Lecture Notes on
C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

- Implementing DATE with a class
- C++ Extensions
  - Namespaces
  - C++ I/O Basics — iostream iomanip fstream
  - Reference — void swap(int& x, int& y)
  - Extensions on C++ Functions
    - Inline Function V.S. Macro
    - Default Arguments
    - Overloading Functions
  - New Delete
  - Exception Handling

- Abstract Data Type
- Class and Object
- Constructor
  - default constructor　　Constructors with default parameters
  - normal constructor
  - copy constructor　　C::C(const C& obj);
- Destructor
  - 对象指针　delete
  - shallow copy　　deep copy
- const
  - const Member Functions
  - const Member Data
  - const object
- 静态（static）成员
- this指针
- Composition

Operator Overloading

(运算符重载)

Operator Overloading

- 类成员运算符重载　　类型 类名::operator 运算符(参数表)
  下标运算符[ ]的重载
- 友元运算符重载　　游离函数
  其它类
  其他类的成员函数
- **Input/Output Overload**
- **Increment/decrement Overload**
- **Overloading function all operator: ()**

运算符重载 (Operator Overloading)

- 对运算符语义的重新定义（即，在不同情况下对同样的运算符做不同的解释）
- 运算符函数：运算符可看作函数

operand1 op operand2 可理解为

op(operand1, operand2) 例如

a+b => +(a,b) => Add(a, b)

[二元运算符是一个具有两个参数的函数

一元运算符是一个具有一个参数的函数]

运算符重载（续）

- 运算符重载的作用：用自然的方式使用用户自定义的类类型。①与基本类型的使用风格一致；②提高程序的可理解性
- 运算符重载的两种形式
  - 类成员运算符重载
  - 友元运算符重载
- 不能重载的运算符：

  ::　　.*　　?:　　.　　sizeof

类成员运算符重载

- 在类中定义运算符函数
- 一般形式：

```
类型  类名::operator 运算符( 参数表 )
{
        …     // 运算符函数体
}
```

类成员运算符重载（续）

- 重载二元运算符时，成员运算符函数只需显式传递一个参数（即二元运算符的右操作数），而左操作数则是该类对象本身，通过**this**指针隐式传递。

c1 + c2 ; ⇔ c1.operator +(c2);

- 重载一元运算符时，成员运算符函数没有参数，操作数是该类对象本身，通过**this**指针隐式传递。

-c1 ; ⇔ c1.operator - ( );

例：复数类

```
class COMPLEX {       // 定义复数类COMPLEX的类界面
public:
        COMPLEX(double r = 0, double i = 0);  // 构造函数一
        COMPLEX(const COMPLEX& other);   // 构造函数二
        void print();                              // 打印复数
        // 与另一个复数相加
        COMPLEX add(const COMPLEX& other);
        // 减去另一个复数
        COMPLEX substract(const COMPLEX& other);
protected:
        double real, image;  // 复数的实部与虚部
};
```

例：复数类

```
COMPLEX COMPLEX::add(const COMPLEX& other)
{
    real = real + other.real;
    image = image + other.image;
    return* this;
}

COMPLEX COMPLEX::substract(const COMPLEX& other)
{
    real = real - other.real;
    image = image - other.image;
    return* this;
}
```

例：复数类

```
int main()
{
    COMPLEX c1(1, 2);    // 定义一个值为1 + 2i的复数c1
    COMPLEX c2(2);       // 定义一个值为2的复数c2

 // 用COMPLEX(const COMPLEX& other)创建一个值同c1的新复数
    COMPLEX c3(c1);

    c3.print();                  // 打印c3原来的值
    c2.add(c1);                  //将c2加上c1
    c3.substract(c2);            // 将c3减去c2
    c3.print();                  // 再打印运算后c3的值

    return 0;
}
```

更多时候，人们希望能使用更贴近习惯的方式

c2 = c2+c1　　c3 = c3 – c2

例：利用了运算符重载机制的复数类

```
class COMPLEX {                                          //complex.h
public:
        COMPLEX(double r = 0, double i = 0);   // 构造函数
        COMPLEX(const COMPLEX& other);    // 拷贝构造函数
        void print();                                    // 打印复数
       // 重载加法运算符（二元）
        COMPLEX operator + (const COMPLEX& other);
       // 重载减法运算符（二元）
        COMPLEX operator - (const COMPLEX& other);
        COMPLEX operator - ();   // 重载求负运算符（一元）
        // 重载赋值运算符（二元）
        COMPLEX operator = (const COMPLEX& other);
protected:
        double real, image;  // 复数的实部与虚部
};
```

例：利用了运算符重载机制的复数类

```
// 程序：COMPLEX.CPP

#include "complex.h"
#include <iostream>

COMPLEX::COMPLEX(double r, double i)
{
        real = r;
        image = i;
}

COMPLEX::COMPLEX(const COMPLEX& other)
{
        real = other.real;
        image = other.image;
}
```

例：利用了运算符重载机制的复数类

```
void COMPLEX::print()
{
        cout << real;
        if (image > 0) cout << "+" << image << "i";
        else if (image < 0) cout << image << "i";
        cout << "\n";
}
```

例：利用了运算符重载机制的复数类

```
COMPLEX COMPLEX::operator + (const COMPLEX& other)
{
        COMPLEX temp;
         temp.real = real + other.real;
         temp.image = image + other.image;
         return temp;
}

COMPLEX COMPLEX::operator - (const COMPLEX& other)
{
        COMPLEX temp;
         temp.real = real - other.real;
         temp.image = image - other.image;
         return temp;
}
```

例：利用了运算符重载机制的复数类

```
COMPLEX COMPLEX::operator - ()
{
    COMPLEX temp;

    temp.real = -real;
    temp.image = -image;
    return temp;
}

COMPLEX COMPLEX::operator = (const COMPLEX& other)
{
    real = other.real;
    image = other.image;
    return *this;
}
```

CPXDEMO.CPP 演示复数类COMPLEX的用法

```
#include "complex.h"
#include <iostream.h>

int main()
{
    COMPLEX c1(1, 2); // 定义一个值为1 + 2i的复数c1
    COMPLEX c2(2);    // 定义一个值为2的复数c2
    COMPLEX c3(c1);   // 用拷贝构造函数创建一个值同c1的新复数

    c3.print();              // 打印c3的值
    c1 = c1 + c2 + c3;       // 将c1加上c2再加上c3赋值给c1
    c2 = -c3;                // c2等于c3求负
    c3 = c2 - c1;            // c3等于c2减去c1
    c3.print();              // 再打印运算后c3的值

    return 0;
}
```

该赋值表达式的值为与c3相等的临时对象

类成员运算符重载（续）

- 赋值运算符的重载
  - 若某个类没有重载赋值运算符，则编译器将自动生成一个缺省的赋值运算符，缺省赋值运算符采用浅复制策略，把源对象逐位地拷贝到目标对象：

```
c1 = c2 ;
```

  - 不含指针成员的类使用缺省赋值运算符即可，含指针成员的类应重载赋值运算符，实现深复制策略。

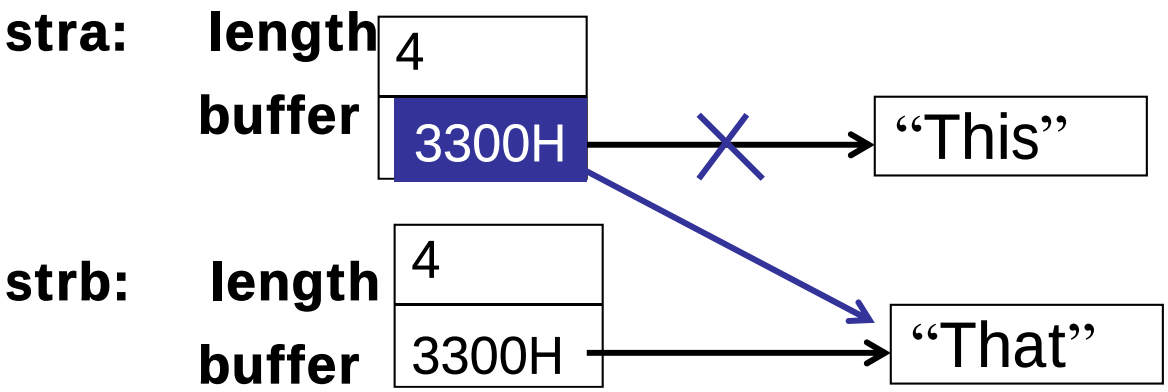
例：自定义的字符串类

```
#include <iostream>
using namespace std;

class STRING {
private:
     int     length;
     char  *buffer;
public:
     STRING();
     STRING(const char *str);
     ~STRING();
};
```

```
STRING::STRING()
{
     length = 0;
     buffer = NULL;
}
STRING::STRING(const char *str)
{
     length = strlen(str);
     buffer = new char[length+1];
     if (buffer!=NULL)
          strcpy(buffer,str);
}
STRING::~STRING()
{
     if (buffer)
          delete []buffer;
}
```

例：自定义的字符串类

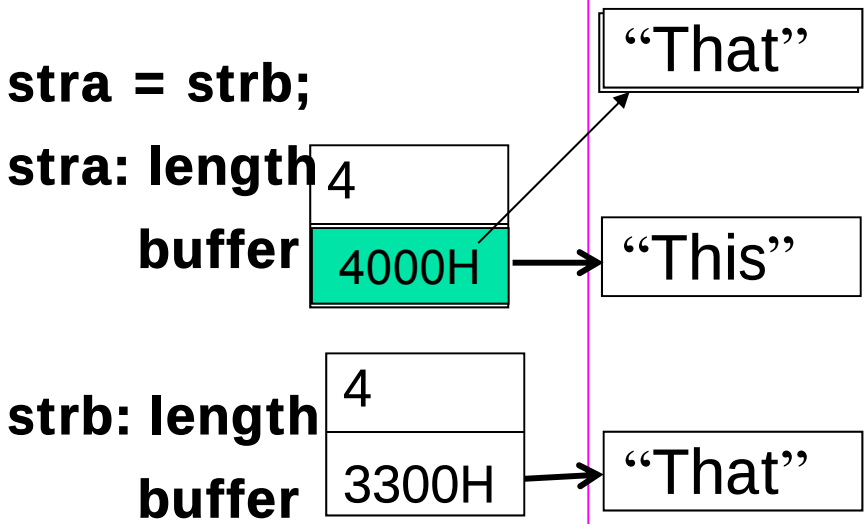
```
void main()
{
    STRING stra("This");
    STRING strb("That");
    stra = strb;
}
```



赋值后**stra**的数据成员**buffer**原来指向的字符串所占用的存储块永久丢失，而且撤销**stra**和**strb**时会导致同一块内存重复释放的问题。

解决方法：重载赋值运算符

```
STRING& STRING::operator=(const STRING &another)
{
    length = another.length;

    if (buffer)
       delete []buffer;

    buffer = new char[length+1];
    if (buffer!=NULL)
       strcpy(buffer,another.buffer);

    return *this;
}
```



- 先释放指针原来指向的内存空间，然后再重新分配内存空间，最后再把需要拷贝的内容置入这个空间内。既避免了内存垃圾，也避免了指针重名。这就是深复制（**deep copy**）。

下标运算符[ ]的重载

- 形参为整型
- 返回值类型为引用类型

例：向量类。演示下标运算符[ ]的重载

```
const int MAX_SIZE = 10;            // 定义符号常量表示向量的大小
class VECTOR {
public:
    VECTOR() // 构造函数
    {       int loop;
            for (loop = 0; loop <= MAX_SIZE - 1; loop = loop + 1)
                table[loop] = loop;
    }
    int& operator [ ] (int index)   // 取向量元素
    {       if ((index < 0) || (index > MAX_SIZE - 1)) {
                cout << "Index out of bounds.\n";
              exit(1);
          }
            return table[index];
    }
protected:
            int table[MAX_SIZE];  // 向量的内容
};
```

例：向量类。演示下标运算符[ ]的重载

```
int main()
{
    VECTOR label;   // 定义向量对象

    cout << label[2] << "\n";   // 输出结果为2

    label[2] = 8;                    // 改变第三个分量的值

    cout << label[2] << "\n";   // 输出结果为8

    // 引起程序异常终止，提示Index out of bounds.
    cout << label[10] << “\n”;

    return 0;
}
```

运算符重载（续）

- 运算符重载的作用：用自然的方式使用用户自定义的类类型。①与基本类型的使用风格一致；②提高程序的可理解性
- 运算符重载的两种形式
  - 类成员运算符重载
  - 友元运算符重载
- 不能重载的运算符：

  ::　　.*　　?:　　.　　sizeof

友元（friend）

- 友元（**friend**）关系允许类的设计者选择出一组其他的类或函数，使得它们可以访问该类的私有和受保护成员。
- 在类的声明中，用**friend**声明的函数或类，即是该类的友元。
- 一个类的友元可以是：
  - 游离函数（不属于任何类的函数）
  - 另一个类
  - 其他类的成员函数。
- 友元关系破坏了类的封装性，不可滥用。

游离函数作为友元

```
// 类VALUE中定义了一个友元函数set()，注意set()不是该类的成员函数
class VALUE
{
    public:
        //声明set()为VALUE的友元
        friend void set(VALUE obj, int x);
    private:
        int value;
};

void set(VALUE obj, int x)    // 实现友元函数set()
{
     obj.value = x;  // set()可以象VALUE成员函数一样访
                     // 问obj的私有和受保护成员
}
```

另一个类作为友元

```
class Y;   // Y类的引用性声明
class X {
    public:
        // 把Y类声明为X类的友元，则Y类的所有成员函数都是X的友元
        friend Y;

    private:
        int k ;
        void m_Xfunc( );
};

class Y {
     public:
        void m_Yfunc( X& obj );
};

void Y::m_Yfunc( X& obj )
{
   obj.k = 100 ; // Y类的成员函数是X的友元，可以访问X的私有和受保护成员
}
```

其它类的成员函数作友元

```
class Y {
      public:
       void Yfunc( );
};

class X {
public:
      friend void Y::Yfunc( );   // 把Y类的Yfunc函数声明为X类的友元
private:
       int k ;
       void m_Xfunc( );
};

void Y::Yfunc( )
{
     X   obj;
     obj.k = 100 ; // 该函数是X的友元，可以访问X的私有和受保护成员
}
```

例子

```
class INTEGER {
public:
        INTEGER(int i = 0)    // 构造函数
        {
            value = i;
        }

        INTEGER(const INTEGER& other)       // 拷贝构造函数
        {     value = other.value;   }

        INTEGER operator +(INTEGER other)  // 重载加法运算符

        {
            INTEGER temp;
            temp.value = value + other.value;
            return temp;
        }
private:
        int value;  // 私有数据
};
```

例子

```
int main()
{
    INTEGER x(10);   // 定义整数对象，用构造函数来初始化
    INTEGER y = x;   // 定义整数对象，用拷贝构造函数来初始化
    INTEGER z;       // 定义整数对象，用构造函数的缺省参数初始化

     y = x + 2;      // 合法调用

     z = 30 + y;     /* 不合法，因为第一操作数（二元运算的左操作数）
                        必须是本类对象*/

     return 0;
}
```

友元运算符重载（续）

- 解决类成员函数运算符重载存在的问题：第一操作数（二元运算的左操作数）必须是本类对象。
- 形参设置规则：
  - 一元运算符必须显式声明一个形参。
  - 二元运算符必须显式声明二个形参。
- 下列运算符不能作为友元重载：

  =　　( )　　[ ]　　->
- 友元函数不是该类的成员，因此在友元函数中不能使用**this**指针。

采用友元运算符重载改进INTEGER

```
class INTEGER
{
public:
        INTEGER(int i = 0);       // 构造函数

        INTEGER(const INTEGER& other);    // 拷贝构造函数

        // 友元重载加法运算符
        friend INTEGER operator +(INTEGER left, INTEGER right);

private:
    int value;     // 私有数据

};
```

采用友元运算符重载改进INTEGER

```
INTEGER::INTEGER(int i)
{
        value = i;
}

INTEGER::INTEGER(const INTEGER& other)
{
        value = other.value;
}

INTEGER operator + (INTEGER left, INTEGER right)
{
    INTEGER temp;
    temp.value = left.value + right.value;
    return temp;
}
```

例（续）

```
int main()
{
    INTEGER x(10);    // 定义整数对象，用构造函数来初始化
    INTEGER y = x;    // 定义整数对象，用拷贝构造函数来初始化
    INTEGER z;        // 定义整数对象，用构造函数的缺省参数初始化

    y = x + 2;        // 合法调用
    z = 30 + y;       // 也是合法的调用

    return 0;
}
```

Input/Output Overload

```
int main()
{
    Fruit fruit1;

    cout << fruit1;
    cin >> fruit1;
    cout << fruit1;

    cout << "Finished!" << endl;

}
```

Input/Output Overload

```
#include <string>
#include <iostream>
using namespace std;

class Fruit;  //对于一些编译器，在重载<<和>>时需要将类和友元额外再声明一次
ostream& operator <<(ostream& out, const Fruit& x);
istream& operator >>(istream& in,  Fruit& x);

class Fruit
{
public:
        Fruit();

        friend ostream& operator << (ostream& out, const Fruit& x);
        friend istream& operator >> (istream& in,  Fruit& x);

private:
        string name, color;
};
```

Input/Output Overload

```
// Fruit.cpp

#include <iostream>
#include "Fruit.h"

using namespace std;

Fruit::Fruit()
{
        name = "apple";
        color = "green";
}
```

Input/Output Overload

```
ostream& operator << ( ostream& out, const Fruit& x)
{
        out << "name: " << x.name
              << "  color: " <<x.color << endl;

        return out;
}

istream& operator >> (istream& in, Fruit& x)
{
        cout << "Please enter the name: " << endl;
        in >> x.name;

        cout << "Please enter the color: " << endl;
        in >> x.color;

        return in;
}
```

Increment/decrement Overload

```
class COMPLEX {                                          //complex.h
public:
      COMPLEX(double r = 0, double i = 0);   // 构造函数
      COMPLEX(const COMPLEX& other); // 拷贝构造函数
       void print();                                          // 打印复数

       COMPLEX & operator++();     //重载前置++
       COMPLEX operator++(int);   //重载后置++
       COMPLEX & operator--();      //重载前置--
       COMPLEX operator--(int);     //重载后置--


protected:
          double real, image;   // 复数的实部与虚部
};
```

Increment/decrement Overload

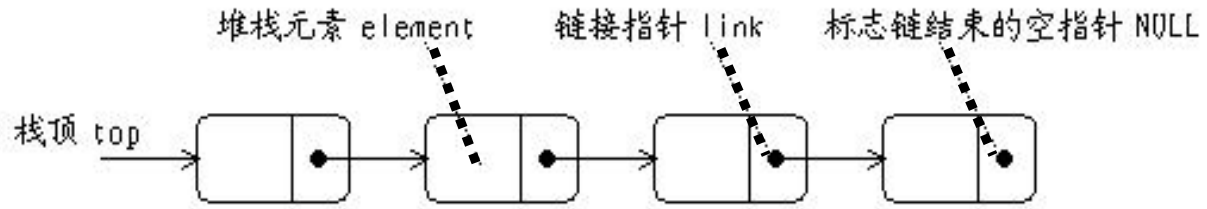
```
COMPLEX& COMPLEX::operator++()      //COMPLEX.CPP
{
        real += 1;
        image += 1;
        return *this;
}

COMPLEX COMPLEX::operator++(int)
{
        COMPLEX before = *this;
        real += 1;
        image += 1;

        return before;
}
```

Increment/decrement Overload

```
COMPLEX& COMPLEX::operator--()      //COMPLEX.CPP
{
        real -= 1;
        image -= 1;
        return *this;
}

COMPLEX COMPLEX::operator -- (int)
{
        COMPLEX before = *this;
        real -= 1;
        image -= 1;

        return before;
}
```

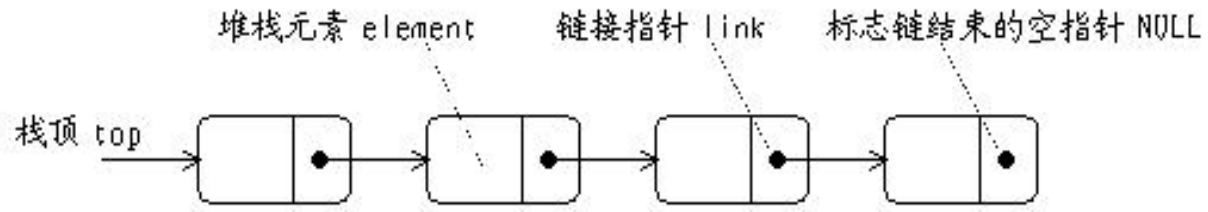
指针的应用：用指针实现堆栈

- 指针的最大作用：构成各种不同的动态数据结构
- 堆栈（**stack**）
  - 后进先出，只能访问栈顶元素的数据结构（如：函数调用堆栈）
  - 堆栈的主要行为
    - **push, pop, get_top, is_empty**
    - **display**（非标准的堆栈行为）
  - 堆栈中存放元素的数据结构
    - 数组
    - 链表



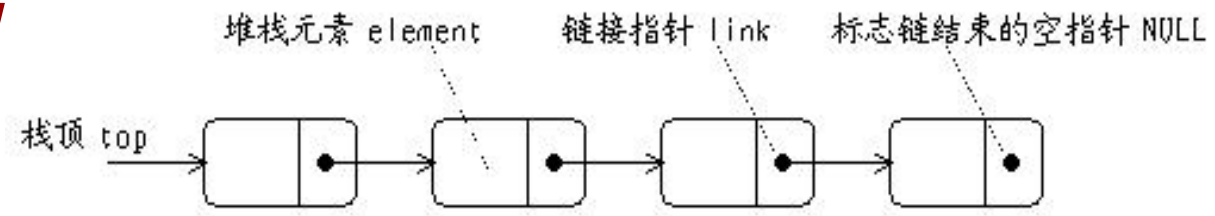
代码 -- 堆栈节点

```
// 堆栈元素类型  程序35
typedef int ELEMENT;  // 为堆栈元素类型起一个别名
struct NODE {
    ELEMENT element;        // 存放堆栈的元素
    NODE* link;             // 指向下一个结点的链接
};
```



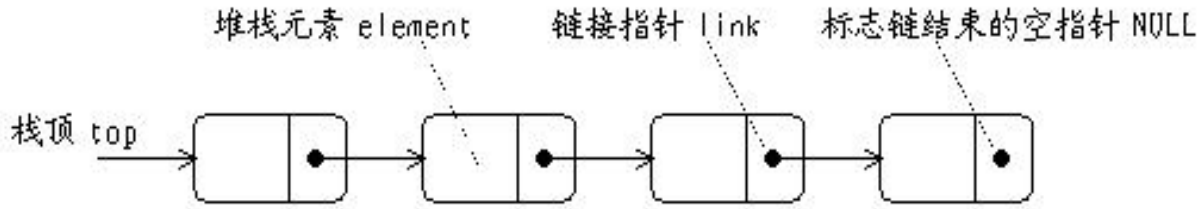
代码--使用指针实现的堆栈类

```
class STACK {
public:
    STACK();  // 构造函数，设置栈顶为空指针
    ~STACK(); // 析构函数，释放堆栈结点占用的存储空间

    void push(ELEMENT obj);  // 将元素obj压入栈顶
    void pop();  // 将当前栈顶的元素弹出栈中。要求：栈不为空。
    ELEMENT get_top();  // 返回当前栈顶的元素值。要求：栈不为空。
    int is_empty();  // 判断当前堆栈是否为空，空则返回1，非空则返回0
    void display();  //
private:
   NODE* top; // 堆栈的栈顶
};
```



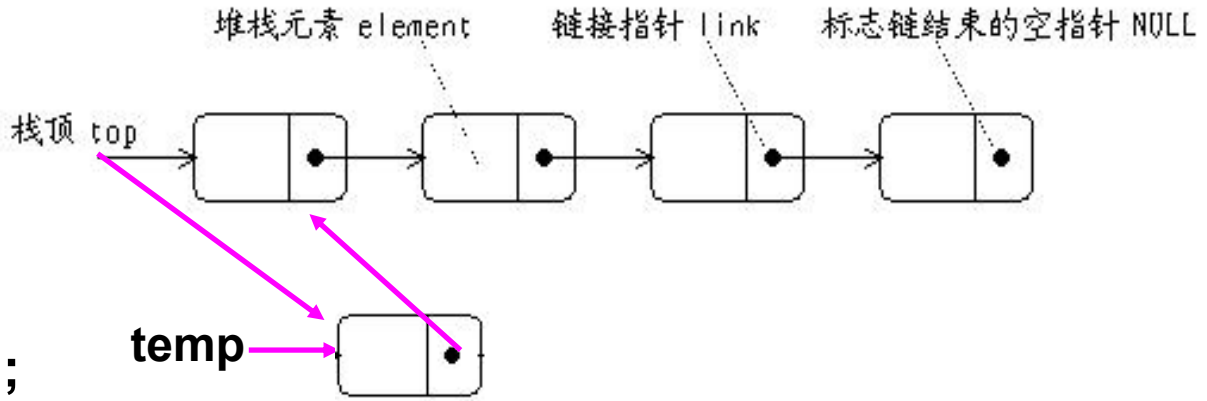
代码

```
STACK::STACK()
{
    top = NULL;              // 将栈顶置为空
}

STACK::~STACK()
{
   NODE* ptr;   // 指向堆栈结点的临时指针

  while (top != NULL)   // 从上到下释放堆栈的结点，注意循环体中语句的次序
  {
      ptr = top;              // 先记住将被摘下来的栈顶结点
      top = top->link;        // 摘下栈顶结点
      delete ptr;             // 释放刚才被摘下来的结点
   }
}
```



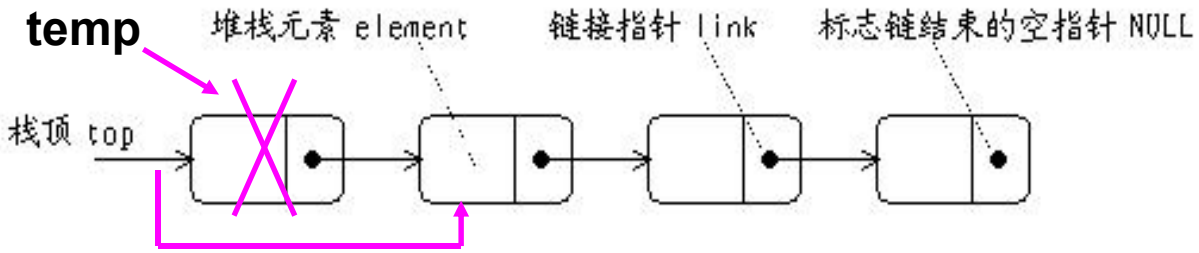
代码

```
void STACK::push(ELEMENT obj)  // 将obj压入堆栈的栈顶
{
   NODE* temp;
   temp = new NODE;
   if (temp != NULL)
  {
      temp->link = top;
      temp->element = obj;
      top = temp;
   }
   else   // 内存分配失败时作异常处理
  {
       cout << "Error: No enough memory.” << endl;
       exit(1);                  // 终止程序
   }
}
```



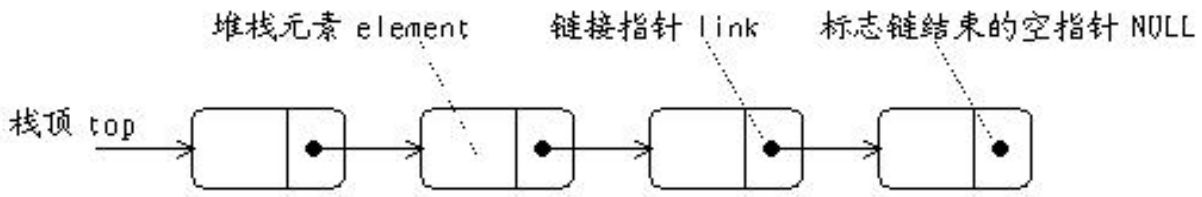
代码

```
void STACK::pop() // 将堆栈当前的栈顶元素弹出
{
     NODE* temp;
     if (top != NULL) //
    {
          temp = top;      // 将栈顶元素弹出堆栈
          top = top->link;
          delete temp;     // 释放被弹出结点占用的存储空间
     }
     else    // 空栈时作异常处理
    {
           cout << "Error: Pop from empty stack.\n";
           exit(1);                    // 终止程序
     }
}
```



代码

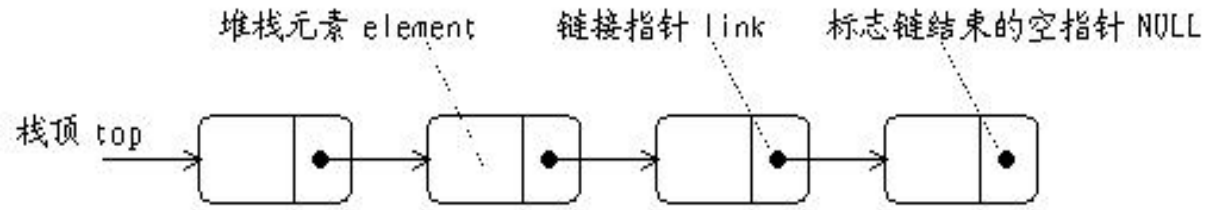
```
ELEMENT STACK::get_top()  // 返回当前栈顶的元素值
{
    if (top == NULL)    // 空栈时作异常处理
    {
        cout << "Error: Get top from empty stack.\n";
        exit(1);        // 终止程序
    }
    return top->element;
}
```




```
int STACK::is_empty() // 判断当前堆栈是否为空，空则返回1，非空则返回0
{
    return (top == NULL);
}
```

代码

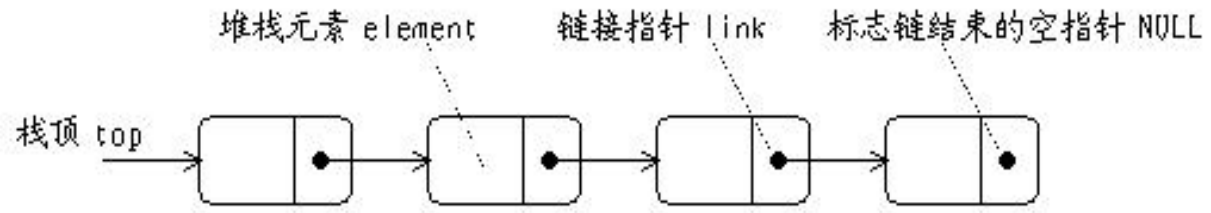
```
void STACK::display()  // 自顶向下显示堆栈中的元素
{
    NODE* loop;
```




```
    loop = top;
    while (loop != NULL)  // 以空指针作为链的结束标记
    {
        cout << loop->element << " ";    // 将当前结点的元素输出
        loop = loop->link;                // 指向下一个结点
    }
    cout << endl;
}
```

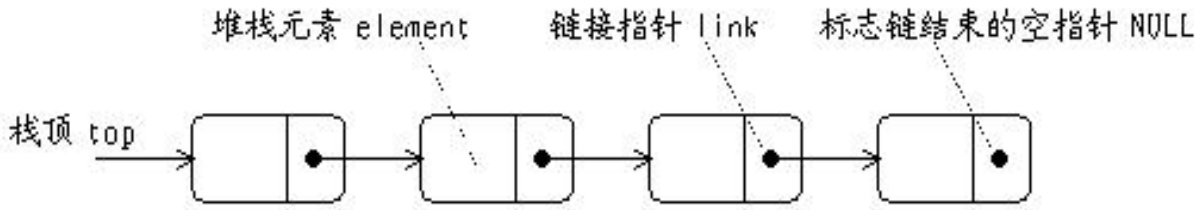
客户代码

```
STACK turner;  // 声明一个元素为整数类型的堆栈
ELEMENT user_input;     // 用户输入的元素（即整数类型）
int loop;               // 循环变量

for (loop = 1; loop <= max_input; loop++)  // 由用户输入若干个元素
{
    cout << "Input no." << loop << ": ";   // 给出输入提示
    cin >> user_input;      // 由用户输入一个元素
    turner.push(user_input);                    // 将用户输入的元素压入栈中
}
```



客户代码

```
turner.display(); // 测试堆栈中的内容

for (loop = 1; loop <= max_input; loop++)
{
      if (! turner.is_empty()) // 仅当堆栈不为空时才处理
      {
           user_input = turner.get_top();       // 取出栈顶元素的值
           turner.pop();                        // 将栈顶元素弹出
           cout << "Output no." << loop << ": ";  // 输出取出来的栈顶元素的值
           cout << user_input << endl;
      }
}
```



Overloading function all operator: ()

```
#include <iostream>
using namespace std;

class Matrix{
private
        double v[3][3];
public:
        Matrix(double [][3], int);
        Matrix() {}
        double& operator()(int, int);
};
```

Overloading function all operator: ()

```
Matrix::Matrix(double ve[][3], int r_size)
{
    if (r_size != 3)
        out << " Error!!";
    else
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                v[i][j] = ve[i][j];
}

double& Matrix::operator()(int i, int j)
{
     return v[i][j];
}
```

Overloading function all operator: ()

```
void main()
{
        double a[3][3];
        int i, j;

        for(i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
             cin >> a[i][j];

        Matrix M(a, 3);

        for (i = 0; i < 3; i++)
        {
                for (j = 0; j < 3; j++)  cout << M(i, j) <<" ";

                cout << endl;
        }
}
```

Overloading function all operator: ()

- ***operator()*** **: Most obvious and probably most important usage—provides the usual *function call* syntax for objects.**
- **Such objects act like functions called *function object*.**
- **An *operator()* must be a member function**

Overloading function all operator: ()

```
#include <iostream>
using namespace std;
class Exchange
{
   public:
        void operator()(int&, int&);
};

void Exchange::operator()(int& x, int& y)
{
        int t;
        t = x;
        x = y;
        y = t;
}
```

Overloading function all operator: ()

```
int main()
{
  Exchange swap;
  int i = 20;
  int j = 30;

  cout << "Before swap:i=" << i << ", j=" << j << endl;

  swap(i, j);

  cout << "After swap:i=" << i << ", j=" << j << endl;

  return 0;
}
```

- Conversion between Derived-Class Object and Base-Class Object
- Name Hiding and Overriding
- protected Members
- protected and private Inheritance
- Constructors and Destructors Under Inheritance
- Multiple Inheritance
- Relationships between Classes