面向对象程序设计基础（OOP）

1. 绪论、基础编程知识

1.1 命令提示符（命令行）

1. 目录结构

```
【Win】C:\example\
【Linux】/home/username/example/
  /home/catherine 进入Linux桌面
  / 进入Linux入口
  /mnt/c 进入C盘（可逐步访问至desktop）
  /mnt/c/Users/Catherine/desktop
```

2. 显示当前目录

```
【Win】cd
【Linux】pwd
```

3. 在当前目录下新建目录
   `【Win/Linux】mkdir`
4. 在当前目录下新建文件

```
【Win】type nul>example.cpp
【Linux】touch example.cpp
```

5. 查看当前目录下的文件

```
【Win】dir
【Linux】ls
```

6. 进入上一层目录
   `【Win/Linux】cd ..`
7. 进入某目录
   `【Win/Linux】cd example`
8. 删除文件

```
【Win】del example.cpp
【Linux】rm example.cpp
```

9. 删除目录及目录下的所有文件

```
【Win】rmdir /s example
【Linux】rm -r example
```

10. 将文件移动到目录

```
【Win】move example.cpp example_folder\
【Linux】mv example.cpp example_folder/
```

11. 将文件拷贝到目录

```
【Win】copy example.cpp example_folder\
【Linux】cy example.cpp example_folder/
```

12. 将目录下的所有文件拷贝到另一目录

```
【Win】xcopy /e folder1 folder2
【Linux】cp -r folder1 folder2
```

13. 打开文件

```
【Win】type example
【Linux】vi example 或 vim example`
```

14. 其他
    sudo： 使用管理这身份发出命令
    ls -a： 显示隐藏目录
    sudo apt-get install：下载软件包
    cat example.cpp：显示文件内容

1.2 Windows Subsystem for Linux (WSL)

一个为在Windows 10上能够原生运行Linux二进制可执行文件的兼容层
[打开WSL的三种方式]

1. 点开Ubuntu
2. cmd输入wsl
3. cmd输入bash

1.3 主流编译器

【Windows】MinGW, MSVC(Visual Studio), TDM-GCC, Clang
【Linux/WSL】g++(配合gdb进行调试)

g++和**gcc**其实是“集成编译器”，会根据语言选择不同的编译器
GNU是一个组织
GCC = GNU Compiler Collection（可以粗糙地理解为C的编译器）
G++ = GNU C++ Compiler（可以粗糙地理解为C++的编译器）
GCC和G++的转换：`g++` = `gcc -xc++ -lstdc++ -shared-libgcc`

环境变量

我们希望在命令行使用g++指令编译程序，当前文件夹下没有g++程序。这时，系统除了在当前目录下面寻找程序外，还会到Path环境变量中的目录去找
【Windows】配置环境变量：此电脑-高级系统设置-高级-环境变量-用户环境变量Path-添加g++.exe所在路径(C:\MinGW\bin)
【Linux】配置环境变量
编辑ubuntu软件更新的源服务器的地址: /etc/apt/sources.list
配置环境变量：terminal中用bash打开Linux，`nano ~/.bashrc`查看
查看当前环境变量：`echo $PATH`

1.4 主流IDE

【IDE】DEV C++, Clion, Xcode, Visual Studio Code
VScode: Ctrl+ ~ 打开终端
【Editor】Sublime Text, Vim

1.5 SSH

SSH(Secure Shell)是建立在应用层基础上的安全协议，可以用其远程登录服务器/其他电脑
登录: `ssh remote_username@remote_address`
可行的远程登录访问: `ssh root@47.102.217.32`
结束远程控制: `exit`
从本地复制到服务器端 (local_file前加-r可以复制文件夹): `scp local_file remote_username@remote_address:remote_folder`
从服务器复制到本地rsa可以实现免密码登录: `scp remote_username@remote_address:remote_file local_folder`

1.6 源程序的结构、编译、链接

源程序 ==> 编译器(compiler) ==> 链接器(linker)
编译器: 生成目标模块（.o或.obj文件）
`g++ -c example.cpp` (Linux) 只编译不链接
链接器: 链接为可执行文件
`g++ -o example.out（也可以是.exe） example.o` (Linux) 链接程序 //-o后紧跟生成文件
`./example.out` (Linux) 可执行文件

1.7 多文件编译和链接过程

【Linux】
直接编译：`g++ a.cpp b.cpp -o test` （g++省略了一些步骤；甚至不需include头文件）
分步编译：（实际运行步骤）

```
g++ -c a.cpp -o a.o （只编译不链接）
g++ -c b.cpp -o b.o
g++ a.o b.o -o test
```

外部函数的**声明**只是令程序顺利通过编译，此时并不需要搜索到外部函数的**实现/定义**；在链接过程中，外部函数的**实现/定义**才会被寻找和添加进程序，一旦没有找到**函数实现**，就无法成功链接。

其他语句：

```
g++ -E main.cpp -o main.ii //预编译命令
g++ -S main.cpp -o main.s //转换成汇编语言文本文件
    //查看汇编代码：(web)Compiler Explorer https://gcc.godbolt.org/
    //可以在 'Compiler Options'中输入-g等语句（优化：-O0 -O1 -O2 -O3 数字越大越"激进"）
g++ -c main.cpp -o main.o //编译为二进制文件
g++ main.cpp -o main //链接
    //链接的过程其实还会链接其他的东西
    //extern "C" int printf(const char *__restrict __fmt, ...)
//其他编译选项：
g++ test.cpp --std=c++11 -fno-elide-constructors -o test
    //禁止编译器进行返回值优化
g++ test.cpp -g -o test -fsanitize=undefined,address
    //查看非法访问、溢出、内存外泄的强大命令
```

使用头文件

头文件里只能进行**函数声明**而不应进行**函数定义**，因为如果有多个cpp文件包含此头文件，在链接时会因为**重复定义**而发生错误。

声明与定义

同一个函数可以有多次声明，但只能有一次实现
变量也可以有声明和定义（`int x;`是定义）
变量的声明：**extern**关键字
`extern int x; //声明变量`
`//也可以用于函数声明，但不是必须的`
定义：定义是在变量声明后，给它分配上内存（即：定义 = 声明 + 内存分配）
（但事实上，变量定义时是否分配内存与编译器的优化行为有关。）

1.8 宏定义

`#define`是C++语言中的一个**预编译指令**，它用来将一个标识符定义为一个字符串，该标识符被称为**宏名**，被定义的字符串称为**替换文本**

宏替换

宏替换：`#define <宏名> <字符串>`
一般使用：`const double PI = 3.1415926`
`cpp example.cpp`（Linux）可以查看宏替换后的代码——预编译指令

宏定义

带参数的宏定义：`#define <宏名>(<参数表>)<字符串>` (example: `#define sqrt(x) ((x) * (x))`
一般使用**内联函数**：`inline double sqrt(double x) {return x * x};`
(`inline`是避免重复定义的好方法)
也可以使用`inline`修饰变量：`inline int const VAR = 123`
(对比`static int const VAR = 123`，会导致两个文件里调用的全局变量地址不一样)

宏定义的使用

1. 防止头文件被重复包含（原因：`#include`的本质是拷贝）
   方法一：header guards

   ```
   #ifndef __BODYDEF_H__
   #define __BODYDEF_H__
    //头文件内容
   #else
    //如果__BODYDEF_H__已经被定义则执行此处语句块
   #endif
   ```

   方法二： #pragma once

   ```
   #pragma once
   //头文件内容
   //用户自定义类型的情况必须使用pragma once或者header guards，即使成功避免在头文件中定义变量和函数也没用
   ```

   头文件中即使正确使用header guards或pragma once，包含函数或变量的定义，也有可能会出现重复定义的问题
2. 用于Debug输出等

   ```
   #ifdef 标识符
       程序段1
   #else
       程序段2
   #endif
   ```

3. 注意事项
   "#"开头的语句（包括宏定义`#define`)不会被名空间限制住。（例如：`namespace a{#define PI 3.14};`则在其他名空间也可以调用PI）

1.9 编写Make工具的脚本程序

Makefile编写规则

格式：：
　　[tab]
Makefile（建立在同一路径目录下）文件内容：

```
all: main test
main: main.cpp student.cpp
    g++ -o main main.o student.o
    //其他可能的语句: g++ -c src/main.cpp -I include -o main.o
    //其中"include"是一个文件夹，里面含有.h文件
main.o: src/main.cpp include/print.hpp
    g++ -c src/main.cpp -I include -o main.o
student.o: student.cpp
    g++ -c student.cpp -o student.o
debug:
    g++ main.cpp product.cpp sum.cpp -o main -DDEBUG
    //或: make -e DEBUG=-DDBUG
    //-D 表示命令后写入定义
    //这里定义了DEBUG对应宏定义中的Debug输出
    //sum.cpp
    //#ifdef DEBUG {}
    //#endif
clean:
    rm main test
rebuild: clean main

(Linux)
//make: 相当于make all
//make 任务名: (Example) make clean, make test
//make -f makefile的文件名
//make -f makefile的文件名 任务名
(Windows)
//把make改成mingw32-make就可以了（在VScodeTerminal或者cmd下，不需要包含路径因为已经在环境变量里了）
//把clean中的rm改成del
```

其他高级语法:

Image unavailable in the original source: note1.png

若在IDE中需要在task.json中配置环境:增加make程序，名称为build

```
//tasks.json
{
      "label": "build",
      "command": "C:\\mingw64\\bin\\mingw32-make.exe",
      "args": [],
      "type": "shell",
},
{
      "label": "clean",
      "command": "C:\\mingw64\\bin\\mingw32-make.exe",
      "args": ["clean"],
      "type": "shell",
},
//launch.json
      "preLaunchTask": "build" (或改成clean)
```

```
//改配置可以实现按F5一键makefile+执行main.exe文件
```

cmake是用来生成makefile的方式

1.10 使用程序主函数的命令行参数

```
#include <iostream>
#include <cstdio> // atoi()
int main(int argc, char** argv)
{
    int a, b;
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    std::cout << a + b << std::endl;
    return 0;
}
```

IDE中输入命令行参数（VScode）：View -> Command Palette -> Open "launch.json" -> 修改args `"args": ["arg1","arg2"]`

1.11 GDB调试工具(Linux)

`g++ example.cpp -o example.out -g` -g在可执行程序中包含标准调试信息

```
gdb example.out                 //调试example.out
r un                            //运行程序
b reak + 行号/函数名             //设置断点
    b reak 10 if (k == 2)       //可以根据具体运行条件设置断点
    d elete break 1             //删除1号断点
info b reak                     //查看已设断点
wa tch x                        //当x的值发生变化时暂停 [!重要! VScode中没有此功能]
awatch x                        //当读取或写入x的时候暂停
set x = 1 or print x = 1        //赋值语句/赋值+输出语句
c ontinue                       //跳至下一个断点
s tep                           //单步执行（进入）
n ext                           //单步执行（不进入）
p rint x                        //输出变量/表达式x
disp lay x                      //持续检测变量/表达式x
l ist                           //列出程序源代码
q uit                           //退出
回车                             //重复上一条指令
//当程序使用-O2 -O3优化后，程序运行结果不同或者报错，不能使用gdb调试。
```

2. 对象的基础知识

OOP核心思想——**数据抽象**：类的接口与实现分离。

2.1 函数重载与缺省值

函数重载： 同一名称的函数，有两个以上不同的函数实现，被称为“函数重载”。（多个同名的函数实现之间，必须保证至少有一个函数参数的类型有区别）
内置类型转换： 如果函数调用语句的实参与函数定义中的形参数据类型不同，且两种数据类型在C++中可以进行自动类型转换，则实参会被转换为形参的类型（float转int：向0取整）
函数参数的缺省值：调用函数时，若不提供相应的实参，则编译自动将相应形参设置成缺省值。

1. 缺省值必须是最后一个参数
2. 缺省值冲突可能导致**二义性**

2.2 基础知识

auto关键字（C++11语法）
1. 编译器根据上下文自动确定变量的类型。

`auto i = 3; //int`

`auto f = 0.4f; //float`

`auto a('c'); //char`

`auto b = a; //char`

`auto *x = new auto(3); //int*`

`auto x = new auto(3); //int*`

2. 追踪返回类型的函数
   可以将函数返回类型的声明信息放到函数参数列表的后面进行声明

`auto func(char* ptr, int val) -> int;`

3. auto变量必须在编译期确定其类型
4. auto变量必须在定义时初始化
5. 函数的参数不能声明为auto
6. 不能使用sizeof或者typeid操作符

decltype （与auto连用）`declare type`

1. 对变量或表达式结果的类型进行推导
2. 重用匿名类型

```
struct {int d; double b;} anon_s;
int main() {
    decltype(anon_s) as;
    cin >> as.d;
}
```

3. 结合auto和decltype，自动追踪返回类型（C++11）

```
auto func(int x, int y) -> decltype(x + y)
{
    return x + y;
}
```

4. 用于代替冗长复杂、变量使用范围专一的变量声明
5. 在定义模板函数时，用于声明依赖模板参数的变量类型

```
template<typename _Tx, typename _Ty>
void Multiply(_Tx x, _Ty y)
{
    auto v = x * y;
    std::cout << v;
}
//or:
/* auto multiply(_Tx x, _Ty y) -> decltype(x * y) {
    return x * y;
}*/
Multiply(2, 3);
Multiply(2, 3.3);
//decltype(Multiply(3, 6.7)) result = Multiply(3, 6.7);
//std::cout << result;
```

内存申请与释放

指针变量所指内存可以通过new/delete运算符在程序运行时动态生成和删除

```
    int * ptr = new int (10); //单个变量
    int * array = new int[10]; //10元素数组
    delete ptr; //删除指针变量所指单个内存单元
    delete[] array; //删除多个单元组成的内存块
```

零指针

1. NULL或者0都可以表示空指针（NULL可以用来表示空指针，也是整数0）
2. **nullptr**(C++11)表示严格意义上的空指针——所以用nullptr而不是NULL

For循环

1. 基于范围的for循环

```
int arr[3] = {1, 3, 9};
for (int e : arr) //auto e : arr也可以
   cout << e << endl;
return 0;
```

2.3 类与对象

对象是由一组属性数据和对这些数据进行特定操作的一组服务所构成的“结合体”
封装 = {结构属性/数据， 服务/函数}

用户定义类型——类class

类class = 成员函数 + 成员变量
成员函数必须在类内声明，但定义（实现）可以在类内或类外
一般在头文件中声明类class，在实现文件中定义成员函数（类外需要类名限定）

```
//matrix.h
#ifndef MATRIX_H
#define MATRIX_H

class Matrix {
    int data[6][6];
public:
    void fill(char dir); //也可以在类内定义成员函数
};

#endif

//matrix.cpp
#include "matrix.h"

void Matrix::fill(char dir) {...}
```

2.4 成员变量与成员函数（理解public和private）

类的成员（数据、函数）可以根据需要分成组，不同组设置不同的访问权限

1. public: 被public修饰的成员可以在类外访问
2. private: 默认权限；被private修饰的成员不允许在类外访问（但可以在类内访问操作）
3. protected

对象：用类来定义的变量通常被称为“对象”，可以使用`对象名.成员名`的形式访问或调用对象的成员函数，用`对象指针->成员名`访问数据成员或成员函数

2.5 this指针

所有成员函数的参数中，隐含着一个指向当前对象的指针变量，其名称为**this**

```
void Matrix::fill(char dir) {
    this -> data[0][0] = 1;
}
```

2.6 内联函数

函数调用要进行一系列的准备和后处理工作（压栈、跳转、退栈、返回等），所以函数调用是一个比较慢的过程，如果大量调用可能会拖慢程序。
使用**内联函数**，编译器自动产生等价的表达式

内联函数和宏定义的区别

1. 内联函数可以执行类型检查，进行编译器错误检查
2. 内联函数可调试，而宏定义的函数不可调试

3. 宏定义的函数无法操作私有数据成员
4. 宏使用的最常见场景：字符串定义、字符串拼接、标志粘贴

内联函数的注意事项

1. 避免对大段代码使用内联修饰符
2. 避免对包含循环或者复杂控制结构的函数使用内联定义
3. 避免将内联函数的声明和定义分开（一般都写在头文件中）
4. 定义在类声明中的函数默认为内联函数
5. 一般**构造函数**、**析构函数**都被定义为内联函数

3. 对象的创建与销毁

3.1 构造函数

构造函数没有返回值类型，函数名与类名相同
类的构造函数可以重载，即可以使用不同的函数参数进行对象初始化
构造函数可以使用**初始化列表**初始化成员数据
　　初始化列表的成员是按照**声明的顺序初始化**的，而不是按照列表中出现的顺序
假如不给成员变量初始化——全局变量和函数静态变量(static)会自动置0；局部变量一般不会自动初始化，其值为未定义，运行结果和系统、编译器有关

```
class A {
private:
    int a;
    int b;
public:
    A(int i) : a(i) {/*函数体*/} //使用初始化列表初始化成员函数
    //example:  A(int i) : b(i), a(b) {}
    //先初始化a，再初始化b，因此a的值不可预测
}
```

委派构造函数： 在构造函数的初始化列表中，还可以调用其他构造函数，称为**委派构造函数**

```
class info{
public:
    info() {Init();}
    info(int i) : info() {id = i;}
    info(char c) : info() {gender = c;}
private:
    void Init() {...} //其他初始化
    int id;
    char gender;
};
```

就地初始化： C++11之前，类中的一般成员变量不能在类定义时进行初始化，只能通过构造函数进行；C++11新增支持**就地初始化**
默认构造函数： 不带任何参数的构造函数，或每个形参提供默认实参的构造函数，也被称为“缺省构造函数”

```
class A {
private:
    int a = 1; //声明+初始化
    double b {2.0}; //声明+初始化
public:
    A() {} //定义默认构造函数
    A(int i, double j) : a(i), b(j) {}
};
A a; //调用默认构造函数
A b = A(); //同样调用默认构造函数
//就地初始化只是一种简便的表达方式，实际操作仍然在对象构造的时候执行
```

先调用成员变量的构造，再执行自己的构造函数

```
class Test{
public:
    Member m;
    Test() {cout << "Test()" << endl;}
};
Test t; //先调用Member的构造函数，再执行Test的构造函数
```

没有手动定义默认构造函数时，编译器会帮我们隐式地合成一个默认构造函数
显式声明默认构造函数：`A() = default;`（编译器会定义隐式默认构造函数，即使有其他构造函数存在）
显示删除构造函数：`A(char cls) = delete;`（用delete显示删除构造函数，避免产生未逾期行为的可能性，当定义`A a('cls');`时，编译错误）
对象数组的初始化：
无参定义对象数组，必须要有默认构造函数
构造函数有参数的数组初始化示例：

```
    A a[3] = {1, 3, 5}; //构造函数只有一个参数
    A a[3] = {A(1,2), A(3, 5), A(0,7)}; //构造函数有两个整型参数
```

3.2 析构函数

对象的清除和释放资源是由编译器在对象作用域结束处自动生成调用析构函数代码来完成的。
一个类只有一个析构函数，名称是“~类名”，没有函数返回值，没有函数参数

```
class ClassRoom {
    int num;
    int * ID_list;
public:
    ClassRoom() : num(0), ID_list(nullptr) {}
    ...///ID_list = new int[10];
    ~ClassRoom() { //析构函数
         if (ID_list) delete[] ID_list; //释放内存
    }
}
```

先执行自己的析构函数，再调用成员变量的析构
隐式定义的析构函数： 注意！隐式定义的析构函数不会delete指针成员，因此有可能造成内存泄露

3.3 对象的构造与析构时机（局部对象和全局对象）

局部对象的构造与析构

1. 局部对象：执行到相应代码时被初始化；所在作用域结束后被析构
2. 作用域：该变量能够引用的区域，例如{}将会形成一个作用域

全局对象的构造与析构

1. 全局对象：在main()函数调用之前进行初始化
2. 在同一编译单元（同一源文件）中，按照定义顺序进行初始化
3. 不同编译单元中，对象初始化顺序不确定
4. 在main()函数执行完return之后，对象被析构

   ！通常建议使用参数来替代全局变量

3.4 引用 reference

`类型名 & 引用名 = 变量名 //Example: int & quote = VAR;`
引用必须在定义时进行初始化，不能修改引用指向
1. 函数参数是引用类型
表示函数的形式参数与实际参数是同一个变量，改变形参将改变实参

```
//传入指针
void swap(int *a, int * b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
swap(&a, &b);
//传入引用
void swap(int &a, int &b) {
    int tmp = a;
    a = b;
    b = tmp;
}
swap(a, b); //注意不是swap(&a, &b)
```

2. 函数返回值是引用类型
但不得指向函数的临时变量（不存在空引用，必须连接到合法的内存）

3.5 运算符重载

可以重载的运算符（加粗运算符只能通过成员函数重载）

1. 双目算术运算符：+，-，*，/，%
2. 关系运算符：==，!=，<，>，<=，>=
3. 逻辑运算符：||，&&，！
4. 单目运算符：+ (正)，- (负)，* (指针)，& (取地址)
5. 自增自减运算符：++，--
6. 位运算符：| (按位或)，& (按位与)，~ (按位取反)，^ (按位异或)，<< (左移)，>> (右移)
7. 赋值运算符：**=**，+=，-=，*=，/=，%=，&=，|=，^=， <<=，>>=
8. 空间申请与释放：new，delete，new[]，delete[]
9. 其他运算符：**( ) (函数调用)**，**-> (成员访问)**，，(逗号)，**[] (下标)**

对同一运算符，只能采用一种实现（全局函数**或**成员函数重载）

全局函数的运算符重载: `ClassName operator+(ClassName a, ClassName b) {...}`

成员函数的运算符重载: `Class ClassName{int data; public: ClassName& operator+(ClassName& b) {...};}`

1. 运算符++的重载：通过函数体中没有使用的哑元参数来区分前缀与后缀的同名重载

```
//++a <==> operator++(a) 前缀没有哑元参数
const A& operator++(A& a) { //返回值const：不希望返回值被修改；&：节约开销；A也不错
    ++a.data;
    return a;
}
//a++ <==> operator++(a, int) 后缀有哑元参数
A operator++(A& a, int) { //返回值不能加引用
    A new_a(a.data);
    ++a.data;
    return new_a;
}
cout << (++a).data << endl;
cout << (a++).data << endl;

//哑元可以没有变量名
int fun(int, int a) {return a;}
```

2. 运算符()的重载：在自定义类中也可以重载函数运算度()，它使对象看上去像是一个函数名

```
//成员函数运算符()重载
public: //class Test里的成员函数
    int operator()(int a, int b) {
        cout << "operator() called " << a << ' ' << b << endl;
        return a + b;
    }
int main() {
    Test sum;
    int s = sum(3, 4); //sum对象看上去像一个函数，故也称为“函数对象”
    int t = sum.operator()(5,6);
    return 0;
```

```
}
```

3. 运算符[]的重载

如果返回类型是**引用**，则数组运算符调用可以出现在等号左边，接受赋值
如果返回类型不是引用，则只能出现在等号右边

```
//class WeekTemperature中的public函数
int& operator[](const char* name) // 字符串作下标
    {
        for (int i = 0; i < 7; i++) {
            if (strcmp(week_name[i], name) == 0)
                return temperature[i];
        }
        return error_temperature; //没有匹配到字符串
    }
int main() {
    WeekTemperature beijing;
    beijing["mon"] = -3; //可以是左值
    return 0;
}
```

4. =，[]，()，->只能通过成员函数来重载

编译错误：`error:'cls& operator[](cls&, cls&)' must be a nonstatic member function`
当没有自定义operator=时，编译器会自动合成一个默认版本的复制操作，在类内定义operator=，编译器则不会自动合成；如果使用全局函数重载，可能会对是否自动合成产生干扰

5. 流运算符<<, >>的重载

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

class Test {
    int id;
public:
    Test(int i) : id(i) { cout << "obj_" << id << " created\n"; }
    friend istream& operator>>(istream& in, Test& dst);
    //friend表示调用该函数可以访问该对象private的东西
    friend ostream& operator<<(ostream& out, const Test& src);
};

istream& operator>>(istream& in, Test& dst) {
    //返回值引用：考虑 cin >> a >> b; 运行完"cin >> a"返回cin才能继续输入
    //传入参数istream& in表示第一个变量：cin
    //传入参数Test& dst引用：输入到dst，dst被改变，必须引用
    in >> dst.id;
    return in;
}

ostream& operator<<(ostream& out, const Test& src) {
    //返回值引用：考虑cout << a << b; 运行完"cout << a"返回cout才能继续输出
```

```
    //传入参数ostream& out表示第一个变量：cout
    //传入参数const Test& src: 常量是因为输入量不会被改变，引用是为了“节省开销”
    out << src.id << endl;
    return out;
}
//返回值引用：（测试代码）cout << obj1 << obj2 << obj3;

int main() {
    Test obj(1);
    cout << obj;  // operator<<(cout,obj)
    cin >> obj;  // operator>>(cin,obj)
    cout << obj;
    return 0;
}
//只能使用全局函数重载（在类内声明，在类外定义）：因为如果是成员函数，传入参数的第一个默认为对象（但事实上应该是cin/cout）
```

3.6 友元

被声明为**友元**的函数或类，具有对**出现友元声明的类**的private及protected成员的访问权限
友元的声明只能在**类内**进行（在public或private里都可以），定义类内类外都可以（友元类不能在类里定义；友元函数定义在类内：全局函数），但一定不是类的成员函数
一个普通函数可以是多个类的友元函数

```
class X {};
class A {
    int data; //默认私有成员
public:
    friend void foo(A &a); //类内声明全局函数为友元
    friend void X::foo(Y); //类内声明X类的成员函数为友元
    friend X::X(Y), X::~X(); //类内声明X的构造函数、析构函数为友元;
    friend class Y; //友元类前置声明，但不能定义
    friend X; //友元类声明
};
class Y {}; //Y能访问A的所有成员
void foo(A &a) { //函数也可以在类内定义
    cout << a.data << endl;
}
```

3.7 静态变量/函数

1. 普通静态变量/函数

```
//a.cpp
static int i = 1;
static int func() {...}
//b.cpp
```

```
extern int i; //链接错误
```

1. 初次定义时必须要初始化（且只能初始化1次）
2. 静态局部变量存储在静态存储区，生命周期持续到整个程序结束
3. 静态全局变量/函数内部可链接（对比非静态全局变量：外部可链接），作用域仅限声明文件，可以避免同名冲突

2. 类的静态数据成员

```
//a.h
class A {
public:
    static int data; //类内声明静态数据成员
};
//a.cpp
int A::data = 0; //类外定义静态数据成员（必须要初始化）//modified 2022.6.14
A::data++; //通过类名访问并修改
A a;
a.data++; //通过对象访问并修改
```

1. 静态数据成员/成员函数被该类的所有对象**共享**
2. 静态数据成员/成员函数可以通过**对象**访问，也可以通过**类名**访问
3. 静态数据成员/成员函数在程序开始前初始化，不依赖于对象实例化
4. 静态数据成员应该在.h文件中**声明**，在.cpp文件中**定义**，否则可能链接失败——一定是类内声明，类外定义（可以不初始化）

3. 类的静态成员函数

```
class A {
public:
    static void func() {} //定义可以在类外
};
//可以用对象或者类名直接调用静态成员函数
```

静态成员函数不能访问非静态成员

4. 静态对象的构造与析构

函数内部定义的静态局部对象：

1. 程序执行到该静态局部对象的代码时被初始化
2. 离开作用域**不析构**
3. 第二次执行到该对象代码时，**不再初始化**，直接使用上一次的对象
4. 在main()函数结束后被析构

静态全局对象和类静态对象：main()函数前初始化，return后析构

```
//类静态对象：类A的对象a是类B的静态变量
//与B是否实例化无关
class A {};
class B {
    static A a;
```

```
};
```

3.8 常量

常量关键字const常用于修饰变量、引用/指针、函数返回值等

1. 普通常量修饰

```
    const int n = 1; //修饰变量必须就地初始化，变量值不能改变
    int a = 1; const int& b = a; //不能通过b改变a的值
    const int* func() {...} //函数返回值的内容不能被修改
```

2. 类的常量数据成员/成员函数

```
class A {
    const int data = 1; //在对象的整个生命周期里不能更改
    void func() const {...} //实现语句不能修改类的数据成员
    //如果类的成员函数试图修改类的数据成员的值，不会报错，不会修改
    //原因：拷贝构造函数
    //改成this->data++;可以观察到报错
};
const A a; //则a只能调用以const修饰的成员函数
```

常量成员函数可以修改静态数据成员
非常量对象的常量成员函数不能访问非常量成员函数
常量数据成员的初始化: 构造函数初始化列表；就地初始化；不允许在函数体中初始化

3. 常量对象：只能调用常量函数，不能修改数据成员

4. 类的常量静态数据成员

不存在常量静态函数:
静态函数隶属于类，可以不实例化而直接通过类名访问
常量/非常量函数的访问权限需要通过实例化后的对象是否为常量对象来决定。常量修饰函数必须绑定在对象上
因此，静态函数和常量函数互相冲突

```
class A {
    static const char* cs; //不可以就地初始化
    static const int i = 2; //可以就地初始化
};
```

类内声明类外定义；例外：int和enum类型可以就地初始化

3.9 对象的构造与析构

1. 常量对象的构造与析构：与普通对象相同
2. 静态对象的构造与析构：静态全局对象与普通全局对象相同，静态局部对象（函数内作用域；类静态对象）不同
3. 参数对象的构造与析构

```
//若传递的是形参
void fun(A b) {
    cout << "In fun: b.s = " << b.s << endl;
}
fun(a);
//在函数被调用时，b被构造，调用拷贝构造函数进行初始化。
//默认情况下，对象b的属性值和a一致。
//函数结束时，调用析构函数，b被析构

//若传递的是类对象的引用
void fun(A &b) {
    cout << "In fun: b.s = " << b.s << endl;
}
fun(a);
//在函数被调用时，b不需要初始化，因为b是a的引用。
//在函数结束时，也不需要调用析构函数，因为b只是一个引用，而不是A的对象。
```

4. 类的指针成员：有指针的类作为函数参数类型要设为引用

```
class A {
public:
    int *data; // 注意这是一个指针
    A(int d) {data = new int(d);}
    ~A() {delete data;} // 注意这里，释放之前申请的内存
};
void fun(A a) {
    cout << *(a.data) << endl;
}
int main() {
    A object_a(3);
    fun(object_a);
    return 0; // 在程序结束时会出错
}
//对象a和对象object_a的data成员一样（地址一样），所以delete的时候释放的是同一块内存地址。
//对象a析构时不会出错。但对象object_a析构时，因为试图释放一块已经释放过的内存，所以会出错。
//修改：传参void fun(A &a) {...}
```

5. 对象的new和delete

```
A *pA = new A(3); //生成一个类对象并返回地址（构造函数被调用）
delete pA; //删除该类对象，释放内存资源（析构函数被调用）

A *pA = new A[3];
delete [] pA; //删除该对象数组及数组中的每个元素，释放内存资源
delete pA; //只调用一次析构函数，造成内存泄漏；造成段错误，导致程序崩溃（因为分配空间的起始地址是pA-4byte，而这样直接释放pA指向的内存空间）
```

4. 对象的引用与复制

4.1 常量引用

最小特权原则：给函数足够的权限去完成相应的任务，但不要给予他多余的权限。
常量引用：函数没有修改权限而只能读取参数值

```
void add(const int& a, const int& b);
```

4.2 拷贝构造函数

拷贝构造函数：特殊的构造函数，它的参数是语言规定的，是同类对象的常量引用。

```
class Person {
    int id;
public:
    Person(const Person& src) {id = src.id; ...} //注意参数类型固定
};
```

拷贝构造函数被调用的三种常见情况：

1. 用一个类对象定义另一个新的类对象：`Person b(a)`，`Person c = a;`
2. 函数调用时以类的对象为形参：`Func(Person a)`
3. 函数返回类对象：`Person Func(void)`
   编译器会自动调用“拷贝构造函数”，在已有对象基础上生成新对象

隐式定义的拷贝构造函数：调用所有数据成员的拷贝构造函数或拷贝赋值运算符
对于**基础类型 (int, double...)**（不包括递归调用基类拷贝构造函数的情况）来说，默认的拷贝方式为**位拷贝**，即对整块内存进行复制。
位拷贝在遇到**指针类型成员**时可能会出错，导致多个指针类型的变量指向同一个地址。因此，为了避免指针被重复删除，不应使用隐式定义的拷贝构造函数。

拷贝构造函数的执行顺序

```
Myclass func(Myclass c)  { //拷贝构造函数(c)
    Myclass tmp; //默认构造函数(tmp)
    return tmp; //拷贝构造函数（返回类对象tmp）
}  //tmp的析构函数；c的析构函数
```

拷贝构造函数的调用时机

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

class Test {
public:
    Test() { //构造函数
        cout << "Test()" << endl;
```

```
    }
    Test(const Test& src) { //拷贝构造
        cout << "Test(const Test&)" << endl;
    }
    ~Test() { //析构函数
        cout << "~Test()" << endl;
    }
};

Test copyObj(Test obj) {
    cout << "func()..." << endl;
    return Test();
}

int main() {
    cout << "main()..." << endl;
    Test t;
    Test a = copyObj(t);//修改代码
    return 0;
}
//运行结果如下:
//g++ test.cpp --std=c++11 -fno-elide-constructors -o test
main()...
Test() //调用构造函数构造t
Test(const Test&) //调用拷贝构造函数拷贝obj = t
func()...
Test() //调用构造函数构造无名对象unnamed
Test(const Test&) //调用拷贝构造函数拷贝返回值unnamed到某无名对象unnamed2
~Test() //析构临时变量unnamed
Test(const Test&) //调用拷贝构造函数拷贝unnamed到对象a
~Test() //析构unnamed2
~Test() //析构obj
~Test() //析构a
~Test() //析构t
```

使用拷贝构造函数

频繁的拷贝构造函数会造成程序效率的显著下降。
解决方法:

1. 使用**引用/常量引用**传参数或返回对象

```
func(MyClass a) -> func(const MyClass& a) //引用或常量引用传递参数
MyClass func(...) -> MyClass& func(...) //返回值为引用
```

2. 将拷贝构造函数声明为private

```
class MyClass {
    MyClass(const MyClass&) {}
public:
    MyClass() = default;
```

```
};
```

3. 用delete关键字让编译器不生成拷贝构造函数的隐式定义版本

```
class MyClass {
public:
    MyClass() = default;
    MyClass(const MyClass&) = delete; //拷贝构造函数显示删除
};
```

4.3 右值引用

左值：可以取地址、有名字的值——可以被&引用
右值：不能取地址、没有名字的值；常见于常值、函数返回值、表达式
右值引用: 右值可以被&&引用 `int &&e = a + b;`，&&不能引用左值
常量左值引用能绑定右值： `const int &e = 3;`
所有的引用（包括右值引用）本身都是左值

```
void ref(int &x) {
    cout << "left " << x << endl;
}
void ref(int &&x) {
    cout << "right " << x << endl;
    ref(x); //调用函数left
}
int main() {
    ref(1); //1是一个常量，调用函数right
    return 0;
}
```

4.4 移动构造函数

右值引用可以延续即将销毁变量的生命周期，用于构造函数可以提升处理效率，在此过程中尽可能少地进行拷贝。**使用右值引用作为参数**的构造函数叫做**移动构造函数**

```
ClassName(const ClassName& VairableName); //拷贝构造函数
ClassName(ClassName&& VariableName); //移动构造函数
```

移动构造函数示例

```
class Test {
public:
    int * buf; //// only for demo.
    Test() {
        buf = new int[10]; //申请一块内存
        cout << "Test(): this->buf @ " << hex << buf << endl;
```

```
    }
    ~Test() {
        cout << "~Test(): this->buf @ " << hex << buf << endl;
        if (buf) delete[] buf;
    }
    Test(const Test& t) : buf(new int[10]) {
        for(int i=0; i<10; i++)
            buf[i] = t.buf[i]; //拷贝数据
        cout << "Test(const Test&) called. this->buf @ "
            << hex << buf << endl;
    }
    Test(Test&& t) : buf(t.buf) { //直接复制地址，避免拷贝
        cout << "Test(Test&&) called. this->buf @ "
            << hex << buf << endl;
        t.buf = nullptr; //将t.buf改为nullptr，使其不再指向原来内存区域
    }
};

Test GetTemp() {
    Test tmp;
    cout << "GetTemp(): tmp.buf @ "
        << hex << tmp.buf << endl;
    return tmp;
}

void fun(Test t) {
    cout << "fun(Test t): t.buf @ "
        << hex << t.buf << endl;
}

int main() {
    Test a = GetTemp();
    cout << "main() : a.buf @ " << hex << a.buf << endl;
    fun(a);
    return 0;
}
//以下为程序运行结果:
Test(): this->buf @ 0x801920
GetTemp(): tmp.buf @ 0x801920
Test(Test&&) called. this->buf @ 0x801920
~Test(): this->buf @ 0
Test(Test&&) called. this->buf @ 0x801920
~Test(): this->buf @ 0
main() : a.buf @ 0x801920
Test(const Test&) called. this->buf @ 0x801d50
fun(Test t): t.buf @ 0x801d50
~Test(): this->buf @ 0x801d50
~Test(): this->buf @ 0x801920
```

1. 右值引用：移动语义

对左值调用移动构造函数加快左值初始化的构造速度——**std::move函数**
输入：左值（包括变量等，该左值一般不再使用）
返回值：该左值对应的右值
【注意】：如果参数为**常量引用**，编译器将采用拷贝构造，move函数失效

```
Test a;
Test b = std::move(a); //可以调用移动构造函数对b进行初始化
```

示例：右值引用结合std::move可以显著提高swap函数的性能

```
template <class T>
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

2. 拷贝/构造函数的调用时机

1. 判断依据：引用的绑定规则
   拷贝构造函数的形参类型：**常量左值引用**，可以绑定常量左值、左值和右值
   移动构造函数的形参类型：**右值引用**，可以绑定右值【优先】
2. 拷贝构造函数的常见调用时机
   用一个类对象/引用/常量引用初始化另一个新的类对象
   以类的对象为函数形参，传入实参为类的对象/引用/常量引用
   函数返回类对象（类中未显示定义移动构造函数，不进行返回值优化）
3. 移动构造函数的常见调用时机
   用一个类对象的右值初始化另一个新的类对象：`Test b = std::move(a);`，`Test b = func(a);`
   以类的对象为函数形参，传入实参为类对象的右值：`func(Test());`，`func(std::move(a));`
   函数返回类对象（显示定义移动构造函数）：`return Test();`

4.5 拷贝赋值运算符与移动赋值运算符

赋值重载函数必须要是**类的非静态成员函数**(non-static member function)，不能是友元函数

1. 拷贝赋值运算符
   可以绑定常量左值、左值和右值

```
Test& operator= (const Test& right) {
    if (this == &right)
        cout << "same obj!\n";
    else {
        for(int i=0; i<10; i++)
            buf[i] = right.buf[i]; //拷贝数据
            cout << "operator=(const Test&) called.\n";
        }
    return *this;
```

```
}
```

2. 移动赋值运算符【优先】
   可以绑定右值（常量、表达式、函数返回）

```
Test& operator= (Test&& right) {
    if (this == &right)
        cout << "same obj!\n";
    else {
        this->buf = right.buf; //直接赋值地址
        right.buf = nullptr;
        cout << "operator=(Test&&) called.\n";
    }
    return *this;
}
```

3. 编译器自动合成的函数/运算符
   类中特殊的成员函数/运算符，即便用户不显示定义，编译器也会根据自身需要自动合成:
   默认构造函数
   拷贝构造函数
   移动构造函数（C++11起）
   拷贝赋值运算符
   移动赋值运算符（C++11起）
   析构函数

4. 小结：利用返回值优化提高运行效率

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

class Test{
public:
    int data = 0;
    Test(){}
    Test(const Test& t){}
    Test(Test&& t){}
};

Test fn1(){
    Test tmp; return tmp;
}
Test&& fn2(){
    Test tmp; return move(tmp); //运行错误，d会指向被析构的tmp
}
Test fn3(){
    Test tmp; return move(tmp); //运行错误，移动构造临时变量
}

int main(){ //接收返回值的三种办法
    const Test& a = fn1(); //常量左值引用
    Test&& b = fn1(); //右值引用
```

```
    Test c = fn1(); //构造新对象
    Test&& d = fn2(); //运行错误，d会指向被析构的tmp
    return 0;
}
```

4.6 类型转换：Srt->Dst

自动类型转换：可以通过定义特定的转换运算符和构造函数来完成
强制类型转换

1. 自动类型转换

方法一：在源类中定义“目标类型转换运算符”

成员函数 + 不能有返回类型 + 参数列表为空 + 返回类型相同

```
class Dst { //目标类
public:
    Dst() {cout << "Dst::Dst()" << endl;}
};

class Src { //源类
public:
    Src() {cout << "Src::Src()" << endl;}
    operator Dst() const { //不需要指定返回类型，因为operator后Dst()已经指明
        cout << "Src::operator Dst() called" << endl;
        return Dst();
    }
};
```

方法二：在目标类中定义“源类对象作参数的构造函数”

```
class Src; //前置类型声明，因为在Src中要用到Src类
class Dst { //目标类
public:
    Dst() {cout << "Dst::Dst()" << endl;}
    Dst(const Src& s) { //常量引用
        cout << "Dst::Dst(const Src&)" << endl;
    }
};

class Src { //源类
public:
    Src() {cout << "Src::Src()" << endl;}
};

int main() {
    Src s;
    Dst d1(s);
```

```
    Dst d2 = s;
    return 0;
}
```

2. 禁止自动类型转换 explicit

如果用**explicit**修饰类型转换运算符或类型转换构造函数，则相应的类型转换必须显示地进行

```
explicit operator Dst() const;
explicit Dst(const Src& s);

Dst d1(s); //可以执行，被认为是显示初始化
Dst d2 = s; //错误，隐式转换
//可以改写为:
Dst d2 = static_cast<Dst>(s);
```

3. 强制类型转换

- const_cast 去除类型的const或volatile属性
- static_cast 类似于C风格的强制转换，无条件转换，静态类型转换
- dynamic_cast 动态类型转换，如派生类和基类之间的多态类型转换
- reinterpret_cast 仅仅重新解释类型，但没有进行二进制的转换

5. 对象的组合与继承

OOP核心思想——**继承**：建立相关类型的层次关系（基类与派生类）。

5.1 组合

如果对象a是对象b的一个组成部分，则称b为a的**整体对象**，a为b的**部分对象**。并把b和a之间的关系称为**“整体 - 部分”**关系（也可称为**“组合”**或**“has-a”**关系）

访问对象组合的两种方式：① 公有数据成员； ② 私有数据成员 + 公有访问接口

```
class Car {
private:
    Wheel w; ///wheel类中有公有接口void set(int n) {_num = n;}
    //可以就地初始化，但是语法是wheel w = wheel(2)，而不是wheel w(2)
public:
    Engine e; /// 公有成员，直接访问其接口
    void setWheel(int n) {w.set(n);} /// 提供私有成员的访问接口
};
```

1. 子对象构造时若需要参数，则应在当前类的构造函数的初始化列表中进行。若使用默认构造函数来构造子对象，则不用做任何处理。
2. 对象构造与析构函数的次序
   先完成子对象构造，再完成当前对象构造
   子对象的构造次序仅由类中**声明的次序**所决定
   析构函数的次序与构造函数相反

3. 对象组合的拷贝与赋值

```
class C1{
public:
    int i;
    C1(int n) : i(n) {}
    C1(const C1 &other) /// 显式定义拷贝构造函数
        {i = other.i; cout << "C1(const C1 &other)" << endl;}
};
class C2{
public:
    int j;
    C2(int n) : j(n) {}
    C2& operator= (const C2& right){ /// 显式定义赋值运算符
        if (this != &right) {
            j = right.j;
            cout << "operator=(const C2&)" << endl;
        }
        return *this;
    }
};
class C3{
public:
    C1 c1;
    C2 c2;
    C3() : c1(0), c2(0) {}
    C3(int i, int j) : c1(i), c2(j) {}
    void print() {cout << "c1.i = " << c1.i << " c2.j = " << c2.j << endl;}
};
int main(){
    C3 a(1, 2);
    C3 b(a);  //C1执行显式定义的拷贝构造，C2执行隐式定义的拷贝构造
    cout << "b: "; b.print(); cout << endl;
    C3 c;
    cout << "c: "; c.print();
    c = a;  //C1执行隐式定义的拷贝赋值，C2执行显式定义的拷贝赋值
    cout << "c: "; c.print();
    return 0;
}
```

5.2 继承

如果类A具有类B全部的属性和服务，而且具有自己特有的某些属性或服务，则称A为B的**特殊类**，B为A的**一般类**。
如果类A的全部对象都是类B的对象，而且类B中存在不属于类A的对象，则A是B的**特殊类**，B是A的**一般类**。
（这是一种**“一般-特殊”结构**，也称**分类结构**，或**”is-a“结构**）

1. 基本概念

基类（base class）：被继承的已有类，也称”父类“
派生类（derived class）：继承的到的新类，也称”子类“、”扩展类“

2. 常见的继承方式

public, private:

```
class Derived : [private] Base {..}; //缺省继承方式为private继承
class Derived : public Base {...};
```

protected: 继承很少被使用

```
class Derived : protected Base {...};
```

3. 不能被继承的函数:

- 构造函数 and 析构函数
  创建派生类对象时，必须调用派生类的构造函数，派生类构造函数调用基类的构造函数，以创建派生对象的基类部分。C++11新增了继承构造函数的机制（使用using），但默认不继承
  释放对象时，先调用派生类析构函数，再调用基类析构函数
- 赋值运算符
  编译器不会继承基类的赋值运算符（参数为基类）
  但会自动合成隐式定义的赋值运算符（参数为派生类），其功能为调用基类的赋值运算符。
- 友元函数（不是类成员）

4. 派生类对象的构造与析构过程

构造过程：若想要显式调用，则只能在派生类构造函数的**初始化列表**中进行派生类对象构造。可以调用含参或不含参的基类默认构造函数；若没有显式调用，则编译器自动调用基类的默认构造函数。【先执行基类构造函数来初始化继承的数据，再执行派生类构造函数】

析构过程: 先执行派生类析构函数，再执行有编译器自动调用的基类析构函数

```
class Base {
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() { cout << "Derive::Derive()" << endl; } //没有显式调用，自动调用基类的默认构造函数
    Derive(int i) : Base(i) {cout << "Derive::Derive()" << endl;}
        //显式调用基类构造函数
    using Base::Base;
        //继承基类构造函数 相当于Derive(int i) : Base(i) {}; Derive() : Base(i) {}
};
int main() {
    Derive obj;  //先后调用Base()和Derive()
    return 0;
}
```

using语句（继承基类构造函数）：如果基类的某个构造函数被声明为**私有成员函数**，则不能在派生类中声明继承该构造函数（但是保护成员函数依然可以继承，且可以被派生类对象直接调用）；如果派生类使用了继承构造函数，编译器就不会再为派生类生成隐式定义的默认构造函数

5. 继承方式的选择

1） public继承

基类中公有成员仍能在派生类中保持公有；原接口可沿用；最常用

is-a：基类对象能使用的地方，派生类对象也能使用

2） private继承

is-implementing-in-terms-of（照此实现）：用基类接口实现派生类功能

移除了is-a关系

通常不使用，用组合替代

可用于**隐藏/公开基类的部分接口**，公开方法：`using`关键字

成员访问权限:

基类中的私有成员，**不允许在派生类成员函数或对象中访问**

基类中的公有成员：

允许在派生类成员函数中被访问

public继承方式：成为派生类的公有成员，派生类对象可以访问

private/protected继承方式：成为派生类私有/保护成员，不能被派生类的对象访问（除非用using声明）

基类中的保护成员：允许在**派生类成员函数**中被访问

```
//私有继承，打开基类公有成员的访问权限
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};
class Derive: private Base { // Base的私有继承
public:
    using Base::baseFunc; /// 私有继承时，在派生类public部分声明基类成员名字
};
int main() {
  Derive obj;
  cout << "calling obj.baseFunc()..." << endl;
  obj.baseFunc(); //基类接口在派生类public部分声明，则派生类对象可调用
  return 0;
}
```

```
//私有继承中，基类中的私有、保护成员访问
class Base{
private:
    int a{0};
protected:
    int b{0};
};
class Derive : private Base{
public:
    void getA() {cout << a << endl;}  ///编译错误，不可访问基类中私有成员
    void getB() {cout << b << endl;}  ///可以访问基类中保护成员
};
```

```
int main()
{
    Derive d;
    d.getB();
    //cout << d.b; ///编译错误，派生类对象不可访问基类中保护成员
    return 0;
}
```

```
//私有继承中，打开基类公有成员的访问权限；不允许私有继承的向上转换
class Base {
private:
    int data{0};
public:
    int getData(){ return data;}
    void setData(int i){ data = i;}
};
class Derive : private Base {
public:
    using Base::getData;
};
int main() {
    Derive d1;
    cout << d1.getData();
    //d1.setData(10);    ///隐藏了基类的setData函数，不可访问
    //Base& b = d1;      ///不允许私有继承的向上转换
    //b.setData(10);     ///否则可以绕过D1，调用基类的setData函数
    return 0;
}
```

public继承： 基类的公有成员，保护成员，私有成员作为派生类的成员时，都**保持**原有的状态。
private继承：基类的公有成员，保护成员，私有成员作为派生类的成员时，都作为**私有**成员。
protected继承：基类的公有成员，保护成员作为派生类的成员时，都成为**保护**成员，基类的私有成员仍然是**私有**的。

Image unavailable in the original source: note2.png

5.3 重写隐藏与重载

重载（overload）
目的：提供同名函数的不同实现，属于**静态多态**
函数名必须相同，函数参数必须**不同**，作用域相同
重写隐藏（redefining）
目的：在**派生类中重新定义基类函数**，实现派生类的特殊功能
屏蔽了基类的**所有**其它**同名**函数
函数名必须相同，函数参数可以不同
重写隐藏发生时，基类中该成员函数的其他重载函数都将被屏蔽掉
可以在派生类中通过**using 类名::成员函数名；**在派生类中“恢复”指定的基类成员函数（即去掉屏蔽），使之重新可用

```
class Base {
```

```
public:
    void f() { cout << "Base::f()\n"; }
    void f(int i) { cout << "Base::f(" << i << ")\n"; }
};
class Derive : public Base {
public:
    using Base::f; //derive的对象可以调用f()
    void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};
```

using关键字

1）继承基类构造函数 `using Base::basefunc();`
2）恢复被屏蔽的基类成员函数 `using Base::f;`
3）指示命名空间 `using namespace std;`
4）将另一个命名空间的成员引入当前命名空间 `using std::cout, std::endl; cout << endl;`
5）定义类型别名 `using a = int;`

5.4 多重继承

派生类同时继承多个基类

```
class File{};
class InputFile: public File{};
class OutputFile: public File{};
class IOFile: public InputFile, public OutputFile{};
```

1. 数据存储：如果派生类D继承自的两个基类A，B是同一基类Base的不同继承，则A，B中继承自Base的数据成员会在D中有两个独立的副本，可能带来数据冗余。
2. 如果派生类D继承的两个基类A，B有同名成员a，则访问D中a时，编译器无法判断访问哪个基类成员而报错，可以用以下方式显示表示： `cout << derive.MiddleB::a << endl;`

6. 虚函数与多态

OOP核心思想——**动态绑定**：统一使用基类指针，实现多态行为。

6.1 向上类型转换与对象切片

1. 向上类型转换

派生类对象/引用/指针 转换成 基类对象/引用指针，只对**public继承**有效。`Base *p = & d`
可以由编译器**自动完成**，是一种隐式类型转换（对任何接受基类对象/引用/指针的地方）
对象的向上类型转换：

```
class Base {
public:
    void print() { cout << "Base::print()" << endl; }
};
```

```
class Derive : public Base {
public:
    void print() { cout << "Derive::print()" << endl; }
};
void fun(Base obj) { obj.print(); }

int main() {
    Derive d;
    d.print();
    fun(d); ///输出：Base::print()，而不是Derive::print()!!!
    return 0;
}
```

2. 对象切片

当派生类的**对象**（不是指针或引用）被转换为基类的对象时（如：传参/赋值时），派生类的对象被**切片**为对应基类的子对象——意味着派生类的独有定义内容被丢失（又称派生类新数据/新方法丢失）。

3. 指针（引用）的向上转换

当派生类的指针/引用被转换为基类指针/引用时，不会创建新的对象，但只保留基类的接口（即只能访问基类接口，但是派生类接口/数据仍然存在）

```
class Instrument {
public:
    void play() { cout << "Instrument::play" << endl; }
};
class Wind : public Instrument {
public:
    // Redefine interface function:
    void play() { cout << "Wind::play" << endl; }
};
void tune(Instrument& i) {
    i.play(); ///编译器将tune中的函数调用i.play()与Instrument::play()绑定
    ///区别：对象切片
}

int main() {
    Wind flute;
    tune(flute); /// 引用的向上类型转换(传参)，编译器早绑定，无对象切片产生
    Instrument &inst = flute;  /// 引用的向上类型转换(赋值)
    inst.play();
    return 0;
}
//两次均输出：Instrument::play
```

总结：（对于基类中有虚函数的情况）

1. 转换为**基类指针**或**引用**，则对应虚函数表仍为派生类的虚函数表（晚绑定）。
   如果基类中没有虚函数：早绑定
2. 转换为**基类对象**，产生**对象切片**，调用基类函数（早绑定）。

6.2 函数调用捆绑与虚函数

捆绑：把函数体与函数调用相联系

将函数体的具体实现代码，与调用的函数名绑定。执行到调用代码时直接进入捆绑好的函数体内部。

早捆绑：捆绑在程序运行之前（由编译器和连接器）完成
晚捆绑/动态捆绑/运行时捆绑：捆绑根据对象的实际类型，发生在程序运行时
要求在运行时能确定对象的实际类型，并绑定正确的函数。
晚捆绑只对类中的**虚函数**起作用，使用**virtual**关键字声明虚函数。

虚函数

```
class Base {
public:
    virtual ReturnType FuncName(argument) {}; //虚函数
    ///虚函数必须定义函数体
};
```

对于**被派生类重新定义的成员函数**，若它在基类中被声明为**虚函数**，则通过基类**指针或引用**调用该成员函数时，编译器将根据所指（或引用）对象的实际类型决定是调用基类中的函数，还是调用派生类重写的函数。
若某成员函数在基类中声明为虚函数，当派生类**重写覆盖**（**同名，同参数函数**）它时，无论是否声明为虚函数，该成员函数都仍然是虚函数。
虚函数要定义函数体！

```
class Instrument {
public:
    virtual void play() { cout << "Instrument::play" << endl; }
};
class Wind : public Instrument {
public:
    void play() { cout << "Wind::play" << endl; }
     /// 重写覆盖（稍后：重写隐藏和重写覆盖的区别）
};

void tune(Instrument& ins) { ///如果传参是对象Instrument ins,无效!
    ins.play();
    /// 由于Instrument::play是虚函数，编译时不直接绑定，运行时根据ins实际类型调用。
}

int main() {
    Wind flute;
    tune(flute); ///向上类型转换，输出：Wind::play
    return 0;
}
```

虚函数表：每个包含虚函数的类用于存储虚函数地址的表

对象自身要包含自己实际类型的信息：用**虚函数表(VTABLE)**表示；运行时通过虚函数表确定对象的实际类型。虚函数表有唯一性，即没有重写虚函数。

虚函数指针(vpointer/VPTR): 每个包含虚函数的类**对象**中，编译器秘密地放一个指针，指向这个类的VTABLE。当通过基类指针做虚函数调用时，编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起**晚捆绑**的发生。
编译期间：建立虚函数表VTABLE，记录每个类或该类的基类中所有已声明的虚函数入口地址。
运行期间：建立虚函数指针VPTR，在构造函数中发生，指向相应的VTABLE。

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

class B{
    int i;
    float j;
public:
    virtual void fun1() {
        cout << "B::fun1()" << endl; }
    virtual void fun2() {
        cout << "B::fun2()" << endl; }
};
class D: public B{
public:
    double k;
    virtual void fun1() {
        cout << "D::fun1()" << endl; } ///对fun1重写覆盖，对fun2没有，则fun2使用基类的虚函数地址
};

int main() {
    B b; D d;
    B *pB = &d;
    pB->fun1(); //输出: D::fun1()
    return 0;
}
```

Image unavailable in the original source: note3.png

```
//存放类型信息
#include <iostream>
using namespace std;
#pragma pack(4) //按照4字节进行内存对齐

class NoVirtual{ //没有虚函数
    int a;
public:
    void f1() const {}
    int f2() const {return 1;}
};
class OneVirtual{ //一个虚函数
    int a;
public:
    virtual void f1() const {}
    int f2() const {return 1;}
```

```
};
class TwoVirtual{//两个虚函数
    int a;
public:
    virtual void f1() const {}
    virtual int f2() const {return 1;}
};

int main(){
    cout << "int: " << sizeof(int) << endl; //4
    cout << "NoVirtual: " << sizeof(NoVirtual) << endl; //4
    cout << "void* : " << sizeof(void*) << endl; //8
    cout << "OneVirtual: " << sizeof(OneVirtual) << endl; //12
    cout << "TwoVirtual: " << sizeof(TwoVirtual) << endl; //12
    ///对带有单个虚函数的类OneVirtual，对象的大小是单个int的大小加上一个void指针(实际上是VPTR)的大小。
    ///带有多个虚函数的类TwoVirtual与OneVirtual大小相同，因为VPTR指向一个存放所有虚函数地址的表。
    return 0;
}
```

6.3 虚函数和构造函数、析构函数

构造函数

当创建一个包含有虚函数的**对象**时，必须初始化它的**VPTR**以指向相应的**VTABLE**。设置**VPTR**的工作由**构造函数**完成。编译器在构造函数的开头秘密的插入能初始化**VPTR**的代码。
构造函数**不能也不必**是虚函数。

```
class Base {
public:
    virtual void foo() {cout << "Base::foo" << endl;}
    Base() {foo();}  ///在构造函数中调用虚函数foo
    void bar() {foo();};    ///在普通函数中调用虚函数foo
};
class Derived : public Base {
public:
    int _num;
    void foo() {cout << "Derived::foo" << _num << endl;}
    Derived(int j) : Base(),_num(j) {}
};

int main() {
    Derived d(0); //输出：Base::foo，构造函数中调用的是foo的“本地版本”
    Base &b = d;
    b.bar(); //输出：Derived::foo0，在普通函数中调用虚函数
    b.foo(); //输出：Derived::foo0，直接调用虚函数
    return 0;
}
```

在构造函数中调用一个虚函数，被调用的只是这个函数的**本地版本**（即当前类的版本），即虚机制在构造函数中不工作。
派生类对象初始化顺序（与构造函数初始化列表顺序无关）：**基类初始化，对象成员初始化，构造函数体**
原因：**基类的构造函数**比**派生类**先执行，调用基类构造函数时派生类中的数据成员还没有初始化（上例中`Derive`中的数据成员`i`）。如果允许调用实际对象的虚函数（如`b.foo()`），则可能会用到未初始化的派生类成员。

析构函数

析构函数能是虚的，且常常是虚的。虚析构函数**需定义函数体**。
虚析构函数的用途：当删除基类对象指针时，编译器将根据指针所指对象的**实际类型**，调用相应的析构函数。
若基类析构**不是虚函数**，则删除基类指针所指派生类对象时，编译器仅自动调用基类的析构函数，而不会考虑实际对象是不是基类的对象。这可能会导致**内存泄漏**。
在析构函数中调用一个虚函数，被调用的只是这个函数的**本地版本**，即虚机制在析构函数中不工作。
重要原则：总是将基类的析构函数设置为虚析构函数

```
class Base1 {
public:
    ~Base1() { cout << "~Base1()\n"; }
};
class Derived1 : public Base1 {
public:
    ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
    virtual ~Base2() { cout << "~Base2()\n"; }
};
class Derived2 : public Base2 {
public:
    ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
    Base1* bp = new Derived1;
    delete bp; /// 只调用了基类的虚析构函数：~Base1()
    Base2* b2p = new Derived2;
    delete b2p; /// 派生类虚析构函数调用完后调用基类的虚析构函数：~Derived2(), ~Base2()
    return 0;
}
```

6.4 重写覆盖，override和final

重载 (overload)：函数名必须相同，函数参数必须不同，作用域相同（同一个类，或同为全局函数），返回值可以相同或不同。
重写覆盖 (override)：派生类重新定义基类中的**虚函数**，函数名必须相同，函数参数必须相同，返回值一般情况应相同。派生类的**虚函数表**中原基类的虚函数指针会被派生类中重新定义的虚函数指针覆盖掉。
某基类成员函数为虚函函数，当派生类重写覆盖该函数后，该函数仍然是虚函数。

重写隐藏 (redefining)：派生类重新定义基类中的函数，函数名相同，但是参数不同或者基类的函数不是虚函数（参数相同+虚函数=>不是重写隐藏）。重写隐藏中**虚函数表**不会发生覆盖。

Image unavailable in the original source: note4.png

重写覆盖 vs 重写隐藏

相同点:
都要求派生类定义的函数与基类**同名**；都会**屏蔽**基类中的同名函数，即派生类的实例无法调用基类的同名函数。
不同点:
重写覆盖要求基类的函数是**虚函数**，且函数**参数相同**，返回值一般情况应相同；**重写隐藏**要求基类的函数**不是虚函数**或者**函数参数不同。**
重写覆盖会使派生类虚函数表中基类的虚函数的指针被派生类的虚函数指针**覆盖。重写隐藏不会。**

```
class Base {
public:
    virtual void foo() {cout << "Base::foo()" << endl;}
    virtual void foo(int) {cout << "Base::foo(int)" << endl;} ///重载
    void bar() {};
};
class Derived1 : public Base {
public:
    void foo(int) {cout << "Derived1::foo(int)" << endl;} /// 是重写覆盖
};
class Derived2 : public Base {
public:
    void foo(float) {cout << "Derived2::foo(float)" << endl;} /// 误把参数写错了，不是重写覆盖，是重写隐藏
};

int main() {
    Derived1 d1;
    Derived2 d2;
    Base* p1 = &d1;
    Base* p2 = &d2;
    //d1.foo(); ///由于派生类都定义了带参数的foo，基类foo()对实例不可见
    //d2.foo();
    p1->foo();  ///但是虚函数表中有继承自基类的foo()虚函数，输出：Base::foo()
    p2->foo();  ///输出：Base::foo()
    d1.foo(3);  ///输出：Derived1::foo(int)
    d2.foo(3.0);   ///调用的是派生类foo(float)，输出：Derived2::foo(float)
    p1->foo(3);  ///重写覆盖，虚函数表中是派生类的 foo(int)，输出：Derived1::foo(int)
    p2->foo(3.0);  ///重写隐藏，虚函数表中继承自基类，输出：Base::foo(int)
    return 0;
}
```

const对重写覆盖和重写隐藏的影响

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

```
class Base1 {
public:
    virtual void f() {cout << "Base1::f" << endl;}
};
class Derive1: public Base1 {
public:
    void f() const {cout << "Derive1::f" << endl;} //重写覆盖失效，其实是重写隐藏
    using Base1::f; //使用using恢复被隐藏的基类函数
};

class Base2{
public:
    virtual void g() {cout << "Base2::g" << endl;}
};
class Derive2 : public Base2 {
public:
    void g() {cout << "Derive2::g" << endl;} //重写覆盖
    using Base2::g; ///d.Base1::g()可以调用基类“被覆盖掉的”函数
};

int main(){
    Derive1 a;
    const Derive1 b;
    a.f(); //Base1::f已被恢复，非常量对象优先匹配Base1::f
    b.f(); //常量对象调用Derive1::f

    Base2 c;
    Derive2 d;
    c.g(); //输出: Base2::g
    d.g(); //重写覆盖，调用Derive2::g
    return 0;
}
```

1. override 关键字

override关键字明确地告诉编译器一个函数是对基类中一个**虚函数**的重写覆盖，编译器将对重写覆盖要满足的条件进行检查，**正确的重写覆盖**才能通过编译。

```
class Derived3 : public Base {
public:
    void foo(int) override {cout << "Derived3::foo(int)" << endl;};
        /// 重写覆盖正确，与Derived1等价
    //void foo(float) override {};
        /// 参数不同，不是重写覆盖，编译错误
    //void bar() override {};
        /// bar 非虚函数，编译错误
};
```

2. final 关键字

在**虚函数声明或定义中**使用时，**final**确保函数为虚且不可被派生类重写。可在继承关系链的“中途”进行设定，禁止后续派生类对指定虚函数重写。在**类定义**中使用时，**final**指定此类不可被继承。

```
class Base {
    virtual void foo() {};
};
class A: public Base {
    void foo() final {}; /// 重写覆盖，且是最终覆盖
    void bar() final {}; /// bar 非虚函数，编译错误
};
class B final : public A{ //但B中仍有虚函数表指针
    void foo() override {}; /// A::foo 已是最终覆盖，编译错误
};
class C : public B{ /// B 不能被继承，编译错误
};
```

3. 派生类虚函数的返回值与基类协变（Covariant）

```
class Instrument {
public:
    virtual Instrument& getObj() { return *this; }
};

class Wind : public Instrument {
public:
    virtual Wind& getObj() { return *this;}  //Wind&和Instrument&协变
};
//去掉引用不能通过编译（不满足协变）
```

协变:

1. 都是指针（不能是多级指针）、都是左值引用或都是右值引用，且在**Derive::f**声明时，**Derive::f**的返回类型必须是**Derive**或其他已经完整定义的类型
2. **ReturnType1**中被引用或指向的类是**ReturnType2**中被引用或指向的类的**祖先类**
3. **Base::f**的返回类型相比**Derive::f**的返回类型**同等或更加cv-qualified**

```
class A{};
class B : public A{};
class C{};
class D : public B, public C{};

class Base {
public:
    virtual Base* f1(){}
    virtual Base** f2(){}
    virtual Base& f3(){}
    virtual A& f4(){}
};
class Derive : public Base {
```

```
public:
    Derive* f1(){} //返回值类型都是指针，Base是Derive的祖先类
    //Derive** f2(){} //编译错误，不能是多级指针
    Base** f2(){} //返回值类型相同
    Derive& f3(){} //返回值类型都是引用，Base是Derive的祖先类
    //Derive* f3(){} //编译错误，类型不同，且非协变
    D& f4(){} //返回值类型都是引用，A是D的祖先类
};
```

6.5 纯虚函数与抽象类

虚函数还可以进一步声明为**纯虚函数**。包含纯虚函数的类，通常被称为**抽象类**。

1. **抽象类**不允许定义对象，定义基类为抽象类的主要用途是为派生类规定**共性“接口”**，能**避免对象切片**：保证只有指针和引用能被向上类型转换。
2. **基类纯虚函数**被派生类重写覆盖之前仍是**纯虚函数**。因此当继承一个**抽象类**时，除**纯虚析构函数**外，必须**实现所有纯虚函数**，否则继承出的类也是**抽象类**。

```
class A {
public:
    virtual void f() = 0; //可以在类外定义，函数体提供默认实现;
    ///派生类通过A::f()调用
};
///A obj; //不准用抽象类定义对象！编译不通过！
class Derive1: public A {}; //Derive1仍为抽象类
```

3. **纯虚析构函数**（与虚析构函数一样）仍然**需要函数体**。目的：使基类成为**抽象类**，不能创建基类的对象。
4. 对于**纯虚析构函数**而言，即使派生类不显式覆盖纯虚析构函数，编译器也会自动合成默认析构函数，只要派生类覆盖了其他纯虚函数，该派生类就**不是抽象类**，可以定义派生类对象。

```
class Base { public: virtual ~Base()=0; };
Base::~Base() {} /// 必须有函数体
class Derive : public Base {};

int main() {
    Base b; /// 编译错误，基类是抽象类
    Derive d1; /// 派生类不必实现纯虚析构函数
  return 0;
}
```

6.6 向下类型转换

基类指针/引用转换成**派生类**指针/引用，称为**向下类型转换**。
借助**动态类型检查**保证基类指针指向的对象也可以被要转换的派生类的指针指向。

1. dynamic_cast

C++提供的一个特殊的显示类型转换，是一种**安全的**向下类型转换。
使用**dynamic_cast**的对象**必须有虚函数**，因为它使用了存储在虚函数表中的信息判断实际的类型——通过**虚函数表**来判断是否能进行向下类型转换。
只允许**指针**和**引用**转换。

```
///obj_p，obj_r分别是T1类型的指针和引用

T2* pObj = dynamic_cast<T2*>(obj_p);
    //转换为T2指针，运行时失败返回nullptr
T2& refObj = dynamic_cast<T2&>(obj_r);
    //转换为T2引用，运行时失败抛出bad_cast异常

///在向下转换中，T1必须是多态类型（声明或继承了至少一个虚函数的类），否则不过编译
```

2. static_cast

static_cast在编译时**静态浏览类层次**，**只检查继承关系**。没有继承关系的类之间，必须具有转换途径才能进行转换（自定义或语言语法支持），否则不过编译。运行时**无法确认**是否正确转换。

```
///obj_p，obj_r分别是T1类型的指针和引用

T2* pObj = static_cast<T2*>(obj_p);
  //转换为T2指针
T2& refObj = static_cast<T2&>(obj_r);
  //转换为T2引用

///不安全：不保证指向目标是T2对象，可能导致非法内存访问。
```

3. dynamic_cast和static_cast比较

相同点：都可以完成向下类型转换
不同点：
static_cast在**编译时**静态执行向下类型转换。
dynamic_cast会在**运行时**检查被转换的对象是否确实是正确的派生类。额外的检查需要 RTTI (Run-Time Type Information)，因此要比static_cast慢一些，但是更**安全**。
dynamic_cast通过**虚函数表**来判断是否能进行**向下类型转换**。

重要原则：清楚指针所指向的真正对象

1. 指针或引用的向上转换总是安全的
2. 向下转换时用**dynamic_cast**，安全检查
3. 避免对象之间的转换。

示例

```
class B { public: virtual void f() {} };
class D : public B { public: int i{2018}; };

///转换失败
```

```
int main() {
    D d; B b;
        //D d1 = static_cast<D>(b); ///未定义类型转换方式
        //D d2 = dynamic_cast<D>(b); ///只允许指针和引用转换
    D* pd1 = static_cast<D*>(&b); /// 有继承关系，允许转换
    if (pd1 != nullptr){
        cout << "static_cast, B*(B) --> D*: OK" << endl;
        cout << "D::i=" << pd1->i << endl;
    } /// 但是不安全：对D中成员i可能非法访问
    D* pd2 = dynamic_cast<D*>(&b);
    if (pd2 == nullptr) /// 不允许不安全的转换
        cout << "dynamic_cast, B*(B) --> D*: FAILED" << endl;
    return 0;
}

///转换成功
int main() {
    D d; B b;
    B* pb = &d;
    D* pd3 = static_cast<D*>(pb);
    if (pd3 != nullptr){
        cout << "static_cast, B*(D) --> D*: OK" << endl;
        cout << "D::i=" << pd3->i <<endl;}
    D* pd4 = dynamic_cast<D*>(pb);
    if (pd4 != nullptr){ /// 转换正确
        cout << "dynamic_cast, B*(D) --> D*: OK" << endl;
        cout << "D::i=" << pd4->i <<endl;
    }
    return 0;
}
```

6.7 多态 (Polymorphism)

多态: 按照**基类**的接口定义，调用**指针或引用**所指对象的接口函数，函数执行过程因对象**实际**所属**派生类**的不同而呈现不同的效果的现象。可以提高程序的可复用性、可拓展性和可维护性。

产生多态效果的条件：继承 && 虚函数 &&（引用 || 指针）
非虚函数或类的对象直接调用函数，均在**编译**时完成绑定，无法呈现“多态”效果。

应用：TEMPLATE METHOD设计模式

> 在接口的一个方法中定义算法的骨架
> 将一些步骤的实现延迟到子类中
> 使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤

模板方法是一种**源代码重用**的基本技术，在类库的设计实现中应用十分广泛。

```
class Animal {
public:
    void action() { ///复用基类借口
        speak();
```

```
        motion();
    }
    virtual void speak() { cout << "Animal speak" << endl; }
    virtual void motion() { cout << "Animal motion" << endl; }
};

class Bird : public Animal {
public:
    void speak() { cout << "Bird singing" << endl; }
    void motion() { cout << "Bird flying" << endl; }
};
class Fish : public Animal {
public:
    void speak() { cout << "Fish cannot speak ..." << endl; }
    void motion() { cout << "Fish swimming" << endl; }
};

int main() {
    Bird bird; Fish fish;
    fish.action();   ///不同调用方法
    bird.action();

    Animal *pBase1 = new Fish;
    Animal *pBase2 = new Bird;
    pBase1->action(); ///同一调用方法，根据
    pBase2->action(); ///实际类型完成相应动作
    return 0;
}
```

7. 模板与STL、STL进阶

7.1 函数模板和类模板

继承与组合提供了**重用对象代码**的方法，而C++的模板特征提供了**重用源代码**的方法。

1. 函数模板

```
template<typename T>
ReturnType Func(Args);
///两个变量相加的函数模板:
template<typename T>
T sum(T a, T b) {return a + b;}
template<class T> //两者皆可
T sum(T a, T b) {return a + b;}

cout << sum(9, 3);
cout << sum(2.1, 5.7);
```

函数模板在调用时，编译器能自动推导出实际参数的类型（这个过程叫做**实例化**）
模板可以支持自定义类型，但调用类型需要**满足函数的要求**：如运算符重载等。
当多个参数的类型不一致时，无法推到：`cout << sum(9, 2.1); //编译错误`
可以手工指定调用类型：`sum<int>(9, 2.1);`

2. 模板原理

对模板的处理是在**编译期**进行的，每当编译器发现对模板的一种参数的使用，就生成对应参数的一份代码。因此：模板库必须在**头文件**中**实现**（声明和定义在一起），不可以分开编译

3. 类模板

在定义类时也可以将一些**类型信息**抽取出来，用**模板参数**来替换，从而使类更具通用性。这种类被称为**类模板**。

```
template<typename T>
class A {  ///其中<typename T>即为“模板参数”
    T data;
public:
    A(T _data): data(_data) {}
    void print() { cout << data << endl; }
};
///或采用类外定义：
template<typename T>
void A<T>::print() { cout << data << endl;}

int main() {
    A<int> a(1);
    a.print();
    return 0;
}
```

类模板的“模板参数”:
类型参数：使用**typename**或**class**标记
非类型参数：整数，枚举（**enum**），指针/引用（用于对象或函数）。其中，无符号整数比较常用。
所有模板参数必须在**编译期**确定，**不可以使用变量。**

```
template<typename T, unsigned size>
class array {
    T elems[size];
};

int main(){
    int n = 5;
    ///array<char, n> array0; //不能使用变量
    const int m = 5;
    array<char, m> array1; //可以使用常量
    array<char, 5> array2; //或具体数值
    return 0;
}
```

```
///类模板示例：排序
```

```cpp
#include <algorithm>

template<class T, unsigned size>
class MyArr
{
    T data[size];
public:
    void sort() {
        for (int i = 0; i < size; i++) { //选择排序
            for (int j = i + 1; j < size; j++) {
                if (data[i] > data[j])
                    std::swap(data[i], data[j]); //交换两者位置
            }
        }
    }
    void input(){
        for(int i = 0; i < size; i++)
            std::cin >> data[i];
    }
    void output(){
        for(int i = 0; i < size; i++)
            std::cout << data[i] << " ";
        std::cout << std::endl;
    }
};

int main()
{
    MyArr<int, 5> arr_a;
    arr_a.input();
    arr_a.sort();
    arr_a.output();

    MyArr<float, 5> arr_b;
    arr_b.input();
    arr_b.sort();
    arr_b.output();
    return 0;
}
```

4. 成员函数模板

普通类的成员函数，也可以定义为**模板函数**

```cpp
class normal_class {
public:
    int value;
    template<typename T>
    void set(T const& v) {
        value = int(v); ///类内定义
    }
    template<typename T> T get(); ///类外定义
```

```
};
template<typename T>
T normal_class::get() { return T(value);}
```

模板类的成员函数，也可有**额外的模板参数**

```
template<typename T0>
class A {
    T0 value;
public:
    template<typename T1>
    void set(T1 const& v) {
        value = T0(v); /// 将T1转换为T0储存
    }         /// 在类内定义
    template<typename T1> T1 get();
};
template<typename T0> template<typename T1> ///注意写法!
T1 A<T0>::get() { return T1(value);} /// 类外定义， 将T0转换为T1返回

int main() {
    A<int> a;
    a.set(5);      //自动推导5为整数类型
    double t = a.get<double>();    //手动指定返回值类型
    return 0;
}
```

多个参数的模板

```
template<typename T0, typename T1> class A {}; ///类模板
template<typename T0, typename T1> void func(T0 a1, T1 a2) {} ///函数模板
```

5. 模板的特化

函数模板的特化

只有**全特化**

```
template<>
int sum<int>(int a, int b) {}
///也可以写成: int sum(int a, int b) {}
///比较:
int sum(int a, int b) {} ///本质上是函数重载
```

类模板的特化

有**全特化**和**偏特化**，可以特化为①绝对类型，②引用或指针类型，③另一个类模板

```
template<class T1, class T2>
class A {};

template<class T1>
```

```
class A<T1, int> {}; ///偏特化
```

6. 模板与多态

相同点：模板使用**泛型标记**，使用同一段代码，来关联不同但相似的特定行为，最后可以获得不同的结果。模板也是**多态**的一种体现。

不同点: 模板的关联是在**编译期**处理，称为**静多态**，基于继承和虚函数的多态在**运行期**处理，称为**动多态**

> **静多态**：模板
> 往往和函数重载同时使用
> 高效，省去函数调用
> 编译后代码增多
>
> **动多态**：继承，虚函数
> 运行时，灵活方便
> 侵入式，必须继承
> 存在函数调用

7.2 命名空间

namespace关键字：避免标识符的命名发生冲突、用于控制标识符作用域的关键字
std命名空间：标准C++库中所包含的所有内容（包括常量、变量、结构、类和函数）
`cout, cin, vector, set, map`

定义和使用命名空间

注意：任何情况下，都不应出现命名冲突

```
namespace A {
    int x, y;
}
///使用using声明简化命名空间使用
///1. 使用整个命名空间
using namespace A;
x = 3; y = 6;
///2. 使用部分成员：所选成员可直接使用
using A::x;
x = 3; A::y = 6;
```

【STL初步】

关于STL的文档和例子

STL/Standard Template Library（标准模板库）：C++软件库，被容纳于C++标准程序库C++ Standard Library中。其中包含四个组件，分别为算法, 容器，函数，迭代器。基于**模板**编写。关键理念：将“在数据上执行的操作”与“要执行操作的数据”分离。

STL的命名空间是std

一般使用`std::name`来使用STL的函数会对象

也可以使用using namespace std来引入STL的命名空间（不推荐在大型工程中使用，容易污染命名空间）

7.3 STL容器

容器是包含、放置数据的工具，通常为数据结构。包括：简单容器 (simple container)（pair, tuple），序列容器 (sequence container)（vector, list），关系容器 (associative container)（set, map）

序列容器与关联容器的区别：

> 序列容器中的元素**有顺序**，可以**按顺序**访问
> 关联容器中的元素**无顺序**，可以**按数值/大小**访问

vector中插入删除操作会使操作位置之后全部的迭代器失效，其他容器中只有被删除元素的迭代器失效。

1. pair

pair：最简单的容器，有两个单独数据组成；在map中大量使用

```
template<class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    //若干其他函数
};
```

通过first, second两个成员变量获取数据。

```
std::pair<int, int> t;
t.first = 4; t.second = 5;
```

1. 创建：使用函数**make_pair**

   ```
   auto t = std::make_pair("abc", 7.8); ///优势：自动推导成员类型
   ```

2. 支持小于、等于等比较运算符
   先比较first，后比较second；要求成员类型支持比较（实现比较运算符重载）
   `std::make_pair(1, 4) < std::make_pair(2, 3);`

2. tuple

tuple：C++11新增，pair的拓展，由若干成员组成的**元组**类型

```
template<class ... Types> class tuple;
```

1. 创建：使用函数**make_tuple**；**tie**函数——返回左值引用的元组

   ```
   auto t = std::make_tuple("abc", 7.8, 123, '3');
   std::string x; double y; int z;
   ```

```
std::tie(x, y, z) = std::make_tuple("abc", 7.8, 123);
```

(*) 创建：**forward_as_tuple**函数——返回右值引用的元组

2. 通过**std::get**函数来获取数据
   【注意】下标需要在编译时确定，不能设定运行时可变的长度（variable i），不能当做数组使用`int i = 0; v = std::hey<i>(tuple); ///编译错误`

```
auto t = std::make_tuple("abc", 7.8, 123, '3');
auto v0 = std::get<0>(t);
auto v1 = std::get<1>(t);
```

3. 用于函数多返回值的传递

```
#include <tuple>
std::tuple<int, double> f(int x) {
    return std::make_tuple(x, double(x)/2);
} ///作为tuple的特例，pair可用于两个返回值的传递

int main() {
    int xval;
    double half_x;
    std::tie(xval, half_x) = f(7);
    return 0;
}
```

(*) `std::tuple`类重载了赋值运算符`=`，`tuple`对象的长度不是在运行时才确定的

3. vector

vector：会自动扩展容器的**数组**，以**循序 (Sequential)** 的方式维护变量集合；STL中最基本的序列容器，提供有效、安全的数组以替代C语言中原生数组；允许直接以**下标**访问（高速）。

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

vector原理：vector是会**自动扩展容量**的数组。除了size，另保存capacity（最大容量限制）。如果size达到了capacity，则另申请一片capacity*2的空间，并整体迁移vector内容。时间复杂度为均摊O(1)。整体迁移过程使多有迭代器失效。

1. 创建：`std::vector<int> x;`
2. 当前数组长度：`x.size();`
3. 清空：`x.clear();`
4. 在末尾添加/删除（高速）：`x.push_back(1); x.pop_back();`
5. 初始化capacity：`x.reserve(100);`
6. 删除`vector`中符合特定条件的元素

```
for (auto it = vec.begin(); it != vec.end(); ) { //注意没有++it
```

```
    if (condition) it = x.erase(it); //返回++it
    else it++; //理解else的使用
}
```

移除位于pos的元素：`x.erase(it);` it为pos位置的迭代器
移除范围[first, last) 中的元素：`x.erase(++x.begin(), --x.end());` 只剩首尾

7. （使用**迭代器**）在中间添加/删除（低速）：`x.insert(x.begin() + 1, 5); x.erase(x.begin() + 1);`

迭代器

迭代器：一种检查容器内元素并遍历元素的**数据类型**
提供一种方法**顺序**访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示
为遍历不同的聚合结构（需拥有相同的基类）提供一个**统一**的接口
使用上类似**指针**

```
template<class T, class Allocator = std::allocator<T>>
class vector {
    class iterator {
        ...
    }
}
```

定义迭代器类型变量：`vector<int>::iterator iter;`

返回vector中第一个元素的迭代器：`x.begin();`
返回vector中最后一个元素**之后的位置**的迭代器：`x.end();`
begin和end函数构成所有元素的**左闭右开**区间

下一个元素：`++iter`
上一个元素：`--iter`
下n个元素：`iter += n`
上n个元素：`iter -= n`

访问元素值——**解引用运算符**：`*iter = 5;`
解引用运算符返回的是左值引用（可以取地址）

迭代器移动-与整数做加法：`iter += 5;`
元素位置差-迭代器相减：`int dist = iter1 - iter2;`
其本质都是**重定义运算符**

遍历vector：`for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)`
常用auto简化代码：`for (auto it = vec.begin(); it != vec.end(); ++it;)`，it理解为指向元素的指针
按范围遍历vector（C++11）：`for (auto & x : vec)`，直接利用vec中元素x

完整代码示例：

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

int main() {
```

```
    vector<int> vec = {1,2,3,4,5};
    cout << vec.end() - vec.begin() << endl; //输出: 5
    for (auto it = vec.begin(); it != vec.end(); ++it){
        *it *= 2; cout << *it << endl;
    } //输出: 2 4 6 8 10
    return 0;
}
```

迭代器的失效:

当迭代器不再指向本应指向的元素时，称此迭代器**失效**

绝对安全的准则：修改过容器后，不使用之前的迭代器

1. 看做纯粹的指针
   - 调用**insert/erase**后，所修改位置之后的所有迭代器失效
   - 调用**push_back**等修改vector大小的方法时，可能会使所有的迭代器失效（push_back到了一定程度之后，可能会造成数组的整体移动，导致所有的内存地址发生改变）

   [push_back对迭代器是否失效的影响](http://cplusplus.com/reference/vector/vector/push_back/)

2. 在遍历的时候增加元素，可能会导致迭代器失效

   ```C++
   for (auto it = vec.begin(); it != vec.end(); ++it)
         vec.push_back(*it); //Error
   ```

3. 使用**erase**删除元素，被删除元素及之后的所有元素均会失效

   vector vec = {1, 2, 3, 4, 5};
   auto first = vec.begin(); //first指向1
   auto second = vec.begin() + 1;
   auto third = vec.begin() + 2;
   auto ret = vec.erase(second); //second和third失效, ret指向3

自定义一个迭代器:

`for (auto batch : D)` == `for (auto it = D.begin(); it != D.end(); ++it))`

```
class D { //类D的迭代器
private:
    Datatype pD;
public:
    Class Iterator { //迭代器Iterator是一个“类中类”
    private:
        DataType * _ptr;
    public:
        Iterator(DataType * ptr) : _ptr(ptr) {} //构造函数
        Iterator operator++() { //重载运算符++
            _ptr++;
            return *this;
        }
        bool operator!=(const Iterator & other) const { //重载运算符!=
            return _ptr != other._ptr;
```

```
        }
        const DataType & operator*() const { //重载运算符*
            return *_ptr;
        }
    };
    Iterator begin() const { //定义函数begin()
        return Iterator(pD);
    }
    Iterator end() const { //定义函数end()
        return Iterator(pD + len);
    }
};
```

```
//一个自定义的更完整的迭代器可以是这样的:
//以下代码确定了一个迭代器的“基调”

//迭代器的分类
1. Input/Output iterator（输入/输出迭代器）：可以执行单次的单向遍历，只能进行输入/输出操作。
2. Forward iterator（前向迭代器）：具有输入迭代器的所有功能；如果不是常量迭代器（constant iterator，不能改变所指向的数据的值），则也具有输出迭代器的功能。只能进行单向遍历。所有标准容器都至少支持前向迭代器类型。
3. Bidirectional iterator（双向迭代器）：具有前向迭代器的所有功能，但也可以向后迭代，即可以进行双向的遍历。
4. Random-access iterator（随机访问迭代器）：具有双向迭代器的所有功能，但还可以“非顺序“地访问容器的元素，即迭代器可以通过一个偏移量直接访问距离较远的元素（直观来说就是类似于指针和整数之间的加减）。“随机”这个说法或许不太恰当，容易误解，它其实只是强调非顺序地进行访问。

#include <iterator> // For std::forward_iterator_tag
#include <cstddef>  // For std::ptrdiff_t

struct Iterator
{
    using iterator_category = std::forward_iterator_tag;
    using difference_type   = std::ptrdiff_t;
    using value_type        = int;
    using pointer           = int*;  // or also value_type*
    using reference         = int&;  // or also value_type&
};
```

8. 排序

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

vector<A> vec;
bool comp(A a, A b) {
    return (a.data > b.data) //从大到小排序
}
sort(vec.begin(), vec.end(), comp);
```

4. 链表容器 list

链表容器 list：底层实现是**双向链表**

```
template<class T, class Allocator = std::allocator<T>>
class list;

std::list<int> l;
```

1. 插入前端：`l.push_front(1);`
2. 插入末端：`l.push_back(2);`
3. 查询：`std::find(l.begin(), l.end(), 2);` 返回迭代器
4. 插入指定位置：`l.insert(it, 4);` it为迭代器
5. 不支持下标等随机访问
   支持在任意位置**高速**插入/删除数据
   其访问主要依赖迭代器
   插入和删除操作**不会导致**迭代器失效（除指向被删除的元素的迭代器外）

5. 无序集合 set

无序集合 set：**不重复元素**构成的**无序**集合

```
template<class Key,
         class Compare = std::less<Key>,
         class Allocator = std::allocator<Key>>
class set;

std::set<int> s;
```

内部**按大小顺序排序**，比较器由函数对象**Compare**完成
注意：**无序**是指不保持**插入顺序**，容器内部排列顺序是根据元素大小排序的。

1. 插入（不允许出现重复元素）：`s.insert(val);`
2. 查询值为val的元素：`s.find(val);` 返回迭代器
3. 删除：`s.erase(s.find(val));` 导致迭代器失效
4. 统计：`s.count(val);` val的个数，总是0或1

6. 关联数组 map

关联数组 map：每个元素由两个数据项组成，map将一个数据项映射到另一个数据项中

```
template<class Key,
         class T,
         class Compare = std::less<Key>,
         class Allocator = std::allocator<std::pair<const Key, T>>
class map;
//其值类型为pair<Key, T>
```

map中的元素key必须互不相同
可以通过下标访问（即使key不是整数），下标访问时如果元素不存在，则创建对应元素
也可以使用**insert**函数进行插入：

```
#include <string>
#include <map>
```

```cpp
int main() {
    std::map<std::string, int> s;
    s["Monday"] = 1;
    s.insert(std::make_pair(std::string("Tuesday"), 2));
    return 0;
}
```

1. 查询键为key的元素：`s.find(key);` 返回迭代器
2. 统计键为key的元素个数：`s.count(key);` 返回0或1
3. 删除：`s.erase(s.find(key));` 导致被删元素的迭代器失效

map常用作稀疏数组或以字符串为下标的数组

> **set**和**map**所用到的数据结构都是**红黑树**（一种二叉平衡树）
> 其几乎所有操作复杂度均为O(logn)

总结：选择合适的容器

1. **算法复杂度**：对于序列容器而言，如果在序列中间存在频繁的插入或删除操作，使用**list**，否则使用**vector**（或**deque**）
2. **元素的顺序**：如果需要在容器的任意位置插入新元素，需要选择**序列**容器而不是**关联**容器
3. **元素查找速度**：如元素的查找速度是关键的考虑因素，可以考虑排序的**vector**或关联容器**set**、**map**等
4. **迭代器、指针或引用失效**：如果希望在元素插入和删除操作后，迭代器、指针或引用失效的情况尽可能少出现，可以考虑使用**list**和关联容器**set**、**map**等

7.4 String 字符串处理

字符串是char数组，string类型使得在没有提前确认字符串长度时也可以定义一个字符串（类似`vector<char>`）

1. 允许简洁的拼接操作：`string fullname = firstname + " " + lastname;`
   使用管用的输入输出方法：`cout << fullname << endl;`
2. 构造方式

```cpp
string s0("Initial String"); //从C风格字符串构造
string s1; //默认空字符串
string s2(s0, 8, 3); //截取“str”，index从8开始，长度为3
string s3("Another character sequence", 12); //截取“Another char”
string s4(10, 'x'); //复制字符：xxxxxxxxxx
string s5(s0.begin(), s0.begin() + 7); //复制截取：Initial
```

3. 转换为C风格字符串：`str.c_str()` 注意返回值为**常量字符指针（const char*）**，不能修改
4. 和**vector**类似

   访问/修改元素：`cout << str[1]; str[1] = 'a';`
   查询长度：`str.size();`
   清空：`str.clear();`
   查询是否为空：`str.empty();`

迭代访问：`for (char c : str);`
向尾部增加：`str.push_back('a'); str.append(s2);`

不同之处：
查询长度也可以使用`str.length();`，与`str.size();`返回值相同
向尾部增加也可以使用`str += 'a';`或者`str += s2;`

5. 三种输入方式
读取可见字符直到遇到空格：`cin >> firstname;`
读一行：`getline(cin, fullname);`
读到指定分隔符为止：`getline(cin, fullnames, '#');`
6. 拼接与比较
拼接：`string fullname = firstname + " "  + lastname;`
[注意]：拼接的**时间复杂度**为生成的字符串长度
使用循环和operator+拼接string类的多个对象，假设长度为常数，则其时间复杂度与对象个数的平方成正比。
拼接多个字符串最好使用**operator+=**或**stringstream**或**str.append(suffix)**
比较：按照**字典序**比较字符串大小
`string a = "alice", b = 'bob'; a < b //True`
7. 数值类型字符串化：`to_string(3.1415926)` //"3.141593"注意精度损失
8. 字符串转数值类型：

```
int a = stoi("2001"); //a = 2001
std::string::size_type sz; //代表长度的类型 无符号整数
int b = stoi("50 cats", &sz); //b = 50, sz = 2 代表读入长度
   //字符串首个子串必须是整型数字（在这个例子里），否则报错
int c = stoi("40c3", nullptr, 16); //c = 16579 十六进制
int d = stoi("0x7f", nullptr, 0); //d = 127 自动检查进制
double e = stod("34.5"); //e = 34.5
```

7.5 iostream/fstream/sstream 输入输出流

1. iostream

iostream: 多重继承自**istream**和**ostream**

[回忆]：重载输出流运算符

```
friend ostream& operator<<(ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}
```

ostream和cout

ostream即output stream，是STL库中所有**输出流**的**基类**
它重载了针对**基础类型**的输出流运算符**<<**
统一了**输出接口**，改善了**C**中输出方式混乱的状况（对比：`printf("%d %f %s", 1, 2.3, "hello");`）

cout是STL中内建的一个ostream对象
它会将数据送到**标准输出流**（一般是屏幕）

```
//实现自己的ostream
class ostream {
public:
    ostream& operator<<(char c) {
        printf("%c", c); //单个字符
        return *this;
    }
    ostream& operator<<(const char* str) {
        printf("%s", str); //字符串常量
        return *this;
    }
} cout;
int main() {
    cout << "hello" //调用第二个函数返回c1（cout的引用）
         << " " //调用第一个函数返回c2（cout的引用）
         << "world"; //调用第二个函数
    return 0;
}
```

1. 格式化输出

```
#include <iomanip>
cout << fixed << 2018.0 << " " << 0.0001 << endl;
    //浮点数 -> 2018.000000 0.000100
cout << scientific << 2018.0 << " " << 0.0001 << endl;
    //科学计数法 -> 2.018000e+03 1.000000e-04
cout << defaultfloat; //还原默认输出格式
cout << oct << 12 << " " << hex << 12 << endl;
    //八进制输出 -> 14  十六进制输出 -> c
cout << dec; //还原十进制
cout << setw(3) << setfill('*') << 5 << endl;
    //设置对齐长度为3,对齐字符为* -> **5
cout << setprecision(2) << 1.05 << endl;
    //保留2位精度，输出1.1

//例：setprecision的实现方式（一种编译器）
class setprecision {
private:
    int precision;
public:
    setprecision(int p) : precision(p) {}
    friend class ostream;
}
//setprecision(2)是一个类的对象
```

2. **流操纵算子**（stream manipulator)
   流操纵算子：借助辅助类，设置成员变量
   1）**setprecision**

```
class ostream {
private:
    int precision; //记录流的状态
public:
    ostream& operator<<(const setprecision &m) {
        precision = m.precision;
        return *this;
    }
} cout;
```

   2）**endl**
   缓冲区：目的是减少外部读写次数；写文件时，只有清空缓冲区或关闭文件才能保证内容正确输入

```
ostream& endl(ostream& os) {
    os.put('\n'); //输出'\n'
    os.flush(); //清空缓冲区
    return os;
} //可以调用endl(cout)

//一种实现方式
ostream& operator<<(ostream& (*fn)(ostream&)) {
    return (*fn)(*this);
}
```

3. 观察**ostream**的复制构造函数

```
ostream(const ostream&) = delete;
ostream(ostream&& x);
//禁止复制，只允许移动
//仅使用cout一个全局变量
```

2. ifstream & ofstream 文件输入输出流

ifstream和**ofstream**: **istream**和**ostream**的子类，功能是从文件中读入数据

1. 打开文件
   `ifstream ifs("input.txt");`
   `ifstream ifs("binary.bin", ifstream::binary);` 以二进制形式打开文件
   `ifstream ifs; ifs.open("file"); ... ifs.close();`
2. 读入文件示例

```
#include <iostream>
#include <string>
#include <cctype>
#include <fstream>
```

```cpp
using namespace std;

int main() {
    ifstream ifs("input.txt");
    while(ifs) { //判断文件是否到末尾 利用了重载的bool运算符
        ifs >> ws;  //除去前导空格 ws也是流操纵算子 ws=whitespace
        int c = ifs.peek(); //检查下一个字符，但不读取
        if (c == EOF) break; //EOF=end of file
        if (isdigit(c)) { //<cctype>库函数
            int n;
            ifs >> n;
            cout << "Read a number: " << n << endl;
        }
        else {
            string str;
            ifs >> str;
            cout << "Read a word: " << str << endl;
        }
    }
    return 0;
}
```

3. 读入行：`getline(ifs, str);`
   读取一个字符：`get();`
   丢弃n个字符，或者直至遇到delim分隔符：`ignore(int n = 1, int delim = EOF);`
   查看下一个字符：`peek();`
   返还一个字符：`putback(char c);`，`unget();`
4. **istream与scanf**

   `scanf("%d %hd %f %lf %s", &i, &s, &f, &d, name);` 不同类型要使用不同的标识符
   注释：`d:int, hd:short, f:float, lf:long double, s:string`
   scanf的安全性较弱（可能写入非法内存），可拓展性不强，性能较差（运行期间需要对格式字符串进行解析，而**istream**在编译期间已经解析完毕）

3. stringstream 字符串输入输出流

stringstream: **iostream**的子类，实现了输入输出流双方的接口

stringstream在对象内部维护了一个**buffer**，使用流输出函数可以将数据写入**buffer**，使用流输入函数可以从**buffer**中读取数据。一般用于程序内部的字符串操作。

1. 构造方式：`stringstream ss;` 空字符串流
   `stringstream ss(str);` 以字符串初始化流

```cpp
#include <sstream>
using namespace std;

int main() {
    stringstream ss;
    ss << "10";
    ss << "0 200";
```

```
    int a, b;
    ss >> a >> b;       //a=100 b=200
    return 0;
}
```

2. 获取**stringstream**的**buffer**： `ss.str();`
   [注意]：**buffer**的内容并不是未读取的内容
   `ss.clear()` 无法清空缓冲区（作用仅仅是清除所有的error state），应该使用 `ss.str("")` 实现清空

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

int main() {
    stringstream ss;
    ss << "100 200";
    cout << ss.str() << endl;  //输出"100 200"
    int a;
    ss >> a; // a = 100
    cout << ss.str() << endl;  //输出"100 200"
    ss >> b; // b = 200
    return 0;
}
```

Image unavailable in the original source: note5.png

3. 实现一个类型转换函数
   to_string：转换为字符串
   stoi：转换为整数
   其他类型： `string x = convert<string>(123); int y = convert<int>("456");`

```
template<class outtype, class intype>
outtype convert(intype val) {
    static stringstream ss; //使用静态变量避免重复初始化
    ss.str(""); //清空缓冲区
    ss.clear(); //清空状态位（不是清空内容）
    ss << val;
    outtype res;
    ss >> res;
    return res;
}
```

4. sstream与字符串快速读入

   将 `sentence` 按照空格切分为若干字符串 `word`

```
#include <sstream>
string sentence, word;
istringstream tmp(sentence);
```

```
while (tmp >> word) {
    ...
}
```

7.6 字符串处理与正则表达式

正则表达式：由字母和符号组成的特殊文本，搜索文本时定义的一种规则

1. 正则表达式的三种模式

1. 匹配：判断整个字符串是否满足条件
   [示例]：`^[a-z0-9_]{3,15}$` 表示3-15位的小写字母与数字组合
2. 搜索：符合正则表达式的子串
   [示例]：在“q123e456w”中找出所有数字串`[0-9]+`，搜索结果为123,456
3. 替换：按规则替换字符串的子串
   [示例]：给定“q123e456w”将所有数字串替换为(number)，替换结果为：q(123)e(456)w

2. 编写正则表达式

正则表达式辅助工具

1. 字符代表其本身
2. 匹配的单个字符在某个范围中
   `[a-z]`：匹配所有**单个**小写字母
   `[0-9]`：匹配所有**单个**数字
3. 连用匹配字符串组合
   `[a-z][0-9]`：匹配所有字母+数字的组合，比如a1，b9
   `[Tt]he`：匹配所有The和the
4. 字符簇-范围取反
   `[^a-z]`：匹配所有非小写字母的单个字符
   `[^c]ar`：The car parked in the garage.
   `^[^0-9][0-9]$`：匹配长度为2的内容，且第一个不为数字，第二个位数字
5. `x{n, m}`代表前面内容出现次数重复n~m次
   `a{4}`：匹配aaaa
   `a{2, 4}`：匹配aa, aaa, aaaa
   `a{2,}`：匹配长度大于等于2的a
   `[a-z]{5-12}`：长度为5-12的英文字母组合
   `.{5}`：长度为5的字符
6. 特殊字符
   `\d`：等价于`[0-9]`，匹配所有单个数字
   `\w`：匹配字母、数字、下划线，等价于`[a-zA-Z0-9_]`
   `.`：匹配除**换行**以外任意字符
   [示例]：`.ar`：The car parked in the garage.
   `\.`：可表示匹配句号
   [示例]：`ge\.`：The car parked in the garage.

`+`：前一个字符至少连续出现**1**次及以上

[示例]：`a\w+`：The car parked in the garage.

`\n`：换行符

`\t`：制表符

`\D`： 等价`[^0-9]`，匹配所有单个非数字

`\s`： 匹配所有**空白字符**，如`\t`,`\n`

`\S`： 匹配所有非空白字符

`\w`： 匹配非字母、数字、下划线，等价`[^a-zA-Z0-9_]`

`^`代表字符串开头，`$`代表字符串结尾

[示例]：`^\t`只能匹配到以制表符开头的内容

[示例]：`^bucket$`只能匹配到只含bucket的内容

`?`：出现0次或1次 **（懒惰模式）**

[示例]：`[T]?he`：The car parked in the garage.

`+`：至少连续出现1次及以上 **（独占模式）**

[示例]：`c.+e`：The car parked in the garage.

`*`：至少连续出现0次及以上 **（贪婪模式）**

[示例]：`[a-z]*`：The car parked in the garage.

7. 或连接符

匹配模式可以使用`|`进行连接：

`(Chapter|Section) [1-9][0-9]?`可以匹配Chapter 1, Section 10等

`0\d{2}-\d{8}|0\d{3}-d{7}`可以匹配010-12345678,0376-2233445等

`(c|g|p)ar`：The car parked in the garage.

使用`()`改变优先级：

`m|food`可以匹配m或者food

`(m|f)ood)`可以匹配mood或者food

`(T|t)he|car`：The car parked in the garage.

3. 正则表达库 regex

1. 创建一个正则表达式对象：`regex re("^[1-9][0-9]{10}$")` 11位数

`^$`：确保匹配在一个**match**里

[注意]：C++的字符串中`\`也是转义字符，如果需要创建正则表达式`\d+`，应该写成`regex re("\\d+")`

2. **原生字符串 R("")**

原生字符串可以**取消转义**，保留字面值

[语法]：`R"(str)"`表示str的字面值

[示例]：`"\\d+"` = `R"(\d+)"` = `\d+`，`string str = R"(Hello换行World)";` -> `str = "Hello\nWorld"`

1）匹配与捕获 regex_match

1. **regex_match(s, re)**

匹配：`regex_match(s, re)` 询问字符串s是否能完全匹配正则表达式re

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;
```

```
int main() {
    string s("subject");
    regex e("sub.*");
    smatch sm;
    if (regex_match(s,e))
        cout << "matched" << endl;
    return 0;
}
```

2. **regex_match(s, sm, re)**
   捕获和分组：使用()进行标识，每个标识的内容被称作分组
   --正则表达式匹配后，每个分组的内容将被捕获
   --用于提取关键信息，例如`version(\d+)`即可捕获版本号
   `regex_match(s, result, re)`：询问字符串s是否能完全匹配正则表达式re，并将捕获结果储存到result中，result需要是**smatch**类型的对象

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;

int main () {
    string s("version10");
    regex e(R"(version(\d+))");
    smatch sm;
    if (regex_match(s,sm,e)) {
        cout << sm.size() << " matches\n"; //输出：2 matches
        cout << "the matches were:" << endl;
        for (unsigned i=0; i<sm.size(); ++i) {
            cout << sm[i] << endl; //输出：version10 10
        }
    }
    return 0;
}
```

   分组会按顺序标号
   0号永远是匹配的字符串本身
   (a)(pple)：0号为apple，1号为a，2号为pple
   用`(sub)(.*)`匹配subject：0号为subject，1号为sub，2号为ject
   如果需要括号，又不要捕获该分组，可以使用`(?:pattern)`
   用`(?:sub)(.*)`匹配subject：0号为subject，1号为ject

   smatch 语法：
   `smatch sm;`：声明smatch对象
   `if (sm.ready())`：如果成功匹配返回true，否则返回false
   `sm.prefix()`：返回锁定match的前缀子串（match_result类型）
   `sm.suffix()`：返回锁定match的后缀子串（match_result类型）
   `sm.suffix.str()`：返回锁定match的后缀子串（string类型）

2）搜索 regex_search

regex_search(s, sm re)

搜索字符串s中能够匹配正则表达式re的**第一个**子串，并将结果存储在result中`regex_search(s, result, re)`
--对于该子串，分组同样会被捕获

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;

int main() {
    string s("this subject has a submarine");
    regex e(R"((sub)([\S]*))");
    smatch sm;
    //每次搜索时当仅保存第一个匹配到的子串
    while(regex_search(s,sm,e)){
        for (unsigned i=0; i<sm.size(); ++i)
            cout << "[" << sm[i] << "] ";
        cout << endl;
        s = sm.suffix().str(); //返回锁定match的后缀子串
    }
    return 0;
}
//输出:
//[subject] [sub] [ject]
//[submarine] [sub] [marine]
```

3) 替换 regex_replace

regex_replace(s, re, s1)

替换字符串s中**所有**匹配正则表达式re的子串，并替换成s1：`regex_replace(s, re, s1)`
s1可以是普通文本，也可以是一些**特殊符号**，代表捕获的分组
--`$&`代表re匹配的子串
--`$1, $2`代表re匹配的第1/2个分组

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;

int main() {
    string s("this subject has a submarine");
    regex e(R"((sub)([\S]*))");
    //regex_replace返回值即为替换后的字符串
    cout << regex_replace(s,e,"SUBJECT") << endl;
    //$&表示所有匹配成功的部分，[$&]表示将其用[]括起来
    cout << regex_replace(s,e,"[$&]") << endl;
    //$i输出e中第i个括号匹配到的值
    cout << regex_replace(s,e,"$1") << endl;
```

```
    cout << regex_replace(s,e,"$2") << endl;
    cout << regex_replace(s,e,"$1 and [$2]") << endl;
    return 0;
}
//输出:
//this SUBJECT has a SUBJECT
//this [subject] has a [submarine]
//this sub has a sub
//this ject has a marine
//this sub and [ject] has a sub and [marine]
```

【例题】学生信息整理

```
void extract(string input) {
    smatch sm;
    regex get_name(R"((My name is |I am )(\w+)\.)");
    regex get_date(R"((\d{4})[\.-](\d{1,2})[\.-](\d{1,2}))");
    regex get_mobile(R"([1-9]\d{10})");
    regex get_email(R"([\w]+@[\w\.]+)");
    if (std::regex_search(input, sm, get_name))
        cout << sm[2] << endl;
    int date[3] = {0};
    if (regex_search(input, sm, get_date)) {
        for (int i = 1; i <= 3; i++)
            date[i - 1] = stoi(sm[i]);
        cout << date[0] << "." << date[1] << "." << date[2] <<endl;
    }
    if (regex_search(input, sm, get_mobile))
        cout << sm[0] << endl;
    if (regex_search(input, sm, get_email))
        cout << sm[0] << endl;
}

#include <iostream>
#include <regex>
#include <string>
using namespace std;

void extract(string input);
int main() {
    string str("I am zhangshuaishuai. \
        I was born on 2000.10.2. \
        My phone number is 18866667777 and you can also \
        reach me by my email: zhangss@tsinghua.edu.cn");
    extract(str);
    return 0;
}

//输出:
//zhangshuaishuai
//2000.10.2
//18866667777
```

```
//zhangss@tsinghua.edu.cn
```

更多内容

1. 预查
   正向预查`(?=pattern) (?!pattern)`
   反向预查`(?<=pattern) (?<!pattern)`
2. 后向引用
   `\b(\w+)\b\s+\1\b`匹配重复两遍的单词，比如go go 或 kitty kitty
3. 贪婪与懒惰
   默认多次重复为贪婪匹配，即匹配次数最多
   在重复模式后加`?`可以变为懒惰匹配，即匹配次数最少

7.7 函数对象和智能指针

1. 函数对象

```
void (*func)(int &); //函数指针的声明
//void: 返回值
//*: 指针符号
//func: 声明的变量名
//(int&): 参数列表

//使用auto自动推断类型
auto func = flag == 1 ? increase : decrease; //inc, dec是两个函数
for (int &x : arr) {func(x);}
```

1. 数组名 = 指向数组第一个元素的指针
   函数名 = 指向函数的指针（和数组类似）
2. 排序函数

```
template <class Iterator, class Compare>
void sort (Iterator first, Iterator last, Compare comp);

bool comp(int a, int b) {
    return a > b; //降序排序
}

template<class T>
class greater {
    bool operator()(const T &a, const T &b) const {
        return a > b; //降序排序
    }
};

#include <algorithm>
#include <iostream>
int arr[5] = {5,2,3,4,1};
std::sort(arr, arr + 5, comp); //comp是函数指针
```

```
    //简化形式：std::sort(arr, arr + 5);
auto func = greater<int>();
std::sort(arr, arr + 5, greater<int>()); //greater是函数对象
```

1）实际上，**Compare**就是**comp**的类型。
Compare是模板类型，可以接受函数指针/函数对象
函数指针：`bool (*)(int, int)`
函数对象：`greater<int>()`
2）STL提供了预定义的比较函数（`#include <functional>`）
从小到大：`sort(arr, arr + 5, less<int>())`
从大到小：`sort(arr, arr + 5, greater<int>())`
`greater<int>()` *为什么带括号?*
3）`greater<int>()` 是一个对象
`greater` 是一个模板类
`greater<int>` 用int实例化的类
`greater<int>()` 该类的一个对象
这种对象被称为函数对象
4）函数对象的要求
需要重载 `operator()` 运算符
该函数需要时 `public` 访问权限
Duck Typing: *如果一个对象，用起来像函数，那么它就是函数对象！*

3. 自定义类型的排序
   1）重载小于运算符
   2）定义比较函数
   3）定义比较函数对象

```
#include <algorithm>
#include <vector>
using namespace std;

class People {
public:
    int age, weight;
    //1.重载小于运算符
    bool operator<(const People &b) const {
        return age < b.age;
    }
};

//2.定义比较函数
bool compByAge(const People &a, const People &b) {
    return a.age < b.age;
}

//3.定义比较函数对象
class AgeComp {
public:
    bool operator()(const People &a, const People &b) const {
```

```
        return a.age < b.age;
    }
};

int main() {
    vector<People> vec = {{18,50}, {16, 40}};
    sort(vec.begin(), vec.end()); //1
    sort(vec.begin(), vec.end(), compByAge); //2
    sort(vec.begin(), vec.end(), AgeComp()); //3
    return 0;
}
```

1）三种设计模式

1）基于虚函数的模板（Template）的设计模式

运行时确定调用函数的地址

```
class CalculatorBase {
public:
    virtual string read();
    virtual string calculate(string);
    virtual void write(string);
    void process() {
        string data = read();
        string output = calculate(data);
        write(output);
    }
};
```

2）基于模板函数的设计模式

编译期确定调用函数的地址

```
#include <iostream>
#include <fstream>
#include <functional>
#include <string>
using namespace std;

//省略readFromScreen/ReadFromFile(类)/calculateAdd/writeToScreen
//类需要重载operator()

template<class ReadFunc, class CalFunc, class WriteFunc>
void process(ReadFunc read, CalFunc calculate, WriteFunc write) {
    string data = read();
    string output = calculate(data);
    write(output);
}

int main() {
    process(readFromScreen, calculateAdd, writeToScreen);
    process(ReadFromFile(), calculateAdd, writeToScreen);
```

```
    return 0;
}
```

3）基于std::function类的设计模式

std::function类来自头文件

function为**函数指针**与**对象**提供了统一的接口

```
//function<返回值(参数列表)> name[] = {};
function<string()> readArr[] = {readFromScreen, readFromFile()};
function<string(string)> calculateArr[] = {calculateAdd, CalculateMul()};
function<void(string)> writeArr[] = {writeToScreen, WriteToFile()};

auto readArr[] = {readFromScreen, ReadFromFile()}
process(readArr[0], calculate, write);
process(readArr[1], calculate, write);
```

使用function

运行时确定调用函数的地址

　　函数可以作为**参数**传递，可以作为**变量**储存

```
//省略readFromScreen/ReadFromFile(类)/calculateAdd/writeToScreen
//类需要重载operator()
void process(function<string()> read,
             function<string(string)> calculate,
             function<void(string)> write) {
    string data = read();
    string output = calculate(data);
    write(output);
}

int main() {
    process(readFromScreen, calculateAdd, writeToScreen);
    process(ReadFromFile(), calculateAdd, writeToScreen);
    return 0;
}
```

2）STL与函数对象

STL有大量函数用到了**函数对象**（`#include<algorithm>`）

```
for_each  对序列进行指定操作
    for_each(vec.begin(), vec.end(), func)
    //其中func为操作语句
find_if   找到满足条件的对象
    Iterator iter = find_if(vec.begin(), vec.end(), UnaryPredicateThing)
    //返回迭代器
    //参数中的UnarPredicateThing有多种可能类型：函数类型，仿函数，lambda表达式，bool自定义函数等

count_if  对满足条件的对象计数
    count_if(arr, arr + size, cmp)
```

```
    bool cmp(int x) {return x > 10;}
    //返回int指，表示满足条件cmp的对象数量

binary_search    二分查找满足条件的对象
    binary_search(arr, arr + size, index)
    //返回bool值

lower_bound      查找第一个≥某个元素的位置
    lower_bound(arr, arr + size, index)
    //返回地址（数组下标）；若所有元素都<index，返回last的位置（last的位置是越界的）
    //可以采用pos = lower_bound(arr, arr + size, index) - arr来确定index插入既定排好序的数组arr的位置

upper_bound      查找第一个>某个元素的位置
    upper_bound(arr, arr + size, index)
    //返回地址（数组下标）
```

STL也有许多预置的**函数对象**（`#include <functional>`）

```
less      比较a<b
equal_to  比较a==b
greater   比较a>b
plus      返回a+b
//实例化如：less<int>()用于sort函数中的compare位置
```

2. 智能指针与引用计数

shared_ptr（智能指针）来自`<memory>`库
智能指针负责动态内存管理的封装查看更多

1. 构造方法

```
shared_ptr<int> p1(new int(1));
shared_ptr<MyClass> p2 = make_shared<MyClass>(2);
shared_ptr<MyClass> p3 = p2;
shared_ptr<int> p4; //空指针
```

2. 访问对象

```
int x = *p1; //从指针访问对象
int y = p2->val; //访问成员变量
```

3. 销毁对象：p2和p3指向同一对象，当两者均出作用域才会被销毁
   实现方法：**引用计数**（当引用计数归0时，销毁对象）`ptr.use_count()`
   对比`weak_ptr`和`unique_ptr`：涉及引用计数，性能较差

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

```
int main() {
    shared_ptr<int> p1(new int(4));
    //辅助指针count
    cout << p1.use_count() << ' '; // 1
    {
        shared_ptr<int> p2 = p1;
        cout << p1.use_count() << ' '; // 2
        cout << p2.use_count() << ' '; // 2
    }   //p2出作用域
    cout << p1.use_count() << ' '; // 1
}
//调用delete，销毁int*（此时辅助指针count = 0）
```

实现自定义的引用计数（智能指针底层原理）：

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

template<typename T>
class SmartPtr;  //声明智能指针模板类

template<typename T>
class U_Ptr { //辅助指针
private:
    friend class SmartPtr<T>; //SmartPtr是U_Ptr的友元类
    U_Ptr(T *ptr) : p(ptr), count(1) { }
    ~U_Ptr() {delete p;}

    int count;
    T *p; //实际数据存放
};

template<typename T>
class SmartPtr { //智能指针
    U_Ptr<T> *rp;
public:
    SmartPtr(T *ptr) : rp(new U_Ptr<T>(ptr)) {}
    SmartPtr(const SmartPtr<T> &sp) : rp(sp.rp) {
        ++rp->count;
    }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count;
        if (--rp->count == 0) //减少自身所指rp的引用计数 pA = pB
            delete rp; //删除所指向的辅助指针
        rp = rhs.rp;
        return *this;
    }
    ~SmartPtr() {
        if (--rp->count == 0)
            delete rp;
    }
```

```
    T& operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};

int main(int argc, char *argv[]) {
    int *pi = new int(2);
    SmartPtr<int> ptr1(pi); //构造函数
    SmartPtr<int> ptr2(ptr1); //拷贝构造
    SmartPtr<int> ptr3(new int(3)); //不能ptr3(pi)，否则会重复delete
    ptr3 =  ptr2; //注意赋值运算
    cout << *ptr1 << endl; //输出2
    *ptr1 = 20;
    cout << *ptr2 << endl; //输出20
    return 0;
}
```

1）不能使用同一裸指针初始化多个智能指针

```
int* p = new int();
shared_ptr<int> p1(p);
shared_ptr<int> p2(p); //会产生多个辅助指针！报错!
```

2）其他用法

`p.get()` 获取裸指针

`p.reset()` 清除指针并减少引用计数

`static_pointer_cast<int>(p)` 转为int类型指针 (和`static_cast`类似，无类型检查)

`dynamic_pointer_cast<Base>(p)` 转为Base类型指针 (和`dynamic_cast`类似，动态类型检测)

4. 弱引用**weak_ptr**：指向对象但不计数

- 弱引用指针的创建

  `shared_ptr<int> sp(new int(3));`

  `weak_ptr<int> wp1 = sp;`
- 弱引用指针的用法

  `wp.use_count()` 获取引用计数

  `wp.reset()` 清除指针

  `wp.expired()` 检查对象/弱引用是否无效

  `sp = wp.lock()` 从弱引用获得一个智能指针（`sp`的类型是`shared_ptr`）

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

class Child;

class Parent {
    shared_ptr<Child> child;
public:
    Parent() {cout << "parent constructing" << endl; }
    ~Parent() {cout << "parent destructing" << endl; }
```

```
    void setChild(shared_ptr<Child> c) {
        child = c;
    }
};

class Child {
    weak_ptr<Parent> parent; //弱引用
public:
    Child() {cout << "child constructing" << endl; }
    ~Child() {cout << "child destructing" << endl; }
    void setParent(shared_ptr<Parent> p) {
        parent = p;
    }
};

void test() {
    shared_ptr<Parent> p(new Parent());
    shared_ptr<Child> c(new Child());
    p->setChild(c);
    c->setParent(p);
    //p和c被销毁
}

int main() {
    test(); //如果没有弱引用，则此处无法析构Parent和Child对象
    return 0;
}
```

5. **unique_ptr**：保证一个对象只被一个指针引用

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

int main() {
    auto up1 = std::make_unique<int>(20);
    //unique_ptr<int> up2 = up1;  //错误，不能复制unique指针
    unique_ptr<int> up2 = std::move(up1); //可以移动unique指针
    int* p = up2.release(); //放弃指针控制权，返回裸指针
    delete p;
    return 0;
}
```

智能指针总结

优点：① 智能指针可以帮助管理内存，避免内存泄漏
② 区分`unique_ptr`和`shared_ptr`能够明确语义
③ 在手动维护指针不可行，复制对象开销太大时，智能指针是唯一的选择

缺点：① 引用计数会影响性能
② 智能指针并不总是智能，需要了解内部原理
③ 需要小心环状结构和数组指针

8. 案例与设计模式

设计模式（Design Pattern）：优秀架构与解决方案。

设计模式的分类：

1. **行为型模式（Behavioral Patterns）**
   关注对象行为功能上的抽象，从而提升对象在行为功能上的可拓展性。
   能以最少的代码变动完成功能的增减。
2. **结构型模式（Structural Patterns）**
   关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性。
   能在结构层面上尽可能的解耦合。
3. **创建型模式（Creational Patterns）**
   将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗。
   能以简短的代码完成对象的高效创建

8.1 行为型模式

1. 模板方法（Template Method）模式

1. 抽象类（父类）定义算法的骨架
   算法的细节由实现类（子类）负责实现
   在使用时，调用抽象类的算法骨架方法，再由这个方法来根据需要调用具体类的实现细节
   当拓展一个新的实现类时，重新继承与实现即可，无需对已有的实现类进行修改
2. 开放封闭原则
   对扩展开放，对修改封闭
   结构层面上解耦，对抽象进行编程
3. 更多适用于逻辑复杂但结构稳定的场景，尤其是其中的某些步骤变化剧烈且没有相互关联时。

2. 策略（Strategy）模式

1. 单一责任原则一个类（接口）
   只负责一项职责，不存在多于一个导致类变更的原因
   功能层面上解耦
2. 更多适用于算法本身灵活多变的场景，且多种算法之间需要协同工作。
3. 策略类只做函数用！且重载函数名称语义与调用派生类函数名称一致！一般不储存变量！

3. 迭代器（Iterator）模式

1. 提供一种方法顺序访问一个聚合对象中的各个元素
   不需要暴露该对象的内部表示（与对象的内部数据结构形式无关，i.e.数组还是链表）
   具体实现相当于用模板方法构建迭代器和数据存储基类，为每种单独的数据结构都实现其独有的迭代器和存储类
   上层执行时只依赖于抽象的迭代器接口，而无需关注最底层的具体数据结构
2. 实现**Iterator**基类
   把数据“访问”设计为一个统一接口，形成迭代器
   这样算法构建就可以不依赖于底层的数据结构

```
//迭代器基类
class Iterator {
```

```
public:
    virtual ~Iterator() { }
    virtual Iterator& operator++() = 0;
    virtual float& operator++(int) = 0;
    virtual float& operator*() = 0;
    virtual float* operator->() = 0;
    virtual bool operator!=(const Iterator &other) const = 0;
    bool operator==(const Iterator &other) const {
        return !(*this != other);
    }
};

vector<int> analyze(Iterator* begin, Iterator* end, AIFramework* ai) {
    vector<int> all_results;
    for (Iterator* p = begin; *p != *end; (*p)++) {
        string inp = ai->processData(**p);
        int result = ai->predict(inp);
        all_results.push_back(result);
    }
    return all_results;
}
```

3. 实现基于数组的Iterator：**ArrayIterator**

```
//继承自迭代器基类并配套ArrayCollection使用的迭代器
class ArrayIterator : public Iterator {
    OneData *_data; //ArrayCollection的数据
    int _index;     //数据访问到的下标
public:
    ArrayIterator(OneData* data, int index) :
        _data(data), _index(index) { }
    ArrayIterator(const ArrayIterator& other) :
        _data(other._data), _index(other._index) { }
    ~ArrayIterator() { }
    Iterator& operator++() { //前缀，返回Itertaor的引用
        _index++;
        return *this;
    }
    OneData& operator++(int) { //后缀，返回OneData的引用
        //不返回Iterator对象的原因：Iterator是抽象类，无法实例化为对象
        //如果返回Iterator&会使用new构建新对象，但难以在外部销毁
        _index++;
        return _data[_index - 1];
    }
    OneData& operator*() {
        return *(_data + _index);
    }
    OneData* operator->() {
        return (_data + _index);
    }
    bool operator!=(const Iterator &other) const {
```

```
        return (_data != ((ArrayIterator*)(&other)) -> _data ||
                _index != ((ArrayIterator*)(&other)) -> _index);
    }
};
```

4. 实现**Collection**（容器）基类
   能够返回代表“头”和“尾”的迭代器
   使用“左闭右开区间”，即`[begin, end)`

```
//Collection基类
class Collection {
public:
    virtual ~Collection() { }
    virtual Iterator* begin() const = 0;
    virtual Iterator* end() const = 0;
    virtual int size() = 0;
};
```

5. 实现基于数组的Collection：**ArrayIterator**

```
class ArrayCollection : public Collection { //底层为数组的存储结构类
    friend class ArrayIterator; //friend可以使得配套的迭代器类可以访问数据
    OneData* _data; //若需要实现float数组迭代器，可将OneData改为float
    int _size;
public:
    ArrayCollection() : _size(10){_data = new OneData[_size]; }
    ArrayCollection(int size, OneData * data) : _size(size) {
        _data = new OneData[_size];          //开辟数组空间用以存储数据
        for (int i = 0; i < size; i++)
            *(_data + i) = *(data + i);
    }
    ~ArrayCollection() { delete[] _data; }
    int size() { return _size; }
    Iterator* begin() const;
    Iterator* end() const;
};

Iterator* ArrayCollection::begin() const {  //头迭代器，并放入相应数据
    return new ArrayIterator(_data, 0);  //注意该迭代器应该由外部销毁
}
Iterator* ArrayCollection::end() const {        //尾迭代器，并放入相应数据
    return new ArrayIterator(_data, _size);
}
```

另一种常见的迭代器模式

```
Iterator* it = collection.iterator();

while (it->hasNext()) {
    it->next();
```

```
    Object object = it->getValue();
    //do something with object;
}
```

STL中的迭代器模式

迭代器模式：模板 vs 继承
目标相同：将算法构建与底层数据结构解耦
区别：
继承：1. 算法中需要使用迭代器的基类指针
模板：1）更加简洁，算法可以使用迭代器对象
2）对每一种迭代器类型都会生成相应代码，使编译速度变慢，可执行文件变大

- 使用迭代器进行循环
  `for (auto i : container)`

4. 其他行为型模式

观察者模式：将事件观察者与被观察者解耦
职责链模式：多个处理器处理按职责处理同一请求
解释器模式：某个语言定义它的语法（或者叫文法）表示，并定义一个解释器用来处理这个语法
备忘录模式：捕捉并存储对象内部状态，以便后续恢复
访问者模式：允许多个操作应用到一组对象上，解耦操作和对象本身

8.2 结构型模式

关心对象组成结构上的抽象，包括**接口**、**层次**、**对象组合**等
抽象结构层次上的不变量，尽可能减少类与类之间的联系与耦合，从而能够以最小的代价支持新功能的增加

1. 适配器（Adapter）模式

功能上满足要求，但是接口不一致->需要进行接口的“转换”
讲一个类的接口转换成客户希望的另一个接口，从而使得原本由于接口不兼容而不能一起工作的类可以在统一的接口环境下工作
方法：通过包装一个需要是配的类，把原借口转换成目标接口
优势：复用现有的类；目标类和适配者类解耦，无需修改原有代码

对象适配器模式：使用组合实现适配

```
class NewModelAdapter : public Model{
public:
    int predict(string input) {
        float result = newModel.predictResult(input);
        return result >= 0.5;
    }
    void load() {
        newModel.loadParameters();
    };
private:
    // 将NewModel组合进来实现相关功能
    NewModel newModel;
```

```
};
```

类适配器模式：使用继承实现适配

```
//直接继承NewModel并改造接口
//采用私有继承可以使得外界只能接触到NewModelAdapter中的接口

class NewModelAdapter : private NewModel, public Model {
public:
    int predict(string input) {
        float res = predictResult(input);
        return res >= 0.5;
    }
    void load() {
        loadParameters();
    }
};
```

2. 代理/委托（Proxy）模式

在被访问对象上加上一个访问层，在访问层上增加新的控制操作，访问层接口保持不变
应用：用于被代理对象进行控制，如引用计数控制、权限控制、远程代理、延迟初始化等
应用：远程代理；智能引用；虚代理（对象的创建开销很大，需要延迟创建。即实际访问该对象内容时才申请资源创建对象）；保护代理（用代理对象控制原始对象的访问权限）

```
class Subject{
public:
    virtual void Request() = 0;
};
class RealSubject : public Subject{
public:
    virtual void Request() {
         ... // request
    }
};

class Proxy : public Subject{
private:
     RealSubject* m_realSub;
public:
    void Request() { //Proxy和Subject有相同的接口，再调用代理类的Request时增加额外功能
        ... // do something
        m_realSub->Request();
        ... // do something
    }
};
```

代理模式：智能指针引用计数

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

```
template <typename T>
class SmartPtr; //提前声明智能指针模板类

//辅助指针，用于存储指针计数以及封装实际指针地址
template <typename T>
class U_Ptr {
private:
    friend class SmartPtr<T>;
    U_Ptr(T *ptr) : p(ptr), count(1) { }
    ~U_Ptr() { delete p; }

    int count;
    T *p; //数据存放地址
};

template <typename T>
class SmartPtr { //智能指针
private:
    U_Ptr<T> *rp;   //进行实际指针操作的辅助指针
public:
    SmartPtr(T *ptr) : rp(new U_Ptr<T>(ptr)) { }
    //调动拷贝构造即增加引用计数
    SmartPtr(const SmartPtr<T> &sp) : rp(sp.rp) { ++rp->count; }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count; //赋值号后的指针引用加1
        if (--rp->count == 0) delete rp;    //原内部指针引用减1
        rp = rhs.rp;        //代理新的指针
        return *this;
    }
    ~SmartPtr() { //只有引用次数为0才会释放
        if (--rp->count == 0) delete rp;
    }
    //对智能指针操作等同于对内部辅助指针操作
    T& operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};

int main(int argc, char *argv[]) {
    //声明指针
    int *i = new int(2);
    //使用代理来包裹指针
    SmartPtr<int> ptr1(i);
    SmartPtr<int> ptr2(ptr1);
    SmartPtr<int> ptr3 = ptr2;
    //之后的操作均通过代理进行
    cout << *ptr1 << endl;
    *ptr1 = 20;
    cout << *ptr2 << endl;
    return 0;
}
```

3. 装饰器（Decorator）模式

创建一个装饰类，用来包装原有的类，并在保持类方法完整性的前提下，提供额外的功能
装饰类与被包装类继承于同一基类，这样装饰之后的类可以被再次包装并赋予更多功能
装饰器≈一连串的代理（有多少新功能就包裹多少次）

```
//数据处理策略基类
class ProcessStrategy {
public:
    virtual string processData(AbstractData* data) = 0;
};

//进行数据处理：分词
class ProcessStrategyTokenize : public ProcessStrategy {
public:
    string processData(AbstractData* data) {…}
};

//装饰器的核心内涵在于用装饰器类整体包裹改动之前的类，以保留原来的全部接口
//在原来接口保留的基础上进行新功能扩充
class ProcessDecorator : public ProcessStrategy {
public:
    ProcessDecorator(ProcessStrategy* component) : component(component) {}
    string processData(AbstractData* data) {
        string out = component->processData(data);
        return extraProcess(out);
    }
    virtual string extraProcess(string lastout) = 0;
private:
    //这里一个基类指针可以能够以递归的形式不断增加新功能
    ProcessStrategy* component;
};

//包裹原component并增加去除停用词处理
class StopWordDecorator : public ProcessDecorator {
public:
    StopWordDecorator(ProcessStrategy* component) : ProcessDecorator(component) {}
    string extraProcess(string lastout) { … } // 去除停用词
};
//包裹原component并增加前缀
class PrefixDecorator : public ProcessDecorator {
public:
    PrefixDecorator(ProcessStrategy* component) : ProcessDecorator(component) {}
    string extraProcess(string lastout) { … } // 增加前缀
};
//包裹原component并增加把所有字母转成小写
class LowerDecorator : public ProcessDecorator {…};

int main(int argc, char** argv) {
    …
     //基础的分词处理
    ProcessStrategyTokenize tk_process;
```

```
    //在分词处理后删除停用词
    StopWordDecorator st_tk_process(&tk_process);
    //在分词、删除停用词后增加前缀
    PrefixDecorator pfx_st_tk_process(&st_tk_process);
    //在分词、删除停用词、增加前缀后转成小写
    LowerDecorator lw_pfx_st_tk_process(&pfx_st_tk_process);

    AIFramework* ai = new AIFramework(&readStrategy,
        &lw_pfx_st_tk_process, &evaluateStrategy, &model);
    …
}
```

4. 其他结构型模式

组合模式：将一组对象组织成树形结构，将单个对象和组合对象都看作树中的节点，以统一处理逻辑
外观模式：它通过封装细粒度的接口，提供组合各个细粒度接口的高层次接口，来提高接口的易用性
享元模式：复用不可变对象，节省内存

8.3 创建型模式

将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗，能以简短的代码完成对象的高效创建
应用：用于对象的创建

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类
建造者模式：建造者模式用来创建复杂对象，可以通过设置不同的可选参数，“定制化”地创建不同的对象。
工厂方法模式：用来创建不同但是相关类型的对象，由给定的参数来决定创建哪种类型的对象
原型模式：利用对已有对象（原型）进行复制（或者叫拷贝）的方式，来创建新对象，以节省时间
单例模式：用来创建全局唯一的对象

8.4 设计原则

开闭原则
一个软件实体，比如类，模块，函数应该对扩展开放，对修改关闭
最基础的设计原则

单一职责原则
每个类应该只有一个职责，只有一个原因可以引起它的改变
例如：迭代器模式使得数据结构与算法分离；可视化程序设计中页面与逻辑分离

里氏代换原则
只要父类出现的地方子类就可以出现，即子类尽量不修改父类的数据与方法，实现基类代码的充分复用

依赖倒转原则
要依赖于抽象，不要依赖于具体。针对接口编程，而不是针对实现编程。具体而言就是上层模块不应该依赖底层模块，使用接口和抽象类指定好规范，剩下的具体细节由实现类来完成
例如：策略模式/模板方法模式不依赖于具体的策略实现，只依赖于抽象

接口隔离原则
不要建立臃肿庞大的接口。即接口尽量细化的同时接口中的方法尽量少
功能拆分粒度太小，将使得类、接口的数量过多；功能拆分粒度太大，将使得类之间耦合度高，程序不灵活

迪米特原则
最少知道原则，一个对象应该对其他对象有最少的了解，使得功能模块相对独立

合成复用原则
合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的
即在实现扩展类功能时，优先考虑使用组合而不是继承；如需要使用继承，则遵守里氏代换原则

---

9. 其它琐碎的知识点

1. 简单计算图：HW4D

2. 哈希算法：HW4B （对比HW2C Map）

哈希算法：一段消息（二进制或字符串） -> 固定长度的数/字符串
常用的哈希算法介绍：MD4，MD5，SHA-1，SHA256...

base64编码：把二进制数据变成一串可读的东西；只是一种信息表示方式，不具有加密作用

3. 快速幂

```
int quickpow(int a, int b){ //可适当改成long long类型
    int ret = 1;
    while (b) {
        if (b & 1) ret *= a;
        a = a * a;
        b >>= 1; //位运算，右移，相当于除以2
    }
    return ret;
}
```

4. Git

Git： 分布式版本控制软件
git history：查看项目开发时间线
git branch：在不影响主代码的情况下进行开发

GitHub、清华Git等是“云盘”的概念，和**Git**并不等同

公钥和私钥

1. 公钥和私钥成对出现，其中一个用来加密，另一个用于解密。一般公开公钥，私有私钥。
2. 自己电脑的公钥和私钥一般存在于用户文件夹下：`.ssh/id_rsa.pub`和`.ssh/id_rsa`下，注意：linux系统和windows系统的公钥私钥不一致（可以改成一致的）
3. 公钥和私钥本质上是一种RSA加密算法。

```
notepad .ssh/id_rsa.pub ///打开公钥记事本
notepad .ssh/id_rsa ///打开密钥记事本
ssh-keygen ///generate public/private rsa key pair
```

git repository

工作区(working directory) => 暂存区(staging) => 本地仓库(local repository) => 远程仓库(remote repository)

`git repo`：包含了一个项目所有内容，可以追踪整条历史修改线

```
mkdir oop-git
cd oop-git
git init //新建仓库，使用git来管理这个文件夹
git add test.cpp //将文件夹里的文件test.cpp到目前为止的修改放入git暂存区
    git add . //添加当前目录的所有的东西
git commit -m "modify test.cpp" //当所有修改加入暂存区后，将其提交至本地仓库，并添加备注内容
git log //查看之前的commit记录以及对应的sha编码
git show //查看此次commit相对于上次的改动
    git show '编码' //查看某次commit相对于上次的改动
git diff //查看此次commit相对于上次的改动（重点显示差别）
git reset --hard '编码' //回退到某次commit

git clone [SSH_code链接]//把一个远程repo克隆到本地 ///仅适用于Windows系统
git reset '编码' ///回退到先前版本（寄存在stash中）
    git reset --hard [hash] ///硬回退到先前版本，当前版本trash
    git reset [hash] <==> git reset -- mixed ///默认回退，当前版本回退到暂存区，可以使用git stash切换到工作区，而将原工作区（最新版本）存放在git stash pop
    git reset --soft ///软回退
git stash ///暂存所有未提交修改，此时当前版本被先前版本覆盖
    git stash list ///列出暂存区版本
    git stash pop ///恢复暂存区版本
git merge ///综合本地仓库和远程仓库信息，merge之后仍需一步一步add, commit
git push //把本地commit的所有修改推送到远程repo/github上
    git push -f ///强制将当前的旧版本push到远程仓库
git pull //把远程repo同步到本地

git status //查看本地暂存区文件状态
git branch //查看repo不同分支情况，开新分支
git merge //合并两个分支

在本地仓库中添加.gitignore文件，写入*.cpp，则所有新建cpp无法add
git checkout run.sh //clone一个远端项目仓库后对run.sh进行本地修改，还没有add的时候可以通过该命令放弃本地修改
```

```
///可能需要用到的绑定用户名:
git config --global user.name "Catherine0120"
git config --global user.email
```

与他人使用远程仓库合作时，使用git fetch会比较远程仓库中的内容和本地仓库的内容，用户在检查是否有冲突以后决定是否合并到本地仓库中；而git pull会直接尝试合并。

5. bash

> 任何在命令行中能正常执行的命令都可以被写进一个BASH脚本并完成一样的事，反之亦然

一些common sense

便于一次性执行大量命令
一般使用`.sh`作为文件后缀
一般在命令行使用`$ bash xxx.sh`启动脚本
如在sh文件第一行通过特定指令指定解释器，如`#!/bin/bash`，则可以在使用`$ ./xxx.sh`启动脚本

示例

```
///批量修改文件名（.txt -> .cpp）
for name in `ls *.txt`; do
    mv $name ${name%.txt}.cpp
done
```

```
///批量输入文件
INPUT_DIR=testcases #测例所在文件夹
OUTPUT_DIR=output #输出文件夹
mkdir -p $OUTPUT_DIR #若不存在输出文件夹，则创建
for input in `ls $INPUT_DIR/case*.txt`; do
        # '<' 用于输入重定向，将$input表示的文件里的内容
        # 输入到test程序里
        # '>' 用于输出重定向。${input##*/}表示获取$input的文件名，
        # 如${aaa/bcd.txt##*/} --> bcd.txt
    ./test < $input > $OUTPUT_DIR/${input##*/}
done
```

BASH基本语法

一般来说，都可以用**Python**解决。

1. 空格或tab区分参数
   `$ command foo bar`表示foo和bar为command的两个参数
2. 使用分号隔开不同命令，表示顺序执行这些命令
   `$ clear; ls`表示先执行clear再执行ls，与两条指令分两行效果相同
3. `$ cmd1 && cmd2`：若cmd1成功，才执行cmd2
   `$ cmd1 || cmd2`：若cmd1失败，才执行cmd2

4. 变量声明：`variable=value`， 等号两边不能有空格（bash没有数据类型的概念，所有的变量值都是字符串）
5. 读取变量：`$variable`或`${variable}`
6. 特殊变量：

```
$?:  上一个命令的退出码，成功为0，失败为非0
$#:  传递给脚本的参数个数
$@:  传递给脚本的全部参数
```

7. 获取脚本参数
   `$ bash script.sh arg1 arg2`
   其中：`$0`为脚本文件名，`$1`为arg1，`$2`为arg2
8. 数组声明：`array=(value1 value2 value3...)`；可以不使用连续下标
9. 读取数组：`${array[n]}`，`${array[@]}`可获得array数组的所有元素，`${#array[@]}可获得array数组的长度
10. 条件判断：`if ... ; then ... elif ...; then ... else ... fi`
11. 循环

```
///while
while condistion; do
    commands
done
///for
for variable in list; do
    commands
done
for ((expression1; expression2; expression3)); do
    commands
done
```

12. 函数、重定向等其他内容

6. 关于换行符

Windows系统下txt文本默认的换行方式是**CRLF**（两个字符：`\r \n`）（`\r` ASCII = 13)
Linux系统下txt文本默认的换行方式是**LF**（一个字符：`\n`）（`\n` ASCII = 10)

7. 动态类型：HW6D

8. MagicArray：题库#96

```
char buff[4096];
int len = sprintf(buff, "arr[%d] += %d", _index1, _num);
//将字符串存入buff中
string line_inst = string(buff, len);
```

```
//转换为string类型
```

9. enum类型

```
enum type {
    kIndex, //0
    kAddEq, //1
    kEq //2
};

if (type == kAddEq) {...}
```