Broadview®
www.broadview.com.cn

```
cout<<"input a b c\n";
cin>>a>>b>>c;
if((a+b<c)||(b+c<a)||(a+c<b))
        cout<<"Not Triangle\n";
else
        cout<<"Triangle\n";
}
```

C/C++
程序缺陷与优化

```
#include<iostream>
using namespace std;
void main()
{
    float a,b,c;
    cout<<"input a b c\n";
```

于秀山 许 峰 李华莹
刘 然 于长钺 杨玲萍 编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

C/C++程序缺陷与优化

- 帮助程序员解决如下问题："为什么我编的程序中有这么多缺陷？如何减少缺陷？"
- 跳出程序员固有思维定势，修正长期的习惯性思维，使其觉察到自身设计中存在的问题
- 书中案例均来自于实际项目，并进行透彻分析。

阅读本书，如同对照一面镜子，
将使作为程序员的你有一种幡然醒悟，相见恨晚的感觉！

博文视点Broadview

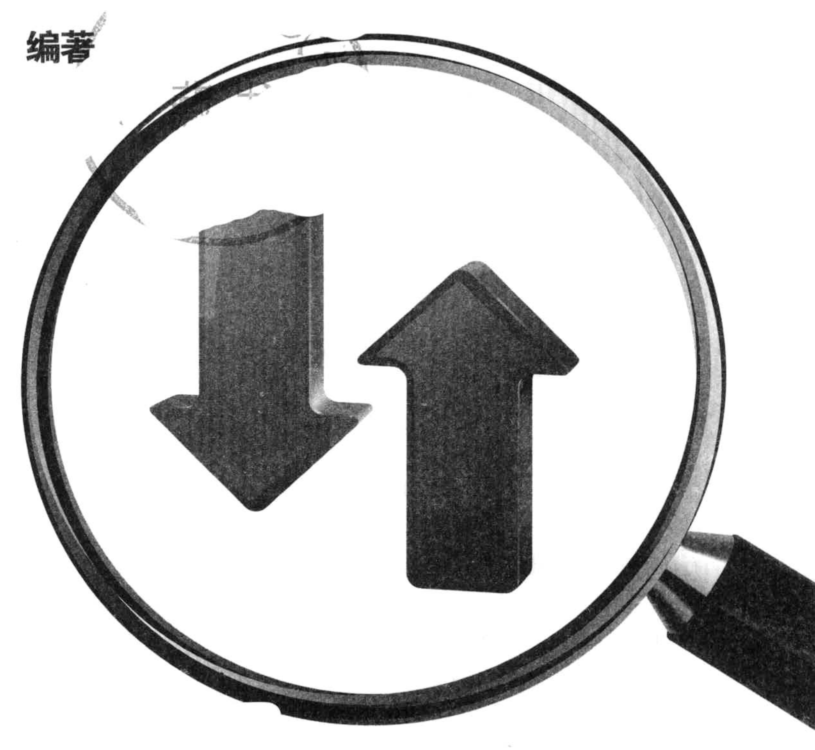
@博文视点Broadview

ISBN 978-7-121-22632-8
9 787121 226328
定价：49.00元

责任编辑：郭 立
封面设计：吴海燕

C/C++ 程序缺陷与优化

于秀山　许　峰　李华莹
刘　然　于长钺　杨玲萍　编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

TP312C
2298

内 容 简 介

程序设计可谓是一个汗牛充栋的话题。与传统的C/C++程序设计方面的书籍不同，本书从另外一个视角——程序缺陷的角度来探讨程序设计与优化。

本书从作者所从事的软件测试项目中精选了与C/C++语言有关的程序缺陷，主要包括编码风格、内存管理、内存泄漏、缓冲区溢出、指针使用、安全等方面。对于每一种缺陷，通过实例分析了缺陷产生的原因，并给出了具体的修改和优化方法。面对这些缺陷，程序员会有一种似曾相识、相见恨晚的感觉。通过这些缺陷，程序员能够跳出固有的程序设计思维定式，使其翻然醒悟，茅塞顿开。

本书适合于有一定编程经验的软件开发人员和测试人员使用，也可作为高等院校计算机相关专业高级程序设计及软件测试课程教材。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

C/C++程序缺陷与优化 / 于秀山等编著. —北京：电子工业出版社，2014.4
ISBN 978-7-121-22632-8

Ⅰ. ①C… Ⅱ. ①于… Ⅲ. ①C语言—程序设计 Ⅳ.①TP312

中国版本图书馆CIP数据核字（2014）第047653号

责任编辑：郭　立
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　印张：17.5　字数：383千字
印　　次：2014年4月第1次印刷
定　　价：49.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

C/C++程序设计是一个既古老又时尚的话题，其古老性表现在几乎任何一个程序员都对其有所了解，都有过使用该语言开发软件的经验；其时尚性表现在虽然历经几十年的演变，这两种语言依然经久不衰，仍然在各个领域得到广泛使用，C/C++程序设计几乎成为每一个程序员的必修课。

虽然大多数程序员都经过了系统的程序设计方面的培训，但编写的软件中仍然存在大量的缺陷，甚至是很低级的缺陷，这些缺陷严重影响了软件质量。

“软件中为何还会存在这样的缺陷？”这是令管理者和程序员经常困惑的一个问题，也是笔者所关注的问题。笔者长期从事软件测试方面的工作，亲历了大量各式各样的软件缺陷，这些缺陷使笔者萌生了从另外一个角度透视程序设计的想法。

与传统的 C/C++程序设计方面的书籍不同，本书从另外一个视角——程序设计缺陷的角度来探讨程序设计。程序员长期形成的习惯性思维，使其难以觉察到自身在程序设计方面存在的问题，可谓“不识庐山真面目，只缘身在此山中”。本书列举了大量来自实际项目中出现的软件缺陷，这些缺陷就像一面镜子，面对这些缺陷，程序员会有一种似曾相识、相见恨晚的感觉。通过这些缺陷，程序员能够跳出固有的程序设计思维定式，使其翻然醒悟，茅塞顿开。

前车之覆，后车之鉴，期望本书能够使读者充分借鉴前人在 C/C++程序设计方面的经验教训，快速提升自己的程序设计水平。

本书由于秀山、许峰、李华莹、刘然、于长钺、杨玲萍编著。在本书的编写过程中，尹浩、严少清、董昕、刘怡静同志参与了部分章节的编写工作，在此向他们表示衷心感谢。

鉴于作者才疏学浅，书中难免有遗漏和错误之处，敬请读者斧正。

作　者

2013 年秋于北京

目　录

第 1 章　语言使用基本问题……1

1.1　变量使用问题……1

1.2　运算符使用问题……24

1.3　函数问题……47

1.4　条件语句问题……57

1.5　循环语句问题……64

1.6　数值类型转换问题……67

第 2 章　内存管理……85

2.1　内存分配与使用……87

2.2　内存泄漏……96

第 3 章　缓冲区溢出……118

3.1　数组越界……119

3.2　数据越界……124

3.3　字符串操作溢出……125

第 4 章　指针问题……141

4.1　空指针解引用……142

4.2　指针非法使用……148

第 5 章　安全缺陷……158

5.1　外部输入安全缺陷……158

5.2　资源泄漏……162

5.3　其他……169

第 6 章 与类有关的编程缺陷 …… 174

第 7 章 其他 …… 208

7.1 预处理 …… 208
7.2 异常 …… 215
7.3 多线程和同步性 …… 226
7.4 代码不可达 …… 229

附录 A 常用静态分析工具 …… 234

A.1 PolySpace——运行时错误静态检查工具 …… 234
A.1.1 PolySpace Verifier …… 235
A.1.2 PolySpace Viewer …… 238
A.2 Klocwork——代码静态检查工具 …… 240
A.2.1 工程创建与分析 …… 241
A.2.2 分析结果查看 …… 244
A.3 Testbed——静态和动态测试工具 …… 250
A.3.1 单个文件分析 …… 251
A.3.2 分析结果查看 …… 254
A.3.3 多个文件批量分析 …… 263
A.4 McCabe IQ2——软件质量保证工具 …… 265
A.4.1 McCabe EQ …… 265
A.4.2 McCabe Test …… 272
A.4.3 McCabe Reengineer …… 273

参考文献 …… 274

第 1 章 语言使用基本问题

C/C++是一种介于汇编语言和高级语言之间的中级语言，自 C/C++语言问世以来，由于其具有运行速度快、可移植、容易理解、便于阅读和书写等特点，使得这两种语言经久不衰，成为很多程序员的首选编程语言。然而，C/C++语言，特别是 C 语言也存在一些固有的缺点，例如，C 语言不是强类型语言，它允许将整型变量赋值给字符型变量、C 语言缺少对字符串和记录的处理、C 语言缺少运行期间检查；为了追求效率，在运行期间，C 语言不检查诸如数组越界之类的错误；C 语言中有相当多的地方容易使程序员产生误解，例如，运算符优先级规则；C 语言中有很多地方的定义是不完备的，程序执行结果可能因编译器的不同而改变，某些情况下，即使在同一个编译器内也会根据上下文而发生变化。上述缺点使得 C/C++程序会产生各种不同的缺陷。其中，有些是变量名字书写错误、有多余的分号等这样的简单缺陷，有些是算法理解方面的复杂缺陷。

使用 C/C++语言可以编写出布局良好的、结构化的、表达性强的程序，也可以编写出不当的和特别难以理解的程序。在本章中，笔者重点介绍在 C/C++语言使用方面存在的基本问题，这些问题看似简单，似乎不屑一顾，而事实上，这些问题在具体项目中频繁出现，严重影响了软件质量。

1.1 变量使用问题

变量是程序处理的基本数据对象之一，它用于保存程序中不断变化的值、从外部接收的数据、保存中间结果及最终结果等。程序中所使用的任何变量和数据都必须遵循先定义后使用的原则。使用未初始化的变量是很常见的程序错误，编译器把变量存放在内存中的某个位置，如果变量没有正确初始化，则该变量存储的数据是未知的，在程序中直接使用该变量可能引起计算错误，甚至软件崩溃等问题。虽然许多编译器在检查出使用未初始化的变量时都会给出一个警告（Warning），但并不要求程序员必须修改此问题，而且，没有一个编译器能够发现所有使用未初始化变量的错误。

1. 未明确定义变量值

所有的变量在使用前都应该有明确定义的值。

例 1：

```
1   void foo( ) {
2   int b;
3   b++;
4   }
```

例 1 中第 3 行语句，变量 b 的值未明确定义却被使用。

修改方法：在第 2 行语句后面增加给变量 b 赋值的语句，如：b＝0;

2. 未初始化所有变量

为了避免使用没有初始化的变量，程序中所有的变量都应该初始化。

如果不初始化声明的整数变量，将得到一个随机值；如果不初始化声明的静态变量，编译器会自动将其初始化为 0；如果不初始化声明的指针变量，那么它所指向的内容是无法确定的，在这种情况下使用该指针变量时，将产生不可预料的后果。

例 2：

```
1   f()
2   {
3     int   *t; // 错误
4     …
5   }
```

例 2 中，f()函数仅仅声明了一个指向整数的指针 t，却没有对 t 进行初始化，指针变量 t 指向的地址将是不确定的，如果其他程序调用 f()函数，使用其中没有初始化的指针变量 t，则可能产生不可预料的错误。

修改方法：将第 3 行语句修改为：int *t=1;

3. 存在多余的变量

程序中定义并初始化了局部变量，但该变量未被使用。

例 3：

```
1   class A {
2   public:
3       void displBalance();
4   };
```

```
5   void foo( ) {
6      A a;        // 错误
7   }

8   void func( ) {
9      int i = 0;  // 错误
10  }
```

修改后的程序如例 4 所示。

例 4：

```
1   class A {
2   public:
3      void displBalance( );
4   };

5   void foo( ) {
6      A a;
7      a.displBalance( );
8   }

9   void func( ) {
10     int i = 0;
11     i++;
12  }
```

4. 使用未初始化的堆内存

程序中使用内存分配函数 malloc()为结构体分配堆内存后，堆内存没有初始化却被使用。

例 5：

```
1   struct student{
2     int num;
3     char[10] name;
4   };
5   int  main(int t)
6   {
7     struct student *s=malloc(sizeof(struct s));
8     int t;
9     t= s->num; // 错误
10  }
```

例 5 中第 9 行语句，结构体 s 还没有初始化却被使用。

5. 读取结构体中未赋值的局部变量

程序中结构体内的局部变量没有被赋值，但却读取该成员的值或将该成员作为参数传递给某个函数。

例 6：

```
1   struct  s
2   {
3     int  a;
4     int  b;
5   };
6   main()
7   {
8     s x;
9     x.b=0;
10    max(x.a,1); // 错误
11  }
```

例 6 中第 10 行语句，结构体 x 中的成员变量 a 还没有赋值，却作为参数传递给函数 max()。

6. 没有使用构造函数初始化类成员

类中所有的成员变量没有在类构造函数中初始化。

例 7：

```
1   class  c
2   {
3     private: int  i;
4              int  j;
5              bool flag;
6     public:  c()
7       {
8         i=0;
9         j=1;
10      }
11  };
```

例 7 中，类 c 的构造函数中缺少对成员变量 flag 的赋值语句。

7. 引用被初始化为地址可变的对象

引用总是指向一个对象，它只能在定义时被初始化一次，之后不可改变。引用和指针之间的区别如下。

（1）指针指向一块内存，它的内容是所指内存的地址，而引用是某块内存的别名。指针是一个实体，而引用仅是一个别名；

（2）使用引用时无须解引用（dereference），而指针需要解引用（*ptr，ptr 为指针）；

（3）引用只能在定义时被初始化一次，之后不可改变，而指针可改变；

（4）引用没有 const 类型，而指针有 const 类型，且 const 类型的指针不可改变；

（5）引用不能为空，而指针可以为空。

例 8：

```
1   void tmp(int* b)
2   {
3      if (b==NULL)
4       {
5          b=new int;
6          *b=200;
7       }
8      else
9       {
10        *b=100;
11      }
12  }
13  main()
14  {
15    int a=0;
16    tmp(&a);
17    int* c=NULL;
18    tmp(&c);    // 指针可以为空，但引用不能为空
19    return;
20  }
```

例 8 的第 18 行语句中，变量 c 是一个空指针，因为引用不能为空，所以“&c”不正确。
修改方法：将第 18 行语句修改为：tmp(c);

8. 静态成员未初始化

类中的静态数据成员是所有类对象共享的内容，其存放的是所有对象的值，而不是某个对象的值。

静态数据成员的初始化在类体外进行，初始化时不需要加访问权限符，因为静态数据成员是类的数据成员，所以在初始化时应指出其类名。

例 9:

```
1    class T
2     {
3      public:
4        T (int a,int b);
5         bb( );
6        private:
7         int x,y;
8         static int s;
9     };
10   T::T (int a,int b)//构造函数的实现部分
11   {
12     x=a;
13     y=b;
14   }
15   void T::bb( )//成员函数的实现部分
16   {
17     s=s+x+y;
18     cout<<"s="<<s<
19   }
20   main( )
21   {
22      T  t1(10,20), t2(5,3);
23      t1.bb( );
        t2.bb( );
24   }
```

例 9 中，T 类中有一个静态数据成员 s，该数据成员应该在类体外被定义。

修改方法：在第 10 行语句之前，增加为静态数据成员 s 赋初值的语句：int T::s=0;

9．赋值运算符（operator=）未给所有的变量赋值

程序中虽然使用赋值运算符（operator=），但没有给所有的变量赋值。

例 10:

```
1    class A
2    {
3       private:
```

```
4      int _x, _y, _z;
5      public:
6      A( ) { }
7      A& operator=( const A& a )
8      {
9          _x = a._x;
10         _y = a._y;
11         return *this;
12     }
13   };
```

例 10 中，第 7 行语句使用“operator=”的本意是给所有的成员赋值，但事实上仅仅对类 A 中的成员变量“_x、_y”进行了赋值，没有对类 A 中所有的成员变量进行赋值。

修改方法：在第 10 行语句后面增加语句：`_z = a._z;`

10. 工程头文件中包含变量的定义

头文件中最好只声明变量，而不定义变量。因为工程中的头文件会被.c 或.cpp 文件多次包含，如果头文件中包含变量的定义，势必造成变量的重复定义。因此，变量的定义应该写在.c 或.cpp 文件内。

例 11：

```
头文件  header.h
   int t=2; // 错误
```

例 11 中，头文件 header.h 包含了对变量 t 的初始化语句，头文件 header.h 如果多次被源代码（.c 或.cpp）文件包含，编译时该变量将被重复定义。

修改方法：将 int t=2 语句修改为：`int t;`

11. 无符号整数初始化为负数

无符号整数不能识别负数，所以初始化无符号整数为负数是错误的。

例 12：

```
1    F()
2    {
3       unsigned int y = -21; // 错误
4    }
```

例 12 中第 3 行语句，无符号整数 y 被错误赋值为负数“-21”。

修改方法：将有符号整数赋值为负数，即将第 3 行语句修改为：`signed int y = -21;`

一般情况下，有符号整数和无符号整数在内存中均占用 16 位，但这两种整数类型的取值范围是不同的，有符号整数的取值范围是-32768 到 32767，无符号整数的取值范围是 0 到 65535。

例 13：

```
1   bool fun(size_t cbSize)
2   {
3     if(cbSize > 1024)
4     rerurn false;
5     char *pBuf = new char[cbSize- 1];              //未对 new 的返回值进行检查
6     memset(pBuf, 0x90, cbSize- 1);
7     …
8     return true;
9   }
```

例 13 中，第 5 行语句调用 new()分配内存后，未对调用结果的正确性进行检查。如果 cbSize 为 0，则“cbSize – 1”为-1，但是 memset()函数中第 3 个参数本身是无符号数，因此会将-1 视为正的 0xFFFFFFFF，导致执行此函数时程序崩溃。

12．显式调用未定义的构造函数

如果程序员自定义的类中没有构造函数，编译器将为该类产生一个默认的构造函数，用于创建类对象时调用，但这个编译器产生的构造函数是无参函数，函数体为空，不做任何操作。为了给类中各成员变量赋初值，应该在创建的类中自定义构造函数。

例 14：

```
1   class A
2    {
3      public:
4          int b;
5          int a;
6    };
7   A* f()
8    {A *a
9      A *a=new A();
10     return new A();
11   }
```

例 14 中，类 A 中并没有定义自己的构造函数，而在 f()函数中调用“A()”。

修改方法：在第 5 行语句后面增加对构造函数的声明语句：

```
A(){ int b=1;  int a=2;};
```

13. 通过赋值方式初始化类的常量成员

调用类的构造函数时，构造函数将参数值传递给类中相应的数据成员。当构造函数的参数值是常量（const）时，常量参数值只能通过初始化方式传递给类中相应的数据成员。

例 15:

```
1   class A {
2   public:
3       A( const char *file, const char *path )
4       {
5           myFile = file;  // 错误
6           myPath = path;  // 错误
7       }
8   private:
9           string myFile;
10          string myPath;
11  };
```

例 15 中的第 5 行语句和第 6 行语句，因为类 A 的构造函数 A()中两个参数值都是常量，所以应该通过初始化方式而非赋值方式给类中成员传值。修改后的程序如例 16 所示。

例 16:

```
1   class A {
2   public:
3       A( const char *file, const char *path ) :
            myFile(file), myPath(path) {};
4   private:
5       string myFile;
6       string myPath;
7   };
```

例 17:

```
1   class  student
2   {
3        public:
4          student();
5        protected:
6          const  int  a;
7          const  int  &b;
8   };
```

```
9   student::student(int  i, int  j) // 错误
10  {
11    a=i;
12    b=j;
13  }
```

例 17 中第 9 行语句，类中常量成员初始化操作没有在初始化表里完成。

修改方法：将第 9 行语句修改为：

```
student::student(int i, int j):a(i),b(j){};
```

14．构造函数内变量初始化和声明的顺序不一致

C++语言中，构造函数有一种特殊的初始化方式——“初始化列表”，构造函数初始化列表以一个冒号开始，接着是以逗号分隔的数据成员列表，每个数据成员后面跟一个放在括号中的初始化值。如果类存在继承关系，派生类必须在其初始化列表里调用其基类的构造函数。

初始化时，编译器会记录成员变量在初始化列表中的初始化顺序，以保证析构函数能够正确释放成员变量。

类的构造函数按照成员在类中的声明顺序执行初始化操作，因此，如果类中构造函数的初始化列表和成员变量的声明顺序不一致，势必造成析构函数释放成员变量的顺序错误。

例 18：

```
1   class A
2   {
3   public:
4      A( int x ) : a( x ), b( a ) {};// 错误
5   private:
6      int b;
7      int a;
8   };
9   class B : public A
10  {
11  public:
12     B( int );
13  private:
14     int t;
15     float u;
16  };
17  B::B( int y ) : u( y ), A( 1 ), t( u ) { }; // 错误
```

例 18 中，类 A 的初始化列表（第 4 行语句）中成员变量的初始化顺序是先 a 成员，再 b 成员；

而类 A 中成员变量声明的顺序是先 b 成员，再 a 成员，两者初始化顺序不一致。第 17 行语句中也存在同样的错误。

修改方法：将第 4 行语句修改为：

```
A( int x ) : b( x ), a (b) {};
```

第 17 行语句修改为：

```
B::B( int y ) : t(y), A( 1 ), u( t ) { };
```

15. 构造函数内成员变量初始化顺序错误

类中构造函数按照成员在类中的声明顺序执行初始化操作，如果忽略了这一点，将导致初始化错误。

例 19：

```
1   class X
2   {
3   public:
4       X( int y );
5   private:
6       int i;
7       int j;
8    };
9   inline X::X( int y ) : j( y ), i( j ) {} ; // 错误
```

例 19 中，构造函数类 X 中成员变量的初始化顺序是先 i，再 j，第 9 行语句初始化列表中将成员变量 i 初始化为值 j ，而此时成员变量 j 还没有被初始化。

修改方法：将第 9 行语句修改为：

```
inline X::X( int y int x ) : j( y ), i( x ) {};
```

16. 外部变量重复定义

工程里同一外部变量在不同源文件中被重复定义时，不同的编译器对这种情况可能有不同的处理方式。为了避免外部变量被重复定义，应保证程序中同一外部变量只能定义一次。

例 20：

```
在 sourc1.c 中定义 int  a=1;
在 sourc2.c 中声明  extern  int  a; //a 是一个外部变量
在 sourc3.c 中定义 int  a=0;
```

当把 sourc1、sourc2 和 sourc3 源代码同时编译链接在一个工程里时，此时源文件 sourc2.c 中引用的外部变量 a 有两个值（在 sourc1 和 sourc3 中均有定义）。

17．同名外部变量类型不一致

当同一个外部变量在两个不同的源文件中被声明为不同的类型时，程序运行结果将不确定，甚至包括程序崩溃等。

例 21：

```
在sourc1.c 中声明 extern  int  a;
在sourc2.c 中定义 long  a;
```

例 21 中，在 sourc1.c 中声明引用 int 类型的外部变量 a，而在 sourc2.c 中定义变量 a 的类型为 long 型，同名变量 a 的类型不一致。

18．使用#define 定义常量

#define 是宏定义指令，预处理器会将源代码中的宏替换为相应的定义，然后编译，这将导致很多负面效应。例如，如果通过#define 定义一个常量，而编译器不识别该常量，在这种情况下，如果该常量用于表示一个表达式，该表达式的值是不相同的，其值依赖于该名字的范围。因此，要使用 const 或 enum，而不要使用 #define 定义常量。

例 22：

```
1    #define PI 3.1416
```

例 22 中使用#define 定义常量是错误的，应使用 const 或 enum 定义常量。

修改方法：将第 1 行语句修改为：const float PI = 3.1416;

19．编译单元中初始化顺序错误

对单个 C 或 CPP 文件进行编译时，预处理器首先递归处理头文件中的声明变量等内容，形成一个含有所有必要信息的单个源文件，这个源文件就是一个编译单元。

C++对定义于不同编译单元内的非局部静态对象的初始化顺序并无明确定义，对于两个不同编译单元中的静态对象 A、B，如果 A 的初始化依赖于 B，则要保证 A 初始化时 B 已经初始化。为了避免不同编译单元中出现初始化顺序错误，需要将每个非局部静态对象放到自己的专属函数内（该对象在此函数内被声明为 static），通过该函数返回一个引用指向它所包含的对象。

例 23：

```
1    #include <iostream>
2    using namespace std;
3    class FileSystem {
4    public:
5        size_t numDisks( ) const;
6    };
```

```
7   extern FileSystem tfs;
8   class Directory {
9   public:
10      Directory( );
11  };
12  Directory::Directory( ) {
13      size_t disks_tfs = tfs.numDisks( );
14  }
15  Directory tempDir( );
```

例 23 中第 7 行语句表示 FileSystem 类变量 tfs 可以在本模块或其他模块中使用，当变量 tfs 在其他模块中使用时，可能导致不同编译单元内非局部静态对象初始化顺序问题。为了避免该问题，将变量 tfs 放到自己的专属函数内。修改后的程序如例 24 所示。

例 24：

```
1   #include <iostream>
2   using namespace std;
3   class FileSystem {
4   public:
5       size_t numDisks( ) const;
6   };
7   FileSystem& tfs_one( ) {
8       static FileSystem fs;
9       return fs;
10  }
11  class Directory {
12  public:
13      Directory( );
14  };
15  Directory::Directory( ) {
16     size_t disks_one = tfs_one().numDisks( );
17  }
18  Directory tempDir( );
```

20. 声明已初始化的数组的大小

初始化数组时，数组大小也随之确定，因此没有必要声明已经初始化的数组的大小。

例 25：

```
1   #define SIZE 4
2   int tab1[SIZE] = {1,2,3,4};
```

例 25 中，第 2 行语句声明初始化的数组 tab1 的大小为 4，而当数组 tabl 在初始化时，其大小就已经确定为 4，不需要再次声明数组 tabl 的大小。

修改方法：直接初始化数组：int tab1[]={1,2,3,4};

21．声明数组时其大小未定义

当声明一个数组时，数组的大小应该被明确声明或者通过初始化隐含定义。

例 26：

```
1   extern int array2[ ];   //错误
2   extern int array1[ ];   //错误
```

例 26 中，没有指定数组 array1[]、array2[]的大小。修改后的程序如例 27 所示。

例 27：

```
1   int array2[ ] = { 0, 10, 15 };
2   extern int array1[ 10 ];
```

22．枚举元素初始化错误

使用 enum 枚举类型时，在枚举值表中应罗列出所有可能的值，这些值也称为枚举元素。枚举元素按照常量方式进行处理，枚举元素又被称为枚举常量。

枚举元素不一定都要初始化。如果不初始化枚举元素，此时枚举元素使用其默认值，即：第一个枚举元素的默认值为 0，其后的枚举元素值依次加 1。也可以对枚举元素进行明确的初始化，但只有两种初始化方式是安全的，一种是初始化全部枚举元素，另一种是只初始化第一个枚举元素。

例 28：

```
1    public enum Test1{x,y,z};
2    public enum Test2{x=1,y,z};
3    public enum Test3{x=2,y=3,z=4};
4    public enum Test4{x,y,z=1};
```

例 28 中，第 1 行语句使用默认值初始化枚举成员，此时，x、y、z 的值分别为 0、1、2；第 2 行语句初始化了第一个成员；第 3 行语句初始化了全部成员。这三种方式都是正确的，而第 4 行语句是不安全的。

23．枚举元素赋值方式错误

使用枚举类型时，如果需要为枚举元素定义其他值，则不能在程序中使用赋值语句对其赋值，只能在声明枚举元素时进行赋值。

例 29：

```
1    enum my_enum1
2     {
3       a,  b
4     };
5    a=1;
6    b=2;
```

例 29 中第 5 行语句，枚举元素 a 和 b 是常量，不能在程序中使用赋值语句进行赋值，只能在声明时对其赋值。

修改方法：在声明该枚举元素 a 和 b 时，对其进行赋值。

24. 同一区域内变量重名

例 30：

```
1    void CBaseProc (int ChooseNumber)
2    {
3       int DefaultValue;
4       bool GetOidMib = false;
5       …
6       if (GetOidMib)
7       {
8          int DefaultValue = 126;
9          DefaultValue = DefaultValue + 1;
10      }
11      DefaultValue = ChooseNumber;
12      …
13   }
```

C 语言允许在不同区域使用重命名的变量，例如在程序块和命名空间中。但在例 30 中，在程序块以及更深一级嵌套使用相同的变量，导致程序无法区分当前变量使用的是哪一个变量。

25. false 定义为非 0 值

在 C++语言中，false 是关键字，其值为 0。如果定义 false，应将其定义为 0。

例 31：

```
#define false 1
```

例 31 中将 false 定义为 1 是错误的，应将其定义为 0。

26. true 定义为非 1 值

在 C++语言中，true 是关键字，其值为 1。如果定义 true，应将其定义为 1。

例 32：

```
#define true 2
```

例 32 中将 true 定义为 2 是错误的，应将其定义为 1。

27. 使用幻数

幻数是没有变量名的数字，它无法表达其准确的含义，影响程序可读性和可维护性。因此，要用有意义的字符替代幻数。

例 33：

```
1   void boo(int);
2   int foo( int a[], int b ) {
3      int e;
4      int f = 0;
5      a[5] = 56;
       …
6   }
```

例 33 第 5 行语句中，数组元素 a[5]中的 5 为幻数。

修改方法：定义一个变量，如 t=5；使用 a[t]的方式访问数组。

28. 未采用 typedef 定义结构体

typedef 是 C 语言的关键字，其作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型和自定义的数据类型。使用 typedef 的目的一般有两个，一个是给变量一个易记且意义明确的新名字，另一个是简化一些比较复杂的类型声明。通过 typedef 定义结构体可以提高代码的可读性和可维护性。

例 34：

```
1   struct A {
2   int i;
3   };
4   struct B {
5   int j;
6    };
```

例 34 中，结构体 A 和 B 都不是由 typedef 定义的。

修改方法：通过 typedef 定义结构体 A 和 B。

29. 数组声明和 malloc()函数调用使用硬编码

硬编码是指将可变变量用一个固定值来代替，一旦使用硬编码，程序被编译后，更改此变量就非常困难了。因此，应避免使用硬编码。

例 35：

```
1   #include <stdlib.h>
2   void myFunction()
3   {
4      int array[10];
5      malloc(20);
6   }
```

例 35 中第 4 行语句定义 array 数组的大小采用的是硬编码，第 5 行语句中，malloc()函数的参数采用的也是硬编码。

30. 命名空间的嵌套层次超过两层

命名空间的嵌套建议不要超过两层，深嵌套的命名空间难以理解。

例 36：

```
1   namespace A
2   {
3       namespace B
4       {
5           namespace C
6           {
7               namespace D
8               {
9               }
10          }
11      }
12  }
```

修改后的程序如例 37 所示。

例 37：

```
1   namespace A
2   {
3       namespace B
4       {
```

```
5        namespace C
6        {
7        }
8        namespace D
9        {
10       }
11     }
12   }
```

31．外观相似的标识符混用

C/C++语言字符集中，有些符号外观很相似，容易混淆，因此，要避免同时使用以下标识符。

- 只有大小写混合差异；
- 存在或缺少下划线字符的差异；
- 字母 O 和数字 0；
- 字母 I 和数字 1；
- 字母 i 和数字 1；
- 字母 I 和数字 l (el)；
- 字母 l (el)和数字 1；
- 字母 S 和数字 5；
- 字母 Z 和数字 2；
- 字母 n 和数字 h；
- 字母 B 和数字 8；
- 字母序列 rn（n 紧跟在 r 之后）和数字 m。

例 38：

```
1   typedef int id1_uint32;
2   short id1_uint3Z;        // Z 和 2 混淆

3   void id2_foo(
4          int id4_paramS,
5          int id4_param5   // S 和 5 混淆
6          ){
7      int id3_abc;
8      int id3_ABC;  // 只有大小写差异
9   }

10  void id2_foo_(){
11     int id5_BO;
```

```
12      int id5_80;  // B 和 8，O 和 0 混淆
13      int id6_modern;
14      int id6_modem; // rn 和 m 混淆
15  }
```

32. 类、联合体、枚举型名称重名

在一段程序中，任何名称都不允许被重用，名称的重用会导致错误和混淆。

例 39：

```
1   struct stag { int a; };
2   void stag(void){}          // 错误
```

修改方法：更改函数 stag(void)名称，使其与结构体名称不一致。

33. 对象或类型的声明与定义未在一个模块中

变量应该在尽可能小的范围内声明，以改善程序的可读性。当一个变量在一个函数的头部声明却在程序的其他地方使用时，难以直接获取此变量的类型。此外，如果一个变量和声明与内部块的局部变量同名，那么此变量将会无意中被隐藏。

如果一个变量被声明在外层，即使不使用该变量，也将占用内存。此外，如果变量在声明时就被初始化，其执行效率比在使用时才赋值更加高效。

例 40：

```
1   void foo( ) {
2       int a;
3       {
4           a = 0;
5       }
6   }
```

修改方法：将 a=0; 同 a 的定义放到同一层。

34. 内部范围变量和外部范围变量使用相同的名称

内部范围的变量不能和外部范围的变量使用相同的名称，否则，外部范围的变量会隐藏内部范围的变量。

例 41：

```
1   int x;
2   void foo( ) {
3       int x;  // 错误
4       x = 3;
5   }
```

修改方法：避免在不同的范围内使用相同的变量名称。

35. 修改字符串内的字符

字符串文字不应该被修改。

例 42：

```
1   void moo(char * p)
2   {
3   }
4   void foo()
5   {
6       char *c1 = "Hello"; // 错误
7       char c2[] = "Hello"; //错误
8       char c3[6] = "Hello"; //错误
9       char *c12;
10       c12 = "Hello"; //错误
11       moo("Hello"); //错误
12  }
```

修改后的程序如例 43 所示。

例 43：

```
1   void moo(const char *p)
2   {
3   }
4   void foo()
5   {
6      const char *c1 = "Hello";
7      const char c2[] = "Hello";
8      const char c3[6] = "Hello";
9      const char *c12;
10       c12 = "Hello";
11       moo("Hello");
12  }
```

36. 存储类型修饰符用于变量或函数

存储类型修饰符包括 auto、static、register、extern、volatile、restrict。存储类型修饰符可以修改标识符的链接和对应对象的生存周期。标识符有链接，没有生命周期；对象有生存周期，没有链接。存储类型修饰符后只能是数据类型而非变量。

例 44：

```
int static i;
```

例 44 中存储类型修饰符 static 后是变量，而不是数据类型。

修改方法：将其修改为：static int i;

37. unsigned 类型常量缺少后缀标识

整型常量与许多因素有关，例如，常量数值大小、整型数的实际大小、是否有后缀、数值表达所采用的进制（十进制、八进制）。整型常量是产生混淆的潜在根源，例如，对于整型常量 40000，在 32 位机器中，其类型是 int，而在 16 位机器中，其类型是 long。同样，对于常量 0x8000，在 16 位机器中，其类型是 unsigned，而在 32 位机器中，其类型是 sign。由此可见，对于同一个数值，在不同的机器环境中，其类型会发生变化。从理论上讲，任何不带后缀的大于或等于 2^{15} 的十六进制数可能是 signed 或 unsigned 类型；任何不带后缀的大于或等于 2^{31} 的十进制数可能是 signed 或 unsigned 类型。为了避免混淆，建议在其后面增加后缀“U”。

例 45：

```
unsigned long var = 02;
```

修改方法：将其修改为：unsigned long var = 02U;

38. 直接使用 volatile 变量

volatile 变量不稳定，直接使用会导致程序出现不可预期的错误。

例 46：

```
1   void StaParPoll(void)
2   {
3     UINT_32 NMC_SNMP_CMD_LEN , MSG_H_LEN = 0, MsgLen = 1u;
4     volatile UINT_32 NewMsgLen = 1U;
5   …
6     NMC_SNMP_CMD_LEN = (NewMsgLen + MsgLen) /
    (NewMsgLen * MSG_H_LEN);
7   }
```

例 46 中，使用了 volatile 变量类型。

修改方法：增加中间变量，获取 volatile 变量的值后再进行运算。

39. 十六进制常量未使用大写字母表示

十六进制常量应该使用大写字母表示，这样可以提高程序的可读性和可维护性。

例 47：

```
int i = 0x3fff;
```

例 47 中，i 常量是十六进制常量，应该使用大写字母表示。

40．a(b)、a[b]和类型转换语句内部使用赋值语句

a(b)、a[b]和类型转换语句内部不要使用赋值语句，这样可以提高程序易读性和运行效率，避免软件缺陷。

例 48：

```
1   void foo( int i )
2   {
3     int *x;
4     x[ i = 0 ];
5     foo( i = 0 );
6     i = (int) (x = 0);
7   }
```

例 48 中，第 4、5、6 行语句中使用了赋值语句。

41．a(b)、a[b]内部使用递增/递减表达式

a(b)、a[b]内部不要使用递增或递减表达式，这样可以提高程序易读性和运行效率，避免软件缺陷。

例 49：

```
1   void foo( int i )
2   {
3   int *x;
4   x[ i++ ];
5   foo( i++ );
6   i = (int) (x++);
7   }
```

例 49 中，第 4、5、6 行语句中使用了递增/表达式。

42．拼接窄的和宽的字符串

C/C++语言中，用 wchar_t 表示宽字符串类型，宽字符串用“L”标识。宽字符串和窄字符串不得相互拼接。

例 50：

```
1   wchar_t array[] = "Hello" L"World";
```

例 50 中，“Hello”是窄字符串，而“World”是宽字符串，宽字符串和窄字符串不得相互拼接。修改后的程序如例 51 所示。

例 51：

```
1   wchar_t array[] = L"Hello" L"World";
```

43．位域类型错误

位域是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。位域结构中，类型说明符应该是布尔类型或者是显式的 unsigned 或 signed 类型。

位域定义形式如下：

```
struct 位域结构名{ 位域列表 };
```

位域列表的形式为：类型说明符 位域名：位域长度。

例 52：

```
1    struct S
2    {
3    char c : 2; // 错误
4    short f : 3; // 错误
5    bool b : 4;
6    };
```

例 52 中，第 3 行和第 4 行语句的位域类型说明符错误，应该是显式的 unsigned 或 signed 类型。修改方法：将第 3 行语句修改为：`unsigned char c : 2;`第 4 行语句修改为：`signed short f : 3;`

例 53：

```
1    enum Color{RED, BLUE, BLACK};
2    struct S
3    {
4    Color n : 2; // 错误
5    …
6    };
```

例 53 中，第 4 行语句的位域类型是枚举类型。位域通常用于由可组合出现的元素组成的列表，

而枚举常数通常用于由互相排斥的元素组成的列表。位域设计为通过按位“或”运算组合来生成未命名的值，而枚举常数则不然。因此，位域不能用于定义枚举类型 Color。

修改方法：将第 4 行语句修改为：`unsigned int n: 2;`

44. signed 整型命名的位域长度不足

位域结构中的位域名类型说明符可以定义为 unsigned，也可定义为 signed，但当位域长度为 1 时，因为一位的长度不可能具有符号，因此，其类型说明符会被认为是 unsigned 类型。如果要将其类型定义为 signed，其长度应该超过一位。

例 54：

```
1  struct  W{
2  signed int S01  : 1;
3  unsigned S: 1;
4  };
```

例 54 的结构体 W 中，signed 类型的 S01 变量长度定义为 1。

修改方法：将第 2 行语句修改为：signed int S01 : 2;

1.2 运算符使用问题

运算符用于指定对象进行的操作。C 语言中定义的运算符及其优先等级如表 1-1 所示。

表 1-1 运算符及其优先级

优先级	运算符	含义	运算符类型	结合性
15	()	圆括号		自左向右
	[]	下标运算符		自左向右
	->	指向结构体成员运算符		自左向右
	.	结构体成员运算符		自左向右
14	!	逻辑非运算符	单目	自右向左
	~	按位取反运算符	单目	自右向左
	++、--	自增、自减运算符	单目	自右向左
	+、-	正、负号运算符	单目	自右向左
	(type)	类型强制转换运算符	单目	自右向左
	*	指针运算符	单目	自右向左
	&	地址运算符	单目	自右向左
	sizeof	类型长度运算符	单目	自右向左

续表

优 先 级	运 算 符	含　义	运算符类型	结 合 性
13	*、/	乘、除运算符	双目	自左向右
	%	求余运算符	双目	自左向右
12	+、-	加、减运算符	双目	自左向右
11	<<、>>	左移、右移运算符	双目	自左向右
10	<、<=、>、>=	关系运算符	双目	自左向右
9	==、!=	等于、不等于关系运算符	双目	自左向右
8	&	按位与运算符	双目	自左向右
7	^	按位异或运算符	双目	自左向右
6	|	按位或运算符	双目	自左向右
5	&&	逻辑与运算符	双目	自左向右
4	||	逻辑或运算符	双目	自左向右
3	?:	条件运算符	三目	自右向左
2	=、+=、-=、*=、/=、%=、>>=、<<=、&=、^、|=	赋值运算符	双目	自右向左
1	,	逗号运算符		自左向右

这些运算符，有的形似，有的存在特定的优先级，有的结合性不同，因此，在使用过程中很容易混淆，导致程序错误。

在使用运算符的过程中，要特别注意以下事项。

（1）优先级最高者（优先级为 15）其实并不是真正意义上的运算符，而且是自左向右结合的。例如，a.b.c 的含义是（a.b）.c，而非 a.（b.c）。

（2）单目运算符优先级高于双目运算符，而且是自右向左结合的。例如，*p++的含义是*(p++)，即取指针 p 所指向的对象后将 p 加 1，而非(*p)++，即取指针 p 所指向的对象，然后将该对象加 1。

（3）双目运算符优先级从高至低依次为：算术运算符→移位运算符→关系运算符→逻辑运算符→赋值运算符。

（4）关系运算符<、<=、>、>=的优先级高于关系运算符==、!=。因此，如果要比较 a 与 b 的相对大小顺序和 c 与 d 的相对大小顺序是否一样，可以采用下面的写法：a<b==c<d。

（5）逻辑运算符的优先级各不相同。虽然同为逻辑运算符，但其优先级却各不相同。这是由历史原因造成的。C 语言是由 B 语言发展而来的，B 语言中的逻辑运算符大致相当于 C 语言中的&和|运算符。虽然这些运算符从定义上而言是按位操作的，但当其出现在条件语句的上下文时，B 语言的编译器会将它们作为 C 语言中的&&和||运算符来处理，而在 C 语言中，这几种用法是有区别的。为了兼容性需要，在 C 语言中没有重新定义这些运算符的优先级。

（6）赋值运算符的优先级仅高于逗号运算符，这一点需要特别注意。

（7）C 语言中只有&&、||、?:和逗号四个运算符存在规定的求值顺序，采取的是最小数目计算原则，即：对于逻辑与运算符&&和逻辑或运算符||，从左向右计算，当计算出一个表达式的值就能确

定整个表达式的值时，则无须计算后续表达式的值，否则，需要继续求后续表达式之值；对于条件运算符 a?b:c，首先求出 a 的值，根据 a 的值再求 b 或 c 之值；对于逗号运算符，首先对左侧操作数求值，然后“丢弃”该值，再求右侧操作数之值。其他运算符对其操作数求值的顺序未做定义，特别是，赋值运算符并不保证任何求值顺序。

例如，对于如下的程序：

```
int x=1,y=0,z=2;
!x&&y+1&&(z+=2);
```

由于!x 为假，因此，无须计算后面的表达式 y+1 和 z+=2 之值，就可以判定出整个表达式的值为假，此时 z 的值为 2 而非 4。

正因为逻辑与运算符&&和逻辑或运算符||存在上述规定的运算顺序，才能保证语句：

```
if (y!=0&&x/y>5)
a=b+c;
```

只有在 y 非 0 时才计算 x/y 之值。

编程时如果不注意运算符的优先级，程序的运行结果将大相径庭。特别是涉及赋值操作时，由于赋值运算符的级别较低，经常出现优先级顺序问题。

对于运算符之间的优先级，记住如下关键两点：

① 所有逻辑运算符的优先级都低于任何一个关系运算符；

② 移位运算符的优先级比算术运算符低，但比关系运算符高。

45. 运算符优先级使用错误

例 55：

```
1  while (c=getc(in)!=EOF)
2       putc(c,out);
```

该语句的本意是复制一个文件到另一个文件，即首先获取函数 getc(in)的值，然后与 EOF 比较是否到达文件尾部以决定是否终止循环。然而，由于赋值运算符的优先级低于!=运算符，因此，c 的值实际上是函数 getc(in)的返回值与 EOF 比较的结果。此处函数 getc(in)的返回值只是一个临时变量，在与 EOF 比较后就被丢弃了，最后得到的文件副本中只包括了一组二进制数为 1 的字节流。

修改方法：将第 1 行语句修改为：While ((c=getc(in))!=EOF)

例 56：

```
1    #include <stdlib.h>
2    main()
3    {
4      int a, b, c;
```

```
5     scanf("%d",&a);
6     scanf("%d",&b);
7     c = ( a & b!=0 ); // 错误
8   }
```

例 56 中，第 7 行语句的本意是“c = ((a & b)!=0)”，即 a 和 b 先做按位与运算，运算后的值再和 0 比较。但由于运算符“!=”的优先级高于“&”，所以条件表达式变为“c=(a&(b!=0))”，即 b 和 0 比较的结果，再和 a 值按位与运算，导致错误的结果。

46. 括号使用不当

C 语言中的运算符优先级很复杂，稍不留心就会误用，使用括号是一个很好的解决方案，但括号使用过多会使程序显得杂乱，降低可读性。下面给出几种不需要使用括号的场合。

① 赋值运算符右边的操作数不需要使用括号，除非右边的操作数中又包含了赋值表达式。例如，没有必要将 x=a+b 写成 x=(a+b);

② 单目运算符中的操作数不需要使用括号。例如，没有必要将 x=a*-1 写成 x=a*(-1);

③ 双目和三目运算符中的操作数应该是强制转换表达式，除非该表达式中所有的运算符是相同的。例如，没有必要将 x = f (a + b, c)写成 x = f ((a + b), c)；没有必要将 x = (uint16_t) a + b 写成 x = ((uint16_t) a) +b。

47. 括号导致结合律不成立

众所周知，即使所有的运算符都是相同的，也可以使用括号控制运算的次序。某些运算符，如加法和乘法，在代数学上是遵循结合律的，而在 C 语言中未必如此。类似地，包含混合类型的整数运算，由于整数提升的原因可能产生不同的结果。

例 57：

```
1   uint16_t a = 10;
2   uint16_t b = 65535;
3   uint32_t c = 0;
4   uint32_t d;
5   d = (a + b) + c;
6   d = a + (b + c);
```

从代数意义上而言，最后两个表达式的结果是相同的，但在本例中，对于 d = (a + b) + c，由于 a、b 均是 16 位整数，a+b 之值因超出了 16 位整数范围而被截断为 9，因此 d = (a + b) + c 之值为 9；对于 d = a + (b + c)，b+c 之值为 65535，整个表达式之值为 65545，表明加法结合律不成立了。

另外，浮点数的四舍五入也会使加法结合律不成立。

例 58：

```
1   f1 = F3 + (F4 + F5);
2   f2 = (F3 + F4 ) + F5;
```

例 58 中，假设 f1 和 f2 是浮点变量，F3、F4 和 F5 是浮点类型的表达式，对于这两个表达式，不能保证赋给 f1 和 f2 的值是相同的，因为紧接加法后的另一个操作数的值取决于该浮点数四舍五入之值。

48. 未指定运算顺序导致运算结果不确定

除了函数调用运算符()、&&、||、?:和逗号等少数几个运算符外，子表达式的运算顺序是未指定的，并且这个顺序是不能通过加括号的方法加以改变的。因此，要确保表达式的值在任何运算顺序下都是相同的。下面给出几类由于运算顺序未指定导致运算结果不确定的实例。

（1）自增或自减运算符所引起的运算结果不确定

例 59：

```
1   x=b[i]+i++;
```

例 59 中，由于没有指定 b[i]和 i++的计算顺序，因此，根据上述两个子表达式计算顺序的不同，整个语句产生的结果也不同。

为了避免这个问题，可以把其中的自增运算作为一条独立的语句，如例 60 所示。

例 60：

```
1   x=b[i]+i;
2   i++;
```

（2）函数中的参数运算顺序未指定引起的结果不确定

考虑语句：`x=func(i++,i);`

该语句的结果因函数中两个参数的运算顺序不同而不同。

（3）函数指针

如果通过函数指针来调用函数，执行结果因指定函数标识符和函数参数运算顺序的不同而不同。例如：

```
p->task_start_fn(p++);
```

（4）函数调用

函数在被调用时可以具有附加的作用，例如，修改某些全局数据。为了避免对运算顺序的依赖，可以在该语句使用之前调用函数，并使用一个临时变量存储语句的运算结果。例如，将 x=f(a)+g(a)改写为：

```
x=f(a);
```

```
x+=g(a);
```

再如，对于语句：push(pop()-pop())；该语句从堆栈中取出两个值，从第一个值中减去第二个值，再把结果放回栈中。该语句的结果取决于先计算哪一个 pop()函数。

（5）嵌套的赋值语句

表达式中嵌套的赋值语句可以产生附加的副作用，因此，不要在表达式中嵌套赋值语句。例如，避免使用下面的语句：

```
x=y=y=z/4;
x=y=y++;
```

（6）访问 volatile 类型

volatile 类型限定符用来表示其值可以独立于程序的运行而自由更改的对象（例如输入寄存器）。对带有 volatile 限定类型对象的访问可能改变它的值。

在计算表达式的值时，经常需要访问作为表达式组成部分的 volatile 数据，此时，运算结果将依赖于运算顺序。因此，建议尽可能将对 volatile 的访问放在简单的赋值语句中。例如：

```
volatile uint16_t v;
/* … */
x = v;
```

（7）子表达式运算次数

例 61：

```
1  #define MAX (a, b) ( ( (a) > (b) ) ? (a) : (b) )
   /* … */
2  z = MAX (i++, j);
```

例 61 中，当 a > b 时，第一个参数计算了两次，当 a <= b 时，只计算了一次。这样，宏调用根据 i 和 j 的值，i 的值增加了一次或两次。

49. 布尔型变量被赋值

例 62：

```
1   void  f(bool flag)
2    {
3     int  a;
4     bool ok;
5     ok=true;
6     if(flag=ok) // flag是一个bool型变量
7       a++;
8    }
```

例 62 中的第 6 行条件判断语句，存在布尔型变量 flag 被赋值的错误。

50. ==运算符与=赋值符混用

由 Algol 语言派生而来的大多数程序设计语言，例如 Pascal、Ada，使用:=作为赋值运算符，使用=作为比较运算符。而 C 语言则不同，它使用=作为赋值运算符，==作为比较运算符。由于赋值运算使用的频率高于比较运算，因此，C 语言将字符数较少的符号=作为赋值操作。此外，在 C 语言中，赋值符号被作为一种运算符对待，使用=作为赋值操作，可以方便地书写类似 a=b=c 这样重复的赋值操作，并且，赋值操作还可以被嵌入到更复杂的表达式中。

这种使用上的便利性以及习惯于将=作为比较运算符的程序员，经常无意中误用这两种运算符。

例 63：

```
1   void foo(int i, int j)
2   {
3     int t;
4     if(i=j)      // 误用
5         t++;
6   }
```

例 63 中，第 4 行语句本意是比较 i 和 j 的值是否相等，而“if(i=j)”执行的操作是把 j 的值赋给 i，再判断 i 的值是否为 0。

例 64：

```
1    class  A
2    {
3      int t;
4      int q();
5    };
6    main()
7    {
8      int num;
9      A  a;
10     if (i=a.q())    // 误用
11     {
12         num++;
13     }
14   }
```

例 64 中的第 10 行条件判断语句，＝＝判断符被误用为＝赋值符，函数调用语句出现在 if 条件

判断语句＝赋值符的右边。

例 65：

```
1    f()
2    {return  1;}
3    main()
4    {
5      int  a;
6      int array[3]={1,12,13};
7      if ((a==array[2])< 0)     // 误用
8        f();
9      else return -1;
10   }
```

例 65 中，第 7 行语句本意是将 array[2]赋值给 a 后，再将 a 的值和 0 作比较，如果条件为真，则执行函数 f()；当使用==后，“(a= =array[2])”的值只能是 0 或 1，不会小于 0，函数 f()永远不会执行。

51．按位运算符&、|和逻辑运算符&&、||误用

例 66：

```
1    f()
2    {return  1;}
3    main()
4    {
5     int  a=8;
6     int  b=1;
7     if(a&b)  f();
8    }
```

例 66 中，第 7 行条件判断语句本意是 a 与 b 做逻辑与运算&&，但被误用为按位与运算符&后，“8&1”结果为 0，函数 f()将不会执行。

52．单引号‘和双引号“字符误用

程序中单引号内的字符，例如'a'代表一个字符，字符在编译器中有其对应字符集中的序列值，也可以说单引号字符代表一个整数。因此，对于采用 ASCII 字符集的编译器而言，'a'的含义与 0141（八进制数）、97（十进制数）是相同的；而用双引号引起的字符串代表的却是一个指向无名数组起始字符的指针，该数组中包含与其对应的字符外加一个二进制值为零的字符\0。

另外，由于整型数（一般为 16 位或 32 位）的存储空间可以容纳多个字符（一般为 8 位），因此，某些 C 编译器允许一个字符常量中包含多个字符，例如，编译器无法区分'yes'和"yes"，但这两者的

含义是不同的。"yes"的含义为依次包含'y'、'e'、's'以及空字符\0 的 4 个连续内存单元的首地址，而'yes'的含义并没有准确地进行定义。大多数编译器将其理解为一个按照特定编译器实现方式组合而成的由'y'、'e'、's'所代表的整数值。

例 67：

```
1    main ()
2    {
3      char *p;
4      p='a';
5    }
6
```

例 67 第 4 行语句中，'a' 代表一个整数，而"a"代表字符串。由于'a'和"a"的误用，导致赋值符号两边的数据类型不匹配。

53. 赋值符错用其他运算符

例 68：

```
1    int  main(int k)
2    {
3     int j=1;
4     k=0;
5     if(j)
6     {
7       j>k;
8       return j;
9     }
10    else
11    {
          k++;
12      return k;
  }
13   }
```

例 68 中，第 7 行语句将赋值表达式运算符“=”错用为运算符“>”，使得该语句没有任何意义。

54. 常量后缀用小写字符

常量后缀应使用大写字母，而不是小写字母，以提高程序的可读性。

例 69：

```
1    void foo(long param = 64l)
2    {
3        const long a = 64l;
4    }
```

修改方法：将 64l 改写成大写的 64L。

55. 自增/自减（++/--）运算符和变量间有空格

程序中无论是自增或自减运算符，运算符和变量之间不能有空格。

例 70：

```
1    main()
2    {
3        int i=6;
4        int j=7;
5        -- i;
6        j ++;
7     }
```

例 70 中，自减和自增运算符与变量 i、j 之间都有空格，为了增加程序的可读性，应该删除自减和自增运算符与变量 i、j 间的空格。

56. 错误使用自增/自减（++/--）运算符

自增/自减运算符（++/--）表达简练，因此在 C/C++编程中经常用到，但就是这个几乎在每个程序中都会用到的运算符，如果不注意细节，就会产生错误。

例 71 中的程序遍历数组 arr[10]的每个元素，如果元素的值不等于 2，则执行自增/自减操作，并打印自增/自减后的值。

例 71：

```
1     #include<stdio.h>
2     int main()
3    {
4         int arr[10] = {2,3,1,2,3,3,1,2,2,3};
5         int tmp = 0;
6
7         for(int i = 0; i<10; i++)
8         {
9              if(arr[i] < 2)
```

```
10          {
11              tmp = arr[i]++;
12              printf("arr[%d] = %d\n", i, tmp);
13          }
14          else
15              if(arr[i] > 2)
16              {
17                  tmp = arr[i]--;
18                  printf("arr[%d] = %d\n", i, tmp);
19              }
20      }
21   return 0;
22  }
```

按照程序设计意图，每次打印语句中的元素值应该都是整数 2，但在例 71 中，第 11 行语句和第 17 行语句赋给 tmp 的值并非是数组元素自增/自减后的值，而是数组元素本身的值。因此，程序实际执行打印出来的值是 arr[10]数组元素本身的值，并不都是整数 2。

对于 a++和++a，单独使用时，其结果是一样的，都相当于 a=a+1，但 tmp= a++和 tmp= ++a 得到的结果是不一样的。

tmp= a++，相当于 tmp = a; a=a+1。

tmp= ++a，相当于 a=a+1; tmp = a。

前者先赋值再自增，后者先自增再赋值，两者所包含的赋值操作顺序不同，结果也就不相同了，同时两者的执行效率也不相同。对于 a++而言，需要用一个临时变量来保存 a 的值，然后执行自增操作；而对于++a 来说，整个表达式的值就是 a 的值，无须进行中间值的复制操作，因此其执行效率要高一些。

在需要使用自增/自减运算符时，如无特殊情况，建议使用++a(--a)格式的语句。

57. 三目运算符中使用混合类型

不要在三目运算符中使用混合数据类型，这样可以避免不同类型的数据间隐式转换时的数据损失。

例 72：

```
1   void foo( ) {
2      int x;
3      int y;
4      x = ((y > 5) ? 1.1 : 0.2);     // 错误
5           }
```

例 72 中第 4 行语句，三目运算符中出现整数和浮点数两种数据类型。

58. 三目运算符执行顺序错误

在计算由三目运算符组成的条件表达式 b?x:y 时，其计算顺序是先计算条件 b，然后进行判断。如果 b 的值为 true，则计算 x 的值，运算结果为 x 的值；否则计算 y 的值，运算结果为 y 的值。运算符的执行方向依次为从右向左。三目运算符“?:”容易造成误解，应谨慎使用。

例 73：

```
1    int main( )
2    {
3        return (1 ? 1 : 0);
4    }
```

例 73 第 3 行语句中，使用“?:”三目运算符容易造成误解。

修改方法：将第 3 行语句修改为：

`if(1) return 1; else return 0;`。

59. 误用字符串结束符

C 语言中没有专门的字符类型，通常用字符数组来存放字符串，以'\0'作为字符串的结束符。如果结束符'\0'被误用为"\0"时，可能导致程序崩溃。

例 74：

```
1    main()
2    {
3      char   a[10];
4      for(i=0;i<9;i++)
5      a[i]=i;
6      a[9]= "\0" // 错误
7        …
8      f(a);
9    }
```

例 74 中，第 6 行语句欲把"\0"作为结束符赋值给字符型数组 a 的最后一个元素。在这种情况下，C++编译器不会把字符"\0"赋值给某个数组元素，但 C 编译器则不然。这种错误在编程时不易察觉，需要注意。

60. 直接比较浮点数是否相等

浮点数在内存中的存储机制和整数不同，有舍入误差，在计算机中用以表示某个近似实数，但无法精确。具体来说，这个实数由一个整数或定点数乘以某个基数（计算机中通常是 2）的整数次幂得到，这种表示方法类似于基数为 10 的科学记数法。因为浮点数是非精确存储的，所以不能用关系

运算符中的“==”、“！=”、“>=”和“<=”比较浮点数。

例 75：

```
1   int  main()
2   {
3     float  a;
4     float  b;
5       …
6     if(a==b)   // 错误
7    return 1;
8      …
9   }
```

例 75 的第 6 行语句中，直接使用关系运算符==比较两个浮点数是否相等。当需要判断两个浮点数是否相等时，应该先设定一个精度，而后比较这两个浮点数差的绝对值是否在这个精度范围内。

修改方法：将第 6 行语句修改为：if(fabs(a-b) < 1.0E-10)

61．枚举类型运算符使用不当

除了[ ]、 =、 ==、 !=、<、<=、 >、 >= 和单目运算符&以外，枚举类型的表达式中不得使用其他运算符。

例 76：

```
1   enum { COLOUR_0, COLOUR_1, COLOUR_2, COLOUR_3, COLOUR_COUNT } colour;
2   void foo()
3   {
4     if ( ( COLOUR_0 + COLOUR_1 ) == colour ){} // 错误
5   }
```

修改后的程序如例 77 所示。

例 77：

```
1   enum { COLOUR_0, COLOUR_1, COLOUR_2, COLOUR_3, COLOUR_COUNT } colour;
2   void foo()
3   {
4     if ( ( COLOUR_0 < colour) && (COLOUR_3 > colour ) ){}
5   }
```

62．普通的 char 类型和 wchar_t 型的运算符使用不当

除了=、==、!= 和单目运算符&之外，带有 char 和 wchar_t（简单的）类型的表达式不能被当作内置的操作数使用。

例 78：

```
1   void foo()
2   {
3       char ch = 't';
4       if ( ( ch >= 'a' ) && ( ch <= 'z' ) ) // 错误
5       {
6       …
7       }
8   }
```

修改后的程序如例 79 所示。

例 79：

```
1   void foo()
2   {
3       char ch = 't';
4       if ( ch == 't' )
5       {
6       …
7       }
8   }
```

63. ~ 和 <<运算符使用不当

当用~和 <<运算符对小整数类型（如无符号字符或无符号短整型）进行操作时，运算结果的高位是不可预测的。

例 80：

```
1    typedef unsigned char uint8_t;
2    typedef unsigned short uint16_t;
3    void foo()
4    {
5        uint8_t port = 0x5aU;
6        uint8_t result_8;
7        uint16_t result_16;
8        uint16_t mode;

9        result_8=(~port)>>4;        // 错误
10       result_16=((port<<4)&mode)>>6;    // 错误
     }
```

修改后的程序如例 81 所示。

例 81：

```
1   typedef unsigned char uint8_t;
2   typedef unsigned short uint16_t;

3   void foo()
4   {
5       uint8_t port = 0x5aU;
6       uint8_t result_8;
7       uint16_t result_16;
8       uint16_t mode;

9       result_8=((uint8_t)(~port))>>4;
10      result_16=((uint16_t)(~(uint16_t)port))>>4 ;
11      result_16=((uint16_t)((uint16_t)port<<4) & mode) >> 6;
12  }
```

64. signed char unsigned char 类型数值的存储和使用错误

signed char 和 unsigned char 类型只能用来存储和使用数字型的值。

例 82：

```
1   signed char a = 'A';          // 错误
2   void foo( ) {
3       unsigned char a = 'B';    //错误

4       if (a == 'C')             //错误
5       {
6       }
7       if (a < 'D')              //错误
8       {
9       }
10          }
```

修改后的程序如例 83 所示。

例 83：

```
1   signed char a = 65;
2   void foo( ) {
```

```
3     unsigned char a = 66;
4         if (a == 67)
5         {
6         }
7         if (a < 68)
8         {
9         }
10               }
```

65. 位、比较、逻辑和逗号运算符声明为非 const

使用位运算符、比较运算符、逻辑运算符和逗号运算符时，因其不会改变调用对象的成员，所以应将其声明为 const。

例 84:

```
1   class A {
2   public:
3   A& operator^( int x );      //错误
4   A& operator==( int x );     //错误
5   A& operator&&( int x );     //错误
6   A& operator,( int x );      //错误
7           };
```

修改方法：将第 3、4、5、6 行语句分别修改为：

```
A& operator^( int x )  const;
A& operator==(int x) const;
A& operator&&(int x) const;
A& operator,( int x ) const;
```

66. !、&&和||运算符的操作数为非布尔类型数据

!、&&和||这些逻辑运算符应使用布尔类型作为操作数，这样可以有效防止逻辑运算符和位运算符（&、|和 ~）的混淆。

例 85:

```
1   void foo(int a, int b, int c, int d, int *ptr)
2   {
3       if ( 1 && ( c < d ) ) {}            //错误
4       if ( !ptr ) {}                      //错误
5       if ( a || ( c + d ) ) {}            //错误
            ...
6   }
```

例 85 第 3 行语句中，&&逻辑运算符含有非布尔类型的操作数 1；第 4 行语句中的!逻辑运算符含有非布尔类型的操作数 ptr；第 5 行语句中的||逻辑运算符含有非布尔类型的操作数 c + d。

修改方法：将第 3、4、5 行语句分别修改为：

```
if ( ( a < b ) && ( c < d ) ){}
if ( ( a == b ) || ( c != d ) ){}
if ( !false ) {}
```

67．对负数进行右移操作

右移运算符>>右移时，移出的低位直接舍弃。因此，在使用右移运算符右移时，需要注意符号位问题。

右移运算符包括算术右移和逻辑右移两种方式，当进行算术右移操作时，对于左边的最高符号位，若原符号位为 0（表示正数），则移入 0，原符号位为 1（表示负数），则移入 1；当进行逻辑右移时，无论最高符号位是 0 还是 1，移入的都是 0。编译器不同，其使用的右移方式可能不同。无论编译器执行的是算术右移还是逻辑右移，对无符号数和正数的右移均无影响，但对负数的右移影响比较大。

例 86：

```
1    main()
2    {
3      signed int  t=-2;
4      t=t>>3;
5      printf("%d\n\n\n",t);
6    }
```

例 86 中，如果是算术右移，输出的将是-1；如果是逻辑右移，输出的将是一个大整数。在编程时，可以通过先对负数取绝对值，然后右移，最后取相反数来实现负数的右移操作，这样可以避免因编译器不同对负数右移产生的影响。

68．sizeof 运算符和 strlen()函数混淆

sizeof(var)是单目运算符，返回变量 var 所占有的字节数。sizeof 运算符一般有两种用法：sizeof(object)和 sizeof(typename)。sizeof(object)是对对象使用 sizeof；sizeof(typename) 是对类型使用 sizeof。

strlen(str)是函数，用来返回以'\0'为结束符的字符串 str 的长度。

例 87：

```
1    #include<string.h>
2    int main()
3    {
4     char str[50]="abcdef\0ghigklmn";
```

```
5    int  i, j;
6    i= strlen(str);  //结果是 6
7    j= sizeof(str);  //结果是 50
8    if(i=j)
9    return  1;
10  }
```

例 87 中，第 6 行语句 strlen()函数返回的是字符串 str 在结束符'\0'之前的字符数，结果是 6；而 sizeof 单目运算符返回字符串 str 的长度，结果是 50，因此第 8 行语句的 if 判断条件永远为假，第 9 行语句永远不会被执行，从而导致程序无返回值，和程序声明不一致。

例 88:

```
1   #include<string.h>
2   int main()
3   {
4    int i, j;
5    int n[4] = {1,2,3,4};
6    i=sizeof(n);     //结果是 16（一个整数占用 4 字节）
7    int n1= 1234;
8    j=sizeof(n1);    //结果是 4
9    if(i=j)
10      return 1;
11   }
```

例 88 中，第 6 行语句 sizeof 返回的是整数数组 n 占用的字节数，n 是有 4 个整数元素的数组，其占用的字节数为“4*sizeof(整数)”，即 16；而第 8 行语句 sizeof 返回的是整数 n1 占用的字节数，其结果为 4，因此第 9 行语句中 if 判断条件永远为假，第 10 行语句永远不会被执行，从而导致程序无返回值，和程序声明不一致。

69. C　x 和 C　x()混淆

“C　x”表示声明一个局部的 C 类对象；“C　x()”表示声明一个返回 C 对象的函数 x()。

例 89:

```
1   class  c{…};
2   class  d{…};
3   void main()
4   {
5    d  t(){…};
6    c x;
7    x=t(); //错误
```

```
8     …
9    }
```

例 89 中，第 5 行语句声明了一个返回类 d 对象的 t()函数，第 6 行语句声明了一个类 c 的对象，第 7 行语句对这两个类的对象进行赋值操作，导致程序错误。

70. 前缀和后缀形式的自增/自减运算符实现不一致

不论是自增（increment）或自减（decrement）的前缀还是后缀，都只有一个参数，其中，后缀形式的参数类型为 int 类型，前缀形式的参数为空。前缀形式是“先操作，后取回”，后缀形式是“先取回，再操作”。前缀和后缀形式的自增/自减操作返回的类型是不同的，前缀形式返回一个引用，后缀形式返回一个 const 类型。

例 90：

```
1    class A {
2    public:
3     explicit A( int i = 0 ) : _i( i ) {}
4     ~A( ) {}
5     A operator++( )     // 前缀形式运算符
6       {
7         ++_i;
8         return *this;
9       }
10    A operator++( int )    // 后缀形式运算符
11      {
12        A temp = *this;
13        ++(*this);
14        return temp;
15      }
16    private:
17    int _i;
18        };
```

例 90 中，第 5 行语句使用的是前缀形式运算符，应该返回一个引用类型，而在本例中返回的是指针而非引用类型；第 10 行语句使用的是后缀形式运算符，应该返回一个 const 类型，而在本例中返回的是非 const 类型。

修改方法：将第 5 行语句修改为：`A& operator++( )` 第 10 行语句修改为：`const A operator++( int )`

71. 重载 &&、||和逗号运算符

在处理 C/C++语言内建的 &&、|| 和逗号运算符时，编译器从左至右对操作数进行求值，如果左操作数为假，那么右操作数将不会被求值。当这些运算符被重载时，编译器将这些运算符视为普通函数，而不是运算符，而普通函数的参数总是会被全部求值的，而且其求值顺序是未定义的，因此，将造成不可预知的运行结果。

例 91：

```
1    class A {
2    public:
3         A( int i ) : _i( i ) {}
4        ~A( );
5        int value( ) { return _i; }
6    private:
7        int _i;
8    };
9    int operator&&( A& lhs, A& rhs ) {
10    return lhs.value( ) && rhs.value( );
11   }
```

例 91 中第 9 行语句重载了运算符&&。

72. 赋值运算符返回的引用类型错误

赋值运算符通常是把已存在的对象指定给其他相同类型的对象，因为赋值运算符会改变左值，所以赋值运算符重载时要返回引用。编程时，必须保证赋值运算符的返回类型是一个指向自身类型的非 const 引用或返回一个指向*this 的引用。

例 92：

```
1    class A {
2    public:
3        A( ) { }
4        void operator=( A& a )
5        {
6            return;
7        }
8    };
9    class C {
10   public:
11       C( ) { }
```

```
12      C operator=( C& c )
13      {
14        C *cp;
15        return *cp;
16      }
17    };
```

例 92 第 4 行和第 12 行语句中，赋值运算符返回的数据类型不是指向其自身类型的引用或一个指向*this 的引用。

修改方法：将第 4 行和第 12 行语句中的赋值运算符返回的数据类型修改为一个指向*this 的引用。修改后的程序如例 93 所示。

例 93：

```
1   class A {
2   public:
3     A( ) { }
4     A& operator=( A& a ) {
5     return *this;
6     }
7   };
8   class C {
9   public:
10    C( ) { }
11    C& operator=( C& c ) {
12      return *this;
13    }
14  };
```

73. 除零错误

例 94：

```
1  static int zeroMethod()
2  {
3     return 0;
4  }
5  static void assignmentRemainderOfMethodResult()
6  {
7     int a = 5;
8     a %= zeroMethod();  //错误
9  }
```

例 94 第 8 行语句中，除数 zeroMethod()函数返回的值为 0，导致程序错误。

74. 分号使用不当

程序中多一个或少一个分号，这个分号也许会作为不会产生任何实际效果的空语句，也许编译器会因此而产生一条警告信息，根据警告信息，很容易删除多余的分号。上述情况对程序运行不会产生太大影响，但在某些情况下，特别是紧跟 if 或 while 子句的一条语句后面多了一个分号时，将导致紧跟在 if 或 while 子句后面的语句成为一条孤立的语句，程序将背离原有结果。

例 95：

```
1   #include <stdlib.h>
2   main()
3   {
4    int  a, c ;
5    int  array[3]={10, 06, 12};
6    scanf("%d",&a);
7    if(a < 0);   //错误
8      c=a;
9   }
```

例 95 中，第 7 行语句多了一个分号，因此紧跟在 if 语句之后的语句“c=a;”就是一条孤立的语句，和条件判断部分完全没有任何关系了。

例 96：

```
1   #include <stdlib.h>
2   void  f()
3   {
4    int  a;
5    float  array[3];
6    scanf("%d",&a);
7     …
8    if  (a < 0)
9      return    //错误
10     array[2]=a;
11  }
```

例 96 中，第 9 行语句少了一个分号，程序会把“array[2]=a;”语句中分号之前的内容作为 return 的返回内容，而该程序中函数 f()不应返回任何内容。

例 97：

```
1   void foo(int param)
```

```
2  {
3     if (param);
4     for(;;)  ;
5     while(param);
6    {
7    }
8  }
```

例 97 中第 3 行 if 语句、第 4 行 for 语句和第 5 行 while 语句中，分号使用不当。

修改后的程序如例 98 所示。

例 98：

```
1     void foo(int param)
2  {
3     if (param)
4    {
5    }
6     for(;;)
7     ;
8     while(param)
9     {
10    }
11    }
```

有时程序中会误用中文分号作为程序中所使用的英文分号，在这种情况下，编译器会对这个错误的分号产生一条告警信息。

75. 逗号使用不当

例 99：

```
1  void FpParser(USHORT *pUnit,USHORT carrierNumber,USHORT frameLen)
2   {
3       …
4       if(snmp_var.syntax==SNMP_SYNTAX_OCTETS)
5       {
6           memcpy(cMsg,snmp_var.value.string.ptr,snmp_var.value.string.len),
7           FpParser((USHORT*)cMsg,g_config.m_CarrNum,g_config m_FramLen); }
8       …
9   }
```

例 99 中第 6 行语句，函数之间用逗号分隔，在这种情况下，只返回最右边函数的值。

修改方法：分开写，都用分号。

1.3 函数问题

C/C++语言程序是由变量或者函数这些外部对象构成的。函数是一个能完成一定功能的可执行代码段，一般使用 return 语句由被调函数向调用函数返回值。如果函数的参数、返回值等处理错误，函数功能将不能正确实现。

76. 无参函数调用错误

函数调用时，即使是调用无参函数，也应该包括参数列表。

例 100：

```
1   #include <stdlib.h>
2   F();  //无参函数
3   main()
4   {
5     int  t;
6     scanf("%d",&t);
7     if (t>=0)
8     F;
9    }
```

例 100 中，第 7、8 行语句本意是，如果 if 条件成立，调用无参函数 F()，而第 8 行语句函数名 F 后未添加参数列表，所以该语句不做任何操作。

77. 函数返回值类型与声明的返回值类型不一致

函数返回值类型不一致，包括函数分支返回值类型不一致和由于函数调用造成返回值不一致两种情况。其中，函数分支返回值类型不一致是指函数如果有返回值，函数的每个分支都应该有返回值；如果没有返回值，函数的每个分支都应该无返回值。函数分支返回值类型不一致见例 101 和 102。函数调用造成返回值不一致，是指程序中主函数规定了返回值类型，而程序中调用函数返回值类型和主函数的返回值类型不一致，导致程序中返回值类型不一致，见例 103。

例 101：

```
1     int  f(int t)
2     {
3      return  t;
4    }
5    void  main()
6      {
```

```
7       int  i;
8       i=4;
9       f(4);     //错误
10    }
```

例 101 中，main()函数没有返回值，但在第 9 行语句调用函数 f()时产生了返回值，造成程序中返回值类型不一致。

例 102：

```
1    void  f(int t)
2    {
3    }
4     int  main()
5     {
6       int  i;
7       i=4;
8       f(4);     //错误
9     }
```

例 102 中，main()函数有 int 类型的返回值，但其调用了没有返回值的函数 f()，导致程序中返回值类型不一致。

例 103：

```
1    #include  <stdio.h>
2     char  g(int t)
3     {
4      char  s;
5      scanf("%c",&s);
6      return  s;
7     }
8    int  main()
9     {
10      int  i;
11      i=4;
12      g(4);     //错误
13     }
```

例 103 中，main()函数显式返回 int 类型值，但在 main()函数内部却调用了显式返回 char 类型值的 g 函数，导致程序中返回值类型不一致。

78. 分支返回值类型不一致

程序中的某个函数含有一组分支语句，其中一种情况是某些分支不提供返回值，而另一些分支却提供返回值；另一种情况是不同分支提供的返回值类型不一致。这种缺陷可能引起潜在的致命错误。

例 104：

```
1   #include  <math.h>
2   #include  <stdio.h>
3   f (int i)
4   {
5    return i;
6   }
7   main ()
8   {
9     int j;
10    scanf("%d",&j);
11    if(j)
12    {
13     …
14      f(j);
15    }
16   }
```

例 104 中，第 11 行 if 语句条件成立时，main()函数将返回一个 int 类型的值；当 if 语句条件不成立时，main()函数将不返回任何值，导致程序中分支返回值不一致。

79. 无返回值的函数有返回值

例 105：

```
1    void t()
2     {
3      int  t;
4     return  t;
5     }
```

例 105 中，第 1 行语句 void 表明 t()是一个没有返回值的函数，但第 4 行语句返回的是 int 类型的值。

80. 有返回值的函数调用无返回值函数

程序中规定了返回值，但程序中调用的函数却没有返回值，例如，一个有返回值的主程序调用

无返回值的函数。

例 106：

```
1   void  g(int t)
2   {
3   }
4     int  main()
5     {
6       int  i;
7       i=4;
8       g(4);
9     }
```

例 106 中，main()函数有 int 类型的返回值，但调用了没有返回值的函数 g()，导致程序中返回值类型不一致。

81．未判断函数返回值

例 107：

```
1   //SV.PAIRS.NO_CHECK
2   void myImpersonateClient (RPC_BINDING_HANDLE &my_ binding_handle) {
3       RPC_STATUS myStatus = RpcImpersonateClient (my_ binding_handle); //错误
4   }
```

例 107 程序是利用服务中的模拟函数使得远程用户能够调用，但第 3 行语句没有确认函数 RpcImpersonateClient()是否调用成功就直接调用，会导致低权限用户无法正常操作。

修改方法：在调用该函数前确认返回值的正确性，保证其返回值正确的情况下再进行调用。

82．非 void 函数中无返回值

对于返回值是非空类型的函数，它的所有退出路径都应该有一个表达式明确地给出返回值，没有此表达式，将会导致未定义的行为（编译器可能不会指出这个错误）。

例 108：

```
1  int foo1(int x){ // 第二重 if 语句没有返回值
2  if (x==0) {
3   if (x==0) {
4     }
5    else {
6      return 0;
7    }
```

```
8   }
9   else {
10    return 0;
11  }
12  }
13  int foo2(int x){ // 在 switch 语句中，没有 default 语句
14  switch(x){
15    case 0: return 1;
16    case 1: return 1;
17    case 2: return 1;
18  }
19  }
20  int foo3(int x){ //错误
21  }
```

修改后的程序如例 109 所示。

例 109：

```
1   int foo1(int x){
2   if (x==0) {
3     if (x==0) {
4       return 0;
5     }
6     else {
7       return 0;
8     }
9   }
10  else {
11    return 0;
12  }
13  }
14  int foo2(int x){
15  switch(x){
16    case 0: return 1;
17    case 1: return 1;
18    case 2: return 1;
19    default: return 1;
20  }
21  }
22  int foo3(int x){
```

```
23    return 0;
24 }
```

83. 函数无参数且无返回值时未使用 void

对于既无参数又无返回值的函数，如果未将其声明为 void，编译器将自动为其分配 int 返回类型，这将和函数实现的语义相冲突。因此，需要明确地指定参数和返回类型均为 void，以清晰地表达函数的意图。

例 110：

```
1    #include <stdio.h>
2    func1 () {
3    printf("%s\n", "Hello World");
4    };
5    void func2() {
6        int a = 2000;
7         …
8        };
9        func3 (void) {
10       return ;
11    };
```

例 110 中第 2 行语句，函数 func1 既无参数也无返回值；第 5 行语句，函数 func2 无参数；第 9 行语句，函数 func3 无返回值。

修改方法：将第 2、5、9 行语句分别修改为：

```
void func1(void)
void func2(void)
void func3(void)
```

84. 形参和实参类型不一致

函数的实参和形参类型必须一致，两者不一致将导致程序错误。

例 111：

```
1    void f(int a, int b)
2    {
3     …
4    }
5    void main(float a, float b)
6    {
7     f(a, b);
8    }
```

例 111 中，main()函数给 f()函数传递的实参类型和 f()函数声明的形参类型不一致，导致程序错误。

85．函数中的参数个数可变

当函数中的参数为省略号时，表明函数中的参数个数不确定，编译器在处理时不检查该函数的实参与形参的个数和类型是否相同，因此，容易引发潜在的问题。

例 112：

```
1    void f (int x, …)
2    {
3    }
```

例 112 第 1 行语句，函数中的参数个数不确定。

修改方法：将可变参数函数修改为确定参数函数。

86．固定长度的数组作为函数参数

数组作为函数参数时，声明数组的长度没有任何意义，因为编译器只是传递参数数组的首地址，并不检查参数数组的长度。

例 113：

```
1    void foo2(int t[30])
2   {
3   }
4    void foo3(char a,int b[30][30][30])
5   {
6   }
```

例 113 中，第 1 行和第 4 行语句分别将声明了数组长度的 t[30]和 b[30][30][30]作为函数参数传递给函数。

修改方法：将第 1 行语句中的数组 t[30]修改为 t[]

第 4 行语句中数组 b[30][30][30]修改为 b[][30][30]

87．在区块内声明函数

函数应该被声明在文件范围内，在区块范围内声明函数可能产生混淆，并且可能导致不明确的行为。

例 114：

```
1   void foo1( ) {
2       void foo2( );  // 错误
3   }
```

修改后的程序如例 115 所示。

例 115：

```
1   void foo2( );
2   void foo1( ) {
3   }
```

88. 定义的函数没有被调用

例 116：

```
1   static void foo()
2   {
3      …
4   }

5   int main()
6   {
7     return (0);
8   }
```

例 116 中，main()函数没有调用已经定义过的 foo()函数。

修改方法：在 main()函数中增加对 foo()函数的调用或去掉 foo()函数。

89. 非虚函数中存在未被使用的参数

例 117：

```
1   int Foo(int i, int k)   // 错误
2   {
3      i = 5;
4      return i;
5   }
```

例 117 中，Foo()函数体中传递的参数 int k 没有被使用。

修改方法：将 Foo(int i,int k)函数的定义修改为 Foo(int i)

90. 重载函数调用混乱

函数重载是指同一个函数名可以对应不同的函数实现方式。函数重载后，编译器一般从重载函数参数的个数和类型上来判断执行时需要调用哪个重载函数。因此，要求重载函数的参数个数或者参数的类型不同。

当类中存在以指针和数值型参数作为唯一不同实参的重载函数，且 0 是给该重载函数传递的唯一不同实参参数值时，编译器将无法确定所调用的重载函数。

例 118：

```
1   class A
2   {
3     public:
4       void f (A a, int *i) {return 0 }
5       void f (A a, long i) {return 1 }
6   };
```

例 118 中，第 4 行语句和第 5 行语句中的重载函数 f()具有两种重载形式，两种重载函数的差别在于第 2 个实参，其中一个为指针型，一个为数据 long 型。当给第 2 个实参传递 0 值时，程序将无法确定函数的重载形式。

91. 运算符重载使用引用作为返回值

运算符重载，当以引用作为返回形式时，不会产生临时变量，运算符重载函数执行结束后其运行过程中产生的结果将被释放，这将导致运算符重载函数返回的引用对象无所指；而当以值作为返回形式时，运算符重载函数运行过程中会产生临时变量，在函数执行结束时，临时变量将结果传递给返回的值，这样就能保证运算符重载函数有值可传。因此，运算符重载时，应避免使用引用作为返回值。

例 119：

```
1   class C
2   {
3    public:
4      C& operator&  (C &x );
5      C& operator^  (C &x );
6      C& operator|  (C &x );
7   };
```

例 119 中，三种符号&、^、|的重载函数使用引用作为返回值，每个运算符重载函数返回的引用，一定是函数中某个已存在对象的引用，且这个对象包含运算符计算后的结果。当运算符重载函数执行结束时，其运行过程中产生的结果将被释放，这将导致返回的引用对象无所指。

修改后的程序如例 120 所示。

例 120：

```
1   class C
2   {
3    public:
4      C operator&  (C &x );
5      C operator^  (C &x );
```

```
6       C  operator|  (C &x );
7    };
```

92. 内联函数体内含有局部静态变量

内联函数在编译时即被插入到调用者代码处，而不像一般函数那样在运行时才被调用。如果内联函数没有参数，但函数体内含有局部静态变量，该静态变量在程序中将具有多个副本。

例 121：

```
1  inline int f()
2  {
3      static int a = 0;
4      a++;
5      return a;
6  }
```

例 121 中，内联函数 f()没有参数，但在其函数体内却含有局部静态变量，这将导致该静态变量在程序中存在多个副本。

修改后的程序如例 122 所示。

例 122：

```
1  int f()
2  {
3      static int a = 0;
4      a++;
5      return a;
6  }
```

93. 幂函数 pow()参数错误

幂函数原型为：

extern float pow(float x, float y);

该函数计算 x 的 y 次幂，要求参数 x 大于零。

例 123：

```
1   #include<stdio.h>
2   #include<math.h>
3
4   int main()
5   {
```

```
6      float x=0;
7      float y=0;
8      scanf("%f", &x);
9      scanf("%f", &y);
10     printf("b%f^%f = %f", x, y, pow(x,y)); // 错误
11     return 0;
12   }
```

例 123 中，程序从用户输入中读入 x 和 y，计算 x 的 y 次幂并打印出来。在第 10 行语句使用 pow()函数时并未对参数进行合法性验证，因此可能出现（x=0，y<0）或者（x<0，0<y<1）等错误情况。

94．函数通过直接或间接方式调用自身

如果函数通过直接或间接方式调用自身，程序的可维护性和可读性较差，而且这种多层嵌套容易使程序隐含潜在的缺陷。

例 124：

```
1   void f ( int i ) {
2       int x = 6;
3       if (i > 0) {
4            f ( x - 1 );
5       }
6   }
```

例 124 中，函数 f()在函数体内调用自身。

95．函数定义与声明的参数名称不一致

函数参数的名称在定义和声明时应该一致，这样可以提高程序的可读性和清晰度。

例 125：

```
1   void f (int a, int b); //函数声明
2   void f (int x, int y ) {} //函数定义
```

例 125 中，函数声明时的参数名称和函数定义时的参数名称不一致。

1.4 条件语句问题

switch 语句和 if 语句是两种常用的条件语句。switch 语句可以产生具有多个分支的控制流程，在使用时经常出现变量类型不一致、省略了 break 语句以及 case 条件后面没有可执行的语句等错误。

96. switch语句变量类型不一致

当多个枚举类型的值被混用在switch分支表达式或switch分支条件语句中时，容易造成case条件变量的值和switch语句中变量类型不一致。

例126：

```
1   void choice ()
2   {
3   enum Q1{Q1Send, Q1Recv}; //枚举类型Q1
4   enum Q2{Q2None, Q2Send, Q2Recv};//枚举类型Q2
5   enum Q1  q;
6    switch (q)
7    {
8    case Q2Send: f(); break; // 错误
9    case Q2Recv: g(); break; // 错误
10   case Q1Send: f(); break;
11   case Q2None: g(); break; // 错误
12   }
13  }
```

例126中，Q1、Q2都是枚举类型，q的取值范围是枚举类型Q1的值，但在第8、9、11行语句中，q的取值是枚举类型Q2的值，导致程序错误。

97. switch语句中遗漏break语句

switch语句中的控制流程能够根据条件判断执行相应的case部分，执行结束后跳出switch语句。当switch语句中遗漏break语句时，程序会执行满足条件的case语句以及其后的case语句，直至整个switch语句结束。因此，在使用switch语句时，不能遗漏break语句。

例127：

```
1   main()
2   {
3   enum  date {Monday, Sunday, Thursday, Weekday} ;
4   date  q;
5   switch  (q)
6     {
7      case Monday: printf("Monday");
8      case Sunday: printf("Sunday");
9      case Thursday: printf("Thursday");
10     case weekday: printf("Weekday");
```

```
11    }
12  }
```

例 127 中，当 q 取值为 Sunday 时，因为在 case 语句中遗漏了 break 语句，程序输出错误的结果：SundayThursdayWeekday。

98. switch 语句中 case 条件后无可执行语句

switch 语句中当变量满足某种 case 条件，却没有可执行的语句时，应该在 switch 语句中略去该 case 语句。

例 128：

```
1   int g(int i)
2    {
3   switch(i)
4     {
5         case 0:  return 1;
6         case 1:     // 错误
7         case 2:     // 错误
8         default: return 0;
9       }
10  }
```

例 128 的 g() 函数中，当变量 i 值为 1 或 2 时，不执行任何语句。

修改方法：在 switch 语句中略去 i 值为 1 或 2 时的 case 分支。修改后的程序如例 129 所示。

例 129：

```
1   int g (int i)
2   {
3     switch (i)
4     {
5       case 0:
6           return 1;
7       default:
8           return 0;
9     }
10  }
```

99. switch 语句中各 case 语句返回值类型不一致

例 130：

```
1   int System_comm( int GetLastCode)
```

```
2   {
3       switch (Communication)
4       {
5         case Nmc_Cfg_Add:
6           return (-1);
7           break;
8         case Nmc_Cfg_Mod:
9           return (1U);
10          break;
11        case Nmc_Cfg_Del:
12          return (1L);
13          break;
14            …
15        default:
16          break;
17      }
18  }
```

例 130 中，各 case 语句的返回值类型不一致。

100．switch 语句中使用布尔表达式

在多条件情况下应该使用 case 语句，若只有 true 和 false 两种情况，应使用 if 和 else 条件语句替代 case 语句。

例 131：

```
1   void UserCfg (void)
2   {
3     bool cUserName = false;
4     …
5     switch (cUserName)
6     {
7       case true:
8     …
9           break;
10      case false:
11    …
12          break;
13      default:
14    …
15          break;
```

```
16      }
17  }
```

例 131 中，只有 true 和 false 两种情况，default 语句变成不可达代码。

101. switch 语句未提供 default 分支

switch 语句中，当变量的取值不满足任何 case 条件时，应该在 switch 语句中增加 default 分支，用于处理 case 条件以外的情况。

例 132：

```
1   void foo( ) {
2       int tag;
3       switch ( tag ) {
4       case 0: {
5           break;
6               }
7       case 1: {
8           foo( );
9           break;
10              }
11      }
12  }
```

例 132 中，switch 语句未包含 default 分支处理 case 条件以外的情况。

修改方法：在 switch 语句中增加 default 分支。

102. if 语句或循环语句中的条件为非布尔类型

如果 if 语句或循环语句的条件中使用非布尔类型(bool)的表达式，那么其结果将会被隐式地转换为布尔类型。为了准确表达程序员的意图，条件表达式应使用显式的判断条件。

例 133：

```
1   void foo()
2   {
3       int i;
4       if (i){} // 错误
5   }
```

修改方法：将 if(i)修改为：`if(i! =0)`

103．条件运算符的第一个操作数是非布尔类型数据

条件运算符的第一个操作数应该是布尔类型，否则，其结果会被隐式转换为布尔类型。

例 134：

```
1  void foo(int i, int j, int k, int l)
2  {
3     i = j ? k : l;
4  }
```

例 134 第 3 行语句中，条件运算符的第一个运算符不是布尔类型。

修改方法：将第 3 行语句修改为：`i = (j != 0) ? k : l;`

104．条件判断语句导致使用未初始化的堆内存

程序中使用内存分配函数 malloc()为结构体分配堆内存后，由于使用了条件判断语句，可能出现分配的堆内存没有初始化就被使用的情况。

例 135：

```
1   struct student{
2     int num;
3     char[10] name;
4   };
5   int  main(int t)
6   {
7    struct student *s=malloc(sizeof(struct s));
8    if (t>0)
9      {
10      s->num=t;
11      s->name="liyue";
12     }
13   return s->num;
14  }
```

例 135 中第 8 行语句，当 if 条件不成立时，第 13 行语句 return 返回的将是没有初始化的“s->num”。

105．条件判断语句导致读取结构体中未赋值的局部变量

程序中的条件判断语句可能导致在结构体中局部成员没有赋值的情况下，读取该成员的值或将该成员作为参数传递给某个函数。

例 136:

```
1    struct  s
2     {
3      int  a;
4      int  b;
5     };
6    main(int t)
7     {
8       struct s x;
9       if(t>0)
10       {
11        x.a=2;
12        x.b=3;
13       }
14      max(x.a, x.b)
15    }
```

例 136 中，当 if 条件不成立时，结构体 x 中的成员变量将没有值，但在第 14 行语句中却被作为参数传递给了 max()函数。

106. 条件判断语句导致类成员未初始化

程序中的条件判断语句可能导致程序中类的构造函数没有对类域内的成员初始化。

例 137:

```
1     class  c
2    {
3     private: int i;
4               int  j;
5               bool flag;
6     public:  c()
7       {
8        if(flag)
9        {
10         i=0;
11         j=1;
12       }
13      }
14   };
```

例 137 中，当 flag 变量取值为真时，类 c 构造函数中 i 和 j 才被赋值；当 flag 取值为假时，将无法在类 c 构造函数中给 i 和 j 赋值。

1.5 循环语句问题

107. 使用复杂的 for 循环表达式

for 循环表达中，只应该完成与循环条件有关的操作。

例 138：

```
1   void Ex_profession(void)
2   {
3     Uint_32 loop;
4     Uint_32 myVar = 0;
5     const Uint_32 max = 10U;
6     …
7     for (loop = 0, myVar = 1U; loop < max; loop++)
8     {
9       …
10    }
11  }
```

例 138 中第 7 行语句，for 循环表达式包含对变量 myVar 的赋值操作，这样做虽然对程序运行结果没有影响，但程序的可读性差。

108. for 循环语句中循环计数器使用不当

如果 for 循环语句中循环计数器不是被-- 或 ++ 修改，那么在循环条件中此循环计数器应该作为 <=，<，> 或 >= 的操作数。

例 139：

```
1   void foo()
2   {
3     int i;
4     for ( i = 1; i != 10; i += 2 ){}
5   }
```

例 139 第 4 行 for 循环语句中，循环计数器 i 不是被-- 或 ++ 修改，循环计数器 i 没有作为 <=，<，> 或 >= 的操作数。

修改方法：将第 4 行语句修改为：`for ( i = 1; i <= 10; i += 2 ){}`

109. for 循环语句使用不当

当 for 循环语句中无初始化或无自增/自减运算符时，采用 while 循环替代 for 循环语句。

例 140：

```
1   void foo()
2   {
3    for(int i=0; i< 10;)
4    {
5       …
6    }
7    int j = 0;
8    for(; j< 10;)
9    {
10      …
11    }
12  }
```

例 140 中，第 3 行 for 循环语句无自增运算符、第 8 行 for 循环语句中无初始化和自增运算符。

修改方法：用 while 循环替代 for 循环语句。

110. 循环控制表达式中非循环变量被修改

变量或参数可能会通过传递其非 const 指针或引用到外部函数而被修改，因此，除了循环计数器以外的循环控制变量不得在判定条件或者表达式中被修改。

例 141：

```
1    bool test(int x);
2    void foo(int x, bool flag)
3    {
4       for ( x = 0; x < 10; flag = test(++x) ) {}
5    }
```

例 141 第 4 行语句中，变量 flag 在 for 循环语句中被修改。

修改方法：将第 4 行语句修改为：`for ( x = 0; x < 10; x++ ) {}`

111. 循环中含有多出口

例 142：

```
1    for(int i = 0;i<num;i++)
2        {
```

```
3        l_pRecordset = m_Ado.ExecuteQuery(cSql,iCount);
4            if(l_pRecordset!=NULL)
5            {
6                    …
7                    if(vtTemp.vt != VT_NULL)
8                    {
9                        …
10                       break;
11                   }
12   …
13           }
14       }
```

例 142 中，第 10 行语句在循环中插入了 break 语句，导致该循环存在多出口，不符合结构化编程要求。

修改方法：把该行单独提取出来进行操作。

112．循环结束条件设置错误导致无限循环

当 for、do-while、while 等循环体中的循环结束条件永远不能为真时，将导致无限循环。

例 143：

```
1    int main()
2     {
3       int n = 5;
4       do
5       {
6         printf("Now the value of n is :%d\n",n--);
7         n--;
8        }while(n!=0);
9      return 0;
10    }
```

例 143 中，由于循环体中的 n 每次循环都减去 2，因此 n 永远不会为 0，导致 do-while 进入无限循环。

113．忽略 while 和 do-while 的区别

while 语句和 do-while 语句是两种很常见的循环结构，在大多数情况下二者实现的结果一致，但在细节上有所区别。

例 144：

```
1    void Func1(int arr[100])
```

```
2  {
3    int i;
4    int sum = 0;
5    scanf("%d", &i);
6    if(i >= 0)
7    {
8      while(i < 100)
9      {
10       sum = sum + arr[i];
11       i++;
12     }
13     printf("%d", sum);
14   }
15 }
16
17 void Func2(int arr[100])
18 {
19   int i;
20   int sum = 0;
21   scanf("%d", &i);
22   if(i >= 0)
23   {
24     do
25     {
26       sum = sum + arr[i];
27       i++;
28     } while(i < 100);
29     printf("%d", sum);
30   }
31 }
```

例 144 中，分别使用 Func1()和 Func2()实现以下功能：求数组 arr[]的某个元素开始的所有元素之和并打印。当 0≤i<100 时，两个函数执行的结果相同。由于 while 循环语句是先判断后执行，而 do-while 语句是先执行后判断，因此，当 i<0 或者 i≥100 时，Func1()中 while 循环条件为假而不执行循环体，而 Func2()中的 do-while 会执行一次循环体再进行判断，从而导致数组索引越界错误。

1.6　数值类型转换问题

在表达式的运算过程中，运算符所处理的数据类型可能不相同。C/C++语言规定，对于不同类型的数据要先转换成相同的数据类型才能进行运算。因此，当一个运算符中的几个操作数的类型不相

同时，就需要通过一些规则把它们转换成某种相同类型的数据。数值类型转换分为隐式转换和强制转换两种，其中强制转换又分为显式强制转换和隐式强制转换。隐式转换也称为自动转换，它是通过一些规则将不同的数据类型转换为某种相同的数据类型；显式强制转换是通过使用强制类型转换的单目转换符“()”进行的；隐式强制转换通过两种方式实现，一种是在赋值语句中，将右边表达式值的类型转换成左边变量的类型；另一种是在函数体中，使用 return 语句返回时，将 return 语句后面表达式值的类型转换为函数值的类型。隐式转换分为如下三种类型。

（1）整数提升（Integral Promotion）转换

整数提升是指当进行算术运算时参与运算的操作数类型必须是有符号或无符号的 int 或 long 型，任何其他整数类型（char、short、bit-field、enum）的操作数在进行算术操作之前，必须转换为 int 或 unsigned int 类型。如果 int 类型能够代表原有类型的值，则将其转换为 int 类型，否则将其转换为 unsigned int 类型。需要注意的是，不是在任何场合下都需要进行整数提升，整数提升仅仅适用于 char、short、bit-field、enum 类型，适用于一目、二目和三目操作数，也适用于 switch 语句的控制表达式，但不能够用于逻辑运算符&&、||、!的操作数上。

由于整数提升的原因，两个类型为 unsigned short 的对象相加的结果总是 signed int 或 unsigned int 类型。事实上，加法是在后两种类型上进行的，因此，对于有些操作，其结果可能超出原始操作数类型的大小范围。例如，如果 int 类型的大小是 32 位，两个 16 位 short 类型的对象相乘可以得到一个无溢出风险的 32 位结果；如果 int 类型的大小仅是 16 位，那么这两个 16 位对象的乘积将只能产生一个 16 位的结果，同时必须对操作数的大小进行适当限制。

整数提升是由 C 语言本质上的不一致性造成的。在 C 语言中，char、short、bit-field、enum 类型的行为与 long 和 int 类型的行为是不同的。因此，建议使用 typedef。

（2）赋值转换

在下列情况下会发生赋值转换。

- 赋值表达式的类型被转换为赋值对象的类型时；
- 初始化表达式的类型被转换为初始化对象的类型时；
- 函数调用参数的类型被转换为函数原型中声明的形式参数的类型时；
- 返回语句中用到的表达式的类型被转换为函数原型中声明的函数类型时；
- switch-case 条件中常量表达式的类型被转化为控制表达式的提升类型时。

对于每一种情况，根据需要，算术表达式的值被无条件转换为另外一种类型。

（3）平衡转换

当双目运算符的两个操作数或三目运算符（?:）中的第二和第三个操作数要平衡为一个公共类型时，ISO C 标准下的“Usual Arithmetic Conversions”给出了一种平衡机制。平衡转换总是涉及两个不同类型的操作数，其中的一个（有些情况下是两个）操作数需要进行隐式转换。与平衡转换明显相关的运算符包括：*、/、% 、+、-、&、^、|、?:、>、>=、<、<=、==、!=。上述大多数运算符会产生一个由平衡机制所确定的结果类型，其中，关系运算符>、>=、<、<=和等值运算符==、!=产生一个类型为 int 的布尔值。需要注意的是，移位运算符<<、>>中的操作数不进行平衡转换，运算结

果被提升为第一个操作数的类型，第二个操作数可以是任何有符号或无符号的整型数。

整数提升经常与平衡转换混淆，实际上，只有在一目运算或双目运算中两个操作数的类型相同时才进行整数提升。

上述类型转换，有些是安全的，有些则存在风险。对于所有数据和所有可能的一致性实现而言，唯一可以确保安全的转换是：①将整型值转换为同符号的更宽类型；②将浮点类型转换为更宽的浮点类型。除此之外的其他类型转换均存在潜在的风险，例如，数据精度丢失、数值丢失、符号丢失、与预期语义不相符等。为了避免类型转换带来的风险，建议采用显式转换。

数值转换错误主要包括三类：浮点数运算、不同数值类型之间的相互转换、void*的类型转换。

114. 隐式转换类型错误

在下列情况下，整型表达式的值不应隐式转换为其他的基本类型。

① 该转换不是一个向同类有符号的更宽整数类型的转换；

② 表达式是复杂表达式；

③ 表达式不是常量而是函数参数；

④ 表达式不是常量而是返回表达式。

在下列情况下，浮点类型表达式的值不应隐式转换为其他的类型。

① 该转换不是一个向更宽浮点类型的转换；

③ 表达式是复杂表达式；

③ 表达式是函数参数；

④ 表达式是返回表达式。

此外，下列隐式转换也是不允许的。

① 有符号和无符号之间的隐式转换；

② 整型和浮点类型之间的隐式转换；

③ 宽类型向窄类型的隐式转换；

④ 函数参数的隐式转换；

⑤ 函数返回表达式的隐式转换；

⑥ 复杂表达式的隐式转换。

例 145：

```
1   extern void foo1 (uint8_t x);
2   int16_t t1 (void)
3   {
4   …
5   foo1 (u8a);
6   foo1 (u8a + u8b);
7   foo1 (s8a);     // 错误
```

```
8   foo1 (u16a);    //错误
9   foo1 (2);    //错误
10  foo1 (2U);
11  foo1 ( (uint8_t) 2 );
12  … s8a + u8a     //错误
13  … s8a + (int8_t) u8a
14  s8b = u8a;     //错误
15  … u8a + 5      //错误
16  … u8a + 5U
17  … u8a + (uint8_t) 5
18  u8a = u16a;     //错误
19  u8a = (uint8_t) u16a;
20  u8a = 5UL;     //错误
21  … u8a + 10UL
22  u8a = 5U;
23  … u8a + 3     //错误
24  … u8a >> 3
25  … u8a >> 3U
26  pca = "P";
27  … s32a + 80000
28  … s32a + 80000L
29  f32a = f64a;     //错误
30  f32a = 2.5;     //错误
31  u8a = u8b + u8c;
32  s16a = u8b + u8b;     //错误
33  s32a = u8b + u8c;     //错误
34  f32a = 2.5F;
35  u8a = f32a;     //错误
36  s32a = 1.0;     //错误
37  f32a = 1;     //错误
38  f32a = s16a;     //错误
39  … f32a + 1      //错误
40  … f64a * s32a   //错误
    …
41  return (s32a);   //错误
    …
42  return (s16a);
    …
43  return (20000);
```

```
   …
44 return (20000L);   //错误
   …
45 return (s8a);    //错误
   …
46 return (u16a);    //错误
47 }
```

例 146:

```
1  void change(long i)
2  {
3    char c;
4    c=i; //错误
5  }
```

例 146 中，对于 32 位系统，第 4 行语句中，i 为 long 型占 4 个字节，c 为 char 类型占 1 个字节，因此，long 型在被隐式转换为 char 型时将导致数据精度损失。

115. 隐式转换枚举类型

程序中枚举和非枚举类型、不同的枚举类型间进行隐式类型转换都是不允许的，这些转换可能导致程序异常。

例 147:

```
1  enum T{A0, B0, C0};
2  void foo1(int arg1){}
3  int foo(int p)
4  {
5      T x;
6      int i = x; // 错误
7      foo1(x);   // 错误
8      return x;  // 错误
9  }
```

例 147 中，第 6、7、8 行语句使用枚举类型做隐式类型转换，可能导致程序错误。

修改后的程序如例 148 所示。

例 148:

```
1  enum T {A0, B0, C0};
2  void foo1(int arg1){}
3  int foo(int p)
```

```
4   {
5       T x;
6       int i = (int)x;
7       foo1((int)x);
8       return (int)x;
9   }
```

116. 显式转换类型错误

整型复杂表达式的值只能强制转换到更窄的类型且与表达式的基本类型具有相同的符号；浮点类型复杂表达式的值只能强制转换到更窄的浮点类型，否则，将导致程序错误。

例 149：

```
1   … (float32_t) (f64a + f64b)
2   … (float64_t) (f32a + f32b)     // 错误
3   … (float64_t) f32a
4   … (float64_t) (s32a / s32b)     // 错误
5   … (float64_t) (s32a > s32b)     // 错误
6   … (float64_t) s32a / (float32_t) s32b
7   … (uint32_t) (u16a + u16b)     // 错误
8   … (uint32_t) u16a + u16b
9   … (uint32_t) u16a + (uint32_t) u16b
10  … (int16_t) (s32a - 12345)
11  … (uint8_t) (u16a * u16b)
12  … (uint16_t) (u8a * u8b)     // 错误
13  … (int16_t) (s32a * s32b)
14  … (int32_t) (s16a * s16b)     // 错误
15  … (uint16_t) (f64a + f64b)    // 错误
16  … (float32_t) (u16a + u16b)  // 错误
17  … (float64_t) foo1 (u16a + u16b)
18  … (int32_t) buf16a[u16a + u16b]
```

117. 位运算符~和<<类型转换错误

如果位运算符~和<<应用于基本类型为 unsigned char 或 unsigned short 的操作数上时，运算结果应立即转换为与操作数同类型的基本类型。

当~和<<运算符应用于 char、short、bit-field、enum 类型（unsigned char 或 unsigned short）时，运算之前要先进行整数提升，因此，运算结果中可能包含并非预期的高端数据位。

例 150：

```
1   uint8_t port = 0x5aU;
```

```
2  uint8_t result_8;
3  uint16_t result_16;
4  uint16_t mode;
5  result_8 = (~port) >> 4;      // 错误
```

例 150 中，第 5 行语句右边的操作执行后，在 16 位机器上的结果是 0xffa5，而在 32 位机器上是 0xffffffa5；将计算结果赋值后，无论哪种情况，其结果都是 0xfa，然而期望的值却是 0x0a。这样的危险可以通过强制转换来避免。

修改后的程序如例 151 所示。

例 151：

```
result_8 = ( ( uint8_t ) (~port ) ) >> 4;
result_16 = ( ( uint16_t ) (~(uint16_t) port ) ) >> 4 ;
```

同样的问题也会出现在<<运算符上，如例 152 所示。

例 152：

```
result_16 = ( ( port << 4 ) & mode ) >> 6 ;    // 错误
```

例 152 中，result_16 的值将依赖于 int 的大小。

修改后的程序如例 153 所示。

例 153：

```
result_16 = ( ( uint16_t ) ( ( uint16_t ) port << 4 ) & mode ) >> 6 ;
```

118．指针类型转换错误

指针类型包括：对象指针、函数指针、void*、空（null）指针常量。

除了下面的两种情况外，其他涉及指针类型的转换都需要显式进行。

- 对象指针和 void*之间进行的转换，并且目标类型包含了源类型所有的类型标识符；
- void*被赋值给任何类型的指针或与其做等值比较时，void*被自动转化为特定的指针类型。

C 语言中只定义了一些特定的指针转换，特定的指针转换包括：

① 除了整型之外，函数指针不能与任何其他类型指针进行转换，这种转换的结果是不确定的；

② 除了整型、其他对象指针和 void*之外，对象指针不能与任何其他类型指针进行转换；

③ 如果指针所指向的类型带有 const 或 volatile 限定符，那么不允许移除限定符进行强制转换。

例 154：

```
1   uint16_t x;
2   uint16_t * const cpi = &x;
3   uint16_t * const * pcpi ;
```

```
4   const uint16_t * * ppci ;
5   uint16_t * * ppi;
6   const uint16_t * pci;
7   volatile uint16_t * pvi;
8   uint16_t * pi;
9   …
10  pi = cpi; // 错误
11  pi = (uint16_t *)pci; // 错误
12  pi = (uint16_t *)pvi ; // 错误
13  ppi = (uint16_t *)pcpi ; // 错误
14  ppi = (uint16_t *)ppci ; // 错误
```

除非是访问内存映射寄存器或其他由硬件实现的功能部件，建议尽量避免在指针类型和整型之间进行强制转换。

建议尽量避免在不同对象类型指针之间进行强制转换。如果新的指针类型需要更严格的分配时这样的转换可能是无效的。

例 155：

```
1   uint8_t p1;
2   uint32_t p2;
3   p2 = (uint32_t *) p1;   //错误
```

119. 整型数字和字符间赋值错误

内存中的字符数据以 ASCII 码存储，即以整数表示，如果把字符型的值赋给整数类型的变量，那么该整数变量的值就是该字符的 ASCII 码。例如，将字符值赋值给整型变量的语句 int a='b'，其执行结果为 a=98。当字符型数据赋给整型变量时，由于字符只占一个字节，而整型变量占两个字节，因此将字符数据放到整型变量的低八位中；如果把整数赋给字符型变量，那么取该整数值的低八位部分所对应的 ASCII 码字符赋给该字符型变量。

例 156：

```
1   main()
2   {
3   char a = 256 ;
4   int d = a ;
5   cout<<d<<endl ;
6   }
```

例 156 中，第 3 行语句把整数 256 赋值给字符 a 后，再把字符 a 的值赋给整数 d，整数 d 的值是 0，而不是 256。

120. 数值运算导致上溢出/下溢出

程序员在编程时会涉及频繁的数值计算，由于计算机（编译器）中每种数据类型都有相应的取值范围，如果在进行数值计算时没有考虑到这一点，就可能造成数值溢出。如果数值大于该类型数据的最大值，称为上溢出（Overflow）；如果数值小于该类型数据的最小值，称为下溢出（Underflow）。32 位机器中几种主要数值类型的取值范围如表 1-2 所示。

表 1-2 32 位机器中几种主要数值类型取值范围

数 值 类 型	取 值 范 围
short	–32768～32767
int	–2147483648～2147483647
long	–2147483648～2147483647
float	–2^127～–2^(–129)（负数）2^(–129)～(1–（2^–23）)*2^127（正数）

例 157：

```
1   void Flow_Sub(char *tmpBuf,…)        //tmpBuf 从外部输入得到
2   {
3     int tmp;
4     int indNum;
5     float f1,f2;
6     …
7     tmp = tmpBuf[4]-1; // 错误
8     …
9     indNum = tmpBuf[5];
10   for(int n = 0; n < indNum; n++)   // 错误
11   {
12     …
13   }
14   …
15   f1 = 3.40282347e+38f;
16   f2 = (float) 3.40282347e+38; // 错误
17   …
18  }
```

例 157 中，由于 tmpBuf 值从外部输入获得，其值无法确定，因此在第 7 行语句中直接使用可能产生整数的下溢出，而在第 10 行语句中的“n++”操作可能产生上溢出。浮点数的最大值用十进制表示应该为：340282346638528859811704183484516925440.0。在第 15 行和第 16 行语句使用两种不同的方式将一个常数转换成浮点数。第 16 行语句使用的是强制转换，编译器在执行此操作时分作两步：第一步将常数转换为 double 类型并将其存到一个临时变量中，第二步再将临时变量转换为 float

类型。由于常量转换为 double 类型时是取符合精度要求的最小较大值，而常量转换为 float 类型时是取符合精度要求的最大较小值，因此第 16 行语句的转换会产生一个浮点数上溢出，而第 15 行语句则不会产生溢出。

121. 赋值语句两端类型不一致

赋值操作应该在相同类型的变量间进行，这样可以避免数值在类型隐式转换过程中的精度丢失。当赋值语句中右操作数来源于函数返回值时，也应该遵守这个原则。

例 158：

```
1    main()
2    {
3      int  a=1;
4      float b=4.1;
5      a=b; // 错误
6      b=a; // 错误
7    }
```

例 158 中，a 是 int 类型的变量，b 是 float 类型的变量，第 5 行语句和第 6 行语句在 int 类型和 float 类型的变量间进行赋值，从而产生错误。

例 159：

```
1    float  f()
2    main()
3    {
4      int  a=1;
5      a=f(); // 错误
6    }
```

例 159 中，a 是 int 类型的变量，f()函数的返回值是 float 类型的变量，在第 5 行语句中，使用 float 类型的变量为 int 类型的变量赋值，从而产生错误。

例 160：

```
1    void f(char *p)
2    {}
3    main()
4    {
5      char *str="sting1";
6      char str1[]="string2";
7      f("sting3");
8    }
```

编译器在处理常量字符串时，不仅把字符串复制存储于静态区中，而且记录了指向该字符串第一个字符的指针。因此，在程序中常量字符串不能直接赋值给变量或作为变量传递给函数。如果需要传递常量字符串给指针、字符数组或函数，可以通过 const 关键字来声明赋值或传值给函数的常量参数方式，从而解决常量给变量赋值或传值不正确的问题。修改后的程序如例 161 所示。

例 161：

```
1   void f(const char *p)
2   {}
3   main()
4   {
5     const  char  *str="sting1";
6     const  char  str1[]="string2";
7     f("sting3");
6   }
```

122. 无符号整型变量赋值为负数

当有符号常量的数值小于 0 时，不能将其赋值给无符号整型变量。

例 162：

```
1   main()
2   {
3     unsigned int y;
4     y = -21; // 错误
5     …
6   }
```

例 162 第 4 行语句中，无符号整型变量 y 被赋值为有符号负数（−21）。

修改方法：将第 3 行语句修改为：signed int y。

123. 有符号和无符号字符赋初值错误

有符号字符（char）的取值范围是−128 到 127，无符号字符（char）的取值范围是 0 到 255，在给字符赋初值时一定要保证在其取值范围内。

例 163：

```
1   void f()
2   {
3     char ch1;
4     unsigned  char ch2;
5     ch1 = -145; // 错误
```

```
6   ch2 = 298; // 错误
7 }
```

例 163 中，第 5 行语句在给有符号字符变量赋初值时，其值超出了有符号 char 类型的取值范围；第 6 行语句在给无符号字符变量赋初值时，其值超出了无符号 char 类型的取值范围。

124．浮点数赋值为整数除运算结果

当两个整数相除（运算符“/”）时，商的小数部分将被截掉。例如：5/4 的结果是 1。如果想要得到更精确的结果（保留小数部分的商），应至少保证除数或被除数中有一个是浮点数类型。

例 164：

```
1  void div()
2  {
3    int a = 2;
4    int b = 5;
5    double d;
6    d = a / b; // 错误
7  }
```

例 164 中第 6 行语句，将整数 a、b 相除的结果（整型数据）赋值给双精度浮点类型。

修改方法：将第 6 行语句修改为：d=((double)a)/b;

125．大小写字符转换和检查函数混淆

大写英文字母检查函数 isupper(c)函数的头文件是“ctype.h”。该函数检查参数 c 是否为大写英文字母。若参数 c 为大写英文字母，则返回非 0 值，否则返回 0。

小写英文字母检查函数 islower(c)函数的头文件是“ctype.h”。该函数检查参数 c 是否为小写英文字母。若参数 c 为小写英文字母，则返回非 0 值，否则返回 0。

大写英文字母转换成小写英文字母函数 tolower(c)函数的头文件是“stdlib.h”。该函数把大写英文字母转换为小写英文字母。若参数 c 为大写英文字母，则将其对应的小写英文字母返回，若不需转换则将参数 c 值返回。

小写英文字母转换为大写英文字母函数 toupper(c)函数的头文件是“stdlib.h”。该函数把小写英文字母转换为大写英文字母。若参数 c 为小写英文字母，则将其对应的大写英文字母返回，若不需转换则将参数 c 值返回。

例 165：

```
1  int  main()
2  {
3   char  a;
4   int  flag;
```

```
5      scanf("%c", &a);
6      flag= toupper(a); // 错误
7      if(flag)  return 1;
8      else      return 0;
9    }
```

例 165 中第 6 行语句，将 isupper()函数误用为字母转换 toupper()函数，致使 flag 始终是非 0 值，从而导致第 8 行语句成为不可达代码。

126．误用强制类型转换运算符 dynamic_cast

强制类型转换 dynamic_cast 运算符的使用方法是：

```
dynamic_cast <type>(expression)。
```

该运算符把 expression 转换成 type 类型的对象，常用于将基类指针（或引用）转换为继承类指针，同时，dynamic_cast 会根据基类指针是否真正指向继承类指针来做相应处理。

例 166：

```
1    class A { …
2    public:
3    int  f(){return 0;}};
4    class B: public  A { … };
5    void g()
6    {
7     …
8      B *pb = new B;
9      B *pv = dynamic_cast<B*>(pb); // 错误
10   }
```

例 166 中，第 9 行语句试图将指针 pv 指向 B 类的对象。每当创建具有虚函数的类或从该类派生一个类时，编译器就为该类创建一个唯一的 VTABLE。在 VTABLE 中放置这个类或其基类中所有虚函数的地址。在转换时，dynamic_cast 依赖于 RTTI(Run-Time Type Identification)信息，检查转换的源对象是否能够转换成目标类型。dynamic_cast 先查看 RTTI 相关部分，通过函数指针表 VTABLE，找到对象的 RTTI 信息。如果基类没有虚函数，也就无法判断基类指针变量所指对象真实的类型，dynamic_cast 转换就会出错。

修改方法：把类的成员函数 f()修改为虚函数，再使用 dynamic_cast 转换。

127．使用 static_cast 强制转换数据的 const 属性

static_cast 运算符的使用方法是：

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

该运算符把 expression 转换为 type 类型。static_cast 主要用于如下几种场合。

① 类层次结构中基类和子类之间指针或引用的转换；

② 基本数据类型之间的转换，例如：int 转换成 char；

③ 空指针转换成目标类型的空指针；

④ 任何类型的表达式转换成 void 类型。

例 167：

```
1  main()
2  {
3   const  char  *x= "hello";
4     …
5   int *t= static_cast<int*>(*x); // 错误
6     …
7  }
```

例 167 中第 5 行语句，static_cast 试图改变常量 x 的属性。事实上，static_cast 并不能转换 expression 的 const、Volatile 和 unaligned 属性。

128. 数据类型 const_cast 转换错误

const_cast 运算符的使用方法是：

```
const_cast<type>(expression);
```

该运算符用来修改类型的 const 或 volatile 属性。其中的 type 和 expression 的类型是相同的。const_cast 主要用于如下几种场合。

① 将常量指针转换成非常量指针，并且仍然指向原来的对象；

② 将常量引用转换成非常量引用，并且仍然指向原来的对象；

③ 将常量对象转换成非常量对象。

例 168：

```
1  main()
2  {
3     …
4    const int a=100;
5    int b=const_cast <int> (a); // 错误
6     …
7  }
```

例 168 中，第 5 行语句使用 const_cast 试图把整型常量 a 转换为整型变量 b，而 const_cast 仅仅用于将指针、引用和对象常量转换为其对应的非常量，不能用于其他数据类型常量和非常量之间的转换。

129．类型转换 reinterpret_cast 使用错误

reinterpret_cast 运算符的使用方法是：

```
reinterpret_cast<type>(expression);
```

type 必须是一个指针，引用算术类型、函数指或者成员指针。该运算符可以将一个指针转换为一个整数，也可以将一个整数转换为一个指针。

例 169：

```
1   main()
2   {
3     int i;
4      string  p = "This is a example";
5      i = reinterpret_cast<int>(p); // 错误
6      …
7   }
```

例 169 中，第 5 行语句使用 reinterpret_cast 试图把 string 型变量 p 通过 reinterpret_cast 转换为 int 类型变量。

修改方法：把变量 p 修改为 string 型指针变量（string *p），再使用 reinterpret_cast 转换。

130．将 const 转换成非 const

const 所修饰的对象不能被改变，例如，const int a=0 中的整型常量 a 不能被改变。变量一旦被定义为常变量，在初始化时就要使用 const 修饰。一般情况下，非 const 类型变量和 const 类型变量间不能相互赋值。const 主要用于以下几种场合。

① const 修饰一般常量、常数组和常对象。表明被修饰的常量值不能被更新；

② const 修饰指针（const int *a 或 int const *a）。表明 const 修饰指向的对象不可变但指针 a 可变。例如，语句 int *const a，表明 const 修饰指针 a，a 不可变，a 指向的对象可变；语句 const int *const a，表明指针 a 和指针 a 指向的对象都不可变；

③ const 修饰引用（const int &a）。表明该引用所引用的对象不能被改变；

④ const 修饰函数的返回值。表明函数的返回值不可变；

⑤ const 修饰类的成员函数。表明在调用被修饰的成员函数时，不能修改类里面的数据；

⑥ 在另一连接文件中引用 const 常量（extern const int i）。表明该常量不能被再次赋值。

例 170：

```
1   void foo( const int a, const int *b ) {
2      int x;
3      int *y;
```

```
4     (int&) a = x;    // 错误
5     y = (int*) b;   // 错误
6    *y = 10;
7  }
```

例 170 中第 4 行语句，将非 const 类型的变量 x 赋值给 const 类型的变量 a；第 5 行语句将 const 类型的变量 b 赋值给非 const 类型的变量 y。

修改方法：避免 const 类型变量和非 const 类型变量间相互赋值。

131. 算术运算操作数类型有误

C/C++语言中的运算分为有符号运算和无符号运算。执行算术运算时，如果操作数的类型不同，在运算过程中，编译器会自动转换操作数类型，使其一致。这种转换因编程语言的标准不同，其转换方式也各异。

例 171：

```
1    f()
2    {
3      unsigned char a=254;
4      signed char b=254;
5      printf("%d \n",a/b);
6    }
```

例 171 中，打印出来的结果不是 1，而是–127。编程时，程序员常常因为无符号数不存在负值而用它表示数量，但在混合运算中，要尽量使用 int 类型的有符号类型，这样就无需担心无符号数的边界问题。

132. HRESULT 与 Boolean 类型强制转换

大多数 COM 函数和一些接口成员函数均使用 HRESULT 返回值类型。HRESULT 类型的成功值（S_OK）等于 0，但 Boolean 类型中的 0 表示失败。因此，如果简单的将这两种类型进行强制转换，则会产生意义上的混淆。

例 172：

```
1    #include <windows.h>
2    void Func()
3    {
4      HRESULT hrlt;
5      LPCOLESTR lpr;
6      LPCLSID lpid;
7      …
8      hrlt = CLSIDFromProgID(lpr, lpid);
```

```
9     if((bool)hrlt) // 错误
10    {
11      …//条件为真时的处理
12    }
13    else
14    {
15      …//条件为假时的处理
16    }
17  }
```

例 172 中，第 9 行语句将 hrlt 强制转换为 Boolean 类型，作为 if 语句的条件表达式，会得到错误的处理结果。应该使用 SUCCEEDED 和 FAILED 宏进行判断。

修改方法：将第 9 行语句修改为：`if(SUCCEEDED(hrlt))`

使用 HRESULT 时，不能将其强制转换为 1、-1、true、false 来使用。如果要指示成功，使用符号常数 S_OK；若要指示失败，应该使用 S_FALSE 或者 E_constant 开头的符号常数；若要指示值-1，应使用 E_FAIL。

133．显式比较位字段和 Boolean 类型

位结构是一种特殊的结构，在需按位访问一个字节或字的多个位时，位结构比按位运算符更加方便。位结构定义的一般形式为：

```
struct 位结构名{
数据类型 变量名1：整型常数;
数据类型 变量名2：整型常数;
…
} 位结构变量;
```

将 1 赋给位域会将它的单个位域保存为 1，但此位域与 1 的任意比较都包括一个将该位域强制转换为 signed int 的隐式操作。这种强制转换会将存储的 1 转换为–1，从而产生意外的结果。

例 173：

```
1   struct bit
2   {
3     int b1:1;
4     int b2:1;
5     int b3:1;
6     int b4:1;
7   }TestBit;
8
9   void Func()
```

```
10  {
11       …
12       if(TestBit.b1 == 1) // 错误
13       {
14       …
15       }
16       …
17  }
```

例 173 中，误认为布尔域和位域是等效的，因此在第 12 行语句将 b1 与 1 进行比较，根据前面的分析，此处会产生错误。如果程序试图判断 b1 是否等于 1，可以将第 12 行语句用如下两行替换：

```
TestBit.b4 = 1;
if(TestBit.b1 == TestBit.b4)
```

第2章 内存管理

C/C++程序运行时，程序分别被加载到内存中几个不同的区域，这些区域分别是代码区、数据区、BSS区、堆和栈区，如图2-1所示。

高地址	栈区	可读写，可执行
	堆区	可读写，可执行
	BSS区	可读写，不可执行
	数据区	可读写，不可执行
低地址	代码区	只读，可执行

图2-1 内存分区

代码区：用来存放可执行代码，只读，可执行。

数据区：用来存放已初始化的全局变量和静态变量，可读写，不可执行，程序结束时释放。

BSS区：用来存放未初始化的全局变量或静态变量，可读写，不可执行，程序结束时释放。

堆区：用于动态分配内存，它是先进先出(FIFO)的数据结构，数据分配从低地址向高地址方向增长，可读写，可执行，一般由程序员分配并释放。若程序员未释放，程序结束时由操作系统回收。

栈区：用来存放函数参数、返回地址等。栈的存储模式是后进先出（LIFO），其数据分配是从高地址向低地址方向递减，可读写，可执行，由编译器自动分配释放。内存中的栈区实际上是系统栈，由系统自动维护。对于类似于C语言这样的高级语言，系统栈的push、pop等堆栈平衡细节是透明的，通常情况下，只有在使用汇编语言时才需要和它直接打交道。函数调用时所建立的栈包含如下信息：函数的返回地址、调用函数的栈顶和栈底、为函数的局部变量分配的空间、为被调函数的参数分配的空间。

不同的芯片，其地址空间不同。对于当今广泛使用的32位X86芯片而言，其空间大小为4GB，地址范围为0x00000000～0xFFFFFFFF。

对于Windows系统，其地址空间结构如表2-1所示。

表 2-1　Windows 系统进程虚拟地址空间结构

分　区	X86 32 位 Windows	3GB 模式下的 X86 32 位 Windows
空指针赋值分区	0x00000000～0x0000FFFF	0x00000000～0x0000FFFF
用户模式分区	0x00010000～0x7FFEFFFF	0x00010000～0xBFFEFFFF
64KB 禁入分区	0x7FFF0000～0x7FFFFFFF	0xBFFF0000～0xBFFFFFFF
内核模式分区	0x80000000～0xFFFFFFFF	0xC0000000～0xFFFFFFFF

空指针赋值分区用于辅助程序员捕获对空指针的赋值。如果进程中的线程试图读取或写入位于这一分区内的内存地址，就会导致访问违规。

用户模式分区是用户进程的地址空间，可用的地址空间和用户模式分区的大小取决于 CPU 的体系结构。对于普通的 X86，用户模式分区的大小约为 2GB，可用的地址范围为 0x00010000～0x7FFEFFFF；对于 X86w/3GB，用户模式分区的大小约为 3GB，可用的地址范围为 0x00010000～0xBFFEFFFF。

进程无法通过指针来读、写或以任何方式访问驻留在该分区中其他进程的数据。由于每个进程都有自己的数据分区，因此，相互破坏的可能性很小。在 Windows 系统中，所有.exe 和动态链接库（DLL）都被载入到用户模式分区，每个进程都有可能将这些 DLL 载入到该分区的不同位置，系统同时把该进程可以访问的所有内存映射文件映射到这一区域。

64KB 禁入分区用于隔离用户和内核空间，防止用户程序跨越到内核空间中。

内核模式分区用于存放操作系统代码，与线程调度、内存管理、文件系统支持、网络支持以及设备驱动程序相关的代码均被载入到这一分区。虽然该分区在每个进程中用户模式分区的上方，但该分区中所有代码和数据都被完全保护起来。如果一个应用程序试图访问位于这一分区中的内存，会引发访问违规，在默认情况下，系统会显示一个消息框，然后结束该应用程序。

对于 Linux 系统，其地址空间大小为 4GB，其中，最高的 1GB（0xC0000000～0xFFFFFFFF）供内核使用，较低的 3GB（0x00000000～0xBFFFFFFF）供各个进程使用。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统的所有进程共享，从具体进程角度来看，每个进程可以拥有 4GB 的虚拟空间，但在实际使用中，内存管理器只是分配给进程一片虚拟地址空间，当需要进行实际的内存操作时，内存管理器才会把虚拟地址和物理地址联系起来。

在内存分配→使用→释放的过程中，可能发生如下错误。

（1）内存分配未成功，但却使用。例如，在从堆中动态申请内存后，许多程序总假设内存分配是成功的，但实际上，内存分配可能不成功，因此，程序必须在使用内存之前检查指针是否为 NULL 以防错误发生。

（2）内存分配虽然成功，但未初始化就引用。在 C 语言中，内存的默认初始值没有统一的标准，其值不固定，如果变量的引用处和其定义处相隔较远，很容易忘记对变量赋初值，从而造成后续引用时出现错误。

（3）内存分配成功并且已经初始化，但访问时越界。越界的内存中存储的数据处于不确定状态，

访问这些内存可能导致程序异常。

（4）未释放内存，造成内存泄漏。内存泄漏是黑客常用的攻击方法，其根本原因在于分配内存后，没有及时释放或释放不正确（详见 2.2 节“内存泄漏”）。

（5）使用释放的内存，即在内存生存期之外访问内存。例如，访问一个已经释放的堆分配单元，由于释放的内存单元归还系统后该内存可以重新分配，因此，再次访问时该内存的值不确定。造成这种情况的原因一是程序中的对象调用关系过于复杂，使程序员难以准确掌握内存的分配与释放情况；二是用 free()或 delete()释放了内存后，没有将指针设置为 NULL；三是函数的 return 语句书写不正确，在 return 语句中返回了指向栈内存的指针或引用。

（6）释放一块并非动态分配的内存。free()函数只能释放动态分配的内存，非动态分配的内存由编译器释放。

（7）释放部分动态分配的内存。动态分配的内存必须整块一起释放，不能单独释放其中的某一部分内存。

内存管理方面的错误具有以下特点。

① 编译器难以发现这类错误，通常只有在程序运行时才能显现；

② 没有明显症状，只有在特殊情况下，这些缺陷才能立即暴露；

③ 引起内存访问错误的异常条件难以再现，增加了错误定位难度。

本章将通过实例对这些错误进行详细论述。

2.1　内存分配与使用

无论是 C 或 C++语言，内存分配都只有三种方式：

（1）从静态存储区域分配。内存在程序编译时就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量、static 变量。

（2）在栈上创建。执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3）从堆上分配，也称动态内存分配。程序在运行时用 malloc()或 new()申请任意大小的内存，由程序员决定何时用 free()或 delete()释放。动态内存分配是程序设计时经常遇到的问题。例如，要输出 1 到 n 内的所有素数。通常的解决方案是声明一个指定长度的数组，但由于数组的长度 n 在运行时才能确定，因此，一般是声明一个足以容纳可能出现的最多元素个数的数组。这种方法的优点是简单，缺点是浪费内存。由于事先声明的数组很大，如果程序需要处理的元素数量较少，无疑会浪费内存空间。为了解决这个问题，C 语言中提供了动态内存分配功能。动态内存的生存期由程序员决定，使用非常灵活，但也是编程时最容易出现问题的地方。

表 2-2 给出了三种内存分配方式的比较。

表 2-2　三种内存分配方式比较

	静态存储分配	栈上分配	堆上分配
管理方式	编译器自动管理	编译器自动管理	程序员控制
空间大小	较大	较大	较小
有无碎片	无	无	有
分配方式	编译时编译器分配	运行时编译器分配	运行时动态分配
分配效率	高	高	低
分配空间是否固定	是	是	否
内存地址生长方向	向上（增加）	向下（减小）	向上（增加）

134. 指针重复释放

例 174:

```
1    struct  st
2     {
3        int  number;
4        char  *s;
5        char  scores;
6      };
7    void  release(st *t)
8      {
9        free(t);
10     }
11     int  main()
12     {
13       st  *p=(st*)malloc(sizeof(st));
14       if(p==NULL) return 0;
15       release(p);
16       free(p);  // 错误
17       return  1;
18     }
```

例 174 中，第 15 行语句通过调用 release()函数释放指针 p，被释放的指针 p 在后面被 free()函数再次释放，导致安全漏洞。

135. 条件判断语句导致指针重复释放

例 175:

```
1    #include <stdlib.h>
```

```
2   int main()
3   {
4    int i;
5    struct student
6    {
7     int num;
8     char name[20];
9     char sex;
10    int age;
11   };
12   struct student *p;
13   p=(struct student*) calloc(1,sizeof(struct student));
14    if(p==NULL)
15    return 0;
16    scanf("%d",&i);
17    if(i!=1000)
18    {
19     p->num=1000;
20     free(p);
21    }
22   else
23   {
24    p->num=i;
25   }
26   free(p);
27   return(1);
28   }
```

例 175 中，第 17 行语句当 if 语句判断条件为真时，指针 p 将被释放，被释放的指针 p 在后面被再次释放，导致安全漏洞。

136．由指针别名引起的指针重复释放

当程序中有两个或两个以上的指针表达式指向同一块内存空间时，则这些指针表达式关于该内存空间形成别名关系。

例 176：

```
1   char *p,*q;
2   p=(char*)malloc(sizeof(chae)*10);
3   strcpy(p, "ok");
```

```
4   q=p;
5   strcpy(q, "hello");
6   free(p);
7   free(q);
```

例 176 中，语句 2 为指针 p 动态分配了空间，语句 4 使指针 q 指向指针 p，这时，指针 p，q 已成为指针别名，但编译器无法识别，因此，不能发现语句 6，7 出现的指针重复释放错误。

同样，如果程序中通过对象间赋值而该对象的构造函数动态申请内存，析构函数动态释放内存；或者两个动态申请的对象，通过相互赋值导致某一对象指向的空间被释放多次；或者由于拷贝构造函数使指针成员指向的内存被释放多次等，由于编译器不进行别名检查，最终导致同一内存重复释放。

137. 函数使用已释放的指针作为参数

例 177：

```
1    #include<stlib.h>
2     struct  st
3     {
4      int  number;
5      char  *s;
6      char  scores;
7      }
8      f(st *t)
9     {
10      int  j;
11      scanf("%d",&j);
12      (t->scores)= (t->scores)+j;
13    }
14    int  main()
15    {
16       int  i;
17       st  *p;
18       st  *p=(st*)malloc(sizeof(st));
19       if(p==NULL) return 0;
20       free(p);
21       f(p); // 错误
22     }
```

例 177 中，第 20 行语句对指针 p 进行了释放，释放后的指针 p 仍被作为参数传递给函数 f()。

138. 条件判断语句导致函数返回被释放的指针

例 178：

```
1   f(char *p)
2    {
3     return p;
4    }
5   char  *main()
6   {
7    char *x=malloc(1);
8    char *y;
9    y=x;
10   if (y!=NULL)
11   free(y);
12   f(y);  //释放了内存后的指针 y 成为“野指针”
13  }
```

例 178 中第 10 行语句，当 if 语句为真时，指针 y 被释放，在这种情况下，在第 12 行语句调用函数 f()，函数 f()将返回已经被释放的指针 y，导致程序错误。

139. 未检查 new()的返回值

在使用 new()、malloc()/calloc()等内存分配函数时，一定要检查其返回值是否为空指针。如果内存分配成功，指针不为空，如果内存分配失败，默认抛出 bad_alloc 异常。因此，程序中需要捕捉此异常语句，以检查内存分配是否成功。

例 179：

```
1   #include <new>
2   void foo()
3   {
4       char *p;
5       p = new char[10*10*10];
6       p[0] = 'x';
7       delete[] p;
8       int *p;
…
9   }
```

例 179 中第 5 行语句，指针 P 通过 new()分配内存，但未检查 new()分配内存后的返回值。

修改方法：在指针 P 通过 new()分配内存后，使用捕获异常语句检查 new()分配内存的情况。修

改后的程序如例 180 所示。

例 180：

```
1   #include <new>
2   void foo()
3   {
4     char *p;
5     try{
6        p = new char[10*10*10];
7     }catch(std::bad_alloc&){}
8   p[0] = 'x';
9   delete[] p;
10    …}
```

140. 函数返回已释放的指针

例 181：

```
1   f(char *p)
2   {return p;}
3   main()
4   {
5    char *x = malloc(1);
6    free(x);
7    f(x);
8    }
```

例 181 中，第 6 行语句释放了指针 x，随后调用函数 f()，函数 f()将返回已经被释放的指针，导致程序错误。

141. 使用已释放的内存

例 182：

```
1   char *GetMemory(void)
2   {
3     char p[] = "hello world";
4     return p;
5   }
6   void Test(void)
7   {
8    char *str= NULL;
```

```
9     str = GetMemory();  // 错误
10    printf(str);
11  }
```

例 182 中，GetMemory()函数返回的是指向“栈内存”的指针，str 变量通过 GetMemory()函数将得不到字符串“hello world”。

142．条件判断语句导致使用已释放的内存

例 183：

```
1    #include<stlib.h>
2    main()
3    {
4       int   *x=malloc(4);
5       scanf("%d",&j);
6       if(j>0)
7       {
8          free(x);
9       }
10      x=&j; // 错误
11   }
```

例 183 中，当条件判断语句成立时，指针 x 将被释放（其指向的内存被释放），第 10 行语句给 x 赋值将产生使用释放内存的错误。

143．使用 delete[]删除单个对象的内存空间

C++语言中，通过 delete()函数回收 new()函数为单个对象分配的内存空间，用 delete[]函数回收 new[]分配的数组空间。

例 184：

```
1   class A {
2   public:
3      A( ) {}
4   };
5
6   void foo( ) {
7      A *a = new A;
8      delete[] a; // 错误
9   }
```

例 184 中，a 是使用 new()为单个对象分配的内存空间，在第 8 行语句中应该使用语句“delete a;”释放该内存空间。

144．使用 delete 删除已分配的数组空间

C++语言中一定要配对使用不同形式的 new 和 delete 类函数。

例 185：

```
1   class A
2   {
3      public:
4      A() {}
5   };
6   void foo() {
7       A *a = new A[100];
8       delete a;    // 错误
9   }
```

例 185 中，a 是由 new[]申请生成的具有 100 个对象的数组空间，删除时应该使用与 new[]配对的 delete[]将其删除。

修改方法：将第 8 行语句修改为：delete[] a;

145．条件判断语句导致释放非堆内存

例 186：

```
1   #include<stlib.h>
2   main()
3   {
4     const int  i=1;
5     int j;
6     int *p;
7     p=&i; //p 是指向常量静态内存的指针
8     scanf("%d",&j);
9     if(j>0)
10    {
11      free(p);
12    }
13  }
```

例 186 中第 11 行语句，当 if 条件成立时，程序试图通过指针 p 释放常量内存。

146. 释放静态内存

例 187：

```
1   main()
2   {
3     const int  i=1;
4     int *p;
5     p=&i;
6     free(p); // 错误
7   }
```

例 187 中第 6 行语句，程序通过指针释放常量内存。

147. 释放未分配的内存

例 188：

```
1   main()
2   {
3      void *p=malloc(10);
4       p+=10;
5      free(p); // 错误
6   }
```

例 188 中，第 5 行语句试图释放未分配的内存。

148. 条件判断语句导致释放未分配的内存

例 189：

```
1   #include<stlib.h>
2   main()
3   {
4     int  j;
5     scanf("%d",&j);
6     void   *p=malloc(j); //p 指向 j 个字节大小的存储区
7     if(j>5)
8      {
9        p+=10; //p 指针向后移 10 个字节
10     }
11  free(p);
12  }
```

例 189 中，当 if 条件判断语句成立时，第 11 行语句将释放未分配的内存。

149. 释放空指针

例 190：

```
1    int main()
2    {
3      char *x;
4      x = NULL;
5      free(x); // 错误
6      return 0;
7    }
```

例 190 中，第 5 行语句释放空指针 x。

150. 条件判断语句导致释放空指针

例 191：

```
1    #include<stlib.h>
2    int main()
3    {
4     int  t;
5     scanf("%d",&t);
6     char *x;
7     x = NULL;
8     if(t>0)
9     free(x); // 错误
10    return 0;
11   }
```

例 191 中，当第 8 行语句中的 if 条件成立时，程序将在第 9 行语句释放空指针 x。

2.2 内存泄漏

内存泄漏（Memory Leaks）是指程序运行时动态分配的内存在程序结束时没有被释放，从而造成该部分内存不可用。程序运行时持续分配内存，如果分配的内存在使用完后没有被释放，系统内存资源将逐渐耗尽，导致系统崩溃。

对于规模较小的程序，内存泄漏并不会导致明显的错误，但对于规模较大的程序或需要连续运行的程序，内存泄漏将最终导致程序因耗尽系统所有内存，无法再进行分配而崩溃。

内存泄漏具有隐蔽性、累积性特征，其症状要通过逐渐积累才能显现。因此，内存泄漏容易被忽略，同时也难以被发现和再现，与其他内存非法访问缺陷相比更加难以检测。

通常将内存泄漏分为两类，一类是物理泄漏，另一类是逻辑泄漏。物理泄漏是指应用程序动态分配内存后，用于指向该内存的指针被释放或用于其他用途，导致程序无法再次访问或释放该内存空间；逻辑泄漏是指应用程序动态分配内存并使用完后，一直未释放该内存，但程序仍然可以访问。

更进一步，还可以将内存泄漏细分为如下 4 类。

① 常发性内存泄漏。造成内存泄漏的代码被多次执行，每次执行都会引起一块内存泄漏；

② 偶发性内存泄漏。造成内存泄漏的代码只有在特定条件下才被执行，代码被执行时会引起内存泄漏。常发性和偶发性是相对的，在一定条件下，偶发性会转变成常发性；

③ 一次性内存泄漏。造成内存泄漏的代码只被执行一次，或者由于算法上的缺陷，导致总会有一块且仅有一块内存发生泄漏；

④ 隐式内存泄漏。程序在运行过程中不断地分配内存，直到结束时才释放。严格来说，这里并没有发生内存泄漏，因为程序最终释放了所分配的内存，但是对于一个需要长时间连续不断运行的程序而言，如果不及时释放这些内存，或由于程序缺陷未能释放这些内存，最终将耗尽系统所有的内存。

造成内存泄漏的原因主要有如下三个。

① 未释放。申请的内存没有被释放；

② 重复释放。申请的内存被多次释放；

③ 释放不彻底。释放的内存空间和申请的内存空间大小不相同。

151. 内存分配和释放函数不匹配

内存分配和释放函数正确的匹配是 malloc()/free()、new()/delete()和 new[]/delete[]（用于分配/释放对象数组）。

C 语言和 C++语言都提供了动态内存分配方法，C 语言提供的是库函数：malloc()、realloc()、alloca()、calloc()、sbrk()、free()等，而 C++语言提供的是运算符：new()、new[]、delete()、delete[]等。

其中，malloc()分配的内存是位于堆中的，并且没有初始化内存的内容，因此，基本上使用 malloc()分配内存之后，都会调用函数 memset()来初始化这部分的内存空间；realloc()则对 malloc()申请的内存大小进行调整；alloca()是向栈申请内存，因此无需释放；calloc()分配的内存位于堆中并初始化其内存（将其设置为 0），申请的内存最终需要通过函数 free()来释放；而 sbrk()则是增加数据段的大小。

malloc()/calloc()/free()基本上都是 C 函数库实现的，与操作系统无关。C 函数库内部通过一定的结构来保存当前有多少可用内存。如果程序使用 malloc()分配的内存大小超出库里所留存的空间，那么将首先调用 sbrk()增加可用空间，然后再分配空间。使用 free()释放的内存并不立即返回给操作系统，而是保留在内部结构中。sbrk()类似于批发，一次性向操作系统申请大块的内存，而 malloc()等函数则类似于零售，分配的内存只要满足程序运行时的要求即可，这套机制类似于缓冲。之所以使

用这套机制，是因为系统调用不能支持任意大小的内存分配（有的系统调用只支持固定大小以及其倍数的内存申请），因此，对于小内存的分配会造成浪费，系统调用申请内存代价昂贵，涉及用户态和核心态的转换。

对于 malloc()/free()这类库函数而言，它们只负责分配或释放一块连续的内存空间，不执行构造/析构函数；对于 new()/delete()这类运算符而言，它们在分配内存空间的同时自动执行了构造/析构函数，主要用于非基本数据类型对象的操作。由于基本数据类型的“对象”没有构造与析构的过程，对它们而言，malloc()/free()和 new()/delete()是等价的。

既然 new()/delete()的功能完全覆盖了 malloc()/free()，为什么 C++语言中还要保留 malloc()/free()呢？这是因为 C++程序经常要调用 C 函数，而 C 程序只能用 malloc()/free()管理动态内存。

由于 C++编译器既能够识别函数，也能够识别运算符，因此，在编程时很容易混淆，造成程序运行错误。此外，由于是动态分配内存，程序员难以确定分配的空间次数是否与释放的次数相等，而编译器也不对这些问题进行检查，因此，容易出现对同一空间释放多次或使用已释放的空间等安全漏洞。

如果用 free()释放 new()创建的动态对象，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete()释放 malloc()申请的动态内存，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc()/free()也一样。

函数 malloc()和 calloc()都可以用来分配动态内存空间，但两者稍有区别。

malloc()函数有一个参数，即要分配的内存空间的大小，函数原型为：

```
void *malloc(size_t size);
```

calloc()函数有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小，函数原型为：

```
void *calloc(size_t numElements, size_t sizeOfElement);
```

如果调用成功，函数 malloc()和 calloc()都将返回所分配内存空间的首地址。

malloc() 函数和 calloc()函数的主要区别是前者不能初始化所分配的内存空间。如果由 malloc()函数分配的内存空间原来没有被使用过，则其中的每一位可能都是 0；反之，如果这部分内存空间曾经被分配、释放和重新分配，则其中可能遗留各种各样的数据，这也就是使用 malloc()函数的程序开始时（内存空间还没有被重新分配）能正常运行，但经过一段时间后（内存空间已被重新分配）可能会出现问题的原因所在。

calloc() 函数会将所分配内存空间中的每一位都初始化为 0，也就是说，如果为字符类型或整数类型的元素分配内存，那么这些元素将保证会被初始化为 0；如果为指针类型的元素分配内存，那么这些元素通常（但无法保证）会被初始化为空指针；如果为实数类型的元素分配内存，那么这些元素可能（只在某些计算机中）会被初始化为浮点型的 0。

malloc() 函数和 calloc()函数的另一点区别是：calloc()函数会返回一个由某种对象组成的数组，但 malloc()函数只返回一个对象。为了明确是为一个数组分配内存空间，有些程序员会选用 calloc()

函数。但是，除了是否初始化所分配的内存空间这一区别之外，绝大多数程序员认为以下两种函数的调用方式没有区别：

```
calloc(numElements, sizeOfElement);
malloc(numElements *sizeOfElement);
```

需要解释的一点是，理论上（按照 ANSI C 标准），指针的算术运算只能在一个指定的数组中进行，但是在实践中，即使 C 编译程序或翻译器遵循这种规定，许多 C 程序还是冲破了这种限制。因此，尽管 malloc()函数并不能返回一个数组，它所分配的内存空间仍然能供一个数组使用（对 realloc()函数来说同样如此，尽管它也不能返回一个数组）。

总之，当在 calloc()函数和 malloc()函数之间做选择时，只需考虑是否要初始化所分配的内存空间，而不需要考虑函数是否能返回一个数组。

例 192：

```
1    #include<stlib.h>
2    main()
3    {
4     int j;
5     int *p;
6     scanf("%d",&j);
7     p=(int*)malloc(sizeof(int));
8     if(j>0)
9     {
10      delete(p); // 错误
11      p=NULL;
12     }
13   if(p!=NULL)
14   free(p);
15   }
```

例 192 中，第 7 行语句使用 malloc()函数为指针 p 申请内存空间，当第 8 行语句 if 条件成立时，第 10 行语句使用与 malloc()函数不匹配的 delete()函数释放指针 p 的内存空间。

例 193：

```
1    main()
2    {
3     int *p;
4     p=(int*)malloc(sizeof(int));
5     delete p; // 错误
6    }
```

例 193 中，第 4 行语句使用 malloc()函数为指针 p 申请内存空间，在第 5 行语句使用与 malloc()函数不匹配的 delete()函数释放指针 p 的空间。

152. 内存申请和释放函数不匹配

类中需要成对使用 operator new()和 operator delete()，operator new[]和 operator delete[]以及 new 和 delete。

例 194：

```
1   #include <stdio.h>
2   class A
3   {
4   public:
5       A() {}
6       void* operator new(size_t size);
7   };
```

例 194 中，类 A 使用 operator new()申请内存，在该类中并没有使用 operator delete()释放申请的内存。

修改方法：在第 6 行语句后面增加语句：

```
void operator delete( void* );
```

153. 构造和析构函数中的 new()/delete()函数不匹配

类中构造函数和析构函数应该成对使用 new()和 delete()函数为某个成员分配内存和释放为其分配的内存。

例 195：

```
1   class A
2    {
3      public:
4         A( )
5         {
6            p1 = new char;
7            p2 = new char;
8         }
9         ~A( )
10        {
11           delete p1;
12        }
13     private:
14        char *p1;
```

```
15    char *p2;
16  };
```

例 195 中，类 A 的构造函数使用 new()函数为 p1 和 p2 分配内存，但在类的析构函数中没有释放为 p2 分配的内存。

修改方法：在类的析构函数中增加释放 p2 内存的语句：delete p2;

154. 连续内存空间释放顺序错误

malloc()或 new[]都是用来动态申请一块连续的空间，包括基本类型和对象类型。释放时必须对空间整体释放，而不能单独释放某个空间，即释放时指向连续空间的指针必须指向连续空间的首地址，否则，该空间不能正确释放。

例 196：

```
1   int *p=(int*)malloc(sizeof(int*10);
2    while (i<10)
3        {
4        *p=i*i;
5        i++;
6        p++;
7        }
8     free(p);
```

例 196 中，释放指针时指针 p 没有返回到申请的内存的首部，因此，指针 p 不能被正确释放。

155. 多维数组释放顺序错误

对于多维连续内存空间，在分配时先申请大范围空间，再申请小范围空间，释放时则刚好相反，应先释放小范围空间，再释放大范围空间。

例 197：

```
1   typedef struct forest{
2           int tnum;
3           tree *t;//tree 是结构体
4           }forest;
5       forest *f=new forest[2];
6           {
7           (f+i->tnum=i;
8           (f+i->t=new tree[(f+i)->tnum];
9           }
10      delete f;
```

例 197 中，在没有释放 forest 结构体内的指针 t 时先释放了 f，这样，为 t 分配的空间将被泄漏。

156．函数参数列表中使用 new()函数

内存资源应该属于某个对象，如果程序中申请的内存没有分配给某个对象，这块内存将会丢失。

例 198：

```
1   class  C
2   {
3   int  *p;
4   }
5   F(C  c1)
6   {}
7   main()
8   {
9      …
10   F(new  int); // 错误
11     …
12  }
```

例 198 中，F()函数的参数是类 C，第 10 行语句中内存分配函数 new()出现在 F()函数参数列表中，而此时内存分配函数 new()申请的内存资源并不属于类的某一个对象，这块内存资源将会丢失。

修改方法：先产生一个对象，再为该对象分配内存资源，对应的语句为：`c1=newC();F(c1);`

157．realloc()函数使用不当

realloc()函数原型为：

```
extern void *realloc(void *mem_address, unsigned int newsize);
```

该函数先按照 newsize 指定的大小分配空间，将原有数据从头到尾复制到新分配的内存区域，而后释放原来 mem_address 所指内存区域，同时返回新分配的内存区域的首地址，即重新分配的存储块的地址。如果重新分配成功则返回指向被分配内存的指针，否则返回空指针 NULL。

例 199：

```
1    #include <malloc.h>
2    #include <windows.h>
3
4    void Func( )
5    {
6     char *source;
7     source = (char *) malloc(128);
```

```
8     if (source != NULL)
9     {
10      source = (char *) realloc(source, 512);
11      …
12      free(source);
13    }
14  }
```

例 199 中，第 10 行语句调用 realloc()为 source 重新分配内存，如果重新分配失败（realloc()返回 NULL），则第 12 行语句 free(source)无法释放原始内存块（第 7 行语句 malloc()函数所分配的内存）。

修改方法：可以将 realloc()的结果赋给一个临时变量，如果分配成功再使用临时变量更新原始指针。

158．使用 malloc()或 realloc()函数构建对象

new()、malloc()和 realloc() 函数都是内存分配函数，new()在申请分配一块内存的同时创建一个对象，而 malloc()和 realloc() 仅仅申请分配一块内存，不产生对象。

例 200：

```
1   #include <stdlib.h>
2   class A
3   {
4    public:
5      A();
6   };
7    void f()
8   {
9     A *a = (A*) malloc( sizeof( A ) );  // 错误
10    b = (A*) malloc( sizeof( A ) );  // 错误
11  }
```

例 200 中，第 9 行和第 10 行语句中的 malloc()函数申请的内存不属于任何对象，所以是没有意义的。

修改方法：使用 new()函数“`A* a = new A( ) ; a = (A*)malloc(sizeof(A));`”，在申请内存的同时创建对象，使得申请的内存资源属于对象。

159．返回局部变量的指针或引用

函数返回局部变量的指针或引用时，局部变量将在函数返回后被销毁，因此返回的引用或指针就处于“无所指”的状态，从而使程序进入未知状态。

例 201：

```
1   #include <iostream.h>
2   float f1(float r)
3   {
4     float temp;
5     temp=(float)(r*r*3.14);
6     return temp;
7   }
8   main()
9   {
10    float &b=f1(10.0); // 错误
11    …
12  }
```

例 201 中，第 10 行语句返回的是 f1()函数中变量 temp 的引用，而该引用指向的 temp 是一个局部变量，在函数返回时该变量被销毁。

修改方法：将第 10 行语句修改为：float b=f1(10.0);

例 202：

```
1   int* f ()
2   {
3     int i;
4     return &i;
5   }
6   void  main()
7    {
8     int  *p;
9     p=f(); // 错误
10  }
```

例 202 中，f()函数返回指针类型值，在第 9 行语句，把函数 f()中局部变量 i 的地址赋给指针 p，而该地址在函数 f()返回后就被销毁，指针 p 进入无所指的状态。

160. 函数返回 new()分配的内存的引用

函数创建新对象仅有两种方法：在栈上或者在堆上。一般情况下，栈上的对象需要通过定义局部变量得到，堆上的对象需要通过 new()函数产生。例 202 中介绍了函数不能返回局部变量的引用或地址的原因，下面主要说明为什么不能返回函数内部由 new()函数分配的内存的引用。

例 203：

```
1    #include <iostream>
2    #include <string>
3    using namespace std;
4    string& f()
5    {
6      string *s = new string("string");
7      return *s;
8    }
9    int main()
10   {
11     string& a=f();
12     cout<<a;
13   }
```

例 203 中，s 是函数 f()中的局部变量，但由 new()创建的*s 不是局部变量，在函数返回时无法被销毁。这个由 new()申请的堆变量，如果不用 delete()释放，这块堆内存就无法释放。例 203 中的函数 f()如果经常被调用，可以观察到程序占用的内存数会一直增长，进而造成内存泄漏。

同样，返回函数内部由 new()分配的内存的引用，虽然不存在局部变量的被动销毁问题，但是被函数返回的引用所指向的空间（由 new()分配）却无法释放，从而造成内存泄漏。

161．条件判断语句导致已分配的内存未释放

例 204：

```
1    void  f (int i)
2    {
3      int  *p=(int *)malloc(12);
4      if(i)
5      {
6        p=NULL;
7      }
8     return  1;
9    }
```

例 204 中，函数 f()为指针 p 申请了内存空间，第 4 行语句当 if 条件成立时，函数 f()通过将指针赋值为空的方式释放指针，但是当 if 条件不成立时，由于指针没有释放，将导致内存泄漏。

162．内存空间二次分配

例 205：

```
1   main()
2   {
3    int  *p=new int (10) ;
4    *p=25;
5    p=new int;  //错误
6    *p=35;
7   }
```

例 205 中，第 3 行语句为指针 p 申请了动态内存空间，但在第 5 行语句中再次为 p 分配动态内存，产生内存泄漏。

163．new operator 与 operator new 误用

new operator 和 operator new 看上去相似，但两者有着很大差别。new operator 在使用时完成申请内存和初始化对象两件事，例如：`string* ps = new string("abc");` 而 operator new 类似于 C 语言中的 malloc()，只是负责申请内存，例如：`void* buffer=operator new(sizeof(string));`

operator new 可以被重载，重载时其返回类型必须声明为 void*，operator new 在重载时可以带其它参数，但其重载函数的第一个参数类型必须为需要分配空间的大小，类型为 size_t。new operator 在 C++语言中不能被重载。

相应地，operator delete 与 delete operator 有相似的特性。

例 206：

```
1   class A
2   {
3   public:
4     …
5       static void* operator new(size_t size)
6     {
7       return ::operator new(size);
8     }
9     …
10  };
11  main()
12  {
13   A *p = operator new A();// 错误
14    …
```

```
15  }
```

例 206 中第 13 行语句，operator new 应为 new operator，它将调用类 A 中的 operator new 为该类的对象分配空间，再调用当前实例的构造函数创建类 A 的对象。

164. 派生类的赋值函数未给基类成员赋值

派生类的赋值函数必须给所有的成员包括基类成员赋值。

例 207：

```
1   class Base
2   {
3     public:
4     …
5     Base & operate =(const Base &other);
6     private:
7     int m_i, m_j, m_k;
8   };
9   class Derived : public Base
10  {
11    public:
12    …
13    Derived & operate =(const Derived &other);//派生类的赋值函数
14    private:
15    int m_x, m_y, m_z;
16  };
17  Derived & Derived::operate =(const Derived &other)
18   {
19    if(this == &other)         //检查自赋值
20      return *this;
21    m_x = other.m_x;  }
22    m_y = other.m_y;  } //给自身的数据成员赋值
23    m_z = other.m_z;  }
24      return *this;
25  }
```

例 207 中，派生类 Derived 的赋值函数完成了对自身数据成员的赋值，但没有对基类的数据成员进行赋值。

修改办法：在第 20 行语句后面增加对基类数据成员的赋值语句：`Base::operate=(other);`

165. 被 delete 删除的指针未赋空值

指针被删除表示其所指向的内存空间被释放，并不代表指针本身被删除，此时，被删除的指针仍指向某一块未知的内存空间。程序中如果引用这种指针，可能会出现难以预料的结果。为避免这种情况发生，应将用 delete 删除的指针赋值为空（NULL 或 0）。

例 208：

```
1    int main(void)
2    {
3       int *p=0;
4       p=new int;
5       *p=100;
6       delete p;
7       return 0;
8    }
```

例 208 中第 6 行语句，调用 delete()释放指针 p 所指向的内存空间后，指针 p 需要被赋值为空。

修改方法：在第 6 行语句后面增加语句：`p =0;`

166. 使用默认的拷贝和赋值构造函数初始化指针变量

C++语言中，当类没有编写其拷贝函数和赋值函数时，编译器在程序编译阶段会自动为类生成默认的拷贝函数和赋值函数，但这种默认的拷贝函数和赋值函数在执行拷贝或赋值操作时，只完成“位复制”而非“值复制”。对于指针变量而言，“位复制”相当于两个指针指向同一块内存区域，一旦该内存区域被释放，那么所有指向这个内存区的指针都会无效；“值复制”是另外再分配一块内存区域，并且复制内容，如果某个指针指向的内存区域被释放，另一个指针不会受影响。

例 209：

```
1    class A
2    {
3      int  *p;
4      public:
5    A(int i){p=new int[i];}
6    };
7    main()
8    {
9      A  a(5);
10     A  b(4);
11     A  c(a);      //调用拷贝函数
12     a=b;          //调用赋值函数
```

```
13    return 0;
14    }
```

例 209 中，类 A 中包含一个指针型变量 p，第 9 行语句和第 10 行语句创建类 A 的对象 a 和 b，并分别为其成员 p 分配内存空间。

本例中第 11 行语句调用默认的拷贝函数（编译器为其自动生成默认的拷贝函数），复制对象 a 给对象 c，使得 a 和 c 的指针 p 指向同一块内存，一旦 a 或 c 的指针 p 被释放，另一个变量的 p 指针指向将无效。

本例中第 12 行语句调用默认的赋值函数，使得 b 的指针 p 指向 a 的指针 p 的内存，导致 b 的指针 p 原来指向的内存空间丢失，造成内存泄漏。

当类中含有指针变量时，编译器调用默认的拷贝构造函数和赋值函数，以“位复制”的方式进行复制和赋值操作，这将给程序带来隐患。

对于任意一个类，如果没有构造函数、析构函数、拷贝函数和赋值函数，C++编译器将自动为类生成如下四个默认的函数：

```
A(void);                          // 默认的无参数构造函数
A(const A &a);                    // 默认的拷贝构造函数
~A(void);                         // 默认的析构函数
A & operate =(const A &a);        // 默认的赋值函数
```

167. 显式调用析构函数释放对象资源

析构函数与类同名但在其前面加上了一个“~”符。在对象被删除或超出作用域时，程序会自动调用类的析构函数完成对象资源的释放。程序中显式调用析构函数时，析构函数只是清除对象本身，并没有释放对象所占用的内存。

例 210：

```
1   class A
2   {
3   public:
4       ~A();
5   };
6   void f()
7   {
8       A a;
9       a.~A();    // 错误
10  }
```

例 210 中，第 9 行语句显式调用析构函数释放对象资源。

168. 未释放分配给 hostent 结构体的内存

在使用 hostent 结构体时，通过 getipnodebyname()或 getipnodebyaddr()函数为 hostent 结构体分配内存；通过 freehostent()函数释放为 hostent 结构体分配的内存。

内存分配函数原型为：

```
struct hostent *getipnodebyname(const char *name, int af, int flags, int *error_num);

struct hostent *getipnodebyaddr(const void *addr, size_t len, int af, int *error_num);
```

内存释放函数原型为：

```
void freehostent(struct hostent *ip);
```

hostent 数据结构为：

```
struct hostent
{
    char*   h_name;          //地址的正式名称
    char**  h_aliases;       //空字节-地址的预备名称的指针
    int     h_addrtype;      //地址类型，通常是AF_INET
    int     h_length;        //地址的比特长度
    char**  h_addr_list;     //零字节-主机网络地址指针，网络字节顺序
}
```

在 hostent 结构体使用完毕后，如果没有动态释放为该结构体分配的内存，会造成内存泄漏。

例 211：

```
1    static void allocationLeak()
2    {
3      …
4    struct hostent *p = getipnodebyname("name", 7, 7, 0);
5      …
7    }
```

例 211 中，使用 getipnodebyname()函数为 hostent 结构体申请了内存空间，在程序最后应该调用 freehostent()函数释放为 hostent 结构体申请的内存空间。

169. 未释放 regcomp()函数分配的内存

C++语言使用 regcomp()函数编译正则表达式，当不再需要已经编译过的正则表达式时，应该调用函数 regfree()将其释放，否则会产生内存泄漏。

regfree()函数原型为：

```
unsigned int regcomp(regex_t *preg, const char *regex, int cflags);
void regfree(regex_t *preg);
```

例 212：

```
1   static void allocationLeak()
2   {
3       regex_t *p;
4       regcomp(p, "regex", 7);
5   }
```

例 212 中，使用 regcomp()函数申请内存空间，使用完毕后未调用 regfree() 函数释放申请的内存空间，因而发生内存泄露。

修改方法：在第 4 行语句后面增加语句：`regfree(p);`

170．VirtualFree()函数参数使用错误

进程的虚拟地址空间内存页面存在三种状态，分别是：空闲（free）、保留（reserved）和提交（committed）。

空闲页面。进程不能访问此类页面，因为此类页面还没有被分配，对任何属于这种页面的虚拟内存地址进行访问都将引发访问异常；

保留页面。页面被保留以备将来之用。此类页面已经被分配，但是还没有使用，物理地址空间中不存在与其对应的物理内存分页。处于保留状态的内存分页也不能被访问；

提交页面。内存已经被分配，并且已经被使用，具有与之对应的物理地址空间中的内存分页。

在实际编程中，常使用 VirtualAlloc()和 VirtualFree()两个函数来申请和释放内存。前者功能是保留或提交当前进程(calling process)的内存页面，即将空闲的内存页面变为保留的或提交的、将保留的页面变为提交的；后者是将当前进程内存状态从提交变为保留，或将保留变为空闲，或同时进行。这里要讨论的是 VirtualFree()函数，其函数声明如下：

```
BOOL VirtualFree(
  LPVOID lpAddress,     //需要改变状态的内存区域的起始地址
  SIZE_T dwSize,        //需要改变状态的内存区域的字节大小
  DWORD dwFreeType      //释放类型
);
```

其中重点关注第三个参数——释放类型，它可以取两个值：MEM_DECOMMIT（将内存变为保留状态）和 MEM_RELEASE（释放内存，将内存变为空闲状态）。如果使用 VirtualFree()来释放内存，但却将其中的第三个参数设置的 MEM_DECOMMIT，程序只是将目标内存变为保留状态，从而造成内存泄漏，如例 213 所示。

例 213：

```
1   …
2
3   void FUNC( )
4   {
5     LPVOID lpAddress;  // 分配基址
6     …
7     lpAddress = VirtualAlloc(
8                     NULL,
9                     SizeOfPage,
10                    MEM_RESERVE,
11                    PAGE_ READWRITE);
12    …
13    if (lpAddress != NULL)
14    {
15      //发生内存泄漏
16      if (VirtualFree(lpAddress, 0, MEM_DECOMMIT ))
17      {
18        …
19      }
20      else
21      {
22        puts("Memory free failed");
23      }
24    }
25    else
26    {
27      puts("Memory allocation failed! ");
28    }
29  }
```

171．使用 TerminateThread()终止线程

Windows API 中明确指出，TerminateThread()是一个非常危险的函数，只应当在最极端的情况下使用。因为使用 TerminateThread()终止线程时，不允许进行相应的线程资源清理，有可能造成内存泄漏。

例 214：

```
1   DWORD __stdcall newthread(void*)
2   {
3     while( true )
4     {
```

```
5        char *ptr = new char[1024];
6        …
7        delete ptr;
8     }
9  }
10
11 int Func(int argc, _TCHAR* argv[])
12 {
13   HANDLE h = CreateThread(NULL, 0, newthread, NULL, 0, NULL);
14     sleep(1000);
15     TerminateThread(h, 0);
16     h = NULL;
17
18     char *lock = new char[1024]; //这里会发生死锁
19
20     delete lock;
21     return 0;
22 }
```

例 214 中，在第 18 行语句处会发生死锁。因为 TerminateThread()不会释放相应的资源，如果一个线程在占用该锁时被杀死，在创建线程时分配的内存并没有释放，则在下一次进行内存分配时就会发生死锁。为防止这种错误，应使用下面的方法正确终止线程：

- 使用 CreateEvent()函数创建一个事件对象；
- 创建多个线程；
- 每个线程都通过调用 WaitForSingleObject() 函数来监视事件状态；
- 设置事件为终止状态（WaitForSingleObject()返回 WAIT_OBJECT_0），自行终止执行每个线程。

172. 程序异常导致内存未释放

程序设计中常常要用到动态内存的分配与释放，如果没有正确处理程序运行过程中发生的异常，内存释放操作将不能正常完成，从而导致内存泄漏。

例 215：

```
1   #define MAX SIZE 1024
2   void Func( )
3   {
4     int *p1 = new int[MAX_SIZE];
5     int *p2 = new int[MAX_SIZE];
6     …
7     delete p1;
```

```
8      delete p2;
9    }
```

例 215 中，如果程序在第 6 行语句所代表的代码处发生异常，程序将直接跳出，导致第 7 行和第 8 行语句的内存释放语句不会执行，从而导致内存泄漏。为避免此类问题，应使用异常处理程序，如例 216 所示。

例 216：

```
1    #define MAX_SIZE 1024
2    void Func( )
3    {
4      int *p1=NULL;
5      int *p2=NULL;
6
7      try
8      {
9        p1 = new int[MAX_SIZE];
10       p2 = new int[MAX_SIZE];
11       …
12       delete  p1;
13       delete p2;
14     }
15     catch (bad_alloc &ba)
16     {
17       if (NULL != p1)
18         delete  p1;
19       if (NULL !=p2)
20         delete  p2;
21     }
22     …
23   }
```

173. 在函数参数表中分配资源

为了确保所有的资源均属于其相应的对象，在资源声明的语句中显式进行资源分配后，需要立即将该资源赋予其管理的对象，否则，由于函数参数的处理顺序未定义，将导致资源泄漏。

例 217：

```
1    #include <boost/shared_ptr.hpp>
2    using boost::shared_ptr;
3    void Fun2( shared_ptr<int> p1, shared_ptr<int> p2 );
```

```
4  void goo( ) {
5  Fun2(shared_ptr<int>(new int), shared_ptr<int>( new int) );
6  }
```

例 217 中第 5 行语句是不安全的。在某些情况下，编译器可以交换 Fun2 函数参数表中构建参数的两个语句的执行顺序：即由 new()运算符调用的内存分配可以先为两个对象 P1 和 P2 分配内存，然后再调用两对象的构造函数，如果两者之一的构造函数抛出了一个异常，则另一个对象的内存将不会被释放掉，这样就很有可能造成泄漏。修改后的程序如例 218 所示。

例 218：

```
1    #include <boost/shared_ptr.hpp>
2    using boost::shared_ptr;
3    void Fun2( shared_ptr<int> p1, shared_ptr<int> p2 );
4    void goo( ) {
5      shared_ptr<int> p1(new int);
6      shared_ptr<int> p2(new int);
7      Fun2(p1, p2);
8    }
```

174．局部变量的地址赋值给全局变量、静态变量或者函数的返回值

程序中如果将局部变量的地址赋值给另一个大范围的局部变量或者静态变量，或者从函数中返回，那么包含这个地址的变量的生存周期可能超过最初变量的生存周期，使程序运行状态不可控。

例 219：

```
1    int *g;
2    int *f () {
3    int i;
4    static int *s;
5    s = &i;
6    g = &i;
7    return &i;
8    }
```

例 219 中第 5 行语句，将 f()函数内的局部变量 i 的引用赋值给静态变量 s；第 6 行语句将局部变量 i 的引用赋值给较大范围的局部变量 g；第 7 行语句将局部变量 i 的引用作为函数的返回值。

175．在循环内调用_alloca()造成堆栈溢出

函数_alloca() 从堆栈分配内存。函数原型为：

```
void *_alloca( size_t size );
```

_alloca()分配内存后，只有当调用函数退出时，该内存才被释放。堆栈是有限的，如果无法提交堆栈页，会导致堆栈溢出。

_resetstkoflw 函数可以将系统从堆栈溢出的状态恢复为正常，从而使程序得以继续运行，而不会由于出现异常错误而失败。如果未调用 _resetstkoflw 函数，则在上一个异常后不会显示保护页，当下次发生堆栈溢出时，不会显示异常，进程将在没有任何警告的情况下终止。

例 220：

```
1    #include <windows.h>
2    #include <malloc.h>
3    #include <excpt.h>
4    #include <stdio.h>
5
6    #define MAX_SIZE 100
7
8    void Func ( int size )
9    {
10     char *str;
11
12     for(int i = 0; i < MAX_SIZE; i++)
13     {
14       str = (char *)_alloca(size);
15       …
16     }
17   }
```

例 220 中，如果分配大小或迭代次数是未知的，应当避免在循环内部调用_alloca()。否则有可能造成内存资源耗尽。修改后的程序如例 221 所示。

例 221：

```
1    #include <windows.h>
2    #define MAX_SIZE 100
3
4    void Func( int size )
5    {
6     char *str;
7
8     for(int i = 0; i < MAX_SIZE; i++)
9     {
10      str = (char *) malloc(size);
11      if (str != NULL)
```

```
12        {
13          …
14          free(str);
15        }
16      }
17    }
```

第3章

缓冲区溢出

缓冲区是程序运行时在内存中开辟的临时存放数据的一块空间，缓冲区溢出是指数据被添加到分配给该缓冲区内存块之外的地方。缓冲区溢出可能造成两个风险，一是与该缓冲区相邻的内存空间很容易被覆盖，如果该空间存储的信息非常关键，可能造成非常严重的安全隐患；二是通过覆盖运行栈中函数返回地址，攻击者可以引诱程序执行恶意代码，从而严重影响程序的安全性。

下面以例 222 中的程序为例加以说明。

例 222：

```
1    #include <string.h>
2    void f(char *p)
3    {
4            char a[4];
5            strcpy(a,p);
6    }
7    int main(int argc,char *argv[])
8    {
9            f(argv[1]);
10           return 0;
11   }
```

例 222 中程序的栈结构为：

a[0]	a[1]	a[2]	a[3]	EBP	EIP	P

一旦输入的字符串长度大于 4，那么将会更改数组 a 后面内存单元的内容。例如，输入 abcdefghijklmno 后，其栈结构为：

a	b	c	d	efgh	ijkl	mno\0

其中，EBP、EIP 和 P 的内容均被修改了，EIP 的内容一旦被修改，函数返回后将指向一个无效的返回地址，导致程序错误。特别地，如果精心构造一个字符串，使修改后的 EIP 为某段程序的入口地址，这样，就可以控制程序的执行，从而造成意想不到的后果。

造成缓冲区溢出的根本原因在于某些高级语言，如 C、C++等固有的缺陷。对语言本身而言，效率是这类语言的重要设计目标之一，为了达到这个目标，在很多方面放开了限制。以 C 语言为例，C 语言允许直接访问内存，而编译器开辟的内存缓冲区常常临近重要的数据结构，如果某个函数的堆栈紧接在内存缓冲区之后，攻击者就可以很容易利用溢出攻击修改这些重要的数据。此外，C 语言编译器不检查数组和缓冲区的边界，而 C++语言为了与 C 语言兼容，也不提供该项功能，因此，一旦访问边界外的内存，将导致程序错误。

在 C/C++语言中，常见的缓冲区溢出包括以下几种类型。

（1）数组越界

这类缺陷主要发生在显式地使用数组元素，包括基本类型和用户自定义类型的数组操作，也包括像 vector 之类的标准库操作引起的越界；另一类缺陷发生在隐式地使用数组元素，如通过指针访问数组。

（2）字符串操作溢出

C/C++语言提供了很多字符串库函数，通过这些库函数，可以方便地实现格式化数据等操作。其中的格式化信息是通过某种字符串来描述的，如 C 语言中的%d、%s 等，称这些字符串为格式化字符串。实际上，格式化字符串是用一种功能有限的数据处理语言来描述的，其功能的有限性导致这些库函数存在固有的不足，如果使用不当，将产生缓冲区溢出。

（3）数值型变量表示范围越界

每一种数据类型都有固定的内存空间，当输入的数据超出了该数据类型的可用空间且未能以预期的方式对该数据进行处理时，便会产生数据越界。常见的造成数据越界的操作包括类型转换、操作符转换、算术操作、比较操作、二进制操作等。几乎所有的编程语言都会受到数据越界的影响。这种问题看似是一种数据操作错误，但由于内存是基于数据进行分配的，当数据越界与内存分配联系到一起时，很有可能产生一个缓冲区溢出漏洞。

3.1　数组越界

数组越界是常见的错误，常常引起不可预见的软件异常。数组是同一类型多个数据的集合，程序中通过数组下标来区分或指定成员。一个拥有 10 个元素的数组，其下标范围是多少呢？

对于这个问题，不同的程序设计语言有着不同的规定。就 Fortran 和 PL/1 程序设计语言而言，该数组的下标默认范围是[1,10]（注：下标起始值允许程序员指定）；而对于 Algol 和 Pascal 程序设计语言，数组下标没有默认的起始值，程序员必须显式指定每个数组的上下边界；在标准的 BASIC 程序设计语言中，声明该数组时，编译器实际上为该数组分配了 11 个元素的空间，其下标范围是[0,10]。

C/C++语言中对于有 n 个成员的数组，数组下标从“0”开始，到“n–1”结束。当为数组传值或

读取数组成员时，需要注意数组的下标是否正确。

数组变量所拥有的内存空间在变量定义时就已经确定，程序在运行时会根据其定义，为其分配相应大小的缓冲区。程序在运行中如果数组变量的下标超过了定义的大小，就会产生数组越界，导致缓冲区溢出。

176. 数组访问越界

例 223：

```
1   #define ARRAY_LEN 6;
2   char str1[ARRAY_LEN];
3   char str2[ARRAY_LEN] = {0,1,2,3,4,5};
4
5   void main(void)
6   {
7     int index;
8     for(index = 0; index < ARRAY_LEN; index ++)
9     {
10      str1[index] = str2[index + 1]; // 错误
11    }
12  }
```

例 223 中，在第 10 行语句处产生了一个缓冲区溢出错误。当 index=5 时，程序会访问 str2[6]，造成数组访问越界。为防止数组越界，应在访问数组时判断下标的范围。

177. 数组下标错误

一个有 n 个元素的数组，其下标范围是 0 到 n−1，不存在下标为 n 的元素，取下标为几的元素的地址是合法的，而如果试图读取这个元素的值，其结果是未知的。

例 224:

```
1   main ()
2   {
3     int  i, a[10];
4     for(i=1;i<=10;i++)
5       a[i]=0; // 错误
6   …
7   }
```

例 224 中，数组 a 中有 10 个元素，其下标范围为 0 到 9。for 语句的比较部分本来是 i<10，却误写为 i<=10，因此，把并不存在的 a[10]置为 0，也就是把内存中数组 a 之后的一个字的内存置为

了 0，如果使用的编译器是按照内存地址递减的方式给变量分配内存，那么内存中数组 a 之后的一个字实际上是分配给了整型变量 i，将 a[10]置为 0 实际上是将计数器 i 的值置为 0，从而程序陷入了死循环。

该类错误常被称为“差一错误”（off-by-one error）。为了避免该类错误，可以采用不对称边界方法计算实际需要的个数。所谓不对称边界，是指用第一个入界点和第一个出界点表示一个数值范围，其中，入界点包含在范围之中，而出界点并不包含在范围之中，这正是不对称边界的由来。例如，对于 x>=10 且 x<=20，如果用不对称边界来表达，则为 x>=10 且 x<21，这里，下边界 10 是取值范围内的第一个点，而上边界 21 是取值范围外的第一个点。对于 C 这样的数组下标从 0 开始的语言，数组的上边界恰是数组元素的个数，正因为此原因，建议将例 224 中的第 4 条语句修改为：

```
for(i=0;i<10;i++)
```

“差一错误”在涉及缓冲区操作、循环次数时也经常出现。

178. 数学运算导致数组越界

程序中的数学运算如果没有考虑数组中元素类型所占的字节大小，将导致运算结果超出数组上下边界，使程序产生不可预知的行为。

例 225：

```
1    void f()
2    {
3     int a[20];
4     int n;
5     n = a [sizeof(a)-1]; // 错误
6    }
```

例 225 中第 5 行语句，a 是 int 类型数组，int 类型在 16 位系统内存中占 2 字节，sizeof(a)相当于 20×2=40，结果 n = a [39]，造成数组越界。

179. 外部输入引起数组越界

当程序员使用未经合法性检查的外部输入进行缓冲区相关操作时，容易造成缓冲区溢出。黑客常利用缓冲区溢出来进行攻击，如通过缓冲区溢出来执行任意代码，或者使系统执行异常从而进行拒绝服务攻击。缓冲区溢出通常都包含数组越界访问的情况，即通过系统调用，将超过缓冲区容量的数据写入缓冲区。

例 226：

```
1     #define GC_SMALL_SIZE    40
2     #define GC_BIG_SIZE     100
3
```

```
4  extern unsigned char  buf_small[GC_SMALL_SIZE];
5  extern unsigned char buf_big[GC_BIG_SIZE];
6
7  void BufOverRun()
8  {
9   unsigned short str_cpy_len = 0;
10  …
11   str_cpy_len = GetStrLength();
12  …
13   LoadBuf(buf_big); //为buf_big赋值
14   memcpy(buf_small, buf_big, str_cpy_len);// 错误
15  }
```

例 226 中，函数 BufOverRun()将一定长度的字符串从 buf_big 复制给 buf_small。由于复制字符串的长度是函数返回值，且没有经过合法性检查，因此在第 14 行语句进行字符串复制操作时可能会将字符数超过 40（buf_small 的大小）的字符串写入字符数组 buf_small，产生缓冲区溢出。

180. 数组转换越界

程序员常常使用指针操作将一个数组的值赋给另一个数组，在进行这种操作时一定要确保类似操作不会造成数组越界。

例 227：

```
1   typedef int arrLong[20];
2   typedef int arrShort [5];
3
4   void main(void)
5   {
6    int i ,value;
7    int arrMiddle [10];
8    for(i = 0; i< 10; i++)
9    {
10    printf("Please input a integer: ");
11    scanf("%d", arrMiddle[i]);
12   }
13   arrLong *long = (arrLong *)arrMiddle; // 错误
14   arrShort *short = (arrShort *)arrMiddle;
15  }
```

例 227 中，由于 arrMiddle 长度比 arrLong 小，在第 13 行语句中，使用前者为后者赋值时，会读取超出 arrMiddle 地址范围的值，产生缓冲区溢出，引起不可预计的后果；而在第 14 行语句中，由

于进行了强制类型转换，因此不会产生缓冲区溢出。

181. 字符串没有以'\0'结束

对于字符串操作，C 语言做了一件很特殊的事情，即给每个字符串加了一个字符串结束符'\0'，用以标志字符串的结束。通常，字符串结束符对于程序员是不可见的，但是，如果忽略了它，程序将无法知道字符串何时结束，从而引起缓冲区溢出。

例 228：

```
1     #include<stdio.h>
2     #include<string.h>
3     void func(char *str);
4     int Call(bool a)
5     {
6       char buf[100];
7       if(a)
8       {initial(buf)}  // initial()初始化字符数组
9     …
10     func(buf);
11    …
12    }
```

例 228 中，如果 a==false，则函数 func()使用的将是一个没有初始化的数组（即没有以'\0'结尾），如果 func()的参数必须是以'\0'结尾的字符串，则此调用会引起缓冲区溢出。为防止这样的错误，应该确认字符串是否以结束符'\0'结尾，如果不是以结束符'/0'结尾，应该手工置上结束符。

修改方法：在第 6 行语句后面增加语句：

```
buf[sizeof(buf)-1] = '\0';
```

需要特殊说明的是，对于程序内部传递的参数，程序员一般会进行必要甚至严格的合法性判断，但会忽略对外部数据（如用户输入或从文件中读取的数据）进行同样的判断，从而引入错误。

例 229：

```
1     FILE *FileRead
2     void ExCauseError(bool inn, char *interStr)
3     {
4      char dest[20];
5      if(inn)
6       {
7      if(strlen(interstr)< 20)
8         strcpy(dest, interStr);
```

```
9      else
10     {
11       strncpy(dest, interStr, 19);
12       dest[19] = '\0';
13      }
14   }
15   else if(fgets(dest, sizeof(dest),FileRead)==NULL)
16         break;
17   }
```

例 229 中，第 6 行至第 13 行语句对于程序传递的字符串进行复制操作时严格检查了缓冲区溢出的情况，但在第 15 行语句中，通过从文件读入字符串时，如果文件字符串长度大于等于 20，就会生成没有以'\0'结束的字符串，从而引起缓冲区溢出。例 229 只是说明了从文件中读取数据时会引起缓冲区溢出错误，实际上，在编程时如果忽略对外部数据的合法性判断，会产生很多的软件错误，在本书中“3.3 字符串操作溢出”一节有详细说明。

3.2 数据越界

当输入的数据超出了该数据类型的可用空间且未能以预期的方式对该数据进行处理时，便会产生数据越界。常见的造成数据越界的操作包括类型转换、操作符转换、算术操作、比较操作、二进制操作等。

182. 整数截断导致缓冲区溢出

将大范围整数（如 32 位）复制给小范围整数（如 16 位）时，容易发生数据截断问题。

例 230：

```
1   bool fun(byte *name, DWORD cbBuf)
2   {
3     unsigned short cSize = cbBuf;
4     byte *buf = new byte[cSize];
5     if (buf == null)
6     return FALSE;
7     memcpy(buf, name, cbBuf); // 错误
8      …
9     return true;
10  }
```

例 230 中，如果 cbBuf 是 0x00010020，第 3 行语句 cSize 只从 cbBuf 整数中复制了低 16 位，赋值后 cSize 结果为 0x20。在第 7 行语句中，将 cbBuf（0x00010020 字节）中的内容复制到新分配的目

标缓冲区 cSize（0x20 字节）中时，将造成缓冲区溢出。

程序中如果发生整数溢出，后续所有相关操作的结果都将发生变化。与缓冲区溢出相比，整数溢出导致的程序异常比较隐蔽，因为发生整数溢出时不会马上产生异常，即使程序执行结果与预期不同，也不容易发现问题。

183. 算术操作导致数据溢出

进行算术加减乘除操作时，计算结果有可能超过预期定义的数据范围而导致数据溢出。

例 231：

```
1    int i,j,sum;
2    i = 32760;
3    j = 8;
4    sum = i+j;    //错误
5    printf("%d",sum);
```

例 231 中，int 类型的范围是-32768~32767，i 和 j 赋值后并没有超过 int 类型的正常范围，但是执行加法运算后，计算结果 sum=32768，超过了 int 类型的范围，从而导致数据溢出。

184. 二进制操作导致数据溢出

无符号整型的左移、右移操作和有符号整型的右移操作都不会引起数据的溢出，然而对于有符号整型的左移操作，则会产生数据溢出。

例 232：

```
1    int i = 0x40000000;
2    i = i << 1;    //错误
```

例 232 中，i = 0x40000000，即 16 进制的 40000000，二进制的 01000000...0000。i 在左移 1 位之后就会变成 0x80000000，也就是二进制的 100000...0000，符号位被置为 1，其他位全是 0，相当于 32 位的-2147483648，导致溢出。

3.3　字符串操作溢出

C/C++语言提供的字符串库函数中，如果目的缓冲区容纳不下源字符串输入的字符个数，将导致程序运行错误。此外，对于包含源字符串和目的字符串两个参数的字符串函数，当目的地址和源地址有重叠时，函数执行后将相互修改地址中的内容或者找不到所指地址。通常将这些函数称为危险函数，可能引起缓冲区溢出的危险函数有三类：输入类函数、输出类函数、字符串处理函数。表 3-1 给出了危险函数的具体分类。

表 3-1　危险函数分类

类　型	函　数
gets	gets，getc，fgets，fgetc，getchar
strcpy	strcpy，lstrcpy，wcscpy，_tcscpy，_mbscpy，strccpy，strecpy，strncpy，lstrcpyn，wcsncpy，_tcsncpy，_mbsnbcpy
strcat	strcat，lstrcat，wcscat，_tcscat，_mbscat，strncat，lstrncat，wcsncat，_tcsncat，_mbsncat，_mbsnbcat
scanf	scanf，sscanf，fscanf，vscnaf，vsscanf，wscanf，swscnaf，fwscanf，_tscanf，vfscanf，_ftscanf，_stscanf，_cscanf
printf	printf，sprint，fprintf，vsprintf，vswprintf，vsnprintf，swprintf，wvnsprintf，wnsprintf，_snprintf，_snwprintf，_vsnprintf
其他	getopt，getopt_long，getopt_long_only，getpass，getwd，getpw，_getts，getenv，oemtochar，oemtocharbuff，oemtoansibuff，gettemppath，syslog，select，mbstowcs，strtrns，strcadd，read，memcpy，copymemory，bcopy，strfrm，wcsxfrm，realpath，chroot，streadd，system，popen

185. 格式化字符串错误导致缓冲区溢出

从第一次接触 C 语言起，程序员便会经常使用格式化字符串。C/C++语言的格式化字符串用于格式化数字的输出、字符串合并和转换等多种场合，如常用的“%s”表示字符串、“%d”表示十进制有符号整数等。如果格式化字符串使用不正确，就会产生潜在危险，例如，当程序员试图将一个字符串传递到一个使用格式化字符串的函数中时，如果不检查字符串内容就直接使用，很有可能引起缓冲区溢出。

例 233：

```
1    #include<stdio.h>
2    void main()
3    {
4      char buf[20];
5      printf("Enter testing string: \n ");
6      scanf("%19s", buf);
7      printf(buf); // 错误
8    }
```

例 233 是一个简单的程序，其功能是输入一个字符串，然后再打印出来。但是如果输入的字符串中包含如“%x”、“%s”等格式的字符串时，程序会打印出错误的结果或者直接异常退出。这是因为在 C/C++语言中，这些字符串（格式化字符串）都有特殊意义，如果程序员能控制该格式化字符串参数（如例 233 中的 buf），则会产生更加严重的后果，这也是黑客对缓冲区溢出进行攻击常用的一个方法。

修改方法：将第 7 行语句修改为：printf("%s", buf);

186. gets()函数使用隐患

gets()函数原型为：

```
char *gets(char *buffer);
```

该函数从标准输入流中读取字符串，直至接收到换行符或 EOF 时为止，并将读取的字符串存储在 buffer 指针所指的字符数组中。由于该函数并不判断所读取字符串的长度，所以使用不当极易引起缓冲区溢出。

与 gets()同类的字符串函数，如 fgets(char*string,int n, FILE*stream)，该类函数从源地址读取相应个数的字符到目的缓冲区，如果实际读入的字符数大于目的缓冲区大小，则程序运行时或者修改与目的缓冲区临近的内存中的数据，或者给出访问内存错误。

例 234：

```
1    #include<stdio.h>
2    void main()
3    {
4       char str[20];
5       gets(str);
6       printf("%s\n",str);
7    }
```

例 234 中，如果用户输入的字符串长度大于等于 20，就会产生缓冲区溢出。

鉴于 gets()函数的不安全性，如果程序员要使用该函数，一定要确保 buffer 的空间足够大。

187. fgets()函数使用隐患

fgets()函数原型为：

```
char *fgets(char *s, int size, FILE *stream);
```

该函数从文件流中读取一行或指定长度字符串到缓冲区 s 中。stream 是文件指针，s 是存放字符串的起始地址，size 是字符串的长度。函数从 stream 中读入“size–1”个字符放到起始地址为 s 的空间内，如果读入的字符串长度等于“size–1”，或在读入过程中遇到一个换行符或 EOF（文件结束标志），则结束本次读操作，读入的字符串中包含最后读取的换行符。读入结束后，系统将自动在字符串 s 后面添加'\0'，并以 s 作为函数返回值。

例 235：

```
1    #include <string.h>
2    #include <stdio.h>
3    int main(void)
4    {
```

```
5    FILE *stream;
6    char string[] = "This is a test";
7    char msg[5];
8    stream = fopen("DUMMY.FIL", "w+"); //打开文件
9    fwrite(string, strlen(string), 1, stream);//写文件
10   fseek(stream, 0, SEEK_SET);//文件的开始处
11   fgets(msg, strlen(string)+1, stream); // 错误
12   printf("%s", msg);
13   fclose(stream);//关闭文件
14   return 0;
15  }
```

例 235 中，fgets()函数将 stream 文件流中长度为“strlen(string)+1”的内容写入 msg 字符串中，因为 msg 字符串内存大小不能容纳该文件流读出的长度，所以导致缓冲区溢出。

188．getchar()函数使用隐患

getchar()函数原型为：

```
int getchar(void);
```

该函数从缓冲区读取一个字符，返回 int 类型值。

例 236：

```
1  main()
2  {
3   char c;
4   c = getchar();  // 错误
5   …
6  }
```

例 236 中，函数 getchar()返回的是 int 类型的值，却赋值给字符类型的变量 c，从而产生数据截断。

修改方法：将第 4 行语句修改为：`int c = getchar();`

许多初学者都习惯用 char 型变量接收 getchar、getc 和 fgetc 等函数的返回值，其实这是错误的，因为 getchar 等函数的返回值是 int 型的，当赋值给 char 型变量时，会发生降级，从而导致数据截断。

189．strcpy()函数使用隐患

strcpy()函数原型为：

```
char *strcpy(char *dest,char *src);
```

该函数将源字符串复制到缓冲区，复制的字符数目取决于源字符串中字符的数目。由于该函数

并不检查缓冲区边界（即 dest 的长度），因此，如果没有限制源字符串大小，容易造成缓冲区溢出。

例 237：

```
1    #include<stdio.h>
2    #include<string.h>
3    void main()
4    {
5      char buf[10];
6      char str[20];
7      printf("Enter testing string: \n ");
8      scanf("%19s", str);
9      strcpy(buf, str);  // 错误
10     printf("%s\n",buf);
11   }
```

例 237 中，程序将用户输入的字符串读入字符数组 str 中，然后用 strcpy()函数将字符串 str 复制到 buf 中。如果用户输入的字符串长度小于 10，程序运行不会有问题，但如果字符串长度大于等于 10 时，则会产生缓冲区溢出错误。

如果事先知道目的缓冲区大小，则在执行此操作前要增加明确的检查语句。对于本例而言，需要在第 8 行语句后面增加判断语句。

修改后的程序如例 238 所示。

例 238：

```
1    #include<stdio.h>
2    #include<string.h>
3    void main()
4    {
5      char buf[10];
6      char str[20];
7      printf("Enter testing string: \n ");
8      scanf("%19s", str);
9     if (strlen(str)>=dst_size)     //判断语句
10    {
11       输出错误提示--
12    }
13    else
14    {
15      strcpy(buf, str);
16      printf("%s\n",buf);
17    }
18   }
```

确保strcpy()不会溢出的另一种方式是随需分配，即通过在源字符串上调用strlen()来分配足够的空间，例如，采用如下的语句：

```
dst=(char*)malloc(strlen(src));
strcpy(dst,src);
```

也可以使用strncpy()函数代替strcpy()，但strncpy()函数的开销比strcpy()函数大。如果程序员执意要使用strcpy()，一定要确保源字符串长度小于目的缓冲区长度。

190. strncpy()函数使用隐患

strncpy()函数有两个原型，其一为：

```
char * strncpy(char * tgtstr, char *srcstr,size_t n);
```

在使用strncpy()函数进行字符串复制时，第三个参数——复制的字符串长度n，必须满足：n>=0&&n<=sizeof(srcstr)&&n<=sizeof(tgtstr)-1，以便为字符串结束符'\0'留出一个位置。

strncpy()函数的另一个原型为：

```
char *strncpy(char *dest, const char *src, size_t n);
```

strncpy()函数把src指向的字符串的前n个字符复制到dest指向的数组中。如果n大于dest所指向的数组内存空间，将造成缓冲溢出。

例239：

```
1    #include<stdio.h>
2    void main()
3    {
4      char str1[10];
5      char str2[20]="string of length 19";
6        …
7      strncpy(str1,str2,sizeof(str2)); // 错误
8      printf("1 str1 is :%s\n", str1);
9       …
10     strncpy(str1,str2,sizeof(str1)); // 错误
11     printf("2 str1 is :%s\n", str1);
12       …
13     strncpy(str1,str2,sizeof(str1)-1); // 错误
14     printf("3 str1 is :%s\n", str1);
15   }
```

例239中，有三处使用了strncpy()函数。在第7行语句中，由于sizeof(str2)>sizeof(str1)，将长度为20的字符串复制到长度为10的str1是错误的；第10行语句由于没有为字符串结束符'\0'预留

位置，也是错误的；对于第 13 行语句，它满足前面讨论的关于 strncpy()第三个参数的条件，但需要特别注意的是，如果源字符串（str2）长度大于要复制的字符串长度（sizeof(str1)-1），strncpy()并不会自动给目的字符串（str1）加上'\0'这个结束符，因此，需要在第 13 行语句后面增加语句：str1[9]='\0';

例 240：

```
1    #inclue <string.h>
2    main()
3    {
4      const char a[30]="string(1)";
5      char b[]="string(2)";
6      strncpy(b, a ,20); // 错误
7    }
```

例 240 中第 6 行语句，strncpy()函数将 a 字符串的前 20 个字符复制到 b 字符串所在的内存空间中，很显然，b 字符串没有能容纳 20 个字符的内存空间，从而引起缓冲区溢出。

191．strlcpy()函数使用隐患

strlcpy()函数原型为：

```
size_t strlcpy(char *dst, const char *src, size_t size);
```

该函数基本上与 strcpy()函数相同，它将源字符串 src 中前 size 个字符复制到目的字符串 dst 中，在遇到字符串结束符时终止。如果 size 大于 src 所指向字符串的长度，则只将 src 复制到 dest 后便返回。

例 241：

```
1    #include <string.h>
2    main()
3    {
4      char  test1[12]="test function strlcpy";
5      char  test2[]="str";
6      printf("%d", strlcpy (test2,test1,sizeof(test1))); // 错误
7    }
```

例 241 中第 6 行语句，strlcpy()函数把字符串 test1 整串内容复制到 test2 中，导致缓冲区溢出。

192．strncat()函数使用隐患

strncat()函数有两个原型，其一为：

```
char *strncat(char *dest, char *src, int n);
```

另一个为：

```
char *strncat(char *dest, const char *src, size_t n);
```

该函数把 src 所指字符串的前 n 个字符添加到 dest 结尾处（覆盖 dest 结尾处的'\0'），并添加'\0'。该函数要求 dest 所指的内存区域有足够空间来容纳 src 的字符串，如果 n 大于 dest 剩余空间的大小，就会导致缓冲区溢出。

例 242：

```
1    #include<string.h>
2    #define SIZE 20
3    void main()
4    {
5      char buf[SIZE];
6      char *str1 = "testingstr1";
7      char *str2 = "testingstr2";
8      strncpy(buf, str1, SIZE);
9      buf[SIZE-1] = '\0';
10     strncat(buf, str2, SIZE); // 错误
11       …
12   }
```

例 242 中，先用 str1 初始化数组 buf，然后将 str2 的前 SIZE 个字符添加到 buf 的结尾处。第 10 行语句使用 strncat()函数时错误地使用 buf 数组的大小作为参数，可能会引起缓冲区溢出。

修改方法：将第 10 行语句修改为：

```
strncat(buf, str2, SIZE-strlen(buf));
```

例 243：

```
1    #include <string.h>
2    main()
3    {
4     char a[15]="string(1)";
5     const char b[]="string[b]";
6     strncat(a,b, sizeof(b));  //错误
7    }
```

例 243 中第 6 行语句，strncat()函数把字符串 b 的所有内容复制到 a 字符串的串尾，但 a 字符串没有足够的剩余空间容纳字符串 b 的内容，造成缓冲区溢出。

193．strcat()函数使用隐患

strcat()函数原型为：

```
char *strcat(char *dest,char *src);
```

该函数把 src 所指字符串添加到 dest 结尾处（覆盖 dest 结尾处的'\0'）并添加'\0'。当 dest 没有足够的空间来容纳 src 中的字符串时，会造成缓冲区溢出。

例 244：

```
1   #include <string.h>
2   …
3   main()
4   {
5     char a[20]="abcdefghijklmn";
6     char b[10]="ok";
7     strcat (b, a); // 错误
8     …
9   }
```

例 244 中第 7 行语句，strcat()函数将 a 字符串内容添加到字符串 b 中，显然，字符串 a 所占内存大于字符串 b 所占内存空间，引起缓冲区溢出。

194．strlcat()函数使用隐患

strlcat()函数原型为：

```
size_t strlcat(char *dst, const char *src, size_t size);
```

该函数基本上与 strcat()函数相同，但只复制前 size 个字符。如果 size 大于 src 所指向字符串的长度，则将 src 复制到 dest 后便返回。src 和 dest 所指内存区域不允许重叠，且 dest 必须有足够的空间来容纳 src 中的字符串。

例 245：

```
1   #include <syslib.h>
2   #include <string.h>
3   main()
4   {
5     char d[20]="Golden Global";
6     char s[ ]=" View";
7     strlcat (s, d, sizeof(d)); // 错误
8     printf("%s",d);
9   }
```

例 245 中第 7 行语句，strlcat()函数将 d 字符串内容复制到字符串 s 中，显然，字符串 d 所占内存大于字符串 s 所占内存空间，引起缓冲区溢出。

195. scanf()函数使用隐患

scanf()函数原型为：

```
int scanf(const char *format, …);
```

scanf()函数是格式化输入函数，它从标准输入设备（键盘）读取输入的信息。例如："%d"代表从键盘输入十进制整数；"%o"代表从键盘输入八进制整数；"%x"代表从键盘输入十六进制整数；"%c"代表从键盘输入一个字符；"%s"代表从键盘输入一个字符串；"%f"代表从键盘输入一个实数。

scanf()函数读取字符串类型时，不检查用户输入字符串的长度，容易造成缓冲溢出。

例 246：

```
1   #include<stdio.h>
2   main()
3   {
4     char a[10];
5     scanf("%s",a); // 错误
6   }
```

例 246 中，字符串 a 的长度是 10，如果 scanf()函数输入 11 个字符给字符串 a，将导致字符串 a 溢出。

196. fscanf()函数使用隐患

fscanf()函数原型为：

```
int  fscanf(FILE *fp,const char *mode, …);
```

fscanf()函数从一个流中执行格式化输入。它从 fp 指向的文本文件中，按 mode 中指定的格式逐个读取文本数据并转换成指定的数据类型，赋给对应指针指向的存储单元。

fscanf()函数读取文本文件时，不检查读取文本数据的长度，容易造成缓冲溢出。

例 247：

```
1   int CCommDamApp (FILE *f, char *x)
2   {
3       if (m_pAppDatabase)
4       {
5           …
```

```
6      fscanf(f, "%10s", x); // 错误
7      …
8      }
9   }
```

例 247 中第 6 行语句，如果在 fscanf()函数中没有指明要从格式化数据流中提取字符串的长度，那么必须限制提取的长度至多是“格式化数据流长度–1”，即提取的长度不能大于格式化数据流的长度，超出这个范围，程序很可能崩溃。

修改方法：限制提取格式化数据流的长度，或者将长度设置为“格式化数据流长度–1”。

197．sprintf()函数使用隐患

sprintf()函数原型为：

```
int sprintf( char *str, char *format [, argument] … );
```

sprintf()函数的作用是将一个格式化的字符串输出到一个目的字符串中，该函数的第一个参数是目的字符串。因为该函数在进行字符串操作时，并不检查目的字符串空间大小，所以可能会出现数组越界而导致程序崩溃的问题。因此，使用 sprintf()函数时，一定要确保目的字符串空间有足够的大小。

例 248：

```
1    #define ARRAY_LEN 10;
2    void StrCopyToBuffer()
3    {
4      char buf[ARRAY_LEN];
5      char str[20]="string of length 19";
6      sprintf(buf, str); // 错误
7      printf("%s\n",buf);
8    }
```

例 248 中第 6 行语句，由于 sprintf()函数并不检查输出目的字符串的大小，因此直接将 str 的全部内容（19 个字符）复制给大小为 10 个字节的 buf，造成缓冲区溢出。

198．snprintf()函数使用隐患

snprintf()函数原型为：

```
int snprintf(char *str, size_t size, const char *format, …);
```

该函数将可变个数的参数(...)按照 format 格式化成字符串，然后将格式化字符串复制到 str 中。如果格式化后的字符串长度小于 size，则将此字符串全部复制到 str 中，并在其后添加一个字符串结

束符'\0'；如果格式化后的字符串长度大于等于 size，则只将其中的(size−1)个字符复制到 str 中，并在其后添加一个字符串结束符'\0'。因为它的第二个参数 size 可以控制向缓冲区中写入的字符串的长度，所以比 sprintf()函数更安全，程序员常常使用 snprintf()函数代替 sprintf()函数，但是，如果参数 size 设置不正确，依然会产生缓冲区溢出。

例 249：

```
1  #define ARRAY_LEN 10;
2  void StrCopyToBuffer()
3  {
4    char buf[ARRAY_LEN];
5    char str[20]="string of length 19";
6    snprintf(buf, 10, str); // 错误
7    printf("%s\n",buf);
8  }
```

例 249 中，第 6 行语句使用 snprintf()函数将字符串 str 的前 9 个字符复制给 buf，在本例中，这个用法是没有问题的，但是，如果在程序其他地方将 buf 的大小（即 ARRAY_LEN）改变为一个小于 10 的数，则会在第 6 行语句产生缓冲区溢出。

修改方法：将第 6 行语句修改为：

```
snprintf(buf, sizeof(buf), str);
```

199．vsnprintf()函数使用隐患

vsnprintf()函数原型为：

```
int vsnprintf(char *str, size_t,const char *format, va_list ap);
```

vsnprintf()函数和 snprintf()函数很类似，只是其中的参数是指针列表。该函数不检查缓冲区边界，因此，当提供过多数据时可能导致缓冲区溢出。

例 250：

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdarg.h>
4  char usermsg[20];
5  char *Func(char *format,…)
6  {
7    va_list arglist;
8    …
9    va_start(arglist,format);
10   vsnprintf(&usermsg,20,format,srglist); // 错误
```

```
11    …
12  }
```

例 250 中，第 10 行语句使用 vsnprintf()函数时，并未对要复制的字符串大小进行检查，存在缓冲区溢出的危险。

200．memcpy()函数使用隐患

memcpy()函数原型为：

```
void *memcpy(void *dest, const void *src, size_t n);
```

该函数将指针 src 指向的前 n 个字节复制到 dest 指向的前 n 个内存区域中。如果 src 和 dest 有重复区域，则会被覆盖。

当程序读取缓冲区内容时，如果 size 参数的值是负数或大于源缓冲区实际的大小时，将超越源缓冲区边界，导致读取缓冲区溢出。

当程序将数据写入缓冲区时，如果 size 参数的值是负数或大于目的缓冲区实际大小时，将超越目的缓冲区边界，导致写入缓冲区溢出。

例 251：

```
1  #include <string.h>
2  void example()
3  {
4    int src[100];
5    int dest[200];
6    …
7    memcpy(dest, src, sizeof(dest));
8   …
9  }
```

例 251 中，memcpy()函数源数据长度是 100，而目的数据长度是 200，当 memcpy()函数复制源数据到目的数据时，产生读取缓冲区溢出。

例 252：

```
1  #include <string.h>
2  void example()
3  {
4    int src[200];
5    int dest[100];
6    …
7    memcpy(dest, src, sizeof(src));
```

```
8    …
9  }
```

例 252 中，memcpy()函数源数据长度是 200，而目的数据长度是 100，当 memcpy()函数复制源数据到目的数据时，产生写入缓冲区溢出。

201．bcopy()函数使用隐患

bcopy()函数原型为：

```
extern void bcopy(const void *src, void *dest, int n);
```

bcopy()函数将字符串 src 的前 n 个字节复制到 dest 中，如果需要的 n 个字节大于 dest 所指的内存空间时，会造成缓冲区溢出错误。

例 253：

```
1    #include<string.h>
2    main()
3    {
4      char dest[10]="string(a)";
5      char src[40]="string\0string";
6      bcopy(src,dest,30); // 错误
7    }
```

例 253 中第 6 行语句，bcopy()函数复制 30 个字节到 dest 中，需要复制的字节数大于 dest 的内存空间，从而造成缓冲区溢出。

202．memset()函数使用隐患

memset()函数原型为：

```
void *memset(void *s, int c, size_t n);
```

该函数将 s 所指内存空间的 n 个字节的值设为 c。

例 254：

```
1    #include<string.h>
2    #include<mem.h>
3    #include<stdio.h>
4    void main(void)
5    {
6      char a[5];
7      …
8      memset(a,1, 5*sizeof(int)); // 错误
```

```
9      …
10   }
```

例 254 中，数组 a 是字符型的，字符型占据 1 字节大小的内存，整数型占据 4 字节（32 位计算机）的内存，而数组 a 字符串大小是 5 字节，因此，第 8 行语句 memset()函数将以 a[0]开始的 20(5*sizeof(int))个字节的内容置为 1，导致缓冲区溢出。

修改方法：将第 8 行语句修改为：

```
memset(a,1,5*sizeof(char));
```

203. getenv()函数使用隐患

getenv()函数原型为：

```
char* getenv(const char* name);
```

该函数返回一个给定的环境变量，环境变量名不区分大小写，如果指定的变量在环境中未定义，则返回空串。

例 255：

```
1    #include<stdlib.h>
2    main()
3    {
4      char buffer[100];
5      char *config = getenv ("CONFIG_FORMAT");
6      if (config == NULL)
7          config = DEFAULT_FORMAT;
8      sprintf(buffer,"%s",config);
9      …
10   }
```

例 255 中，来自 getenv()函数中未作合法性检查的数据作为 sprintf()函数的格式化字符串，很容易导致缓冲区溢出错误。

204. read()函数使用隐患

read()函数原型为：

```
ssize_t read(int fd, void *buf, size_t count);
```

read ()函数从 fd 指定的已打开文件中传送 count 个字节到 buf 指针所指的内存中。

例 256：

```
1    main()
```

```
2    {
3       static char filename[]="t1.txt" ;
4       char buffer[10] ;
5       int handle ;
6       int i ;
7       int total = 0 ;
8       handle = open(filename,O_RDONLY) ;
9       if ( handle == -1 )
10      {
11         printf("[%s] create fail !!!!\n",filename) ;
12         exit(1) ;
13      }
14      else
15      {
16         read(handle,buffer,20); // 错误
17      }
18      return(0) ;
19   }
```

例 256 中，read()函数从文件“t1.txt”中读取 20 字节，放到内存空间只有 10 字节大小的 buffer 中，导致 buffer 溢出。

第 4 章 指针问题

指针是 C/C++语言的灵魂，姑且不论这种说法是否准确，但它道出了指针的两个特性：① 指针是 C/C++语言中非常重要的技术；② 指针是 C/C++语言中最难以驾驭的技术。因此，只有深入掌握指针才能真正掌握 C/C++语言。

C/C++语言没有限制指针的指向，因此，指针可能指向无效或非法的地址。如果指向的内容不合法，引用或修改该地址的内容会引起程序执行紊乱，出现不可预料的后果。

指针非法引用是由于对内存空间的不合法操作或管理不当引起的，指针非法引用错误大致可分为指针的无效引用、释放非法内存和释放未申请资源指针指向的内存空间。

指针的无效引用是指引用无效的指针，即引用指针的值是无法确定的。无效引用不一定引起程序异常终止，程序仍可能运行，但运行结果是未知的，其结果取决于许多因素。例如，指针存储的无效地址在操作系统中指向的内存地址、不同编译器代码生成机制对空指针的定义等。指针的无效引用主要包括以下几种情况。

（1）指针已定义但未初始化。此时指针指向的地址是无效的，直接引用此类指针必然导致错误。

（2）指针指向的空间被释放。指向某一有效空间的指针在程序执行过程中被释放，此时指针指向的地址是不确定的，直接引用此类指针可能导致错误，在没有其他指针对内存空间操作时，释放该指针指向的空间后即刻引用可能不会出现错误。

（3）指向局部变量的指针或指向的变量超出作用域的指针。要使指针有效，必须确保指针所指的内存块有效。由于每个语句块中都允许程序员自定义变量，这些变量只在被定义的语句块内有效，当语句块结束时，这些变量就会被自动释放。如果在此语句块内将某个语句块外的指针指向语句块内定义的某个局部变量，当语句块结束时，该指针指向的内存会被自动释放，此时，指针将指向无效的内存块，在该语句块外对该指针的引用可能会导致不可预料的错误。

（4）空指针，即指针值为 NULL 的指针。C 程序提倡程序员在释放一个指针后将其值置为 NULL，这样，一方面便于对 NULL 指针的取值和写入非法操作进行检查，另一方面，由于 C 程序允许 NULL 指针调用 free()函数，且不会出现潜在错误，从而可以有效防止对一个指针变量多次释放资源的错误

操作。

（5）调用库函数 malloc()、realloc()等不成功时，未判断返回值 NULL 而直接引用该指针。

释放非法内存是由于指针本身不能确定指向的究竟是哪一类内存，因此，程序员可以释放任何指针。如果程序员使用库函数释放的内存指针不是指向一个合法的堆空间首地址，调用该库函数的释放操作可能破坏内存中的有关信息。释放非法内存主要包括以下几种情况。

（1）释放栈或静态存储区中的内存空间。栈空间和静态存储区的空间是自动分配的，如果释放这些内存空间，可能改写其他变量的内存空间，引起未知的错误。

（2）释放某个无效指针指向的内存空间。由于无效指针指向的地址是不确定的，因此其合法性也无法确定，对其进行释放内存操作产生的后果也是无法预料的。

（3）释放动态分配的内存空间中的某个中间地址。根据 C/C++语言内存分配策略，当分配堆内存空间时，返回的是分配成功的内存首地址，内存的分配函数根据该首地址写入其对应的数据；在回收内存空间时，内存回收函数必须通过首地址才能够正确找到相应的内存数据。如果程序员释放的是堆内存空间的某个中间地址，则内存释放函数必然无法找到正确的内存数据，从而引发不可预料的错误。

释放未申请资源指针指向的内存空间产生的问题是由于这类指针变量指向的是随机地址，对他们的释放操作可能导致未知的结果或错误。

4.1 空指针解引用

C/C++ 语言中允许一个整数代表指针，其中的一个特殊情况就是常数 0；当指针等于 0 时，就意味着这个指针为空指针，因为空指针没有指向一块有意义的内存，所以无法对它进行存取操作。

空指针并不指向任何对象。因此，除了赋值或比较运算，其他情况下使用空指针都是非法的。例如，如果 p 或 q 是一个空指针，strcmp(p,q)的执行结果是未定义的。违反规定使用空指针时，不同的编译器将给出不同的结果，有些编译器对内存 0 位置施加了硬件级的读保护，如果错误使用了空指针，程序将立即终止执行；而有些器编译对内存 0 位置只允许读，不允许写，在这种情况下，其读出的内容常常是无效信息；还有些编译器对内存 0 位置既允许读也允许写，在这种情况下，很可能使用空指针覆盖操作系统中的内容，从而造成系统崩溃。

205．空指针被解引用或作为参数传递给函数

例 257：

```
1   main()
2   {
3   int *p = 0; //指针 P 是空指针
4   *p = 1;     //解引用空指针 p
5   }
```

例 257 中，第 3 行语句将指针 p 定义为空指针，第 4 行语句没有判断指针是否为空就解引用该指针，从而产生错误。

例 258：

```
1    int copy(char *dst,char *src)
2    {
3      if (!src)  return 0;
4      else
5      {
6        dst=src;
7        return 1;
8      }
9    }
10   main()
11   {
12     char  *p=0;
13     copy(p,"Hello");// 错误
14   }
```

例 258 中，指针 p 被赋值为空指针，第 13 行语句没有判断指针 p 是否为空，就作为参数传递给 copy()函数，导致程序错误。

206. 条件判断语句导致空指针被解引用或作为参数传递给函数

例 259：

```
1    main()
2    {
3      char *p;
4      scopy(char *dst, char *source)
5      {
6        if (!source) return 0;
7      dst= source;
8      }
9      chchar(int i,char *t)
10     {
11       if(i) p=NULL;
12       scopy(p,"copychar");// 错误
13     }
14   }
```

例 259 中，当第 11 行条件判断语句成立时，p 赋值为空指针，第 12 行语句在没有判断 p 是否为空的情况下，作为参数传递给了 scopy()函数。

207. 函数返回的空指针被解引用或作为参数传递给函数

当程序中调用函数返回结果为空指针时，在没有判断该指针值是否为空的情况下被解引用或作为参数传递给函数。

例 260：

```
1    main()
2    {
3     foo()
4      {
5      return  0;
6      }
7      int *p;
8      p=foo();
9      *p=1; // 错误
10   }
```

例 260 中，调用 foo()函数后，指针 p 的值为空，而在第 9 行语句中指针 p 被解引用。

208. 条件判断语句导致调用函数返回的空指针被解引用或作为参数传递给函数

当程序中调用函数返回结果为空指针时，程序中的条件判断语句可能导致在没有判断指针值是否为空的情况下被解引用或作为参数传递给函数。

例 261：

```
1    foo()
2    {
3      return  0;
4    }
5    #include<stlib.h>
6    main()
7    {
8     int j;
9     char *p;
10    scanf("%d",&j);
11    scopy(char *dst, char *source);
12      if (!source) return 0;
```

```
13      dst= source;
14    p=foo();
15    if(j)
16    scopy(p,"copychar");// 错误
17  }
```

例 261 中，指针 p 的值由 foo()函数得到，当第 15 行语句条件判断成立时，空指针作为参数传递给 scopy()函数，产生程序错误。

209. 通过赋值得到的空指针被解引用或作为参数传递给函数

当程序中的指针值通过直接赋值或间接函数赋值得到，并且程序中存在判断指针值是否为空的语句，但这个判断语句并未改变空指针的值时，程序中可能存在空指针被解引用或者作为参数传递给函数的错误。

例 262：

```
1    getpointer()
2    {
3      return 0;
4    }
5    void  main()
6    {
7      char  *p=getpointer();
8      if(p!=NULL) {}
9       *p=1; // 错误
10   }
```

例 262 中，当第 7 行语句得到的 p 指针值为空时，程序中第 8 行语句虽然对指针 p 进行了非空判断，但并未改变空指针 p 的值，随后空指针 p 仍被解引用。

例 263：

```
1   int copy(char *dst,char *src)
2   {
3     if (!src)  return 0;
4     else
5     {
6       dst=src;
7       return 1;
8     }
9   }
```

```
10  check(char *q)
11  {
12   char *p=getpointer();
13   if(p!=NULL){}
14   if(q) {p=q;}
15   copy(p,"Hello");      // 错误
16  }
```

例 263 中，通过 getpointer()函数得到指针 p 的值，第 13、14 行语句分别对指针 p、q 的值进行了非空判断。当指针 p 和 q 都是空指针时，第 13、14 行语句并未改变指针 p、q 的值（仍然为 0），第 15 行语句将空指针 p 作为参数传递给函数 copy()，导致程序错误。

210. 函数中没有判断指针参数是否为空

当函数返回的指针以参数形式传递给另一个函数时，函数内部没有判断指针是否为空而使用了该指针，可能出现程序错误。

例 264：

```
1    int copy(char *dst,char *src)
2    {
3       if (!src)  return 0;
4       else
5      {
6        dst=src;
7        return 1;
8      }
9    }
10  getpointer()
11  {
12   return 0;
13  }
14   main(char *q)
15  {
16    char *p=getpointer();
17    copy(p,"Hello");          // 错误
18  }
```

例 264 第 16 行语句，指针 p 通过函数 getpointer()获得空值，第 17 行语句中以参数形式将其传递给 copy()函数，而 copy()函数内部并没有判断该指针是否为空就直接使用，导致程序错误。

211．函数解引用未作判断的指针值

程序中存在判断指针值是否为空的语句，但该判断并未改变空指针的值，该指针又以参数形式传递给程序中其他函数时，该函数对传递的指针值不作非空判断就进行解引用，导致程序错误。

例 265：

```
1    int copy(char *dst,char *src)
2    {
3      dst=src;
4      return 1;
5    }
6    main(char *q)
7    {
8      char *p=0;
9      if(p!=NULL) *p="abc";
10     copy(p,"Hello");
11   }
```

例 265 中，指针 p 在第 8 行语句中被赋值为空，在第 10 行语句中以参数形式传递给 copy()函数，在 copy()函数内部对其“dst”参数不作非空判断就直接使用，导致程序错误。

212．函数引用空指针常量

空指针常量传递给函数，函数在没有检查指针是否为空的情况下就引用该指针。

例 266：

```
1    int copy(char *dst,char *src)
2    {
3      dst=src;
4      return 1;
5    }
6    main(char *q)
7    {
8      char *p;
9      *p="hello";
10     copy(0, p); // 错误
11   }
```

例 266 中，第 10 行语句给 copy()函数传递了一个空指针常量 src，copy()函数在没有判断 src 指针是否为空的情况就引用了指针，导致程序错误。

213. 函数解引用空指针

程序中指针以参数的形式传递给函数进行指针解引用，在函数内部虽然对指针进行了是否为空的判断，但并未改变指针的值，指针在被解引用时仍为空。

例 267：

```
1   int copy(char *dst,char *src)
2   {
3      if(dst!=NULL){
4      dst=src;
5      return 1;
6      }
7   }
8   main()
9   {
10     char *p=0;
11     copy(p,"Hello");    // 错误
12  }
```

例 267 第 11 行语句中，空指针 p 传递给 copy()函数，在 copy()函数内部虽然存在判断指针是否为空的语句，但并未改变指针值，指针 p 在解引用时仍然为空，导致程序错误。

4.2 指针非法使用

214. 指针修改常量字符串

如果使用指针修改常量字符串，程序可能出现不可预料的后果。

例 268：

```
1   main ()
2   {
3     string  *p,*q;
4     p="xyz"; //字符串
5     q=p;
6     p[1]='A'; // 错误
7     …
8   }
```

例 268 中，字符型变量指针 p、q 指向同一常量字符串“xyz”，在第 6 行语句中，指针 p 修改了

常量字符串的内容。

215. 含有指针型数据成员的类或结构体对象值传递错误

当类或结构体中存在指针变量成员且没有声明拷贝构造函数时，如果需要传递类或结构体的对象，只能通过引用方式进行传递。引用是变量的别名，当传递给函数的参数是引用时，传递的就是这个变量的内存地址。

例 269：

```
1     struct s
2     {
3        char  name[10];
4        float  value;
5        int    *p;
6      } ;
7      void  f(s);
8      main()
9      {
10        s  s1;
11        f(s1);
12     }
```

例 269 中，结构体 s 类型的变量 s1 作为参数传递给函数 f()，在 s1 对象作为参数传递时，其指针变量 p 完成的是位复制。位复制可能会造成指针指向被释放的内存，进而导致程序异常。为了避免位复制可能带来的问题，建议使用对象的引用作为参数进行传递。修改后的程序如例 270 所示。

例 270：

```
1    struct s
2      {
3        char  name[10];
4        float  value;
5        int   *p;
6     } ;
7      void  f(&s);
8      main()
9      {
10      s  s1;
11      f(s1);
12   }
```

216. 使用指向数据类型的指针作为 sizeof()的参数

当使用 sizeof()作为运算符来确定由 alloc()、calloc()或 realloc()函数分配的内存大小时，如果不将真实数据类型作为 sizeof()的操作数，而是将指向数据类型的指针作为 sizeof()的操作数，则 sizeof()返回的是指针的大小。

例 271：

```
1   main(int n)
2   {
3     typedef struct student
4     {
5       int num;
6       char name[20];
7       char sex;
8     } std1, *pt;
9     pt tmp=(pt)malloc(n*sizeof(pt)); // 错误
10    free(tmp);
11  }
```

例 271 中第 9 行语句，sizeof(pt)返回的是指针 pt 的大小，而不是指针所指向结构体 student 的内存大小。

217. 指针类型和数组类型不一致

数组是有序数据的集合，数组中的每一个元素都属于同一个数据类型。数组作为参数传递给函数时，实际上传递的是数组中第一个元素（下标为 0 的元素）的地址。如果指针 p 指向一个数组（指向数组中的第一个元素），指针 p 的值就是数组中第一个元素的地址。从某种意义上说，任何一个数组下标运算都等同于一个对应的指针运算。

例 272：

```
1   main ()
2   {
3     int a[3][4];  //a 数组是一个包含 3 个数组类型元素的数组，它的每个数组类型元素又是一个包含 4
                    //个整型元素的数组
4     int *p;
5     p=a; // 错误
6     …
7   }
```

例 272 中，a 是一个二维数组，数组名 a 被转换为一个指向数组的指针，而 p 是一个指向整型

变量的指针，语句“p=a;”试图将一种类型的指针类型赋值给另一种类型的指针，从而产生错误。修改后的程序如例 273 所示。

例 273：

```
1   main ()
2   {
3     int  a[3][4];
4     int  (*p)[4];
5     p=a;
6     …
8   }
```

218. 使用指向数组的指针计算数组大小

指针指向数组，但在计算数组大小时，并不能用指向该数组的指针代替数组进行计算。

例 274：

```
1   main()
2   {
3     int  a[12];
4     int  *p;
5     p=a;
6     sizeof(p);
7     …
8   }
```

例 274 中，sizeof (p)计算的是指针的大小，而不是数组的大小，计算数组的大小应该用 sizeof(a)。

219. 比较指针大小

例 275：

```
1   void CreateDocOfxt(UINT_32 *p1_ptr, UINT_32 *p2_ptr)
2   {
3     if ( p1_ptr > p2_ptr )
4     {
5       …
6     }
7   }
```

例 275 中第 3 行语句，用>比较两个指针的大小。指针之间只能用“==”或者“!=”比较两者是否相等，而不能比较其大小。

220. 直接对指针进行类型转换操作

例 276：

```
1    void CCoolTabCtrl (CDC *pDC, UINT nStyle, bool bActive)
2    {
3       …
4      CBrush brush(GetSysColor(COLOR_3DFACE));
5      nStyle = UINT( pDC +1); // 错误
6       …
7    }
```

例 276 中，第 5 行语句试图把指针类型转换成整型再赋给“nStyle”，导致程序不安全。

修改方法：利用中间变量提取指针数值再进行赋值。

221. 非整型指针变量直接与 NULL 或者 0 进行比较

例 277：

```
1    void SnmpTrapPoll(void)
2    {
3       char *SendCfgCmd = "NMC_CFG_CMD";
4        …
5       if  (SendCfgCmd != NULL ) // 错误
6        {
7         …
8        }
9    }
```

例 277 中第 5 行语句，错误地使用指针变量与 NULL 进行比较。

修改方法：将第 5 行语句修改为：

```
if (SendCfgCmd != (char *)(NULL) );
```

222. 非法使用指针访问变量或结构的成员

根据 ANSI C 语言编程规范，不允许使用指向一个变量（或结构成员）的指针去访问另一个变量（或结构成员）。只有当 ptr+i 是 ptr 所指向对象的组成部分的地址时（如 ptr 指向一个数组，而 ptr+i 指向该数组的一个元素；或 ptr 指向一个结构，而 ptr+i 指向该结构的一个成员），可以使用解引用访问 ptr+i 所指向的变量。

例 278：

```
1    int n;
2    int test[10];
3    struct strt {
4     int e1;
5     int e2;
6    };
7
8    int main()
9    {
10     int i;
11     scanf("%d", i);
12
13     *(&n + 1) = i; // 错误
14     *(test + 1) = i;
15     *(&strt.e1 + 1) = i;     // 错误
16
17     return 0;
18   }
```

例 278 中，在第 13、14、15 行语句的指针操作中，第 13 行语句和第 15 行语句是错误的。在第 13 行语句中，地址“&n + 1”程序员已无法控制；在第 15 行语句中，部分编译器可能不会报错，但使用“*(&strt.e1 + 1)”访问 strt.e2 是不安全的；在第 14 行语句中，因为 test 是数组的地址，因此可以通过这样的方式来访问数组的第 2 个元素。

223．误用 NULL 和整数值 0

在 C++语言中，整数值 0 既是整数类型又是空指针常数，然而，在实际使用过程中，NULL 仅作为空指针常数，而 0 仅作为整数值零，两者不可混用。

例 279：

```
1   #include <cstddef>
2   void f1( int );
3   void foo( )
4   {
5     f1( NULL ); // 错误
6   }
```

例 279 中第 5 行语句，误用 NULL 为整数值 0 作为函数 f1()的参数。

修改方法：将 5 行语句修改为：f1(0);

例 280:

```
1  #include <cstddef>
2  void f1( int* );
3  void foo( )
4  {
5    f1( 0 ); // 错误
6  }
```

例 280 中第 5 行语句，误用整数值 0 为 NULL 作为函数 f1()的参数。

修改方法：将 5 行语句修改为：f1(NULL);

224. 将指针赋值为 NULL

NULL 表示当前指针没有指向任何位置，即允许不指向特定位置的指针存在且合法。在 C++语言中，NULL 不代表 0，不能直接将 NULL 赋值给指针。

例 281:

```
1  #include <stddef.h>
2  void foo( ) {
3  int *lp = NULL; // 错误
4  }
```

例 281 中第 3 行语句，指针变量 lp 错误地被赋值为 NULL。

修改方法：将第 3 行语句修改为：`int *lp = 0;`

225. 使用 delete（）函数删除未分配内存的对象

delete()函数只用于释放内存，如果误用 delete()函数删除未分配内存的对象，将导致程序错误。

例 282:

```
1  class  A
2  {
3    Public:
4       A();
5       ~A();
6     Private:
7       int  a;
8       int  b;
9  };
10      void f()
11 {
```

```
12    A a1;
13    delete(a1); // 错误
14  }
```

例 282 中第 13 行语句，对象 a1 在没有调用 new()函数为其分配内存空间的情况下，就被删除。修改方法：在第 13 行语句前面增加为对象 a1 申请内存空间的语句：

```
A *a1 = new A ();
```

226. 通过基类指针或引用操作派生类数组

虽然类的继承性使得基类的指针或引用能够操作派生类，但在编程时应禁止使用基类指针或引用操作派生类对象组成的数组。因为编译器在通过指针或引用查找派生类数组对象时，仍认为每个数组对象的间隔是 sizeof（基类对象）。事实上，派生类的长度比其基类长，因此，派生类对象占用的内存也比基类对象占用的内存大，在编译过程中，编译器无法正确找到派生类数组中所有的成员。

例 283：

```
1   class BST {
2   public:
3       void cleanBSTArray(BST array[], int numElements)
4       {
5           for (int i = 1; i < numElements; ++i)
6           {
7               array[i] = array[0];
8           }
9       }
10
11      void deleteArray(BST array[])
12      {
13          delete[] array;
14      }
15  };
16
17  class BalancedBST: public BST {};
18
19  void foo()
20  {
21      BalancedBST *p;
22      BST BSTArray[10];
23      BalancedBST bBSTArray[10];
24        …
```

```
25      p->cleanBSTArray(bBSTArray, 10);  // 错误
26      p->deleteArray(bBSTArray);        // 错误
27  }
```

例 283 第 25 行语句和 26 行语句，通过指针来操作派生类数组 bBSTArray。编译器认为数组元素之间间隔大小仍然是 sizeof(BST 对象)，因此，在执行数组元素赋值和删除操作时，程序结果将难以预料。

修改方法：将第 25 行和第 26 行语句分别修改为：

```
p->cleanBSTArray(BSTArray, 10);
p->deleteArray (BSTArray);
```

227. 使用已经释放的句柄

例 284：

```
1   void message_released(const char *name, const char *data1, const char *data2)
2    {
3       char c;
4       mqd_t h;
5       if ((h = mq_open(name, O_RDWR)) != (mqd_t)-1)
6       {
7   …
8           mq_close(h);
9           printf("OK.\n");
10          mq_send(h, "OK.", 3, 2); // 错误
11      }
12  …
13   }
```

例 284 中，第 10 行语句引用了句柄"h"，但该句柄已经在第 8 行语句被释放，引用已经释放的句柄导致了程序出错。

修改方法：在引用任何句柄前检查该句柄是否可用。

228. 解引用指针类型表达式

表达式的结果如果是指针，称这个表达式为指针类型的表达式。解引用指针类型的表达式是指取表达式结果指针地址对应的变量值。避免解引用指针类型的表达式，可以增强程序的安全性和可读性。

例 285：

```
1   int *pFun();
2   void goo()
```

```
3    {
4        int *ptr;
5        int a, b, c;
6        a = *ptr++;      // 错误
7        b = *pFun();      // 错误
8        c = *(ptr + 5);  // 错误
9    }
```

例 285 中第 6、7、8 行语句分别解引用指针类型的表达式。

修改后的程序如例 286 所示。

例 286:

```
1    int *pFun();
2    void goo()
3    {
4        int *ptr;
5        int *ptr2;
6        int a, b, c;
7        ptr++;
8        a = *ptr;
9        ptr2 = pFun();
10       b = *ptr2;
11       ptr2 = ptr + 5;
12       c = *ptr2;
13   }
```

第 5 章 安全缺陷

软件安全缺陷是指软件设计和实现过程中引入的、在数据访问或行为逻辑等方面的疏漏。攻击者可以利用这些缺陷对程序进行攻击，使程序产生意想不到的结果。

程序中的安全缺陷形式多样、种类繁多，如缓冲区溢出、内存泄漏，指针使用、类型转换等也都可能存在安全缺陷，这些内容已在前面的章节中做了介绍。本章重点介绍程序在外部输入、目录、文件和动态资源泄漏等方面存在的安全缺陷。

5.1 外部输入安全缺陷

229. 未经合法性验证的外部输入用于驱动命令执行

一些函数会驱动系统执行相应命令，如 SQL 语句、文件管理函数等，当使用不可信数据（未经合法性验证的数据）作为这些函数的参数时，可能造成安全缺陷。

SQL 注入攻击就是利用 SQL 语法，巧妙地构造 SQL 请求语句，获取超过自己权限的信息，甚至控制服务器，任意对数据库进行编辑。

例 287：

```
1     void SQLInjection(char *name,…)
2     {
3       …
4       EXEC SQL DECLARE SX CURSOR FOR
          SELECT * FROM FinancialData
          WHERE UName = 'name'; // 错误
5       …
6     }
```

例 287 中，第 4 行语句执行一条简单的查询语句，查询出表 FinancialData 中指定用户名字的记录。由于参数 name 是由外部传入的，程序在使用之前没有做合法性检查，因此如果用户恶意构造一些特殊的字符串作为 name 传入，将会得到超过其权限的结果。如：name = 'Tom' or '1' = '1'，则此时查询语句变为：

```
SELECT * FROM FinancialData
WHERE UName = 'Tom' or '1' = '1';
```

该语句的判断条件恒为真，因此程序将返回 FinancialData 的全部记录。

类似地，当不可信数据作为文件管理函数的参数时，也会产生安全缺陷。如 Windows NT4.0 Server 文件管理函数有时会拒绝服务，该缺陷就是由函数接收了精心制作的参数导致内存不能被释放而引起的。

230. 未经合法性验证的外部输入传递给内存分配函数

当使用 malloc()等函数进行内存分配的时候，如果使用不安全数据作为参数决定分配内存的大小，将会给系统带来安全缺陷。

例 288：

```
1   void MemorySet()
2   {
3      int size=0;
4      char *buf = NULL;
5      …
6      printf("Enter memory size to be allocated:");
7      scanf("%d",&size); // 错误
8      buf = malloc(size * sizeof(int));
9      …
10  }
```

例 288 中，malloc()函数的参数 size 是由用户输入的，且没有经过任何合法性检查，因此在第 7 行语句会产生一个外部注入缺陷。例如，攻击者可以利用 size * sizeof(int)操作产生整数溢出来重写任意内存，使用攻击代码的地址来覆盖程序需要跳转的地址，从而实现对程序的控制和破坏。

231. 用未经合法性验证的用户输入访问数组

数组通过其下标对其进行访问，数组下标有不同的设置方式，此处特指使用未经合法性判断的用户输入数据作为数组下标访问数组引起的安全漏洞。当使用函数返回值或用户输入等外部数据作为数组下标时，一定要判断数组下标范围。

例 289：

```
1  int buf[8];
2
3   void putValue(void)
4   {
5     int index ,value;
6     printf("Please input array index:  ");
7     scanf("%d", &index);
8     printf("Please input value of the element: ");
9     scanf("%d", &value);
10    buf[index] = value; // 错误
11  }
```

例 289 中，程序使用用户输入作为数组下标，并为相应的数组元素赋值。当用户输入的下标值大于等于 8 时，第 10 行语句对数组元素的赋值操作会引起缓冲区溢出。

232. 用未经合法性验证的用户输入作为循环边界

循环边界用于指定循环结束的条件，循环边界有不同的设置方式，此处指直接将用户的输入作为循环边界或者作为参数传递给另一个函数作为循环边界。

例 290：

```
1    #define GC_SMALL_SIZE    40
2    #define GC_BIG_SIZE      100
3
4    extern unsigned char buf_small[GC_SMALL_SIZE];
5
6    void StrCopy(char *str_tgt, char *str_src)
7    {
8      int i = 0;
9      char tmp_char;
10     while(str_src != NULL)
11     {
12       str_tgt[i++] = *str_src++;
13     }
14   }
15   void Loop_Bound_Violate()
16   {
17     …
18     char buf_big[GC_BIG_SIZE];
```

```
19      scanf("%s", buf_big);
20      StrCopy(buf_small, buf_big); // 错误
21      …
22   }
```

例 290 中，buf_big 之值从用户输入读取，在函数 StrCopy()中被用作循环边界。如果 buf_big 的长度超过了 buf_small 的大小 GC_SMALL_SIZE，将会产生缓冲区溢出。

又如常见的 for 循环体：

```
for(int i = 0; i<n; ++){}。
```

如果循环边界 n 来自用户输入，但没有对 n 做合法性验证，就可能造成循环无法结束，恶意用户可以利用此缺陷进行拒绝服务攻击。

233. 命令参数注入缺陷

在 C/C++语言中，有些函数以 shell 命令为参数，如 popen()、system()。调用该函数就是执行参数所对应的命令并得到执行结果。如果该参数（字符串）从用户输入得到而未对其长度、内容等进行合法性检查，函数的执行可能产生不可预计的后果，甚至被攻击者利用进行恶意代码攻击。

例 291：

```
1     int main()
2     {
3       char cmd_sys[100];
4       char cmd_pop[100];
5       scanf("%s", cmd_sys);
6       system(cmd_sys);     // 错误
7       …
8       scanf("%s", cmd_pop);
9       popen(cmd_pop, "w");     // 错误
10      …
11      return 0;
12    }
```

例 291 中，由于 system()和 popen()函数的指定执行命令参数是直接来自于用户输入，因此在第 6 行和第 9 行语句都会产生命令参数注入缺陷。

234. 使用危险的进程创建函数

在创建进程时，使用不安全的函数会引入注入缺陷，如 execlp()函数，其函数原型为：

```
int execlp(const char * file, const char *arg[0], …);
```

execlp()函数从 PATH 环境变量所指的目录中查找符合参数 file 的文件名（命令），找到后便执行该文件。

例 292：

```
1    #include<unistd.h>
2    void main()
3    {
4       …
5       execlp("/bin/ls", "ls", "-a", null); // 错误
6       …
7    }
```

例 292 中，由于 execlp()函数的参数（要执行的命令及其参数）均是从外部读取，因此在第 5 行语句处会产生一个注入缺陷。攻击者可以通过修改原文件（命令）及其参数来执行其攻击代码。

修改方法：使用更安全的 CreateProcess()函数来代替 execlp()函数。

5.2 资源泄漏

235. 目录资源泄漏

当目录流与某个目录关联后，在使用完毕时，如果没有关闭该目录流，将会造成目录资源泄漏。

opendir()函数用于打开参数 name 指定的目录，并返回 DIR*形态的目录流，其函数原型为：

```
DIR *opendir(const char *name);
```

函数 closedir()用于关闭 DIR*形态的目录，其函数原型为：

```
int closedir(DIR *dir);
```

例 293：

```
1    static void A()
2    {
3     …
4     DIR* d = opendir("dir.name");      // 错误
5    }
```

例 293 中，使用 opendir()函数打开目录，在使用完毕后没有调用 closedir()函数关闭该目录。

修改方法：增加关闭目录的语句：closedir(d);

236. 动态库资源泄漏

动态库被加载使用完成后，如果没有关闭加载的动态库，将会造成动态库资源泄漏。

dlopen()函数用于动态加载函数和类，其函数原型为：

```
void *dlopen(const char *filename, int flag);
```

dlclose()函数用于关闭动态库，其函数原型为：

```
int dlclose(void *handle);
```

例 294：

```
1   int call_dev_func(char *func_name, int opt_flag, void *value, int locate[])
2   {
3       int ret = -1;
4       void *handle;
5       int (*dev_func)(int, void *, int[]);
6       const char *error;
7       handle = dlopen(dev_lib_path, RTLD_LAZY);
8       if (! handle) {
9        LOG(m_handler, ERROR, "dlopen dll failed\n");
10       return ret;
11      }
12      …
13      }
```

例 294 中，使用 dlopen()函数成功加载动态库，使用后应该调用 dlclose()函数释放该动态库资源。

修改方法：增加释放动态库资源的语句：`dlclose(handle);`

237. 文件资源泄漏

文件被打开使用后，没有关闭打开的文件流，而在很多系统中，允许同时打开的文件数是有限的，因此，当这种现象积累过多时，会造成文件资源泄漏。

以下三个函数都可用于打开指定文件，并返回 FILE *形态的文件流。函数原型分别为：

```
FILE *fopen(const char *path, const char *mode);
FILE *fdopen(int fildes, const char *mode);
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

fclose()函数用于关闭文件，其函数原型为：

```
int fclose(FILE *stream);
```

例 295:

```
1    #include <sys\stat.h>
2    #include <stdio.h>
3    #include <fcntl.h>
4    #include <io.h>
5    int main(void)
6    {
7      int handle;
8      FILE *stream;
9      handle = open("DUMMY.FIL", O_CREAT,  S_IREAD | S_IWRITE);
10     stream = fdopen(handle, "w");  //把流与一个文件句柄相接
11     if (stream == NULL)
12       printf("fdopen failed\n");
13     else
14     {
15       fprintf(stream, "Hello world\n");
16     }
17     return 0;
18   }
```

例 295 中，第 15 行语句在使用完文件流后，未调用 fclose(stream)关闭打开的文件流。
修改方法：在 16 行语句后面增加关闭文件流的语句：fclose(stream);

238. 未判断文件打开操作是否成功

当涉及文件打开操作时，应该判断文件打开操作是否成功，否则可能引起文件 I/O 操作错误。
例 296:

```
1    #include <stdio.h>
2    int foo()
3    {
4        int c;
5        FILE *data = fopen("data.txt", "r");
6        do {
7            c = getc(data);
8            …
9        } while (c != EOF);
10       FILE *data2;
11       data2 = fopen("data2.txt", "r");
12       do {
```

```
13            c = getc(data2);
14            …
15        } while (c != EOF);
16        if (!data2)
17        {
18            …
19            return 0;
20        }
21        return 1;
22    }
```

例 296 中，第 5 行语句和第 11 行语句分别执行打开文件操作，但未对文件打开是否成功进行判断。修改后的程序如例 297 所示。

例 297:

```
1    #include <stdio.h>
2    int foo()
3    {
4        int c;
5        FILE *data = fopen("data.txt", "r");
6        if (data) {
7            do {
8                c = getc(data);
9                …
10           } while (c != EOF);
11       }else{
12           printf( "File opening error" );
13           return 0;
14       }
15       FILE *data2;
16       data2 = fopen("data2.txt", "r");
17
18       if (!data2) {
19           printf( "File opening error" );
20           return 0;
21       }else{
22           do {
23               c = getc(data2);
24               …
25           } while (c != EOF);
```

```
26      }
27      return 1;
28  }
```

239. 加载文件时未使用完整路径

例 298:

```
1   HINSTANCE myLoadLibraryX()
2   {
3     …
4     return LoadLibrary("X.dll"); // 错误
5     …
6   }
```

例 298 中第 4 行语句，由于使用的是相对路径，使得程序可以加载任意路径的动态链接库文件，如果被攻击者利用，程序安全将受到威胁。

修改方法：使用完整路径，如：c:\X.dll，限制其使用条件。

240. 未使用随机变量导致程序安全受到威胁

例 299:

```
1   HANDLE myCreateFile() {
2      Return CreateFile("C:\\test.tmp",
3               GENERIC_READ | GENERIC_WRITE,
4               0,
5               NULL,
6               CREATE_NEW,
7               FILE_TEM,// temporary file
8               NULL);
9   }
```

例 299 中，函数 CreateFile()使用宏"FILE_TEM"作为文件名创建文件，但第 2 行语句中的文件名很容易被攻击者破解并利用，导致程序安全受到威胁。

修改方法：利用函数 GetTempPath()和 GetTempFileName()取得随机的路径和文件名，尽量避免被攻击者破解。

241. 包含空格的可执行文件路径用作函数参数

CreateProcess()等函数使用应用程序名（lpApplicationName）和可执行文件命令行（lpCommand Line）作为参数，其中的参数 lpApplicationName 可以为空。在这种情况下，lpCommandLine 中必须

包含可执行文件的名字。函数解析参数 lpCommandLine 时，将第一个空格前的字符串作为可执行文件的完整路径，如果可执行文件或路径中包含空格，则存在执行非目标文件的风险。

例 300：

```
1    #include <windows.h>
2    #include <stdio.h>
3
4    void Func ()
5    {
6      …
7      LPTSTR szCmdline = _tcsdup(TEXT("C:\\Program Files\\ TestApp -L -S"));    // 错误
8      if( !CreateProcess ( NULL, szCmdline,…) )
9        {
10        …
11        return;
12       }
13       …
14   }
```

例 300 中，第 7 行语句指定了可执行文件的路径和名字："C:\\Program Files" 下的 TestApp.exe，-L 和-S 是可执行文件 TestApp.exe 的参数。如果恶意用户将执行攻击的程序命名为 Program.exe，并放到 C 盘根目录下，每次程序调用例 300 中的函数时，就会执行 Program.exe，而不是 TestApp.exe。要避免这个问题，可以在 lpApplicationName 参数中指定具体应用程序，或者在 lpCommandLine 中将可执行文件的全路径名放在双引号（转义符）中，如下所示：

```
CreateProcess(NULL,"\"C:\\Program Files\\MyApp.exe\" -L -S",…);
```

242. 使用 access()函数直接访问文件

例 301：

```
1   int main() {
2          if (access("/etc/context",w_ok)) // 错误
3          {
4              chown("/etc/getinfo", 0, 0);
5          }
6          return 0;
7   }
```

例 301 中，第 2 行语句使用函数 access()直接访问文件，会导致攻击者在文件创建和访问之间的时间间隙控制文件。

修改方法：在涉及文件的时候，使用文件句柄或者文件描述符调用文件。

243. fopen()和 fclose()函数未成对使用

类中构造函数和析构函数要成对使用 fopen()和 fclose()函数，类中的构造函数如果使用 fopen()函数打开一个文件，那么在类的析构函数中就要调用 fclose()函数关闭该文件，避免资源泄漏。

例 302：

```
1   #include <iostream>
2   using namespace std;
3   class File_ptr
4   {
5       public:
6           File_ptr (const char *n, const char *a)
7           {
8               p = fopen(n,a);
9           }
10          ~File_ptr () {
11          }
12      private:
13          FILE *p;
14  };
```

例 302 中，在 File_ptr 类的构造函数中调用 fopen()函数打开一个文件，在该类的析构函数中没有调用 fclose()函数关闭该文件，造成资源泄漏。

修改方法：在类的析构函数中增加关闭文件语句：`fclose(p);`

244. 管道资源泄漏

当管道被加载，但程序在使用完毕后并未释放该管道资源时，会造成管道资源泄漏。

popen()函数用于加载管道，其函数原型为：

```
FILE *popen(const char *command, const char *type);
```

pclose()函数用于关闭管道，其函数原型为：

```
int pclose(FILE *stream);
```

例 303：

```
1   #include <stdio.h>
2   int main(int argc, char *argv[])
3   {
```

```
4   char buf[128];
5   FILE *pp;
6   if( (pp = popen("ls -l", "r")) == NULL )
7   {
8     printf("popen() error!\n");
9     exit(1);
10  }
11   while(fgets(buf, sizeof buf, pp))
12   {
13    printf("%s", buf);
14   }
15    return 0;
16 }
```

例 303 中，管道使用完毕后，未关闭该管道。

修改方法：在 14 行语句后面增加管道关闭语句：pclose(pp)；

5.3 其他

245．忽略 ANSI 与 Unicode 的区别

编程时一般使用的都是 ANSI 编码，也常常使用 strcpy()、strncat()等标准 ANSI 函数。但在有些情况下（如在 COM 组件中），需要使用宽字符类型 Unicode，如果不注意 ANSI 与 Unicode 的区别，往往在不经意间给程序埋下致命隐患。

例 304：

```
1   #include <string.h>;
2   #define SiZE 200
3   void StrCpyToBuffer()
4   {
5     char buf[SIZE];
6     wchar_t wbuf[SIZE];
7     char *str="string of ANSI Char";
8     wchar_t *wstr= L"string of Unicode Char";
9     …
10    strncpy(buf, str, sizeof buf);
11    wcsncpy(wbuf, wstr, sizeof wbuf); // 错误
12  }
```

例 304 中第 10 行和第 11 行语句，分别使用 strncpy()函数和 wcsncpy()函数对两种类型的字符串

执行复制操作。第 10 行语句不存在问题，但第 11 行语句则有严重的安全隐患。因为 sizeof 计算的是目标对象的字节数。对于 ANSI 字符串，字节数与字符个数是一致的；而对于 Unicode 字符串，字节数则是字符个数的两倍。因此使用“sizeof wbuf”作为 wcsncpy()的参数，会造成内存越界写入。

修改方法：将第 11 行语句修改为：

```
wcsncpy(wbuf, wstr, (sizeof wbuf)/(sizeof wbuf[0]));
```

246. 从 try 块中直接跳转

在进行异常处理时常常使用 try-finally 程序块，如果程序不是顺序执行完 finally 块后再跳转，而是在 try 块中直接跳转，将会强制展开局部堆栈，展开局部堆栈需要 1000 条机器指令，这将严重影响程序的性能。

例 305：

```
1   #include<stdbool.h>
2   void ThrowExc();
3   void FuncFinal();  //finally 块中做一些清理善后工作
4  int Func(char *ptr)
5  {
6    …
7    try
8   {
9    if(ptr)
10   {
11    …
12     return 0; // 错误
13   }
14    ThrowExc();
15   }
16  finally
17   {
18    FuncFinal();
19   }
20  return 1;
21 }
```

例 305 中，如果 ptr 不为 NULL，则程序在第 12 行语句直接返回，程序性能将受影响。为避免这种情况发生，应该使用“leave”退出受保护的 try-finally 块。修改后的程序如例 306 所示。

例 306：

```
1  try
2    {
3     if(ptr)
4     {
5      …
6       leave;
7     }
8      ThrowExc();
9    }
```

247．修改只读字符串

编译器在静态只读内存中分配常量字符串，所以，尝试修改该内存会导致运行时错误（访问冲突或随机崩溃）。程序员一般不会直接修改只读字符串，但如果误将只读字符串用作可写字符串参数传递给某个函数，则可能发生错误。

例 307：

```
1   void StrEd(char *c1, char *c2)
2   {
3     …
4     c1[0] =  'Y';
5     c2[0] =  'N';
6     …
7   }
8   void Func()
9   {
10    char arry[] = "Const string";
11    char *str = "Const string";
12
13    StrEd(arry, str); // 错误
14    …
15  }
```

例 307 中，分别将一个字符数组和字符指针传递给函数 StrEd()，在 StrEd()中修改两个字符串的第一个字符。修改 arry[0]没有问题，但修改 str[0]时将会发生运行时错误。这是因为对于用来初始化字符数组的字符串常量，编译器会在栈中为字符数组分配空间，然后把字符串中的所有字符复制到数组中；而用来初始化字符指针的字符串常量则会被编译器安排到只读数据存储区，并按字符数组的形式来存储。执行“arry[0] = 'Y'”修改的不是只读数据区中的字符串常量，而是由字符串常量复

制而来的存在于栈中的字符数组 arry 的一个元素；但执行“str[0] = 'N'”时却试图修改用于初始化字符指针的位于只读数据区的字符串常量，所以会发生运行时错误。

248. 使用简单加密操作

例 308：

```
1  #include <unistd.h>
2  void myEncrypt(char block[], const char *key) {
3     setkey(mypassword);
4     encrypt(block, 0);
5  …
6  }
```

例 308 中，第 3 行语句利用函数 setkey()直接给出密钥，密钥过于简单，攻击者很容易破解，给程序安全造成威胁。

修改方法：利用 MD5 对密钥进行加密。

249. 使用安全性较低的函数

例 309：

```
1  #include <netdb.h>
2  extern int h_errno;
3  struct hostent * myGethostbyname(const char *name) {
4     return gethostbyname(name); // 错误
5  }
```

例 309 第 4 行语句中，函数 gethostbyname()依靠服务器发送的数据运行，如果攻击者模拟服务器，就会利用发送的数据给程序安全造成威胁。

修改方法：利用类似“TOCTOU”防止攻击的方法，尽量避免直接引用服务器数据。

250. 使用弱加密函数

例 310：

```
1  #include <crypt.h>
2
3  void main()
4  {
5    …
6    crypt();
7    …
8  }
```

例 310 中，第 6 行语句利用函数 crypt()进行加密，但该函数是弱加密函数，会给程序安全造成威胁。

修改方法：使用类似 MD5 加密方法。

251. 未正确处理函数使用的资源

一个现有进程通过调用函数 fork()创建一个新进程。由函数 fork()创建的新进程被称为子进程。子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本，这意味着父子进程间不共享存储空间。由于子进程复制了父进程的堆栈段，两个进程都停留在函数 fork()中等待返回，因此，函数 fork()被调用一次但返回两次，两次返回的唯一区别是子进程中返回 0 值，父进程中返回子进程 ID。

例 311：

```
1   #include <sys/types.h>
2   #include <unistd.h>
3   void bufferclean(void) {
4     pid_t id = fork();
5     if (id == -1) { … }
6     else if (id) {
7       …         // 清除缓冲区
8     } else {
9       …         // 不清除缓冲区
10    }
11  }
```

例 311 中，函数 fork()在使用过程中的中间值和结果没有被清除，如果该函数在敏感位置使用，程序安全将受到威胁。

修改方法：不使用该函数，或者随时清除中间值和结果。

第 6 章 与类有关的编程缺陷

C++语言中的类是某种事物抽象出来的结果，类具有成员变量和成员方法，其成员变量相当于属性，成员方法相当于该类能完成的某些功能。继承性、封装性和多态性是类的三大特性。类的继承性使得只需在新类中增加原有类中没有的成分，就可以在新类中获得其父类的操作和数据结构；封装性将类的属性和方法封装在对象内部，形成一个独立的整体，并尽可能隐蔽对象的内部细节；多态性使得父类中定义的属性或行为被派生类继承之后，可以具有不同的数据类型或表现出不同的行为特性。

C++语言这些独有的特性既给程序员带来了方便，也使程序员容易犯错误，本章总结归纳了 C++语言中与类操作有关的编程缺陷。

252. 基类和派生类位于同一文件内

被用作其他类的基类和其他类的成员函数，必须和使用它的其他类放在不同的文件（头文件或源程序文件）内。

例 312：

源程序文件：source.cpp //source.cpp 文件内存放基类和派生类

```
1    …
2    class A {};
3    class B : public A
4    {
5          A *a;
6    };
```

例 312 中将基类 A 和其派生类 B 放在相同的文件内，程序可读性差。

修改后的程序如例 313 所示。

例 313：

```
头文件：header.h
class A {};    // header.h 文件内存放基类 A;
源程序文件：source.cpp  //source.cpp 文件内存放派生类
1    #include "header.h"
2    class B : public A
3    {
4          A *a;
5    };
```

例 313 中将基类 A 和其派生类 B 放在不同的文件内，这样有利于提高程序可读性和可维护性。

253．被用作返回类型的类未被放在独立的头文件内

将被用作返回类型的类包含在独立的头文件内，可以提高程序可读性和可维护性。

例 314：

```
源程序文件：source.cpp
1    …
2    class A {};
3    A *moo( );
4    class B
5    {
6          A *foo( );
7    };
```

例 314 中类 A 作为返回类型的类未放在独立的头文件内，程序可读性差。

修改后的程序如例 315 所示。

例 315：

```
头文件：header.h
class A {};
源程序文件：source.cpp
1    #include "header.h"
2    A *moo( );
3    class B
4    {
5          A *foo( );
6    };
```

例 315 中类 A 作为返回类型的类独立包含在 header.h 头文件内，这样，可以提高程序的可读性和可维护性。

254. 作为入口参数类型的类未放在独立的头文件内

被用作函数或成员函数入口参数的类放在独立的头文件内，可以提高程序的可读性和可维护性。

例 316：

```
源程序文件： source.cpp
1    …
2    class A {};
3    void moo( A *a );
4    class B {
5    void foo( A *a );
6    };
```

例 316 中类 A 作为函数 moo()和类 B 中成员函数 foo()入口参数的类，未放在独立的头文件内，程序可读性差。

修改后的程序如例 317 所示。

例 317：

```
头文件： header.h
class A {};
源程序文件： source.cpp
1    #include "header.h"
2    void moo( A *a );
3    class B {
4     void foo( A *a );
5    };
```

例 317 中类 A 作为函数 moo()和类 B 中成员函数 foo()入口参数的类，独立包含在 header.h 头文件内，这样，可以提高程序可读性和可维护性。

255. 内联成员函数内部的函数声明未放在独立的头文件内

在内联函数中，用作函数或成员函数的函数声明应该放在独立的头文件内，这样，可以提高程序可读性和可维护性。

例 318：

```
源程序文件： source.cpp
1    void moo( );
2    …
3    class A
4    {
```

```
5      inline void foo( )
6      {
7         moo( );
8      }
9   };
```

例 318 中 moo()函数作为类 A 内联函数 foo()内的函数，未放在单独的头文件内，程序可读性差。修改后的程序如例 319 所示。

例 319：

头文件：header.h

```
void moo( );
```

源程序文件：source.cpp

```
1   #include "header.h"
2   class A
3   {
4      inline void foo( )
5      {
6         moo( );
7      }
8   };
```

例 319 中 moo()函数作为类 A 内联函数 foo()的函数，独立包含在 header.h 头文件内，这样，可以提高程序可读性和可维护性。

256．使用 struct 关键字声明 C++变量

C++语言中建议不要使用 struct 关键字声明变量，以提高程序的可读性。

例 320：

```
1   struct Position_t {…};
2   Position_t Pos;
```

例 320 中先将 Position_t 声明为结构体数据类型，再将变量 Pos 声明为 Position_t 类型的数据，影响程序的可读性。

257．全局变量、常量、枚举类型和自定义类型变量未封装在一个类中

在一个类中封装全局变量、常量、枚举类型和自定义 typedef 类型，可以提高程序的可读性和可维护性。

例 321：

```
1    class A {
2    public:
3       enum Days {yesterday};
4       typedef int MyInt;
5    };
6       static MyInt glob;
```

例 321 中，将全局变量 MyInt glob 放在类 A 外声明，影响程序的可读性。

修改方法：将第 6 行语句放在类 A 中声明。

258. 使用全局函数时未使用::作用域运算符

使用全局函数时，通过::作用域运算符来明确所使用的函数，从而解决全局函数的重名问题，

例 322：

```
1    namespace N {
2       void globalFoo1( );
3    }
4    void globalFoo1( ) {
5    }
6    void globalFoo2( ) {
7       ::globalFoo1( );
8       ::N::globalFoo1( );
9    }
```

例 322 中有重名全局函数 globalFoo1()，通过::作用域运算符可以区分所使用的函数。

259. 公共函数为虚函数

公共函数应为非虚函数。如果继承类需要访问基类，虚函数应为私有成员函数或保护类成员函数。但对于析构函数则不然，因为析构函数有特殊的执行顺序。

例 323：

```
1    class A{
2    public:
3    virtual void foo();
4    void goo();
5    };
```

例 323 第 3 行语句中，foo()是公共函数，不能为虚函数。

修改方法：将虚函数 foo()放在私有成员 private 中。

260．类和其非成员函数接口不在同一命名空间

非成员函数也是函数，如果将非成员函数（特别是操作符和辅助函数）设计成类 X 接口的一部分，那么就必须在类 X 的命名空间内定义，以便正确调用。

例 324：

```
1    namespace N{
2        class X{
3            public:
4            void foo();
5        };
6    }
7    N::X operator+(const N::X&, const N::X&);
```

例 324 中第 7 行语句，X 类的非成员函数操作符和 X 类不在同一命名空间。

修改方法：将 X 类的非成员函数操作符和 X 类放在同一命名空间 N 中。

261．在指针上使用 static_cast

static_cast 是类层次的静态转换，没有运行时类型检查来保证转换的安全性。指向动态对象的指针应避免使用 static_cast。

dynamic_cast 是动态转换，可以在运行时确定真正的类型来保证转换的安全性。指向动态对象的指针使用 dynamic_cast，可以使程序更加安全。

static_cast 的特性使其在程序执行过程中有时是安全的，有时是危险的，而 dynamic_cast 在程序执行过程中始终是安全的。

例 325：

```
1    class B {};
2    class D : public B {};
3    void foo( B *pb, D *pd ) {
4    B *pb2 = static_cast<B*>( pd ); // 错误
5    }
```

例 325 中，第 4 行语句使用 static_cast 定义动态对象的指针，给程序带来了安全隐患。

修改方法：将第 4 行语句修改为：

```
B* pb2 =dynamic_cast <B*>( pd );
```

262．在函数内定义类、结构体或联合体

如果在函数内定义类、结构体或联合体，那么被定义的类、结构体或联合体就相当于一个局部

数据类型，该局部数据类型将不能被编译器重复使用，同时也将导致同类型定义的不相容。

例 326：

```
1   void foo(){
2   class A {
3           …
4       };
5       …
6   }
```

例 326 中第 2 行语句，类 A 定义在函数 foo()内。

修改方法：将类 A 定义在函数 foo()外。

263．未声明引用参数为 const 引用

如果不希望引用参数在被调用的函数内部被修改，那么把参数声明为 const 引用，可以防止当函数返回时变量被意外修改，这样既能够提高程序的可读性，也可以防止程序维护时修改调用函数的数据。

例 327：

```
1   struct Foo {
2       int x;
3       int y;
4   };
5   int A( Foo &f ) // 错误
6   {
7     return f.x;
8   }
```

例 327 中第 5 行语句，A 函数中 Foo 类型的参数为一个不会被修改的引用，因此，应该将其声明为 const 引用而非仅仅是引用。

修改方法：将第 5 行语句修改为：`int A ( const Foo &f );`

264．const 成员函数返回指向类数据的非 const 指针或引用

当 const 放在成员函数的后面时，表示该成员函数是“只读”函数，这时，既不能更改数据成员的值，也不能调用那些能引起数据成员值变化的成员函数，只能调用 const 成员函数。

当声明一个对象为 const 类的类型时，只有 const 成员函数可以调用该对象，const 成员函数要求对象的状态不能在调用函数的时候被修改。

例 328:

```
1   class Test
2   {
3      public:
4         …
5     int *GetI() const        // 错误
6     {
7       return &_i;
8      }
9       …
10  };
```

例 328 第 5 行语句中，GetI()是 const 成员函数，该函数只能返回指向类数据的 const 指针或引用。

修改方法：将第 5 行语句修改为：`const int *GetI1() const`

265. 返回静态数据成员的成员函数定义为静态成员函数

static 成员函数与类是分开的，其函数指针不包含对象信息。一般类中的成员函数都具有附加的隐含实参，即指向该类对象的一个 this 指针。因为 static 成员函数不是任何对象的组成部分，所以 static 成员函数没有 this 形参。被声明为 static 的成员函数只能存取静态的数据成员，这样可以预防无意中修改数据。

例 329:

```
1   class A
2   {
3   public:
4   int f1 ()          // 错误
5    {
6     return m_s;
7    }
8    private:
9      static int m_s;
10  };
```

例 329 中第 4 行语句，f1()成员函数返回的是类的静态数据成员 m_s，但 f1()成员函数是非 static 成员函数。

修改方法：将第 4 行语句修改为：`static int f1 ();`

266. 使用指向指针的指针

指针指向变量和对象的地址，指针变量存放的是其他对象的内存地址。指向指针的指针就是指

针变量中存放的地址是另一个指针变量自身的地址。避免使用指向指针的指针，可以提高程序可读性，降低程序缺陷率。但在下述情况下，可以使用指向指针的指针，即 main()函数的第二个参数，其类型一般为 char*[]。

例 330：

```
1  void foo() {
2    int** a;
3  }
```

例 330 中，第 2 行语句声明了一个指向指针的指针变量 a。

修改方法：声明一个指针类型成员变量的类。

267. 多重继承导致数据成员或成员函数不一致

多重继承，是指一个类可以同时继承多个类，从而使其本身同时具有这些类的特征。在多重继承中，同名的数据成员或成员函数因继承途径不同而在内存中有不同的副本，从而造成数据或函数不一致。为了解决这个问题，C++语言中使用虚基类的概念，即在指定派生类基类时，使用 virtual 关键字指明该基类作为虚基类来继承，这时，从不同的路径继承而来的同名数据成员在内存中就只有一个副本，同一函数名也只有一个映射，这样不仅解决了二义性问题，也节省了内存，避免了数据不一致的问题。

例 331：

```
1  class A {};
2  class B : public A {};
3  class C {};
4  class D: public B, public C {};
5  class E: public A {};
6  class F: public D, public E, public C{};
```

例 331 中，类 F 是由类 D、E、C 多重继承而来的，而 E 类是 A 类的派生类。因 D 类是由类 B、C 多重继承而来的，所以在类 F 对类 D、E、C 多重继承过程中，对 C 类中的所有成员产生二义性。修改后的程序如例 332 所示。

例 332：

```
1  class A {};
2  class B : public virtual A {};
3  class C {};
4  class D: public virtual B, public virtual C {};
5  class E: public A {};
6  class F: public D, public E, public C{};
```

268. 多重继承中存在个非虚基类

类的多重继承需要遵守一个原则，即多个被继承的父类中只能有一个非接口类。

接口类是指没有成员变量并且所有的成员函数都是虚（vitural）函数的类。接口类没有成员变量是因为接口类仅用于描述行为而不是数据，这样，在多重继承时，可以避免数据成员从同一个接口类多次继承。将接口类里所有成员函数定义为虚函数，可以确保程序后面的派生类能够根据需要重新定义函数。

例 333：

```
1    class A
2    {
3     public:
4       void f1();
5    };
6    class B
7    {
8     public:
9       void f1();
10      void f2();
11   };
12   class C: public A, public B
13   { };
```

例 333 中，类 A 和类 B 都不是接口类，而类 C 多重继承类 A 和 B 后，A 和 B 两个类中的同名函数 f1()将产生二义性。修改后的程序如例 334 所示。

例 334：

```
1    class A
2    {
3    public:
4       virtual  void f1()=0;
5    };
6    class B
7    {
8       public:
9       void f1();
10      void f2();
11   };
12   class C: public B, public A
```

```
13  {};
```

269. 构造函数使用全局变量传递数据

C++语言中类变量是按顺序初始化的，即首先完成基类静态变量的初始化，然后是它的派生类，直到所有的静态变量都被初始化。然而，因为静态变量和全局变量都被放在公共内存区，因此，全局变量和静态变量的初始化是不分次序的。如果在类的构造函数中通过全局变量传递数据给静态变量，因两者初始化顺序不确定，程序可能进入不确定的状态。因此，在类的构造函数中不能使用全局变量传递数据。

例 335：

```
1    int a;
2    class A
3    {
4      public:
5      A( )
6      {
7         b = a;
8      }
9      private:
10       int b;
11   };
```

例 335 中，第 7 行语句在类 A 构造函数中使用全局变量 a 传递数据给成员 b，而全局变量 a 和成员 b 的初始化顺序不确定，成员 b 有可能先于 a 被初始化，此时构造函数执行时产生的异常可能使程序进入不确定的状态。

270. 改变虚函数中默认参数值

编译时，编译器对虚函数中默认参数值和虚函数本身的处理是不相同的，虚函数是程序运行时动态绑定的，而其默认参数值是编译阶段静态绑定的。当基类中存在带有默认参数值的虚函数时，派生类不能改变继承而来的虚函数中默认参数的值。

例 336：

```
1    class A
2      {
3       virtual int fun(int i=1024)
4        {
5        return i;
6        }
7      };
```

```
8  class B :public A
9    {
10    virtual  int fun(int i=2048) // 错误
11     {
12       return  i;
13     }
14   };
```

例 336 中第 10 行语句，派生类 B 更改了基类 A 中虚函数的默认参数值，即将默认参数 i 的值修改为 2048。

271. 构造函数或析构函数调用类自身的虚成员函数

当类中虚成员函数被类自身的构造函数或析构函数调用时，为了避免在构造函数和析构函数中该虚函数的“多态”发生，将虚函数自动变成了普通函数。

例 337:

```
1    class  A
2    {
3      A() {f();}
4      ~A() {f();}
5      virtual f();
6    };
7    class  B : public A
8    {
9      B() {}
10     virtual f();
11   };
12   main()
13   {
14     A  *t=new B;      // 错误
15     delete t;        // 错误
16   }
```

例 337 中，第 14 行语句在使用 new()函数创建 B 实例时，调用 A 的构造函数，但是在 A 的构造函数中，被调用的是 A::f()而不是 B::f()。同样，第 15 行语句使用 delete()函数释放 t 时，将调用 A::f ()而非 B::f ()。

272. 通过基类指针删除其派生类对象

C++语言标准中规定，通过基类的指针删除派生类对象时，如果基类没有虚析构函数，那么结果将是不确定的，可能导致程序崩溃。

如果某个类包含虚函数，一般表示该类将作为一个基类来使用。如果该类的析构函数没有定义为虚函数形式，那么不能通过该类的指针删除其派生类的对象。

例 338：

```
1    class A
2     {
3       public:
4          virtual void f();
5          ~A( );
6     };
7    class B : public A{};
8    main()
9    {
10     A  *t=new B;
11     delete t; // 错误
12   }
```

例 338 中，第 11 行语句试图通过基类的指针删除派生类的对象（t 是一个指向派生类的基类指针），但该基类并没有虚析构函数，所以导致程序失败。

修改方法：把类 A 中的析构函数变为虚析构函数：virtual ~A( );

273. 派生类指针指向基类对象

C++语言允许基类指针指向其派生类对象，但不允许一个派生类指针指向其基类对象。

例 339：

```
1    class B
2    {
3       public: void  f();
4    };
5    class D : public B
6    {
7       public: void  f();
8    };
9     main( )
10   {
11      D *d = new B; // 错误
12      delete  d;
13   }
```

例 339 中，第 11 行语句将派生类指针指向基类对象，导致程序错误。

274. 数据成员访问权限设置不规范

C++语言采用三个关键词设置其成员的访问权限，分别是 public、private 和 protected。public 意味着其后的数据或函数对该类的实例开放；private 则说明其后的数据或函数只能被该类内部的成员函数引用；protected 则说明其后的数据或函数只对该类本身和其子类可见。

因为 public 或 protected 类型的数据成员封装性很差，因此类中应该避免使用这两种类型的数据成员。

例 340：

```
1   class A
2   {
3     public:
4      int i;
5      f();
6   };
```

例 340 中，类 A 的数据成员被声明为 public 类型，使得该类的数据成员封装性很差，应该声明该类中的数据成员为 private 类型。

275. 构造函数调用类自身的方法

声明类的对象时，类的构造函数不仅为该对象分配存储空间，而且也对该对象进行必要的初始化操作。在类的构造函数中调用类自身的方法，可能导致类的对象在被使用时还处于未完全初始化的状态。

例 341：

```
1   class C
2   {
3     public:
4       C();
5       void f();
6       private:
7       int a;
8   };
9   C::C()
10  {
11    f();          // 错误
12  }
```

例 341 中，类 C 的构造函数调用类的 f()方法，而 f()方法需要类的对象完全初始化作为其调用的前提条件。在这种情况下，有可能导致程序初始化不完整，或互相等待初始化而死锁。

276. 派生类的拷贝构造函数定义错误

派生类的拷贝构造函数必须能够拷贝基类和自身所有的数据成员，所以当为派生类定义拷贝构造函数时，需要显式的调用基类的拷贝构造函数。

通过下面的方法将基类的拷贝构造函数传递给其派生类：

Derived(const Derived& d) : Base(d)，其中类 Derived 是 Base 的派生类。

例 342：

```
1   class Base
2   {
3       public:
4           Base();
5           Base (int x) : base_member (x) {}
6           Base (const Base& rhs) : base_member
7         (rhs.base_member) {}
8       private:
9           int base_member;
10  };
11  class Derived : public Base
12  {
13  public:
14          Derived (int x, int y, int z) : Base (x),
15          derived_member_1 (y),
16          derived_member_2 (z) {}        // 错误
17          Derived(const Derived& rhs)
18          {
19          }
20      private:
21          int derived_member_1;
22          int derived_member_2;
23  };
```

例 342 中，第 16 行语句没有将基类的拷贝构造函数传递给派生类 Derived。正确的派生类拷贝构造函数语句是：

```
Derived(const Derived& rhs): Base(rhs); //复制基类的成员变量;
derived_member_1 (rhs.derived_member_1); //复制派生类自身的成员变量;
derived_member_2 (rhs.derived_member_2); //复制派生类自身的成员变量。
```

277. 由成员函数返回指向类数据的非 const 句柄

指向类数据的句柄一般有三种形式（这里的类数据不包括类中的静态数据）：①指向成员变量的引用；②指向成员变量的指针；③在构造函数中指向数据分配或在析构函数中指向数据释放的指针/引用。因为句柄比对象生命周期长，一旦使用成员函数返回指向类数据的非 const 句柄，则可能存在运行风险。

例 343：

```
1    class Test
2   {
3     public:
4         Test(int & p) : _i(p)
5         {
6             _k = new int;       }
7            ~Test()
8         {
9               if (_k) {
10               delete _k;
11           }
12        }
13            int *GetI1()
14          {
15      return &_i; // 错误
16    }
17      int& GetI2() const
18    {
19      return _i; // 错误
20     }
21     protected:
22      const int *GetI3() const
23      {
24          return _k; // 错误
25        }
26    private:
27        int & _i;
28     int * _k;
29   };
```

例 343 中第 15、19、24 行语句，在类 Test 的成员函数中分别返回了指向类数据的非 const 句柄。

278. 非 POD 对象使用 memcpy()或 memcmp()函数

在 C++语言中，常常把传统 C 风格的数据类型叫做 POD（Plain Old Data）对象，即一种古老的纯数据。在 C 语言中根本没有 POD 这一概念，因为 C 语言中的所有对象都是 POD。对于 POD 对象而言，其二进制内容是可以随意复制的，无论在什么地方，只要其二进制内容存在，就能准确无误地还原出 POD 对象。正是由于这个原因，对于任何 POD 对象，都可以放心大胆地使用 memset()、memcpy()、memcmp()等函数对对象的内存数据进行操作。然而，对于非 POD 对象，使用 memcpy()、memcmp()函数或者其他类似的内存初始化函数，会导致内存和资源的泄漏或者是无法预知的行为。

例 344：

```
1   #include <memory.h>
2   class A {
3      int *p;
4   };
5   class B {
6      int *p;
7   };
8   void foo( ) {
9      A p1;
10     B p2;
11     memcpy( &p1, &p2, sizeof( p1 ) );   // 错误
12  }
```

例 344 中，指向类 A 的指针 p1 和指向类 B 的指针 p2 都是非 POD 对象，不能使用 memcpy()函数在不同对象间拷贝。

279. 类中存在指向动态内存的指针时，未定义拷贝构造函数、赋值函数和析构函数

当类中存在指向动态内存的指针时，如果这个类未定义拷贝构造函数、赋值函数和析构函数，类会产生默认的拷贝构造函数、赋值函数和析构函数，而这些默认的函数，只是一种“浅”赋值、“浅”拷贝。说他们“浅”，是因为他们只是简单的赋值和拷贝，可能导致程序崩溃。例如，类的对象 A 中有一个指针成员变量 p 指向一块动态申请的内存区域，当对象 A 复制给对象 B 时，对象 B 中的指针变量将会得到和对象 A 中的指针一样的地址值。如果对象 A 中指针 p 指向的内存空间被释放，那么对象 B 中的指针指向的位置也一同被释放（因为他们指向同一片内存空间），从而导致程序崩溃。赋值函数和析构函数也是同样的道理。

因此，当类中有指向动态内存的指针时，类中需要显式定义拷贝构造函数、赋值函数和析构函数。

例 345：

```
1   class A
```

```
2   {
3   public:
4       A(int size) {
5           p = new char[size];
6       }
7       A(const A& other) {
8           p = new char[strlen(other.p) + 1];
9           strcpy(p, other.p);
10      }
11      ~A(){
12          delete []p;
13      }
14  private:
15      char *p;
16  };
```

例 345 类 A 构造函数中，指针 p 通过 new()函数申请内存空间，因此需要在类 A 中定义拷贝构造函数、赋值函数和析构函数。

修改方法：在类 A 中增加赋值函数。

280. 拷贝构造函数未拷贝所有的数据成员及其基类型

类因其继承性可能含有很多数据成员，这些数据成员既包括派生类本身的数据成员，也包括继承层次结构中基类的数据成员。因此，拷贝构造函数必须拷贝所有的成员，包括其中的基类型，这样才能确保类在拷贝对象时能够正确地处理数据成员及其基类型。

例 346：

```
1   class Base
2   {
3       public:
4           Base();
5           Base (int x) : base_member (x) { }
6           Base (const Base& rhs) : base_member (rhs.base_member) {}
7       private:
8           int base_member;
9   };
10  class Derived : public Base
11  {
12      public:
13          Derived (int x, int y, int z) : Base (x);
```

```
14          derived_member_1 (y);
15          derived_member_2 (z) { };
16        Derived(const Derived& rhs) // 错误
17          {
18          }
19     private:
20          int derived_member_1;
21          int derived_member_2;
22  };
```

例 346 中，第 16 行语句拷贝构造函数未拷贝基类型的数据成员。

修改方法：将第 16 行语句修改为：

```
Derived(const Derived& rhs) : Base(rhs)
```

281. 赋值运算符未对所有成员赋值

在继承的层次结构中，某个特定类可能含有很多数据成员。因此，其赋值运算符必须对所有的成员，包括基类型中的成员赋值。

例 347：

```
1   typedef unsigned int int32;
2   class Base
3   {
4   public:
5      Base (int32 x) : base_member (x) {}
6      Base &operator=(const Base& rhs)
7      {
8         if (this != &rhs)
9         {
10           base_member = rhs.base_member;
11        }
12        else
13        {
14        }
15        return *this;
16     }
17   private:
18     int32 base_member;
19  };
20   class Derived : public Base
```

```
21  {
22    public:
23      Derived (int32 x, int32 y, int32 z) : Base (x),
24          derived_member_1 (y),
25          derived_member_2 (z) {}
26    Derived& operator=(const Derived& rhs)
27         {
28        if (this != &rhs)
29        {
30        derived_member_1 = rhs.derived_member_1;
31            derived_member_2 = rhs.derived_member_2;
32        }
33        else
34        {
35        }
36        return *this;
37      }
38    private:
39      int32 derived_member_1;
40      int32 derived_member_2;
41  };
```

例 347 中，类 Derived 的赋值运算符未对基类型中的成员赋值。

修改方法：在类 Derived 的赋值运算符中为基类型中的成员赋值，即在 30 行语句之前增加语句：Base::operator=(rhs);

282. 从虚类派生类

例 348：

```
1   class B {};
2   class D: public virtual B {}; // 错误
```

例 348 第 2 行语句中，类 D 从虚类 B 派生而来。

修改方法：将第 2 行语句修改为：

```
class D: virtual public B {};
```

283. 被重定义的虚函数缺少 virtual 关键字

当派生类对基类中的虚函数进行重新定义时，必须保证派生类中重定义的函数也是虚函数，并且与基类的虚函数具有相同的形参个数和形参类型。

例 349：

```
1    class A {
2    public:
3        virtual int foo( );
4    };
5
6    class B : public A {
7    public:
8        int foo( );          // 错误
9    };
10
11   int B::foo() {
12       return 5;
13   }
```

例 349 中，类 B 是类 A 的派生类，在类 B 中重定义类 A 中虚函数 foo()时缺少 virtual 关键字。修改后的程序如例 350 所示。

例 350：

```
1    class A {
2    public:
3        virtual int foo( );
4    };
5
6    class B : public A {
7    public:
8       virtual int foo( );
9    };
10
11   int B::foo() {
12       return 5;
13   }
```

284. 纯虚函数覆盖错误

纯虚函数只能覆盖纯虚函数，不能覆盖非纯虚函数。

例 351：

```
1    class A
2    {
```

```
3       public:
4       virtual void foo ( );
5   };
6   class B: public A
7   {
8       public:
9       virtual void foo ( ) = 0; // 错误
10  };
```

例 351 中，类 A 中的 foo()为虚函数，而类 B 中 foo()为纯虚函数。B 类作为 A 类的派生类，派生类 B 中的纯虚函数 foo()将覆盖基类中的虚函数 foo()。

修改方法：将类 A 的 foo()函数声明为纯虚函数：`virtual void foo ( ) = 0;`

285. 派生类构造函数未显式调用基类的构造函数

例 352：

```
1   class A
2   {
3       public:
4           A ( )
5           {
6           }
7   };
8   class B : public A
9   {
10      public:
11          B ( ) // 错误
12          {
13          }
14  };
```

例 352 中，类 B 是类 A 的派生类，第 11 行语句中类 B 中的构造函数 B()未显式调用基类 A 的构造函数 A()。

修改方法：将第 11 行语句修改为`B():A()`

286. 在拷贝构造函数内修改全局或静态成员变量

禁止在拷贝构造函数内修改全局或静态成员变量。

例 353：

```
1   class A
```

```
2   {
3   public:
4       A ( A const & rhs )
5       : m_i ( rhs.m_i )
6       {
7           ++m_static;   // 错误
8       }
9   private:
10      int m_i;
11      static int m_static;
12  };
```

例 353 第 11 行语句中，声明 m_static 是类 A 的静态成员变量；第 7 行语句在类 A 的拷贝构造函数内，通过自增运算符修改 m_static 的值。

287. 抽象类中拷贝赋值运算符未被声明为 protected 或 private

在抽象类中，如果未将拷贝赋值运算符设置为 protected 或 private，将允许从类的层次结构之外访问该类，而非从衍生类访问该类，这样极易破坏类的封装性，引起程序异常。

例 354：

```
1   class B1
2   {
3      public:
4          virtual void f( ) = 0;
5          B1 & operator= ( B1 const & rhs );  //错误
6   };
```

例 354 第 5 行语句中，类 B1 的拷贝赋值运算符未被声明为 protected 或 Private。

修改后的程序如例 355 所示。

例 355：

```
1   class B1
2   {
3      public:
4          virtual void f( ) = 0;
5   protected:
6          B1 & operator= ( B1 const & rhs );
7   };
```

288. 类中存在包含单个泛型参数的模板构造函数时，未声明拷贝构造函数

例 356：

```
1    typedef signed int int32_t;
2    class A                          // 错误
3    {
4       public:
5          A ( );
6          template <typename T>
7          A ( T const & rhs )
8          : i ( new int32_t )
9          {
10            *i = *rhs.i;
11         }
12      private:
13            int32_t * i;
14   };
```

例 356 第 6 行语句中声明了单个泛型参数的模板构造函数，但在类 A 中未声明拷贝构造函数。

修改方法：类 A 中增加拷贝构造函数声明：`A ( A const & rhs );`

289. 含有虚函数的类通过值方式传递对象

虚函数只对指针和引用有效，只要类中含有虚函数，就只能通过指针或引用形式传递或返回该类的对象。

例 357：

```
1    class A {
2    public:
3       virtual void f();
4      g(A a)
5      {
6        return  a;
7    }
8    };
```

例 357 中，类 A 成员函数中含有虚函数 f()，但在其另一个成员函数 g()中却使用值传递该类的对象。修改后的程序如例 358 所示。

例 358:

```
1    class A {
2    public:
3        virtual void  f();
4     A& g(A &a)
5     {
6        return  a;
7     }
8    };
```

290. 无默认构造函数的类声明为类数组

无默认构造函数的类不能动态分配数组的元素类型，因为类中如果没有默认构造函数，程序中就不能声明类数组。

无参构造函数可以作为默认构造函数，如果构造函数带有参数，而且这个参数具有默认值，那么这个构造函数也可以作为默认构造函数。

例 359:

```
1    class c {
2     public:
3      c(int i, int j);
4     };
5     int main()
6     {
7       c a[10]; // 错误
8        …
9     }
```

例 359 中，类 c 中并没有默认的构造函数，但在第 7 行语句却声明了一个类数组。

修改方法：为类 c 定义一个默认的构造函数，即将第 3 行语句修改为：`c();`或 `c(int i=2; int j=3);`

291. 同一联合体的不同成员使用相同对象相互赋值

联合体的特征是其所有数据变量共享同一段内存。当同一联合体的不同成员使用相同对象相互赋值时，因联合体成员间内存区域重叠，将导致程序行为异常或数据丢失。

例 360:

```
1    #include <string.h>
2    union U {
3       int        iValue;
```

```
4     long      lValue;
5     double    dValue;
6   };
7
8   int main( ) {
9     union U a, b;
10    union U *p;
11    (void)memcpy( &a.dValue,  &a.lValue, 8 );//错误
12    (void)memcpy( &p->dValue, &p->iValue, 8 );//错误
13    return 0;
14  }
```

例 360 中，联合体 U 的三个成员 iValue、lValue 和 dValue 共享内存区域，在第 11、12 行语句中，联合体 U 内不同成员使用相同对象相互赋值。

修改后的程序如例 361 所示。

例 361：

```
1   #include <string.h>
2   union U {
3     int       iValue;
4     long      lValue;
5     double    dValue;
6   };
7
8   int main( ) {
9     union U a, b;
10    union U *p, *q;
11    (void)memcpy( &a.dValue,  &b.lValue, 8 );
12    (void)memcpy( &p->dValue, &q->iValue, 8 );
13    return 0;
14  }
```

292. 非成员对象或具有静态存储期的函数的标识符名称重用

在一个程序里静态对象或者函数标识符不能被重用。带有静态存储期的对象或者函数标识符必须是唯一的。

例 362：

```
1   static float a;
2   static void foo()
3   {
```

```
4   int a;   // 错误
5   int foo;  //错误
6   }
7
8   static float a;
9   static void foo()
10  {
11  static int a ;  // 错误
12  static int foo; //错误
13  }
```

293．union 使用错误

union 是一种特殊的类，也是一种构造类型的数据结构。在 union 内可以定义多种不同的数据类型。union 类型的变量中，允许装入该 union 所定义的任何一种数据，所以 union 变量的长度等于各成员中最长的长度。union 的特征是其所有数据变量共享同一段内存，所谓的共享不是指把多个成员同时装入一个联合变量内，而是该联合变量可被赋予任一个成员值，但每次只能赋一种值，赋新值时将覆盖旧值。

使用联合体时要注意以下两种情况。

（1）由于联合体内的成员共享内存，而静态数据成员或引用成员均不能共享内存，所以不能使用这些类型的成员。

（2）联合体内不允许存放带有构造函数、析构函数、复制操作符等的类，因为这些类对象共享内存时，编译器无法保证这些对象不被破坏。

为了防止 union 使用错误，应尽量避免使用联合体。

例 363：

```
1   #include <iostream>
2    void main()
3    {
4     union t{
5       int test;
6       char c;
7      } a;
8     a.test=5;
9     a.c='b';
10    cout<<a.test<<endl;
11    cout<<a.c<<endl;
12   }
```

例 363 中，联合体 union 的成员共用一段内存，这样，一个成员被赋值整个内存就会改变，程序只能输出最后赋值成员的值，而无法输出 a.test 的值。

294. 不同源文件对象间相互调用

当同一工程中存在不同源文件的两个对象 X 和 Y，Y 的构造函数调用 X 的某些方法时，如果使用不当，会造成严重后果。例如，Y 对象的构造函数调用 X 对象的某些方法时，如果 X 对象所在的编辑单元先被初始化，这种对象之间的调用就不会出现问题；如果 Y 对象所在的编辑单元先被初始化，那么当 Y 的构造函数调用 X 对象的方法时，X 对象还没有被构造函数创建，在这种情况下，对象之间的调用将引起程序崩溃。

例 364：

源文件 x.cpp 中定义了 x 对象：

```
File x.cpp
#include "c.hpp"
c x。
```

源文件 y.cpp 中文件定义了 y 对象：

```
File y.cpp
#include "b.hpp"
b y。
```

类 b 的构造函数为：

```
File b.cpp
1    #include "b.hpp"
2    b::b()
3    {
4     …
5     x.f(); //x 是类 c 的对象，调用类 c 的成员函数 f();
6     …
7    }
```

例 364 中，由于对象 x 和 y 位于不同的源文件中，并且程序未对 x 和 y 的创建顺序作出说明，因此可能出现先创建 y，从而导致程序崩溃的情况。为了避免出现这种情况，可以通过返回类.c 对象引用的全局函数"→x()"，取代全局类 c 的对象 x，如例 365 所示的 x.cpp 文件，同时将例 364 中第 5 行语句修改为：x().f();

例 365：

```
File x.cpp
1    #include "c.hpp"
```

```
2  C& x()
3  {
4    static c *p = new c();
5    return *p;
6  }
```

295. 将数组视为多态

数组操作几乎都包含指针运算，因此，数组和多态不能混合使用。

例 366:

```
1   class BST {
2   public:
3   void cleanBSTArray(BST array[], int numElements)
4   {
5     for (int i = 1; i < numElements; ++i)
6     {
7       array[i] = array[0];
8     }
9   }
10  void deleteArray(BST array[])
11  {
12     delete [] array;
13  }
14  };
15  class BalancedBST: public BST {};
16  void foo()
17  {
18    BalancedBST *p;
19    BST BSTArray[10];
20    BalancedBST bBSTArray[10];
21    p->cleanBSTArray(bBSTArray, 10); // 错误
22    p->deleteArray(bBSTArray); // 错误
23  }
```

例 366 中，第 21 行和第 22 行语句将数组 bBSTArray 视为多态。

修改方法：将第 21 行和 22 行语句分别修改为：

```
p->cleanBSTArray(BSTArray, 10);
p->deleteArray(BSTArray);
```

296. 类模板和函数模板实例化时参数个数错误

C++语言为实现代码重用提供了模板机制。使用模板，程序员可以利用一个函数或类的定义产生不同的版本以应对不同的数据类型。在对模板进行实例化时，实参个数要正确。

例 367：

```
1  template <class T, class S>
2  class CLS{};
3  void Func()
4  {
5     A<int, int> cls1;
6     A<int, int, int> cls2; // 错误
7     A<int> cls2;              // 错误
8  }
```

例 367 中，第 5 行语句对类模板的实例化是正确的，而第 6 行语句和第 7 行语句则是错误的。因为类模板定义的参数个数为 2，而在第 6 行语句实例化时使用了 3 个参数，第 7 行语句实例化时只有一个参数，只给出了 T 的类型，没有指定 S 的类型。

也可以在模板定义中使用默认的形参类型，如例 368 所示。

例 368：

```
1  //使用默认的形参类型
2  template <typename T, typename S = int>
3  class CLS{};
4
5  void Func()
6  {
7     A<int> cls1;
8  }
```

例 368 中，显式地给出了 S 的类型 int，T 使用默认的类型 int，因此该 CLS 类的实例化是正确的。

在对函数模板进行实例化时，需要显式指定实参的类型或者通过模板参数推演（template argument deduction）确定实参的类型。

例 369：

```
1  //函数模板
2  template <typename T, typename S >
3  void f(T, S);
4
```

```
5   void Func()
6   {
7     f<int, int> (1, 2);
8     f<int> (3, 2);
9   }
```

例 369 中，第 7、8 两行语句对函数模板 f 的实例化都是正确的。第 7 行语句显式指定了两个参数的类型，第 8 行语句显式指定了 T 的类型 int，函数调用时通过模板参数推演确定了 S 的类型为 int。

例 370：

```
1    //函数模板
2    template <typename T, typename S >
3    T f(S);
4
5   void Func()
6   {
7     f<int> (a);
8     f(a); // 错误
9   }
```

当函数模板的返回类型中使用类型形参时，因为在函数调用中不会出现与之匹配的实参，因此实例化时无法通过模板参数推演确定该参数的类型，必须显式指定该参数的类型。在例 370 中，第 7 行语句的实例化是正确的（显式指定 T 的类型 int，S 的类型通过模板参数推演得到），而第 8 行语句的实例化是错误的。

297. 含有非静态指针的类未通过引用传递类对象

含有指针成员的对象使用编译器生成的拷贝构造函数传递类对象时，是以值传递形式进行的，这样将导致拷贝成员时指针逐位拷贝的问题，如果以引用来传递对象就可以避免该问题。当类中含有非静态指针并且没有声明拷贝构造函数时，应使用引用来传递类对象。

例 371：

```
1   class A;
2   void f (A a);
3   class A
4   {
5   int *x;
6   };
7   void foo(void)
8   {
9   A a;
```

```
10   f (a); // 错误
}
```

例 371 中，类 A 含有非静态指针 x，且没有声明拷贝构造函数，在第 10 行语句以值方式传递类对象，可能导致位拷贝。

修改后的程序如例 372 所示。

例 372：

```
1    class A;
2    void foo1(A &a);
3    class A
4    {
5    int *x;
6    };
7    void foo(void)
8    {
9    A a;
10   foo1(a);
11   }
```

298．拷贝构造函数参数形式错误

类中如果没有定义拷贝构造函数，编译器会自动为类生成一个默认的拷贝构造函数。拷贝构造函数的作用就是用已经实例化的该类对象实例化该类的另一个对象。对于某个类 C 而言，其拷贝构造函数有如下四种形式：

① C&;

② const C&;

③ volatile C&;

④ const volatile C&。

如果程序中某个函数只含有上述四种拷贝构造函数的参数形式或虽有其他参数但其他参数都有默认值，那么这个函数就是拷贝构造函数，类中可以有多个拷贝构造函数。

例 373：

```
1    class C
2    {
3      C(const C& A, int);       //类中声明的构造函数
4    };
5    C::C(const C& , int i =0)  //类体外实现的拷贝构造函数
6    {
7      …
```

```
8    }
```

例 373 中，类 C 体内并没有显式的声明一个拷贝构造函数，编译器将为其隐式声明一个拷贝构造函数，但在类体外却定义了拷贝构造函数的实现。当程序调用该类的拷贝构造函数时，将产生错误。修改后的程序如例 374 所示。

例 374:

```
1    class A
2    {
3        A(const A&, int i=0);  //类的拷贝构造函数
4        A(const A&);
5    };
6
7    A::A(const A& C, int i)  //类体外实现的构造函数
8    {
9         …
10   }
```

299．static 或 const 成员函数未被定义为 static 或 const 形式

声明成员函数为 static 或者 const 形式，能够限制其存取非静态的数据成员，从而防止修改数据成员的值。因此，如果成员函数可以定义为 static 或 const 形式时，就应该将其定义为 static 或 const 形式。

例 375:

```
1    class A
2    {
3     public:
4     int f1 ()
5     {
6     return a;
7     }
8    int f2 ()
9     {
10    return b;
11   }
12    private:
13    const int b;
14    static int a;
15    };
```

例 375 中，成员函数 f1()因为返回 static 类型的数据成员 a，因此可以定义为 static 类型；成员函数 f2()因为返回 const 类型的数据成员 b，因此可以定义为 const 类型。

修改方法：定义函数 f1()为 static、函数 f2()为 const。

第 7 章 其 他

C/C++语言在使用过程中存在各种各样的陷阱与缺陷，前 6 章从几个不同的侧面对其进行了归纳总结，但难免挂一漏万。本章介绍 C/C++语言在其他方面存在的陷阱与缺陷以及优化方法。

7.1 预处理

300．#include 指示符未在文件头部

在程序文件中，所有的#include 语句都应该被组织在一起并放在靠近文件头的位置，在一个文件中，#includ 语句之前的语句只能是预处理指令或者注释。

例 376：

```
1    void foo();
2    int g;
3    #include "test.h"  // 错误
```

例 376 中，#include 语句在程序的中间，没有在文件头部。

修改方法：将第 3 行语句移到文件头部。

301．未在全局的命名空间中使用#define 或#undef 宏

尽管在程序文件的任何地方放置#define 或者#undef 都是合法的，建议不要将其放在程序块中。

例 377：

```
1    void foo( int *x ) {
2    #define CHECKPARAM(p)  (p != 0)
```

```
3     if (CHECKPARAM(x)) {
4       …
5     }
6   #undef CHECKPARAM
7   }
```

修改后的程序如例378所示。

例378：

```
1   #define CHECKPARAM(p)  (p != 0) // 在任何块的外面定义宏
2   void foo( int *x ) {
3     if (CHECKPARAM(x)) {
4       …
5     }
6   }
7   #undef CHECKPARAM              // 在块的外面撤销宏定义
```

302. 定义类函数宏错误

尽管宏比函数的处理速度快，但是函数提供了一种更加安全和更加稳健的机制，非常适合于检查函数参数的类型。

例379：

```
1   #define SUM(A,B)  ((A)+(B))  // 错误
2
3   void foo( int x, int y ) {
4       …
5     SUM( x, y );
6    …
7   }
```

修改方法：将宏定义的SUM修改成函数定义形式。

303. 传递给函数宏的实参包含预处理指令符号#

如果任何参数与预处理相似，那么当发生宏置换时，后果是不可预测的。

例380：

```
1   #define MACRO1(x)
2   #define MACRO2(x, y)
3     …
4   void foo( void ) {
```

```
5       int i = 0;
6       MACRO1( #foo );       // 错误
7       MACRO2( i, #foo );    // 错误
8       MACRO2( i, "#foo" );  // 错误
9   }
```

例 380 中，第 6、7、8 行语句中的函数 MACRO1 和 MACRO2 包含带有预处理指令符号#的实参 foo。

修改方法：将实参修改为普通变量类型的实参。

304．defined 预处理运算符错误

“defined (标识符)”和“defined 标识符”是两种仅有的可允许的预处理运算符定义形式，其他形式的预处理运算符定义将使程序的行为未知。

例 381：

```
1   #if defined X > Y                // 错误
2   #endif
3   #define DEFINED defined
4   #if DEFINED(X)              // 错误
5   #endif
```

修改后的程序如例 382 所示。

例 382：

```
1   #if defined X
2   #endif
3   #if defined (X)
4   #endif
```

305．#endif 预处理指令与#if 或 #ifdef 不匹配

#endif 预处理指令与#if 或 #ifdef 要匹配。

例 383：

```
1   #define A
2   #ifdef A
3   #include "file1.h"
4   #endif
5   #if 1
6   #include "file2.h"
```

例 383 中，#if/ifdef 的个数与#endif 的个数不匹配。

修改方法：在第 6 行语句后面加上#endif 语句。

306．单个的宏定义中出现多个#或##操作符

有关#和##预处理操作符，其赋值的顺序一直存在争议。为了避免这类问题，在单个的宏定义中，这两个操作符仅允许出现一次，例如一个#或者一个##或者两者均无。

例 384：

```
1   #define TEST1(A,B,C) A # B # C    // 错误
2   #define TEST2(A,B,C) A ## B # C   //错误
3   #define TEST3(A,B,C) A ## B ## C   // 错误
```

修改后的程序如例 385 所示。

例 385：

```
1   #define TESTa(A,B) A # B
2   #define TESTb(A,B) A ## B
```

307．重用标准库中宏和对象的名称

程序员使用新版本标准库中的宏或者对象时，这些修改过的宏或者对象需要有一个新的命名，以区分新旧标准宏或者对象。

程序中不允许使用下面给出的两类中的名称。

（1）来自 C 标准库头文件的宏和 typedef 名称，例如 assert.h，complex.h，ctype.h，errno.h，float.h，iso646.h，limits.h，locale.h，math.h，setjmp.h，signal.h，stdarg.h，stddef.h，stdio.h，stdlib.h，string.h，time.h，wchar.h，wctype.h，stdint.h，inttypes.h，fenv.h，stdbool.h，tgmath.h。

（2）以下划线字符开头的标识符。

例 386：

```
1   #define NULL ( a > b )     // 错误
2   #define _NULL ( a > b )    // 错误
```

例 386 中第 1 行语句和第 2 行语句定义新宏时，均重复使用标准库中已有的宏 NULL。

修改后的程序如例 387 所示。

例 387：

```
1   #define MY_NULL ( a > b )
2   #define MY_NULL1 ( a > b )。
```

例 388：

```
1   int printf ( int a, int b )        // 错误
```

```
2    {
3       return ( ( a > b ) ? a : b );
4    }
```

例 388 中，printf()是 C 标准库中的函数，不允许在此作为函数名使用。

修改方法：将函数名命名为其他的名字，例如 my_printf()，以区别于标准库中的函数名称。

308. 头文件包含的内容错误

头文件应该用来声明对象、函数、内联函数、函数模板、类型定义(typedef)、宏、类和类模板，不得包含或者产生需要占据存储空间的对象或者函数的定义。

例 389:

```
1    /* file.h */
2    void f1(){}              // 错误
3    int var;                // 错误
4    class C {
5      void f2();
6    };
7    void C::f2() {}         // 错误
8
9    /* file.cpp */
10   #include "file.h"
```

修改后的程序如例 390 所示。

例 390:

```
1    /* file.h */
2    void f1();
3    extern int var;
4    class C {
5      void f2();
6    };
7    template <typename T>
8    void f3 ( T ) { }
9
10   /* file.cpp */
11   #include "file.h"
12   void f1(){}
13   int var;
14   void C::f2() {}
```

309. 头文件中包含多个类

一个头文件只能声明一个类。

例 391:

```
1   #ifndef STATIC_258_H
2   #define STATIC_258_H
3   typedef unsigned int Uint_32;
4   typedef unsigned short Uint_16;
5
6   class Person
7   {
8    public:
9     Person();
10    explicit Person(const Uint_32 personNum);
11    explicit Person(const Person &person);
12    Person & operator=(const Person &person);
13    ~Person();
14   protected:
15   private:
16    Uint_32 personalNumber;
17  };
18
19  class MalePerson : public Person
20  {
21   public:
22    MalePerson();
23    explicit MalePerson(const Uint_32 personNum);
24    MalePerson(const Uint_32 personNum,
25           const Uint_16 weight);
26    explicit MalePerson(const MalePerson &mperson);
27    MalePerson & operator=(const MalePerson &mperson);
28    ~MalePerson();
29   protected:
30   private:
31    Uint_16 weightAmount;
32  };
33  #endif
```

例 391 中，在同一个头文件中声明了多个类。

修改方法：在不同的头文件中分别声明这些类。

310. 头文件 stdio.h 使用错误

编程时，避免使用头文件 stdio.h 中所声明的函数（如 scanf()/printf()），建议使用 iostream.h 提供的具有相同功能的函数（如<</>>）。

例 392：

```
1   #include <stdio.h>
2   int main( )
3   {
4       printf("%s\n", "Hello World");
5       return 0;
6   }
```

例 392 中，第 4 行语句使用 printf()函数输出数据。为避免使用该函数输出数据，建议在头文件中使用 iostream.h 中定义的输出函数。

修改后的程序如例 393 所示。

例 393：

```
1   #include <iostream>
2   using namespace std;
3   int main( )
4   {
5       cout<<"Hello World"<<endl;
6       return 0;
7   }
```

311. 具有外部链接的对象或函数未定义在头文件中

将外部链接的对象或函数的声明放到头文件中，其目的是能够从其他单元来访问。如果外部链接不是必须的，那么该对象或函数应该被声明在一个不同名的命名空间中或者声明为静态的。

例 394：

```
1   // file.cpp
2   int a1 = 0;   // 错误
3   void fun(){}  // 错误
```

修改的程序如例 395 所示。

例 395：

```
1   // file.h
```

```
2  extern int a1;
3  extern void fun();
4
5  // file.cpp
6  #include "file.h"
7  int a1 = 0;
8  void fun(){}
```

312. 在多个文件中声明同一外部对象

外部对象应该在头文件中声明，然后由包含对象声明所在头文件的源文件使用这些对象，这样可以提高程序的可读性。

例 396：

```
1  m.h
2  extern int a;
3  ma.c
4  #include "m.h"
5  extern int a;
```

例 396 中，头文件“m.h ”和源文件“ma.c”中均声明变量 a，外部对象 a 被声明在多个文件内。修改后的程序如例 397 所示。

例 397：

```
1  m.h
2  extern int a;
3  ma.c
4  #include "m.h"
5  int a;
```

7.2　异常

C++语言异常处理机制由 3 部分组成，分别是 try(检查)、throw(抛出)和 catch(捕获)。通常将需要检查的语句放在 try 模块中，用于检查语句是否发生错误；throw 抛出异常，用于发出错误信息；catch 用于捕获异常信息，并对异常加以处理。

313. 从析构函数、释放函数以及 swap 函数抛出异常

析构函数不能抛出异常有两个原因：① 析构函数抛出异常后，异常点之后的程序将不会执行，如果析构函数在异常点之后需要执行某些必要的操作，如释放某些资源，则这些操作将不会执行，

从而引发资源泄漏问题；② 当异常发生时，C++语言的处理机制会调用构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又产生新的异常，将会造成程序崩溃。释放函数以及 swap 函数内抛出异常，会面临同样的问题。

例 398：

```
1    class Exceptions{};
2    class A{
3        A();
4        ~A();
5        void operator delete(void*);
6    };
7    A::~A(){
8       throw Exceptions();           // 错误
9    }
10   void A::operator delete(void*){
11   throw Exceptions();           // 错误
12   }
13   …
```

例 398 中，第 8 行语句在类 A 的析构函数内抛出异常，第 11 行语句在类 A 中重载释放函数 delete()内抛出异常，引起资源泄漏。

修改方法：删除第 8 行和第 11 行语句。

314. 抛出指针类型异常对象

如果抛出的异常对象是指针类型，而指针类型指向的是动态创建的对象，那么将无法确定这个对象是由哪个函数负责销毁以及何时销毁。例如，如果该对象是建立在堆内存上，则必须销毁，否则会造成内存泄漏；如果该对象是建立在栈内存上，则不能销毁，否则程序的行为将不可预测。因此，不能抛出指针类型的异常对象。

例 399：

```
1    class Exception {
2    public:
3        Exception( char* );
4    };
5    bool a();
6    void foo_Violation() {
7        Exception *exp = new Exception("error message");
8        if (!a()) {
9          throw exp;
10   }
11   …
```

例 399 中，第 9 行语句抛出的异常对象是指针类型。

修改方法：将第 7 行语句修改为：Exception exp("error message");

315．程序中存在未处理的异常

程序中应该至少有一个异常处理程序来捕获所有其他未处理的异常。如果程序抛出一个未被处理的异常，程序将会终止，而终止前调用栈是否被“展开”、动态对象能不能被析构，所有这些都依赖于编译器。因此，不仅要处理预期抛出的异常，其他可能被抛出的异常也要有相应的处理措施。

例 400：

```
1   class Exception{};
2   int main( )
3   {
4       try
5       {
6           …
7       }
8      catch ( Exception e )
9       {
10          …
11      }
12      return 0;
13  }
```

例 400 中第 8 行语句，catch 块只能捕获其上面 try 块中的异常，如果在对象 Exception 的构造函数或析构函数中抛出异常，并不能被此 catch 块捕获，将会导致程序终止。

修改方法：在返回语句 return 前增加 catch 块，用于捕获第 8 行语句 catch 块中对象 Exception 可能抛出的异常。

316．catch 语句块外存在空的 throw 语句

空的 throw 语句用来将捕获的异常再次抛出，可以实现多个处理程序间异常的传递。然而，如果在 catch 语句外使用空的 throw 语句，该异常将不能被捕获，从而也就不能再次抛出，程序将以不确定的方式终止。因此，空的 throw 语句只能出现在 catch 语句块中。

例 401：

```
1   class Exception{};
2   void foo(int a)
3   {
```

```
4      Exception E;
5      if(a)
6      {
7        throw;  //错误
8      }
9    }
```

例 401 中第 7 行语句，空的 throw 语句出现在 catch 语句块外会导致程序以不确定的方式终止。

修改方法：将第 7 行语句修改为：throw E;

317. 在程序启动前和终止后抛出异常

在执行 main()函数体之前，是初始化阶段，用于构造和初始化静态对象；在 main()函数返回后，是终止阶段，用于销毁静态对象。在这两个阶段执行期间如果抛出异常，程序将以不确定的方式终止。因此，只能在程序启动之后和终止之前抛出异常，即异常只能在初始化之后、程序结束之前抛出。

例 402：

```
1    class C
2    {
3    public:
4       C ( )
5       {
6         throw ( 0 );  //错误
7       }
8       ~C ( )
9       {
10        throw ( 0 );  //错误
11      }
12   };
13   C c;
14   int main( … )
15   {
16     …
17   }
```

例 402 中，第 6 行语句在 main()函数开始之前抛出异常，第 10 行语句在 main()函数退出之后抛出异常。

318. goto 或 switch 开关语句将控制传递到 try 或 catch 模块中

使用 goto 或 switch 开关语句将控制传递到 try 或 catch 块中，这样的程序是不规范的。

例 403：

```
1   class Exception{};
2   void f ( int i )
3   {
4       if ( 10 == i )
5       {
6           goto Label_10;
7       }
8       switch ( i )
9       {
10          case 1:
11              try
12              {
13          Label_10:
14              case 2:
15              break;
16              }
17              catch ( Exception e )
18              {
19              }
20      }
21  }
```

例 403 中，switch 开关语句 case 1 条件中含有 try 和 catch 模块，第 6 行语句将控制传递到 try 模块内。

319. throw 抛出异常时诱发新的异常

当构造异常对象或计算赋值表达式时诱发新的异常，那么新的异常会在本来要抛出的异常之前被抛出，导致异常抛出逻辑混乱。

例 404：

```
1   class E
2   {
3   public:
4       E ( )
5       {
6           throw 10;
7       }
8   };
```

```
9   int foo1()
10  {
11      try
12      {
13          if ( 0 )
14          {
15              throw E ( );
16          }
17      }
18      catch(…)
19      {
20
21      }
22  }
```

例 404 中第 15 行语句，throw 抛出异常对象 E 时，在异常对象 E 的构造过程中，在第 6 行语句处又抛出异常，异常抛出逻辑混乱。

修改方法：删除第 6 行语句。

320. NULL 被显式抛出

因为 throw(NULL)=throw(0)，因此 NULL 会被当作整型捕获，而不是空指针常量。

例 405：

```
1   #define NULL 0
2   void foo()
3   {
4       try
5       {
6       throw ( NULL ); // 错误
7       }
8       catch ( int i )
9       {
10          …
11      }
12  }
```

例 405 中第 6 行语句，将 throw(0)误用为 throw(NULL)。

321. 函数声明的异常和抛出的异常类型不一致

函数抛出异常的程序结构为：

返回值类型 函数名(形参表) throw(类型名表){函数体}。

函数声明的异常类型和其抛出的异常之间存在如下关系。

① 函数在声明时指定了具体的异常类型，那么它只能抛出指定类型的异常；

② 函数在声明时未指定异常的类型，那么它可以抛出任意类型的异常对象；

③ 函数在声明时异常类型为空，则表示不抛出任何类型的异常。

其中，第二种情况是没有 throw(类型名表)语句，第三种情况是有 throw(类型名表)语句，只是类型名表为空。

对应于上述三种情况，其形式如下：

形式一：int A( ) throw(myexception, int)，表明只能抛出 myexception 和 int 两种类型的异常。

形式二：int A( )，表明可以抛出任何异常，也可以不抛出异常。

形式三：int A( ) throw( )，表明不抛出任何异常。

此外，函数原型中的异常声明要与实现中的异常声明一致，否则会引起异常冲突。

由于异常机制只有在运行出现异常时才发挥作用，因此如果函数的实现中抛出了没有在其异常声明列表中列出的异常，编译器也许无法发现此类错误。在这种情况下，编译器将自动调用 unexpected()函数，unexpected()函数会抛出 std::bad_exception 类型的异常；如果 std::bad_exception 类型的异常也不在异常声明列表内，那么编译器将自动调用 terminate()终止程序运行。

例 406：

```
1   int SetGetValue(int a, int b, int c) throw(bool);
2   int SetGetValue(int a, int b, int c)
3   {
4   try{
5      if(a<1) throw a;
6   }
7   catch(int err)
8   {
9   printf("a less 1 err: %d", err);
10  }
11  return a + b + c;
12  }
```

例 406 中第 1 行语句，函数声明的异常类型是 bool 类型，而第 5 行语句 throw 抛出的是 int 类型，函数声明的异常类型和抛出的异常类型不一致。

修改方法：将第 1 行语句中函数声明的异常类型修改为 int 类型。

322. 未捕获显式抛出的异常

程序中显式抛出的每个异常应该在可能导致异常的所有调用路径上存在捕获异常的处理程序，从而使程序中所有预期抛出的异常均能够被捕获。

例 407：

```
1    class A {};
2    class B {};
3    void main ( int i ) throw ( )
4    {
5        try
6        {
7            if ( i > 10 )
8            {
9            throw A ( );
10           }
11           else
12           {
13       throw B ( );
14       }
15       }
16       catch ( A const & )
17       {
18       }
19   }
```

例 407 中的第 13 行语句抛出异常，但程序中没有与其对应的 catch 捕获语句。

修改方法：在程序中增加捕获“throw B ( );”异常的 catch 捕获语句。

323. 未按照引用方式捕获类(class)类型异常

如果使用非引用方式捕获类类型的对象，那么会切断基类和派生类之间的派生关系，违反 C++语言自身的规范，即基类和派生类之间存在派生关系。也就是说，如果使用非引用方式捕获类类型的对象，当派生类的异常对象被作为基类异常对象捕获时，那么基类和派生类间的派生类行为就被切断了，这样派生对象实际上成为一个基类对象，它不再具有派生类的数据成员和函数，而且当调用它的虚函数时，程序解析后调用的是基类对象的函数。因此，如果异常捕获的对象是类类型，只能以引用类类型的方式捕获该对象。

例 408：

```
1    class ExpBase
2    {
3    };
4    class ExpD1: public ExpBase
5    {
```

```
6   };
7   void foo()
8   {
9      try
10     {
11        throw ExpD1 ( );
12        throw ExpBase ( );
13     }
14       catch ( ExpBase b )  //错误
15     {
16     }
17  }
```

例 408 中，第 11 行和第 12 行分别抛出基类 ExpBase 和派生类 ExpD1 两种异常情况，但只有一条异常捕获语句（第 14 行语句），因此，派生类 ExpD1 的异常对象被作为基类异常对象捕获，这将切断基类和派生类间的派生行为。

修改方法：将第 14 行语句修改为：catch (ExpBase &b)。

324．function-try-block 结构中 catch 处理程序错误

Function-try-block 结构是指函数体整个包含在一个函数 try 块中。

如果在构造函数创建对象的过程中抛出一个内存分配异常，那么当处理程序试图去访问其成员时，该对象还未被创建；相反，如果在析构函数释放对象的过程中抛出异常，该对象可能在异常被处理之前已经被销毁。当处理程序访问未被创建或已销毁对象的成员时，程序将退出。因此，不允许在类的构造函数或析构函数中的 try 功能模块访问类或基类的非静态成员。

例 409：

```
1   typedef int int32_t;
2   class C
3   {
4      public:
5         int32_t x;
6         C( );
7      try
8      {
9         …
10     }
11     catch (… )
12     {
13        if ( 0 == x )
```

```
14          {
15              …
16          }
17      }
18      ~C ( )
19      try
20      {
21          …
22      }
23      catch ( … )
24      {
25          if ( 0 == x )
26      {
27          …
28  }
29   }
30  };
```

例 409 中，在类 C 的构造函数 C( )和析构函数~C ( )中的 catch 模块内存在判断语句“if (0 == x)”，此时，如果类 C 中的变量 x 还未创建，那么该处理程序中的 catch 模块将导致程序退出。

325. function-try-block 结构中处理程序执行顺序错误

当类（派生类和基类）程序中的 function-try-block 块有多个处理程序时，处理程序的执行顺序应遵循先执行最底层派生类的处理程序，待所有派生类的处理程序均执行后，再执行其基类的处理程序的原则。因为派生类的异常会匹配其基类的处理程序，如果基类的处理程序在派生类的处理程序之前被找到，就会使用基类的处理程序，那么派生类的处理程序将成为不可达代码。

例 410：

```
1   class B { };
2   class D: public B { };
3   void foo()
4   {
5       try
6       {
7           …
8       }
9           catch ( B &b )
10      {
11          …
```

```
12        }
13        catch ( D &d )
14        {
15            …
16        }
17    }
```

例 410 中，try-catch 语句块有多个处理程序（第 9 行和第 13 行语句），但处理顺序不是遵循从最底层的派生类到基类的顺序。

修改方法：将第 9 行和第 13 行语句分别修改为：

catch (D &d)和 catch (B &b)。

326. 断言的布尔表达式值为假

在编程过程中，程序员常常会预先做一些假定，然后使用断言来对这些假定进行检查。断言被是异常处理的一种高级形式，用于创建更稳定、更高质量且易于排查错误的程序。在 C 语言中使用 assert()函数来实现断言，函数原型为：

```
void assert(bool expression);
```

assert()计算表达式 expression 的值，如果值为假，那么它先向 stderr 打印一条出错信息，然后通过调用 abort()函数来终止程序运行。

断言有两种形式，第一种形式是 assert(expression)，其中 expression 的结果总是一个布尔值；第二种形式是 assert(expression1：expression2)，其中如果 expression1 为假，则抛出表示断言失败消息的字符串 expression2。

断言常被用来调试程序，但如果程序在发布后未关闭断言且存在断言表达式为假的情况，则被视为不可控异常。

例 411：

```
1   #define CONSTANT 100
2
3    void asrt_test()
4    {
5      int n,m;
6      …
7      scanf("%d", &n);
8      if(n<100)
9       n=100;
10     assert(n>=100);
11     m=sqrt(n);
```

```
12    assert(m>10); // 错误
13    assert(m>10);
14    …
15  }
```

例 411 中，由于第 8、9 行语句的存在，使第一个断言的表达式恒为真。但是第 12 行语句中的断言表达式当 m=10 时断言为假，从而使程序终止运行；如果程序运行到了第 13 行语句，断言是不会有问题的，因为该断言的表达式已经通过了第 12 行语句的判断。

例 412：

```
1   #include <assert.h>
2   void foo( ) {
3     int i = 0;
4     assert(i=5);
5     …
6   }
```

例 412 第 4 行断言语句中，表达式“i=5”的结果不是布尔值。

修改方法：将第 4 行语句修改为：assert(i!=0);

7.3 多线程和同步性

327. 多线程死锁

多线程死锁是两个或两个以上的线程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。多线程应用程序中，在没有资源释放的情况下，针对同一资源的锁不允许被申请两次，以免引发多线程死锁。

例 413：

```
1   #include <pthread.h>
2   pmutex_t *mutex;
3   bool preconditionHolds();
4   void exclusivelyCompute()
5   {
6      pthread_mutex_lock(mutex);
7     if (preconditionHolds())
8      {
9         exclusivelyCompute();
10     }
```

```
11      pthread_mutex_unlock(mutex);
12    }
13    void run()
14    {
15       mutex = new pthread_mutex_t;
16      pthread_mutex_init(mutex,(const   pmutexattr_t*)0);
17       exclusivelyCompute();
18    }
```

例 413 中第 7 行语句，当 if 条件判定为真时，递归调用 exclusivelyCompute()函数，同一个锁将被申请两次，出现应用程序死锁现象。

修改方法：在不同的函数里增加加锁和解锁语句，但必须确保执行解锁的函数只能被执行相应加锁的函数来调用。

修改后的程序如例 414 所示。

例 414:

```
1    #include <pthread.h>
2    pmutex_t *mutex;
3    bool preconditionHolds();
4    static void nonexclusivelyCompute()
5    {
6       if (preconditionHolds())
7       {
8          nonexclusivelyCompute();
9       }
10   }
11   void exclusivelyCompute()
12   {
13      pthread_mutex_lock(mutex);
14       nonexclusivelyCompute();
15      pthread_mutex_unlock(mutex);
16   }
17   void run()
18   {
19      mutex = new pthread_mutex_t;
20      pthread_mutex_init(mutex, (const pmutexattr_t*)0);
21      exclusivelyCompute();
22   }
```

328. 加锁的互斥资源未被解锁

对于多线程应用程序，一个互斥资源被加锁但是没有被解锁，将造成应用程序死锁。

例 415：

```
1   #include <pthread.h>
2   pthread_mutex_t *mutex;
3   bool preconditionHolds();
4   void exclusivelyCompute()
5   {
6       pthread_mutex_lock(mutex);
7       if (preconditionHolds()) {
8        pthread_mutex_unlock(mutex);
9       }
10  }
```

例 415 中第 7 行语句，当 if 条件判定为假时，被加锁的互斥资源 mutex 未被解锁，将导致程序死锁。

修改方法：将解锁语句 pthread_mutex_unlock(mutex)放到 if 条件语句外执行。

修改后的程序如例 416 所示。

例 416：

```
1   #include <pthread.h>
2   pthread_mutex_t *mutex;
3   bool preconditionHolds();
4   void exclusivelyCompute()
5   {
6      pthread_mutex_lock(mutex);
7      if (preconditionHolds())
8      {
9         …
10     }
11     pthread_mutex_unlock(mutex);
12  }
```

329. 线程持有加锁资源的同时使用阻塞函数

程序中的某些函数，例如 sleep()函数，使持有加锁资源的线程在某些重要的时刻停止执行一段时间，因此增加了线程间不必要的资源争夺，导致死锁发生。

例 417：

```
1   #include <pthread.h>
```

```
2    #include <unistd.h>
3    pthread_mutex_t *mutex;
4    static void testLock()
5    {
6         pthread_mutex_lock(mutex);
7        sleep(10);
8        …
9        pthread_mutex_unlock(mutex);
10   }
```

例 417 中，第 6 行语句对资源进行了加锁，在第 7 行语句使用等待函数 sleep()，使持有加锁资源的线程停止执行并等待一段时间，在这个等待时间里，如果有其他线程需要已被加锁的资源，那么线程间将进入死锁状态。

7.4　代码不可达

程序中分支语句的判定条件始终为真或假、循环语句的条件始终不满足，或在 return、break、continue、goto 等特殊语句后执行的语句，都可能导致程序中存在不可达代码。

330．return、break、continue 和 goto 语句之后包含代码

例 418：

```
1    void foo();
2    int myFunction(int i, int j)
3    {
4        switch(i)
5        {
6           case 1:
7               j = 5;
8               break;
9               foo();  // 不可达代码
10          case 2:
11              j = 3;
12              return j;
13              foo();        // 不可达代码
14       }
15   }
```

例 418 中，foo()函数始终不能被执行，为不可达代码。

331．if/else 语句中判断条件有误导致代码不可达

例 419：

```
1   void foo()
2   {
3       int unreachable_code = 1;
4       if(0)
5       {
6           unreachable_code = 2; //不可达代码
7       }
8   }
```

例 419 中，由于 if 条件始终不满足，导致第 6 行语句不可达。

修改方法：修改 if 条件语句中的判断条件。

332．switch 语句有误导致代码不可达

例 420：

```
1   void foo( int i ) {
2       switch (i) {
3           i = 0;
4       case 1:
5           i = 1;
6           break;
7       default:
8           i = 2;
9           break;
10      }
11  }
```

修改后的程序如例 421 所示。

例 421：

```
1   void foo( int i ) {
2       switch (i) {
3       case 0:
4           i = 0;
5           break;
6       case 1:
7           i = 1;
```

```
8           break;
9       default:
10        i = 2;
11          break;
12      }
13  }
```

333．for/while 循环语句中循环条件有误导致代码不可达

例 422：

```
1   void foo( int i, int j )
2   {
3       for ( i = 0; i < 0; i++ )  // 错误
4       {
5         j = 1;
6       }
7   }
```

例 422 中，for 循环中的循环条件设计有误，导致 for 循环始终不能被执行，成为不可达代码。修改方法：修改 for 循环语句的判断条件，例如 i<5。

334．for/while/catch 模块外部的 if 或 switch 之后包含不可达代码

例 423：

```
1   int foo1( int c ) {
2       if ( c > 2 ) {
3           return 0;
4       } else {
5           return 1;
6       }
7       return c;           // 不可达代码
8   }
9
10  int foo2( int i ) {
11      switch(i){
12      case 1:
13          i++;
14          return 0;
15      case 2:
16          i = i + 2;
```

```
17          return 1;
18      default:
19          return 2;
20      }
21      return i;           // 不可达代码
22  }
```

修改方法：删除不可达代码或将不可达代码移到判断语句中。

335. for /while/catch 模块内部的 if 或 switch 之后包含不可达代码

例 424：

```
1   int foo( int c ) {
2       while ( c > 1 ) {
3           if ( c > 2 ) {
4               continue;
5           } else {
6               break;
7           }
8           c++;                // 不可达代码
9       }
10
11      for (int i = 0; i > 1; i++ ) {
12          switch(i){
13          case 1:
14              i++;
15              return i;
16          case 2:
17              i = i + 2;
18              return i;
19          default:
20              return i;
21          }
22          c++;                // 不可达代码
23      }
24      return c;
25  }
```

修改后的程序如例 425 所示。

例 425：

```
1   int foo( int c ) {
2      while ( c > 1 ) {
3         c++;
4         if ( c > 2 ) {
5            continue;
6         } else {
7            break;
8         }
9      }
10
11     for (int i = 0; i > 1; i++ ) {
12        c++;
13        switch(i){
14        case 1:
15           i++;
16           return i;
17        case 2:
18           i = i + 2;
19           return i;
20        default:
21           return i;
22        }
23     }
24     return c;
25  }
```

336. 判断条件自身错误或矛盾导致不可达代码

例 426：

```
1   void myFunction(int a)
2   {
3      if((a <= 1)&&(a >= 2)); //错误
4      if((a < 1)&&(a > 2));   //错误
5      if((a >= 2)&&(a <= 1)); //错误
6      if((a > 2)&&(a < 1));   //错误
7   }
```

例 426 中，if 语句自身出现矛盾，表达式产生的结果始终是 true 或者 false，导致后续部分代码不可达。

修改方法：修改 if 条件语句中的判断条件，例如 `if((a <= 1)&&(a >= -2));`

附录 A 常用静态分析工具

A.1 PolySpace——运行时错误静态检查工具

PolySpace 软件是由 PolySpace 公司（2007 年被 Mathworks 收购）开发的一个程序运行时错误（run-time errors）静态检查工具。该工具采用基于源代码静态检查的方法来检查程序在运行时可能出现的错误，可以大幅度提高软件的可靠性，降低测试成本，缩短软件的开发周期。

PolySpace 能对 C、C++、Ada 语言进行检查，检查的运行时错误种类包括：

- 非法的指针解引用；
- 数组越界；
- 函数返回未初始化的值；
- 使用未初始化变量；
- “除 0”等数学运算错误；
- 断言语句为假；
- 整数、浮点数上溢出/下溢出；
- 参数个数错误；
- 无限循环；
- 与面向对象编程相关的动态错误（C++）；
- 与异常处理相关的错误（C++）；
- 不可达代码；
- 其他（与安全有关的项目）。

PolySpace 采用高级形式化分析技术，对源代码进行形式化验证，能够保证分析的完备性。与其他动态测试工具相比，它不需要测试用例，不需要使用仪器，更不需要运行程序，就能对源程序进行运行时错误检查，降低了查错的成本。由于该工具采用模拟程序动态运行的方式对软件进行分析，

因此需要耗费大量的时间和资源。根据笔者经验，在一台配置较高的机器上（CPU：2GHz；内存：2GB），使用 PolySpace 工具分析一个 10000 行的程序，需要两个半小时。

下面以某型号嵌入式软件为例，介绍 PolySpace 4.1 的使用方法，主要介绍两个工具：PolySpace Verifier 和 PolySpace Viewer。前者依据源代码创建一个工程并进行分析，后者用来查看分析结果。软件的安装属于典型的“Yes—Next”安装流程，在此不作介绍。

A.1.1　PolySpace Verifier

通过开始菜单或者桌面快捷方式打开“PolySpace Launcher”，如图 A.1 所示。

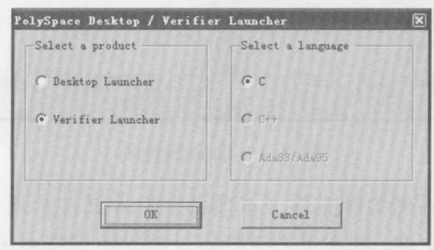


图 A.1　Verifier Launcher 语言选择界面

选择“Verifier Launcher”和相应的开发语言，单击“OK”进入 PolySpace Verifier 主界面，单击菜单“File→New Project”新建一个工程，如图 A.2 所示。

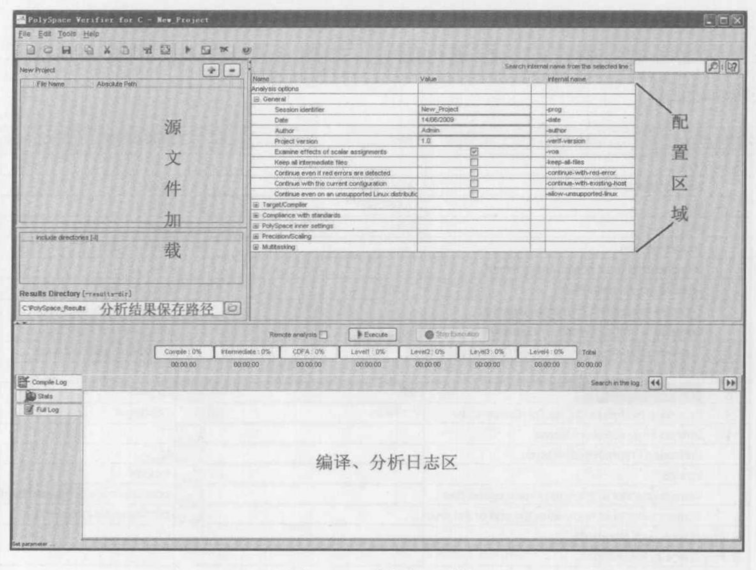


图 A.2　新建工程

在主界面左上侧单击按钮进入源程序文件加载对话框，如图 A.3 所示。

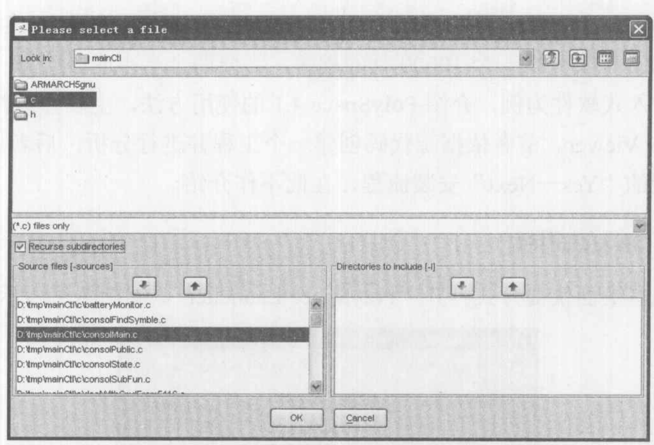


图 A.3　选择源文件

对话框下部有两个区域，左侧为源文件（本项目中为.c 文件）列表，右侧为头文件（.h 文件）列表。可以每次添加一个.c/.h 文件，即在上方文件目录列表中选择要添加的文件，单击相应↓按钮；也可以每次添加一个文件夹中的所有文件，即在上方文件目录列表中选择要添加的文件夹，单击相应↓按钮，该文件夹中的所有文件就显示在下方的列表中。添加文件后可以使用↑按钮来移除相应文件。需要注意的是，在添加头文件时，除了原项目中自定义的头文件外，还必须包含所有相关的系统头文件。

接下来需要在配置区域对新建的工程进行配置，如图 A.4、图 A.5 和图 A.6 所示。不正确的配置将导致工程无法编译通过。

	Name	Value		Internal name
	Analysis options			
	⊟ General			
	Session identifier	New_Project		-prog
	Date	15/06/2009		-date
	Author	Admin		-author
	Project version	1.0		-verif-version
	Examine effects of scalar assignments	☑		-voa
	Keep all intermediate files	☐		-keep-all-files
1	Continue even if red errors are detected	☑		-continue-with-red-error
	Continue with the current configuration	☐		-continue-with-existing-host
	Continue even on an unsupported Linux distributic	☐		-allow-unsupported-linux
	⊟ Target/Compiler			
2	Target processor type	i386	...	-target
3	Operating system target for PolySpace stubs	VxWorks		-OS-target
4	Defined Preprocessor Macros		...	-D
	Undefined Preprocessor Macros		...	-U
	Include		...	-include
	Command/script to apply to preprocessed files		...	-post-preprocessing-command
	Command/script to apply after the end of the analy		...	-post-analysis-command
	⊞ Compliance with standards			
	⊞ PolySpace inner settings			
	⊞ Precision/Scaling			
	⊞ Multitasking			

图 A.4　工程配置 1

图 A.4 主要用于基本设置，其中：

“1”：必须勾选此项，否则分析时遇到第一个标记为红色的错误（red error）就会停止分析，等待用户修改后重新分析。

“2”：选择目标机的处理器类型。该选项决定数据类型的长度。

“3”：选择目标机的操作系统。该选项确定 PolySpace 用到的一些基本函数的原型。

“4”：添加编译阶段的宏定义标记。

Name	Value	Internal name
Examine effects of scalar assignments	☑	-voa
Keep all intermediate files	☐	-keep-all-files
Continue even if red errors are detected	☑	-continue-with-red-error
Continue with the current configuration	☐	-continue-with-existing-host
Continue even on an unsupported Linux distributic	☐	-allow-unsupported-linux
⊞ Target/Compiler		
⊞ Compliance with standards		
⊞ PolySpace inner settings		
⊟ Precision/Scaling		
⊟ Precision		
5 Quick Analysis	☐	-quick
6 Analysis Precision Level	2	-O
Specific Precision	...	-modules-precision
Launch PolySpace Verifier from beginning of	scratch	-from
To end of	Software Safety Analysis level 4	-to
Sensitivity context	...	-context-sensitivity
Automatic selection for sensitivity context	☐	-context-sensitivity-auto
Improve precision of interprocedural analysis		-path-sensitivity-delta
Retype variables of pointer types	☐	-retype-pointer
Retype symbols of integer types	☐	-retype-int-pointer
⊞ Scaling		
⊞ Multitasking		

图 A.5 工程配置 2

图 A.5 主要设置分析精度，共有 0、1、2、3、4 五个级别。其中，

“5”：快速分析模式。选择该项，则接下来四项变为灰色不能编辑。第一次分析时，一般采用“Quick”模式。

“6”：如不采用“Quick”模式，则在此处选择分析精度级别。分析的精度越高，所需时间也越长。一般采用级别 2 即可。

图 A.6 用于多任务设置，其中：

“7”：多任务入口。PolySpace 认为多任务和中断是一个概念。此处填写多任务的函数名称，多任务函数的原型必须是：void proc(void)。

“8”：互斥设置。此处指定不能同时运行的任务列表，任务间以空格隔开，按运行先后顺序排列。

至此，主要的配置就完成了，此外还有重要的一点需要说明，要能够对源代码进行成功的分析，源工程需要有一个函数入口——main()函数，当没有 main()函数入口时，需手工添加 main()函数，并在其中调用被分析软件的主函数。

单击 Execute 按钮对工程进行编译、分析。如果编译、分析过程出错，依据编译、分析日志信息作相应修改后再次运行。分析结束后就可以使用 PolySpace Viewer 来查看分析结果。

Name	Value	Internal name
Analysis options		
⊟ General		
Session identifier	New_Project	-prog
Date	15/06/2009	-date
Author	Admin	-author
Project version	1.0	-verif-version
Examine effects of scalar assignments	☑	-voa
Keep all intermediate files	☑	-keep-all-files
Continue even if red errors are detected	☑	-continue-with-red-error
Continue with the current configuration	☑	-continue-with-existing-host
Continue even on an unsupported Linux distribution	☑	-allow-unsupported-linux
⊞ Target/Compiler		
⊞ Compliance with standards		
⊞ PolySpace inner settings		
⊞ Precision/Scaling		
⊟ Multitasking		
7 Entry point or Interruption	...	-entry-points
Critical section details	...	-critical-section-begin/end
8 Temporal exclusion point (separated by space characters)	...	-temporal-exclusions-file

图 A.6　工程配置 3

A.1.2　PolySpace Viewer

PolySpace Verifier 分析结束后，既可以使用工具栏按钮直接打开 PolySpace Viewer 查看分析结果，也可以打开 PolySpace Viewer 后通过“File→Open”打开分析结果文件进行查看，如图 A.7 所示。

图 A.7　分析结果查看界面

PolySpace Viewer 主界面主要有 5 个显示区域。分别是错误信息统计列表、源代码查看、函数调用关系、错误详细信息和变量查看。

错误信息统计列表（见图 A.8）。按照源文件、函数分别列出分析结果的统计信息，包括问题类别、问题等级、在源文件中的行数和列数、问题详细描述。PolySpace 使用四种不同的颜色标识不同类型的问题：红色表示每次运行都会出现的错误；橙色表示满足一定条件会出现的错误；灰色表示

不可达代码；绿色表示安全代码，即每次运行都不会出问题的代码。

Procedural entities	!	✗	?	✓	Line	Col	%	Details
✓ OVFL.781				1	6641	73		[*] overflow range: {2147483647 >= [exp
✓ UNFL.782				1	6641	73		[*] underflow range: {-2147483648 <= [e
✓ OVFL.783				1	6642	70		[*] overflow range: {2147483647 >= [exp
✓ UNFL.784				1	6642	70		[*] underflow range: {-2147483648 <= [e
✗ OBAI.785		1			6643	54		array index within bounds: [0..399]
! NIVL.786	1				6643	59		
✗ OVFL.787		1			6644	72		[*] overflow range: {2147483647 >= [exp
✗ UNFL.788		1			6644	72		[*] underflow range: {-2147483648 <= [e
✓ OVFL.789				1	6646	73		[*] overflow range: {2147483647 >= [exp
✓ UNFL.790				1	6646	73		[*] underflow range: {-2147483648 <= [e
? OBAI.791			1		6647	58		array index within bounds: [0..9]
✓ OVFL.792				1	6647	59		[*] overflow range: {2147483647 >= [exp
? UNFL.793			1		6647	59		[*] underflow range: {-2147483648 <= [e
✓ NIV.794				1	6647	63		
✓ NIV.795				1	6647	69		
✓ OBAI.796				1	6647	69		array index within bounds: [0..254]
? OBAI.797			1		6649	63		array index within bounds: [0..9]
✓ OVFL.798				1	6649	64		[*] overflow range: {2147483647 >= [exp
? UNFL.799			1		6649	64		[*] underflow range: {-2147483648 <= [e
✓ NIV.800				1	6649	68		
✓ NIV.801				1	6649	74		
✓ OBAI.802				1	6649	74		array index within bounds: [0..254]
✓ OVFL.803				1	6651	75		[*] overflow range: {2147483647 >= [exp
✓ UNFL.804				1	6651	75		[*] underflow range: {-2147483648 <= [e
✗ OBAI.805		1			6652	54		array index within bounds: [0..399]
! NIVL.806	1				6652	59		
✗ OVFL.807		1			6653	72		[*] overflow range: {2147483647 >= [exp
✗ UNFL.808		1			6653	72		[*] underflow range: {-2147483648 <= [e

图 A.8　错误信息统计列表

单击错误信息统计列表中某一条错误信息，就会链接源代码并显示在源代码查看区域，如图 A.9 所示，同时在错误详细信息区域显示该错误的详细信息，在函数调用关系区域显示相关的函数调用关系。可以通过这些信息来分析、追踪错误。

图 A.9 中，使用红色和橙色标记了在文件 comm.c 第 5543 行至 5560 行中发现的 3 类共 7 个错误。

① 第 5543 行和第 5552 行中，分别存在一个确定的错误（红色）。

原因分析：使用未初始化变量 temp。

修改方法：在使用变量 temp 前对其进行初始化。

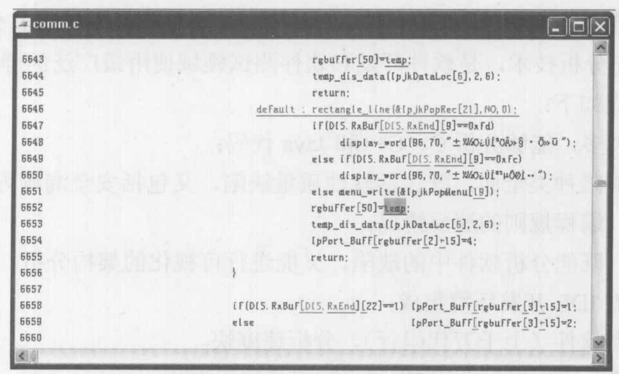


图 A.9　链接错误代码行

② 第 5547、5549 和第 5558 行，存在三个可能的错误（橙色）。

原因分析：使用未初始化变量 DIS。出现这个错误是因为前文对该结构体进行初始化的代码可能不会被执行。

修改方法：检查程序逻辑，确保该结构体在使用前被正确初始化。

③ 第 5558 行和第 5559 行，存在两个可能的错误（橙色）。

原因分析：数组越界引起的缓冲区溢出。

修改方法：对数组 pPort_BuFF[]的下标“rgbuffer[3]+15”进行检查，确保该值不会造成数组越界。

此时也可以通过单击工具栏按钮来图形化显示函数调用关系，如图 A.10 所示。

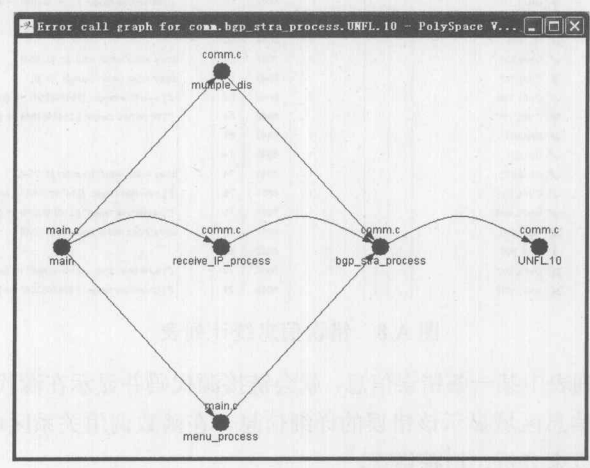


图 A.10　函数调用关系

A.2　Klocwork——代码静态检查工具

Klocwork 软件是由 Klocwork 公司基于其专利技术—分析引擎开发的静态分析工具。该工具综合应用了多种先进的静态分析技术，是软件开发和软件测试领域使用最广泛的静态分析工具之一。

Klocwork 主要优点如下：

- 支持的语言种类多，能够分析 C、C++和 Java 代码；
- 能发现的软件缺陷种类全面，既包括软件质量缺陷，又包括安全漏洞方面的缺陷，还可以分析对软件架构、编程规则的违反情况；
- 软件功能全面，既能分析软件中的缺陷，又能进行可视化的架构分析、优化；
- 能够与多种主流 IDE 开发环境集成；
- 能够分析超大型软件（上千万代码行），分析速度快。

下面以某数据管理系统为例，介绍 Klocwork 工具使用方法。

A.2.1 工程创建与分析

首先，双击桌面 Start klocwork Servers 图标启动 Klocwork 的四个服务，其中 lm 是 License 服务，mysql 是数据库服务，tomcat 是 Web 服务，Klocwork 分析调度引擎 rmimanager。Klocwork 开启服务界面如图 A.11 所示。

```
Start Klocwork Servers
Using projects root: D:\K72
Local Host is: LHY [192.168.0.23]
Starting lm              [started on LHY:27000] (projects root is D:\K72)
Starting mysql           [started on LHY:3309] (projects root is D:\K72)
Starting tomcat          [started on LHY:8064]
Starting rmimanager
```

图 A.11 Klocwork 服务启动界面

Klocwork 服务成功开启后，双击 Klocwork Management Console 图标， 进入 Klocwork 控制台界面，如图 A.12 所示。

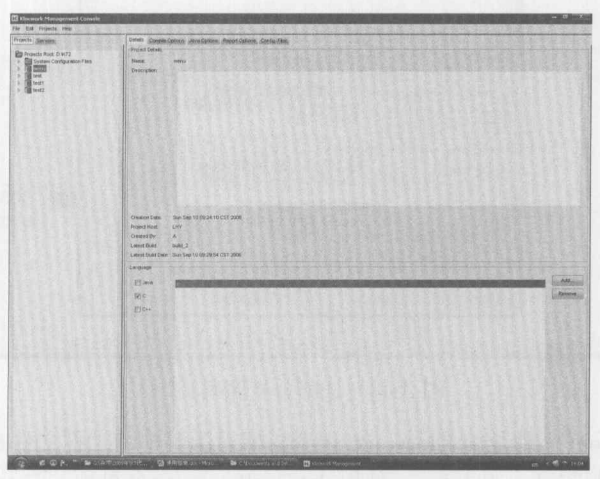


图 A.12 Klocwork 控制台界面

在 Klocwork 控制台界面中新建一个源代码分析工程，并选择语言类型：C、C++或 Java，如图 A.13 所示，在 Klocwork 控制台界面新建一个源码分析工程“test”。

单击图 A.13 中的“Next”按钮，进入选择分析源码类型界面，如图 A.14 所示。

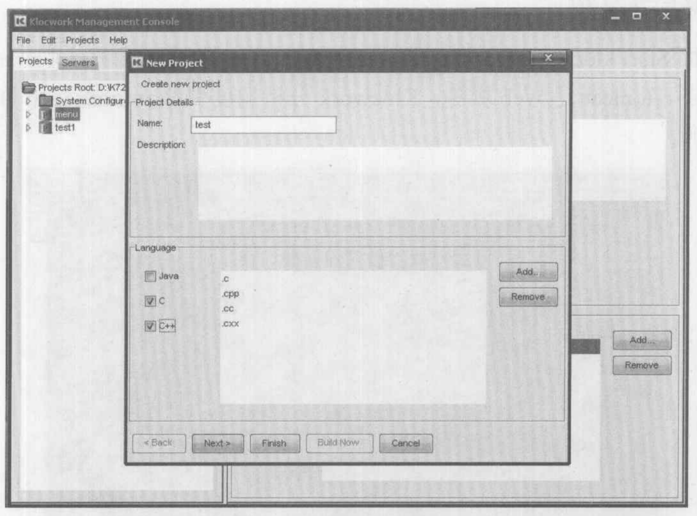


图 A.13 建立源码分析工程

图 A.14 选择分析源码类型

源码分析选择的*.out 文件是由 Klocwork 中分析器对源码中的工程文件（*.dsp、*.dsw、*.vcproj、*.sln）分析得到的；对于*.dsp 和*.dsw，使用 kwDspParser 命令得到；对于*.vcproj 和*.sln，使用 kwVcprojParser.exe 命令得到。

可以选择图 A.14 中任一种源码文件类型进行缺陷分析。下面以第一种源码文件为例进行源码缺陷分析。

单击图 A.14 中的“Next”按钮，进入添加源码需要的包含文件和宏定义界面，如图 A.15 所示。

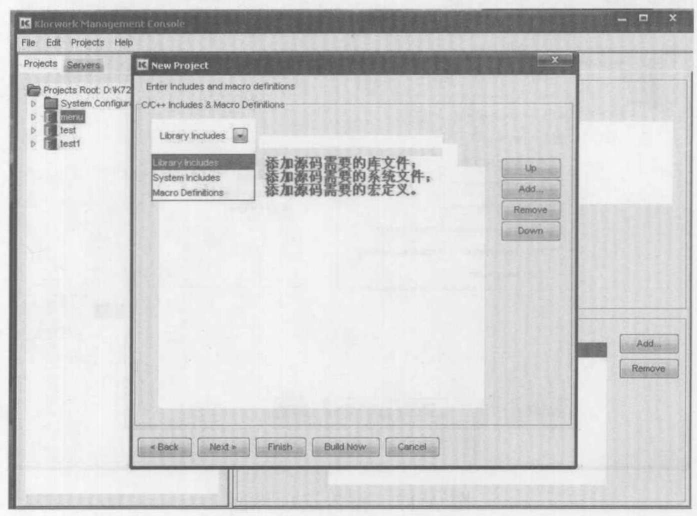


图 A.15 包含文件和宏定义添加界面

单击图 A.15 中的“Next”按钮，进入编译选项界面，如图 A.16 所示。

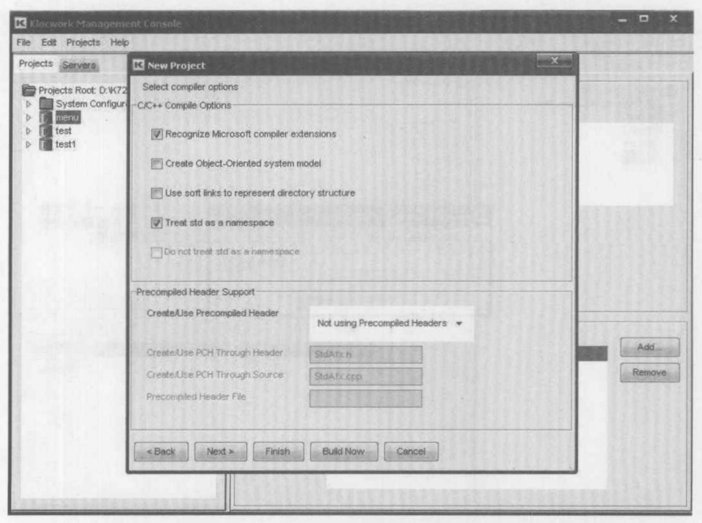


图 A.16 编译选项界面

单击图 A.16 中的“Next”按钮，进入报告选项界面，如图 A.17 所示。

在图 A.17 中完成报告选项后（一般选择第 1、6、7、8 项），单击“Build Now”按钮，进入图 A.18；或单击“Finish”按钮，回到新建工程界面，在界面左边工程列表中选择新建的工程名，单击鼠标右键，在弹出的右键菜单中选择“Build”选项，进入图 A.18。

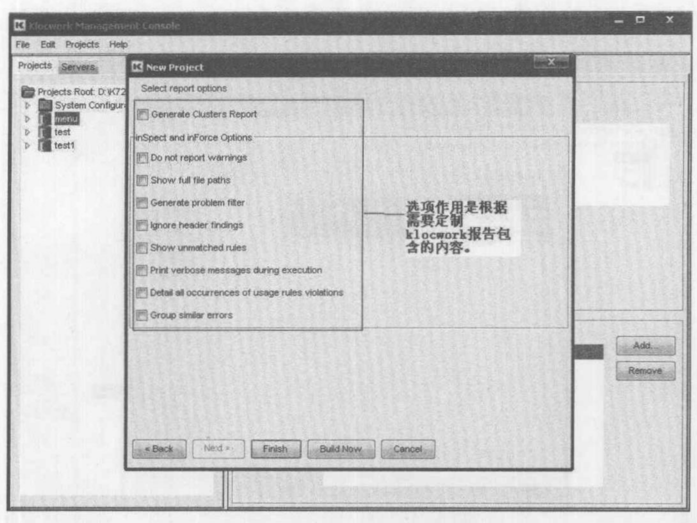


图 A.17　报告选项界面

在图 A.18 中，为源码分析结果选择空文件夹，单击“Build”按钮后，进入源码分析阶段，如图 A.19 所示。

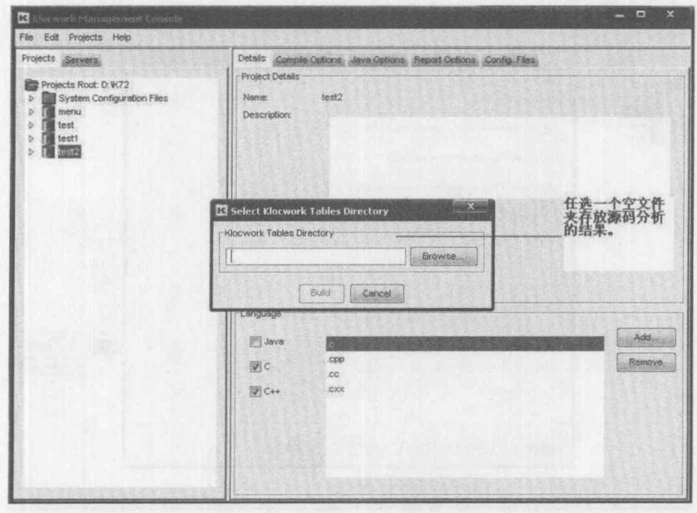


图 A.18　选择空文件夹存放源码分析结果界面

A.2.2　分析结果查看

工程分析结束后，通过 Web 方式查看分析结果。打开 IE 浏览器，在地址栏中输入 Klocwork 源码分析工具所在的 IP 地址和端口号(8064)，如果服务器在本地，IE 地址栏中输入 http://localhost:8064

即可进入图 A.20 所示的用户名、密码输入界面。

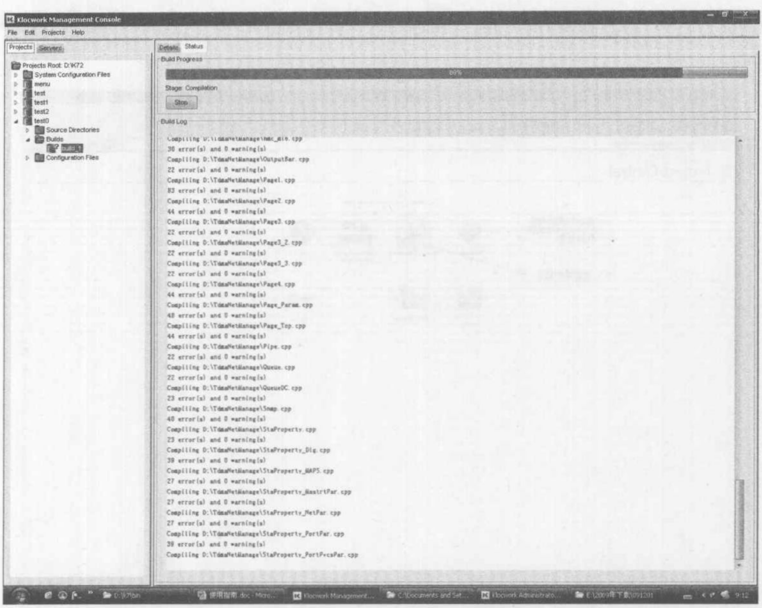


图 A.19　源码分析阶段

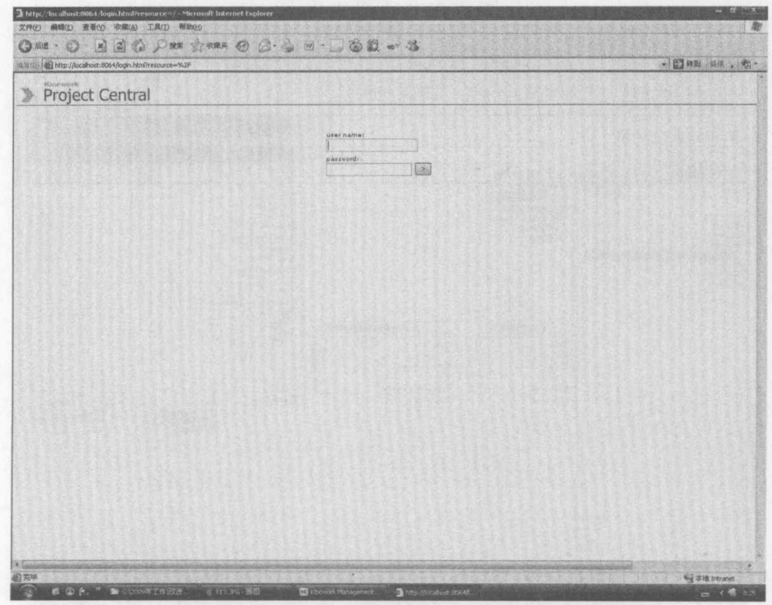


图 A.20　用户名密码输入界面

在图 A.20 所示界面中，输入用户名和密码，单击 图标，进入图 A.21，在图 A.21 中可以根据需要查看源码分析结果，单击 图标进入图 A.22 设置分析报告内容；单击 图标进入图 A.23 查看源码分析报告。

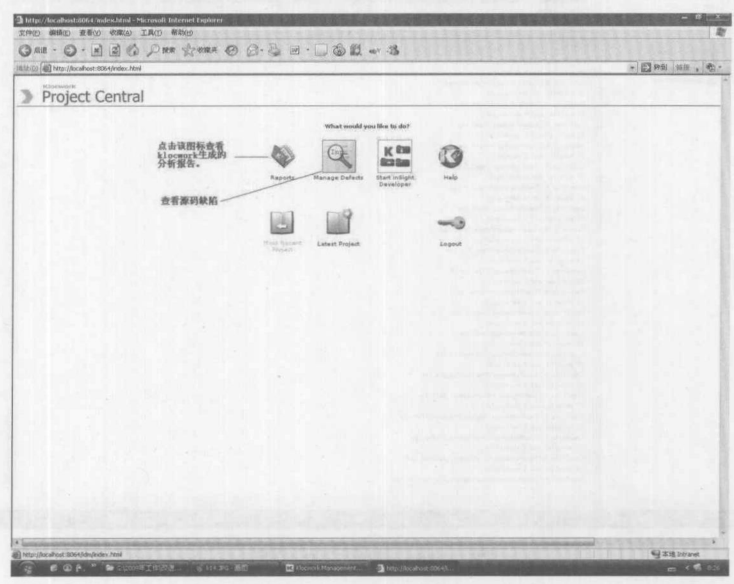


图 A.21　查看主界面

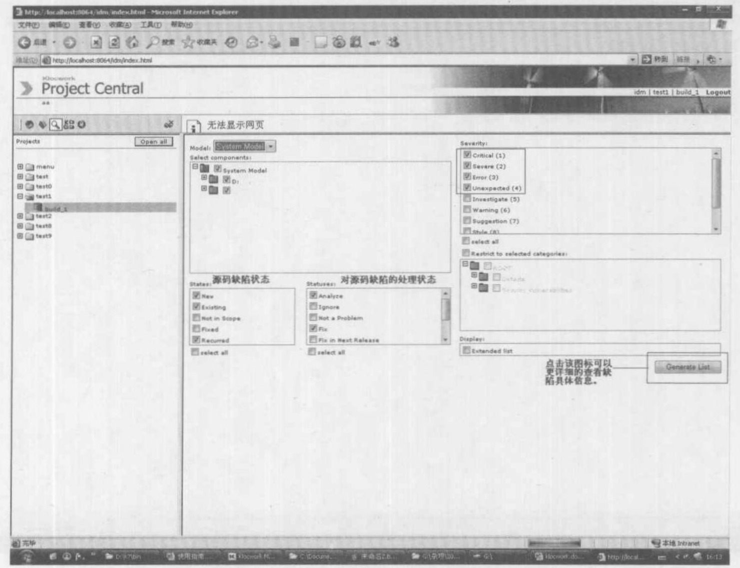


图 A.22　分析报告显示内容配置

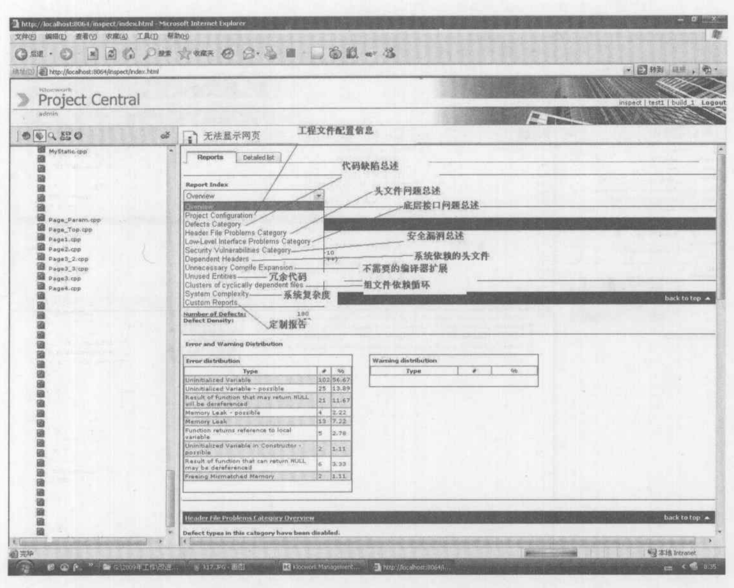


图 A.23 分析报告查看

单击图 A.22 中的“Generate List”按钮，进入代码缺陷总述界面，如图 A.24 所示。在图 A.24 界面中，单击界面上的🔍图标，可以详细查看该行缺陷的追踪信息，如图 A.25 所示，单击界面上的“Code”栏的缺陷类型可以查看 Klocwork 工具对该缺陷类型的详细定义，如图 A.26 所示。

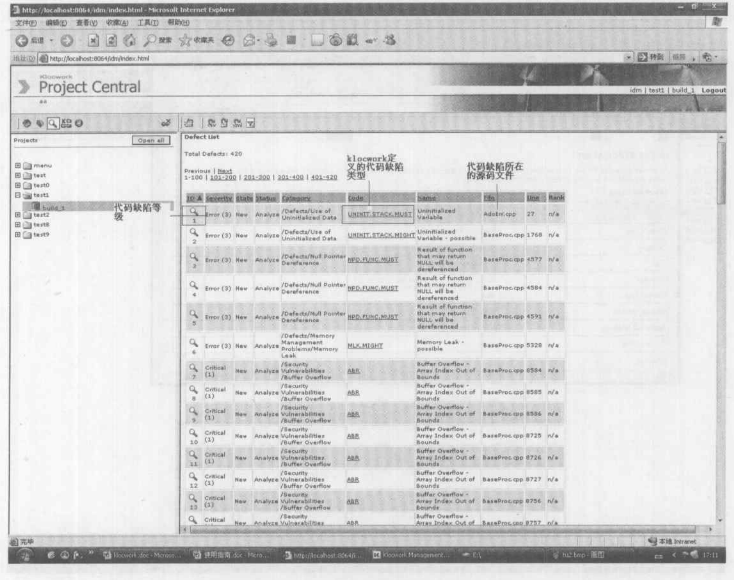


图 A.24 代码缺陷总述界面

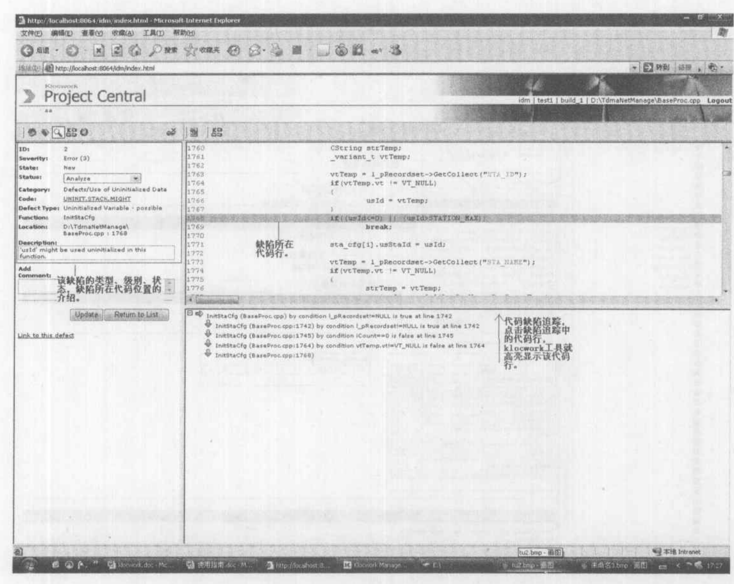


图 A.25　代码缺陷详细追踪界面

新建的工程分析成功后，可以通过“InSight Architect”查看被分析源码的框架结构。

单击桌面上的图标，进入图 A.27，在图 A.27 中选择需要查看的工程，单击“Ok”按钮可以进一步查看源码的框架结构，如图 A.28 所示。

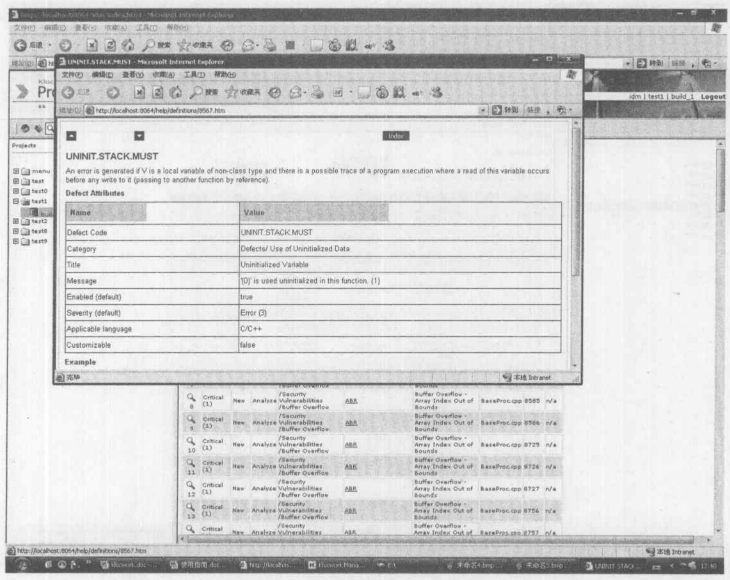


图 A.26　Klocwork 工具缺陷类型定义

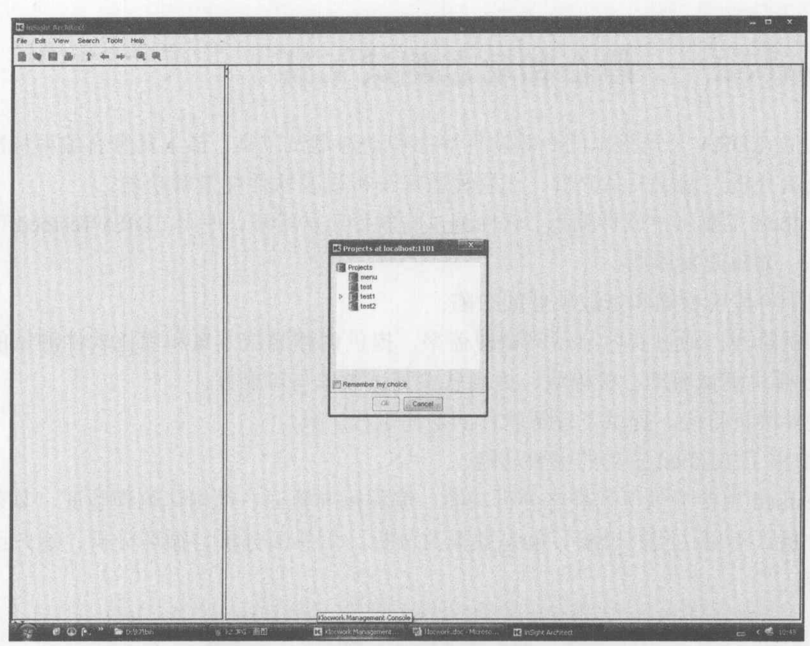


图 A.27　查看源码框架结构

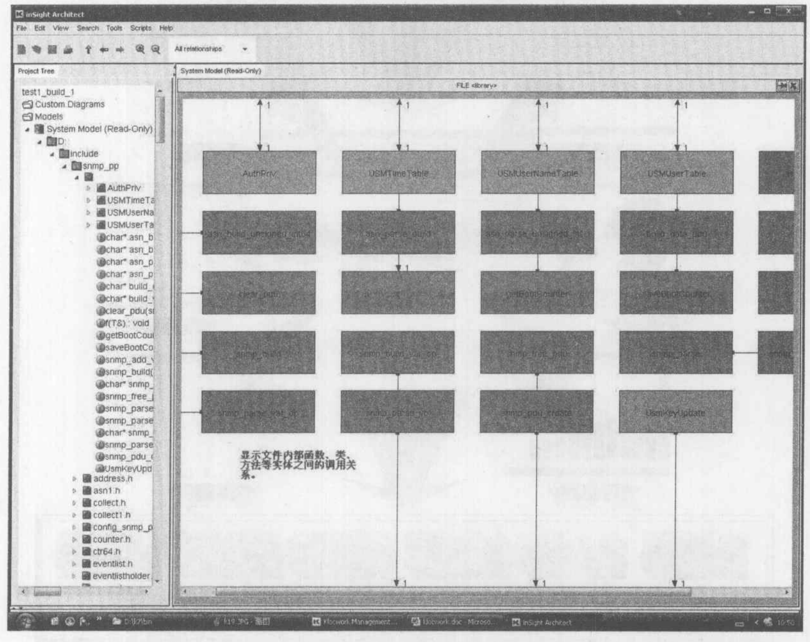


图 A.28　源码框架结构

A.3 Testbed——静态和动态测试工具

Testbed 是由 LDRA 公司推出的一款软件静态和动态测试工具，该工具具有编程标准验证、数据流分析、控制流分析、调用关系分析、代码覆盖率分析以及软件度量等功能。

LDRA Testbed 主要用于软件编程、软件测试与软件维护阶段，使用 LDRA Testbed 可以提高软件产品开发效率、缩短开发周期。

（1）在编程阶段可检测和修改软件的缺陷；

（2）在测试阶段，通过实时显示测试覆盖率，提供调整测试方案和优化软件测试的必要信息；自动生成单元/模块测试驱动、桩模块，提高软件测试效率与可靠性；

（3）在软件维护阶段，提供了理解软件的逆向工程工具。

图 A.29 给出了 Testbed 工具的主要功能。

LDRA Testbed 主要提供下列静态分析功能：编程标准验证、结构化编程验证、复杂性度量、变量交叉引用分析、不可达代码分析、静态数据流分析、信息流分析、循环分析、递归函数分析、函数接口分析等。

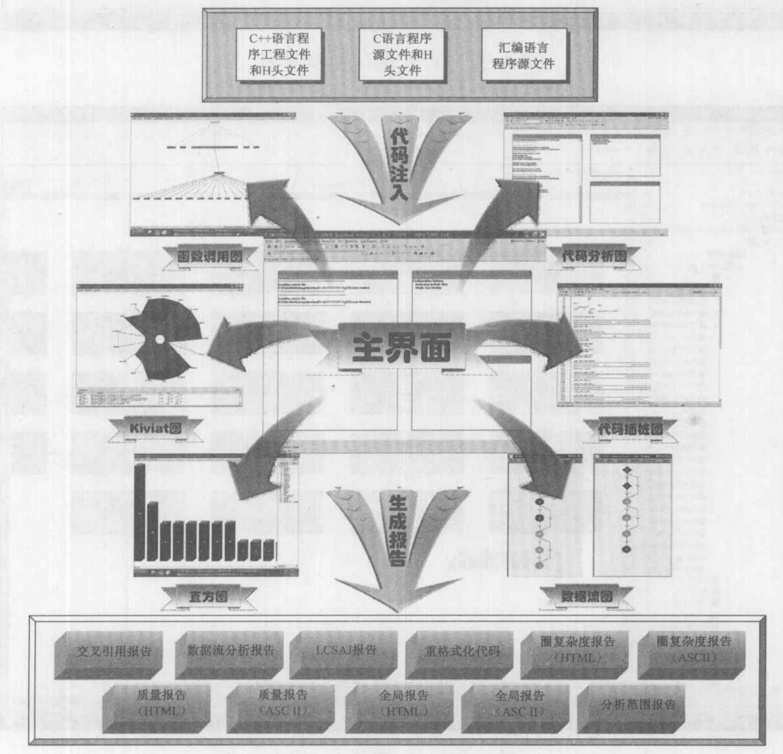


图 A.29　Testbed 工具主要功能

A.3.1 单个文件分析

Testbed 既能分析单个文件也能通过集（set）的方式同时分析多个文件。在分析之前，要保证源代码没有语法错误，即能通过编译，同时，还要求源文件所在的路径中没有中文和空格。

下面以一个简单的 C 语言程序为例，介绍 Testbed 的主要功能和使用方法。例子程序完成一个简单的功能：输入一个数字，按顺序插入一个已排序的数组中并打印出来。代码如下：

```
1    #include <stdio.h>
2    void main()
3    {
4         int a[11]={1,4,6,9,13,16,19,28,40,100};
5         int temp1,temp2,number,end,i,j,k;
6         printf("original array is:\n");
7         for(i=0;i<10;i++)
8              printf("%5d",a[i]);
9         printf("\n");
10        printf("insert a new number:");
11        scanf("%d",&number);
12        end=a[9];
13        if(number>end)
14             a[10]=number;
15        else
16             {
17             for(i=0;i<10;i++)
18                  {
19                       if(a[i]>number)
20                       {
21                            temp1=a[i];
22                            a[i]=number;
23                            for(j=i+1;j<11;j++)
24                            {
25                                 temp2=a[j];
26                                 a[j]=temp1;
27                                 temp1=temp2;
28                            }
29                            break;
30                       }
31                  }
32        }
```

```
33        for(i=0;i<11;i++)
34            printf("%6d",a[i]);
35        return;
36   }
```

启动软件后进入工具主界面，如图 A.30 所示。

其中：

1——标题栏，显示版本和版权信息；

2——菜单栏，用户通过点击菜单项可完成文件选取、分析，以及查看结果等各项功能；

3——工具栏，常用功能的快捷按钮；

4——输入源文件，显示当前选中的源文件或集的名字，如果当前分析的是一个集，则显示该集中正被分析的文件名；

5——分析信息窗口，显示当前正在进行的分析操作；

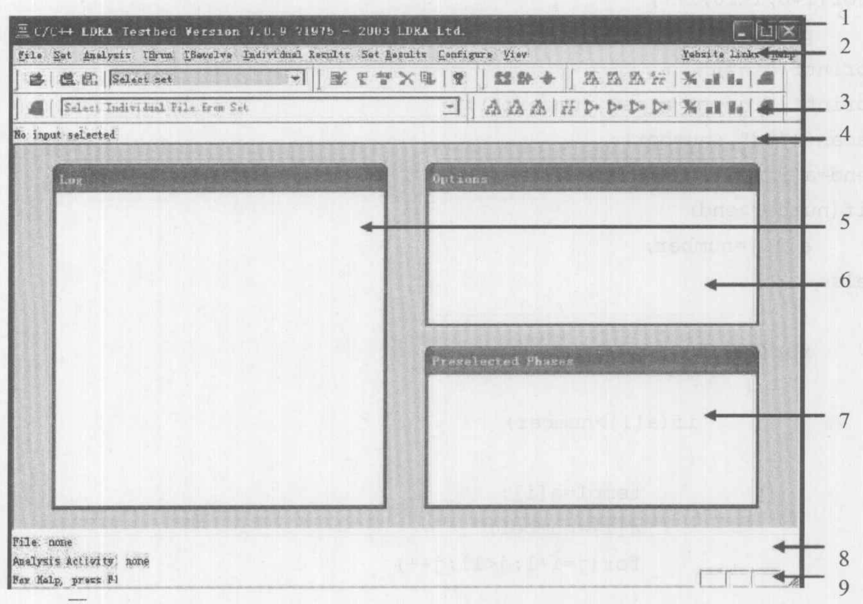


图 A.30 Testbed 主界面

6——选项窗口，显示当前可选的分析项；

7——执行状态窗口，显示已执行的分析项；

8——分析状态栏，显示当前正在执行的分析项；

9——状态栏，显示当前选中的菜单项的功能和快捷键信息。

选择菜单“File→Select File”，弹出文件选择对话框，如图 A.31 所示。

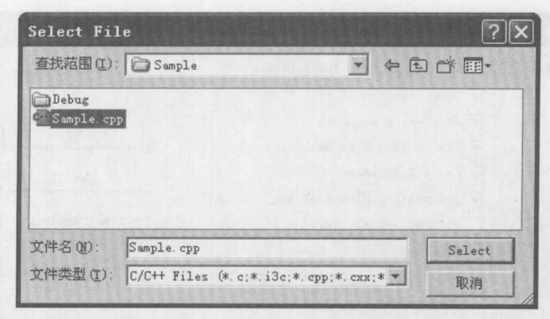


图 A.31　文件选择对话框

选择要分析的源文件“Sample.cpp”，单击“Select”按钮，弹出分析范围设置对话框，如图 A.32 所示。用户可编辑与当前源文件相关的 Sysppvar.dat 文件和 Sysearch.dat 文件。前者可编辑，可在其中罗列用户定义的宏（格式为标准 C/C++宏定义格式）；选中后者，分析时会自动包括当前源文件所在目录下的所有头文件。

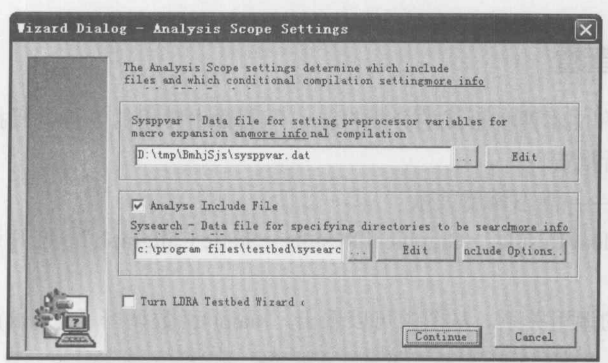


图 A.32　分析范围设置

单击分析范围设置对话框中的“Continue”按钮，返回主界面，选择菜单“Analysis→Select Analysis”，打开分析选项对话框，如图 A.33 所示。

一般选择前 5 项：

- 主要静态分析——编程规则检查；
- 复杂度分析——分析节点数、圈复杂度等；
- 静态数据流分析——对变量进行分析，发现潜在错误；
- 交叉引用——分析变量在程序中的使用情况；
- 信息流分析——分析变量间的依赖关系。

设置完成后，单点击“Start Analysis”按钮，工具开始对选中的源程序进行分析，分析完成后可查看分析结果。

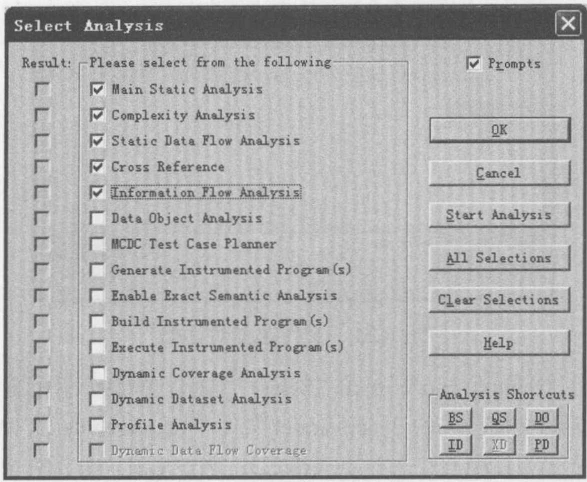


图 A.33　分析项选择

A.3.2　分析结果查看

Testbed 提供图形和文本两种形式的分析报告，下面给出常用的 5 种分析报告。

（1）主要静态分析报告

① 图表形式报告——静态调用关系

选择菜单“Individual Results→Graphical Results→Static Callgraph”，打开系统调用图窗口，如图 A.34 所示。

该图显示了程序的调用关系。从图中可以看出，main()函数调用了 scanf()和 printf()两个函数。

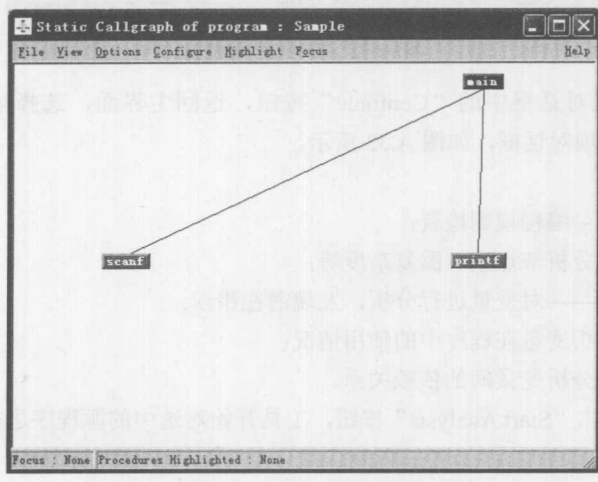


图 A.34　系统调用图

② 文本形式报告——总体报告

文本形式报告中，总体报告、质量报告和度量报告三种报告均有 HTML 和 ASCII 两种格式可供选择，其余报告只有 ASCII 格式。

选择菜单“Individual Results→Text Results→Overview Report(HTML)”，打开 HTML 格式的 Overview Report 窗口，如图 A.35 所示。该报告显示了 Sample 程序中关于指定的编程规则的通过情况。

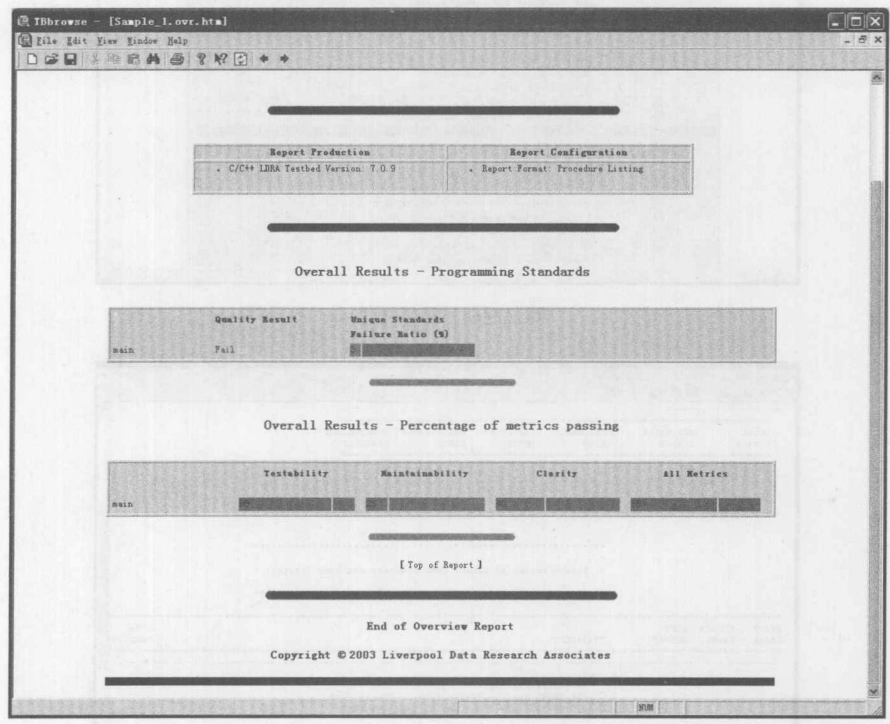


图 A.35 总体报告

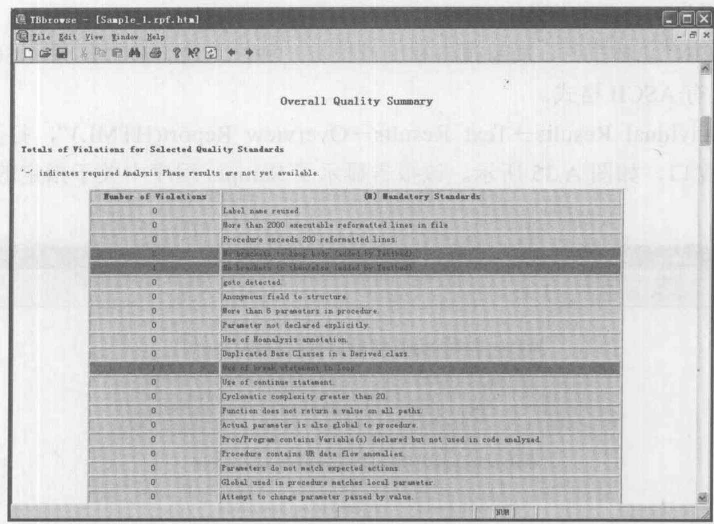
③ 文本形式报告——质量报告

选择菜单“Individual Results→Text Results→Quality Report(HTML)”，打开 HTML 格式的质量报告窗口，如图 A.36 所示。

该报告显示了 Sample 程序中违反编程规则的详细情况（图中深灰色部分），具体信息为：

- 第 9 行和第 35 行的 for 循环体没有使用花括号“{}”；
- 第 15 行 if 后面的执行语句未使用花括号“{}”；
- 第 30 行在 for 循环体中使用 break 语句。

前两条是关于代码可维护性方面的要求，第 3 条是关于代码结构化方面的要求。单击列表中的规则，可查看关于该规则的说明和示例。



Overall Quality Summary

Totals of Violations for Selected Quality Standards

'-' indicates required Analysis Phase results are not yet available.

Number of Violations	(M) Mandatory Standards
0	Label name reused.
0	More than 2000 executable reformatted lines in file
0	Procedure exceeds 200 reformatted lines.
2	No brackets to loop body (added by Testbed).
1	No brackets to then/else (added by Testbed).
0	goto detected.
0	Anonymous field to structure.
0	More than 6 parameters in procedure.
0	Parameter not declared explicitly.
0	Use of Noanalysis annotation.
0	Duplicated Base Classes in a Derived class.
1	Use of break statement in loop.
0	Use of continue statement.
0	Cyclomatic complexity greater than 20.
0	Function does not return a value on all paths.
0	Actual parameter is also global to procedure.
0	Proc/Program contains Variable(s) declared but not used in code analyzed.
0	Procedure contains UR data flow anomalies.
0	Parameters do not match expected actions.
0	Global used in procedure matches local parameter.
0	Attempt to change parameter passed by value.

图 A.36　质量报告

```
--------------------
 ======================================================================
 TOTAL        REACHABLE    UNREACHABLE  MAX. LCSAJ   UNREACHABLE  UNREACHABLE
 LCSAJS       LCSAJS       LCSAJS       DENSITY      LINES        BRANCHES
 ======================================================================
 31           27           4            6            0            0

-----------------------------------------------------------------------------------

                   ****************************************************
                   *                                                  *
                   *  Determination of Linear Code Sequence and Jump Triples  *
                   *                                                  *
                   ****************************************************

==========================================================================================
START   FINISH  LINE                                                            LCSAJ
LABEL   LABEL   NUMBER        STATEMENT                                         DENSITY
==========================================================================================
------------------------------------------------------------------------------------------
                1              /*                                               0
                2              C++ TESTBED VERSION  : 7.0.9                     0
                3              FILE UNDER TEST  : "D:\tmp\Sample\Sample.cpp"    0
                4              DATE OF ANALYSIS : Tue Dec 07 14:38:14 2010      0
                5              */                                               0
------------------------------------------------------------------------------------------
START           6 (1)                                                           2
                7              #include <stdio.h>                               2
                8 (2)                                                           2
                9              void                                             2
                10               main()                                         2
                11 (3)           {                                              2
                12 (4)             int                                          2
                13                   a [ 11 ] = { 1 , 4 , 6 , 9 , 13 , 16 , 19 , 28 , 40 , 100 } ;   2
                14 (5)             int                                          2
                15                   temp1 ,                                    2
                16                   temp2 ,                                    2
                17                   number ,                                   2
                18                   end ,                                      2
                19                   i ,                                        2
                20                   j ;                                        2
                21 (6)             printf ( "original array is:\n" ) ;          2
                22 (7)             for                                          2
                23                   {                                          2
                24                   i = 0                                      2
                25                   ;                                          2
START           26                   i < 10                                     4
        FINISH  27                   ;                                          4
------------------------------------------------------------------------------------------
START           28                   i ++                                       1
        FINISH  29                   }                                          1
------------------------------------------------------------------------------------------
START           30 (8)               {                                          1
                31                     printf ( "%5d" , a [ i ] ) ;             1
For Help, press F1                    Ln 81, Col 81          NUM
```

图 A.37　LCSAJ 密度报告 1

④ 文本形式报告——LCSAJ 报告

选择菜单“Individual Results→Text Results→LCSAJ Report(HTML)”，可查看源代码的 LCSAJ 密度信息，如图 A.37 和 A.38 所示。

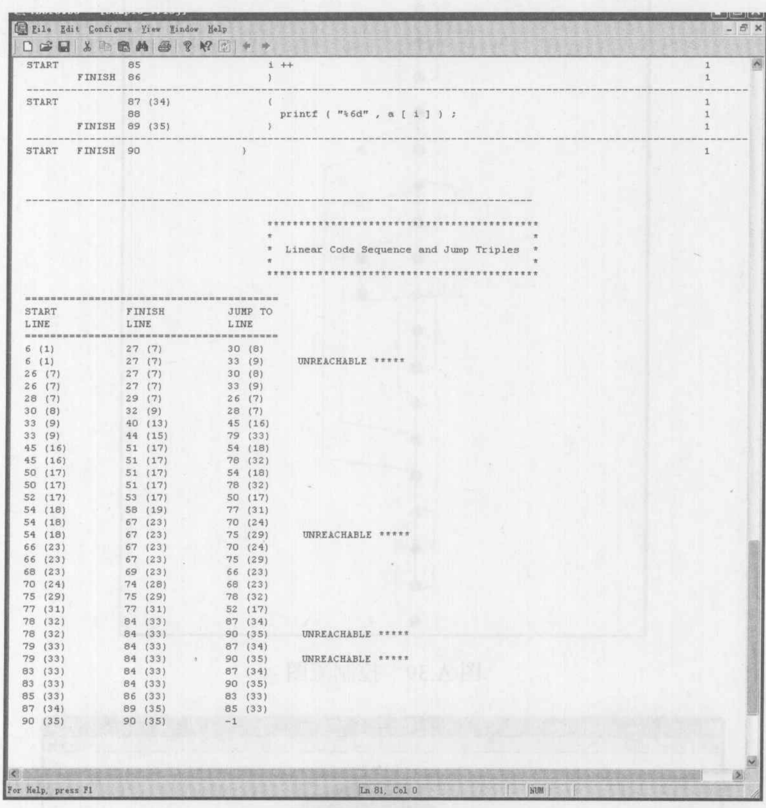


图 A.38 LCSAJ 密度报告 2

（2）复杂度分析报告

① 图表形式报告——静态控制流

选择菜单“Individual Results→Graphical Results→Static Flowgraph”，打开程序控制流图窗口，也可单击系统调用图中的深灰色节点查看相应函数的控制流图，如图 A.39 所示。

在控制流图中单击节点可以调出相应的格式化源代码，图中菱形节点代表该节点所包含的源代码有违反编码规则的情况存在。

② 图表形式报告——Kiviat 图

选择菜单“Individual Results→Graphical Results→Standard Kiviat”，打开标准 Kiviat 图窗口，如图 A.40 所示。Kiviat 图显示被分析的代码在软件质量度量方面和预设的质量模型之间的符合情况。

Kiviat 图的每根轴表示一个度量元，报告以图形和列表形式显示每个度量元的预设下限（Lvb）、

预设上限（Upd）和实际值（Val）。度量元实际值超出预设范围时在 Kiviat 图中该轴显示深灰色。

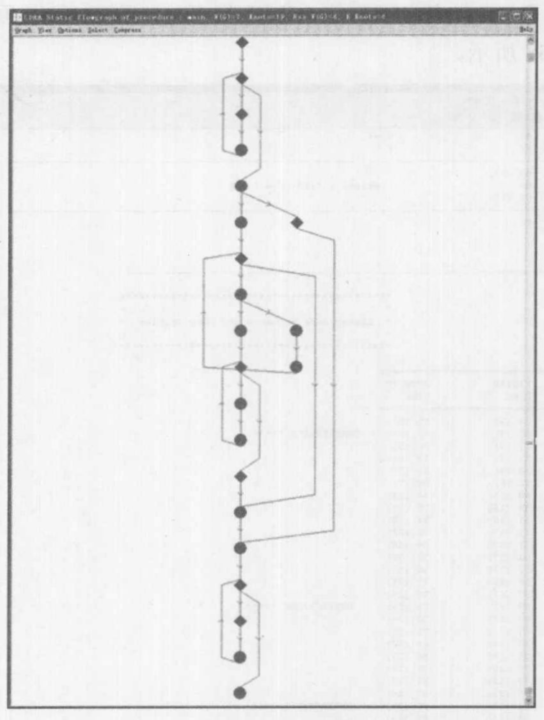


图 A.39　控制流图

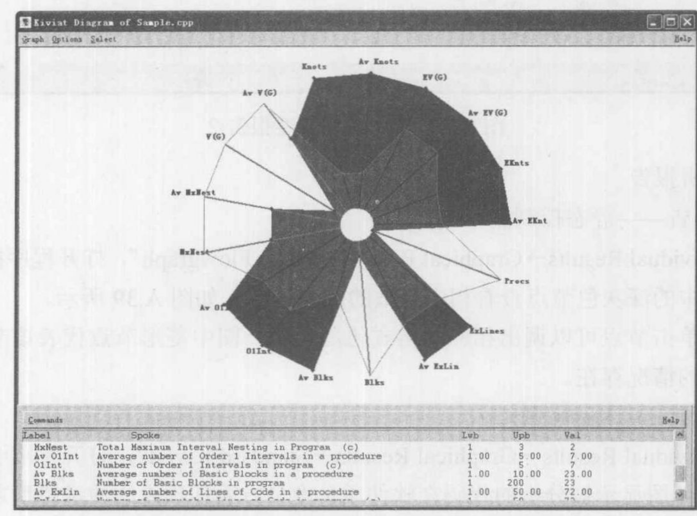


图 A.40　标准 Kiviat 图

“Graphical Results”菜单项下还有四种详细的 Kiviat 图：Clairty Kiviat、Maintainability Kiviat、Testability Kiviat、OO Kiviat。分别报告被测代码的清晰性（可读性和易理解性）、可维护性、可测试性和面向对象特性。

对于本章所用的例子，可分别查看前三种 Kiviat 图，如图 A.41、图 A.42 和图 A.43 所示。

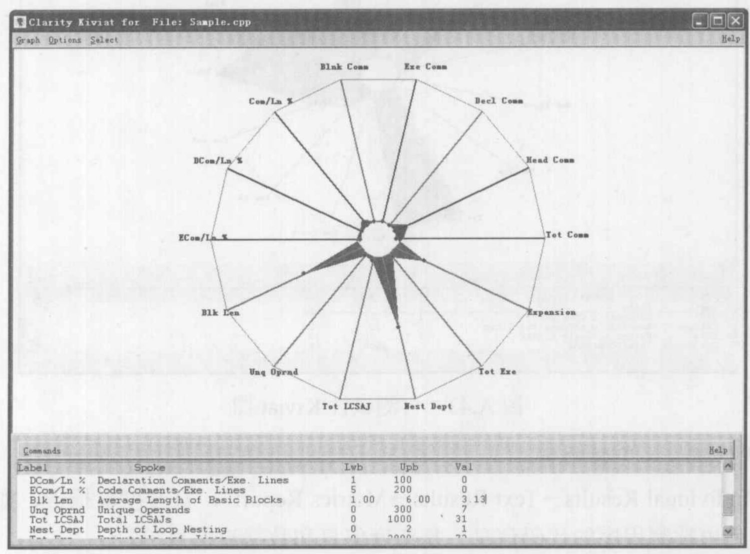


图 A.41 清晰性 Kiviat 图

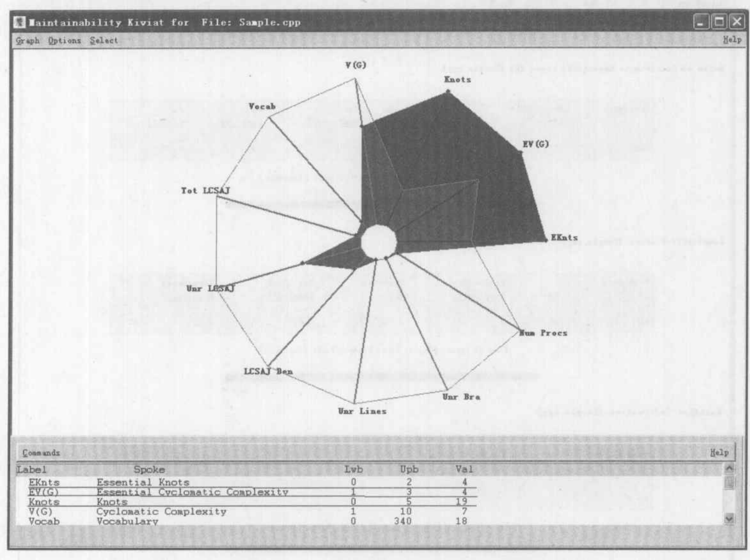


图 A.42 可维护性 Kiviat 图

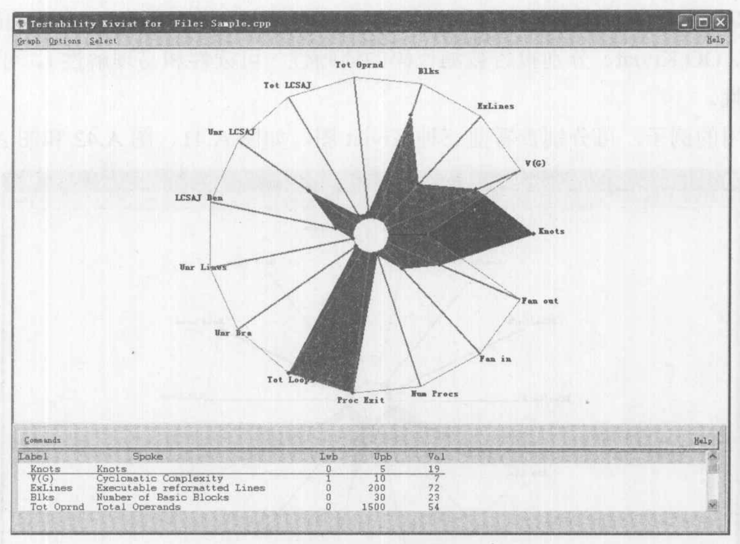


图 A.43 可测试性 Kiviat 图

③ 图表形式报告——度量报告

选择菜单“Individual Results→Text Results→Metrics Report”，打开度量报告，如图 A.44 所示。度量报告按函数列出被测程序的代码信息、复杂度信息和数据流信息等。

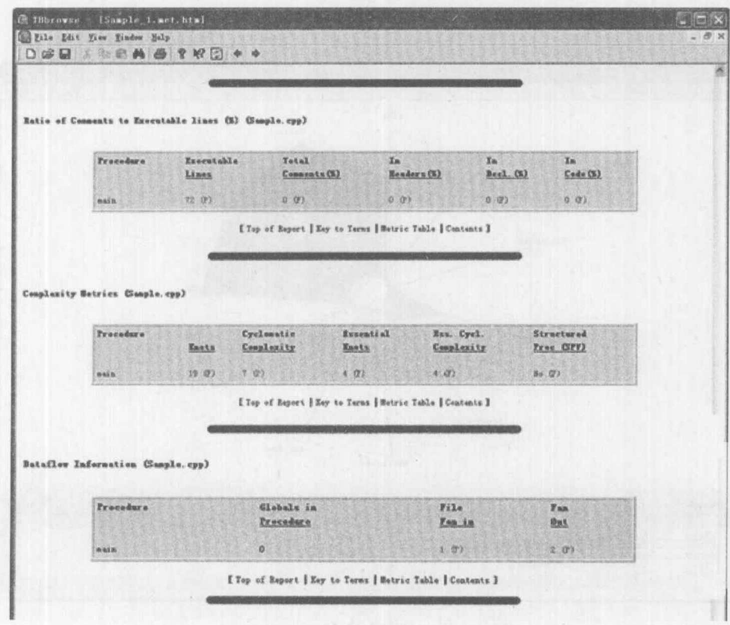


图 A.44 代码度量报告

代码信息中可查看被测程序的可执行格式化代码总行数、注释率等。

复杂度信息可查看控制流节点数、圈复杂度等。控制流节点数和圈复杂度均反映程序的复杂性。在多数质量模型中，推荐控制流节点数小于 5，圈复杂度小于 10。本例中程序节点数为 19，圈复杂度为 7，这是由于在 for 循环中使用了 break 语句（第 30 行），造成该 for 结构块非结构化。

数据流信息可查看被测程序中模块（函数）的扇入扇出数。扇入数是指该函数被其它函数调用的数目，扇出数是指该函数调用其他函数的数目。它们是衡量程序调用关系复杂度的一个指标。模块的扇出一般应控制在 7 以下。为避免程序代码重复，可适当增加模块的扇入；应使高层模块有较高的扇出、低层模块有较高的扇入。

（3）静态数据流分析报告

选择菜单"Individual Results→Text Results→Data Flow Analysis Report"，打开静态数据流分析报告，如图 A.45 所示。

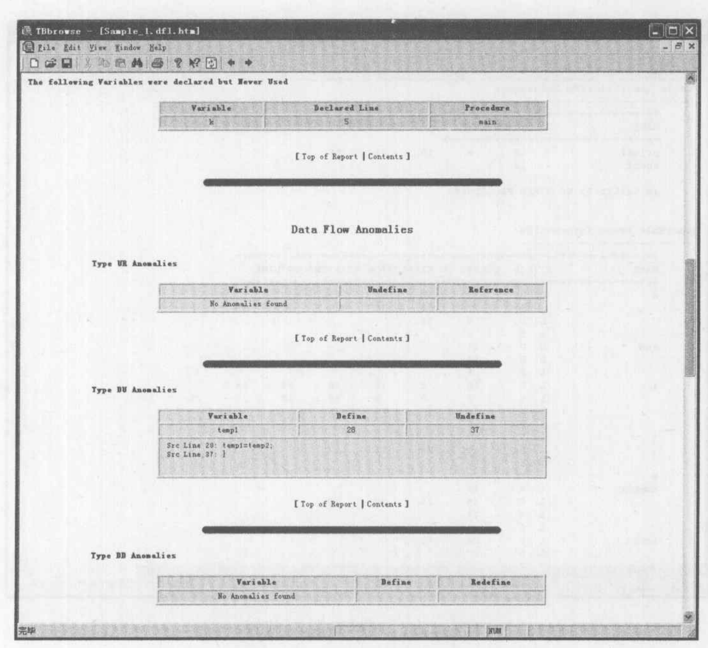


图 A.45 静态数据流分析报告

静态数据流分析报告列出 4 种数据流异常，分别是：

① 未使用变量：定义了变量但从未使用；

② UR：变量未初始化就引用；

③ DU：初始化后未被引用就出作用域了；

④ DD：初始化后未被引用而再次被初始化。

本例中的被测程序有一个未使用变量异常（k）和一个 DU 异常，即最后一次使用 temp2 为 temp1

赋值后就跳到第 34 行的 for 循环，最后为 temp1 赋的值未被引用。

要消除未使用变量异常的缺陷，可直接删掉变量 k 或者在其中一个 for 循环中使用 k。

要消除 DU 异常的缺陷，本例中可在最后增加打印 temp1 的语句。

（4）交叉索引分析报告

选择菜单“Individual Results→Text Results→Cross Reference Report”，打开交叉索引报告，如图 A.46 所示。

交叉索引分析对被分析的文件或工程的所有数据项都进行全面的交叉索引分析，并且给出每个数据项的具体使用情况——是全局的，还是局部的或者是参数，同时以文本的形式给出完整的调用关系。

交叉索引报告最先给出的是调用关系情况，它是以函数为基础的，包括被该函数调用的所有函数和调用该函数的所有函数。

```
TBbrowse - [Sample_1.xrf]
File  Edit  Configure  View  Window  Help

Calls the Following Procedures
------------------------------
    ==================================
    NAME             CALLED ON LINE
    ==================================
    printf              6     9    10    11    35
    scanf              12

    IS CALLED BY NO OTHER PROCEDURES


Variable Usage Information
--------------------------
    ==========================================================
    NAME             ATTRIB  OCCURS IN EXPRESSION STARTING ON LINE
    ==========================================================
    a                 L  D     4    15    23    27
                      L  R     9    13    20    22    26    35
                      L  O     9    35
                      L  E     4
    end               L  D    13
                      L  R    14
                      L  E     5
    i                 L  D     8     8    18    18    34    34
                      L  R     8     8     9    18    18    20    22    23
                      L  R    24    34    34    35
                      L  E     5
    j                 L  D    24    24
                      L  R    24    24    26    27
                      L  E     5
    k                 L  E     5
    number            L  D    12
                      L  R    14    15    20    23
                      L  I    12
                      L  E     5
    temp1             L  D    22    28
                      L  R    27

For Help, press F1          Ln 1, Col 1          NUM
```

图 A.46　交叉索引报告

交叉索引报告还给出被分析的数据项的详细情况，格式如下：

<item name> <attribute code> < list of line numbers>。

其中<attribute code>的具体含义如下：

L——局部变量，在函数体内声明，只在函数体里使用；

G——全局变量，在程序的其他地方声明，在函数体内作为全局变量使用；

P——参数，通过函数调用来使用；

LG——局部全局变量，变量在函数内部声明，但是在程序的其他地方如果该函数作为全局变量

使用，所有对该变量引用的具体位置都会被标识出来；

E——变量在该行声明；

D——变量在该行定义；

R——变量在该行被引用；

I——变量在该行作为输入；

O——变量在该行作为输出。

（5）信息流分析报告

选择菜单“Individual Results→Text Results→Information Flow Analysis Report”，打开信息流分析报告，如图 A.47 所示。信息流分析报告给出各种信息流依赖关系，依赖关系包括：

① m 强直接依赖 n；

② m 强条件依赖 n；

③ m 弱直接依赖 n；

④ m 弱条件依赖 n。

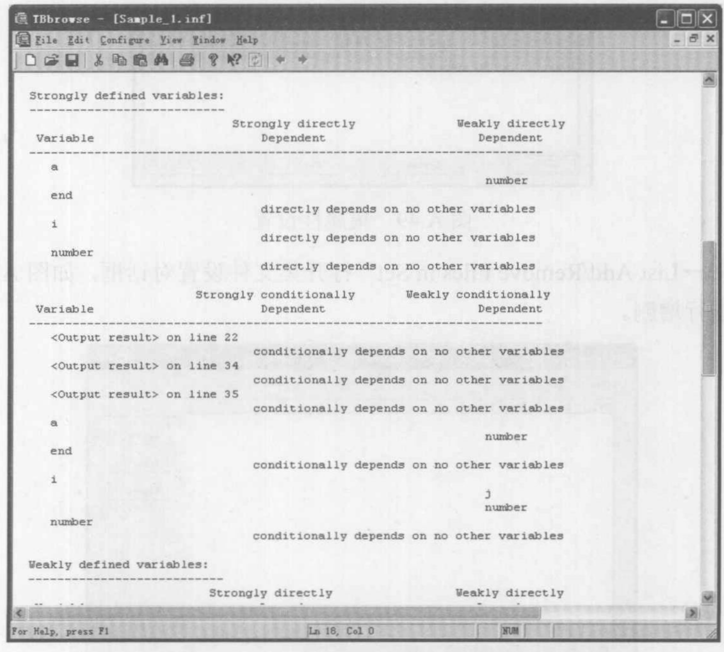


图 A.47　信息流分析报告

A.3.3　多个文件批量分析

Testbed 可以用集（set）的方式同时分析多个文件。

选择菜单“Set→Select/Create/Delete Set”打开集选择/创建/删除对话框，如图 A.48 所示。可以

输入集名创建一个新的集，也可以选择一个已创建的集。

在图 A.48 中输入集名后单击“Create”按钮，弹出集属性对话框，如图 A.49 所示。可以将集设置为“Group”（文件间无关）或“System”（集中的文件作为一个工程进行分析）。

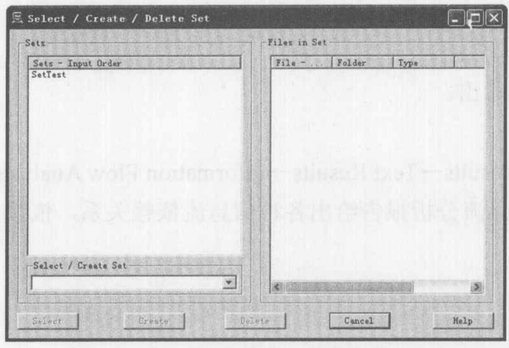


图 A.48　集选择/创建/删除窗口

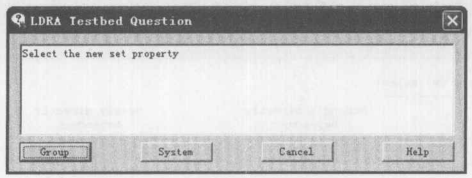


图 A.49　集属性设置

选择菜单“Set→List/Add/Remove Files in Set”打开集文件设置对话框，如图 A.50 所示，可以对属于该集的文件进行增删。

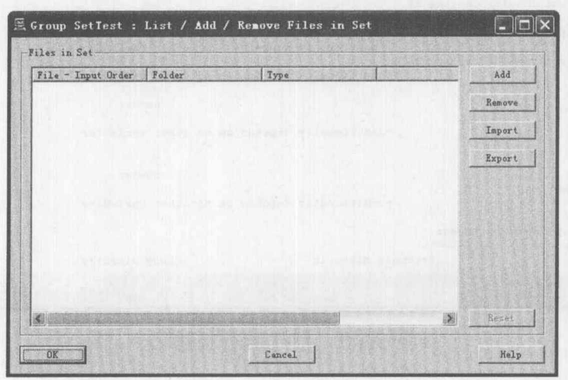


图 A.50　集文件设置

集文件设置结束后就可以对该集进行分析了，分析设置同单个文件分析。

分析结束后，可在“System Results”菜单下查看分析报告，具体内容同单个文件分析。

A.4 McCabe IQ²——软件质量保证工具

McCabe IQ² 是美国 McCabe Software 公司推出的软件质量管理解决方案，该公司由 Thomas McCabe Jr 于 1977 年创建。Thomas 是软件质量方面的专家，1976 年发表了软件复杂性度量理论，1982 年发表了著名的“结构化测试：一种利用圈复杂度进行测试的方法”论文，该论文由美国国家标准技术局（NIST）出版并被 NIST 采纳作为测试标准。

McCabe IQ² 是一款关于软件质量度量、质量分析、静态度量、动态测试和软件再工程的专业工具套件，主要包括：

- McCabe EQ（Enterprise Quality）：软件质量静态分析模块，包括质量度量、质量趋势分析、报告生成等功能；
- McCabe Test：软件动态测试模块，包括路径覆盖、MC/DC（DO178-B A 级）覆盖、分支、语句、条件等覆盖测试；
- McCabe Reengineer：软件再工程质量分析模块，包括对数据变量的分析、变更趋势分析、功能比较分析和回归测试管理等功能。

A.4.1 McCabe EQ

McCabe EQ 为软件系统计算 McCabe 复杂度，在容易理解的可视化环境中评估整个软件的质量，了解需要改进质量的模块软件。McCabe EQ 包含 McCabe QA 和 McCabe Data 两部分。

1. Battlemap（程序结构图）生成

Battlemap 图是一个可视化的程序结构查看工具，通过 Battlemap 图，能够显示程序模块之间的调用关系。Battlemap 图与程序结构图相似，但其包含的功能比程序结构图多，Battlemap 中的每个对象都是“活灵活现”的，将鼠标移到任何连线或者模块单元，都会弹出相应的菜单。

从开始菜单中，选择“程序→McCabe IQ 8.0→Battlemap”，显示空白的 Battlemap 窗口，创建一个新工程，打开现有的工程，Battlemap 界面如图 A.51 所示。

Battlemap 是软件的系统结构图，显示了软件的结构和模块之间的调用关系。Battlemap 中的模块有不同的颜色，红色、绿色、黄色等。不同的颜色代表不同的含义：

红色：表示基本复杂度（ev）超出了门限值；

黄色：表示圈复杂度（V）超出了门限值；

绿色：表示基本复杂度（ev）和圈复杂度（V）都没有超出门限值。

图 A.51 中的连线是模块之间的调用关系，线的颜色表示调用的类型：条件、无条件和循环。在线上单击右键会弹出菜单，如图 A.52 所示。

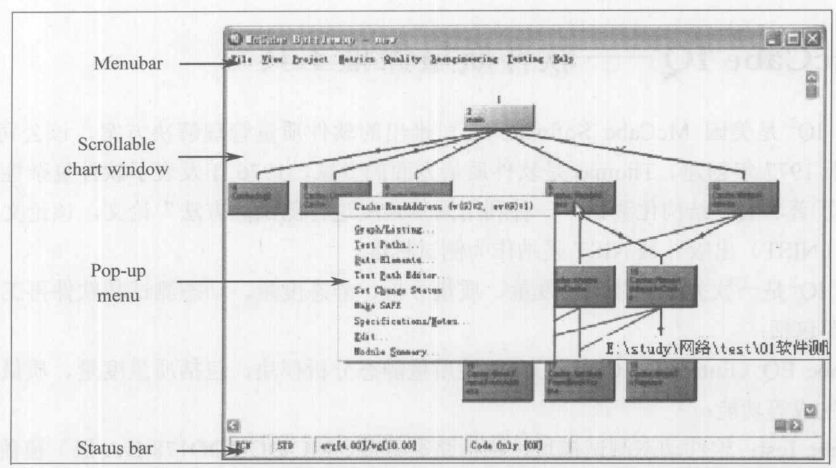


图 A.51 Battlemap 主界面

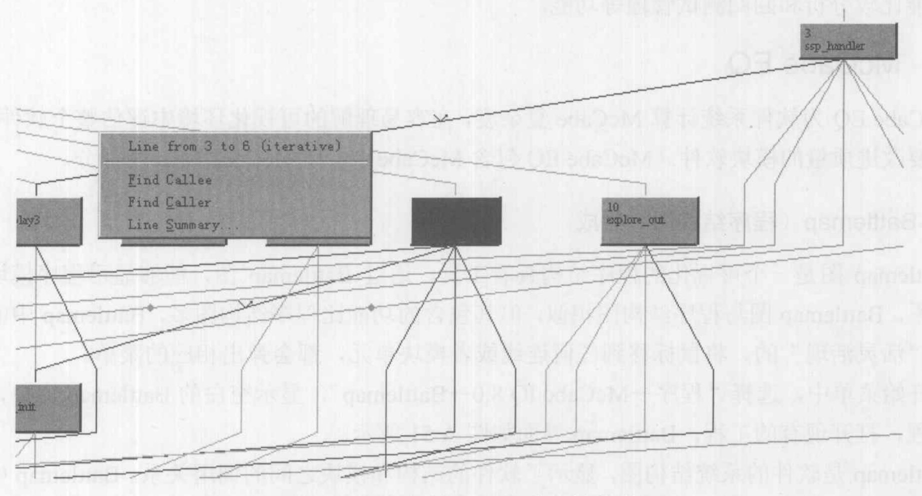


图 A.52 线上单击右键弹出的菜单

2. 程序流程图生成

程序流程图是描述程序控制流的一种图示方法。在程序控制流图的基础上，通过分析控制结构的环路复杂性，导出基本的可执行路径集合，从而设计测试用例。

打开 Battlemap，在图中任意一个模块上单击右键，选择 Graph/Listing，如图 A.53 所示。

Graph/Listing 显示了模块的完整控制结构图，如图 A.54 所示。

图 A.54 中，左侧的 Flowgragh 显示了模块控制结构图，由附带星号的高亮数字显示调用节点；

右侧的 ASL（Annotated Source Listings）显示模块中的注释源代码列表。

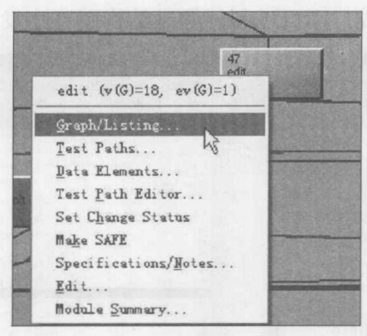


图 A.53 右键弹出菜单

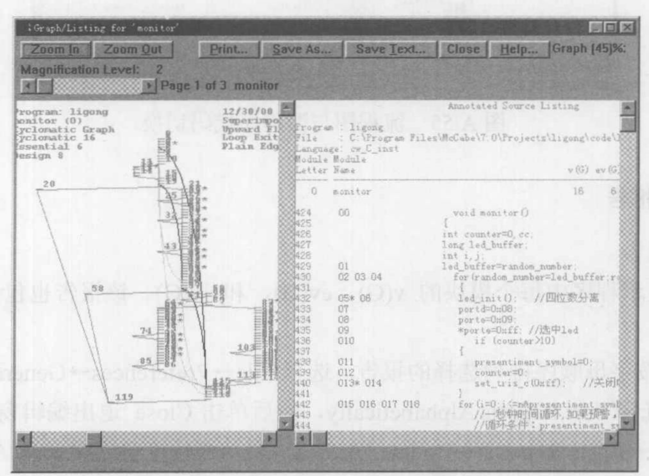


图 A.54 程序流程图及源程序代码

这两个窗口是互动的，可以来回切换，如图 A.55 所示。

在图 A.55 中，通过在 2#节点右键选择“Highlight ASL”，可以快速找到与之对应的 for 语句起始点。

Flowgragh 和 ASL 联合提供下列信息：

- 程序名称和文件名称；
- 程序所用语言类型；
- 模块区分码和名称；
- 模块的圈复杂度，基本和设计圈复杂度度量；
- 模块的起始行号和代码总行数；
- 连续行数；
- 模块中包含的代码列表（如果代码行与 flowgraph 中的多个节点相关，该行的前端将列出节点

标识）。

若程序或文件由多个模块组成，则可以利用控制结构图迅速发现那些极其复杂的模块。

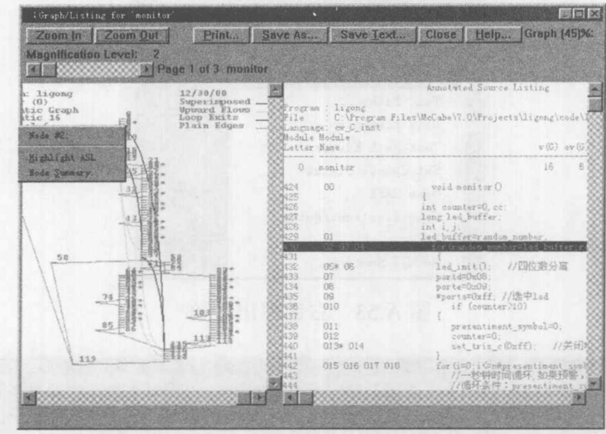


图 A.55　流程图与源程序代码切换

3. 单元级度量报告

（1）模块度量报告

模块度量报告显示程序中每个模块的 v(G)，ev(G)，和 iv(G)。该报告也包含起始行和每个模块的代码行数。

① 打开工程，按字母顺序排列选择的报告。选择 File→Preferences→General，在弹出的 General Preferences 窗口中选择 Sort Reports Alphabetically，然后单击 Close 退出编辑窗口。

② 选择 Metrics→Basic Reports→Module Metrics，显示模块度量结果如图 A.56 所示。

Module Metrics

Print... | Save As... | Save Text... | Save Data... | Apply To Chart | Close | Help...

Page 1　22/08/08

Module Metrics

Program: less

Module Name	Mod #	v(G)	ev(G)	iv(G)	# Lines	Line #
add_back_pos	89	2	1	1	13	277
add_forw_pos	102	2	1	1	13	260
add_line	90	1	1	1	5	591
back_line	88	15	14	13	102	298
back_raw_line	55	6	8	6	51	508
backspace	82	1	1	1	10	689
backward	32	2	1	2	19	454
bell	119	2	1	2	8	609
carat_char	123	2	1	1	6	300
ch:buffered	120	3	3	1	11	373
ch:fch_get	121	12	5	6	92	277
ch_back_get	113	6	5	5	18	469
ch_end_seek	60	3	1	3	16	412
ch_forw_get	109	4	1	2	13	452
ch_init	72	6	1	5	63	492
ch_length	118	2	1	2	7	432
ch_seek	108	4	4	2	19	389
ch_tell	111	1	1	1	5	443
clear	99	1	1	1	5	632
clear_eol	131	1	1	1	5	642
cmd_reset	22	1	1	1	4	247
command:cmd_char	28	9	6	6	38	293

图 A.56　模块度量结果

（2）散点图（Scatter plot）报告

散点图是两个复杂度的坐标图，从系统的角度反映模块复杂度的分布情况，便于对系统质量进行评估。

在 Battlemap 中，选择 Metrics→Graphical→Scatterplot，显示散点图结果如图 A.57 所示。

图 A.57 是基于圈复杂度门限设置为 10，基本复杂度门限设置为 4 而得到的。右半部是文本显示的圈复杂度和基本复杂度的数值；左半部分是采用坐标形式显示的。由于门限设置为 10 和 4，过于复杂、结构很差的模块位于坐标的右上，这些模块需要特别注意，因为它们过于复杂且结构不好，调试、测试、维护都很困难。

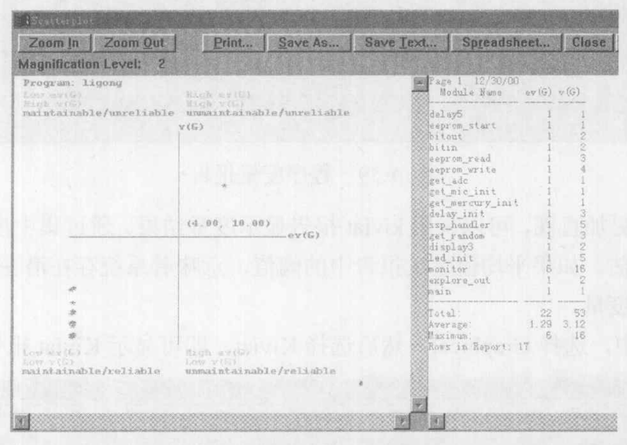


图 A.57　散点图

（3）柱状图（Histogram）报告

柱状图报告使用条形图形式给出系统中选择的度量值。

从 Metrics 菜单中，选择 Graphical→Histogram，即可显示柱状图报告，如图 A.58 所示。

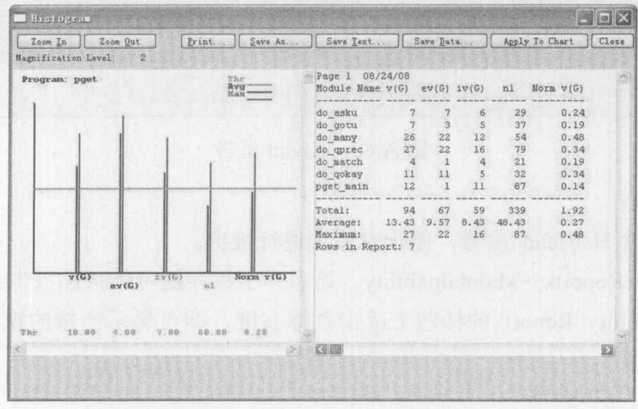


图 A.58　柱状图报告

（4）程序度量报告

程序度量报告显示程序设计复杂度（S_0）（设计测试路径的数量）和集成复杂度（S_1）（子树的数量）。这些度量可以有助于确定整个程序所需的测试资源（子树和设计测试路径的数量越多，所需的测试资源越多）。

选择 Metrics→Basic Reports→Program Metrics，显示程序度量报告如图 A.59 所示。

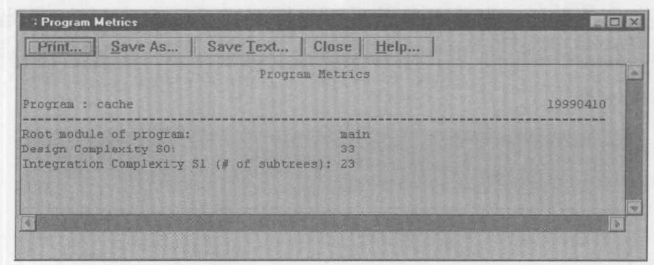


图 A.59　程序度量报告

为了使度量结果更加直观，可以采用 kiviat 报告显示度量结果。通过集中度量平均值，可以对系统的整体质量进行评估。如果平均值超过报告中的阈值，意味着系统存在潜在问题。在 Kiviat 报告中必须至少包含三种度量。

从 Metrics 菜单中，选择 Graphical，然后选择 Kiviat，即可显示 Kiviat 报告，如图 A.60 所示。

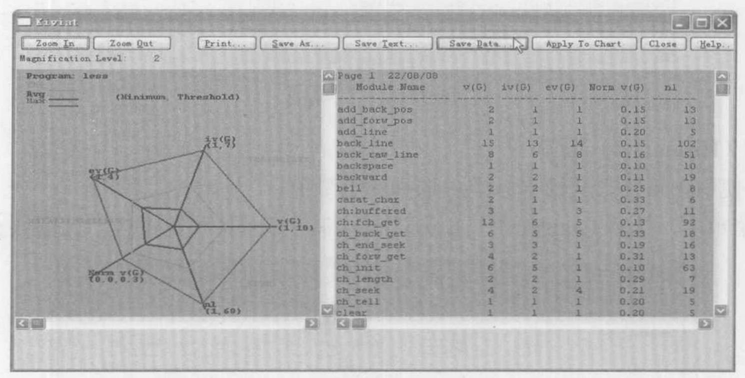


图 A.60　Kiviat 报告

（5）可维护性报告

可维护性报告包含 Halstead 度量，便于对系统进行维护。

① 选择 Quality→Reports→Maintainability，显示一个程序的可维护性报告；

② 在 Maintainability Report 的标题上单击鼠标右键，即可显示链接的模块级维护报告，如图 A.61 所示。

（6）密度报告

密度报告显示采用 Halstead 度量出的圈复杂度、基本复杂度和设计密度复杂度。

选择 Quality→Reports→Density，显示密度报告如图 A.62 所示。

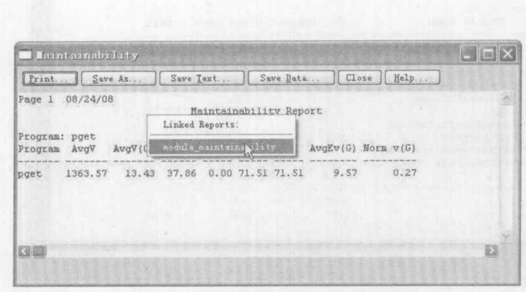


图 A.61 可维护性报告

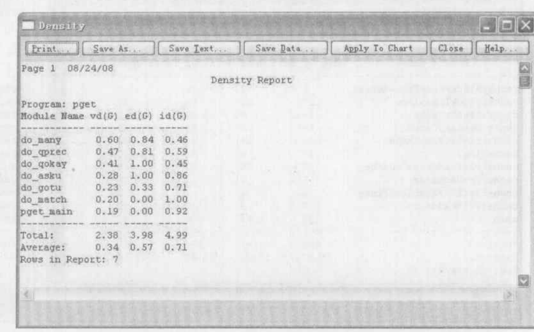


图 A.62 密度报告

（7）Halstead 报告

McCabe QA 提供三种 Halstead 报告，分别是 Halstead 短格式、Halstead 长格式以及操作数计数报告。其中，前两种报告内容相似，只是格式不同；操作数计数报告包含单个操作符和操作数个数的信息，以及操作符和操作数的总数。

选中 Metrics→Basic Reports→Halstead，然后指定报告种类，即可显示选择的报告，如图 A.63、图 A.64 和图 A.65 所示。

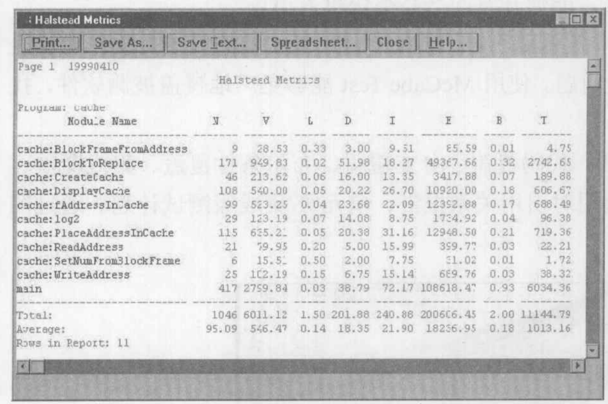


图 A.63 Halstead 短格式报告

图 A.64 Halstead 长格式报告

（8）行度量报告

行度量报告显示每个模块的代码行数，包含代码，注释，空行以及代码注释混合行的统计数据。

选择 Metrics→Basic Reports→Line Count，即可显示行度量报告，如图 A.66 所示。

: Operand Count

Print... Save As... Save Text... Spreadsheet... Close Help...

Page 1 19990410

Operand Count Metrics

Program: cache

Module Name	Uniq Op	Uniq Opnd	Total Op	Total Opnd
cache:BlockFrameFromAddress	6	3	6	3
cache:BlockToReplace	27	20	94	77
cache:CreateCache	16	9	28	18
cache:DisplayCache	14	18	56	52
cache:fAddressInCache	20	19	54	45
cache:log2	13	6	16	13
cache:PlaceAddressInCache	20	26	62	53
cache:ReadAddress	7	7	11	10
cache:SetNumFromBlockFrame	4	2	4	2
cache:WriteAddress	9	8	13	12
main	32	73	240	177
Total:	168	191	584	462
Average:	15.27	17.36	53.09	42.00

Rows in Report: 11

图 A.65 操作数计数报告

: Line Count Metrics

Print... Save As... Save Text... Spreadsheet... Close Help...

Page 1 19990410

Line Count Metrics

Program: cache

Module Name	code	comment	blank	code & comm
cache:BlockFrameFromAddress	5	3	1	0
cache:BlockToReplace	36	1	3	0
cache:CreateCache	11	1	2	0
cache:DisplayCache	17	1	2	0
cache:fAddressInCache	19	1	3	1
cache:log2	9	1	2	0
cache:PlaceAddressInCache	23	1	2	0
cache:ReadAddress	11	1	3	0
cache:SetNumFromBlockFrame	5	3	1	0
cache:WriteAddress	11	1	3	1
main	94	4	16	0
Total:	241	18	38	2
Average:	21.91	1.64	3.45	0.18

Rows in Report: 11

图 A.66 行度量报告

A.4.2 McCabe Test

McCabe Test 是一种计划、监控、彻底性地测试软件的交互式可视化环境。以 NIST 出版的测试标准为基础，McCabe Test 能彻底地对系统进行测试，找出系统中的错误。在覆盖率测试模式下，提供图形化的覆盖测试信息。通过将测试过程自动化、标准化，可以缩短测试周期，可以对测试的完整性进行审查、对测试资源进行精确的计划分配。McCabe Test 集高精度、高集中性、高可靠性的特点于测试过程中，费用低、速度快、测试彻底，能使开发的产品很快投入市场。

McCabe Test 的白盒测试对软件的可靠性非常重要。对于新的软件，最初始的测试是软件功能性检查，直到用户使用时才会发现软件的可靠性问题。使用 McCabe Test 能够更广地覆盖被测软件，找到更多错误，从而提高被测软件的可靠性。

McCabe Test 提供的动态覆盖率报告包括：代码覆盖、分支覆盖、布尔条件覆盖、集成覆盖、McCabe 测试路径覆盖、数据覆盖等，提供调用对引用关系报告、单元级/集成级测试计划。通过图形显示未能测试到的代码，如图 A.67 所示。

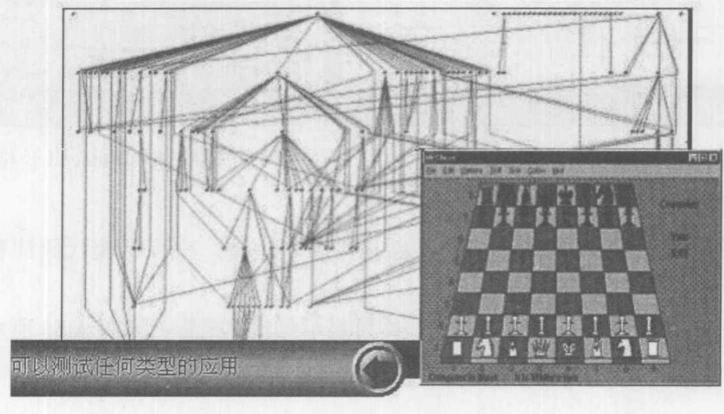


图 A.67 动态覆盖测试结果

A.4.3　McCabe Reengineer

McCabe Reengineer 包括以下几个功能模块：McCabe Slice、McCabe Data、McCabe Compare & Change。

（1）McCabe Slice

- 可以非常直观地显示出已测试和未测试的路径，同时还可以将先前测试过的信息加载进来；
- 分解复杂的系统。可以通过 Slice 将一些模块按功能分开进行测试，然后再合起来进行分析；
- 可以通过 Slice 按照规定要求对代码进行精简。例如，要寻找完成某一特定功能的所有代码，通过 Slice，可以从整个代码中把符合这些功能的代码精简出来，该功能在再工程中非常有用；
- 追踪需求。要完成程序的某个需求，可以通过 Slice 生成已执行代码报告；
- 定义死代码。在 Battlemap 图中，结合覆盖率分析，可以定义出软件中从来没有执行过的代码。

（2）McCabe Data

- 数据字典。通过该模块，可以得到所有数据的报告，特别是一些重要的全局变量和局部变量，同时，可以定义某个数据，并追踪其在整个程序中的运行情况；
- 数据变量管理（DVM）。

（3）McCabe Compare & Change

- 提供了丰富的图形分析虚拟环境。在该环境下，可以有效地分析和修改代码。通过可视化技术和分析方法，在大量逆向工程中节省时间和资源；
- 便于对它人留下的软件系统进行修改。通过充分理解系统结构，对系统改变产生的影响进行分析；
- 精确发现系统维护方面存在的危险性；
- 冗余代码定位。庞大、难于修改的系统通常包括 30%的冗余代码，本工具能识别并准确查找这些冗余代码，从而减小系统的复杂性和规模，提高系统的性能。