directory

review

- 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

纯虚函数

- **纯虚函数**：把虚函数进一步声明为纯虚函数
  - `virtual 返回类型 函数名(形式参数) = 0;`
- **抽象类**：包含纯虚函数的类，通常被称为“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”**
  - 避免对象切片：保证只有指针和引用能够被向上类型转换

```
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！
```

- 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(**不能定义类对象**)

```
#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;
}
```

纯虚析构函数

review：虚函数与析构函数

- 析构函数常常是虚函数，虚析构函数仍需要定义函数体
- 虚析构函数的用途：删除基类对象指针时，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

- 纯虚析构函数仍然徐娅函数体
- 目的：使基类成为抽象类，不能创建基类的对象
- 只要派生类覆盖了抽象类中其他的纯虚函数，该派生类就不是抽象类，不必显式实现纯虚析构函数（编译器自动合成默认析构函数）

向下类型转换

review：向上类型转换

- 编译器自动完成，隐式类型转换
- **派生类**对象/引用/指针 ---> **基类**对象/引用/指针
- 只对public继承有效

向下类型转换

- **基类**指针/引用 ---> **派生类**指针/引用
- 目的：
  - 使用基类指针表示各种派生类时，仅保留共性(基类)，但丢失派生类特性
  - 可以使用**基类指针数组**，对各种派生类对象进行管理
- 如何确保转换的正确性？
  - 借助虚函数表进行动态类型检查
- 使用方法：dynamic_cast与static_cast

dynamic_cast

- Base 为至少包含一个虚函数的基类
  - **必须有虚函数**
  - 因为dynamic_cast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：

    - `Derived *dp = dynamic_cast<Derived *>(bp)`
        - 指针
        - Base为包含至少一个虚函数的基类
        - 如果有一个指向Base的指针bp，可以在**运行**时将它转换成指向Derived的指针
        - 转换为T2指针，运行时失败返回**nullptr**
    - `Derived &dp = dynamic_cast<Derived &>(bp)`
        - 左值引用
        - 转换为T2引用，运行时失败抛出**bad_cast异常**
            - 异常定义在头文件typeinfo中
    - `dynamic_cast< type&& >(e)`
        - 右值

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

static_cast

- 在编译时**只检查继承关系**，用于非多态的转换
- 相当于强制转换
- 运行时无法确认是否正确转换
- 使用方法：
    - `static_cast< new_type >(expression)`
    - `obj_p`，`obj_r`分别是`T1`类型的指针和引用
    - 引用：`T2* pObj = static_cast<T2*>(obj_p);`
        - //转换为T2指针
    - 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。
- 用途：
    - 基类-->派生类之间指针或引用的转换
    - 基本数据类型之间的转换
        - `char a = 'a'; int b = static_cast<char>(a);`
    - 把空指针转换成目标类型的空指针
    - 把任何类型的表达式转换成void类型

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

Base* pB = new Base();

if(Derived* pD = static_cast<Derived*>(pB)) {}//下行转换是不安全的(坚决抵制这种方法)

Derived* pD = new Derived();
if(Base* pB = static_cast<Base*>(pD)) {}//上行转换是安全的
```

两种向下类型转换：

- `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
- 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

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

int main() {
    B b; // 基类
    D d; // 派生类

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

```
#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;
}
```

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：
  - 派生类-->基类
  - 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**，产生对象切片，调用基类函数（早绑定）。
- 向下类型转换：
  - 基类-->派生类
  - dynamic_cast通过虚函数表来判断是否能进行向下类型转换。

```
#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;
}
```

多态

- 静态多态
  - 函数重载
  - 函数模板
- 动态多态
  - 动态绑定
    - 目前所指对象的类型，运行时确定
      - 静态类型：对象声明时的类型，编译时确定
      - 动态类型：目前所指对象的类型，运行时确定
    - 必须是虚函数（派生类一定要重写基类中的虚函数）
    - 通过**基类的引用或指针调用虚函数**：根据指针(引用)实际指向(引用)的类型确定调用基类还是派生类的虚函数
    - 调用非虚函数：只能调用基类的函数

riview 多重继承

- 虚函数
  - 最多继承一个非抽象类（is-a）
  - 可以继承多个抽象类（接口）

```
#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;
}
```

多态

- 按照基类的接口定义，调用**指针或引用**所指对象的接口函数，函数执行过程因对象实际所属派生类的不同而呈现不同的效果（表现）
- 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)**
- 作用：
  - 提高代码复用性
    - 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
  - 提高可拓展性和可维护性
    - 不同派生类对同一接口的实现不同，能达到不同的效果

```
#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;
}
```

- 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。

函数模板与类模板

- 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"
- 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; }
```

- 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
  - 模板库必须在头文件中实现，不可以分开编译

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
- 所有模板参数必须在**编译期**确定，不可以使用变量。

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

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

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);
}
```

- 模板类的成员函数，也可有额外的模板参数

```
template<typename T0> class A {
    T0 value;
public:
    template<typename T1> void set(T1 const& v){
    value = T0(v); /// 将T1转换为T0储存
    }         /// 在类内定义
    template<typename T1> T1 get();
};

template<typename T0> template<typename T1>
T1 A<T0>::get(){ return T1(value);}
/// 类外定义，  将T0转换为T1返回
```

- 注意不能写成：

```
template<typename T0, typename T1>
 T1 A<T0>::get(){ return T1(value);} /// 错误，与多个参数的模板混淆
```

```
 template<typename T0> template<typename T1>
 T1 A<T0>::get(){ return T1(value);} /// 正确
```

- 多个参数的模板
- 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) {…}
```

- 模板使用中通常可以从参数自动推导类型，无法推导时需要指定
- 普通类模板的成员函数，也可有额外的模板参数

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

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