

C++第八天课堂笔记

一、回顾知识点

1.1 模板方面

模板机制：

编译器对最初的模板进行编译一次，对具体化模板再一次编译
类模板的声明与实现不能分离

函数模板： 函数参数模板化，即参数化

类模板： 对类成员变量进行模板化，类模板在整个类的内部都可以使用，类模板派生类时，必须具体化类模板。类模板可以派生模板，具体化父类模板应该使用派生模板的类型。

```
template<typename T>
class A{};

class B: public A<int>{};

template<typename U>
class C: public A<U>{}; // 使用派生类模板类型具体化父类模板的类型
```

类模板中遇到友元函数：

- 1) 内部声明并实现友元函数，类模板的类型可以用于友元函数（属于全局函数）
- 2) 外部友元函数模板，类模板声明友元函数时，必须声明<>空泛型，表示外部友元函数是模型函数
类模板：`friend void show<>(A<T> &a);`

外部的全局函数：

```
template<typename U>
void show(A<U> &a){}
```

条件：友元函数必须在类模板声明，且类模板也要提前声明。

- 3) 声明友元函数函数模板

类模板：

```
template<typename U> friend void show(A<U> &a);
```

外部全局函数：

```
template<typename U>
void show(A<U> &a){}
```

类模板的成员函数在外部实现时： 指定函数为模板函数，模板的类型必须是类模板的类型（泛型）。

```
template<typename T>
A<T>::A(参数列表){}

template<typename T>
void A<T>::show(){

}
```

1.2 类型转换函数

静态转换： `static_cast<目标类型>(转换的变量或指针或引用)`

支持：

- 1) 基本数据类型 安全的
- 2) 上行转换 安全的
- 3) 下行转换 不安全的

不支持：

- 1) 基本数据类型的指针
- 2) 不相关的类之间的转换

动态转换： `dynamic_cast<> ()`

支持：

- 1) 上行转换 安全的

不支持：

- 1) 基本数据类型
- 2) 基本数据类型的指针
- 3) 不相关的类之间的转换
- 4) 下行转换 不安全

重新解释转换: `reinterpret_cast<>()`

支持：

- 1) 上行转换 安全的
- 2) 基本数据类型的指针 不安全
- 3) 不相关的类之间的转换 不安全
- 4) 下行转换 不安全

不支持：

- 1) 基本数据类型

常量转换： `const_cast<>()`

条件： 非指针或引用不能去掉 `const`

可以将 `const`变量引用或指针 转化为 变量引用或指针

可以将 变量引用或指针 转化为 `const`变量引用或指针

二、C++异常

2.1 异常基本概念

C语言中处理异常的方式： 不会中断程序的执行

- 1) 返回常量值， 如 0成功， 1失败
- 2) 宏 `errno` （类似于全局整数类型的变量）， 记录程序出现异常的标识。
`perror()` 进行打印错误信息
- 3) `NULL` 指针的`NULL`表示，表示异常的情况
`FILE *f = fopen("不存在的文件", "r");`

C++处理异常的方式： 抛出异常、 捕获异常

2.2 c++异常语法

2.2.1 throw抛出异常

语法： `throw` 异常数据

异常数据包含基本数据类型、 类、 `struct`等

【注意】如果抛出的异常没有处理时，则会中断程序的执行

如1：抛出基本类型

```
int div(int a, int b)
{
    if (b == 0)
    {
        throw "除数不能为0";
    }
    return a / b;
}
```

2.2.2 处理异常的语法

语法格式：

```
try{
    执行可能存在异常的语句块；
}catch(异常数据的类型1 变量){ // 捕获相应数据类型的异常信息
    // 处理异常的语句；
}catch(异常数据的类型2 变量){
    // 处理异常的语句；
}
...
catch(...){ // 必须放在最后
    // 以上异常类型都没有匹配成功时，执行的语句的。
}
```

如2：捕获基本数据类型的异常

```

#include <iostream>

using namespace std;

int div(int a, int b)
{
    if (b == 0)
    {
        throw "除数不能为0";
    }
    return a / b;
}

int main(int argc, char const *argv[])
{
    cout << "-----aaaaa-----" << endl;
    int ret = 0;
    try
    {
        ret = div(20, 0);
    }
    catch (const char *error)
    {
        cout << "异常: " << error << endl;
    }

    cout << "ret=" << ret << endl;
    return 0;
}

```

【小结】如果异常被捕获之后，程序则不会中断。捕获异常信息的时候，一定考虑异常信息的数据类型。

2.2.3 throw的限制与严格类型异常匹配

在使用throw抛出异常信息时，受到函数声明处的throw()声明的可抛出异常类型的限制。

如1：函数内可以抛出任何异常

```

#include <iostream>
using namespace std;

class A
{
public:
    int n;
    A(int n) : n(n) {}
};

void show(int x)
{
    if (x == 1)
        throw 0;
    else if (x == 2)
        throw 'a';
    else if (x == 3)
        throw "abc";
}

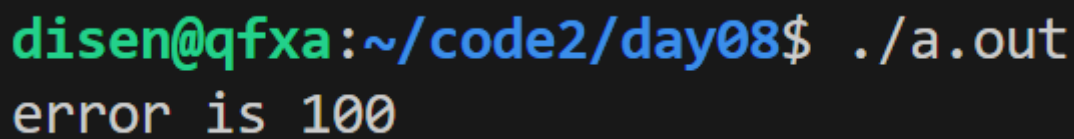
```

```

        else if (x == 4)
            throw 1.25;
        else if (x == 5)
            throw A(100);
        cout << "x=" << x << endl;
    }

    int main()
    {
        try
        {
            show(5);
        }
        catch (int error)
        { // 与throw int 匹配
            cout << "error is " << error << endl;
        }
        catch (const char &error)
        {
            cout << "error is " << error << endl;
        }
        catch (A &error)
        {
            cout << "error is " << error.n << endl;
        }
    }
}

```



```

disen@qfxa:~/code2/day08$ ./a.out
error is 100

```

如2: 限制函数抛出的异常类型

```

#include <iostream>
using namespace std;

class A
{
public:
    int n;
    A(int n) : n(n) {}
};

void show(int x) throw(int,char)
{
    if (x == 1)
        throw 0;
    else if (x == 2)
        throw 'a';
    else if (x == 3)
        throw "abc";
    else if (x == 4)
        throw 1.25;
    else if (x == 5)
        throw A(100);
    cout << "x=" << x << endl;
}

```

```

}

int main()
{
    try
    {
        show(5);
    }
    catch (int error)
    { // 与throw int 匹配
        cout << "error is " << error << endl;
    }
    catch (const char &error)
    {
        cout << "error is " << error << endl;
    }
    catch (A &error)
    {
        cout << "error is " << error.n << endl;
    }
}

```

```

disen@qfxa:~/code2/day08$ ./a.out
terminate called after throwing an instance of 'A'
Aborted (core dumped)

```

如3：限制函数抛出异常

在函数的声明位置 使用 throw()

```

#include <iostream>
using namespace std;

void show(int x) throw()
{
    if (x == 1)
        throw 0;
    cout << "x=" << x << endl;
}

int main()
{
    try
    {
        show(1);
    }
    catch (int error)
    { // 与throw int 匹配
        cout << "error is " << error << endl;
    }
}

```

```
disen@qfxa:~/code2/day08$ ./a.out
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

2.3.4 栈解旋(unwinding)

异常被抛出后，从进入 try 块起，到异常被抛掷前，这期间在栈上构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反，这一过程称为栈的解旋。

如：

```
#include <iostream>

using namespace std;

class A
{
public:
    A()
    {
        cout << "A()" << endl;
    }
    ~A()
    {
        cout << "~A()" << endl;
    }
};

int main(int argc, char const *argv[])
{
    try
    {
        A a1;
        throw 0; // 抛出异常时，则会回收 a1栈中的空间（解旋）
    }
    catch (...)
    {
        cout << "异常被处理" << endl;
    }
    return 0;
}
```

```
A()
~A()
异常被处理
```

2.3.5 异常接口声明

在声明函数时，可以声明throw()可抛出的异常接口（基本数据类型、类、结构体）。即为函数内限制 throw 抛出异常信息的类型。

如：结构体变量可以作为异常信息抛出

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct ERROR_S
{
    string title;
    int errnum;
} error1;

void test1(int n) throw(ERROR_S, int)
{
    if (n == 0)
    {
        error1.title = "n参数不能为0";
        error1.errnum = 0;
        throw error1;
    }
    else if (n == 1)
    {
        throw 2;
    }
    cout << "n=" << n << endl;
}

int main(int argc, char const *argv[])
{
    try
    {
        test1(atoi(argv[1])); // 从命令行获取第一个参数
    }
    catch (ERROR_S &error)
    {
        cout << "error no: " << error.errnum << ", msg:" << error.title << endl;
    }
    catch (...)
    {
        cout << "n参数不能为1 " << endl;
    }

    return 0;
}
```



```
.disen@qfxa:~/code2/day08$ ./a.out 1
n参数不能为1
disen@qfxa:~/code2/day08$ ./a.out 0
error no: 0, msg:n参数不能为0
disen@qfxa:~/code2/day08$ ./a.out 5
n=5
```

2.3.6 异常变量生命周期

catch() 捕获异常信息的对象的生成周期，如果声明是变量或对象时，会在捕获到时，则会在栈中临时会创建变量或对象空间。

【建议】使用引用方式接收异常信息的变量或对象。

设计Exception类：

```
class Exception
{
private:
    string msg;

public:
    Exception()
    {
        msg = "";
        std::cout << "Exception()" << std::endl;
    }
    Exception(const string &msg)
    {
        this->msg = msg;
        cout << this << " Exception(cosnt string &)" << endl;
    }
    Exception(const Exception &other)
    {
        cout << this << " Exception(const Exception &)" << endl;
        this->msg = other.msg;
    }
    ~Exception()
    {
        cout << this << " ~Exception()" << endl;
    }
    string getMsg() const
    {
        return msg;
    }
};
```

如1：抛出对象，接收对象；会执行拷贝构造函数

```
#include <iostream>

using namespace std;
```

```

class Exception
{
private:
    string msg;

public:
    Exception()
    {
        msg = "";
        std::cout << "Exception()" << std::endl;
    }
    Exception(const string &msg)
    {
        this->msg = msg;
        cout << this << " Exception(cosnt string &)" << endl;
    }
    Exception(const Exception &other)
    {
        cout << this << " Exception(const Exception &)" << endl;
        this->msg = other.msg;
    }
    ~Exception()
    {
        cout << this << " ~Exception()" << endl;
    }
    string getMsg() const
    {
        return msg;
    }
};

int main(int argc, char const *argv[])
{
    try
    {
        throw Exception("测试123"); // 在栈中创建对象
    }
    catch (Exception error)
    {
        cout << error.getMsg() << endl;
    }
    return 0;
}

```

```

disen@qfxxa:~/code2/day08$ ./a.out
0x196cca0 Exception(cosnt string &)
0x7ffe10c84950 Exception(const Exception &)
测试123
0x7ffe10c84950 ~Exception()
0x196cca0 ~Exception()
disen@qfxxa:~/code2/day08$ 

```

throw Exception(...);
 catch(Exception error)
 创建 析构

如2：抛出对象（栈），接收引用

```

#include <iostream>

using namespace std;

int main(int argc, char const *argv[])

```

```

{
    try
    {
        throw Exception("测试123"); // 在栈中创建对象
    }
    catch (Exception &error)
    {
        cout << error.getMsg() << endl;
    }
    return 0;
}

```

```

disen@qfxa:~/code2/day08$ ./a.out
0x186eca0 Exception(const string &)
测试123
0x186eca0 ~Exception()

```

如3：抛出对象（堆），捕获指针；在处理完异常之后，需要手动delete

```

#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    try
    {
        throw new Exception("测试123"); // 在堆中创建对象
    }
    catch (Exception *error)
    {
        cout << error->getMsg() << endl;
        delete error;
    }
    return 0;
}

```

```

disen@qfxa:~/code2/day08$ ./a.out
0x1ea8cb0 Exception(const string &)
测试123
0x1ea8cb0 ~Exception()

```

2.3.7 异常的多态使用

throw抛出的异常信息是属于子类对象，catch捕获异常的类型是父类的引用或指针

如：对Exception类进行派生出 ZeroDivisionException和OutOfRangeException两个类

ZeroDivisionException是除数为零的异常

OutOfRangeException 元素下标越界的异常

```

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

class Exception
{
private:
    string msg;

public:
    Exception()
    {
        msg = "";
        std::cout << "Exception()" << std::endl;
    }
    Exception(const string &msg)
    {
        this->msg = msg;
        cout << this << " Exception(const string &)" << endl;
    }
    Exception(const Exception &other)
    {
        cout << this << " Exception(const Exception &)" << endl;
        this->msg = other.msg;
    }
    ~Exception()
    {
        cout << this << " ~Exception()" << endl;
    }
    string getMsg() const
    {
        return msg;
    }
};

class ZeroDivisionException : public Exception
{
public:
    ZeroDivisionException() {}
    ZeroDivisionException(const string &msg) : Exception(msg) {}
};

class OutOfRangeException : public Exception
{
public:
    OutOfRangeException() {}
    OutOfRangeException(const string &msg) : Exception(msg) {}
};

int main(int argc, char const *argv[])
{
    int n = atoi(argv[1]);
    try
    {
        // 抛出子类异常的对象
        if (n == 0)
            throw ZeroDivisionException("除数不能为0");
        else if (n >= 20)
    }
}

```

```

        throw OutOfRangeException("范围越界了");
    else if (n == 10)
        throw Exception("n不能为10");
    cout << "argv[1]=" << n << endl;
}
catch (ZeroDivisionException &error1) // 子类异常
{
    cout << "精准捕获到子类的异常" << error1.getMsg() << endl;
}
catch (Exception &error2) // 父类的引用，父类异常
{
    cout << error2.getMsg() << endl;
}
return 0;
}

```

```

disen@qfxxa:~/code2/day08$ ./a.out 0
0x253bca0 Exception(cosnt string &)
精准捕获到子类的异常除数不能为0
0x253bca0 ~Exception()
disen@qfxxa:~/code2/day08$ ./a.out 20
0x12acca0 Exception(cosnt string &)
范围越界了
0x12acca0 ~Exception()
disen@qfxxa:~/code2/day08$ ./a.out 10
0xf56ca0 Exception(cosnt string &)
n不能为10
0xf56ca0 ~Exception()
disen@qfxxa:~/code2/day08$ ./a.out 5
argv[1]=5
disen@qfxxa:~/code2/day08$

```

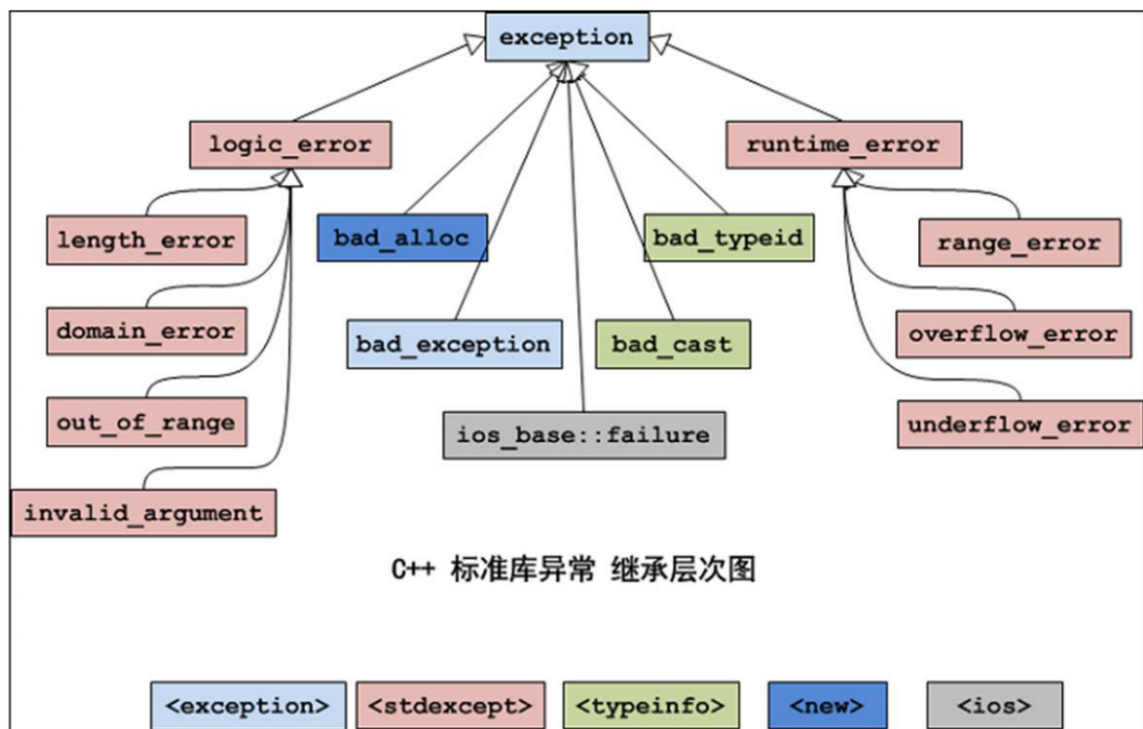
【小结】如果捕获异常的类型即为子类又有父类时，建议将父类放在最后，以防止抛出子类的异常时，无法精准处理。

2.3 C++标准异常库

C++提供了异常类的基类： `exception`

2.3.1 标准异常库的说明

c++提供的异常类的常用类： 使用时引入 `<exception>` 头文件



标准异常类的成员：

- 1) 每个类都提供了构造函数、复制构造函数、和赋值操作符重载。
- 2) `logic_error` 类及其子类、`runtime_error` 类及其子类，它们的构造函数是接受一个 `string` 类型的形式参数，用于异常信息的描述
- 3) 所有的异常类都有一个 `what()` 方法，返回 `const char*` 类型（C 风格字符串）的值，描述异常信息。

标准异常类的具体描述：

`exception` 所有标准异常类的父类

`bad_alloc` 当 `operator new` and `operator new[]`，请求分配内存失败时

`bad_exception` 这是个特殊的异常，如果函数的异常抛出列表里声明了 `bad_exception` 异常，当函数内部抛出了异常抛出列表中没有的异常，这是调用的 `unexpected` 函数中若抛出异常，不论什么类型，都会被替换为 `bad_exception` 类型。

`bad_typeid` 使用 `typeid` 操作符，操作一个 `NULL` 指针，而该指针是带有虚函数的类，这时抛出 `bad_typeid` 异常

`bad_cast` 使用 `dynamic_cast` 转换引用失败的时候

`ios_base::failure` io 操作过程出现错误

`logic_error` 逻辑错误，可以在运行前检测的错误

`runtime_error` 运行时错误，仅在运行时才可以检测的错误

`logic_error` 的子类：

`length_error` 试图生成一个超出该类型最大长度的对象时，例如 `vector` 的 `resize` 操作

`domain_error` 参数的值域错误，主要用在数学函数中。例如使用一个负值调用只能操作非负数的函数

`out_of_range` 超出有效范围

`invalid_argument` 参数不合适。在标准库中，当利用 `string` 对象构造 `bitset` 时，而 `string` 中的字符不是 '0' 或 '1' 的时候，抛出该异常

`runtime_error` 的子类：

range_error 计算结果超出了有意义的值域范围
overflow_error 算术计算上溢
underflow_error 算术计算下溢

如1:

```
#include <iostream>
#include <exception>
#include <cstdlib>
#include <cstdio>

using namespace std;

int maxVal(int a, int b)
{
    if (a == b)
    {
        char msg[100];
        sprintf(msg, "两个参数不能相同: %d == %d", a, b);
        throw invalid_argument(msg);
    }

    return a > b ? a : b;
}

int main(int argc, char const *argv[])
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    try
    {
        int ret = maxVal(a, b);
        cout << "maxValue is " << ret << endl;
    }
    catch (exception &error)
    {
        cout << "error:" << error.what() << endl;
    }

    return 0;
}
```

```
disen@qfxa:~/code2/day08$ ./a.out 4 4
error:两个参数不能相同: 4 == 4
disen@qfxa:~/code2/day08$ ./a.out 4 5
maxValue is 5
```

如2:

```
#include <iostream>
#include <exception>

using namespace std;
```

```

template <typename T>
class Stack
{
private:
    T *mData;        // 数据
    int mCapacity;    // 最大空间
    int mIndex;       // 当前元素的操作的下标

public:
    Stack(int capacity) : mCapacity(capacity)
    {

        mData = new T[capacity];
        mIndex = -1;
    }
    ~Stack()
    {
        delete[] mData;
        mData = NULL; // c++中的nullptr空指针
    }
    T pop() throw(out_of_range)
    {
        if (mIndex == -1)
            throw out_of_range("当前栈是空的，请先添加数据再操作");
        return mData[mIndex--];
    }
    Stack<T> &push(const T &item) throw(out_of_range)
    {
        if (mIndex == mCapacity - 1)
            throw out_of_range("栈已满，不能再存放数据");

        mData[++mIndex] = item;
        return *this;
    }
    T &at(int index) throw(out_of_range)
    {
        if (mIndex == -1 || index < 0 || index > mIndex)
            throw out_of_range("栈是空或位置无效");

        return mData[index];
    }
    int size()
    {
        return mIndex + 1;
    }
};

int main(int argc, char const *argv[])
{
    Stack<int> s1(5);
    s1.push(10).push(20).push(15);
    for (int i = 0; i < s1.size(); i++)
    {
        cout << s1.at(i) << endl;
    }
    cout << "弹出数据" << endl;
    while (1)
    {

```



```

        try
        {
            cout << s1.pop() << endl;
        }
        catch (exception &error)
        {
            cout << "error: " << error.what() << endl;
            break;
        }
    }
    cout << "-----清空之后打印数据-----" << endl;
    for (int i = 0; i < s1.size(); i++)
    {
        cout << s1.at(i) << endl;
    }
    return 0;
}

```

```

disen@qfxa:~/code2/day08$ ./a.out
10
20
15
弹出数据
15
20
10
error: 当前栈是空的，请先添加数据再操作
-----清空之后打印数据-----

```

2.3.2 自定义标准异常类

① 自己的异常类要继承标准异常类。

因为 C++ 中可以抛出任何类型的异常，所以我们的异常类可以不继承自标准异常，但是这样可能会导致程序混乱，尤其是当我们多人协同开发时。

② 当继承标准异常类时，应该重载父类的 what 函数和虚析构函数。

③ 因为栈展开的过程中，要复制异常类型，那么要根据你在类中添加的成员考虑是否提供自己的复制构造函数

如：自定义 OutOfRangeError

```

class OutOfRangeError : public exception
{
public:
    virtual const char *what() const throw()
    {
        return "访问位置越界";
    }
};

```

应用示例:

```
#include <iostream>
#include <exception>

using namespace std;

class OutOfRangeError : public exception
{
public:
    virtual const char *what() const throw()
    {
        return "访问位置越界";
    }
};

template <typename T>
class Stack
{
private:
    T *mData;          // 数据
    int mCapacity;     // 最大空间
    int mIndex;        // 当前元素的操作的下标

public:
    Stack(int capacity) : mCapacity(capacity)
    {
        mData = new T[capacity];
        mIndex = -1;
    }
    ~Stack()
    {
        delete[] mData;
        mData = NULL; // c++中的nullptr空指针
    }
    T pop() throw(OutOfRangeError)
    {
        if (mIndex == -1)
            throw OutOfRangeError();
        return mData[mIndex--];
    }
    Stack<T> &push(const T &item) throw(OutOfRangeError)
    {
        if (mIndex == mCapacity - 1)
            throw OutOfRangeError();

        mData[++mIndex] = item;
        return *this;
    }
    T &at(int index) throw(OutOfRangeError)
    {
        if (mIndex == -1 || index < 0 || index > mIndex)
            throw OutOfRangeError();

        return mData[index];
    }
};
```

```

int size()
{
    return mIndex + 1;
}

};

int main(int argc, char const *argv[])
{
    Stack<int> s1(5);
    s1.push(10).push(20).push(15);
    for (int i = 0; i < s1.size(); i++)
    {
        cout << s1.at(i) << endl;
    }
    cout << "弹出数据" << endl;
    while (1)
    {
        try
        {
            cout << s1.pop() << endl;
        }
        catch (exception &error)
        {
            cout << "error: " << error.what() << endl;
            break;
        }
    }
    cout << "-----清空之后打印数据-----" << endl;
    for (int i = 0; i < s1.size(); i++)
    {
        cout << s1.at(i) << endl;
    }
    return 0;
}

```

晚上任务：

- 1) `const` 能否修饰`class`类
- 2) `const`修饰的成员函数，能否在子类重写

三、STL标准模板库开发

3.1 STL概念

为了建立数据结构和算法的一套标准，并且降低他们之间的耦合关系，以提升各自的独立性、弹性、交互操作性(相互合作性,interoperability),诞生了 STL。

STL(Standard Template Library,标准模板库)，是惠普实验室开发的一系列软件的统称。现在主要出现在 c++中，但是在引入 c++之前该技术已经存在很长时间了。

STL(Standard Template Library)标准模板库,在我们 c++标准程序库中隶属于 STL 的占到了 80%以上。

六大组件:

容器: 数据结构, 用于存放数据; 如 `vector`、`list`、`deque`、`set`、`map` 【类模板】
算法: 操作数据的各种功能, 如删除、排序、查询等 【函数模板】
迭代器: 算法借助 迭代器 操作数据 (主要读数据) 【各种运算符重载】
仿函数: 算法的某种策略, 增强算法的功能。 【()运算符重载】
适配器: 用于扩展容器、算法、迭代器的接口
空间管理器: 负责空间的配置与管理 【类模板】

STL 优点:

- 1) STL 是 C++的一部分, 不需要安装外部库
- 2) STL 将数据和操作分离
- 3) STL 具有高可重用性, 高性能, 高移植性, 跨平台的优点。
高可重用性: 采用了模板类和模板函数
高性能: 可以高效地从大量的数据中快速查找, 如`map`采用红黑树的结构。
高移植性: 只要存在C++的编译环境的操作系统, 都可以编译和运行STL模块。

STL 之父 Alex Stepanov 亚历山大·斯特潘诺夫(STL 创建者)。

3.2 STL 三大组件的基本用法

容器 (数据结构, `vector`、`set`、`map`、`queue`)、算法 (插入数据、删除数据、修改数据、排序等)、迭代器 (算法操作容器)。

3.2.1 容器

容器: 用于存放数据的

常用的数据结构:

数组(`array`)
链表(`list`)
树 (`tree`)
栈(`stack`),
队列(`queue`)
集合(`set`)
映射表(`map`)

根据数据在容器中的排列特性: 序列式容器、关联式容器

序列式容器:

强调值的排序, 每个元素均有固定的位置, 除非用删除或插入的操作改变这个位置,
如 `vector`, `deque/queue`, `list`;

关联式容器:

非线性, 更准确的说是二叉树结构, 各元素之间没有严格的物理上的顺序关系; 在数据中选择一个关键字`key`, 这个`key`对数据起到索引的作用, 方便查找。
如: `Set/multiset`, `Map/multimap` 容器

【注意】容器可以嵌套容器

3.2.2 算法

算法(Algorithm): 用于解决问题的

STL 收录的算法经过了数学上的效能分析与证明, 是极具复用价值的, 包括常用的排序, 查找等。特定的算法往往搭配特定的数据结构, 算法与数据结构相辅相成。

算法分为: 质变算法和非质变算法。

质变算法: 是指运算过程中会更改区间内的元素的内容; 例如拷贝、替换、删除等
非质变算法: 是指运算过程中不会更改区间内的元素内容, 如查换、统计、求极值等

3.2.3 迭代器

迭代器(iterator)是一种抽象的设计概念, 使之能够依序寻访某个容器所含的各个元素, 而又无需暴露该容器的内部表示方式。

简之, 迭代器是依次遍历容器中所有的元素。

迭代器的种类:

输入迭代器: 只读数据, 支持 ++、==, !=
输出迭代器: 只写数据, 支持 ++
向前迭代器: 读写数据 (向前), 支持 ++、==, !=
双向迭代器: 读写数据 (向前、向后), 支持 ++、--
随机迭代器: 提供读写操作 (跳跃式访问任意位置), 支持 ++、--、[], -n, <, <=, >, >=

3.2.4 初次使用

如1: 存放基础类型的数据

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

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);
    v1.push_back(4);
    v1.push_back(5);
    v1.push_back(6);

    // 创建迭代器
    vector<int>::iterator it;
    for (it = v1.begin(); it != v1.end(); it++)
    {
        cout << *it << endl;
    }

    return 0;
}
```

```
disen@qfxa:~/code2/day08$ ./a.out
1
2
3
4
5
6
```

如2：存放类对象的数据

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

using namespace std;

class Student
{
    friend ostream &operator<<(ostream &cout, Student &s);

private:
    string name;
    int age;

public:
    Student(const string &name, int age)
    {
        this->name = name;
        this->age = age;
    }
};

ostream &operator<<(ostream &cout, Student &s)
{
    cout << "name=" << s.name << ",age=" << s.age;
    return cout;
}

int main(int argc, char const *argv[])
{
    vector<Student> v1;
    v1.push_back(Student("disen", 20));
    v1.push_back(Student("jack", 18));
    v1.push_back(Student("lucy", 21));
    v1.push_back(Student("mack", 23));

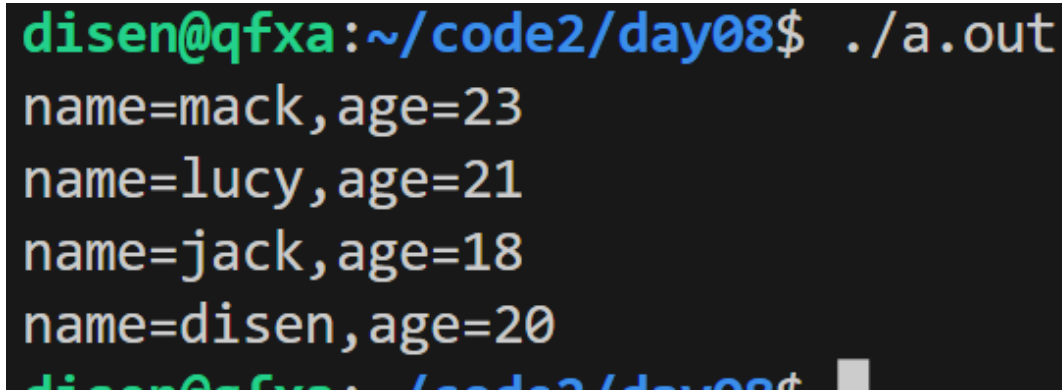
    // 创建迭代器
    vector<Student>::iterator it = v1.end();
    // 倒序打印
```

```

while (it != v1.begin())
{
    it--;
    cout << *it << endl;
}

return 0;
}

```



```

disen@qfxa:~/code2/day08$ ./a.out
name=mack,age=23
name=lucy,age=21
name=jack,age=18
name=disen,age=20
disen@qfxa:~/code2/day08$

```

如3: 存放对象的指针

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Student
{
    friend ostream &operator<<(ostream &cout, Student *s);

private:
    string name;
    int age;

public:
    Student(const string &name, int age)
    {
        this->name = name;
        this->age = age;
    }
};

ostream &operator<<(ostream &cout, Student *s)
{
    cout << "name=" << s->name << ",age=" << s->age;
    return cout;
}

int main(int argc, char const *argv[])
{
    vector<Student *> v1;
    v1.push_back(new Student("disen", 20));
    v1.push_back(new Student("jack", 18));
    v1.push_back(new Student("lucy", 21));
}

```

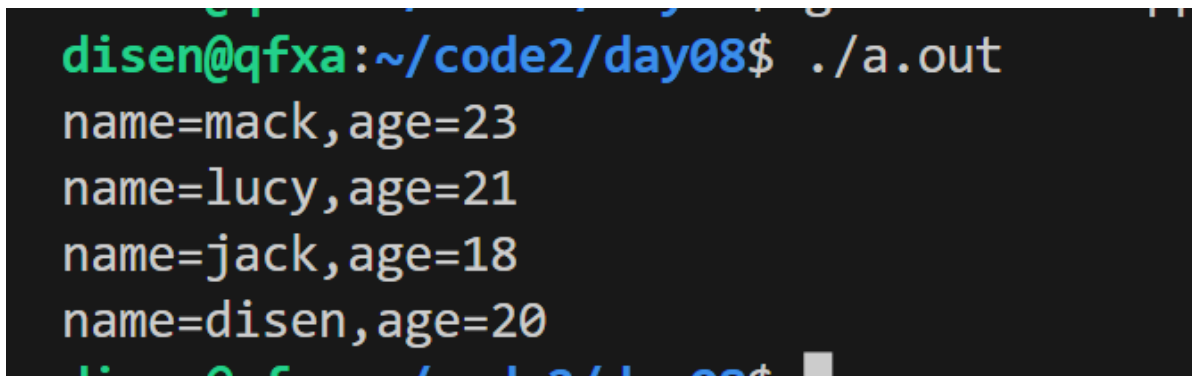
```

v1.push_back(new Student("mack", 23));

// 创建迭代器
vector<Student *>::iterator it = v1.end();
// 倒序打印，并释放vector容器中存储的对象指针
while (it != v1.begin())
{
    it--;
    cout << *it << endl;
    delete *it; // 取出vector存储的元素 (Student *)
}

return 0;
}

```



```

disen@qfxa:~/code2/day08$ ./a.out
name=mack,age=23
name=lucy,age=21
name=jack,age=18
name=disen,age=20
disen@qfxa:~/code2/day08$

```

如4：嵌套容器的用法

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(int argc, char const *argv[])
{
    vector<vector<int>> > vs;
    vector<int> v1, v2, v3;

    for (int i = 0; i < 9; i++)
    {
        switch (i / 3)
        {
            case 0:
                v1.push_back(i + 1);
                break;
            case 1:
                v2.push_back(i + 1);
                break;
            case 2:
                v3.push_back(i + 1);
                break;
        }
    }

    vs.push_back(v1);
    vs.push_back(v2);

```



```

vs.push_back(v3);

vector<vector<int>>::iterator it1 = vs.begin();
while (it1 != vs.end())
{
    vector<int>::iterator it2 = (*it1).begin();
    while (it2 != (*it1).end())
    {
        cout << *it2 << "\t";
        it2++;
    }
    cout << endl;

    it1++;
}

return 0;
}

```

```

1       2       3
4       5       6
7       8       9

```

【注意】vscode会自动格式化代码，g++编译器在编译 `>>` 认为是运算符重载，在表达 `vector<vector<int>>` 容器嵌套时，应该将两个 `>` 中间加个空格（vim编辑器修改）。