多态与模板

多态与模板

By 曹菡雯（计 03）、赵晨阳（计 06）、罗华坤（软 02）、李晨宇（软 13）

一、抽象类与纯虚函数

1.1 定义

虚函数还可以进一步声明为纯虚函数，只要包含有一个纯虚函数的类，即为“抽象类”。语法如下：

```
virtual 返回类型 函数名(形式参数) = 0;
```

抽象类不允许定义对象。（new 也算是定义对象。）

定义基类为抽象类的主要用途是为派生类规定共性“接口”。能避免赋值型对象切片：保证只有指针和引用能被向上类型转换。（因为抽象类无法定义对象，但是可以定义指针和引用）

例子

```
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() 调用
};
A obj; /// 不准抽象类定义对象！编译不通过！
```

```
#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; }
};//重写覆盖后，也可以通过如 d1.Base::foo();来调用基类函数。

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;
}
Output:Pet motion:
dog run
Pet motion:
bird fly
```

注意到在 L8 3.5 中有提到过，重写覆盖 override 与重写隐藏 redefining 相似，也可以通过如 d1.Base::foo();来调用基类函数。此例中虽然发生了重写覆盖，但是通过直接调用 Pet::motion();来调用了基类函数。

1.2 抽象类继承

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（随后解释），必须实现所有纯虚函数，否则继承出的类也是抽象类。

1.3 纯虚析构函数

析构函数也可以是纯虚函数，**纯虚析构函数仍然需要函数体**。

目的：使基类成为抽象类，不能创建基类的对象。如果有其他函数是纯虚函数，则析构函数无论是否为纯虚的，基类均为抽象类。

1.4 纯虚析构函数和其他纯虚函数的区别

一般的纯虚函数被派生类重写覆盖之前仍是纯虚函数。如果派生类不覆盖纯虚函数，那么派生类也是抽象类。

但对于对于纯虚析构函数而言，即便派生类中不显式实现析构函数，编译器也会自动合成默认析构函数，完成重写覆盖，使得派生类不是抽象类。故而，即使派生类不显式覆盖纯虚析构函数，只要派生类完全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。

例一

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

class Base {
```

```
public:
    virtual void func()=0;
};
class Derive1: public Base {}; //Derive1 仍为抽象类
class Derive2: public Base {
public:
    void func() {
        cout << "Derive2::func" << endl;
}
};

int main()
{
    // Derive1 d1; //编译错误，Derive1 仍为抽象类
    Derive2 d2;
    d2.func();
    return 0;
}
Output：
Derive2::func
```

例二

```
#include <iostream>
using namespace std;
class Base{
public:
    virtual ~Base()=0;
};
Base::~Base() {cout<<"Base destroyed"<<endl;}//纯虚函数不能在类内写函数体，
需要挪到类外

class Derive1: public Base {};
class Derive2: public Base {
public:
    virtual ~Derive2() {cout<<"Derive2 destroyed"<<endl;} };
int main()
{
Base* p1 = new Derive1;
Base* p2 = new Derive2;
delete p1;
```

```
cout << “------” << endl;
delete p2;
return 0;
}
Output：
Base destroyed
------
Derive2 destroyed
Base destroyed
```

例三

下面关于虚函数和抽象类的描述，正确的是：

A 通过类的指针或引用调用类内函数，无论是否使用虚函数，都可实现晚绑定(运行时绑定)

B 抽象类的派生类必须显式实现抽象类中的所有纯虚函数（包括纯虚析构函数），否则会出现编译错误

C 抽象类不允许定义对象

D 抽象类的成员函数都是纯虚函数

A.晚捆绑依赖于虚函数表与虚函数，只对类中虚函数起作用，并且只对基类指针和引用起作用。

D.一个纯虚函数就会使得一个类成为纯虚函数

二、向下类型转换

2.1 定义

指向派生类对象的基类指针/引用转换成派生类指针/引用，则称为向下类型转换。（类层次中向下移动）

当我们用基类指针表示各种派生类时(向上类型转换),保留了他们的共性，但是丢失了他们的特性。如果此时要表现特性，则可以使用向下类型转换。

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。

但是需要注意到，这里是说用基类指针可以管理派生类，故而可以把这个指针向下转换，生成实际使用的派生类指针。但是，如果是基类指针管理基类对象，也就是基类指针本来就指向基类对象时，基类对象本身就没有派生类多出的数据和服务，还要把这个基类指针向下转换为派生类指针，要么是错的，要么是危险的。

2.2 转换方式

2.2.1 安全向下类型转换（基类向派生类的转换）

C++提供了一个特殊的显式类型转换，称为 dynamic_cast，是一种安全的向下类型转换。使用 dynamic_cast 的对象必须有虚函数，因为它使用了存储在虚函数表中的信息判断实际的类型。

所谓对象必须有虚函数，实际上是指有继承关系时，基类和派生类都必须有虚函数。基类有虚函数，是多态的，那么派生类必然也是多态的。重写覆盖是用虚函数覆盖虚函数，仍然可以用 Base::的方式调用基类虚函数，不破坏多态性。而重写隐藏更不会破坏其多态性。

语法

T2 是 T1 的派生类，obj_p，obj_r 分别是 T1 类型的指针和引用，二者都指向一个派生类 T2 的对象。

```
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，运行失败时抛出 bad_cast 异常
```

2.2.2 快速向下类型转换

如果我们知道目标的操作是安全的，可以使用 static_cast 来加快速度。

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。因为快速，static_cast 运行时无法确认是否正确转换。

语法

T2 是 T1 的派生类，obj_p，obj_r 分别是 T1 类型的指针和引用，二者都指向一个派生类 T2 的对象。

```
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：如果基类指针本来就指向基类对象，向下类型转换就是不合法的。此时不保证指向目标是 T2 对象，可能导致非法内存访问。

例子一：第一类不合法转换

```
#include <iostream>
using namespace std;
class B {
```

```
    public:
    virtual void f() {}
};
class D : public B {
    public:
    int i{2018};
};
int main() {
    D d; B b;
    //D d1 = static_cast<D>(b); ///未定义类型转换方式,注意这个直接是对象的转换
而不是指针的转换
    //    D d2 = dynamic_cast<D>(b); ///这是合法的，dynamic_cast 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,让 D 类型指针 pd1 指
向了 B 类型的对象 b

    if (pd1 != nullptr){
        cout << "static_cast, B*(B) --> D*: OK" << endl;
        cout << "D::i=" << pd1->i <<endl;
    } /// 但是不安全：pd1 访问 D 中成员 i 时，可能是非法访问

    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;

    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;
    }

Output：
static_cast, B*(B) --> D*: OK
D::i=4817112
dynamic_cast, B*(B) --> D*: FAILED
static_cast, B*(D) --> D*: OK
D::i=2018
dynamic_cast, B*(D) --> D*: OK
D::i=2018
```

注意到前文提过危险转换的情况，如果基类指针本来就指向基类对象，向下类型转换就是不合法的。从而我们观察到，使用 dynamic_cast 将&b 转为 D 的指针是错误的，返回了空指针。而 static_cast 虽然实现了转换，但是输出的完全是非法内存。且在大多数编译器上，static_cast 这样操作会 RE。

另一方面，我们现在讨论的的都是指针转换，而对象的直接转换是类型转换，故而不加其他定义时用 static_cast 和 dynamic_cast 都是不安全的。

例子二：第二类不合法转换

```
#pragma once
#include <string>
#include <iostream>

class Animal {
private:
    std::string name, type;
    Animal(const std::string &_name, const std::string &_type): name(_name), type(_t
ype) {}
};
class Dog: public Animal {
public:
    Dog(const std::string &_name): Animal(_name, "dog") {};
};
class Bird: public Animal {
public:
    Bird(const std::string &_name): Animal(_name, "bird") {};
};
void action(Animal* animal, std::vector<Dog> & dogzone, std::vector<Bird> & birdz
one){
    Dog* point1 =dynamic_cast<Dog*>(animal);
```

```
    Bird* point2 =dynamic_cast<Bird*>(animal);
    if(point1!=nullptr){
        dogzone.push_back(std::move(*point1));
    }
    else {
       birdzone.push_back(std::move(*point2));
    }
}
```

这个题放上了伪代码，给出了第二类非法转换的例子。一个基类派 D 生出了多个派生类 A,B,C。比如基类指针 d 指向了派生类 A 对象 a，但是如果我们把 d 向下类型转换为 B 类指针，也是非法转换。从这个角度可以看出，在上例中，Animal 类的 type 是 private，没有接口是没法访问的。我们本来需要根据 type 来决定向下转换的类型，但是访问不了。于是我们直接向下类型转换，直接通过是否转为 nullptr 来检测是否转换正确。

2.2.3 总结

dynamic_cast 与 static_cast

相同点：都可完成向下类型转换。

不同点：static_cast 在编译时静态执行向下类型转换。而 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。

一般使用 dynamic_cast 进行向下类型转换。

指针转换的重要原则：

清楚指针所指向的真正对象

1）指针或引用的向上转换总是安全的；

2）向下转换时用 dynamic_cast，安全检查；

3）避免对象之间的转换。尽可能用指针、引用进行转换。这样可以通过验证虚函数表判断转换是否安全。

引例——历史遗留问题，关于传参的本质

（下面这个程序有很大问题）

```
#include <iostream>
using namespace std;
void swap(int *a, int *b)
{
 int tmp = *a;
 *a = *b;
 *b = tmp;
}
int main() {
 int x=1,y(2);
 swap(x,y);
 cout<<x<<endl;
```

```
return 0;
}
Output:
2
```

我们试图运行该程序，很显然，输出了 2。我们似乎已经察觉到了地址和指针之间的美妙关系，也就是直接输出指针的值就是指针指向的地址。我们隐约察觉到，这里 int*a 貌似会指向 x 的地址，所以我们试图把地址打印出来。

（下面这个程序也有很大问题）

```
#include <iostream>
using namespace std;
void swap(int *a, int *b)
{   cout<<a<<endl;
 int tmp = *a;
 *a = *b;
 *b = tmp;
}
int main() {
 int x=1,y(2);
 swap(x,y);
 cout<<&x<<endl;
 cout<<x<<endl;
return 0;
}
Output：
0x7ffebcbe46e8
2
```

看上去很对，但是，似乎少了点什么？我们 swap 函数里对 int*a 的输出呢？这些程序有什么问题？

原因是，在 std 空间里本身就包含有 swap 函数，这么写 swap 根本无法调用我们自己定义的 swap，故而不可使用 std 空间。（这也告诫了我们少用 std 空间，或者把函数名字起的复杂点）

我们禁用了 std 空间后

```
#include <iostream>
//using namespace std;
void swap(int *a, int *b)
{
 int tmp = *a;
 *a = *b;
 *b = tmp;
}
int main() {
 int x=1,y(2);
```

```
 swap(x,y);
 std::cout<<x<<std::endl;
return 0;
}
Output：
main.cpp:11:2: error: no matching function for call to 'swap'
 swap(x,y);
 ^~~~
main.cpp:3:6: note: candidate function not viable: no known conversion from 'int' to 'i
nt *' for 1st argument; take the address of the argument with &
void swap(int *a, int *b)
     ^
1 error generated.
```

我们终于发现了问题，在调用参数的时候，我们根本无法让一个 int x 传入 int*a。首先，传参数的过程可以理解为赋值语句：比如我们函数定义 void swap(int *a, int *b);那么调用时，swap(x,y);需要 int*a=x，int*b=y;这两个赋值语句都没法实现。但是我们调用 swap(&x,&y)时，实际上是 int*a=&x，int*b=&y，这两个赋值都可以实现，且&x 和&y 实际上是取地址而不是取引用的意思。

修改好的程序如下：

```
#include <iostream>
//using namespace std;
void swap(int *a, int *b)
{
 std::cout<<a<<std::endl;//调用 swap 函数时，构造 int*型的两个形参 a,b.这两个指

针分别指向主函数里的两个 int 型变量 a,b 的地址。（by abuse of notation...)

 int tmp = *a;//*a 是指针 a 指向的内容，把它赋值给 tmp ；

 *a = *b;//修改*a 就是修改 a 指向的内容的值，也就是修改主函数里面变量 a 的

值。
 *b = tmp;//同理。
}
int main() {
 int a=1,b(2);
 std::cout<<&a<<std::endl;
 swap(&a,&b);
 std::cout<<a<<std::endl;
return 0;
}
```

```
output：
0x7fff8f992eb8
0x7fff8f992eb8
2
```

从这个例子可以看出来，对实参取地址，然后 int*a=&x，就会使得 a 指针确切的指向 x 的内存空间，所以可以通过指针 a 控制 x 的内存空间。哪怕 x 的生命周期结束了，退出了函数体之后把 a 指针析构掉，因为我们没有人为定义 delete a，而默认析构函数不会 delete a，故而也不会影响 x 的内存空间。综上所述，终于解决了历史遗留问题。我们再做两个实验来验证下：

实验一，在函数体内 delete a，果然清除了 x 内存空间，导致 x 非法访问。

```
#include <iostream>
void swap(int *a, int *b)
{
 std::cout<<a<<std::endl;
 int tmp = *a;
 *a = *b;
 *b = tmp;
 delete a;
 return;
}
int main() {
 int x=1,y(2);
 std::cout<<&x<<std::endl;
 swap(&x,&y);
 std::cout<<x<<std::endl;
return 0;
}
Output：
free(): invalid pointer
```

实验二，传引用也是如此。

```
#include <iostream>

void swap(int &a, int &b) //此时函数体里的形参是主函数里两个变量 a,b 的引用（

也就是别名）

{
 std::cout<<&a<<std::endl;
 int tmp = a;
 a = b;
 b = tmp;
 return;
}
```

```
int main() {
 int x=1,y(2);
 std::cout<<&x<<std::endl;
 swap(x,y);
 std::cout<<x<<std::endl;
return 0;
}
Output：
0x7fff2af01638
0x7fff2af01638
2
```

基于上述引例，考察下这个例子

```
#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  if(d)等价于 if (d!=nullptr)
        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;
}
```

```
Output：
dog run
bird fly
```

注意到，在这个例子里面，向下类型转换都不是重点，重点在传参的这一步。因为我们的 p[i]是的类型是 Pet*，指向的内存空间是 Dog 或者 Bird，所以这个传参是合法的！也就是赋值语句，pet* p=p[i]是合法的，这个传参就是合法的！(将传参视为给函数参数列表里的形参赋值的过程)

三、抽象类与纯虚函数

3.1 多重继承

优点：清晰，符合直觉；结合多个接口

弊：

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。

钻石型继承树（DOD：Diamond Of Death）带来的数据冗余：右图中如果 InputFile 和 OutputFile 都含有继承自 File 的 filename 变量，则 IOFile 会有两份独立的 filename，而这实际上并不需要。

优化方案：

最多继承一个非抽象类（has-a），可以继承多个抽象类（接口）。这样可以避免多重继承的二义性，也可以一个对象可以实现多个接口。

3.2 例子

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

class WhatCanSpeak {
public:
    virtual ~WhatCanSpeak() {}
    virtual void speak() = 0;  };
class WhatCanMotion {
public:
    virtual ~WhatCanMotion() {}
    virtual void motion() = 0;   };
class Human : public WhatCanSpeak, public WhatCanMotion
{
    void speak() { cout << "say" << endl; }
    void motion() { cout << "walk" << endl; }
```

```
};
void doSpeak(WhatCanSpeak* obj) { obj->speak(); }
void doMotion(WhatCanMotion* obj) { obj->motion(); }
int main()
{
    Human human;
    doSpeak(&human); doMotion(&human);
    return 0;
}
Output：
say
walk
```

结合晚绑定体现多重继承。

四、多态 Polymorphism

4.1 定义

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”。这个定义非常繁琐，实际上就是按照实际类型调用函数罢了。

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，故而调用在指针权限范围内优先级最高的函数。关于指针的权限范围在 L8 已经叙述过。

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。

产生多态效果的条件：继承+虚函数+(引用或指针)

4.2 优势

全部设计抽象类指针指向派生类后，C++语言可以用一段相同的代码，在运行时完成不同的任务，这些不同运行结果的差异由派生类之间的差异决定，不必对每一个派生类特殊处理，只需要调用抽象基类的接口即可。大大提高程序的可复用性。另一方面，不同派生类对同一接口的实现不同，能达到不同的效果，提高了程序可拓展性和可维护性。

例子

```cpp
#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;
}
```

Output：

```
Fish cannot speak ...
Fish swimming
Bird singing
Bird flying
Fish cannot speak ...
```

```
Fish swimming
Bird singing
Bird flying
```

很有意思的是，直接通过派生类对象调用 action 函数时，由于派生类没有重定义 action，故而直接调用了基类的 action。之后以 fish 为例，调用了 fish.speak() 和 fish.motion()。然而注意到，Fish 类已经重写覆盖了 Animal 的虚函数 speak()，故而会调用自身重定义的 speak()，motion()同理。

下方的通过基类指针调用就是典型的重写覆盖+动态绑定。

五、函数模板与类模板

5.1 意义与定义

如果我们想实现对于整数、浮点数、自定义类的排序函数，这些排序本质上算法是相同的，但是我们可能需要写很多个相似的函数。

有些算法实现与类型无关，所以可以将函数的参数类型也定义为一种特殊的“参数”，这样就得到了“函数模板”。

定义函数模板的方法（可以不写在分开的两行）

```
template <typename T>
 ReturnType Func(Args) ;
```

如：任意类型两个变量相加的“函数模板”

```
template <typename T>
 T sum(T a, T b) { return a + b; }
```

注：typename 也可换为 class

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

模板必须在头文件中实现，原因比较复杂，涉及链接原理，不做赘述。

5.2 实例化与自动推导

函数模板在调用时，编译器能自动推导出实际参数的类型（这个过程叫做实例化）。所以，形式上调用一个函数模板与普通函数没有区别，如

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

调用类型需要满足函数的要求。本例中，要求类型 T 定义了加法运算符。

当多个参数的类型不一致并且不强制要求类型推导时，无法推导：

```
cout << sum(9, 2.1); //编译错误
```

可以强制指定类型推导方式：

```
#include<iostream>
```

```
using namespace std;
template <class T>
T sum(T a, T b) { return a + b; }
int main()
{
    cout<<sum<int>(9.9,2.5)<<endl;
    cout<<sum<float>(9.9,2.5)<<endl;
    cout<<sum<double>(9.9,2.5)<<endl;
}
Output：
11
12.4
12.4
```

从上例子可以看出先转换了再进行加法。

例一

```
#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
        }
    }
}
template<class T>
void output(T* data, int len)
{
    for(int i = 0; i < len; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
}
int main()
{
    int arr_a[] = {3,2,4,1,5};
    sort(arr_a, 5);  //调用 int 类型的 sort

    output(arr_a, 5); //调用 int 类型的 output
```

```
    float arr_b[] = {3.2, 2.1, 4.3, 1.5, 5.7};
    sort(arr_b, 5); //调用 float 类型的 sort

    output(arr_b, 5);  //调用 float 类型的 output

    return 0;
}
Output：
1 2 3 4 5
1.5 2.1 3.2 4.3 5.7
```

例二

```
#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

        }
    }
}
template<class T>
void output(T* data, int len)
{
    for(int i = 0; i < len; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
}
class MyInt
{
public:
    int data;
    MyInt(int val): data(val) {};
};
int main()
{
    MyInt arr_c[] = {3, 2, 4, 1, 5};
```

```
    sort(arr_c, 5);
    output(arr_c, 5);
    return 0;
}
Output：
main.cpp: In instantiation of ‘void sort(T*, int) [with T = MyInt]’:
main.cpp:33:15:      required from here
main.cpp:9:15: error: no match for ‘operator>’ (operand types are ‘MyInt’ and
‘MyInt’)
```

模板编译错误会引起上百行的错误，关注上方几行。sort 中需要 operator> 但 MyInt 并不支持，no match for ‘operator>’ (operand types are ‘MyInt’ and ‘MyInt’)

稍作修改：

```
#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
        }
    }
}
template<class T>
void output(T* data, int len)
{
    for(int i = 0; i < len; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
}
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);
    output(arr_c, 5);
    return 0;
}
Output：
1 2 3 4 5
```

完全重载了流运算符和大小运算符。

5.3 模板原理

对函数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**。

```
template<typename T>
T sum(T a, T b) {return a + b;}
int main() {
    int a = sum(1, 2); //生成并编译 int sum(int, int)

    double b = sum(1.0, 2.0); //生成并编译 double sum(double, double)

    return 0;
}
```

六、类模板

6.1 定义

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”。

```
#include <iostream>
using namespace std;
template <typename T> class A {
    T data;
public:
    A(T _data): data(_data) {}
    void print() { cout << data << endl; }
```

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

成员函数也可以如此定义

```
#include <iostream>
using namespace std;
template <typename T> class A {
    T data;
public:
    A(T _data): data(_data) {}
    void print();
};
template<typename T>
void A<T>::print() { cout << data << endl; }
int main() {
    A<int> a(1);
    a.print();
    return 0;
}
```

6.2 模板参数

我们可以将类模板视为一种特殊的“函数”，需要向其中传入参数才能够定义一个完整的类，而这些传入的参数有两种。一种是类型参数，也就是在 Typename 或者 Class 之后的参数(下例中的小 T)，第二种时非类型参数，这个可能有非常多种，比如整数，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)等等，比如下例就是无符号整数来定义了数组的大小。

实际上，模板比较复杂时，尖括号里可以放很多，可以是类型也可以是非类型。当然类型参数也可以有多个。

```
template<typename T, unsigned size>
 class array {
    T elems[size];
 };
 array<char, 10> array0;
```

注意到，size 只是一个无符号整数的名字而已，unsigned 和 Typename 才是关键，完整的传入两个参数才可形成一个完整的类，并定义这个类的对象。

一个模板可能会写出多个新的类，这些类是不同的，不过碰巧都用了同一个模板的名字。（比如上例中的 array 模板名字）

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

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

所有模板参数必须在编译期确定，因而不可以使用变量。

例子

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

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

    MyArr<float, 5> arr_b;
    arr_b.input();
    arr_b.sort();
    arr_b.output();
    return 0;
}
Input：3 2 4 1 5
3.2 2.1 4.3 1.5 5.7
Output：
1 2 3 4 5
1.5 2.1 3.2 4.3 5.7
```

6.3 模板与多态

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

但模板的关联是在编译期处理，称为**静多态**。

模板的特点：

往往和**函数重载**同时使用；高效，省去函数调用；编译后代码增多

基于继承和虚函数的多态在运行期处理，称为动多态

虚函数的特点：运行时，灵活方便；侵入式，必须继承；存在函数调用

所谓的省去函数调用是指，如果 doSomethingOnA 和 doSomethingOnB 的实现除了类型不同，其他基本一致的话则可省去如下代码。

```
if(typeof(a)==A){
 doSomethingOnA(a);
}
else if(typeof(a)==B){
 doSomethingOnB(a);
}
```

直接写一个模板然后 dosomethingOnWhatever