

C - 2

- **More on RAM**
- **Example programs using pointers**
 - Multiple level pointers
 - Building a linked list in Java versus C
 - Use of malloc(), free()
- **Heap versus stack storage**

C Data Types

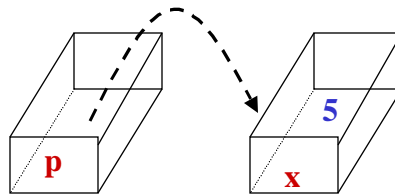
- **Primitive: char, int, double (+others)**
 - Without boolean type, any nonzero value is *true*; zero is *false*
- **Aggregates: arrays, structs**
 - homogeneous arrays
`char a[10]; int b[2][10]`
 - heterogeneous structs

<code>struct employee{ int age; double payrate; }</code>	<code>struct rectangle{ struct point p1; struct point p2; }</code>
--	--

C Data Types

- **Pointers: variables whose value is the L-value of a variable**
 - address-of operator &
 - dereference operator for a pointer to obtain its R-value *

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



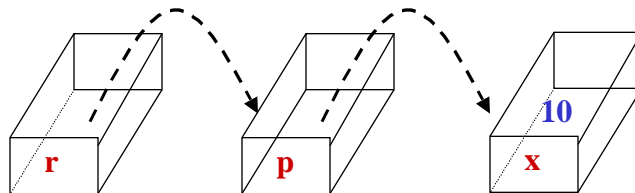
C2, CS314 Fall0, BGR

3

C Data Types

- **Pointers can point to pointer variables (called multi-level pointers)**

```
int *p;  
int **r;  
p = &x;  
*p = 5;  
r = &p;  
**r = 10;
```



C2, CS314 Fall0, BGR

4

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
- Special relation to arrays
- 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

Pointers and Arrays

- An array name is considered pointer to first element
 - a is pointer to $a[0]$
 - $pa = \&a[0]$ and $pa = a$ mean the same thing
 - $a+1$ means L-value of $a[0]$ plus as many bytes as are needed to store value of elements of a 's type
 - Pointer arithmetic is an address calculation with respect to the underlying architecture
- An array name is not a variable
 - $a++$ and $a = pa$ are ILLEGAL!

RAM

- **A universal computing device, similar to a Turing Machine**
- **Components:**
 - Program-sequence of instructions
 - Memory-sequence of locations
 - Control-current location in program
 - Input file- sequence of values
 - Output file-sequence of values
- **Control flows from one instruction to next**

L-values and R-values

- **L-value of location $M[j]$ is location j**
- **R-value of location $M[j]$ is contents of location j**
 - $M[j] = M[k]$ means fetch the r-value of location k and store it into location j (the l-value of $M[j]$)
 - **Indirect addressing**
 - L-value of $M[M[j]]$ is location $M[j]$;
 - R-value of $M[M[j]]$ is contents of location $M[j]$

RAM Instructions

- **Assignment:** **corresponding C stmt**
 M[j]=k; j = k
 M[j]=M[l]+M[n]; j= l+n
 M[j]=M[l]-M[n]; j= l-n
 M[j]=M[M[l]]; j=*l
 M[M[j]]=M[k]; *j=k
- **Input:** read M[k];
- **Output:** write M[k];
- **Branch:** if M[k] >= 0 then goto loop
- **Halt:** halt;

C2, CS314 Fall0, BGR

9

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
47 scherzo!c> gcc multipleptr.c
48 scherzo!c> a.out
99 99 99 */
```

C2, CS314 Fall0, BGR

10

Problems with Pointers

- **Uninitialized pointers**

- `int *a; *a = 12;`

- Can get *segmentation fault* error when value in **a** is meaningless address
 - Can actually store 12 into some memory location accessible to your program, whose address corresponds to the random bits in **a**

Problems with Pointers

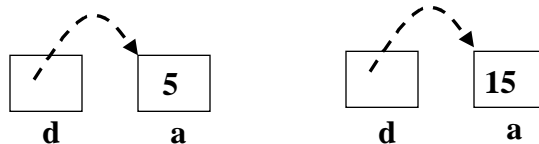
- **Null** doesn't point to anything by definition so it cannot be dereferenced

- `a = 0;` /* makes pointer **a**'s value NULL */

- `if (a == NULL) ...` /* tests for a NULL pointer value*/

- **Multiple level pointers**

- Can be used as L-values or R-values



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

Exerpt from List in Java

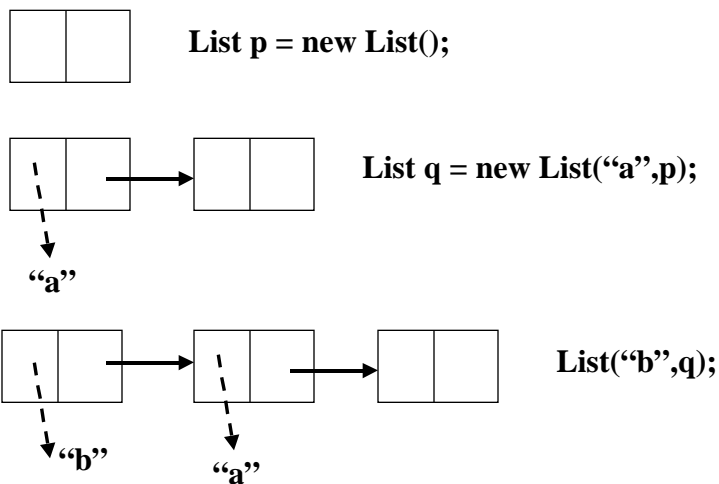
```
public class List extends Object {
protected Object element;
protected List subList;
/**
 * Create an new List, initially empty.
 */
public List() {
element = null;
subList = null;
}

//cons operation
public List(Object newElement,List oldList){
element = newElement;
subList = oldList;
} }
}
```

C2, CS314 Fall0, BGR

13

How List building works?



C2, CS314 Fall0, BGR

14

list.c

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

C2, CS314 Fall0, BGR

15

list.c, cont.

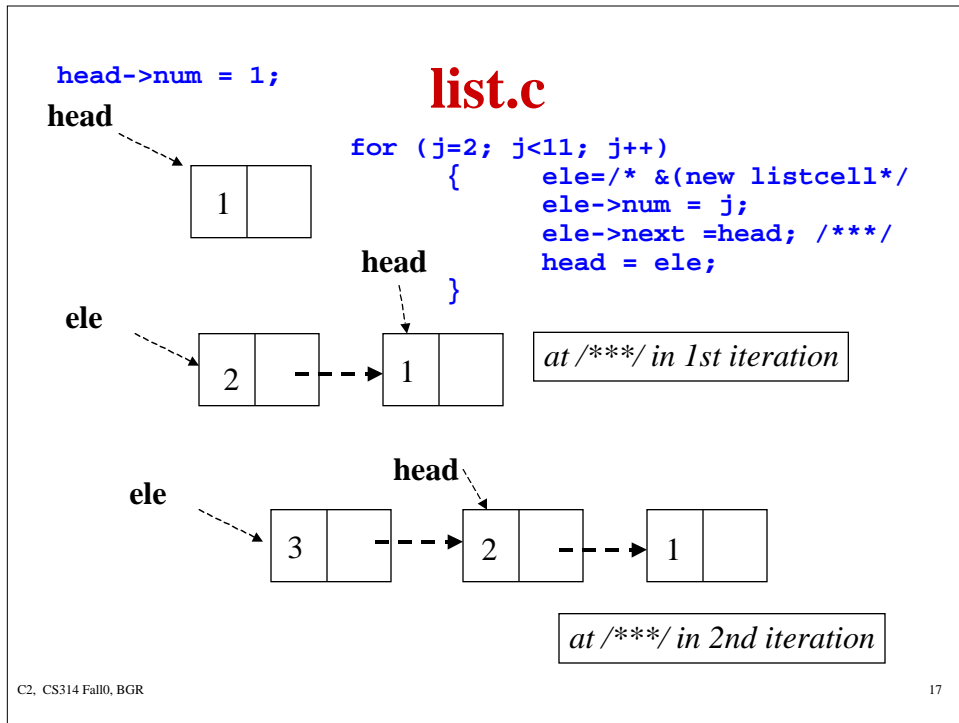
```
main(void)
{ int j;
  /*create first node in list*/
  head = (listcell *) malloc(sizeof (listcell));
  head->next = NULL;
  /*now create entries in list of numbers from 1 to 10*/
  head->num = 1;
  for (j=2; j<11; j++)
  {
    ele = (listcell *) malloc(sizeof (listcell));
    ele->num = j;
    ele->next =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");
}
```

cast → (listcell *)

→ malloc(sizeof (listcell)); → **Allocates heap storage**

C2, CS314 Fall0, BGR

16



list.c output

```

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

```

The output of the program shows the numbers 10 through 1 in descending order, indicating that the program successfully built a linked list with 10 nodes.

C2, CS314 Fall0, BGR 18

Review: *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

Heap Storage

*void *malloc (size_t n)*

- returns pointer to block of contiguous storage of *n* bytes (chars), if possible
- if not enough memory left for allocation, *malloc* returns a NULL pointer
 - So you **ALWAYS** have to check return the value
- to allocate storage of a different type requires sending *malloc* the proper amount of bytes needed and casting the return pointer value appropriately

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

listwithfree.c

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

Same declarations as **list.c**

listwithfree.c, cont.

```
main(void)
{ int j;
  /*create first node in list*/
  head = (listcell *) malloc(sizeof (listcell));
  /*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");
}
```

Same building of the list
and printing it out as **list.c**

listwithfree.c, cont.

```
/*now delete the first 2 elements of the list and
free their storage */
ele = head->next; /*1*/
free (head); /* free 1st list element storage*/
head = ele;
ele = head->next; /*2*/
free (head); /* free 2nd list element storage*/
head = ele; /*3*/

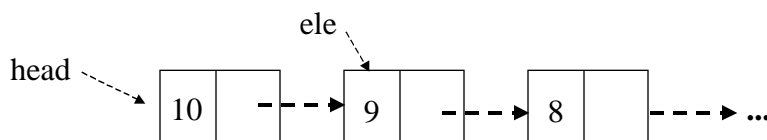
/*now traverse the list and print the elements*/
for (p=head; p!=NULL; p=p->next)
    printf("%d ",p->num);
printf("\n");
}
```

C2, CS314 Fall0, BGR

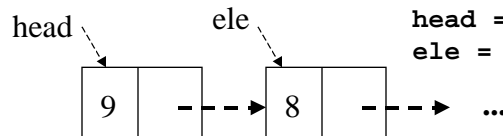
23

Trace

After `ele = head->next; /*1*/`



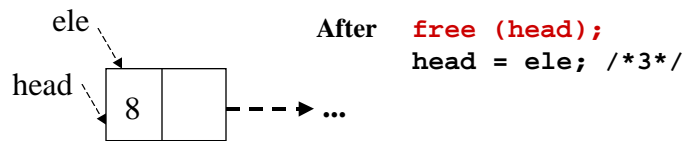
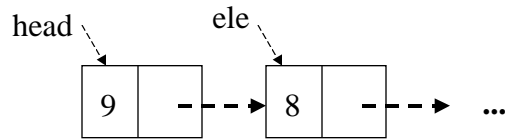
After `free (head);`
`head = ele;`
`ele = head->next; /*2*/`



C2, CS314 Fall0, BGR

24

listwithfree.c, cont.



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