
第三课 类和对象(封装)

内容概述

1. 封装的概念
2. 访问控制
3. 栈类的封装
4. 构造与析构
5. myString构造函数
6. 构造与析构的次序
7. 类文件写法
8. 对象的内存
9. this指针初探
10. 构造函数初始值列表
11. 拷贝构造和赋值运算符重载
12. 浅拷贝
13. 深拷贝
14. 成员函数内联
15. 友元
16. const和static成员
17. 单例模式
18. 类类型隐式转换
19. 类类型传参和返回值
20. 练习:myList链表

封装

类的基本思想：**数据抽象**和**封装**。

数据抽象是一种依赖于**接口**和**实现**分离的编程技术。

接口：类的用户所能执行的操作

实现：类的数据成员、接口函数的实现及其他私有函数的实现

封装：实现了类的接口和实现的分离。

封装后的类隐藏了实现细节；

类的用户只能使用接口而无法访问实现部分。

面向对象三大特性：**封装**、**继承**、**多态**。

访问控制

struct结构体 → 结构体变量
class类 → 对象

- 1.数据与行为不分离
(成员变量, 成员函数)
- 2.权限控制:
类内开放, 类外控制
保证数据完整性、正确性
- 3.成员函数,调用内部变量不需要传参

访问说明符:

public: 公共的

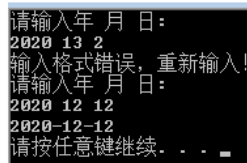
protected: 保护的

private: 私有的

C++中, class 和 struct没有本质区别, 只是默认权限不同

```
#include <stdio.h>
typedef struct _Date {
    int y, m, d;
}Date;
int isValid(Date *pd) {
    if (pd->y > 0 && pd->y < 9999 &&
        pd->m >0 && pd->m < 13 &&
        pd->d>0 && pd->d < 32)
        return 1;
    return 0;
}
void init(Date *pd) {
    while (1) {
        printf("请输入年 月 日:\n");
        scanf("%d%d%d", &pd->y, &pd->m, &pd->d);
        if (isValid(pd)) break;
        printf("输入格式错误, 重新输入!\n");
    }
}
void print(Date *pd) {
    printf("%d-%d-%d\n", pd->y, pd->m, pd->d);
}
int main()
{
    Date d1;
    init(&d1);
    print(&d1);
    return 0;
}
```

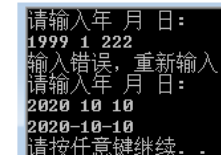
C语言



```
请输入年 月 日:
2020 13 2
输入格式错误, 重新输入!
请输入年 月 日:
2020 12 12
2020-12-12
请按任意键继续. . .
```

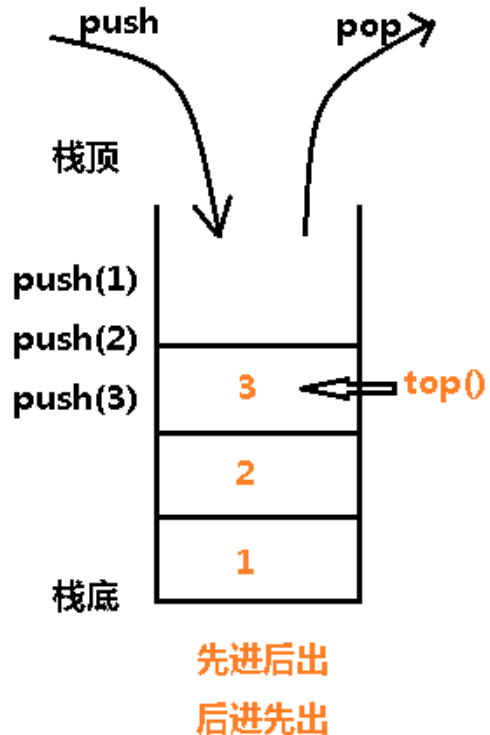
```
#include <iostream>
using namespace std;
class Date {
public:
    void init() {
        while (1) {
            cout << "请输入年 月 日:" << endl;
            cin >> y >> m >> d;
            if (isValid()) break;
            cout << "输入错误, 重新输入!" << endl;
        }
    }
    void print() {
        cout << y << "-" << m << "-" << d << endl;
    }
private:
    bool isValid() {
        if (y > 0 && y < 9999 && m>0 &&
            m < 13 && d>0 && d < 32)
            return true;
        return false;
    }
    int y, m, d;
};
int main() {
    Date d1;
    d1.init();
    d1.print();
    return 0;
}
```

C++语言



```
请输入年 月 日:
1999 1 222
输入错误, 重新输入!
请输入年 月 日:
2020 10 10
2020-10-10
请按任意键继续. . .
```

栈类



```
int main() {  
    Stack S;  
    S.init();  
    for (int i = 0; i < 5; i++) {  
        if (!S.isFull())  
            S.push(i);  
    }  
    while (!S.isEmpty()) {  
        cout << S.top() << " ";  
        S.pop();  
    }  
    //S.topidx = 10; //错误, 无法修改  
    S.destroy();  
    return 0;  
}
```

权限控制的重要性
提供的接口:
isEmpty(), isFull(), init(), destroy(),
top(), push(), pop()

成员函数const

```
class Stack {  
public:  
    bool isEmpty()const { return topidx == 0; }  
    bool isFull()const { return topidx == size; }  
    void init(int len = 1024) {  
        ps = new int[len];  
        size = len;  
        topidx = 0;  
    }  
    void destroy() {  
        if (ps) delete[] ps;  
        ps = NULL;  
    }  
    int top()const {  
        return ps[topidx - 1];  
    }  
    void push(int data) {  
        ps[topidx++] = data;  
    }  
    void pop() {  
        topidx--;  
    }  
private:  
    int *ps;  
    int topidx;  
    int size;  
};
```

构造函数、析构函数

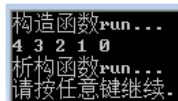
构造函数：每个类都定义了它的对象被初始化的方式，类通过一个或多个特殊的成员函数来控制其对象的初始化。（生成对象时自动调用）

析构函数：释放对象使用的资源。（对象销毁时自动调用）

构造函数：与类名相同，无返回，可以有参数

析构函数：~与类名相同，无参数，无返回

```
int main() {
    Stack S;
    //S.init(); //不再需要，被构造函数替代了
    for (int i = 0; i < 5; i++) {
        if (!S.isFull())
            S.push(i);
    }
    while (!S.isEmpty()) {
        cout << S.top() << " ";
        S.pop();
    }
    cout << endl;
    //S.destroy(); //不再需要，被析构函数替代了
    return 0;
}
```



```
class Stack {
public:
    Stack(int len = 1024) {
        ps = new int[len];
        size = len;
        topidx = 0;
        cout << "构造函数run..." << endl;
    }
    ~Stack() {
        if (ps) delete[] ps;
        ps = NULL;
        cout << "析构函数run..." << endl;
    }
    .....其他代码
private:
    int *ps;
    int topidx;
    int size;
};
```

构造函数初步

- 1.可以有参数，有默认参数，可以重载
- 2.若未提供构造函数，系统默认生成一个无参空构造
若提供，则不再生成默认无参空构造函数

类名 a; //调用无参构造[不能写成类名 a()],编译器会认为是函数声明]

类名 a(xx); //调用有参构造 a{xx}也可以。

通过 new 在堆空间创建对象，同样会自动调用构造函数

```
int main() {
    Stack S1;          //调用无参构造 不能写 Stack S1();
    Stack S2(100);    //调用带参构造
    Stack S3{ 10 };  //调用带参构造
    Stack *p1 = new Stack; //无参构造
    Stack *p2 = new Stack(10); //带参构造
    Stack *p3 = new Stack{ 10 }; //带参构造
    //对照：
    int a1; //不能写 int a1(); 这是函数声明
    int a2(10);
    int a3{ 10 };
    int *pa1 = new int;
    int *pa2 = new int(10);
    int *pa3 = new int{ 20 };
    return 0;
}
```

```
class Stack {
public:
    Stack() { //无参构造函数
        ps = new int[1024];
        size = 1024;
        topidx = 0;
        cout << "Stack() run" << endl;
    }
    Stack(int len) { //带参构造函数
        ps = new int[len];
        size = len;
        topidx = 0;
        cout << "Stack(int len) run" << endl;
    }
    /*Stack(int len = 1024) {
        ps = new int[len];
        size = len;
        topidx = 0;
    }*/
    .....//其他代码
};
```

默认参数：
将上面两个构造函数统一

构造函数初步: myString类的构造函数

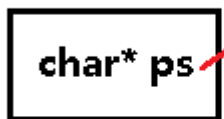
思考: string类是怎么实现的?

模仿标准库的string类:

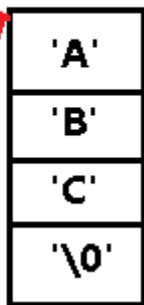
string s1; //无参构造

string s2("abc") //有参构造

string对象: s1



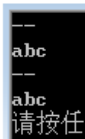
堆



```
#include <iostream>
#include <string>
#include <string.h>
using namespace std;
class myString {
public:
    myString() {
        ps = new char[1];
        ps[0] = '\0';
    }
    myString(const char *str) {
        int len = strlen(str) + 1;
        ps = new char[len];
        strcpy(ps, str);
    }
    ~myString() {
        delete[] ps;
    }
    const char* c_str()const {
        return ps;
    }
private:
    char *ps;
};
```

```
myString(const char *str = NULL) {
    if (str == NULL) { 默认参数
        ps = new char[1];
        ps[0] = '\0';
    }
    else {
        int len = strlen(str) + 1;
        ps = new char[len];
        strcpy(ps, str);
    }
}
```

```
int main() {
    string s1;
    string s2("abc");
    cout << "-" << s1.c_str() << "-" << endl;
    cout << s2.c_str() << endl;
    myString ms1;
    myString ms2("abc");
    cout << "-" << ms1.c_str() << "-" << endl;
    cout << ms2.c_str() << endl;
    return 0;
}
```



析构函数初步

析构函数：释放对象使用的资源。（对象销毁时自动调用）

1. 无参，无返回(不可重载)
2. 若未提供，系统默认生成一个空析构函数

通过 **delete** 销毁堆空间上的对象，同样会自动调用析构。

设计原则：
自己申请的资源，自己负责释放。

```
struct Stu {
    char *name;
    int age;
};
int main() {
    //申请st1的内存
    struct Stu *st1 =
        (struct Stu*)malloc(sizeof(struct Stu));
    //申请st1中name的内存
    st1->name = (char*)malloc(sizeof(char)*20);
    //释放st1中name的内存
    free(st1->name);
    //释放st1本身的内存
    free(st1);
    return 0;
}
```

C代码:

要逐级分别释放各自申请的资源

```
class Stu {
public:
    Stu() {
        name = new char[20];
        age = 10;
    }
    ~Stu() {
        delete[] name;
    }
private:
    char *name;
    int age;
};
int main() {
    //申请st1的内存
    Stu *st1 = new Stu;
    delete st1;
    return 0;
}
```

C++代码:

只负责释放自己申请的资源

构造与析构：次序

1. 多个对象，按次序构造，析构次序相反。

2. 类中有成员变量也是类对象的时候，先运行成员类的构造函数，再运行本类的构造函数。析构次序与构造次序相反。

3. 注意类中成员变量是类的指针类型的话，不会调用构造函数

```
class A {
public:
    A(int i = 0) {
        num = i;
        cout << "A()" << num << endl;
    }
    ~A() { cout << "~A()" << num << endl; }
private:
    int num;
};
class B {
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
};
int main() {
    A a1;
    A a2(1);
    B b;
    A *pa = new A[2]{ 2,3 }; //连续构造2次
    delete[] pa;           //连续析构2次
    return 0;
}
```

```
A<>0
A<>1
B<
A<>2
A<>3
~A<>3
~A<>2
~B<
~A<>1
~A<>0
请按任
```

```
class myString {
public:
    myString(const char *str = NULL) {
        if (str == NULL) {
            ps = new char[1];
            ps[0] = '\0';
        }
        else {
            int len = strlen(str) + 1;
            ps = new char[len];
            strcpy(ps, str);
        }
        cout << "myString构造" << endl;
    }
    ~myString() {
        if (ps) delete[] ps;
        cout << "myString析构" << endl;
    }
private:
    char *ps;
};
class Stu {
public:
    Stu() { cout << "Stu构造" << endl; }
    ~Stu() { cout << "Stu析构" << endl; }
private:
    myString name;
    int age;
};
int main() {
    Stu st1;
    return 0;
}
```

```
myString构造
Stu构造
Stu析构
myString析构
请按任意键继续
```

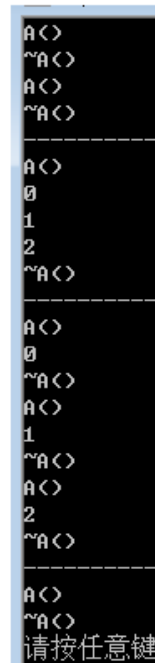
构造与析构：次序

1. 栈空间中的对象脱离作用域时，析构。

2. 堆空间中的对象delete时，析构。

```
int main() {
    {
        A a1;
    }
    //到这里，a1已经析构了。
    fun();
    //到这里，fun中的a2已经析构了。
    cout << "-----" << endl;
    int i = 0;
    //注意a3的作用域
    for (A a3; i < 3; i++) {
        cout << i << endl;
    }
    cout << "-----" << endl;
    //注意a4的作用域
    for (int j = 0; j < 3; j++) {
        A a4; //每次循环都会构造
        cout << j << endl;
    } //while循环也是如此
    cout << "-----" << endl;
    A *pa = new A; //构造
    delete pa; //调用delete析构
    return 0;
}
```

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
};
void fun() {
    A a2;
}
```



```
A()
~A()
A()
~A()
-----
A()
0
1
2
~A()
-----
A()
0
~A()
A()
1
~A()
A()
2
~A()
-----
A()
~A()
请按任意键
```

类文件写法

通常将一个类分为2个文件:

类的声明写在 **类名.h**

类的实现写在 **类名.cpp**

一个类就是一个作用域

在类的外部定义成员函数时:

返回值类型 **类名::函数名(参数)**

myString.h

```
#include <iostream>
class myString {
public:
    myString(const char *str = NULL);
    ~myString();
    const char* c_str()const;
private:
    char *ps;
};
```

调用:

```
#include <iostream>
#include "myString.h"
using namespace std;

int main() {
    myString s1;
    return 0;
}
```

myString.cpp

```
#include <string.h>
#include "myString.h"
//注意, 默认参数要写在函数声明中
myString::myString(const char *str) {
    if (str == NULL) {
        ps = new char[1];
        ps[0] = '\0';
    }
    else {
        int len = strlen(str) + 1;
        ps = new char[len];
        strcpy(ps, str);
    }
}
myString::~myString() {
    delete[] ps;
}
const char* myString::c_str()const {
    return ps;
}
```

对象的内存

类 → 对象, 模具 → 产品

创建对象时:

只有成员变量开辟内存。

没有成员变量时,占1个字节。

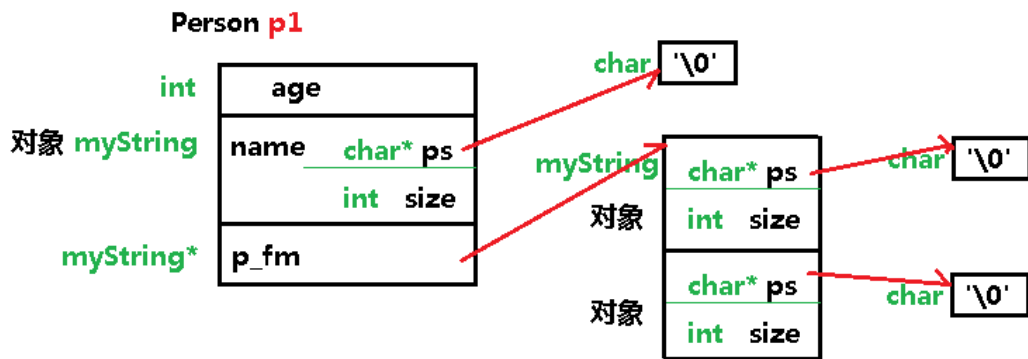
成员函数并不占用对象的空间。

class 中的成员变量和 C 中的 struct 一样, 要对齐、补齐。

```
#include <iostream>
using namespace std;
class A1 {};
class A2 { void fun() {} };
class A3 { int num; void fun() {} };
class A4 { int num; char name[6]; };
int main() {
    cout << sizeof(A1) << endl; //1
    cout << sizeof(A2) << endl; //1
    cout << sizeof(A3) << endl; //4
    cout << sizeof(A4) << endl; //12
    return 0;
}
```

```
class Person {
public:
    Person() { age = 20; p_fm = NULL; }
    void set_fm(){
        p_fm = new myString[2];
    }
    ~Person() {
        cout << "~Person()" << endl;
        delete[] p_fm;
        cout << "======" << endl;
    }
private:
    int age; //年龄
    myString name; //自己的名字
    myString *p_fm; //父母的名字
};
```

```
~Person()
~myString()
~myString()
====
~myString()
请按任意键继续
```



```
class myString {
public:
    myString(const char *str = NULL);
    ~myString();
private:
    char *ps; int size;
};
```

```
myString::myString(const char *str) {
    if (str == NULL) {
        ps = new char[1];
        ps[0] = '\0';
    }
    else {
        int len = strlen(str) + 1;
        ps = new char[len];
        strcpy(ps, str);
    }
}

myString::~myString() {
    delete[] ps;
    cout << "~myString()" << endl;
}
```

```
int main() {
    Person p1;
    p1.set_fm();
    return 0;
}
```

this指针初探

This指针可认为是顶层const,不能修改。

this = xxx; 错误

```
class A {
public:
    A(int num) { A::num = num; }
    A(double num) { this->num = num; }
    int num;
};
int main() {
    A a1(20);
    cout << a1.num << endl; //20
    A a2(1.2);
    cout << a2.num << endl; //1
    return 0;
}
```

```
class A {
public:
    A(int num) { this->num = num; }
    A& add(int n) { num += n; return *this; }
    int get_num()const { return num; }
private:
    int num;
};
int main() {
    A a1(1);
    a1.add(2).add(3).add(4).add(5);
    cout << a1.get_num() << endl; //15
    return 0;
}
```

通过返回*this的引用,实现连加。

```
class A {
public:
    A(int num) { this->num = num; }
    int show()const { return num; }
private:
    int num;
};
int main() {
    A a1(20);
    cout << a1.show() << endl; //20
    A a2(1);
    cout << a2.show() << endl; //1
    return 0;
}
```

show函数如何知道返回的是 a1还是a2的 num?

可想象为:

```
int A::show( A* this );
```

调用 a1.show(&a1); a2.show(&a2);

成员函数隐式地传递了一个当前对象的指针参数。

this指针: 指向的是本对象的地址。

构造函数初始值列表

有些类中，初始化和赋值的区别事关底层效率的问题。

初始化的先后次序，与成员变量出现的先后次序一致。

```
class A {
public:
    A(int n, double f) { num = n; fd = f; }
private:
    int num;
    double fd;
};
class B {
public:
    B(int n, double f) :num(n), fd(f) {}
private:
    int num;
    double fd;
};
int main() {
    A a(10, 1.2);
    B b(10, 1.2);
    return 0;
}
```

//构造函数的初始值列表

A类在创建对象时，先分别对 num 和 fd 调用默认初始化；

然后再通过赋值语句给 num 和 fd 赋值。

B类在创建对象时，直接初始化 num 和 fd。

A的流程：int num; num=10;

B的流程：int num(10);

假如是类成员变量：A--> myString ms; ms = 'abc';

B--> mySreing ms('abc');

初始值列表对成员变量的初始化先后次序，与成员变量在类中出现的先后次序一致。

尝试用排序靠后的成员变量来初始化前面的成员变量，会得到未知的结果。

```
class B {
public:
    B(double f) :fd(f), num(fd) {}
    int get_num()const { return num; }
private:
    int num;
    double fd;
};
int main() {
    B b(1.2);
    cout << b.get_num() << endl;
    //没有得到预期中的 1
    return 0;
}
```



还有一种写法，C++11支持

```
class B {
public:
    B(){}
private:
    int num = 10;
    //可理解为默认值,不可 int num(10);
    double fd = 1.2;
};
```


拷贝构造和赋值运算符重载

拷贝构造函数:

```
class 类名{  
    类名(const 类名 & another);  
}
```

1. 系统提供默认的拷贝构造, 若自己提供, 则不复存在。
2. 默认拷贝构造是等位拷贝, 也就是所谓的浅拷贝。
- 3 要实现深拷贝, 必须要自己实现。

赋值运算符重载:

```
类名{  
    类名& operator=(const 类名&  
    源对象)  
    ...; return *this;  
} 与拷贝构造类似。
```

```
string s1("abc"); //直接初始化  
string s2("1234");  
string s3(s2); //直接初始化  
string s4 = s2; //拷贝初始化  
string s5;  
s5 = s2; //赋值操作
```

思考: 上面是string类的一些操作, 如何实现?

```
class A {  
public:  
    A(int n = 0) : num(n) {}  
    A(const A &other) : num(other.num) {}  
private:  
    int num;  
};  
int main() {  
    A a1;  
    A a2(a1); //直接初始化  
    A a3 = a1; //拷贝初始化  
    const A a4;  
    A a5 = a4; //参数没有const, 就错误  
    return 0;  
}
```

a对象还没有创建 ; b对象已经存在。

拷贝构造函数: A a(b); //直接初始化 (调用拷贝构造函数)

A a = b; //拷贝初始化

以对象作为参数和返回值时, 调用拷贝构造

```
class A {  
public:  
    A(int n = 0) : num(n) {}  
    A(const A &other) : num(other.num) { }  
    A& operator=(const A &other) {  
        num = other.num;  
        return *this;  
    }  
private:  
    int num;  
};  
int main() {  
    A a1;  
    A a2;  
    a2 = a1; //赋值  
    return 0;  
}
```

赋值运算符重载:

用一个已有对象, 给另外一个已有对象赋值。两个对象均已创建结束后, 发生的赋值行为。

赋值运算符, a = b 可以解释为:

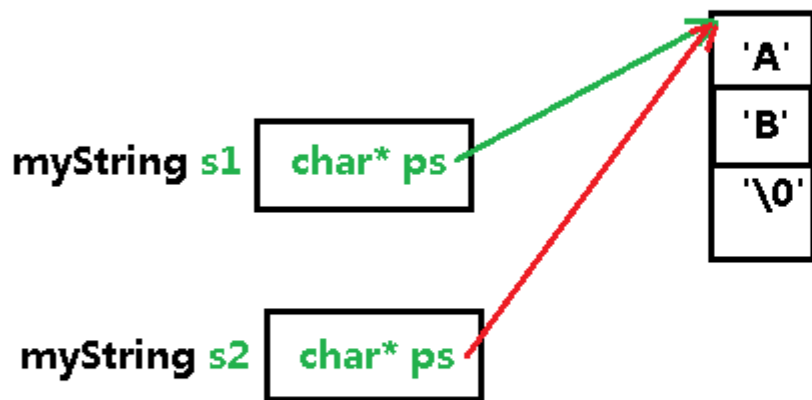
operator=(a, b); //普通函数

a.operator=(b); //成员函数

浅拷贝

浅拷贝可能会造成内存泄漏、重析构。

```
myString(const char *str = NULL) {  
    ...  
    ps = new char[len];  
}
```



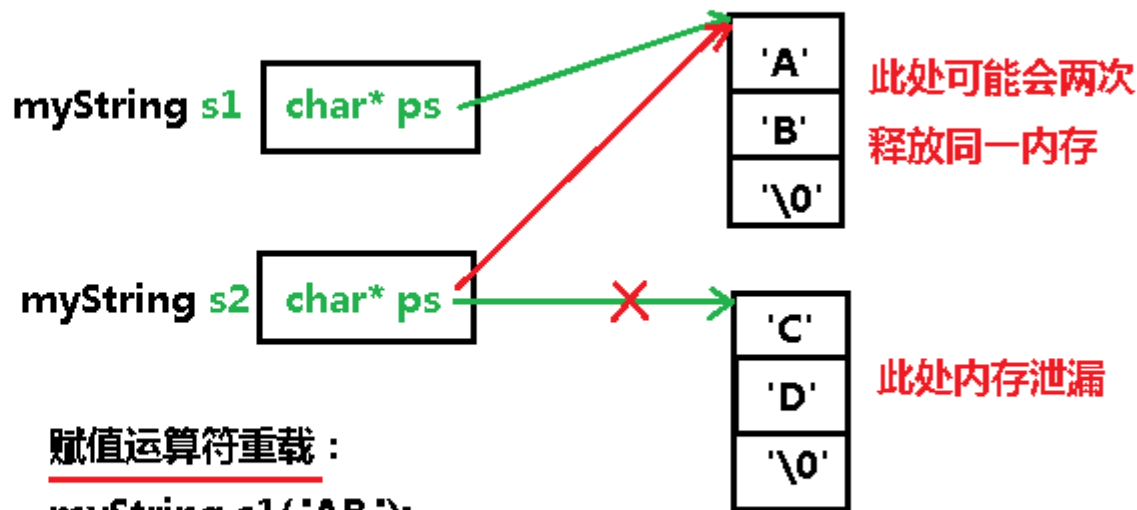
拷贝构造 : myString s2=s1;

等位拷贝 : s1的ps指针值赋给s2的ps

s1.ps和s2.ps同时指向同一个内存

s1,s2分别析构时, 会两次释放同一内存

```
~myString() {  
    if (ps) delete[] ps;  
}
```



赋值运算符重载 :

```
myString s1('AB');
```

```
myString s2('CD');
```

```
s2 = s1;
```

内存泄漏, 重析构

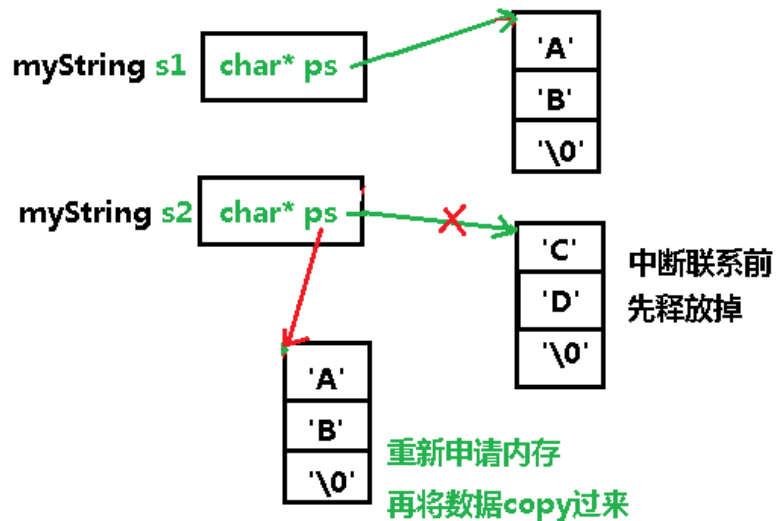
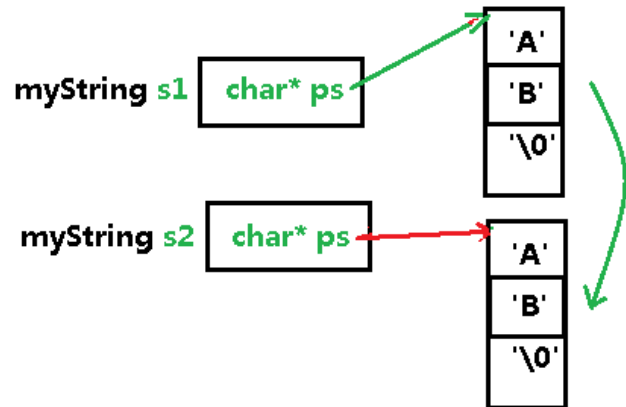
深拷贝实现

```
class myString {
public:
    myString(const char *str = NULL) {
        if (str == NULL) {
            ps = new char[1];
            ps[0] = '\0';
        }
        else {
            int len = strlen(str) + 1;
            ps = new char[len];
            strcpy(ps, str);
        }
    }
    myString(const myString &other);
    myString& operator=(const myString &other);
    ~myString() {
        delete[] ps;
    }
private:
    char *ps;
};
```

```
myString::myString(const myString &other) {
    ps = new char[strlen(other.ps) + 1];
    strcpy(ps, other.ps);
} //拷贝构造
```

赋值运算符重载：要注意自己给自己赋值的情况。

```
myString& myString::operator=(const myString &other) {
    if (this == &other) return *this;
    delete[] ps;
    ps = new char[strlen(other.ps) + 1];
    strcpy(ps, other.ps);
    return *this;
} //赋值运算符重载
```



成员函数内联

内联函数: `inline`修饰, 编译时展开。

Stack.h

```
class Stack {
public:
    Stack(int len = 1024);
    ~Stack();
    bool isEmpty()const {
        return topidx == 0;
    } //隐式内联
    inline bool isFull()const; //内联
    int get_size()const; //内联
    int top()const;
    void push(int data);
    void pop();
private:
    int *ps;
    int topidx;
    int size;
};
inline bool Stack::isFull()const{
    return topidx == size;
}
inline int Stack::get_size()const{
    return size;
}
```

Stack.cpp

```
#include <iostream>
#include "Stack.h"
Stack::Stack(int len): topidx(0),
                    size(len) {
    ps = new int[len];
}
Stack::~Stack() {
    if (ps) delete[] ps;
    ps = NULL;
}

int Stack::top()const {
    return ps[topidx - 1];
}

void Stack::push(int data) {
    ps[topidx++] = data;
}

void Stack::pop() {
    topidx--;
}
```

main.cpp

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
    Stack S;
    for (int i = 0; i < 5; i++) {
        if (!S.isFull())
            S.push(i);
    }
    cout << S.get_size() << endl;
    while (!S.isEmpty()) {
        cout << S.top() << " ";
        S.pop();
    }
    cout << endl;
    return 0;
}
```

友元

封装: 类的数据成员一般定义为私有成员, 外部要访问类的私有成员只能通过**接口** (如get,set方法)
特殊情况: 类外有些函数需要**频繁地访问类的数据成员**, 此时可以将这些函数定义为该类的友元函数。
友元的好处: 提高了程序的运行效率 (减少类型和安全性检查及调用的时间开销)
友元的副作用: 破坏了类的封装性和隐藏性, 使得非成员函数可以直接访问类的私有成员。
友元可以是一个函数, 称为**友元函数**; 友元也可以是一个类, 称为**友元类**。

```
friend void set_Person(Person& person, double m, int t);  
friend double calc_Person(const Person& person);
```

友元的声明仅仅指定了访问的权限。
通常把友元函数本身的声明和类的头文件放在一起。

```
class Person {  
public:  
    void set_m_p_piece(int m) {  
        if (m >= 10.0 && m <= 100.0)  
            m_p_piece = m;  
        else m_p_piece = 10.0;  
    }  
    double get_m_p_piece()const { return m_p_piece; }  
    void set_t_piece(int t) {  
        if (t <= 0) t_piece = 0;  
        else t_piece = t;  
    }  
    int get_t_piece()const { return t_piece; }  
private:  
    double m_p_piece; //单件工资  
    int t_piece;     //总件数  
};
```

```
void set_Person(Person& person, double m, int t) {  
    person.set_m_p_piece(m);  
    person.set_t_piece(t);  
}  
double calc_Person(const Person& person) {  
    return  
        person.get_m_p_piece()*person.get_t_piece();  
}  
int main() {  
    Person person1;  
    for (int i = 0; i < 100000; i++) {  
        set_Person(person1, i, i);  
        cout << calc_Person(person1) << endl;  
    }  
    return 0;  
}
```

person.m_p_piece = m;
person.t_piece = t;

person.m_p_piece
* person.t_piece;

友元

同类之间可直接访问。例：`A(const A &other) { num = other.num; }`

全局函数、类成员函数可作为友元函数。类的所有成员函数都可直接访问：友元类。

```
#include <iostream>
#include <cmath>
using namespace std;
class Point;          //前向声明
class ManagerPoint{ //管理Point的类
public:
    double distance(Point &a, Point &b);
};

class Point{         //Point类
public:
    friend double ManagerPoint::distance(Point &a, Point &b);
    Point(double x, double y):x(x),y(y) { }
    void print()const;
    const double &get_x()const { return x; }
    const double &get_y()const { return y; }
private:
    double x, y;
};

void Point::print()const {
    cout << "(" << x << ", " << y << ")" << endl;
}
```

```
double ManagerPoint::distance(Point &a, Point &b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx*dx + dy*dy);
}

int main(){
    Point p1(1.0, 1.0), p2(4.0, 5.0);
    p1.print();
    p2.print();
    ManagerPoint mp;
    double d = mp.distance(p1, p2);
    cout << "Distance = " << d << endl;
    return 0;
}
```

```
<1,1>
<4,5>
Distance = 5
请按任意键继续
```

前向声明：是一种不完全型声明

1. 不能定义类的对象
2. 可以用于定义指向这个类类型的指针或引用。
3. 用于声明，使用该类类型作为形参类型或者返回值类型。

友元

- (1) 友元关系不能被继承。
- (2) 友元关系是单向的，不具有交换性。若 B 是 A 的友元，A 不一定是 B 的友元
- (3) 友元关系不具有传递性。若 B 是 A 的友元，C 是 B 的友元，不能推出 C 是 A 的友元

B 是 A 的友元，不等于 A 是 B 的友元!

若 B 是 A 的友元 (B 可访问 A 的私有成员)

C 是 B 的友元 (C 可访问 B 的私有成员)

不等于 C 是 A 的友元 (C 可访问 A 的私有成员?不行)

```
class Point {
public:
    friend class Stack; //友元类
    Point(double x = 0.0, double y = 0.0)
        :x(x), y(y) {}
private:
    double x, y;
};
```

```
class Stack {
public:
    Stack(int len = 1024);
    ~Stack();
    bool isEmpty()const;
    bool isFull()const;
    const Point &top()const;
    void push(const Point& data) {
        ps[topidx++] = data;
        cout << "(" << data.x << ", "
            << data.y << ")" << endl;
    }
    void pop();
private:
    Point *ps;
    int topidx;
    int size;
};
```

类的const常量成员

const 成员变量，不能修改，只能在**初始值列表**中初始化。

```
class A {
public:
    A(int i = 0) :ci(i) { }
    //const成员变量,只能通过初始值列表初始化
    //A(int i = 0) { ci = i; } //错误!
    //void set_ci(int i) { ci = i; } //错
private:
    const int ci;
};
```

const 成员函数:

1. const放在函数声明之后，实现体之前 **void fun()const { }**
2. 承诺在本函数内部不会修改类内的数据成员，不会调用其它非 const 成员函数
3. const构成函数**重载**(why? **this指针底层const**)

const对象，只能调用 **const 成员函数**。可访问所有数据成员，不可修改。

```
#include <iostream>
using namespace std;
class A {
public:
    A(int i = 0) :ci(i),num(i) { }
    int get_ci()const { return ci; }
    void show()const {
        //set_num(0); //错误,调用非const函数
        //num = 0; //错误,修改数据成员
        cout << num*ci << endl; //正确,调用不修改
        get_ci(); //正确,调用const函数
        cout << "show() const" << endl;
    }
    void show() { cout << "show()" << endl; } //重载
    int set_num(int i) { num = i; }
private:
    const int ci;
    int num;
};
int main(){
    A a; //普通对象
    const A ca; //const对象
    //ca.set_num(1); //错误,const对象只能调用const函数
    ca.show(); // show() const
    a.show(); //show() 非const
    //[非const对象也可调用const函数]
    return 0;
}
```


类的static静态成员

static 成员变量：同类对象间信息共享，类外存储，必须类外初始化，可通过类名访问，也可通过对象访问。

静态成员函数：管理静态数据成员，对静态数据成员封装（对外提供接口）。

静态成员函数：只能访问静态数据成员。静态成员函数**属于类**，**没有 this 指针**。

```
#include <iostream>
using namespace std;
class Person {
public:
    Person(int _age) :age(_age) { sum += _age; count++; }
    static int get_count();
    static double get_avg() { return double(sum) / count; }
private:
    int age;           //年龄
    static int count; //总人数
public:
    //不能 static int sum=0;
    static int sum;   //年龄总和
};

//此处不能加 static
int Person::get_count() { return count; }
//此处不能加 static
int Person::sum = 0;    //必须初始化
int Person::count = 0; //数据区(bss,rw?)
```

```
int main(){
    //外部直接访问方式，类名::静态变量名
    cout << Person::sum << endl;
    //cout << Person::count; //错误,同样有权限控制
    //静态成员变量不占用对象的内存空间
    cout << sizeof(Person) << endl; //4

    Person p1(10);
    //静态变量，也可以通过 对象.变量名 来访问
    //类外需要权限，类内部都行
    cout << p1.sum << endl;
    Person p2(20);

    //静态成员函数，可以通过 类名::函数名 来访问
    cout << Person::get_avg() << endl;
    //也可以通过 对象.函数名 来访问
    cout << p1.get_count() << endl; //2
    cout << p2.get_count() << endl; //与上面输出一致
    return 0;
}
```


单例模式

1. 将默认构造函数和析构函数声明为私有，外部无法创建，无法销毁（只能自己销毁自己）
2. 使用一个私有的静态本类类型的指针变量，用来指向该类的唯一实例。
3. 用一个公有的静态方法来获取该实例，第一次调用该方法时，创建实例并返回（懒汉式），以后调用直接返回。
4. 用一个公有的静态方法来删除该实例，以保证该实例只会被删除一次

```
#include <iostream>
using namespace std;
class A {
private:
    A() {}; //构造和析构私有
    ~A() {};
public:
    static A * GetInstance() { //公有静态方法，可以获取该唯一实例
        if (NULL == m_pInstance) m_pInstance = new A; //(多线程需要加锁)
        return m_pInstance;
    }
    static void DeleteInstance() { //公有静态方法，可以删除该实例
        if (m_pInstance != NULL)
            delete m_pInstance;
        m_pInstance = NULL;
    }
private:
    static A *m_pInstance; //私有静态指针变量：指向类的唯一实例
    int count; //其他成员变量...
};
```

```
A *A::m_pInstance = NULL; //懒汉式
//A *A::m_pInstance = new A; //饿汉式

int main() {
    //A a; //错误，外部无法创建该类对象
    A *pa = A::GetInstance(); //通过调用 类静态成员函数 来获取类对象
    A *pb = A::GetInstance(); //可多次调用
    cout << pa << pb << endl; //地址相同，pa,pb指向的是同一个对象
    //delete pa; //错误，外部无法直接销毁该类对象
    A::DeleteInstance(); //通过调用 类静态成员函数 来析构类对象
    return 0;
}
```

类类型隐式转换

```
class A{};
class myString {
public:
    myString(const char* s = NULL) { cout << "myString(const char* s)" << endl; }
    myString(int size, char c = ' ') { cout << "构建size个c组成的字符串" << endl; }
    explicit myString(A a) { cout << "explicit myString(A a)" << endl; }
    myString(const myString& other) { cout << "copy初始化" << endl; }
    myString &operator=(const myString& other) { cout << "赋值函数" << endl; return *this; }
};
```

```
int main() {
    //自己定义的类如何实现这些?
    myString s1("abc"); //直接初始化
    cout << "-----" << endl;
    //拷贝初始化[拷贝前先隐式类型转换](编译器可能会优化)
    myString s2 = "abc";
    cout << "======" << endl;
    s2 = "abc"; //先隐式转换为 myString,再 赋值
    //调用 myString(const char*) 函数进行 隐式转换
    cout << "-----" << endl;
    myString s3 = 20; //这样,也行!
    //调用了 myString(int size,char c=' ')
    //实际上,是创建一个 20个空格组成的字符串
    //但是,我们看起来:是 数字20 赋值 给了 myString
    //容易引起混乱,所以这样的构造函数,前面要加 explicit
    cout << "======" << endl;
}
```

观察:

```
//int a{ 1.2 }; //错误,列表初始化
//观察下面的隐式类型转换
int a1(1.2); //直接初始化
int a2 = 1.2; //拷贝初始化
int a3;
a3 = 1.2; //赋值
```

```
A a;
//s3 = a; //错误,由于explicit修饰,A类型无法隐式转换为myString
s3 = myString(a);
s3 = (myString)a; //只能显式地转换
s3 = static_cast<myString>(a); //只能显式地转换
cout << "-----" << endl;
return 0;
```



```
myString(const char* s)
-----
myString(const char* s)
=====
myString(const char* s)
赋值函数
构建size个c组成的字符串
=====
explicit myString(A a)
赋值函数
explicit myString(A a)
赋值函数
explicit myString(A a)
赋值函数
请按任意键继续. . .
```

- 要实现 A -> B 的隐式转换, B中必须有非Explicit构造函数,参数是A
1. explicit: 抑制构造函数定义的隐式转换
 2. explicit构造函数,只能用于直接初始化
 3. explicit关键字:只能在类内声明使用,类外不能加

不能用explicit构造函数来实现 myString s = xxx;这样的拷贝构造
也就是说:假如 explicit myString(const char* s = NULL)
那么:myString S("abc"); //ok myString S = "abc"; //error

类类型临时量

临时量：内置类型是const的，类类型不一定

```
#include <iostream>
using namespace std;
class A {
public:
    A(const char* s = NULL){}
    A& operator+(const A& other) {
        cout << "+++" << endl;
        return *this;
    }
    const A& operator-(const A& other) {
        cout << "---" << endl;
        return *this;
    }
};
```

```
void fun(const int &i) { cout << "const int" << endl; }
void fun(int &i) { cout << "int" << endl; }
void fun(const A &a) { cout << "const A" << endl; }
void fun(A &a) { cout << "A" << endl; }
int main() {
    fun(1+1); //调用const版本
    cout << "-----" << endl;
    int i = 10;
    fun(i); //调用非const版本
    cout << "-----" << endl;
    A a;
    fun(a + "aa"); //调用非const
    fun(a - "aa"); //调用const
    return 0;
}
```

类类型临时量，在表达式结束后自动析构。

类类型传参

```
#include <iostream>
using namespace std; // 临时量：内置类型是const的，类类型不一定
class myString {
public:
    myString(const char* s = NULL) { cout << "const char* 构造" << endl; }
    myString(const myString &other) { cout << "copy 构造" << endl; }
    myString &operator=(const myString &other) { cout << "赋值操作" << endl; }
    ~myString() { cout << "析构" << endl; }
};
void fun(myString ss){}
//void fun(myString &ss) {} //引用参数，相当于 myString &ss = s1; 没有开辟内存
//void fun(myString *ss) {} //指针参数，只是复制了4字节的指针数据
int main() {
    myString s1 = "abc"; // 语义是：先隐式转换，然后copy构造。
                        // 编译器优化：直接 const char* 构造
    fun(s1); //copy构造：语义：myString ss = s1;
    fun("abc"); //语义：myString ss = "abc"; 与s1构造类似
    return 0;
}
```

```
const char* 构造
copy 构造
析构
const char* 构造
析构
析构
请按任意键继续...
```

类类型返回值

```
#include <iostream>
using namespace std; // 临时量：内置类型是const的，类类型不一定
class myString {
public:
    myString(const char* s = NULL) { cout << "const char* 构造" << endl; }
    myString(const myString &other) { cout << "copy 构造" << endl; }
    myString &operator=(const myString &other) {
        cout << "赋值操作" << endl; return *this; }
    ~myString() { cout << "析构" << endl; }
};

myString fun() { myString ss; return ss; }
//myString &fun() {} //返回的是引用
//myString *fun() {} //返回的是指针
int main() {
    myString s1; //const char* 构造
    s1 = fun(); //myString 临时量 = ss;(copy构造) --> s1 = 临时量;(赋值操作)
    //fun();
    return 0;
}
```

```
const char* 构造
const char* 构造
copy 构造
析构
赋值操作
析构
析构
请按任意键继续.
```

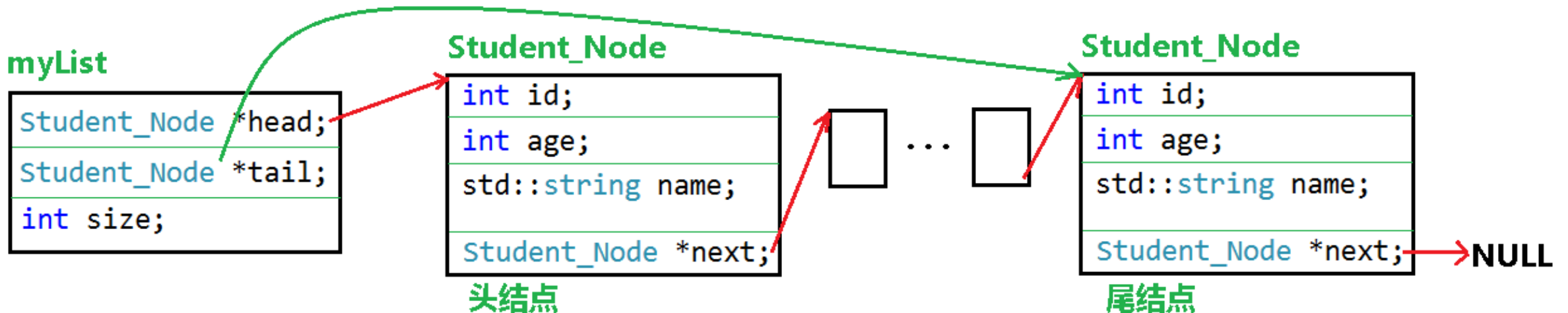

练习：myList链表1

实现一个**链表类**：

1. 带头结点
2. 数据类型是Student类，学号起始编号是100
3. 实现下列功能：
myList(const myList& other); //深拷贝构造
myList &operator=(const myList& other); //赋值重载
void insert_tail(int _age,const string& _name); //尾插
bool del_node(int _id); //按id删除节点
void print()const; //遍历打印
inline int get_size()const; //获取链表中元素个数

```
class myList {  
    .....  
private:  
    Student_Node *head; //头指针  
    Student_Node *tail; //尾指针  
    int size; //节点数量  
};
```

```
class Student_Node {  
    .....  
private:  
    int id;  
    int age;  
    std::string name;  
    Student_Node *next;  
};
```



练习：myList链表2

Student_Node类（结点）：

1. 学号自动增长。**static** 静态变量和静态函数
2. Student_Node类没有实现析构、拷贝构造、赋值运算符重载，可行吗？
3. 假如学员姓名用自己写的 myString类，那么要在哪里自己实现上面的那些函数？
4. 声明了 myList是 Student_Node的友元类，假如不允许使用友元，该类还需要提供哪些函数？

StudentNode.cpp

```
#include "StudentNode.h"
int Student_Node::now_id = 100;
```

```
#ifndef _STUDENTNODE_H
#define _STUDENTNODE_H
#include <string>
class Student_Node {
    friend class myList;
public:
    Student_Node(bool flag = false, int _age = 20,
        const std::string &_name = "")
        :id(1), age(_age),
        name(_name), next(NULL) {}
    2 int calc_id(bool flag) {
        if(flag) return now_id++;
        return 0;
    }
private:
    int id;
    int age;
    std::string name;
    Student_Node *next;
private:
    static int now_id;
};
#endif
```

StudentNode.h

| Student_Node |
|---------------------|
| int id; |
| int age; |
| std::string name; |
| Student_Node *next; |

答案: 1: calc_id(flag)
2: static

练习：myList链表3

myList类:

1. 内联函数为啥写在 myList.h 头文件中?
2. 链表要求带头结点，构造函数怎么写?
3. 拷贝构造函数和赋值运算符重载实现上有什么区别?
4. 要如何析构myList?

```
#include <string>
#include "StudentNode.h"           myList.h
using namespace std;
class myList {
public:
    myList();
    ~myList();
    myList(const myList& other);
    myList &operator=(const myList& other);
    void insert_tail(int _age,
                    const string& _name); //尾插
    bool del_node(int _id); //按id删除节点
    void print()const; //遍历打印
    inline int get_size()const;
private:
    void copy_from(const myList& other);
private:
    Student_Node *head; //头指针
    Student_Node *tail; //尾指针
    int size; //节点数量
};
inline int myList::get_size()const {
    return size;
}
```


练习：myList链表4

```
#include <iostream>
#include "myList.h"
using namespace std;
myList::myList() :size(0) {
    head = new Student_Node;
    tail = head;
}
myList::~myList() {
    while (head) {
        Student_Node *tmp = head->next;
        delete head;
        head = tmp;
    }
    head = tail = NULL;
}
```

myList.cpp

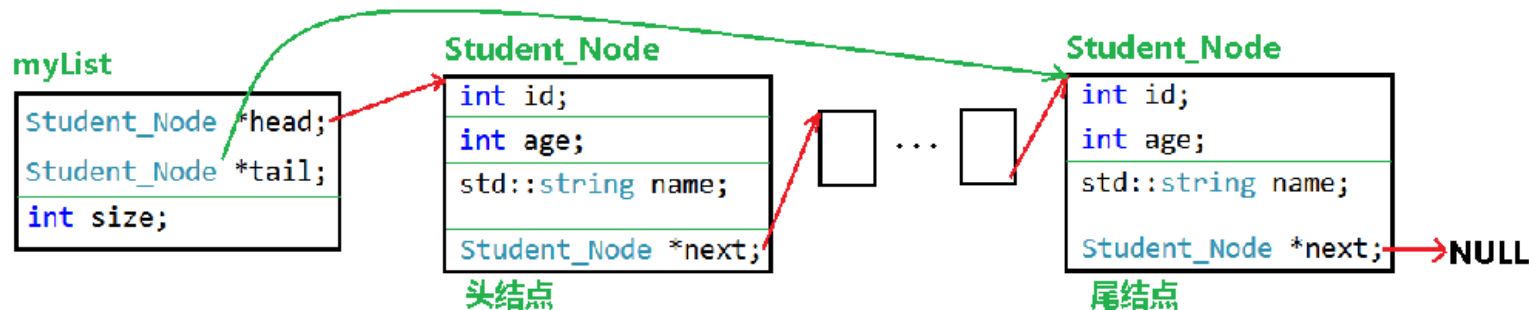
```
void myList::print()const { //遍历打印
    Student_Node* p = head->next;
    cout << "Head -> ";
    while (p) {
        cout << "(" << p->id << "," <<
            p->name << "," << p->age << ") -> ";
        p = p->next;
    }
    cout << "NULL" << endl;
}
```

myList

```
Student_Node *head;
Student_Node *tail;
int size;
```

Student_Node

```
int id;
int age;
std::string name;
Student_Node *next;
```



练习：myList链表5

```
void myList::copy_from(const myList& other) {  
    Student_Node *p = other.head->next;           myList.cpp  
    while (p) {  
        Student_Node *newNode = new Student_Node(*p);  
        tail->next = newNode;  
        tail = newNode;  
        size++;  
        p = p->next;  
    }  
}
```

```
myList::myList(1) :myList() {  
    //委托构造  
    copy_from(other);  
}
```

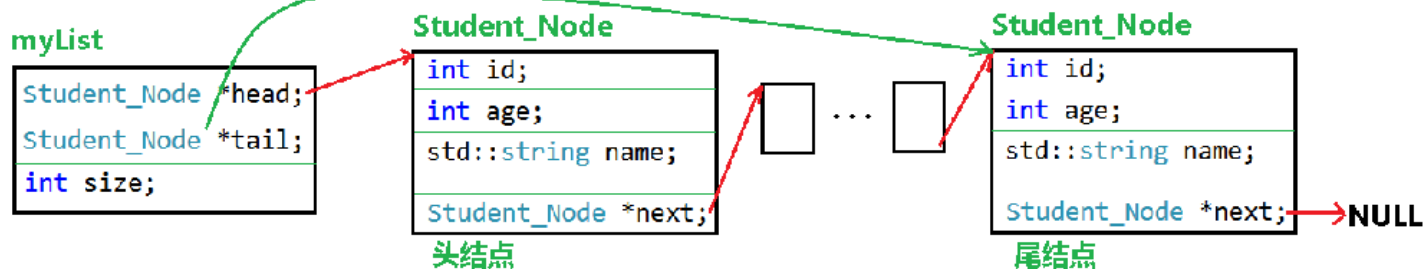
```
myList &myList::operator=(const myList& other) {  
    if (2) return 3;  
    Student_Node *p = head->next;  
    while (p) {  
        Student_Node *q = p->next;  
        delete p;  
        p = q;  
    }  
    head->next = NULL;  
    tail = head;  
    size = 0;  
    copy_from(other);  
    return 4;  
}
```

myList

| |
|---------------------|
| Student_Node *head; |
| Student_Node *tail; |
| int size; |

Student_Node

| |
|---------------------|
| int id; |
| int age; |
| std::string name; |
| Student_Node *next; |



答案：1 const myList& other
2 this == &other
3 *this
4 *this

练习：myList链表6

myList其他函数：

insert_tail：尾插

del_node：根据学号删除

```
void myList::insert_tail(int _age, const string& _name) {
    Student_Node *newNode = new Student_Node(true, _age, _name);
    tail->next = newNode;
    tail = newNode;
    size++;
}

bool myList::del_node(int _id) { //按id删除
    Student_Node *p = head, *q = head->next;
    while (q && q->id != _id) {
        p = q;
        q = q->next;
    }
    if (!q) return false; //没找到,删除失败
    if (q->next == NULL) tail = p;
    p->next = q->next;
    delete q;
    size--;
    return true;
}
```

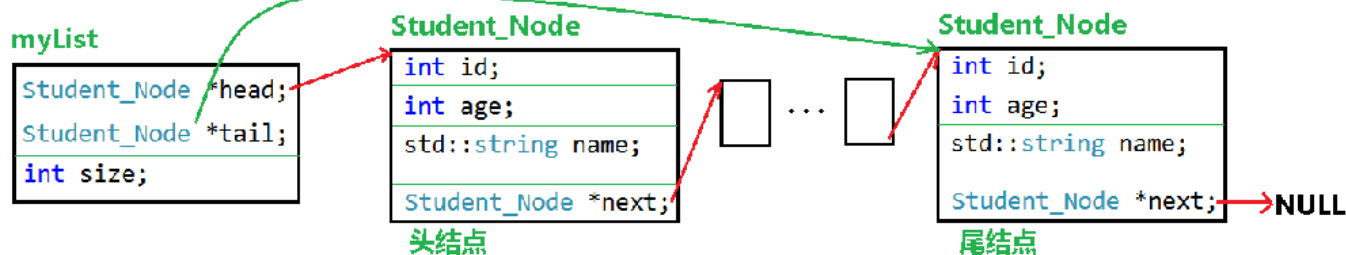
myList.cpp

myList

```
Student_Node *head;
Student_Node *tail;
int size;
```

Student_Node

```
int id;
int age;
std::string name;
Student_Node *next;
```



练习：myList链表7

```
#include <iostream>
#include "StudentNode.h"
#include "myList.h"
using namespace std;

int main() {
    cout << "p1-----" << endl;
    myList* p1 = new myList;
    p1->insert_tail(30, "赵一");
    p1->insert_tail(21, "张三");
    p1->print();
    p1->del_node(100);
    p1->print();
    p1->insert_tail(19, "钱五");
    p1->print();

    cout << "p2-----" << endl;
    myList* p2 = new myList(*p1);
    p2->print();
    p2->insert_tail(18, "李六");
    p2->print();
```

test.cpp
测试

```
cout << "p3-----" << endl;
myList* p3 = new myList;
*p3 = *p2;
p3->print();
p3->del_node(103);
p3->print();
p3->insert_tail(19, "费大");
p3->print();
delete p1;
delete p2;
delete p3;
return 0;
}
```

```
p1-----
Head -> <100, 赵一, 30> -> <101, 张三, 21> -> NULL
Head -> <101, 张三, 21> -> NULL
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> NULL
p2-----
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> NULL
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> <103, 李六, 18> -> NULL
p3-----
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> <103, 李六, 18> -> NULL
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> NULL
Head -> <101, 张三, 21> -> <102, 钱五, 19> -> <104, 费大, 19> -> NULL
请按任意键继续. . .
```