中山大學
SUN YAT-SEN UNIVERSITY

数据抽象和类

中山大学计算机学院

主讲人：

指导老师：

目录

CONTENTS

传值和传引用

C++的函数参数传递方式，可以是传值方式，也可以是传引用方式。

传值的本质是：形参是实参的一份复制。

传引用的本质是：形参和实参是同一个对象。

```
void fun_1(int a);     //int类型，传值（复制产生新变量）
void fun_2(int& a);    //int类型，传引用（形参和实参是同一个东西）
void fun_3(int* arr); //指针类型，传值（复制产生新变量）
void func_4(int*& arr); //指针类型，传引用（形参和实参是同一个东西）
```

传值和传引用

```
//code15
void swap_pass_by_reference(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}

void swap_pass_by_value(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
originally
a=1, b=2
after swap_pass_by_reference
a=2, b=1
after swap_pass_by_value
a=1, b=2
```

```
int main() {
    int a = 1, b = 2;
    cout << "originally" << endl;
    cout << "a=" << a << ", b=" << b << endl;

    swap_pass_by_reference(a, b);
    cout << "after swap_pass_by_reference" << endl;
    cout << "a=" << a << ", b=" << b << endl;

    swap_pass_by_value(&a, &b);
    cout << "after swap_pass_by_value" << endl;
    cout << "a=" << a << ", b=" << b << endl;

    return 0;
}
```

常（const 限定）引用

首先进一步理解引用：

int &a=b 可以理解为 **int *const a=b**。

即引用是一个指针常量（又称常指针，即一个常量，其类型是指针）。

每当编译器遇到引用变量a，就会自动执行 * 操作。

常引用：**const int &a=b**就相当于 **const int * const a=b**。

不仅仅是a这个地址不可修改，而且其指向的内存空间也不可修改。

常引用

```
//code16

int const_ref_without_casting() {
    int b = 10;
    const int &a = b;
//  a = 12;  //error: assignment of
read-only reference
    b = 11; //这可以改!
    printf("a=%d,b=%d\n", a, b);
}
```

```
int const_ref_with_constant() {
    // (2) const引用 常量
    const int &c = 15;
    //编译器会给常量15开辟一片内存，并将引用名作为这片
内存的别名
//  int &d=15;  //error: invalid initialization of
non-const reference of type 'int&' from an rvalue
of type 'int'
    int* p = (int *)&c;  // must casting explicitly
    *p = 10;
    cout << c << endl;

}
```

常引用

```
int const_ref_with_casting() {
    double b = 3.14;
    const int &a = b; // equal to: int temp = b; const int &a = temp
    b = 11;
    printf("a=%d,b=%f\n", a, b);
}
```

常引用做了类型转换后，就没有关联了，所以b的值改变后a没变

```
// const 限定引用参数
int const_ref_in_para(const string& s)
{
    cout << s << endl;
}

int no_const_ref_in_para(string& s) {
    cout << s << endl;
}
```

```
int main()
{
    const_ref_without_casting();
    const_ref_with_constant();
    const_ref_with_casting();

    const_ref_in_para("I love c++");
//  no_const_ref_in_para("I love c++"); //error
}
```

输出：

```
a=11,b=11
10
a=3,b=11.000000
I love c++
```

常（const 限定）方法

```
class Test {
public:
        void Test1(int _a)const {  //常方法 其中this指针由Test* const--> const Test*const
                cout<<"Test1(int) const"<<endl;
//              a = _a;            //error  常方法不能修改普通成员变量的值
                int d = _a;         //可以访问a，但不能修改a的值
                int e = c;            // const函数访问const成员变量
                }
         void Test2(int x) {
                int mf = c;         //普通函数访问const成员变量
                a =  x ; //变量成员可以被修改
                cout<< "Test2(int)" << endl;  //输出该函数
                }
        Test(int _a,int _b,int _c):a(_a),b(_b),c(_c){  //构造函数(常成员只能由初始化表进行初始化)
                std::cout<< "Test(int,int,int)" <<std::endl;     //输出该函数
                }
private:
   int a, b;
   const int c; //C++中常变量必须初始化（类中的常变量可以使用构造函数的初始化表进行初始化）
};
```

常方法

```
int main()
{
    Test tmp(10,20,30);      //调用构造函数生成对象tmp
    tmp.Test1(10);           //const函数访问常数据成员变量
    tmp.Test2(10);             //普通函数访问常数据成员变量
    return 0;
}                            //code18
```

常对象：常对象只能调用该类中的常方法

常方法：

①可以访问对象中的常成员，也可以访问普通成员

②该方法不允许修改任何数据的值

常数据成员：

①只能通过构造函数的初始化表就行初始化，其他方式皆不允许

②可以被const成员函数访问，也可以被该类中普通函数访问（但不允许修改其中的值）

目录

CONTENTS

以下请先自学

案例研究：链表（Linked Table）与栈（Stack）

- **链表**是一种数据结构用于存储序列数据
- 链表的特征
  - 有头指针指向 LinkNode 类型对象
  - LinkNode对象包含指向下一个的指针
  - 直到 NULL
- LinkNode 类型
  - 包含数据成员保存用户数据
  - LinkNode 类型指针
- 链表与数组都是保存一组数据，哪个好？

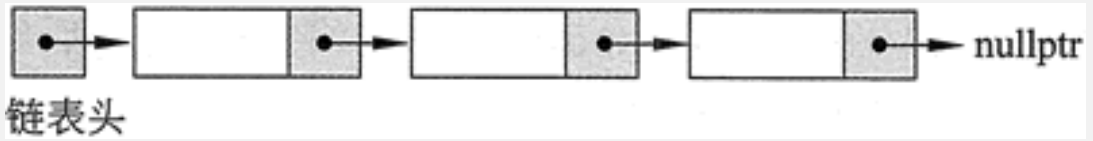


图 1 链表结构图解

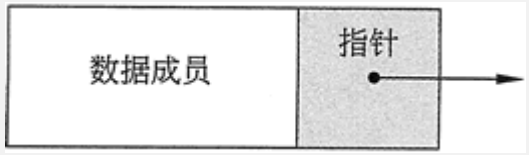


图 2 链表结点的结构

案例研究：链表节点（Linked Node）定义

link-node.hpp

- 请正确书写头文件，避免循环嵌套
- 使用 typedef 使得程序更通用
- 在类名使用下划线开头，表明不希望普通开发者使用它。
- = delete 禁止使用默认构造
- 添加了一些新的操作

问题：

- 链表有哪些基本操作？
- 链表节点是否需要深复制？（这里有 next 指针）

```
#ifndef LINK_NODE_H
#define LINK_NODE_H
typedef int UserData;
class _LinkNode {
    public:
        _LinkNode() = delete;
        _LinkNode(UserData item): item(item) {};
        // … get & set
        // append item after *this, return this->next
        _LinkNode* insert(UserData item);
        // delete item after *this, return this->next
        _LinkNode* erase();
        //new item before the *head, return new head
        static _LinkNode* Push(_LinkNode *head, UserData item);
        //delete *head, return head->next as new head
        static _LinkNode* Pop(_LinkNode *head);
    private:
        UserData item;
        _LinkNode *next;
};
#endif
```

便于修改（只用改这的数据类型）

类声明加哨兵指令

案例研究：链表的基本操作

link-node.cpp

基本操作就两个：

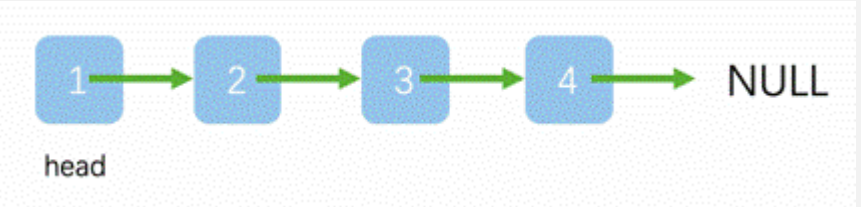
- Push 给定链表头的链表，创建新的表头节点，新节点的next指向原链表，返回新的表头
- Pop 给定链表头的链表，如果不是空链表，新的表头指向表头的下一个节点，释放原表头节点，返回新的表头

实现选择

- 普通函数（如c语言）
- 节点静态成员（本案例选择）
- 高级数据结构如栈、列表的成员函数

```
_LinkNode* _LinkNode::Push(_LinkNode *head, UserData item) {
    auto new_head = new _LinkNode(item);
    new_head->setNext(head);
    return new_head;
}

_LinkNode* _LinkNode::Pop(_LinkNode *head) {
    if (head) {
        auto new_head = head->next;
        delete head;
        return new_head;
    }
    return nullptr;
}
```



案例研究：链表的常用操作

link-node.cpp

常用操作很多，这里只写了两个

- insert 在链表节点后插入。把 this->next 看作子链表头（注：链表是递归的定义），则直接调用前面的 Push 即可
- erase 在链表节点后删除，方法类似

注意事项：

- 链表是单向的，无法在前面插入
- 不能直接删除自身节点

```
_LinkNode* _LinkNode::insert(UserData item) {
    this->next = Push(this->next,item);
    return this->next;
}

_LinkNode* _LinkNode::erase() {
    this->next = Pop(this->next);
    return this->next;
}
```

基础训练（leetcode）：

21. 合并两个有序链表

83. 删除排序链表中的重复元素

206. 反转链表

237. 删除链表中的节点

案例研究：指针应用的调试

W6-link-node.dev

指针应用常见问题：

1. 结果正确，内存测试不能通过
2. 有结果，但输出乱码
3. 无结果，终止代码不为 0

项目简介

- w6-test-mem.cpp 内存检测工具
- w6-linknode hpp 和 cpp
- w6-linknode-test.cpp 测试驱动的程序案例

内存测试不能通过：

1. 将 test-mem.cpp 加入项目
2. 测试（主）程序添加 extern int new_del_count;
3. 在合适位置检查 new_del_count 值。0 表示 new 和 delete 配对
4. 如果认为指示不清，修改test-mem.cpp 。去除注释 `//#define PRINT_MEM`
5. 会打印所有 new 和 delete 信息。检查是否 new[] 和 delete 错配等

案例研究：链表操作与问题追踪

W6-link-node.dev

- w6-linknode-test.cpp

1. 分析测试程序，如右
2. 请打开 main 函数中注释，程序能部分执行，但异常终止。显然，2，3 两种情况就复杂多了
3. 程序检查大致可分为
   - 无复制构造
   - 深复制策略构造

test_link_node 函数问题分析：

14-17 Push 创建 node_1=[1,5,9]

19-24 temp 指针遍历链表，检查

26-27 插入操作 node_1=[1,3,5,7,9]

29-36 temp 指针遍历链表，检查

38-39 删除操作 node_1=[1,5,9]

41-45 temp 指针遍历链表，检查

47-48 Pop 2次 node=[9]

49 检查是否 9

结果，程序退出前 9 遗留内存且无法访问，请改正程序，使得测试程序正常退出

中山大學
SUN YAT-SEN UNIVERSITY

案例研究：链表与栈（Stack）

w6-link-stack-test.dev

项目简介

- w6-test-mem.cpp 内存检测工具
- w6-linknode hpp 和 cpp
- w6-linksatck.hpp 栈的链表实现
- w6-arraystack.hpp 栈的数组实现
- w6-stack-test.cpp 测试程序

Stack 编程难度分类

- 不支持复制和赋值
- 支持复制和赋值
- 栈元素是基本数据类型
- 栈元素已实现深拷贝，如 string
- 栈元素是需要实现深拷贝，如包含 cstring
- 栈包含 static 或 const 数据成员

栈基础：

栈就像一堆书，你只能取最上面一本；或在上面再放一本；或看最上面的书名。你甚至无法知道这堆书有多少本。然而这个简单的特性，却是解决许多问题的关键。如函数调用

基础训练（leetcode）：

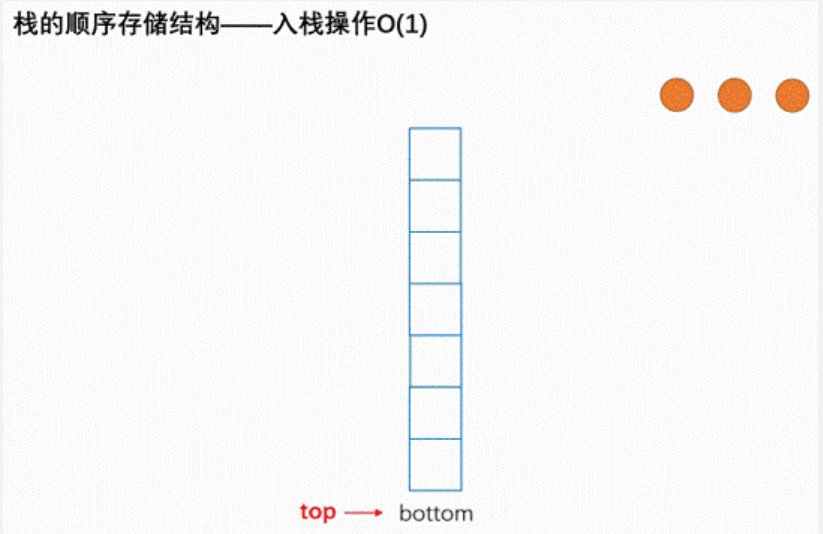
20. 有效的括号

155. 最小栈

234. 回文链表

682. 棒球比赛

注：matrix 考试不支持模板哦



案例研究：栈的链表实现

w6-link-stack-test.dev

- w6-linkstack.hpp

1. 简单，禁止了复制与赋值（=delete），元素是整数
2. 每个功能基本就一条语句
   - push 在表头加一个元素
   - pop 释放表头元素
   - top 取表头元素的值
   - …

```
class LinkedStack {
    public:
        LinkedStack(): head(nullptr) {};
        LinkedStack(const LinkedStack& s) = delete;
        LinkedStack& operator=(const LinkedStack& other) = delete
        ~ LinkedStack() { clear(); };
        void push(UserData item) {
            head = head->Push(head,item);
        };
        void pop() {
            if (!isEmpty()) head = head->Pop(head);
        };
        UserData top() {
            if (!isEmpty()) return head->getData();
            return 0;
        };
        bool isEmpty() { return !head; };
        void clear() { while (!isEmpty()) pop(); };
    private:
        _LinkNode *head;
};
```

案例研究：链表实现测试

w6-link-stack-test.dev

- w6-stack-test.cpp

1. 分析测试程序，如右

挑战自己：

- 把 typedef int UserData; 改为 typedef string UserData; 思考如何修改
- 把 typedef int UserData; 改为 typedef char* UserData; 思考如何修改

link_stack_test 函数分析：

10-12 s 空栈检测

14-19 1对 push-pop 检测 s=[]

21-23 多个push和pop序列检测，s=[3,7]

32-34 clear 测试 s=[]

36-37 destructor 测试 s=[3,7]

main 运行后显示，尽管链表中有元素，但stack析构函数正确的释放了它们。

案例研究：栈的数组实现

w6-link-stack-test.dev

- w6-arrystack.hpp

1. 先不考虑构造函数，元素是整数
2. 每个功能基本就一条语句
   - push 加一个元素
   - pop head--
   - top 取头元素的值
   - …

```
class ArrayStack {
    public:
        void push(UserData item) {
            if (head < MAX_SIZE)
                stack[++head] = item;
        };
        void pop() {
            if (!isEmpty()) head--;
        };
        UserData top() {
            if (!isEmpty()) return stack[head];
            return 0;
        };
        bool isEmpty() { return head == -1; };
        void clear() { while (!isEmpty()) pop();};
    private:
        UserData *stack;
        int head = -1 ; // just as initial list in constructors
        const static int MAX_SIZE = 100;
};
```

据说，只要C++上机考试，先打一段栈的基本实现具有凝气安神之功效

案例研究：栈的数组实现

w6-link-stack-test.dev

- w6-arrystack.hpp

1. 考虑构造函数
2. 构造与析构必须配对
3. 要注意成员的初始化。改成 const int MAX_SIZE; 好吗?
4. 不要企图在拷贝构造中调用默认构造
5. 如果元素是 string，请问 new UserData[MAX_SIZE] 对吗?
6. 简述 = 操作的实现步骤

```
class ArrayStack {
    public:
        ArrayStack() {
            stack = new UserData[MAX_SIZE]{0};
        };
        ArrayStack(const ArrayStack& other) {
            //ArrayStack(); //correct?
            stack = new UserData[MAX_SIZE];
            memcpy(stack, other.stack, sizeof(UserData) * MAX_SIZ
            head = other.head;
        } ;//= delete;
        ArrayStack& operator=(const ArrayStack& other) = delete;
        ~ ArrayStack() {
            clear();
            delete []stack;
        };
    private:
        UserData *stack;
        int head = -1 ; // just as initial list in constructors
        const static int MAX_SIZE = 100;
};
```

案例研究：数组实现测试

w6-link-stack-test.dev

- w6-stack-test.cpp

1. 自己分析测试程序，

挑战自己：

- 完成赋值运算重载
- 把 typedef int UserData; 改为 typedef string UserData; 思考如何修改
- 把 typedef int UserData; 改为 typedef char* UserData; 思考如何修改

array_stack_test 函数分析：

？

中山大學
SUN YAT-SEN UNIVERSITY

谢谢

中山大学计算机学院

主讲人：