C++编程(13)

唐晓晟
北京邮电大学电信工程学院

第15章 类层次结构

- 引言和概述
- 多重继承
- 访问控制
- 运行时类型信息
- 指向成员的指针
- 自由存储
- 忠告

15.1 引言和概述

- 本章讨论派生类和虚函数如何与其他语言功能相互作用，例如访问控制、名字查找、自由存储管理、构造函数、指针和类型转换等

15.2 多重继承

- 一个类可以有多个直接基类，采用多个直接基类的情况通常称为多重继承
- 假设Satellite从Task和Displayed多重继承，那么在实际应用中，就可以将一个Satellite传递给那些期望Task或者Displayed的函数

15.2.1 歧义性解析

```
class Task{
    //...
    virtual debug_info* get_debug();
};
class Displayed {
    //...
    virtual debug_info* get_debug();
};
void f(Satellite* sp){
    debug_info* dip = sp->get_debug();
    // 错误，歧义
    dip = sp->Task::get_debug();
    dip=sp->Displayed::get_debug();
}
```

通过左边这种明确写出的方式消除歧义性显得比较混乱，最好是通过在派生类里定义新函数的方法消除这类问题

例如：在Satellite中定义get_debug()函数即可消除此类问题

15.2.2 继承和使用说明

- 重载解析的使用不会跨越不同类的作用域(7.4)，特别地，来自不同基类的函数之间的歧义性不能基于参数类型完成解析

```
class Task{ void debug(double p); };
class Displayed{ void debug(int v); };
class Satellite:public Task, public Displayed{/*...*/}
// 注意：Satellite中没有定义debug()
void g(Satellite* p){
    p->debug(1); // 歧义，错误
    p->Task::debug(1); // ok
    p->Displayed::debug(1); // ok
}
```

继承和使用说明示例

- 假如在不同基类中使用同样名字是一种精心筹划的设计决策，可以如下解决

```
class A{ public: int f(int); char f(char); };
class B{ public: double f(double); };
class AB:public A, public B{
public:
    using A::f;
    using B::f;
    char f(char);
    // 遮蔽A::f(char)
    AB f(AB);
};
```

```
void g(AB& ab){
    ab.f(1); // A::f(int)
    ab.f('a');// AB::f(char)
    ab.f(2.0);// B::f(double)
    ab.f(ab); // AB::f(AB)
}
```

15.2.3 重复的基类

- 一个类两次作为基类是可能的，例如：Task和Displayed都是从Link派生而来，在一个Satallite中就会出现两个Link
- 当引用Link中的元素时，注意必须加以限定

```
struct Link { Link *next; };
void mess_with_links(Satellite* p){
    p->next = 0; // 错误，歧义性
    p->Link::next = 0; // 错误，歧义性
    p->Task::next = 0; // ok
    p->Displayed::next = 0; // ok
}
```

15.2.3.1 覆盖

- 重复基类中的虚函数可以由派生类里的函数覆盖

```
class Storable{
    public: virtual void write() = 0; virtual void read()=0;
    virtual const char* get_file() = 0; };
class Transimitter: public Storable{ public: void write();};
class Receiver:public Storable{ public: void write(); };
class Radio:public Transimitter, public Receiver{
    void write(); };
void Radio::write(){
    Transmitter::write();
    Receiver::write();
}// 一种典型应用方式，覆盖函数首先调用基类中的版本，然后做
  // 有关派生类的特殊工作
```

15.2.4 虚基类

```
class Storable{
public:
    Storable(const char* s);
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
private:
    const char* store;
    Storable(const Storable&);
    Storable& operator=(const Storable&);
};
class Transmitter:public virtual Storable{
    public: void write();  };
class Receiver:public virtual Storable{
    public: void write();  };
```

```
class Radio:
    public Transmitter,
    public Receiver{
public:
    void write();
};
```

Storable稍加修改，继承方式需要改变，以便能处理一个对象的被存储了多个副本的共享问题

解决方法是使用虚基类

15.2.4.1 用虚基类的程序设计

- 在为存在虚基类的类定义函数时，一般来说，程序员并不知道这个基类是与其他派生类共享的
- 在实现某种服务，其中要求调用基类的某个函数恰好一次时，这种情况就可能引起问题

用虚基类的程序设计示例

```
class A{ /*...*/ }; // 无构造函数
class B{ public: B(); }; // 默认构造函数
class C{ public: C(int); }; // 无默认构造函数
class D:virtual public A, virtual public B, virtual public C{
    D(){/*...*/} // 错误，没有调用C的构造函数
    D(int i):C(i){/*...*/}; // ok
};
```

语言保证对于一个虚基类的构造函数将调用恰好一次
虚基类的构造函数将从最终派生类的构造函数里调用，而且要在派生类的构造函数之前调用

用虚基类的程序设计示例

```
class Window{    // 拥有基本功能
    virtual void draw(); };
class Window_with_border:
    public virtual Window{
    // 边框功能
    void own_draw(); // 显示边框
    void draw();
};
class Window_with_menu:
    public virtual Window{
    // 菜单功能
    void own_draw(); // 显示菜单
    void draw();
};// own_draw()不必是虚函数，
它只是被draw()调用，而draw()知
道调用哪一个
```

```
class Clock:
    public Window_with_border,
    public Window_with_menu
{
    // 时钟功能
    void own_draw();
    void draw();
};
```

用虚基类的程序设计示例

```
void Window_with_border::draw(){
    Window::draw();
    own_draw();
}

void Window_with_menu::draw(){
    Window::draw();
    own_draw();
}

void Clock::draw(){
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw();
}
```

借助own_draw()函数写出draw()函数，可以使任何对draw()的调用者都只能让Window::draw()被调用一次，做到这些并不依赖于到底对哪种Window调用draw()

15.2.5 使用多重继承

- 多重继承的最简单最明显的应用，就是利用它将两个原本不相干的类“粘合”起来，作为第三个类的实现的一部分
- 回顾以前讲到过的BB_ival_slider的(多重继承)实现方式，其一个基类作为公用的抽象基类，提供界面，另外一个则是受保护的具体类，提供细节
- 多重继承也使兄弟类之间能够共享信息，而又不会在程序里引进对同一基类的依赖性，通常被称为“钻石型继承diamond-shaped inheritance”，如刚才提到的Radio和Clock

使用多重继承

- 如果虚基类或者由虚基类直接派生的类是抽象类，钻石型继承将特别容易控制
- 将此思想进一步推广，得到的逻辑结论就是，组成应用的界面的抽象类的所有派生都**应该**是虚的，这是最符合逻辑、最具有一般性、也最灵活的解决方案
- 但是，由于历史的原因，作者没有这么设计

15.2.5.1 覆盖虚基类的函数

- 派生类可以覆盖其直接虚基类或者间接虚基类中的虚函数，特别地，两个不同的类也可能覆盖了来自虚基类的不同虚函数
- 不同的派生类可能覆盖其虚基类中的同一个函数，这当且仅当某个覆盖类是从覆盖此函数的类派生时，才可以这么做
- 如果两个类覆盖了同一个虚基类中的函数，但是它们又互不覆盖，这个类层次结构就是错误的
- 一个为虚基类提供部分实现(但是并非全部实现)的类通常被称为“混入类”(mixin.)

15.3 访问控制

- 类中的一个成员可以是private,protected或者public
- private:它的名字只能由其声明所在类的成员函数和友元使用
- protected:它的名字只能由其声明所在类的成员函数和友元，以及由该类的派生类的成员函数和友元使用
- public:它的名字可以由任何函数使用

15.3.1 保护成员

- 派生类只能访问它这种类型对象基类中的保护成员，这样能防止由于一个派生类破坏了属于另一个派生类的数据而产生的微妙错误

```
class Buffer{
    protected:    char a[128];
};
class Linked_buffer:public Buffer{/*...*/};
class Cyclic_buffer:public Buffer{
    void f(Linked_buffer* p){
        a[0] = 0; // 可以，自己的保护成员
        p->a[0] = 0; //  不可以，其他类型的保护成员
    }
};
```

15.3.1.1 保护成员的使用

- 简单的public/private模型就能很好的满足具体类型概念的需要，派生的引入使得这种模型无法迎合派生类的特殊需要
- 声明一些protected数据成员通常都是设计错误，将位于一个公共类的大量数据提供给派生类使用，将使这些数据容易遭到破坏
- private通常使更好的选择，也使默认情况
- 以上所有批评对于protected函数都不重要，protected是描述供派生类使用的操作的极好方式

15.3.2 对基类的访问

- 象成员一样，基类也可以是private, protected或者public

```
class X:public B{};
class Y:protected B{};
class Z:private B{};
```

- public派生使得派生类称为基类的一个子类型，protected在经常需要进一步派生的类层次结构中非常有用，private则提供更强的限定

对基类的访问

- 考虑从基类B派生出的类D
- 如果B是private基类，那么它的public和protected成员只能由D的成员函数和友元访问，只有D的成员和友元能将D*转换到B*
- 如果B是protected基类，那么它的public和protected成员只能由D的成员函数和友元、以及由D派生出的类的成员函数和友元访问。只有D的成员和友元以及由D派生出的类的成员和友元能将D*转换到B*
- 如果B是public基类，那么它的public成员可以由任何函数使用，此外，它的protected成员能由D的成员函数和友元，以及由D派生出的类的成员函数和友元访问，任何函数都能将D*转换到B*

15.3.2.1 多重继承和访问控制

- 在一个派生类的继承层次中，如果一个名字或者基类可以从多条路径到达，那么若有一条路径使它能够访问，它就是可访问的

```
struct B{
    int m;
    static int sm;
};

class D1:public virtual B{/*...*/};
class D2:public virtual B{/*...*/};
class DD:public D1, private D2{/*...*/};

DD* pd = new DD;
B* pb = pd;      // 可以：通过D1访问
int i1 = pd->m;// 可以：通过D1访问
```

多重继承和访问控制示例

- 如果一个实体可以通过多条途径到达，我们还是可能无歧义地引用它

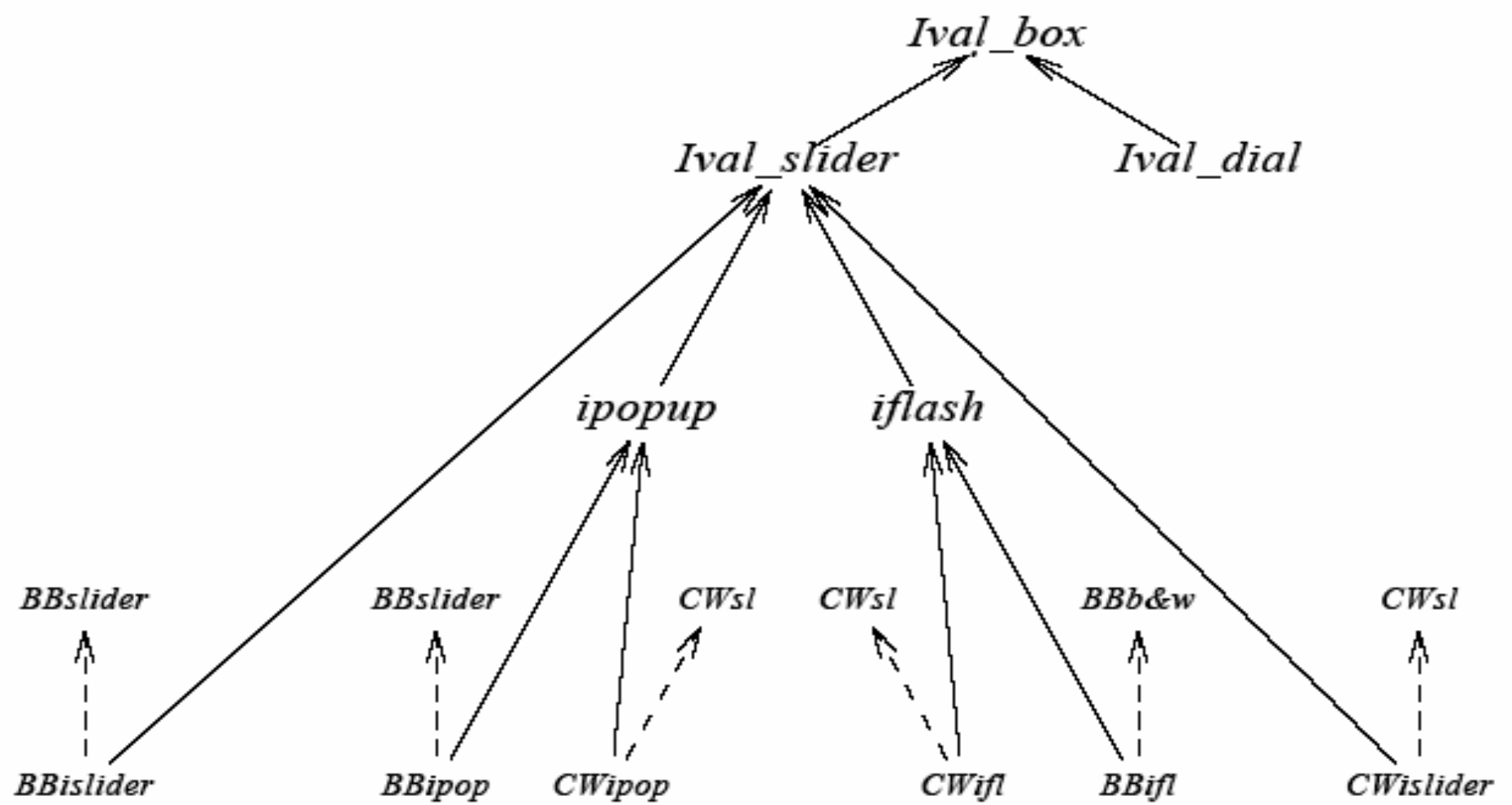
```
struct B{
    int m;
    static int sm;
};

class X1:public B{/*...*/};
class X2:public B{/*...*/};
class XX:public X1,public X2{/*...*/};

XX* pxx = new XX;
int i1 = pxx->m; // 错误：歧义
int i2 = pxx->sm; // 可以：在XX里只有B::sm
```

15.3.2.2 使用声明和访问控制

- 不能通过使用声明(using)取得对更多信息的访问权，using语句只是一种能使信息的使用更方便的机制

```
class B{
    private: int a;
    protected: int b;
    public: int c;
};
class D:public B{
public:
    using B::a; // 错误：B::a为private
    using B::b; // 可以：使B::b在整个D中可以直接使用
};
```

15.4 运行时类型信息


Ival_box
Ival_slider
Ival_dial
ipopup
iflash
BBslider
BBslider
CWsl
CWsl
BBb&w
CWsl
BBislider
BBipop
CWipop
CWifl
BBifl
CWislider

运行时类型信息

- 考虑以前设计过的Ival_box类层次结构
- 一般使用方式是将Ival_box的派生类生成的对象指针传递给一个需要Ival_box类型指针的控制显示屏系统进行绘制，然后，某些活动后，系统将对象送回应用程序，但是，用户系统对Ival_box一无所知，这当然是必要的，也是正确的，但是，我们确实会遗失掉送给系统而后又被送还我们的对象的类型

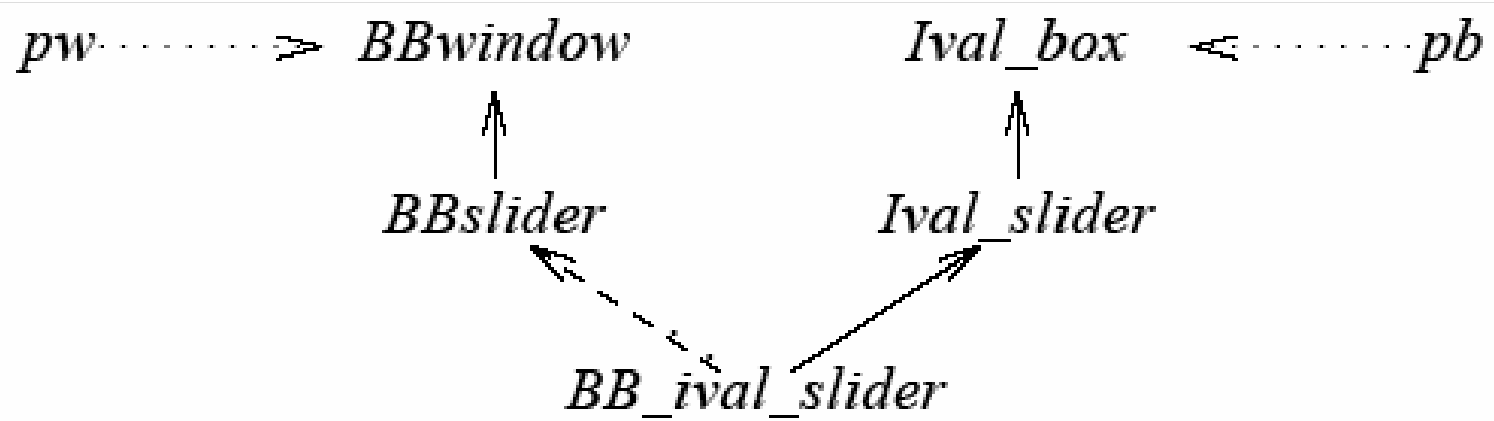
运行时类型信息示例

- 要寻回一个对象遗失的类型，需要能够以某种方式去向对象询问其类型，dynamic_cast做的就是这类事情
- 注意，在下例中，pw的实际类型应该是某种特殊的Ival_box，比如说Ival_slider或者BBslider等

```
void my_event_handler(BBWindow* pw){
    if(Ival_box* pb = dynamic_cast<Ival_box*>(pw)){
        pb->do_something();
    } // 若pw指向一个Ival_box，则do_something()
    else{
        //...
    }
}
```

运行时类型信息示例

- 动作pb = dynamic_cast<Ival_box*>(pw)可以表示为



- 出自pw和pb的箭头表示的是被传递的对象指针，其他箭头表示被传递对象中的不同部分之间的继承关系
- 在运行时对类型信息的使用一般称为运行时类型信息(RTTI RunTime Type Information)

运行时类型信息示例

- 从基类到派生类的强制转换通常被称为向下强制(downcast)，因为一般画继承树时，根在上面
- 从派生类向基类的强制转换称为向上强制(upcast)
- 从一个基类向其兄弟类的强制，例如上图中从BBwindow到Ival_box，称为交叉强制(crosscast)

15.4.1 dynamic_cast

- dynamic_cast与以前讲到过的static_cast类型，都是有两个参数

```
class BB_ival_slider:public Ival_slider, protected BBslider{};
void f(BB_ival_slider* p){
    Ival_slider* pil = p; // ok
    Ival_slider* pi1 = dynamic_cast<Ival_slider*>(p); // ok
    BBslider* pbb1 = p; // 错误：BBslider是保护的基类
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);
    // 可以：pbb2变成0
}
// 上述例子说明：dynamic_cast也不能违背对private和protected基类的保护
```

dynamic_cast

- dynamic_cast的专长是处理那些编译器无法确定转换正确性的情况
- dynamic_cast<T*>(p)将查看p指向的对象，如果这个对象属于类T或者有唯一的类型为T的基类，将返回指向该对象的类型T*的指针；否则就返回0(要求该转换能唯一确定一个对象，否则转换失败，返回0)

dynamic_cast示例

```
class My_slider:public Ival_slider{// 多态基类(有虚函数)
};
class My_date:public Date{ // 基类不是多态类(无虚函数)
};
void g(Ival_box* pb, Date* pd){
    My_slider* pd1 = dynamic_cast<My_slider*>(pb); // ok
    My_date* pd2 = dynamic_cast<My_date*>(pd);
    // 错误：Date不是多态类
}
```

- dynamic_cast要求一个到多态类型的指针或者引用，以便做upcast或者crosscast
- dynamic_cast的目标类型不必是多态的
- 到void*的dynamic_cast可以用于确定多态类型的对象的起始地址

15.4.1.1 引用的dynamic_cast

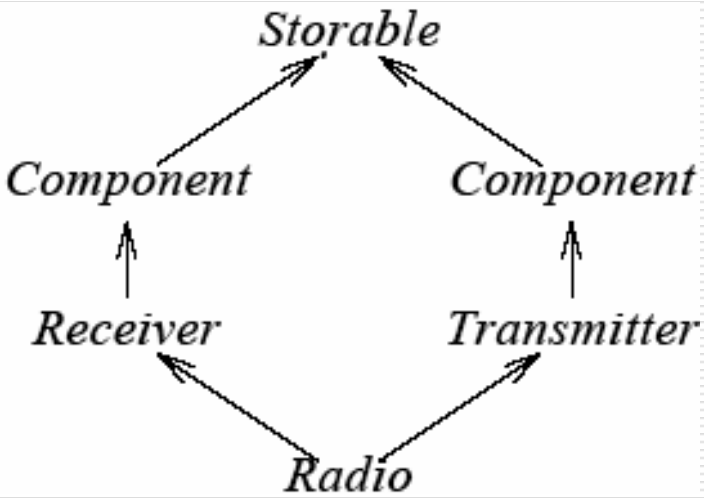
- dynamic_cast<T*>(p)时，对返回只必须检查是否为0，这表示一个询问：“p所指的对象的类型是T吗？”
- dynamic_cast<T&>(r)，不是询问而是断言：“由r引用的对象的类型是T”，其转换结果隐式地由dynamic_cast本身的实现去检查，如果引用的对象不具有所需要的类型，就会抛出一个bad_cast异常

引用的dynamic_cast示例

```
void f(Ival_box* p, Ival_box& r){
    if(Ival_slider* is = dynamic_cast<Ival_slider*>(p)){
        // p是指向一个Ival_slider吗？
    }
    else{ // *p不是一个slider
    }
    Ival_slider& is = dynamic_cast<Ival_slider&>(r);
    // r引用一个Ival_slider!
}
void g(){
    try { f(new BB_ival_slider, *new BB_ival_slider); // ok
          f(new BBdial, *new BBdial); // 抛出bad_cast异常
    }
    catch(bad_cast) {/*…*/}
}
```

15.4.2 在类层次结构中漫游

```
class Component:public virtual Storable{/*...*/};
class Record:public Component{/*...*/};
class Transmitter:public Component{/*...*/};
class Radio:public Receiver, public Transmitter{/*...*/};

void h1(Radio& r){
    Storable* ps = &r;
    Component* pc =
    dynamic_cast<Component*>(ps);
} // pc = 0
```



15.4.2.1 静态和动态强制(casts)

- dynamic_cast能从多态性的虚基类强制到某个派生类或者兄弟类，而static_cast不检查被强制的对象，所以它做不到这些

```
void g(Radio& r){
    Receiver* prec = &r; // Receiver是Radio的常规基类
    Radio* pr = static_cast<Radio*>(prec); // 可以，不检查
    pr = dynamic_cast<Radio*>(prec); // 可以，运行时检查

    Storable* ps = &r;  // Storable是Radio的虚基类
    pr = static_cast<Radio*>(ps); // 错误：不能从虚基类强制
    pr = dynamic_cast<Radio*>(ps); // 可以：运行时检查
}
```

静态和动态强制(casts)

- dynamic_cast要求多态性的操作对象，这是因为在非多态对象里没有存储有关的信息，而从一个对象触发，找到以它作为基类子对象的那些派生类对象，需要这种信息
- 之所以还存在使用static_cast进行的类层次结构的漫游，这是因为存在着数以百万计的代码是在dynamic_cast可以使用之前写出的，大都依靠C风格强制来保证合法性，常常遗留着隐蔽的错误
- 因此，尽管dynamic_cast有一些运行之外的开销，在所有可能的地方，还是应该去用更安全的dynamic_cast

静态和动态强制(casts)示例

- 编译器不能对由void*所指向的存储提供任何保证，这意味着dynamic_cast不能从void*出发进行强制，因为它必须去查看对象，以便确定其类型
- 这种情况需要使用static_cast
- dynamic_cast和static_cast都遵守const和访问控制规则

```
Radio* f(void* p){
    Storable* ps = static_cast<Storable*>(p);
    // 相信程序员！
    return dynamic_cast<Radio*>(ps);
}
```

15.4.3 类对象的构造和析构

- 一个类对象并不简单地就是一块存储
- 类对象是通过其构造函数从“原始存储”中构筑起来的，并通过其析构函数的执行使它重归于“原始存储”
- 构造使一个自下而上的过程，析构则自上而下
- 一个类对象也就是在它能得以构造或析构的意义上才称其为对象

15.4.4 typeid和扩展的类型信息

- 有时候，需要知道一个对象的确切类型，此时，可以使用typeid运算符，它取得一个对象，该对象代表着对应运算对象的类型
- typeid返回一个到标准库类型type_info的引用，该类型在头文件<typeinfo>里面定义
- typeid()经常被用于找出由一个引用或者指针所引用的对象的确切类型

typeid和扩展的类型信息

- 如果typeid是个函数，其声明大致如下：

```
class type_info;
const type_info& typeid(type_name) throw();
const type_info& typeid(expression) throw(bad_typeid);
```

- 如果一个多态类型的指针或者引用的操作对象的值是0，typeid()将抛出一个bad_typeid异常
- 如果typeid()的操作对象的类型不是多态的，或者它不是一个lvalue，其结果将在编译时确定

typeid和扩展的类型信息

```
class type_info{
public:
    virtual ~type_info(); // 多态的
    bool operator==(const type_info&)const; //可以比较
    bool operator!=(const type_info&)const;
    bool before(const type_info&)const; // 有序
    const char* name() const; // 类型名
private:
    type_info(const type_info&); // 禁止复制
    type_info& operator=(const type_info&); // 禁止赋值
};
#include <typeinfo>
void g(Component* p){
    cout << typeid(*p).name(); }
```

15.4.4.1 扩展的类型信息

- 典型情况下，找到一个对象的确切类型，只不过是作为获取和使用有关该类型的更详细信息的第一步
- 用户可以使用任何方式来对这类信息进行处理和建立关联

15.5 指向成员的指针

- C++提供了一种能间接引用类成员的功能，指向成员的指针就是一种标识类成员的值，可以将它们想象成位于该类的一个对象里的那个成员的位置，在这里，实现需要负责去处理数据成员、虚函数、非虚函数之间的差异
- 将地址运算符应用到完全限定的类成员名，就能得到指向成员的指针，要声明此类指针，需要使用形式为X::*声明符
- 静态成员不与任何对象相关联，指向静态成员的指针是一个常规指针

指向成员的指针示例

```
class Std_interface{
public:
    virtual void start() = 0;      virtual void suspend() = 0;
    virtual void resume() = 0;     virtual void quit() = 0;
    virtual void full_size() = 0;    virtual void small() = 0;
    virtual ~Std_interface() {}
};
typedef void (Std_interface::*Pstd_mem)();
void f(Std_interface* p){
    Pstd_mem s = &Std_interface::suspend;
    p->suspend();
    (p->*s)();
}
```

15.6 自由存储

- 我们可以重新定义operator new()和operator delete()来取代全局的操作符或者为某个特定的类提供这些操作

```
class Employee{ // operator new和operator delete默认为static
public:  // size_t表示实际分配或者删除的对象的大小
    void* operator new(size_t);
    void operator delete(void*, size_t);
};
void* Employee::operator new(size_t s){
    // 分配s字节的内存，返回一个到它的指针
}
void Employee::operator delete(void* p, size_t s){
    // p指向new分配的内存，delete负责释放它们以便重用
}
```

自由存储示例

```
class Manager:public Employee {
    int level;
};
void f(){
    Employee* p = new Manager;
    // 确切类型丢失
    delete p; // 无法得到正确大小
}
// 解决方法，提供虚析构函数
class Employee {
public:
    void* operator new(size_t);
    void operator delete(void*, size_t);
    virtual ~Employee();
};
```

在Employee里给出了析构函数，能够保证每个它派生出的类都将提供一个析构函数(这样就能保证大小正确)，即使派生类里没有用户定义的析构函数

甚至是空的析构函数也可以

```
Employee::~Employee
{}
```

15.7 忠告

- [1] 利用常规的多重继承表述特征的合并
- [2] 利用多重继承完成实现细节与界面的分离
- [3] 用***virtual***基类表达在类层次结构里对某些类（不是全部类）共同的东西
- [4] 避免显式的类型转换（强制）
- [5] 在不可避免地需要漫游类层次结构的地方，使用***dynamic_cast***
- [6] 尽量用***dynamic_cast***而不是***typeid***

忠告

- [7] 尽量用***private***而不是***protected***
- [8] 不要声明***protected***数据成员
- [9] 如果某个类定义了***operator delete***( )，它也应该有虚析构函数
- [10] 在构造和析构期间不要调用虚函数
- [11] 尽量少用为解析成员名而写的显式限定词，最好是在覆盖函数里用它