

C/C++ Programming Language

CS219 Spring

Feng Zheng

2019.05.28



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



Content

- Review
- Friends
- Nested Classes
- Exceptions (object)
- Runtime Type Identification (object)
- Type Cast Operators

Brief Review



Review

- Classes with Object Members
- Private Inheritance
- Multiple Inheritance
- Class Templates



Friends



Friend Classes

- Friend functions?
 - The **extended interface** for a class
 - A common kind of Friend: overloading the << operator (left operand)
- Friends (neither is-a nor has-a)
 - 1. **Any method** of the friend class can access **private** and protected members of the original class
 - 2. Designate **particular** member functions of a class to be friends to another class
 - **Cannot** be imposed from the **outside**
- An example
 - A television and a remote control
 - ✓ **is-a** relationship of public inheritance **doesn't** apply
 - ✓ **has-a** relationship of containment or of private or protected inheritance **doesn't** apply



Friend Declaration

- Run tv.h, tv.cpp, use_tv.cpp
 - The **Remote** methods are implemented by **using the public interface** for the **Tv** class
 - Provide the class with methods for **altering** the settings
 - A remote control should **duplicate the controls** built in to the television
- Friend declaration
 - A friend **declaration** can **appear** in a public, private, or protected section
 - The **location** makes no difference for as a friend but is different for **the devised class** or for **the outside**

```
friend class Remote;
```



Friend Member Functions

- A problem?
 - The only **Remote** method that accesses a **private Tv** member directly is **Remote::set_chan()**, so that's the only method that needs to be a friend

- A **second** solution

- Make **Remote::set_chan()** a friend to the **Tv** class
- Declare it as a **friend** in the **Tv** class declaration

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

- A new problem of circular dependence?

- If **Tv** defined in front, compiler needs to see the **Remote** definition
- But the fact that **Remote** methods mention **Tv** objects

- The **third** solution: **forward** declaration

```
class Tv; // forward declaration
class Remote { ... };
class Tv { ... };
```

Could you use the following arrangement **instead**?

```
class Remote; // forward declaration
class Tv { ... };
class Remote { ... };
```





Friend Member Functions

- Another difficulty remains

Remote \longrightarrow `void onoff(Tv & t) { t.onoff(); }`

- Compiler needs to **have seen** the Tv class declaration at this point
- But the declaration necessarily **follows** the Remote declaration.
- The **fourth** solution
 - Restrict Remote to method declarations and to place the **actual** definitions **after** the Tv class.

```
class Tv; // forward declaration
class Remote { ... }; // Tv-using methods as prototypes only
class Tv { ... };
// put Remote method definitions here
```

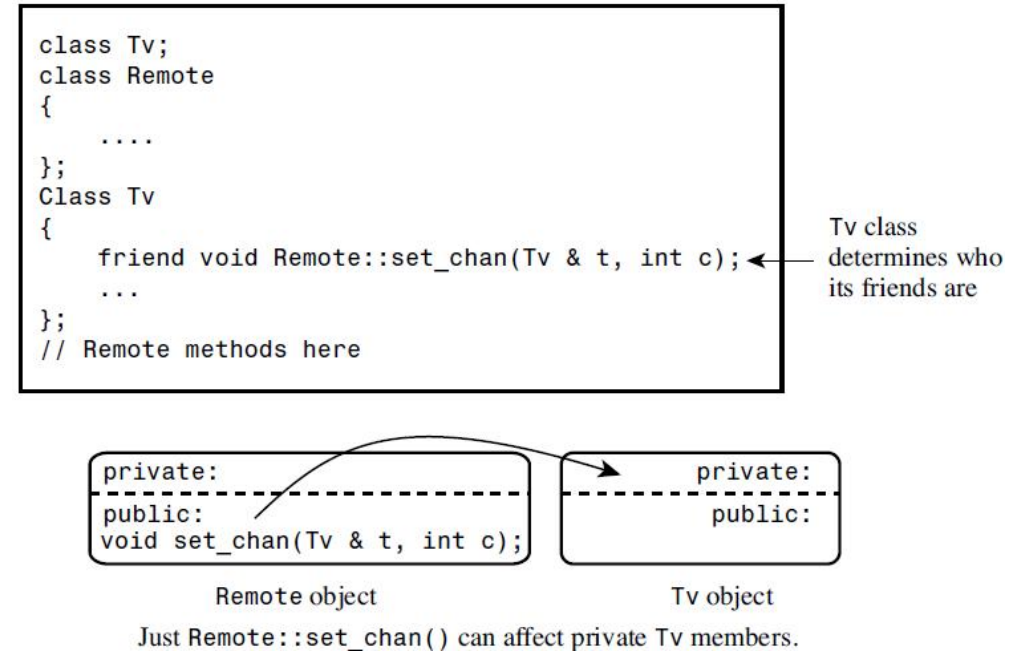
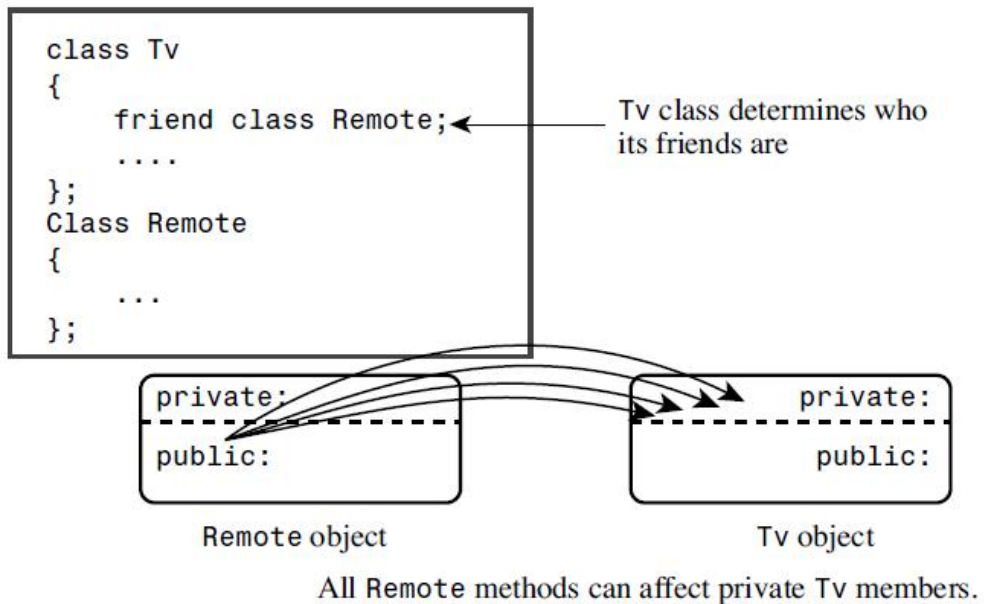
The Remote prototypes look like this:

```
void onoff(Tv & t);
```



Comparison

- **Class** friends versus **class member** friends





Other Friendly Relationships

- 1. Interactive controls

- Make the classes friends to **each other**
- Eg.: the television might activate a **buzzer** in your remote control if your response is wrong
- `Tv::buzz()` method has to be defined outside the `Tv` declaration so that the definition can **follow the Remote declaration**
- If you **don't** want `buzz()` to be **inline**, you need to define it in a separate method **definitions file**

```
class Tv
{
    friend class Remote;
public:
    void buzz(Remote & r);
    ...
};

class Remote
{
    friend class Tv;
public:
    void Bool volup(Tv & t) { t.volup(); }
    ...
};

inline void Tv::buzz(Remote & r)
{
    ...
}
```

→ A problem

One point to keep in mind is that a `Tv` method that uses a `Remote` object can be **prototyped** before the `Remote` class declaration but must be defined after the declaration so that the compiler will have enough information to compile the method.



Other Friendly Relationships

- A problem

- A **function** needs to access **private** data in **two separate classes** while it is **impossible** to be a member function of each class
- It could be a member of one class and a friend to the other

- **2. Shared friends (better solution)**

- Eg.: Probe class represents some sort of programmable measuring device and an Analyzer class represents some sort of programmable analyzing device. Each has an internal clock, and you would like to be able to **synchronize the two clocks**

```
class Analyzer; // forward declaration
class Probe
{
    friend void sync(Analyzer & a, const Probe & p); // sync a to p
    friend void sync(Probe & p, const Analyzer & a); // sync p to a
};
class Analyzer
{
    friend void sync(Analyzer & a, const Probe & p); // sync a to p
    friend void sync(Probe & p, const Analyzer & a); // sync p to a
    ...
};
```

```
// define the friend functions
inline void sync(Analyzer & a, const Probe & p)
{
    ...
}
inline void sync(Probe & p, const Analyzer & a)
{
    ...
}
```

Nested Classes



Nested Classes

- What is the nested class?
 - Place a class declaration **inside another** class
 - ✓ Member functions of the class containing the declaration can **create and use objects** of the nested class
 - ✓ The outside world can use the nested class only if the declaration is in the **public** section
 - Why? Assist in the implementation of other class and to **avoid name conflicts**
 - ✓ Why not a containment?
- Nesting classes is **not** the same as **containment**
 - ✓ Containment: have a class **object** as a member
 - ✓ Nesting class: define a **type** locally to the class
 - ✓ What is the **difference**?

```
class Queue
{
    // class scope definitions
    // Node is a nested class definition local to this class
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i) : item(i), next(0) { }
    };
    ...
};
```



Nested Classes and Access

- Two kinds of access
 - **Where** a nested class is declared controls the **scope** of the nested class
 - The **public**, **protected**, and **private** sections of a nested class **provide** access control to class members
- Scope
 - In a **private** section, it is known **only** to that **containing** class
 - In a **protected** section, it is visible to containing class but **invisible** to the outside world. While, a **derived** class would know about it
 - In a **public** section, it is available to the containing class, to derived classes, and to the outside world

Where Declared in Nesting Class	Available to Nesting Class	Available to Classes	
		Derived from the Nesting Class	Available to the Outside World
Private section	Yes	No	No
Protected section	Yes	Yes	No
Public section	Yes	Yes	Yes, with class qualifier



Access Control

- The **same** rules that govern access to a regular class govern access to a nested class
 - A containing class **object** can access only the public **members** of a nested class **object** explicitly
 - The **location** of a class declaration determines the **scope or visibility** of a class
 - The **usual access control rules** (public, protected, private, friend) determine the access a program has to members of the nested class
- Nesting in a **template**

```
template <class Item>
class QueueTP
{
private:
    enum {Q_SIZE = 10};
    // Node is a nested class definition
    class Node
    {
public:
        Item item;
        Node * next;
        Node(const Item & i):item(i), next(0){ }
    };
    Node * front;    // pointer to front of Queue
    Node * rear;     // pointer to rear of Queue
};
```




Nesting in a Template

- Remember class template (template argument)?
 - Templates are a good choice for implementing container classes such as the Queue class

```
// queuetp.h -- queue template with a nested class
#ifndef QUEUETP_H_
#define QUEUETP_H_

template <class Item>
class QueueTP
{
private:
    enum {Q_SIZE = 10};
    // Node is a nested class definition
    class Node
    {
    public:
        Item item;
        Node * next;
        Node(const Item & i):item(i), next(0){ }
    };
    Node * front;      // pointer to front of Queue
    Node * rear;       // pointer to rear of Queue
    int items;         // current number of items in Queue
    const int qsize;   // maximum number of items in Queue
    QueueTP(const QueueTP & q) : qsize(0) {}
    QueueTP & operator=(const QueueTP & q) { return *this; }
```

```
public:
    QueueTP(int qs = Q_SIZE);
    ~QueueTP();
    bool isempty() const
    {
        return items == 0;
    }
    bool isfull() const
    {
        return items == qsize;
    }
    int queuecount() const
    {
        return items;
    }
    bool enqueue(const Item &item); // add item to end
    bool dequeue(Item &item);      // remove item from front
};
```

How to define the method?

```
Queue::Node::Node(const Item & i) : item(i), next(0) { }
```

Exceptions



Rudimentary Options

- An example: **harmonic** mean of two numbers

$$2.0 \times x \times y / (x + y)$$

- Calling **abort()**: **run error1.cpp**
 - Send a **message** such as "abnormal program termination" to the standard error stream and **terminate** the program
 - **Return** an implementation-dependent value that indicates failure to the **operating** system
- Returning an **error code**: **run error2.cpp**
 - **Return** values to **indicate** a problem
 - We have used it in the previous examples



Throw-Catch Mechanism

- An **exceptional** circumstance arises while a program is **running**
- Exception mechanism provide a way to **transfer** control from one part of a program to another
 - Throwing an exception
 - ✓ **throw** keyword indicates the **throwing** of an exception
 - ✓ A throw statement, in essence, is a **jump** (other jump operators??)
 - Catching an exception with a handler
 - ✓ **catch** keyword indicates the **catching** of an exception
 - ✓ Followed by a **type declaration** that indicates the **type of exception**
 - Using a try block
 - ✓ A **try** block identifies a block of code for which **particular exceptions** will be activated
 - ✓ Followed by **one or more** catch blocks
- Run **error3.cpp**



Throw-Catch Mechanism

- Can we use more complex **types**? YES!
- Using objects as exceptions
 - Advantage: use **different exception types** to distinguish among **different functions and situations** that produce exceptions
 - An object can **carry** information with it, and you can use this information to help identify the conditions that caused the exception to be thrown
 - A catch block could use that information to **decide** which course of action to pursue
- Run **exc_mean.h, error4.cpp**
 - Geometric and harmonic means



More Exception Features of Throw-Catch Mechanism

- Differences to the normal function

- One difference

- ✓ A return statement: **transfer** execution to the **calling** function
 - ✓ A throw: **transfer** execution to the **first** function having a **try-catch**

- Second difference

- ✓ The compiler always creates a **copy** when throwing an exception

- The **exception** class

- Define an exception class that C++ uses as a **base** class
 - One **virtual** member function is named **what()**, and it returns a string

```
class problem {...};  
...  
void super() throw (problem)  
{  
    ...  
    if (oh_no)  
    {  
        problem oops;    // construct object  
        throw oops;      // throw it  
    }  
    ...  
}  
...  
try {  
    super();  
}  
catch(problem & p)  
{  
    // statements  
}
```

```
#include <exception>  
class bad_hmean : public std::exception  
{  
public:  
    const char * what() { return "bad arguments to hmean()"; }  
    ...  
};
```



More Exception Features

- The **stdexcept** exception classes
 - The **stdexcept** header file defines **several** more exception classes
 - **logic_error** and **runtime_error** classes
 - ✓ **logic_error family**: **domain_error**, **invalid_argument**, **length_error**, **out_of_bounds**
 - ✓ **runtime_error family**: **range_error**, **overflow_error**, **underflow_error**
- The **bad_alloc** exception and **new**
 - Have **new** throw a **bad_alloc** exception
 - **new** returned a null pointer when it couldn't allocate the memory
- Run **newexcp.cpp**

```
class logic_error : public exception {  
public:  
    explicit logic_error(const string& what_arg);  
    ...  
};
```

```
class domain_error : public logic_error {  
public:  
    explicit domain_error(const string& what_arg);  
    ...  
};
```

```
int * pi = new (std::nothrow) int;  
int * pa = new (std::nothrow) int[500];
```

Runtime Type Identification



What Is RTTI For?

- Runtime type identification (RTTI)
 - One of the more **recent** additions to C++
 - **Isn't** supported by many older implementations
- Why RTTI?
 - Provide a standard way to **determine the type** of object during runtime
 - Allow future libraries to be **compatible** with each other
- How Does RTTI Work?
 - The **dynamic_cast** operator generates a pointer of a **base** type from a pointer of a **derived** type. Otherwise, it returns the null pointer.
 - The **typeid** operator returns a value **identifying** the type of an object.
 - A **type_info** structure **holds** information about a particular type.



RTTI

- The **dynamic_cast** operator

- Safely assign the **address** of an object to a **pointer** of a **particular** type
 - ✓ Invoke the **correct** version of a class **method**
 - ✓ Keep **track** of which **kinds** of objects were generated

```
class Grand { // has virtual methods};  
class Superb : public Grand { ... };  
class Magnificent : public Superb { ... };
```

```
Grand * pg = new Grand;
```

```
Grand * ps = new Superb;
```

```
Grand * pm = new Magnificent;
```

```
Magnificent * p1 = (Magnificent *) pm;           // #1  safe
```

```
Magnificent * p2 = (Magnificent *) pg;           // #2  not safe
```

```
Superb * p3 = (Magnificent *) pm;                // #3  safe
```

```
Superb * pm = dynamic_cast<Superb *>(pg);
```

NULL



RTTI

- The **typeid** operator

- Let you determine whether two objects are the **same** type
- Accept **two** kinds of **arguments**
 - ✓ The **name** of a **class**
 - ✓ An **expression** that evaluates to an **object**
- The typeid operator **returns** a reference to a **type_info object**

- The **type_info** class

- Defined in the **typeinfo** header file
- Overload the **==** and **!=** operators so that you can use these operators to **compare** types

```
typeid(Magnificent) == typeid(*pg)
```

Type Cast Operators



Type Cast Operators

- Select an **operator** that is suited to a particular **purpose**
- Examples
 - None of them make much sense
 - In C, all of them are allowed
- **Four** type cast operators
 - **dynamic_cast**
 - ✓ Allow **upcasts** within a class hierarchy
 - ✓ **is-a** relationship
 - ✓ **Disallow** other casts
 - **const_cast**
 - ✓ Type cast for **const** or **volatile** value
 - ✓ An error if any other aspect of the type is altered
- Run **constcast.cpp**

```
struct Data
{
    double data[200];
};

struct Junk
{
    int junk[100];
};

Data d = {2.5e33, 3.5e-19, 20.2e32};
char * pch = (char *) (&d);    // type cast #1 - convert to string
char ch = char (&d);          // type cast #2 - convert address to a char
Junk * pj = (Junk *) (&d);     // type cast #3 - convert to Junk pointer
```

```
dynamic_cast < type-name > (expression)
```

```
const_cast < type-name > (expression)
```

```
High bar;
const High * pbar = &bar;

...

High * pb = const_cast<High *> (pbar);    // valid
const Low * pl = const_cast<const Low *> (pbar);    // invalid
```



Type Cast Operators

➤ `static_cast`

- ✓ It's valid **only** if `type_name` can be converted **implicitly** to the **same** type that **expression** has, or vice versa
- ✓ Otherwise, the type cast is an **error**

➤ `reinterpret_cast`

- ✓ Do **implementation-dependent** things
- ✓ Cast a **pointer** type to an **integer** type that's large enough to hold the pointer representation
- ✓ **Can't** cast a **pointer** to a **smaller integer** type or to a floating point type
- ✓ **Can't** cast a **function** pointer to a **data** pointer or vice versa

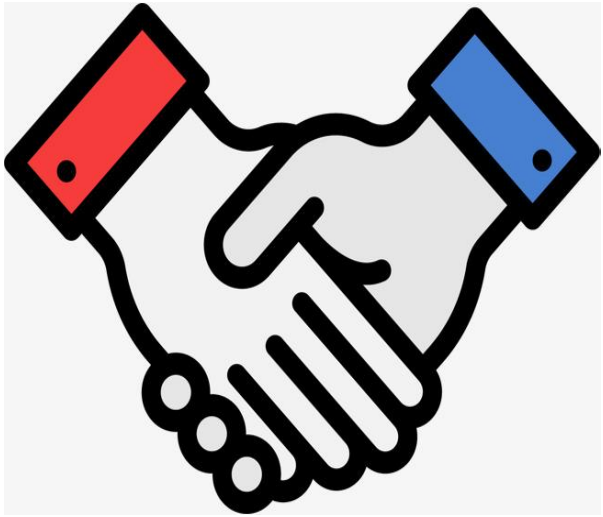
```
static_cast < type-name > (expression)
```

```
High bar;  
Low blow;  
...  
High * pb = static_cast<High *> (&blow);    // valid upcast  
Low * pl = static_cast<Low *> (&bar);        // valid downcast  
Pond * pmer = static_cast<Pond *> (&blow);    // invalid, Pond unrelated
```

High is a **base** class to Low
and that Pond is an
unrelated class

```
reinterpret_cast < type-name > (expression)
```

```
struct dat {short a; short b;};  
long value = 0xA224B118;  
dat * pd = reinterpret_cast< dat *> (&value);  
cout << hex << pd->a;    // display first 2 bytes of value
```



Thanks



zhengf@sustech.edu.cn