15 友元、异常和其他

内容提要

- 友元类、友元成员函数和嵌套类
  - 它们是在其他类（原始类）中声明的类
- C++异常处理提供了处理特殊情况的机制，如果不对其进行处理，将导致程序终止
- RTTI是一种确定对象类型的机制。新的类型转换运算符提高了类型转换的安全性

友元

1 友元

- 原始类并除了能拥有友元函数，也可以将类作为友元
  - 原始类中声明友元函数Fun()。Fun()不是原始类的函数，但可以访问原始类的私有和保护成员
  - 原始类中申明友元类CF，类CF可以直接访问原始类的私有和保护成员
- 原始类可对友元类透明，也可作更严格限制
  - 哪些函数、成员函数或类为友元，是由原始类定义的，而不能从原始类外部强加友元属性
  - 也就是说，访问私有和保护成员的授权，是在原始类内部进行的
- 友元机制并不违背面向对象的思想，反而提高了公有接口的灵活性

1.1 友元类

P15.1 tv.h　P15.2 tv.cpp　P15.3 use_tv.cpp

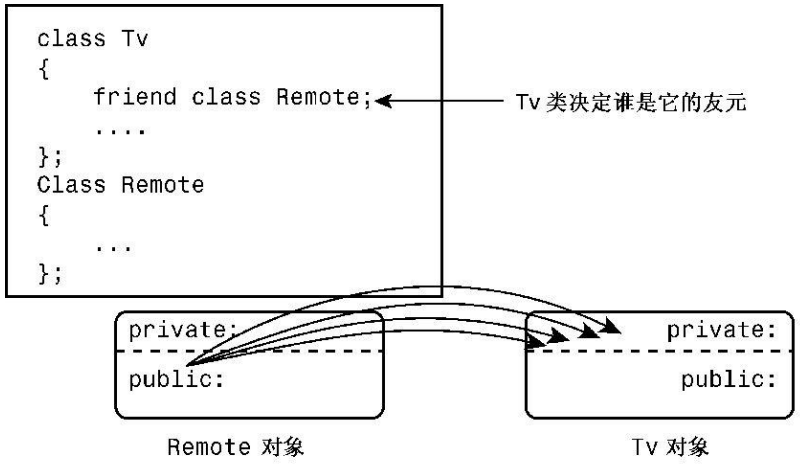
- 什么时候希望一个类成为另一个类的友元呢？
- 考虑电视机和遥控器的关系
  - 不是公有继承的is-a关系
  - 不是私有和保护继承的has-a关系
  - 遥控器可以改变电视机的状态，这表明应将Remote类作为Tv类的一个友元
    - Remote需要访问Tv的私有或者保护成员
- 友元声明可以位于公有、私有或保护部分，其所在的位置无关紧要
- 类友元是一种自然用语，用于表示一些关系
  - 如果不使用某些形式的友元关系，则必须将Tv类的私有部分设置为公有，或者创建一个笨拙的、大型类来包含电视机和遥控器
  - 这种解决方法无法反应这样的事实，即同一个遥控器可用于多台电视机

1.2 友元成员函数

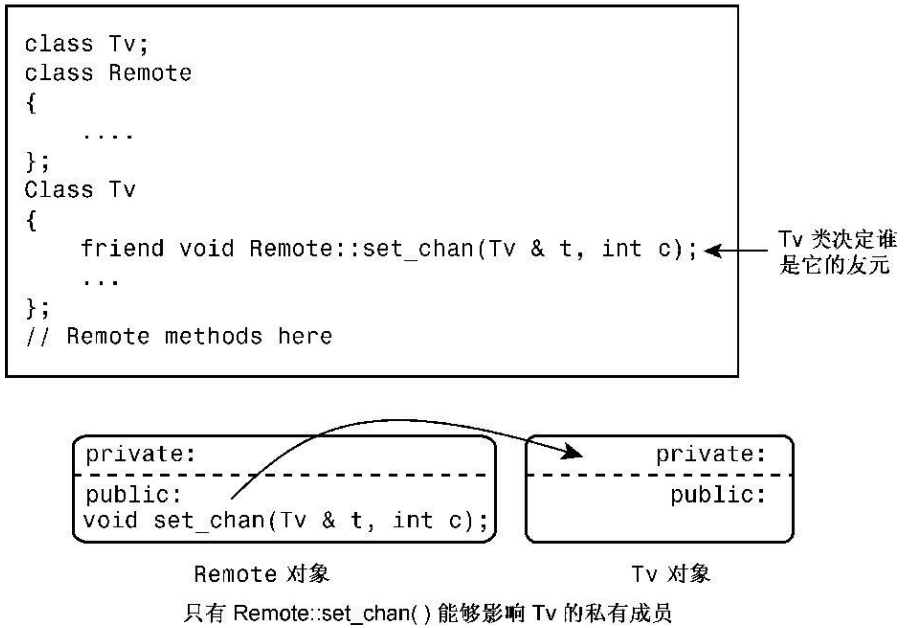
- P15.4 Mm.h
- 可以选择仅让特定的类成员成为另一个类的友元，但要注意声明和定义的排列
- 让 Remote::set_chan()成为 Tv 类的友元的方法是，在 Tv 类声明中将其声明为友元：

```
class Tv{
    friend void Remote::set_chan(Tv &t, int c);
}
```

- 要使编译器能够处理这条语句，它必须知道 Remote 的定义。



Remote的所有方法都可影响 Tv 的私有成员



只有 Remote::set_chan( ) 能够影响 Tv 的私有成员

友元成员函数

- 前向申明

1.3 其他友元关系

- 交互式遥控器
  - 让类彼此成为对方的友元，同样注意声明和定义的顺序

嵌套类

2 嵌套类

- 在另一个类(原始)中声明的类被称为嵌套类
  - Node
- 原始类的成员函数可以使用被嵌套类的对象
  - Queue的成员函数可以实例化和使用Node对象
- 仅当嵌套类声明位于公有部分，才能在包含类的外面使用嵌套类，且必须使用作用域解析运算符
  - Queue::Node node;
- 对类嵌套与包含不同
  - 嵌套只是声明，包含是组成

```
1.  class Queue
2.  {
3.  public:
4.      // class scope definitions
5.      // Node is a nested class definition local to
    this class
6.      class Node
7.      {
8.      private:
9.          Item item;
10.         Node *next;
11.     }
12. }
```

2.1 嵌套类和访问权限

15.5 queuetp.h　P15.6 nested.cpp

- 作用域
  - 如果嵌套类是在包含它的原始类的私有部分声明的，则只有包含它的原始类知道嵌套类
  - 如果嵌套类是在原始类的公有部分声明的，则允许原始类、原始类的派生类以及外部世界使用它
    - `Team::Coach forhire;`
- 类可见后，访问控制规则将决定程序对嵌套类成员的访问控制
  - 与对常规类相同

```
1. class Team
2. {
3. public:
4.     class Coach
5.     {
6.     };
7. };
```

异常

3 异常

- 程序有时会遇到运行阶段错误，导致程序无法正常运行下去
- 程序员都会试图预防这种意外情况。C++异常为处理这种情况提供了一种功能强大而灵活的工具

3.1 调用abort( )

- P15.7 error1.cpp

- abort()函数
  - 实现向标准错误流发送消息，然后终止程序，还返回给操作系统一个随实现而异的值

```
1.  double hmean(double a, double b);
2.  int main(){
3.      double x, y, z;
4.      std::cout << "Enter two numbers: ";
5.      while (std::cin >> x >> y)    {
6.          z = hmean(x, y);
7.      }
8.      std::cout << "Bye!\n";
9.      return 0;
10. }

11. double hmean(double a, double b)
12. {
13.     if (a == -b){
14.         std::abort();
15.     }
16.     return 2.0 * a * b / (a + b);
17. }
```

3.2 返回错误码

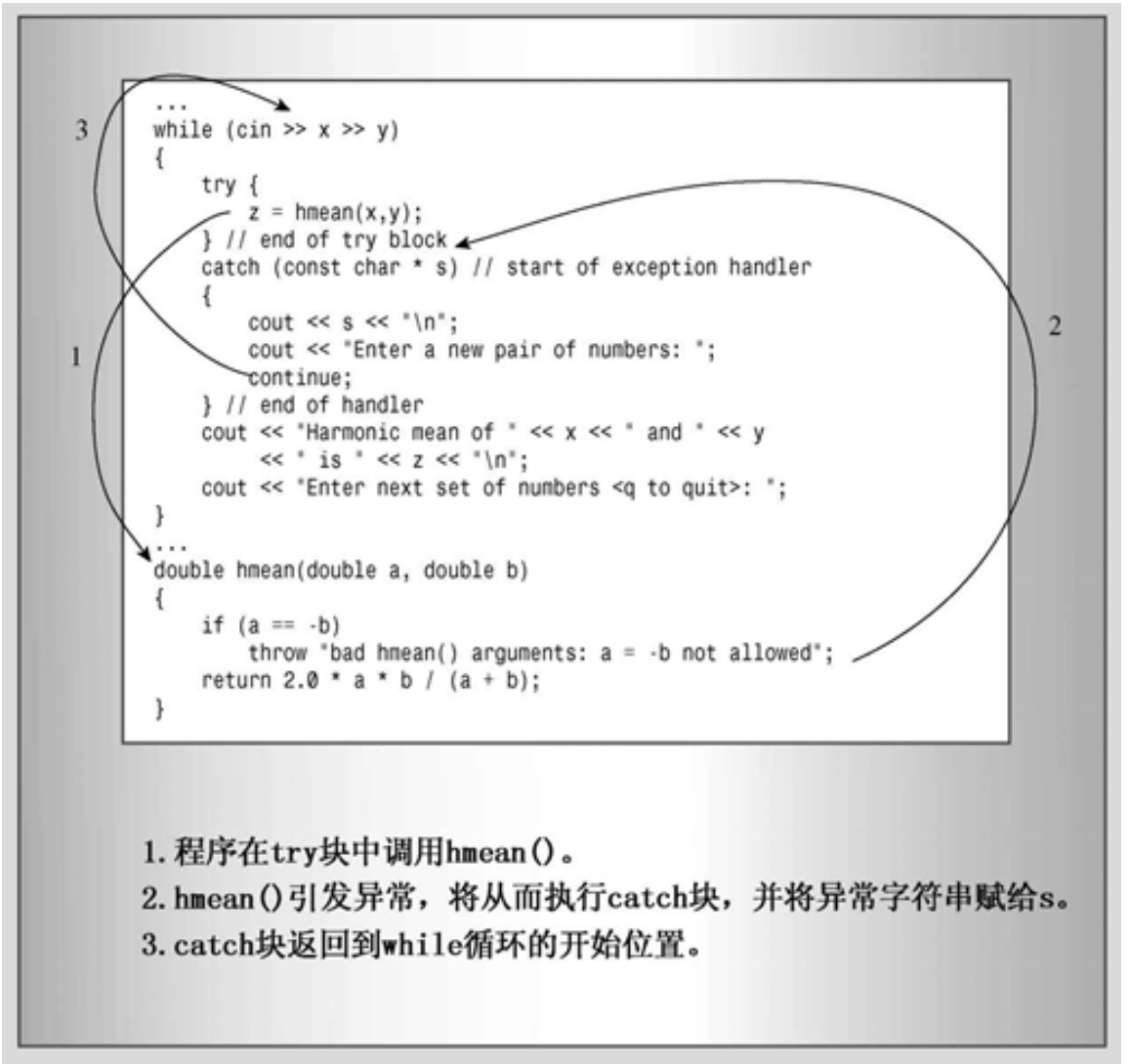
P15.8 error2.cpp

- 使用函数的返回值指出问题将比异常终止更灵活

3.3 异常机制

P15.9 error3.cpp

- C++异常，是对程序运行过程中发生的异常情况的一种响应
  - 引发异常
  - 使用处理程序捕获异常
  - 使用try块
- try块
  - 标识**其中特定的异常可能被激活的**代码块
  - 它后面跟一个或多个catch块
- 代码块
  - 出现问题时用throw抛出异常，带参数
- 异常处理程序(exception handler) catch
  - 捕获异常，根据异常参数进行处理



3.4 将对象用作异常类型

P15.10 exc_mean.h　　P15.11 error4.cpp

- 引发异常的函数throw something，异常参数传递一个对象，其优点
  - 不同异常类型，对应不同情况下的异常
  - 抛出异常，对象可以携带信息

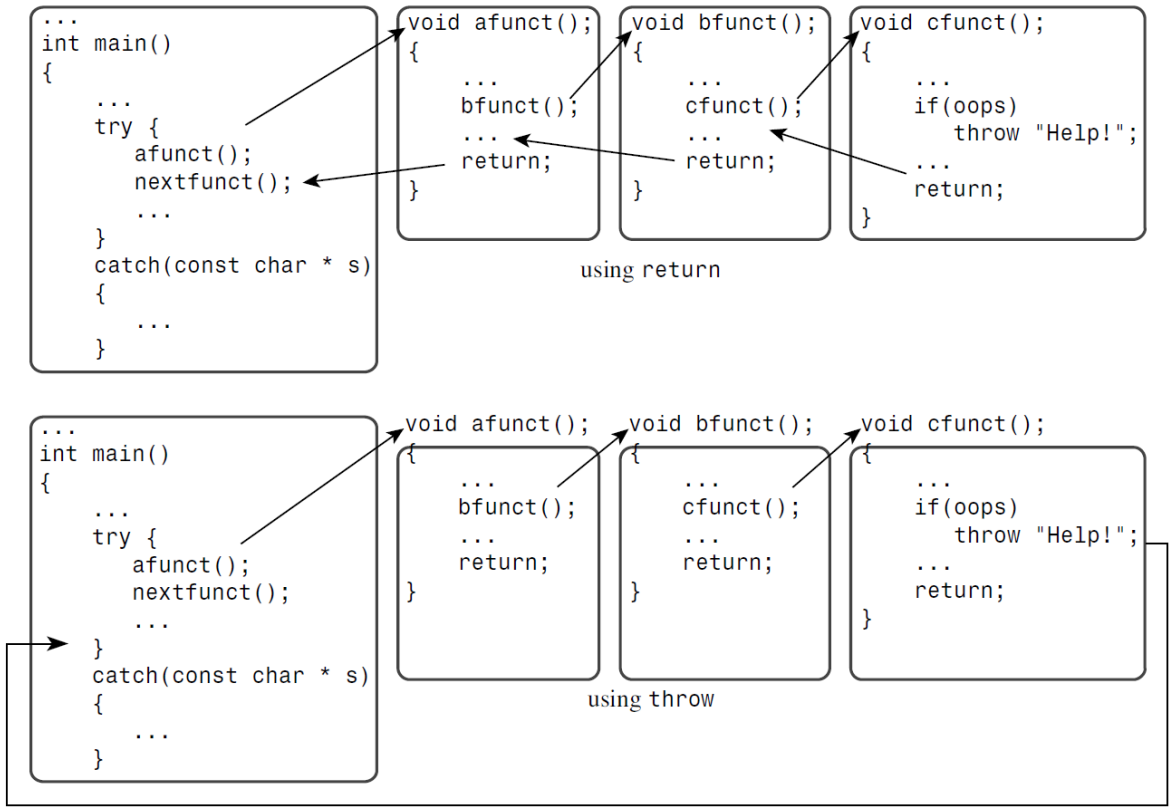
3.5 异常规范和C++11

- 异常规范（exception specification），这是 C++98 新增的一项功能，但 C++11 却将其摒弃了。这意味着 C++11仍然处于标准之中，但以后可能会从标准中剔除，因此不建议使用

3.6 栈解退

P15.12 error5.cpp

- 函数异常终止，程序不会在释放栈第一个返回地址后停止，而是继续释放直到一个try块中的返回地址
- return和throw的区别



3.7 其他异常特性

P15.13 newexcp.cpp

- 引发异常时编译器会创建一个临时拷贝
- 按照异常类继承层次结构排列catch块（子类在前，父类在后）

3.8 exception类

- 可以派生exception类
- C++库定义很多基于exception的异常类型
  - stdexcept异常类
  - bad_alloc异常和new
  - 空指针和new

3.9 异常、类和继承

P15.14 sales.h P15.15 sales.cpp P15.16 use_sales.cpp

- 异常、类和继承以三种方式相互关联
  - 从一个异常类派生出另一个
  - 在类定义中嵌套异常类声明组合异常
  - 嵌套声明本身可被继承，可用作基类

3.10 异常何时会迷失方向

- 在带异常规范函数中引发，必须与规范列表中的异常匹配，否则unexpected exception
- 不是在函数中引发（或者函数没有异常规范），则必须捕获，否则uncaught exception

3.10 有关异常的注意事项

- 在设计程序时就加入异常处理，而不是以后，但会有些缺点
  - 增加程序代码
  - 降低运行速度
  - 与动态内存分配不总能协同

RTTI

4 RTTI

- RTTI是运行阶段类型识别（Runtime Type Identification）的简称
- RTTI 旨在为程序在运行阶段确定对象的类型提供一种标准方式。很多类库已经为其类对象提供了实现这种功能的方式，但由于C++内部并不支待，因此各个厂商的机制通常互不兼容。
- 创建一种 RTTI 语言标准将使得未来的库能够彼此兼容

4.1 RTTI的用途

P15.17 rtti1.cpp P15.18 rtti2.cpp

- 为何要知道类型？可能希望调用类方法的正确版本(包含派生类情况下)
  - 不真正需要知道对象的类型
    - 基类指针或者引用，调用虚函数
  - 需要知道对象的类型
    - 派生类中包含的非继承方法，只有某些对象才可以使用的方法，需要知道对象类型
    - 出于调试目的，跟踪一个生成的对象的类型
- RTTI

4.2 RTTI的工作原理

- RTTI只适用于包含虚函数的类，三个元素
  - dynamic_cast
  - typeid
  - type_info

dynamic_cast

- dynamic_cast运算符，最常用的RTTI组件
  - 不能回答“指针指向的是哪类对象”
  - 但能够回答 “是否可以安全地将对象的地址赋给特定类型的指针”
    - 要调用方法，类型并不一定要完全匹配，定义了方法的虚版本的基类类型就可以
      - dynamic_cast<Type *>(pt)
      - 将指针pt转换为Type类型的指针，如果不能，结果为0，即空指针
- 即使编译器支持RTTI，在默认情况下，它也可能关闭该特性。如来该特性被关闭，程序可能仍能够通过编译，但将出现运行阶段错误
- 也可以将dynamic_cast用于引用，其用法稍微有点不同（尽量用指针！）

typeid运算符和type_info类

- typeid运算符
  - 参数为类名或者对象
  - 返回一个对type_info对象的引用
  - typeid(Magnificent) == typeid(*pg)
- type_info类的实现随厂商而异，但包含一个name()成员，该函数返回一个随实现而异的字符串：通常（但并非一定）是类的名称

类型转换运算符

5 类型转换运算符

P15.19 constcast.cpp

- 在Stroustrup看来， C语言中的类型转换运算符太过松散。C++采取的措施是，更严格地限制允许的类型转换，并添加4个类型转换运算符，使转换过程更规范：
  - dynamic_cast（确认能否进行，更安全）
  - const_cast（转换后不能修改）
  - static_cast（表示知道转换的风险）
  - reinterpret_cast（无关类型的转换）
- 可以根据目的选择一个适合的运算符，而不是使用通用的类型转换。这指出了进行类型转换的原因，并让编译器能够检查程序的行为是否与设计者想法吻合
- High和Low是两个类，ph和pl的类型分别为High*和Low*，则当且仅当Low是High的可访问基类（直接或间接）时，下面的语句才将Low*指针赋给pl：
  - pl = dynamic_cast<Low *> ph;
  - 否则，该语句将空指针赋给pl

6 总结

- 友元使得能够为类开发更灵活的接口
  - 类可以将其他函数、其他类和其他类的成员函数作为友元。在某些情况下，可能需要使用前向声明，需要特别注意类和方法声明的顺序，以正确地组合友元
- 嵌套类是在其他类中声明的类，它有助于设计助手类，即实现其他类，但不必是公有接口的组成部分。
- C++异常机制为处理拙劣的编程事件，提供了一种灵活的方式
  - 引发异常将终止当前执行的函数，将控制权传给匹配的catch块。 catch 块紧跟在try块的后面，为捕获异常，直接或间接导致异常的函数调用必须在try块中。这样程序将执行 catch 块中的代码
- RTTI（运行阶段类型信息）特性让程序能够检测对象的类型
  - dynamic_cast 运算符用于将派生类指针转换为基类指针，其主要用途是确保可以安全地调用虚函数。typeid 运算符返回一个 type_info 对象。可以对两个 typeid的返回值进行比较，以确定对象是否为特定的类型
- 与通用转换机制相比，dynamic_cast、static_cast、const_cast和reinterpret_cast提供了更安全、更明确的类型转换