Link Lists

孫雅麗
台灣大學 資管所
Pointer

- Two most important operators
  - & : address
  - * : de-referencing (or indirection)

- Examples
  
  if i is a variable
  - &i : address of i
  - pi = &i; /* assign address of i to a pointer variable pi */
  - i = 10;
  - cout << *pi;
Dynamic Storage Allocation

Problem:
- Allocate memory space on-demand
- Useful for programs that do not know how much space needed in prior to execution and do not want to allocate large area that may never be required.

Solution:
- Allocate storage at run-time
  - library function calls
  - C++ “new”, C “malloc”
    - to request the system to allocate a block of memory space;
    - the function returns a pointer pointing to the allocated block.
  - C++ “del”, C “free”:
    - to return a storage area to the system.
Figure 4-1
(a) A linked list of integers; (b) insertion; (c) deletion
Look in location 342 for what you want

A pointer to an integer

Memory locations

\[ \text{Address} \]

\[
\begin{array}{cccc}
\ldots & 26 & 10 & 5 \\
340 & 341 & 342 & 343
\end{array}
\]

\( P \)
(a) Declaring pointer variables;
(b) pointing to statically allocated memory;
(c) assigning a value;
(d) allocating memory dynamically;
(e) assigning a value;

(a) `PtrType P, Q;
    int X;`

(b) `P = &X;`

(c) `*P = 6;`

(d) `P = new int;`

(e) `*P = 7;`
(f) copying a pointer;

\[ Q = P; \]

\[ Q = \text{new int}; \quad *Q = 8; \]

(h) assigning \textbf{NULL} to a pointer variable;

\[ P = \text{NULL} \]

\[ \text{delete } Q; \quad Q = \text{NULL}; \]
Programming with pointer variables that reference dynamically allocated memory

```c
typedef int* ptrType;
ptrType P, Q;

P = new int; // Allocate a cell of type int.

*P = 1; // Assign a value to the new cell.

Q = new int; // Allocate a cell of type int
```
(continued)

*Q = 2; // Assign a value to the new cell.

cout << *P << " " // Output line contains: 1 2
  << *Q << endl; // These values are in
  // the cells to which P // and Q point.

*P = *Q + 3; // The value in the cell to
  // which Q points, 2 in this
  // case, 3 are added together.
  // The result is assigned to the
  // cell to which p points.

cout << *P << " " // Output line contains: 5 2
  << *Q << endl;
P = Q; // P now points to the same
    // cell as Q. The cell P formerly
    // pointed to is lost; it cannot
    // be referenced.
    cout << *P << " "    // Output line
    << *Q << endl; // contains: 2 2

*P = 7; // The cell to which P points
    // (which is also the cell to
    // which Q points) now contains
    // the value 7.
    cout << *P << " "    // Output line
    << *Q << endl; // contains: 7 7

P = new int; // This changes what P
    // points to, but not what
    // Q points to.
(continued)

```cpp
delete P;  // Return to the system the
    // cell to which P points.
```

```cpp
P = NULL;  // Set P to NULL, a good
    // practice following delete.
```

```cpp
Q = NULL;  // The cell to which Q
    // previously pointed is
    // now lost. You cannot
    // reference it.
```
Linked Lists

- A list is a sequence of elements of the same type
  \[ L = \{X_0, \ldots, X_{n-1}\} \]

- One-dimensional array implementation
  - \(i^{th}\) element -> \(Loc_i\)
  - \((i+1)\) element -> \((Loc_i + c) = Loc_{i+1}\)
  - Sequential mapping
  - Expensive insertion and deletion operations
    (lots of move !!)
Linked Lists (cont’d)

- **Linked Lists: an alternate implementation**
  - an element -> a node
    - data component
    - a “pointer” to the next item in the list (a.k.a “links”)
    - can be anywhere in the memory
    - \( Loc_{i+1} \neq (Loc_i + c) \)
  
Logical relationship!!
A node in a linked list is usually a `struct`

Example declaration #1:

```c
struct node {
    int data; // data component
    node* next;
}; // end struct

typedef node *ptrType;
```
Example declaration #2:

- Define ptrType before node and use it within the definition of node.
- Must declare node first, as follows:

```c
struct node;  // declare here, define later
typedef node* ptrType;

struct node
{
    int data;
    ptrType next;
};  // end struct
```
Programming with Linked Lists (1/8)

• Define a head pointer

Figure 4-7
A head pointer to a list

Figure 4-8
A lost cell

Head = new node;

Head = NULL;
Display the Contents of a Linked List

1. Let a current pointer point to the first node in the linked list
2. while (the current pointer is not NULL)
3. { Display the data portion of the current node
4. Set the current pointer to the next pointer of the current node
} //end while

in C++:

```cpp
for (ptrType cur=head; cur!=NULL; cur=cur->next)
    cout<< cur->item << end;
```
Programming with Linked Lists (3/8)

- **Delete a specified node from a linked list (interior node)**

```c
prev = NULL;
cur = head;
```

**Figure 4-10**
Deleting a node from a linked list

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Delete a specified node from a linked list (first node)

```
prev = NULL;
cur = head;
head = cur->next
```
Before change the value of `cur`, you should use the statements

```c
cur->next = NULL;
delete cur; // Return deleted node to the system
cur = NULL;
```
Insert a node into a specified position of a linked list

1. `newPtr->next = cur;`
2. `prev->next = newPtr;`
Inserting a node into a specified position of a linked list (first node)

newPtr->next = head;
head = newPtr;

Figure 4-13
Inserting at the beginning of a linked list
Programming with Linked Lists (8/8)

- Inserting a node into a specified position of a linked list (last node)

\[ \text{newPtr->next = cur; } \]
\[ \text{prev->next = newPtr; } \]

**Figure 4-14**
Inserting at the end of a linked list
Determine the point of insertion into a sorted linked list

// initialize prev, cur to start the traversal from the beginning
prev = NULL;
cur = head;

// advance prev and cur as long as newValue > the current data item
// Loop invariant: newValue > data items in all nodes at and before
// node to which prev points
while (cur != NULL && newValue > cur->item) {
  prev = cur;
  cur = cur->next;
}
  // end while
Linked Lists: find the position (2/2)

- Another way of coding

```c
for (prev = NULL, cur = head;
     (cur != NULL) && (newValue > cur->item);
      prev = cur, cur = cur->next);
```
A Pointer-Based ADT List:

listNode (1/12)

// ***********************************************
// Implementation file ListP.cpp for the ADT list.
// Pointer-based implementation.
// ***********************************************
#include "ListP.h" // header file
#include <stddef.h> // for NULL
#include <cassert.h> // for assert()

struct listNode // a node on the list
{
    listItemType item; // a data item on the list
    ptrType next; // pointer to next node
} // end struct
A Pointer-Based ADT List: constructor (2/12)

**Default Constructor**

```cpp
listClass::listClass(): size(0), head(NULL) {
} // end default constructor
```

**Copy Constructor**

```cpp
listClass (const listClass& L); // copy constructor
```
*Figure 4-17*

A pointer-based implementation of the ADT list
Figure 4-18
Copies of the linked list in Figure 4-17: (a) a shallow copy; (b) a deep copy
A Pointer-based ADT List: deep copy constructor (3/12)

```cpp
listClass::listClass(const listClass& L): size(L.size)
{
    if (L.head == NULL)
        head = NULL;  // original list is empty
    else
    {
        // copy first node
        head = new listNode;
        assert(head != NULL);  // check allocation
        head->item = L.head->item;  // assignment? - overloading
    }
}
```
A Pointer-based ADT List: deep copy constructor (4/12)

// copy rest of list
ptrType newPtr = head; // new list pointer
// origPtr points to nodes in original list
for (ptrType origPtr = L.head->next; origPtr != NULL; origPtr = origPtr->next)
{
    newPtr->next = new listNode;
    assert(newPtr->next != NULL);
    newPtr = newPtr->next;
    newPtr->item = origPtr->item;
}
// end for
newPtr->next = NULL;
} // end if
} // end copy constructor
A Pointer-based ADT List: destructor (5/12)

listClass::~listClass()  //destructor
{
    bool success;

    while (!listIsEmpty())
        listDelete(1, success);

}  // end destructor
A Pointer-based ADT List: list length (6/12)

bool listClass::listIsEmpty() const
{
    return bool (size == 0);
}  // end listIsEmpty

int listClass::listLength() const
{
    return size;
}  // end listLength
ptrType listClass::ptrTo(int position) const
// Locates a specified node in a linked list.
// Precondition: position is the number of the desired node.
// Postcondition: Returns a pointer to the desired node.
// If position < 1 or position > the number of nodes in the list, returns NULL.
{
    if ( (position < 1) || (position > listLength()) ) return NULL;
    else // count from the beginning of the list
    {
        ptrType cur = head;
        for (int skip = 1; skip < position; ++skip)
            cur = cur->next;
        return cur;
    } // end if
} // end ptrTo
A Pointer-based ADT List: retrieve a node (8/12)

```cpp
void listClass::listRetrieve(int position, itemType& dataItem, bool& success) const
{
    success = bool((position >= 1) && (position <= listLength()));

    if (success)
    {
        // get pointer to node, then data in node
        ptrType cur = ptrTo(position);
        dataItem = cur->item; // assignment operator overloading
    } // end if
}
} // end listRetrieve
```
void listClass::listInsert(int newPosition, listItemType newItem, bool& success)

{ int newLength = listLength() + 1;
  success = bool( (newPosition >= 1) &&
        (newPosition <= newLength) );

if (success)
{
  // create new node and placeNewItem in it
  ptrType newPtr = new listNode;
  assert(newPtr != NULL); // check allocation
  size = newLength;
  newPtr->item = newItem;
  }
A Pointer-based ADT List: insert a node (10/12)

// attach new node to list
if (newPosition == 1)
{
    // insert new node at beginning of list
    newPtr->next = head;
    head = newPtr;
}
else {
    ptrType prev = ptrTo(newPosition-1);
    // insert new node after node to which Prev points
    newPtr->next = prev->next;
    prev->next = newPtr;
}
} // end if

} // end if

} // end ListInsert
A Pointer-based ADT List: delete a node  (11/12)

void listClass::listDelete(int position, bool& success)
{
  prtType cur;
  success = bool ( (position >=1) &&
                   (position <= listLength() ) );
  if(success)
  {
    --size;
    if(position ==1)
    {
      // delete the first node from the list
      cur = head; // save pointer to node
      head = head->next;
    }
  }
}
A pointer-based ADT List: delete a node (12/12)

else
    {
        ptrType prev = ptrTo(position-1);
        // delete the node after the node to which Prev points
        cur = prev->next; // save pointer to node
        prev->next = cur->next;
    } // end if

// return node to system
cur -> next = NULL;
delete cur;
cur = NULL;
} // end if
} // end lstDelete
A Pointer to the Tail of the List

- Application needs
A Head Pointer as a Value Argument

- Procedure call
- Call by value (e.g., func_name(int i));
  - A separate variable (memory location) is created with the same value as the input argument
  - Any modification to the value of the variable will not change the value of the variable in the calling procedure
- Call by reference (e.g., func_name(char a[]));
procedure_1 (head); //calling program
...

procedure_1 (headPtr); //called program
{
...
}

"Actual argument"
Head

HeadPtr
"Formal argument"
A value argument vs. a reference argument

- A Value argument allows called function to modify the nodes of the list and even to insert and delete nodes.
- To change header node, a Reference argument is needed.
Circular Linked List: list pointer points to the first node (1/3)

Before: NULL
Now: to the first node
Circular Linked List: list pointer points to the first node (2/2)

1. Insert at front
   - new_node -> link = ptr;
   - last_node -> link = new_node;
   - ptr = new_node;

2. Insert at rear
   - last_node -> link = new_node;
   - new_node -> link = ptr
Circular Linked List: list pointer points to the first node (3/3)

- In “insert”, one must traverse from the first node to the last node (i.e. last_node) -> O(n).

- Inefficient “insert” operation.

- A performance improvement: let list pointer points to the last node of the list.
A Circular Linked List with pointer pointing to the last element (1/2)

Figure 4-26
A circular linked list with an external pointer to the last node
A Circular Linked List with pointer pointing to the last element (2/2)

- Insert a new node at either the **front** or the **rear** of a circularly linked list

```c
1. new_node->link = ptr->link;
2. ptr->link = new_node;
```

- Insert at front

```c
1. new_node->link = ptr->link;
2. ptr->link = new_node;
```

- Insert at rear

```
ptr = new_node;
```
A Doubly Linked List (1/3)

vs. singly linked list

Figure 4.29
A doubly linked list
A Doubly Linked List – “delete” a node (2/3)

```c
cur->llink->rlink = cur->rlink;
cur->rlink->llink = cur->llink;
cur->rlink = null; cur->llink = null;
```

```
... del cur; cur=NULL;
```
A Doubly Linked List: “add” a node (3/3)

```c
newPtr->rlink = cur;   // 2
cur->llink->rlink = newPtr;  // 4
cur->llink = newPtr;     // 1

newPtr->llink = cur->llink;  // 3
```
Chains: concatenation (1/2)

Chains: Concatenating Two Singly Linked Lists

ptr1

ptr2

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Chains: concatenation (2/3)

list_pointer concatenate(list_pointer ptr1, list_pointer ptr2)
{
    /*produce a new list that contains the list ptr1 followed by the list ptr2.
The list pointed to by ptr1 is changed permanently*/

    list_pointer temp;

    if (IS_EMPTY(ptr1)) return ptr2;
    else {
        if (!IS_EMPTY(ptr1)) {
            for (temp = ptr1; temp->link; temp = temp->link)
                ;
            temp->link = ptr2;
        }
        return ptr1;
    }
}

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document
may be reproduced, stored in a retrieval system, or transmitted in any form,
or by any means without the prior written permission of the author.
Chains: Inverting a singly linked list (1/2)

lead: points to the first of the original remaining list;
middle: points to the node to be inverted;
trail: points to the first of the inverted list

Reverse linking

advancing
Chains: inversion (2/2)

```c
list_pointer invert(list_pointer lead) {
    /*invert the list pointed to by lead*/

    list_pointer middle, trail;
    middle = NULL;
    while(lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```

Chains: Inverting a Singly Linked List

- **advancing**
- **reverse linking**
The end. 😊
Homework 2008

- Pages 242-243: #22 and #23.
- Pages 244-245: #8, #11