规则 1

```
/*
   * Rule 1:View C++ as a federation of languages.
   *
   * 1.C
   * 2.Object-Oriented C++.
   * 3.Template C++.
   * 4.STL
   *
   * C++高效编程守则视情况而定，取决于你使用 C++的哪一部分
   *  对于内置类型而言 pass-by-value 比 pass-by-reference 高效
   *  其他的若存在构造函数和析构函数，pass-by-reference-to-const 高效
   */
```

规则 2

```
 // 尽量以 const,enum,inline 代替#define
/*
 * 尽量用 const double Aspect_ratio 1.653;代替 #define ASPECT_RATIO 1.653;
 * 因为 const double Aspect_ratio 1.653; 一定会被编译器看到记录到符号表而
 *  #define ASPECT_RATIO 1.653;则不一定。
 */

  #define ASPECT_RATIO 1.653;//不是我们我想要的形式
  const double Aspect_Ratio=1.653;//尽量使用这种形式


/*
 * 定义常量指针，因为常量的定义常放在头文件中，所以指针定义为 const
 * class 专属常量的定义
 */
  const char * const authorName1="Scott Meyers";
  const std::string authorName2="Scott Meyers";//比较常用


 //新的编译器
 class GamePlayer{
  static const int Numturns=5;//声明式
  int scores[Numturns];//使用该常量
};
//老的编译器
class GamePlayer{
  static const int Numturns;//声明式
  int scores[Numturns];//使用该常量

};
const int Numturns=1.653;

//没有 所谓的 private #define 这种声明
//获取一个 const 常量的地址或引用是合法的 ，但是获取一个 enum 的地址或引用是不合
//法的，获取一个#define 也是不合法的
class GamePlayer{
  enum { Numturnes=5};//Numturnes 只是 5 的一个记号
  int scores[Numturns];

};

//尽量不要用#define 调用函数，要用 template
#define CALL_MAX(a,b) f( (a)>(b)? (a):(b))
```

```
int a=5,b=0;
CALL_MAX(++a,b);//a 累加 2 次
CALL_MAX(++a,b+10)//a 累加 1 次

tempalte <typename T>
inline void    CallMax(const T& a,const T& b)
{
   f(a>b?a:b);
}

//注意  #include #ifdef/#ifndef 仍是必备品
```

规则 3

```
/*尽可能使用  const*/
    /*
     *    1.const 修饰的意义
     */

     char greeting[]="hello";
     char* p=greeting;//non-const pointer,non-const data
     const char* p=greeting;//non-const pointer,const data
     char* const p=greeting;//const pointer,non-const data
     const char* const p=greeting;//const pointer,const data


     void f(const Widget* pw);//获取一个指针，只想一个常量的 Widget 对象
     void f(Widget const* pw);//同上


     //声明一个迭代器为 const，表示这个迭代器不能指向别的东西，但可以改变其值
     vector<int>    vec;

     const vector<int>::iterator iter=vec.begin;
     *iter=10;//正确  可以改变其值
     ++iter;//错误  不可以改变指向

     vector<int>::const_iterator iter=vec.begin;
     *iter=10;//错误  不可以改变其值
     ++iter;//正确    可以改变指向

     /*令函数返回一个常量值  可以减少莫名其妙的错误
      *如果不声明为 const
      * Rational a,b,c;
         ...
         (a*b)=c;//合法
      */

     class Rational{

     };
     const Rational operator*(const Rational& lhs,const Rational &rhs)
     {
           return Rational();

     }
```

```
/*
*    2.const  成员函数
*/


class TextBlock{

      public:
            ...
      const char& operator[](size_t position)const
      {
            return text[position];
      }
char& operator[](size_t position)
      {
            return text[position];
      }
      private:
            string text;

};

TextBlock tb("hello");
cout<<tb[0];//调用  non-const TextBlock::operator[]

const Textblock ctb("world");
cout<<ctb[0];//调用  const TextBlock::operator[]

/*
 *    non-const operator[]  的返回类型是 reference to char  不是 char。如果 operator[]
 *  只是返回一个 char  则 tb[0]='x';无法通过编译，因为函数返回是内置类型，改动函数
 *  返回值本来就不合法
 *
 *

cout<<tb[0];//ok  读一个 non-const TextBlock
tb[0]='x';//ok  写一个 non-const TextBlock
cout<<ctb[0];//ok  读一个 const TextBlock
ctb[0]='x';//wrong  写一个 const TextBlock
*/

/*
 * 3.如何在 const 成员函数中，更改成员变量
 */
```

```
 class CtextBlock{

      public:

      std::size_t length() const;
      private:
            char *pText;
            std::dize_t textLength;
            bool lengthIsValid;
};
std::size_t CTextblock::length()const
      {
            if(!lengthIsValid)
                  {
                        textLength=std:strlen(pText);//在 const 成员函数中不能给
                                                      // textLength 和 lengthIsValid 赋值
                        lengthIsValid=true;
                  }
            return textLength;
      }

//解决方法是将要改变的值声明为 mutable
 class CtextBlock{

      public:
            ...
      std::size_t length() const;
      private:
            char *pText;
            mutable std::size_t textLength;
            mutable bool lengthIsValid;
};
std::size_t CTextblock::length()const
      {
            if(!lengthIsValid)
                  {
                        textLength=std:strlen(pText);//在 const 成员函数中不能给
                                                      //textLength 和 lengthIsValid 赋值
                        lengthIsValid=true;
                  }
            return textLength;
      }
```

```
/*
 * 4.const 和 non-const 成员函数避免重复
 */


    class TextBlock{

    public:
        ...
    const char& operator[](size_t position)const
    {
        ...
        ...
        ...
        return text[position];

    }
   char& operator[](size_t position)
    {
        //避免重复的简洁代码
        return const_cast<char&>(static_cast<const TextBlock&>(*this)[position];

    }

};
```

规则 4
/*确定对象使用前已经初始化*/

```
 class PhoneNumber{

     };
 class ABEntry{

     private:
          string theName;
          string theAddress;
          list<PhoneNumber> thePhones;
          int numTimesconsulted;
     public:
         ABEntry(const string& name,const string &address,const list<PhoneNumber>& phones);


};
```

//下面的只是赋值　不是初始化　尽量不要使用这种写法

```
ABEntry::ABEntry(const string& name,const string &address,const list<PhoneNumber>& phones)
     {

          theName=name;
          theAddress=address;
          thePhones=phones;
          numTimeconsulted=0;
     }
```

//初始化的写法　尽量采用这种写法

```
ABEntry::ABEntry(const string& name,const string &address,const list<PhoneNumber>& phones):
     theName(name),theAddress(address),thePhones(phones),numTimeconsulted(0)
     {

     }
```

//不同编译单元内定义的 non-local static 对象的初始化顺序

```
//在一个文件中
  class FileSystem{

    public:
          size_t numDisks() const;
};
extern FileSystem tfs;
//在另一个文件中
  class Directory{
    public:
          Directory(params);
};
Directory::Directory(params)
    {
          size_t disks=tfs.numDisks();//这里有可能 tfs 还没有初始化
    }

//解决上述问题的解决办法
  class FileSystem{

};//同上
FileSystem& tfs()
{
    static FileSystem fs;
    return fs;
}
class Directory{

};//同上
Directory::Directory(params)
    {
          size_t disks=tfs().numDisks();//这里 tfs 已经初始化
    }
Directory& tempDir()
{
    static Dirrectory td;
    return td;
}
```

规则 5

```
/*    了解 c++  默默编写并调用那些函数*/

 class Empty{
};
 //与上面的类相同
 class Empty{

      public:
           Empty(){    }
           Empty(const Empty& rhs){    }
           ~Empty(){    }

           Empty& operator=(const Empty&rhs){      }
};

template<class T>
   class NameObject{
      public:
           NameObject(string name,const T& value):nameValue(name),objectValue(value)
                {
                }

      private:
           string nameValue;
            T objectValue;
   };

   NameObject<int> p1(newDog,2);
   NameObject<int> s1(oldDog,36);

   p1=s1;//不会产生编译错误


//下面的类  一些函数将不会被构造
   template<class T>
   class NameObject{
      public:
           NameObject(string name,const T& value):nameValue(name),objectValue(value)
                {
                }

      private:
           string& nameValue;//如今这个是  reference
```

```
        const T objectValue;//如今这个为 const
};
```

```
string newDog("Persephone");
string oldDog("Satch");
NameObject<int> p(newDog,2);
NameObject<int> s(oldDog,36);

p=s;//会产生编译错误  c++不容许 reference 指向不同的对象
```

规则 6

/*　如不想让编译器自动生成函数 ， 就该明确拒绝*/

```
 //第一种做法
 class HomeForSale{

      public:

      private:
           HomeForSale(const HomeForSale&);//声明为 private  而且不定义
           HomeForSale& operator=(const HomeforSale&);//声明为 private  而且不定义
};

 //第二种做法
 class Uncopyable{
      protected:
           Uncopyable(){}
           ~Uncopyable(){}
      private:
           Uncopyable(const Uncopyable&);
           Uncopyable& operator=(const Uncopyable&);

};
//为了阻止 HomeForSale　我们只需要从 Uncopyable 继承

 class HomeForSale1:private Uncopyable{
              //不用写任何代码
};
```

规则 7
/*为多态基类声明 virtual 析构函数*/

//为类声明 virtual 析构函数时可以减少"局部销毁"问题

```
 class TimeKeeper{

      public:
           TimeKeeper(){}
           virtual ~TimeKeeper(){}

};
clasee Derived:public TimeKeeper{

      public:
           Drived(){}
           ~Derived(){};
};
```

//任何一个 class 只要确定带有 virtual 函数，几乎确定应该有一个 virtual 析构函数

```
/*
 * 如果一个 class 不含 virtual 函数，说明他不想被用作一个基类。当不想被用作基类时候
 * 定义 virtual 析构函数往往是个坏主意。
 *
 *
 */
```

//心得：只有当 class 内至少含有一个 virtual 函数时，才声明 virtual 析构函数

```
      TimeKeeper *tk=new Derived();

      delete tk;//如果 TimeKeeper 没有 virtual   那么 Derived 的部分可能没有被销毁
```

规则 8

/*不要让异常逃离析构函数*/

//具体的原因见教材

//解决好的办法

```
class DBConn{

    public:
        ...
        void close()
        {
            db.close();
            closed=true;
        }
      ~DBConn(){
        if(!closed)
            {
                try{
                    db.close();
                }
                catch(...){
                    制作运转记录，记下对 close 的调用失败;
                    ...
                }
            }

      }
      private:
        DBconnection db;
        bool closed;

};
```

规则 9
/*绝不在构造函数和析构函数中调用 virtual 函数*/

```
//下面的做法是不好的
class    Transaction{

      public:
            Transaction();
            virtual void logTransaction()const=0;
};

Transaction::Transaction()
      {
            ...
            logTransaction();
      }

class BuyTransaction:public Transaction{

      public:
            virtual void logTransaction() const;

};
class SellTransaction:public Transaction{

      public:
            virtual void logTransaction() const;

};

      BuyTransaction b;//后果见教材

      //解决方法

      class    Transaction{

      public:
            explicit Transaction(const std::string &logInfo);
            void logTransaction(const std::string &logInfo)const;
     };

      Transaction::Transaction(const std::string &logInfo)
      {
            ...
```

```
        logTransaction(logInfo);
}

class BuyTransaction:public Transaction{

    public:
        BuyTransaction(parameters):Transaction(createLogString(parameters))
                {
                }
        //virtual void logTransaction() const;
    private:
        static std:string createLogString(parameters);
};
```

规则 10
/*令 operator=返回一个 reference to *this*/

//只是一个协议 不是强制的

```
class Widget{

    public:
        Widget& operator=(const Widget& rhs)
        {
            ...
            return *this;
        }
        Widget& operator+=(const Widget& rhs)
        {
            ...
            return *this;
        }
        Widget& operator=(int rhs)
        {
            ...
            return *this;
        }


};
```

规则 11

/*在 operator= 处理自我赋值*/

//可能出现自我赋值的情况

```
a[i]=a[j];//i==j
*px=*py;//px,py 指向同一个对象

class Base {};
class Derived:public Base {};
void function(const Base& rb,Derived * pd);//rb 和*pd 是同一对象
 //给出自我赋值的情况和解决办法
 class Bitmap{
};
class Widget{
  ...
private:
    Bitmap *pb;
}
//一份不安全的 operator=实现版本
Widget& Widget::operator=(const Widget& rhs)
    {
        delete pb;
        pb=new Bitmap(*rhs.pb);
        return *this;
    }
//解决方法
    Widget& Widget::operator=(const Widget& rhs)
    {
        if(this==&rhs)
            return *this;
        delete pb;
        pb=new Bitmap(*rhs.pb);
        return *this;
    }
//解决方法
    Widget& Widget::operator=(const Widget& rhs)
    {
        Bitmap *pOrig=pb;
        pb=new Bitmap(*rhs.pb);
        delete pOrig;
        return *this;
    }
```

规则 12
/*复制对象时勿忘其每一个成分*/

```
/*
 *   当自己定义 copy  和  copy assign 函数的时候，不要忽略类中每一个
 *   成员变量，即使是继承类在定义 copy  和  copy assign 函数的时候
 *   也要显示的对基类的成员变量的 copy  和  copy assign 函数的进行
 *   显示的初始化。
 *   具体的代码见教材
 */

 class Cuetomer{
      public:
           ...
           Customer(const Customer& rhs);
           Cuetomer& operator=(const Customer& rhs);
           ...
      private:
           string name;
};
Customer::Customer(const Customer& rhs):name(rhs.name)
{

}
Customer& Customer::operator=(const Customer& rhs)
{
           name=rhs.name;
           return *this;

}

class PriorotyCustomer:public Customer{
      public:
           ...
                PriorotyCustomer(const PriorotyCustomer& rhs);
                PriorotyCustomer& operator=(const PriorotyCustomer& rhs);
           private:
                int priority;
};

PriorotyCustomer::PriorotyCustomer( const PriorotyCustomer& rhs)
                                        :Customer(rhs), priority(rhs.priority)
{
```

```
}
PriorotyCustomer& PriorotyCustomer::operator=(const PriorotyCustomer& rhs)
{
        Customer::operator=(rhs);
        priority=rhs.priority;
        return *this;
}
```

规则 13

```
/*以对象管理资源*/
 class Investment{

};
Investment *createInvestment();//工厂函数

//可能函数在...部分遇到异常而无法删除 pInv
void f()
{
      Investment *pInv=createInvestment();
      ...
      delete pInv;
}
//利用 auto_ptr 来避免此类问题

void f()
{
      std::auto_ptr<Investment> pInv(createInvestment());
      ...

}

//auto_ptr  和  shared_ptr 的区别

   std::auto_ptr<Investment> pInv1(createInvestment());//pInv1 指向 createInvestment()的对象
   std::auto_ptr<Investment> pInv2(pInv1);//pInv2 指向对象，pInv1 设置为 null
   pInv1=pInv2;//pInv1 指向对象，pInv2 设置为 null

   std::tr1::shared_ptr<Investment> pInv1(createInvestment());//pInv1 指向 createInvestment()
                                                              //的对象
   std::tr1::shared_ptr<Investment> pInv2(pInv1);//pInv2 指向对象，pInv1 指向对象
   pInv1=pInv2;//pInv2 指向对象，pInv1 指向对象
```

规则 14
/*在资源管理类中小心 copying 行为*/

```
void lock(Mutex* pm);//锁定 pm 指向的互斥器
void unlock(Mutex* pm);//互斥器解除锁定

class Lock{

    public:
        explicit Lock(Mutex*pm):mutexPtr(pm)
            {
                lock(mutexPtr);
            }
        ~Lock()
          {
                unlock(mutexPtr);
        }
    private:
        Mutex *mutexPtr;


};

Mutex m;//定义互斥器
...
{

    Lock m1(&m);
    ...
}

//如果发生下列行为 结果不可预知
Lock m1(&m);
Lock m2(m1);


//解决办法 1
    class Lock:private Uncopyable{

    public:
        explicit Lock(Mutex*pm):mutexPtr(pm)
            {
                lock(mutexPtr);
            }
```

```
        ~Lock()
         {
                unlock(mutexPtr);
        }
    private:
        Mutex *mutexPtr;


};

//解决方法 2
class Lock{

    public:
        explicit Lock(Mutex*pm):mutexPtr(pm,unlock)
            {
                lock(mutexPtr.get());
            }

    private:
        std::tr1::shared_ptr<Mutex> mutexPtr;


};
```

规则 15
/*在资源类中提供对原始资源的访问*/

```
std::tr1::shared_ptr<Investment> pInv(createInvestment());
int daysHeld(const Investment *pi);//返回投资天数
int days=daysHeld(pInv);//error

//auto_ptr 和 tr1::shared_ptr 提供 get 函数来返回智能指针内部的原始指针
int days=daysHeld(pInv.get());

//auto_ptr 和 tr1::shared_ptr 重载了  operator->和 operatot*

class Investment{

      public:
            bool isFree()const;
            ...
}

Investment *createInvestment();

std::tr1::shared_ptr<Investment> pInv1(createInvestment());
bool tab1=!(pInv1->isFree());
std::auto_ptr<Investment> pInv1(createInvestment());
bool tab2=!( (*pInv2).isFree());
```

规则 16
/*成对的使用 new 和 delete */

```
std::string* stringPtr1=new std::string;
std::string* stringPtr1=new std::string[100];

delete stringPtr1;
delete [] stringPtr2;

typedef std::string AddressLines[4];
std::string * pa1=new AddressLines;

delete    [] pal;
```

规则 17
/*以独立的语句将 newed 对象植入智能指针*/

//下面的写法可能造成内存泄漏
```
processWidget(std::tr1::shared_ptr<Widget>(new Widget),priority());
```

//改进的算法 不会造成内存泄漏
```
std::tr1::shared_ptr<Widget> pw(new Widget);
processWidget(pw,priority());
```

规则 18
/*让借口设计的更容易使用*/

//设计一个 Date 类 你也许会这样设计

```
class Date{
    public:
        Date(int month,int day,int year);
        ...
};
```

//也许下面的设计会更好点

```
 struct Day{                    struct Month{                    struct Year{

    explicit Day(int d):val(d)      explicit Month(int m):val(m)    explicit Year(int y):val(y)

    {                                   {                                {
    }                                   }                                }
    int val;                        int val;                        int val;
};                                  };                               };
 class Date{
    public:
        Date(const Month& m,const Day& day,const Year& year);
        ...
};
```

//一年只有 12 个月 最好定义好所有的月份

```
class Month{
    public:
        static Month Jan() { return Month(1);}
        static Month Feb() { return Month(2);}
        ...
        static Month Dec() { return Month(12);}
    private:
        explicit Month(int m);
        ...
};
```

规则 19
/*设计 class 犹如设计 type*/

1.新的 type 的对象应该如何被创建和销毁
2.对象的初始化和对象的赋值有什么区别
3.新的 type 的对象如果被 passed by value 意味者什么
4.什么是新的 type 的合法值
5.你的新 type 需要配合某个继承图表吗？
6.你的新的 type 需要什么样的转换？
7.什么样的操作符和函数对此新的 type 而言是合理的
8.什么样的标准函数应该驳回
9.谁该取用新的 type 的成员
10.什么是新的 type 的未声明的接口？
11.你的新的 type 有多么一般化？
12.你真的需要新的 type 吗?

规则 20

/*宁以 pass-by-reference-to-const 替代 pass-by-value*/

```
//下列代码将会非常耗时
class Person{

    public:
        Person();
        virtual ~Person();
        ...
    private:
        std::string name;
        std::string address;
};
class Student:public Person{

    public:
        Student();
        ~Student();
        ...
    private:
        std::string schoolName;
        std::string schoolAddress;
};

bool validStudent(Student s);
Student plato;
bool platoIsOK=validStudent(plato);
//将会调用一次 Student copy 构造函数，一次 Person Copy 构造函数和四次 string copy 构造
//函数

//解决方法
bool validStudent(const Student& s);//该方法还可以防止类型的截断
```

规则 21

/*必须返回对象时，别妄想返回 reference*/

```
class Rational{
    public:
        Rational(int numerator=0,int denominator=1);
        ...
    private:
        int n,d;
        friend const Rational operator*(const Rational& lhs,const Rational& rhs);
};

//如果改用 reference

const Rational& operator*(const Rational& lhs,const Rational& rhs)
{
    Rational result(lhs.n*rhs.n,lhs.d*rhs.d);//糟糕的代码
    return result;
}

const Rational& operator*(const Rational& lhs,const Rational& rhs)
{
    Rational *result=new Rational(lhs.n*rhs.n,lhs.d*rhs.d);//更糟糕的代码
    return *result;
}

const Rational& operator*(const Rational& lhs,const Rational& rhs)
{
    static Reational result;//又是一个更糟糕的代码

    result=...;

    return result;
}
//基于上面的代码
bool operator==(const Rational& lhs,const Rational& rhs);
Rational a,b,c,d;

if((a*b)==(c*d))//将永远返回真值，具体解释见教材
    {
    }
else
    {
    }
```

规则 22
/*将成员变量设为 private*/

```
class AccessLevels{

    public:
        ...
        int getReadOnly()const
        {
            return readOnly;
        }
        void setReadWrite(int val)
        {
            readWrite=value;
        }
        int getReadWrite() const
        {
            return readWrite;
        }
        void setWriteOnly(int val)
        {
            writeOnly=val;
        }
    private:
        int noAccess;//对此 int 无任何访问
        int readOnly;//对此 int 只读访问
        int readWrite;//对此 int 读写访问
        int writeOnly;//对此 int 惟写访问

};
//如果成员变量设为 protected 和 public  会造成不可预知的维护成本
```

规则 23

/*宁以 non-member non-friend 替代 member 函数*/

```
class WebBrowser{
        public:
                ...
                void clearCache();
                void clearHistory();
                void removeCokies();
                ...
};

//让客户一个一个执行
   WebBrowser wb;
   wb.clearCache();
   wb.clearHistory();
   wb.removeCokies();

 //提供一个整套动作的函数
 class WebBrowser{
        public:
                ...
                void clearEverything();//调用上面的 3 个函数
                ...
};

//提供一个 non-member 函数(这种写法是最好的)
void clearBorowser(WebBrowser& wb)
{
        wb.clearCache();
   wb.clearHistory();
   wb.removeCokies();
}
```

规则 24

/*若所有的参数都需要转型，请为此采用 non-member 函数*/

```
class Rational{

    public:
        Rational(int numerator=0,int donominator=1);
        int    numerator()const;
        int donominator()const;
        const Ration operator* (const Rational& rhs)const;
    private:
        ...
};

Rational oneEight(1,8);
Rational oneHalf(1,2);

Rational result=oneHalf*oneEight;//OK
result=result*oneEight;//OK

result=oneHalf*2;//OK
result=2*oneHalf;//BAD

//解决办法是

class Rational{
    ...
};
const Rational operator*(const Rational &lhs,const Rational &rhs)
{
    return Rational(lhs.numerator()*rhs.numerator(),lhs.donominator()*rhs.donominator());
}

Rational oneEight(1,8);
Rational oneHalf(1,2);

Rational result=oneHalf*oneEight;//OK
result=result*oneEight;//OK

result=oneHalf*2;//OK
result=2*oneHalf;//OK
```

规则 26

/*尽量延后变量定义出现的时间*/

```
void encrypt(std::string& s);//在其中适当的地点为其加密

//以下程序过早的定义 encrypted
std::string encryptPasswrod(const std::dtring& password)
{
        using namespace std;
        string encrypted;
        if(password.length()<MinumumPasswordLength){
            throw logic_error("password is too short");
        }
        encrypt(encrypted);
        return encrypted;
}

    //比较好的定义方式
    std::string encryptPasswrod(const std::dtring& password)
    {
        if(password.length()<MinumumPasswordLength){
            throw logic_error("password is too short");
        }
        string encrypted(password);
        encrypt(encrypted);
        return encrypted;
    }



    //A 定义于循环外                          //B 定义于循环内
     Widget w;
     for(int i=0;i<n;i++){                  for(int i=0;i<n;i++){
          w=取决于 i 的某个值;                        Widget w(取决于 i 的某个值);
          ...                                  ...
        }                                    }

        //做法 A 的代价  1 个构造函数+1 个析构函数+n 个赋值函数
        //做法 B 的代价  n 个构造函数+n 个析构函数
        //一般情况下我们认为 B 较好
```

规则 27
/*尽量少做转型动作*/

```
const_cast<T>(expression);
dynamic_cast<T>(expression);
reinterpret_cast<T>(expression);
static_cast<T>(expression);

class Widget{
    public:
        explicit Widget(int size);
        ...
};

void doSomework(const Widget& w);
doSomework(Widget(15));
doSomework(staitc_cast<Widget>(15));//尽量使用这种写法

//下面的代码有问题
 class Window{
    public:
        virtual void onReszie()
        {
            ...
        }
};
 class SpecialWindow:public Window{
    public:
        virtual void onResize()
        {
            static_cast<Window>(*this).onResize();

            ...
            ...
            ...
        }
};
    //具体的问题见教材 可以改为
    class Window{
    public:
        virtual void onReszie()
        {
            ...
        }
```

```
        };
  class SpecialWindow:public Window{
        public:
              virtual void onResize()
              {
                    Window::onResize();

                    ...
                    ...
                    ...
              }
};
```

规则 28
/*避免返回 handles 指向对象内部*/

```
//下面的程序有问题
class Point{
    public:
        Point(int x,int y);
        ..
        void setX(int newX);
        void setY(int newY);
        ...
};
struct RectData{
    Point ulhc;
    Point lrhc;
};

class Rectangle{
    public:
        Point& upperLeft()const{ return pData->ulhc;}//我们本意是不容许修改 upperLeft 的
                                                      //值  但是我们却能修改其值
        Point& lowRight()const{ return pData->lrhc;}

    ...
    private:
        std::tr1::shared_ptr<RectData> pData;
};

Point coord1(0,0);
Point coord2(100,100);
const Ractangle rec(coord1,coord2);//rec 是  (0,0)--->(100,100)
rec.upperLeft().setX(50);//rec 是  (50,0)--->(100,100)

//我们可以这样修改程序
class Point{
    public:
        Point(int x,int y);
        ..
        void setX(int newX);
        void setY(int newY);
        ...
};
struct RectData{
    Point ulhc;
```

```
    Point lrhc;
};

class Rectangle{
    public:
        const Point& upperLeft()const{ return pData->ulhc;}//我们本意是不容许修改
                                                            //upperLeft 的值  但是我们却能修改其值
        const Point& lowRight()const{ return pData->lrhc;}

    ...
    private:
        std::tr1::shared_ptr<RectData> pData;
};

Point coord1(0,0);
Point coord2(100,100);
const Ractangle rec(coord1,coord2);//rec 是  (0,0)--->(100,100)

rec.upperLeft().setX(50);//wrong  但是会出现空悬 handles 的情况
                                //总之  避免返回 handles 指向对象内部
```

规则 29

/*为"异常安全"而努力是值得的*/

//下面的代码对出现异常安全问题

```
class PreettyMenu{
    public:
        ...
        void changeBackground(std::istream& imgSrc);
            ...
    private:
        Mutex mutex;//互斥器
        Image* bgImage;//目前的背景图片
        int imageChange;//背景图片改动的次数
};
void PreetyMenu::changeBackground(std::istream& imgSrc)
    {
        Lock(&mutex);//锁定互斥器
        delete bgImage;//拜托旧的北京图象
        ++imageChanges;//修改图像变更次数
        bgImage=new Image(imgSrc);//安装新的背景图像
        unlock(&mutex);//释放互斥器

    }
```

//修改后迦结果

```
 class PreettyMenu{
    public:
        ...
        void changeBackground(std::istream& imgSrc);
        ...
    private:
        Mutex mutex;//互斥器
        std::tr1::shared_ptr<Image> bgImage;//目前的背景图片
        int imageChange;//背景图片改动的次数
};
void PreetyMenu::changeBackground(std::istream& imgSrc)
    {
        Lock m1(&mutex);//锁定互斥器
        bgImage.reset(new Image(imgSrc));
        ++imageChanges;//修改图像变更次数


    }
```

规则 30
/*透彻了解 inlining 的里里外外*/

```
class Person{

    public:
        ...
        int age() const//一个隐喻的 inline 申请
        {
            return theAge;
        }
        ...
    private:
        int theAge;
};
//通常声明 inline 函数的做法是在函数前加上 inline
 template <typename T>
 inline const T& std::max(const T&a,const T&b)
    {
        return a<b?b:a;
        }

    //    Inline 函数一般放在头文件内
    //    template 一般放在头文件内
    //
    //inline 函数的调用可能被 inlined 也可能不被 inlined

    //编译器通常不对"通过函数指针而进行的调用"实施 inlining

    inline void f() {...}
    void (*pf)()=f;

    ...
    f();//这个调用被 inlined
    pf();//这个调用或许不被 inlined
```

规则 32

```
/*确定你的 public 继承塑膜出 is-a 关系*/

//其实不能反映出 is-a 关系，企鹅不会飞
class Bird{
    public:
        virtual void fly();//鸟可以飞
        ...
};

class Penguin:public Bird{//企鹅也是一种鸟

    ....
};

//也许下面的设计更合理一点
class Bird{
            ...
};

class FlyingBird:public Bird{//企鹅也是一种鸟

    public:
        virtual void fly();
    ....
};

class Penguin:public Bird{//企鹅也是一种鸟


    ....
};

//rectangle 和 square 的设计

class Rectangle{

    public:
        virtual void setHeight(int newHeight);
        virtual void setWidth(int newHeight);
        virtual int height() const;//返回当前值
        virtual int width() const;
};
void makeBigger(Rectangle& r)
```

```
{
    int oldHeight=r.height();
    r.setWidth(r.width()+10);//宽度加 10
    assert(r.height()==oldHeight);//判断 r 的高度是否未曾改变
}

class Square:public Rectangle{ ...};

Square s;
...
assert(s.width()==s.height());//正方形一定为真
makeBigger(s);//由于继承  是是一个矩形
assert(s.width()==s.height());//正方形一定为真,遗憾的是并不为真
```

规则 33

```
/*避免遮掩继承而来的名字*/
class Base{
        private:
                int x;
        public:
                virtual void mf1()=0;
                virtual void mf1(int);
                virtual void mf2();
                void mf3();
                void mf3(double);
                ...

};

class Derived:public Base{
        public:
                virtual void mf1();
                void mf3();
                void mf4();
                ...
};

        Derived d;
        int x;
        ...
        d.mf1();
        d.mf1(x);//错误！Derived::mf1 遮掩 Base::mf1
        d.mf2();
        d.mf3();
        d.mf3(x);//错误！Derived::mf3 遮掩 Base::mf3

        //我们可以这样做解决上述问题

        class Base{
        private:
                int x;
        public:
                virtual void mf1()=0;
                virtual void mf1(int);
                virtual void mf2();
                void mf3();
                void mf3(double);
                ...
```

```
};

class Derived:public Base{
    public:
        using Base::mf1;
        using Base::mf3;
        virtual void mf1();
        void mf3();
        void mf4();
        ...
};

    Derived d;
    int x;
    ...
    d.mf1();
    d.mf1(x);//OK！调用 Base::mf1
    d.mf2();
    d.mf3();
    d.mf3(x);//OK！调用 Base::mf3


    //我们也可以使用转交函数实现，因为我们不想继承 Base 的所有函数
    class Base{
    private:
        int x;
    public:
        virtual void mf1()=0;
        virtual void mf1(int);
        virtual void mf2();
        void mf3();
        void mf3(double);
        ...

};

class Derived:public Base{
    public:
        virtual void mf1()
        {
            Base::mf1();//转交函数   暗自成 inline
        }
        ...
```

```
};

    Derived d;
    int x;
    ...
    d.mf1();
    d.mf1(x);//错误！Base::mf1 被遮掩
```

规则 34

```
/*接口继承和实现继续*/
class Shape{

    public:
        virtual void draw()const=0;
        virtual void error(const std::string& msg);
        int objectID()const;
        ...
};

class Rectangle:public Shape{ ...};
class Ellipse:public Shape{ ...};

Shape* ps=new Shape;//Shape 是抽象类
Shape* ps1=new Rectangle;
ps1->draw();//Rectangle::draw()
Shape* ps1=new Ellipse;
ps2->draw();//Ellipse::draw()

ps1->Shape::draw();
ps2->Shape::draw();

//考虑以下代码的缺陷和改进方法
class Airport { ...};
class Airplane{

    public:
        virtual void fly(const Airport& destination);
        ...
};

void fly(const Airport& destination)
{
    缺省代码 将飞机飞至指定的目的地
}
class ModelA:public Airplane{ ...};
class ModelB:public Airplane{ ...};

class ModelC:public Airplane{
      ...
};//C 的飞行方式可能不一样，这是灾难的原因

Airport PDX(...);
```

```
Airplane* pa=new ModelC;
...
pa->fly(PDX);//Airplane::fly()

//下面是一种解决方法

class Airport { ...};
class Airplane{

    public:
        virtual void fly(const Airport& destination);
        ...
    protected:
        void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
    缺省代码 将飞机飞至指定的目的地
}

class ModelA:public Airplane{
    public:
        virtual void fly(const Airport& destination)
        {
            defaultFly(destination);
        }
    ...
};
class ModelB:public Airplane{

    public:
        virtual void fly(const Airport& destination)
        {
            defaultFly(destination);
        }
     ...
};

class ModelC:public Airplane{
    virtual void fly(const Airport& destination);
     ...
};
void ModelC::fly(const Airport& destination)
```

```
{
    缺省代码 将飞机 C 飞至指定的目的地
}

//另一种解决方法
class Airport { ...};
class Airplane{

    public:
        virtual void fly(const Airport& destination);
        ...
};

void Airplane::fly(const Airport& destination)
{
    缺省代码 将飞机飞至指定的目的地
}

class ModelA:public Airplane{
    public:
        virtual void fly(const Airport& destination)
        {
            Airplane::fly(destination);
        }
    ...
};
class ModelB:public Airplane{

    public:
        virtual void fly(const Airport& destination)
        {
            Airplane::fly(destination);
        }
     ...
};

class ModelC:public Airplane{
    virtual void fly(const Airport& destination);
     ...
};
void ModelC::fly(const Airport& destination)
{
    缺省代码 将飞机 C 飞至指定的目的地
}
```

规则 36

/*绝对不重新定义继承而来的 non-virtual 函数*/

```
//不重新定义继承而来的 non-virtual 函数
class B {
	public:
		void mf();
};

class D:public B {...};

	D x;
	B* pB=&x;
	pB->mf();//B:mf()
	D* pD=&x;
	pD->mf();//B:mf()

//	重新定义继承而来的 non-virtual 函数

class B {
	public:
		void mf();
};

class D:public B {
	puhlic:
		void mf();
	...
};

	D x;
	B* pB=&x;
	pB->mf();//B:mf()
	D* pD=&x;
	pD->mf();//D:mf()
```

规则 37

/*绝不重新定义继承而来的缺省参数*/

```
class Shape{

    public:
        enum ShapeColor{Red,Green,Blue};
        //所有的形状都必须提供一个函数  来绘制出自己
        virtual void draw(ShapeColor color=Red) const=0;
        ...
};

class Rectangle:public Shape{
    public:
        //这里赋予不同的缺省参数  很糟糕!
        virtual void draw(ShapeColor color=Green) const;

};

class Circle:public Shape{
    public:
        //这里赋予不同的缺省参数  很糟糕!
        virtual void draw(ShapeColor color) const;

};

Shape* ps; //静态类型  Shape*

Shape* pr=new Rectangle; //静态类型  Shape*

Shape* pc=new Circle; //静态类型  Shape*

ps=pc;          //动态类型  Circle*
ps=pr;          //动态类型  Rectangle*

pr->draw();//Rectangle::draw(Shape::Red);

//也许我们可以改变这种设计结构

class Shape{

    public:
        enum ShapeColor{ Red,Blue,Green};
        void draw(ShapeColor color=Red) const
```

```
        {
            doDraw(color);
        }
    private:
        virtual void doDraw(ShapeColor color=Red) const=0;//真正的工作再次完成

};
class Rectangle:public Shape{
    public:
        //这里赋予不同的缺省参数 很糟糕！
        virtual void doDraw(ShapeColor color) const;//不需要指定缺省参数

};
```

规则 38

/*通过复合塑膜出 has-a 或"根据某物实现出"*/

```
class Address {...};
class PhoneNumber {...};
class Person{
	public:
		...
	private:
		string name;
		Address address;
		PhoneNumber voiceNumber;
		PhoneNumber faxNumber;
};
```

//构造一个 template 希望构造出一组 classes 用来表示不重复对象组成的 sets

```
template <class T> class Set{

	public:
		bool member(const T& item)const;
		void insert(const T& item);
		void remove(const T& item);
		size_t size() const;
	private:
		list<T> rep;//原来表述 Set 数据
};

template<typename T>
bool Set<T>::member(const T& item)const
{
		return find(rep.begin(),rep.end(),item)!=rep.end();
}

template<typename T>
void    Set<T>::insert(const T& item)
{
		if(!member(item)) rep.push_back(item);
}
template<typename T>
void    Set<T>::remove(const T& item)
{
		typename list<T>::iterator it=find(rep.begin(),rep.end(),item);
		if(it!=rep.end()) rep.erase(it);
```

```
}

template<typename T> size_t Set<T>::size() const
{
	return rep.size();
}
```

规则 39
/*明智而审慎的使用 private*/

//如果类之间是 private 继承关系，那么编译器不会将 derived class 转换为 base class
//尽可能的使用复合，必要时才使用 private 继承（当 protected 成员或 virtual 函数牵扯进来的时候）

```
class Timer{
	public:
		explicit Timer(int tickFrequency);
		virtual void onTick();//定时器每滴答一次 此函数调用一次
		...
};

class Widget : private Timer{
	private:
		virtual void onTick();
};

//面对上面的代码 下面的代码可能更好一些

class Widget{

	private:
		class WidgetTimer: public Timer{
			public:
			  virtual void onTick()const;//定时器每滴答一次 此函数调用一次

		};
		WidgetTimer timer;
		...
};

//下面这种设计到空间最优化的时候 你要选择 private 继承

class Empty{//没有数据 不应该占内存

};
class HoldAnInt{//应该值需要一个 int 空间
	private:
		int x;
		Empty e;//应该是不需要内存
};//其实 sizeof( HoldAnInt)>sizeof(int)   大多数编译器 sizeof(Empty)==1
```

```
//选择 private 继承
class HoldAnInt:private Empty {//应该值需要一个 int 空间
        private:
                int x;
};//sizeof( HoldAnInt)==sizeof(int)
```

规则 40
/*明智而审慎的使用多重继承*/

//A 是 B ,C 的基类 B，C 是 D 的基类的写法

```
class A {

};
class B:virtual public A{

};
class C:virtual public A{

};
class D: public B,public C{

}; //若 A 是虚基类 D 的构造函数应该对 A 进行初始化
```

//public 和 private 继承集合在一起

```
class IPerson{
    public:
        virtual ~IPerson();
        virtual string name() const=0;
        virtual string birthDate() const=0;
};
class DatabaseID{ ...};//稍后使用 细节不重要

class PersonInfo{
    public:
        explicit PersonInfo(DatabaseID pid);
        virtual ~PersonInfo();
        virtual const char* theName() const;
        virtual const char* theBirthData()const;
        virtual const char* valueDelimOpen()const;
        virtual const char* valueDelimClose()const;

};

class CPerson:public IPerson,private PersonInfo{

    public:
        CPerson(DatabaseID pid):PersonInfo(pid){}
```

```
        virtual string name() const
        {
            return PersonInfo::theName;
        }
        virtual string birthDate() const
        {
            return PersonInfo::birthDate;
        }
    private:
        const char* valueDelimOpen()const {return " ";}
        const char* valueDelimClose()const{return " ";}

};

//如果使用单一继承可以完成 尽量不要使用多重继承
```

规则 41
/*了解隐式接口和编译期多态*/

//对于 template 而言，接口是隐式的，基于有效表达式。
//多态则是通过 template 具现化和函数重载解析发生于编译期

```
template<typename T>
void doProcessing(T& w)
{
    if(w.size()>0 && w!=someNastyWidget){
        {
            T temp(w);
            temp.normalize();
            temp.swap(w);
        }
}
```

规则 42

```
/*了解 typename 的具体意义*/
//以下定义是等价的
template<class T> class Widget;
template<typename T> class Widget;


tempalte<typename C>
void print2nd(const C& container)
{
    if(container.size()>=2){
        typename C::const_iterator iter(container.begin());//此处必须用 typename
        ...
    }
}

tempalte<typename C>
void print2nd(const C& container,//不容许使用 typename
typename C::iterator iter）//一定要使用 typename
{

}


//typename 必须做为嵌套从属类型名称的前缀词的例外是，typename 不可以出现在 base
//class list 内的嵌套从属类型之前
//也不可以在成员初值列中做为 base class 的修饰符

template<typename T>
class Derived: public Base<T>::Nested{//不容许使用 typename

    public:
        explicit Derived(int x):Base<T>::Nested(x)//不容许使用 typename
            {
                typename Base<T>::Nested temp;//必须使用 typename
                    ..
            }
            ..
};


//注意以下写法
template<typename IterT>
void workWithIterator(IterT iter)
```

```
{
    typedef typename std::iterator_traits<IterT>::value_type value_type;
    value_type temp(*iter);//IterT 是什么类型 temp 就是什么类型
}
```

规则 43

```
/*学会处理模版化基类内的名称*/
class CompanyA{
    public:
        void sendClearText(const string& msg);
        void sendEncryptedText(const string& msg);

};
class CompanyB{
    public:
        void sendClearText(const string& msg);
        void sendEncryptedText(const string& msg);

};
class Info{ ....};//用来保存信息
template<typename Company>
class MsgSender{

    public:
        void sendClear(const Info& info)
        {
            string msg;
            //根据 info 产生信息
            Company c;
            c.sendClearText(msg);
        }
        void sendSecret(const Info& info)
        {
            string msg;
            //根据 info 产生信息
            Company c;
            c.sendClearText(msg);
        }
};

template<typename Company>
class LoggingMsgSender:public MsgSender<Company>{

    public:
        void sendClearMsg(const Info& info)
        {
            //将传送前的信息写入 log
            sendClear(info);//调用 base class 函数  无法通过编译
            //将传送后的信息写入 log
```

```
        }
};


//模版全特化
class CompanyA{
    public:
        void sendClearText(const string& msg);
        void sendEncryptedText(const string& msg);

};
class CompanyB{
    public:
        void sendClearText(const string& msg);
        void sendEncryptedText(const string& msg);

};
class Info{ ....};//用来保存信息
template<typename Company>
class MsgSender{

    public:
        void sendClear(const Info& info)
        {
            string msg;
            //根据 info 产生信息
            Company c;
            c.sendClearText(msg);
        }
        void sendSecret(const Info& info)
        {
            string msg;
            //根据 info 产生信息
            Company c;
            c.sendClearText(msg);
        }
};
class CompanyZ{
    public:
        void sendEncryptedText(const string& msg);

};
```

```
template<>
class MsgSender<CompanyZ>{

   public:

        void sendSecret(const Info& info)
        {
            string msg;
            //根据 info 产生信息
            Company c;
            c.sendClearText(msg);
        }
};

template<typename Company>
class LoggingMsgSender:public MsgSender<Company>{

   public:
        void sendClearMsg(const Info& info)
        {
            //将传送前的信息写入 log
            sendClear(info);//如果 Company==CompanyZ  这个函数不存在
            //将传送后的信息写入 log

        }
};

//我们有三种办法苓 C++ "不进入 templatized base classes 观察"的行为失效
   //第一种
   template<typename Company>
class LoggingMsgSender:public MsgSender<Company>{

   public:
        void sendClearMsg(const Info& info)
        {
            //将传送前的信息写入 log
            this->sendClear(info);//成立  假设 sendClear 将被继承
            //将传送后的信息写入 log

        }
};
//第二种
template<typename Company>
class LoggingMsgSender:public MsgSender<Company>{
```

```
    public:
        using MsgSender<Company>::sendClear;//告诉编译器 sendClear 位于基类
        void sendClearMsg(const Info& info)
        {
            //将传送前的信息写入 log
            sendClear(info);//成立  假设 sendClear 将被继承
            //将传送后的信息写入 log

        }
};
//第三种
    template<typename Company>
class LoggingMsgSender:public MsgSender<Company>{

    public:

        void sendClearMsg(const Info& info)
        {
            //将传送前的信息写入 log
             MsgSender<Company>::sendClear(info);//成立  假设 sendClear 将被继承
            //将传送后的信息写入 log

        }
};
```

规则 44

/*将与参数无关的代码抽离 template*/

```
template<typename T,size_t n>
class SquareMatrix{

    public:
        void invert();
};

SquareMatrix<double,5> sm1;
...
sm1.invert();//SquareMatrix<double,5>::invert

SquareMatrix<double,10> sm2;
...
sm2.invert();//SquareMatrix<double,10>::invert

//以上代码可能会造成代码膨胀  可以用以下方法改进
    //1.
     template<typename T>
     class SquareMatrixBase{

         protected:
             void invert(size_t matrixSize);

     };
     template<typename T,size_t n>
     class SquareMatrix:private SquareMatrixBase{
         private:
             using SquareMatrixBase<T>::invert;
         public:
             void invert() { this->invert(n);};//调用 base 的 invert
    };
    //2.
    template<typename T>
     class SquareMatrixBase{

         protected:
             SquareMatrixBase(size_t n,T* pMem):size(n),pData(pMem){}
             void setDataPtr(T* ptr){pData=ptr;};
         private:
             size_t size;//矩阵大小
             T* pData;//指向矩阵的内容
```

```
    };
    template<typename T,size_t n>
    class SquareMatrix:private SquareMatrixBase{
        private:
            T data[n*n];
        public:
            SquareMatrix():SquareMatrixBase<T>(n,data);
            void invert() { this->invert(n);};//调用 base 的 invert
};
  //3.
  template<typename T>
    class SquareMatrixBase{

        protected:
            SquareMatrixBase(size_t n,T* pMem):size(n),pData(pMem){}
            void setDataPtr(T* ptr){pData=ptr;};
        private:
            size_t size;//矩阵大小
            T* pData;//指向矩阵的内容

    };
    template<typename T,size_t n>
    class SquareMatrix:private SquareMatrixBase{
        private:
            T data[n*n];
        public:
            SquareMatrix():SquareMatrixBase<T>(n,0),pData(new T[n*n])
                {
                    this->setDataPte(pData.get());
                }
        private:
            boost::scoped_array<T> pData;

};
```

规则 45
/*运用成员函数模版接受所有兼容类型*/

```
class Top{
};
class Middle:public Top{

};
class Bottom:public Middle{

};
Top* pt1=new Middle;//Middle*转到 Top*
Top* pt2=new Bottom;//Bottom*转到 Top*

const Top* pct2=pt1;//Top*转到 const Top*

//
 template <typename T>
 class SmartPtr{//智能指针
      public:
           explicit SmartPtr(T* realPtr);
};
SmartPtr<Top> pt1=SmartPtr<Middle>(new Middle);//SmartPtr<Top>  转换为 SmartPtr<Middle>
SmartPtr<Top> pt2=SmartPtr<Bottom>(new Bottom);//SmartPtr<Top>  转换为 SmartPtr<Bottom>
SmartPtr<const Top> pt2=pt1;//SmartPtr<Top>  转换为  SmartPtr<const Top> //由于上面的函数
//构造函数无法完成  我们采用一下做法
   template<typename T>
   class SmartPtr{
      public:
           template<typename U>
           SamrtPtr(const SmartPtr<U>& other);
           ...
   };//无法阻止 SmartPtr<double>到 SmartPtr<int>的转换

   //针对上面的问题做出改变
   template<typename T>
   class SmartPtr{
      public:
           template<typename U>
           SamrtPtr(const SmartPtr<U>& other):heldPtr(other.get())
                {
                }
           T* get() const {return heldPtr;}
      private:
```

```
        T* heldPtr;
        ...
};
```

规则 46

/*需要类型转换时请为模版定义非成员函数*/

//template 实参推导过程中从不将隐式的类型转换纳入考虑

```
template<typename T>
class Rational{
	public:
		Rational(const T& numerator=0,const T& denominator=1);
		const T numerator()const;
		const T denominator()const;
		...
};
 template<typename T>
 const Rational<T> operator*(const Rational<T>& lhs, const Rational<T>& rhs)
 {
	...
}

Rational<int> oneHalf(1,2);
Rational<int> reault=oneHalf*2;//编译错误
```

//解决方法

```
template<typename T> class Rational;//声明 Rational    template

template<typename T>
const Rational<T> doMultiply( const Rational<T>& lhs,const Rational<T>& rhs);
                        //声明 helper template
template <typename T>
class Rational{
	public:
		Rational(const T& numerator=0,const T& denominator=1);
		const T numerator()const;
		const T denominator()const;
		friend const Rational<T> operator*(const Rational<T>& lhs,const Rational<T>& rhs)
		{
			return doMultiply(lhs,rhs);
		}
		...
};
template<typename T>
const Rational<T>    doMultiply( const Rational<T>& lhs , const Rational<T>& rhs)
{
	return Rational<T>(lhs.numerator()*rhs.numerator(),lhs.denominator()*rhs.denominator());
}
```

规则:49
/*了解 new-handler 的行为*/

//set_new_handler 的参数是个指针，指向 operator new 无法分配足够的内存时该被调用的函数。
//其返回值也是个指针，指向 set_new_handler 被调用前正在执行的那个 new-handler 函数

//set_new_handler 的用法

```
#include<new>
#include<iostream>
using namespace std;
void outMem(){

    cerr<<"内存分配失败";
    abort();

}

int main(){
    set_new_handler(outMem);
    try
   {
       while ( 1 )
       {
           new int[5000000];
           cout << "Allocating 5000000 ints." << endl;
       }
   }
   catch ( exception e )
   {
       cout << e.what( ) << " xxx" << endl;
   }

}
```

//new_handler 和 operator new 可以在普通类中被实现，为了使该方案可以被复合使用，可以将其模板化：

```
template<typename T>                    // "mixin-style" base class for
class NewHandlerSupport{                // class-specific set_new_handler
public:                                 // support
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void * operator new(std::size_t size) throw(std::bad_alloc);
  ...                                   // other versions of op. new —
                                        // see Item 52
```

```
private:
   static std::new_handler currentHandler;
};
template<typename T>
std::new_handler NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
std::new_handler oldHandler = currentHandler;
currentHandler = p;
return oldHandler;
}
template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
   throw(std::bad_alloc)
{
   NewHandlerHolder h(std::set_new_handler(currentHandler));
   return ::operator new(size);
}
// this initializes each currentHandler to null
template<typename T> std::new_handler NewHandlerSupport<T>::currentHandler = 0;
//需要将 new_handler 和 operator new 特别处理的类直接从 NewHandlerSupport 派生即可：
class Widget: public NewHandlerSupport<Widget> {
   ...                                    // as before, but without declarations for
};                                        // set_new_handler or operator new
//有一种“不抛出异常”的 operator new 的实现：
class Widget { ... };
Widget *pw1 = new Widget;                 // throws bad_alloc if
                                          // allocation fails
if (pw1 == 0) ...                         // this test must fail
Widget *pw2 =new (std::nothrow) Widget;   // returns 0 if allocation for
                                          // the Widget fails
if (pw2 == 0) ...                         // this test may succeed
//需要注意的是，上述代码只能保证 pw2 在执行 Widget *pw2 =new (std::nothrow) Widget 时
//不抛出异常，
//却不能保证 pw2 中的 operator new 操作不抛出异常。
```

规则:50
//重写 new 和 delete 的 8 个理由：
1)检测运用上的错误.
2)为了强化效能.
3)为了收集使用上的统计数据.
4)为了检测运用错误.
5)为了收集动态分配内存之使用统计信息.
6)为了增加分配和归还的速度.
7)为了降低缺省内存管理器带来的空间额外开销.
8)为了弥补缺省分配器中非最佳齐位.
9)为了将相关对象成簇集中
10)为了获得非传统的行为

规则 51:

/*编写 new 和 delete 要固守常规*/

//一段 operator new 的伪代码：

```
void * operator new(std::size_t size) throw(std::bad_alloc)
{                                                  // your operator new might
   using namespace std;                          // take additional params
   if (size == 0) {                            // handle 0-byte requests
      size = 1;                                  // by treating them as
   }                                                 // 1-byte requests
   while (true) {
    attempt to allocate size bytes;
      if (the allocation was successful)
           return (a pointer to the memory);
      // allocation was unsuccessful; find out what the
      // current new-handling function is (see below)
      new_handler globalHandler = set_new_handler(0);
      set_new_handler(globalHandler);
      if (globalHandler) (*globalHandler)();
      else throw std::bad_alloc();
   }
}


class Base{
      public:
            static void* operator new(std::size_t size) throw(std::bad_alloc);
            ...

};
class Derived:publicBase{

      //什么也没写
};

Derived *p =new Derived;//调用的是 Base::operator new
```

//对于 base class 的 operator new 操作，在其 derived class 不重写的情况下，可以这样写 base //class 的 operator new：non-menber 版本

```
void * Base::operator new(std::size_t size) throw(std::bad_alloc)
{
   if (size != sizeof(Base))                  // if size is "wrong,"
       return ::operator new(size);             // have standard operator
                                                          // new handle the request
   ...                                                  // otherwise handle
```

```
                                                    // the request here
}
```

//独立对象的 size 不会为 0。operator delete 的唯一要求就是“保证 delete NULL pointer 永远安全”：

```
void operator delete(void *rawMemory) throw()
{
   if (rawMemory == 0) return;                    // do nothing if the null
                                                        // pointer is being deleted
   deallocate the memory pointed to by rawMemory;
}
//member 版本
class Base{
      public:
            static void* operator new(std::size_t size) throw(std::bad_alloc);
            static void* operator delete(void* raeMemory,std::size_t size) throw();
            ...

};
void Base::operator delete(void *rawMemory, std::size_t size) throw()
{
   if (rawMemory == 0) return;                    // check for null pointer
   if (size != sizeof(Base)) {                  // if size is "wrong,"
        ::operator delete(rawMemory);           // have standard operator
        return;                                     // delete handle the request
   }
```

规则:52
/*写了 placement new 也要写 placement delete*/
//operator new 和 operator delete 要成对使用，不仅仅是指形式上要搭配，在本质上 new 和
//delete 也要是对应的

```
// normal forms of new & delete
void* operator new(std::size_t) throw(std::bad_alloc);
void operator delete(void *rawMemory) throw();    // normal signature at global scope
void operator delete(void *rawMemory,std::size_t size) throw();   // normal signature at class
                                                                  //scope

// placement version of new & delete
void* operator new(std::size_t, std::ostream& logStream) throw(std::bad_alloc);
    // "placement new"
void operator delete(void *rawMemory, std::ostream&) throw();
    // "placement delete"
```

//规则：如果一个带额外参数的 operator new 没有带额外参数的对应版本的 operator delete ，
//那么内存分配动作需要取消并恢复旧观时就没有任何 operator delete 会被调用

```
class Widget{

    public:
        ...
    static void* operator new(std::size_t size,std::ostream& logstream) throw(std::bad_alloc);
                                                        // "placement new"
    static void operator delete(void *pMemory) throw();//normal signature
    static void operator delete(void *rawMemory, std::ostream&logStream) throw();
                                                        // "placement delete"
    ...

};
Widget *pw=new(std::cerr) Widget;//不会内存泄漏
delete pw;//调用正常的 operator delete
```

//防止作用域名字遮掩问题

```
class StandardNewDeleteForm{
    //内含所有的正常形式的 new 和 delete

    //normal new/delete
    static void* operator new(std::size_t size) throw(std::bad_alloc);
    {
            return ::operator new(size);
```

```
    }
    static void operator delete(void *rawMemory) throw();
    {
        return ::operator delete(rawMemory);
    }

    //placement new/delete
    static void* operator new(std::size_t size,void* ptr) throw();
    {
        return ::operator new(size,ptr);
    }
    static void operator delete(void *rawMemory,void* ptr) throw();
    {
        return ::operator delete(rawMemory,ptr);
    }

    //nothrow new/delete
    static void* operator new(std::size_t size,const std::nothrow_t& nt) throw();
    {
        return ::operator new(size,nt);
    }
    static void operator delete(void *rawMemory,const std::nothrow_t& nt) throw();
    {
        return ::operator delete(rawMemory,nt);
    }
};

class Widget:public StandardNewDeleteForm{
    public:
    using StandardNewDeleteForm::operator new;
    using StandardNewDeleteForm::operator delete;
    static void* operator new(std::size_t size,std::ostream& logstream) throw(std::bad_alloc);
                                                    // "placement new"
    static void operator delete(void *rawMemory, std::ostream&logStream) throw();
                                                    // "placement delete"



};
```

规则 53

```
/*不要忽视编译信息*/
class B{
	public:
		virtual void f() const;
};

class D:public B{

	public:
		virtual void f();//由于未声明为 const　只是对 B::f 的覆盖　而不是重新声明
};
```