

# oop

---

## oop

### 第一章、基本介绍

- 1.1 C语言优缺点
- 1.2 C++进步
- 1.3 字符串
- 1.4 动态内存分配
- 1.5 题目

### 第二章、Class

- 2.1 引用
- 2.2 类
  - 2.2.1 `::` resolver
  - 2.2.2 声明和定义
  - 2.2.3 this指针
  - 2.2.4 OOP 特性
- 2.3 构造和析构
  - 2.3.1 构造函数
  - 2.3.2 析构函数
  - 2.3.4 存储分配
- 2.4 题目

### 第三章 类和对象

- 3.1类的定义
  - 3.1.1 编译单元
  - 3.1.2 头文件

### 第四章、容器

- 4.1 STL (标准模板库)
  - 4.1.1 优点
  - 4.1.2 模板库内容:
  - 4.1.3 vector
  - 4.1.4 list
  - 4.1.5 map
  - 4.1.6 stl的使用
- 4.2 Iterators
- 4.3 Algorithms
- 4.4 Typdefs
- 4.5 Pitfalls
- 4.6 题目

### 第五章、Functions

- 5.1 局部变量
- 5.2 C++ access control
  - 5.2.1 访问形式
  - 5.2.2 Friends
  - 5.2.3 class vs. struct
- 5.3 Initialization
  - 5.3.1 初始化列表
  - 5.3.2 Overloaded constructors
  - 5.3.3 Pitfall of default arguments
- 5.4 inline function
- 5.5 题目

## 第六章、常量

### 6.1 const关键字

#### 6.1.1 Run-time constants

#### 6.1.2 聚合 (Aggregates)

### 6.2 Pointers and const

#### 6.2.1 String Literals

#### 6.2.2 Conversions

### 6.3 Passing and returning addresses

#### 6.4.1 Const member functions

#### 6.4.2 重载 **const** and none-const functions

#### 6.4.3 constants in classes

### 6.5 **static**静态

#### 6.5.1 全局对象 (Global objects)

#### 6.5.2 Static Initialization Dependency

#### 6.5.2 静态成员Static members

### 6.6 Namespace (看成一个类就行, , )

### 6.7 题目

## 第七章 继承 (Inheritance)

### 7.1 组合 (Composition)

### 7.2 Inheritance

### 7.3 题目

## 第八章 多态 (Polymorphism)

### 8.1 Virtual functions

### 8.2 虚函数表

### 8.3 应用

### 8.4 tips

### 8.5 Abstract class

### 8.6 Protocol classes

### 8.7 题目

## 第九章 拷贝构造 (Copy Constructor)

### 9.1 Copying

### 9.2 types of function parameters and return value

### 9.3 tips

### 9.4 Left Value vs Right Value

### 9.5 移动拷贝函数

### 9.6 对象初始化的形式

### 9.7 题目

## 第十章 运算符重载 (Overloaded operators )

### 10.1 Overloaded operators

### 10.2 C++ overloaded operator

### 10.3 How to overload

### 10.4 Copying vs. Initialization

### 10.5 User-defined Type conversions

### 10.6 题目

## 第十一章 模板 (Templates)

### 11.1 函数模板 (Function Templates)

### 11.2 模板实例化 (Template Instantiation)

### 11.3 类模板 (Class templates)

### 11.4 Templates and inheritance

### 11.5 题目

## 第十二章 异常 (Exceptions)

### 12.1 Standard library exceptions

# 第一章、基本介绍

---

## 1.1 C语言优缺点

优点：

- Efficient programs (程序高效)
- Direct access to machine, suitable for OS and ES (操作底层)
- Flexible (灵活)

缺点：

- Insufficient type checking (类型检查不够)
- Poor support for programming-in-the-large (难以支持大程序)
- Procedure-oriented programming (面向过程)

## 1.2 C++进步

- Data abstraction
- Access control
- Initialization & cleanup
- References
- Function overloading
- Streams for I/O
- Name control
- Operator overloading
- More safe and powerful memory management
- Templates
- Exception handling

## 1.3 字符串

- 字符串是个类 (加 `#include<string>`)
- `find` 函数

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
```

```

5  {
6      string str;
7      cin>>str;
8      //主要字符串是从0开始计数的
9      cout<<"ab在str中的位置:"<<str.find("ab")<<endl;
10     //返回第一次出现ab的位置,没有则返回一串乱码
11     cout<<"ab在str[2]到str[n-1]中的位置:"<<str.find("ab",2)<<endl;
12     //返回第一次从str[2]开始出现ab的位置,没有则返回一串乱码
13     cout<<"ab在str[0]到str[2]中的位置:"<<str.rfind("ab",2)<<endl;
14     //返回ab在str[0]到str[2]中的位置,没有则返回一串乱码
15     return 0;
16 }

```

- substr 函数

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string str;
7      cin>>str;
8      cout<<"返回str[3]及其以后的子串:"<<str.substr(3)<<endl;
9      //若小于限制长度则报错
10     cout<<"从ste[2]开始由四个字符组成的子串:"<<str.substr(2,4)<<endl;
11     //若小于限制长度则报错
12     return 0;
13 }

```

- replace 函数

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string line = "this@ is@ a test string!";
7      line = line.replace(line.find("@"), 1, ""); //从第一个@位置替换第一个@为空
8      cout<<line<<endl;
9      return 0;
10 }

```

- insert 函数

```

1  #include <string>
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6      string str;
7      cin>>str;
8      cout<<"从2号位置插入字符串jkl并形成新的字符串返回:"<<str.insert(2, "jkl")<<endl;
9      return 0;
10 }

```

- `append` 函数

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string str;
7      cin>>str;
8      cout<<"在字符串str后面添加字符串ABC:"<<str.append("ABC")<<endl;
9      return 0;
10 }

```

- `swap` 函数

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string str1,str2;
7      cin>>str1>>str2;
8      cout<<"str1:"<<str1<<endl;
9      cout<<"str2:"<<str2<<endl;
10     swap(str1, str2);
11     cout<<"str1:"<<str1<<endl;
12     cout<<"str2:"<<str2<<endl;
13 }

```

## 1.4 动态内存分配

- 为什么需要动态内存分配?

存在程序的内存需求只能在**运行时确定**的情况

- 特点:

1. C++中通过new关键字进行动态内存申请
2. C++中的动态内存分配是**基于类型**进行的

### 3. delete关键字用于内存释放

- new关键字与malloc函数的区别

new关键字是C++的一部分	malloc是由C库提供的函数
new以具体类型为单位进行内存分配	malloc以字节为单位进行内存分配
new在申请单个类型变量时可进行初始化	malloc不具备内存初始化的特性

例子见[https://blog.csdn.net/qg\\_40416052/article/details/82493916](https://blog.csdn.net/qg_40416052/article/details/82493916)

- Tips:
  - Don't use delete to free memory that new **didn't allocate**.
  - Don't use delete to free the **same** block of memory twice in succession.
  - Use delete [] if new [] was used to allocate an array.
  - Use delete (no brackets) if new was used to allocate a single entity.
  - It's **safe** to apply delete to the null pointer (nothing happens).
- delete 运算符的结果类型为 void，因此它不返回值
- 使用 delete 释放 C++ 类对象的内存时，将在释放该对象的内存之前调用该对象的析构函数（如果该对象具有析构函数）。
- 如果 delete 运算符的操作数是可修改的左值，则在删除该对象后未定义其值
- 对于不是类类型（class、struct 或 union）的对象，将调用全局 delete 运算符。对于类类型的对象，如果 delete 表达式以一元范围解析运算符 (::) 开始，则会在全局范围中解析解除分配函数的名称。否则，delete 运算符将在释放内存之前为对象调用析构函数（如果指针不为 null）。可为每个类定义 delete 运算符；如果给定类不存在这种定义，则会调用全局 delete 运算符。如果删除表达式用于释放其静态对象具有虚拟析构函数的类对象，则将通过对象的动态类型的虚拟析构函数解析释放函数。

## 1.5 题目

### ▶ 2-1 分数 2

( ) 不是面向对象程序设计的主要特征。

- ☐ A. 封装
- ☐ B. 继承
- ☐ C. 多态
- ☒ D. 结构

2-2 分数 2

关于C++与C语言关系的描述中，（）是错误的。

- ☐ A. C语言是C++语言的一个子集
- ☐ B. C语言与C++语言是兼容的
- ☐ C. C++语言对C语言进行了一些改进
- ☒ D. C++语言和C语言都是面向对象的

2-3 分数 2

下列关于cin和cout的说法中，错误的是\_\_\_\_\_。

- ☐ A. cin用于读入用户输入的数据
- ☐ B. cout用于输出数据
- ☒ C. cin比C语言中的scanf()函数更有优势，它可以读取空格
- ☐ D. cout通常与<<运算符结合

说反了

2-4 分数 2

关于delete运算符的下列描述中，（）是错误的。

- ☐ A. 它必须用于new返回的指针；
- ☐ B. 使用它删除对象时要调用析构函数；
- ☒ C. 对一个指针可以使用多次该运算符；
- ☐ D. 指针名前只有一对方括号符号，不管所删除数组的维数。

2-5 分数 2

以下程序中，new语句干了什么。

```
int** num;
```

```
num = new int* [20];
```

- ☐ A. 分配了长度为20的整数数组空间，并将首元素的指针返回。
- ☐ B. 分配了一个整数变量的空间，并将其初始化为20。
- ☒ C. 分配了长度为20的整数指针数组空间，并将num[0]的指针返回。
- ☐ D. 存在错误，编译不能通过。

2-6 分数 2

以下程序存在的问题是：

```
void fun()
{
    int *num1, *num2;
    num1 = new int[10];
    num2 = new int[20];
    num1[0] = 100;
    num2[0] = 300;
    num1 = num2;
    delete [] num1;
}
```

- ☐ A. num2不能给num1赋值
- ☒ B. num2最初指向的空间没有释放
- ☐ C. num1最初指向的空间没有释放
- ☐ D. 程序没有问题

答案错误: 0 分

num1指向其他地方，原来指向的空间没有释放

2-7 分数 2

作者 李志明 单位 燕山大学

关于new 和 delete 关键字功能的叙述，不正确的是（ ）

- ☐ A. C++程序的内存空间，可以分为代码区(text segment)、静态存储区(Data Segment)、栈区(Stack)、堆区(Heap)。new关键字用于从堆区中动态申请创建对象所需的内存空间。
- ☐ B. new动态申请内存空间成功后，返回该内存区域的首地址；同时，也会自动调用相关类的构造函数。
- ☐ C. delete用于删除new建立的对象，并释放指针所指向的内存空间，同时，也会自动调用对象的析构函数。
- ☒ D. B \* ptr=new B(5); delete ptr; 假设上述语句中，new申请的内存空间首地址为Addr,存放ptr指针变量值的内存空间首地址为 PAddr，则执行delete ptr 语句后，Addr、PAddr指向的内存区域均会被系统收回。



只收回PAddr的空间

2-8 分数 2

用new关键字动态申请一个三维数组，则下列语句正确的是（）

- ☐ A. float \*fp; fp= new float[10][25][10];
- ☒ B. float (\* fp)[25][10]; fp=new float[10][25][10];
- ☐ C. float (\* fp)[10]; fp=new float[10][25][10];
- ☐ D. float \*fp [25][10]; fp= new float[10][25][10];

二维指针数组表示三维数组

2-9 分数 2

下列语句中，不能连续输出3个值的是。

- ☐ A. cout<<x<<y<<z;
- ☒ B. cout<<x,y,z;
- ☐ C. cout<<x; cout<<y; cout<<z;
- ☐ D. cout<<(x,y,z)<<(x,y,z)<<(x,y,z);

2-10 分数 2

在C++中，cin是（）。

- ☐ A. 预定义的类
- ☐ B. 预定义的函数
- ☐ C. 一个标准的语句
- ☒ D. 预定义的对象

## 第二章、Class

---

## 2.1 引用

- Reference is a new way to manipulate objects in C++: 引用引入了对象的一个**同义词**。定义引用的表示方法与定义指针相似，只是用&代替了\*。引用（reference）是c++对c语言的**重要扩充**。引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。其格式为：**类型 &引用变量名 = 已定义过的变量名**。
- 引用的特点：
  1. 一个变量可取**多个**别名。
  2. 引用必须**初始化**。
  3. 引用只能在初始化的时候引用一次，**不能更改**为转而引用其他变量。
- Tips:
  1. &在这里不是求地址运算，而是起**标识**作用
  2. 对引用求地址，就是对目标变量求地址。即引用名是目标变量名的一个别名。引用在**定义上**是说引用**不占据任何内存空间**，但是编译器在一般将其实现为**const指针**，即指向位置不可变的指针，所以引用实际上与一般指针同样占用内存
  3. **不能建立引用的数组**。因为数组是一个由若干个元素所组成的集合，所以无法建立一个由引用组成的集合，但是可以建立**数组的引用**
  4. 引用常见的使用用途：作为函数的参数、函数的返回值
  5. Bindings don't change at run time（运行时不变），unlike pointers; Assignment changes the object referred-to
- Pointers vs. References  
References: can't be null（非空）

are dependent on an existing variable, they are an alias for an variable（依赖）

can't change to a new "address" location（不能改）

Pointers: can be set to null（可空）

pointer is independent of existing objects（独立）

can change to point to a different address（能改）
- 限制：没有引用的引用；没有指针的引用

```
1 int&* p; // illegal
2 void f(int& p); // 可以
```

## 2.2 类

### 2.2.1 :: resolver

::  
::

### 2.2.2 声明和定义

The class declaration, along with the prototype of the member functions should be put into a **header file** (声明和函数原型放在头文件)

The definition of the member functions should be put into **another source file** (定义放在另一个文件)

### 2.2.3 this指针

this 是 C++ 中的一个关键字，也是一个 **const 指针**，它指向**当前对象**，通过它可以访问当前对象的所有成员。

this 虽然用在类的内部，但是只有在对象被**创建以后**才会给 this 赋值，并且这个赋值的过程是编译器**自动完成**的，不需要用户干预，用户也**不能**显式地给 this 赋值 (const指针)

this 实际上是成员函数的一个**形参**，在调用成员函数时将对象的地址作为实参传递给 this。不过 this 这个形参是**隐式**的，它并不出现在代码中，而是在编译阶段由编译器默默地将它添加到参数列表中。

this is a natural local variable of all structs member functions that you can not define, but can use it directly. (不能定义，但可以直接使用，自然局部变量)

Inside member functions, you can use this as the pointer to the variable that calls the function. (函数变量的指针)

### 2.2.4 OOP 特性

1. Everything is an object.
2. A program is a bunch of objects telling each other what to do by sending messages.
3. Each object has its own memory made up of other objects.
4. Every object has a type.
5. All objects of a particular type can receive the same messages.

## 2.3 构造和析构

### 2.3.1 构造函数

- 构造函数是一种特殊的**成员函数**，不需要用户来调用，**定义**对象时被自动执行。
- 构造函数名字与**类名相同**，**无返回类型**。
- 如果我们没有定义构造函数，系统会为我们自动定义一个无参的默认构造函数的，它不对成员属性做任何操作，如果我们自己定义了构造函数，系统就不会为我们创建默认构造函数了。

```

1 struct Y {
2     float f;
3     int i;
4     Y(int a);
5 };
6 Y y1[] = { Y(1), Y(2), Y(3) }; // OK
7 Y y2[2] = { Y(1) }; Y y3[7]; // 错误
8 Y y4; // 错误

```

### 2.3.2 析构函数

- 析构函数执行对象的**清理**工作，当对象的生命周期结束的时候，会自动调用。析构函数的作用并不是删除对象，在对象撤销它所占用的内存之前，做一些清理的工作。清理之后，这部分内存就可以被系统回收再利用了。在设计这个类的时候，系统也会默认提供一个析构函数。在对象的生命周期结束的时候，程序就会自动执行析构函数来完成这些工作。同构造函数，用户自己定义，系统自动调用。

```

1 class Y {
2     public:
3     ~Y();
4 };

```

- 析构函数没有返回值，**没有参数**；
- 没有参数，所以**不能重载**，一个类仅有一个析构函数；
- 析构函数除了释放工作，还可以做一些用户希望它做的一些工作，比如输出一些信息
- 调用时间：当对象**超出作用域**时，编译器自动调用析构函数。例如：main函数里创建对象，在退出main函数之后，析构函数被调用。在main函数的外面创建的对象，这个对象的销毁是在我们退出程序之后。析构函数销毁对象的顺序与构建对象的顺序是**相反**的。因为对象的存储是在**栈**中的，栈的特性就是先进后出。

### 2.3.4 存储分配

- The compiler allocates all the storage for a scope at the opening brace of that scope. (在作用域的开始大括号处为该作用域分配所有存储空间)
- The constructor call doesn't happen until the sequence point where the object is defined

## 2.4 题目

2-1 分数 2

假定AA为一个类，a()为该类公有的函数成员，x为该类的一个对象，则访问x对象中函数成员a()的格式为（）

- ☐ A. x.a
- ☒ B. x.a()
- ☐ C. x->a()
- ☐ D. (\*x).a()

2-2 分数 2

下列关于类定义的说法中，正确的是

- ☒ A. 类定义中包括数据成员和函数成员的声明
- ☐ B. 类成员的缺省访问权限是保护的
- ☐ C. 数据成员必须被声明为私有的
- ☐ D. 成员函数只能在类体外进行定义


缺省=默认，所以是私有

2-3 分数 2

下列关于类和对象的叙述中，错误的是

- ☒ A. 一个类只能有一个对象
- ☐ B. 对象是类的具体实例
- ☐ C. 类是对某一类对象的抽象
- ☐ D. 类和对象的关系是一种数据类型与变量的关系

2-4 分数 2

Resolver  is used to:

- ☐ A. Define a member function outside class declaration
- ☐ B. Access a member of a namespace
- ☐ C. Access a static member of a class
- ☒ D. All of the others

2-5 分数 2

类的实例化是指（ ）。

- ☐ A. 定义类
- ☒ B. 定义对象
- ☐ C. 调用类的成员函数
- ☐ D. 访问对象的数据成员

2-6 分数 2

C++ 函数的声明和定义可以分开。函数声明不需要( )。

- ☐ A. 返回类型
- ☐ B. 函数名
- ☐ C. 参数表
- ☒ D. 函数体

2-7 分数 2

在面向对象的程序设计中，首先需要在问题域中识别出若干个（ ）

- ☐ A. 函数
- ☒ B. 类
- ☐ C. 文件
- ☐ D. 过程

## 第三章 类和对象

---

### 3.1 类的定义

#### 3.1.1 编译单元

- The compiler sees only one .cpp file, and generates .obj file
- The linker links all .obj into one executable file
- To provide information about functions in other .cpp files, use .h

#### 3.1.2 头文件

- If a function is declared in a header file, you must include the header file **everywhere**
- If a class is declared in a header file, you must include the header file everywhere the class is used and where class member functions are defined
- Header = interface  
The header is a **contract** between you and the user of your code.  
The compiler enforces the contract by requiring you to declare all structures and functions before they are used.
- 内容: extern variables; function prototypes; class/struct declaration
- 形式:

`#include`: include is to **insert** the included file into the .cpp file at where the `#include` statement is.

`include "xx.h"`: search in the **current directory** firstly, then the directories declared somewhere (优先找当前文件夹)

`include <xx.h>`: search in the **specified** directories

`include <xx>`: same as include <xx.h>

## 第四章、容器

---

### 4.1 STL (标准模板库)

#### 4.1.1 优点

- Reduce development time.– Data-structures already written and debugged.
- Code readability (可读性)
- Fit more meaningful stuff on one page (紧凑)
- Robustness– STL data structures grow automatically. (自动增长)
- Portable code; Maintainable code (可移植, 可维护)

- Easy

#### 4.1.2 模板库内容:

- A Pair class (pairs of anything, int/int, int/char, etc)
- Containers: Vector (expandable array); Deque (expandable array, expands at both ends); List (double-linked); Sets and Maps
- Basic Algorithms (sort, search, etc)
- All identifiers in library are in std namespace: using namespace std;

#### 4.1.3 vector

- It is able to **increase its internal capacity** as required: as more items are added, it simply makes enough room for them. (自动增长的内存)
- It keeps its own private count of how many items it is currently storing. Its size method returns the number of objects currently stored in it. (统计有多少项, `.size()` 函数返回个数)
- It maintains the order of items you insert into it. You can later retrieve them in the same order (顺序储存)

函数:

```
1 v.assign(2, 3); //将 2 个 3 赋值给 v
2 v.swap(v1); //交换两个容器的内容
3 v.push_back(4); //在末尾添加元素 4
4 v.pop_back(); //删除最后一个元素
5 v.front(); //返回第一元素
6 v.back(); //返回最末元素
7 v.clear(); //清空容器
8 v.empty(); //容器为空返回true, 否则返回 false
```

初始化方式:

```
1 vector<int> v(100); //预先定义
2 v[80]=1; // okay
3 v[200]=1; // bad
4
5 vector<int> v2; //自动增长
6 int i;
7 while (cin >> i)
8 v.push_back(i);
```

#### 4.1.4 list

- 用法和vector等容器基本相同, 但是内部结构区别很大

函数:



```

1 l.push_front(3); //在头部插入元素 3
2 l.push_back(3); //在末端插入元素 3
3 l.pop_front(); //删除第一元素
4 l.pop_back(); //删除最末元素
5 l.front(); //返回第一元素
6 l.back(); //返回最末元素
7 l.clear(); //清空容器
8 l.empty(); //容器为空返回 true, 否则返回 false

```

### 4.1.5 map

- Maps are collections that contain pairs of values. (一对值)
- Pairs consist of a key and a value. (键值与值的映射)
- Lookup works by supplying a key, and retrieving a value.

### 4.1.6 stl的使用

- `vector<Student> v1;`: 定义了一个存储 `Student` 类型对象的 `vector` 容器, 对象直接保存在容器
- `vector<Student&> v2;`: 定义了一个存储 `Student` 类型的引用的 `vector` 容器, 但是由于引用类型不能被拷贝和赋值, 所以这种写法不符合语法规范。
- `vector<Student*> v3;`: 定义了一个存储指向 `Student` 类型对象的指针的 `vector` 容器, 即对象的地址保存在容器中。

## 4.2 Iterators

- Declaring

```

1 list<int>::iterator li;
2 list<int> L;
3 li = L.begin(); //Front of container
4 li = L.end(); //Past the end

```

- 迭代器都可以进行 `++` 操作。反向迭代器和正向迭代器的区别在于：
  - 对正向迭代器进行 `++` 操作时, 迭代器会指向容器中的后一个元素;
  - 而对反向迭代器进行 `++` 操作时, 迭代器会指向容器中的前一个元素。
- 正向迭代器: 假设 `p` 是一个正向迭代器, 则 `p` 支持以下操作: `++p`, `p++`, `*p`。此外, 两个正向迭代器可以互相赋值, 还可以用 `==` 和 `!=` 运算符进行比较。
- 双向迭代器: 双向迭代器具有正向迭代器的全部功能。除此之外, 若 `p` 是一个双向迭代器, 则 `--p` 和 `p--` 都是有定义的。 `--p` 使得 `p` 朝和 `++p` 相反的方向移动。

## 4.3 Algorithms

- Advantages of foreach loop:
  - It eliminates the possibility of errors and makes the code more readable.(不会错)
  - Easy to implement (可读性)

Does not require pre-initialization of the iterator (不用初始化迭代器)

- Disadvantages of foreach loop:

Cannot directly access the corresponding element indices (不能直接访问元素)

Cannot traverse the elements in reverse order (无法逆序)

It doesn't allow the user to skip any element as it traverses over each one of them (无法跳过)

## 4.4 Typdefs

- Annoying to type long names, like:

```
1 typedef PB map<Name, list<PhoneNum> >;
2 PB phonebook;
3 PB::iterator finger;
```

Easy to change implementation.

## 4.5 Pitfalls

- Accessing an invalid

```
1 vector<int> v;
2 v[100]=1; // 不一定能取到
```

Solutions:

- use push\_back()
  - Preallocate with constructor. (预先分配空间)
  - Reallocate with reserve() (重新分配内存)
  - Check capacity()
- Use invalid iterator

```
1 list<int> L;
2
3 list<int>::iterator li;
4
5 li = L.begin();
6
7 L.erase(li);
8
9 ++li; // WRONG
10
11 // Use return value of erase to advance
12
13 li = L.erase(li); // RIGHT
```

## 4.6 题目

### ▶ 2-1 分数 2

若有下面的语句：

```
vector<int> v;  
for (int i = 0; i < 4; i++)  
    v.push_back(i + 1);  
cout << v.size() << endl;
```

则执行后程序的输出结果是

- ☐ A. 1
- ☐ B. 2
- ☐ C. 3
- ☒ D. 4

### 2-2 分数 2

设有定义 `vector<string> v(10);`

执行下列哪条语句时会调用构造函数？

- ☐ A. `v[0] += "abc";`
- ☐ B. `v[0] = "2018";`
- ☒ C. `v.push_back("ZUCC");`
- ☐ D. `cout << (v[1] == "def");`

当执行 `v.push_back("ZUCC");` 时，如果 vector 容器的大小不足以容纳新元素，则会动态分配更多的空间，并将新元素插入到容器的末尾。这种情况下，会调用字符串类的拷贝构造函数来创建一个临时字符串对象，然后将其插入 vector 中

### 2-3 分数 2

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

循环结束后, `std::map m` 中实际上只有一个元素, 即最后一个读入的字符串作为键, 并且它的值为9。因此输出的 `m.size()` 结果为1。

### 2-4 分数 2

下列关于STL的描述中, 错误的是。

- ☐ A. STL的内容从广义上讲分为容器、迭代器、算法三个主要部分
- ☐ B. STL的一个基本理念就是将数据和操作分离
- ☐ C. STL中的所有组件都由模板构成, 其元素可以是任意类型
- ☒ D. STL的容器、迭代器、算法是三个完全独立的部分, 彼此也无任何联系

### 2-5 分数 2

下列创建vector容器对象的方法中, 错误的是。

- ☐ A. `vector<int> v(10);`
- ☐ B. `vector<int> v(10, 1);`
- ☐ C. `vector<int> v{10, 1};`
- ☒ D. `vector<int> v = (10, 1);`

`vector<int> v(10, 1);` 的意思是创建了一个包含 10 个元素的整数向量 `v`, 并将每个元素初始化为 1

2-6 分数 2

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

2-7 分数 2

设void f1(int \* m, long & n); int a; long b; 则以下调用合法的是 ( ) 。

- ☐ A. f1(a, b);
- ☒ B. f1(&a, b);
- ☐ C. f1(a, &b);
- ☐ D. f1(&a, &b);

2-8 分数 2

下面程序片段,哪一个是正确的?

- ☐ A. int n=4; int &r=n\*3;
- ☐ B. int m=5; const int &r=m; r=6;
- ☐ C. int n=8; const int &p=n; int &q=p ;
- ☒ D. int n=8; int &p=n; const int q=p ;

引用初始化不能用表达式，const做右值左边一定也是const

2-9 分数 2

下面程序段 `int a=1,b=2; int &r=a; r=b; r=7; cout<<a<<endl;` 的输出结果是？

- ☐ A. 1
- ☐ B. 2
- ☒ C. 7
- ☐ D. 无法确定

2-10 分数 2

已知： `float b = 34.5;` ，则下列表示引用的方法中，正确的是（ ）。

- ☒ A. `float &x = b;`
- ☐ B. `float &y = 34.5;`
- ☐ C. `float &z;`
- ☐ D. `int &t = &b;`

能将引用绑定到无法修改的临时值或字面量上。

## 第五章、Functions

### 5.1 局部变量

Local variables are defined **inside a method**, have a scope limited to the method to which they belong.

- 如果一个局部变量和类的成员变量(字段)拥有相同的名字，这样会导致方法内部无法直接访问这个字段。这是因为，当编译器遇到这样的情况时，它会默认使用局部变量而不是类成员变量。这种现象被称为“名称遮掩 (Name Hiding) ”。

例如，假设我们定义了一个名为 `Foo` 的类，并在其中声明了一个整型成员变量 `value`：

```
1 class Foo {
2     public:
3         int value;    // 成员变量（字段）
4         void bar() {
5             int value = 42;    // 局部变量
6             // ...
7         }
8     };
```

在上面例子中，`bar()` 函数内部定义了一个局部变量 `value` 来存放值为42的整数，此时如果需要在该函数中访问 `value` 成员变量，就会产生名称遮掩，即编译器会自动**取局部变量**而非class的成员 `value`。

- Fields（成员变量），parameters, local variables比较：
  1. Fields：成员变量是定义在**类或结构体**中的变量，在**实例化**该类或结构体时会被**创建**。成员变量可以被整个类内的函数所使用，同时也可以被类的外部所访问，并且成员变量的值在整个**类的生存期**内都是存在的。
  2. Parameters：函数参数是在调用函数时传递给该函数的一组值。它们只在函数内部起作用，并且在函数**执行完成后就被销毁了**。
  3. Local Variables：局部变量是在函数内部定义的变量。与函数参数一样，它们也只在函数内部起作用，并且在函数结束时就会销毁。
  4. Fields are defined outside constructors and methods
  5. Formal parameters are defined in the header of a constructor or method. They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call（传参的时候由特定的构造函数初始化形参）

## 5.2 C++ access control

### 5.2.1 访问形式

通过使用访问修饰符来限制类成员变量或函数的可见性。C++ 共有三种形式的访问控制：`public`、`private` 和 `protected`。

- `Public`：公共的成员变量和函数可以被类的任何地方的代码所访问，包括**该类的对象、子类、其他非子类**等。
- `Private`：私有的成员变量和函数只能由包含它们的**类本身所访问**，其他任何类型的对象都无法直接访问。继承自该类的子类及其他任何类都无法访问这些成员。
- `Protected`：受保护的成员变量和函数和私有成员一样，不能被类外部的代码直接访问，但是可以被继承自该类的**子类所访问**。而其他非子类的代码仍然不能访问。
- **默认情况下**，C++ 类定义的成员变量和成员函数的访问级别为 `private`

### 5.2.2 Friends

可以通过友元函数或友元类(friends)来实现某些函数或类对于其他类的**私有/受保护成员的访问**。友元是一种特殊的访问修饰符，与 `public`、`private` 和 `protected` 不同。

```
1 class MyClass {
2     private:
3         int myVar;
4         // 友元类能够访问该类的私有成员
5         friend class MyFriendClass;
6         // 友元函数能够访问该类的私有成员
7         friend void setMyVar(MyClass &obj, int val);
8 };
9 void setMyVar(MyClass &obj, int val) {
10     obj.myVar = val;    // 在友元函数中设置私有成员变量的值
```

```

11 }
12 class MyFriendClass {
13 public:
14     void setMyVar(MyClass &obj, int val) {
15         obj.myVar = val;    // 从友元类中设置私有成员变量的值
16     }
17 };
18 int main() {
19     MyClass obj;
20     MyFriendClass frd;
21     frd.setMyVar(obj, 42);    // 在main函数中使用friend类设定值
22     return 0;
23 }

```

### 5.2.3 class vs. struct

- class defaults to private
- struct defaults to public.

## 5.3 Initialization

### 5.3.1 初始化列表

初始化列表是用于定义类构造函数时指定初始值的一种方式。初始化列表也可以用于结构体和联合体初始化

当对象创建时，成员变量的初始化被放在它们的申明中或者在构造函数内部。如果我们直接在构造函数中通过赋值语句来初始化成员变量，那么就会多出一个**额外的步骤**：首先调用默认构造函数来初始化这些成员变量，然后再用指定的表达式进行覆盖复制。

- pseudo-constructor calls for built-ins(用内置函数的伪构造函数调用)
- Order of initialization is order of declaration – Not the order in the list（类成员变量的初始化顺序始终按照它们在类中声明的顺序进行，而不是在构造函数初始化列表中指定的顺序）
- Destroyed in the reverse order.（销毁顺序**相反**）
- Initialization vs. assignment

```

1 | Student::Student(string s):name(s) {}//initialization

```

before constructor

```

1 | Student::Student(string s) {name=s;}//assignment

```

inside constructor

string must have a default constructor



### 5.3.2 Overloaded constructors

可以为一个类定义多个不同的构造函数，这些构造函数使用相同的名称但带有**不同的参数列表**。

- 可以为参数提供**默认值**。如果调用该函数时**省略了**默认参数，则将使用预定义的默认值来**自动初始化**参数。

下面是默认参数规则的几个要点：

1. **默认参数必须出现在函数声明中的非默认参数之后**，即所有有默认值的参数都在末尾，否则编译器会报错。
2. 函数调用时，如果有缺省参数，那么可以不传递该参数，并按照函数原型中所列出的默认值赋值，例如：`myFunction()`；
3. 可以直接跳过某个有默认值的参数并在调用时给定其后面的参数，例如：`myFunction(42)`；
4. 在声明函数时，可以指定参数的默认值。示例代码如下：

```
1 int myFunction(int x, int y=5, bool z=true)
2 int chico(int n, int m = 6, int j); //illegal
3 int result1 = myFunction(10);          //y被隐式设置为5, z被隐式设置为true
4 int result2 = myFunction(10,20,false); //y被设置为20, z被设置为false
```

### 5.3.3 Pitfall of default arguments

- The default arguments are declared in its prototype, not defined in its function head: Can not put default arguments in definition (默认参数只需要在函数原型或者函数声明中提供，而**不需要**在函数定义中再次列出)

```
1 // 非法：定义中不应该有默认参数
2 void myFunction(int a=0, int b) { ... }
3 // 非法：同时出现了函数声明和定义中的默认参数
4 void myFunction(int a=0, int b=5) {
5     ...
6 }
7 // 非法定义例子，超过了函数原型的作用域范围
8 void myFunction(int a, int b=5) {
9     // 函数原型/outside的默认b=5.
10    ...
11    void innerFunction(int x, int y=b); // error-不能使用外部函数的默认参数
12    ...
13 }
14
```

## 5.4 inline function

内联 (inline) 函数是一些较小的函数，通常包含少量语句并不需要执行长时间运算。其实现过程是，编译器会将“内联函数”的代码**插入**到程序中的每个调用处，而**不是实际上去创建该函数**，并在程序执行时动态调用它，从而节省了函数调用本身所需的时间

- Repeat inline keyword at declaration and definition.
- An inline function definition may not generate any code in .obj file.

- So you can put inline functions' bodies in **header file**. Then #include it where the function is needed.

Never be afraid of multi-definition of inline functions, Definitions of inline functions are just **declarations**.

- The compiler does not have to honor your request to make a function inline. It might decide the function is too large or notice that it calls itself (recursion is not allowed or indeed possible for inline functions), or the feature might not be implemented for your particular compiler (编译器自己判断是否内联, 递归或过大就不内联)

Any function you define inside a class declaration is automatically an inline (类里定义的函数自动变成内联函数)

- You can put the definition of an inline member function out of the class braces.

But the definition of the functions should be put before where they may be called. (编译器必须能够看到内联函数的函数体, 否则就无法展开该函数的实现代码)

## 5.5 题目

▶ 1-1 分数 2

在C++语言中引入内联函数 (inline function) 的主要目的是降低空间复杂度, 即缩短目标代码长度。

☐ T ☒ F

空间复杂度增加

1-2 分数 2

作者 李志明 单位 燕山大学

主程序调用内联函数 (inline) 时, 不发生控制转移, 无需保存和恢复环境变量等, 因此, 节省了系统开销。内联函数的声明以及最终的生效, 是由程序员决定的。

☐ T ☒ F

编译器决定

1-3 分数 2

int Sum (int a,int b=5,int c); 这个函数原型的声明没有什么不合适的地方。

☐ T ☒ F

默认参数必须出现在函数声明中的非默认参数之后

1-4 分数 2

两个以上的函数, 具有相同的函数名, 且形参的个数或形参的类型不同, 或返回的数据类型不同, 则称之为函数的重载。

☐ T ☒ F

《返回类型》

1-5 分数 2

带有默认值的函数F的函数原型为F(int x=5, int x=9, int y, float m=10.0), 则该函数在编译时会报错。答案为T, 分值为2分。

☒ T ☐ F

▶ 2-1 分数 2

如果默认参数的函数声明为“ void fun(int a,int b=1,char c='a',float d=3.2);”, 则下面调用写法正确的是 ( ) 。

- ☐ A. fun();
- ☒ B. fun(2,3);
- ☐ C. fun(2, 'c',3.14)
- ☐ D. fun(int a=1);

2-2 分数 2

重载函数在调用时选择的依据中, 错误的是 ( ) 。

- ☐ A. 函数的参数
- ☐ B. 参数的类型
- ☐ C. 函数的名字
- ☒ D. 函数的类型

2-3 分数 2

在 ( ) 情况下适宜采用inline定义内联函数。

- ☐ A. 函数体含有循环语句
- ☐ B. 函数体含有递归语句
- ☒ C. 函数代码少、频繁调用
- ☐ D. 函数代码多、不常调用

2-4 分数 2

在C++中，关于下列设置缺省参数值的描述中，（）是正确的。

- ☐ A. 不允许设置缺省参数值；
- ☒ B. 在指定了缺省值的参数右边，不能出现没有指定缺省值的参数；
- ☐ C. 只能在函数的定义性声明中指定参数的缺省值；
- ☐ D. 设置缺省参数值时，必须全部都设置；

2-5 分数 2

下面说法正确的是（）。

- ☐ A. 内联函数在运行时是将该函数的目标代码插入每个调用该函数的地方
- ☒ B. 内联函数在编译时是将该函数的目标代码插入每个调用该函数的地方
- ☐ C. 类的内联函数必须在类体内定义
- ☐ D. 类的内联函数必须在类体外通过加关键字inline定义

2-6 分数 2

对定义重载函数的下列要求中，（）是错误的。

- ☐ A. 要求参数的个数不同
- ☐ B. 要求参数中至少有一个类型不同
- ☒ C. 要求函数的返回值不同
- ☐ D. 要求参数个数相同时，参数类型不同

2-7 分数 2

当一个函数功能不太复杂，但要求被频繁调用时，选用\_\_\_\_。

- ☐ A. 重载函数
- ☒ B. 内联函数
- ☐ C. 递归函数
- ☐ D. 嵌套函数

2-8 分数 2

如有函数定义：void func(int x = 0, int y = 0){ .... }, 则下列函数调用中会出现问题的是\_\_\_\_\_。

- ☒ A. func(1,2, 3);
- ☐ B. func(1,2);
- ☐ C. func(1);
- ☐ D. func();

一眼丁真

2-9 分数 2

以下有关函数的叙述中正确的是（ ）。

- ☐ A. 函数必须返回一个值
- ☐ B. 函数体中必须有return语句
- ☒ C. 两个同名函数，参数表相同而返回值不同不算重载
- ☐ D. 函数执行中形参的改变会改变实参

2-11 分数 2

以下选项中，是正确的函数默认形参设置的是。

- ☐ A. int fun(int a,int b,int c);
- ☒ B. int fun(int a,int b,int c=1);
- ☐ C. int fun(int a,int b=1,int c);
- ☐ D. int fun(int a=1,int b,int c);

## 第六章、常量

### 6.1 const关键字

declares a variable to have a **constant** value

```

1  const int x = 123;
2  x = 27; // illegal!
3  x++; // illegal!
4  int y = x; // Ok, copy const to non-const
5  y = x; // Ok, same thing
6  const int z = y; // ok, const is safer

```

- Constants are variables, observe scoping rules (常量是一种变量，因此遵守作用域法则)
- the compiler tries to avoid creating storage for a const -- holds the value in its symbol table. (编译器不给常量空间，使用 `extern` 强制分配存储空间)
- 必须初始化，除非用 `extern`

Compile time constants

```

1  const int bufsize = 1024; // value must be initialized
2  extern const int bufsize;

```

### 6.1.1 Run-time constants

数组的长度必须是一个常量表达式(constant expression)，也就是说，在编译时可以确定的一个常量值。因此，在定义数组时，我们不能使用**运行时变量**来指定数组的长度。

```

1  const int class_size = 12;
2  int finalGrade[class_size]; // ok
3  int x;
4  cin >> x;
5  const int size = x;
6  double classAverage[size]; // error!

```

### 6.1.2 聚合 (Aggregates)

聚合 (Aggregate) 是指一些简单的数据类型的集合。聚合类型具有以下特性：

1. 是一个类或结构体；
  2. 没有用户自定义的构造函数、析构函数和赋值运算符重载；
  3. 所有成员都是 public 的；
  4. 没有虚函数；
- It's possible to use const for aggregates, but storage will be allocated. In these situations, const means a piece of storage that **cannot be changed**.

However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time (可以使用 `const` 关键字将聚合类型定义为常量，使其分配存储空间并不能够更改其值。但是在编译时，因为编译器无法知道其内容，所以不能将其用于数组大小或结构体成员的初始化参数等需要在编译时执行的地方。)

```

1 | const int i[] = { 1, 2, 3, 4 };
2 | float f[i[3]]; // illegal
3 | struct S { int i, j; };
4 | const S s[] = { { 1, 2 }, { 3, 4 } };
5 | double d[s[1].j]; // illegal

```

## 6.2 Pointers and const

指针 (Pointers) 和 `const` 常量 (Constants) 可以结合使用, `const` 关键字可以应用于指针本身或指针所指向的对象。具体来说, 如果是将 `const` 应用于**指针本身**, 则表示该指针变量**本身不能被更改**; 而如果是将 `const` 应用于指针所指向的**对象**, 则表示该**对象不能被修改**。

```

1 | char * const q = "abc"; // q is const
2 | *q = 'c'; // OK
3 | q++; // ERROR
4 | const char *p = "ABCD"; // (*p) is a const char
5 | *p = 'b'; // ERROR! (*p) is the const

```

	<code>int i;</code>	<code>const int ci = 3;</code>
<code>int *ip;</code>	<code>ip = &amp;i;</code>	<code>ip = &amp;ci; // ERROR</code>
<code>const int *cip</code>	<code>cip = &amp;i;</code>	<code>cip = &amp;ci;</code>

```

1 | *ip = 54; // always legal since ip points to int
2 | *cip = 54; // never legal since cip points to const int

```

### 6.2.1 String Literals

```

1 | char* s = "Hello, world!";

```

- This is actually a **`const char *s`** but compiler accepts it without the `const`
- If you want to change the string, put it in an **array**

### 6.2.2 Conversions

- Can always treat a non-const value as const
- cannot treat a constant object as non-constant without an explicit cast

```

1 | void f(const int* x);
2 | int a = 15;
3 | f(&a); // ok
4 | const int b = a;
5 | f(&b); // ok
6 | b = a + 1; // Error!

```

## 6.3 Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it's possible for the programmer to take it and modify the original value.
- In fact, whenever you're passing an address into a function, you should make it a const if at all possible

### 6.4 const object

#### 6.4.1 Const member functions

- Cannot modify their objects

```
1  int Date::set_day(int d){
2  //...error check d here...
3  day = d; // ok, non-const so can modify
4  }
5  int Date::get_day() const {
6  day++; //ERROR modifies data member
7  set_day(12); // ERROR calls non-const member
8  return day; // ok
9  }
```

- Const member function definition: Repeat the const keyword in the definition **as well as the declaration**

```
1  int get_day () const;
2  int get_day() const { return day };
```

#### 6.4.2 重载 const and none-const functions

```
1  void f() const;
2  void f();
```

#### 6.4.3 constants in classes

```
1  class HasArray {
2  const int size;
3  int array[size]; // ERROR!
4  };
5  class HasArray {
6  enum { size = 100 };
7  int array[size]; // OK!
8  };
9  class HasArray {
10 static const int size = 100;
11 int array[size];
12 }
```



## 6.5 static静态

Two basic meanings

- Static storage
- allocated once at a fixed address

Visibility of a name

internal linkage

type	meaning
static free function	internal linkage(deprecated)
static global variables	internal linkage(deprecated)
static local variables	persistent storage
static member variables	shared by instances
static member functions	static member functions shared by instances, can only access static members

- **静态局部函数** (static free function , 用 static 显式修饰函数) : 在全局命名空间下定义一个只能在**当前编译单元内使用**的静态函数。这种类型的函数拥有内部链接 (internal linkage) , 即只能在当前编译单元内使用, 不能被其他编译单元访问, 而且它**不作为类成员函数**使用。
- **静态全局变量** (static global variables internal linkage, 用 static 显式修饰变量) : 在函数内部定义的静态变量不会每次进入函数都创建一次, 只会在程序运行期间**初始化一次**, 并且其生命周期将持续整个程序执行期间。这些变量拥有内部链接 (internal linkage) , 意味着它们只能在**当前编译单元中使用**。
- **静态数据成员** (static member variables , 用 static 关键字声明类的成员变量) : 静态数据成员是针对类定义内的数据成员添加 static 修饰后得到的成员变量。所有该类对象**共享同一份静态数据成员**的副本, 即使该类没有被实例化或者有多前的实例化也始终如此。另外, 由于这种变量是属于类而不是属于某个对象的, 因此它们必须在类定义外部进行**初始化**, 而且可以通过 类名::静态成员变量名 来访问。
- **静态成员函数** (static member functions shared by instances, can only access static members, 用 static 关键字声明类的成员函数) : 静态成员函数是属于类本身而不是属于某个对象的。这种函数不能访问非静态的成员变量或函数, **只能访问被声明为 static 的数据成员**以及其他静态函数。

### 6.5.1 全局对象 (Global objects)

- Constructors are called **before main()** is entered
- Order controlled by appearance in file
- main() is no longer the first function called
- Destructors called when called `main()` `exit()`

## 6.5.2 Static Initialization Dependency

使用静态变量时存在 Static Initialization Dependency 问题，简称 SID。这种问题可能会导致程序运行期间出现未定义行为或崩溃等问题。

一般而言，编译器会根据一些规则确定文件静态变量的**初始化顺序**，以避免由于不良的初始化顺序造成程序运行期间的错误。其中一个关键点是：同一编译单元内的所有静态变量构造的顺序是已知的（即按定义的顺序构造），但是**不同编译单元之间的静态变量初始化顺序是未指定的**。也就是说两个不在同一个源文件的静态全局变量，其初始化到底哪一个先执行是没有明确规定的，具体顺序取决于编译器和链接器的实现细节及操作系统等其他因素。所以，当非本地静态对象（如全局/命名空间范围中声明的静态对象）在不同的文件之间存在依赖时，就会产生错误，这种依赖性也称为跨文件静态初始化顺序问题。

## 6.5.2 静态成员 Static members

Static member variables:

1. Global to all class member functions
  2. Initialized once, at file scope
  3. provide a place for this variable and init it in .cpp
  4. No static in .cpp
- Static member functions:
    - Have no implicit receiver `this` (不依赖对象)
    - Can access only static member variables(or other globals)

## 6.6 Namespace（看成一个类就行，，）

命名空间（Namespace）是一种将标识符分组的机制。它可以避免全局空间命名冲突，也可以避免不同库间进行命名冲突。

如果没有命名空间，当几个文件都定义了具有**相同名称**的变量或函数时就会产生**冲突**。通过使用命名空间，我们可以将它们分成各个独立的范围来解决这个问题。

作用：

- 表示类、函数、变量等的逻辑分组（Expresses a logical grouping of classes, functions, variables, etc.）
- 名称空间是一个类似于类的作用域（A namespace is a scope just like a class）
- 仅在需要名称封装时使用（Preferred when only name encapsulation is needed）

```
1 // old1.h
2 namespace old1 {
3 void f();
4 void g();
5 }
6 // old2.h
7 namespace old2 {
8 void f();
9 void g();
10 }
```

```

11
12 namespace A {
13     int a = 100;
14 }
15 namespace B {
16     int a = 200;
17 }
18 void test02()
19 {
20     //A::a a是属于A中
21     cout<<"A中a = "<<A::a<<endl;//100
22     cout<<"B中a = "<<B::a<<endl;//200
23 }

```

- 命名空间只能全局范围内定义（以下为**错误写法**）

```

1 void test02()
2 {
3     namespace A { //不能局部定义
4         int a = 100;
5     }
6     namespace B {
7         int a = 200;
8     }
9     //A::a a是属于A中
10    cout<<"A中a = "<<A::a<<endl;//100
11    cout<<"B中a = "<<B::a<<endl;//200
12 }

```

- 命名空间中的函数 可以在“命名空间”外定义

```

1 namespace A {
2     int a=100;//变量
3
4     void func();
5 }
6
7 void A::func()//成员函数 在外部定义的时候记得加作用域
8 {
9     //访问命名空间的数据不用加作用域
10    cout<<"func遍历a = "<<a<<endl;
11 }
12
13 void funb()//普通函数
14 {
15     cout<<"funb遍历a = "<<A::a<<endl;
16 }
17 void test06()
18 {
19     A::func();
20     funb();

```

- 无名命名空间，意味着命名空间中的标识符只能在本文件内访问，相当于给这个标识符加上了static，使得其可以作为内部连接（等于正常写变量和函数）

```

1 namespace{
2     int a = 10;
3     void func(){
4         cout<<"hello namespace"<<endl;
5     }
6 }
7 void test(){
8     //只能在当前源文件直接访问a 或 func
9     cout<<"a = "<<a<<endl;
10    func();
11 }

```

### Using的用法

- Introduces a local synonym for name（把命名空间里的东西用一个同名的东西代替）
- States in one place where a name comes from（在一个地方说明，其他地方不用再说）
- Eliminates redundant scope qualification（减少冗余的限定符书写）

```

1 void main() {
2     using MyLib::foo;
3     using MyLib::Cat;
4     foo();
5     Cat c;
6     c.Meow();
7 }

```

### Using-Directives

- Makes all names from a namespace available.
- Can be used as a notational convenience.

```

1 void main() {
2     using namespace std;
3     using namespace MyLib;
4     foo();
5     Cat c;
6     c.Meow();
7     cout << "hello" << endl;
8 }

```

### Ambiguities（Using-Directives可能出现歧义）

- Using-directives only make the names available.
- Ambiguities arise only when you make calls.（调用时出现歧义）

- Use scope resolution to resolve. (加: :)

```
1 // Mylib.h
2 namespace XLib {
3 void x();
4 void y();
5 }
6 namespace YLib {
7 void y();
8 void z();
9 }
10
11 void main() {
12 using namespace XLib;
13 using namespace YLib;
14 x(); // OK
15 y(); // Error: ambiguous
16 XLib::y(); // OK, resolves to XLib
17 z(); // OK
18 }
```

### Namespace aliases (别名)

- Namespace names that are too short may clash (太短可能冲突)
- names that are too long are hard to work with (太长不方便使用)
- Use aliasing to create workable names (用一个短的别名)
- Aliasing can be used to version libraries. (别名还可以用于对库进行版本控制。当更新库的新版本时，我们可以使用别名来引用特定的版本，从而确保代码与所需的库版本匹配。这样可以提供更多灵活性，并避免由于库的升级而导致的潜在问题。)

```
1 namespace supercalifragilistic {
2 void f();
3 }
4 namespace short = supercalifragilistic;
5 short::f();
```

### Namespace composition (类似于类的组合)

- Compose new namespaces using names from other ones.
- Using-declarations can resolve potential clashes. (组合的命名空间里相同的東西要聲明用哪個里面的，这个是必须的，而且应该是运行时出错<可能是动态链接的>，你如果不调用这个重复的东西什么都不会发生，但是调用了之后直接报错)
- Explicitly defined functions take precedence. (明确定义的函数会覆盖其他命名空间中相同名称的函数)

```

1 namespace first {
2     void x();
3     void y();
4 }
5 namespace second {
6     void y();
7     void z();
8 }
9 namespace mine {
10    using namespace first;
11    using namespace second;
12    using first::y(); // resolve clashes to first::x()
13    void mystuff();
14    // ...
15 }

```

Namespace selection (选择用哪些东西, 没什么可说)

```

1 namespace mine {
2     using orig::Cat; // use Cat class from orig
3     void x();
4     void y();
5 }

```

Namespaces are open (不同文件里定义相同的命名空间, 然后就十分智能地把它们里面的东西合一起了)

```

1 //header1.h
2 namespace X {
3     void f();
4 }
5 // header2.h
6 namespace X {
7     void g(); // X now has f() and g();
8 }

```

## 6.7 题目

▶ 1-1 分数 2

Order of initialization in the initial list is the order of their declaration in the list.

☐ T ☒ F

和列表的顺序无关, 要看声明顺序

1-2 分数 2

类的组合关系可以用“Has-A”描述；类间的继承与派生关系可以用“Is-A”描述。

☒ T ☐ F

1-3 分数 2

一个类的友元函数是这个类的成员。

☐ T ☒ F

2-9 分数 2

对于以下关于友元的说法

- ☐ A. 如果函数fun被声明为类A的友元函数，则该函数成为A的成员函数
- ☐ B. 如果函数fun被声明为类A的友元函数，则该函数能访问A的保护成员，但不能访问私有成员
- ☐ C. 如果函数fun被声明为类A的友元函数，则fun的形参类型不能是A。
- ☒ D. 以上答案都不对

2-10 分数 2

对于类之间的友元关系：

- ☐ A. 如果类A是类B的友元，则B的成员函数可以访问A的私有成员
- ☐ B. 如果类A是类B的友元，则B也是A的友元。
- ☐ C. 如果类A是类B的友元，并且类B是类C的友元，则类A也是类C的友元。
- ☒ D. 以上答案都不对。

---

2-11 分数 2

友元的作用是

- ☒ A. 提高程序的运用效率
- ☐ B. 加强类的封装性
- ☐ C. 实现数据的隐藏性
- ☐ D. 增加成员函数的种类

2-12 分数 2

下面关于友元的描述中，错误的是：

- ☐ A. 友元函数可以访问该类的私有数据成员
- ☐ B. 一个类的友元类中的成员函数都是这个类的友元函数
- ☐ C. 友元可以提高程序的运行效率
- ☒ D. 类与类之间的友元关系可以继承

友元关系不能继承

2-13 分数 2

已知类A是类B的友元，类B是类C的友元，则：

- ☐ A. 类A一定是类C的友元
- ☐ B. 类C一定是类A的友元
- ☐ C. 类C的成员函数可以访问类B的对象的所有成员
- ☒ D. 类A的成员函数可以访问类B的对象的所有成员

2-17 分数 2

Suppose a class is defined without any keywords such as public, private and protected, all members default to

- ☐ A. public
- ☐ B. protected
- ☒ C. private
- ☐ D. static

2-18 分数 2

Who can access a private member of a class?

- ☐ A. Only member functions of that class.
- ☒ B. Only member functions of that class and friend functions or member functions of friend classes
- ☐ C. Only member functions of that class and derived classes
- ☐ D. None of the others



可以访问私有成员的东西：成员函数、友元、友元类的成员函数

2-19 分数 2

静态成员函数没有：

- ☐ A. 返回值
- ☒ B. this指针
- ☐ C. 指针参数
- ☐ D. 返回类型

全局的，不属于某个对象，自然没有this

2-20 分数 2

For the code below:

```
class A {  
    static int i;  
    //...  
};
```

Which statement is NOT true?

- ☒ A. All objects of class A reserve a space for i
- ☐ B. All objects of class A share the space of i
- ☐ C. i is a member variable of class A
- ☐ D. i is allocated in global data space

所有类的对象共享一个静态变量

## 第七章 继承 (Inheritance)

### 7.1 组合 (Composition)

Composition: construct new object with existing objects (It is the relationship of **has-a**)

- Objects can be used to build up other objects
- 可以全组合，也可以组合一部分

```
1 class Person {  
2     public:  
3         Person(){};  
4         Person(string name, string address){
```

```

5         this->name=name;
6         this->address=address;
7     }
8     void set_name(string name){
9         this->name=name;
10    }
11    void set_address(string address){
12        this->name=address;
13    }
14    void print(){
15        cout<<name<<address<<endl;
16    }
17 private:
18     string name;
19     string address;
20 };
21 class Currency {
22 public:
23     void set_cents( int cents ){
24         num=cents;
25     }
26     void print(){
27         cout<<num<<endl;
28     }
29 private:
30     int num;
31 };
32 class SavingsAccount {
33 public:
34     SavingsAccount(
35         const string name,
36         const string address,
37         int cents );
38     ~SavingsAccount(){};
39     void print();
40 private:
41     Person m_saver;
42     Currency m_balance;
43 };
44
45
46 SavingsAccount::SavingsAccount (
47     const string name,
48     const string address,
49     int cents ) {
50     m_saver.set_name( name );
51     m_saver.set_address( address );
52     m_balance.set_cents( cents );
53 }
54 void SavingsAccount::print() {
55     m_saver.print();
56     m_balance.print();

```

## 7.2 Inheritance

Inheritance is to take the existing class, clone it, and then make additions and modifications to the clone (继承是获取现有的类，克隆后进行添加和修改。)

- Allows sharing of design for
  1. Member data
  2. Member functions
  3. Interfaces
- Advantages of inheritance
  1. Avoiding code duplication
  2. Code reuse (重用但不重复)
  3. Easier maintenance (维护性)
  4. Extendibility (扩展性)
- 继承的构造
  - Think of inherited traits as an embedded object
  - Base class is mentioned by class name

```

1  class Employee {
2  public:
3      Employee( const std::string& name, const std::string& ssn );
4          const std::string& get_name() const;
5      void print(std::ostream& out) const;
6      void print(std::ostream& out, const std::string& msg) const;
7  protected:
8      std::string m_name;
9      std::string m_ssn;
10 };
11 Employee::Employee( const string& name, const string& ssn )
12 : m_name(name), m_ssn( ssn) {
13 // initializer list sets up the values!
14 }
15 inline const std::string& Employee::get_name() const {
16     return m_name;
17 }
18 inline void Employee::print( std::ostream& out ) const {
19     out << m_name << endl;
20     out << m_ssn << endl;
21 }
22 inline void Employee::print(std::ostream& out, const std::string& msg) const {
23     out << msg << endl;
24     print(out);
25 }
26 //下面是寄存类

```

```

27 class Manager : public Employee {
28 public:
29     Manager(const std::string& name, const std::string& ssn, const std::string&
        title);
30     const std::string title_name() const;
31     const std::string& get_title() const;
32     void print(std::ostream& out) const;
33 private:
34     std::string m_title;
35 };
36 Manager::Manager( const string& name, const string& ssn, const string& title =
    "" )
37 :Employee(name, ssn), m_title( title ) {
38 }
39 inline void Manager::print( std::ostream& out ) const {
40     Employee::print( out ); // call the base class print
41     out << m_title << endl;
42 }
43 inline const std::string& Manager::get_title() const {
44     return m_title;
45 }
46 inline const std::string Manager::title_name() const {
47     return string( m_title + ": " + m_name ); // access base m_name
48 }

```

- More on constructors
  - Base class is always constructed first (基类先构造)
  - If no explicit arguments are passed to base class, Default constructor will be called
  - Destructors are called in exactly the reverse order of the constructors (析构顺序相反)

#### • Name Hiding

If you redefine a member function in the derived class, all other overloaded functions in the base class are inaccessible.(在继承类中重新定义基类的函数，基类的函数无法访问)

#### • What is not inherited

##### Constructors

- synthesized constructors use memberwise initialization (在合成的构造函数中，成员按逐个成员的方式进行初始化)
- In explicit copy ctor, explicitly call base-class copy ctor or the default ctor will be called instead. (在显式定义的拷贝构造函数中，我们可以明确调用基类的拷贝构造函数，确保正确复制派生类和基类的数据。)

##### Destructors

##### Assignment operation

- synthesized operator= uses memberwise assignment
- explicit operator= be sure to explicitly call the base class version of operator=

Private data is hidden, but still present (也继承了)

- Access protection
  - 默认继承方式是私有继承

Inheritance Type (B is)	public (不变)	protected (最多保护)	private (全为私有)
public A	public in B	protected in B	hidden
private A	private in B	private in B	hidden
protected A	protected in B	protected in B	hidden

- When is protected not protected?
  - When your derived classes are ill-behaved!
  - Protected is public to all derived classes (protected 是对所有派生类公开的)

因此

- make member functions protected (成员函数可以被自己的子类(派生类)直接访问,不被外部直接访问)
- keep member variables private
- Subclasses and subtyping
  - Objects of subclasses can be used where objects of supertypes are required (子类的对象可以用于需要父类的参数, “IS—A”, 需要一个动物, 给你一个狗, 这种做法叫 substitution)
  - Subclass object may be assigned to superclass pointer variables

```

1  vehicle *v1 = new vehicle();
2  vehicle *v2 = new car();
3  vehicle *v3 = new Bicycle();
4  public class Database
5  {
6      public void addItem(Item &theItem)
7      {
8          ...
9      }
10 }
11 DVD dvd = new DVD(...);
12 CD cd = new CD(...);
13 database.addItem(dvd);
14 database.addItem(cd);

```

- Up-casting (向上转型)
  - Is to regard an object of the derived class as an object of the base class (把子类看成基类)
  - Upcasting is the act of converting from a Derived reference or pointer to a base class reference or pointer. (将子类对象指针或引用赋值给父类指针或引用)

```

1  class Animal {

```

```

2 public:
3     virtual void makeSound() {
4         cout << "Animal is making a sound" << endl;
5     }
6 };
7 class Dog : public Animal {
8 public:
9     void makeSound() override {
10         cout << "Dog is barking" << endl;
11     }
12 };
13 int main() {
14     Dog dog;
15     Animal* animalPtr = &dog; // Up-casting, 将Dog对象指针赋值给Animal指针
16     animalPtr->makeSound();    // 调用的是Dog的重写函数
17 }

```

- Static type and dynamic type
  - 静态类型 (Static Type) : 指的是在编译时就确定下来的变量或表达式的类型
  - 动态类型 (Dynamic Type) : 指的是在运行时确定的变量或表达式的类型 (引出多态)

## 7.3 题目

1-1 分数 1

In C++, inheritance allows a derived class to directly access all of the functions and data of its base class.

☐ T ☒ F

答案正确: 1 分

一眼丁真

One class can have more than one super classes.

☒ T ☐ F

答案正确: 1 分

super class就是父类

```

1 #include <iostream>
2 #include <string>
3 using namespace std ;
4 class Testing
5 {
6 private:
7     string words;
8     int number ;
9 public:
10     Testing(const string & s = "Testing")
11     {

```

```

12     words = s ;
13     number = words.length();
14     if (words.compare("Testing")==0)
15         cout << 1;
16     else if (words.compare("Heap1")==0)
17         cout << 2;
18     else
19         cout << 3;
20 }
21 ~Testing()
22 {
23     cout << 0;
24 }
25 void show() const
26 {
27     cout << number;
28 }
29 };
30 int main()
31 {
32     Testing *pc1 , *pc2;
33     pc1 = new Testing ;           //1
34     pc2 = new Testing("Heap1"); //2
35     pc1->show(); //3
36     delete pc1 ; //4
37     delete pc2 ; //5
38     return 0;
39 }

```

the output at //1 is 1 (用默认的Testing) //2 is 2 //3 is 7 //4 is 0 //5 is 0

```

1 class A{
2     int i;
3 public:
4     A(int ii=0):i(ii) { cout << 1; }
5     A(const A& a) {
6         i = a.i;
7         cout << 2;
8     }
9     void print() const { cout << 3 << i; }
10 };
11 class B : public A {
12     int i;
13     A a;
14 public:
15     B(int ii = 0) : i(ii) { cout << 4; }
16     B(const B& b) {
17         i = b.i;
18         cout << 5;
19     }
20     void print() const {

```

```

21         A::print();
22         a.print();
23         cout << 6 << i;
24     }
25 };
26 int main()
27 {
28     B b(2);           //1
29     b.print();        //2
30     B c(b);           //3
31     c.print();        //4
32 }

```

the output at //1 is 114 (B类里有个A类对象，所以构造B的时候A构造两次，输出11)

//2 is 303062 //3 is 115 //4 is 303062

## 第八章 多态 (Polymorphism)

Upcast: take an object of the derived class as an object of the base one

Dynamic binding:

- Binding: which function to be called
  - Static binding: call the function as the code (编译时进行。根据变量或表达式的静态类型来确定应该调用哪个函数)
  - Dynamic binding: call the function of the object (运行时进行。根据变量或表达式的动态类型来确定要调用的函数)

### 8.1 Virtual functions

- Non-virtual functions
  - Compiler generates static, or direct call to stated type – Faster to execute
- Virtual functions
  - Can be transparently overridden in a derived class (在派生类里覆写)
  - Objects carry a pack of their virtual functions (产生一组虚函数)
  - Compiler checks pack and dynamically calls the right function (编译器在运行时动态调用)
  - If compiler knows the function at compile-time, it can generate a static call (编译时能确定就静态链接)
- Polymorphic variables
  - Pointers or reference variables of objects are polymorphic variables
  - They can hold objects of the declared type, or of subtypes of the declared type



## 8.2 虚函数表

- 虚函数表实现原理

虚函数的实现是由两个部分组成的，虚函数指针与虚函数表。

- 虚函数指针从本质上来说就只是一个指向函数的指针。它指向用户所定义的虚函数，具体是在子类里的实现，当**子类调用虚函数**的时候，实际上是通过调用该虚函数指针从而找到接口。

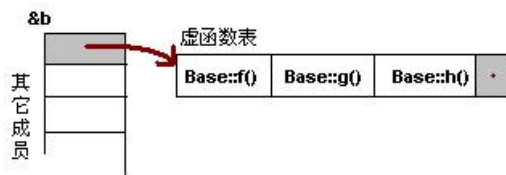
在一个被实例化的对象中，它总是被存放在该对象的**地址首位**，这种做法的目的是为了保证运行的**快速性**。与对象的成员不同，虚函数指针对外部是完全不可见的，除非通过直接访问地址的做法或者在DEBUG模式中，否则它是**不可见的也不能被外界调用**。

只有拥有虚函数的类才会拥有虚函数指针，每一个虚函数也都会对应一个虚函数指针。所以拥有虚函数的类的所有对象都会因为虚函数产生**额外的开销**，并且也会在一定程度上**降低程序速度**

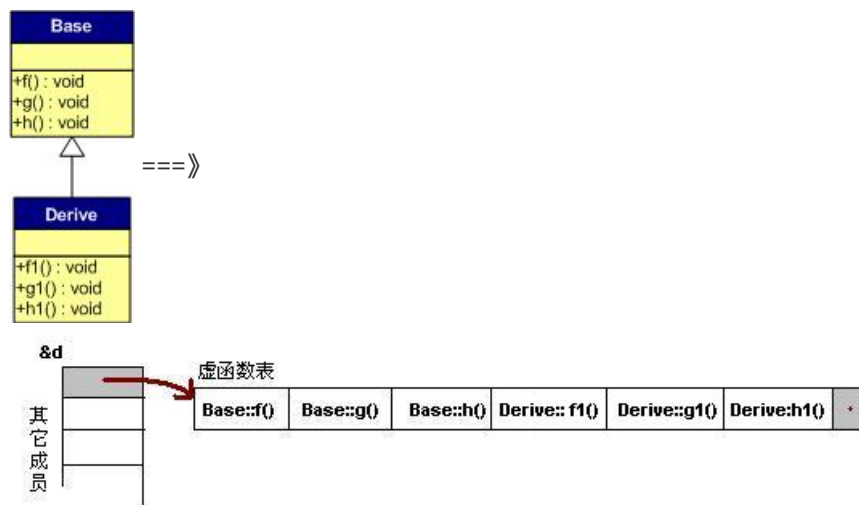
- 虚函数表 (Virtual Table)

```
1  class Base {  
2      public:  
3          virtual void f() { cout << "Base::f" << endl; }  
4          virtual void g() { cout << "Base::g" << endl; }  
5          virtual void h() { cout << "Base::h" << endl; }  
6  
7  };
```

对应的虚函数表：



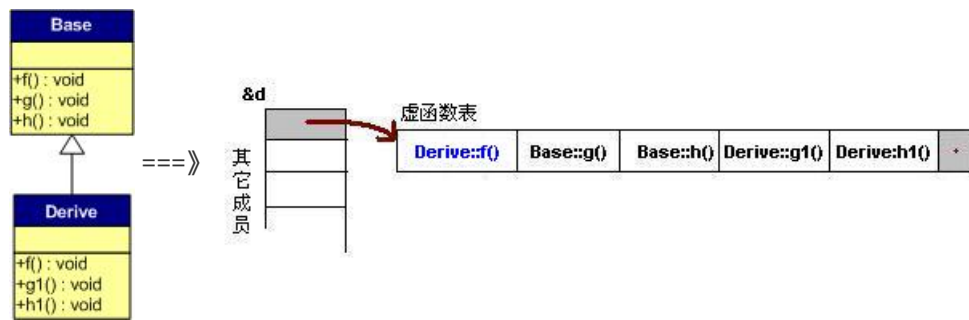
- 一般继承（无虚函数覆盖）



虚函数按照其**声明顺序**放于表中

**父类**的虚函数在子类的虚函数**前面**

- 一般继承（有虚函数覆盖）



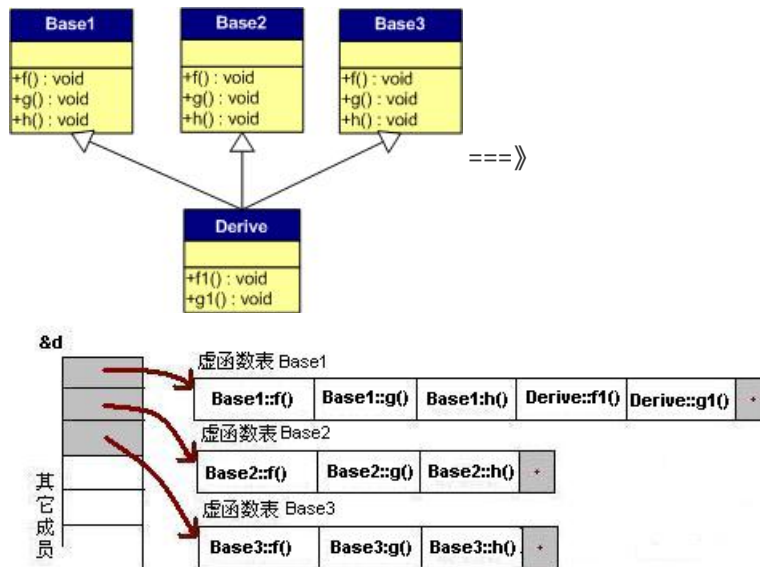
覆盖的f()函数**替换**了虚表中原来父类虚函数的位置。

没有被覆盖的函数**依旧**。

可以通过基类指针b调用子类的f()

```
1 Base *b = new Derive();
2 b->f();
```

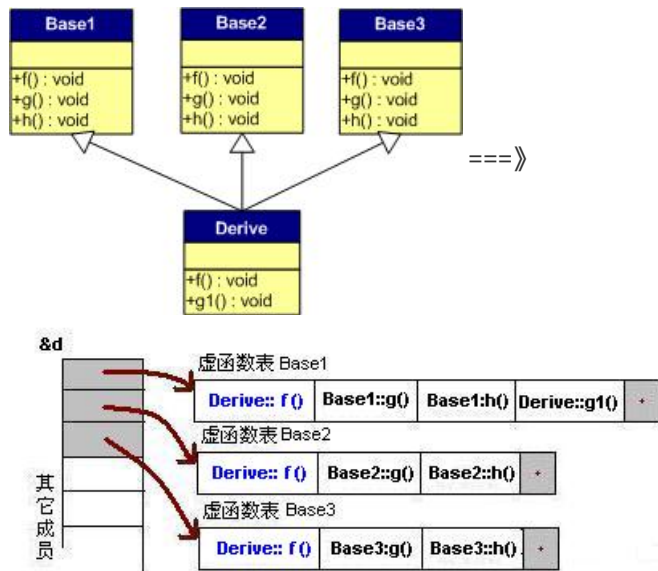
#### ■ 多重继承（无虚函数覆盖）



**每个**父类都有自己的虚函数表。

子类的成员函数被放到了**第一个**父类的表中（第一个父类是按照**声明顺序**来判断的）

#### ■ 多重继承（有虚函数覆盖）



父类虚函数表中的被覆函数的位置都被替换成了子类的函数指针。可以用任一静态类型的父类来指向子类，并调用子类的f()

```

1      Derive d;
2      Base1 *b1 = &d;
3      Base2 *b2 = &d;
4      Base3 *b3 = &d;
5      b1->f(); //Derive::f()
6      b2->f(); //Derive::f()
7      b3->f(); //Derive::f()
8      b1->g(); //Base1::g()
9      b2->g(); //Base2::g()
10     b3->g(); //Base3::g()

```

- ppt例子:

```

1  class Shape {
2  public:
3      Shape();
4      virtual ~Shape();
5      virtual void render();
6      void move(const int&);
7      virtual void resize();
8  protected:
9      int center;
10 };
11 class Ellipse : public Shape {
12 public:
13     Ellipse(float majr, float minr);
14     virtual void render();
15 protected:
16     float major_axis;
17     float minor_axis;

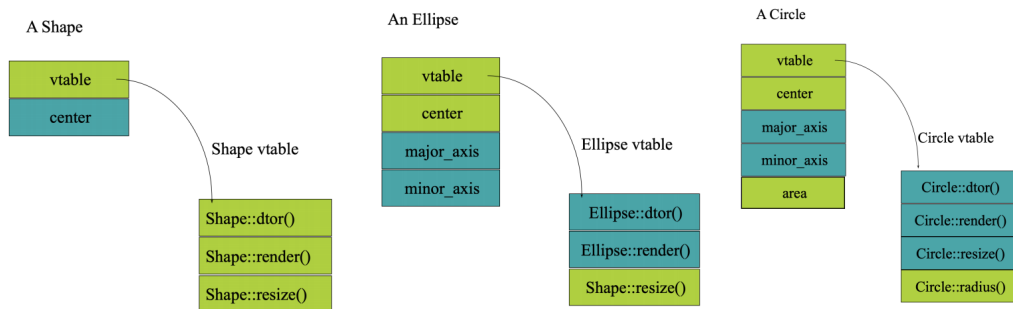
```

```

18 };
19 class Circle : public Ellipse {
20 public:
21   Circle(float radius);
22   virtual void render();
23   virtual void resize();
24   virtual float radius();
25 protected:
26   float area;
27 };

```

虚函数表变化:



What happens if

```

1 Ellipse elly(20F, 40F);
2 Circle circ(60F);
3 elly = circ; // 10 in 5?

```

- Area of circ is sliced off(Only the part of circ that fits in elly gets copied)
- Vtable from circ is **ignored**
- the vtable in elly is the Ellipse vtable ( `elly.render(); // Ellipse::render()` )

## 8.3 应用

- What happens with pointers?

```

1 Ellipse* elly = new Ellipse(20F, 40F);
2 Circle* circ = new Circle(60F);
3 elly = circ;

```

- the original Ellipse for elly is **lost**
- elly and circ point to the **same Circle object!** ( `elly->render(); // Circle::render()` )
- Virtuals and reference arguments

```

1 void func(Ellipse& elly)
2 {
3     elly.render();
4 }
5 Circle circ(60F);
6 func(circ);

```

- References act like pointers(类似传入指针)
- `Circle::render()` is called
- Virtual destructors (基本一般函数一样)

```

1 Shape *p = new Ellipse(100.0F, 200.0F); ...
2 delete p;

```

- Must declare `Shape::~~Shape()` virtual, If `Shape::~~Shape()` were not virtual, only `Shape::~~Shape()` will be invoked! (如果基类析构不是虚函数, 很显然就调用不了子类析构)
- It will call `Shape::~~Shape()` automatically (自动调用父类析构)
- Overriding

Overriding redefines the body of a virtual function (重构)

```

1 class Base {
2 public:
3     virtual void func();
4 }
5 class Derived : public Base {
6 public:
7     virtual void func(); //overrides Base::func()
8 }

```

- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass satisfies static type check.
- Subclass method is called at **runtime** – it overrides the superclass version.
- Return types relaxation

看下面例子就明白了, 只有指针引用可以

```

1  class Expr {
2  public:
3      virtual Expr* newExpr();
4      virtual Expr& clone();
5      virtual Expr self();
6  };
7  class BinaryExpr : public Expr {
8  public:
9      virtual BinaryExpr* newExpr(); // ok
10     virtual BinaryExpr& clone(); // ok
11     virtual BinaryExpr self(); // Error!
12 };

```

## 8.4 tips

- Never redefine an inherited non-virtual function
  - Non-virtuals are statically bound
  - No dynamic dispatch!
- Never redefine an inherited default parameter value
  - They're statically bound too!

## 8.5 Abstract class

- Some class is to create a common interface for all the classes derived from it. (有些类只作公共接口)
- An abstract method is incomplete. It has only a declaration and no method body. (纯虚函数: 只有声明, like `virtual void render() = 0;`)
- A class containing abstract methods is called an abstract class. (包含纯虚函数的是虚类)
  - Abstract base classes cannot be instantiated(需类不能实例化, 只能实例化派生类)
- 使用原因和时机: Modeling/interface
  - Force correct behavior
  - Define interface without defining an implementation
  - When designing for interface inheritance
- Mix and match
  - Members are duplicated
  - Derived class has access to full copies of each base class

```

1  class B1 { int m_i; };
2  class D1 : public B1 {};
3  class D2 : public B1 {};
4  class M : public D1, public D2 {};
5  void main() {
6      M m; //OK
7      B1* p = new M;

```

```

8      // 这里创建的是一个M对象，并将其赋值给一个B1类型的指针p。由于M同时继承自D1和D2，
9      // 基类B1在M中有两个副本，因此编译器不知道应该选择哪一个基类的指针给变量p。
10     // 编译器无法判断是要将p指向D1的部分还是D2的部分，因此会产生错误
11     B1* p2 = dynamic_cast<D1*>(new M); // OK
12     // 这里使用dynamic_cast进行类型转换，将M对象的指针转换为D1类型的指针p2。
13     // 因为M类继承自D1和D2，所以指针p2可以正确地指向M对象中的D1部分。
14     // 注意，使用dynamic_cast进行类型转换时，必须保证基类具有虚函数（或虚析构函数）。
15     // 此处dynamic_cast成功，因为M继承自D1，它是合法的。
16 }

```

- Replicated bases

Replication becomes a problem if replicated data makes for confusing logic:

```

1  M m;
2  m.m_i++; // ERROR: D1::B1.m_i or D2::B1.m_i?

```

## 8.6 Protocol classes

Abstract base class with

- All non-static member functions are pure virtual except destructor（全是纯虚函数或静态函数）
- Virtual destructor with empty body
- No non-static member variables, inherited or otherwise（只有静态的成员变量）

Using virtual base classes

- Virtual base classes are shared（虚基类被共享，而派生类只有一个虚基类的拷贝）
- Derived classes have a single copy of the virtual base
- Full control over sharing
  - Up to you to choose
- Cost is in complications

```

1  class B1 { int m_i; };
2  class D1 : virtual public B1 {};
3  class D2 : virtual public B1 {};
4  class M : public D1, public D2 {};
5  void main() {
6      M m; //OK
7      m.m_i++; // OK, there is only one B1 in m.
8      B1* p = new M; // OK
9  }

```

Complications of MI

- 名称冲突：当派生类从多个基类继承时，可能会出现名称冲突的情况。如果两个或多个基类具有相同的成员名称，派生类必须明确指定要使用的成员。
- 优先规则（Dominance Rule）：在多重继承中，如果多个基类提供了相同的成员函数，派生类在调用该函数时会根据某种规则选择要调用哪个基类的版本。

- 构造顺序：在存在多个基类的情况下，决定构造函数和析构函数的执行顺序可能变得复杂。程序员需要注意控制派生类和各个基类的构造顺序，以避免潜在的错误和未定义行为。
- 谁构造虚基类？：当一个派生类继承了拥有虚基类的多个子类时，问题出现在哪个派生类负责构造虚基类的实例。这由编译器自动处理，但可能会导致不直观或令人困惑的行为。
- 虚基类在需要时未声明：如果在派生类的构造函数中尝试访问虚基类的成员，而虚基类尚未被构造，将无法访问到虚基类的成员。
- 虚基类中的代码调用多次：当派生类通过多个路径继承相同的虚基类时，虚基类中的代码可能会被调用多次，导致执行逻辑上的重复操作。
- 编译器支持度各不相同：不同的编译器对多重继承的处理方式可能有所差异，一些编译器可能实现不完全或存在行为上的偏差。

Virtual bases 评价

- Use of virtual base imposes some **runtime and space** overhead.
- If replication isn't a problem then you don't need to make bases virtual (能重复就不需要)
- Abstract base classes (that hold no data except for a vptr) can be replicated with no problem - virtual base can be eliminated.

## 8.7 题目

虚函数是用virtual 关键字说明的成员函数。

☒ T ☐ F

将构造函数说明为纯虚函数是没有意义的。

☒ T ☐ F

没法构造基类了

抽象类是指一些没有说明对象的类。

☐ T ☒ F

动态绑定是在运行时选定调用的成员函数的。

☒ T ☐ F

因为静态成员函数不能是虚函数，所以它们不能实现多态。

☒ T ☐ F

共有的没法再重载了



在多继承中，派生类的构造函数需要依次调用其基类的构造函数，调用顺序取决于定义派生类时所指定的各基类的顺序。

☒ T ☐ F

虚函数具有继承性。

☒ T ☐ F

静态成员函数可以声明为虚函数。

☐ T ☒ F

1-10 分数 2

如果一个类的函数全部都是纯虚函数，则这个类不能有自己类的实现（包括引用和指针），只能通过派生类继承实现。

☐ T ☒ F

▶ 2-1 分数 2

Which one is the characteristic of abstract class?

- ☐ A. May have virtual functions
- ☐ B. May have constructors overloaded
- ☐ C. May have friend function
- ☒ D. Can not make instance of this class

选特性，其他不是特性？

2-2 分数 2

Given:

```
class A {  
    A() {};  
    virtual f() {};  
    int i;  
};
```

which statement is NOT true:

- ☐ A. i is private
- ☐ B. f() is an inline function
- ☐ C. i is a member of class A
- ☒ D. sizeof(A) == sizeof(int)

还记得虚指针吗

2-4 分数 2

Given:

```
class X {  
    int i;  
    virtual void f() {};  
};
```

If sizeof(int \*) == sizeof(int) == 4, then sizeof(X)==?

- ☐ A. 4
- ☐ B. 6
- ☒ C. 8
- ☐ D. Undetermined

如果一个类至少有一个纯虚函数，那么就称该类为（ ）。

- ☒ A. 抽象类
- ☐ B. 虚函数
- ☐ C. 派生类
- ☐ D. 具体类

假设A为抽象类，下列声明（）是正确的。

- ☐ A. A fun(int);
- ☒ B. A \*p;
- ☐ C. int fun(A);
- ☐ D. A Obj;

在C++中，可以声明指向抽象类的指针，因为指针本身并不需要知道具体派生类的信息，只要指针的类型能够匹配抽象基类即可

#### 2-8 分数 2

在创建派生类对象时，构造函数的执行顺序是（）。

- ☐ A. 对象成员构造函数、基类构造函数、派生类本身的构造函数
- ☒ B. 基类构造函数、对象成员构造函数、派生类本身的构造函数
- ☐ C. 基类构造函数、派生类本身的构造函数、对象成员构造函数
- ☐ D. 派生类本身的构造函数、基类构造函数、对象成员构造函数

---

#### 2-9 分数 1

派生类中的私有成员

若采用私有继承方式，则派生类对象中的私有成员不可能为\_\_\_\_\_。

- ☒ A. 基类中定义的私有成员
- ☐ B. 基类中定义的保护成员
- ☐ C. 基类中定义的公有成员
- ☐ D. 派生类中新增的私有成员

私有成员私有继承后无法直接访问，自然不是私有成员

以下说法正确的是？

- ☐ A. 在虚函数中不能使用this指针
- ☒ B. 在构造函数中调用虚函数，不是动态联编
- ☐ C. 抽象类的成员函数都是纯虚函数
- ☐ D. 构造函数和析构造函数都不能是虚函数

构造的时候没法动态联编知道调用什么函数

#### 5-1 B Fill in the blanks 分数 3

Run the following program, the output is: B::f()

```
#include <iostream>
using namespace std;
class A{
public:
    virtual void f() 1 分 { cout<<"A::f()\n"; }
};
class B:public A{
public:
    void f() {cout<<"B::f()\n"; }
};
int main()
{
    B b;
    A &p =b 1 分 ;
    p. 1 分 f();
    return 0;
}
```

## 第九章 拷贝构造 (Copy Constructor)

### 9.1 Copying

Create a new object from an existing one

- The copy constructor
  - Copying is implemented by the copy constructor (复制是由复制构造函数实现)
  - Has the unique signature

```
1 | T::T(const T&);
```

- Call-by-reference is used for the explicit argument (使用引用作为参数允许在函数内部修改传递的变量，并使这些修改对调用函数可见)
- C++ builds a copy ctor for you if you don't provide one (有默认的拷贝构造)

Copies each member variable

- Good for numbers, objects, arrays– Copies each **pointer**
- Data may become shared

ex:

```
1 | Person::Person( const char *s ) {  
2 |     name = new char[sizeof(s) + 1]; //The copy ctor initializes  
   | uninitialized memory (分配新内存)  
3 |     //Accesses s across client boundary (要有限权)  
4 |     ::strcpy(name, s); //No value returned  
5 | }  
6 | Person::~~Person() {  
7 |     delete [] name;  
8 | }
```

- 复制构造函数被调用的三种情况

- 当用一个对象去初始化同类的另一个对象时，会引发复制构造函数被调用。

```
1 | Complex c2(c1);  
2 | Complex c2 = c1; //这两条语句是等价的。  
3 | //注意，第二条语句是初始化语句，不是赋值语句。赋值语句的等号左边是一个早已有定义的变量，赋值语句不会引发复制构造函数的调用。  
4 | Complex c1, c2; c1 = c2 ;  
5 | c1=c2; //这条语句不会引发复制构造函数的调用，因为 c1 早已生成，已经初始化过了
```

- 如果函数 F 的参数是类 A 的对象，那么当 F 被调用时，类 A 的复制构造函数将被调用。换句话说，作为形参的对象，是用复制构造函数初始化的，而且调用复制构造函数时的参数，就是调用函数时所给的实参。

```

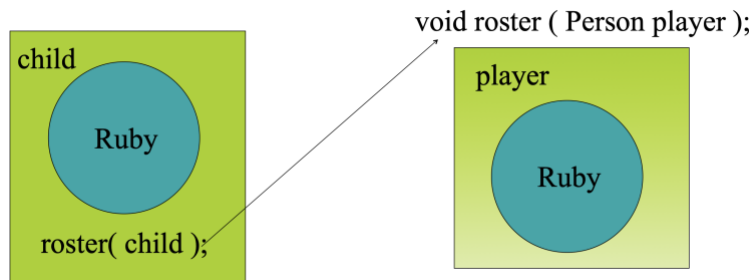
1  class A{
2  public:
3      A(){};
4      A(A & a){
5          cout<<"Copy constructor called"<<endl;
6      }
7  };
8  void Func(A a){ }
9  int main(){
10     A a;
11     Func(a);
12     return 0;
13 }//程序的输出结果为: Copy constructor called

```

```

1  void roster( Person ); // declare function
2  Person child( "Ruby" ); // create object
3  roster( child ); // call function

```

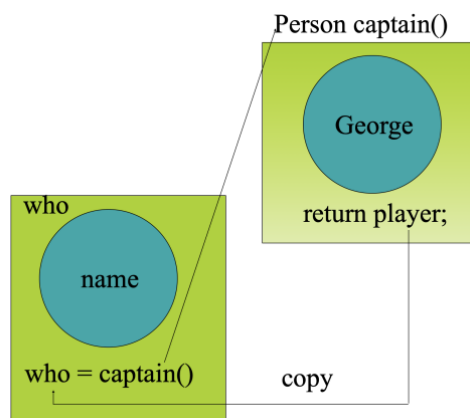


- 如果函数的返回值是类 A 的对象，则函数返回时，类 A 的复制构造函数被调用。

```

1  Person captain() {
2      Person player("George");
3      return player;
4  }
5  ...
6  Person who("");
7  ...

```



- Copies and overhead
  - Compilers can "optimize out" copies when safe! (没必要就别用)

Programmers need to

- Program for "dumb" compilers
- Be ready to look for optimizations

```

1 Person copy_func( char *who ) {
2     Person local( who );
3     local.print();
4     return local; // copy ctor called!
5 }
6 Person nocopy_func( char *who ) {
7     return Person( who );
8 } // no copy needed!

```

- Constructions vs. assignment
  - Every object is constructed once
  - Every object should be destroyed once (调不了delete或者delete多次不行)
  - Once an object is constructed, it can be the target of many assignment operations (对象被构造就可以对其进行多次赋值操作)
- What if the name was a string
  - In the default copy ctor, the compiler recursively calls the copy ctors for all member objects (and base classes)
  - default is memberwise initialization (默认情况下是按成员初始化进行)
- Copy ctor guidelines
  - In general, be explicit: Create your own copy ctor -- don't rely on the default
  - If you don't need one declare a private copy ctor (不需要拷贝构造函数, 则声明为私有)
    - prevents creation of a default copy constructor
    - generates a compiler error if try to pass-by-value - don't need a definition

## 9.2 types of function parameters and return value

传参类型

- a new object is to be created in f (要创建新对象)

```

1 void f(Student i);

```

- better with const if no intend to modify the object (要是不想修改就串指针)

```

1 void f(Student *p);

```

- better with const if no intend to modify the object (不想) 修改就加const

```
1 | void f(Student& i);
```

## 返回类型

- `Student f();`: 返回一个 `Student` 类型的对象。在函数体内部, 会创建一个新的 `Student` 对象, 并将其作为返回值返回给调用者。调用者会得到一个复制后的新对象, 它与原始对象互不影响的。
- `Student* f();`: 这个函数声明的返回类型是 `Student*`, 表明它将会返回一个指向 `Student` 类型对象的指针。在函数体内部, 会创建一个新的 `Student` 对象, 并在堆上分配内存。然后将该对象的指针作为返回值返回给调用者。调用者会得到指向堆上创建的对象指针, 可以通过该指针来访问和操作对象。
- `Student& f();`: 这个函数声明的返回类型是 `Student&`, 表明它将会返回一个指向 `Student` 类型对象的引用。在函数体内部, 会创建一个新的 `Student` 对象, 并将其引用作为返回值返回给调用者。调用者会得到对新创建对象的引用, 可以直接使用该引用进行访问和修改。

## 9.3 tips

很自然的选择:

- Pass in an object if you want to store it
- Pass in a const pointer or reference if you want to get the values
- Pass in a pointer or reference if you want to do something to it
- Pass out an object if you create it in the function
- Pass out pointer or reference of the passed in only
- Never new something and return the pointer (内存泄漏)

## 9.4 Left Value vs Right Value

可以简单地认为能出现在赋值号左边的都是左值:

1. 变量本身、引用
2. `*`、`[]` 运算的结果

只能出现在赋值号右边的都是右值:

1. 字面量
  2. 表达式
- 引用只能接受左值;  $\rightarrow$  引用是左值的别名
  - 调用函数时的传参相当于参数变量在调用时的初始化
  - 右值引用



```

1  int x=20; // 左值
2  int&& rx = x * 2; // x*2的结果是一个右值，rx延长其生命周期
3  int y = rx + 2; // 因此你可以重用它:42
4  rx = 100; // 一旦你初始化一个右值引用变量，该变量就成为了一个左值，可以被赋值
5  int&& rrx1 = x; // 非法:右值引用无法被左值初始化
6  const int&& rrx2 = x; // 非法:右值引用无法被左值初始化

```

- 右值参数

```

1  // 接收左值
2  void fun(int& lref) {
3      cout << "l-value" << endl;
4  }
5  // 接收右值
6  void fun(int&& rref) {
7      cout << "r-value" << endl;
8  }
9  int main() {
10     int x = 10;
11     fun(x); // output: l-value reference
12     fun(10); // output: r-value reference
13 }

```

- const引用参数 (也能接受右值)

```

1  void fun(const int& clref) {
2      cout << "l-value const reference\n";
3  }

```

## 9.5 移动拷贝函数

参数为右值引用的拷贝构造函数

```

1  vector<int> v1{1, 2, 3, 4};
2  vector<int> v2 = v1; // 此时调用复制构造函数，v2是v1的副本
   vector<int> v3 =
   std::move(v1); // 此时调用移动构造函数

```

## 9.6 对象初始化的形式

```

1  //小括号初始化
2  string str("hello");
3  //等号初始化
4  string str = "hello";
5  //大括号初始化
6  struct Studnet
7  {
8      char *name;
9      int age;
10 };

```

```

11 Studnet s = {"dablelv"
12 , 18}; //Plain of Data类型对象
13 Studnet sArr[] = {"dablelv"
14 , 18}, {"tommy"
15 , 19}}; //POD数组

```

- 列表初始化

```

1 class Test
2 {
3     int a;
4     int b;
5 public:
6     Test(int i, int j);
7 };
8 Test t{0, 0}; //C++11 only, 相当于 Test t(0,0);
9 Test *pT = new Test{1, 2}; //C++11 only, 相当于 Test* pT=new Test(1,2);
10 int *a = new int[3]{1, 2, 0}; //C++11 only

```

- 容器初始化

```

1 // C++11 container initializer
2 vector<string> vs={ "first", "second", "third"};
3 map<string,string> singers = { {"Lady Gaga", "+1 (212) 555-7890"}, {"Beyonce
Knowles", "+1 (212) 555-0987"}};

```

- `std::swap()`

```

1 void swap(T& a, T& b) {
2     T tmp{a}; // 调用用拷贝构造函数
3     a = b; // 拷贝赋值运算符
4     b = tmp; // 拷贝赋值运算符
5 }
6 void swap(T& a, T& b) {
7     T temp{std::move(a)};
8     a = std::move(b);
9     b = std::move(temp);
10 }

```

- Delegating Ctor

```

1 class class_c {
2 public:
3     int max;
4     int min;
5     int middle;
6     class_c(int my_max) {
7         max = my_max > 0 ? my_max : 10;
8     }
9     class_c(int my_max, int my_min) : class_c(my_max) {

```

```

10     min = my_min > 0 && my_min < max ? my_min : 1;
11 }
12 class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
13     middle = my_middle < max && my_middle > min ? my_middle : 5;
14 }
15 };
16 int main() {
17     class_c c1{ 1, 3, 2 };
18 }

```

## 9.7 题目

1-1 分数 2

当用一个对象去初始化同类的另一个对象时,要调用拷贝构造函数。

☒ T ☐ F

1-2 分数 2

对象间赋值将调用拷贝构造函数。

☐ T ☒ F

▶ 2-1 分数 2

下列各类函数中, 不是类的成员函数的是

- ☐ A. 构造函数
- ☐ B. 析构函数
- ☒ C. 友元函数
- ☐ D. 拷贝构造函数

2-2 分数 2

设类AA已定义, 假设以下语句全部合法, 哪些语句会触发调用拷贝构造函数 ( )。

```

AA a, b; //1
AA c(10, 20); //2
AA d(c); //3
AA e = d; //4

```

- ☐ A. 2
- ☐ B. 3
- ☐ C. 4
- ☒ D. 3 和 4

2-3 分数 2

假设MyClass是一个类，则该类的拷贝初始化构造函数的声明语句为（ ）

- ☐ A. MyClass&(MyClass x);
- ☐ B. MyClass(MyClass x);
- ☒ C. MyClass(MyClass &x);
- ☐ D. MyClass(MyClass \*x);

2-5 分数 2

下列哪一个说法是错误的？

- ☐ A. 当用一个对象去初始化同类的另一个对象时,要调用拷贝构造函数
- ☐ B. 如果某函数有一个参数是类A的对象,那么该函数被调用时,类A的拷贝构造函数将被调用
- ☐ C. 如果函数的返回值是类A的对象时，则函数返回时，类A的拷贝构造函数将被调用
- ☒ D. 拷贝构造函数必须自己编写

2-6 分数 2

假设A是一个类的名字,下面哪段程序不会用到A的拷贝构造函数？

- ☒ A. A a1,a2; a1=a2;
- ☐ B. void func( A a) { cout<<"good"<< endl; }
- ☐ C. A func() { A tmp; return tmp;}
- ☐ D. A a1; A a2(a1);

2-7 分数 2

如果某函数的返回值是个对象，则该函数被调用时，返回的对象？

- ☒ A. 是通过拷贝构造函数初始化的
- ☐ B. 是通过无参数的构造函数初始化的
- ☐ C. 用哪个构造函数初始化，取决于函数中return 语句是怎么写的
- ☐ D. 不需要初始化

程序填空比较简单(\*^▽^\*)

## 第十章 运算符重载 (Overloaded operators)

Allows user-defined types to act like built in types

Another way to make a **function call**

## 10.1 Overloaded operators

- operators **can be** overloaded (Unary and binary) :

+ - \* / % ^ & | ~ = < > += -= \*= /= %= ^= &= |= << >> >>= <=< ==

!= <= >= ! && || ++ -- , -> \* -> () []

operator new operator delete

operator new[] operator delete[]

- can't overload:**

. .\* :: ?:

sizeof typeid

static\_cast dynamic\_cast const\_cast

reinterpret\_cast

- Restrictions:
  - Only existing operators can be overloaded (只能重载现有运算符, 不能自创)
  - Operators must be overloaded on a class or enumeration type (必须在类或枚举类型上重载)
  - Preserve number of operands & Preserve precedence (操作数和优先级会保留)

## 10.2 C++ overloaded operator

- Just a function with an operator name (只是个函数)
  - Use the operator keyword as a prefix to name `operator * (...)`
- Can be a member function (可以是成员函数)
  - Implicit first argument (。 。 。 )

```
1 | const String String::operator +(const String& that);
```

- Can be a global (free) function (可以是正常的函数)
  - Both arguments explicit

```
1 | const String **operator+**(const String& r, const String& l);
```

## 10.3 How to overload

- As member function
  - Implicit first argument
  - No type conversion performed on receiver
  - Must have access to class definition

```

1  class Integer {
2  public:
3      Integer( int n = 0 ) : i(n) {}
4      const Integer operator+(const Integer& n) const{
5          return Integer(i + n.i);
6      }
7      ...
8  private:
9      int i;
10 };

```

- For binary operators (+, -, \*, etc) member functions require one argument. (二元需要一个参数)
- For unary operators (unary -, !, etc) member functions require no arguments (一元需要一个)

```

1  const Integer operator-() const {
2      return Integer(-i);
3  }
4  ...
5  z = -x; // z.operator=(x.operator-());

```

- As a **global** function
  - Explicit first argument
  - Type conversions performed on both arguments

```

1  const Integer operator+(
2      const Integer& rhs,
3      const Integer& lhs);
4  Integer x, y;
5  x + y ==> operator+(x, y);

```

-Can be made a **friend**

```

1  class Integer {
2      friend const Integer operator+ (
3          const Integer& lhs,
4          const Integer& rhs);
5      ...
6  }
7  const Integer operator+(
8      const Integer& lhs,
9      const Integer& rhs){
10     return Integer( lhs.i + rhs.i );
11 }

```

- binary operators require two arguments
- unary operators require one argument
- Tips: Members vs. Free Functions

- **Unary** operators **should be** members (通常在单个对象上操作, 并且不需要额外的参数)
- **= () [] -> \* must be** members
- assignment operators should be members (涉及访问或操作对象内部的成员)
- All **other binary** operators as **non-members**
- The prototypes of operators

- `- */%^&|~`

`- const T operatorX(const T& l, const T& r);`

- `!&& || < <= == >= >`

`- bool operatorX(const T& l, const T& r);`

```

1  bool Integer::operator==( const Integer& rhs ) const {
2      return i == rhs.i;
3  }
4  // implement lhs != rhs in terms of !(lhs == rhs)
5  bool Integer::operator!=( const Integer& rhs ) const {
6      return !(*this == rhs);
7  }
8  bool Integer::operator<( const Integer& rhs ) const {
9      return i < rhs.i;
10 }

```

- `[]`

`- E& T::operator[](int index);`

- operators `++` and `--`

```

1  const Integer& Integer::operator++() {
2      *this += 1; // increment
3      return *this; // fetch
4  }
5  // int argument not used so leave unnamed so
6  // won't get compiler warnings
7  const Integer Integer::operator++( int ){
8      Integer old( *this ); // fetch
9      ++(*this); // increment
10     return old; // return
11 }

```

- Defining a stream extractor / stream inserter
  - Has to be a 2-argument free function
    - First argument is an `istream&`
    - Second argument is a *reference* to a value

```

1  istream& operator>>(istream& is, T& obj) {
2      // specific code to read obj
3      return is;
4  }

```

```

1  ostream& operator<<(ostream& os, const T& obj) {
2      // specific code to write obj
3      return os;
4  }

```

## 10.4 Copying vs. Initialization

- Automatic operator= creation

The compiler will automatically create a `type::operator=(type)` if you don't make one.

- Assignment Operator
  - Must be a member function
  - Be sure to assign to all data members
  - Return a reference to **\*this**

```

1  T& T::operator=( const T& rhs ) {
2      // check for self assignment
3      if ( this != &rhs) {
4          // perform assignment
5      }
6      return *this;
7  }

```

- For classes with dynamically allocated memory declare an assignment operator (and a copy constructor)
- To prevent assignment, explicitly declare operator= as **private**

## 10.5 User-defined Type conversions

- A conversion operator can be used to convert an object of one class into
  - an object of another class
  - a built-in type
- Compilers perform implicit conversions using:
  - Single-argument constructors
  - implicit type conversion operators
- Preventing implicit conversions (只用定义过的显示转换)

`explicit` 关键字



```

1  class PathName {
2      string name;
3  public:
4      explicit PathName(const string&);
5      ~ PathName();
6  };
7  ...
8  string abc("abc");
9  PathName xyz(abc); // OK!
10 xyz = abc; // error!

```

- C++ type conversions

- Built-in conversions

Primitive: char ⇒ short ⇒ int ⇒ float ⇒ double ⇒ int ⇒ long

–Implicit (for any type T):

```

1  T⇔T&
2  T&→T
3  T*⇔void*
4  T→ const T
5  T[]⇔T*
6  T*⇔T[]

```

- Overloading and type conversion

- C++ checks each argument for a "best match"

- Best match means cheapest

–Exact match is cost-free

–Matches involving built-in conversions

–User-defined type conversions

- Overloading tips

- Just because you can overload an operator doesn't mean you should. (可以但不一定好)

- Overload operators when it makes the code easier to read and maintain. (可读性和可维持性)

- Don't overload && || or , (the comma operator)

## 10.6 题目

1-1 分数 1

多数运算符可以重载，个别运算符不能重载，运算符重载是通过函数定义实现的。

☒ T ☐ F

1-2 分数 1

对每个可重载的运算符来讲，它既可以重载为友元函数，又可以重载为成员函数，还可以重载为非成员函数。

☐ T ☒ F

C++规定有四个运算符 =, ->, [], ()不可以是全局域中的重载（即不能重载为友元函数），这几个太常用了？？

1-3 分数 1

对单目运算符重载为友元函数时，可以说明一个形参。而重载为成员函数时，不能显式说明形参。

☒ T ☐ F

1-4 分数 1

重载运算符可以保持原运算符的优先级和结合性不变。

☒ T ☐ F

1-5 分数 5

预定义的提取符和插入符是可以重载的。

☒ T ☐ F

1-6 分数 5

重载operator+时，返回值的类型应当与形参类型一致。  
比如以下程序中，operator+的返回值类型有错：

```
class A {  
    int x;  
  
public:  
    A(int t=0):x(t){ }  
    int operator+(const A& a1){ return x+a1.x; }  
};
```

☐ T ☒ F

1-7 分数 1

重载关系运算符一般都返回true或false值。

☒ T ☐ F

1-8 分数 1

The operator :: can not be overloaded.

☒ T ☐ F

1-9 分数 1

若重载为友元函数，函数定义格式如下：

```
<类型>operator<运算符> (<参数列表>)  
{  
    <函数体>  
}
```

☐ T ☒ F

没有friend？？

► 2-1 分数 2

下列运算符中，（ ）运算符不能重载。

- ☐ A. & &
- ☐ B. []
- ☒ C. ::
- ☐ D. <<

2-2 分数 2

下列关于运算符重载的描述中，（ ）是正确的。

- ☐ A. 运算符重载可以改变操作数的个数
- ☐ B. 运算符重载可以改变优先级
- ☐ C. 运算符重载可以改变结合性
- ☒ D. 运算符重载不可以改变语法结构

2-3 分数 2

为了能出现在赋值表达式的左右两边，重载的"[]"运算符应定义为：

- ☐ A. A operator [] (int);
- ☒ B. A& operator [] (int);
- ☐ C. const A operator [] (int);
- ☐ D. 以上答案都不对

2-4 分数 2

在C++中不能重载的运算符是

- ☒ A. ?:
- ☐ B. +
- ☐ C. -
- ☐ D. <=

2-5 分数 1

下列关于运算符重载的表述中，正确的是（ ）。

- ☐ A. C++已有的任何运算符都可以重载
- ☐ B. 运算符函数的返回类型不能声明为基本数据类型
- ☒ C. 在类型转换函数的定义中不需要声明返回类型
- ☐ D. 可以通过运算符重载来创建C++中原来没有的运算符

2-6 分数 2

能用友元函数重载的运算符是（）。

- ☒ A. +
- ☐ B. =
- ☐ C. []
- ☐ D. ->

2-8 分数 2

下列哪一项说法是不正确的？

- ☐ A. 运算符重载的实质是函数重载
- ☐ B. 运算符重载可以重载为普通函数,也成员可以重载为成员函数
- ☐ C. 运算符被多次重载时,根据实参的类型决定调用哪个运算符重载函数
- ☒ D. 运算符被多次重载时,根据函数类型决定调用哪个重载函数

2-9 分数 2

如何区分自增运算符重载的前置形式和后置形式？

- ☐ A. 重载时，前置形式的函数名是++operator，后置形式的函数名是operator ++
- ☒ B. 后置形式比前置形式多一个 int 类型的参数
- ☐ C. 无法区分，使用时不管前置形式还是后置形式，都调用相同的重载函数
- ☐ D. 前置形式比后置形式多一个 int 类型的参数

这个int就是为了区分

2-10 分数 2

下列关于运算符重载的描述正确的是（）。

- ☐ A. 运算符重载可以改变操作数的个数
- ☐ B. 可以创造新的运算符
- ☒ C. 运算符可以重载为友元函数
- ☐ D. 任意运算符都可以重载

2-11 分数 2

在重载一个运算符时，如果其参数表中有一个参数，则说明该运算符是（）。

- ☐ A. 一元成员运算符
- ☐ B. 二元成员运算符
- ☐ C. 一元友元运算符
- ☒ D. 二元成员运算符或一元友元运算符

2-12 分数 2

下列关于运算符重载的描述中，错误的是（ ）。

- ☐ A. 运算符重载不改变优先级
- ☒ B. 运算符重载后，原来运算符操作不可再用
- ☐ C. 运算符重载不改变结合性
- ☐ D. 运算符重载函数的参数个数与重载方式有关

2-13 分数 2

若需要为xv类重载乘法运算符,运算结果为xv类型,在将其声明为类的成员函数时,下列原型声明正确的是\_\_\_\_\_。

- ☐ A. xv\*(xv);
- ☐ B. operator\*(xv);
- ☒ C. xv operator\*(xv);
- ☐ D. xv operator\*(xv,xv);

2-14 分数 2

下列运算符中，不可以重载的是（ ）。

- ☐ A. new
- ☐ B. ++
- ☒ C. .\*
- ☐ D. []

## 第十一章 模板 (Templates)

- Why templates: 不同类型的相同对象  
Reuse source code （重用）  
-generic programming （泛型编程）  
-use types as parameters in class or function definitions （类型做参数）

### 11.1 函数模板 (Function Templates)

- Perform similar operations on different types of data. （都是这样）  
EX: swap function template

```
1  template < class T >
2  void swap( T& x, T& y ) {
3      T temp = x;
4      x = y;
5      y = temp;
6  }
```

- The `template` keyword introduces the template
- The `class T` specifies a parameterized type name  
class means any built-in type or user-defined type (class表示任何内置类型或自定义的类型)
- Inside the template, use T as a type name (模板中用T做类型名称)
- Function Template Syntax (语法)
  - Parameter types represent: (T可以做参数、返回类型、新定义的变量)
    - types of arguments to the function
    - return type of the function
    - declare variables within the function

## 11.2 模板实例化 (Template Instantiation)

Generating a declaration from a template class/function and template arguments:

- Types are substituted into template (传入的类型替换模板)
- New body of function or class definition is created (建立一个新函数体, 编译器会根据提供的参数将模板中的类型进行替换, 并生成相应的函数或类定义)
- Specialization -- a version of a template for a particular argument (基本变成正常函数了??)

```

1 // 模板通用实现
2 template<typename T>
3 void print(T value) {
4     std::cout << "Generic Print: " << value << std::endl;
5 }
6 // 字符串特化实现
7 template<>
8 void print<const char*>(const char* value) {
9     std::cout << "Specialized Print for const char*: " << value << std::endl;
10 }
```

- Interactions
  - Only *exact* match on types is used
  - No conversion operations are applied

```

1 swap(int, int); // ok
2 swap(double, double); // ok
3 swap(int, double); // error!
```

- Even *implicit* conversions are ignored (就是不能传double给int这样的)
- Overloading rules (优先普通函数, 其次在寻找模板函数)
  - Check first for unique function match
  - Then check for unique function template match
  - Then do overloading on functions

```

1 void f(float i, float k) {};
2 template <class T>
3 void f(T t, T u) {};
4 f(1.0, 2.0); //调void f(T t, T u) {}; (1.0不是float。。。)
5 f(1, 2); //调f(T t, T u) {};
6 f(1, 2.0); //f(float i, float k) {};

```

- The compiler deduces the template type from the actual arguments passed into the function (编译器从传递到函数中的实际参数中推导出模板类型)
- Can be explicit:  
 -for example, if the parameter is not in the function signature (older compilers won't allow this... 我的就不允许??)

```

1 template < class T >
2 void foo( void ) { /* ... */ }
3 foo<int>(); // type T is int
4 foo<float>(); // type T is float

```

## 11.3 类模板 (Class templates)

Classes parameterized by types

- Abstract operations from the types being operated upon (来自正在操作的类型的抽象操作)
- Define potentially infinite set of classes
- Another step towards reuse!

EX:

```

1 template <class T>
2 class Vector {
3 public:
4     Vector(int);
5     ~Vector();
6     Vector(const Vector&);
7     Vector& operator=(const Vector&);
8     T& operator[](int);
9 private:
10     T* m_elements;
11     int m_size;
12 };
13 Vector<int> v1(100);
14 Vector<Complex> v2(256);
15 v1[20] = 10;
16 v2[20] = v1[20]; // ok if int->Complex
17 // defined

```

- Templates can use **multiple** types

```

1 | template< class Key, class Value>
2 | class HashTable {
3 |     const Value& lookup(const Key&) const;
4 |     void install(const Key&, const Value&);
5 |     ...
6 | };

```

## 11.4 Templates and inheritance

- Templates can inherit from non-template classes （模板可以从非模板类中继承）

```

1 | template <class A>
2 | class Derived : public Base { ...}

```

- Templates can inherit from template classes （模板可以从模板类中继承）

```

1 | template <class A>
2 | class Derived : public List<A> { ...}

```

- Non-template classes can inherit from templates （非模板类可以从模板中继承）

```

1 | class SupervisorGroup : public
2 | List<Employee*> { ...}

```

## 11.5 题目

▶ 1-1 分数 1

pair类模板的作用是将两个数据组成一个数据，用来表示一个二元组或一个元素对，两个数据可以是同一个类型也可以是不同的类型。

☒ T ☐ F



► 2-1 分数 2

现有声明：

template

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;

特化模板，需要具体类型

2-2 分数 2

Given:

```
template < class T >
void swap( T& x, T& y ) {
    T temp = x;
    x = y;
    y = temp;
}
int i,j;
float f,m;
```

Which statement is incorrect?

- ☐ A. swap(i,j);
- ☐ B. swap(j,i);
- ☐ C. swap(f,m)
- ☒ D. swap(i,f);

2-3 分数 2

Given:

```
void f(int i) { cout << "Func1" << endl; }  
template<class T>  
void f(T t) { cout << "Func2" << endl; }  
main() {  
    f(2);  
}
```

The result is :

- ☒ A. Func1
- ☐ B. Func2
- ☐ C. *nothing*
- ☐ D. undetermined

2-4 分数 2

下列的模板说明中，正确的是。

- ☐ A. template < typename T1, T2 >
- ☐ B. template < class T1, T2 >
- ☒ C. template < typename T1, typename T2 >
- ☐ D. template ( typedef T1, typedef T2 )

两个得一致

2-5 分数 2

关于类模板，描述错误的是。

- ☒ A. 一个普通基类不能派生类模板
- ☐ B. 类模板可以从普通类派生，也可以从类模板派生
- ☐ C. 根据建立对象时的实际数据类型，编译器把类模板实例化为模板类
- ☐ D. 函数的类模板参数需生成模板类并通过构造函数实例化

2-6 分数 2

建立类模板对象的实例化过程为。

- ☐ A. 基类-派生类
- ☐ B. 构造函数-对象
- ☒ C. 模板类-对象
- ☐ D. 模板类-模板函数

2-7 分数 2

下列有关模板的描述，错误的是\_\_\_\_\_。

- ☐ A. 模板把数据类型作为一个设计参数，称为参数化程序设计
- ☐ B. 使用时，模板参数与函数参数相同，是按位置而不是名称对应的
- ☐ C. 模板参数表中可以有类型参数和非类型参数
- ☒ D. 类模板与模板类是同一个概念

2-8 分数 2

模板函数的真正代码是在哪个时期产生的\_\_\_\_\_。

- ☐ A. 源程序中声明函数时
- ☐ B. 源程序中定义函数时
- ☒ C. 源程序中调用函数时
- ☐ D. 运行执行函数时

2-9 分数 2

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

2-10 分数 2

声明模板的关键字为\_\_\_。

- ☐ A. static
- ☒ B. template
- ☐ C. typename
- ☐ D. class

2-11 分数 2

下列对模板的声明，正确的是\_\_\_。

- ☐ A. template<T>
- ☐ B. template<class T1, T2>
- ☒ C. template<class T1, class T2>
- ☐ D. template<class T1, class T1>

2-12 分数 2

下列选项中，哪一项是类模板实例化的时期\_\_\_。

- ☒ A. 在编译时期进行
- ☐ B. 属于动态联编
- ☐ C. 在运行时进行
- ☐ D. 在连接时进行

编译时就确定好类型了

2-13 分数 2

下列选项中，哪一个函数可以定义为对许多数据类型完成同一任务\_\_\_\_\_。

- ☐ A. 函数模板
- ☐ B. 递归函数
- ☒ C. 模板函数
- ☐ D. 重载函数

答案错误：0 分

???

2-14 分数 2

一个\_\_\_\_\_允许用户为类定义一种模式，使得类中的某些数据成员及某些成员函数的返回值能取任意类型。

- ☐ A. 函数模板
- ☐ B. 模板函数
- ☒ C. 类模板
- ☐ D. 模板类

模板来定义，函数来使用

2-15 分数 2

下列关于pair<>类模板的描述中，错误的是。

- ☐ A. pair<>类模板定义头文件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<>类模拟提供了两个成员函数first与second来访问这两个数据

pair<> 是 C++ 标准库中提供的一个类模板，用于表示包含两个值的有序对。它位于 <utility> 头文件中。

可以使用以下语法来创建和初始化 pair<> 对象：

```
1 std::pair<T1, T2> p; // 默认构造函数创建一个空的 pair 对象
2 std::pair<T1, T2> p(val1, val2); // 使用给定的 val1 和 val2 值初始化 pair 对象
```

其中，`T1` 和 `T2` 是实际类型参数，可以是任何合法的 C++ 类型。

以下是一些使用 `pair<>` 的示例：

```
1 #include <iostream>
2 #include <utility>
3 int main() {
4     // 创建并初始化 pair 对象
5     std::pair<int, std::string> p1(42, "hello");
6     // 获取 pair 对象的成员元素
7     int first = p1.first;
8     std::string second = p1.second;
9     // 修改 pair 对象的成员元素
10    p1.first = 24;
11    p1.second = "world";
12    // 输出 pair 对象的值
13    std::cout << "First: " << p1.first << std::endl;
14    std::cout << "Second: " << p1.second << std::endl;
15    return 0;
16 }
```

2-16 分数 2

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

假设声明了一下的函数模板：

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

并定义了int i; char c;

错误的调用语句是（ ）。

- ☐ A. max(i,i);
- ☐ B. max(c,c);
- ☐ C. max((int)c,i);
- ☒ D. max(i,c);

## 第十二章 异常 (Exceptions)

At the point where the problem occurs, you might not know what to do with it, but you do know that you can't just continue on merrily; you must stop, and somebody, somewhere, must figure out what to do.

- Why exception?
  - they clean up error handling code. (代码好看)
  - It separates the code that describes what you want to do from the code that is executed (描述代码和执行代码分开)
- How to raise an exception

```
1  template <class T>
2  T& vector<T>::operator[](int indx) {
3  if (indx < 0 || indx >= m_size) {
4      // throw is a keyword
5      // exception is raised at this point
6      throw <<something>>;
7  }
8      return m_elements[indx];
9  }
```

- What about your caller

**Case 1)** Doesn't care

-Code never even suspects a problem

```

1 | int func() {
2 |     vector<int> v(12);
3 |     v[3] = 5;
4 |     int i = v[42]; // out of range
5 |     // control never gets here!
6 |     return i * 5;
7 | }

```

**Case 2)** Cares deeply (分辨具体类型进行处理)

```

1 | void outer() {
2 |     try {
3 |         func(); func2();
4 |     } catch (VectorIndexError& e) {
5 |         e.diagnostic();
6 |         // This exception does not propagate
7 |     }
8 |     cout << "Control is here after
9 |     exception";
10 | }

```

**Case 3)** Mildly interested (单一处理)

```

1 | void outer2() {
2 |     String err("exception caught");
3 |     try {
4 |         func();
5 |     } catch (VectorIndexError) {
6 |         cout << err;
7 |         throw; // propagate the exception
8 |     }
9 | }

```

**Case 4)** Doesn't care about the particulars (全部)

```

1 | void outer3() {
2 |     try {
3 |         outer2();
4 |     } catch (...) {
5 |         // ... catches ALL exceptions!
6 |         cout << "The exception stops here!";
7 |     }
8 | }

```

- Throw statement **raises** the exception
  - Control propagates back to first handler for that exception (回到该异常的第一个处理程序)
  - Propagation follows the **call chain**
  - Objects on **stack** are properly destroyed



- Try blocks

```
1 try { ... }
2 catch ...
3 catch ...
```

Establishes any number of handlers

- Exception handlers

```
1 catch (SomeType v) { // handler code
2 }
3 catch (...) { // handler code
4 }
```

- Select exception by type (根据类型选择)
  - Can re-raise exceptions (可重复)
  - Take a single argument (单个参数)
- Selecting a handler

Handlers are checked in order of appearance:

1. Check for **exact match**
2. Apply **base class** conversions Reference and pointer types, only
3. Ellipses (...) match all

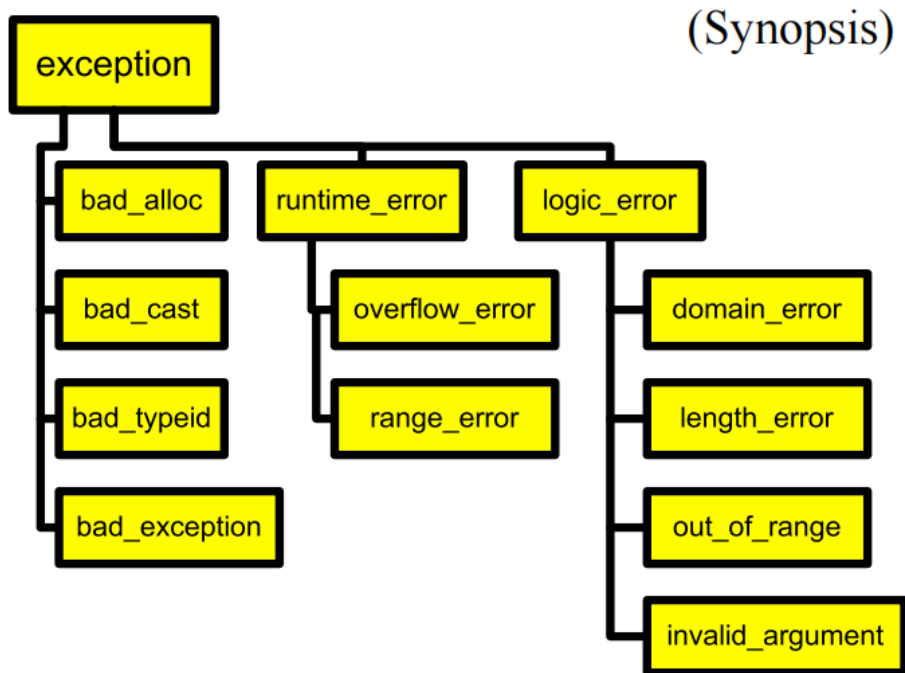
Inheritance can be used to structure exceptions (继承异常)

```
1 class MathErr {
2 ...
3 virtual void diagnostic();
4 };
5 class OverflowErr : public MathErr { ... }
6 class UnderflowErr : public MathErr { ... }
7 class ZeroDivideErr : public MathErr { ... }
8
9 try {
10 // code to exercise math options
11 throw UnderFlowErr();
12 } catch (ZeroDivideErr& e) {
13 // handle zero divide case
14 } catch (MathErr& e) {
15 // handle other math errors
16 } catch (...) {
17 // any other exceptions
18 }
```

- Exceptions and new

new raises a bad\_alloc() exception (new可能引发内存分配异常)

## 12.1 Standard library exceptions



- Exception specifications

Declare which exceptions function might raise (对函数可能引发的异常进行声明的一种方式，它是函数原型的一部分)

- Not checked at compile time (At **run time**)
- if an exception not in the list propagates out, the `unexpected` exception is raised (如果一个未在异常规格列表中的异常传播到函数外部，那么就会引发未知异常)

```
1 Printer::print(Document&) : throw(PrinterOffline, BadDocument)
2 PrintManager::print(Document&) : throw (BadDocument) { ...
3 // raises or doesn't handle BadDocument
4 void goodguy() : throw () {
5 // handles all exceptions
6 void average() { } // no spec, no checking,
```

## 12.2 Summary

- Exceptions provide the mechanism
  - Propagated dynamically (运行时动态地传播异常)
  - Objects on stack destroyed properly (确保栈上的对象被正确销毁)
  - Act to terminate the problematic function (终止出现问题的函数的执行)
- Another big use:
  - Constructors that can't complete their work

## 12.3 More exceptions

- Failure in constructors: Better to Throw an exception
  - Dtors for objects whose ctor didn't complete won't be called. (对象的析构函数将不会被调用。这是因为对象尚未完全构造成功, 所以析构函数的执行没有意义)。
  - Clean up allocated resources before throwing (适当的清理)
- Two stages construction
  1. Do normal work in ctor (始化所有成员对象、初始化所有基本类型成员和将指针初始化为 0)
    - Initialize all member objects
    - Initialize all primitive members
    - Initialize all pointers to 0
    - NEVER request any resource (不应该请求任何资源, 如文件、网络连接或内存等。通过避免在构造函数中进行可能引发异常或错误的操作)
  2. Do addition initialization work in Init() (资源请求更加安全)
- Exceptions and destructors
  - Throwing an exception in a destructor that is itself being called as the result of an exception will invoke `std::terminate()`. (在析构函数中抛出异常并且该析构函数本身是作为另一个异常的结果被调用时, 会触发 `std::terminate()` 函数, 导致程序的终止。)
  - Allowing exceptions to escape from destructors should be avoided. (在析构函数中抛出异常的同时, 堆栈上可能还有其他未处理的异常, 而这样的情况无法恢复或继续执行。因此, 应该尽量避免从析构函数中抛出异常)

## 12.4 Programming with exceptions

- Prefer catching exceptions by **reference** : Throwing/catching by value involves slicing

异常对象是按值进行传递的, 当我们以基类类型 `x` 进行捕获时, 实际上会发生对象切片。这意味着只会复制 `y` 对象的 `x` 部分, 而被称为切片

```
1 struct X {};  
2 struct Y : public X {};  
3 try {  
4     throw Y();  
5 } catch(X x) {  
6     // was it X or Y?  
7 }
```

- Throwing/catching by pointer introduces coupling between normal and handler code (正常代码和异常处理代码之间引入耦合关系)

```
1 try {  
2     throw new Y();  
3 } catch(Y* p) {  
4     // whoops, forgot to delete..  
5 }
```

- Catch exceptions by reference

```
1 struct B {
2     virtual void print() { /* ... */ }
3 };
4 struct D : public B { /* ... */ };
5 try {
6     throw D("D error");
7 }
8 catch(B& b) {
9     b.print() // print D's error.
10 }
```

- Exception Hierarchies

```
1 try {
2     ... throw SomethingElse();
3 }
4 catch(This& t) { /* ... */ }
5 catch(That& t) { /* ... */ }
6 catch(Other& t) { /* ... */ }
7
8 class B {};
9 class D1 : public B {};
10 class D2 : public B {};
11 ...
12 try {
13     ... throw D1();
14 }
15 catch(D2& t) { /* catch specific class here */ }
16 catch(B& t) { /* anything else here. */ }
```

- Unexpected exceptions (定义了函数可能抛出的异常类型)

```
1 void f() throw(x, y) { /* may throw X and Y */}
2 void g() throw() { /* throws no exceptions */}
3 void h() { /* may throw any exception*/}
```

如果函数 `f()` 抛出了与其异常规范不一致的异常类型，即抛出了除 `x` 和 `y` 之外的其他类型的异常，则会触发异常规范的违反（exception specification violation）。这将导致程序的终止并调用 `std::unexpected()` 函数，该函数默认会调用 `std::terminate()` 来终止程序。

对于函数 `g()`，在其异常规范为 `throw()`、即**不允许抛出任何异常**的情况下，如果函数 `g()` 确实抛出了异常，也会触发异常规范的违反。同样，这将导致程序的终止。

- Offers a guarantee (and firewall) to callers.
- `unexpected()` behavior can be intercepted

- Uncaught exceptions

- If an exception is thrown by not caught `std::terminate()` will be called (当异常在程序的调用链上未被处理 (即没有匹配的异常处理代码) 时, 就会引发未处理的异常)
- `terminate()` can also be intercepted.
- Exceptions wrapup
  - Develop an error-handling strategy early in design.
  - Avoid over-use of try/catch blocks. Use **objects** to acquire/release resources.
  - Don't use exceptions where local control structures will suffice (别添足)
  - Not every function can handle every error.
  - Use exception-specifications for major interfaces. (主要接口用异常规范)
  - Library code should not decide to terminate a program. Throw exceptions and let caller decide

## 12.5 题目

### ▶ 1-1 分数 1

If you are not interested in the contents of an exception object, the catch block parameter may be omitted.。

☒ T ☐ F

### catch (...)

#### 1-2 分数 1

作者 张德慧 单位 西安邮电大学

catch (type p) acts very much like a parameter in a function. Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.。

☒ T ☐ F

### 1-3 分数 1

异常处理的catch{ }语句块必须紧跟try{ }语句块之后, 这两个语句之间不能插入另外语句。

☒ T ☐ F

### 1-4 分数 1

有如下语句序列

```
第1行:    int a=1;
第2行:    try{
第3行:        if(a==1) throw(a);
第4行:        a++;
第5行:    }
第6行:    catch(int b){
第7行:        cout << "error! a = " << b << endl;
第8行:    }
```

以上语句的第6行有编译错误, 只能写成catch(int)。

☐ T ☒ F

► 2-1 分数 2

One of the major features in C++ is ( ) handling, which is a better way of handling errors.

- ☐ A. data
- ☐ B. pointer
- ☐ C. test
- ☒ D. exception

2-2 分数 2

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].

Segmentation fault! !

## 2-3 分数 2

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

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

int main()
{
    // Read two integers

    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

## 2-4 分数 2

The function what() is defined in \_\_\_\_\_.

- ☒ A. exception
- ☐ B. runtime\_error
- ☐ C. overflow\_error
- ☐ D. bad\_exception

`exception` 类提供了一个名为 `what()` 的虚成员函数，用于返回与异常相关的描述或消息的字符串。

派生类，如 `runtime_error`、`overflow_error` 和 `bad_exception`，都继承自 `exception` 类。它们可以重写 `what()` 函数，以提供自己特定的实现来返回有关错误的信息。

2-5 分数 2

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`.

2-6 分数 2

Suppose `Exception2` is derived from `Exception1`. Analyze the following code.

```
try {
```

```
    statement1;
```

```
    statement2;
```

```
    statement3;
```

```
}
```

```
catch (Exception1 ex1)
```

```
{
```

```
}
```

```
catch (Exception2 ex2)
```

```
{
```

```
}
```

- ☒ A. If an exception of the `Exception2` type occurs, this exception is caught by the first catch block.
- ☐ B. If an exception of the `Exception2` type occurs, this exception is caught by the second catch block.
- ☐ C. The program has a compile error because these two catch blocks are in wrong order.
- ☐ D. The program has a runtime error because these two catch blocks are in wrong order.

`Exception2` is derived from `Exception1` (没看题想半天)



2-7 分数 2

Suppose that statement2 throws an exception of type Exception2 in the following statement:

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

- ☐ A. statement2
- ☐ B. statement3
- ☐ C. statement4
- ☒ D. statement5

问题：statement2执行完后，下一条被执行的语句是什么？

2-9 分数 1

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。

在函数声明的形参表后没有任何表示异常类型的说明，那就表示该函数可以抛出任意类型的异常；`throw()`表示该函数不会抛出任何异常

2-10 分数 1

以下关于异常处理的描述错误的是（）。

- ☒ A. C++程序中出现异常时，编译器不会进行提示
- ☐ B. 将可能产生异常的代码放在try语句块内
- ☐ C. 使用catch关键字接收并处理异常
- ☐ D. 重抛异常可以在try语句块或者catch语句块中调用throw实现

答案错误: 0 分

重抛异常是指在异常处理块（catch）内部将当前捕获的异常重新抛出

2-11 分数 2

下列关于异常的描述中，错误的是（）。

- ☒ A. 编译错属于异常，可以抛出
- ☐ B. 运行错属于异常
- ☐ C. 硬件故障也可当异常抛出
- ☐ D. 只要是编程者认为是异常的都可当异常抛出

异常是运行时错误??

2-12 分数 2

作者 刘莹 单位 威海

下列关于异常类的说法中，错误的是。

- ☒ A. 异常类由标准库提供，不可以自定义
- ☐ B. C++的异常处理机制具有为抛出异常前构造的所有局部对象自动调用析构函数的能力
- ☐ C. 若catch块采用异常类对象接收异常信息，则在抛出异常时将通过拷贝构造函数进行对象复制，异常处理完后才将两个异常对象进行析构，释放资源
- ☐ D. 异常类对象抛出后，catch块会用类对象引用接收它以便执行相应的处理动作

2-13 分数 2

下列关于重抛异常的描述中，错误的是。

- ☐ A. 处理不了的异常，可以通过在catch结构中调用throw重新抛出异常，将当前异常传递到外部的try-catch结构中
- ☐ B. 重抛异常时只能从catch语句块或从catch块中的调用函数中完成
- ☒ C. 重抛的异常可以被同一个catch语句捕捉
- ☐ D. 可以单独使用throw关键字完成异常重抛

应该在更高的异常处理层级中寻找匹配的 catch 语句进行处理

2-14 分数 2

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

2-15 分数 2

C++处理异常的机制是由（）3部分组成。

- ☐ A. 编辑、编译和运行
- ☒ B. 检查、抛出和捕获
- ☐ C. 编辑、编译和捕获
- ☐ D. 检查、抛出和运行

程序填空：

```
1  #include <iostream>
2  using namespace std;
3  enum ERROR{UnderFlow,OverFlow};
4  template<typename T>
5  class StackTemplate {
6      enum { ssize = 100 };
7      T stack[ssize];
8      int top;
9  public:
10     StackTemplate() : top(0) {}
```

```

11     void push(const T& i) {
12         if (top >= ssize)
13             【throw OverFlow】;
14         stack[top++] = i;
15     }
16     T pop() {
17         if (top 【<=0】) throw UnderFlow;
18         return stack[--top];
19     }
20     int size() const
21     { return top; }
22 };
23 int fibonacci(int n);
24
25 int main() {
26
27     【try】{
28         【StackTemplate<int>】 is;
29         for(int i = 0; i < 20; i++)
30             is.push(fibonacci(i));
31         for(int k = 0; k < 20; k++)
32             cout << is.pop() << "\t";
33     }
34     catch( ERROR e ) {
35         switch( 【e】 )
36         {
37             case OverFlow:
38                 exit;
39             case UnderFlow:
40                 exit;
41         }
42     }
43     catch(...)
44     {
45         exit;
46     }
47     return 0;
48 }
49
50 int fibonacci(int n)
51 {
52     【const】 int sz = 100;
53     int i;
54     static int f[sz];
55     if (n >= sz) 【exit】;
56     f[0] = f[1] = 1;
57     for(i = 0; i < sz; i++)
58         if(f[i] == 0) break;
59     while(i <= n) {
60         【f[i】】 = f[i-1] + f[i-2];
61         i++;
62     }

```

```
63     return f[n];  
64 }
```