# 3.6 Evaluation of Expressions

# 3.6.1 Expressions

###  X = A / B - C + D \* E - A \* C

#### to fix the order of evaluation, assign to each operator a priority

# 3.6.2 Postfix notation

## infix notation

### the operators come *in*-between the operands

### tree에서 다시 공부하므로 철저히 학습 필요

## convert the expression into a postfix notation

### postfix form : call for each operator to appear after its operands

### infix A\*B / C -> postfix AB\*C/

#### evaluate the postfix form by using the stack

#### the virtues of postfix notation that enable easy evaluation of expressions

####  infix: A / B - C + D \* E - A \* C

####  postfix: A B / C - D E \* + A C \* -

## evaluating the postfix notation by using the stack



# 3.6.3 Infix to postfix

## an algorithm for producing postfix from infix

### 1) fully parenthesize the expression

### 2) move all operators for replacing their corresponding right parentheses

### 3) delete all parentheses

####  infix: A / B - C + D \* E - A \* C

####  when fully parenthesized: ((( A / B ) - C ) + ( D \* E )) - ( A \* C ))

####  postfix: A B / C - D E \* + A C \* -

## form the postfix by stacking

### infix: A + B \* C

####  next token stack output

####  ------------- -------- --------

####  none empty none

####  A empty A

####  + + A

####  B + AB

## if \* get placed on top of the stack or if the + gets taken off

### priority of \* is higher than +

### stacking \*

####  \* +\* AB

####  C +\* ABC

## if the input expression is empty, then output all remaining operators in the stack

####  ABC\*+

## infix: A \* (B + C) \* D

####  next token stack output

####  ------------- ------- ----------

####  none empty none

####  A empty A

####  \* \* A

####  ( \*( A

####  B \*( AB

####  + \*(+ AB

####  C \*(+ ABC

## unstack down to the corresponding left parenthesis

####  ) \* ABC+

####  \* \* ABC+\*

####  D \* ABC+\*D

####  done empty ABC+\*D\*



## a priority-based scheme for stacking and unstacking operators

### establish two priorities for operators : isp(in-stack priority) and icp(in-coming priority)



### isp( '(' ) = 8, icp( '(' ) = 0, isp ( '#' ) = 8

### stacking rule : operators are taken out of the stack as long as their in-stack priority is numerically less than or equal to the in-coming priority of the new operator



# Chap 4 Linked Lists

# 4.1 Singly linked lists

## a sequential mapping used for ordered lists

### expensive insertion and deletion of arbitrary elements

#### require to move elements in the list either one location higher or lower

## sequential representation for several ordered lists of varying sizes

### waste an unused space

### compute the size of the dynamically allocated list

## linked representation

### pointer or link

### the elements no longer occur in sequential order



### to draw linked lists as an ordered sequence of nodes with links being represented by arrows



## why it is easier to make arbitrary insertions and deletions using a linked lists rather than a sequential list





# 4.2 Representing chains in C++

## a set of objects를 정의하는 클래스?

### object를 정의하는 class

### list를 정의하는 class

# 4.2.1 Defining a list node in C++

#### class ThreeLetterNode {

#### private:

####  char data[3];

####  ThreeLetterNode \*link;

#### };

Example 4.1: class definitions for link lists

#### class nodeb {

#### private:

####  int data;

####  nodeb \*link;

#### };

#### class nodea {

#### private:

####  int data1;

####  char data2;

####  float data3;

####  nodea \*linka;

####  nodeb \*linkb;

#### };



4.2.2 Designing a list in C++

## how to represent a single node in C++

Design attempt 1

#### ThreeLetterNode \*first; // considered to be a global variable

##### reference the data members of the node pointed to by first

#####  first→data, first→link

#####  first→data[0], first→data[1]



#### compile-time errors because private data members cannot be accessed from outside the class

#### Example 4.1: private representation

Design attempt2

## make the data members public

### do the trick

### violate data encapsulation

## to define public member functions GetLink(), SetLink(), GetData(), SetData()

### used to indirectly access the data members

### not a good solution because any function in the program can access data members of ThreeLetterNode

## a good solution : only grant the functions that perform list manipulation operations: insert a node, delete a node

### no other functions should have access to data members of ThreeLetterNode

Design attempt 3

## need to tackle data structure design problem

## to implement a singly linked list

### should contain a class corresponding to the entire list data structure

#### this class should support member functions that perform list manipulation operations

### ThreeLetterList has-a ThreeLetterNode

### 다항식을 정의하는 class와 유사함

### class term {

### friend Polynomial;

### private:

### float coef;

### int exp;

### };

### class Polynomial {

### ...

### private:

###  static term termArray[MaxTerms];

### };

## Definition: A has-a B iff A contains B or B is a part of A



#### Objects of ThreeLetterNode are declared as data members of ThreeLetterList

#### ThreeLetterList physically contains many objects of class ThreeLetterNode

##### # of nodes in a linked list is not a constant

##### be impossible to know in advance the number of ThreeLetterNodes in ThreeLetterList



#### contain only the access pointer

##### ThreeLetterNode objects are not physically contained inside ThreeLetterList

## a solution of dilemma of how to define class ThreeLetterNode so that only list manipulation operations have access to its data members

### achieved by declaring ThreeLetterList to be a friend of ThreeLetterNode

#### friend: member functions of ThreeLetterNode and ThreeLetterList can access the private data members of ThreeLetterNode

#### prog 4

# 4.2.3 Pointer manipulations in C++

## create nodes of a type by using the C++ command new

#### ThreeLetterNode\* y, z;

#### y = new nodea;

#### z = new nodeb;

#### \*y, \*z denote the nodes of type NodeA and NodeB

#### delete y; delete z;

## allow pointer variables to be assigned the null pointer constant 0

## permit addition of integers to pointer variables (have no logical meaning)

### y++

## compare two pointer variables of the same type

### y == z, y != z, y !=0

### Figure 4.9 Effect of Pointer assignments

# 4.2.4 List manipulation operations

## must declare list manipulation operations as member function of the list class

#### class ListNode {

#### private:

####  int data;

####  ListNode \*link;

#### };

#### class xxList {

#### public:

####  // list manipulation operations

#### private:

####  ListNode \*first;

#### }

## assume the access pointer first to be a private data member of List

Example 4.2

## create a linked list with two nodes of type ListNode





Example 4.3, 4.4

## Program 4.4, Chain::Insert50( )

## Program 4.5 Chain::Delete( )

### see the page 184: problem3,4

# 4.3 A Reusable Linked List Class

# 4.3.1 Implementing linked lists with templates

## a linked list is a container class

### a good candidate for implementation with templates



### List<Type> : a friend of ListNode<Type>

### declare first to be a pointer to object ListNode<Type>

## define an empty linked list of integers intlist

#### List <int> intlist;

#### 임의의 클래스가 있을 때 template으로 정의할 수 있을 것-이것이 프로그래밍 능력임

4.3.2 Linked lists iterators(Chain Iterators)

## motivate the need for iterators of a container class

### 1) print all integers in class C

### 2) obtain the max, min, mean, median of all integers in C

### 3) obtain the sum, product, or sum of squares of all integers in C

### 4) obtain all integers in C that satisfy some property P

### 5) obtain the integer x from C such that, for some function f, f(x) is max

## an iterator

### an object that is used to traverse all the elements of a container class one by one

### while loop를 object로 인식하는 개념이 필요

### OLE/COM, ODBC, JDBC이해에 매우 중요

## the pseudo-code for traversing all elements of the container class

#### initialization step;

#### for each item in C

#### {

####  current = current item of C;

####  body;

#### }

#### postprocessing step;

##### 적색 라인: depend on the container class

##### all operations of the type have to be implemented as member functions of the particular container class

## to find the maximum of all elements in the container class

#### for (int I=0;I<n;I++)

#### {

#### current = a[I];

#### x = max(current,x);

####  }

#### int x = -MAXINT;

#### for (ListNode<int> \*ptr = first; ptr != 0; ptr = ptr->link)

#### {

####  current = ptr->data;

####  x = max(current, x);

#### }

#### return x;

The max element have to be implemented as member functions of the container class as these operations access private data members of the container class. There are some drawbacks to this approach:

## 1) all operations will be member functions of List<Type> if the container class is the template list class

### all of its operations should be independent of the type of object to which T is instantiated

#### if the body: computing the sum of all objects, then do not make sense when the objects are of type Rectangles

## 2) It is not feasible for the class designer to predict all the operations required by a particular user of the class

#### He would be forced user to add that operation to the container class

#### list<Type>에 모든 operations을 미리 만들어 놓을 수 없음

## 3) have to learn how to traverse the container class, even if the added member functions are allowed

## suggest that the iterative operations be moved outside the definition of List<Type>

### the iterative operations

#### no modify the contents of the linked list

#### need to access private data members of List<Type> and ListNode<Type>

## define a third class ListIterator<Type>

### handle the details of the linked list traversal

### retrieve elements stored in the list



Program 4.9 List iterator functions



#### do not require access to the private data members of List<Type> or ListNode<Type>

## examine why it is better to define a third class *ListIterator* than to augment the *List* class with the private variable *current* and the four public member functions

### page 194: exercise 4.

#### Not possible for multiple iterators to be defined on the same list if the iterator operations are implemented as member functions of List<Type>

#### Let x1,x2,…,xn be the elements of a chain. To compute the sum(i=1, n-5) (xi \* xi+5)

# 4.3.3 Linked list operations

## choose which operations to include in reusable classes

### important to provide enough operations so that the class can be used in many applications

### important not to include too many operations

#### constructors

#### a destructor

#### an assignment operator (operator =)

#### the test-for-equality operator (operator ==)

#### operators to input and output a class object (operator >>, operator <<)

## see how to implement the destructor for linked lists

## useful operations in reusable classes

### insertions to, deletions from, and other manipulations of the linked list

#### attach an element to the end of a list



### invert a chain "in place"

#### make use of three pointers

#### prog 4

### concatenate two chainprog 4

# 주요 학습 내용:

## invert( ), concatenate( ) 알고리즘을 기억하는 것이 중요한 것이 아니고 두개의 list가 있을 때 merge하는 메카니즘을 이해하고 이를 활용할 수 있는 능력을 가지는 것이 중요함

### -> merge with two sorted lists

# 4.3.4 Reusing a class

## illustrate how class *Polynomial* can be implemented by reusing the linked list class: section 4.6

# some scenarios where one should not attempt to reuse a class

## 1) less efficient than directly implementing class

### if efficiency is of paramount importance, then implement another class

### use linked lists to implement stacks and queues: section 4.6 uses chains to implement stacks and queues

#### data structure가 다른 응용에서도 재사용되지만 매우 중요한 데이터 구조라면, 가장 효율적인 구조로 직접 구현하는 것이 바람직

## 2) if the operations required by the applications are complex and specialized, do not reuse an existing classes

### implement equivalence classes directly

### section 4.8 implements equivalence classes directly

본 강의 자료의 그림 및 알고리즘 발췌

저자 : HOROWITZ

타이틀 : FUNDAMENTALS OF DATA STRUCTURES IN C++ 2nd Edition (2006)

공저 : SAHNI, MEHTA

 출판사 : Silicon Press