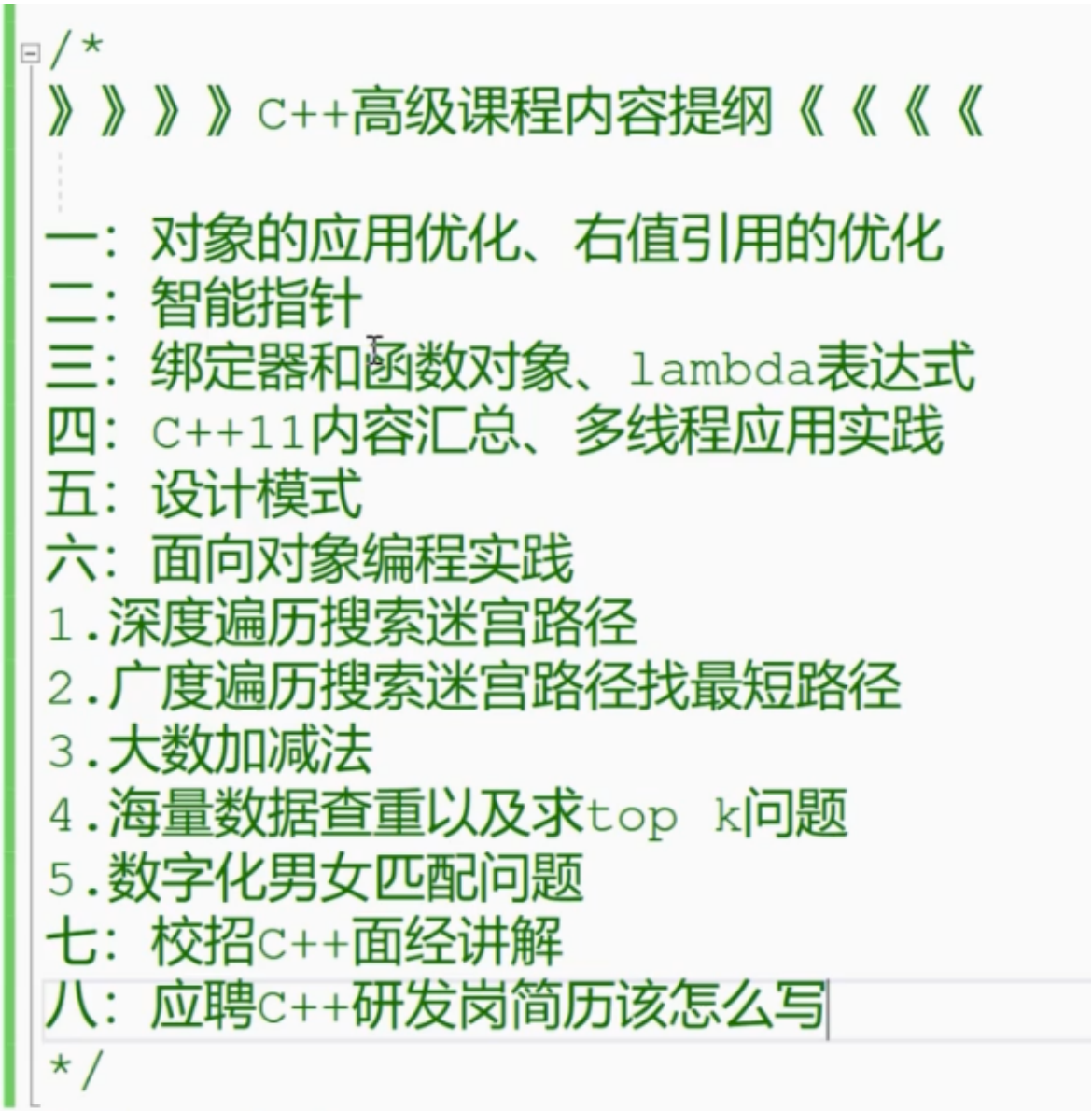
一、内容提纲



二、对象被优化以后才是高效的C++编程

1. 对象使用过程中背后调用了哪些方法

```
#include <iostream>

using namespace std;

class Test
{
public:
    Test(int a = 10) :ma(a) { cout << "Test()" << endl; }
    ~Test() { cout << "~Test()" << endl; }
    Test(const Test& t) : ma(t.ma) { cout << "Test(const Test&)" << endl; }
    Test& operator=(const Test& t)
```

```
    {
        cout << "operator =" << endl;
        ma = t.ma;
        return *this;
    }
private:
    int ma;
};

int main()
{
    Test t1;
    Test t2(t1);
    Test t3 = t1;
    t3 = t1;
    cout << "-------t3-------" << endl;
    t3 = Test(30); //临时对象必须生成 生存周期:所在语句
    cout << "-------t3-------" << endl;
    //显示生成临时对象
    //C++编译器对于对象构造的优化:用临时对象生成新对象时,
    //临时对象就不产生了,直接构造新对象就可以了
    //隐式生成临时对象
    Test t4 = Test(20); //和 Test t4(20); 没有区别

    //显示生成临时对象
    //构造 =重载赋值 析构
    t4 = (Test)30; //int -> Test(int)
    t4 = 30; //int -> Test(int)
    cout << "----------" << endl;

    Test* p = &Test(40); //出了语句对象析构了 不应该这样 不安全
    const Test& ref = Test(50); //没有析构,相当于别名

    cout << "----------" << endl;
    return 0;
}
/*
Test()
Test(const Test&)
Test(const Test&)
operator =
-------t3-------
Test()
operator =
~Test()
-------t3-------
Test()
Test()
operator =
~Test()
Test()
operator =
~Test()
----------
Test()
~Test()
Test()
----------
```

```
~Test()
~Test()
~Test()
~Test()
~Test()
*/
```

```
#include <iostream>

using namespace std;

class Test
{
public:
    //Test() / Test(10)只传了a / Test(int, int)
    Test(int a = 10, int b = 10) :ma(a), mb(b) { cout << "Test(int, int)" <<
endl; }
    ~Test() { cout << "~Test()" << endl; }
    Test(const Test& t) : ma(t.ma), mb(t.mb) { cout << "Test(const Test&)" <<
endl; }
    Test& operator=(const Test& t)
    {
        cout << "operator =" << endl;
        ma = t.ma;
        mb = t.mb;
        return *this;
    }
private:
    int ma, mb;
};

Test t1(10, 10); //1. Test(int, int)
int main()
{
    Test t2(20, 20); //3. Test(int, int)
    Test t3 = t2; //Test (const Test&)
    static Test t4 = Test(30, 30); //static Test t4(30, 30);
    t2 = Test(40, 40); //Test(int, int) -> operator= -> ~Test()

    //类型强转 (a=50, b=10)
    //括号表达式是最后一个
    t2 = (Test)(50, 50); //Test(int, int) operator= ~Test()
    t2 = 60;

    Test* p1 = new Test(70, 70); //Test(int, int)
    Test* p2 = new Test[2]; //两次构造
    Test* p3 = &Test(80, 80); //指针指向临时对象 语句结束析构
    const Test& p4 = Test(90, 90); //Test(int, int)
    cout << "-----------" << endl;
    delete p1; //~Test()
    delete []p2; //两次~Test()
    cout << "-----------" << endl;
    return 0;
}
Test t5(100, 100); //2.Test(int, int)
```

```
/*
Test(int, int)
Test(int, int)
Test(int, int)
Test(const Test&)
Test(int, int)
Test(int, int)
operator =
~Test()
Test(int, int)
operator =
~Test()
Test(int, int)
operator =
~Test()
Test(int, int)
Test(int, int)
Test(int, int)
Test(int, int)
~Test()
Test(int, int)
-----------
~Test()
~Test()
~Test()
-----------
~Test()
~Test()
~Test()
~Test()
~Test()
~Test()
*/
```

2. 函数调用过程中对象背后调用的方法太多

```
#include <iostream>

using namespace std;

class Test
{
public:
    Test(int data = 10)
        :ma(data)
    {
        cout << "Test(int)" << endl;
    }
    ~Test()
    {
        cout << "~Test()" << endl;
    }
    Test(const Test& t) :ma(t.ma)
    {
        cout << "Test(const Test&)" << endl;
```

```
    }
    void operator=(const Test& t)
    {
        cout << "operator=" << endl;
        ma = t.ma;
    }
    int getData() const { return ma; }
private:
    int ma;
};

//不能返回局部或临时的指针或引用
Test GetObject(Test t) //3. Test(const Test&)
{
    int val = t.getData();
    Test tmp(val); //4. Test(int)
    return tmp; //5. Test(const Test&)  tmp不能被拿出来
    //6. ~Test() tmp
    //7. ~Test() t
}

int main()
{
    Test t1; //1. Test(int)
    Test t2; //2. Test(int)
    //实参到形参,初始化(两个对象)
    t2 = GetObject(t1); //8 operator=
                        //9 ~Test() 临时对象析构
    cout << "-----" << endl;
    return 0;
}
/*
Test(int)
Test(int)
Test(const Test&)
Test(int)
Test(const Test&)
~Test()
~Test()
operator=
~Test()
-----
~Test()
~Test()
*/
```

3. 总结三条对象优化的规则

1. 函数参数传递过程中，对象优先按引用传递，不要按值传递
2. 当函数返回对象的时候，应该优先返回一个临时对象，而不要返回一个定义过的对象
3. 接收返回值是对象的函数调用的时候，优先按初始化的方式接收，不要按赋值的方式接收

用临时对象拷贝构造同类型的新对象时有优化。从上面的 11 行优化成了 4 行

```
Test GetObject(const Test& t)
{
    int val = t.getData();
    /*Test tmp(val);   //减少构造和析构
    return tmp;*/
    return Test(val); //直接构造main()上的临时对象
}

int main()
{
    Test t1; //1. Test()
    Test t2; //2. Test()

    t2 = GetObject(t1); //3. Test() 直接构造在main上
                        //4. operator=
                        //5. ~Test()
    cout << "-----" << endl;
    //6. ~Test()
    //7. ~Test()
    return 0;
}
```

```
Test GetObject(const Test& t)
{
    int val = t.getData();
    return Test(val); //直接构造main()上的临时对象
}

int main()
{
    Test t1; //1. Test()
    //之前是先定义t2, t2要构造
    //这里是初始化,不是赋值
    Test t2 = GetObject(t1); //Test() 直接构造t2
    cout << "-----" << endl;
    //3. ~Test()
    //4. ~Test()
    return 0;
}
```

4. CMyString的代码问题

想法：B 找 A 要东西，如果 A 不要了，A 大可直接把东西给 B，而不是让 B 拷贝一份后， A 把原来的丢掉

当这个东西特别大时，原来的方法需要改变

5. 添加带右值引用参数的拷贝构造和赋值函数

通俗来讲：

左值：有内存、有名字

右值：没名字（临时量）**或** 没内存

```
//数字没有内存，在寄存器
/*
    int tmp = 20;
    const int &&b = tmp;
    */
int&& a = 10;
/*
    int tmp = 20;
    const int &b = tmp;
    */
const int& b = 20;
```

```
String&& a = String("aaa"); //String("aaa")是右值：没有名字
String& d = a; //右值本身是左值：既有名字又有内存
String b = String("bbb");
String& c = b; //左值引用
```

具体例子：用 **带右值引用参数的拷贝构造 和 带右值引用参数的赋值重载函数** 实现

```
#include <iostream>

using namespace std;
#if 1
class String
{
public:
    String(const char* p = nullptr)
    {
        cout << "String()" << endl;
        if (p != nullptr)
        {
            _pstr = new char[strlen(p) + 1];
            strcpy(_pstr, p);
        }
        else
        {
            _pstr = new char[1];
            *_pstr = '\0';
        }
    }
    ~String()
    {
        cout << "~String()" << endl;
        delete[]_pstr;
        _pstr = nullptr;
    }
    //带左值引用参数的拷贝构造
    String(const String& str)
    {
        cout << "String(const String& str)" << endl;
        _pstr = new char[strlen(str._pstr) + 1];
        strcpy(_pstr, str._pstr);
```

```
    }
    //带右值引用参数的拷贝构造
    String(String&& str) //str引用的是一个临时变量
    {
        cout << "String(String&& str)" << endl;
        _pstr = str._pstr;
        str._pstr = nullptr;
    }
    //带左值引用参数的赋值重载函数
    String& operator=(const String& src)
    {
        cout << "String& operator=(const String&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = new char[strlen(src._pstr) + 1];
        strcpy(_pstr, src._pstr);
        return *this;
    }
    //带右值引用参数的赋值重载函数
    String& operator=(String&& src)
    {
        cout << "String& operator=(String&&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = src._pstr;
        src._pstr = nullptr;
        return *this;
    }


    const char* c_str() const { return _pstr; }
private:
    char* _pstr;
};

String GetString(String& str)
{
    const char* pstr = str.c_str();
    String tmpStr(pstr); //3. String()
    return tmpStr;
    //4. String(&&) 带出去给main
    //5. ~String()
}

int main()
{
    String str1("aaaaa"); //1. String()
    String str2; //2. String()
    str2 = GetString(str1); //6.String& operator=(String&&)
                            //7.~String() 析构传出来的

    cout << str2.c_str() << endl;
    return 0;
    //8. ~String()
    //9. ~String()
}
```

```
#endif
```

6. CMyString在vector上的应用

```
#include <iostream>

using namespace std;
#if 1
class String
{
public:
    String(const char* p = nullptr)
    {
        cout << "String()" << endl;
        if (p != nullptr)
        {
            _pstr = new char[strlen(p) + 1];
            strcpy(_pstr, p);
        }
        else
        {
            _pstr = new char[1];
            *_pstr = '\0';
        }
    }
    ~String()
    {
        cout << "~String()" << endl;
        delete[]_pstr;
        _pstr = nullptr;
    }
    //带左值引用参数的拷贝构造
    String(const String& str)
    {
        cout << "String(const String& str)" << endl;
        _pstr = new char[strlen(str._pstr) + 1];
        strcpy(_pstr, str._pstr);
    }
    //带右值引用参数的拷贝构造
    String(String&& str) //str引用的是一个临时变量
    {
        cout << "String(String&& str)" << endl;
        _pstr = str._pstr;
        str._pstr = nullptr;
    }
    //带左值引用参数的赋值重载函数
    String& operator=(const String& src)
    {
        cout << "String& operator=(const String&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = new char[strlen(src._pstr) + 1];
        strcpy(_pstr, src._pstr);
        return *this;
```

```
    }
    //带右值引用参数的赋值重载函数
    String& operator=(String&& src)
    {
        cout << "String& operator=(String&&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = src._pstr;
        src._pstr = nullptr;
        return *this;
    }
    const char* c_str() const { return _pstr; }
private:
    char* _pstr;
    friend String operator+(const String& lhs, const String& rhs);
    friend ostream& operator<<(ostream& out, const String& str);
};

String operator+(const String& lhs,
    const String& rhs)
{
    String tmpStr;
    tmpStr._pstr = new char[strlen(lhs._pstr) + strlen(rhs._pstr) + 1];
    strcpy(tmpStr._pstr, lhs._pstr);
    strcat(tmpStr._pstr, rhs._pstr);
    return tmpStr; //右值引用, 存放数据的地方不改变
}

ostream& operator<<(ostream& out, const String& str)
{
    out << str._pstr;
    return out;
}

String GetString(String& str)
{
    const char* pstr = str.c_str();
    String tmpStr(pstr);
    return tmpStr;
}

int main()
{
    String str1 = "Hello ";
    String str2 = "world!";
    cout << "---------" << endl;
    String str3 = str1 + str2;
    cout << "---------" << endl;
    cout << str3 << endl;
    return 0;
}
#endif

/*
String()
String()
---------
```

```
String()
String(String&& str)
~String()
---------
Hello world!
~String()
~String()
~String()
*/
```

右值引用对效率的优化体验

```
String str1 = "aaa";
vector<String>vec;
vec.reserve(10);
cout << "-----------" << endl;
vec.push_back(str1);
vec.push_back(String("bbb"));
cout << "-----------" << endl;

/*
String(const char*)
-----------
String(const String& str)
String(const char*)
String(String&& str)
~String()
-----------
~String()
~String()
~String()
*/
```

7.move移动语义和forward类型完美转发

```
#include <iostream>

using namespace std;

/*
容器的空间配置器
*/
template<typename T>
class Allocator
{
public:
    T* allocate(size_t size) //负责内存开辟
    {
        return (T*)malloc(sizeof(T) * size);
    }
    void deallocate(void* p) //负责内存释放
    {
        free(p);
    }
```

```
    void construct(T* p, const T& val) //负责对象构造
    {
        new (p) T(val); //定位new  指定内存上构造
    }
    void construct(T* p, T&& val) //右值 负责对象构造
    {
        new (p) T(val);
    }
    void destroy(T* p) //负责对象析构
    {
        p->~T(); //~T()代表了T类型的析构函数
    }
}
;


/*
实现 vector 向量容器
容器底层内存开辟,内存释放,对象构造和析构,都通过allocator空间配置器来实现
*/
template<typename T, typename Alloc = Allocator<T>>
class vector
{
public:
    vector(int size = 10)
    {
        //需要把内存开辟和对象构造分开处理
        //_first = new T[size];
        _first = _allocator.allocate(size);
        _last = _first;
        _end = _first + size;
    }
    ~vector()
    {
        //delete[]_first;
        for (T* p = _first; p != _last; p++)
        {
            _allocator.destroy(p); //把_first指针指向的有效元素析构
        }
        _allocator.deallocate(_first); //释放堆上的内存
        _first = _last = _end = nullptr;
    }
    vector(const vector<T>& rhs)
    {
        int size = rhs._end - rhs._first;
        //_first = new T[size];
        _first = _allocator.allocate(size);
        int len = rhs._last - rhs._first;
        for (int i = 0; i < len; ++i)
        {
            //_first[i] = rhs._first[i];
            _allocator.construct(_first + 1, rhs._first[i]);
        }
        _last = _first + len;
        _end = _first + size;
    }
    vector<T>& operator=(const vector<T>& rhs)
    {
```

```
        if (this == &rhs)
            return *this;
        //delete[]_first;
        for (T* p = _first; p != _last; p++)
        {
            _allocator.destroy(p); //把_first指针指向的有效元素析构
        }
        _allocator.deallocate(_first);

        int size = rhs._end - rhs._first;
        //_first = new T[size];
        _first = _allocator.allocate(size);
        int len = rhs._last - rhs._first;
        for (int i = 0; i < len; ++i)
        {
            //_first[i] = rhs._first[i];
            _allocator.construct(_first + 1, rhs._first[i]);
        }
        _last = _first + len;
        _end = _first + size;

        return *this;
    }

    void pop_back()
    {
        if (empty())
            return;
        //--_last;
        //析构删除的元素
        --_last;
        _allocator.destroy(_last);
    }
    T back() const
    {
        return *(_last - 1); //空的情况没写
    }
    bool full() const { return _last == _end; }
    bool empty() const { return _first == _last; }
    int size() const { return _last - _first; }

    ////////////////////////////
    //左值
    void push_back(const T& val)
    {
        if (full())
            expand();
        _allocator.construct(_last, val);
        _last++;
    }
    //右值
    void push_back(T&& val) //一个右值引用变量本身还是一个左值
    {
        if (full())
            expand();
        _allocator.construct(_last, val);
        _last++;
    }
```

```
private:
    T* _first; //起始
    T* _last; //有效元素的后继位置
    T* _end; //数组空间的后继位置
    Alloc _allocator; //定义容器的空间配置器对象

    void expand()
    {
        int size = _end - _first;
        //T* ptmp = new T[2 * size];
        T* ptmp = _allocator.allocate(2 * size);
        for (int i = 0; i < size; i++)
        {
            _allocator.construct(ptmp + i, _first[i]);
            //ptmp[i] = _first[i];
        }
        //delete[]_first;
        for (T* p = _first; p != _last; ++p)
        {
            _allocator.destroy(p);
        }
        _allocator.deallocate(_first);
        _first = ptmp;
        _last = _first + size;
        _end = _first + size * 2;
    }
};

class Test
{
public:
    Test() { cout << "Test()" << endl; }
    ~Test() { cout << "~Test()" << endl; }
    Test(const Test&) { cout << "Test(const Test&)" << endl; }
};

class String
{
public:
    String(const char* p = nullptr)
    {
        cout << "String()" << endl;
        if (p != nullptr)
        {
            _pstr = new char[strlen(p) + 1];
            strcpy(_pstr, p);
        }
        else
        {
            _pstr = new char[1];
            *_pstr = '\0';
        }
    }
    ~String()
    {
        cout << "~String()" << endl;
        delete[]_pstr;
        _pstr = nullptr;
```

```
    }
    //带左值引用参数的拷贝构造
    String(const String& str)
    {
        cout << "String(const String& str)" << endl;
        _pstr = new char[strlen(str._pstr) + 1];
        strcpy(_pstr, str._pstr);
    }
    //带右值引用参数的拷贝构造
    String(String&& str) //str引用的是一个临时变量
    {
        cout << "String(String&& str)" << endl;
        _pstr = str._pstr;
        str._pstr = nullptr;
    }
    //带左值引用参数的赋值重载函数
    String& operator=(const String& src)
    {
        cout << "String& operator=(const String&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = new char[strlen(src._pstr) + 1];
        strcpy(_pstr, src._pstr);
        return *this;
    }
    //带右值引用参数的赋值重载函数
    String& operator=(String&& src)
    {
        cout << "String& operator=(String&&)" << endl;
        if (this == &src)
            return *this;
        delete[]_pstr;
        _pstr = src._pstr;
        src._pstr = nullptr;
        return *this;
    }
    const char* c_str() const { return _pstr; }
private:
    char* _pstr;
    friend String operator+(const String& lhs, const String& rhs);
    friend ostream& operator<<(ostream& out, const String& str);
};

String operator+(const String& lhs,
    const String& rhs)
{
    String tmpStr;
    tmpStr._pstr = new char[strlen(lhs._pstr) + strlen(rhs._pstr) + 1];
    strcpy(tmpStr._pstr, lhs._pstr);
    strcat(tmpStr._pstr, rhs._pstr);
    return tmpStr; //右值引用，存放数据的地方不改变
}

ostream& operator<<(ostream& out, const String& str)
{
    out << str._pstr;
    return out;
```

```
}

String GetString(String& str)
{
    const char* pstr = str.c_str();
    String tmpStr(pstr);
    return tmpStr;
}

int main()
{
    String str1 = "aaa";
    vector<String>vec;
    cout << "-----------" << endl;
    vec.push_back(str1);
    vec.push_back(String("bbb"));
    cout << "-----------" << endl;
    return 0;
}
/*
String()
-----------
String(const String& str)
String()
String(const String& str)
~String()
-----------
~String()
~String()
~String()
*/
```

采用 `move()` 将左值强转为右值

```
void construct(T* p, T&& val) //右值 负责对象构造
{
    new (p) T(move(val));
}

void push_back(T&& val) //一个右值引用变量本身还是一个左值
{
    if (full())
        expand();
    _allocator.construct(_last, std::move(val));
    _last++;
}
/*
String()
-----------
String(const String& str)
String()
String(String&& str)
~String()
-----------
~String()
~String()
~String()
```

```
*/
```

引用折叠：左值加右值为左值；右值加右值为右值

```
template<typename Ty>
void construct(T* p, Ty&& val)
{
    new (p) T(forward<Ty>(val));
}

template<typename Ty>
void push_back(Ty&& val) //引用折叠
{
    if (full())
        expand();
    //move：移动语义，得到右值类型
    //forward：类型的完美转发
    _allocator.construct(_last, std::forward<Ty>(val)); //左值变左值 右值变右值
    _last++;
}
```

三、体验一下智能指针的强大

1. 基础知识

原来用的叫“裸指针”，在释放时需要手动释放。可能出现些问题。

智能指针 保证能做到资源的自动释放

利用栈上的对象出作用域自动析构的特征，来做到资源的自动释放。

```
#include<iostream>

using namespace std;

template<typename T>
class CSmartPtr
{
public:
    CSmartPtr(T *ptr = nullptr)
        :mptr(ptr) {
        //cout << "CSmartPtr(T *ptr)" << endl;
    }
    ~CSmartPtr() {
        //cout << "~CSmartPtr()" << endl;
        delete mptr;
    }
    T& operator*() { return *mptr; } //返回引用,可改变内存
    T* operator->() { return mptr; } //将->的返回值返回
private:
    T* mptr;
};

class Test
```

```
{
public:
    void test() { cout << "call Test::test()" << endl; }
};

int main()
{
    CSmartPtr<int>ptr1(new int);
    *ptr1 = 20;
    cout << *ptr1 << endl;
    CSmartPtr<Test>ptr2(new Test());
    (*ptr2).test();
    ptr2->test();
    return 0;
}
```

2. 不带引用计数的智能指针

怎么解决浅拷贝的问题?

不带引用计数的智能指针

auto_ptr : C++库里面

C++11新标准: **scoped_ptr**　　**unique_ptr**

auto_ptr

```
//不推荐使用 auto_ptr
//如: 容器经常拷贝　　所以除非使用场景非常简单,不要用auto_ptr
auto_ptr<int> ptr1(new int);
auto_ptr<int> ptr2(ptr1); //ptr2代替ptr1 ptr1置空
*ptr2 = 20;
cout << *ptr1 << endl; //error　被释放了
```

scoped_ptr

```
scoped_ptr(const scoped_ptr<T>&) = delete;
scoped_ptr<T>& operator=(const scoped_ptr<T>&) = delete;
```

任何地方使用拷贝构造 或 赋值，编译器报错。

unique_ptr

```
unique_ptr(const unique_ptr<T>&) = delete;
unique_ptr<T>& operator=(const unique_ptr<T>&) = delete;
unique_ptr(unique_ptr<T>&&src) = delete;
unique_ptr<T>& operator=(unique_ptr<T>&&src) = delete;
```

```
unique_ptr<int> p1(new int);
//std::move得到当前变量的右值类型
unique_ptr<int> p2(move(p1));  //把p1的资源给p2 所以不要再访问p1
*p2 = 1;
cout << *p2;
```

3. 实现带引用指针计数的智能指针

shared_ptr 和 weak_ptr

带引用计数：多个智能指针可以管理同一个资源

带引用计数：给每个对象资源，匹配一个引用计数

最后一个析构时释放资源

```
#include <iostream>
#include <memory>

using namespace std;

//对资源进行引用计数的类
template<typename T>
class RefCnt
{
public:
    RefCnt(T *ptr = nullptr)
        :mptr(ptr)
    {
        if (mptr != nullptr)
            mcount = 1;
    }
    void addRef() { mcount++; } //添加引用计数
    int delRef() { return --mcount; }
private:
    T* mptr;
    int mcount; //atomic_int 原子整形类
};

template<typename T>
class CSmartPtr
{
public:
    CSmartPtr(T *ptr = nullptr)
        :mptr(ptr) {
        //cout << "CSmartPtr(T *ptr)" << endl;
        mpRefCnt = new RefCnt<T>(mptr);
    }
    ~CSmartPtr() {
        //cout << "~CSmartPtr()" << endl;
        if (0 == mpRefCnt->delRef())
        {
            delete mptr;
            mptr = nullptr;
        }
```

```
    }
    T& operator*() { return *mptr; } //返回引用,可改变内存
    T* operator->() { return mptr; } //将->的返回值返回

    CSmartPtr(const CSmartPtr<T>& src)
        :mptr(src.mptr), mpRefCnt(src.mpRefCnt)
    {
        if (mptr != nullptr)
            mpRefCnt->addRef();
    }
    CSmartPtr<T>& operator=(const CSmartPtr<T>& src)
    {
        if (this == &src)
            return *this;
        mpRefCnt->delRef();
        if (0 == mpRefCnt->delRef())
        {
            delete mptr;
        }
        mptr = src.mptr;
        mpRefCnt = src.mpRefCnt;
        mpRefCnt->addRef();
        return *this;
    }
private:
    T* mptr; //指向资源的指针
    RefCnt<T>* mpRefCnt; //指向该资源引用计数对象的指针
};

int main()
{
    CSmartPtr<int>ptr1(new int);
    CSmartPtr<int>ptr2(ptr1);
    CSmartPtr<int> ptr3;
    CSmartPtr<int> ptr4(new int);
    ptr3 = ptr2;

    *ptr1 = 20;
    *ptr4 = 30;
    cout << *ptr1 << ' ' << *ptr2 << ' ' << *ptr3 << ' ' << *ptr4 << endl;
    return 0;
}
```

4. shared_ptr的交叉引用问题

shared_ptr：强智能指针 可以改变资源的引用计数

weak_ptr：弱智能指针 不会改变资源的引用计数（**只能观察**资源还活着没）

weak_ptr =》shared_ptr =》资源（内存）

强智能指针循环引用（交叉引用）是什么问题？什么结果？怎么解决？

```
#include <iostream>
#include <memory>
```

```
using namespace std;

class B;
class A
{
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
    shared_ptr<B> _ptrb;
};
class B
{
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    shared_ptr<A> _ptra;
};

int main()
{
    shared_ptr<A> pa(new A());
    shared_ptr<B> pb(new B());

    pa->_ptrb = pb;
    pb->_ptra = pa;

    cout << pa.use_count() << endl;
    cout << pb.use_count() << endl;

    return 0;
}
/*
A()
B()
2
2
*/
```

new 出来的资源无法释放，资源泄露

定义对象的时候，用强智能指针；引用对象的地方使用弱智能指针。

```
#include <iostream>
#include <memory>

using namespace std;

class B;
class A
{
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
    void funcA() { cout << "funcA()" << endl; }
    weak_ptr<B> _ptrb;
};
class B
```

```
{
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
    void func()
    {
        shared_ptr<A>ps = _ptra.lock(); //提升方法
        if (ps != nullptr) //资源还在
        {
            ps->funcA();
        }

    }
    weak_ptr<A> _ptra;
};

int main()
{
    shared_ptr<A> pa(new A());
    shared_ptr<B> pb(new B());

    pa->_ptrb = pb;
    pb->_ptra = pa;

    cout << pa.use_count() << endl;
    cout << pb.use_count() << endl;

    pb->func();

    return 0;
}
```

5. 多线程访问共享对象的线程安全问题

可能存在访问弱指针指向的对象时，内容已经不在了的情况。

```
class A
{
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
    void testA() { cout << "非常好用的方法" << endl; }
};

//子线程
void handler01(weak_ptr<A> pw) //一定要使用->换为强指针接手
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    //q访问A对象的时候,需要侦测下A对象是否存活
    shared_ptr<A> sp = pw.lock();
    if (sp != nullptr)
    {
        sp->testA();
    }
    else
```

```
    {
        cout << "A已经析构" << endl;
    }
}


int main()
{
    {
        shared_ptr<A>p(new A());
        thread t1(handler01, weak_ptr<A>(p));
        t1.detach(); //分离线程
        std::this_thread::sleep_for(std::chrono::seconds(3));
    }
    std::this_thread::sleep_for(std::chrono::seconds(20));
    return 0;
}
/*
A()
非常好用的方法
~A()
*/
```

将线程的弱指针改为强指针试下

```
//子线程
void handler01(shared_ptr<A> sp) //一定要使用->换为强指针接手
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
    sp->testA();

}

int main()
{
    {
        shared_ptr<A>p(new A());
        thread t1(handler01, shared_ptr<A>(p));
        t1.detach(); //分离线程
    }
    cout << "main里的析构" << endl;
    std::this_thread::sleep_for(std::chrono::seconds(20));
    return 0;
}
/*
A()
main里的析构
非常好用的方法
~A()
*/
```

6. 自定义删除器

删除堆、关闭文件等

可以一种类型写一个类

```
#include <iostream>
#include <memory>
#include <thread>

using namespace std;

template<typename T>
class MyDeletor
{
public:
    void operator()(T* ptr) const
    {
        cout << "call MyDeletor.operator()" << endl;
        delete[]ptr;
    }
};

template<typename T>
class MyFileDeletor
{
public:
    void operator()(T* ptr) const
    {
        cout << "call MyFileDeletor.operator()" << endl;
        fclose(ptr);
    }
};

int main()
{
    unique_ptr<int, MyDeletor<int>> ptr1(new int[100]);
    unique_ptr<FILE, MyFileDeletor<FILE>> ptr2(fopen("D:/qt.txt", "r"));
    return 0;
}
/*
call MyFileDeletor.operator()
call MyDeletor.operator()
*/
```

采用 lambda 表达式

```
int main()
{
    unique_ptr<int, function<void(int*)>> ptr1(new int[100],
        [](int* p)->void {
            cout << "call lambda release int [100]" << endl;
            delete[]p;
        });

    unique_ptr<FILE, function<void(FILE*)>> ptr2(fopen("D:/qt.txt", "r"),
        [](FILE* p)->void {
            cout << "call lambda release new open" << endl;
            fclose(p);
        });
    return 0;
```

```
}
/*
call lambda release new open
call lambda release int [100]
*/
```

四、C++11中引入的bind绑定器和function函数对象

1. bind1st 和 bind2nd 什么时候会用到

bind1st : operator() 的第一个形参变量绑定成一个确定的值

bind2nd : operator() 的第二个形参变量绑定成一个确定的值

C++11从Boost库中引入了 bind 绑定器和 function 函数对象机制

lambda表达式 底层依赖函数对象的机制实现的

使用 bind 例子

```
#include <iostream>
#include <memory>
#include <thread>
#include <functional>
#include <algorithm>
#include <vector>

using namespace std;

template<typename Container>
void showContainer(Container& con)
{
    //类型还没有实例化 编译器不知道后面套的是类型还是变量
    //加上typename 告诉编译器后面的是类型
    typename Container::iterator it = con.begin();
    for (; it != con.end(); ++it) {
        cout << *it << ' ';
    }
    cout << endl;
}

int main()
{
    vector<int>vec;
    srand(time(nullptr));
    for (int i = 0; i < 20; i++) {
        vec.push_back(rand() % 100 + 1);
    }

    sort(vec.begin(), vec.end());
    showContainer(vec);
```

```
    sort(vec.begin(), vec.end(), greater<int>());
    showContainer(vec);

    //把70按序插入
    //库里只有二元的，但是我们需要一元的
    //绑定器+二元函数对象 => 一元函数对象
    auto it1 = find_if(vec.begin(), vec.end(),
        bind1st(greater<int>(), 70));
    if (it1 != vec.end())
    {
        vec.insert(it1, 70);
    }
    else
        vec.push_back(70);
    showContainer(vec);

    return 0;
}
```

2. bind1st 和 bind2nd 的底层实现原理

封装了下，底层填充，变成二元。

```
#include <iostream>
#include <memory>
#include <thread>
#include <functional>
#include <algorithm>
#include <vector>

using namespace std;

template<typename Container>
void showContainer(Container& con)
{
    //类型还没有实例化 编译器不知道后面套的是类型还是变量
    //加上typename 告诉编译器后面的是类型
    typename Container::iterator it = con.begin();
    for (; it != con.end(); ++it) {
        cout << *it << ' ';
    }
    cout << endl;
}

template <typename Compare, typename T>
class _mybind1st
{
public:
    _mybind1st(Compare comp, T val)
        :_comp(comp), _val(val)
    { }
    bool operator()(const T& second)
    {
        return _comp(_val, second); //底层是二元函数对象
    }
```

```
private:
    Compare _comp;
    T _val;
};

template <typename Compare, typename T>
_mybind1st<Compare, T> mybind1st (Compare comp, const T& val)
{
    return _mybind1st<Compare, T>(comp, val);
}

template<typename Iterator, typename Compare>
Iterator my_find_if(Iterator first, Iterator last, Compare comp)
{
    for (; first != last; ++first)
    {
        if (comp(*first))
        {
            return first;
        }
    }
    return last;
}

int main()
{
    vector<int>vec;
    srand(time(nullptr));
    for (int i = 0; i < 20; i++) {
        vec.push_back(rand() % 100 + 1);
    }

    sort(vec.begin(), vec.end());
    showContainer(vec);

    sort(vec.begin(), vec.end(), greater<int>());
    showContainer(vec);

    //把70按序插入
    //库里只有二元的，但是我们需要一元的
    //绑定器+二元函数对象 => 一元函数对象
    auto it1 = my_find_if(vec.begin(), vec.end(),
        bind1st(greater<int>(), 70));
    if (it1 != vec.end())
    {
        vec.insert(it1, 70);
    }
    else
        vec.push_back(70);
    showContainer(vec);

    return 0;
}
```

3. function函数对象类型的应用实例

绑定器本身还是一个函数对象

function : 绑定器，函数对象，lambda表达式 它们只能使用在一条语句中

体验下 function 的用法

```
#include <iostream>
#include <functional>

using namespace std;

void hello1()
{
    cout << "hello" << endl;
}

void hello2(string str)
{
    cout << str << endl;
}

int sum(int a, int b)
{
    return a + b;
}

class Test
{
public: //调成员方法必须依赖一个对象 void (Test::*pfunc)(string)
    Test() { cout << "Test()" << endl; }
    void hello(string str) { cout << str << endl; }
};

int main()
{
    function<void()>func1 = hello1;
    func1();    //func1.operator() => hello1()
    function<void(string)>func2 = hello2;
    func2("hi~");   //func2.operator()(string str) => hello2(string str)
    function<int(int, int)>func3 = sum;
    cout << func3(10, 20) << endl;
    function<int(int, int)> func4 = [](int a, int b)->int {return a + b; };
    cout << func4(20, 30) << endl;

    function<void(Test*, string)> func5 = &Test::hello;
    func5(&Test(), "call Test::hello");

    return 0;
}
/*
hello
hi~
30
50
Test()
call Test::hello
*/
```

用 function + 哈希表 取代 switch

```
#include <iostream>
#include <functional>
#include <map>

using namespace std;

void doA() { cout << "A" << endl; }
void doB() { cout << "B" << endl; }
void doC() { cout << "C" << endl; }

int main()
{
    map<int, function<void()>>m;
    m.insert({1,  doA });
    m.insert({ 2, doB });
    m.insert({ 3, doC });
    while (1)
    {
        int x; cin >> x;
        auto it = m.find(x);
        if (it == m.end()) {
            cout << "无效选项" << endl;
        }
        else {
            it->second();
        }
    }

    return 0;
}
```

4. 模板的完全特例化和部分特例化

完全特例化：原模版的 T 已知。

```
template<typename T>
bool compare(T a, T b)
{
    cout << "template compare" << endl;
    return a > b;
}

//完全特例化 这样写的前提是compare是模板
template<>
bool compare<const char*>(const char* a, const char* b)
{
    cout << "compare<const char*>" << endl;
    return strcmp(a, b) > 0;
}
```

部分特例化：如针对指针类型特例化

```cpp
#include <iostream>

using namespace std;

template<typename T>
class Vector
{
public:
    Vector() { cout << "Vector() init" << endl; }
};
//对char* 类型提供的完全特例化版本
template<>
class Vector<char*>
{
public:
    Vector() { cout << "Vector<char*> init" << endl; }
};
//对指针类型提供的部分特例化版本
template<typename Ty>
class Vector<Ty*>
{
public:
    Vector() { cout << "Vector<Ty*> init" << endl; }
};
//对函数指针类型提供的部分特例化版本
template<typename R, typename A1, typename A2>
class Vector<R(*)(A1, A2)>
{
public:
    Vector() { cout << "call Vector<R(*)(A1, A2)>" << endl; }
};

int sum(int a, int b) { return a + b; }

int main()
{
    Vector<int>vec1;
    Vector<char*>vec2;
    Vector<int*>vec3;
    Vector<int(*)(int, int)>vec4; //函数指针

    typedef int(*PFUNC1)(int, int);
    PFUNC1 pfunc = sum;
    cout << pfunc(10, 20) << endl;

    typedef int PFUNC2(int, int);
    PFUNC2* pfunc2 = sum;
    cout << (*pfunc2)(10, 20) << endl;

    return 0;
}
/*
Vector() init
Vector<char*> init
Vector<Ty*> init
call Vector<R(*)(A1, A2)>
30
30
```

```
*/
```

观察函数指针类型

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

template<typename T>
void func(T a)
{
    cout << typeid(T).name() << endl;
}
int sum(int a, int b) { return a + b; }
int main()
{
    func(sum); //int (__cdecl*)(int,int) 函数指针

    return 0;
}
```

5. function的实现原理

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <functional>

using namespace std;

void hello(string str) { cout << str << endl; }

template<typename Fty>
class myfunction{};

template<typename R, typename A1>
class myfunction < R(A1) >
{
public:
    using PFUNC = R(*)(A1); //函数指针类型
    myfunction(PFUNC pfunc) :_pfunc(pfunc) { } //外面传进来的保存到自己的成员变量上
    R operator() (A1 arg)
    {
        return _pfunc(arg);
    }
private:
    PFUNC _pfunc;
};

template<typename R, typename A1, typename A2>
class myfunction < R(A1, A2) >
{
public:
    using PFUNC = R(*)(A1, A2); //函数指针类型
```

```
    myfunction(PFUNC pfunc) :_pfunc(pfunc) { } //外面传进来的保存到自己的成员变量上
    R operator() (A1 arg1, A2 arg2)
    {
        return _pfunc(arg1, arg2);
    }
private:
    PFUNC _pfunc;
};

int sum(int a, int b) { return a + b; }

int main()
{
    myfunction<void(string)> func1 = hello;
    func1("hello"); //func1.operator()("hello")
    myfunction<int(int, int)> func2 = sum;
    cout << func2(10, 20) << endl;
    return 0;
}
```

用模板语法

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <functional>

using namespace std;

void hello(string str) { cout << str << endl; }
int sum(int a, int b) { return a + b; }

template<typename Fty>
class myfunction{}; //要先定义模板

template<typename R, typename... A> //个数是任意的
class myfunction<R(A...)>
{
public:
    using PFUNC = R(*)(A...); //函数指针类型
    myfunction(PFUNC pfunc) :_pfunc(pfunc) { } //外面传进来的保存到自己的成员变量上
    R operator() (A... arg)
    {
        return _pfunc(arg...);
    }
private:
    PFUNC _pfunc;
};

int main()
{
    myfunction<void(string)> func1 = hello;
    func1("hello"); //func1.operator()("hello")
    myfunction<int(int, int)> func2 = sum;
    cout << func2(10, 20) << endl;
    return 0;
}
```

6. bind 和 function 实现线程池

C++11 bind绑定器 => 返回的结果还是一个函数对象

bind 是函数模板，可以自动推演模板类型参数

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <functional>

using namespace std;

void hello(string str) { cout << str << endl; }
int sum(int a, int b) { return a + b; }
class Test
{
public:
    int sum(int a, int b) { return a + b; }
};

int main()
{
    //bind 是函数模板，可以自动推演模板类型参数
    bind(hello, "hello")();
    cout << bind(sum, 10, 20)() << endl;
    cout << bind(&Test::sum, Test(), 20, 30)() << endl;

    //参数占位符 绑定器出了语句无法继续使用
    bind(hello, placeholders::_1)("测试占位符");
    cout << bind(sum, placeholders::_1, placeholders::_2)(10, 20) << endl;

    //此处把bind返回的绑定器复用起来了
    function<void(string)> func1 = bind(hello, placeholders::_1);
    func1("hello t1");
    func1("hello t2");

    return 0;
}
/*
hello
30
50
测试占位符
30
hello t1
hello t2
*/
```

线程池

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <functional>
```

```cpp
#include <vector>
#include <thread>

using namespace std;

//线程类
class Thread
{
public:
    Thread(function<void()>func) :_func(func) {}
    thread start()
    {
        thread t(_func);
        return t;
    }
private:
    function<void()>_func;
};

//线程池类
class ThreadPool
{
public:
    ThreadPool(){}
    ~ThreadPool()
    {
        //释放Thread对象占用的堆资源
        for (int i = 0; i < _pool.size(); ++i)
        {
            delete _pool[i];
        }
    }
    //开启线程池
    void startPool(int size)
    {
        for (int i = 0; i < size; ++i)
        {
            _pool.push_back(
                new Thread(bind(&ThreadPool::runInThread,this,i))); //绑定给当前对
象
        }
        for (int i = 0; i < size; ++i)
        {
            _handler.push_back(_pool[i]->start());
        }
        for (thread& t : _handler)
        {
            t.join();
        }
    }

private:
    vector<Thread*> _pool;
    vector<thread> _handler;
    //runInThread充当线程函数
    void runInThread(int id)
    {
        cout << "call runInThread id:" << id << endl;
```

```
    }
};

int main()
{
    ThreadPool pool;
    pool.startPool(10);
    return 0;
}
/*
call runInThread id:call runInThread id:1call runInThread id:5
call runInThread id:call runInThread id:9
call runInThread id:6
call runInThread id:4

call runInThread id:3
8
call runInThread id:7
0
call runInThread id:2
*/
```

7. lambda 表达式的实现原理

函数对象的升级版 =》lambda 表达式

函数对象的缺点：需要定义出一个类出来等

lambda表达式的语法：

`[捕获外部变量](形参列表)->返回值{操作代码};`

```
auto f = []()->void {cout << "hi"; };
f();

//不需要返回值的话可以将其省略
auto f = [](){cout << "hi"<< endl; };
```

类似于这个

```
template<typename T = void>
class TestLambda
{
public:
    TestLambda() {}
    void operator()() //第一个括号运算符重载 第二个括号形参列表
    {
        cout << "hi" << endl;
    }
};
```

`[]`：表示不捕获外部的变量

`[=]`：表示以传值的方式捕获外部的所有变量

`[&]`：以传引用的方式捕获外部的所有变量

`[this]`：捕获外部的 this 指针

`[=,&a]`：以传值的方式捕获外部的所有变量，但是 a 变量以传引用的方式捕获

`[a,b]`：以值传递的方式捕获外部变量a 和 b

`[a,&b]`：a 以值传递捕获，b以传引用的方式捕获

```
int a = 10, b = 20;
auto f3 = [=]() {  //error 常方法里试图改变变量的值
    int tmp = a;
    a = b;
    b = tmp;
};
```

将 const 加上 mutable，通过了语法编译，但是并没有交换，将地址的变量输出

```
int a = 10, b = 20;
printf("%p %p\n", &a, &b);
auto f3 = [a, b]() mutable
{
    int tmp = a;
    a = b;
    b = tmp;
    printf("%p %p\n", &a, &b);
};
f3();
printf("%d %d\n", a, b);
/*
00CFFCE4 00CFFCD8
00CFFCC8 00CFFCCC
10 20
*/
```

改为引用传递

```
auto f3 = [&a, &b]()
{
    int tmp = a;
    a = b;
    b = tmp;
};
f3();
```

8. lambda 表达式的应用实践

既然 lambda 表达式只能使用在语句中，如果想跨语句使用之前定义好的 lambda 表达式怎么办？用什么类型来表示 lambda 表达式?

用 function 类型来表示函数对象的类型

lambda表达式 =》函数对象

换种方式写 if-else

```
map<int, function<int(int, int)>> caculateMap;
caculateMap[1] = [](int a, int b)->int {return a + b; };
caculateMap[2] = [](int a, int b)->int {return a - b; };
caculateMap[3] = [](int a, int b)->int {return a * b; };
caculateMap[4] = [](int a, int b)->int {return a / b; };
```

智能指针自定义删除器

```
unique_ptr<FILE, function<void(FILE*)>>
    ptr1(fopen("data.txt", "w"), [](FILE* pf) {fclose(pf); });
```

多元组的优先级队列

```
class Data
{
public:
    Data(int val1 = 10, int val2 = 10) :ma(val1), mb(val2) {}
    //bool operator>(const Data& data) const { return ma > data.ma; }  //方法一
    //bool operator<(const Data& data) const { return ma < data.ma; }
    int ma;
    int mb;
private:
};

//main
//优先级队列
using FUNC = function<bool(Data&, Data&)>; //方法二
priority_queue<Data, vector<Data>, FUNC>
    queue([](Data& d1, Data& d2)->bool
          {
              return d1.ma > d2.ma;
          });
queue.push({ 10, 20 });
queue.push({ 15, 25 });
```

五、C++11知识点汇总

1. C++11常用知识点整理总结

一、关键字和语法

auto：可以根据右值，推导出右值的类型，然后左边变量的类型也就已知了。

nullptr：指针专用，能够和整数（NULL）区别

foreach：可以遍历数组、容器等。底层通过指针或迭代器实现的

右值引用：move移动语义函数 和 forward 类型完美转发函数

模板的一个新特性：`typename... A` 表示可变参

二、绑定器和函数对象

function：函数对象

bind：绑定器

lambda 表达式

三、智能指针

`shared_ptr` 和 `weak_ptr`

四、容器

STL

五、C++语言级别支持的多线程编程

2. 通过thread类编写C++多线程程序

语言级别支持多线程，说明代码可以跨平台。

thread / mutex / condition_variable

lock_guard / unique_lock

atomic 原子类型

sleep_for

```
#include <iostream>
#include <thread>

using namespace std;

void threadHandle1()
{
    //让子线程睡眠两秒
    std::this_thread::sleep_for(std::chrono::seconds(2));
    cout << "hello thread1" << endl;
}

int main()
{
    //创建了一个线程对象 传入了一个线程函数
    std::thread t1(threadHandle1);
    //主线程等待子线程结束,主线程继续往下运行
    //t1.join();

    //或者 把子线程设置为分离线程，主线程结束，整个进程结束，所有子线程都自动结束了
    t1.detach();   //这个看不到子线程输出信息了

    cout << "main" << endl;

    return 0;
}
```

再贴个例子

```
#include <iostream>
#include <thread>

using namespace std;

void threadHandle1(int t)
{
    //让子线程睡眠t秒
    std::this_thread::sleep_for(std::chrono::seconds(t));
    cout << "hello thread1" << endl;
}

void threadHandle2(int t)
{
    //让子线程睡眠t秒
    std::this_thread::sleep_for(std::chrono::seconds(t));
    cout << "hello thread2" << endl;
}

int main()
{
    //创建了一个线程对象 传入了一个线程函数
    std::thread t1(threadHandle1, 2);
    std::thread t2(threadHandle2, 3);
    //主线程等待子线程结束,主线程继续往下运行
    t1.join();
    t2.join();
    cout << "main" << endl;

    return 0;
}
```

3. 线程间互斥-mutex互斥锁和lock_guard

```
#include <iostream>
#include <thread>
#include <mutex>
#include <list>

using namespace std;

int ticketCount = 100;
std::mutex mtx; //全局的一把互斥锁

void sellTicket(int index)
{
    while (ticketCount > 0)
    {
        { //定义一个作用域
            lock_guard<std::mutex>lock(mtx); //出了作用域就自动释放 类似智能指针
            if (ticketCount > 0) //可能出现最后一个都进来了
            {
```

```
                cout << "窗口" << index << "卖出第:" << ticketCount << endl;
                ticketCount--;
            }
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

int main()
{
    list<std::thread>tlist;
    for (int i = 1; i <= 3; i++) {
        tlist.push_back(std::thread(sellTicket, i));
    }
    for (auto& i : tlist)
    {
        i.join();
    }
    cout << "结束" << endl;
    return 0;
}
```

4. 线程间同步通信-生产者消费者模型

1. 线程间互斥 （竞态条件 =》临界区代码段 =》保证原子操作 =》互斥锁mutex）
2. 线程间的同步通信

两者生产一个就消费一个，结束后互相通知对方。

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue> //STL所有容器都不是线程安全

using namespace std;

std::mutex mtx; //定义互斥锁, 线程间互斥
std::condition_variable cv; //定义条件变量, 线程间通信

class Queue
{
public:
    void put(int val)
    {
        //lock_guard<std::mutex>guard(mtx);//error scoped_ptr 左值拷贝构造和赋值都
delete了
        unique_lock<std::mutex> lck(mtx); //防止这把锁被释放了
        while (!que.empty())
        {
            cv.wait(lck); //线程进入等待, 并把mtx锁释放
        }
        que.push(val);
        cv.notify_all(); //其他线程得到该通知,从等待变为阻塞,获取锁后继续执行
        cout << "生产者 生产:" << val << " 号物品" << endl;
```

```
    }
    int get()
    {
        //lock_guard<std::mutex>guard(mtx);
        unique_lock<std::mutex>lck(mtx);
        while (que.empty())
        {
            cv.wait(lck);
        }
        int val = que.front();
        que.pop();
        cv.notify_all();
        cout << "消费者 消费:" << val << "号物品" << endl;
        return val;
    }
private:
    queue<int> que;
};

void producer(Queue* que)
{
    for (int i = 0; i < 10; i++)
    {
        que->put(i);
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void consumer(Queue* que)
{
    for (int i = 0; i < 10; i++)
    {
        que->get();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

int main()
{
    Queue que;

    std::thread t1(producer, &que);
    std::thread t2(consumer, &que);

    t1.join();
    t2.join();

    return 0;
}
```

5. 再谈lock_guard和unique_lock

lock_guard 和 unique_lock

unique_lock：不仅可以使用在简单的临界区代码段的互斥操作中，还能用在函数调用过程中

lock_guard：不可能用在函数参数传递或者返回过程中，只能用在简单的临界区代码段的互斥操作中

condition_variable wait 和 notify_all 方法

notify_all：通知在 cv 上等待的线程，条件成立了，起来干活。其它 cv 上等待的线程收到通知，从等待态到阻塞态。

6. 基于CAS操作的atomic原子类型

互斥锁比较重，临界区代码做的事情稍稍复杂时用。

无锁队列

```
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <atomic> //包含了很多原子类型
#include <list>

using namespace std;

//volatile 防止多线程对共享变量进行缓存
volatile std::atomic_bool isReady = true;
volatile std::atomic_int ccount = 0;
//int ccount = 0;

void task()
{
    while (!isReady)
    {
        std::this_thread::yield(); //线程出让当前的cpu时间片,等待下一次调度
    }
    for (int i = 0; i < 1000; i++)
        ccount++;
}

int main()
{
    list<std::thread> tlist;
    for (int i = 0; i < 10; i++) {
        tlist.push_back(std::thread(task));
    }
    for (auto& t : tlist)
    {
        t.join();
    }
    cout << ccount << endl;
    return 0;
}
```

六、设计模式

1.单例模式代码设计

单例模式：一个类不管创建多少次对象，永远只能得到该类型一个对象的实例。

常用的：日志模块、数据库模块

饿汉式单例模式：还没有获取实例对象，实例对象就已经产生了

懒汉式单例模式：唯一的实例对象，直到第一次获取它的时候，才产生

饿汉式一定是线程安全的；但是会延长软件的启动时间

```
#include <iostream>

class Singleton
{
public:
    static Singleton* getInstance() //#3 获取类的唯一实例对象的接口方法
    {
        return &instance;
    }
private:
    static Singleton instance; //#2 定义一个唯一的类的实例对象
    Singleton() //#1 构造函数私有化
    {

    }
    Singleton(const Singleton&) = delete; //#4 禁止深拷贝、运算符重载
    Singleton& operator=(const Singleton&) = delete;
};
Singleton Singleton::instance;

int main()
{
    Singleton* p1 = Singleton::getInstance();
    Singleton* p2 = Singleton::getInstance();
    Singleton* p3 = Singleton::getInstance();
    return 0;
}
```

懒汉式模式：变成指针。

```
class Singleton
{
public:
    static Singleton* getInstance() //#3 获取类的唯一实例对象的接口方法
    {
        if (nullptr == instance)
        {
            instance = new Singleton();
        }
        return instance;
    }
private:
    static Singleton *instance; //#2 定义一个唯一的类的实例对象
    Singleton() //#1 构造函数私有化
```

```
    {

    }
    Singleton(const Singleton&) = delete; //#4 禁止深拷贝、运算符重载
    Singleton& operator=(const Singleton&) = delete;
};
Singleton *Singleton::instance = nullptr;
```

2. 线程安全的懒汉单例模式

可重入：没有执行完又被调用一次

开辟内存 => 构造对象、赋值（这两个不保证执行顺序）

有可能出现线程 1 运行后还没来得及赋值线程 2 也进去了

```
class Singleton
{
public:
    static Singleton* getInstance() //#3 获取类的唯一实例对象的接口方法
    {
        if (nullptr == instance)
        {
            std::lock_guard<std::mutex> guard(mtx);
            if (nullptr == instance)
                instance = new Singleton();
        }
        return instance;
    }
private:
    static Singleton* volatile instance; //#2 定义一个唯一的类的实例对象
    Singleton() //#1 构造函数私有化
    {

    }
    Singleton(const Singleton&) = delete; //#4 禁止深拷贝、运算符重载
    Singleton& operator=(const Singleton&) = delete;
};
Singleton* volatile Singleton::instance = nullptr;
```

也可以用 static 实现

```
class Singleton
{
public:
    static Singleton* getInstance()
    {
        // 函数静态局部变量的初始化,在汇编上已经自动添加线程互斥指令了
        static Singleton instance; //运行到这里才会初始化
        return &instance;
    }
private:
    Singleton()
    {

```

```
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};
```

3. 简单工厂和工厂方法

简单工厂，一个工厂把所有的产品都造，同时也不符合“开闭”原则

```
#include <iostream>
#include <memory>

/*
简单工厂 Simple Factory
工厂方法 Factory Method
抽象工厂 Abstract Factory

工厂模式：主要是封装了对象的创建
*/

class Car
{
public:
    Car(std::string name) : _name(name){}
    virtual void show() = 0;
protected:
    std::string _name;
};

class Bmw : public Car
{
public:
    Bmw(std::string name) : Car(name){}
    void show()
    {
        std::cout << "获取了一辆宝马 " << _name << std::endl;
    }
};

class Audi : public Car
{
public:
    Audi(std::string name) : Car(name) {}
    void show()
    {
        std::cout << "获取了一辆奥迪 " << _name << std::endl;
    }
};

enum CarType
{
    BMW, AUDI
};

class SimpleFactory
```

```
{
public:
    Car* createCar(CarType ct)
    {
        switch (ct)
        {
        case BMW:
            return new Bmw("x1");
            break;
        case AUDI:
            return new Audi("y1");
            break;
        default:
            std::cerr << "传入工厂的参数不正确 " << ct << std::endl;
            break;
        }
    }
};

int main()
{
    std::unique_ptr<SimpleFactory> factory(new SimpleFactory());
    std::unique_ptr<Car>p1(factory->createCar(BMW));
    std::unique_ptr<Car>p2 (factory->createCar(AUDI));
    p1->show();
    p2->show();

    return 0;
}
```

可以通过一个基类向外拓展。实现了修改关闭、拓展打开

```
#include <iostream>
#include <memory>

/*
简单工厂 Simple Factory
工厂方法 Factory Method
抽象工厂 Abstract Factory

工厂模式：主要是封装了对象的创建
*/

class Car
{
public:
    Car(std::string name) : _name(name){}
    virtual void show() = 0;
protected:
    std::string _name;
};

class Bmw : public Car
{
public:
    Bmw(std::string name) : Car(name){}
    void show()
```

```
    {
        std::cout << "获取了一辆宝马 " << _name << std::endl;
    }
};

class Audi : public Car
{
public:
    Audi(std::string name) : Car(name) {}
    void show()
    {
        std::cout << "获取了一辆奥迪 " << _name << std::endl;
    }
};

class Factory
{
public:
    virtual Car* createCar(std::string name) = 0;
};

class BMWFactory : public Factory
{
public:
    Car* createCar(std::string name)
    {
        return new Bmw(name);
    }
};

class AudiFactory : public Factory
{
public:
    Car* createCar(std::string name)
    {
        return new Audi(name);
    }
};

int main()
{
    std::unique_ptr<Factory> bmwFactory(new BMWFactory());
    std::unique_ptr<Factory> audiFactory(new AudiFactory());
    std::unique_ptr<Car>p1(bmwFactory->createCar("x1"));
    std::unique_ptr<Car>p2(audiFactory->createCar("y1"));
    p1->show();
    p2->show();

    return 0;
}
```

4. 抽象工厂

现在考虑生产一类产品

缺点：重写接口很麻烦

```
#include <iostream>
#include <memory>

/*
简单工厂 Simple Factory
优：客户不用自己负责new对象
缺：接口函数不闭合，不能对修改关闭

工厂方法 Factory Method
优：提供了一个纯虚函数（创建产品），定义派生类（具体产品的工厂）负责创建对应的产品，可以做到不同的产品在不同的工厂里面创建，能够对现有工厂以及产品的修改关闭。
缺：很多产品是有关联关系的，属于一个产品簇，不应该放在不同的工厂里去创建，且工厂类太多，不好维护

抽象工厂 Abstract Factory
把有关联关系的，属于一个产品簇的所有产品创建的接口函数放在一个抽象工厂里面，派生类（具体产品的工厂）应该负责创建该产品簇里面所有的产品。

工厂模式：主要是封装了对象的创建
*/

//系列产品1
class Car
{
public:
    Car(std::string name) : _name(name){}
    virtual void show() = 0;
protected:
    std::string _name;
};

class Bmw : public Car
{
public:
    Bmw(std::string name) : Car(name){}
    void show()
    {
        std::cout << "获取了一辆宝马 " << _name << std::endl;
    }
};

class Audi : public Car
{
public:
    Audi(std::string name) : Car(name) {}
    void show()
    {
        std::cout << "获取了一辆奥迪 " << _name << std::endl;
    }
};

//系列产品2
class Light
{
public:
    virtual void show() = 0;
```

```
};

class BmwLight : public Light
{
public:
    void show() { std::cout << "Bmw light" << std::endl; }
};

class AudiLight : public Light
{
public:
    void show() { std::cout << "Audi light" << std::endl; }
};

//抽象成抽象工厂 => 对有一组关联关系的产品簇提供产品对象的统一创建
class AbstractFactory
{
public:
    virtual Car* createCar(std::string name) = 0; //工厂方法 创建汽车
    virtual Light* createCarLight() = 0; //创建车灯
};

class BMWFactory : public AbstractFactory
{
public:
    Car* createCar(std::string name)
    {
        return new Bmw(name);
    }
    Light* createCarLight()
    {
        return new BmwLight();
    }
};

class AudiFactory : public AbstractFactory
{
public:
    Car* createCar(std::string name)
    {
        return new Audi(name);
    }
    Light* createCarLight()
    {
        return new AudiLight();
    }
};

int main()
{
    std::unique_ptr<AbstractFactory> bmwFactory(new BMWFactory());
    std::unique_ptr<AbstractFactory> audiFactory(new AudiFactory());
    std::unique_ptr<Car>p1(bmwFactory->createCar("x1"));
    std::unique_ptr<Car>p2(audiFactory->createCar("y1"));
    std::unique_ptr<Light>p3(bmwFactory->createCarLight());
    std::unique_ptr<Light>p4(audiFactory->createCarLight());
    p1->show();
    p2->show();
```

```
    p3->show();
    p4->show();

    return 0;
}
/*
获取了一辆宝马 x1
获取了一辆奥迪 y1
Bmw light
Audi light
*/
```

5. 代理模式

```
#include <iostream>
#include <memory>

using namespace std;

/*
代理Proxy模式：通过代理类，来控制实际对象的访问权限
客户      助理Proxy      老板 委托类
*/
class VideoSite
{
public:
    virtual void freeMovie() = 0; //免费电影
    virtual void vipMovie() = 0; //vip电影
    virtual void ticketMovie() = 0; //券电影
};
class FixBugVideoSite : public VideoSite  //委托类
{
public:
    virtual void freeMovie()  //免费电影
    {
        cout << "免费电影" << endl;
    }
    virtual void vipMovie()  //vip电影
    {
        cout << "vip电影" << endl;
    }
    virtual void ticketMovie()  //券电影
    {
        cout << "券电影" << endl;
    }
};

class FreeVideoSitProxy : public VideoSite
{
public:
    FreeVideoSitProxy(){ pVideo = new FixBugVideoSite(); }
    ~FreeVideoSitProxy() { delete pVideo; }
    virtual void freeMovie() //免费电影
    {
        pVideo->freeMovie(); //通过代理对象的freeMovie，来访问真正委托类对象的freeMovie
```

```
    }
    virtual void vipMovie()//vip电影
    {
        cout << "请升级vip" << endl;
    }
    virtual void ticketMovie() //券电影
    {
        cout << "请充值" << endl;
    }
private:
    VideoSite* pVideo;
};

class VipVideoSitProxy : public VideoSite
{
public:
    VipVideoSitProxy() { pVideo = new FixBugVideoSite(); }
    ~VipVideoSitProxy() { delete pVideo; }
    virtual void freeMovie() //免费电影
    {
        pVideo->freeMovie(); //通过代理对象的freeMovie，来访问真正委托类对象的freeMovie
    }
    virtual void vipMovie() //vip电影
    {
        pVideo->vipMovie();
    }
    virtual void ticketMovie()  //券电影
    {
        cout << "请充值" << endl;
    }
private:
    VideoSite* pVideo;
};

void watchMovie(unique_ptr<VideoSite>& ptr)
{
    ptr->freeMovie();
    ptr->vipMovie();
    ptr->ticketMovie();
}

int main()
{
    unique_ptr<VideoSite> p1(new FreeVideoSitProxy());
    unique_ptr<VideoSite> p2(new VipVideoSitProxy());

    watchMovie(p1);
    watchMovie(p2);

    return 0;
}
/*
免费电影
请升级vip
请充值
免费电影
vip电影
请充值
```

```
*/
```

6. 装饰器模式

```
#include <iostream>
#include <memory>

using namespace std;

/*
装饰器模式
*/

class Car  //抽象基类
{
public:
    virtual void show() = 0;
};
//三个实体的汽车类
class Bmw : public Car
{
public:
    void show()
    {
        cout << "宝马,配置有：基类配置";
    }
};
class Audi : public Car
{
public:
    void show()
    {
        cout << "奥迪,配置有：基类配置" ;
    }
};
class Benz : public Car
{
public:
    void show()
    {
        cout << "奔驰,配置有：基类配置" ;
    }
};

//装饰器 定速巡航
class ConcreteDecorator01 : public Car
{
public:
    ConcreteDecorator01(Car* p) :pCar(p) {}
    void show()
    {
        pCar->show();
        cout << ",定速巡航" ;
    }
private:
```

```
    Car* pCar;
};

class ConcreteDecorator02 : public Car
{
public:
    ConcreteDecorator02(Car* p) :pCar(p) {}
    void show()
    {
        pCar->show();
        cout << ",自动刹车";
    }
private:
    Car* pCar;
};

class ConcreteDecorator03 : public Car
{
public:
    ConcreteDecorator03(Car* p) :pCar(p) {}
    void show()
    {
        pCar->show();
        cout << ",车道偏离" ;
    }
private:
    Car* pCar;
};


int main()
{
    Car* p1 = new ConcreteDecorator01(new Bmw());
    p1 = new ConcreteDecorator02(p1);
    p1 = new ConcreteDecorator03(p1);
    Car* p2 = new ConcreteDecorator02(new Audi());
    Car* p3 = new ConcreteDecorator03(new Benz());

    p1->show();
    cout << endl;
    p2->show();
    cout << endl;
    p3->show();

    return 0;
}
/*
宝马,配置有：基类配置,定速巡航,自动刹车,车道偏离
奥迪,配置有：基类配置,自动刹车
奔驰,配置有：基类配置,车道偏离
*/
```

7. 适配器模式

```
#include <iostream>
```

```cpp
#include <memory>

using namespace std;

/*
适配器模式：让不兼容的接口可以在一起工作
电脑 =》 投影到 =》 投影仪
*/
class VGA //VGA接口类
{
public:
    virtual void play() = 0;
};

//进了一批新的投影仪，但是新的投影仪只支持HDMI接口
class HDMI
{
public:
    virtual void play() = 0;
};

//TV01 表示支持VGA接口的投影仪
class TV01 : public VGA
{
public:
    void play()
    {
        cout << "通过VGA接口连接投影仪，进行视频播放" << endl;
    }
};

//TV02 表示支持HDMI接口的投影仪
class TV02 : public HDMI
{
public:
    void play()
    {
        cout << "通过HDMI接口连接投影仪，进行视频播放" << endl;
    }
};

//实现一个电脑类(只支持VGA接口)
class Computer
{
public:
    //由于电脑只支持VGA接口，所以该方法的参数只支持VGA接口的指针/引用
    void playVideo(VGA* pVGA)
    {
        pVGA->play();
    }
};

/*
方法1：换一个支持HDMI接口的电脑，代码重构
方法2：买一个转换头（适配器），把VGA信号转为HDMI信号，添加适配器类
*/
class VGAToHDMIAdpter : public VGA
{
```

```
public:
    VGAToHDMIAdpter(HDMI* p) :pHdmi(p){}
    void play() //转换头
    {
        pHdmi->play();
    }
private:
    HDMI* pHdmi;
};

int main()
{
    Computer computer;
    computer.playVideo(new TV01());
    computer.playVideo(new VGAToHDMIAdpter(new TV02()));
    return 0;
}
```

8. 观察者模式

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <list>

using namespace std;

/*
行为形模式：主要关注的是对象之间的通信
观察者-监听者模式（发布-订阅模式）设计模式：主要关注的是对象的一对多的关系，
也就是多个对象都依赖一个对象，当该对象的状态发生改变时，其它对象都能接收到
相应的通知

一组数据（数据对象） => 通过这一组数据 => 曲线图/柱状图/圆饼图
当数据对象改变时，对象1、对象2、对象3应该及时地收到相应的通知
*/

//观察者抽象类
class Observer
{
public:
    //处理消息的接口
    virtual void handle(int msgid) = 0;
};

//第一个观察者实例 1 2
class Observer1 : public Observer
{
public:
    void handle(int msgid)
    {
        switch(msgid)
        {
        case 1:
            cout << "Observer1 recv 1 msg!" << endl;
```

```
            break;
        case 2:
            cout << "Observer1 recv 2 msg!" << endl;
            break;
        default:
            cout << "Observer1 recv unknow msg!" << endl;
            break;
        }
    }
};

//第二个观察者实例 2
class Observer2 : public Observer
{
public:
    void handle(int msgid)
    {
        switch (msgid)
        {
        case 2:
            cout << "Observer2 recv 2 msg!" << endl;
            break;
        default:
            cout << "Observer2 recv unknow msg!" << endl;
            break;
        }
    }
};

//第三个观察者实例  1 3
class Observer3 : public Observer
{
public:
    void handle(int msgid)
    {
        switch (msgid)
        {
        case 1:
            cout << "Observer3 recv 1 msg!" << endl;
            break;
        case 3:
            cout << "Observer3 recv 3 msg!" << endl;
            break;
        default:
            cout << "Observer3 recv unknow msg!" << endl;
            break;
        }
    }
};

class Subject
{
public:
    //给主题增加观察者对象
    void addObserver(Observer* obser, int msgid) //观察者和它感兴趣的id
    {
        _subMap[msgid].push_back(obser);
    }
```

```
    //主题检测发生改变，通知相应的观察者对象处理事件
    void dispatch(int msgid)
    {
        auto it = _subMap.find(msgid);
        if (it != _subMap.end())
        {
            for (auto tmp : it->second)
            {
                tmp->handle(msgid);
            }
        }
    }
private:
    unordered_map<int, list<Observer*>> _subMap;
};

int main()
{
    Subject subject;
    Observer* p1 = new Observer1();
    Observer* p2 = new Observer2();
    Observer* p3 = new Observer3();

    subject.addObserver(p1, 1);
    subject.addObserver(p1, 2);
    subject.addObserver(p2, 2);
    subject.addObserver(p3, 1);
    subject.addObserver(p3, 3);

    int msgid = 0;
    for (;;)
    {
        cout << "输入消息id：";
        cin >> msgid;
        if (msgid == -1)
            break;
        subject.dispatch(msgid);
    }
    return 0;
}
/*
输入消息id：1
Observer1 recv 1 msg!
Observer3 recv 1 msg!
输入消息id：2
Observer1 recv 2 msg!
Observer2 recv 2 msg!
输入消息id：3
Observer3 recv 3 msg!
输入消息id：4
输入消息id：-1
*/
```

七、C++代码应用实践

1. dfs路径

没有难度

2. bfs最短路径

bfs 记录路径可以新开一个二维数组，每个地方记录前驱的下标

3. 大数的加减法

模拟高精度

4. 海量数据查重问题解决方案汇总

1. 哈希表
2. 分治思想
3. Bloom Filter：布隆过滤器
4. 字符串类型：Trie树

```
/*
#1 哈希表
*/

/*
* # 2
有一个文件 50亿个整数，内存限制400M，请你找出文件中重复的数和次数
50亿 5G*4=20G  *2(哈希表地址索引) = 40G
分治：大文件划分为小文件，把结果写入到一个存储重复元素的文件当中

大文件划分为小文件的个数：40G/400M = 120个小文件
data0.txt
data1.txt
...
data126.txt

遍历大文件的元素，把每一个元素根据哈希映射函数，放到对应序号的小文件当中
data % 127 = file_index
*/

/* #3
a,b 两个文件，里面都有10亿个整数，内存限制400M，求出a，b两个文件中
重复的元素有哪些？
10亿 -》1G*4
把a和b两个大文件划分成个数相等的一系列（27个）小文件（分治）
a0.txt a1.txt a2.txt ... a26.txt
b0.txt b1.txt b2.txt ... b26.txt
*/
```

5. 海量数据求top k问题解决方案汇总

1. 求最大的/最小的前 k 个元素
2. 求最大的/最小的第 k 个元素

解法一：大根堆/小根堆 如求前十大元素，维持一个大小为 10 的小根堆。

解法二：快排分割函数

有一个大文件，内存限制200M，求最大的前10个。

分治思想。小文件，每个小文件前 10 个，合并起来就是结果了。

6. 海量数据查重和top k综合应用

重复次数最多的前 10 个。

哈希表后堆

八、C++11容器emplace方法原理剖析

对 push/insert 更新成了 emplace。传入参数时直接在容器的底层构造。

vector 用下

```
#include <iostream>
#include <vector>

using namespace std;

class Test
{
public:
    Test(int a) { cout << "Test(int)" << endl; }
    Test(int a, int b) { cout << "Test(int, int)" << endl; }
    Test(const Test&) { cout << "Test(const Test&)" << endl; }
    Test(Test&&) { cout << "Test(Test&&)" << endl; }
    ~Test() { cout << "~Test()" << endl; }
};

int main()
{
    vector<Test> vec;
    vec.reserve(100);
    vec.push_back(10);
    cout << "--------" << endl;
    vec.emplace_back(10); //没有拷贝构造，直接在vec底层构造
    cout << "--------" << endl;
    return 0;
}
```