OOP HW4 错题整理：Template, STL, String & Exception

Notes

2026 年 6 月 17 日

总览

本份笔记按照前面整理出的 **1–100 题编号**记录，不重新编号。整理范围包括：

1, 5, 17, 18, 25, 27, 30, 31, 32, 34, 41, 43, 44, 49, 53, 55, 66, 67, 71–72, 87

以及前面单独问过的异常题：

69, 73, 78, 86, 93, 94.

核心主线：

- **template**：模板不是具体代码，实例化后才形成具体函数或具体类。
- **class template**：使用类模板创建对象时，一般需要写模板实参，如 `MyClass<int> a(10);`。
- **STL**：容器、迭代器、算法相互配合，不是完全孤立的三部分。
- **string**：注意 `cin >> s`、`getline`、下标越界和字符运算。
- **exception**：多个 `catch` 从上到下匹配；派生类异常应写在基类异常之前；`throw;` 表示重抛当前异常。

一、模板与类模板

Question 1

原题：现有声明：

```
template <class T>
class Test { ... };
```

则以下哪一个声明不可能正确？

- **A.** `Test a;`
- B. `Test<int> a;`
- C. `Test<float> a;`

- D. `Test<Test<int>> a;`

答案：A

解析：类模板不是具体类，使用时通常要给出模板实参：

```
Test<int> a;
Test<float> b;
Test<Test<int>> c;
```

而

```
Test a;
```

没有说明 T 是什么类型，因此不正确。

易错点：函数模板可以通过函数实参自动推导类型，但类模板创建对象时，传统 C++ 语境下一般不能直接省略模板实参。本题按基础 C++ 题库语境处理。

Question 5

原题：关于函数模板，描述错误的是。

- A. 函数模板必须由程序员实例化为可执行的函数模板
- B. 函数模板的实例化由编译器实现
- C. 一个类定义中，只要有一个函数模板，则这个类是类模板
- D. 类模板的成员函数都是函数模板，类模板实例化后，成员函数也随之实例化

严格看：A、C 都有问题；若单选，题库多半想考 A

解析：

- A. “函数模板必须由程序员实例化”错误。函数模板通常由编译器根据调用自动实例化。
- B. “函数模板的实例化由编译器实现”正确。
- C. “一个类定义中只要有一个函数模板，则这个类是类模板”错误。普通类中也可以有成员函数模板。
- D. “类模板的成员函数都是模板函数”是教材中的粗略说法，严格术语下不完全准确。

例如：

```
template<class T>
T myMax(T x, T y) {
    return x > y ? x : y;
}
```

```
int main() {
    myMax(1, 2);        // 编译器自动实例化 myMax<int>
    myMax(1.0, 2.0);    // 编译器自动实例化 myMax<double>
}
```

补充：普通类也可以有成员函数模板。

```
class A {
public:
    template<class T>
    void print(T x) {
        cout << x << endl;
    }
};
```

这个 A 不是类模板。

Question 17

原题：类模板的使用实际上是将类模板实例化成一个_____。

- A. 函数
- B. 对象
- **C. 类**
- D. 抽象类

答案：C

解析：类模板本身不是一个具体类。使用时，编译器把类模板实例化成具体类：

```
template<class T>
class Box {
    T value;
};

Box<int> a;       // Box<int> 是具体类
Box<double> b;    // Box<double> 是另一个具体类
```

一句话：类模板 → 模板类/具体类 → 对象。

Question 18

原题：下列关于模板的说法中，错误的是_____。

- A. 用模板定义一个对象时，不能省略参数

- **B. 类模板只能有虚拟参数类型**
- C. 类模板的成员函数都是模板函数
- D. 类模板在编译中不会生成任何代码

答案：B

解析：这里的“虚拟参数类型”是老教材说法，大致指 **类型模板参数**，例如：

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

其中 `T` 是类型模板参数。

但类模板不只能有类型参数，还可以有 **非类型模板参数**：

```
template<class T, int N>
class Array {
    T data[N];
};

Array<int, 10> a;  // T = int, N = 10
```

关于“类模板的成员函数都是模板函数”：

```
template<class T>
class A {
public:
    void test() {
        cout << "hello";
    }
};
```

这里 `test()` 没有自己的 `template<class U>`，所以严格说它不是“函数模板”，而是**类模板的普通成员函数**。但它会随着 `A<int>`、`A<double>` 等类模板实例化而生成对应版本。所以教材有时粗略地说“类模板的成员函数都是模板函数”。

更准确的说法：类模板的成员函数会随类模板实例化而实例化；只有自己带 `template<class U>` 的成员函数才严格叫成员函数模板。

Question 25

原题：下列关于类模板的定义，正确的是_____。

- **A.** `template<class T, int i = 0>`
- B. `template<class T, class int i>`
- C. `template<class T, typename T>`

- D. `template<class T1, T2>`

答案：A

解析：

```
template<class T, int i = 0>
class A {};
```

是合法的。这里 `T` 是类型模板参数，`i` 是非类型模板参数，并且 `i` 有默认值。

为什么 `template<class T, typename T>` 错？

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

两个模板参数都叫 `T`，在同一作用域内重名，非法。类似于不能写：

```
void f(int x, double x) {}
```

正确写法应为：

```
template<class T, typename U>
class PairLike {};
```

Question 32

原题：模板的使用是为了（）。

- **A.** 提高代码的可重用性
- B. 提高代码的运行效率
- C. 加强类的封装性
- D. 实现多态性

答案：A

解析：模板让同一套代码适用于不同类型：

```
template<class T>
T add(T a, T b) {
    return a + b;
}
```

这样既可以处理 `int`，也可以处理 `double`，主要目的就是**代码复用**。

Question 34

原题：关于函数模板，描述错误的是（　）。

- A. 函数模板的实例化由编译器实现
- B. 函数模板必须由程序员实例化为可执行的函数模板
- C. 类模板的成员函数都是函数模板，类模板实例化后，成员函数也随之实例化
- D. 一个类定义中，只要有一个函数模板，这个类就是类模板

严格看：B、D 都有问题；若单选，题库通常想考 B

解析：

- A. 函数模板的实例化由编译器实现。正确。
- B. 函数模板必须由程序员实例化。错误。通常编译器会根据调用自动实例化。
- C. 类模板的成员函数随类模板实例化而实例化。教材语境中通常认为正确。
- D. 一个类定义中只要有一个函数模板，这个类就是类模板。严格错误。

不用程序员自己实例化的例子：

```
template<class T>
T maxValue(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = maxValue(3, 5);           // 自动生成 maxValue<int>
    double b = maxValue(3.1, 5.2);    // 自动生成 maxValue<double>
}
```

程序员没有写：

```
maxValue<int>(3, 5);
maxValue<double>(3.1, 5.2);
```

但编译器会自动推导并实例化。

Question 49

原题：A template class can have _____.

- **A.** More than one generic data type
- B. Only one generic data type

- C. At most two data types
- D. Only generic type of integers and not characters

答案：A

解析：类模板可以有多个类型参数：

```
template<class T1, class T2>
class MyPair {
    T1 first;
    T2 second;
};
```

使用时：

```
MyPair<int, double> p;
```

Question 53

原题：Which is the most significant feature that arises by using template classes?

- A. Code readability
- B. Ease in coding
- **C.** Code reusability
- D. Modularity in code

答案：C

解析：类模板的核心价值是用一份类定义适配多种类型：

```
vector<int> vi;
vector<double> vd;
vector<string> vs;
```

`vector` 只写一套模板，但可以实例化出多种具体容器类型。

Question 55

原题：How is function overloading different from template class?

- A. Overloading is multiple function doing same operation, Template is multiple function doing different operations
- B. Overloading is single function doing different operations, Template is multiple function doing different operations

- **C.** Overloading is multiple function doing similar operation, Template is multiple function doing identical operations
- D. Overloading is multiple function doing same operation, Template is same function doing different operations

答案：C

解析：函数重载是多个函数使用同一个函数名，但参数表不同；模板则是用同一个模式处理不同类型。题库选项中 C 最接近：

Overloading：多个函数做相似操作；Template：同一模式处理不同类型。

例如重载：

```
int maxValue(int a, int b);
double maxValue(double a, double b);
```

模板：

```
template<class T>
T maxValue(T a, T b);
```

易错点：重载强调“多个函数”；模板强调“一套代码模式，多种实例化”。

二、STL、迭代器与 pair

迭代器知识总括

STL 的核心结构：

- **容器 container**：存数据，例如 `vector`, `list`, `map`, `set`。
- **迭代器 iterator**：像指针一样访问容器中的元素。
- **算法 algorithm**：通过迭代器操作容器，例如 `sort`, `find`, `copy`。

常见迭代器类型：

- **Input Iterator**：只能读，只能向前走。
- **Output Iterator**：只能写，只能向前走。
- **Forward Iterator**：可多次遍历，只能向前。
- **Bidirectional Iterator**：可前进也可后退，如 `list` 迭代器。
- **Random Access Iterator**：可随机访问，如 `vector` 迭代器。

不存在“删除迭代器”这种基本迭代器类别。

Question 27

原题：设有如下代码段：

```
std::map<char *, int> m;
const int MAX_SIZE = 100;
int main() {
    char str[MAX_SIZE];
    for (int i = 0; i < 10; i++) {
        std::cin >> str;
        m[str] = i;
    }
    std::cout << m.size() << std::endl;
}
```

读入 10 个字符串，则输出的 `m.size()` 为：

- A. 0
- **B. 1**
- C. 10

答案：B

解析：`map<char*, int>` 的 key 是 `char*` 指针。这里每次输入都写入同一个数组 `str`，所以 `str` 转成指针后地址始终相同。于是 map 认为 key 一直是同一个，最终只有一个元素。

如果想按字符串内容作为 key，应使用：

```
std::map<std::string, int> m;
```

Question 30：pair 模板

原题：下列关于 `pair<>` 类模板的描述中，错误的是。

- A. `pair<>` 类模板定义在头文件 `utility` 中
- B. `pair<>` 类模板作用是将两个数据组成一个数据，两个数据可以是同一个类型也可以是不同的类型
- C. 创建 `pair<>` 对象只能调用其构造函数
- D. `pair<>` 类模板提供了两个成员函数 `first` 与 `second` 来访问这两个数据

题库多半选：C；但 D 的表述也不严谨

解析：

- A. `pair` 定义在头文件 `<utility>` 中。正确。
- B. `pair` 可以把两个数据组合成一个数据，二者类型可以相同也可以不同。正确。
- C. 创建 `pair` 对象只能调用构造函数。错误，因为还可以用 `make_pair`。
- D. `first` 和 `second` 不是成员函数，而是数据成员；所以这句话表述也不严谨。

示例：

```
#include <utility>
using namespace std;

pair<int, string> p1(1, "Tom");
auto p2 = make_pair(2, string("Jerry"));

cout << p1.first << "␣" << p1.second << endl;
```

严格记法：

```
p.first;   // 数据成员
p.second;  // 数据成员
```

不是：

```
p.first();   // 错
p.second();  // 错
```

Question 31

原题：下列选项中，哪一项不是迭代器。

- A. 输入迭代器
- B. 前向迭代器
- C. 双向迭代器
- **D. 删除迭代器**

答案：D

解析：输入迭代器、前向迭代器、双向迭代器都是常见迭代器类别；没有“删除迭代器”这一基本类别。

注意：`erase` 是容器的删除操作，不是迭代器类别：

```
vector<int> v = {1, 2, 3};
auto it = v.begin();
v.erase(it);
```

三、string 与输入输出

Question 41

原题：有代码如下：

```
string s;
s[0] = '1';
```

则关于以上语句说法正确的是（　）。

- **A.** 语句 `"s[0]='1';"` 有问题
- B. s 的值为字符串 `"1"`
- C. s 是空格串
- D. s 是空串

答案：A

解析：默认构造出的 s 是空字符串，没有第 0 个字符。直接访问 `s[0]` 越界，属于未定义行为。

正确做法：

```
string s;
s.push_back('1');      // 方法一

string t;
t.resize(1);
t[0] = '1';            // 方法二
```

Question 43

原题：有代码如下：

```
int n;
string s;
cin >> n;
getline(cin, s);
cout << s.size() << endl;
```

则在输入以下数据后得到结果是（　）。

```
1
Hello World
```

- A. 11

- **B. 0**
- C. 5
- D. 12

答案：B

解析： `cin >> n` 只读走数字 1，但留下后面的换行符 `'\n'`。紧接着 `getline(cin, s)` 会读到这个换行符之前的空内容，所以 `s` 是空串，长度为 0。

正确写法：

```
int n;
string s;
cin >> n;
cin.ignore();           // 丢掉换行符
getline(cin, s);
```

如果担心缓冲区中还有更多字符，可以写：

```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

Question 44

原题： 以下代码的输出结果是（　　）。

```
string res="";
string s,t="123456";
s=string(3,'0'); //相当于s="000";
s=s+"123";
for(int i=5;i>=0;i--) {
      char c=s[i]+t[i]-'0';
      res=c+res;
}
cout<<res<<endl;
```

- A. 975321
- B. 236456
- C. 654632
- **D. 123579**

答案：D

解析：这里不是整数加法，而是字符运算。逐位计算：

$$0+1 \to 1,$$
$$0+2 \to 2,$$
$$0+3 \to 3,$$
$$1+4 \to 5,$$
$$2+5 \to 7,$$
$$3+6 \to 9.$$

因此结果是：

```
123579
```

易错点：代码没有处理进位，所以不是普通的大整数加法模板，只是每一位单独相加。

四、异常处理

Question 66

原题：What is wrong in the following code?

```
vector<int> v;
v[0] = 2.5;
```

- A. The program has a compile error because there are no elements in the vector.
- B. The program has a compile error because you cannot assign a double value to `v[0]`.
- **C.** The program has a runtime error because there are no elements in the vector.
- D. The program has a runtime error because you cannot assign a double value to `v[0]`.

答案：C

解析：`vector<int> v;` 创建的是空容器，此时没有第 0 个元素。访问 `v[0]` 越界。

严格 C++ 说法：`v[0]` 越界是未定义行为，不一定稳定表现为“运行时错误”。但考试题一般按运行时错误处理。

注意：`2.5` 赋给 `int` 可以发生类型转换，不是主要错误。

正确做法：

```
vector<int> v;
v.push_back(2);  // 先插入元素
v[0] = 2;
```

Question 67

原题： If you enter `1 0`, what is the output of the following code?

```
#include "iostream"
using namespace std;

int main()
{
    cout << "Enter␣two␣integers:␣";
    int number1, number2;
    cin >> number1 >> number2;

    try
    {
        if (number2 == 0)
            throw number1;

        cout << number1 << "␣/␣" << number2 << "␣is␣"
             << (number1 / number2) << endl;

        cout << "C" << endl;
    }
    catch (int e)
    {
        cout << "A";
    }

    cout << "B" << endl;
    return 0;
}
```

- A. A
- B. B
- C. C
- **D. AB**

答案：D

解析： 输入 `number1 = 1, number2 = 0`，所以抛出 `int` 类型异常。程序跳到：

```
catch (int e) {
    cout << "A";
}
```

输出 A。然后 catch 结束后，继续执行：

```
cout << "B" << endl;
```

所以最终输出：

```
AB
```

Question 69

原题： Which of the following statements are true?

- **A.** A custom exception class is just like a regular class.
- B. A custom exception class must always be derived from class exception.
- C. A custom exception class must always be derived from a derived class of class exception.
- D. A custom exception class must always be derived from class runtime_error.

答案：A

解析： C++ 中自定义异常类本质上就是普通类，不要求必须继承 std::exception：

```
class MyException {};

throw MyException();
```

工程上推荐：

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "MyException";
    }
};
```

这样可以用：

```
catch (const std::exception& e) {
    cout << e.what();
}
```

Question 71 与 Question 72 对比

共同代码：

```
try {
    statement1;
    statement2;
    statement3;
}
catch (Exception1 ex1)
{
}
catch (Exception2 ex2)
{
}
catch (Exception3 ex3)
{
    statement4;
    throw;
}
statement5;
```

Question 71

原题： Suppose that `statement2` throws an exception of type `Exception2` in the above statement. Which statement is executed after `statement2` is executed?

- A. statement2
- B. statement3
- C. statement4
- **D. statement5**

Question 71 答案：D

执行流程：

- 执行 `statement1`。
- 执行 `statement2`，抛出 `Exception2`。
- `statement3` 不执行。
- 进入 `catch(Exception2 ex2)`。
- 该 `catch` 中没有 `throw;`，所以异常处理结束。
- 继续执行 `statement5`。

Question 72

原题： Suppose that `statement3` throws an exception of type `Exception3` in the above statement. Which statements are executed after `statement3` is executed?

- A. statement2
- B. statement3
- **C. statement4**
- D. statement5

Question 72 答案：C

执行流程：

- 执行 `statement1`。
- 执行 `statement2`。
- 执行 `statement3`，抛出 `Exception3`。
- 进入 `catch(Exception3 ex3)`。
- 执行 `statement4`。
- 执行 `throw;`，重抛当前异常。
- 当前 `try-catch` 后面的 `statement5` 不执行。

核心区别：

`catch(Exception2)` 没有重抛，所以继续执行 statement5；

`catch(Exception3)` 有 `throw;`，所以异常继续向外传播。

Question 73

原题： 下列关于异常处理的说法不正确的是（　）。

- A. 异常处理的 `throw` 与 `catch` 通常不在同一个函数中，实现异常检测与异常处理的分离。
- B. `catch` 语句块必须跟在 `try` 语句块的后面，一个 `try` 语句块后可以有多个 `catch` 语句块。
- **C.** 在对函数进行异常规范声明时，若形参表后没有任何表示抛出异常类型的说明，它表示该函数不能抛出任何异常。
- D. `catch` 语句块中,`catch(...)` 表示该 `catch` 可以捕捉任意类型的异常,必须将 `catch(...)` 放在 `catch` 结构的最后。

答案：C

解析：普通函数声明：

```
void f();
```

并不表示 `f` 不能抛出异常。它只是没有写异常说明。

如果要表示函数不抛异常，C++11 以后应写：

```
void f() noexcept;
```

旧式写法是：

```
void f() throw();
```

Question 78

原题：下列关于断言的描述中，错误的是。

- A. 断言是调试程序的一种手段
- B. 若断言情况发生，一般会终止程序
- C. 在 C++ 中，宏 `assert()` 用来在调试阶段实现断言
- **D.** 断言在程序调试与发布版本中都可以使用断言

答案：D

解析：`assert()` 主要用于调试阶段。若断言失败，一般会终止程序：

```
#include <cassert>

int x = -1;
assert(x > 0);  // 断言失败，通常终止程序
```

如果发布版本中定义了 `NDEBUG`，断言通常会被关闭：

```
#define NDEBUG
#include <cassert>
```

一句话：断言是给程序员调试用的，不是给用户处理运行时错误用的。

Question 86

原题：How many catch blocks can a single try block can have?

- A. Only 1
- B. Only 2

- C. Maximum 127
- **D. As many as required**

答案：D

解析：一个 `try` 后面可以跟多个 `catch`：

```
try {
    // code
}
catch (const runtime_error& e) {
}
catch (const logic_error& e) {
}
catch (const exception& e) {
}
catch (...) {
}
```

多个 `catch` 从上往下匹配，只执行第一个匹配成功的。一般顺序：

派生类异常 → 基类异常 → `catch(...)`

Question 87

原题：To catch the exceptions _____.

- A. An object must be created to catch the exception
- **B.** A variable should be created to catch the exception
- C. An array should be created to catch all the exceptions
- D. A string have to be created to store the exception

考试答案：B

解析：题库想表达的是常见写法：

```
try {
    throw 10;
}
catch (int e) {
    cout << e;
}
```

这里 e 是 catch 参数变量。

但严格 C++ 说法：这个题目本身不严谨。因为 C++ 中可以不写变量名：

```
catch (const runtime_error&) {
    cout << "caught";
}
```

也可以用：

```
catch (...) {
    cout << "caught␣anything";
}
```

所以：

- A. “必须创建对象来捕获异常”不准确，异常对象是在 throw 时产生的。
- B. “应该创建一个变量”更符合题库语法意图，但不是严格必要条件。

Question 93

原题：If catching of base class exception is done before derived class in C++ _____.

- A. It gives compile time error
- B. It doesn’t run the program
- **C. It may give warning but not error**
- D. It always gives compile time error

答案：C

解析：如果基类异常写在派生类异常前面：

```
try {
    throw runtime_error("error");
}
catch (const exception& e) {
    cout << "base";
}
catch (const runtime_error& e) {
    cout << "derived";
}
```

由于 runtime_error 是 exception 的派生类，第一个 catch 已经能捕获它，后面的派生类 catch 就不可达。

通常不是编译错误，但编译器可能 warning。

正确顺序：

```
catch (const runtime_error& e) {
}
catch (const exception& e) {
}
```

Question 94

原题： If a catch block accepts more than one exceptions then _____.

- A. The catch parameters are not final
- **B. The catch parameters are final**
- C. The catch parameters are not defined
- D. The catch parameters are not used

答案：B

解析： 这题是 **Java multi-catch** 语法，不是 C++ 语法。例如 Java 中：

```
try {
    // code
}
catch (IOException | SQLException e) {
    // e is implicitly final
}
```

这里 e 是隐式 final，不能重新赋值。

C++ 注意： C++ 不支持这种写法：

```
catch (Exception1 | Exception2 e) { }  // C++ 错误
```

C++ 通常写多个 catch，或者捕获共同基类。

五、最开始单独问过的题号映射

你最开始问的问题	对应题号	核心结论
自定义异常类是否必须继承 exception	Question 69	不必须，答案 A
try 后可以有几个 catch	Question 86	按需多个，答案 D
catch 中为什么还能 throw;	Question 72	throw; 是重抛当前异常
异常处理说法不正确	Question 73	void f(); 不代表不能抛异常，答案 C
断言是什么	Question 78	assert 调试用，发布版可能关闭，答案 D
基类异常先于派生类异常捕获	Question 93	可能 warning，但通常非 error，答案 C
一个 catch 捕获多个异常	Question 94	Java multi-catch，参数隐式 final，答案 B

六、最后速记

- 类模板创建对象一般要写模板实参：`Test<int> a;`。
- 函数模板通常由编译器根据调用自动实例化，不需要程序员手动实例化。
- 类模板可以有类型参数，也可以有非类型参数，如 `template<class T, int N>`。
- `pair` 的 `first` 和 `second` 是数据成员，不是成员函数。
- STL 不是容器、迭代器、算法三者互不相干；算法靠迭代器操作容器。
- `map<char*, int>` 比较的是指针地址，不是字符串内容。
- 空 `string` 不能直接写 `s[0]='1'`。
- `cin >> n` 后接 `getline`，要先处理换行符。
- 空 `vector` 不能直接访问 `v[0]`。
- 多个 `catch` 从上往下匹配；派生类异常放前，基类异常放后。
- `throw;` 只能在处理异常时用于重抛当前异常。
- `assert` 是调试手段，不应作为发布版错误处理机制。
- Java 的 multi-catch 和 C++ 的异常语法不要混在一起。