Item M1：指针与引用的区别

- 适用引用的效率高于指针：因为不存在指向空值的引用，在使用引用之前不需要测试它的合法性。 （例外：`char *pc = 0;char& rc = *pc`）
- 应用场合：始终指向一个对象，或者在重载操作符时为防止不必要的语义误解时，你应该使用引用。而在除此之外的其他情况下，则应使用指针。

```
vector<int> v(10);
v[5] = 10; //如果是引用，无需*
*v[5] = 10;//如果是指针，则需要添加*
```

Item M2：尽量使用C++风格的类型转换

- `static_cast` 在功能上基本上与C风格的类型转换一样强大，含义也一样。
- `const_cast` 用于类型转换掉表达式的 `const` 或 `volatileness` 属性。
- `dynamic_cast` 用于安全地沿着类的继承关系向下进行类型转换。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时）
- `reinterpret_casts` 的最普通的用途就是在函数指针类型之间进行转换。

Item M3：不要对数组使用多态

```
class BST { ... };
class BalancedBST: public BST { ... };
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements) {
  for (int i = 0; i < numElements; ) {
    s << array[i];
  }
}
BalancedBST bBSTArray[10];
printBSTArray(cout, bBSTArray, 10);          // 还会运行正常么？
```

- 在这种情况下，编译器原先已经假设数组中元素与 `BST` 对象的大小一致，但是现在数组中每一个对象大小却与 `BalancedBST` 一致，执行的结果无法预料。
- 通过一个基类指针来删除一个含有派生类对象的数组，结果将是不确定的（执行基类的析构函数而不执行派生类的）。

Item M4：避免无用的缺省构造函数

- 缺省构造函数则可以不利用任何在建立对象时的外部数据就能初始化对象。对于很多对象来说，不利用外部数据进行完全的初始化是不合理的。
- 如果一个类的构造函数能够确保所有的部分被正确初始化，则成员函数不用去测试所有的部分是否被正确初始化。缺省构造函数一般不会提供这种保证。
- 特别是如果一个类没有缺省构造函数，就会存在一些使用上的限制。

1、建立数组时，一般来说，没有一种办法能在建立对象数组时给构造函数传递参数（可以避开该限制）。

2、它们无法在许多基于模板（template-based）的容器类里使用。

3、设计虚基类时所面临的要提供缺省构造函数还是不提供缺省构造函数的两难决策。不提供缺省构造函数的虚基类，很难与其进行合作。因为几乎所有的派生类在实例化时都必须给虚基类构造函数提供参数。

- 无缺省构造函数的对象数组的构造方法

方法 1：
```
EquipmentPiece bestPieces[] = {
  EquipmentPiece(ID1),
  ...,
  EquipmentPiece(ID10)
};
```
方法 2：
```
typedef EquipmentPiece* PEP;
PEP bestPieces[10];
PEP *bestPieces = new PEP[10];
for (int i = 0; i < 10; ++i)
  bestPieces[i] = new EquipmentPiece( ID Number );
```
方法 3：placement new
```
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);
for (int i = 0; i < 10; ++i)
  new (&bestPieces[i]) EquipmentPiece( ID Number );
//删除
for (int i = 9; i >= 0; --i)
  bestPieces[i].~EquipmentPiece();
operator delete[](rawMemory);
```

Item M5：谨慎定义类型转换函数

- 允许编译器进行隐式类型转换的两种函数：单参数构造函数（有缺省值的多参数也行）和隐式类型转换运算符。
- operator char *()，返回类型默认就是 char *，避免使用其他类型。
- 建议用命名函数来替代 operator char *()（比如 string::c_str()）
- 用 explicit 来申明单参数的构造函数来避免隐式类型转换

Item M6：自增(increment)、自减(decrement)操作符前缀形式与后缀形式的区别

- ++/--：C++规定后缀形式有一个 int 类型参数，当函数被调用时，编译器传递一个 0 给该函数

- 前缀形式返回一个引用，后缀形式返回一个 const 类型（避免此类调用：i++++; i.operator++(0).operator++(0); ）。
- 为确保前缀和后缀形式行为一致，后缀 increment 和 decrement 应该根据它们的前缀形式来实现。

Item M7：不要重载“&&”，“||”，或“,”

- C、C++布尔表达式使用短路求值法(short-circuit evaluation)，如果重载“&&”，“||”则变为函数调用，破坏了该原则（1、两个参数都需要计算，2、参数的计算顺序不定）。
- 逗号表达式首先计算左边的表达式，然后计算右边的表达式；结果是右边表达式的值。
- 如果你没有一个好理由重载操作符，就不要重载。

Item M8：理解各种不同含义的 new 和 delete

- new 操作符完成的功能分成两部分。第一部分是分配足够的内存以便容纳所需类型的对象。第二部分是它调用构造函数初始化内存中的对象。new 操作符总是做这两件事情，你不能以任何方式改变它的行为。
- new 操作符调用一个函数来完成必需的内存分配，你能够重写或重载这个函数来改变它的行为。new 操作符为分配内存所调用函数的名字是 operator new。函数 operator new 通常这样声明：void * operator new(size_t size);
- placement new，在已分配的内存中构造一个对象

```
void * operator new(size_t, void *location){
  ...
  return location;
}
```

Item M9：使用析构函数防止资源泄漏

- 利用 auto_ptr 或者类似的机制

Item M10：在构造函数中防止资源泄漏

- 利用 auto_ptr 包装堆成员（需要 new）来避免部分构造时（构造过程抛出异常）已构造的对象无法释放（此时不会调用析构函数，只会调用::operator delete(obj,sizeof(T))清理内存，意味着会清理栈不会清理异常前面分配的堆内存，auto_ptr 把堆伪装成栈从而得到清理）；

Item M11：禁止异常信息（exceptions）传递到析构函数外

- 异常传递的堆栈展开（stack-unwinding）过程中，可能会导致对象删除。如果在删除的过程中，析构函数也抛出异常，并导致程序控制权转移到析构函数外，C++将调用 terminate 函数。

- 如果一个异常被析构函数抛出而没有在函数内部捕获住，那么析构函数就不会完全运行（它会停在抛出异常的那个地方上）

```
Session::~Session(){
  try {
    logDestruction(this);
  }
  catch (...) {  }
}
```

Item M12：理解“抛出一个异常”与“传递一个参数”或“调用一个虚函数”间的差异

- 当异常对象被拷贝时，调用的是对象的静态类型（static type）所对应类的拷贝构造函数。

```
SpecialWidget localSpecialWidget;
...
Widget& rw = localSpecialWidget;
throw rw;  //它抛出一个类型为Widget的拷贝，不同于函数调用
```

- 带有 const void* 指针的 catch 子句能捕获任何类型的指针类型异常。
- catch 子句匹配异常类型时不会进行隐式的类型转换，但支持多态。

Item M13：通过引用（reference）捕获异常

- 优点：

  1．不需要删除异常对象（通过指针 throw 和 catch）

  2．能够避开异常对象切片（通过值 catch）

  3．能够捕获标准异常类型

  4．减少了异常对象被拷贝的次数

Item M14：审慎使用异常规格(exception specifications)

- 违反异常规范的函数默认调用 unexpected，其缺省行为是调用 terminate（新 C++规则允许抛出一个 un_expected 异常）。
- 编译器仅仅能部分地检测异常规格的使用是否一致
- 异常规格会阻止 high-level 异常处理器来处理 unexpected 异常
- 异常规范是理想主义，是不现实的

Item M15：了解异常处理的系统开销

- 不使用任何异常处理特性：你需要空间建立数据结构来跟踪对象是否被完全构造，你也需要 CPU 时间保持这些数据结构不断更新。
- try 块：粗略地估计，使用 try 块后，代码的尺寸将增加 5%－10%并且运行速度

也同比例减慢。

- 异常规格：编译器为异常规格生成的代码与它们为 try 块生成的代码一样多，所以一个异常规格一般花掉与 try 块一样多的系统开销。
- 抛出异常：与一个正常的函数返回相比，通过抛出异常从函数里返回可能会慢三个数量级。
- map<string, int>::iterator iter;

  建议使用(*iter).second 形式而非 iter->second 形式。

  简单地说，iterator 是一个对象，不是指针，所以不能保证”->”被正确应用到它上面，而 STL 要求"."和"*"在 iterator 上是合法的。

Item M16：牢记 80—20 准则（80—20 rule）

- 80—20 准则说的是大约 20%的代码使用了 80%的程序资源；大约 20%的代码耗用了大约 80%的运行时间；大约 20%的代码使用了 80%的内存；大约 20%的代码执行 80%的磁盘访问；80%的维护投入于大约 20%的代码上。
- 用 profiler 程序而不是主观猜测来识别出令人讨厌的程序的 20%部分。

Item M17：lazy evaluation（懒惰计算法）

- 应用场合：当你必须支持某些操作而不总需要其结果时，lazy evaluation 是在这种时候使用的用以提高程序效率的技术。
- 引用计数（比如：base_string 的写时复制技巧）
- 懒惰提取（比如：大对象从数据库初始化）

  如果编译器不支持 mutable，可以使用 fakeThis 技巧

```
const string& LargeObject::field1() const{
  // 声明指针, fakeThis, 其与 this 指向同样的对象
  // 但是已经去掉了对象的常量属性
  LargeObject * const fakeThis =
    const_cast<LargeObject* const>(this);
  if (field1Value == 0) {
    fakeThis->field1Value = <from database>;
  }
  return *field1Value;
}
```

- 懒惰表达式计算（比如：矩阵运算）

Item M18：over-eager evaluation（过度热情计算法）

- 应用场合：当你必须支持某些操作而其结果几乎总是不止一次地需要时，在这种时候 over-eager 是提高程序效率的一种技术。
- cache 计算结果：跟踪集合的平均值，最大值，最小值，然后需要时无需计算。

- Prefetch：内存的分配，分配内存时预先分配一些备用。

Item M19：理解临时对象的来源

- 当传值或传递常量引用时，才会发生这些类型转换。
- C++语言禁止为非常量引用 （reference-to-non-const）产生临时对象（如果函数修改了参数，则修改的是临时对象）。
- 返回函数对象也需要创建临时对象

Item M20：协助完成返回值优化（return value optimization）

- 编译器会优化无名的临时对象，优化方案：调用构造函数返回临时对象

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

Item M21：通过重载避免隐式类型转换

- 操作符重载

```
class UPInt {
public:
  UPInt();
  UPInt(int value);
  ...
};
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);
const UPInt operator+(const UPInt& lhs, int rhs);
const UPInt operator+(int lhs, const UPInt& rhs);
UPINT upi2, upi1;
upi2 = 10 + upi1;
```

- 在C++中有一条规则是每一个重载的operator必须带有一个用户定义类型的参数。
- 不过，必须谨记80－20规则。没有必要实现大量的重载函数，除非你有理由确信程序使用重载函数以后其整体效率会有显著的提高。

Item M22：考虑用运算符的赋值形式（operator +=）取代其单独形式（operator +）

```
class Rational {
public:
  ...
  Rational& operator+=(const Rational& rhs);
  Rational& operator-=(const Rational& rhs);
```

```
};
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  return T(lhs) += rhs; //进行了返回值优化
}
template<class T>
const T operator-(const T& lhs, const T& rhs)
{
  return T(lhs) -= rhs; //进行了返回值优化
}
```

Item M23：考虑变更程序库

- 如果你的程序有 I/O 瓶颈，你可以考虑用 stdio 替代 iostream，如果程序在动态分配和释放内存上使用了大量时间，你可以想想是否有其他的 operator new 和 operator delete 的实现可用。因为不同的程序库在效率、可扩展性、移植性、类型安全和其他一些领域上蕴含着不同的设计理念，通过变换使用给予性能更多考虑的程序库，你有时可以大幅度地提高软件的效率。

Item M24：理解虚拟函数、多继承、虚基类和 RTTI 所需的代价

- 大多数程序和程序库由多个 object 文件连接而成，但是每个 object 文件之间是独立的。哪个 object 文件应该包含给定类的 vtbl 呢？

  1. IDE 的厂商：为每一个可能需要 vtbl 的 object 文件生成一个 vtbl 拷贝，连接程序然后去除重复的拷贝。

  2. 采用启发式算法（更普遍）：包含该类的第一个非内联、非纯虚拟函数定义的 object 文件包含 vtbl。

- RTTI：运行时类型信息，只有多态类型才有，比如：vtbl 数组的索引 0 处可以包含一个 type_info 对象的指针，这个对象属于该 vtbl 相对应的类。

Item M25：将构造函数和非成员函数虚拟化

```
class NewsLetter {
public:
  ...
private:
  static NLComponent * readComponent(istream& str);
   ...
};
NewsLetter::NewsLetter(istream& str){
    while (str) {
        components.push_back(readComponent(str));
    }
 }
```

- readComponent 根据所读取的数据建立了一个不同的对象，它的行为与构造函数相似，我们称它为虚拟构造函数。
- 特殊的虚拟构造函数（虚拟拷贝构造函数 clone）

```
class NLComponent {
public:
  virtual NLComponent * clone() const = 0;
  ...
};
class TextBlock: public NLComponent {
public:
  virtual TextBlock * clone() const
  { return new TextBlock(*this); }
  ...
};
class Graphic: public NLComponent {
public:
  virtual Graphic * clone() const
  { return new Graphic(*this); }
  ...
};
//虚拟拷贝构造函数的应用
NewsLetter::NewsLetter(const NewsLetter& rhs){
  for (list<NLComponent*>::const_iterator it =
         rhs.components.begin();
       it != rhs.components.end();
       ++it) {
    components.push_back((*it)->clone());
  }
}
```

- 非成员的虚拟函数（调用虚拟函数）

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...
};
class TextBlock: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...
};
class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...
};
inline
ostream& operator<<(ostream& s, const NLComponent& c){
  return c.print(s);
}
```

Item M26：限制某个类所能产生的对象数量

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};                          //异常类
  static int objectCount() { return numObjects; }
protected:
  Counted(){init();};
  Counted(const Counted& rhs){init();}
  ~Counted() { --numObjects; }
private:
  static int numObjects; //用模板的静态成员来计数
  static const size_t maxObjects;
  //构造函数抛出异常，要注意避免内存泄漏，
  //（确保抛出异常前的new资源得到释放）
  void init(){
    if (numObjects >= maxObjects) throw TooManyObjects();
    ++numObjects;
  }
};
//实现文件定义numObjects，自动初始化为0
template<class BeingCounted>
int Counted<BeingCounted>::numObjects;

//下面的定义让用户来完成，GCC需要在前面加 template<>
const size_t Counted<Printer>::maxObjects = 10;
//使用
class Printer: private Counted<Printer> {
public:
  Printer();
  Printer(const Printer& rhs);
  ~Printer();
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
  using Counted<Printer>::objectCount; //恢复处于public接口下
  using Counted<Printer>::TooManyObjects;
}
```

- 此处私有继承优点

  1. 继承实现，用户不关心细节

  2. 基类可以避免申明虚拟的析构函数

Item M27：要求或禁止在堆中产生对象

- 要求在堆中建立对象（申明析构函数为非public，然后定义一个伪析构函数）

```
class UPNumber {
public:
```

```
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);
  // 伪析构函数 (一个 const 成员函数， 因为
  // 即使是 const 对象也能被释放。)
  void destroy() const { delete this; }
  ...
private:
  ~UPNumber();
};
```

- 检查对象是否在堆中，实现一个基类，然后从基类继承

```
class HeapTracked {
public:
  class MissingAddress{};
  virtual ~HeapTracked() = 0;
  static void *operator new(size_t size){
    void *memPtr = ::operator new(size);
    addresses.push_front(memPtr);
    return memPtr;
  }
  static void operator delete(void *ptr){
    list<RawAddress>::iterator it =
      find(addresses.begin(), addresses.end(), ptr);
    if (it != addresses.end()) {
      addresses.erase(it);
      ::operator delete(ptr);
    } else {
      throw MissingAddress();
    }
  }
  bool isOnHeap() const{
    //dynamic_cast 此时目的是为了得到对象的开始地址
    //dynamic_cast 只能用于“多态对象”的指针上
    const void *rawAddress = dynamic_cast<const void*>(this);
    list<RawAddress>::iterator it =
      find(addresses.begin(), addresses.end(), rawAddress);
    return it != addresses.end();
  }
private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
list<HeapTracked::RawAddress> HeapTracked::addresses;
HeapTracked::~HeapTracked() {}
```

- 禁止堆对象（无法达到目的）

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
```

```
  ...
};
```

1. 作为其他类（比如 Contain）的成员时调用的是 Contain::operator new 或::operator new
2. 可以通过派生类重新定义 new、delete 来绕过

Item M28：灵巧（smart）指针

- 通过传值方式传递 auto_ptr 对象是一个很糟糕的方法。函数返回后，传入的参数已经失去了所有权。
- 只有在你确实想把对象的所有权传递给一个临时的函数参数时，才能通过传值方式传递 auto_ptr。
- 在 C++类库标准中，void*类型的隐式转换已经被 bool 类型的隐式转换所替代，operator bool 总是返回与 operator!相反的值。
- 在同一时间编译器进行隐式类型转换最多一次，但是可以构造多次。
- 作为参数传递时，灵巧指针不能代替指针，不支持多态（不能把 smart_ptr<Derive>传递给 void fun(smart_ptr<Base>)）。

Item M29：引用计数(写时拷贝)

- C++编译器没有办法告诉我们一个特定的 operator[]是用作读的还是写的，所以我们必须保守地假设“所有”调用非 const operator[]的行为都是为了写操作。

```
String s1 = "Hello";
String s2 = s1;
char *p = &s1[1];
```

- 解决办法（降低值共享于对象间的次数）：在每个 StringValue 对象中增加一个标志以指出它是否为可共享的。在最初（对象可共享时）将标志打开，在非 const 的 operator[]被调用时将它关闭。一旦标志被设为 false，它将永远保持在这个状态。

```
class String {
private:
  struct StringValue {
    int refCount;
    bool shareable;                 // add this
    char *data;
    StringValue(const char *initValue);
    ~StringValue();
  };
...
};
String::StringValue::StringValue(const char *initValue)
:   refCount(1),
    shareable(true){
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
```

```
}
String::StringValue::~StringValue(){
  delete [] data;
}
String::String(const String& rhs){
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }
  else {
    value = new StringValue(rhs.value->data);
  }
}
//const 用来标明是读函数，非 const 用来标明是写函数是不精确的。
//编译器仅仅根据调用对象的 const 属性来选择此成员函数的 const
//和非 const 版本，而不考虑调用时的环境。
const char& String::operator[](int index) const {
  return value->data[index];
}
//意味着这里也有读函数调用它
char& String::operator[](int index){
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }
  value->shareable = false;
  return value->data[index];
}
```

Item M30：代理类

- 通过代理类实现[][]，对用户透明的代理类

```
template<class T>
class Array2D {
public:
  class Array1D {
  public:
    Array2D(int dim1, int dim2);
    T& operator[](int index);
    const T& operator[](int index) const;
    ...
  };
  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...
};

Array2D<float> data(10, 20);
...
cout << data[3][6]; //通过代理类 Array1D 实现
```

- 通过代理类阻止隐式类型转换，适用于不支持 mutable 关键字的编译器

```
template<class T>
class Array {
public:
  class ArraySize {                        // 这个类是新的
  public:
    //利用了在同一时间编译器进行隐式类型转换最多一次，可以构造多次特性
    ArraySize(int numElements): theSize(numElements) {}
    int size() const { return theSize; }
  private:
    int theSize;
  };
Array(int lowBound, int highBound);
  Array(ArraySize size);                   // 注意新的声明
...
};
```

- 利用代理类区分 operator[]的读形式和写形式

```
class String {
public:
  class CharProxy {
  public:
    CharProxy(String& str, int index);
    CharProxy& operator=(const CharProxy& rhs);//write
    CharProxy& operator=(char c);              //write
    operator char() const;                     //read
  private:
    String& theString;
    int charIndex;
  };
  const CharProxy
    operator[](int index) const;
  CharProxy operator[](int index);
  ...
friend class CharProxy;
private:
  RCPtr<StringValue> value;
};
//
const String::CharProxy String::operator[](int index) const{
  return CharProxy(const_cast<String&>(*this), index);
}
String::CharProxy String::operator[](int index){
  return CharProxy(*this, index);
}
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
String::CharProxy::operator char() const {
  return theString.value->data[charIndex];
}
String::CharProxy&
```

```
String::CharProxy::operator=(const CharProxy& rhs) {
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];
  return *this;
}
String::CharProxy& String::CharProxy::operator=(char c){
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }
  theString.value->data[charIndex] = c;
  return *this;
}
//上面实现的局限性，建议在没有其它方法可采用的情况使用。
String s1 = "Hello";
char *p = &s1[1]; //不能编译通过，除非你重载 operator &

String s2 = "Hello";
s2[1] += s2[2];  //不能编译通过，除非你重载 operator +=

void swap(char& a, char& b);
String s = "+C+";
swap(s[0], s[1]); //不能编译通过，隐式转换返回的是临时对象，
                  //无法将临时对象传递给非 const 的引用参数
```

Item M31：让函数根据一个以上的对象来决定怎么虚拟

- 二重调度（double dispatch）问题：如果处理过程只取决于 object1 的动态类型，可以将 process() 设为虚函数，并调用 object1.process(object2)，如果只取决于 object2 的动态类型，也可以同样处理。如果取决于两个对象的动态类型。虚函数体系只能作用在一个对象身上，它不足以解决问题。
- 实现方法（最好从设计上避免该情况出现）

  1．虚函数＋RTTI

```
void SpaceShip::collide(GameObject& otherObject) {
  const type_info& objectType = typeid(otherObject);
  if (objectType == typeid(SpaceShip)) {
     ...
  }
  else if (objectType == typeid(SpaceStation)) {
    ...
  }
  ...
}
```

  2．虚函数＋重载

```
class SpaceShip: public GameObject {
public:
```

```
  virtual void collide(GameObject&        otherObject);
  virtual void collide(SpaceShip&         otherObject);
  virtual void collide(SpaceStation&      otherObject);
  virtual void collide(Asteroid&          otherobject);
  ...
};
```

Item M32：在未来时态下开发程序

- 用C++语言自己来表达设计上的约束条件，而不是用注释或文档。
- 避免 “用户要求型”的虚函数，应该判断一个函数的含意，以及它被派生类重定义的话是否有意义。如果是有意义的，申明它为虚，即使没有人立即重定义它。如果不是的话，申明它为非虚，并且不要在以后为了便于某人而更改。
- 处理类的赋值和拷贝构造函数，即使“从没人这样做过”。
- 基于最小惊讶法则：努力提供这样的类，它们的操作和函数有自然的语法和直观的语义。和内建数据类型的行为保持一致：拿不定主意时，仿照 `int` 来做。
- 一般而言，只要能编译通过，就有人会这么做。所以，要使得自己的类易于被正确使用而难以误用，所以要将你的类设计得可以防止、检测或修正这些错误。
- 努力于可移植的代码：只有在性能极其重要时采用不可移植的结构才是可取的。优点：更换平台是比较容易，扩大你的用户基础，吹嘘支持开放平台。这也使得你赌错了操作系统时比较容易补救。
- 将你的代码设计得当需要变化时，影响是局部的。尽可能地封装；将实现细节申明为私有。
- 只要可能，使用无名的命名空间和文件内的静态对象或函数。
- 避免导致虚基类的设计，因为这种类需要每个派生类都直接初始化它——即使是那些间接派生类。
- 避免需要 `RTTI` 的设计，它需要 `if...then...else` 型的瀑布结构。
- 你需要虚析构函数，只要有人 `delete` 一个实际值向 `D` 的 `B *`。
- 非尾端类应该是抽象类。在处理外来的类库时，你可能需要违背这个规则；但对于你能控制的代码，遵守它可以提高程序的可靠性、健壮性、可读性、可扩展性。

Item M34：如何在同一程序中混合使用C++和C

- 名变换：解决方案

```
#ifdef __cplusplus
extern "C" {
#endif

...

#ifdef __cplusplus
}
#endif
```

- 静态初始化

  在 main 执行前和执行后都有大量代码被执行。只要程序的任意部分是 C++写的，你就应该用 C++写 main()函数。如果不能用 C++写 main()，你就有麻烦了，因为没有其它办法确保静态对象的构造和析构函数被调用了。不是说没救了，只是处理起来比较麻烦一些。编译器生产商们知道这个问题，几乎全都提供了一个额外的体系来启动静态初始化和静态析构的过程。
- 动态内存分配

  strdup 函数，它并不在 C 和 C++标准（运行库）中，却很常见：
  ```
  char * strdup(const char *ps);
  ```
  用 delete？用 free？如果你调用的 strdup 来自于 C 函数库中，那么是后者。如果它是用 C++写的，那么恐怕是前者。
  尽可能避免调用那些既不在标准运行库中的函数
- 数据结构的兼容性

  只有非虚函数的结构或类（不能有虚函数，不能继承）兼容于它们在 C 中的孪生版本

Item M35：让自己习惯使用标准 C++语言

- ISO/ANSI C++标准是厂商实现编译器时将要考虑的，是作者们准备出书时将要分析的，是程序员们在对 C++发生疑问时用来寻找权威答案的。