封装与接口

一.函数重载与缺省值

1.函数重载

- 同一名称的函数，有两个以上不同的函数实现，叫“函数重载”
- 重载函数必须保证至少有一个函数参数的类型有区别。返回值、参数名称等不能作为区分标志

2.缺省值

- 函数参数可以在定义时设置默认值（缺省值），这样在调用该函数时，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
- 有缺省值的函数参数，必须是最后一个参数

```
void print(char* name, int score,char* msg = "pass")
{
    cout << name << ": " << score<< ", " << msg << endl;
}
```

- 如果有多个带缺省值的函数参数，则这些函数参数都只能在没有缺省值的参数后面出现
- 缺省值存在冲突问题，如下代码不通过编译

```
void fun(int a,int b=1){}
void fun(int a){}
fun(2);
```

二.auto与decltype

1.auto关键字

- 由编译器根据上下文自动确定变量的类型
- 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，如`auto func(char* ptr, int val) -> int`；追踪返回类型在原本函数返回值的位置使用auto关键字
- auto变量必须在定义时初始化：`auto a`报错

- 函数参数不生声明为auto：`void func(auto a) {…}`报错
- auto并不是一个真正的类型。不能使用一些以类型为操作数的操作符，如sizeof或者typeid：`cout << sizeof(auto) << endl`报错
- auto的作用：
  1.用于代替冗长复杂、变量使用范围专一的变量声明。如vector中的迭代器
  2.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; //临时变量
std::cout << v;
}
//使用时
Multiply(2, 3); //Multiply(int, int)
Multiply(2, 3.3); //Multiply(int, double)
```

  3.与delctype结合

2.decltype关键字

- decltype可以对变量或表达式结果的类型进行推导
- 重用匿名类型

```
struct { int d ;
double b;
} anon_s,anon_t[100]; //没有名字的结构体，定义了一个变量

int main() {
decltype(anon_s) as ;//定义了一个上面匿名的结构体
decltype(anon_t) as;//定义了一个上面匿名的结构体构成的数组
}
```

3.auto+delctype

- 结合auto和decltype，自动追踪返回类型
  1.可以推导返回类型（C++11）

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

2.C++14中不再需要显式指定返回类型

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

三.类成员的访问权限 与 this指针

1.类成员的访问权限:

- public:可以被任意实体访问
- private:只允许本类的成员函数访问
- protected:只允许子类及本类的成员函数访问

2.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。这也是成员函数与普通函数的重要区别

四.内联函数

- 使用内联函数，编译器自动产生等价的表达式。

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
cout << max(a, b) << endl;
//上方代码与下方代码等价
cout << (a > b ? a : b) << endl;
```

- 内联函数和宏定义的区别
    1. 宏定义：编译器会将所有宏定义的代码，直接拷贝到被调用的地方。

```
#define MAX(a, b) (a) > (b) ? (a) : (b)
cout << MAX(a, b) << endl;
//上方代码与下方代码等价
#define MAX(a, b) (a) > (b) ? (a) : (b)
cout << (a) > (b) ? (a) : (b) << end;
```

2. 内联函数可以执行类型检查，进行编译期错误检查。内联函数可调试，而宏定义的函数不可调试。
    - 在Debug版本，内联函数没有真正内联，而是和一般函数一样，因此在该阶段可以被调试。
    - 在Release版本，内联函数实现了真正的内联，增加执行效率。
3. 宏定义的函数无法操作私有数据成员。

- 内联函数的注意事项
    1. 避免对大段代码使用内联修饰符。
        - 内联修饰符相当于把该函数在所有被调用的地方**拷贝**了一份，所以大段代码的内联修饰会增加负担。（代码膨胀过大）
    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。
        - 因为内联函数优化的，只是在函数调用的时候，会产生的压栈、跳转、退栈和返回等操作。所以如果函数内部执行代码的时间比函数调用的时间长得多，优化几乎可以忽略。
    3. 避免将内联函数的**声明和定义分开**
        - 编译器编译时需要得到内联函数的实现，因此多文件编译时内联函数先需要将实现写在头文件中，否则无法实现内联效果。
    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。
    6. 内联修饰符更像是**建议**而不是命令。编译器“有权”拒绝不合理的请求，例如编译器认为某个函数不值得内联，就会忽略内联修饰符。
    7. 编译器会对一些没有内联修饰符的函数，自行判断可否转化为内联函数，一般会选择短小的函数。

创建与销毁

一.构造函数

1.构造函数

- 没有返回值，函数名和类名相同
- 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

```
class Student {
int ID;//声明
public:
    Student(int id) : ID(id) { }
    Student(int year, int order) {
            ID = year * 10000 + order;
    }
    ...
};
```

- 初始化列表的成员是**按照声明的顺序初始化**的，而不是按照出现在初始化列表中的顺序
    - 在下面的代码中，编译器先初始化ID1，再初始化ID2，因此ID1的值将不可预测

```
class Student {
int ID1; //声明
int ID2; //声明
public:
    Student(int id) : ID2(id), ID1(ID2) { }
    ...
};
```

- C++11新增支持如下初始化操作，称为就地初始化,被初始化的变量不能是静态成员变量:

```
class A {
private:
    int a = 1; //声明+初始化
double b {2.0}; //声明+初始化
public:
    A() {} //a=1 b=2.0
    A(int i):a(i) {} //a=i b=2.0
    A(int i, double j):a(i), b(j) {}         //a=i b=j
};
```

注意：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

- 对象数组的初始化：
  1.无参定义对象数组，必须要有默认构造函数：`A a[50];`
  2.若构造函数只有一个参数：`A a[3] = {1, 3, 5};`三个实参分别传递给3个数组元素的构造函数
  3.若构造函数有多个参数：`A a[3] = {A(1, 2), A(3, 5), A(0, 7)};`

2.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;
    ...
};
```

3.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，被称为“默认构造函数”，也称“缺省构造函数”：`A(){}`
- 默认构造函数是隐式定义的，但若用户已经定义了其他构造函数，编译器将不会隐式合成默认构造函数。
  自C++11起，可以通过`Student()=default;`手动指定生成默认版本的构造函数。

    - 为什么不直接写`Student(){};`?
    - C++11中将POD数据类型分为了两个基本概念的集合，即平凡的（trival）和标准布局的（standard layout）。满足这两个基本概念的被称为POD类型。编译器对此数据类型具有优化作用。
    - 平凡的默认构造函数（trival constructor）即什么都不干。通常情况下，不定义类的构造函数, 编译器就会自动生成一个trivial constructor, 而一旦定义了构造函数, 即使构造函数中不包含任何参数，函数体中也没有任何代码，**那么该构造函数都不再是trivial的**，但是我们可以使用C++11中的新的关键字“=defualt”来显式的声明缺省版本的构造函数从而**使类型恢复平凡化**。
    - 详情见链接：https://blog.csdn.net/u011475134/article/details/72900890
- 默认构造函数的调用：
  `ClassName a;ClassName a = ClassName();`等价
  而`ClassName c();`声明了一个ClassName为返回值的函数
- 在类的构造函数中，除了执行函数体内声明的语句，编译器还会做一些额外操作，如调用成员变量（在此类中声明另一个类的对象）的默认构造函数。且先调用成员变量的构造，再调用自己的构造。
- 显式删除构造函数：

```
class Student {
private:
    int ID = 1;
    char class = 'a';
public:
    Student() = default;
    Student(int i):ID(i) {}
    Student(char cls) = delete;//不加这一句会通过编译，因为'c'会作为ASCII码传入上方的形参为int的函数，从
};

Student s('c'); //编译错误
```

4.拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，它的参数是语言规定的，是同类对象的常量引用;作用：用参数对象的内容初始化当前对象

```
class Person {
    int id;
    ...
public:
    Person(const Person& src) { id = src.id;  ... }
    ...
};
```

- 拷贝构造函数被调用的三种常见情况：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。
    1. 用一个类对象定义另一个新的类对象
       `Test a; Test b(a);Test c = a;`
    2. 函数调用时以类的对象为形参：`Func(Test a)`
    3. 函数返回类对象 `Test Func(void)`
- 如果调用拷贝构造函数且当前没有给类显式定义拷贝构造函数，编译器将自动合成“隐式定义的拷贝构造函数”，其功能是调**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等)来说，默认的拷贝方式为位拷贝(Bitwise Copy)，即直接对**整块内存进行复制**。对于自定义类，会递归调用所有数据成员的拷贝构造函数或拷贝赋值运算符。
- 位拷贝在遇到指针可能会出错，导致多个指针指向同一个地址。此时不应该使用隐式定义的拷贝构造函数。
- 正常情况下应避免使用拷贝构造函数，以防止运算效率下降与指针错误发生。
    1. 使用引用/常量引用传参数或返回对象
       `func(MyClass a) => func(const MyClass& a)`
       ，`MyClass func(…) => MyClass& func(…)`
    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;
    ......
}
```

- 以上情况建立在禁用返回值优化（RVO）的情况下。如果返回值为对象，理论上会调用对象的构造和析构函数，导致效率损耗。
  如`A test(){return A();}`后`A a=test()`但是编译器会采用RVO技术，将test函数里A的构造过程直接用在a上，从而只付出一个构造函数的代价，节省了构造临时对象的代价。
    - 编译器进行ROV的**两个条件**：1.return的值的类型与函数签名的返回值类型相同。2.return的是一个局部对象。
    - `g++ test.cpp --std=c++11 -fno-elide-constructors -o test`可以禁用返回值优化

5. 移动构造函数：

- 右值引用：

1. **左值**：可以取地址、有名字的值。
   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
   如`int a=1`,`int b=func()`,`int c=a+b`z中a,b,c为左值，1,func(),a+b为右值
2. 左值可以取地址并可以被&引用。

```
    int *d = &a; int &d = a;//正确
    int *e = &(a + b);int &e = a + b;//错误
```

3. 右值无法取地址，但是可以被&&引用（右值引用）：`int &&e = a+b;`,但右值引用无法绑定左值,如`int &&e = a;`是错误的。
4. 左值引用可以绑定左值，右值引用可以绑定右值。但是常量左值引用（其实就是常引用）也可以绑定右值。`cosnt int &e = 3;const int &e = a*b;`
   常见引用绑定规则：

\	非常量左值	常量左值	右值
非常量左值引用	√		√

\	非常量左值	常量左值	右值
常量左值引用	√	√	√
右值引用			√

注意：所有的引用（含右值引用）本身都是左值

5. 示例：

```
void ref(int &x) {
    cout << "left " << x << endl;
}

void ref(int &&x) {
    cout << "right " << x << endl;
    ref(x);//调用left的
}

int main() {
    int a = 1;
    ref(a);//left 1
    ref(2);//right 2
           //left 2
    return 0;
}
```

- 移动构造函数:
    1. 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(ClassName&& VariableName);`
    2. 拷贝构造函数为新对象**重新开辟堆内存**并拷贝临时对象的数据，**再析构掉临时对象**；而移动构造函数直接将新对象指向那片原有的堆内存，从而对于即将析构的临时类，移动构造函数直接利用**原来临时对象**的堆内存，**新对象无需开辟内存，临时对象无需释放内存**，大大提高效率。
    3. 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;
}
```

在不禁用返回值优化时的返回值：

```
Test(): this->buf @ 0x7fa908c04b90
GetTemp(): tmp.buf @ 0x7fa908c04b90
main() : a.buf @ 0x7fa908c04b90
Test(const Test&) called.
    this->buf @ 0x7fa908c04ba0
fun(Test t): t.buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04ba0
~Test(): this->buf @ 0x7fa908c04b90
```

在禁用返回值优化时的返回值:

```
Test(): this->buf @ 0x7f8951c04b90
GetTemp(): tmp.buf @ 0x7f8951c04b90
Test(Test&&) called. this->buf @ 0x7f8951c04b90
~Test(): this->buf @ 0x0 (tmp)
Test(Test&&) called. this->buf @ 0x7f8951c04b90 ~a=GetTemp()
~Test(): this->buf @ 0x0 ~GetTemp()
main() : a.buf @ 0x7f8951c04b90
Test(const Test&) called. this->buf @ 0x7f8951c04ba0
fun(Test t): t.buf @ 0x7f8951c04ba0
~Test(): this->buf @ 0x7f8951c04ba0
~Test(): this->buf @ 0x7f8951c04b90
```

1. 第一个移动构造：把tmp的内容交给了 GetTemp的返回值,GetTemp返回值占用了tmp的内存
2. ~Test(): this->buf @ 0x0 (tmp) =>第一次的时候，指针为空不是被自己删除的,而是在移动构造函数里被GetTemp的返回值删除的
3. Test a=GetTemp(); a又占用了GetTemp()的内存；

删除移动构造函数、并且禁止编译器优化的输出结果：

```
Test(): this->buf @ 0x7fabf8c04b50
GetTemp(): tmp.buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf @ 0x7fabf8c04b50
~Test(): this->buf @ 0x7fabf8c04b60
main() : a.buf @ 0x7fabf8c04b50
Test(const Test&) called. this->buf @ 0x7fabf8c04b60
fun(Test t): t.buf @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b60
~Test(): this->buf @ 0x7fabf8c04b50
```

注意：`Test a=GetTemp()`一句话进行了两个内存的三次反复分配。

- 移动构造函数加快了右值初始化构造速度，左值也可以显式调用移动构造函数加快速度。如`Test a;Test b = std::mobe(a);`move函数本身不对对象做任何操作，仅做类型转换，即转换为右值。

6.拷贝/移动构造函数的调用时机

- 可以引用的绑定规则为判断依据：
    1. 拷贝构造函数的形参类型为常量左值引用，可以绑定常量左值、左值和右值
    2. 移动构造函数的形参类型为右值引用，可以绑定右值
    3. 引用的绑定**存在优先级**，例如常量左值引用和右值引用均能绑定右值，当传入实参类型为**右值时优先匹配形参类型为右值引用的函数**
- 拷贝构造函数的常见调用时机：
    1. 用一个类对象/引用/常量引用初始化另一个新的类对象
    2. 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(因为都是左值)**
    3. 函数返回类对象（**类中未显式定义移动构造函数，不进行返回值优化**）
- 移动构造函数的调用时机:
    1. 用一个类对象的右值初始化另一个新的类对象（常配合std::move函数一起使用）：
       `Test b = func(a); Test b = std::move(a);`与`Test b = a;`不同。
    2. 以类的对象为函数形参，传入实参为类对象的右值（常配合std::move函数一起使用）：`func(Test());func(std::move(a));`与`func(a)`不同
    3. 函数返回类对象（**类中显式定义移动构造函数，不进行返回值优化**）：`return Test();` 或 `return tmp;`均调用移动构造

7.拷贝/移动赋值运算符

1. 拷贝赋值运算符：

- 已定义的对象之间相互赋值，可通过调用对象的“拷贝赋值运算符函数”来实现的

```
    ClassName& operator= (const ClassName& right) {
        if (this != &right) {// 避免自己赋值给自己
                // 将right对象中的内容拷贝到当前对象中
            }
        return *this;
    }
```

注意： 以下代码上方是拷贝赋值，下方是拷贝构造。

```
    ClassName a;
    ClassName b;
    a = b;
    //
    ClassName a = b;
```

- 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。**（因为同类对象赋值代表已经是成员函数，没必要友元）

2. 移动赋值运算符：

```
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;
    }
    return *this;
}
swap(Test& a, Test& b) {
     Test tmp(std::move(a)); // 第一行调用移动构造函数
     a = std::move(b);       // std::move的结果为右值引用，
     b = std::move(tmp); // 后两行均调用移动赋值运算
}
```

3. 拷贝/移动赋值运算符的调用时机：
    - 和拷贝/移动构造函数的调用时机类似，主要判断依据是**引用的绑定规则**
        1. 拷贝赋值运算符函数的形参类型为常量左值引用，可以绑定常量左值、左值和右值
        2. 移动赋值运算符函数的形参类型为右值引用，可以绑定右值(常量、表达式、函数返回)
        3. 引用的绑定存在优先级，例如常量左值引用和右值引用均能

绑定右值，当赋值运算符右侧为右值时优先匹配形参类型为右值引用的赋值运算符函数

4. 根据赋值运算符右侧变量的类型决定调用拷贝或移动赋值运算符函数

二.析构函数

- 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。编译器在对象生命期结束时自动调用类的析构函数，以便释放对象占用的资源，或其他后处理
- 和默认构造函数一样，析构函数除了执行函数体内声明的语句，编译器还会做一些额外操作,如自动调用成员变量的析构函数，且先执行自己的析构函数，在调用成员变量的析构函数。
- 和构造函数类似，当用户没有自定义析构函数时，编译器会自动合成一个隐式的析构函数。但隐式定义的析构函数不会delete指针成员。
- 1. 局部对象：在程序执行到该局部对象的代码时被初始化；在局部对象生命周期结束、即所在作用域结束后被析构。
  2. 全局对象：在main()函数调用之前进行初始化；在同一编译单元中，按照定义顺序进行初始化（编译单元：通常同一编译单元就是同一源文件）；不同编译单元中，对象初始化顺序不确定；在main()函数执行完return之后，对象被析构。
- 尽量避免使用全局对象：1.构造顺序不能完全确定，所以全局对象间不能形成依赖关系。2.全局对象会增大代码的耦合性（耦合性越大，代码间独立性越差），导致程序难以复用或测试。
- 含有指针的类实例的析构：

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

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，因为`fun(A a)`调用了拷贝构造函数，但是指针指向的是同一片内存，在函数结束时释放了一次，`return 0;`会释放第二次，从而这时候报错。
将函数改为`fun(A &a)`就可以避免调用拷贝构造函数，从而避免错误，同时可以减少时间开销。

三.引用

- 引用必须在定义时初始化，且不能修改引用指向
- 引用和变量在内存中是同一单元的两个不同名字，修改其中任一者，另一者也会修改
- 函数参数可以是引用类型，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。如调用以下函数将交换实参的值：

```
void swap(int& a, int& b)
{  int tmp = b; b = a; a = tmp; }
```

- 函数返回值可以是引用类型，但不得指向函数的临时变量
- 引用和指针的区别：
  1.不存在空引用。引用必须连接到一块合法的内存。
  2.一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。

3.引用必须在创建时被初始化为一个对象。指针可以在初始化时置空，之后再指向对象。

四.运算符重载

- 可以重载的运算符：
    - 双目算术运算符
      + (加)，-(减)，*(乘)，/(除)，% (取模)
    - 关系运算符
      ==(等于)，!= (不等于)，< (小于)，> (大于)，<=(小于等于)，>=(大于等于)
    - 逻辑运算符
      ||(逻辑或)，&&(逻辑与)，!(逻辑非)
    - 单目运算符
      + (正)，-(负)，*(指针)，&(取地址)
    - 自增自减运算符
      ++(自增)，--(自减)
    - 位运算符
      | (按位或)，& (按位与)，~(按位取反)，^(按位异或)，<< (左移)，>>(右移)
    - 赋值运算符
      =, +=, -=, *=, /= , % = , &=, |=, ^=, <<=, >>=
    - 空间申请与释放
      new, delete, new[] , delete[]
    - 其他运算符
      ()(函数调用)，->(成员访问)，,(逗号),
- 运算重载一般有两种方式（注意参数不同）：

1. 全局函数的运算符重载
   `A operator+(A a, A b) {…}`
2. 成员函数的运算符重载

```
class A{
int data;
public:
    A operator+(A b) {…};
};
```

3. =,[],(),->只能通过成员函数来重载。以“=”为例：当没有自定义operator=时，编译器会自动合成一个默认版本的赋值操作;在类内定义operator=，编译器则不会自动合成;如果允许使用全局函数重载，可能会对是否自动合成产生干扰。

- “+”的三种重载：同一运算符只能采用一种实现

```
#include <iostream>
using namespace std;
class A {
public:
    int data;
    A(int i) { data = i; }
    ① A& operator+(A& a) { data += a.data; return *this;}
    ②// A operator+(A& a) {
     //     A new_a(data + a.data);
     //     return new_a;
     // }
};
③//A operator+(A& a1, A& a2) {
 //A new_a(a1.data + a2.data);
 //return new_a;
 //}
int main() {
    A a1(2), a2(3);
    a1 += a2; // 调用operator+=()
    cout << a1.data << endl; // 5
    cout << (a1 + a2).data << endl; // 调用operator+()
    return 0;
}
```

- 前缀与后缀的“++”，“--”的重载

1.前缀运算符重载声明

```
    ClassName operator++();
    ClassName operator--();
```

2.后缀运算符重载声明

```
    ClassName operator++(int dummy);
    ClassName operator--(int dummy);
```

3.通过函数体中的哑元参数dummy区分前缀和后缀的同名重载，哑元可以没有变量名，如`int fun(int, int a){ return a/10*10; }`,函数在调用时第一个值随便给，最终会被丢弃(某个参数如果在子程序或函数中没有用到，那就被称为哑元。这是程序设计语言中的一个术语，函数的形参又称“哑元”，实参又称“实元”。在C++的运算符重载中，就会用到哑元以区分i++与++i的区别。哑元表示虚无的元素，没有实际空间，甚至连名字都可以没有，它只有联系上实元才有意义。)

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

class A {
public:
    int data;
    A() { data = 0; }
    A(int i) { data = i; }
};

A operator++(A& a) {//前缀
    ++a.data;
    return a;
}
A operator++(A& a, int) { //哑元，后缀
    A new_a(a.data);
    ++a.data;
    return new_a;
}

int main() {
    A a(1);
    cout << (++a).data << endl; // 2
    cout << (a++).data << endl; // 2
    cout << a.data << endl; // 3
    return 0;
}
```

- “()”的重载

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

class Test {
public:
    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对象看上去象是一个函数，故也称“函数对象”
    //上下等价
    int s = sum.operator()(3, 4);
    return 0;
}
```

- “[]”的重载
    - 函数声明形式:返回类型 operator[] (参数);
    - 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，即`Obj[index] = value;`
    - 如果返回类型不是引用，则只能出现在等号右边`value = Obj[index];`
    - **注意**：这里Obj是一个对象，而不是一个数组

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

char week_name[7][4] = { "mon", "tu", "wed", "thu", "fri", "sat", "sun"};

class WeekTemperature {
    int temperature[7];
    int error_temperature;
public:
    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;
    cout << "Monday Temperature: " << beijing["mon"] << endl;
    return 0;
}
//结果为：Monday Temperature -3
```

- 流运算符“>>”,“<<”的重载

```
istream& operator>> (istream& in, Test& dst );

ostream& operator<< (ostream& out, const Test& src );
```

1. 函数名为：operator>> 和 operator<<
2. 不修改istream和ostream类的情况下，只能使用全局函数重载
3. 返回值为：istream& 和 ostream&，均为引用
4. 参数分别：流对象的引用、目标对象的引用。对于输出流，目标对象一般是常量引用。

```
#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 ostream& operator<< (ostream& out, const Test& src);
};
istream& operator>> (istream& in, Test& dst) {
    in >> dst.id;
    return in;
}
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}

int main() {
    Test obj(1);
    cout << obj;  // operator<<(cout,obj)
    cin >> obj;         // operator>>(cin,obj)
    cout << obj;
    return 0;
}
```

5. 为什么返回值使用引用？
   避免复制。ostream的复制构造函数：`ostream& (const ostream&) = delete;ostream& (ostream&& x);`禁止了复制，只允许移动，仅使用cout一个全局对象。从而减少复制的开销，同时一个对象对应一个标准输出更合理，多个对象无法同步输出状态。

```
ostream& operator<< (ostream& out, const Test& src) {
    out << src.id << endl;
    return out;
}
//测试代码
cout << obj << obj2 << obj3 << endl;

//等价于
ostream& out = operator<<(cout, obj); //return cout;
ostream& out1 = operator<<(out, obj2); //return cout;
ostream& out2 = operator<<(out1, obj3); //return cout;
```

五.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，即可以访问该类的一切成员。友元的声明只能在类内进行。

```
class A {
    int data;//默认私有成员
    friend void foo(A &a);
};

void foo(A &a) {
    cout << a.data << endl;
}//可以访问A的私有成员和保护成员
```

- 被友元声明的函数一定不是当前类的成员函数，即使该函数的定义写在当前类内

```
#include <iostream>
using namespace std;
class A {
private:
    int data;
public:
    A(int i) : data(i) {}
    void print() { cout << data << " inside\n"; }
    friend void print(A a)  // 这一行的print是全局函数
    { cout << a.data << " outside\n"; }
};
int main() {
    A c(1);
    c.print(); // 1 inside
    print(c); // 1 outside
    return 0;
}
```

- 可以声明别的类的成员函数，为当前类的友元。其中，构造函数、析构函数也可以是友元。

```
class Y {
    int data;
    friend void X::foo(Y);
    friend X::X(Y), X::~X();
};
//X的构造函数X::X()和析构函数X::~X()为Y的友元函数，则在它们的函数体内可直接访问/修改Y的私有成员。
```

- 友元的声明与当前所在域是否为private或public无关
- 一个普通函数可以使多个类的友元函数
- 可对class/struct/union进行友元声明，代表该类的所有成员函数均为友元函数

```
class Y {}; // 定义类Y，且Y能访问A的所有成员
class A {
    int data;  // 私有数据成员

    friend class X;  // 友元类前置声明（详细类型指定符）
    friend Y;  // 友元类声明（简单类型指定符）（C++11起）
};
class X {}; // 定义类X，X能访问A的所有成员
//注意X,Y声明的差别
```

- **注意事项:**

1. 友元非对称：A中声明B为A的友元，则A不可以访问B的私有成员
2. 友元不传递
3. 友元不继承
4. 友元声明不能定义新的class，如在类中：`friend class Y{};`是错的

六.静态成员:

- 1. 静态变量：使用static修饰的变量
     - 初始化：初次定义时需要初始化，且只能初始化一次。
     - 静态局部变量存储在静态存储区，生命周期将持续到整个程序结束
     - 静态全局变量是内部可链接的，作用域仅限其声明的文件，不能被其他文件所用，可以避免和其他文件中的同名变量冲突
  2. 静态函数：使用static修饰的函数
     - 静态函数是内部可链接的，作用域仅限其声明的文件，不能被其他文件所用，可以避免和其他文件中的同名函数冲突
  - 在a.cpp中`static int i = 1;`在b.cpp中`extern int i;`会报错。而在.h中定义static可以在多.cpp中使用，因为相当于在多个cpp中定义了自己范围内的static变量或函数。include.h是将代码copy到对应的cpp中去。
  - 类的静态数据成员，也叫“类变量”,使用static修饰的数据成员，是隶属于类的,**需要类内声明，类外赋初始值**

  1. 静态数据成员被该类的所有对象共享（即所有对象中的这个数据域处在同一内存位置）
  2. 类的静态成员（数据、函数）既可以通过对象来访问，也可以通过类名来访问，如ClassName::static_var或者a.static_var（a为ClassName类的对象）
  3. 类的静态数据成员要在实现文件中赋初值，格式为：`Type ClassName::static_var = Value;`
  4. 和全局变量一样，类的静态数据成员在程序开始前初始化
  5. 静态数据成员应该在.h中声明（`class Test{static int count;};`），在.cpp中定义（`int Test::count=0`）。如果在.h中同时声明定义，由于包含了该头文件的所有源文件都定义了这些静态成员变量，则被包含多少次，同一变量就被定义多少次，从而导致链接错误，编译失败。

  - 类的静态成员函数，在返回值前面添加static修饰的成员函数，称为类的静态成员函数

1. 和静态数据成员类似，类的静态成员函数既可以通过对象来访问，也可以通过类名来访问，如`ClassName::static_function`或者`a.static_function`(a为ClassName类的对象）
2. 静态成员函数不能访问非静态成员静态成员函数属于整个类，在类实例化对象之前已经分配了内存空间。类的非静态成员必须在类实例化对象后才分配内存空间。如果使用静态成员函数访问非静态成员，相当于没有定义一个变量却要使用它。

- 类的静态成员如果是私有的，也可以且需要在类外访问并定义，这是私有变量能在类外访问的特例。于此同时，也可以利用这个特例达到其他的特例，比如在类外调用私有的构造函数：

```
#include<iostream>
using namespace std;
class T {
private:
    T() { cout << "created!"; }
    static T t;
};

T T::t;

int main()
{
    return 0;//输出“created!”
}
```

七.常量成员

- 常用于修饰变量、引用/指针、函数返回值等，可以就地初始化或者在构造函数的初始化列表中初始化
- 常量数据类型
    1. 修饰变量时（如const int n = 1;），必须**就地初始化**，该变量的值在其生命周期内都不会发生变化
    2. 修饰引用/指针时（如int a=1; const int& b=a;），不能通过该引用/指针修改相应变量的值，常用于函数参数以**保证函数体中无法修改参数**的值
    3. 修饰函数返回值时（如const int* func() {…}），函数返回值的内容（或其指向的内容）不能被修改
- 常量成员函数

  `ReturnType Func(…) const {…}`

注意区别：`const ReturnType Func(…) {…}` (返回值为常量)

1. 常量成员函数的访问权限：实现语句不能修改类的数据成员，即不能改变对象状态
2. 若对象被定义为常量`const ClassName a;`，则它只能调用以const修饰的成员函数。否则如果能访问一般函数，则一般函数可能对成员变量进行修改，与常量对象的值维持不变矛盾。

- 常量引用
    1. 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。
    2. 如果不想给予函数修改权限，则可以在参数中使用常量/常量引用。`void add(const int& a, const int& b)`

八.常量静态变量

- 常量静态变量需要在类外定义（和静态变量一样）。但有例外：int和enum类型可以就地初始化
- 常量静态变量和静态变量一样，满足访问权限的任意函数均可访问，但都不能修改
- 不存在常量静态函数：静态函数隶属于类，可以不实例化直接通过类名访问。但是常量/非常量函数的访问权限需要通过实例化后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,所以两种属性相冲突。
-

常量、静态成员总结

	静态数据成员	常量数据成员	常量静态数据成员(除int, enum外)	常量静态数据成员(int, enum)
初始化				
就地初始化		√		√
初始化列表初始化		√		
构造函数体内初始化				
类外初始化	√		√	√
访问				
普通成员函数	√	√	√	√
静态成员函数	√		√	√
常量成员函数	√	√	√	√
修改				
普通成员函数	√			
静态成员函数	√			
常量成员函数	√			

九.对象的new和delete

- `A *pA = new A(some parameters);`生成类对象并返回地址，调用构造函数。`delete pA;`删除类对象，释放内存资源，调用析构函数。
- 生成类对象的数组:`A *pA = new A[3]`
    1. 调用operator new[]标准库函数来分配足够大的原始未类型化的内存。注意：会多出4个字节来存放数组的大小。
    2. 在刚分配的内存上运行构造函数对新建的对象进行初始化构造。
    3. 返回指向新分配并构造好的对象数组的指针。
- 删除类对象数组：`delete []pA;`
    1. 对数组中各个对象运行析构函数，数组的维数保存在pA前4个字节里。
    2. 调用operator delete[ ]标准库函数释放申请的空间。不仅仅释放对象数组所占的空间，还有记录维数的4个字节。
- 如果使用了`new[]`和`delete`，会造成：
    1. 只调用**一次**析构函数。如果类对象中有大量申请内存的操作，那么因为没有调用析构函数，这些内存无法被释放，造成**内存泄漏**。

2. 直接释放pA指向的内存空间，这个会造成严重的**段错误**，程序必然会崩溃。因为分配空间的起始地址是pA-4byte。（delete[] pA的释放地址自动转换为pA-4byte）

十.类型转换

- 当编译器发现表达式和函数调用所需的数据类型和实际类型不同时，便会进行自动类型转换。自动类型转换可通过定义特定的转换运算符和构造函数来完成。
- 自动类型转换方法：(不可同时使用)
    1. 源类中定义“目标类型转换运算符”

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

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

class Src { //源类Source
public:
    Src(){ cout << "Src::Src()" << endl; }
    operator Dst() const {
        cout << "Src::operator Dst() called" << endl;
        return Dst();
    }
};
```

注意：无需指定返回类型，因为必然是Dst这一转换后类型

2. 目标类中定义“源类对象作参数的构造函数”

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

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

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

3.以上写法都可以成功运行以下代码：

```
void Transform(Dst d) { }

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

    Dst d2 = s;
    Transform(s);
    return 0;
}
```

- 示例1:

```
class SmallInt;
operator int(SmallInt&); //错误：不是成员函数
class SmallInt{
public:
    int operator int() const;        //错误：不能返回类型
    operator int(int = 0) const; //错误：参数列表不为空
    operator int*() const {return 42;} //错误：42不是一个合法指针
                                       //本意：将SmallInt对象转换为int* 指针
};
```

- 示例2：

```
class SmallInt{
public:
//构造函数:从int转换为SmallInt
    SmallInt (int i=0): val(i){
        cout<<"SmallInt_Init"<<endl;
    }
    operator int() const {//转换运算符:从SmallInt 转换为int
        cout<<"Int_Transform"<<endl;
        return val;
    }
    void print() {
        cout << val << endl;
    }
private:
    size_t val;
};
int main()
{
    SmallInt si;
    si = 4.10;
    si = si + 3;
    si.print();
    return 0;
}
```

最终输出：

```
SmallInt_Init
//SmallInt si, 调用构造函数
SmallInt_Init
//si = 4.10, 首先内置类型转换将double转换为int,
//然后调用构造函数隐式地将4转换成SmallInt
Int_Transform
//si + 3, 调用类型转换运算符将si隐式地转换成int
SmallInt_Init
//si = si + 3, 调用构造函数隐式地将si + 3的结果转换成SmallInt
7
```

注意：`si=si+3;`一句因为编译器不会自动生成SmallInt类的加号重载，所以不会将3转化为SmallInt后再相加。如果加上这样一个加号重载就会报错：模糊调用。当去除int->SmallInt的类型转换时，只用加号重载，就会把3转换后相加。

- 禁用自动转换类型：
  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

式地进行

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

此时最初的例子中`Dst d1(s);`是显式初始化，可以执行，

而`Dst d2 = s;`，`Transform(Dst d);`是隐式转换，报错。

- 可以使用强制类型转换:
    - 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。
    - 则之前例子改

      为：`Dst d2 = static_cast<Dst>(s);`

      和`Transform(static_cast<Dst>(s));`即可正常运行。

组合与继承

一.组合

```
class A{};
class B{};
class C
{
private:
    A a;
public:
    B b;
};
```

- 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。
- 先完成子对象的构造（多个子对象的构造次序仅由在类中声明的次序决定），再完成当前对象的构造；析构相反
- 如果调用拷贝构造函数且没有给类显式定义拷贝构造函数，编译器将提供“隐式定义的拷贝构造函数”。该函数的功能为：递归调用子对象拷贝构造函数，对于基础类型，采用位拷贝；拷贝赋值运算符同理。

二.继承

- 不能（或在部分版本中不能）被继承的函数：
    - 构造函数：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。构造时，先调用基类，再派生类。
    - 析构函数：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

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

class Base{
public:
    int k = 0;
    void f(){cout << "Base::f()" << endl;}
    Base & operator= (const Base &right){
        if(this != &right){
            k = right.k;
            cout << "operator= (const Base &right)" << endl;
        }
        return *this;
    }
};
class Derive: public Base{};
int main(){
    Derive d, d2;
    cout << d.k << endl; //Base数据成员被继承
    d.f(); //Base::f()被继承

    Base e;
    //d = e; //编译错误，Base的赋值运算符不被继承
    d = d2;  //调用隐式定义的赋值运算符
    return 0;
}
//结果:0
       Base::f()
       operator= (const Base &right)
```

- 派生类对象的构造：
  基类数据成员需要在构造派生类的过程中调用**基类**构造函数来正确初始化。若没有显式调用，编译器会自动调用基类默认构造函数。若想要显式调用，需要在派生类构造函数的**初始化成员列表**进行；或者使用`using Base::Base`直接继承基类的所有构造函数。
  (1) 若没有显式调用，则编译器会自动生成一个对基类的默认构造函数的调用。

```
class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }
    /// 默认构造函数，如果没有，在此段代码中会报错
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() { cout << "Derive::Derive()" << endl; }
    /// 无显式调用基类构造函数，则调用基类默认构造函数(需要存在)
};
int main() {
    Derive obj;
    return 0;
}
```

（2）若想要显式调用，则只能在派生类构造函数的初始化成员列表中进行；上方派生类的构造函数改为：`Derive(int i) : Base(i) { cout << “Derive::Derive()” << endl; }` 即可

（3）在派生类中使用 using Base::Base; 来继承基类构造函数，相当于给派生类“定义”了相应参数的构造函数

```
class Base
{
    int data;
public:
    Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
    Base(int i, int j)
        { cout << "Base::Base(" << i << “," << j << ")\n";}
};
class Derive : public Base {
public:
    using Base::Base;
    ///相当于 Derive(int i):Base(i){};
    ///加上 Derive(int i, int j):Base(i, j){};
};
int main() {
    Derive obj(356);
    Derive obj2(356, 789);
    ///不能用Derive obj构造对象了，因为派生类没有默认构造函数
    return 0;
}
```

（4）如果基类构造函数为私有则不可以在派生类中声明继承该构造函数。

- 继承的方式：
    - 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。**所有接口都如同自身函数使用。
    - 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**。可用于隐藏/公开基类的部分接口，通过使用using关键字。当私有继承时，我们是“照此实现”（is-implementing-in-terms-of）；也就是说，创建的新类**具有基类的所有数据和功能，但这些功能是隐藏的**，所以它只是部分的内部实现。该类的用户访问不到这些内部功能，并且一个对象不能被看做是这个基类的实例。**相当于重新实现了一遍基类的功能，而且它们是私有的**。
      此时不允许向上转换`Derived d;Base& ptr = d.`否则ptr作为实例化对象可以绕过d的权限限制调用base里的函数。
    - 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。基本不用，存在是为了语言的完备性。
- 成员访问权限：
    - **基类私有成员**也不允许在派生类成员函数中或者派生类对象访问。
    - **基类公有成员**
        1. 允许在派生类成员函数中被访问
        2. 如果是public继承，则成为派生类公有成员，可以由派生类对象访问
        3. 若是使用private/protected继承方式，则成为派生类私有/保护成员，**不能被派生类的对象访问**。若想让某成员能被派生类的对象访问，可在派生类public部分用关键字using声明它的名字。
    - **基类保护成员**保护成员允许在派生类成员函数中被访问，但不能被外部函数访问。
- 私有继承示例
    1. 访问基类公有成员

```
#include <iostream>
using namespace std;
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive2: private Base
{/// 私有继承， is-implementing-in-terms-of：用基类接口实现派生类功能
public:
void deriveFunc() {
    cout << “in Derive2::deriveFunc(), calling Base::baseFunc()..." << endl;
    baseFunc(); /// 私有继承时，基类接口在派生类成员函数中可以使用
    }
};

int main() {
    Derive2 obj2;
    cout << "calling obj2.deriveFunc()..." << endl;
    obj2.deriveFunc();
    //obj2.baseFunc(); 错误：基类接口不允许从派生类对象调用

    return 0;
}
```

2. 打开派生类对象对基类公有成员的访问权限方式：在派生类中写`using Base::baseFunc;`,则上方示例代码中`obj2.baseFunc()`不会报错。
3. 访问基类私有与保护成员

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

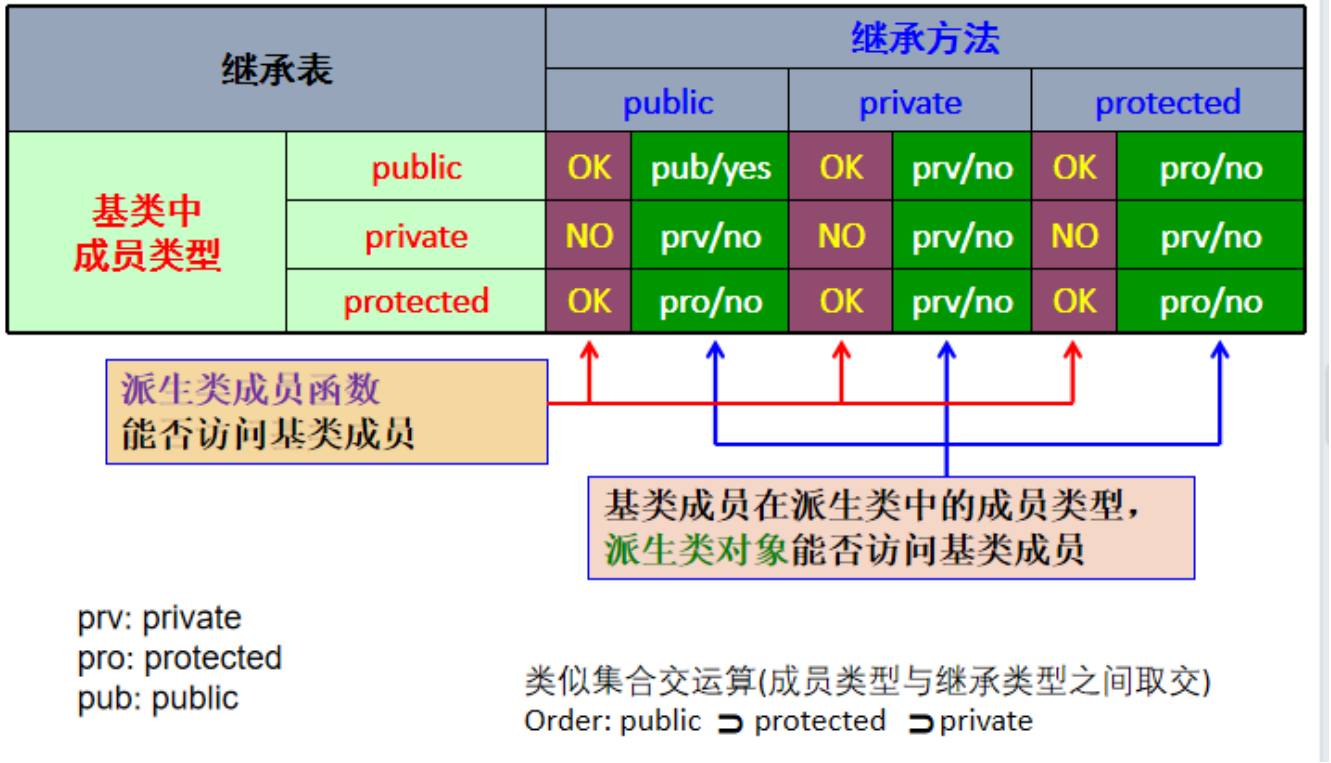
class Base{
    private:
    int a{0};
    protected:
    int b{0};
};

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;
}
```

-

成员访问权限



下方“类似集合交运算”的理解：private成员和public继承相交得到private;public成员和private继承得到private

三.重写隐藏与重载

- 重写隐藏的在派生类中重新定义基类函数，实现派生类特殊功能。其屏蔽了基类所有其他同名函数。需要派生类中函数名和基类相同，参数可以不同。而重载需要在相同域里，函数名相同，参数必须不同。
- 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，不能提供给派生类对象使用。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

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
    void f() { cout << "B::f()\n"; }
    void f(int i) { cout << "Base::f(" << i << ")\n"; } /// 重载
    void f(double d) { cout << "Base::f(" << d << ")\n"; } ///重载
    void f(T) { cout << "Base::f(T)\n"; } ///重载
};
class Derive : public Base {
public:
    void f(int i) { cout << "Derive::f(" << i << ")\n"; } ///重写隐藏
};
int main() {
    Derive d;
    d.f(10);
    d.f(4.9);         /// 编译警告。执行自动类型转换。
    //d.f();         /// 被屏蔽，编译错误
    //d.f(T());         /// 被屏蔽，编译错误
    return 0;
}
//在Derive中写using Base::f;即可使main中后三个f恢复正常调用基类的f函数。
```

四.多重继承

1. **数据存储**： 如果派生类D继承的两个基类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有两份独立的副本，可能带来数据冗余。
2. **二义性**： 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，编译器无法判断要访问的哪一个基类成员。
3. 因此最好采用：最多继承一个非抽象类，继承多个抽象类，从而避免二义性，利用一个对象实现多个接口

五.虚函数

（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，称为向上类型转换。只对public继承有效，在继承图上是上升的；对private、protected继承无效。
- 向上类型转换（派生类到基类）可以由编译器自动完成，是一种隐式类型转换。
- 凡是接受基类对象/引用/指针的地方（如函数参数），都可以使用派生类对象/引用/

指针，编译器会自动将派生类对象转换为基类对象以便使用。

（2）对象切片：

- 当派生类的对象（非指针/引用）被转换为基类对象时，派生类的对象被切片为对应基类的子对象，即派生类特有的数据丢失。

（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，派生类的接口无法使用。

（4）函数调用捆绑：

- 把函数体和函数调用相联系称为捆绑，即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绑定好的函数体内部
- 当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑(early binding)。即在运行之前已经决定了函数调用到底进入哪个函数。
- 当捆绑根据对象的实际类型（说明对象自身要包含自己实际类型的信息，即虚函数表），发生在程序运行时，称为晚捆绑(late binding)，又称动态捆绑或运行时捆绑。要求在运行时能确定对象的实际类型，并绑定正确的函数。晚绑定只对类中虚函数起作用，只对指针和引用有效。

（5）虚函数：

- 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（如下所示），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。

```
class Base {
    public:
    virtual ReturnType FuncName(argument); //虚函数
    ...
};
```

- 若某成员函数在基类中声明为虚函数，当派生类重写覆盖它时(同名，同参数函数)，无论是否声明为虚函数，该成员函数都仍然是虚函数。
- 虚函数表：
    - 对象自身要包含自己实际类型的信息：用虚函数表表示。运行时通过虚

函数表确定对象的实际类型。
    - 虚函数表(VTABLE)：每个包含虚函数的类用于存储虚函数地址的表(虚函数表有唯一性，即使没有重写虚函数)。
    - 每个包含虚函数的类对象中，编译器秘密地放一个指针，称为虚函数指针(vpointer/VPTR)，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VTABLE，记录每个类/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。VPTR是void*类型，占8个字节，打印含虚函数的类的sizeof可以发现比不带多8个字节。
- 内存对齐：`pragma pack(n)`
    - 为啥要内存对齐？
    1. 平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
    2. 性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。
    3. 详见http://www.cnblogs.com/TenosDoIt/p/3590491.html
- 虚函数与构造函数
    - 设置VPTR的工作由构造函数完成。编译器在构造函数开头秘密插入能初始化VPTR的代码。
    - 构造函数不必也不能是虚函数。不能：如果构造函数是虚函数，则创建对象时需要先知道VPTR，而此时VPTR还没初始化；不必：构造函数的作用就是初始化类中成员，调用时明确指定要创建对象的类型，没必要是虚函数。
    - 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。因为基类先初始化，派生类对象此时还未初始化。所以如果允许调用实际对象的虚函数则可能用到未初始化的派生类成员（基类初始化-->对象成员初始化-->构造函数体）

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

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
};

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);
    Base &b = d;
    b.bar();
    b.foo();
    return 0;
}
//结果：Base::foo
//Derived::foo0 普通函数中调用虚函数
//Derived::foo0 直接调用虚函数
```

- 虚函数与析构函数
    - 析构函数能是虚的，且常常是虚的。虚析构函数仍需定义函数体。
    - 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。因为析构和构造顺序相反，最晚派生的析构函数会被最先调用，所以调用到基类的析构函数时，其所调用析构函数的实际对象可能已经被删除了，会造成非法访问。
    - 编程的重要原则就是总将基类的析构函数设置为虚析构函数。

（6）重载、重写覆盖与重写隐藏

- 概念辨析：

1.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。
2.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。
3.重写隐藏(redefining)：派生类重新定义基类中的函数，函数名相同，但是参数不同或者参数相同且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。

	重载(overload)	重写隐藏(redefining)	重写覆盖(override)
作用域	相同(同一个类中，或者均为全局函数)	不同(派生类和基类)	不同(派生类和基类)
函数名	相同	相同	相同
函数参数	不同	相同/不同	相同
返回值	不能仅返回值不同	无要求	相同或协变的
其他要求	—	如果函数参数相同，则基类函数不能为虚函数	基类函数为虚函数

重写覆盖同样会隐藏掉基类的同名函数!

注:对协变的解释： Base和Derived的返回值应该满足：

```
class Base {
public:
    virtual ReturnType1 f(argument){}
};
class Derive : public Base {
    ReturnType2 f(argument)()
};
```

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
};
```

- 重写覆盖和重写隐藏的相同与不同：
    - 相同:

      1.都要求派生类定义的函数与基类同名

      2.都会屏蔽掉基类中的同名函数，即派生类的实例无法调用基类的同名函数
    - 不同：

      1.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是虚函数但函数参数不同。

      2.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。
- 示例：

```
#include<iostream>
using namespace std;
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;}//重写覆盖
};
class Derived2 : public Base{
public:
    void foo(float ) {cout<<"Derived2::foo(float )"<<endl;}//重写隐藏
};
int main(){
    Derived d1;
    Derived d2;
    Base* p1 = &d1;
    Base* p2 = &d2;
    //d1.foo();//派生类定义了带参数的foo所以基类的foo()对实例不可见，编译错误
    //d2.foo();
    p1->foo();//但是虚函数表里有继承自基类的foo()虚函数
    p2->foo();
    d1.foo(3);//重写覆盖
    d2.foo(3.0);//正常调用派生类的foo(float )
    p1->foo(3);//重写覆盖，虚函数表中是派生类的
    p2->foo(3.0);//重写隐藏，虚函数表中是基类的foo(int ),
                 //所以会将3.0转化为int后调用
    return 0;
}
//结果为:
Base::foo()
Base::foo()
Derived1::foo(int )
Derived2::foo(float )
Derived1::foo(int )
Base::foo(int )
```

注意： 上方d1.foo(),d2.foo()很重要。

- const对重写覆盖和重写隐藏的影响：使用const修饰成员函数可能导致重写覆盖失败

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

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

class Base2{
public:
    virtual void g()
        {cout << "Base2::g" << endl;}
};
class Derive2:public Base2{
public:
    void g() //重写覆盖
        {cout << "Derive2::g" << endl;}
    using Base2::g;
};
//结果:
Base1::f
Derive1::f
Base2::g
Derive2::g
```

- override关键字：
    - 重写覆盖要满足的条件很多，很容易写错，可以使用override关键字辅助检查。
    - 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。
    - 如果没有override关键字，但是满足了重写覆盖的各项条件，也能实现重写覆盖。**它只是编译器的一个检查**，正确实现override时，对编译结果没有影响。

- 在之前示例中加上：`class Derived3 : public Base {public:void foo(int ) override {cout<<"Deriv`则重写覆盖正确，与Derived1等价。如果类中改为`void foo(float ) override {};`由于参数不同不符合重写覆盖会编译错误。

（7）final关键字

- 1.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。
  2.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{
    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
};
```

多态与模板

一.纯虚函数

- 虚函数可以声明为纯虚函数:`virtual 返回类型 函数名(形式参数) = 0;`包含纯虚函数的类通常被称为抽象类

```
class A {
public:
    virtual void f() = 0;
    //可在类外定义函数体提供默认实现。
    //派生类通过 A::f() 可调用此纯虚函数
};
A obj; //不准抽象类定义对象，编译不通过
```

- 示例：

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

class Pet {
public:
    virtual void motion()=0;
};
void Pet::motion(){ cout << "Pet motion: " << endl; }
class Dog: public Pet {
public:
    void motion() override {Pet::motion(); cout << "dog run" << endl; }
};
class Bird: public Pet {
public:
    void motion() override {Pet::motion(); cout << "bird fly" << endl; }
};
int main() {
    Pet* p = new Dog; /// 向上类型转换
    p->motion();
    p = new Bird; /// 向上类型转换
    p->motion();
    //p = new Pet; /// 不允许定义抽象类对象
    return 0;
}
//运行结果:
Pet motion:
dog run
Pet motion:
bird fly
```

- 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”
    2. 只能为派生类提供接口

3. 能避免对象切片：保证只有指针和引用能被向上类型转换
4. 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。

- 纯虚析构函数：
    - 纯虚析构函数必须要在类外定义函数体
    - 作用是使基类称为抽象类的同时，让派生类不必实现纯虚析构函数的覆盖也能成为非抽象类
    - 和一般纯虚函数的区别：对于纯虚析构函数而言，即便派生类中不显式实现，编译器也会自动合成默认析构函数。因此，即使派生类不显式覆盖纯虚析构函数，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。

二.向下类型转换

- 基类指针/引用转换成派生类指针/引用，则称为向下类型转换。
- 意义：当用基类指针表示派生类时（向上）会丢失派生类特性，为了重获特性，进行向下类型转换。比如我们可以使用基类指针数组对各种派生类对象进行管理，当具体处理时我们可以将基类指针转换为实际的派生类指针，进而调用派生类专有的接口。
- 需要借助虚函数表进行动态类型检查，以确保基类指针指向的对象也可以被派生类的指针指向。
- **dynamic_cast** ：
    - 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。
    - 使用方法：

      obj_p，obj_r分别是T1类型的指针和引用，T2是T1的派生类

      `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

      注意： 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）**，否则不过编译，而T2不必为多态类型。T1，T2没有继承关系也能通过编译，但是会转换失败
- **static_cast** ：

    - 如果我们**知道正在处理的是哪些类型**，可以使用static_cast来避免这种开销。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（要么自定义，要么是语言语法支持），否则不过编译。**运行时无法确认是否正确转换**。
    - 使用方法：

      obj_p，obj_r分别是T1类型的指针和引用，T2是T1的派生类

      `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。
- 示例：

```
1.
#include <iostream> using namespace std;
class B { public: virtual void f() {} };
class D : public B { public: int i{2018}; };

int main() {
    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;
}
//结果：
static_cast, B*(B) --> D*:OK
D::i=124455624
dynamic_cast, B*(B) --> D*: FAILED

2.
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;
}
//结果：
```

```
static_cast, B*(D) --> D*: OK
D::i=2018
dynamic_cast, B*(D) --> D*: OK
D::i=2018

3.
#include <iostream>
using namespace std;

class Pet { public:  virtual ~Pet() {} };
class Dog : public Pet {
public:         void run() { cout << "dog run" << endl; }
};
class Bird : public Pet {
public:         void fly() { cout << "bird fly" << endl; }
};

void action(Pet* p) {
    auto d = dynamic_cast<Dog*>(p);         /// 向下类型转换
    auto b = dynamic_cast<Bird*>(p);         /// 向下类型转换
    if (d) /// 运行时根据实际类型表现特性
        d->run();
    else if(b)
        b->fly();
}
int main() {
    Pet* p[2];
    p[0] = new Dog; /// 向上类型转换
    p[1] = new Bird; /// 向上类型转换
    for (int i = 0; i < 2; ++i) {
        action(p[i]);
    }
    return 0;
}
//结果:
dog run
bird fly
```

- dynamic_cast与static_cast都可以完成向下类型转换，而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,所以一般用dynamic_cast。
- dynamic_cast与static_cast也能对指针或引用进行向上类型转换，然而较少使用，因为向上转换支持式子转换，static_cast也能对不同对象类型进行转换。

三.多态

（1）多态梗概

- 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（表现），这个现象被称为“多态”。
- 当利用基类指针/引用调用函数时，虚函数在运行时确定执行哪个版本，取决于引用或指针对象的真实类型，非虚函数在编译时绑定；当利用类的对象直接调用函数时，无论什么函数，均在编译时绑定。
- **产生多态需要：继承+虚函数+指针或引用**
- 示例：

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

class Animal{
public:
void action() {
    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() {
Fish fish;
Bird bird;
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;
}
//结果:
Fish cannot speak ...
Fish swimming
Bird singing
Bird flying
Fish cannot speak ...
Fish swimming
Bird singing
```

```
Bird flying
```

（2）函数模板与类模板

0.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，即声明和定义不可分开，因为如果分开编译，则编译模板库的源代码时不知道应该生成哪些模板实例；而编译模板实例时，又没有模板库的源代码来生成，所以产生链接错误。

1. 函数模板：

（1）普通函数模板:

- `template <typename T> ReturnType Func(Args);` 其中的typename换成class效果一样；多参数：`template<typename T0, typename T1>`
- 如`template <typename T>` `T sum(T a, T b) { return a + b; }` 调用类型应该满足函数要求，如果`sum(9,2.1)`因为类型不同所以无法通过编译。可以手动指定:`sum<int>(9,2.1)`
- 函数模板支持自定义类型，但是应该实现相应的运算符重载，比如函数模板中用到了大于号，则在自定义类型中，如一个class应该针对想实现的比较目标重载大于号。

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

template<class T>
void sort(T* data, int len)
{
    for(int i = 0; i < len; i++){ //选择排序
        for(int j = i + 1; j < len; j++) {
            if(data[i] > data[j])
                std::swap(data[i], data[j]); //交换元素位置
        }
    }
}

class MyInt
{
public:
    int data;
    MyInt(int val): data(val) {};
    bool operator>(const MyInt& b){ //用于sort
        return data > b.data;
    }
    friend std::ostream&  //用于output
            operator<<(std::ostream& out, const MyInt& obj){
        out << obj.data;
        return out;
    }
};

int main()
{
    MyInt arr_c[] = {3, 2, 4, 1, 5};
    sort(arr_c, 5);
    return 0;
}
```

（2）成员函数模板:

1.
```
class normal_class {
public:
    int value;
    template<typename T> void set(T const& v) {
        value = int(v);
    }/// 在类内定义
    template<typename T> T get();
};
template<typename T>/// 在类外定义
T normal_class::get() {
    return T(value);
}
```

2.模板类的成员函数加以额外的参数

```
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>
T1 A<T0>::get(){ return T1(value);}
/// 类外定义，将T0转换为T1返回，此处T0是为了完整定义一个类，必须加
```

2.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”,如

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

template<typename T>
void A<T>::print() { cout << data << endl; }

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

- 模板参数：类型参数：使用typename或class标记；非类型参数：整数，枚举，指针（指向对象或函数），引用（引用对象或引用函数）。无符号整数(unsigned)比较常用。

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

array<char, 10> array0;
//此外模板参数还可以使用常量，但是不能用变量
```

3.函数模板与类模板特化

- 有时，有些类型并不合适，则需要对模板在某种情况下的具体类型进行特殊处理，这称为“模板特化”。

（1）函数模板

- 方法：以`template <typename T> T sum(T a, T b)`为例
    1. 在函数名后用<>括号括起具体类型：`template<> char* sum<char*>(char* a, char* b)`
    2. 由编译器推导出具体类型，函数名为普通形式`template<> char* sum(char* a, char* b)`

- ```
  template<class T>
  T div2(const T& val)
  {
      cout << "using template" << endl;
      return val / 2;
  }

  template<>
  int div2(const int& val) //函数模板特化
  {
      cout << "better solution!" << endl;
      return val >> 1; //右移取代除以2
  }
  ```

- **注意**：对于函数模板，如果有多个模板参数，则特化时必须提供所有参数的特例类型，**不能部分特化。但可以用重载来替代部分特化**。

```
template<class T, class A>
T sum(const A& val1, const A& val2)
{
    cout << "using template" << endl;
    return T(val1 + val2);
}

template<class A>
int sum(const A& val1, const A& val2)
{   //不是部分特化，而是重载函数
    cout << "overload" << endl;
    return int(val1 + val2);
}
```

- 函数模板重载解析顺序：
  类型匹配的普通函数->基础函数模板->全特化函数模板
  - 如果有普通函数且类型匹配，则直接选中，重载解析结束
  - 如果没有类型匹配的普通函数，则选择最合适的基础模板
  - 如果选中的基础模板有全特化版本且类型匹配，则选择全特化版本，否则使用基础模板
  - 示例1：
```

```
template<class T> void f(T) {
//func1为基础模板
cout<< “full template” <<endl;};
template<class T> void f(T*) {
//func2为func1的重载，仍是基础模板
cout<< “full template -> overload template” <<endl;};
template<> void f(char*) {
//func3为func2的特化版本(T特化为char)
cout<< “overload template -> specialized” <<endl;};
int main() {
    char *p;
    f(p);
    return 0;
}
//实则调用的func3
//因为没找到类型匹配的普通函数，然后找到了func2这一基础函数模板，
//发现其全特化版本func3，且类型匹配
```

- 示例2：

```
template<class T> void f(T) {
//func1为基础模板
cout<< “full template” <<endl;};
template<> void f(char*) {
//仅仅改换了func3和2的顺序
//现在，func3为func1的特化版本(T特化为char*)
cout<< “full template -> specialized” <<endl;};
template<class T> void f(T*) {
//func2为func1的重载,仍是基础模板
cout<< “full template -> overload template” <<endl;}
int main() {
    char *p;
    f(p);
    return 0;
}
//主函数调用的是func2
//先从基础模板func1和func2中选择更匹配的模板实例，
//func2参数类型更匹配，因此优先选中。
//函数模板func2无特化版本，因此直接调用模板func2。
```

（2）类模板

- 对于以下模板`template<typename T1, typename T2> class A { ... };`与函数模板类似，可以进行全部特化：`template<> class A<int, int> { ... };`

```
template<typename T1, typename T2> class Sum { //类模板
public:
Sum(T1 a, T2 b) {cout << "Sum general: " << a+b << endl;}
};
template<> class Sum<int, int> { //类模板全部特化
public:
Sum(int a, int b) {cout << "Sum specific: " << a+b << endl;}
};
```

- 类模板允许部分特化：
  如通用模板为：
  `template<typename T1, typename T2> class A { ... };`
  部分特化：第二个类型指定为int
  `template<typename T1> class A<T1, int> {...};`
  对比全部特化：指定所有类型
  `template<> class A<int, int> { ... };`

（3）小结

1. **类模板可以部分特化或者全部特化**，编译器会根据调用时的类型参数自动选择合适的模板类。
2. **函数模板只能全部特化**，但可以通过**重载**代替部分特化的实现。编译器在编译阶段决定使用特化函数或者标准模板函数。
3. 函数模板的**全特化版本的匹配优先级可能低于重载的非特化基础函数模板**，因此最好不要使用全特化函数模板而直接使用重载函数。

4.静多态和动多态

- 模板的关联在编译器处理，称为静多态，其往往和函数重载同时使用，能够高效省去函数调用，在编译后代码会增多
- 基于继承和虚函数的多态在运行期处理，称为动多态。其运行时方便灵活，必须有继承，存在函数的调用。

（3）OOP核心思想：

1. 数据抽象：类的接口与实现分离
2. 继承：建立相关类型的层次关系（基类与派生类）
3. 动态绑定：统一使用基类指针，实现多态行为

命名空间,stl与字符串处理

一.命名空间

- 为了避免在大规模程序的设计中，以及在程序员使用各种各样的C++库时，标识符的命名发生冲突，标准C++引入了关键字namespace（命名空间），可以更好地控制标识符的作用域。

```
namespace A {
        int x, y;
    }
A::x = 3;
A::y = 6;

简化使用:
using namespace A;
x = 3; y = 6;

使用部分:
using A::x;
x = 3; A::y = 6;

任何情况下，都不应出现命名冲突
```

二.STL拾遗

- 标准模板库（英文：Standard Template Library，缩写：STL），是一个高效的C++软件库，它被容纳于C++ 标准程序库C++ Standard Library中。**其中包含4个组件，分别为算法、容器、函数、迭代器**。
- 命名空间为std，一般用std::name来使用其中对象。也可以用using namespace std以引入（**不推荐在大型工程中使用，易污染命名空间**）

1.pair

- `auto t = std::make_pair("abc", 7.8);`自动推导，比`pair<string,double> t`更方便
- 支持比较运算符，先比较first，后second

2.tuple

- 创建tuple四种方法:

1. `tuple<int, bool, string, Person> t1 = tuple<int, bool, string, Person>(11, true, "ok", P`
   Perosn是一个类，`auto t1`也行；
2. `tuple<int, bool, string, Person> t2 = std::make_tuple(11, true, "ok", Person("ok", 11));`
3. `tuple<int, bool, string, Person> t3 = std::forward_as_tuple(11, true, "ok", Person("ok",`
   返回右值引用的元组
4. ```
   int i = 11;
   bool b = true;
   string s = "ok";
   Person p("ok", 11);
   tuple<int, bool, string, Person> t7 = std::tie(i, b, s, p);
   //返回左值引用的元组
   ```

- 获取元素的方法

```
auto t = std::make_tuple(“abc”, 7.8, 123, ‘3’);
auto v0 = std::get<0>(t);
auto v1 = std::get<1>(t);
//尖括号里不能是变量，必须在编译时确定
```

- 也可以用于多返回值的传递

```
#include <tuple>
std::tuple<int, double> f(int x){
    return std::make_tuple(x, double(x)/2);
}
int main() {
    int xval;
    double half_x;
    std::tie(xval, half_x) = f(7);
    //相当于int xval = x;double half_x=double(x/2);
    return 0;
}
```

3.vector

- `int distance = iter1 - iter2;`获取迭代器间距离
```

- `for(auto & x:vec)`等价于`for(vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)`
- 迭代器的失效：
    - 调用insert/erase后，所修改位置之后的所有迭代器失效。（原先的内存空间存储的元素被改变）其中，erase后，会返回下一个元素的迭代器（可以是end）。
    - 调用push_back等修改vector大小的方法时，可能会使所有迭代器失效；因为push_back到预置内存满后，会发生内存翻倍，然后整体复制到新内存，因而所有地址改变，此时所有迭代器失效。
    - 绝对安全的准则:在修改过容器后，不使用之前的迭代器

4.list

- 本质为双向链表,用push_front插入前端，push_back插入尾部，find(itr1,itr2,element)在两个迭代器范围间查找element并返回迭代器,insert(itr,element)在迭代器位置插入element。
- 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，且不会引起迭代器失效（除了指向被删除元素的迭代器

5.set/multiset

- 前者不可重复，后者可重复

6.map

- 元素为pair<Key,T>
- map中元素key必须互为不同，但是T可以随意相同，所以不是双射
- 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；也可使用insert函数进行插入。

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

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

7.序列容器和关联容器小结

- 1. 序列容器：vector、list
  2. 关联容器：set、map
  3. 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）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

8.string

- 构造方式
  `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
- 转换为c风格字符串:`str.c_str()`返回值为const char*

- 三种输入方式
    - Mike William
      Andy William
      #
    - 读取可见字符直到遇到空格`cin >> firstname;//Mike`
    - 读一行 `getline(cin, fullname);//Mike William`
    - 读到指定分隔符为止（可以读入换行符）
      `getline(cin, fullnames, '#');//"Mike William\nAndy William\n"`
- 数值类型字符串化

```
to_string(1)        //"1"
to_string(3.14)        //"3.14"
to_string(3.1415926)//"3.141593" 注意精度损失
to_string(1+2+3)        //"6"
```

- 字符串转数值类型

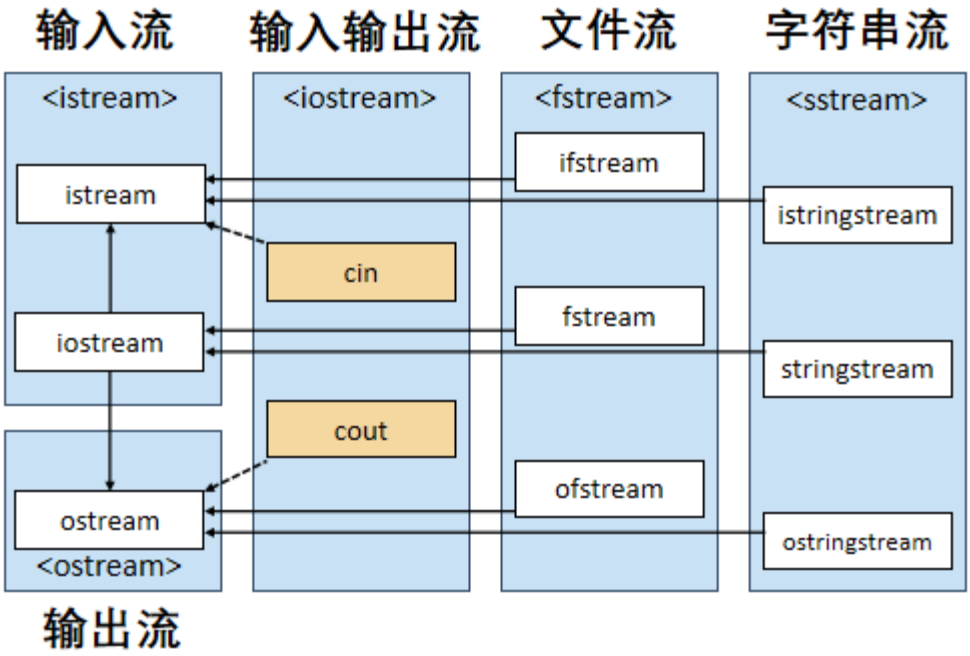
```
int a = stoi("2001")                  //a=2001
std::string::size_type sz;  //代表长度的类型（类似int），是无符号整数
int b = stoi("50 cats", &sz)  //b=50 sz=2 代表读入长度
int c = stoi("40c3", nullptr, 16) //c=0x40c3 十六进制
int d = stoi("0x7f", nullptr, 0)  //d=0x7f 自动检查进制

double e = stod("34.5")          //e=34.5
float e = stof("1.233234234")   //同理
```

iostream输入输出流

- 类间关系图，其中ostream，istream是最顶层基类。

STL输入输出流



（1）ostream

```
class ostream//简易自实现版
{
public:
    ostream& operator<<(char c)
    {
        printf("%c", c);
        return *this;
    }
    ostream& operator<<(const char* str)
    {
        printf("%s", str);
        return *this;
    }
}cout;
int main()
{
    cout << "hello" << ' '
        << "world";
    return 0;
}
```

- **实现原理**：<<运算符为左结合
  先执行cout << "hello" 调用第二个函数 返回c1（cout的引用）；再执行c1 << ' ' 调用第一个函数 返回c2 (cout的引用）；最后执行c2 << "world" 调用第二个函数
- 重载流运算符时返回的是引用，是为了避免复制。实际中，ostream中`ostream(const ostream&) = delete;ostream(ostream&& x);`以禁止复制，只使用cout一个全局对象，可以移动。是为了减少复制开销，且多个对象无法同步输出状态。

（2）流操纵算子

- 示例：

```
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 << setprecision(2) << 3.1415926 << endl;
                //输出精度设置为2 -> 3.2
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
```

这都是流操纵算子：fixed、scientific、defaultfloat、oct、hex、dec实现方式和endl一样，这是标准中定义的。

- 按道理来讲，在同一编译器内，流操纵算子实现方式就是两种：
  一种是不带参数的（以endl为代表，规范有定义）；一种是带参数的（以setprecision为代表，规范没有定义，不同编译器实现不同）
- 实现方式：
  1.以serprecision为例

```
class setprecision//辅助类
{
private:
    int precision;
public:
    setprecision(int p) : precision(p) {}
    friend class ostream;
};
// setprecision(2) 是一个类的对象

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

2.以endl为例

- C++标准中endl的声明：`ostream& endl(ostream& os);`可见endl是一个函数
- ```
  //先输出'\n',再清空缓冲区
  ostream& endl(ostream& os){
      os,put('\n);
      os.flush();
      return os;
  }
  //因此可以调用endl(cout)
  ```
- 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写入
- 同时，endl是函数也要作为流操纵算子，一种实现方法：

```
ostream& operator<<(ostream& (*fn)(ostream&)) {
        //流运算符重载，函数指针作为参数
    return (*fn)(*this);
}
```
```

（3）fstream

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

int main() {
ifstream ifs("input.txt");
//ifstream ifs;
//ifs.open("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
    int c = ifs.peek();                //检查下一个字符，但不读取
    if (c == EOF) break;
    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;
}
```

- 其他操作：
    - ifstream是istream的子类，故getline(ifs, str)仍然有效
    - get() 读取一个字符
    - ignore(int n=1, int delim=EOF) 丢弃n个字符，或者直至遇到delim分隔符
    - peek() 查看下一个字符
    - putback(char c) 返还一个字符
    - unget() 返还一个字符

（4）stringstream

- stringstream是iostream的子类，iostream是istream和ostream的子类，则stringstream实现了输入输出流双方的接口

- 它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中读出数据；**注意buffer并非未读取的内容**
- 构造方式
    - stringstream ss; //空字符串流
    - stringstream ss(str); //以字符串初始化流
- 示例：

```
#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;
}
```

- 可以连接字符串；可以将字符串转换为其他类型的数据；配合流操作算子，可以达到格式化输出效果
- ss.str()：返回一个string对象，内容为stringstream的buffer

```
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"
    return 0;
}
```

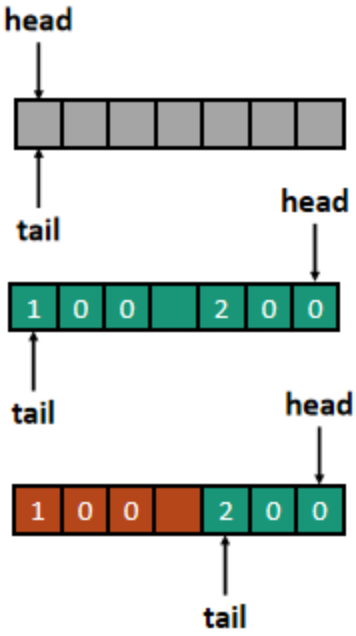
获取stringstream的buffer

```
int main()
{
    stringstream ss;

    ss << "100 200";
    cout << ss.str() << endl;
     //"100 200"

    int a, b;
    ss >> a;  // a = 100
    cout << ss.str() << endl;
    //"100 200"

    ss >> b; // b = 200
    return 0;
}
```



head和tail间代表未读取的部分

head 是等待读入的最后位置(还没有读过的在开头；已经读过的在尾巴）；head到tail表示当前等待读入的缓冲区，所以head在后面。

- 利用stringstream实现一个类型转换函数。

```
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;
}
```

当到了end-of-file的位置，此时stringstream会为其**设置一个eofbit的标记位**，标记其为已经到达eof。 当stringstream设置了eofbit，任何读取eof的操作都会失败。同时，读取失败会**设置failbit的标记位**，标记为失败状态。所以后面的操作都失败了。clear函数的作用就是**清除掉所有的**

error state(状态位)，所以在代码前面加一个ss.clear()即可达到预期结果。一般清空缓冲区和状态位同时使用。

正则表达式

- regex_search只能用于找第一个，迭代找多次的方法：

1. `for (sregex_iterator it(test_str.begin(),test_str.end(),r), end_it;it != end_it;++it) {c`
   test_str是被搜索字符串，r是正则表达式
2. while(regex_search(test_Str,sm,r)){}

其余细节见L12-ppt/CSDN

函数对象

- sort里的预置函数是啥？
  `sort(arr, arr+5, greater<int>())`其中greater是一个模板类，greater是int实例化的类,greater()是该类的一个对象；其表现的像一个函数：`auto func = greater<int>();cout << func(2, 1) << endl;`输出为True
- 函数对象的实现：
  1. 需要重载operator()运算符
  2. 并且该函数需要是public访问权限

```
template<class T>
class Greater {
public:
    bool operator()(const T &a, const T &b) const        {
        return a > b;
    }//注意三个const，排序中，comp不可修改数据，comp也不能修改自身
};
int main(){
    auto func = Greater<int>();
    cout << func(2, 1) << endl;        //True
    cout << func(1, 1) << endl;        //False
    cout << func(1, 2) << endl;        //False
    return 0;
}
```

- sort函

数

：`template <class Iterator, class Compare>void sort (Iterator first, Iterator last,Compa`

其中comp可以传入自定义的bool类型函数的函数指针(`bool(*)(int,int)`)，也可以传函数对象(greater)，其可接收2种值

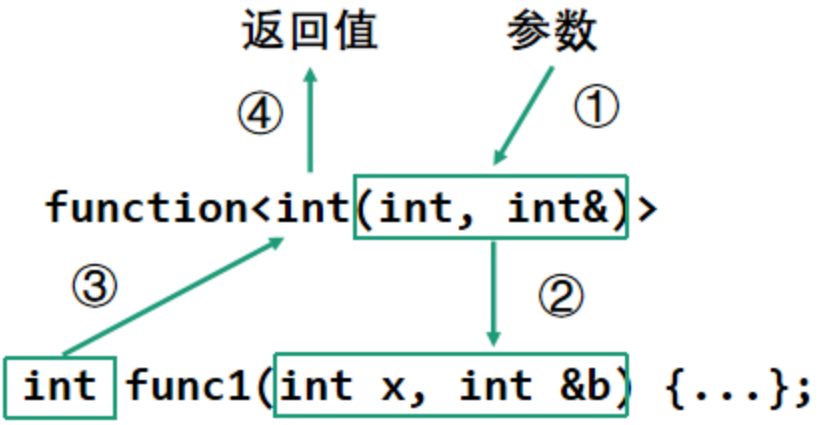
- 至此，自定义类的大小比较可以由三种方法：
    1. 类内重载小于运算符
    2. 定义比较函数
    3. 定义比较函数对象（一个新类
- 将函数对象和函数指针统一：std::function类

`function<ReturnType(param list)> paramName = 函数指针/函数对象`，在库里

示例：

```
#include <iostream>
#include <fstream>
#include <functional>
#include <string>
using namespace std;
string readFromScreen()
{
    string input; getline(cin, input);
    return input;
}
class ReadFromFile
{
public:
    string operator()(){
        string input;
        getline(ifstream("input.txt"), input);
        return input;
    }
};
string calculateAdd(string x)
{
    return x;
}
void writeToScreen(string x)
{
    cout << x;
}
void process(
        function<string()> read,
        function<string(string)> calculate,
        function<void(string)> write)
{
    string data = read();
    string output = calculate(data);
    write(output);
}
int main()
{
    function<string()> readArr[] =
            {readFromScreen, ReadFromFile()};
    process(readArr[0], calculateAdd, writeToScreen);
    process(readArr[1], calculateAdd, writeToScreen);
    return 0;
}
```

函数指针牵引出的“严格”、“宽松”问题



准则： function参数 比 实际参数 更严格
function返回值 比 实际返回值 更宽松

- 准则存在的原因：
    1. 所有能被function接受的参数，都应被实际函数接受
    2. 所有可能的实际函数的返回值，都可能被function返回

	左值	常量左值	右值	常量右值
	lvalue	const lvalue	rvalue	const rvalue
int	√	√	√	√
int&	√			
const int&	√	√	√	√
int&&			√	
const int&&			√	√

一般不使用const int&&，无实际意义

同时可见const int&和int一样宽松

```
int func1(const int &x, int &b) {...};
function<______> pf1 = func1;

A.int(int, int)
B.int(int, int&)
C.int(int&, int&)
D.int(const int&, int&)
E.int(const int, int)
F.int&(int, int&)
//B,C,D可通过编译


class Func2{
public:
    int& operator()(int &&b) const {...}
};
function<______> pf2 = Func2();

A.int&(int&&)
B.int(int&&)
C.int&(int&)
D.int(int&)
E.int&(int)
//A,B,E可通过编译
```

注意： 上方E可以的原因是,虽然int无法直接传给Func2(),但实际上， pf2的实现方式是拷贝一份参数， 然后将该参数的右值传给Func2()

智能指针

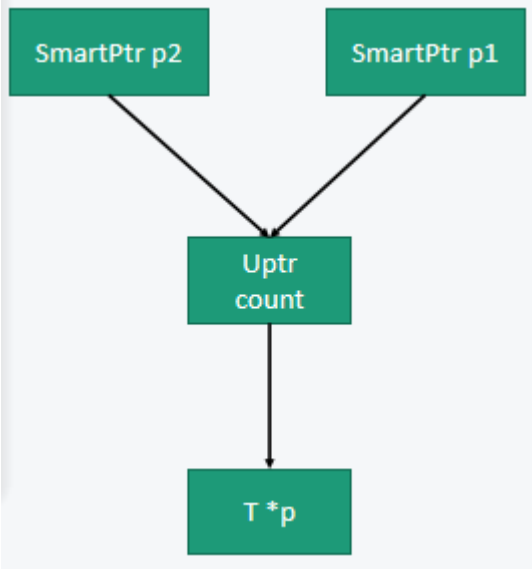
一.shared_ptr(<memory>库)

- 构造：
    - `shared_ptr<int> p1(new int(1));`
    - `shared_ptr<MyClass> p2 = make_shared<MyClass>(构造函数初始化值);`
    - `shared_ptr<MyClass> p3 = p2;`
    - `shared_ptr<int> p4;` //空指针
- 访问：
    - `int x = *p1;` //从指针访问对象
    - `int y = p2->val;` //访问成员变量
- 销毁：p2和p3指向同一对象， 当两者均出作用域才会被销毁

- 引用计数：

```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    shared_ptr<int> p1(new int(4));
    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
}
```

- 实现自己的智能指针：



```
#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) //减少自身原本所指(将指向新的)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)); //不能再ptr3(pi)；原因见后方"注意"
    ptr3 =  ptr2; //注意赋值运算
    cout << *ptr1 << endl; //输出2
    *ptr1 = 20;
    cout << *ptr2 << endl; //输出20
return 0;
}
```

- **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int类型指针(和dynamic_cast类似，动态类型检查）
    - 数组管理方面：shared_ptr可以使用*，->，但是不能用[]访问元素(只能用get()获取裸指针后配合*，->获取元素)，unique_ptr相反；
- **注意**：
    - 不能使用同一裸指针初始化多个智能指针
      如
      ：`int* p = new int(); shared_ptr<int> p1(p); shared_ptr<int> p2(p);`
      会产生多个辅助指针！
    - 不能直接使用智能指针维护对象数组，智能指针的所有删除方式都是`delete p;`但是数组需要`delete[] p;`来删除
      修改方法及访问方式：3种删除方式

```
void ArrayDeleter(TestClass *array) {
    delete [] array;
}
int main()
{
    const size_t size = 100;

    //使用函数指定
    std::shared_ptr<TestClass []> spFunc(new TestClass[size], ArrayDeleter);

    //使用lambda表达式
    std::shared_ptr<TestClass []> spLambda(new TestClass[size], [] (TestClass * tc) {delete []

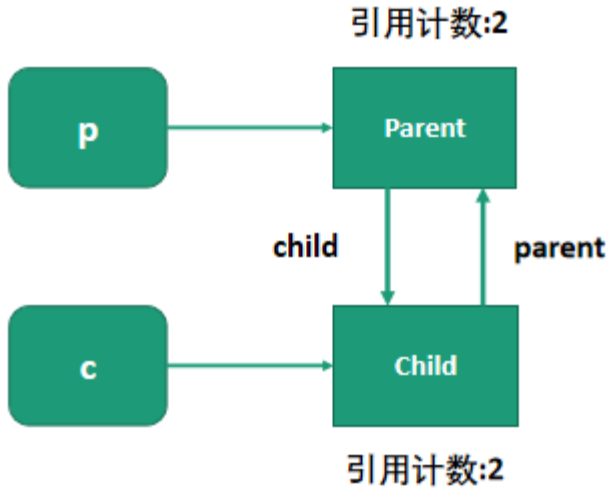
    {
    //使用默认删除
    std::shared_ptr<TestClass []> spDefaultDeleter(new TestClass[size], std::default_delete<Te
    (spDefaultDeleter.get())->b = 10;
    }

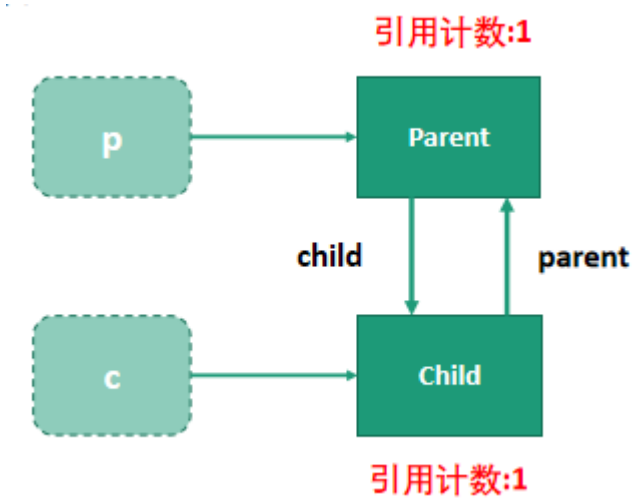
    //访问成员
    //spFunc[0] = 10; //error
    (spFunc.get())->b = 10;
    (spFunc.get() + 1)->b = 20;
    std::cout << "the first element: " << (spFunc.get())->b << "\n";
    std::cout << "the second element: " << (spFunc.get() + 1)->b << "\n";

    return 0;
}
```

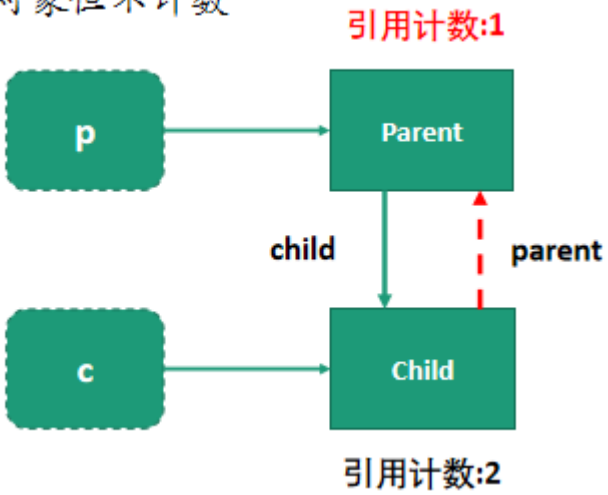
二.weak_ptr

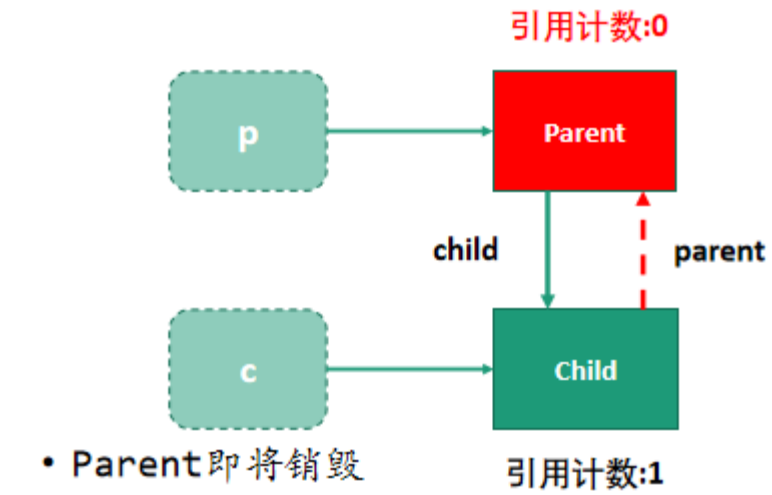
- shared_ptr存在问题：循环指向会导致内存泄漏(最后因为引用计数都为1而都不删除)

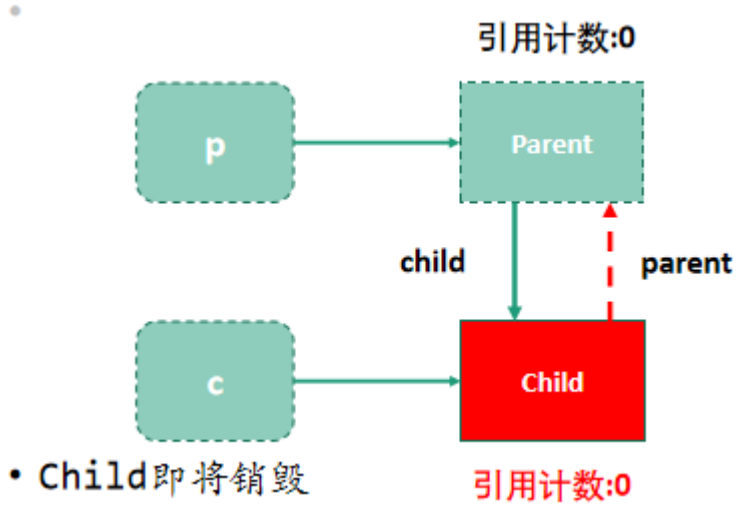




- 将其中一个类的指针修改为弱引用weak_ptr，其不会使被指对象获得引用计数
  - 指向对象但不计数







- 使用方法：
  - 创建：`shared_ptr<int> sp(new int(3));weak_ptr<int> wp1 = sp;`
  - 用法：

    `wp.use_count()` //获取引用计数

    `wp.reset()` //清除指针

    `wp.expired()` //检查对象是否无效

    `sp = wp.lock()` //从弱引用获得一个智能指针

三.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;
}
```