

Imperative Programming: C

- Structures
- Memory allocation
- Linked list example
- Function pointers
- Arrays revisited
 - strings

1

Structures

- Defining new structure types:

```
typedef struct {
    float x_coord;
    float y_coord;
} Point;

struct POLYGON {
    Point vertices[10];
    int N;
} p;
```

2

Structures

- **Pointers to Structures:**

```
typedef struct {
    int age;
    double payrate;
} Employee;
Employee joe, *maryp;
maryp = &joe;
```

- **Accessed as:**

```
(*maryp).payrate = 75000.00;
(*maryp).payrate += 5000.00;
maryp->payrate += 5000.00;
```

3

Recursive Structures

- **This is legal:**

```
typedef struct NODE {
    int data;
    struct NODE *next;
} Cell;
```

- **These are not legal:**

<pre>typedef struct NODE { int data; struct NODE next; } Cell;</pre>	<pre>typedef struct { int data; Cell *next; } Cell;</pre>
--	---

4

Memory Allocation

- Remember the Stack and the Heap?
- Parameters and local variables can be allocated on the stack, as part of the stack frame:

```
int n, *pn;  
pn = &n;
```

- But consider:

```
Employee *maryp;
```

If you want to create the `Employee` structure that is pointed to by `maryp`, where does it go?

On the Heap!

5

Memory Allocation

- This is similar to the storage of lists in Scheme, except that you have to do it yourself!
- Standard allocation procedure:

```
Employee *maryp;  
maryp = (Employee *)malloc(sizeof(Employee));
```

Note: `malloc` returns the `NULL` pointer if it cannot find enough space.

- To release memory:

```
free ( maryp );
```

6

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;
    }

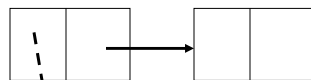
    /* cons operation */
    public List(Object newElement, List oldList) {
        element = newElement;
        subList = oldList;
    }
}
```

7

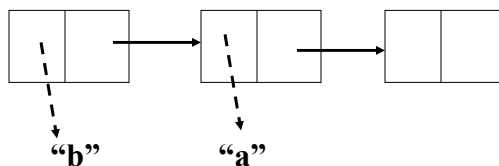
How List building works?



List p = new List();



List q = new List("a",p);



List("b",q);

8

list2.c

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

9

list2.c, cont.

```
main()
{ int j;

  /*create first node in list*/
  head = (listcell *) malloc(sizeof (listcell));
  head->next = NULL;

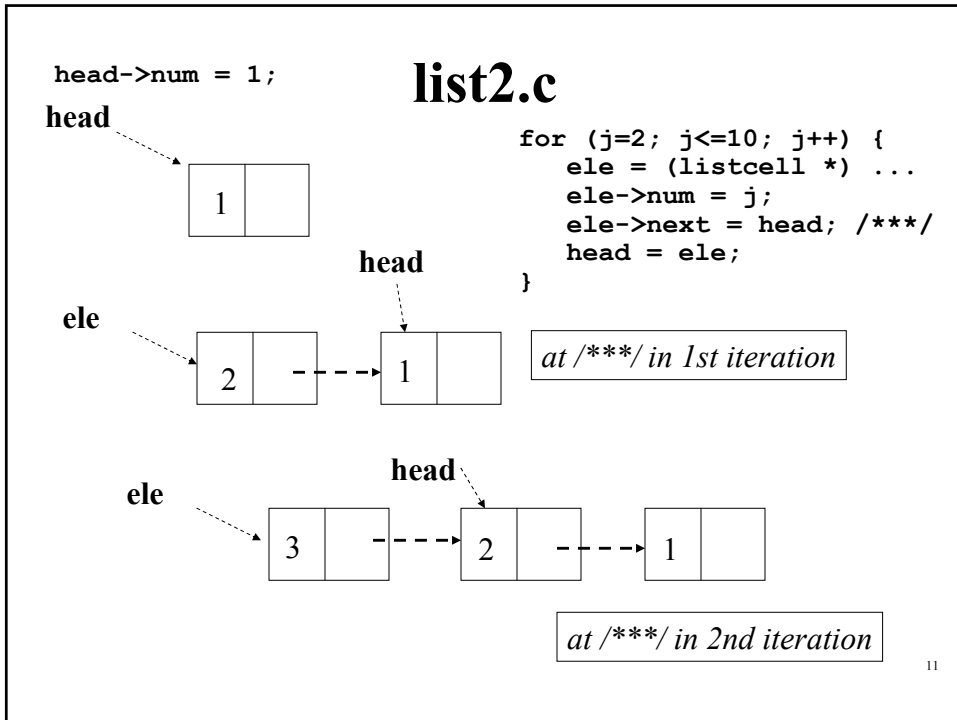
  /*now create entries in list of numbers from 1 to 10*/
  head->num = 1;
  for (j=2; j<=10; j++) {
    ele = (listcell *) malloc(sizeof (listcell));
    ele->num = j;
    ele->next = head; /**/
    head = ele;
  }

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

cast

Allocates heap storage

10



list2.c output

```

% gcc list2.c
% ./a.out
10 9 8 7 6 5 4 3 2 1
%

```

12

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

13

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 should check the return 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));  
if (head == NULL) {  
    ...  
}
```

14

Pointers vs. References

- **listcell and listcell* are different**
- **Explicit dereference operator**
- **Pointers needed for recursive definitions**
- **Casting results in type conversion:**
`int x = (int) (&w);`
- **Memory management is performed by programmer**
- **Everything is a reference**
- **Dereference is implicit**
- **Recursive definitions are automatic**
- **Casting just satisfies the type checker**
- **Memory management is automatic**

15

Pointers to Functions

- **Syntax:**
`int foo(int x){ ... }`
`int (*pf)(int x);`
`pf = &foo;`
- **Motivation: “Polymorphic” Processing**
 - e.g., sorting an array of integers in increasing vs. decreasing order
 - e.g., sorting an array of integers vs. an array of floats vs. an array of Employee structs vs. ...
- **How can we do this?**

16

Pointers to Functions

- Suppose `listcell` is defined as before, but has a data field could be an `int`, a `char`, a `struct`, or ...
- Use a “callback” function:

```
listcell *searchlist(listcell *c, void *value,
                    int (*compare)(void *, void *)) {
    for ( ; c!=NULL; c = c->next) {
        if ((*compare)(&(c->data), value))
            break;
    }
    return c;
}
```

17

Pointers to Functions

- To search a list of integers, define:

```
int equalints(void *a, void *b)
{
    return (*(int *)a == *(int *)b);
}
```

- Then call:

```
listcell *foundit;
int k=5;
foundit = searchlist(head, &k, equalints);
```

18

Pointers to Functions

- To search a list of integers, define:

```
int equalchars(void *a, void *b)
{
    return (*(char *)a == *(char *)b);
}
```

- Then call:

```
listcell *foundit;
char ch = 'a';
foundit = searchlist(head, &ch, equalchars);
```

19

Arrays

- **Array: Typed collection of values indexed by nonnegative integers.**
- **Type and size must be known at compile time, except in the case of formal parameters:**

```
int a[20]; void sort(int b[], int n);
```
- **Arrays in C are stored one-dimensionally:**

```
piece chess[8][8];    (chess has type piece *)
```
- **Strings are arrays of characters:**

```
char msg[21] = "Enter your password:";
```

20