

# 多态与模板

## (OOP)

刘知远

`liuzy@tsinghua.edu.cn`

`http://nlp.csai.tsinghua.edu.cn/~lzy/`

课程团队：刘知远 姚海龙 黄民烈

# 上期要点回顾

- 向上类型转换
- 对象切片
- 函数调用捆绑
- 虚函数和虚函数表
- 虚函数和构造函数、析构函数
- 重写覆盖，`override`和`final`

# 本讲内容提要

- 纯虚函数与抽象类
- 向下类型转换
- 多重继承的虚函数表，多重继承的利弊
- 多态
- 函数模板与类模板

# 纯虚函数

- 虚函数还可以进一步声明为纯虚函数（如下所示），包含纯虚函数的类，通常被称为“**抽象类**”。

**virtual** 返回类型 函数名(形式参数) = 0;

- 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**”

```
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！

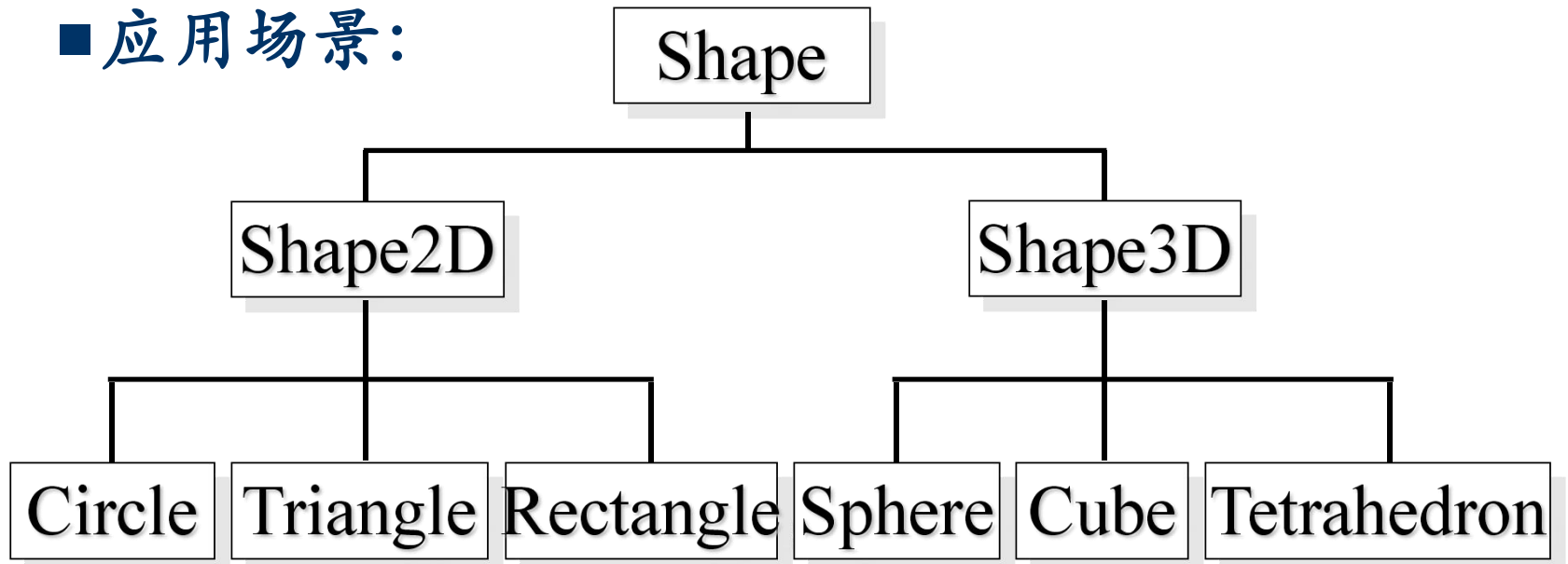
# 抽象类

■ 定义：含有至少一个纯虚函数。

■ 特点：

- 不允许定义对象。
- 只能为派生类提供接口。
- 能避免对象切片：保证只有指针和引用能被向上类型转换。

■ 应用场景：



# 纯虚函数与抽象类示例

```
#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

- 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，必须实现所有纯虚函数，否则继承出的类也是抽象类。
- 纯虚析构函数除外

# 抽象类示例

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

class Base {
public:
    virtual void func()=0;
};

class Derive1: public Base {};
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;
}
```

运行结果

*Derive2::func*



# 纯虚析构函数

## ■ 回顾：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。

# 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，则析构函数不必是纯虚的。

```
class Base { public: virtual ~Base()=0; };
```

```
Base::~~Base() {} /// 必须有函数体
```

```
class Derive : public Base {};
```

```
int main() {  
    Base b; /// 编译错误，基类是抽象类  
    Derive d1;  
    return 0;  
}
```

# 纯虚析构函数

## ■ 纯虚析构函数和一般纯虚函数

- 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。
- 纯虚析构函数除外
- 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，即使派生类不覆盖纯虚析构函数，**派生类可以不是抽象类，可以定义派生类对象。**

# 纯虚析构函数 示例

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

class Base{
public:
    virtual ~Base()=0;
};
Base::~~Base() {cout<<"Base destruct"<<endl;}

class Derive1: public Base {};
class Derive2: public Base {
public:
    virtual ~Derive2() {cout<<"Derive2 destruct"<<endl;}
};

int main()
{
    Base* p1 = new Derive1;
    Base* p2 = new Derive2;
    delete p1;
    delete p2;
    return 0;
}
```

## 运行结果

*Base destruct*  
*Derive2 destruct*  
*Base destruct*

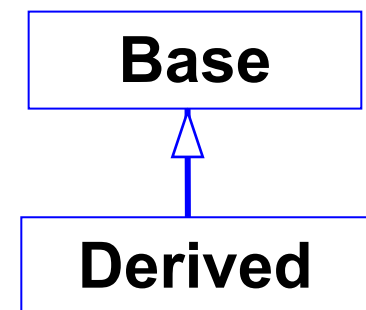
下面关于虚函数和抽象类的描述，错误的是：

- ☒ A 通过类的指针或引用调用类内函数，均可实现晚绑定(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，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

提交

# 回顾：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。



# 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**。（类层次中向下移动）
- 为什么要向下类型转换？
  - 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。
- 如何确保转换的正确性？
  - 如何保证基类指针指向的对象也可以被要转换的派生类的指针指向？—— 借助虚函数表进行**动态类型检查**！

# 向下类型转换

- C++提供了一个特殊的显式类型转换，称为 `dynamic_cast`，是一种安全类型向下类型转换。
  - 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。
- 使用方法：
  - `obj_p`，`obj_r` 分别是 `T1` 类型的指针和引用
  - `T2* pObj = dynamic_cast<T2*>(obj_p);`  
// 转换为 `T2` 指针，运行时失败返回 `nullptr`
  - `T2& refObj = dynamic_cast<T2&>(obj_r);`  
// 转换为 `T2` 引用，运行时失败抛出 `bad_cast` 异常
  - `T1` 必须是多态类型（声明或继承了至少一个虚函数的类），否则不过编译；`T2` 不必。`T1`, `T2` 没有继承关系也能通过编译，只不过运行时会转换失败。



# 向下类型转换

- 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。

- `static_cast` 使用方法：

- `obj_p`, `obj_r` 分别是 `T1` 类型的指针和引用
- `T2* pObj = static_cast<T2*>(obj_p);`  
// 转换为 `T2` 指针
- `T2& refObj = static_cast<T2&>(obj_r);`  
// 转换为 `T2` 引用
- 不安全：不保证转换后的目标是 `T2` 类型的，可能导致非法内存访问。

# 示例

```
#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
    //    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;

}
```

```
static_cast, B*(B) --> D*:OK
D::i=124455624
dynamic_cast, B*(B) --> D*: FAILED
```

# 示例

```
#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
    //    D d2 = dynamic_cast<D>(b); ///只允许指针和引用转换

    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;
}
```

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

# 向下类型转换

## 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。
- 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

# 向下类型转换

重要原则(清楚指针所指向的真正对象):

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

# 向上向下类型转换与虚函数表

## ■ 对于基类中有虚函数的情况：

## ■ 向上类型转换：

- 转换为基类**指针或引用**，则对应虚函数表仍为派生类的虚函数表（晚绑定）。
- 转换为基类**对象**，则对应虚函数表是基类的虚函数表（早绑定）。

## ■ 向下类型转换：

- `dynamic_cast`通过虚函数表来判断是否能进行向下类型转换。

# 示例

```
#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;
}
```

运行结果

*dog run*  
*bird fly*

## 下面关于向下类型转换的说法，正确的是

A

使用dynamic\_cast对基类指针向下类型转换时，转换失败返回空指针

B

dynamic\_cast只在运行时确认是否满足正确类型转换的条件

C

static\_cast只在编译时检查继承关系是否允许类型转换

D

static\_cast无法在运行时确定转换目标类型是否正确，因此可能会导致非法内存访问

提交



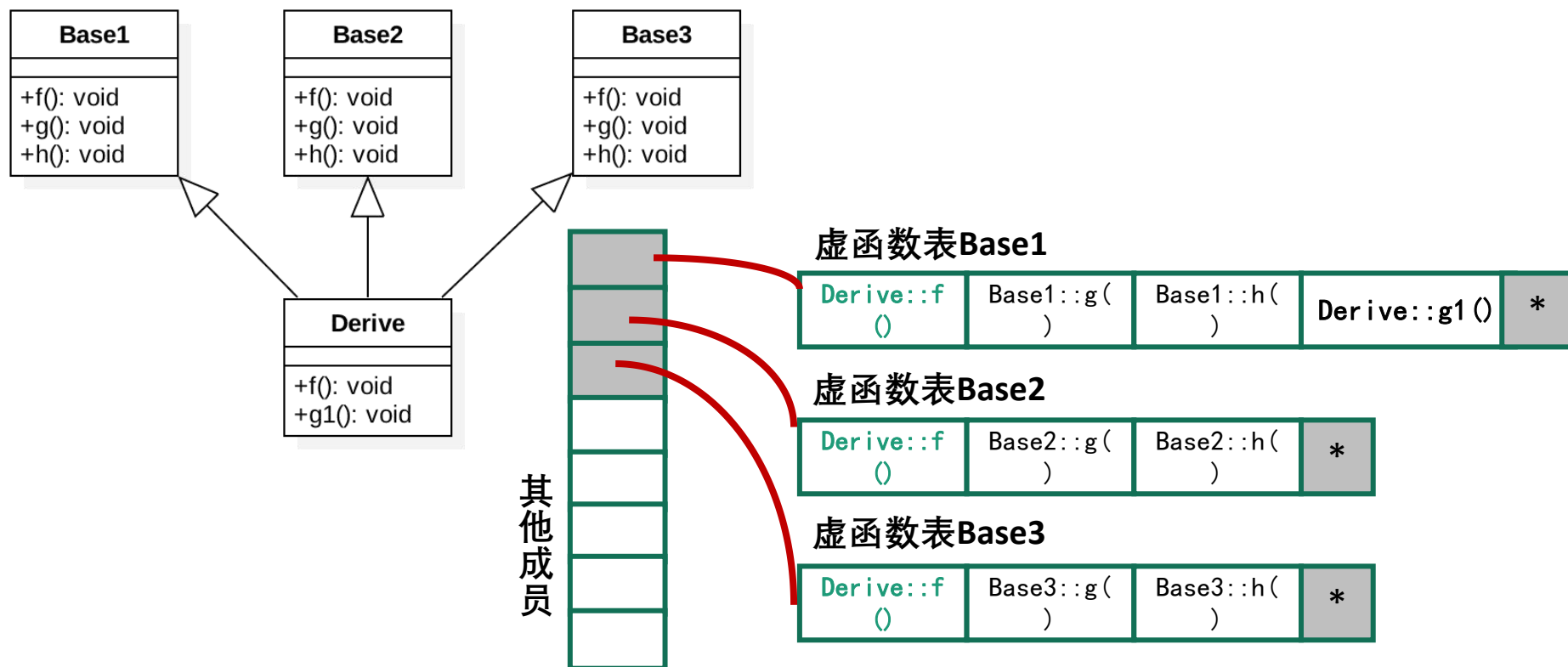
# 多重继承的虚函数表

- 多重继承会有多个虚函数表，几重继承，就会有几个虚函数表。这些表按照派生的顺序依次排列。
- 如果子类改写了父类的虚函数，那么就会用子类自己的虚函数覆盖虚函数表的相应位置，如果子类有新的虚函数，那么就添加到第一个虚函数表的末尾。

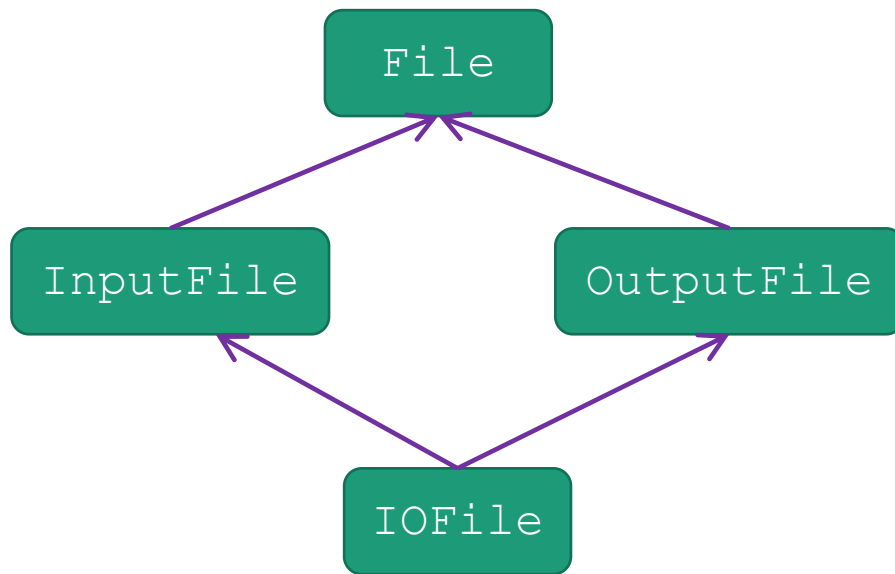
# 多重继承

- 左图是派生关系，右图是 **Derive** 对象的虚函数指针，以及 **Derive** 类的虚函数表。
- **Derive** 重写了所有基类的 **f()**，新定义了一个虚函数 **g1()**。

```
Class Derive: public Base1,  
              public Base2, public Base3 {};
```



# 多重继承



## ■ 利:

- 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。

# 多重继承

## ■ Best Practice:

- 最多继承一个非抽象类 (is-a)
- 可以继承多个抽象类 (接口)

## ■ 为什么?

- 避免 多重继承的二义性
- 利用 一个对象可以实现多个接口

# 多重继承示例

```
#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;
}
```

运行结果

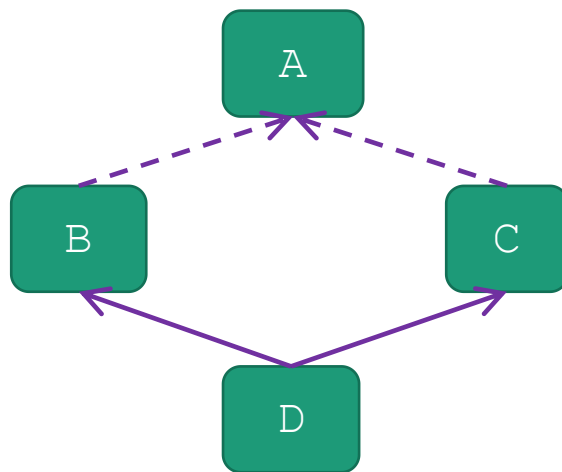
say  
walk

# 多重继承

## ■ 虚拟继承 (仅了解)

- 用于解决多重继承中**二义性**和**数据冗余**的问题：从不同途径继承来的同一基类，会在子类中存在多份拷贝。
- 虚基类并不是在声明基类时声明的，而是在**声明派生类**时，指定继承方式时声明的。

```
#include <iostream>
using namespace std;
class A {public: int a;};
//声明基类A
class B: virtual public A {};
//A为B的虚基类
class C: virtual public A {};
//A为C的虚基类
class D: public B, public C
{public: void func(){cout<<a<<endl;}};
//D访问直接基类的成员时无二义性
```



# 多态 (Polymorphism)

- 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
- 产生多态效果的条件：**继承 && 虚函数 && (引用 或 指针)**

# 多态 (Polymorphism)


- 多态，使得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。
- 好处：
  - 通过基类定好接口后，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。
  - 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。



# 多态示例

```
#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; }
};
```



复用基类接口

```
class Bird : public Animal
{
public:
    void speak() { cout << "Bird singing" << endl; }
    void motion() { cout << "Bird flying" << endl; }
};
```

# 多态示例

```
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;
}
```

## 运行结果

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

# 多态 (Polymorphism)


- 应用：TEMPLATE METHOD设计模式
  - 在接口的一个方法中定义算法的骨架
  - 将一些步骤的实现延迟到子类中
  - 使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。
- 模板方法是一种**源代码重用**的基本技术，在类库的设计实现中应用十分广泛，因为这个设计模式能有效地解决“**类库提供公共行为**”与“**用户定制特殊细节**”之间的折中平衡。

# 模板设计模式

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

```
class Base{
public:
    void action() {
        step1();
        step2();
        step3();
    }
    virtual void step1() { cout << "Base::step1" << endl; }
    virtual void step2() { cout << "Base::step2" << endl; }
    virtual void step3() { cout << "Base::step3" << endl; }
};

class Derived1 : public Base{
    void step1() { cout << "Derived1::step1" << endl; }
};
```



算法骨架

# 模板设计模式

```
class Derived2 : public Base{  
    void step2() { cout << "Derived2::step2" << endl; }  
};
```

```
int main(){  
    Base* ba[] = {new Base, new Derived1, new Derived2};  
    for (int i = 0; i < 3; ++i) {  
        ba[i]->action();  
        cout<<"==="<<endl;  
    }  
    return 0;  
}
```

运行结果

```
Base::step1  
Base::step2  
Base::step3  
===  
Derived1::step1  
Base::step2  
Base::step3  
===  
Base::step1  
Derived2::step2  
Base::step3  
===
```

下列关于多态的说法，正确的是

A

利用基类的指针或引用调用函数时，虚函数和非虚函数的绑定时间不一致

B

多态可以提高代码接口的复用性

C

虚函数的目标是实现函数地址的早绑定

D

利用基类对象调用虚函数时，也可产生多态现象

提交

# 函数模板和类模板

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

# 模板：引入

- 实现一个整数排序算法接口：

```
void sort(int *data, int len);
```

- 实现一个浮点数排序算法接口：

```
void sort(float *data, int len);
```

- 实现一个自定义类型排序算法接口：

```
void sort(myClass *data, int len);
```

- 明明实现是一样的，为什么要写多遍？



# 模板分类

- 函数模板
- 类模板
- 成员函数模板

# 函数模板

- 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

```
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

# 函数模板

- 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时，无法推导：  
`cout << sum(9, 2.1);` **// 编译错误**
- 手工指定调用类型：`sum<int>(9, 2.1)`

# 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”。例如：

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

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

# 类模板

## ■ 类模板中成员函数的类外定义

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

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

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

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

# 类模板

## ■ 类模板的“模板参数”

- 类型参数：使用typename或class标记
- 非类型参数：**整数**，枚举，指针（指向对象或函数），引用（引用对象或引用函数）。整数型比较常用。如：

```
template<typename T, unsigned size>
```

```
class array {  
    T elems[size];
```

```
    ...
```

```
};
```

```
array<char, 10> array0;
```

# 成员函数模板

- 普通类的成员函数，也可以定义为模板函数，如：

```
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);  
}
```

# 成员函数模板

- 模板类的成员函数，也可有额外的模板参数

```
template<typename T0> class A {  
    T0 value;  
public:  
    template<typename T1> void set(T1 const& v){  
        value = T0(v);  
    }           ///  
    template<typename T1> T1 get();  
};  
template<typename T0> template<typename T1>  
T1 A<T0>::get(){ return T1(value);}  ///  

```



# 成员函数模板

■ 注意不能写成：

```
template<typename T0, typename T1>
```

```
T1 A<T0>::get(){ return T1(value);}
```

```
/// 类外定义
```

多个参数的类模板：

```
template<typename T0, typename T1> class A  
{}
```

多个参数的函数模板

```
template<typename T0, typename T1> void  
func( T0 a1, T1 a2) {}
```

# 成员函数模板

- 模板使用中通常可以自动推导类型，必要时也可以指定

```
template<typename T0> class A {  
    T0 value;  
public:  
    template<typename T1> void set(T1 const& v){  
        value = T0(v);  
    }           ///  
    template<typename T1> T1 get();  
};  
template<typename T0> template<typename T1>  
T1 A<T0>::get(){ return T1(value);} ///  
  
int main() {  
    A<int> a;  
    a.set(5);           ///  
    double t = a.get<double>();    ///  
}
```

下列函数模板的声明中，正确的是

☐ A `template<typename T1, T2> void func(T1 a, T2 b) {...}`

☐ B `template<class T1, T2> void func(T1 a, T2 b) {...}`

☒ C `template<class T1, class T2> void func(T1 a, T2 b) {...}`

☒ D `template<typename T1, typename T2> void func(T1 a, T2 b) {...}`

提交

# 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。
- 这意味着所有模板参数必须在编译期确定，不可以使用变量。

```
int n = 5; myClass<n> a; //错误
```

```
const int n = 5; myClass<n> b; //正确
```

- 也带来了问题：模板库必须在头文件中实现，不可以分开编译（**请思考为什么？**）

# 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**。
  - 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**
  - 运行时，灵活方便
  - 侵入式，必须继承
  - 存在函数调用

下列关于多态性的说法正确的是：

☒ A C++语言的多态性分为编译时的多态性和运行时的多态性

☐ B 运行时的多态性可通过模板和虚函数实现

☒ C 编译时的多态性可通过函数重载实现

☐ D 实现编译时多态性的机制称为动态多态性

提交

# OOP核心思想

■ OOP的核心思想是数据抽象、继承与动态绑定

■ 数据抽象：类的接口与实现分离

■ 继承：建立相关类型的层次关系（基类与派生类）

■ 动态绑定：统一使用基类指针，实现多态行为

# OOP核心思想

■ OOP的核心思想是数据抽象、继承与动态绑定

■ 数据抽象：类的接口与实现分离

- 回顾Animal/模板设计的例子

■ 继承：建立相关类型的层次关系（基类与派生类）

- Is-a、is-implementing-in-terms-of：客观世界的认知关系

■ 动态绑定：统一使用基类指针，实现多态行为

- 虚函数
- 类型转换，模板



# 课后阅读

## ■ 《C++编程思想》

- 多态性与虚函数，p364-p390
- 模板，p400-p435

**结 束**