C++

个人笔记总结

王
小
龙

Good luck

Believe yourself

Just go

概述：

一、C++语言语法基础(6)
1.从 C 到 C++的过渡
2.类和对象
3.操作符重载
4.继承与多态
5.异常和 I/O 流

二、数据结构和算法
1.基本数据结构，堆栈、队列、链表、二叉树，实现和应用(2)
2.排序和查找算法

三、模板和 STL
1.模板语法
2.STL

四、阶段项目
简化的企业管理信息系统(MIS)

第一课 从C到C++的过渡

一、背景介绍

算盘 - 面向硬件的编程
电子计算机 - 机器语言的编程1010
- 汇编语言的编程ADD
- 高级语言的编程Fortran
  printf ("%d", 12);
- 结构化程序设计C/PASCL
  顺序、分支、循环、函数
- 面向对象的程序设计C++/Java/C#
- 面向问题的程序设计

1960 - Algol 60，算法语言,远离硬件，不适合进行系统开发
1963 - 剑桥大学，CPL，在Algol 60的基础上增加对系统开发的支持，复杂，不易掌握，不易使用
1970 - MIT，BCPL，CPL的精华版，易学易用，太慢，不实用
1972 - Ken Thomposon，B语言，通过运行时支持优化BCPL的性能，缺少类型
1973 - Dennis Ritchie，C语言，用C重新实现UNIX内核
1978 - 《The C Programming Language》，第一个C语言的事实标准
1989 - ANSI C，C89
1990 - ISO C，C90
1999 - ISO C 修订，C99
197X - Bajarne Stroustrup，simula早期的面向对象语言，性能低下，B语言。
1979 - 贝尔实验室，多核UNIX系统仿真，Cpre，
通过扩展宏为C语言增加类似simula的面向对象机制。C with Class:
simula - 类
Alogo 68 - 操作符重载
Ada - 模板、名字空间
Smalltalk - 引用、异常
C是C++的子集，C++是对C的扩展。
1983 - C++命名
1985 - CFront 1.0，第一款商用C++编译器
1987 - GNU C++
1990 - Borland C++
1992 - Microsoft C++，IBM C++
1998 - ISO C++98
2003 - ISO C++03
2011 - ISO C++2011/C++11/C++0x

二、C++语言的使用领域：

1.游戏开发：强建模能力，性能高。
2.科学计算：FORTRAN，C++算法库。
3.网络和分布式：ACE 框架。
4.桌面应用：VC/MFC，Office，QQ，多媒体
5.操作系统和设备驱动：优化编译器的发明使 C++在底层开发方面可以和 C 向媲美。
6.移动终端
既需要性能，同时又要有面向对象的建模。

三、C++比 C 更丰富

1.支持面向对象，将问题域和方法域统一化。宏观面向对象，微观面向过程。
2.支持泛型编程。

```
int add (int a, int b) { ... }
template<typename T>
T add (T a, T b) { ... }
```

3.支持异常机制。

```
int func (void) {
  ...
}
int main (void) {
  if (func () == -1) {
     错误处理;
  }
}
```

4.操作符重载

四、第一个 C++程序

1.编译器：g++，如果用 gcc 需要带上-lstdc++，
指定其使用标准 c++的运行库。

2.源文件扩展名：.cpp/.cc/.C/.cxx/.c++，最好用.cpp

3.头文件：#include <iostream>
大多数标准库头文件都没有.h 后缀。

4.输出：cout - 标准输出对象
输入：cin - 标准输入对象
插入运算符：<<
提取运算符：>>
5.std：所有标准库的函数、对象、类型都位于 std 名字空间中。

五、名字空间

1.对程序中的标识符（类型、函数、变量），

按照某种逻辑规则划分成若干组。

2.定义名字空间

```
namespace 名字空间名 {
  名字空间成员;
}
```

3.使用名字空间

1 作用于限定符：名字空间名::名字空间成员，

表示访问特定名字空间中的特定成员。

例子：

```
#include <iostream>
int main (void) {
    std::cout << "Hello, World !" << std::endl;
    int i;
    double d;
    char s[256];
   //   scanf ("%d%lf%s", &i, &d, s);
    std::cin >> i >> d >> s;
   //   printf ("%d %lf %s\n", i, d, s);
    std::cout << i << ' ' << d << ' ' << s << '\n';
    return 0;
}
```

---

2 名字空间指令：

using namespace 名字空间名;
在该条指令之后的代码对指令所指名字空间中的所有成员都可见，
可直接访问这些成员，无需加“::”。
例：using namespace std;

3 名字空间声明：

using 名字空间名::名字空间成员;
将指定名字空间中的某个成员引入当前作用域，
可直接访问这些成员，无需加“::”。

4.匿名名字空间

如果一个标识符没有被显示地定义在任何名字空间中，
编译器会将其缺省地置于匿名名字空间中。
对匿名名字空间中的成员通过“::名字空间成员”的形式访问。

5.名字空间合并

6.名字空间嵌套

```
namespace ns1 {
  namespace ns2 {
    namespace ns3 {
      void foo (void) { ... }
    }
  }
}
ns1::ns2::ns3::foo ();
using namespace ns1::ns2::ns3;
foo ();
```

例子：名字空间

```
#include <iostream>
using namespace std;
//namespace {
    void print (int money) {
        cout << money << endl;
    }
//}
// 农行名字空间
namespace abc {
    int balance = 0;
    void save (int money) {
        balance += money;
    }
    void draw (int money) {
        balance -= money;
    }
}
namespace abc {
    void salary (int money) {
        balance += money;
    }
```

```
    void print (int money) {
        cout << "农行：";
        ::print (money);
    }
}
// 建行名字空间
namespace ccb {
    int balance = 0;
    void save (int money) {
        balance += money;
    }
    void draw (int money) {
        balance -= money;
    }
    void salary (int money) {
        balance += money;
    }
}
int main (void) {
    using namespace abc; // 名字空间指令
    save (5000);
    cout << "农行：" << balance << endl;
    draw (3000);
    cout << "农行：" << balance  << endl;
    ccb::save (8000);
    cout << "建行：" << ccb::balance << endl;
    ccb::draw (5000);
    cout << "建行：" << ccb::balance << endl;
    using ccb::salary; // 名字空间声明
//  using abc::salary;
    salary (6000);
    cout << "建行：" << ccb::balance << endl;
    abc::print (abc::balance);
    return 0;
}
```

六、C++中的结构、联合和枚举

1.结构

和 C 语言的不同：
1)定义结构型变量时，可以省略 struct 关键字。
2)结构内部可以定义函数——成员函数。
3)sizeof (空结构) -> 1

例子：

```
#include <iostream>
using namespace std;
struct Student {
    char name[128];
    int age;
    void who (void) {       //成员函数
        cout << "我叫" << name << ", 今年" << age
            << "岁了。" << endl;
    }
};
int main (void) {
    Student student = {"张飞", 25}, *ps = &student;
    student.who ();
    ps->who ();
    struct A {};
    cout << sizeof (A) << endl;
    return 0;
}
```

2.联合

增加了匿名联合的概念。借用联合语法形式，
描述一些变量在内存中的布局方式。

```
int main()
{
    UNION
    {
    int a;
    char ch[4];
    };
    a=0x12345678;
}
```

定义联合变量时，可以不加 union

例子：

```
#include <iostream>
using namespace std;
int main (void) {
    // 匿名联合
    union {
        int x;
        char c[4] /*= {'A', 'B', 'C', 'D'}*/;
    };
    cout << (void*)&x << ' ' << (void*)c << endl;
    x = 0x12345678;
    for (int i = 0; i < 4; ++i)
        cout << hex << (int)c[i] << ' ';
    cout << endl;
    return 0;
}
```

3.枚举

枚举是一个独立的数据类型。

C:

```
enum E {a, b, c};
enum E e;
e = a;
e = 1000;
```

C++:

```
enum E {a, b, c};
E e;
e = a;
e = b;
e = c;
b=1;      // ERROR !
e = 1000; // ERROR !
e = 1;    // ERROR !
```

例子：

```
#include <iostream>
using namespace std;
int main (void) {
    enum E {a, b, c};
    E e;
    e = a;
    e = b;
    e = c;
  //e = 1000;//报错
  //e = 1;   //报错      return 0;}
```

七、C++的布尔类型，跟在 cout 后面可以 boolalpha

```
bool b = true;
b = false;
cout << sizeof (b) << endl; // 1
b = 100;
b = 1.234;
b = "hello";
b = 'A';
```

八、C++中的运算符别名

```
&& - and
|| - or
&  - bitand
^  - xor
{  - <%
}  - %>
[  - <:
]  - :>
```

九、C++中的函数

1.重载：条件

在同一个作用域中，
函数名相同，
参数表不同的函数，
构成重载关系。

C++编译器会对程序中的函数做换名，
将参数表中的类型信息汇合到函数名中，以保证函数名的唯一。
通过 extern "C"，可以要求编译器不做 C++换名，以方便在 C 语言的模块中使用 C++编译生成的代码。

方式一：

```
extern "C" {
int add (int a, int b) {
    return a + b;
  }
int sub (int a, int b) {
    return a - b;
  }
          }
```

方式二：

```
extern "C" int add (int a, int b, int c) {

    return a + b + c;          }
```

2.缺省参数和哑元参数

1)如果调用一个函数时，没有提供实参，那么对应形参就取缺省值。
2)如果一个参数带有缺省值，那么它后边的所有参数必须都带有缺省值。
3)如果一个函数声明和定义分开，那么缺省参数只能放在声明中。
4)避免和重载发生歧义。
5)只有类型而没有名字的形参，谓之哑元。

```
i++ - operator++
++i
V1: void decode (int arg) { ... }
V2: void decode (int) { ... }
```

例子 1：重载与缺省值

```
#include <iostream>
using namespace std;
void foo (int a = 10, double b = 0.01,
          const char* c = "tarena");    //函数 1
void foo (void) {}                      //函数 2
//函数 1 与函数 2 构成重载关系
void bar (int) {                        //函数 3
```

```
    cout << "bar(int)" << endl;
}
void bar (int, double) {                //函数 4
    cout << "bar(int,double)" << endl;
}
//函数 3 与函数 4 构成重载关系
int main (void) {
    foo (1, 3.14, "hello");//调用函数 1
    foo (1, 3.14);          //调用函数 1

    foo (1);                //调用函数 1
//  foo (); // 歧义 ，可以调用函数 2，但也可以调用函数 1，因为函数 1 在不提供实参的情况下，可以取缺省值。
    bar (100);              //调用函数 3
    bar (100, 12.34);       //调用函数 4
    return 0;
}
```

例子 2：重载与作用域

```
#include <iostream>
using namespace std;
namespace ns1 {
    int foo (int a) {                       函数 1
        cout << "ns1::foo(int)" << endl;
        return a;
    }
};
namespace ns2 {
    double foo (double a) {                 函数 2
        cout << "ns2::foo(double)" << endl;
        return a;
    }
};
int main (void) {
    using namespace ns1;  // 名字空间指令
    using namespace ns2;  // 名字空间指令
    cout << foo (10) << endl;     //10 调用函数 1，作用域可见 ns2 与 ns1，所以与函数 2 构成重载
    cout << foo (1.23) << endl;  //1.23 调用函数 2，作用域可见 ns2 与 ns1，所以与函数 1 构成重载
```

```
    using ns1::foo;                 //名字空间声明
（当同时出现名字指令与名字空间声明，则名字空间声明会隐藏名字空间指令）
    cout << foo (10) << endl;       //10，调用函数 1，只可见名字空间 ns1 的 foo(),所以
也并不构成重载。

   cout << foo (1.23) << endl;  //10，调用函数 1，只可见名字空间 ns1 的 foo(),所以也
并不构成重载。

    using ns2::foo;                 //名字空间声明
    cout << foo (10) << endl;       //10，调用函数 1，可见名字空间 ns1 与名字空间 ns2
的 foo(),所以构成重载。

  cout << foo (1.23) << endl;  //1.23，调用函数 2，可见名字空间 ns1 与名字空间 ns2 的
foo(),所以构成重载。
    return 0;
}
```

---

3. 内联

1)编译器用函数的二进制代码替换函数调用语句，减少函数调用的时间开销。这种优化策略成为内联。

2)频繁调用的简单函数适合内联，而稀少调用的复杂函数不适合内联。

3)递归函数无法内联。

4)通过 inline 关键字，可以建议编译对指定函数进行内联，但是仅仅是建议而已。

```
inline void foo (int x, int y){...}
```

十、C++的动态内存分配

malloc/calloc/realloc/free
1.new/delete：对单个变量进行内存分配/释放
2.new[]/delete[]：对数组进行内存分配/释放

例子：new 与 delete

```
#include <iostream>
using namespace std;
int main (void) {
  //int* pi = (int*)malloc (sizeof (int));
  //free (pi);          //c 中的方法
    int* pi = new int;
    *pi = 1000;
    cout << *pi << endl;
    delete pi;                //一定要释放内存，否则会造成内存泄露，很严重
    pi = NULL;                //不要忘记这个，虽然不会报错，但是要有好习惯
    /*
    *pi = 2000;
    cout << *pi << endl;      //pi 指向的内存地址已经被释放，被初始化为指向 NULL
    */
   pi = new int[10];
    for (size_t i = 0; i < 10; ++i)
        pi[i] = i;
    for (size_t i = 0; i < 10; ++i)
        cout << pi[i] << ' ';
    cout << endl;
    delete[] pi;              //千万记住 new[]要用 delete[]来释放内存
    pi = NULL;
   pi = new int (1234);       //用 new 分配内存的同时初始化赋一个值。
    cout << *pi << endl;      //1234
    delete pi;
    pi = NULL;
   char buf[4] = {0x12,0x34,0x56,0x78};
    pi = new (buf) int;
    cout << hex << *pi << endl;
//  delete pi;
   cout << (void*)pi << ' ' << (void*)buf << endl;
    int (*p)[4] = new int[3][4];
    delete[] p;
    int (*q)[4][5] = new int[3][4][5];
    delete[] q;
    return 0;
}
```

十一、引用

1.引用即别名。

```
int a = 20;
int& b = a; // int* b = &a;
b = 10; // *b = 10;
cout << a << endl; // 10
```

2.引用必须初始化。

```
int a;
int* p;
a = 20;
p = &a;
int& b; // ERROR !
int& b = a; // OK
```

3.引用一旦初始化就不能再引用其它变量。

```
int a = 20, c = 30;
int& b = a;
b = c; // c => b/a
```

4.引用的应用场景

1)引用型参数

a.修改实参

b.避免拷贝，通过加 const 可以防止在函数中意外地修改实参的值，同时还可以接受拥有常属性的实参。

2)引用型返回值

```
int b = 10;
int a = func (b);
func (b) = a;
```

从一个函数中返回引用往往是为了将该函数的返回值作为左值使用。但是，一定要保证函数所返回的引用的目标在该函数返回以后依然有定义，否则将导致不确定的后果。

不要返回局部变量的引用，可以返回全局、静态、成员变量的引用，也可以返回引用型形参变量本身。

5.引用和指针

1)引用的本质就是指针，很多场合下引用和指针可以互换。

2)在 C++层面上引用和指针存在以下不同：

A.指针式实体变量，但是引用不是实体变量。

```
int& a = b;
sizeof (a); // 4
double& d = f;
sizeof (d); // 8
```

B.指针可以不初始化，但是引用必须初始化。

C.指针的目标可以修改，但是引用的目标的不能修改。

D.可以定义指针的指针，但是不能定义引用的指针。

```
int a;
int* p = &a;
int** pp = &p; // OK
int& r = a;
int&* pr = &r; // ERROR
```

E.可以定义指针的引用，但是不能定义引用的引用。

```
int a;
int* p = &a;
int*& q = p; // OK
int& r = a;
int&& s = r; // ERROR
```

F.可以定义指针的数组，但是不能定义引用的数组。

```
int a, b, c;
int* parr[] = {&a, &b, &c}; // OK
int& rarr[] = {a, b, c};    // ERROR
```

可以定义数组的引用。

```
int arr[] = {1, 2, 3};
int (&arr_ref)[3] = arr;    // OK
```

例子：指针与引用

```
#include <iostream>
using namespace std;
void swap1 (int a, int b) {
    int c = a;
    a = b;
    b = c;
}
void swap2 (int* a, int* b) {
    int c = *a;
    *a = *b;
    *b = c;
}
void swap3 (int& a, int& b) {
    int c = a;
    a = b;
    b = c;
```

```
}
void swap4 (const char* x, const char* y) {
    const char* z = x;
    x = y;
    y = z;
}
void swap5 (const char** x, const char** y) {
    const char* z = *x;
    *x = *y;
    *y = z;
}
void swap6 (const char*& x, const char*& y) {
    const char* z = x;
    x = y;
    y = z;
}
struct Student {
    char name[1024];
    int age;
};
void print (const struct Student& s) {
    cout << s.name << ", " << s.age << endl;
//  s.age = -1;
}
void foo (const int& x) {
    cout << x << endl;
}
int main (void) {
    int a = 100, b = 200;
//  swap1 (a, b);
//  swap2 (&a, &b);
    swap3 (a, b);
    cout << a << ' ' << b << endl; // 200 100
    const char* x = "hello", *y = "world";
//  swap4 (x, y);
//  swap5 (&x, &y);
    swap6 (x, y);
    cout << x << ' ' << y << endl;
    Student s = {"张飞", 22};
    print (s);
    print (s);
    foo (100);
    return 0;
}
```

十二、显示类型转换运算符

C：目标类型变量 = (目标类型)源类型变量；

1.静态类型转换

static_cast<目标类型> (源类型变量)

如果在目标类型和源类型之间某一个方向上可以做隐式类型转换，那么在两个方向上都可以做静态类型转换。反之如果在两个方向上都不能做隐式类型转换，那么在任意一个方向上也不能做静态类型转换。

```
int* p1 = ...;
void* p2 = p1;
p1 = static_cast<int*> (p2);
char c;
int i = c;
```

如果存在从源类型到目标类型的自定义转换规则，那么也可以使用静态类型转换。

例子：静态类型转换

```
#include <iostream>
using namespace std;
int main (void) {
    int* p1 = NULL;                 //p1 为 int 型的指针
    void* p2 = p1;                  //p2 为 void 型的指针
    p1 = static_cast<int*> (p2);    //将 void 型的 p2 指针转换为 int 型指针并复制给 int 型的 p1 指针。
    return 0;
}
```

2.动态类型转换

dynamic_cast<目标类型> (源类型变量)

用在具有多态性的父子类指针或引用之间。

3.常类型转换

const_cast<目标类型> (源类型变量)

给一个拥有 const 属性的指针或引用去常

```
const int a = 100;
const int* p1 = &a;
*p1 = 200; // ERROR
int* p2 = const_cast<int*> (p1);
*p2 = 200; // OK
```

4.从解释类型转换

reinterpret_cast<目标类型> (源类型变量);

在不同类型的指针或引用之间做类型转换，以及在指针和整型之间做类型转换。

5.目标类型变量 = 目标类型(源类型变量)；

```
int a = int (3.14);
```

例子一：常类型转换

```
#include <iostream>
using namespace std;
int main (void) {

    const volatile int a = 100;
```

(关键字 volatile 在描述变量时使用,阻止编译器优化那些以 valatile 修饰的变量,volatile 被用在一些变量能被意外方式改变的地方,例如:抛出中断,这些变量若无 volatile 可能会和编译器执行的优化 相冲突.)

```
//  a = 200;
    const volatile int* p1 = &a;
//  *p1 = 200;
    int* p2 = const_cast<int*> (p1); //去常，去掉常属性
    *p2 = 200;
    cout << *p2 << endl;    // 200
    cout << a << endl;      // 200
    // cout << 100 << endl;
    return 0;

}
```

例子二：解释型类型转换

```
#include <iostream>
using namespace std;
int main (void) {
    int i = 0x12345678;
    char* p = reinterpret_cast<char*> (&i);
    for (size_t i = 0; i < 4; ++i)
        cout << hex << (int)p[i] << ' ';  //78 56 34 12
    cout << endl;
    float* q = reinterpret_cast<float*> (&i);
    cout << *q << endl;
    void* v = reinterpret_cast<void*> (i);
    cout << v << endl;
    return 0;
}
```

十三、C++之父的建议

1.少用宏，多用 const、enum 和 inline

```
#define PAI 3.141519
const double PAI = 3.14159;
#define ERROR_FILE -1
#define ERROR_MEM  -2
enum {
  ERROR_FILE = -1,
  ERROR_MEM = -2
};
#define max(a,b) ((a)>(b)?(a):(b))
inline int double max (double a, double b) {
  return a > b ? a : b;
}
```

2.变量随用随声明同时初始化。

3.少用 malloc/free，多用 new/delete。

4.少用 C 风格的强制类型转换，多用类型转换运算符。

5.少用 C 风格的字符串，多用 string。

6.树立面向对象的编程思想。

附加：

string 类：

例子：

```
#include <iostream>
#include <cstring>
#include <cstdio>
using namespace std;
int main (void) {
    string s1 ("hello");
    string s2 = "world";
    (s1 += " ") += s2;
    cout << s1 << endl;
    string s3 = s1;
    cout << s3 << endl;
    cout << (s1 == s3) << endl;
    s3[0] = 'H';
    cout << s3 << endl;
    cout << (s1 > s3) << endl;
    cout << s1.length () << endl;
    cout << (s1 == s3) << endl;
    cout << (strcasecmp (s1.c_str (),
        s3.c_str ()) == 0) << endl;
    cout << sizeof (s1) << endl;
    FILE* fp = fopen ("string.txt", "w");
//  fwrite (&s3, sizeof (s3), 1, fp);
    fwrite (s3.c_str (), sizeof (char),
        s3.length (), fp);
    fclose (fp);
    return 0;
}
```

第二课　类和对象

一、什么是对象

1.万物皆对象
2.程序就是一组对象，对象之间通过消息交换信息
3.类就是对对象的描述和抽象，对象就是类的具体化和实例化

二、通过类描述对象

属性：姓名、年龄、学号
行为：吃饭、睡觉、学习
类就是从属性和行为两个方面对对象进行抽象。

三、面向对象程序设计(OOP)

现实世界　　虚拟世界
对象　　-> 抽象 -> 类 -> 对象
1.至少掌握一种 OOP 编程语言
2.精通一种面向对象的元语言—UML
3.研究设计模式，GOF

四、类的基本语法

1.类的定义

```
class 类名 {
};
```

如

```
class Student {
};
```

2.成员变量——属性

```
class 类名 {
  类型 成员变量名;
};
```

如

```
class Student {
  string m_name;
  int    m_age;
};
```

3.成员函数——行为

```
class 类名 {
  返回类型 成员函数名 (形参表) {
    函数体;
  }
};
```

如

```
class Student {
  string m_name;
  int    m_age;
  void eat (const string& food) {
    ...
  }
};
```

4.访问控制属性

1)公有成员：public，谁都可以访问。
2)私有成员：private，只有自己可以访问。
3)保护成员：protected，只有自己和自己的子类可以访问。
4)类的成员缺省访控属性为私有，而结构的成员缺省访控属性为公有。

例子：

```
#include <iostream>
using namespace std;
class Student {
private:          //声明为私有部分
    string m_name;
    int m_age;
public:           //声明为私有部分
    void eat (const string& food) {
        cout << m_age << "岁的" << m_name
            << "同学正在吃" << food << "。" << endl;
    }
    void setName (const string& name) {  //为接口
        if (name == "2")
            cout << "你才" << name << "! " << endl;
        else
            m_name = name;
    }
    void setAge (int age) {              //为接口
        if (age < 0)
            cout << "无效的年龄! " << endl;
        else
            m_age = age;
    }
};
int main (void) {
    Student student;
   student.setName ("2");    //你才 2
   student.setAge (-100);    //无效年龄
   student.setName ("张飞"); //将其赋值给成员变量 m_name
```

```
    student.setAge (20);        //将其赋值给成员变量 m_age
     student.eat ("包子");      //20 岁的张飞同学正在吃包子
     return 0;
}
```

5.构造函数

```
class  {
  ...
  类名 （行参表） {
    构造函数体;
  }
};
```

当一个对象被创建时，构造函数会自动被执行,其参数来自构造实参。

6.构造函数可以通过构造参数实现重载。

7.如果一个类没有定义任何构造函数，那么系统就会缺省地为其提供一个无参构造函数，该构造函数对于基本类型的成员变量不做初始化，对于类类型的成员变量，调用其相应类型的无参构造函数初始化。

8.对象创建过程

分配内存->调用构造函数->调用类类型成员的构造函数->构造函数的代码

9.初始化表

```
class 类名 {
  类名(...) :初始化表 {
    构造函数体
  }
};
const int x = 100;
x = 100;
int& a = b;
a = b;
```

1)**如果类中含有常量或引用型的成员变量，必须通过初始化表对其初始化。**

2)成员变量的初始化顺序仅于其被声明的顺序有关，而与初始化表的顺序无关。

```
class A {
public:
    A (char* psz) : m_str (psz),
       m_len (m_str.length()) {}
private:
   size_t m_len;
   string m_str;
}
```

例子 1：类的声明与定义以及使用可以不在一个文件

这是 s.h 文件

```
#ifndef _S_H
#define _S_H
#include <string>
using namespace std;
// 声明 Student 类
class Student {
public:
    Student (const string& name = "", int age = 0);
    void eat (const string& food);
private:
    string m_name;
    int m_age;
};
#endif // _S_H
```

这是 s.cpp 文件

```
#include <iostream>
using namespace std;
#include "s.h"
// 实现 Student 类
Student::Student (const string& name /* = "" */,
    int age /* = 0 */) : m_name (name),
    m_age (age) {}
void Student::eat (const string& food) {
    cout << m_name << ", " << m_age << ", " << food
        << endl;
}
```

这是 main.cpp 文件：

```
#include "s.h"
// 使用 Student 类
int main (void) {
    Student s ("张飞", 25);
    s.eat ("包子");
    return 0;
}
```

例子 2：不同的构造函数

```
#include <iostream>
using namespace std;
class A {
public:
    A (int a) {}
};
class Student {
private:
    string m_name;
    int m_age;
    A m_a;          //类类型的成员变量
public:
    void eat (const string& food) {
        cout << m_age << "岁的" << m_name
            << "同学正在吃" << food << "。" << endl;
    }

//  void _ZN7Student3eatERKSs (Student* this,
//      const string& food) {
//      cout << this->m_age << "岁的"<<this->m_name
//          << "同学正在吃" << food << "。" << endl;
//  }     //这是计算机中编译的样子

    void setName (const string& name) {
        if (name == "2")
            cout << "你才" << name << "! " << endl;
        else
            m_name = name;
    }
    void setAge (int age) {
        if (age < 0)
            cout << "无效的年龄! " << endl;
        else
            m_age = age;
    }

  //如果同时有多种构造函数存在，则根据构造参数来确定调用哪个构造函数，既构造函数可以通过构造参数实现重载
  // 构造函数
    Student (const string& name, int age) :
        m_a (100) {
        m_name = name;
```

```
        m_age = age;
    }

    // 无参构造
    Student (void) : m_a (100) {      //没有参数
        m_name = "没名";
        m_age = 0;
    }

   // 单参构造
    Student (const string& name) : m_a (100),
        m_name (name),   m_age (0) {
        m_name = "哈哈";
    }
};
int main (void) {
    Student s1 ("张飞", 25);
    s1.eat ("包子");
//  _ZN7Student3eatERKSs (&s1, "包子");   //编译器中的样子

   Student s2 = Student ("赵云", 22);
   s2.eat ("烧饼");
//  _ZN7Student3eatERKSs (&s2, "烧饼");   //编译器中的样子

   Student s3;                           //调用的无参构造
    s3.eat ("煎饼果子");


   Student* s4   = new Student ("关羽", 30);//调用有参构造，分配内存，并初始化
    s4->eat ("烤鸭"); //当访问地址（指针或迭代器）的成员或数据时，用->
    delete s4;

   Student& s5 = *new Student ();//调用无参构造初始化
    s5.eat ("面条");              //当访问直接对象的成员或数据时，用“.”
    delete &s5;

    Student sa1[3] = {s1, s2};    //用 s1 与 s2 给数组初始化，但第三个元素没有赋值
    sa1[0].eat ("KFC");           //25 岁的张飞同学正在吃 KFC
    sa1[1].eat ("KFC");           //22 岁的赵云同学正在吃 KFC
    sa1[2].eat ("KFC");           //0 岁的无名同学正在吃 KFC

   Student* sa2 = new Student[3] {s1, s2}; // c++2011 支持
    sa2[0].eat ("KFC");
    sa2[1].eat ("KFC");
```

```
    sa2[2].eat ("KFC");
    delete[] sa2;

Student s6 ("刘备");//调用单参构造
    s6.eat ("米饭");//
    return 0;
}
```

练习：实现一个 Clock 类支持两种工作模式，计时器和电子钟。

00:01:00
14:05:37
#include <iomanip>
cout << setw(4) << setfill('0') << 12;
0012
Clock
  时、分、秒
  走 - 显示、滴答

练习答案：

```
#include <iostream>
#include <iomanip>
using namespace std;
class Clock {
public:
    Clock (bool timer = true) :
        m_hour (0), m_min (0), m_sec (0) {
        if (! timer) {
            time_t t = time (NULL);
            tm* local = localtime (&t);
            m_hour = local->tm_hour;
            m_min = local->tm_min;
            m_sec = local->tm_sec;
        }
    }
    void run (void) {
        for (;;) {
            show ();
            tick ();
        }
    }
private:
    void show (void) {
```

```
        cout << '\r' << setfill ('0')
            << setw (2) << m_hour << ':'
            << setw (2) << m_min << ':'
            << setw (2) << m_sec << flush;
//      printf ("\r%02d:%02d:%02d", m_hour,
//          m_min, m_sec);
    }
    void tick (void) {
        sleep (1);
        if (++m_sec == 60) {
            m_sec = 0;
            if (++m_min == 60) {
                m_min = 0;
                if (++m_hour == 24)
                    m_hour = 0;
            }
        }
    }
    int m_hour;
    int m_min;
    int m_sec;
};
int main (void) {
    Clock clock (false);
    clock.run ();
    return 0;
}
```

五、this 指针

1.一般而言，在类的构造函数或成员函数中，关键字 this 表示一个指针，对于构造函数而言，this 指向正在被构造的对象，对于成员函数而言，this 指向调用该函数的对象。
2.this 指针的用途
1)在类的内部区分成员变量。
2)在成员函数中返回调用对象自身。
3)在成员函数内部通过参数向外界传递调用对象自身，以实现对象间交互。
老 -问题-> 学
师 <-答案- 生

```
class A {
   B m_b;
};
class B {
   A m_a;
};
sizeof(A)  //error
class C {
   C m_c;
};
```

例子 1：

```
#include <iostream>
using namespace std;
class A {
public:
    A (int data) : data (data) {
        cout << "构造: " << this << endl;
//      this->data = data;
    }
    void foo (void) {
        cout << "foo: " << this << endl;
        cout << this->data << endl;
    }
    int data;
};
int main (void) {
 A a (1000);                //创建对象调用了构造函数，并输出 this 的地址，输出“构造：0xbf9b24d8”
cout << "main: " << &a << endl;//输出该对象的地址，输出“main:0xbf9b24d8”
a.foo ();                   //该对象调用 foo 函数，输出 this 的值，以及输出 this->data 的值，输出“foo：0xbf9b24d8  1000”
A* pa = new A (1000);       //创建对象调用构造函数，输出 this 的地址，输出“构造：
```

0x95cc008”
cout << "main: " << pa << endl; //输出该对象的地址，输出“main：0x95cc008”
pa->foo (); //该对象调用 foo 函数，输出 this 以及 this->data 的值，输出“foo：0x95cc008 1000”
delete pa;
}

例子 2：

```
#include <iostream>
using namespace std;
class Counter {
public:
    Counter (void) : m_data (0) {}
    Counter& inc (void) {  //返回的是一个别名，不加&的话，返回的就是一个拷贝
        ++m_data;
        return *this;
    }
    void print (void) {
        cout << m_data << endl;
    }
private:
    int m_data;
};
int main (void) {
    Counter c;
//  c.inc ();
//  c.inc ();
//  c.inc ();
    c.inc ().inc ().inc ();//函数返回的是一个别名，是一个左值，可以用来调用函数
    c.print ();            // 输出为 3，如果前面的函数不加&，返回的只是拷贝，输出为 1。
    return 0;
}
```

例子 3：学生与老师

```
#include <iostream>
using namespace std;
class Student;   //因为在 Teacher 中会用到 Student，所以提前声明一下
class Teacher {
public:
    void educate (Student* s);//可以声明在类的内部，定义在类的外部
    void reply (const string& answer) {
        m_answer = answer;
    }
private:
    string m_answer;
};
class Student {
public:
    void ask (const string& question, Teacher* t) {
        cout << "问题：" << question << endl;
        t->reply ("不知道。");
    }
};
void Teacher::educate (Student* s) {
    s->ask ("什么是 this 指针？", this);//将问题 question 和 Teacher 类变量的地址作为参数传递给 Student 类中的 ask 成员函数，并在 ask 函数中得到一个值作为参数传递给 Teacher 类中的 replay 函数，将值赋给 m_answer，最后完成输出。
    cout << "答案：" << m_answer << endl;
}
int main (void) {
    Teacher t;
    Student s;
    t.educate (&s);
    return 0;
}
```

六、常函数与常对象

1.如果在一个类的成员函数的参数表后面加上 const 关键字，那么这个成员函数就被成为常函数，常函数的 this 指针是一个常指针。在常函数内部无法修改成员变量，除非该变量具有 mutable 属性。而且在常函数内部也无法调用非常函数。
2.常对象:拥有 const 属性的对象，兑现引用或指针。
常对象只能调用常函数。
同型的常函数和非常函数可以构成重载关系。常对象调用常版本，非常对象调用非常版本。如果没有非常版本，非常对象也可以调用常版本。

```
const XXX 函数名 (const YYY yyy)
const {
  ...
}
```

例子：

```
#include <iostream>
using namespace std;
class A {
public:
//  void bar (void) {            //函数 1
//      cout << "非常 bar" << endl;
//  }
//函数 1 与函数 2 构成函数重载
    void bar (void) const {      //函数 2
        cout << "常 bar" << endl;
    }
//  void XXXbarYYY (A* this) {}
      void foo (void) const {      //函数 3
//      m_i = 100;  //在常函数内部无法修改成员变量，除非那个成员变量有 mutable 属性，例：mutable int m_i;
        const_cast<A*>(this)->m_i = 100;//也可以通过去常来解决
    }
    void print (void) const {
        cout << m_i << endl;
    }
//  _ZNK1A3fooEv (const A* this) {
//      const_cast<A*>(this)->m_i = 100;
//  }
    int m_i;
};
void func (void) /*const*/ {}
int main (void) {
    A a;
```

```
    a.foo ();  //调用的是函数 3
    a.print ();//“100”

  const A& r = a;//r 为常对象，a 为非常对象
    r.bar ();//“常 bar”，r 为常对象，常对象只能调用常函数
//  XXXbarYYY (&r); // const A*

  a.bar ();//“常 bar”，a 为非常对象，可以调用常函数，也可调用非常函数
//  XXXbarYYY (&a); // A*
    return 0;
}
```

七、析构函数

```
class 类名 {
  ~类名 (void) {
     析构函数体;
  }
};
```

当一个对象被销毁时自动执行析构函数。
局部对象离开作用域时被销毁，堆对象被 delete 时被销毁。
如果一个类没有定义任何析构函数，那么系统会提供一个缺省析构函数。缺省析构函数对基本类型的成员变量什么也不干，对类类型的成员变量，调用相应类型的析构函数。
一般情况下，在析构函数中释放各种动态分配的资源。
构造：基类->成员->子类
析构：子类->成员->基类

例子：

```
#include <iostream>
using namespace std;
class Double {
public:
    Double (double data) :
        m_data (new double (data)) {
        cout << "构造" << endl;
    } //构造函数早开始时执行，既创建对象时执行
    ~Double (void) {
        cout << "析构" << endl;
        delete m_data;
    }//析构函数在对象结束时执行
    void print (void) const {
        cout << *m_data << endl;
```

```
    }
private:
    double* m_data;
    string m_lable;
};
int main (void) {
//  {
        Double d1 (3.14);
        d1.print ();//“构造,3.14”（d1 调用的构造）
//  }
    Double* d2 = new Double (1.23);//“构造”（d2 调用的构造）
    delete d2;      //“析构”（d2 调用的析构）
    cout << "再见！" << endl;//“再见”
    return 0;       //“析构”,（程序结束，d1 调用的析构）
}
```

八、拷贝构造函数和拷贝赋值运算符

1. 拷贝构造：用一个已有的对象，构造和它同类型的副本对象——克隆。

2. 拷贝构造函数

形如

```
class X {
  X (const X& that) { ... }
};
```

的构造函数成为**拷贝构造函数**。如果一个类没有定义拷贝构造函数，系统会提供一个缺省拷贝构造函数。缺省拷贝构造函数对于基本类型的成员变量，按字节复制，对于类类型的成员变量，调用相应类型的拷贝构造函数。

3. 在某些情况就下，**缺省拷贝构造函数只能实现浅拷贝**，如果需要获得深拷贝的复制效果，就需要**自己定义拷贝构造函数**。

例子：拷贝函数

```
#include <iostream>
using namespace std;
class Integer {
public:
    Integer (int data = 0) : m_data (data) {}//构造函数
    void print (void) const {
        cout << m_data << endl;
    }
```

```
//拷贝构造（自己定义的）：
    Integer (const Integer& that) :
        m_data (that.m_data) {
        cout << "拷贝构造" << endl;
    }

private:
    int m_data;
};
void foo (Integer i) {    //用 Inerger 类变量时实参给函数中 Interger 类的形参赋值，同样会调用拷贝构造函数
    i.print ();
}
Integer bar (void) {
    Integer i;
    return i;
}
int main (void) {
    Integer i1 (10);
    i1.print ();//正常创建对象，输出“10”
    Integer i2 (i1); // 调用拷贝构造，输出“拷贝构造”
    i2.print ();  //调用 print 函数，输出“10”
    Integer i3 = i1; // 调用拷贝构造，输出“拷贝构造”
    i3.print ();  //调用 print 函数，输出“10”
//  Integer i4 (10, 20);
    cout << "调用 foo()函数" << endl;
    foo (i1);     //调用拷贝构造函数，且调用 print 函数输出，所以输出为“拷贝构造 10”
    cout << "调用 bar()函数" << endl;
    Integer i4 (bar ());
    return 0;
}
```

4.拷贝赋值运算符函数

形如

```
class X {
  X& operator= (const X& that) {
    ...
  }
};
```

的成员函数称为**拷贝赋值运算符函数**。如果一个类没有定义拷贝赋值运算符函数，系统会提供一个缺省拷贝赋值运算符函数。缺省拷贝赋值运算符函数对于基本类型的成员变量，按字节复制，对于类类型的成员变量，调用相应类型的拷贝赋值运算符函数。

5.在某些情况就下，缺省拷贝赋值运算符函数只能实现浅拷贝，如果需要获得深拷贝的复制效果，就需要自己定义拷贝赋值运算符函数。

例子：拷贝赋值运算符函数

```
#include <iostream>
using namespace std;
class Integer {
public:
    Integer (int data) : m_data (new int (data)) {}
//构造函数
    ~Integer (void) {    //析构函数
        if (m_data) {
            delete m_data;
            m_data = NULL;
        }
    }
    void print (void) const {
        cout << *m_data << endl;
    }
    Integer (const Integer& that) :   //拷贝构造函数
        m_data (new int (*that.m_data)) {}
    void set (int data) {
        *m_data = data;
    }
```

```
//拷贝赋值运算符函数（运算符重载）
    Integer& operator= (const Integer& that) {
            // 防止自赋值
        if (&that != this) {
            // 释放旧资源
            delete m_data;
            // 分配新资源
            m_data = new int (*that.m_data);
            // 拷贝新数据
        }
        // 返回自引用
        return *this;
    }
private:
    int* m_data;
};
int main (void) {
    Integer i1 (10);
    i1.print ();
    Integer i2 (i1);
    i2.print ();
    i2.set (20);
    i2.print ();
    i1.print ();
    Integer i3 (30);
    i3.print (); // 30
    i3 = i1; // 拷贝赋值
    // i3.operator= (i1);
    i3.print (); // 10
    i3.set (40);
    i3.print (); // 40
    i1.print (); // 10
    /*
    int a = 10, b = 20, c = 30;
    (a = b) = c;
    cout << a << endl;
    */
    (i3 = i1) = i2;
//  i3.operator=(i1).operator=(i2);
    i3.print ();
    i3 = i3;
    i3.print ();
    return 0;
}
```

练习：实现一个简化版的字符串类 String 支持：

1.用字符指针形式的字符串构造
2.拷贝构造和拷贝赋值
3.获取字符指针形式字符串的成员函数，类似 string::c_str
说明：不得使用 std::string!

练习答案：

```
#include <iostream>
#include <cstring>
using namespace std;
class String {
public:
    String (const char* str = NULL) {   //构造函数
        m_str = new char[strlen(str?str:"")+1];
        strcpy (m_str, str ? str : "");
    }
    ~String (void) {                    //析构函数
        if (m_str) {
            delete[] m_str;
            m_str = NULL;
        }
    }
    String (const String& that) :       //拷贝构造函数
        m_str (strcpy (
            new char[strlen(that.m_str)+1],
            that.m_str)) {}
    /* 菜鸟
    void operator= (const String& that) {
        m_str = new char[strlen(that.m_str)+1];
        strcpy (m_str, that.m_str);
    }*/
    String& operator= (const String& that) {
        if (&that != this) {
            /* 小鸟
            delete[] m_str;
            m_str = new char[strlen(that.m_str)+1];
            strcpy (m_str, that.m_str);
            */

            /* 大鸟
            char* str =
                new char[strlen(that.m_str)+1];
            delete[] m_str;
```

```
            m_str = strcpy (str, that.m_str);
            */

            // 老鸟
            String temp (that);
            swap (m_str, temp.m_str);
        }
        return *this;
    }
    const char* c_str (void) const {
        return m_str;
    }
private:
    char* m_str;
};
int main (void) {
    String s1 ("Hello, World !");
    cout << s1.c_str () << endl;
    String s2 = s1;
    cout << s2.c_str () << endl;
    String s3 ("Hello, Linux !");
    try {
        s1 = s3;
    }
    catch (exception& ex) {
        cout << ex.what () << endl;
    }
    cout << s1.c_str () << endl;
    return 0;  }
```

九、静态成员

1.静态成员变量和静态成员函数是属于类的而非属于对象。
2.静态成员变量，为多个对象所共享，只有一份实例，可以通过对象访问也可以通过类访问，必须在类的外部定义并初始化。静态成员变量本质上与全局变量并没有区别，只是多了类作用域的约束，和访控属性的限制。

```
class Account {
private:
  string m_name;
  double m_balance;
  static double m_rate;
};
```

3.静态成员函数，没有 this 指针，无法访问非静态成员。
4.单例模式

例子 0：

```
#include <iostream>
using namespace std;
class A {
public:
    static int m_i;
    static void foo (void) {
        cout << "foo：" << m_i << endl;
//      m_d = 3.14;//报错，静态成员函数不能访问非静态成员成员
//      bar ();    //报错，理由同上
    }
    double m_d;
    void bar (void) {
        m_i = 1000;//OK，非静态成员函数可以访问非静态成员，也可以访问静态成员
        foo ();   //OK，理由同上
    }
};
int A::m_i = 1;//在外部定义
int main (void) {
    A::m_i = 10;//通过类访问静态成员变量

    A a1, a2;
    cout << ++a1.m_i << endl;//通过对象访问静态成员变量，输出为“11”
    cout << a2.m_i << endl;  //因为静态成员变量，为多个对象共享，只有一个实例，所以上面 a1 将 m_i 修改为 11，则通过 a2 访问的 m_i 也是 11，输出为“11”

    A::foo ();//输出为“foo:11”，通过类访问静态成员函数
    a1.foo ();//输出为“foo:11”，通过对象访问静态成员函数
    a1.bar ();//先调用 bar，将 m_i 修改为 1000，再调用 foo，输出为“foo：1000”
```

```
    return 0;
}
```

例子 1：单例模式（饿汉方式）：

```
#include <iostream>
using namespace std;
// 饿汉方式
class Singleton {
public:
    static Singleton& getInst (void) {
        return s_inst;  //当调用这个函数时，不会新创建对象，不会调用构造函数，只是返回创建好的那个对象
    }
private:
    Singleton (void) {}          //构造函数
    Singleton (const Singleton&);//拷贝构造函数
    static Singleton s_inst;     //（类的内部）静态成员变量的声明，所有对象公用
};
Singleton Singleton::s_inst;//（类的外部）静态成员变量的定义，这个时候已经创建了对象，并调用构造函数
int main (void) {
    Singleton& s1 = Singleton::getInst ();//不会创建新的对象，返回已经创建好的 Singletion 类对象，并没有新分配内存和地址
    Singleton& s2 = Singleton::getInst ();//与上面一样，还是返回那个对象，内存地址没有变
    Singleton& s3 = Singleton::getInst ();//还是一样滴
    cout << &s1 << ' ' << &s2 << ' ' << &s3 << endl;
//输出的都是 0x804a0d4
    return 0;
}
```

例子 2：单例模式（懒汉模式）

```
#include <iostream>
using namespace std;
// 懒汉方式
class Singleton {
public:
    static Singleton& getInst (void) {
        if (! m_inst)   //m_inst 指针被初始化为 NULL，第一次调用时，执行下面那一行代码来分配内存创建一个对象。否则跳过那一行代码。一旦执行过一次下一行代码，则 m_inst 就不会再为 NULL，也就是说只会分配一次内存，只有一个对象
```

```
        m_inst = new Singleton;//分配内存，创建对象，执行一次
        ++m_cn;//调用一次该函数，则数量加一
        return *m_inst;//返回创建好的 m_inst
    }
    void releaseInst (void) {
        if (m_cn && --m_cn == 0)//调用了几次 getInst，就要调用几次该函数，才能真正把对象释放掉
            delete this;
    }
private:
    Singleton (void) {  //构造函数
        cout << "构造：" << this << endl;
    }
    Singleton (const Singleton&);//析构函数
    ~Singleton (void) {
        cout << "析构：" << this << endl;
        m_inst = NULL;
    }
    static Singleton* m_inst;//声明该指针(静态)
    static unsigned int m_cn;//声明该变量(静态)
};
Singleton* Singleton::m_inst = NULL;//先将指针初始化，并没有创建对象，分配内存
unsigned int Singleton::m_cn = 0;//创建对象并初始化，调用构造，这个变量什维利计算 m_inst 的数量
int main (void) {
    Singleton& s1 = Singleton::getInst ();//第一次调用 getInst 函数，所以执行哪一行代码来分配内存创建对象，并返回该对象
    Singleton& s2 = Singleton::getInst ();//因为不是第一次调用，m_inst 已经不再指向 NULL，所以根据条件语句，不执行创建对象的代码，直接返回原先第一次调用还函数时创建好的对象，地址神马的都不变
    Singleton& s3 = Singleton::getInst ();//同上
    cout << &s1 << ' ' << &s2 << ' ' << &s3 << endl;
//输出的地址都是一样的，因为只分配了一次内存，创建了一次对象
    s3.releaseInst ();//调用时，m_cn 为 3，结束时为 2，没有释放掉
    s2.releaseInst ();//调用时，m_cn 为 2，结束时为 1，没有释放掉
    s1.releaseInst ();//调用时，m_cn 为 1，满足条件语句，执行 delete this ，真正释放掉
    return 0;
}
```

十、成员指针

```
Student s;
string* p = &s.m_name; // 不是
Student s2;
```

(一)成员变量指针

1.定义

成员变量类型 类名::*指针变量名;

```
string Student::*pname;
int Student::*page;
```

2.初始化/赋值

指针变量名 = &类名::成员变量名

```
pname = &Student::m_name;
page = &Student::m_age;
```

3.解引用

对象.*指针变量名

对象指针->*指针变量名

```
Student s, *p = &s;
s.*pname = "张飞";
cout << p->*page << endl;
```

(二)成员函数指针

1.定义

成员函数返回类型 (类名::*指针变量名) (参数表)

```
void (Student::*plearn) (const string&) const;
```

2.初始化/赋值

指针变量名 = &类名::成员函数名;

```
plearn = &Stduent::learn;
```

3.解引用

(对象.*指针变量名) (实参表);

(对象指针->*指针变量名) (实参表);

```
(s.*plearn) ("C++");
(p->*plearn) ("UNIX");
```

例子：

```
#include <iostream>
#include <cstring>
using namespace std;
class Student {
public:
    Student (const string& name, int age) : //构造函数
        m_name (name), m_age (age) {}

    double m_weight;
    string m_name;
```

```
    Int m_age;
    void learn (const string& lesson) const {
        cout << "我在学" << lesson << endl;
    }
    static void hello (void) {
        cout << "你好！" << endl;
    }
};
int main (void) {
    //成员变量指针使用对象的地址作为基础，然后用成员变量的偏移量得到该对象中的成员变量的绝对地址。
    string Student::*pname = &Student::m_name;//声明一个指向 Student 类中 m_name 成员变量的成员变量指针
    void* pv;
    memcpy (&pv, &pname, 4);
    cout << pv << endl;//输出“0x8”

    int Student::*page = &Student::m_age;  //声明一个指向 Student 类中 m_age 成员变量的成员变量指针
    memcpy (&pv, &page, 4);
    cout << pv << endl;//输出“0xc”
    Student s ("张飞", 25), *p = &s;//p 是指向对象的指针。存的是对象 s 的地址
    cout << s.*pname << endl;//成员变量指针，输出“张飞”
    cout << p->*page << endl;//成员变量指针，输出“25”

    Student s2 ("赵云", 22);
    cout << s2.*pname << endl;//输出“赵云”

    void (Student::*plearn) (const string&) const =
        &Student::learn;//成员函数指针
    (s.*plearn) ("C++");//对象是用地址提供 this 的实参，输出“我在学 c++”
    (p->*plearn) ("UNIX");//输出“我在学 UNIX”

    void (*phello) (void) = Student::hello;
    phello ();//静态的函数就不需要用对象的地址来提供 this 的实参，输出“你好”
    return 0;
}
```

第三课　操作符重载

复数：3+4i

Complex
c1 - (c2 + c3)
c1.sub (c2.add (c3))

一、操作符标记和操作符函数

1.双目操作符：L#R

成员函数形式：L.operator# (R)
左调右参
全局函数形式：::operator# (L, R)
左一右二

2.单目操作符：#0/0#

成员函数形式：0.operator# ()
全局函数形式：::operator# (0)

3.三目操作符：不考虑

二、双目操作符

1.　+/-/*//

操作数在计算前后不变。
表达式的值是右值。

例子：

```
#include <iostream> //操作符重载
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }
    // 第一个 const：返回右值，返回的对象不能接受赋值。
    // 第二个 const：支持常量型右操作数，可以接受有常属性的变量为参数，将引用声明为常引用才能指向常变量，常引用也可以指向非常变量。
    // 第三个 const：支持常量型左操作数，使 this 指针变为常指针，也可以指向有常属性的变量。
    const Complex operator+ (const Complex& r) const { //操作符重载的成员函数形式：
```

L.operator+(R)

```
        return Complex (m_r + r.m_r, m_i + r.m_i);
    }
private:
    int m_r;
    int m_i;
    friend const Complex operator- (const Complex&,
        const Complex&);   //将该函数声明为友远这样该函数就可以访问类的私有部分
};
const Complex operator- (const Complex& l,const Complex& r) {          //操作符重
```

载的全局函数形式，::operator-(L,R)

```
    return Complex (l.m_r - r.m_r, l.m_i - r.m_i);
}
int main (void) {
    const Complex c1 (1, 2);
    c1.print ();
    const Complex c2 (3, 4);
    Complex c3 = c1 + c2; // c3 = c1.operator+ (c2)
    c3.print (); // 4+6i
//  (c1 + c2) = c3;
    c3 = c2 - c1; // c3 = ::operator- (c2, c1)
    c3.print (); // 2+2i
    return 0;
}
```

---

2. +=/-=/*=...

左变右不变。

表达式的值是左值，左操作数的引用。

(a += b) = c;

例子：

```
#include <iostream>
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }
        //成员函数形式:
    Complex& operator+= (const Complex& r) {  //返回的是左操作数的引用，是可以赋值
```

的，所以最前面不加 const，调用该函数的左操作数是要改变的也就是 this 会修改，也就是调用函数的参数被修改，所以{前也不加 const。

```
        m_r += r.m_r;
        m_i += r.m_i;
        return *this;//返回该调用操作数
    }
        //全局函数形式:（把定义与声明合二为一，因为有 friend，所以是全局函数）
    friend Complex& operator-= (Complex& l,
        const Complex& r) {            //第一个参数 l 为左操作数引用，可被修改，所以不加 const，而第二个参数 r 为右操作数，值不会被修改，所以加 const。
        l.m_r -= r.m_r;
        l.m_i -= r.m_i;
        return l;//返回左操作数
    }
private:
    int m_r;
    int m_i;
};
int main (void) {
    Complex c1 (1, 2), c2 (3, 4);
    c1 += c2; // c1.operator+= (c2)
    c1.print (); // 4+6i
    Complex c3 (5, 6);
    (c1 += c2) = c3;//返回的是左操作数 c1 的引用，把 c3 赋值给 c1.
    c1.print (); // 5+6i
    c1 -= c2; // ::operator-= (c1, c2)
    c1.print (); // 2+2i
    (c1 -= c2) = c3;
    c1.print (); // 5+6i;
    return 0;
}
```

---

3. <</>>

```
int i = 10;
float f = 1.23;
Complex c (...);
cout << c << i << f << endl;
cin >> c;
```

左操作数 ostream/istream 类型，不能是常量，不能拷贝。
右操作数自定义类型，对于<<可以是常量，对于>>不能是常量。
表达式的值是左操作数的引用。

```
::operator<< (cout, c).operator<< (i).operator<< (f).operator<<
(endl);
```

例子：

```
#include <iostream>
//输入与输出。
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }
        //全都使用全局函数的形式
    friend ostream& operator<< (ostream& os,  //返回的是左操作数的引用
        const Complex& r) {              //因为右操作数是要输出的，所以右操作数可
以是常量，所以声明为常引用，加 const，也可做非常变量的引用。
        return os << r.m_r << '+' << r.m_i << 'i';//返回左操作数的引用
    }

        friend istream& operator>> (istream& is,//返回的是左操作数的引用
        Complex& r) {                    //因为右操作数是要输入的，所以右操作数不能
是常量，不加 const
        return is >> r.m_r >> r.m_i;     //返回左操作数的引用
    }
private:
    int m_r;
    int m_i;
};
int main (void) {
    Complex c1 (1, 2), c2 (3, 4);
    cout << c1 << endl << c2 << endl;
//  ::operator<<(::operator<<(cout,c1).operator<<(
//      endl),c2).operator<<(endl);
    cin >> c1 >> c2;
    cout << c1 << endl << c2 << endl;
    return 0;
}
```

三、单目操作符

1.-(取负)/!/~

操作数不变。
表达式的值是右值。

例子：

```
#include <iostream>
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }

        //成员函数形式：
    const Complex operator- (void) const {  //只有一个操作数，且作为调用参数，所以没有形参。因为返回的是右值，是常量，所以要加 const（第一个 const）。操作数不改变，就是 this 不可改变，也加 const（第二个 const）。
        return Complex (-m_r, -m_i);//返回的是右值（常数）
    }
        //全局函数形式：
    friend const Complex operator~ (
        const Complex& o) {       //就一个操作数，且不改变，加 const
        return Complex (o.m_i, o.m_r);//交换实部与虚部
    }
private:
    int m_r;
    int m_i;
};
int main (void) {
    const Complex c1 (1, 2);
    Complex c2 = -c1; // c2=c1.operator-()
    c2.print (); // -1+-2i
    Complex c3 = ~c1; // c3=::operator~(c1)
    c3.print (); // 2+1i;
    return 0;
}
```

2. 前++/前--

操作数变。
表达式的值是运算以后的值。
表达式的值是左值，操作数的引用。
(++i) = 100;
++++++i;

例子：

```
#include <iostream>
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }
        //成员函数形式：
    Complex& operator++ (void) {   //只有一个操作数，且该操作数是改变的，所以前{不加 const。而返回的是一个可以改变的左值，既操作数的引用，所以可以改变，最前面不加 const。
        ++m_r;
        ++m_i;
        return *this;
    }
        //全局函数形式：
    friend Complex& operator-- (Complex& o) {  //返回的是操作数本身，既引用。
        --o.m_r;
        --o.m_i;
        return o;       //返回操作数的引用。
    }
private:
    int m_r;
    int m_i;
};
int main (void) {
    Complex c1 (1, 2);
    Complex c2 = ++c1; // c2=c1.operator++() //成员函数形式
    c1.print (); // 2+3i
    c2.print (); // 2+3i
    (++c1) = Complex (10, 20);
    c1.print (); // 10+20i;
    (++++++c1).print (); // 13+23i
```

```
    c2 = --c1; // c2=::operator--(c1)  //全局函数形式
    c1.print (); // 12+22i
    c2.print (); // 12+22i
    return 0;
}
```

3.后++/后--

操作数变。
表达式的值是运算以前的值。
表达式的值是右值。

例子：

```
#include <iostream>
using namespace std;
class Complex {
public:
    Complex (int r = 0, int i = 0) :
        m_r (r), m_i (i) {}
    void print (void) const {
        cout << m_r << '+' << m_i << 'i' << endl;
    }
    const Complex operator++ (int) {   //返回的是一个右值，不变，所以最前面家一个const。int 是哑元，占个位置，没有实际意义，与有形参的进行匹配。
        Complex old (*this);
        ++m_r;
        ++m_i;
        return old;
    }
    friend const Complex operator--(Complex& o,            int) {   //操作数会被修改，所以用引用，不加 const。Int 是占位置的，没有实际意义

        Complex old (o);
        --o.m_r;
        --o.m_i;
        return old;
    }
private:
    int m_r;
    int m_i;
};
int main (void) {
    Complex c1 (1, 2);
    Complex c2 = c1++; // c2=c1.operator++(0) //成员函数形式
```

```
    c1.print (); // 2+3i;
    c2.print (); // 1+2i;
//  (c1++) = c2;
//  c1++++++;
    c2 = c1--; // c2=::operator--(c1,0) //全局函数形式
    c1.print (); // 1+2i //c1 进行了--，得到新的值。
    c2.print (); // 2+3i //c2 得到表达式之前的值。
    return 0;
}
```

四、其它操作符

1.下标操作符：[]

```
int arr[10] = { ... };
arr[1] = 10;
cout << arr[1] << endl;
-----------------------
class Array { ... };
Array arr (...);
arr[1] = 10;
cout << arr[1] << endl;
```

双目操作符，左操作数是一个具有容器特性的对象，右操作数是容器中特定数据元素的索引(基零的下标)。

下标表达式的值可以是左值，也可以是右值，由容器对象的常属性决定。常容器下标表达式的值是右值，非常容器下标表达式的值是左值。

例子：

```
#include <iostream>
using namespace std;
class Array {
public:
    Array (size_t size = 1) :   //构造函数
        m_data (new int[size]) {}

  ~Array (void) {           //析构函数
        if (m_data) {
            delete m_data;
            m_data = NULL;
        }
    }
    int& operator[] (size_t i) {  //不带有常属性的容器对象，没有常属性，所以最前面不加 const，返回的是一个引用，可以修改，所以{前不加 const
        return m_data[i];   //返回类型为该数组中第 i 个元素的引用
```

```
    }
    const int& operator[] (size_t i) const {  //带有常属性的容器对象。是个常量，所以最前面加 const，因为返回的是一个不可修改的右值，所以加了第二个 const。
        return const_cast<Array&> (*this)[i];  //去常，转换成不具有常属性，可以调用上面那个重载。
    }
private:
    int* m_data;
};
int main (void) {
    Array arr (10);//不具有常属性
    for (size_t i = 0; i < 10; ++i)
        arr[i] = i; // arr.operator[](i) = i;
    arr[0]++;//没有常属性，可以修改
    const Array& cr = arr;//具有常属性
    for (size_t i = 0; i < 10; ++i)
        cout << cr[i] << ' ';
    cout << endl;
//  cr[0]++;  //具有常属性，不可修改，所以报错。
    return 0;
}
```

2.函数操作符：()

如果为一个类定义了形如：
返回类型 operator() (形参表) {...}
的操作符函数，那么这个类所实例化的对象就可以被当做函数使用。

例子：

```
include <iostream>
using namespace std;
class Square {
public:
    double operator() (double x) {
        return x * x;
    }
};
class Integer {
public:
    explicit Integer (int i = 0) : m_i (i) {} //explicit 强制要求用这个构造函数进行类型转换时必须要用显式类型转换

        void print (void) const {
        cout << m_i << endl;
```

```
    }
    Integer& operator() (int i) {
        m_i += i;
        return *this;//返回自引
    }

    Integer& operator, (int i) {
        m_i += i;
        return *this;
    }

    operator int (void) const {    //可以将 Integer 转换为 int
        return m_i;
    }
private:
    int m_i;
};
void foo (const Integer& i) {
    i.print ();
}
Integer bar (void) {
    return Integer (300);//因为返回的是 Integer，所以将 300 转换为 Integer 类型的
}
int main (void) {
    Square square;
    cout << square (3) << endl;//这个类的对象可以直接当函数使用。输出为“9”
//  cout << square.operator() (3) << endl;
    Integer i (10);
    i (1) (2) (3) (17);//10+1+2+3+17
    i.print (); // 33
    i, 1, 2, 3, 17;    //33+1+2+3+17
    i.print (); // 56
    i = (Integer)100;//i 是 Integer 类型的，所以要将 100 进行转换
    i.print ();
    foo (static_cast<Integer> (200));//静态类型转换
    bar ().print ();
    int n = i;
    cout << n << endl;
    return 0;
}
```

3.解引用(*)和间接访问(->)操作符

以指针的方式使用类类型的对象。

例子：

```
#include <iostream>
#include <memory>
using namespace std;
class A {
public:
    A (void) {
        cout << "构造" << endl;
    }
    ~A (void) {
        cout << "析构" << endl;
    }
    void hello (void) {
        cout << "Hello, World !" << endl;
    }
};
class PA {
public:
    PA (A* p = NULL) : m_p (p) {}
    ~PA (void) {                    //智能指针，在栈内的 mp 消灭时，同时将它所指向的
堆中的内存释放（delete）
        if (m_p) {
            delete m_p;
            m_p = NULL;
        }
    }

    A& operator* (void) const {    //返回引用
        return *m_p;
    }
    A* operator-> (void) const {   //返回指针，既地址
//      return &**this;       //调用上面的重载函数
        return m_p;           //返回的只是一个地址
    }

    PA (PA& that) : m_p (that.release ()) {}//拷贝构造函数，将自己指向 NULL，并将自
己指向的内存返回
    PA& operator= (PA& that) {
        if (&that != this)
            reset (that.release ());//reset 的参数为 that 所返回的内存地址
```

```
        return *this;
    }
private:
    A* release (void) {
        A* p = m_p;//先将内存保存下来
        m_p = NULL;//
        return p;
    }
    void reset (A* p) {
        if (p != m_p) {
            delete m_p;
            m_p = p;
        }
    }
    A* m_p;
};
void bar (auto_ptr<A> pa) {
    cout << "bar: ";
    pa -> hello ();
}
void foo (void) {
    PA pa (new A);//PA 保存分配的内存的地址，当在栈中的 PA 销毁时，调用 PA 的析构函数，自动执行其中的代码将其 delete 把内存释放，是智能指针

//  A* pa = new A;
//  pa -> hello ();
//      (*pa).hello ();
//  A* pb = pa;
        pa -> hello (); // pa.operator->()->hello();//pa 返回的只是一个地址，所以用“->”
    (*pa).hello (); // pa.operator*().hello(); //（*pa）返回的是引用，所以可以用“.”访问
    PA pb = pa;//如果没有自己写的拷贝赋值函数，则会出现浅拷贝，出现 double free，因为有两个指针都指向同一个内存地址，所以会执行两次析构函数，会出问题。
       //上面自己写的拷贝构造函数，使得 pa 指向 NULL，并返回自己指向的内存地址
       //然后调用操作符重载函数（“=”的），先看 pa 指向的内存与执行拷贝构造所返回的内存（pb 原先指向的内存）是否相同，不相同，则将 pb 所指向的内存释放，再使 pb 指向 pa 原先指向的内存
    pb -> hello ();
    auto_ptr<A> pa1 (new A);//auto_ptr 只能用于单个变量，不能用于数组，但 smart_ptr 都可以
    pa1 -> hello ();
    auto_ptr<A> pa2 = pa1;
    (*pa2).hello ();
```

```
    bar (pa2);//实参与 bar 函数中的形参类似与（PA pb=pa），执行后，pa2 就指向了
NULL，用不了了，gai 内存的析构有 bar 函数中的形参来实现
    cout << pa2.get () << endl;
}
// smart_ptr
int main (void) {
    foo ();
    return 0;
}
```

4.自定义类型转换和类型转换操作符

1)通过单参构造实现自定义类型转换
如果 A 类中有一个可以接受 B 类对象做为唯一参数的构造函数，那么 B 类型的对象就可以根据该构造函数被转换为 A 类型。
通过 explicit 关键字，可以强制使用该构造函数所完成的类型转换必须显示进行。
2)通过类型转换操作符函数实现自定义类型转换
如果 A 类中有一个形如
operator B (void) const { ... }
的操作符函数，那么 A 类型的对象就可以根据该函数被转换为 B 类型。
3)如果目标类型是类类型，源类型是基本类型，那么就只能通过在目标类型中定义以源类型为单参的构造函数实现类型转换。
　如果目标类型是基本类型，源类型是类类型，那么就只能通过在源类型中定义以目标类型为函数名的类型转换操作符函数实现类型转换。
　如果目标类型和源类型都是类类型，那么以上两种方法任取其一，但是不能同时使用。
　如果目标类型和源类型都是基本类型，那么无法实现自定义类型转换。

例子：

```
#include <iostream>
using namespace std;
class Square {
public:
    double operator() (double x) {
        return x * x;
    }
};
class Integer {
public:
    explicit Integer (int i = 0) : m_i (i) {} //explicit 强制要求用这个构造函数进行
类型转换时必须要用显式类型转换
```

```
    void print (void) const {
        cout << m_i << endl;
    }
    Integer& operator() (int i) {
        m_i += i;
        return *this;//返回自引
    }

    Integer& operator, (int i) {
        m_i += i;
        return *this;
    }

    operator int (void) const {    //可以将 Integer 转换为 int
        return m_i;
    }
private:
    int m_i;
};
void foo (const Integer& i) {
    i.print ();
}
Integer bar (void) {
    return Integer (300);//因为返回的是 Integer，所以将 300 转换为 Integer 类型的
}
int main (void) {
    Square square;
    cout << square (3) << endl;//这个类的对象可以直接当函数使用。输出为“9”
//  cout << square.operator() (3) << endl;
    Integer i (10);
    i (1) (2) (3) (17);//10+1+2+3+17
    i.print (); // 33
    i, 1, 2, 3, 17;    //33+1+2+3+17
    i.print (); // 56
    i = (Integer)100;//i 是 Integer 类型的，所以要将 100 进行转换
    i.print ();
    foo (static_cast<Integer> (200));//静态类型转换
    bar ().print ();
    int n = i;
    cout << n << endl;
    return 0;
}
```

5.new/delete 操作符

例子：

```
#include <iostream>
#include <cstdlib>
using namespace std;
class A {
public:
    ~A (void) {}//析构函数

    static void* operator new (size_t size) {    // size_t 等于 unsigned int，但是
一般要用 size_t。
        void* p = malloc (size);  //这是调用构造函数，创建对象
        cout << "我的 new   ：" << p << ' '
            << size << endl;
        return p;                 //完成内存分配，返回分配好的内存地址
    }
    static void operator delete (void* p) {
        cout << "我的 delete：" << p << endl;
        free (p);                 //完成内存释放
    }
    static void* operator new[] (size_t size) {
        void* p = malloc (size);  //这时调用构造函数，创建对象
        cout << "我的 new[]   ：" << p << ' '<< size << endl;
     //打印出分配的内存地址
        return p;                 //完成内存分配，返回分配好的内存地址
    }
    static void operator delete[] (void* p) {
        cout << "我的 delete[]：" << p << endl;
        free (p);    //完成内存释放
    }
private:
    int m_i;
    double m_d;
    char m_c;
};  // IIIIDDDDDDDDCXXX  sizeof(A)为 16
int main (void) {
    cout << sizeof (A) << endl;//16
    A* pa = new A;      //我的 new：0x85bd008  16
    cout << pa << endl;//0x86bd008
    delete pa;          //我的 delete：0x85bd008
    pa = new A[2];      //我的 new[]:0x9fc2008  32
    cout << pa << endl;//0x8fc2008
```

```
    delete[] pa;        //我的 delete[]:0x9fc2008
   //delete pa         //会崩溃，错误，因为 delete 的地址与 new[]的地址不相同，会偏移四个字节，造成局部释放
        return 0;
}
```

五、关于操作符重载的限制

1.至少有一个操作数是类类型的。

```
int a = 10, b = 20;
int c = a + b; // 200
int operator+ (int a, int b) {
  return a * b;
} // ERROR !
```

2.不是所有的操作符都能重载。

```
:: - 作用域限定
.  - 直接成员访问
.* - 直接成员指针解引用
?: - 三目运算符
sizeof - 获取字节数
typeid - 获取类型信息
```

3.不是所有的操作符都可以用全局函数的方式实现。

```
=  - 拷贝赋值
[] - 下标
() - 函数
-> - 间接成员访问
```

4.不能发明新的操作符，不能改变操作数的个数

```
x ** y; // x^y
#   @
```

第四课　继承与多态

一、继承的基本概念

人类：姓名、年龄、吃饭
学生是人：学号、学习
教师是人：工资、讲课

```
    人类        - 基类，共性
   /    \         派生 V^继承
学生      教师 - 子类，个性
```

二、继承的语法

```
class 子类名 : 继承方式 1 基类 1, 继承方式 2 基类 2, ... {
  ...
};
```

继承方式：
公有继承 - public - 最常用方式
私有继承 - private - 缺省方式
保护继承 - protected - 特殊的私有继承

三、公有继承

1.通过继承，在基类中定义的任何成员，也都成为了子类的成员，但是基类的私有成员，子类虽然拥有却不能直接访问。
2.基类中的保护成员，可以被子类直接访问，但不能在无关的类和全局域中被访问。
3.任何一个子类对象中都包含着它的基类子对象。如果在子类的构造函数中没有明确指明其基类子对象如何被构造，系统将采用无参的方式构造该子对象。如果在初始化表中指明了基类子对象的构造方式，就调用相应的构造函数构造该子对象。
4.子类对象的构造和析构顺序
按照继承表的顺序依次构造每个基类子对象->按照声明的顺序依次构造每个成员变量->执行子类构造函数体中的代码
析构的过程与构造严格相反
5.一个子类对象在任何都可以被视为它的基类对象——IsA。
任何时候，一个子类对象的指针或者引用，都可以被隐式地转换为它的基类类型的指针或者引用，但是反过来，将基类类型的指针或者引用转换为它的子类类型必须显示地(static_cast)完成。

```
Student s (...);
Human* h = &s; // OK !
```

6.在子类中定义的任何和基类成员同名的标识符，都可以将基类中的该成员隐藏起来。通过作用域限定操作符“::”，可对该成员解隐藏。

四、继承方式对访控属性的影响

```
class A {
X:
   void foo (void) { ... }
};
class B : Y A {
  void bar (void) {
    foo (); // 仅需考虑 X
  }
};
int main (void) {
  B b (...);
  b.foo(); // 不仅要考虑 X 还要考虑 Y
}
class C : Z B {
  void fun (void) {
    foo(); // 不仅要考虑 X 还要考虑 Y
  }
};
```

当通过一个子类对象(B)访问它从基类(A)中继承过来的成员(foo)的时候，需要考虑子类(B)从基类(A)继承时所采用的继承方式。

1.访控属性

关键字	属性	基类	子类	外部	友员
public	公有	OK	OK	OK	OK
protected	保护	OK	OK	NO	OK
private	私有	OK	NO	NO	OK

2.基类的成员被继承到子类中以后，其访控属性会因不同的继承方式而异。

基类	公有继承	保护继承	私有继承
公有	公有	保护	私有
保护	保护	保护	私有
私有	私有	私有	私有

五、子类的构造函数和析构函数

1.子类隐式地调用基类的构造函数

在子类的构造函数中没有显示地指明其基类部分如何构造，隐式地调用基类的无参构造函数。如果子类没有定义任何构造函数，其缺省无参构造函数同样会隐式地调用基类的无参构造函数。

2.子类显式地调用基类的构造函数

在子类构造函数的初始化表中指明其基类部分的构造方式。

```
class A {
public:
  A (void) : m_data (0) {}
  A (int data) : m_data (data) {}
private:
  int m_data;
};
class B : public A {
public:
  B (int data) : A (data) {}
};
class A { ... };
class B : public A { ... };
class C : public B { ... };
C c (...);
```

构造：A->B->C

析构：C->B->A**3.继承链的构造和初始化顺序**

任何时候子类中基类子对象的构造都要先于子类构造函数中的代码。

4.delete 一个指向**子类对象**的基类指针，实际被执行的**基类的析构函数**，基类的析构函数不会调用子类析构函数，因此子类所特有的资源将形成内存泄漏。

```
Human* p = new Student (...);
delete p; // ->Human::~Human()
delete static_cast<Student*> (p);
```

六、子类的拷贝构造和拷贝赋值

子类的缺省拷贝构造和拷贝赋值除了复制子类的特有部分以外，还会复制其基类部分。如果需要自己定义子类的拷贝构造和拷贝赋值，一定不要忘记在复制子类特有部分的同时，也要复制其基类部分，否则将无法得到完整意义上的对象副本。

七、私有继承和保护继承

用于防止或者限制基类中的公有接口被从子类中扩散。

```
class DCT {
public:
  void codec (void) { ... }
};
class Jpeg : protected DCT { //只有自己类内可以用

public:
  void render (void) {
    codec (...);
  }
};
Jpeg jpeg;
jpeg.codec (...); // ERROR !防止公有接口被从子类中扩散

Jpeg Has A DCT，实现继承

class Jpeg2000 : public Jpeg {
public:
  void render (void) {
    codec (...); // OK ，code 在 Jpeg 类中是保护型的，而通过公有继承，可以访问
  }
};
```

例子：

```
#include <iostream>
using namespace std;
class Human {
public:
    Human (const string& name, int age) :
        m_name (name), m_age (age) {} //构造函数
    void who (void) const {
        cout << m_name << ", " << m_age << endl;
    }
    void eat (const string& food) const {
        cout << "我在吃" << food << endl;
```

```
    }
protected:
    string m_name;
    int m_age;
};


class Student : public Human {
public:
    Student (const string& name, int age, int no) :
        Human (name, age), m_no (no) {} //正确的构造函数创建 Student 类的对象的同时先创建基类 Huamn 类，所以会调用 Human 类的构造函数，这里指定 Human 所调用的构造函数与所写的构造函数相匹配，所以不会出错。
    /*
  Student (const string& name, int age, int no) :
        m_no (no) {
      m_name=name;
      m_age=age;
  }
  */ 错误的构造函数会报错，原因是这里调用 Human 的构造函数时没有指定方式，默认使用无参构造，但是在基类 Human 类中没有无参构造，所以会报错



    Student (const Student& that) :
        Human (that), m_no (that.m_no) {} //拷贝构造，显式的指明了调用基类的拷贝构造函数

Student& operator= (const Student& that) { //操作符重载
        if (&that != this) {
            Human::operator= (that);//显式的调用基类的拷贝赋值
            m_no = that.m_no;
        }
        return *this;
    }
    void learn (const string& lesson) const {
        cout << "我(" << m_name << ", " << m_age
            << ", " << m_no << ")在学" << lesson
            << endl;
    }
    using Human::eat;//如果没有这句，则 Human 的 eat 与这里的 eat 作用域不再一起，则下面的 eat 不会与 Human 中的 eat 构成重载，而是构成隐藏关系
                     //但是有了这句之后，将 Human 的 eat 在这里可见，既作用域也被声明在这里，则两个 eat 构成了重载关系
```

```
    void eat (void) const {
        cout << "我绝食！" << endl;
    }
//  int eat;
private:
    int m_no;
};
int main (void) {
    Student s1 ("张飞", 25, 1001);
    s1.who ();
    s1.eat ("包子");
    s1.learn ("C++");
    Human* h1 = &s1;//子类的指针可以隐式转换为基类的指针，因为访问范围缩小了，是安全的
    h1 -> who ();
    h1 -> eat ("KFC");
//  h1 -> learn ("C");//基类的指针或对象不可以访问子类中的成员
  Student* ps = static_cast<Student*> (h1);//基类的指针不可以隐式的转换为子类的指针，因为访问范围扩大，不安全，所以必须显式的进行转换，但是这样有风险
    ps -> learn ("C");
    Student s2 = s1;
    s2.who (); //子类的指针或对象可以方位基类的成员
    s2.learn ("英语");
    Student s3 ("赵云", 20, 1002);
    s3 = s2;
    s3.who ();
    s3.learn ("数学");
    return 0;
}
```

七、私有继承和保护继承

用于防止或者限制基类中的公有接口被从子类中扩散。

```
class DCT {
public:
  void codec (void) { ... }
};
class Jpeg : protected DCT {
public:
  void render (void) {
    codec (...);
  }
};
Jpeg jpeg;
jpeg.codec (...); // ERROR !
//Jpeg Has A DCT，实现继承
class Jpeg2000 : public Jpeg {
public:
  void render (void) {
    codec (...); // OK !
  }
};
```

八、多重继承

从多于一个基类中派生子类。

电话 媒体播放器 计算机

\ | /

智能手机

1. 多重继承的语法和语义与单继承并没有本质的区别，只是子类对象中包含了更多的基类子对象。它们在内存中按照继承表的先后顺序从低地址到高地址依次排列。

2. 子类对象的指针可以被隐式地转换为任何一个基类类型的指针。无论是隐式转换，还是静态转换，编译器都能保证特定类型的基类指针指向相应类型基类子对象。但是重解释类型转换，无法保证这一点。

3. 尽量防止名字冲突。

例子：智能手机

```
#include <iostream>
using namespace std;
class Phone {                    //基类 1
public:
    Phone (const string& numb) : m_numb (numb) {}
    void call (const string& numb) {
        cout << m_numb << "致电" << numb << endl;
    }
    void foo (void) {
        cout << "Phone::foo" << endl;
    }
private:
    string m_numb;
};

class Player {                   //基类 2
public:
    Player (const string& media) : m_media (media){}
    void play (const string& clip) {
        cout << m_media << "播放器播放" << clip
            << endl;
    }
    void foo (int data) {
        cout << "Player::foo" << endl;
    }
private:
    string m_media;
};
```

```
class Computer {          //基类 3
public:
    Computer (const string& os) : m_os (os) {}
    void run (const string& prog) {
        cout << "在" << m_os << "上运行" << prog
            << endl;
    }
private:
    string m_os;
};

class SmartPhone : public Phone, public Player,
    public Computer {          //多重继承
   public:
    SmartPhone (const string& numb,
        const string& media, const string& os) :
        Phone (numb), Player (media),
        Computer (os) {}
    using Phone::foo;  //将 Phone 中的 foo 函数的作用域声明到这里
    using Player::foo; //将 Player 中的 foo 函数的作用域声明到这里
   //这样就构成了重载
};
int main (void) {
    SmartPhone sp ("13910110072", "MP3", "Android");
    sp.call ("01062332018");
    sp.play ("High 歌");
    sp.run ("愤怒的小鸟");
    Phone* p1 = reinterpret_cast<Phone*> (&sp);
    Player* p2 = reinterpret_cast<Player*> (&sp);
    Computer* p3 = reinterpret_cast<Computer*>(&sp);
    cout << &sp << ' '<< p1 << ' ' << p2 << ' '
        << p3 << endl;  //地址都相同，但如果不用 reinterpret 的话，用隐式或者静态转
换，p1 p2 p3 将 sp 地址段一分为三，所以 p1 p2 p3 地址会不同
    sp.foo ();
    sp.foo (100);
    return 0;
}
```

4.钻石继承和虚继承

1)钻石继承

```
  A
 / \
B   C
 \ /
  D
```

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

在最终子类(D)对象中存在公共基类(A)子对象的多份实例，因此沿着不同的继承路径访问公共基类子对象中的成员，会发生数据不一致的问题。

2)虚继承

在继承表中通过 virtual 关键字指定从公共基类中虚继承，这样就可以保证在最终子类对象中，仅存在一份公共基类子对象的实例，避免沿着不同的继承路径访问公共基类子对象中的成员时，所引发的数据不一致的问题。

只有当所创建对象的类型回溯(su)中存在钻石结构时，虚继承才起作用，否则编译器会直接忽略 virtual 关键字。

例子：钻石继承和虚继承

```
#include <iostream>
using namespace std;
class A {    //公共基类
public:
    A (int i) : m_i (i) {}
protected:
    int m_i;
};
class B : virtual public A {
public:
    B (int i) : A (i) {}
    void set (int i) {
        m_i = i;
    }
};
class C : virtual public A { //virtual 是虚继承
public:
    C (int i) : A (i) {}
    int get (void) {
        return m_i;
    }
};
```

```
class D : public B, public C {
public:
    D (int i) : B (i), C (i), A (i) {}//真正起作用的是A(i),不用B(i),C(i),但要写
};
int main (void) {
    D d (1000);//B里的m_i与C里的m_i都存的是2000
    cout << d.get () << endl; // 1000，调用C中的get，返回C中的m_i的值
    d.set (2000);//调用B类中的set，给B中的m_i赋值
    cout << d.get () << endl; // 输出为2000，---如果B，C没有virtual调用C中的
get，D的初始化表中没有A(i),返回C中的m_i的值,则会输出1000.----因为有了，所以制定
从公共基类中虚继承，所以在最终子对象中只有一份公共基类子对象的实例
//B b(3000);  //B创建的对象没有钻石结构，所以写了virtual也不起作用，依然拥有A的
基类子对象
    return 0;
}
```

九、虚函数与多态

如果将基类中的一个成员函数声明为虚函数，那么子类中的同型函数就也成为虚函数，并且对基类版本形成覆盖。这时，通过一个指向子类对象的基类指针，或者一个引用子类对象的基类引用，调用该虚函数时，实际被调用的函数不由该指针或引用的类型决定，而由它们的目标对象决定，最终导致子类中覆盖版本被执行。这种现象称为多态。

图形：位置，绘制

**/ **

矩形：宽和高 圆：半径

绘制 绘制

例子：图形绘制

```
#include <iostream>
using namespace std;
class Shape {
public:
    Shape (int x, int y) : m_x (x), m_y (y) {}
    virtual void draw (void) {
        cout << "形状(" << m_x << ',' << m_y << ')'
            << endl;
    }
protected:
    int m_x, m_y;
};
```

```
class Rect : public Shape {    //矩形
public:
    Rect (int x, int y, int w, int h) :
        Shape (x, y), m_w (w), m_h (h) {}
    void draw (void) {    //隐藏 shape 中的 draw，构成隐藏关系
        cout << "矩形(" << m_x << ',' << m_y << ','
            << m_w << ',' << m_h << ')' << endl;
    }
private:
    int m_w, m_h;
};
class Circle : public Shape {  //圆形
public:
    Circle (int x, int y, int r) :
        Shape (x, y), m_r (r) {}
    void draw (void) {
        cout << "圆形(" << m_x << ',' << m_y << ','
            << m_r << ')' << endl;
    }
private:
    int m_r;
};
void render (Shape* shapes[]) {
    for (size_t i = 0; shapes[i]; ++i)//挨个解析
        shapes[i]->draw ();}
```

//因为在 shape 中的 draw()有 virtual 修饰为虚函数，而另外两个子类中的同名 draw 也变为虚函数，覆盖了基类 shape 中的 draw

且调用时，有指针的指向的目标类型决定执行哪一个函数，真正执行的是覆盖版本的 draw

所以就通过指针调用各自的 draw()，这样就可以用各自的绘制方法画出图形；（有了 virtual 修饰，则是按照指针指向的对象来找 draw）

但是如果没有在基类 shape 中的 draw()没有用 virtual 修饰，则 shape 类型的指针会访问 shape 类中的 draw，则全部会用基类 shape 中的 draw 绘制图形（是根据指针类型来找 draw）

```
int main (void) {
    Shape* shapes[1024] = {};  //定义的基类类型的指针数组，这样就可以指向不同子类类型的对象
    shapes[0] = new Rect (1, 2, 3, 4);
    shapes[1] = new Circle (5, 6, 7);
    shapes[2] = new Circle (8, 9, 10);
    shapes[3] = new Rect (11, 12, 13, 14);
    shapes[4] = new Rect (15, 16, 17, 18);
    render (shapes);
    return 0;
}
```

十、函数覆盖的条件

overload - 重载
override - 覆盖、重写、改写
1.基类版本必须是虚函数。
2.函数名、形参表和常属性必须严格一致。
3.如果返回基本类型或者对象，那么也必须严格一致。如果返回类类型的指针或引用，那么子类版本也可以返回基类版本的子类。

```
class B : public A { ... };
基类：virtual A* foo (void) {...}
子类：A* foo (void) { ... }
      B* foo (void) { ... }
```

4.子类的覆盖版本不能比基类版本声明更多的异常抛出。
5.子类覆盖版本的访控属性与基类无关。

```
class A {
public:
  virtual void foo (void) { ... }
};
class B : public A {
private:
  void foo (void) { ... }
};
int main (void) {
  B* b = new B;
  b->foo (); // ERROR !foo 在 B 中是私有的
  A* a = new B;
  a->foo (); // OK ! -> B::foo  访控属性是看指针类型的，在 A 中，foo 是公共部分的，所以可以访问，但真正执行的是覆盖版本的 B 中的 foo。
}
```

十一、多态=虚函数+指针/引用

```
Rect rect (...);
Shape shape = rect;//shape 只能代表 shape 代表不了 rect。
shape->draw ();     // Shape::draw
Shape& shape = rect;
shape->draw ();     // Rect::draw
```

```
class A {
public:
  A (void) {
    bar (); // A::bar    //构造函数中调用虚函数，永远没有多态型，构造 A 的时候，B 还没有构造好，没法调用 B 中的尚未构造好的覆盖版本。
  }
  ~A (void) {
    bar (); // A::bar  //析构函数中调用虚函数，永远没有多态性，因为析构的顺序和构造相反，当执行基类中的析构函数时，子类已经析构后释放完了，无法调用析构后的覆盖版本
  }
  void foo (void) {
    This->bar (); // B::bar
  }
  virtual void bar (void) {
    cout << 'A' << endl;
  }
};
class B : public A {
  void bar (void) {
    cout << 'B' << endl;
  }
};
int main (void) {
  B b; // A
  b.foo (); // B    因为 foo 函数是 A 类中的成员函数，所以 this 指针是 A 类型的，这个 this 指针指向 B 类型的对象 b，调用那个虚函数的覆盖版本看指针指向的目标对象，所以调用 B 中的 bar
  return 0;
}
```

十二、纯虚函数、抽象类、纯抽象类

形如:
virtrual 返回类型 成员函数名 (形参表) = 0;
的虚函数被称为纯虚函数。
一个包含了纯虚函数类称为抽象类，抽象类不能实例化为对象。
如果一个类继承自抽象类，但是并没有为其抽象基类中的全部纯虚函数提供覆盖，那么该子类就也是一个抽象类。

```
class A { // 纯抽象类
  virtual void foo (void) = 0;
  virtual void bar (void) = 0;
  virtual void fun (void) = 0;
};
class B : public A { // 抽象类
  void foo (void) { ... }
};
class C : public B { // 抽象类
  void bar (void) { ... }
};
class D : public C { // 具体类
  void fun (void) { ... }
};
```

除了构造和析构函数以外，所有的成员函数都是纯虚函数的类称为纯抽象类。

例子：

```
#include <iostream>
using namespace std;
class Shape {
public:
    Shape (int x, int y) : m_x (x), m_y (y) {}
    virtual void draw (void) = 0;  //空函数，纯虚函数，因为有了这个纯虚函数，所以 shape 为抽象类。所以 shape 不能实例化，不能创建对象，如果该类中除了构造和析构函数之外，都是纯虚函数，则该类为纯抽象类，同样不能实例化。
protected:
    int m_x, m_y;
};
class Rect : public Shape {
public:
    Rect (int x, int y, int w, int h) :
        Shape (x, y), m_w (w), m_h (h) {}
    void draw (void) {
        cout << "矩形(" << m_x << ',' << m_y << ','
            << m_w << ',' << m_h << ')' << endl;
    }
```

```
//      int draw (void){}          //不构成任何合法关系
//  int draw (void) const {}  //会隐藏，因为形参不同（这里的 this 是 const 类型）
//  int draw (int){}           //隐藏
private:
    int m_w, m_h;
};
class Circle : public Shape {
public:
    Circle (int x, int y, int r) :
        Shape (x, y), m_r (r) {}
    void draw (void) {
        cout << "圆形(" << m_x << ',' << m_y << ','
            << m_r << ')' << endl;
    }
private:
    int m_r;
};
void render (Shape* shapes[]) {
    for (size_t i = 0; shapes[i]; ++i)
        shapes[i]->draw ();
}
int main (void) {
    Shape* shapes[1024] = {};
    shapes[0] = new Rect (1, 2, 3, 4);
    shapes[1] = new Circle (5, 6, 7);
    shapes[2] = new Circle (8, 9, 10);
    shapes[3] = new Rect (11, 12, 13, 14);
    shapes[4] = new Rect (15, 16, 17, 18);
    render (shapes);
//  Shape shape (1, 2);
    return 0;
}
```

十三、动态绑定（后期绑定、运行时绑定）

1.虚函数表

```
class A {
public:
  virtual void foo (void) { ... }
  virtual void bar (void) { ... }
};
class B : public A {
public:
  void foo (void) { ... }
};
A* pa = new A;
pa->foo (); // A::foo
pa->bar (); // A::bar
---------------------
A* pa = new B;
pa->foo (); // B::foo
pa->bar (); // A::bar
```

2.动态绑定

当编译器看到通过指向子类对象的基类指针或者引用子类对象的基类引用，调用基类中的虚函数时，并不急于生成函数调用代码，相反会在该函数调用出生成若干条指令，这些指令在程序的运行阶段被执行，完成如下动作：

1)根据指针或引用的目标对象找到相应虚函数表的指针；

2)根据虚函数表指针，找到虚函数的地址；

3)根据虚函数地址，指向虚函数代码。

由此可见，对虚函数的调用，只有运行阶段才能够确定，故谓之后期绑定或运行时绑定。

3.动态绑定对程序的性能会造成不利影响。如果不需要实现多态就不要使用虚函数。

例子：

```
#include <iostream>
using namespace std;
class A {
public:
    virtual void foo (void) {
        cout << "A::foo()" << endl;
    }
    virtual void bar (void) {
        cout << "A::bar()" << endl;
    }
};
class B : public A {
```

```
public:
    void foo (void) {            //覆盖
        cout << "B::foo()" << endl;
    }
};
int main (void) {
    A a;   //a
    void (**vft) (void) = *(void (***) (void))&a;  //使 vft(二级指针)指向 a 的虚函数表，因为 a 是 A 类型的，所以强制转换为 void (***)，虚函数表是一个函数指针数组，要指向他，应该用指向指针的指针--二级指针。
    cout << (void*)vft[0] << ' '
        << (void*)vft[1] << endl; //A 类中 foo 函数与 bar 函数的地址
    vft[0] ();//调用了 A 类的 foo
    vft[1] ();//调用了 A 类的 bar

    B b;
    vft = *(void (***) (void))&b;//使 vft 指向 B 类的虚函数表
    cout << (void*)vft[0] << ' '
        << (void*)vft[1] << endl;//B 类中的 foo 函数与 A 类中的 bar 函数地址
    vft[0] ();//调用了 B 类的 foo 函数
    vft[1] ();//调用了 A 类的 bar 函数
    return 0;
}
```

十四、运行时类型信息（RTTI）

1.typeid 操作符

A a;
typeid (a)返回 typeinfo 类型的对象的常引用。
typeinfo::name() - 以字符串的形式返回类型名称。
typeinfo::operator==() -类型一致
typeinfo::operator!=() -类型不一致
#include <typeinfo>

例子：

```
#include <iostream>
#include <typeinfo>
#include <cstring>
using namespace std;
class A {
public:
    virtual void foo (void) {}
```

```
};
class B : public A {};
void print (A* pa) {
    //  if (! strcmp (typeid (*pa).name (), "1A"))
    if (typeid (*pa) == typeid (A))
        cout << "pa 指向 A 对象！" << endl;
    else
    //  if (! strcmp (typeid (*pa).name (), "1B"))
    if (typeid (*pa) == typeid (B))
        cout << "pa 指向 B 对象！" << endl;
}
int main (void) {
    cout << typeid (int).name () << endl;           //'i'
    cout << typeid (unsigned int).name () << endl;  //'j'
    cout << typeid (double[10]).name () << endl;    //A10_d
    cout << typeid (char[3][4][5]).name () << endl; //A3_A4_A5_c
    char* (*p[5]) (int*, short*);                   //函数指针数组
    cout << typeid (p).name () << endl;             //A5_PFPcPiPsE
    cout << typeid (const char* const* const).name (//
        ) << endl;     //PKPKc  指针指向一个常量，这个常量是个指针，这个指针指向一个常量，这个常量是 char 类型的。
    cout << typeid (A).name () << endl;//1A
    A* pa = new B;
    cout << typeid (*pa).name () << endl;//A 中有虚函数，所以'1B'，如果 A 中没有虚函数，则是'1A'。没有多态，则会按照指针本身的类型，有多态则会按照指针指向的目标对象类型。
    print (new A);//"pa 指向 A 对象"
    print (new B);//"pa 指向 B 对象"
}
```

2.dynamic_cast

例子：

```
#include <iostream>
//动态类型转换
using namespace std;
class A { virtual void foo (void) {} };
class B : public A {};
class C : public B {};
class D {};
int main (void) {
    B b;
    A* pa = &b;//指向子类对象的基类指针。
```

```
    cout << pa << endl; //地址
    cout << "-------- dc --------" << endl;
                //动态类型转换,运行期间检查。
    // A 是 B 的基类，pa 指向 B 对象，成功
    B* pb = dynamic_cast<B*> (pa);
    cout << pb << endl;     //地址
    // A 不是 C 的基类，pa 没有指向 C 对象，失败，安全
    C* pc = dynamic_cast<C*> (pa);
    cout << pc << endl;     // 0
    A& ra = b;      //引用子类对象的基类引用。
    try {
        C& rc = dynamic_cast<C&> (ra);
    }
    catch (exception& ex) {
        cout << "类型转换失败：" << ex.what ()
            << endl;
        // ...
    }
    // pa 没有指向 D 对象，失败，安全
    D* pd = dynamic_cast<D*> (pa);
    cout << pd << endl;

        cout << "-------- sc --------" << endl;
          //静态类型转换，编译期间检查。
          // B 是 A 的子类，成功
    pb = static_cast<B*> (pa);
    cout << pb << endl;
    // C 是 A 的孙子类，成功，危险！
    pc = static_cast<C*> (pa);
    cout << pc << endl;

     // D 不是 A 的后裔，失败，安全
     //  pd = static_cast<D*> (pa);   //两个方向都不能做隐式转换，所以任何方向也不
能做静态转换。
     //  cout << pd << endl;

        cout << "-------- rc --------" << endl;
              //重解释类型转换
        // 无论在编译期还是在运行期都不做检查，危险！
    pb = reinterpret_cast<B*> (pa);
    cout << pb << endl;
    pc = reinterpret_cast<C*> (pa);
    cout << pc << endl;
    pd = reinterpret_cast<D*> (pa);   cout << pd << endl;   return 0;  }
```

十五、虚析构函数

将基类的析构函数声明为虚函数，delete 一个指向子类对象的基类指针，实际被执行的将是子类的析构函数，而子类的析构函数可以自动调用基类的析构函数，进而保证子类特有的资源，和基类子对象中的资源都能够得到释放，防止内存泄漏。
如果基类中存在虚函数，那么就有必要为其定义一个虚析构函数，即使该函数什么也不做。

思考：

1.虚函数可以内联吗？ **不可以**
2.一个类的构造函数可以被定义为虚函数吗？ **不可以（调用虚函数要用虚函数表，但调用构造函数时还没有创建好对象，没有虚函数表，自相矛盾）**
3.一个类的静态成员函数可以被定义为虚函数吗？ **不可以**
4.一个类的成员函数形式的操作符函数可以被定义为虚函数吗？ **可以**
5.一个全局函数可以被定义为虚函数吗？ **不可以**

```
class PDFParser {
public:
  void parse (const char* file) {
    // 解析出一行文字
    on_text (...);
    // 解析出一幅图片
    on_image (...);
    // 解析出一张图形
    on_graph (...);
  }
private:
  virtual void on_text (...) = 0;
  virtual void on_image (...) = 0;
  virtual void on_graph (...) = 0;
};
class PDFRender : public PDFParser {
private:
  void on_text (...) { ... }
  void on_image (...)  { ... }
  void on_graph (...)  { ... }
};
PDFRender render (...);
render.parse ("test.pdf");
```

模板方法模式

MFC

例子：

```
#include <iostream>
using namespace std;
class A {
public:
    A (void) {
        cout << "A 构造" << endl;
    }
    virtual ~A (void) {      //析构函数只有一个，声明为 virtual 则会形成覆盖，这个叫做虚析构函数
        cout << "A 析构" << endl;
    }
};
class B : public A {
public:
    B (void) {
        cout << "B 构造" << endl;
    }
    ~B (void) {
        cout << "B 析构" << endl;
    }
};
int main (void) {
       B* pb=new B;      //先调用 A 再调用 B 的构造
       delete pb;        //调用 B 的析构，再自动调用 A 的析构

       A* pa = new B;    //指向子类的基类指针，先调用 A 的构造，再调用 B 的构造
       delete pa;        //因为基类中的析构函数是虚函数，调用 B 的析构，B 的析构自动调用 A 的构析。如果基类中的析构函数没有被声明为虚函数，则会调用基类的析构，不会调用子类的析构，会出现泄露
    return 0;
}
```

第六课　异常和 I/O 流

一、为什么要有异常——WHY？

1.通过返回值表达错误

像 malloc 会返回 0 或 1.

局部对象都能正确的析构

层层判断返回值，流程繁琐

例子：

```
#include <iostream>
#include <cstdio>
using namespace std;
int func3 (void) {
    FILE* fp = fopen ("none", "r");//fopen 失败会返回控指针 NULL。
    if (! fp)
        return -1;
    // ...
    fclose (fp);
    return 0;
}
int func2 (void) {
    if (func3 () == -1)
        return -1;
    // ...
    return 0;
}
int func1 (void) {
    if (func2 () == -1)
        return -1;
    // ...
    return 0;
}
int main (void) {
    //层层判断返回值
    if (func1 () == -1) {
        cout << "执行失败！改天再见！" << endl;
        return -1;
    }
    // ...
    cout << "执行成功！恭喜恭喜！" << endl;
    return 0;
}
```

2.通过 setjmp/longjmp 远程跳转

一步到位进入错误处理，流程简单
局部对象会失去被析构的机会

例子：

```
#include <iostream>
#include <cstdio>
#include <csetjmp> //标 c 的函数，跳转
using namespace std;
jmp_buf g_env;      //jmp 是专门为 c 量身定造的，有类的情况不适用，会跳转，因为不执行右括号，局部对象失去执行析构的机会,不会调用析构函数，会造成内存泄露
class A {
public:
    A (void) {
        cout << "A 构造" << endl;
    }
    ~A (void) {
        cout << "A 析构" << endl;
    }
};
void func3 (void) {
    A a;
    FILE* fp = fopen ("none", "r");
    if (! fp)
        longjmp (g_env, -1);    //（没有定义类的时候）这个时候是的 g_env 变为-1，但是不在这返回，在 main 函数的 setjmp 处返回
    // ...
    fclose (fp);
}
void func2 (void) {
    A a;
    func3 ();
    // ...
}
void func1 (void) {
    A a;
    func2 ();
    // ...
}
int main (void) {
    if (setjmp (g_env) == -1) {     //（没有定义类的时候）第一次到这，genv 是 0，所以执行下面的 func1()，执行了后在 fun3 中的 longjmp 处在缓冲区使得 g_env 变为 1，并在这使 g_env 返回
```

```
        cout << "执行失败！改天再见！" << endl;
        return -1;
    }
    func1 ();
    // ...
    cout << "执行成功！恭喜恭喜！" << endl;
    return 0;
}
```

3.异常处理

局部对象都能正确的析构

一步到位进入错误处理，流程简单

二、异常的语法——WHAT？

1.异常的抛出

throw 异常对象;

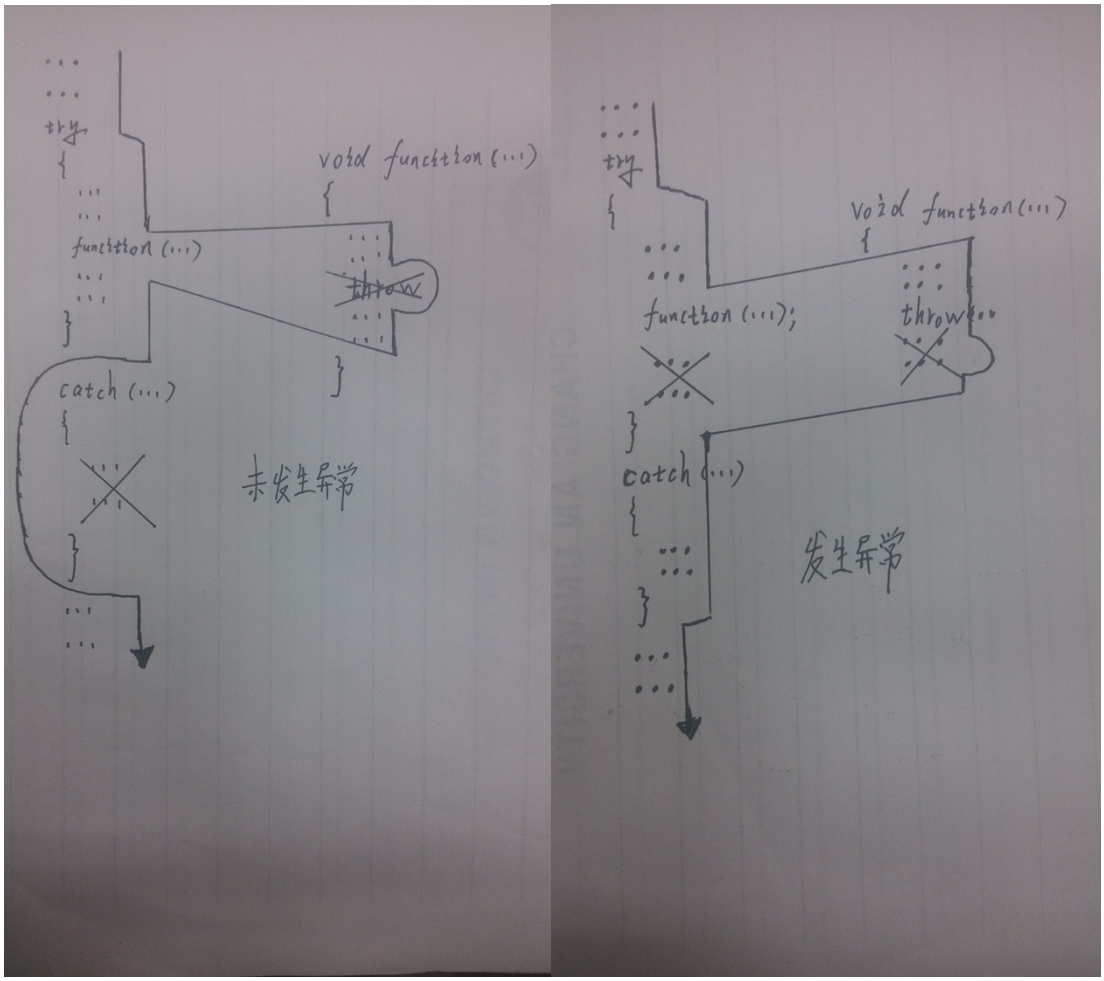
异常对象可以是基本类型的变量，也可以是类类型的对象。

当程序执行错误分支时抛出异常。

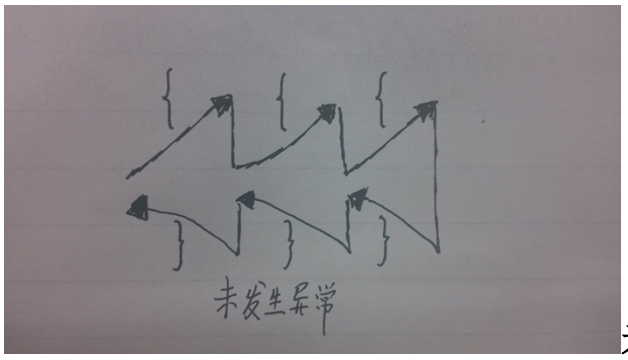
2.异常的捕获

```
try {
  可能抛出异常的语句块;
}
catch (异常类型 1 异常对象 1) {
  处理异常类型 1 的语句块;
}
catch (异常类型 2 异常对象 2) {
  处理异常类型 2 的语句块;
}
...
catch (...) {
  处理其它类型异常的语句块;
}
```

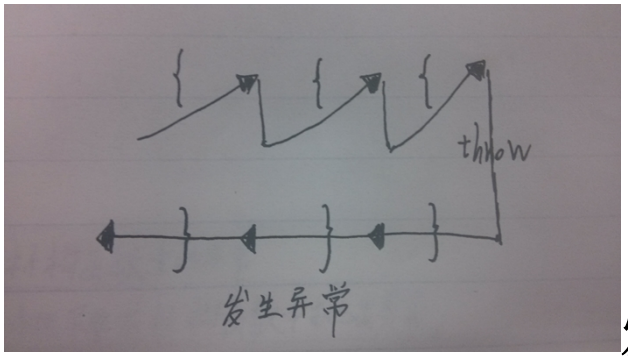
异常处理的流程，始终沿着函数调用的逆序，依次执行右花括号，直到 try 的右花括号，保证所有的局部对象都能被正确地析构，然会根据异常对象的类型，匹配相应的 catch 分支，进行有针对性的错误处理。



---不发生异常与发生异常的示意图---



未发生异常



发生异常

例子：

```
#include <iostream>
#include <cstdio>
using namespace std;
class A {
public:
    A (void) {
        cout << "A 构造" << endl;
    }
    ~A (void) {
        cout << "A 析构" << endl;
    }
};
void func3 (void) {
    A a;
    FILE* fp = fopen ("none", "r");
    if (! fp) {       //如果不发生异常，不执行 throw，直接执行 throw 后面的语句
        cout << "throw 前" << endl;
        throw -1;     //如果有异常，throw 之后的语句不执行，直接右括号
        cout << "throw 后" << endl;
    }
    cout << "文件打开成功！" << endl;
    // ...
    fclose (fp);
}
void func2 (void) {
    A a;
    cout << "func3()前" << endl;
    func3 ();     //如果有异常，则直接右括号
    cout << "func3()后" << endl;
    // ...
}
void func1 (void) {
    A a;
    cout << "func2()前" << endl;
    func2 ();     //有异常，直接右括号
    cout << "func2()后" << endl;
    // ...
}
```

```
int main (void) {
    try {
        cout << "func1()前" << endl;
        func1 ();       //之后进入 func1，先创建 a，执行构造，再进入 func2，又创建 a，执行构造，再进入 func3，又创建 a，执行构造，然后执行 throw，抛出-1；结束 func3，释放 func3 中的 a，调用析构，然后 func2 结束，释放 func2 的 a，调用析构，然后 func1 结束，释放 func1 的 a，调用析构。然后直接到 try 的右花括号，然后执行异常处理，根据异常对象的类型匹配相应的 catch，这里是“执行失败”。
        cout << "func1()后" << endl;
    }
    catch (int ex) {
        if (ex == -1) {
            cout << "执行失败！改天再见！" << endl;
            return -1;
        }
    }
    // ...
    cout << "执行成功！恭喜恭喜！" << endl;
    return 0;
}
```

三、异常处理的使用方法——HOW？

1.抛出基本类型的异常，用不同的值代表不同的错误。

例子：

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
void foo (void) {
    FILE* fp = fopen ("none", "r");
    if (! fp)
        throw "打开文件失败！";
    void* pv = malloc (0xFFFFFFFF);
    if (! pv)
        throw "内存分配失败！";
    // ...
}
int main (void) {
    try {
        foo ();                    //可能引发异常的语句快
    }
    catch (const char* ex) {    //char 类型的异常对象类型
        cout << ex << endl;
        return -1;    //
    }        //有 none 文件，显示内存分配失败，没有 none 文件，显示打开文件失败
    return 0;
}
```

2.抛出类类型的异常，用不同的类型表示不同的错误。

例子：

```
#include <iostream>/
#include <cstdio>
#include <cstdlib>
using namespace std;
class Error {};
class FileError : public Error {};
class MemError : public Error {};
void foo (void) {
    FILE* fp = fopen ("none", "r");
    if (! fp)
        throw FileError ();    //放到安全区
```

```
    void* pv = malloc (0xFFFFFFFF);
    if (! pv)
        throw MemError ();
    // ...
}
int main (void) {
    try {
        foo ();
    }
    catch (FileError& ex) {          //用引用，效率高，避免拷贝构造
        cout << "打开文件失败！" << endl;
        return -1;
    }
    catch (MemError& ex) {//用引用
        cout << "内存分配失败！" << endl;
        return -1;
    }
    catch (Error& ex) {              //如果放在最前面：会有警告，会捕获所有异常，一个子类的对象可以用基类的引用去引用，虽然后面有更适合的匹配，但是最先匹配原则
        cout << "一般性错误！" << endl;
        return -1;
    }
    return 0;
}
```

3.通过类类型的异常携带更多诊断信息。

例子：

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
class Error {
public:
    virtual void print (void) const = 0;
};
class FileError : public Error {
public:
    FileError (const string& file, int line) :
        m_file (file), m_line (line) {}
    void print (void) const {
        cout << "在" << m_file << "文件的第"
            << m_line << "行，发生了文件错误！"
            << endl;
    }
```

```
private:
    string m_file;
    int m_line;
};
class MemError : public Error {
public:
    void print (void) const {
        cout << "内存不够啦！！！" << endl;
    }
};
void foo (void) {
    FILE* fp = fopen ("none", "r");
    if (! fp)
        throw FileError (__FILE__, __LINE__);  //这里抛出，所以不执行下面的语句，
所以下面的异常没有抛出
    void* pv = malloc (0xFFFFFFFF);
    if (! pv)
        throw MemError ();
    // ...
}
int main (void) {
    try {
        foo ();
    }
    catch (Error& ex) {
        ex.print ();
        return -1;
    }
    return 0;
}
```

4.忽略异常和继续抛出异常。

例子：

```
#include <iostream>
//继续抛出  记住用引用
using namespace std;
void foo (void) {
    throw 10;
}
void bar (void) {
    try {
        foo ();
    }
    catch (int& ex) {
        --ex;  //安全区中的 ex 变为 9
        throw; // 继续抛出 抛出的 ex 为 9.
    }
    // ...
}
int main (void) {
    try {
        bar ();
    }
    catch (int& ex) {
        cout << ex << endl; // 9
    }
    return 0;
}
```

5.异常说明

在一个函数的形参表后面写如下语句：

...形参表) throw (异常类型 1, 异常类型 2, ...) { ... }

表示这个函数可以被捕获的异常。

throw () - 这个函数所抛出的任何异常均无法捕获。

没有异常说明 - 这个函数所抛出的任何异常均可捕获。

```
class A {
  virtual void foo (void)
    throw (int, double) { ... }
  virtual void bar (void)
    throw () { ... }
};
```

```
class B : public A {
  void foo (void)
    throw (int, char) { ... }  // ERROR  不能比基类抛出更多的异常

void bar (void) { ... }        // ERROR  不可以这样写覆盖函数
  void bar (void)
    throw () { ... }
};
```

例子：

```
#include <iostream>
using namespace std;
void foo (void) throw (int, double, const char*){ //可以被捕获到的类型
//  throw 1;
//  throw 3.14;
    throw "Hello, Exception !";
}
int main (void) {
    try {
        foo ();
    }
    catch (int ex) {            //可以捕获 int 的异常
        cout << ex << endl;
    }
    catch (double ex) {         //可以捕获 double 的异常
        cout << ex << endl;
    }
    catch (const char* ex) {    //可捕获 const char*的异常
        cout << ex << endl;
 //               cout<<__LINE__<<endl;
    }
    return 0;
}
```

6.使用标准异常

```
#include <stdexcept>
```

四、构造函数中的异常

构造函数可以抛出异常，而且有些时候还必须抛出异常，已通知调用者构造过程中所发生的错误。
如果在一个对象的构造过程中抛出了异常，那么这个对象就称为不完整对象。
不完整对象的析构函数永远不会被指向，因此需要在 throw 之前，手动释放动态的资源。

例子：

```
#include <iostream>
#include <stdexcept>
#include <cstdio>
//构造函数中的异常
using namespace std;
class FileError : public exception {
private:
    const char* what (void) const throw () {  //
        return "文件访问失败！";
    }
};
class B {
public:
    B (void) {
        cout << "B 构造" << endl;
    }
    ~B (void) {
        cout << "B 析构" << endl;
    }
};
class C {
public:
    C (void) {
        cout << "C 构造" << endl;
    }
    ~C (void) {
        cout << "C 析构" << endl;
    }
};
class A : public C {
public:
    A (void) : m_b (new B) {
        FILE* fp = fopen ("none", "r");
```

```
        if (! fp) {
            delete m_b;             //再抛出异常前，将资源释放，因为一旦抛出异常，就
不会执行析构函数。
            throw FileError ();     //会有回滚机制，所有调用的构造函数，会反向执行一
遍析构函数，但除了动态变
        }
        // ...
        fclose (fp);
    }
    ~A (void) {
        delete m_b;
    }
private:
    B* m_b;
//  C m_c;
};
int main (void) {
    try {
        A a; //执行 A 的拷贝构造,
        // ...
    }
    catch (exception& ex) {
        cout << ex.what () << endl;//执行覆盖版本
        return -1;
    }
    return 0;
}
```

五、析构函数中的异常

永远不要在析构函数中抛出异常。

```
class A {
public:
  ~A (void) {
    //throw -1;  //错误，抛出后，直接到析构函数最后的花括号（1），然后直接到 main 函数中的花括号（2），花括号（2）结束，又调用此析构函数，形成死循环
    try {

      sysfunc ();
    }
    catch (...) {}
  } (1)
};
try {
  A a;
  a.foo ();
}   (2)
catch (...) { ... }
```

通过 try-catch 拦截所有可能引发的异常。

六、C++的 I/O 流库

C：fopen/fclose/fread/fwrite/fprintf/fscanf/fseek/ftell...
C++：对基本的 I/O 操作做了类的封装，其功能没有任何差别，用法和 C 的 I/O 流也非常近似。

sscanf

七、格式化 I/O

<< / >>

例子：

```
#include <iostream>
#include <fstream>
//格式化 I/O
using namespace std;
int main (void) {
        //格式化的写
    ofstream ofs ("format.txt");     //相当与 c 中的 w，新加的内容会覆盖原有内容，打开这个文件，打开失败，ofs 为 false，成功为 true
```

```
    if (! ofs) {
        perror ("打开文件失败");
        return -1;
    }
    ofs << 1234 << ' ' << 56.78 << ' ' << "tarena"
        << '\n';      //将这写写入到文件里
    ofs.close ();     //关闭文件，如果不写也可以，结束后，调用 ofstream 析构函数会关闭掉

        ofs.open ("format.txt", ios::app);     //相当于 c 中的以 a 方式打开，可以在文件中追加，不会覆盖原有的内容
    if (! ofs) {
        perror ("打开文件失败");
        return -1;
    }
    ofs << "append_a_line\n";
    ofs.close ();

        //格式化的读
    ifstream ifs ("format.txt");     //要求文件必须存在，否则报错
    if (! ifs) {
        perror ("打开文件失败");
        return -1;
    }
    int i;
    double d;
    string s1, s2;
    ifs >> i >> d >> s1 >> s2;     //读取文件中的内容到程序中。
    cout << i << ' ' << d << ' ' << s1 << ' '
        << s2 << endl;
    ifs.close ();      //关闭文件
    return 0;
}
```

八、非格式化 I/O

put / get
例子：

```
#include <iostream>
#include <fstream>
//非格式化 I/O
using namespace std;
int main (void) {
    ofstream ofs ("putget.txt");          //定义一个 ofstream 类
    if (! ofs) {
        perror ("打开文件失败");
        return -1;
    }
    for (char c = ' '; c <= '~'; ++c)     //前加加返回的是一个引用，效率高，后加加返回的是拷贝，要进行一次拷贝，所以效率低。
        if (! ofs.put (c)) {       //ofs.put 成功返回 true，否则 false。向文件中写入。

            perror ("写入文件失败");
            return -1;
        }
    ofs.close ();

    ifstream ifs ("putget.txt");
    if (! ifs) {
        perror ("打开文件失败");
        return -1;
    }
    char c;
    while ((c = ifs.get ()) != EOF)      //读如字符，直到返回 EOF（表示读取完毕）。
        cout << c;
    cout << endl;
    if (! ifs.eof ()) {        //或者 if(ifs.error())都可以判断是否出错
        perror ("读取文件失败");
        return -1;
    }
    ifs.close ();
    return 0;
}
```

九、随机 I/O

seekp / seekg　　（p->put,g->get）

tellp / tellg

例子：

```
#include <iostream>
#include <fstream>
//随机 I/O
using namespace std;
int main (void) {
    fstream fs ("seek.txt", ios::in | ios::out);      //即可读也可写,相当于 c 中的 r+，要求文件必须存在。
    if (! fs) {
        perror ("打开文件失败");      //打印错误信息。最近的一次错误的原因。
        return -1;
    }
    fs << "0123456789";  //向文件里输入。
    cout << fs.tellp () << endl;    //获取写指针的位置，在下一个接受数据的位置，最后一个 9 所在的位置是 9，所以写的位置为 10
    cout << fs.tellg () << endl;    //获取读指针的位置，虽然没有读，但是会随着写指针一起走

        //seekp()与 seekg()函数分别有两个参数，第一个是偏移量，正负代表前后方向，第二个是从哪个位置开始
    fs.seekp (-3, ios::cur);   //调整写指针的位置，表示从当前位置往文件头移动三个字符
    fs << "XYZ";    //覆盖 789
    fs.seekg (4, ios::beg);    //调整读指针的位置，表示从文件头开始偏移四个位置
    int i;
    fs >> i;      //从 4 开始读，读到 6，后面是 xyz 所以不读，结束。
    cout << i << endl;
    cout << fs.tellg () << endl;   //7
    cout << fs.tellp () << endl;   //7

        fs.seekg (-6, ios::end);     //从文件尾开始向文件头偏移 6 个位置
    fs << "ABC";
    fs.close ();
    return 0;
}
```

十、二进制 I/O

read / write
K 0 - 255
A^K=B
B^K=A
PKI
HAS MD5

例子：加密文件（异或机制）

```
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <cstdlib>
using namespace std;
#define BUFSIZE (1024*10)
int _xor (const char* src, const char* dst,
    unsigned char key) {    //源文件，目标文件，密钥
    ifstream ifs (src, ios::binary);   //以二进制方式读
    if (! ifs) {
        perror ("打开源文件失败");
        return -1;
    }
    ofstream ofs (dst, ios::binary);
    if (! ofs) {
        perror ("打开目标文件失败");
        return -1;
    }

    char* buf = NULL;   //创建缓冲区
    try {
        buf = new char[BUFSIZE];
    }
    catch (bad_alloc& ex) {      //bad_alloc 是标准库里的
        cout << ex.what () << endl;
        return -1;
    }
    while (ifs.read (buf, BUFSIZE)) {    //缓冲区的地址，和大小
        for (size_t i = 0; i < BUFSIZE; ++i)
            buf[i] ^= key;      //将每一个字符都与 key 异或
        if (! ofs.write (buf, BUFSIZE)) {   //以二进制方式写进去，缓冲区地址，和希望写的大小
            perror ("写入文件失败");
            return -1;
```

```
        }
    }
    if (! ifs.eof ()) {   //判断是否正常
        perror ("读取文件失败");
        return -1;
    }

    for (size_t i = 0; i < ifs.gcount (); ++i)   //gcount 函数返回剩下的大小
        buf[i] ^= key;      //将缓冲区中的剩下的也与 key 异或
    if (! ofs.write (buf, ifs.gcount ())) {
        perror ("写入文件失败");
        return -1;
    }
    delete[] buf;  //释放缓冲区
    ofs.close ();  //关闭文件
    ifs.close ();  //关闭文件
    return 0;
}
int enc (const char* plain, const char* cipher) {
    srand (time (NULL));
    unsigned char key = rand () % 256;  //0 到 256sui 随机数
    if (_xor (plain, cipher, key) == -1)  //为-1 则失败
        return -1;
    cout << "密钥：" << (unsigned int)key << endl; //告诉密钥是什么。转换为数的形
式
    return 0;
}
int dec (const char* cipher, const char* plain,
    unsigned char key) {
    return _xor (cipher, plain, key);
}
int main (int argc, char* argv[]) {
    if (argc < 3) {
        cerr << "用法：" << argv[0]
            << " <明文文件> <密文文件>" << endl;
        cerr << "用法：" << argv[0]
            << " <密文文件> <明文文件> <密钥>"
            << endl;
        return -1;
    }
    if (argc < 4)
        return enc (argv[1], argv[2]);
    else
        return dec (argv[1], argv[2],
```

```
        atoi (argv[3]));      return 0;     }
```

十一、格式控制

```
in a;
printf ("%d%x\n", a, a)
cout << a << endl;
```

流函数

流控制符

```
cout << hex << a;
cout << setw (10) << a;
```

例子：

```
#include <iostream>
#include <iomanip>   //要加这个头文件
#include <cmath>     //数学库
#include <fstream>   //文件头文件
#include <sstream>   //字符串流
//格式控制
using namespace std;
int main (void) {
    cout << sqrt (2) << endl;   //求平方根，只输出六位有效数字,1.41421
    cout.precision (10);        //将精度设为 10，10 位有效数字
    cout << sqrt (2) << endl;   //输出十位。1.414213562
    cout << sqrt (2) * 100 << endl;   //141.4213562
    cout << setprecision (5) << sqrt (2) << endl    //将精度改为 5，1.4152
        << sqrt (2) * 100 << endl;   //141.42
    cout << "当前精度：" << cout.precision ()       //可以返回当前精度
        << endl;
    cout << setprecision (2) << 1.24 << ' ' << 1.25
        << ' ' << 1.26 << endl;        //将精度设为 2，1.2 1.2 1.3，只有大于 5 才会入，小于等于 5 都舍去。
    cout << showbase << hex << 127 << endl;   //打印进制标志，hex 为十六进制
    cout << oct << 127 << endl;   //oct 为 8 进制
    cout << dec << 127 << endl;   //dec 为 10 进制
    cout << noshowbase << hex << 127 << dec << endl;   //关闭现实进制，并恢复为十进制，一次性起作用，输出紧跟着的后一个输出，后面的不会管。
    cout << setw (12) << 127 << 721 << endl;   //设置域宽，默认靠右，填充空格，只对 127 起作用，对 721 不起作用，一次性。
    cout << setfill ('$') << left << setw (12)    //用$填充，左对齐，域宽为 12
        << 127 << endl;
    cout.precision (10);    //精度修改为 10，对科学计数法和定点形式意义不一样
    cout.setf (ios::scientific);  //以科学计数法输出
    cout << sqrt (2) << endl;   //1.4142135624e+00//小数部位为十位
```

```
    cout.setf (ios::fixed);     //一定点小数形式输出（正常的）
    cout << sqrt (2) << endl;     //1.414213562 有效数字为十位
    cout << 12.00 << endl;      //12.00000000  因为前面精度设为 10
    cout << showpoint << 12.00 << endl;     //显示小数点
    cout << noshowpoint << 12.00 << endl;    //不显示小数点

        ifstream ifs ("stream.txt");  //打开文件
    ifs.unsetf (ios::skipws);   //取消跳过空白，否则默认将空格等制表符认为是分隔作用，不读取
    char c;
    while (ifs >> c)
        cout << c;
    ifs.setf (ios::skipws);   //又设置回跳过空白
    ifs.clear ();      // 复位，将流复位，状态恢复到文件头，否则只改位置指针没有用
    ifs.seekg (ios::beg);   //此时位置指针不再文件头，这可以将位置指针返回文件头
    while (ifs >> c)
        cout << c;
    ifs.close ();
    cout << endl;

    int i = 1234;
    double d = 56.78;
    string s = "tarena";
    ostringstream oss;      //定义输出字符串流对象
    oss << i << ' ' << d << ' ' << s;
    string str = oss.str ();  //将字符串流流中的内容拿出来
    cout << str << endl;   //1234 56.78 tarena
    str = "hello 3.14 pai";
    istringstream iss;  //定义输入字符串流对象
    iss.str (str);       //将 str 的内容读入字符串流
    iss >> s >> d >> str;
    cout << s << ' ' << d << ' ' << str << endl;
    return 0;
}
```

十二、字符串流

```
class Student {
  ...
private:
  string m_name;
  int m_age;
};
Student s ("张飞", 25);
ofs.write (&s, sizeof (s));  //不能直接以二进制的方式写，这样只是写入了地址
```

例子：

```
#include <iostream>
#include <fstream>
#include <cstring>
//类中有字符串，如何写入文件
using namespace std;
class Dog {
public:
    Dog (const string& name = "", int age = 0) :
        m_age (age) {
        strcpy (m_name, name.c_str ());
    }
    void print (void) const {
        cout << m_name << ", " << m_age << endl;
    }
private:
    char m_name[128];
    int m_age;
};
int main (void) {
    ofstream ofs ("dog.dat");
    Dog dog ("小白", 25);
    ofs.write ((char*)&dog, sizeof (dog));   //写进去的只是地址，指针
    ofs.close ();
    ifstream ifs ("dog.dat");
    Dog dog2;
    ifs.read ((char*)&dog2, sizeof (dog2));
    dog2.print ();
    ifs.close ();
    return 0;
}
```

初学

C++ Primer Plus

进阶

C++ Primer
Effective C++
More Effective C++

深入

C++程序设计语言，Bjarena Stroustrup，机械版
深度探索 C++对象模型

休闲

C++语言 99 个常见错误
C++语言的设计与演化，Bjarena Stroustrup

NOT OVER

【Keep moving】

微信小程序：鼠友。国内首个大学生在线交流社区，既可以看外校新鲜事，又可以和本校同学沟通交流，实时发布信息，这是鼠于大学生们的友谊树洞~

如果您对金融领域一知半解，想学习金融领域相关知识，提高自身综合投资水平，获取相关金融服务，请关注微信公众号：财醒来

微信直接搜索关注微信公众号：财醒来，您可以获得以下服务：

1. 私人财富管理咨询服务，您通过公众号添加号主个人微信后，可结合自身情况咨询财富管理服务等。

2. 公众号会分享原创的宏观、股票、期货等二级市场复盘和投资参考，助力您发现投资机会。

3. 公众号不定期会分享号主自己的投资心得，投资策略等，带给您不一样的金融评论和金融思维。

4. 公众号后台回复：证券、期货，即可免费在行业龙头企业用超低手续费开户，开启你的投资生涯！

重磅福利：头部券商，万 1.1 开户

无论是买股票、基金还是期货，交易成本都是我们不可忽视的重要元素。

如果你自己去应用市场下载 XX 证券、XX 期货或者在同花顺开户，交易佣金一般默认为较高的万 2.5 或万 3 且没有客服服务。

现在，如果你从我们这边的专属渠道二维码开股票账户，可以享受到万 1.1 的开户优惠，并且有专属客服服务！

对于大部分人来说，每年至少可以省出一部苹果最新款手机的钱了，下面简单介绍一下该券商：

【AA 类券商，全国前五大券商，安全可靠】

【步骤简单，无需排队，全国都能开户，没有时间和地域的限制】

【营业部遍布全国，业务可以异地办理】

如何开户并享受最低佣金优惠？

搜索微信 ID：daxueshengqiang 或扫描下方二维码，添加客服微信并备注：开户，客服会辅助你开户！

扫一扫上面的二维码图案，加我微信

记住了，只有从上面二维码首次开户注册的，才可以享受最低佣金优惠！开户后客服会主动联系您进行佣金调整。

如果你自个儿去应用市场下载注册，就无法享受到这边的专属渠道福利了。

同时，我强烈推荐你开一个期货账户！期货账户最大的好处是，既可以做空又可以做多！

与股票账户相同，如果你从我们这边的专属渠道二维码开期货账户，也可以享受到超低手续费的开户优惠，并且有专属客服服务！下面简单介绍一下：

【AA 类期货公司，全国前五大期货公司，安全可靠】

【步骤简单，手机快速开户，足不出户即可开通】

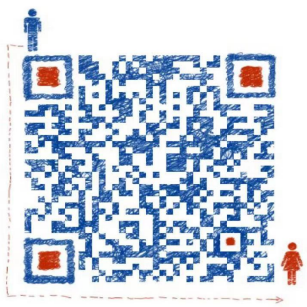
【营业部遍布全国，业务可以异地办理】

【超低手续费的开户优惠】

【一对一专属客服服务】

如何开户并享受最低佣金优惠？

搜索微信 ID：daxueshengqiang 或扫描下方二维码，添加客服微信并备注：开户，客服会辅助你开户！



扫一扫上面的二维码图案，加我微信