带参数的构造函数

范 懿

目录

1. 带参数的构造函数
2. 变长字符串
3. 迭代器
4. 队列

```cpp
#include <iostream>
using namespace std;
class MyClass {
        int a, b;
    public:
        MyClass(int i, int j) {a = i; b = j;}
        void show() {cout << a << " " << b;}
};
int main(){
    MyClass ob(3, 5);
    ob.show();
    return 0;
}
```

- 传入参数 i 和 j 用来给参数 a 和 b 赋值。
- MyClass ob(3, 5); 创建一个对象 ob 并把 3 和 5 赋给 MyClass() 的参数 i 和 j
- 也可以这样传参数：MyClass ob = MyClass(3, 5);
- 两种写法有细微区别：详见拷贝构造函数

```
#include <iostream>
#include <cstring>
using namespace std;
const int IN = 1;
const int CHECKED_OUT = 0;
class Book {
        char author[40];
        char title[40];
        int status; // 是否在馆
    public:
        Book(char *n, char *t, int s);
        int get_status() {return status;}
        void set_status(int s) {status = s;}
        void show();
};


Book::Book(char *n, char *t, int s)
{
    strcpy(author, n);
    strcpy(title, t);
```

```cpp
    status = s;
}
void Book::show()
{
    cout << title << " by " << author;
    cout << " is ";
    if(status==IN) cout << "in.\n";
    else cout << "out.\n";
}
int main()
{
    Book b1("Twain", "Tom Sawyer", IN);
    Book b2("Melville", "Moby Dick", CHECKED_OUT);
    b1.show();
    b2.show();
    return 0;
}
```

带参数的构造函数直接可以完成全部初始化

```
#ifndef MY_STRING_H
#define MY_STRING_H
class MyString{
    private:
        char *str; // 字符串首元素地址
        int len; // 字符串长度
        int capacity; // 字符串容量
        // 获取字符串首元素地址
        // 涉及内存，设为私有比较保险
        char* get_str()
        {
            return str;
        }
    public:
        MyString();
        MyString(const char* s);
        ~MyString();
        // 把MyString类型对象s赋值给
        // 当前MyString类型对象
        void assign(MyString &s);
        // 把对象s连接到当前的对象之后
        void append(MyString &s);
        void show();
};
#endif
```

```
#include <iostream>
#include <cstring>
#include "MyString.h"

using namespace std;

// 无参构造函数
MyString::MyString()
{
    str = NULL;
    len = 0;
    capacity = 0;
}

//带参构造函数
MyString::MyString(const char* s)
{
    len = strlen(s); // 确定长度
    capacity = len; // 确定容量
    str = new char[capacity + 1]; // 分配空间
    strcpy(str, s); // 复制
```

```
}

// 析构函数
MyString::~MyString()
{
    delete []str; // 释放内存
}

// 把MyString型对象s赋值给调用本成员函数的对象
void MyString::assign(MyString &s)
{
    len = strlen(s.get_str());
    if(capacity < len) // 容量不足
    {
        capacity = len;
        delete[] str; // 清理旧空间
        str = new char[capacity + 1]; // 空间扩充
        if(!str) // 注意检查是否分配成功
        {
            cout << "insufficient memory" << endl;
            exit(1);
        }
    }
    strcpy(str, s.get_str()); // 复制
}

// 把MyString型对象添加到调用本成员函数的对象的后面
void MyString::append(MyString &s)
{
    len += strlen(s.get_str()); // 扩充长度
    if(capacity < len){
        capacity = len; // 扩充容量
        // 扩充空间
        char* temp_str = new char[len + 1];
        if(!temp_str) // 注意检查是否分配成功
        {
            cout << "insufficient memory" << endl;
            exit(1);
        }
        // 把原字符串复制过去
        strcpy(temp_str, str);
        delete[] str; // 回收原字符串占据的空间
        str = temp_str; // 指针重定向
    }
    strcat(str, s.get_str()); // 连接
}

void MyString::show()
{
    cout << str;
}
```

```cpp
#include <iostream>
#include "MyString.h"

using namespace std;

int main()
{
    MyString str1("long"), str2("-term");
    str1.show();
    cout << endl;
    str2.show();
    cout << endl;
    str1.append(str2);
    str1.show();
    cout << endl;
    str1.assign(str2);
    str1.show();
    cout << endl;
    // 析构函数自动释放内存
    return 0;
}
```

```
#ifndef MY_VECTOR_H
#define MY_VECTOR_H
#include <iostream>

using namespace std;

const int OK = 1;
const int ERROR = 0;
const int INIT_CAPACITY = 4;

class MyVector{
    private:
        int *arr; // 数组首元素地址
        int array_size; // 数组大小
        int capacity; // 数组容量

    public:
        MyVector();
        MyVector(int *a, int sz);
        ~MyVector();

        /*返回下标为i的元素（即第i+1个元素）的值*/
        int get_element(int i, int &e);

        /*返回最后一个值为e的元素的下标；若满足条件的元素不存在，则返回-1，*/
        int locate_element(int e);

        /*在下标为i的位置之前插入元素e，i的范围是[0, array_size]*/
        int insert_element(int i, int e);

        /*删除下标为i的元素*/
        int delete_element(int i);

        /*输出迭代器的内容*/
        void show();
};

#endif
```

```cpp
#include <iostream>
#include "MyVector.h"

// 无参构造函数
MyVector::MyVector()
{
    capacity = INIT_CAPACITY;
    arr = new int[capacity];
    if(!arr)
    {
        cout << "insufficient memory" << endl;
        exit(1);
    }
    array_size = 0;
}
```

```
// 带参构造函数
MyVector::MyVector(int *a, int sz)
{
    // 确定容量
    capacity = INIT_CAPACITY>sz?INIT_CAPACITY:sz;
    // 分配空间
    arr = new int[capacity];
    if(!arr)
    {
        cout << "insufficient memory" << endl;
        exit(1);
    }
    // 复制
    for(int i = 0; i < sz; i++)
    {
        arr[i] = a[i];
    }
    array_size = sz; // 维护数组大小
}
```

```cpp




// 析构函数
MyVector::~MyVector()
{
    delete[] arr;
}

// 取值
int MyVector::get_element(int i, int &e)
/*返回下标为i的元素（即第i+1个元素）的值*/
{
    if(i < 0 || i >= array_size) // 访问越界
    {
        return ERROR;
    }
    e = arr[i];
    return OK;
}
```

```
// 定位
int MyVector::locate_element(int e)
/*返回最后一个值为e的元素的下标；若满足条件的元素不存在，则返回-1，*/
{
    int i;
    // 从后往前搜查
    for(i = array_size - 1; i >= 0; i--)
    {
        if(arr[i] == e) // 找到
        {
            break;
        }
    }
    return i;
}
```

```






// 插入
int MyVector::insert_element(int i, int e)
/*在下标为i的位置之前插入元素e, i的范围是[0,
    array_size], 允许在末端插入元素*/
{
    if(i < 0 || i > array_size)
    {
        return ERROR; // 不在合法范围内
    }

    if(capacity < array_size + 1) // 容量不足
    {
        capacity *= 2; // 容量增大为2倍
        int* new_arr = new int[capacity]; // 分配空
    间
```

```
        if(!new_arr)
        {
            cout << "insufficient memory" << endl;
            exit(1);
        }
        int j;
        // 把前面i个旧元素复制过去
        for(j = 0; j < i; j++)
        {
            new_arr[j] = arr[j];
        }
        new_arr[j++] = e; // 插入元素
        for(; j < array_size + 1; j++)
        {
            // 复制发生偏移
            new_arr[j] = arr[j - 1];
        }
        delete[] arr; // 销毁原数组
        arr = new_arr; // 关联新数组
    }
    else
    {
        for(int j = array_size - 1; j >= i; --j)
        {
            arr[j + 1] = arr[j]; // 元素后移，包括下标为i的元素
        }
        arr[i] = e;
    }
    array_size++;
    return OK;
}

// 删除
int MyVector::delete_element(int i)
/*删除下标为i的元素*/
{
    if(i < 0 || i > array_size - 1)
    {
        return ERROR;
    }
    // 下标从i+1开始是后面的元素，直到最后一个
    for(int j = i + 1; j < array_size; j++)
    {
        arr[j - 1] = arr[j]; // 前移
    }
    array_size--;
    return OK;
}

// 打印
void MyVector::show()
{
    for(int i = 0; i < array_size; i++)
    {
        cout << arr[i] << '\t';
    }
}
```

```cpp
#include <iostream>
#include "MyVector.h"

using namespace std;

const int FAILURE = 1;

int main()
{
    // 创建
    int a1[] = {1, 2, 3};
    int a2[] = {5, 4, 3, 2, 1};
    MyVector vec1(a1, 3), vec2(a2, 5);
    cout << "vec1:" << endl;
    vec1.show();
    cout << endl;
    cout << "vec2:" << endl;
    vec2.show();
    cout << endl;

    // 取值
    int e;
    if(!vec1.get_element(2, e))
    {
        return FAILURE;
    }
    cout << "The 3rd element in vec1 is " << e << endl;
    if(!vec2.get_element(3, e))
    {
        return FAILURE;
    }
    cout << "The 4th element in vec2 is " << e << endl;

    // 定位
    int index = vec1.locate_element(4);
    if(index == -1)
    {
        cout << "4 is not found in vec1" << endl;
    }
    else
    {
        cout << "the index of 4 in vec1 is " <<
   index << endl;
    }

    index = vec2.locate_element(4);
    if(index == -1)
    {
        cout << "4 is not found in vec2" << endl;
    }
    else
    {
        cout << "the index of 4 in vec2 is " <<
   index << endl;
    }

    // 插入
    cout << "try to insert 8 into vec1 before
   location 4" << endl;
    if(!vec1.insert_element(4, 8))
    {
        cout << "inserting 8 into vec1 before location 4 causes failures" << endl;
    }
    cout << "vec1:" << endl;
    vec1.show();
    cout << endl;

    cout << "try to insert 8 into vec1 before location 2" << endl;
    if(!vec1.insert_element(2, 8))
    {
        cout << "inserting 8 into vec1 before location 2 causes failures" << endl;
    }
    cout << "vec1:" << endl;
    vec1.show();
    cout << endl;

    cout << "try to insert 7 into vec2 before location 5" << endl;
    if(!vec2.insert_element(5, 7))
    {
        cout << "inserting 7 into vec2 before location 5 causes failures" << endl;
    }
    cout << "vec2:" << endl;
    vec2.show();
    cout << endl;

    cout << "try to delete the element at location 1 in vec1" << endl;
    if(!vec1.delete_element(1))
    {
        cout << "deleting the element at location 1 in vec1 failed" << endl;
    }
    cout << "vec1:" << endl;
    vec1.show();
    cout << endl;

    cout << "try to delete the element at location 5 in vec2" << endl;
    if(!vec2.delete_element(5))
    {
        cout << "deleting the element at location 5 in vec2 failed" << endl;
    }
    cout << "vec1:" << endl;
    vec2.show();
    cout << endl;

    // 无需手工调用析构函数
    return 0;
} /*g++ .\testMyVector.cpp .\MyVector.cpp -o .\test */
```

```
#ifndef MY_QUEUE_H
#define MY_QUEUE_H

const int INIT_SIZE = 4;
const int OK = 1;
const int ERROR = 0;

```

```
class MyQueue{
/*用长度为n+1的数组来实现长度不超过n的动态队列，
当空间不足时，动态扩充*/
    private:
        int *arr; // 数组首元素地址
        int array_capacity; // 数组容量
        int head; // 队首下标
        int tail; // 队尾的下一个元素的下标
        /*当head==tail时，队列为空；当tail+1==head时，队列为满。*/

    public:
        MyQueue();

        // 根据数组构造队列
        MyQueue(int *a, int n);
        ~MyQueue();

        // 入队，空间不足时扩充
        void enQueue(int x);
        // 出队，返回出队的元素的值
        int deQueue();
        void show();
};

#endif
```

```cpp
#include <iostream>
#include <stdio.h>
#include "MyQueue.h"

using namespace std;

// 无参构造函数
MyQueue::MyQueue()
{
    array_capacity = INIT_SIZE; // 确定容量
    arr = new int[array_capacity + 1]; // 分配空间
    if(!arr)
    {
        cout << "insufficient memory" << endl;
        exit(1);
    }
    head = tail = 0; // 空队列的头尾指针相等
}



```

```cpp


// 带参构造函数
MyQueue::MyQueue(int* a, int n)
{
    // 确定容量
    array_capacity = INIT_SIZE > n ? INIT_SIZE : n;
    // 分配空间
    arr = new int[array_capacity + 1];
    if(!arr)
    {
        cout << "insufficient memory" << endl;
        exit(1);
    }
    // 空队列的头尾指针相等
    head = 0;
    for(int i = 0; i < n; i++)
    {
        arr[i] = a[i]; // 填入
    }
    tail = n; // 最后一个元素的下标加1
```

```
}



// 析构函数
MyQueue::~MyQueue()
{
    delete[] arr;
}

// 打印
void MyQueue::show()
{
    for(int i = head; i != tail;)
    {
        cout << arr[i] << '\t';
        // 注意数组长度为capacity+1
        if(i == array_capacity)
        {
            i = 0; // 自增后回到数组首元素位置
        }
        else
        {
            i++;
        }
    }
}

// 入队
void MyQueue::enQueue(int x)
{
    if(head == (tail + 1) % (array_capacity + 1)) //满队
    {
        // 多一倍空间，肯定不需要折回，可按通常数组处理
        int* new_arr = new int[array_capacity * 2 + 1];
        int j = head;
        /*i用于扫描原数组，j用于扫描新数组*/
        for(int i = head; i != tail; j++)
        /*head下标保持不变，因此i和j起点相同*/
        {
            new_arr[j] = arr[i];
            if(i == array_capacity) // 原数组已扫到尽头
            {
                i = 0; // 折回
            }
            else
            {
                i++;
            }
        } // 循环结束时，j为有效元素的下一个位置的下标
        delete[] arr; // 销毁原数组
        arr = new_arr; // 关联新数组
        tail = j; // 维护队尾下一个位置的下标
        array_capacity *= 2;
    } // 循环结束，不满队
    arr[tail] = x;
    if(tail == array_capacity)
    {
```

```cpp
        tail = 0;
    }
    else
    {
        tail++;
    }
}








// 出队
int MyQueue::deQueue()
{
    if(tail == head)
    {
        cout << "underflow" << endl;
        return ERROR;
    }
    int x = arr[head]; // 获取队首元素
    if(head == array_capacity)
    {
        head = 0; // 折回
    }
    else
    {
        head++;
    }
    return x;
}
```

```cpp
#include <iostream>

#include "MyQueue.h"

using namespace std;

int main()
{
    int a1[] = {2, 3, 5, 7};
    int a2[] = {11, 13, 17, 19};
    MyQueue que1(a1, 4), que2(a2, 4);

    cout << "que1:" << endl;
    que1.show();
    cout << endl;

    cout << "que2:" << endl;
    que2.show();
    cout << endl;

    int x = 8;
    que1.enQueue(x); // que1.enQueue(4);
    cout << "appended " << x << " into que1" << endl;
    cout << "que1:" << endl;
    que1.show();
    cout << endl;

    x = 6;
    que2.enQueue(x);
    cout << "appended " << x << " into que2" << endl;
    x = 4;
    que2.enQueue(x);
    cout << "appended " << x << " into que2" << endl;
    cout << "que2:" << endl;
    que2.show();
    cout << endl;

    x = que1.deQueue();
    cout << "obtained " << x << " from que1" <<
```

```
   endl;
    cout << "que1:" << endl;
    que1.show();
    cout << endl;

    x = que2.deQueue();
    cout << "obtained " << x << " from que2" <<
   endl;
    x = que2.deQueue();
    cout << "obtained " << x << " from que2" <<
   endl;
    cout << "que2:" << endl;
    que2.show();
    cout << endl;

    x = 3;
    que1.enQueue(x);
    cout << "appended " << x << " into que1" <<
   endl;
    cout << "que1:" << endl;
    que1.show();
```

```cpp
    cout << endl;

    x = 4;
    que2.enQueue(x);
    cout << "appended " << x << " into que2" << endl;
    cout << "que2:" << endl;
    que2.show();
    cout << endl;

    return 0;
}
```