Friend Classes

friend Classes

- A class is a friend of another class.
- The friend class can access all members even private members.
- A friend class can be public, protected and private.

```
class Supplier
{
    int storage;
public:
    Supplier(int storage = 1000);
    bool provide(Sniper & sniper)
    {
        // bullets is a private member
        if (sniper.bullets < 20)
        // ...
    }
};
```

```
class Sniper
{
private:
    int bullets;
public:
    Sniper(int bullets = 0);
    friend class Supplier;
};
```

friend.cpp

友元类能访问所有成员，包括私有成员.
友元类可以是 public、protected、private.
修饰规则同理.

Friend Member Functions

类似于友元函数，但声明时要加上类名.

- A single member function of a class is a friend.
- Different from friend functions.
- But very similar to a normal friend function.
- But... declaration problem ...

```
class Sniper
{
private:
    int bullets;
public:
    Sniper(int bullets = 0): bullets(bullets){}
    friend bool Supplier::provide(Sniper &);
};
```

Nested Types (内部类型)

Nested Enumerations (C++11) (内部枚举)

作用域同样有 public、protected、private.

- It can be accessed outside of the class, but with the class name scope qualifier.

```
class Mat
{
public:
    enum DataType
    {
        TYPE8U,
        TYPE8S,
        TYPE32F,
        TYPE64F
    };
private:
    DataType type;
    void * data;
public:
    Mat(DataType type) : type(type), data(NULL){}
    DataType getType() const { return type; }
};
```

DataType::TYPE8U → 在类外声明

Mat::DataType::TYPE8U → 在类内声明

nested-enum2.cpp

Nested Types: Scope

Private:

- Only visible to the containing class

Protected:

- Visible to the containing class and its derived class.

Public:

- Visible to the containing class, to its derived classes, and to the outside world.

Nested Classes (内部类)

- Nested classes: The declaration of a class/struct or union may appear inside another class.

```
class Storage
{
public:
    class Fruit
    {
        string name;
        int weight;
    public:
        Fruit(string name="", int weight=0);
        string getInfo();
    };
private:
    Fruit fruit;
public:
    Storage(Fruit f);
};
```

nestedclass.cpp

实例化内部类：外部类名::内部类名 变量名；

```
Storage::Fruit apple("apple", 100);
```

RTTI (Runtime Type Identification) 存在于多态中，只有C++有，C无。

对象类型会在运行时确定。

```
class Person;
class Student: public Person;

Person person("Yu");
Student student("Sam", "20210212");
Person* pp = &student;
Person& rp = student;
Student * ps = (Student*)&person; //danger!
```

这样强转可能有危险，可通过RTTI避免

Type cast Operators

- dynamic_cast operator: conversion of polymorphic types.
- typeid operator: Identify the exact type of an object.
- type_info class. the type information returned by the typeid operator.

typeid、type_info

- typeid operator
  - determine whether two objects are the same type
  - Accept: the name of a class, an expression that evaluates to an object
- type_info class
  - The typeid operator returns a reference to a type_info object
  - Defined in the <typeinfo> header file
  - Comparing type using the overloaded == and != operators

```
Derived *dp = new Derived;
Base *bp = dp;

// compare the type of two objects at run time
if (typeid(*bp) == typeid(*dp))
{ ...                }
```

the operands to the typeid are objects, so use *dp not dp

```
// test whether the run-time type is a specific type
if (typeid(*bp) == typeid(Derived))
{...                                }
```

type_info class includes a ***name()*** member that returns an string that is typically the name of the class.

```
#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();

    Inherit* pInherit = dynamic_cast<Inherit*>(pBase);

    std::cout << "The type of pBase pointed to is " << typeid(*pBase).name() << std::endl;
    std::cout << "The type of pBase is " << typeid(pBase).name() << std::endl;

    std::cout << "The type of pInherit pointed to is " << typeid(*pInherit).name() << std::endl;
    std::cout << "The type of pInherit is " << typeid(pInherit).name() << std::endl;


    delete pBase;

    return 0;
}
```

```
The type of pBase pointed to is 7Inherit
The type of pBase is P4Base
The type of pInherit pointed to is 7Inherit
The type of pInherit is P7Inherit
```

dynamic_cast

- It can safely assign the address of an object to a pointer of a particular type.
- Invoke the correct version of a class method (remember virtual functions)

```
Person person("Yu");
Student student("Sam", "20210212");
Person* pp = NULL;
Student * ps = NULL;

ps = dynamic_cast<Student*>(&person); // NULL
pp = dynamic_cast<Person*>(&student);
```

There must be at least one virtual function in the class B, otherwise it fails to compile.

```
class B { ... };
class D : public B { ... };
void f()
{
  B* pb = new D;
  B* pb2 = new B;
  D* pd = dynamic_cast<D*>(pb);      // ok: pb points to D
  ...
  D* pd2 = dynamic_cast<D*>(pb2);   // fail,pb2 points to B not D
                                    //  pd2 is NULL
  ...
}
```

You can check whether the downcast is successful by if statement.

```
if (Derived *dp = dynamic_cast<Derived*>(bp))   //bp is a base class pointer
{
}
```

If **bp** points to a Derived object, then the cast will initialize dp to point to the Derived object to which bp points. In this case, it is safe for the code inside the if to use Derived operations. Otherwise, the result of the cast is 0.

```
#include <iostream>

#ifndef _CAST_H_
#define _CAST_H_

class Base
{
public:
    Base(){}
    virtual ~Base(){}

    void show()
    {std::cout << "Base funtion" << std::endl;}
};

class Inherit :public Base
{
public:
    Inherit(){}
    ~Inherit(){}

    void show()
    { std::cout << "Inherit funtion" << std::endl;}
};

#endif
```

```
#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();

    pBase->show();

    delete pBase;

    return 0;
}
```

Invoke the show() of base class, though pBase points to the derived object.

```
#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();

    if(Inherit* pInherit = dynamic_cast<Inherit*>(pBase))
    {
        pInherit->show();
    }

    delete pBase;

    return 0;
}
```

Invoke the show() of derived class, because pBase is converted to the derived pointer.

其他的 Type Cast Operators

- const_cast:
  - Type cast for const or volatile value

  const_cast.cpp
- 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

```
Base * pB = static_cast<Base*>(derived); //valid
Derived * pD = static_cast<Derived*>(base); //valid
UnRelated * pU = static_cast<UnRelated*>(base); //invalid
```

- reinterpret_cast
  - Converts between types by reinterpreting the underlying bit pattern.

const_cast 多用于将 const 数据传入非 const 参数的函数时.

四种 cast 辨析:

① static_cast<typeid>(expression): 将 expression 转成 typeid 类型, 但无运行时检查保证转换安全

(a) 类层次结构中的转换: up-casting 是安全的, down-casting 不安全

(b) 基本 data type 的转换: 不安全

(c) 一般空指针 ⟶ typeid 类型空指针: 安全.

(d) 任何类型 ⟶ void 类型: 安全

static_cast 不能转换掉 expression 中的 const, volatile, __unaligned 属性

② dynamic_cast<typeid>(expression): 将 expression 转成 typeid 类型

(a) typeid 必须是类的指针、类的引用、void*. 且若 typeid 是类的指针/引用, 则 expression 也必须是类的指针/引用.

(b) 依赖于 RTTI 信息. 在转换前会检查是否可转. 通常编译器会通过 vtable 找到对象的 RTTI 信息, 若基类无 virtual 方法, 就无法判定基类指针变量所指对象的真实类型. 这时, dynamic_cast 只能用做安全转换 (实际上这样的转换也用不到 dynamic_cast 了)

(c) 类层次间的 up-casting 和 down-casting

(d) 类间交叉转换.

dynamic_cast 不能转换掉 expression 中的 const, volatile, __unaligned 属性

③ reinterpret_cast<typeid>(expression): 在非相关类型间转换, 不做任何检查, 操作结果只是简单从一个指针到别的指针的值的二进制拷贝.

reinterpret_cast 不能转换掉 expression 中的 const, volatile, __unaligned 属性

④ const_cast<typeid>(expression): 将 expression 的 const 属性设置或移除. 但并不转换原 expression 本身, 只是将操作后结果赋给新变量.