

Ve 280

Programming and Introductory Data Structures

Operator Overloading;
Linear List; Stack

Outline

- Operator overloading
- Linear List
- Stack
 - Implementation
 - Application

Operator Overloading

Introduction

- C++ lets us **redefine** the meaning of the operators when applied to objects of **class type**.
- This is known as **operator overloading**.
- We have already seen the overloading of the assignment operator.
- Operator overloading makes programs much easier to write and read:

```
IntSet is;  
int x = is[5]; // overload [] operator  
           // access the IntSet element by index  
cout << is << endl; // overload << operator  
           // print all the IntSet elements
```

Operator Overloading

Basics

- Overloaded operators are functions with special names: the keyword **operator** followed by the symbol (e.g., +, -, etc.) of the operator being redefined.
- Like any other function, an overloaded operator has a return type and a parameter list.

```
A operator+(const A &l, const A &r) ;
```

Operator Overloading

Basics

- Most overloaded operators may be defined as ordinary **nonmember** functions or as class **member** functions.

```
A operator+(const A &l, const A &r);  
// returns l "+" r
```

```
A A::operator+(const A &r);  
// returns *this "+" r
```

- Overloaded functions that are members of a class may appear to have **one fewer** parameter than the number of operands.
 - Operators that are member functions have an implicit **this** parameter that is bound to the **first operand**.

Operator Overloading

Basics

- An overloaded **unary** operator has **no** (explicit) parameter if it is a member function and **one** parameter if it is a nonmember function.
- An overloaded **binary** operator would have **one** parameter when defined as a member and **two** parameters when defined as a nonmember function.

Example

- Overload **operator+=** for a class of complex number.

```
class Complex {  
    // OVERVIEW: a complex number class  
    double real;  
    double imag;  
public:  
    Complex(double r=0, double i=0); // Constructor  
    Complex &operator += (const Complex &o);  
    // MODIFIES: this  
    // EFFECTS: adds this complex number with the  
    // complex number o and return a reference  
    // to the current object.  
};
```

Example

```
Complex &Complex::operator += (const Complex &o)
{
    real += o.real;
    imag += o.imag;
    return *this;
}
```


Example

- `operator+=` is a member function.
- We can also define a nonmember function that adds two numbers.

```
Complex operator + (const Complex &o1,  
                  const Complex &o2)  
{  
    Complex rst;  
    rst.real = o1.real + o2.real;  
    rst.imag = o1.imag + o2.imag;  
    return rst;  
}
```

- However, there is a problem with this. What is it?
- Since `operator+` is a nonmember function, it cannot access the private data members.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

The function **baz** has access to **f**, which would otherwise be private to class **foo**.

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

Note: a friend function is **NOT** a member function; it is an ordinary function.

Note: NOT `void foo::baz() { ... }`

Friend

- So, we'll need some other mechanism to make the function as a "**friend**".
- The "friend" declaration allows you to expose the **private** state of one class to another function (and only that function) explicitly.

```
class foo {  
    friend void baz();  
    int f;  
};  
void baz() { ... }
```

Note: "friend void baz ();" goes inside foo. It means foo gives friendship to function baz ().

Friend

- Besides function, we can also declare a class to be friend.

```
class foo {  
    friend class bar;  
    int f;  
};  
class bar {  
    ...  
};
```

Then, objects of class `bar` can access private member `f` of `foo`.

Friend

```
class foo {  
    friend class bar;  
    friend void baz();  
    int f;  
};  
class bar { ... };  
void baz() { ... }
```

Friendship of both
class and function.

- Note: Although “**friendship**” is declared inside **foo**, **bar** and **baz()** are not the members of **foo**!
- “**friend**” declaration may appear anywhere in the class.
 - It is a good idea to **group** friend declarations **together** either at the beginning or end of the class definition.

Example

- In our example of complex number class, we will declare **operator+** as a friend:

```
class Complex {  
    // OVERVIEW: a complex number class  
    double real;  
    double imag;  
public:  
    Complex(double r=0, double i=0);  
    Complex &operator += (const Complex &o);  
    friend Complex operator+(const Complex &o1,  
                             const Complex &o2);  
};
```

Its implementation is the same as before.

Overloading Operator []

- We want to access each individual element in the IntSet through **subscript operator []**, just like how we access an ordinary array.
 - For example, **is[5]** accesses the sixth element in the IntSet **is**.
- We need to overload the **operator []**.
 - It is a binary operator: The first operand is the IntSet object and the second one is the index.

Overloading Operator []

- We write two versions with bound checking

```
const int &IntSet::operator[](int i) const {  
    if(i >= 0 && i < numElts) return elts[i];  
    else throw -1;  
}
```

const version returning a const reference to int

```
int &IntSet::operator[](int i) {  
    if(i >= 0 && i < numElts) return elts[i];  
    else throw -1;  
}
```

nonconst version returning a reference to int

Overloading Operator []

- Why we need a nonconst version that returns a reference to int?
 - We need to assign to an element through subscript operation
`is[5] = 2;`
- Why we need a const version that returns a const reference to int?
 - We may call the subscript operator with some const IntSet objects or within some const member function. Const objects/const member function can only call their const member functions.
 - Furthermore, the return type should be const reference because we cannot use a const object (`elts[i]` in this case is a const int) to initialize a non-const reference.
- Which version is called if both are OK?
 - Answer: the non-const version.

Overloading Output Operator <<

- We want to redefine the **operator<<** for the `IntSet` class, so that it prints all the elements in the set in sequence.
- Convention of the IO library
 - The **operator<<** should take an **ostream&** as its first parameter and a **const** reference to a object of the class type as its second.

```
os << obj;
```

- The **operator<<** should return a reference to its **ostream parameter**.

```
ostream &operator<<(ostream &os, const IntSet &is){  
    ...  
    return os;  
}
```

Overloading Output Operator <<

```
ostream &operator<<(ostream &os, const IntSet &is){  
    ...  
    return os;  
}
```

- Why should **operator<<** return a reference to its **ostream parameter**?
 - Because **operator<<** can be **chained together**:
`cout << "hello " << "world!" << endl;`
 - It is equivalent to
`cout << "hello ";`
`cout << "world!";`
`cout << endl;`

Overloading Output Operator <<

- **operator<<** must be a nonmember function!
 - The first operand is not of the class type.
- We can implement **operator<<** as follows

```
ostream &operator<<(ostream &os, const IntSet &is){  
    for(int i = 0; i < is.size(); i++)  
        os << is[i] << " ";  
    return os;  
}
```

Question: Which version of operator[] is called?

- Now we can write **cout << is << endl;**

Do we need to declare << as a friend of IntSet?

Overloading Input Operator >>

- Convention of the IO library
 - The **operator>>** should take an **istream&** as its first parameter and a **nonconst** reference to a object of the class type as its second.

Question: why nonconst?

```
is >> obj;
```
 - The **operator>>** should return a reference to its **istream parameter**.

Question: why returning reference?

```
istream &operator>>(istream &is, foo &obj){  
    ...  
    return is;  
}
```

Outline

- Operator overloading
- Linear List
- Stack
 - Implementation
 - Application

Linear List ADT

- Recall the IntSet ADT
 - A collection of zero or more integers, with **no duplicates**.
 - It supports insertion and removal, but by value.
- A related ADT: linear list
 - A collection of zero or more integers; **duplicates possible**.
 - $L = (e_0, e_1, \dots, e_{N-1})$
 - It supports insertion and removal **by position**.

Linear List ADT

Insertion

```
void insert(int i, int v) // if  $0 \leq i \leq N$   
// (N is the size of the list), insert v at  
// position i; otherwise, throws BoundsError  
// exception.
```

Example: $L1 = (1, 2, 3)$

$L1.insert(0, 5) = (5, 1, 2, 3);$

$L1.insert(1, 4) = (1, 4, 2, 3);$

$L1.insert(3, 6) = (1, 2, 3, 6);$

$L1.insert(4, 0)$ throws **BoundsError**

Linear List ADT

Removal

```
void remove(int i) // if  $0 \leq i < N$  (N is  
// the size of the list), remove the i-th  
// element; otherwise, throws BoundsError  
// exception.
```

Example: `L2 = (1, 2, 3)`

`L2.remove(0) = (2, 3);`

`L2.remove(1) = (1, 3);`

`L2.remove(2) = (1, 2);`

`L2.remove(3) throws BoundsError`

Outline

- Operator overloading
- Linear List
- Stack
 - Implementation
 - Application

Stack

- A “pile” of objects where new object is put on **top** of the pile and the top object is removed first.
 - LIFO access: last in, first out.
 - Restricted form of a **linear list**: insert and remove only at the end of the list.



Methods of Stack

- **size()** : number of elements in the stack.
- **isEmpty()** : checks if stack has no elements.
- **push(Object o)** : add object **o** to the top of stack.
- **pop()** : remove the top object if stack is not empty; otherwise, throw **stackEmpty**.
- **Object &top()** : return a reference to the top element.

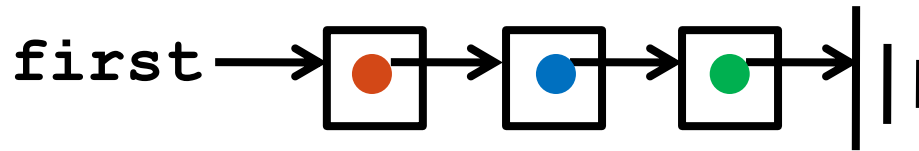
Stacks Using Arrays

Array[**MAXSIZE**] :

2	3	1	4		
---	---	---	---	--	--

- Maintain an integer **size** to record the size of the stack.
- **size():return size;**
- **isEmpty():return (size == 0);**
- **push(Object o):** add object **o** to the end of the array and increment **size**. Allocate more space if necessary.
- **pop():** If **isEmpty()**, throw **stackEmpty**; otherwise, decrement **size**.
- **Object &top():** return a reference to the top element **Array[size-1]**

Stacks Using Linked Lists

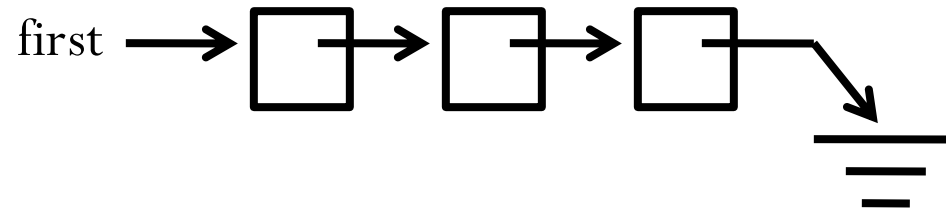


For single-ended linked list, which end is preferred to be the top? Why?

- **`size() : LinkedList::size() ;`**
- **`isEmpty() : LinkedList::isEmpty() ;`**
- **`push(Object o) :`** insert object at the beginning
`LinkedList::insertFirst(Object o) ;`
- **`pop() :`** remove the first node
`LinkedList::removeFirst() ;`
- **`Object &top() :`** return a reference to the object stored in the first node.

LinkedList::size()

- How to get the size of a linked list?



```
int LinkedList::size() {  
    int count = 0;  
    node *current = first;  
    while(current) {  
        count++;  
        current = current->next;  
    }  
    return count;  
}
```


Array vs. Linked List: Which is Better?

- They both have advantages and disadvantages
- Linked list
 - memory-efficient: a new item just needs extra constant amount of memory
 - not time-efficient for size operation
- Array
 - time-efficient for size operation
 - not memory-efficient: need to allocate a big enough array

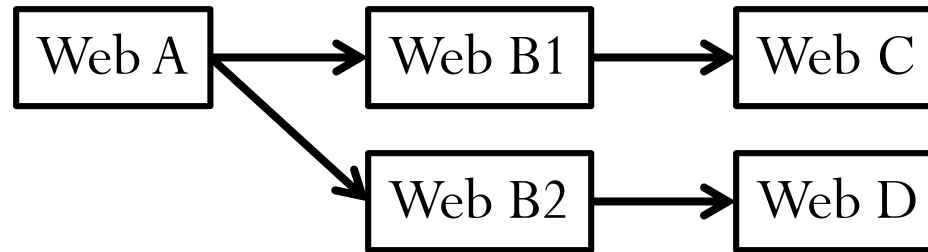
Outline

- Operator overloading
- Linear List
- Stack
 - Implementation
 - Application

Application of Stacks

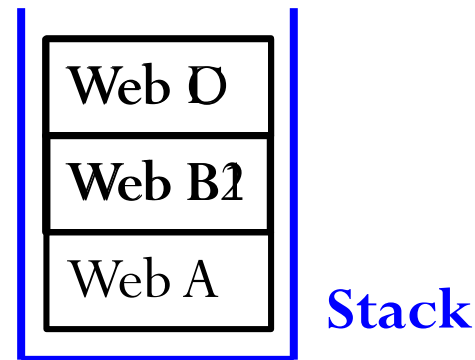
- Function calls in C++
- Web browser's “back” feature
- Parentheses Matching

Web Browser's “back” Feature



Visiting order

- Web A
- Web B1
- Web C
- Back (to Web B1)
- Back (to Web A)
- Web B2
- Web D



Parentheses Matching

- Output pairs (u, v) such that the left parenthesis at position u is matched with the right parenthesis at v .

$((a + b) * c + d - e) / (f + g)$
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18

- Output is: $(1, 5); (0, 12); (14, 18);$

$(a + b)) * ((c + d)$
0 1 2 3 4 5 6 7 8 9 10 12

- Output is

$(0, 4);$

Right parenthesis at 5 has no matching left parenthesis;

$(8, 12);$

Left parenthesis at 7 has no matching right parenthesis

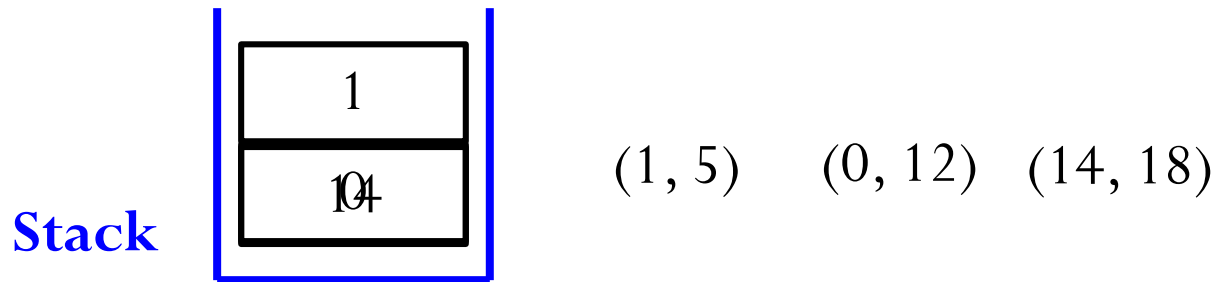
How to Realize Parentheses Matching?

((a + b) * c + d - e) / (f + g)
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18

- Scan expression from left to right.
- When a **left** parenthesis is encountered, push its position to the stack.
- When a **right** parenthesis is encountered, pop the top position from the stack, which is the position of the **matching left** parenthesis.
 - If the stack is empty, the **right** parenthesis is not matched.
- If string is scanned over but the stack is not empty, there are not-matched **left** parentheses.

Parentheses Matching

((a + b) * c + d - e) / (f + g)
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18



Reference

- **C++ Primer (4th Edition)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
 - Chapter 12.5 **Friends**
 - Chapter 14 **Overloaded Operations and Conversions**