§12.1-12.5: Pointers

24 Nov 2008
CMPT14x
Dr. Sean Ho
Trinity Western University

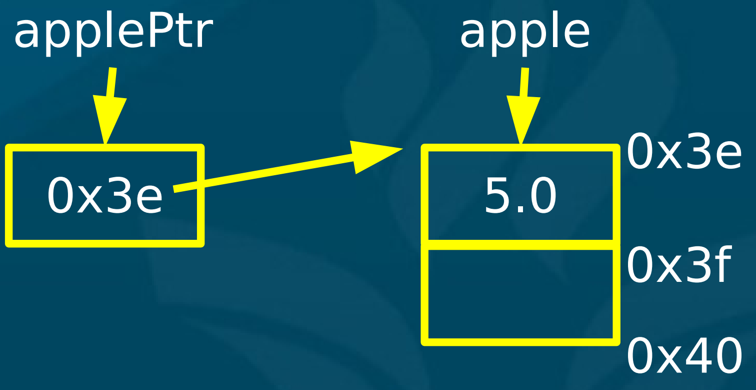
What's on for today (12.1-12.5)

- Pointers (in Modula-2 and C)
  - Creating pointers, dereferencing pointers
  - Assignment compatibility
  - Pointer arithmetic
  - NIL (in C: NULL)
- Static vs. dynamic allocation of memory
  - Activation records
  - Stack, stack pointer
- Dynamic variables: NEW(), DISPOSE()

Pointers

- Values are stored in locations in memory
- These locations are accessed by their addresses, which point to a spot in memory
- A pointer is a variable whose value is a memory address:

```
VAR
    applePtr : POINTER TO REAL;
    apple : REAL;
BEGIN
    apple := 5.0;
    applePtr := SYSTEM.ADR (apple);
```



Dereferencing pointers

- The last example shows how to make a pointer:

```
VAR
    applePtr : POINTER TO REAL;
    apple : REAL;
BEGIN
    apple := 5.0;
    applePtr := SYSTEM.ADR (apple);
```

In C:

```
float apple;
float* applePtr;
apple = 5.0;
applePtr = &apple;
```

- How do we get at the memory pointed to?

```
applePtr^ := 4.0;   (* same as apple := 4.0 *)
```

  - (C syntax: *applePtr)
  - The “hat” operator ^ is called the dereferencing operator

Operations on pointers

- Different pointer types are not compatible
  - But can always cast from one type to another:

```
float* applePtr;
int* pearPtr;
applePtr = (float*) pearPtr;
```

- NIL points to nothing at all
  - Handy for initializing pointers: ptr1 := NIL;
  - Dereferencing NIL raises sysException
  - In C, use NULL (which is just 0)

Pointers and C arrays

- An array in C is really just a pointer to a location in memory that stores consecutive entries of the array:

```
float appleSizes[4];
appleSizes[0] = 2.5;
```

  - Indexing into the array is really done by adding to the pointer to the head of the array:

```
appleSizes[2]
```

    - Is the same as:

```
*(appleSizes + 2)
```

Pointers and call-by-reference

- Pointers are how call-by-reference is done in C:

```
int increment (int* x) {        /* takes a pointer to an int */
    *x = *x + 1;
    return *x;
}
int x;
x = 5;
increment (&x);                 /* pass a pointer to x */
```

- In C++, can specify in the function definition:

```
int increment (int &x) {        /* call-by-reference */
    x = x + 1; ....
increment (x);
```

Static vs. dynamic memory

- Static variables are allocated at the beginning of the program run
  - Their size in memory is fixed at compile-time
  - Variables named in declaration section
- Dynamic variables are allocated during the running of a program
  - May also be deallocated during program
  - Size need not be predetermined
  - Reference them via pointers

Dynamic variables

- You can make your own dynamically allocated variables, using NEW() and DISPOSE():

```
VAR
    applePtr : POINTER TO REAL;
BEGIN
    NEW (applePtr);
```

  - Allocates memory for a REAL, and stores the address in applePtr

```
    DISPOSE (applePtr);
```

  - Deallocates the memory, and sets applePtr to NIL
- Dynamic variables are in the heap:
  - Open space for program to allocate/deallocate
- If heap is full, NEW sets pointer to NIL

A caution about pointers

- Pointers are a powerful tool and a quick way to shoot yourself in the foot:

```
VAR
    applePtr : POINTER TO REAL;
BEGIN
    applePtr^ := 5.0;        (* yipes! *)
```

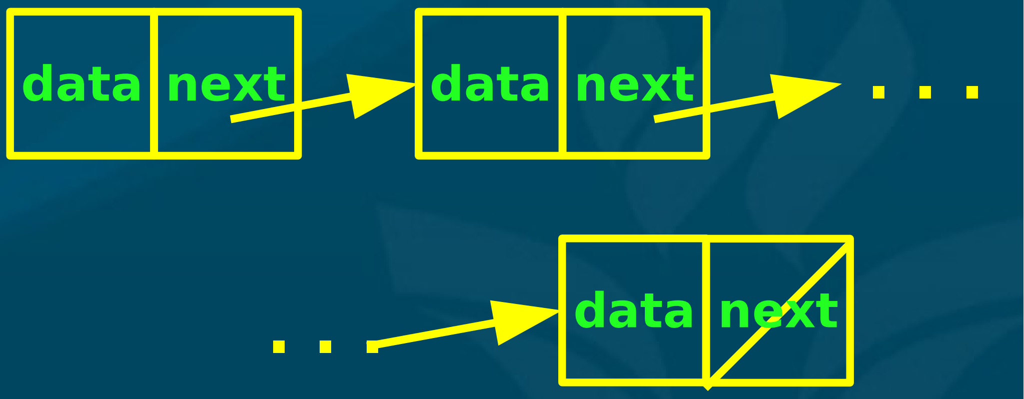
  - Uninitialized pointer could point to anywhere in memory: dereferencing it can potentially modify any accessible memory!
    - Can crash older Windows; core dump in Unix

Linked lists: creating

- A linked list is a dynamic ADT where each item in the list contains a pointer to the next item:

```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

n1 = Node()
n2 = Node()
n1.next = n2
```



Operations on linked lists

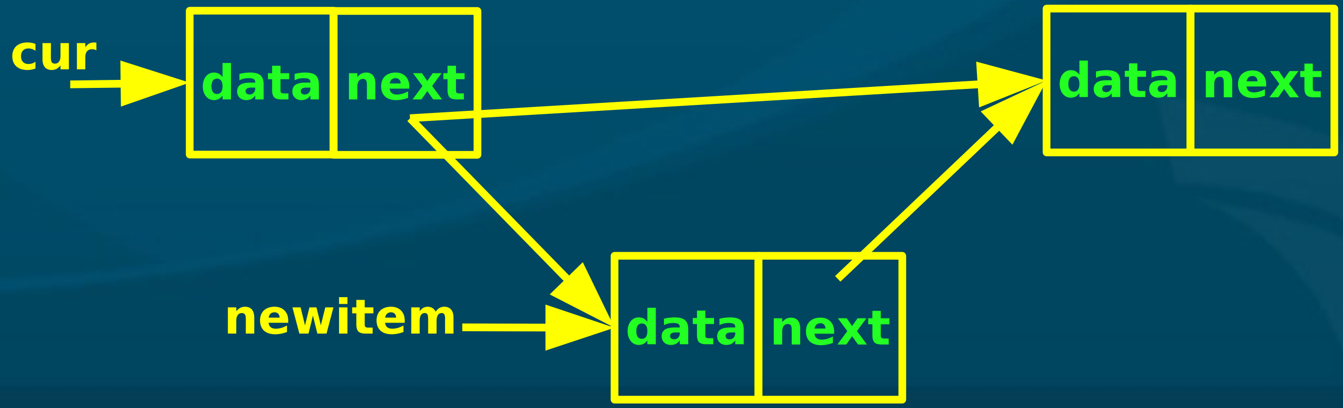
- Index into list (get a reference to n^{th} node)
- Print out the list
- Search list for given data (cargo/payload)
- Insert a new node into a linked list
- Delete a node from a linked list
  - By index (0, 1, 2, ...) or by cargo

Inserting a node into a linked list

- Follow pointers to get to the right spot
  - Create a new node with the given cargo
  - Thread new node into the list

```
newitem = Node(data)
newitem.next = cur.next
cur.next = newitem
```

  - What about inserting at head of list?



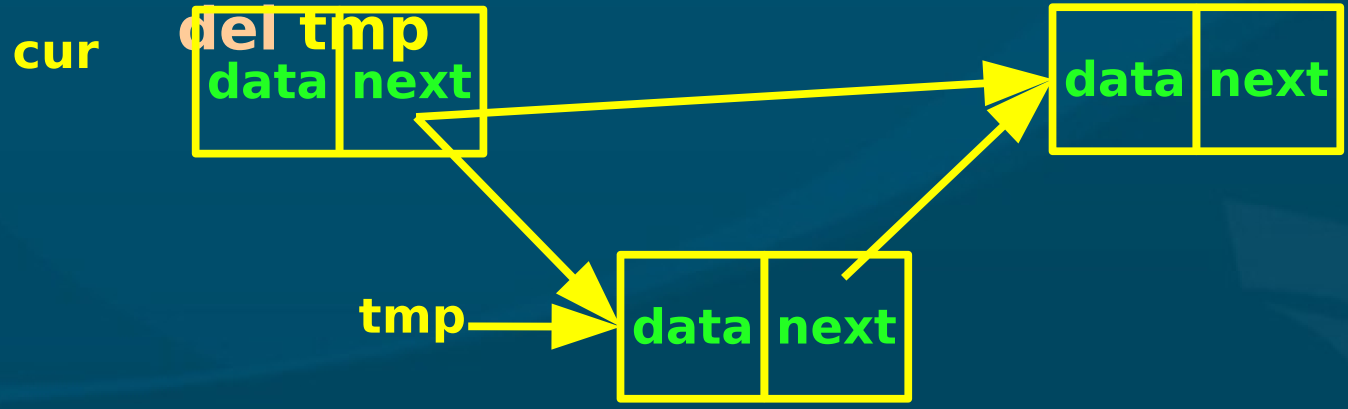
Insert() method: code

```
def insert (self, n, data=None):
    """Insert a new node into linked list at position n."""
    newitem = Node(data)
    if n == 0:                          # new head: modify self
        newitem.next = self
        self = newitem
    else:
        cur = self
        for idx in range(n-1):          # get to proper position
            cur = cur.next
        newitem.next = cur.next
        cur.next = newitem
```

Deleting from a linked list

- Follow pointers to find the item we want to delete
  - Sew up links to skip over the item
  - Deallocate the item from memory

```
tmp = cur.next
cur.next = tmp.next
del tmp
```



Linked lists: algorithmic efficiency

- Big-O notation: O(n) means # operations varies linearly with n
- For a linked list with n items:
  - Insert at head: don't have to traverse list: O(1)
  - Append to tail: must walk list: O(n)
  - General insert:
    - Worst-case: O(n)
    - Average-case: O(n/2), which is also O(n)
  - Deleting: also O(n)
- Double-headed list (keep a tail pointer):
  - Speeds up append-to-tail to O(1)

Variants of linked lists

- Circularly linked list:
    - tail.next = head
  - How to keep from infinite loop?
- Bidirectional linked list:

```
class Node:
    def __init__(self, data=None, prev=None,
      next=None):
        self.data = data
        self.prev = prev
        self.next = next
```