标准C++语言

PART 2

DAY01

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	其它操作符的重载及限制
	10:30 ~ 11:20	
	11:30 ~ 12:20	
下午	14:00 ~ 14:50	继承的基本概念和语法
	15:00 ~ 15:50	公有继承的基本特点
	16:00 ~ 16:50	继承方式与访问控制
	17:00 ~ 17:30	总结和答疑

其它操作符的重载及限制

其它操作符的重载及限制

- 其它操作符的重载
  - 下标操作符
  - 函数操作符
  - 解引用和间接成员访问操作符
  - 智能指针
  - auto_ptr
  - 自定义类型转换
  - 对象创建操作符
  - 对象销毁操作符
- 操作符重载的限制
  - 不能重载
  - 基本类型与优先级
  - 操作数个数
  - 发明操作符
  - 一致性
  - 可读性

其它操作符的重载

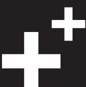
下标操作符

- 常用于在容器类型中以下标方式获取数据元素
- 非常容器的元素为左值，常容器的元素为右值

知识讲解

```
class Array {
public:
    int& operator[] (size_t i) {
        return m_array[i];
    }
    const int& operator[] (size_t i) const {
        return const_cast<Array&> (*this)[i];
    }
private:
    int m_array[256];
};
```

```
Array array;
array[100] = 1000;
// array.operator[] (100) = 1000;
const Array& carr = array;
cout << carr[100] << endl;
// cout << carr.operator[] (100) << endl;
```



支持[]操作符的数组类

【参见：TTS COOKBOOK 】

课堂练习

- 支持[]操作符的数组类

函数操作符

- 如果一个类重载了函数操作符，那么该类的对象就可以被当做函数来调用，其参数和返回值就是函数操作符函数的参数和返回值
- 参数的个数、类型以及返回值的类型，没有限制
- 唯一可以带有缺省参数的操作符函数

知识讲解

```
class Less {
public:
    bool operator() (int a, int b) const {
        return a < b;
    }
};
```

```
Less less;
cout << less (100, 200) << endl;
// cout << less.operator() (100, 200) << endl;
```

支持()操作符的平方类

【参见：TTS COOKBOOK 】

课堂练习

- 支持()操作符的平方类

解引用和间接成员访问操作符

- 如果一个类重载了解引用和间接成员访问操作符，那么该类的对象就可以被当做指针来使用

知识讲解

```
class Integer {
public:
  Integer (const int& val = 0) : m_val (val) {}
  int& value (void) {
    return m_val;
  }
  const int& value (void) const {
    return m_val;
  }
private:
  int m_val;
};
```

```
class IntegerPointer {
public:
  IntegerPointer (Integer* p = NULL) : m_p (p) {}
  Integer& operator* (void) const {
    return *m_p;
  }
  Integer* operator-> (void) const {
    return m_p;
  }
private:
  Integer* m_p;
};
```

```
IntegerPointer ip (new Integer (100));
(*ip).value ()++;
// ip.operator* ().value ()++;
cout << ip->value () << endl;
// cout << ip.operator-> ()->value () << endl;
```

智能指针

知识讲解

- 常规指针的缺点
  - 当一个常规指针离开它的作用域时，只有该指针变量本身所占据的内存空间(通常是4个字节)会被释放，而它所指向的动态内存并未得到释放
  - 在某些特殊情况下，包含free/delete/delete[]的代码根本就执行不到，形成内存泄漏
- 智能指针的优点
  - 智能指针是一个封装了常规指针的类类型对象，当它离开作用域时，其析构函数负责释放该常规指针所指向的动态内存
  - 以正确方式创建的智能指针，其析构函数总会执行

智能指针（续1）

- 智能指针与常规指针的一致性
  - 为了使智能指针也能象常规指针一样，通过“*”操作符解引用，通过“->”操作符访问其目标的成员，就需要对这两个操作符进行重载
- 智能指针与常规指针的不一致性
  - 任何时候，针对同一个对象，只允许有一个智能指针持有其地址，否则该对象将在多个智能指针中被析构多次(double free)
  - 智能指针的拷贝构造和拷贝赋值需要做特殊处理，对其所持有的对象地址，以指针间的转移代替复制
  - 智能指针的转移语义与常规指针的复制语义不一致

知识讲解

简化版的智能指针类

【参见：TTS COOKBOOK 】

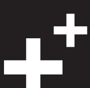
课堂练习

- 简化版的智能指针类

auto_ptr

- 标准库的auto_ptr<T>，实现了智能指针的基本功能
  - #include <memory>
  - auto_ptr<目标类型> 智能指针对象 (
    new操作符返回的目标对象地址);
  - auto_ptr<Integer> p (new Integer (100));
- auto_ptr<T>的局限性
  - 不能跨作用域使用
  - 不能放入标准容器
  - 不能指向对象数组

知识讲解



auto_ptr的基本用法和局限

【参见：TTS COOKBOOK 】

课堂练习

- auto_ptr的基本用法和局限

自定义类型转换

知识讲解

- 通过构造函数实现自定义类型转换
  - class Integer {
    [explicit] Integer (int const& data) : m_data (data) {}
    };
  - “explicit”关键字可将这种类型转换强制为显式类型转换
- 通过类型转换操作符函数实现自定义类型转换
  - class Integer {
    [explicit] operator int (void) const { return m_data; }
    };
  - “explicit”关键字可将这种类型转换强制为显式类型转换，但需要编译器支持C++11标准

自定义类型转换（续1）

- 源类型是基本类型，只能通过构造函数实现自定义类型转换
- 目标类型是基本类型，只能通过类型转换操作符函数实现自定义类型转换
- 源类型和目标类型都不是基本类型，既可以通过构造函数也可以通过类型转换操作符函数实现自定义类型转换，但不要两者同时使用，引发歧义
- 源类型和目标类型都是基本类型，则无法实现自定义类型转换，基本类型间的类型转换规则完全由编译器内置

知识讲解

在基本类型和类类型之间做类型转换

【参见：TTS COOKBOOK 】

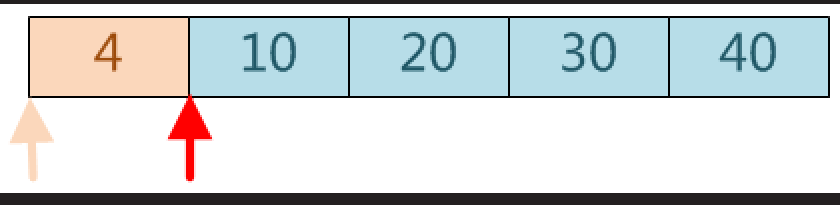
课堂练习

- 在基本类型和类类型之间做类型转换

对象创建操作符

知识讲解

- 如果一个类重载了new/new[]操作符，那么当通过new/new[]创建该类的对象/对象数组时，将首先调用该操作符函数分配内存，然后再调用该类的构造函数
  - class Dummy {
        static void* operator new (size_t size) { ... }
        static void* operator new[] (size_t size) { ... }
    };
- 包含自定义析构函数的类，通过new[]创建对象数组，所分配的内存会在低地址部分预留出sizeof(size_t)个字节，存放数组长度



对象创建操作符（续1）

知识讲解

- 使用new/new[]操作符的形式代码

```
class Dummy {
public:
  Dummy (void) {}
  ~Dummy (void) {}
  static void* operator new (size_t size) {
    return malloc (size);
  }
  static void* operator new[] (size_t size)
    return malloc (size);
  }
};
```

```
Dummy* dummy = new Dummy;
// Dummy* dummy = (Dummy*)Dummy::operator new (sizeof (Dummy));
// dummy->Dummy ();
Dummy* dummies = new Dummy[10];
// Dummy* dummies = (Dummy*)((size_t*)Dummy::operator new[] (
//   sizeof (size_t) + 10 * sizeof (Dummy)) + 1);
// *((size_t*)dummies - 1) = 10;
// for (size_t i = 0; i < *((size_t*)dummies - 1); ++i)
//   (dummies + i)->Dummy ();
```

对象销毁操作符

知识讲解

- 如果一个类重载了delete/delete[]操作符，那么当通过delete/delete[]销毁该类的对象/对象数组时，将首先调用该类的析构函数，然后再调用该操作符函数释放内存
  - class Dummy {

```
class Dummy {
    static void operator delete (void* p) { ... }
    static void operator delete[] (void* p) { ... }
};
```

- 包含自定义析构函数的类，通过delete[]销毁对象数组，会根据低地址部分预存的数组长度，从高地址到低地址依次对每个数组元素调用析构函数

对象销毁操作符（续1）

知识讲解

- 使用delete/delete[]操作符的形式代码

```
class Dummy {
public:
  Dummy (void) {}
  ~Dummy (void) {}
  static void operator delete (void* p) {
    free (p);
  }
  static void operator delete[] (void* p) {
    free (p);
  }
};
```

```
delete dummy;
// dummy->~Dummy ();
// Dummy::operator delete (dummy);
delete[] dummies;
// for (size_t i = *((size_t*)dummies - 1) - 1; ; --i) {
//   (dummies + i)->~Dummy ();
//   if (i == 0) break;
// }
// Dummy::operator delete[] ((size_t*)dummies - 1);
```

重载new和delete操作符

【参见：TTS COOKBOOK 】

课堂练习

- 重载new和delete操作符

操作符重载的限制

不能重载

知识讲解

- 不是所有的操作符都能重载，以下操作符不能重载
  - 作用域限定操作符(::)
  - 直接成员访问操作符(.)
  - 直接成员指针解引用操作符(.*)
  - 条件操作符(?:)
  - 字节长度操作符(sizeof)
  - 类型信息操作符(typeid)

基本类型与优先级

- 无法重载所有操作数均为基本类型的操作符
  - int operator+ (int lhs, int rhs) { return lhs - rhs; }
  - x = 1 + 1; // x = 0 ?
- 无法通过操作符重载改变操作符的优先级
  - // z = x^y

```
Integer const operator^ (Integet const& x, int y) {
    Integer z (1);
    for (int i = 0; i < y; ++i)
        z.m_n *= x.m_n;
    return z;
}
```

  - d = a + b ^ c; // d = a + b^c ?

知识讲解

操作数个数

- 无法通过操作符重载改变操作数的个数
  - double operator% (Integer const& opd) {
        return opd.m_n / 100.0;
    }
  - x = 50%; // x = 0.5 ?

知识讲解

发明操作符

知识讲解

- 无法通过操作符重载发明新的操作符
  - // $c^2 = a^2 + b^2$
    ```
    Double const operator@ (Double const& a,
        Double const& b) {
        return Double (sqrt (a.m_d * a.m_d + b.m_d * b.m_d));
    }
    ```
  - c = 3.0 @ 4.0; // c = 5.0 ?

一致性

- 操作符重载源自对一致性的追求，任何违反既有规则，甚至有悖于人类常识的实现，都应该尽可能地避免
  - Complex const operator- (Complex const& lhs,
        Complex const& rhs) {
        return Complex (rhs.m_r - lhs.m_r, rhs.m_i - lhs.m_i);
    }
  - x = (3 + 4i) - (1 + 2i); // x = -2 - 2i ?

知识讲解

可读性

- 操作符重载的价值在于提高代码的可读性，而不是成为某些别有用心的算符控们赖以卖弄的奇技淫巧

知识讲解

```
class Stack {
public:
    Stack (void) { ... };
    ~Stack (void) { ... };
    void operator+ (int data) { ... }  // 压入
    int operator- (void) { ... }       // 弹出
    int operator* (void) const { ... } // 栈顶
    operator bool (void) const { ... } // 判空
    ...
};
```

```
Stack stack;
for (int i = 0; i < 10; ++i)
    stack + i;
while (! stack) {
    cout << *stack << endl;
    -stack;
}
```

继承的基本概念和语法

- 继承的基本概念和语法
  - 继承的基本概念
    - 共性与个性
    - 超集与子集
    - 基类与子类
    - 继承与派生
  - 继承的语法
    - 继承表
    - 继承方式

继承的基本概念

共性与个性

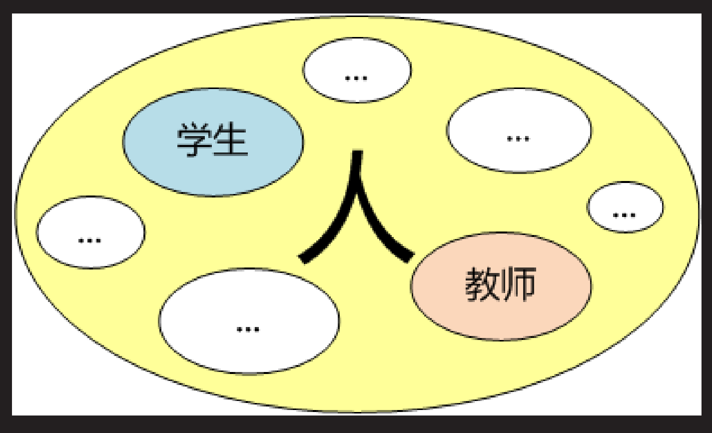
- 共性表达了不同类型事物之间共有的属性和行为
- 个性则着意刻画每种类型事物特有的属性和行为

知识讲解

超集与子集

- 超集体现了基于共性的一般
- 子集体现了针对个性的特殊

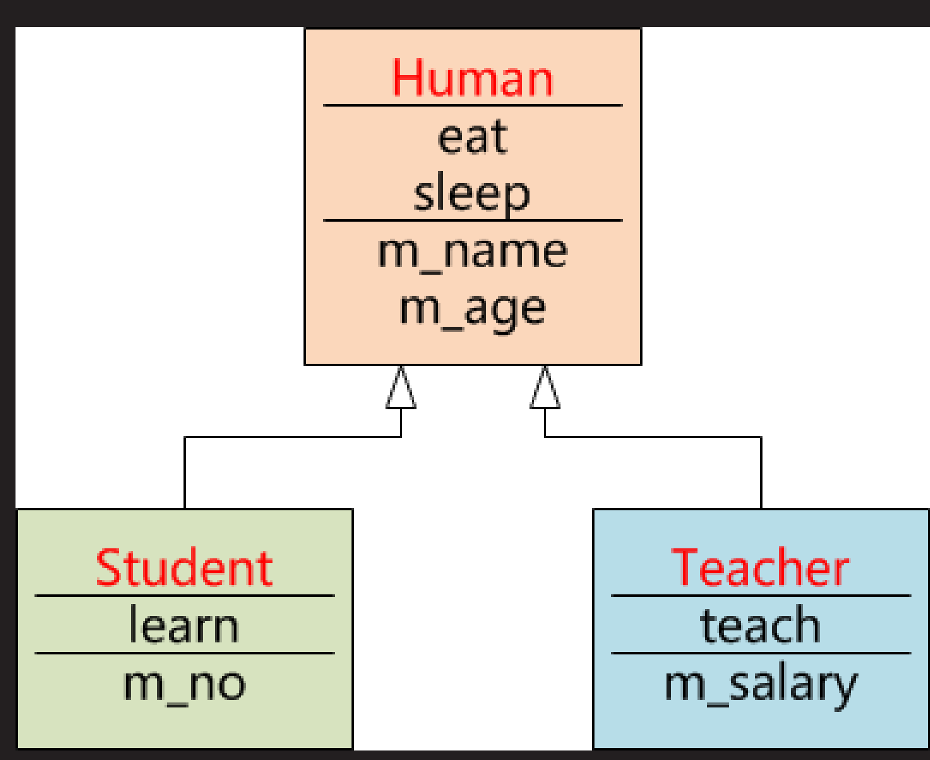
知识讲解

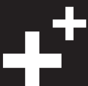


基类与子类

- 基类表示超集，体现共性，描述共有的属性和行为
- 子类表示子集，体现个性，描述特有的属性和行为

知识讲解





继承与派生

- 子类继承自基类
- 基类派生出子类

知识讲解

```
class Human {
public:
  void eat (const string& food) { ... }
  void sleep (int hours) { ... }
  string m_name;
  int m_age;
};
```

```
class Student : public Human {
public:
  void learn (
    const string& course) { ... }
  int m_no;
};
```

```
class Teacher : public Human {
public:
  void teach (
    const string& course) { ... }
  float m_salary;
};
```

继承的语法

继承表

- 一个子类可以同时从零到多个基类继承
- 子类从每一个基类继承的继承方式可以相同也可以不同
  - class 子类 : 继承方式1 基类1, 继承方式2 基类2, ... {

    ...

    };

知识讲解

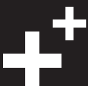
```
class/struct 类名 : 继承方式 基类, ... {
访问控制限定符:
        类名 (形参表) : 成员变量 (初值), ... {
                函数体;
        }
        ~类名 (void) {
                函数体;
        }
        返回类型 函数名 (形参表) 常属性 异常说明 {
                函数体;
        }
        数据类型 变量名;
};
```

类
继承表
初始化表
构造函数
析构函数
成员函数
成员变量

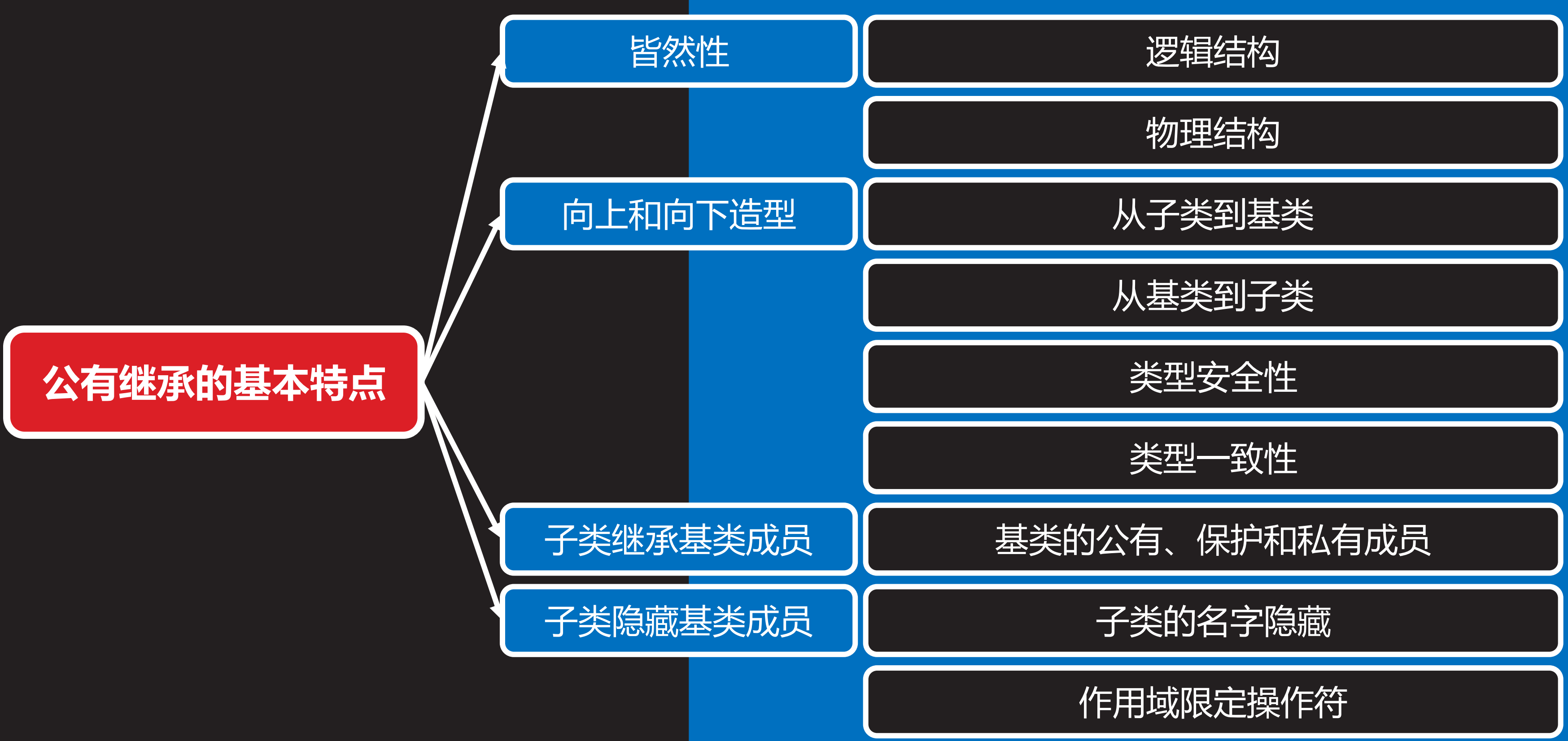
继承方式

知识讲解

- 公有继承
  - public
  - 基类的特性将通过子类向外扩散
- 保护继承
  - protected
  - 基类的特性仅在继承链的范围内扩散
- 私有继承
  - private
  - 基类的特性仅为子类所有，不向任何方面扩散



公有继承的基本特点

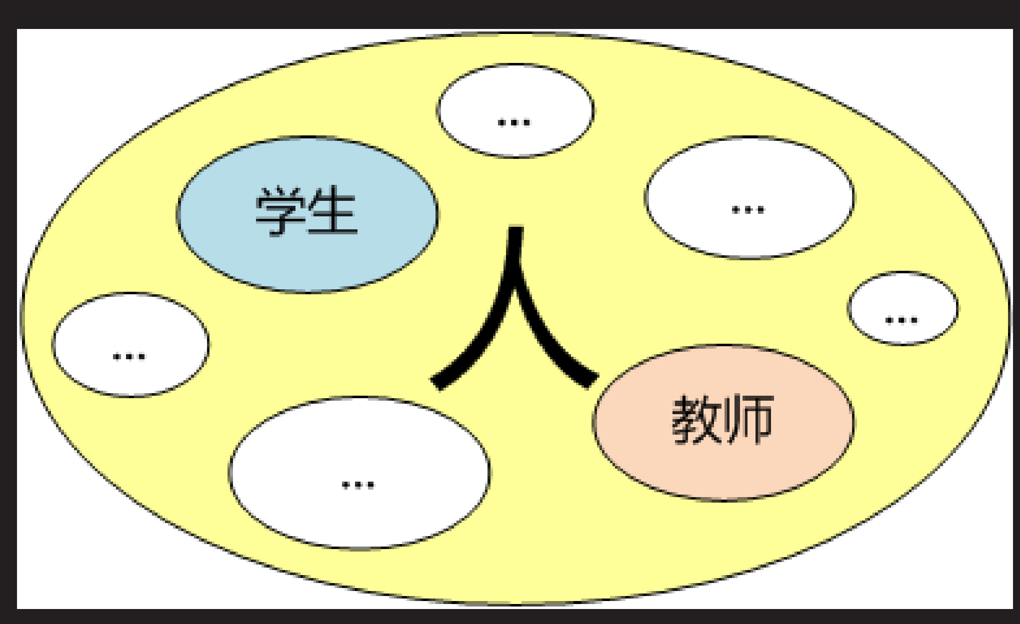


皆然性

逻辑结构

- 子类对象任何时候都可以被当做其基类类型的对象
- 子类的逻辑空间小于基类
- 子类对象 IS A 基类对象

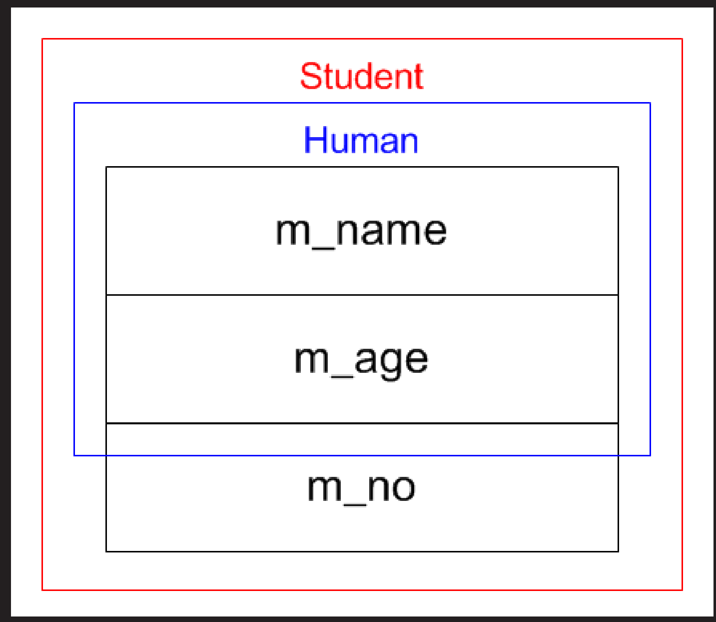
知识讲解



物理结构

- 基类的子类对象是其基类子对象的父对象
- 子类的物理空间大于基类
- 子类对象中的基类子对象与通常意义上的基类对象并没有本质性的区别

知识讲解

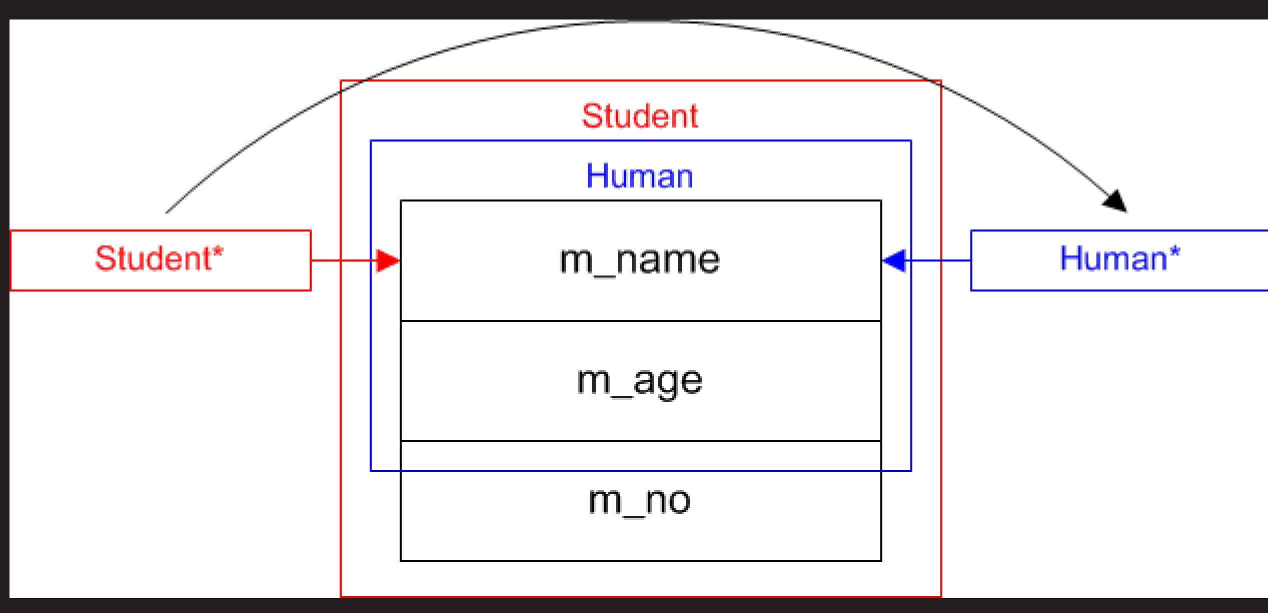


向上和向下造型

从子类到基类

知识讲解

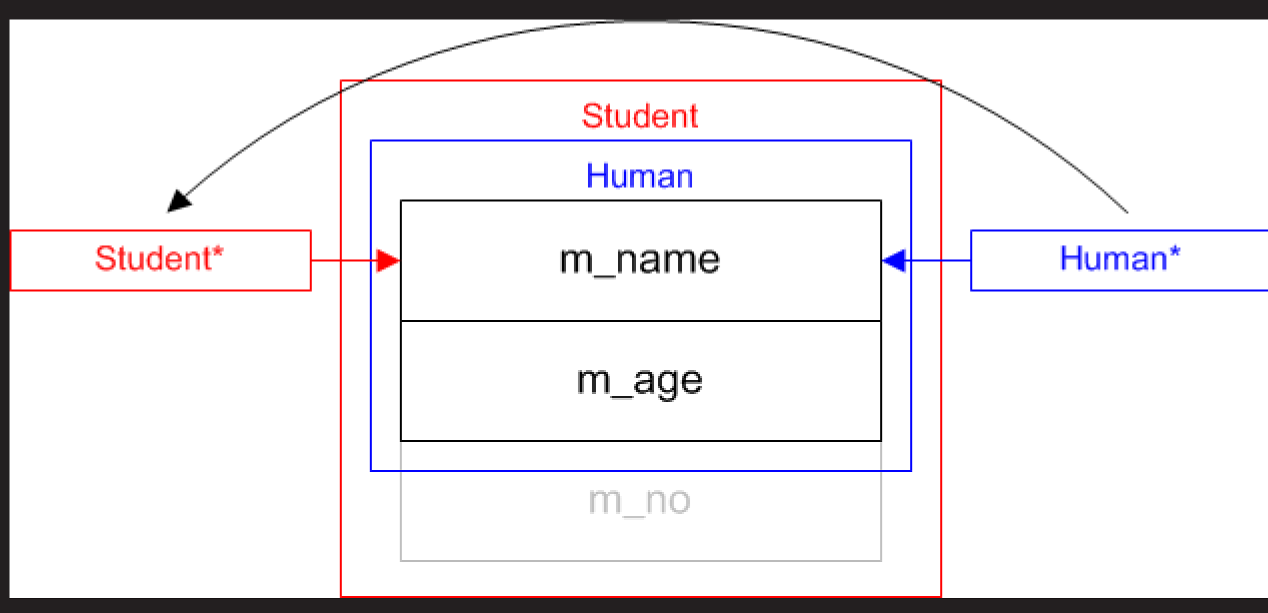
- 子类类型的指针或引用总可以被隐式地转换为其基类类型的指针或引用
- 这种操作性缩小的类型转换，在编译器看来是安全的



从基类到子类

- 基类类型的指针或引用不可以被隐式地转换为其子类类型的指针或引用
- 这种操作性扩大的类型转换，在编译器看来是危险的

知识讲解



类型安全性

知识讲解

- 编译器对类型安全所做的检测，仅仅基于指针或引用本身的数据类型，而与其目标对象的实际类型无关
  - Student student (...);
    Human* phuman = &student;
    Student* pstudent = phuman; // 错误
    Human& rhuman = student;
    Student& rstudent = rhuman; // 错误

类型一致性

知识讲解

- 基类指针或引用实际目标对象的类型，究竟是否与需要转换的指针或引用的目的类型一致，完全由程序员自己判断
- 静态类型转换典型应用：基类指针/引用->子类指针/引用
  - ```
    Student student (…);
    Human* phuman = &student;
    Student* pstudent = static_cast<Student*> (phuman);
    Human& rhuman = student;
    Student& rstudent = static_cast<Student&> (rhuman);
    ```

    ```

# 子类继承基类成员

# 基类的公有、保护和私有成员

知识讲解

- 在子类中或通过子类，可以直接访问基类的所有公有和保护成员，就如同它们是在子类中声明的一样
  - Student student ("张飞", 25);
    student.eat ("饺子");
    student.sleep (6);
- 基类的私有成员在子类中虽然存在却不可见，故无法直接访问
  - string const& Human::name (void) const {
        return m_name; }
  - void Student::learn (string const& course) const {
        cout << m_name << "学" << course << endl; // 错误
        cout << name () << "学" << course << endl; }

# 子类隐藏基类成员

# 子类的名字隐藏

知识讲解

- 尽管基类的公有和保护成员在子类中直接可见，但仍然可以在子类中重新定义这些名字，子类中的名字会隐藏所有基类中的同名定义
  - class Human {
        void eat (string const& food) { ... }
    };
  - class Student : public Human {
        int eat;
    };
  - Student student ("张飞", 25);
    student.eat ("饺子"); // 错误

# 作用域限定操作符

知识讲解

- 如果需要在子类中或通过子类访问一个在基类中定义却为子类所隐藏的名字，可以借助作用域限定操作符“::”实现
  - class Human {
        void eat (string const& food) { ... }
    };
  - class Student : public Human {
        int eat;
    };
  - Student student ("张飞", 25);
    student.Human::eat ("饺子");

# 人、学生和教师

【参见：TTS COOKBOOK 】

课堂练习

- 人、学生和教师

# 继承方式与访问控制

继承方式与访问控制

- 继承方式影响访控属性
  - 访控限定符与访控属性
  - 继承方式的影响范围
  - 继承方式的作用场合

# 继承方式影响访控属性

# 访控限定符与访控属性

- 访控限定符规定了一个类的特定成员，是否具有被从类的内部、类的子类，以及类的外部进行访问的能力

知识讲解

| 访问控制限定符 | 访问控制属性 | 内部 | 子类 | 外部 | 友元 |
| --- | --- | --- | --- | --- | --- |
| public | 公有成员 | OK | OK | OK | OK |
| protected | 保护成员 | OK | OK | NO | OK |
| private | 私有成员 | OK | NO | NO | OK |

# 继承方式的影响范围

- 基类中的公有、保护和私有成员，在其公有、保护和私有子类中的访问控制属性，会因继承方式而异

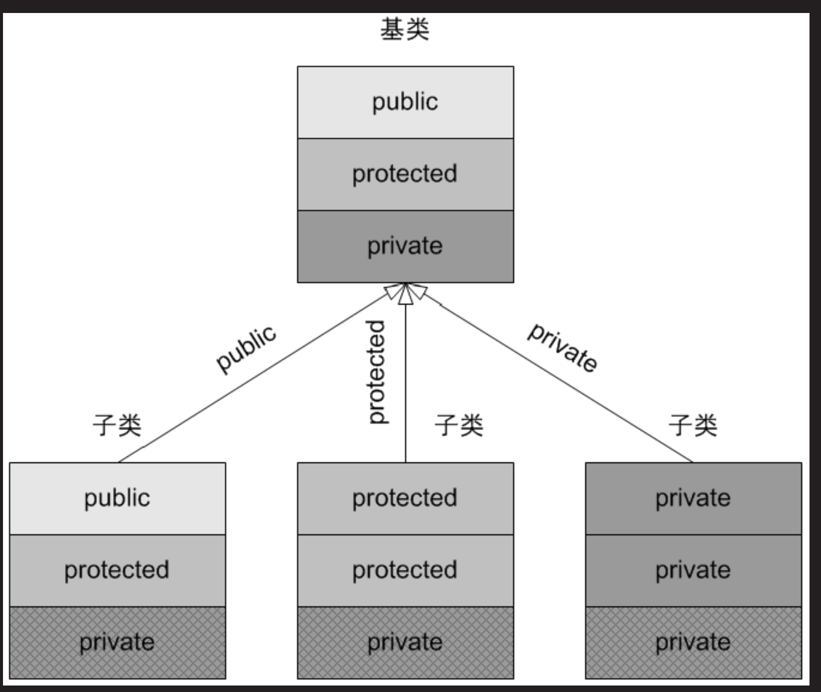
知识讲解

| 基类中的 | 在公有子类中 变 成 | 在保护子类中 变 成 | 在私有子类中 变 成 |
|---|---|---|---|
| 公有成员 | 公有成员 | 保护成员 | 私有成员 |
| 保护成员 | 保护成员 | 保护成员 | 私有成员 |
| 私有成员 | 私有成员 | 私有成员 | 私有成员 |

# 继承方式的作用场合

- 当通过子类访问其所继承的基类成员时，需要考虑继承方式对访问控制属性的影响

知识讲解




```
-------------------------------+---
class A |
{ |
public: |
    int m_pub; |
protected: |
    int m_pro; |
private: |
    int m_pri; |
}; |
-------------------------------+--------------+---------------+------------------
class B : | public A | protected A | private A
{   +--------------+---------------+------------------
    void foo (void) | | |
    { | | |
        m_pub = 10; | Ok | Ok | Ok
        m_pro = 10; | Ok | Ok | Ok
        m_pri = 10; | No | No | No
    } | | |
}; | | |
-------------------------------+--------------+---------------+------------------
class C : | public/protected/private B
{   +--------------+---------------+------------------
    void foo (void) | | |
    { | | |
        m_pub = 10; | Ok | Ok | No
        m_pro = 10; | Ok | Ok | No
        m_pri = 10; | No | No | No
    } | | |
}; | | |
-------------------------------+--------------+---------------+------------------
int main (void) { | | |
    B b; | | |
    b.m_pub = 10; | Ok | No | No
    b.m_pro = 10; | No | No | No
    b.m_pri = 10; | No | No | No
    return 0; | | |
} | | |
-------------------------------+--------------+---------------+------------------
```

公有继承、保护继承和私有继承

【参见：TTS COOKBOOK 】

课堂练习

- 公有继承、保护继承和私有继承

总结和答疑