[本文部分内容和例子都来自于 PC-Lint 用户手册，翻译得时候加上了点自己的理解]

摘要：C/C++语言的语法拥有其它语言所没有的灵活性，这种灵活性带来了代码效率的提升，但相应增加了代码中存在隐患的可能性。静态代码检查工具 PC-Lint 则偏重于代码的逻辑分析，它能够发现代码中潜在的错误，比如数组访问越界、内存泄漏、使用未初始化变量等。本文将介绍如何安装和配置 PC-Lint 代码检查工具以及如何将 PC-Lint 与常见的代码编辑软件集成。

关键词：代码检查 PC-Lint 规则 选项

目　　录

一 引言

C/C++语言的语法拥有其它语言所没有的灵活性，这种灵活性带来了代码效率的提升，但相应也使得代码编写具有很大的随意性，另外 C/C++编译器不进行强制类型检查，也不做任何边界检查，这就增加了代码中存在隐患的可能性。如果能够在代码提交测试之前发现这些潜在的错误，就能够极大地减轻测试人员的压力，减少软件项目的除错成本，可是传统的 C/C++编译器对此已经无能为力，这个任务只能由专用的代码检查工具完成。目前有很多 C/C++静态代码检查工具，其中 Logiscope RuleChecker 和 PC-Lint 是目前应用比较广泛的两个工具。这两个检查工具各有特色，Logiscope RuleChecker 倾向于代码编码规范的检查，比如代码缩进格式、case 语句书写规范、函数声明和布尔表达式

的编写规则等，而 PC-Lint 则偏重于代码的逻辑分析，它能够发现代码中潜在的错误，比如数组访问越界、内存泄漏、使用未初始化变量等。本文将介绍如何安装和配置 PC-Lint 代码检查工具以及将 PC-Lint 与常见的代码编辑软件，如 Visual C++，Source Insight 集成的方法，同时还将简要介绍一些 PC-Lint 常用的代码检查选项。

二 PC-Lint 介绍

PC-Lint 是 GIMPEL SOFTWARE 公司开发的 C/C++软件代码静态分析工具，它的全称是 PC-Lint/FlexeLint for C/C++，PC-Lint 能够在 Windows、MS-DOS 和 OS/2 平台上使用，以二进制可执行文件的形式发布，而 FlexeLint 运行于其它平台，以源代码的形式发布。PC-lint 在全球拥有广泛的客户群，许多大型的软件开发组织都把 PC-Lint 检查作为代码走查的第一道工序。PC-Lint 不仅能够对程序进行全局分析，识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针以及冗余的代码，还能够有效地帮你提出许多程序在空间利用、运行效率上的改进点。

通过下面的例子就可以看出 PC-Lint 工具的强大功能：

```
1：
2：char *report( int m, int n, char *p )
3：{
4：  int result;
5：  char *temp;
6：  long nm;
7：  int i, k, kk;
8：  char name[11] = "Joe Jakeson";
9：
10：  nm = n * m;
11：  temp = p == "" ? "null" : p;
12：  for( i = 0; i<m; I++ ) {
14：  k++;
15：  kk = i;
16：  }
17：
18：  if( k== 1 ) result = nm;
19：  else if( kk > 0 ) result = 1;
20：  else if( kk < 0 ) result = -1;
21：
22：  if( m == result ) return( temp );
23：  else return( name );
24：}
```

这是一段 C 代码，可以通过大多数常见的 C 语言编译器的检查，但是 PC-Lint 能够发现其中的错误和潜在的问题：第 8 行向 name 数组赋值时丢掉了结尾的 nul 字符，第 10 行的乘法精度会失准，即使考虑到 long 比 int 的字长更长，由于符号位的原因仍然会造成精度失准，第 11 行的比较有问题，第 14 行的变量

k 没有初始化，第 15 行的 kk 可能没有被初始化，第 22 行的 result 也有可能没有被初始化，第 23 行返回的是一个局部对象的地址。

随着 C++语言的出现，C/C++编译器有了更严格的语法检查，但是仍然不能避免出现有 BUG 的程序。C++的类型检查依然不如 Pascal 那么严格。对于一个小程序，多数程序员都能够及时发现上面出现的错误，但是从一个拥有成千上万行代码的大型软件中找出这些瑕疵将是一项烦琐的工作，而且没有人可以保证能找出所有的这类问题。如果使用 PC-Lint，只需通过一次简单的编译就可以检查出这些错误，这将节省了大量的开发时间。从某种意义上说。PC-Lint 是一种更加严格的编译器，它除了可以检查出一般的语法错误外，还可以检查出那些虽然符合语法要求，但很可能是潜在的、不易发现的错误。

三 PC-Lint 的代码检查功能

PC-Lint 能够检查出很多语法错误和语法上正确的逻辑错误，PC-Lint 为大部分错误消息都分配了一个错误号，编号小于 1000 的错误号是分配给 C 语言的，编号大于 1000 的错误号则用来说明 C++的错误消息。表 1 列出了 PC-Lint 告警消息的详细分类：

表 1 列出了 PC-Lint 告警消息分类

错误说明	C	C++	告警级别
语法错误	1-199	1001-1199	1
内部错误	200-299		0
致命错误	300-399		0
告警	400-699	1400-1699	2
消息	700-800	1700-1899	3
可选信息	900-999	1900-1999	4

以 C 语言为例，其中的编号 1-199 指的是一般编译器也会产生的语法错误；编号 200-299 是 PC-Lint 程序内部的错误，这类错误不会出现在代码中的；编号 300-399 指的是由于内存限制等导致的系统致命错误。编号 400-999 中出现的提示信息，是根据隐藏代码问题的可能性进行分类的：其中编号 400-699 指的是被检查代码中很可能存在问题而产生的告警信息；编号 700-899 中出现的信息，产生错误的可能性相比告警信息来说级别要低，但仍然可能是因为代码问题导致的问题。编号 900-999 是可选信息，他们不会被默认检查，除非你在选项中指定检查他们。

PC-Lint/FelexLint 提供了和许多编译器类似的告警级别设置选项 -wLevel，它的告警级别分为以下几个级别，缺省告警级别为 3 级：

-w0 不产生信息（除了遇到致命的错误）
-w1 只生成错误信息 -- 没有告警信息和其它提示信息
-w2 只有错误和告警信息
-w3 生成错误、告警和其它提示信息（这是默认设置）
-w4 生成所有信息

PC-Lint/FelexLint 还提供了用于处理函数库的头文件的告警级别设置选项 -wlib(Level)，这个选项不会影响处理 C/C++源代码模块的告警级别。它有和 -wLevel 相同的告警级别，缺省告警级别为 3 级：

-wlib(0) 不生成任何库信息
-wlib(1) 只生成错误信息（当处理库的源代码时）
-wlib(2) 生成错误和告警信息
-wlib(3) 生成错误、告警和其它信息（这是默认设置）
-wlib(4) 产生所有信息

PC-Lint 的检查分很多种类，有强类型检查、变量值跟踪、语义信息、赋值顺序检查、弱定义检查、格式检查、缩进检查、const 变量检查和 volatile 变量检查等等。对每一种检查类型，PC-Lint 都有很多详细的选项，用以控制 PC-Lint 的检查效果。PC-Lint 的选项有 300 多种，这些选项可以放在注释中（以注释的形式插入代码中），例如：

/*lint option1 option2 ... optional commentary */ 选项可以有多行
//lint option1 option2 ... optional commentary 选项仅为一行（适用于 C++）

选项间要以空格分开，lint 命令一定要小写，并且紧跟在/*或//后面，不能有空格。如果选项由类似于操作符和操作数的部分组成，例如-esym(534, printf, scanf, operator new)，其中最后一个选项是 operator new，那么在 operator 和 new 中间只能有一个空格。PC-Lint 的选项还可以放在宏定义中，当宏被展开时选项才生效。例如：

#define DIVZERO(x) /*lint -save -e54 */ ((x) /0) /*lint -restore */ 允许除数为 0 而不告警

下面将分别介绍 PC-Lint 常用的，也是比较重要的代码检查类型，并举例介绍了各个检查类型下可能出现的告警信息以及常用选项的用法：

3.1 强类型检查

强类型检查选项“-strong”和它的辅助（补充）选项“-index”可以对 typedef 定义的数据类型进行强类型检查，以保证只有相同类型之间的变量才能互相赋值，强类型检查选项 strong 的用法是：

-strong(flags[, name] ...)

strong 选项必须在 typedef 定义类型之前打开，否则 PC-Lint 就不能识别 typedef 定义的数据类型，类型检查就会失效。flags 参数可以是 A、J、X、B、b、l 和 f，相应的解释和弱化字符在表 2 中列出：

表 2 强类型检查 strong 选项和参数表

A	对强类型变量赋值时进行类型检查，这些赋值语句包括：直接赋值、返回值、参数传递、初始化 。 A 参数后面可以跟以下字符，用来弱化 A 的检查强度： i 忽略初始化 r 忽略 Return 语句 p 忽略参数传递 a 忽略赋值操作 c 忽略将常量赋值（包括整数常量、常量字符串等）给强类型的情况 z 忽略 Zero 赋值，Zero 定义为任何非强制转换为强类型的 0 常量。例如：0L 和(int)0 都是 Zero，

	但是(HANDLE)0 当 HANDLE 是一个强类型的时候就不是 Zero。(HANDLE *)0 也不是例如使用 -strong(Ai,BITS)设置，PC-Lint 将会对从非 BITS 类型数据向 BITS 类型数据赋值的代码发出告警，但是忽略变量初始化时的此类赋值。
X	当把强类型的变量赋指给其他变量的时候进行类型检查。弱化参数 i, r, p, a, c, z 同样适用于 X 并起相同的作用。
J	选项是当强类型与其它类型进行如下的二进制操作时进行检查，下面是 J 的参数： e 忽略==、!=和?:操作符 r 忽略>、>=、<和<= o 忽略+、-、*、/、%、|、&和^ c 忽略该强类型与常量进行以上操作时的检查 z 忽略该强类型与 Zero 进行以上操作时的检查 使用忽略意味着不会产生告警信息。举个例子，如果 Meters 是个强类型，那么它只在判断相等和其他关系操作时才会被正确地检查，其它情况则不检查，在这个例子中使用 J 选项是正确的。
B	B 选项有两个效果： 1. 出于强类型检查的目的，假设所有的 Boolean 操作返回一个和 Type 兼容的类型，所谓 Boolean 操作就是那些指示结果为 true 或 false 的操作，包括前面提到的四种关系运算符和两种等于判断符，取反操作符！，二元操作符&&和||。 2. 在所有需要判断 Bolean 值的地方，如 if 语句和 while 语句，都要检查结果是否符合这个强类型，否则告警。 例如 if(a)...当 a 为 int 时，将产生告警，因为 int 与 Bolean 类不兼容，所以必须改为 if(a != 0)。
b	仅仅假定每一个 Bolean 类操作符都将返回一个与 Type 类型兼容的返回值。与 B 选项相比，b 选项的限制比较宽松。
l	库标志，当强类型的值作为参数传递给库函数等情况下，不产生告警。
f	与 B 或 b 连用，表示抑止对 1bit 长度的位域是 Boolean 类型的假定，如果不选该项表示 1bit 长度的位域被缺省假定为 Boolean 类型。

这些选项字符的顺序对功能没有影响。但是 A 和 J 选项的弱化字符必须紧跟在它们之后。B 选项和 b 选项不能同时使用，f 选项必须搭配 B 选项或 b 选项使用，如果不指定这些选项，-strong 的作用就是仅仅声明 type 为强类型而不作任何检查。下面用一段代码演示-strong 选项的用法：

```
//lint -strong(Ab,Bool) <选项是以注释的形式插入代码中>
typedef int Bool;
Bool gt(int a, b)
{
  if(a) return a > b; // OK
  else return 0; // Warning
}
```

例子代码中 Bool 被声明成强类型，如果没有指定 b 选项，第一个 return 语句中的比较操作就会被认为与函数类型不匹配。第二个 return 语句导致告警是因为 0 不是各 Bool 类型，如果添加 c 选项，例如-strong(Acb,Bool)，这个告警就会被抑制。再看一个例子：

```
/*lint -strong( AJXl, STRING ) */
typedef char *STRING;
STRING s;
...
s = malloc(20);
strcpy( s, "abc" );
```

由于 malloc 和 strcpy 是库函数，将 malloc 的返回值赋给强类型变量 s 或将强类型变量 s 传递给 strcpy 时会产生强类型冲突，不过 l 选项抑制了这个告警。

强类型也可用于位域，出于强类型检查的目的，先假定位域中最长的一个字段是优势 Boolean 类型，如果没有优势 Boolean 或位域中没有哪个字段比其它字段长，这个类型从位域被切开的位置开始成为“散”类型，例如：

```
//lint -strong( AJXb, Bool )
//lint -strong( AJX, BitField )
typedef int Bool;
typedef unsigned BitField;
struct foo
{
unsigned a:1, b:2;
BitField c:1, d:2, e:3;
} x;
void f()
{
x.a = (Bool) 1; // OK
x.b = (Bool) 0; // strong type violation
x.a = 0; // strong type violation
x.b = 2; // OK
x.c = x.a; // OK
```



```
x.e = 1; // strong type violation
x.e = x.d; // OK
}
```

上面例子中，成员 a 和 c 是强类型 Bool，成员 d 和 e 是 BitField 类型，b 不是强类型。为了避免将只有一位的位域假设成 Boolean 类型，需要在声明 Boolean 的-strong 中使用 f 选项，上面的例子就应该改成这样：-strong(AJXbf,Bool)。

另一个强类型检查选项是 index，index 的用法是：

-index(flags, ixtype, sitype [, sitype] ...)

这个选项是对 strong 选项的补充，它可以和 strong 选项一起使用。这个选项指定 ixtype 是一个排除索引类型，它可以和 Strongly Indexed 类型 sitype 的数组（或指针）一起使用，ixtype 和 sitype 被假设是使用 typedef 声明的类型名称。flags 可以是 c 或 d，c 允许将 ixtype 和常量作为索引使用，而 d 允许在不使用 ixtype 的情况下指定数组的长度（Dimensions）。下面是一个使用 index 的例子：

```
//lint -strong( AzJX, Count, Temperature )
//lint -index( d, Count, Temperature )
// Only Count can index a Temperature
typedef float Temperature;
typedef int Count;
Temperature t[100]; // OK because of d flag
Temperature *pt = t; // pointers are also checked
// ... within a function
Count i;
t[0] = t[1]; // Warnings, no c flag
for( i = 0; i < 100; i++ )
t[i] = 0.0; // OK, i is a Count
119
pt[1] = 2.0; // Warning
i = pt - t; // OK, pt-t is a Count
```

上面的例子中，Temperature 是被强索引类型，Count 是强索引类型。如果没有使用 d 选项，数组的长度将被映射成固有的类型：

```
Temperature t[ (Count) 100 ];
```

但是，这是个小麻烦，像下面那样将数组长度定义成常量更好一些：

```
#define MAX_T (Count) 100
Temperature t[MAX_T];
```

这样做还有一个好处就是同样的 MAX_T 还可以用在 for 语句中，用于限制 for 语句的范围。需要注意的是，指向强被索引类型的指针（例如上面的 pt）如果用在[]符号（数组符号）中也会被检查类型。其实，无论何时，只要将一个值加到一个指向强被索引类型的指针时，这个值就会被检查以确认它是一个强索引类型。此外，强被索引指针如果减去一个值，其结果被认为是平常的强索引，所以下面的例子就不会产生告警：

```
i = pt - t;
```

3.2 变量值跟踪

3.2.1 变量值初始化跟踪

早期的变量值跟踪技术主要是对变量值的初始化进行跟踪，和变量初始化相关的 LINT 消息主要是 644, 645 ("变量可能没有初始化"), 771, 772 ("不可靠的初始化"), 530 ("未初始化的"), and 1401 - 1403 ("成员 ... 未初始化")。以下面的代码为例：

```
if( a ) b = 6;
else c = b;     // 530 message
a = c;          // 645 message
```

假设 b 和 c 在之前都没有初始化，PC-Lint 就会报告 b 没有初始化（在给 c 赋值的时候）和 c 可能没有被初始化（在给 a 赋值的时候）的消息。而 while 和 for 循环语句和上面的 if 语句稍微有所不同，比较下面的代码：

```
while ( n-- )
{
b = 6;
...
}
c = b;  //772 message
```

假设 b 在使用之前没有被初始化，这里会报告 b 可能没有初始化的消息（当给 c 赋值时）。之所以会有这样的区别，是因为程序设计者可能知道这样的循环体总是会被至少执行一次。相反，前面的 if 语句，对于程序设计者来说比较难以确定 if 语句是否总会被执行，因为如果是这样的话，这样的 if 语句就是多余的，应该被去掉。While 语句和 if 比较相似，看下面的例子：

```
switch ( k )
{
case 1: b = 2; break;
case 2: b = 3;
/* Fall Through */
case 3: a = 4; break;
default: error();
}
c = b;   //645 message
```

尽管 b 在两个不同的地方被赋值，但是仍然存在 b 没有被初始化的可能。因此，当 b 赋值给 c 的时候，就会产生可能没有初始化的消息。为了解决这个问题，你可以在 switch 语句之前给 b 赋一个默认值。这样 PC-Lint 就不会产生告警消息，但是我们也失去了让 PC-Lint 检查后续的代码修改引起的变量初始化问题的机会。更好的方法是修改没有给 b 赋值的 case 语句。

如果 error()语句代表那些“不可能发生”的事情发生了，那么我们可以让 PC-Lint 知道这一段其实是不可能执行的，下面的代码表明了这一点：

```
switch ( k )
{
case 1: b = 2; break;
```

```
case 2:
case 3: b = 3; a = 4; break;
default: error();
/*lint -unreachable */
}
c = b;
```
注意：这里的-unreachable 应该放在 error()后面，break 的前面。另外一个产生”没有初始化”告警的方式是传递一个指针给 free（或者采用相似的方法）。比如：
```
if( n ) free( p );
...
p->value = 3;
```
在访问 p 的时候会产生 p 可能没有被初始化的消息。对于 goto 语句，前向的 goto 可能产生没有初始化消息，而向后的 goto 会被忽略掉这种检查。
```
if ( a ) goto label;
b = 0;
label: c = b;
```
当在一个大的项目中使用未初始化变量检查时，可能会产生一些错误的报告。这种报告的产生，很大一部分来自于不好的程序设计风格，或者包括下面的结构：
```
if( x ) initialize y
...
if( x ) use y
```
当出现这种情况时，可以采用给 y 赋初始值的方式，或者利用选项-esym(644,y)关掉变量 y 上面的初始化检查。

3.2.2 变量值跟踪

变量值跟踪技术从赋值语句、初始化和条件语句中收集信息，而函数的参数被默认为在正确的范围内，只有在从函数中可以收集到的信息与此不符的情况下才产生告警。与变量值跟踪相关的消息有：

(1) 访问地址越界消息（消息 415，661，796）
(2) 被 0 除消息（54，414，795）
(3) NULL 指针的错误使用（413，613，794）
(4) 非法指针的创建错误（416，662，797）
(5) 冗余的布尔值测试（774）

看下面的例子：
```
int a[10];
int f()
{
int k;
k = 10;
return a[k]; // Warning 415
}
```
这个语句会产生警告 415（通过 '[' 访问越界的指针），因为 PC-Lint 保存了

赋给 k 的值，然后在使用 k 的时候进行了判断。如果我们把上面的例子稍加修改：

```
int a[10];
int f( int n )
{
int k;
if ( n ) k = 10;
else k = 0;
return a[k]; // Warning 661
}
```

这样就会产生告警 661 (可能访问越界指针)。使用“可能”是因为不是所有的路径都会把 10 赋值给 k。PC-Lint 不仅收集赋值语句和初始化，还从条件语句中收集值的信息。比如下面的例子：

```
int a[10];
int f( int k, int n )
{
if ( k >= 10 ) a[0] = n;
return a[k]; // Warning 661 -- k could be 10
}
```

这里仍然产生 661 告警，因为 PC-Lint 检测到，在使用 k 的时候，k 的值>=10。另外，对于函数来说，它总是假设 K 是正确的，程序使用者知道他们要做些什么，所以下面的语句不会产生告警：

```
int a[10];
int f( int k, int n )
{ return a[k+n]; } // no warning
```

和检查变量没有初始化一样，还可以检查变量的值是否正确。比如，如果下面例子中的循环一次都没有运行，k 可能会超出范围。这时候会产生消息 796 (可预见的地址访问越界).

```
int a[10];
int f(int n, int k)
{
int m = 2;
if( k >= 10 ) m++; // Hmm -- So k could be 10, eh?
while( n-- )
{ m++; k = 0; }
return a[k]; // Info 796 - - k could still be 10
}
```

下面的例子演示了可能使用 NULL 指针的问题：

```
int *f( int *p )
{
if ( p ) printf( "\n" ); // So -- p could be NULL
printf( "%d", *p ); // Warning
return p + 2; // Warning
}
```

这里会产生两个告警，因为可能使用了 NULL 指针，很明显，这两个语句应该在 if 语句的范围内。为了使你的程序更加健壮，你可能需要打开 Pointer-parameter-may-be-NULL 这个开关（+fpn）。这个选项假设所有传递到函数中的指针都有可能是 NULL 的。数组边界值在高位被检测，也就是说
int a[10]; ... a[10] = 0;
被检测了，而 a[-1]却检测不到。PC-Lint 中有两个消息是和指针的越界检查有关的，一个是越界指针的创建，另外一个是越界指针的访问，也就是通过越界指针获取值。在 ANSI C([1]3.3.6)中，允许创建指向超过数组末尾一个单元的指针，比如：

```
int a[10];
f( a + 10 ); // OK
f( a + 11 ); // error
```

但是上面创建的两个指针，都是不能访问的，比如：

```
int a[10], *p, *q;
p = a + 10; // OK
*p = 0; // Warning (access error)
p[-1] = 0; // No Warning
q = p + 1; // Warning (creation error)
q[0] = 0; // Warning (access error)
```

布尔条件检查不象指针检查那么严格，但是它会对恒真的布尔条件产生告警，比如：

```
if ( n > 0 ) n = 0;
else if ( n <= 0 ) n = -1; // Info 774
```

上面的代码会产生告警（774），因为第二个条件检查是恒真的，可以忽略。这种冗余代码不会导致问题，但它的产生通常是因为逻辑错误或一种错误可能发生的征兆，需要详细的检查。

3.2.3 使用 assert（断言）进行补救

在某些情况下，虽然根据代码我们可以知道确切的值，但是 PC-Lint 却无法获取所有情况下变量的值的范围，这时候会产生一些错误的告警信息，我们可以使用 assert 语句增加变量取值范围信息的方法，来抑制这些错误的告警信息的产生。下面举例来说明：

```
char buf[4];
char *p;
strcpy( buf, "a" );
p = buf + strlen( buf ); // p is 'possibly' (buf+3)
p++; // p is 'possibly' (buf+4)
*p = 'a'; // Warning 661 - possible out-of-bounds reference
```

PC-Lint 无法知道在所有情况下变量的值是多少。在上面的例子中，产生告警的语句其实并不会带来什么危害。我们可以直接使用

```
*p = 'a'; //lint !e661
```

来抑制告警。另外，我们还可以使用 assert 工具来修正这个问题：

```
#include <assert.h>
...
```

```
char buf[4];
char *p;
strcpy( buf, "a" );
p = buf + strlen( buf );
assert( p < buf + 3 ); // p is 'possibly' (buf+2)
p++; // p is 'possibly' (buf+3)
*p = 'a'; // no problem
```

由于 assert 在 NDEBUG 被定义时是一个空操作，所以要保证 Lint 进行的时候这个宏没有被定义。

为了使 assert()和你的编译器自带的 assert.h 一起产生上面的效果，你需要在编译选项文件中添加一个选项。例如，假设 assert 是通过以下的编译器宏定义实现的：

```
#define assert(p) ((p) ? (void)0 : __A(...))
```

考虑到__A()会弹出一个消息并且不会返回，所以这个需要添加的选项就是：

```
-function( exit, __A )
```

这个选项将 exit 函数的一些非返回特征传递给__A 函数。做为选择结果，编译器可能将 assert 实现成一个函数，例如：

```
#define assert(k) _Assert(k,...)
```

为了让 PC-lint 知道_Assert 是一个 assert 函数，你需要使用

```
-function( __assert, _Assert )
```

选项或

```
-function( __assert(1), _Assert(1) )
```

选项复制__assert()函数的语义

许多编译器的编译选项文件中已经存在这些选项了，如果没有的话，你可以复制一个 assert.h 文件到 PC-lint 目录下（这个目录由于使用了-i 选项，文件搜索的顺序优先于编译器的头文件目录）。

3.2.4 函数内变量跟踪

PC-Lint 的函数值跟踪功能会跟踪那些将要传递给函数（作为函数参数）变量值，当发生函数调用时，这些值被用来初始化函数参数。这种跟踪功能被用来测定返回值，记录额外的函数调用，当然还可以用来侦测错误。考察下面的例子代码：

t1.cpp:

```
1 int f(int);
2 int g()
3 { return f(0); }
4 int f( int n )
5 { return 10 / n; }
```

在这个例子中，f()被调用的时候使用 0 作为参数，这将导致原本没有问题的 10/n 语句产生被 0 除错误，使用命令 lin -u t1.cpp 可以得到以下输出：

```
--- Module: t1.cpp
During Specific Walk:
File t1.cpp line 3: f(0)
t1.cpp 5 Warning 414: Possible division by 0 [Reference:File t1.cpp: line 3]
```

你第一个注意到的事情是短语“During Specific Walk”，紧接着是函数调用发生的位置，函数名称以及参数，再下来就是错误信息。如果错误信息中缺少了错误再现时的错误行和用来标记错误位置的指示信息，这是因为检查到错误的时候代码（被调用函数的代码）已经走过了。如果像下面一样调换一下两个函数的位置：

t2.cpp:

```
1 int f( int n )
2 { return 10 / n; }
3 int g()
4 { return f(0); }
```

这种情况下就不会出现被 0 除的告警，因为此时 f(0)在第四行，函数 f()的代码已经过了，在这种情况下就需要引入 multi-pass 选项。如果在刚才的例子中使用 lin -u -passes(2) t2.cpp 命令，那么输出就变成：

```
--- Module: t2.cpp
/// Start of Pass 2 ///
--- Module: t2.cpp
During Specific Walk:
File t2.cpp line 4: f(0)
t2.cpp 2 Warning 414: Possible division by 0 [Reference:File t2.cpp: line 4]
```

使用-passes(2)选项将会检查代码两遍，一些操作系统不支持在命令行中使用-passes(2)，对于这样的系统，可以使用-passes=2 或 -passes[2]代替。通过冗长的信息可以看出来，以 pass 2 开始表示第一次检查没有产生告警信息。这一次得到的错误信息和前一次不同，在某种情况下我们可以推断出指定函数调用的返回值，至少可以得到一些返回值的属性。以下面的模块为例：

t3.cpp:

```
1 int f( int n )
2 { return n - 1; }
3 int g( int n )
4 { return n / f(1); }
```

使用命令 lin -u -passes(2) t3.cpp，可以得到以下输出信息：

```
--- Module: t3.cpp
/// Start of Pass 2 ///
--- Module: t3.cpp

{ return n / f(1); }
t3.cpp 4 Warning 414: Possible division by 0 [Reference:File t3.cpp: lines 2, 4]
```

第一遍检查我们知道调用函数 f()传递的参数是 1，第二遍检查先处理了函数 f()，我们推断出这个参数将导致返回结果是 0，当第二遍检查开始处理函数 g()的时候，产生了被 0 除错误。应该注意到这个信息并不是在短语“During Specific Walk”之前出现的，这是因为错误是在对函数 g()进行正常的处理过程中检测到的，此时并没有使用为函数 g()的参数指定的值。指定的函数调用能够产生附加

的函数调用，如果我们 pass 足够多的检测次数，这个过程可能会重复发生，参考下面的代码：

t4.cpp:

```
1 int f(int);
2 int g( int n )
3 { return f(2); }
4 int f( int n )
5 { return n / f(n - 1); }
```

第五行的分母 f(n-1)并不会引起怀疑，直到我们意识到 f(2)调用将导致 f(1)调用，最终会调用 f(0)，迫使最终的返回值是 0。使用下面的命令行：

lin -u -passes(3) t4.cpp，

输出结果如下：

```
--- Module: t4.cpp
{ return f(2); }
t4.cpp 3 Info 715: Symbol 'n' (line 2) not referenced
/// Start of Pass 2 ///
--- Module: t4.cpp
/// Start of Pass 3 ///
--- Module: t4.cpp
During Specific Walk:
File t4.cpp line 3: f(2)
File t4.cpp line 5: f(1)
t4.cpp 5 Warning 414: Possible division by 0 [Reference:File t4.cpp: lines 3, 5]
```

到这里已经处理了三遍才检测到可能的被 0 除错误，想了解为什么需要处理三遍可以看看这个选项-specific_wlimit(n)。需要注意的是，指定的调用序列，f(2)，f(2)，是作为告警信息的序言出现的。

3.3 赋值顺序检查

当一个表达式的值依赖于赋值的顺序的时候，会产生告警 564。这是 C/C++语言中非常普遍的一个问题，但是很少有编译器会分析这种情况。比如

```
n++ + n
```

这个语句是有歧义的，当左边的+操作先执行的话，它的值会比右边的先执行的值大一，更普遍的例子是这样的：

```
a[i] = i++;
f( i++, n + i );
```

第一个例子，看起来好像自加操作应该在数组索引计算以后执行，但是如果右边的赋值操作是在左边赋值操作之前执行的话，那么自加一操作就会在数组索引计算之前执行。虽然，赋值操作看起来应该指明一种操作顺序，但实际上是没有的。第二个例子是有歧义的，是因为函数的参数值的计算顺序也是没有保证的。能保证赋值顺序的操作符是布尔与（&&）或（||）和条件赋值（? :）以及逗号(,)，因此：

```
if( (n = f()) && n > 10 ) ...
```

这条语句是正确的，而：

```
if( (n = f()) & n > 10 ) ...
```
将产生一条告警。

3.4 弱定义检查

这里的弱定义包含是以下内容：宏定义、typedef 名字、声明、结构、联合和枚举类型。因为这些东西可能在模块中被过多定义且不被使用，PC-Lint 有很多消息用来检查这些问题。PC-Lint 的消息 749-769 和 1749-1769 都是保留用来作为弱定义提示的。

(1) 当一个文件#include 的头文件中没有任何引用被该文件使用，PC-Lint 会发出 766 告警。

(2) 为了避免一个头文件变得过于大而臃肿，防止其中存在冗余的声明，当一个头文件中的对象声明没有被外部模块引用到时，PC-Lint 会发出 759 告警。

(3) 当变量或者函数只在模块内部使用的时候，PC-Lint 会产生 765 告警，来提示该变量或者函数应该被声明为 static。

如果你想用 PC-Lint 检查以前没有检查过的代码，你可能更想将这些告警信息关闭，当然，如果你只想查看头文件的异常，可以试试这个命令：

```
lint -w1 +e749 +e?75? +e?76? ...
```

3.5 格式检查

PC-Lint 会检查 printf 和 scanf(及其家族)中的格式冲突，例如：

```
printf( "%+c", ... )
```
将产生 566 告警，因为加号只在数字转换时有用，有超过一百个这样的组合会产生告警，编译器通常不标记这些矛盾，其他的告警还有对坏的格式的抱怨，它们是 557 和 567。我们遵循 ANSI C 建立的规则，可能更重要的是我们还对大小不正确的格式进行标记（包括告警 558, 559, 560 和 561）。比如 %d 格式，允许使用 int 和 unsigned int，但是不支持 double 和 long（如果 long 比 int 长），同样，scanf 需要参数指向的对象大小正确。如果只是参数的类型（不是大小）与格式不一致，那将产生 626 和 627 告警。-printf 和 -scanf 选项允许用户指定与 printf 或 scanf 函数族类似的函数，-printf_code 和 -scanf_code 也可以被用来描述非标准的 % 码。

3.6 缩进检查

根据代码中的缩进问题，PC-Lint 也会产生相应的告警，因为缩进的问题有很大一部分是由于代码结构不良或者大括号的遗漏造成的。比如下面的例子：

```
if( ... )
if( ... )
statement
else statement
```
很明显这里的 else 是和第一个 if 语句对应的，而在这里编译器则把它和第二个 if 对应起来。PC-Lint 会对这种情况产生告警。和这样的缩进检查相关的告警主要有三个 725(no positive indentation)、525(negatively indented from)、539（Did not expect positive indentation from Location）要进行缩进检查，我们首先要设置文件中的 tab 键所对应的空格数，默认的是占用 8 个空格，

这个参数可以用-t#选项进行修改。比如-t4 表示 tab 键占用 4 个空格长度。另外，缩进检查还和代码的编码格式策略相关，需要进行必要的调整。

3.7 const 变量检查

对于 const 变量的检查，PC-Lint 是完全支持的。使用 const 变量，对于提高代码的质量非常有好处，看一下下面的例子：

```
char *strcpy( char *, const char * );
const char c = 'a';
const char *p = &c;
void main()
{
char buf[100];
c = 'b';
*p = 'c';
strcpy( p, buf );
...
```

这里的 c 和*P 指向的内容都是静态变量，不可修改。上面的代码明显违反了这个规定，会产生 Error(11)，另外，把 P 作为第一个参数传入 strcpy 中，会产生告警 605（Increase in pointer capability），而把 buf 作为第二个参数传入 strcpy 函数中，会产生告警 603（Symbol 'Symbol' (Location) not initialized），因为 buf 没有初始化，而作为静态变量的第二个参数，是不能在 strcpy 函数中再被初始化的。

3.8 volatile 变量检查

对于 volatile 变量的检查，在 PC-Lint 中有这样的规定，如果一个表达式中同时使用了两次相同的 volatile 变量，那么就会给出 564 告警，因为这时候会产生赋值顺序的问题。

```
volatile char *p;
volatile char f();
n = (f() << 8) | f(); /* Warning 564 */
n = (*p << 8) | *p; /* Warning 564 */
```

四 PC-Lint 软件使用方法

4.1 安装与配置

PC-lint 软件性价比高，易于学习，容易推广和固化到软件开发测试流程中去，所以在全世界得到了广泛的应用。PC-lint 使用方法很简单，可以用命令行方式进行，例如 lint-nt –u std.lnt test1.c test2.c test3.c 也可以使用 MAKEFILE 的方式。此外，它还可以集成到很多开发环境或常用的代码编辑软件中，比如集成到 Source Insight/SLICKEDIT/MS VC6.0/KEIL C..等。

PC-Lint 还支持 Scott Meyes 的名著（Effective C++/More Effective C++）中说描述的各种提高效率和防止错误的方法。

PC-lint 的安装非常简单，以 PC-lint 8.0 为例，运行安装程序将其释放到指定的安装目录即可，比如 c:\pclint8。然后需要运行 PC-lint 的配置工具 config.exe 生成选项和检查配置文件，以刚才的安装路径为例，config.exe 应该位于：C:\pclint8\config.exe。配置文件是代码检查的依据，PC-lint 自带了一个标准配置文件 std.lnt，但是这个文件没有目录包含信息（头文件目录），通常对代码检查的时候都需要指定一些特殊的包含目录，所以要在标准配置的基础上生成针对某个项目代码检查的定制配置。下面就以 Microsoft Visual C++ 6 的开发环境为例，介绍一下定制配置的过程。

运行 C:\pclint8\config.exe 后出现一个欢迎界面，提示版权信息，如图 4.1 所示：

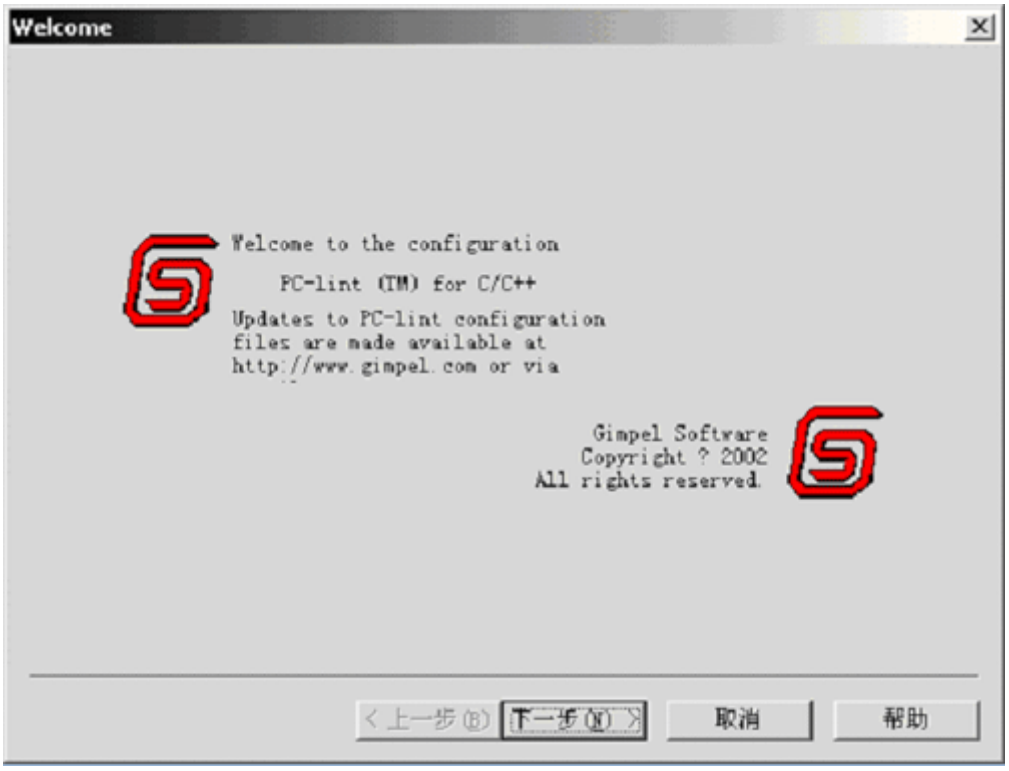

图 4.1 配置欢迎窗口

点击“下一步”按钮出现 pc-lint.exe 命令行使用说明窗口（图 4.2 所示）：

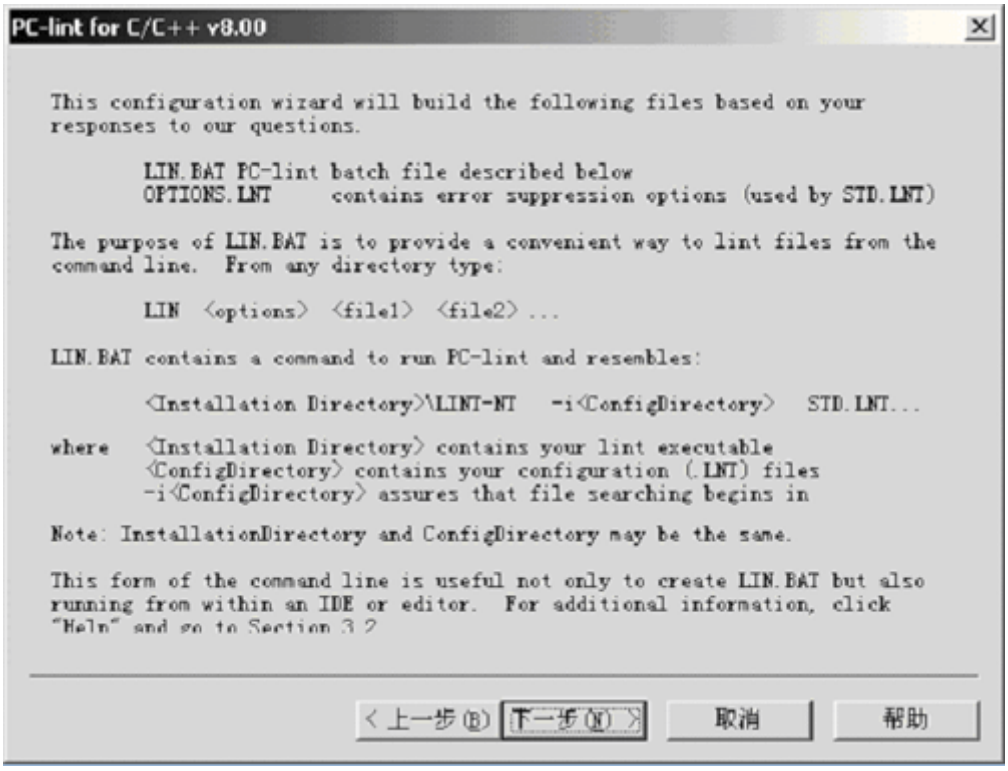

图 4.2 pc-lint.exe 命令行使用说明窗口

点击“下一步”按钮继续，接着是选择创建或修改已有配置文件 STD.LNT 的选项：

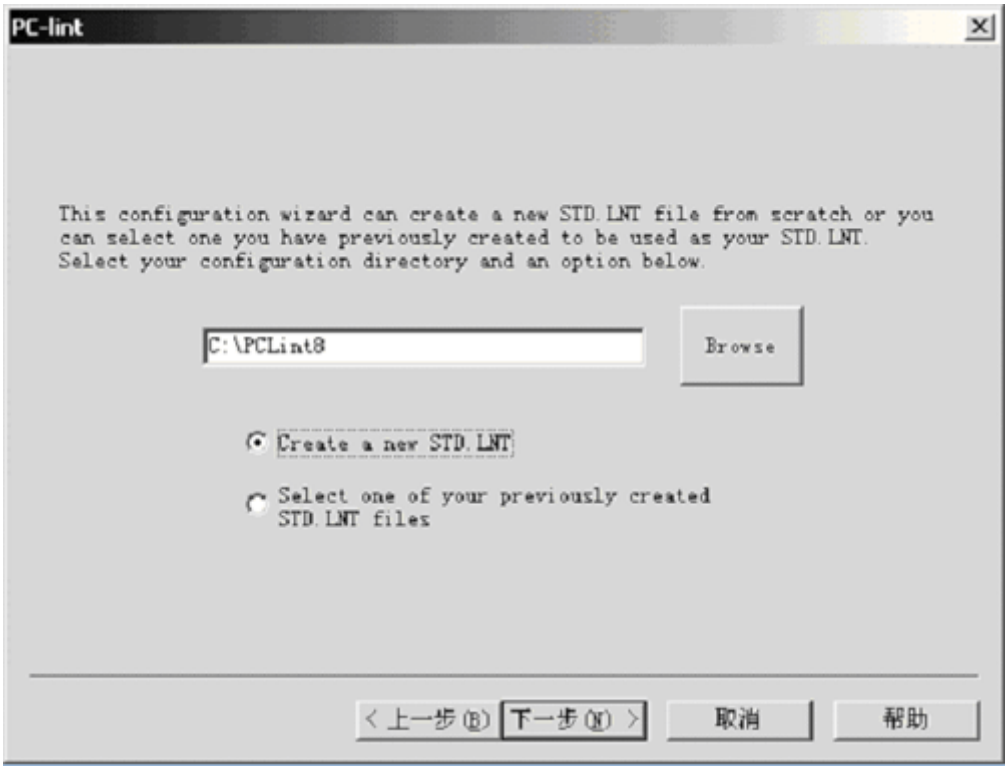

图 4.3 选择如何使用配置文件 STD.LNT

因为我们是第一次配置，所以选择上面一个选项“Create a new STD.LNT”，这样做不会修改已有配置文件 STD.LNT 的内容，而是创建一个新的 STD_x.LNT 文件，文件名中的 x 是从“a”到“z”26 个英文字符中的任意一个，一般是按顺序排列，从“a”开始。STD_x.LNT 文件的内容被初始化为 STD.LNT 内容的拷贝。如图 4.3 所示，使用默认的 PC-Lint 路径，然后点击“下一步”按钮选择编译器：

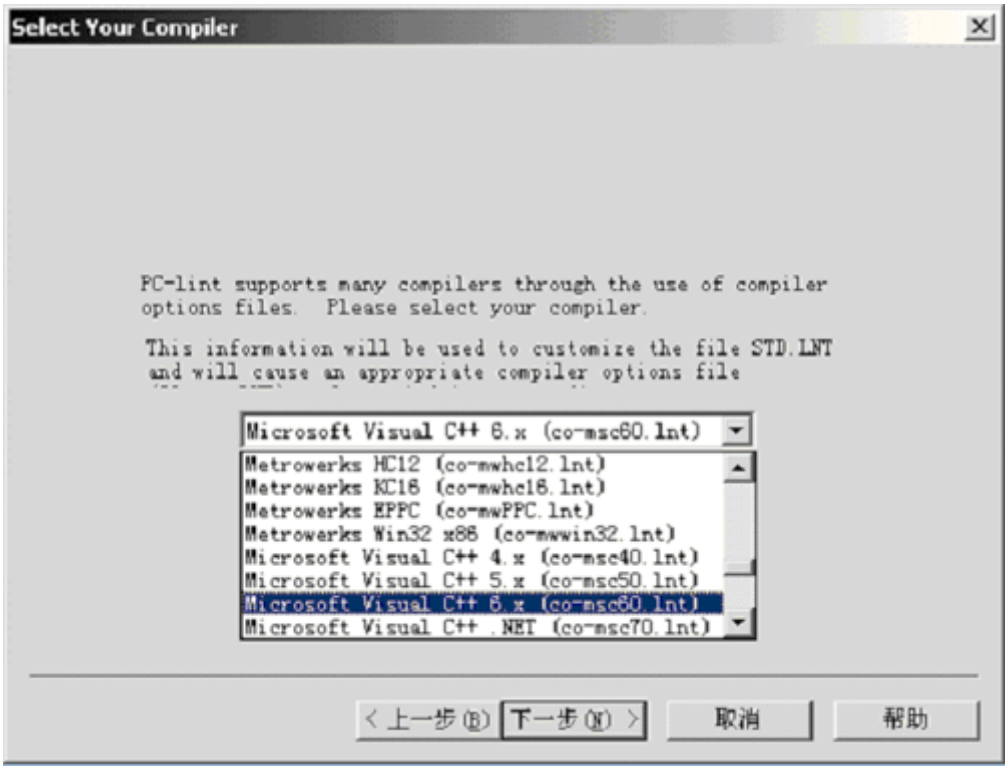

图 4.4 选择编译器

接下来是选择编译器，在下拉框中选择自己使用的编译器。这里我们选择“Microsoft Visual C++ 6.x (co-msc60.lnt)”。如果没有自己使用的编译器，可选择通用编译器“Generic Compilers”。这个选项会体现在 co-xxx.lnt 文件中，并存放在前面我们选择的配置路径（C:\PCLint8）下，在后面配置选项我们所选择的***.LNT 均会被存放到这个路径下。点击“下一步”按钮选择内存模式：

图 4.5 选择内存模式

可以根据自己程序区和数据区的实际大小选择一个恰当的内存模型，内存模型的选项会体现在 STD.LNT 文件或新创建的 STD_x.LNT 中。因为我们的开发环境是 32 位的 Windows，所以选择“32-bit Flat Model”，然后点击“下一步”按钮选择所要的支持库的配置信息：

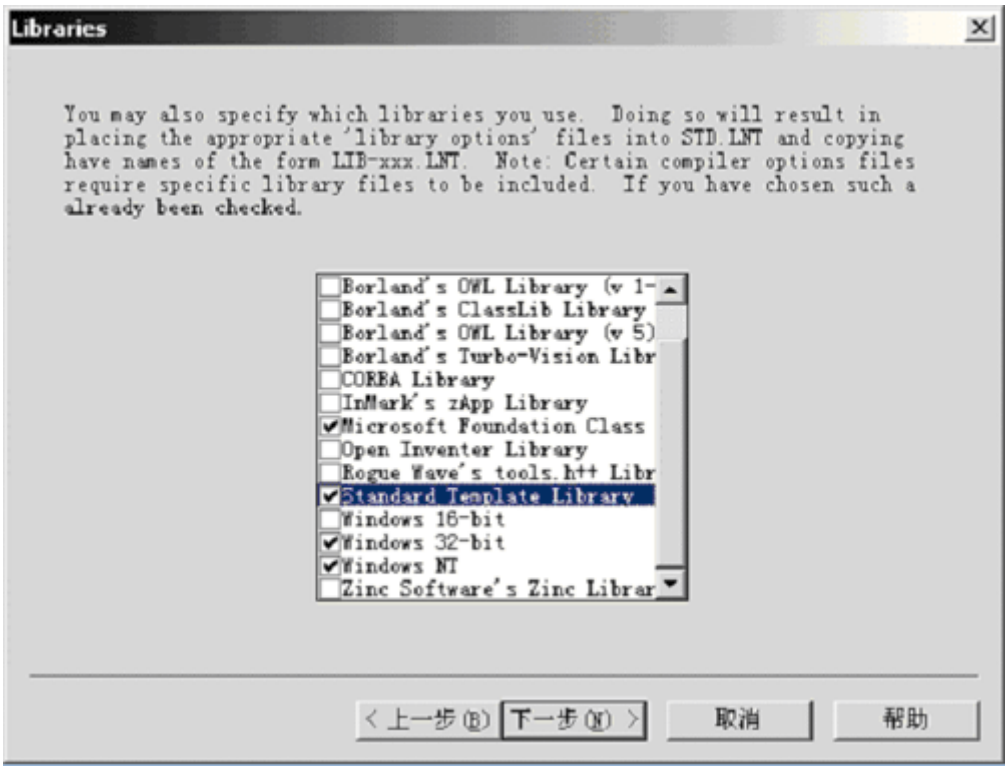

图 4.6 选择软件库的配置信息

PC-Lint 对现在常用的一些软件库都提供了定制的配置信息，选择这些定制信息有助于开发人员将错误或信息的注意力集中在自己的代码中，选择的支持库配置将被引入到 STD.LNT 文件或新创建的 STD_x.LNT 文件中。选择常用的 ATL、MFC、STL 等配置，然后点击“下一步”按钮：

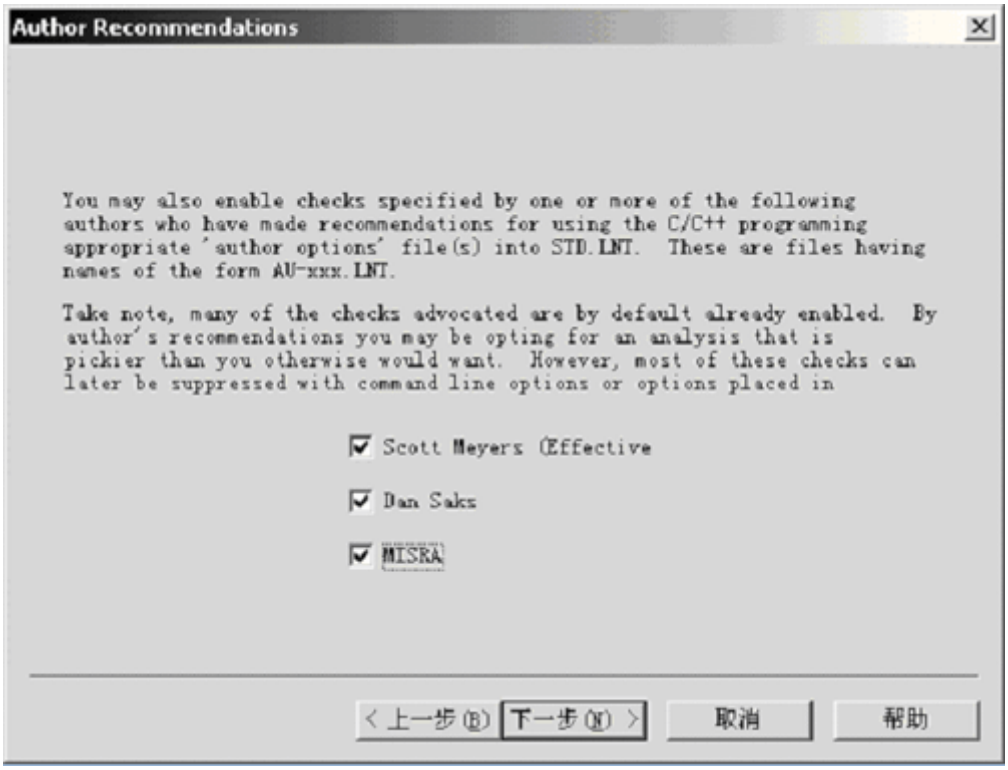

图 4.7 选择软件名人的编程建议

这是一个比较有意思的选项，就是让你选择是否支持为使用 C/C++编程提出过重要建议的作者的一些关于编程方面的个人意见。如果选择某作者的建议，那么他提出的编程建议方面的选项将被打开，作者建议的配置名为 AU-xxx.LNT，建议全部选择，然后点击“下一步”按钮：

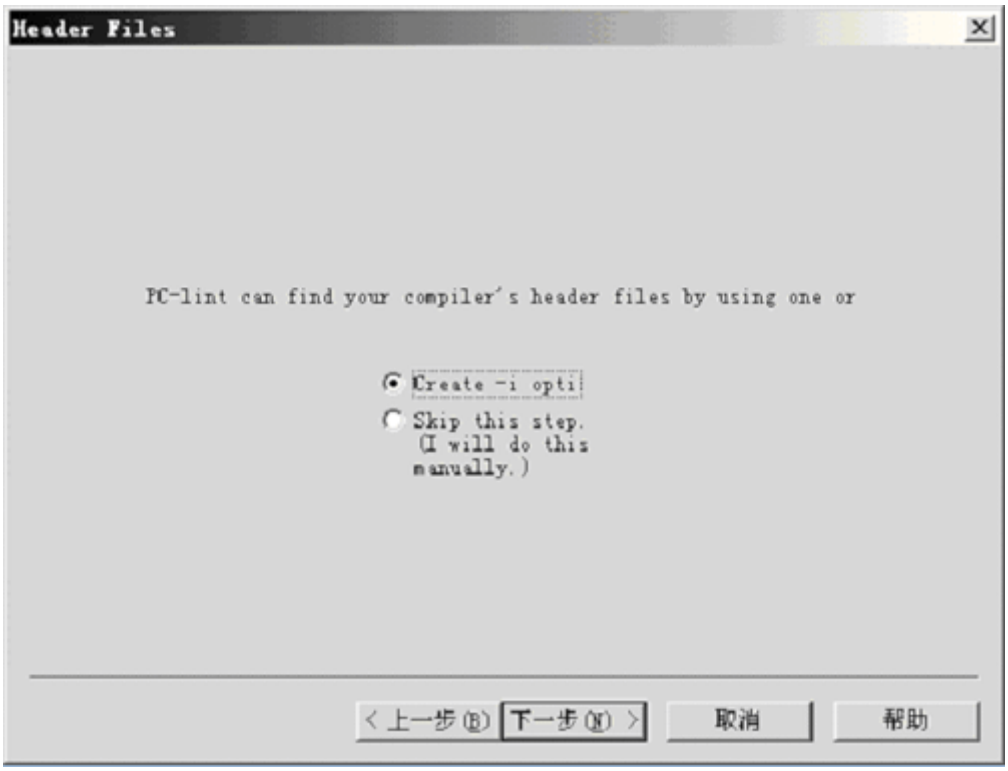

图 4.8 选择是否现在设置包含文件目录

接下来是选择用何种方式设置包含文件目录，如果选择使用-i 方式协助设置包含文件选项，下一步就会要求输入一个或多个包含路径。也可以跳过这一步，以后手工修改配置文件，-i 选项体现在 STD.LNT 文件或新创建的 STD_x.LNT 文件中，每个目录前以-i 引导，目录间以空格分隔，如果目录名中有长文件名或包含空格，使用时要加上双引号，如-i“E:\Program Files\Microsoft Visual C++\VC98\Indlue”。这里我们选择用-i 方式协助我们来设置，然后点击“下一步”按钮：

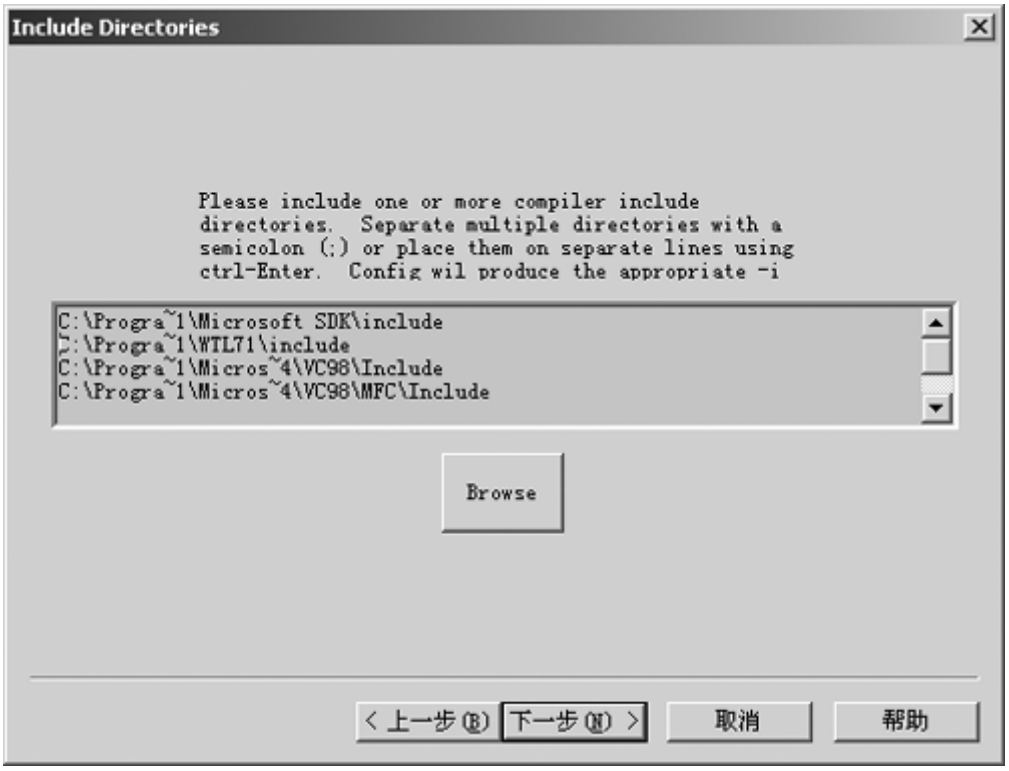

图 4.9 选择是否现在设置包含文件目录

这一步就是在下面的文本框里可手工输入文件包含路径，用分号“;”或用 ctrl+Enter 换行来分割多个包含路径，或者可以点中 Brows，在目录树中直接选择。填完后点击“下一步”按钮：

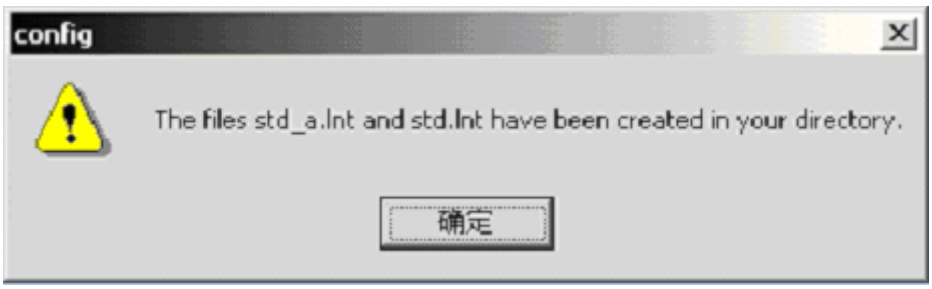

图 4.10 提示 std_x.lnt 已经被创建

因为第三步选择了“Create a new STD.LNT”选项，所以出现以下对话框，表示 std_x.lnt，std.lnt 在配置路径下已被创建，这里的 std_a.lnt 实际上包含了 std.lnt 的信息，除此之外还有我们选择的包含路径和库配置信息。单击“确定”按钮继续：

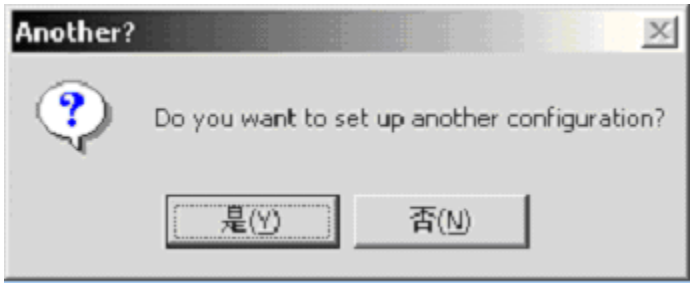

图 4.11 提示是否为其它编译环境创建配置文件

选择“确定”后，会接着提示是否为其它编译环境创建配置文件，如果选择“是”将从第四步开始创建一个新的配置文件。这里我们选择“否”：

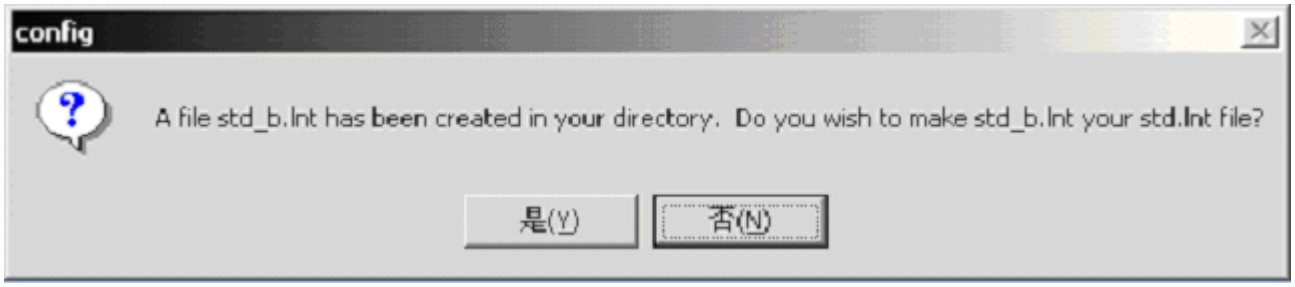

图 4.12 是否替换 std.lnt 文件

接下来会提示是否使用现在生成的 std_x.lnt 文件取代 std.lnt 文件。如果选择“是”将会用 std_x.lnt 文件的内容覆盖 std.lnt 文件的内容，使得当前创建的配置选项成为以后创建新的配置文件时的缺省配置。通常我们选择“否”继续下一步：

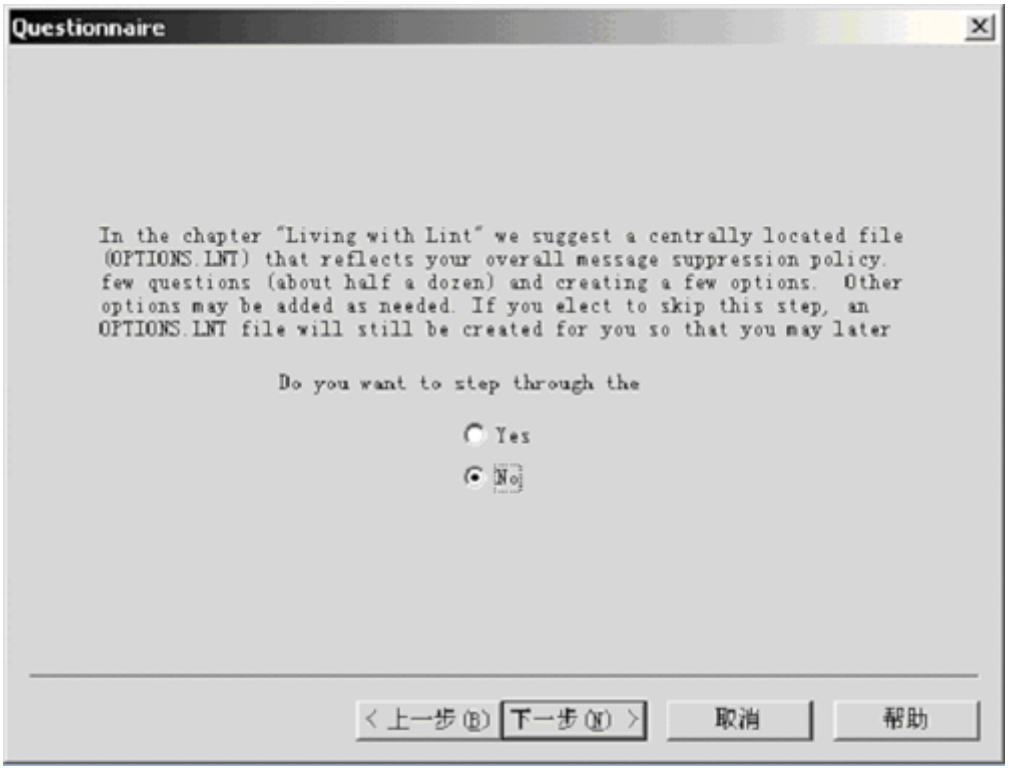

图 4.13 生成全局代码检查选项文件 OPTIONS.LNT

接下来将会准备产生一个控制全局编译信息显示情况的选项文件 OPTIONS.LNT，该文件的产生方式有两种，一种是安装程序对几个核心选项逐一解释并提问你是否取消该选项，如果你选择取消，则会体现在 OPTIONS.LNT 文件中，具体体现方式是在该类信息编码前加-e，后面有一系列逐一选择核心选项的过程。如果选择第二种选择方式，安装文件会先生成一个空的 OPTIONS.LNT 文件，等你以后在实际应用时加入必要的选项。这里选择“No”选项，即不取消这些选项，然后单击“下一步”：

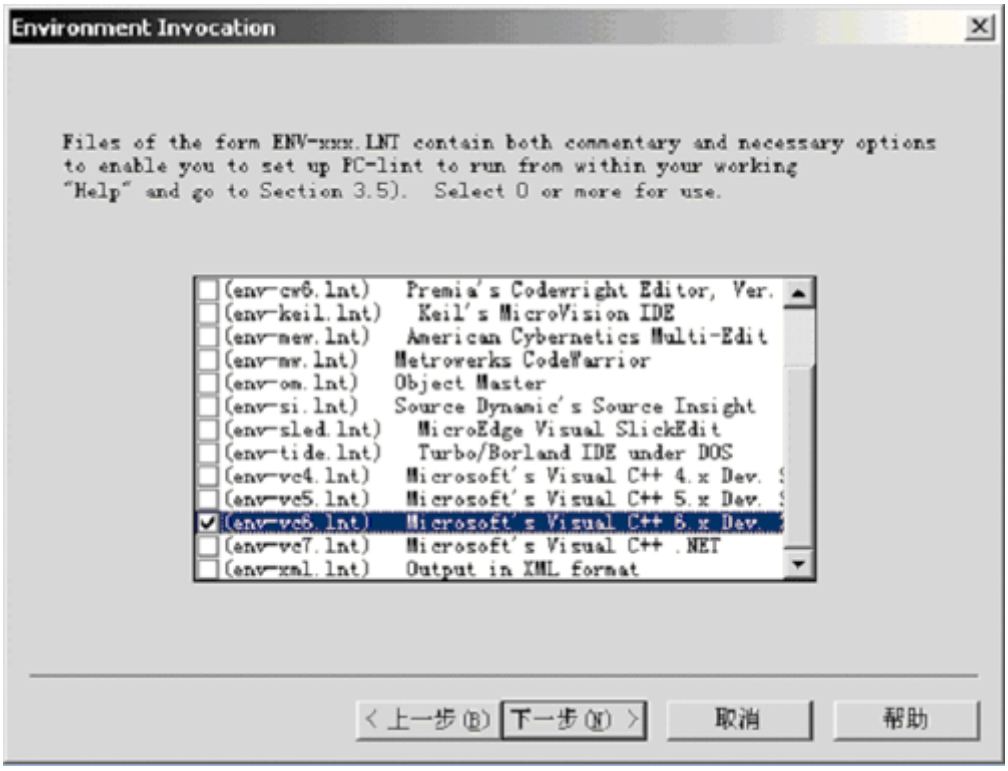

图 4.14 选择所支持的集成开发环境

接着选择所支持的集成开发环境选项，可选多个或一个也不选，PC-Lint 提供了集成在多种开发环境中工作的功能，例如可集成在 VC、BC、Source Insight 中。这里我们选择 Microsift Visual C++ 6.0，这样 env-v6.lnt 就会被拷贝到配置路径中。然后单击“下一步”：

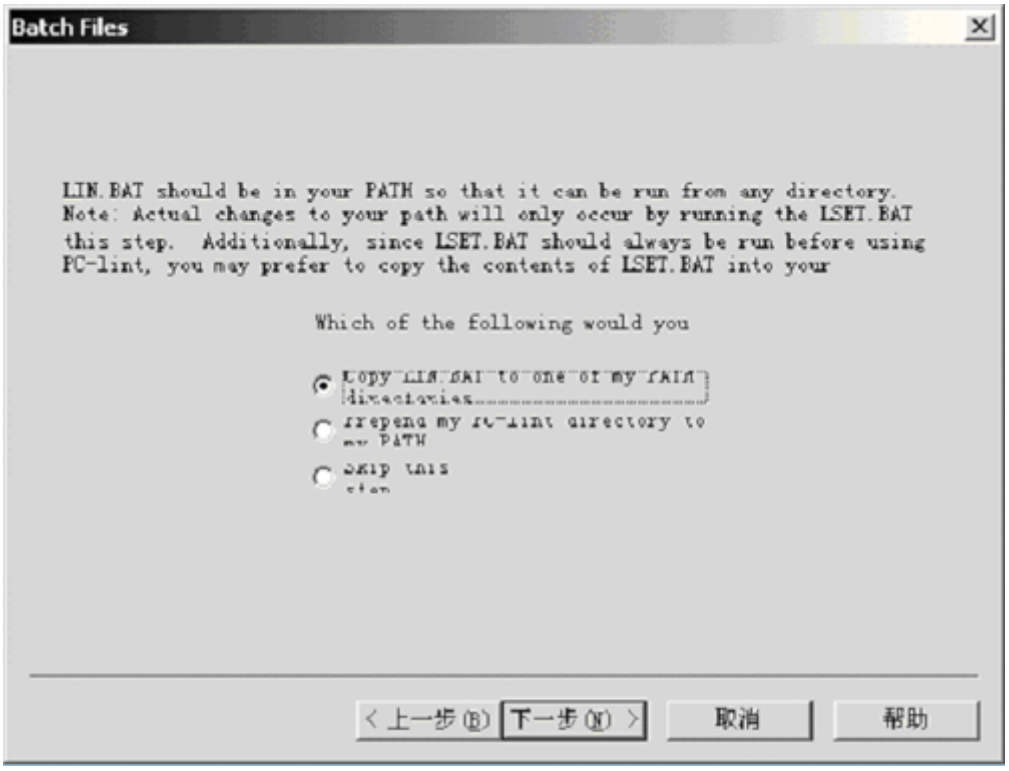

图 4.15 选择 LIN.BAT 文件的使用方式

安装程序会生成一个 LIN.BAT 文件，该文件是运行 PC-Lint 的批处理文件，为了使该文件能在任何路径下运行，安装程序提供了两种方法供你选择。第一种方法是让你选择把 LIN.BAT 拷贝到任何一个 PATH 目录下。第二种方法是生成一个 LSET.BAT 文件，在每次使用 PC-LINT 前先运行它来设置路径，或者把 LSET.BAT 文件的内容拷贝到 AUTOEXEC.BAT 文件中。建议选择第一种方法，指定的目录为当前 PC-Lint 的安装目录。我们选择第一种方式：“copy LIN.BAT to one of my PATH directory”，然后单击“下一步”输入 PATH 目录：

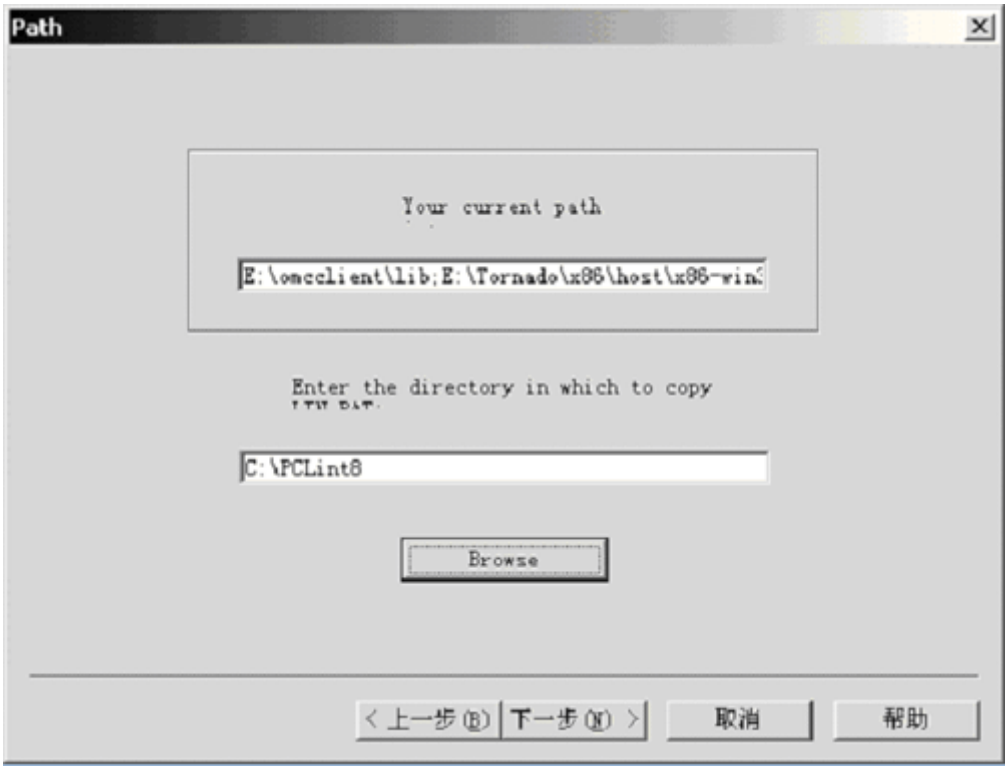

图 4.16 指定 PATH 目录

输入安装目录 C:\PCLint8 作为 PATH 目录，然后单击“下一步”按钮进入最后的确认窗口：

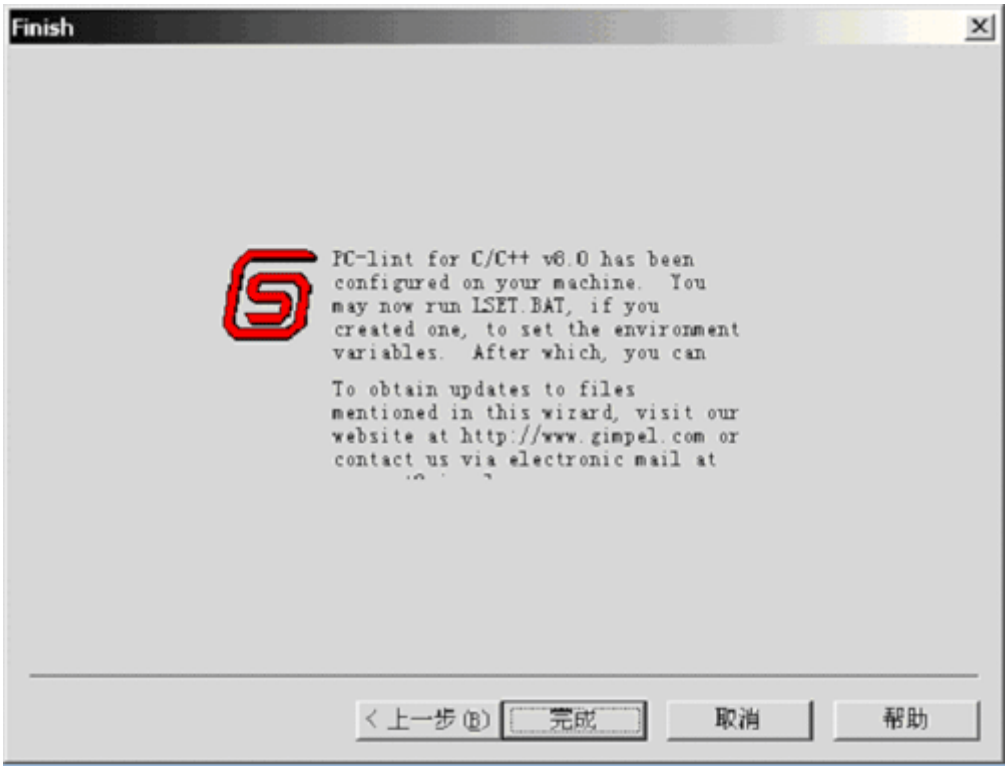

图 4.17 确认完成配置

到此就完成了 PC-Lint 的安装配置工作，单击“完成”按钮就可以使用 PC-Lint 了。以上配置过程中在配置路径下产生的多个*.lnt 文件，除了 std.lnt、std_x.lnt 和 option.lnt 为配置向导所生成，其它 co-xxx.lnt、lib-xxx.lnt、env-xxx.lnt 均是从原始安装目录中拷贝出来的，在这个目录下还有其它 PCLint 所支持的编译器、库及集成开发环境的 lnt 配置文件，所有的 lnt 文件均为文本文件。

上面的配置方法适合于刚开始接触 PC-lint 时使用，对于熟练的使用者可以直接编辑、编写各*.lnt 配置文件安成上面的配置工作，或者定制出更适合自己使用的配置环境。

4.2 PC-Lint 与常用开发工具的集成

PC-Lint 的使用方法很简单，可以用命令行方式进行，也可以集成到开发环境中，下面就分别介绍这些用法

4.2.1 使用命令行方式

命令行的使用方式是 PC-lint 最基本的使用方式，也是其他各种集成使用方式的基础，通过命令行可以完成 PC-lint 的全部代码分析工作。PC-lint 的命令行有下列形式：

Lint-nt option file1 [file1 file3 …]

其中的 Lint-nt 是 PC-lint 在 Windows NT/2000/XP 平台上的可执行程序 Lint-nt.exe，它完成 PC-lint 的基本功能；option 代表 PC-lint 可接受的各种选项，这是 PC-lint 最为复杂的部分，它的选项有 300 多种，可以分为：错误信息禁止选项、变量类型大小选项、冗余信息选项、标志选项、输出格式选项和其他选项等几类，这些选项在本文的第三部分已经介绍过了；file 为待检查的源文件。

另外值得注意的一点是，在命令行中可以加入前面提到的*.lnt 配置文件名，并可以把它看作是命令行的扩展，其中配置的各种选项和文件列表，就和写在命令行中具有一样的效果。

4.2.2 PC-Lint 与 Visual C++集成开发环境（IDE）集成

在所有集成开发环境中，PC-Lint 8.0 对 VC++6 和 VC++7.0 的支持是最完善的，甚至支持直接从 VC 的工程文件（VC6 是*.dsp，VC7 是*.vcproj）导出对应工程的.Lnt 文件，此文件包含了工程设置中的预编译宏，头文件包含路径，源文件名，无需人工编写工程的.Lnt 文件。

PC-Lint 与 VC 集成的方式就是在 VC 的集成开发环境中添加几个定制的命令，添加定制命令的方法是选择“Tools”的“Customize...”命令，在弹出的 Customize 窗口中选择“Tools”标签，在定制工具命令的标签页中添加定制命令。首先要为 VC 的集成开发环境添加一个导出当前工程的.Lnt 配置文件的功能，导出.Lnt 文件的命令行是：

lint-nt.exe +linebuf $(TargetName).dsp>$(TargetName).lnt

参数+linebuf 表示加倍行缓冲的大小，最初是 600 bytes，行缓冲用于存放当

前行和你读到的最长行的信息。$(TargetName)是 VC 集成开发环境的环境变量，表示当前激活的 Project 名字，注意要选中“Use Output Window”选项，这样 PC-Lint 就会将信息输出到 Output 窗口中。填写效果如图 4.18 所示：

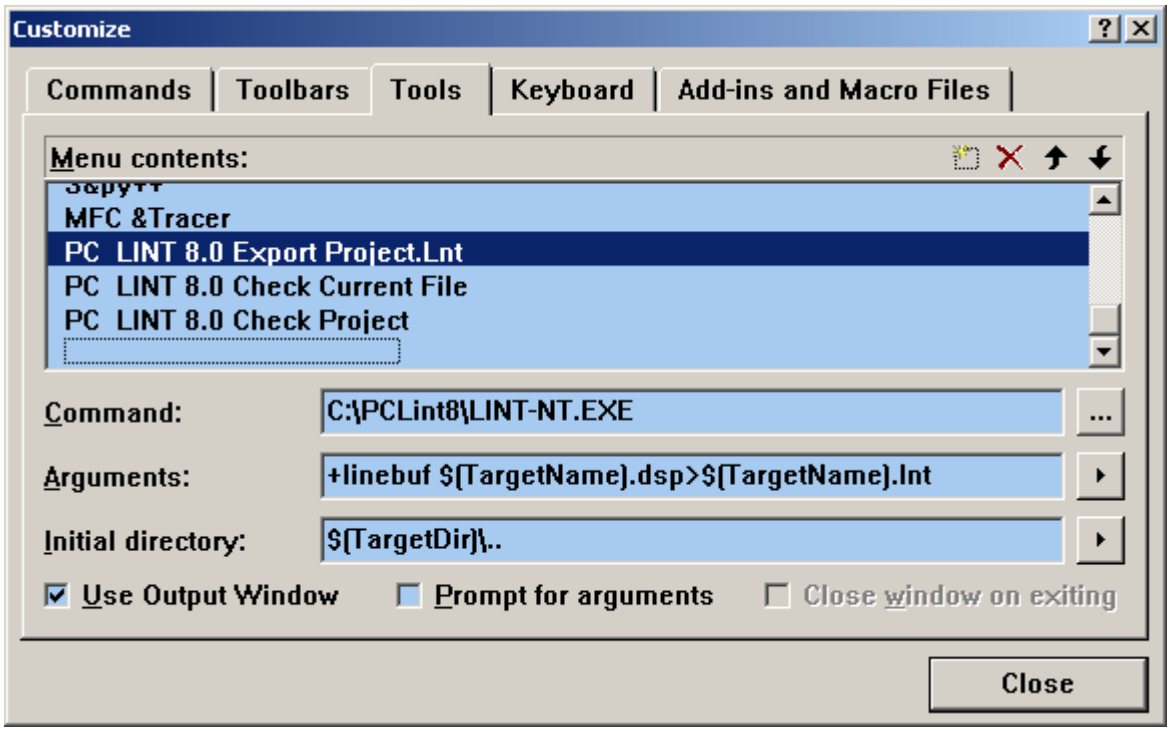

图 4.18 添加导出项目.Lnt 文件的定制命令

接着添加一个检查当前文件的定制命令，检查文件的命令行为：

lint-nt.exe -i"C:\PCLint8" -u std_g.lnt env-vc6.lnt "$(FileName)$(FileExt)"

第一个参数-i"C:\PCLint8"为 PC-Lint 搜索*.lnt 文件的目录，这里就是我们的配置路径。std_g.lnt 是为 VC 编译环境定制的配置文件，$(FileName)和$(FileExt)是 VC 集成开发环境的环境变量，"$(FileName)$(FileExt)"表示当前文件的文件名。和导出.Lnt 命令一样，这个命令也要使用 VC 集成环境的 Output 窗口输出检查信息，所以要选中“Use Output Window”选项，如图 4.19 所示：

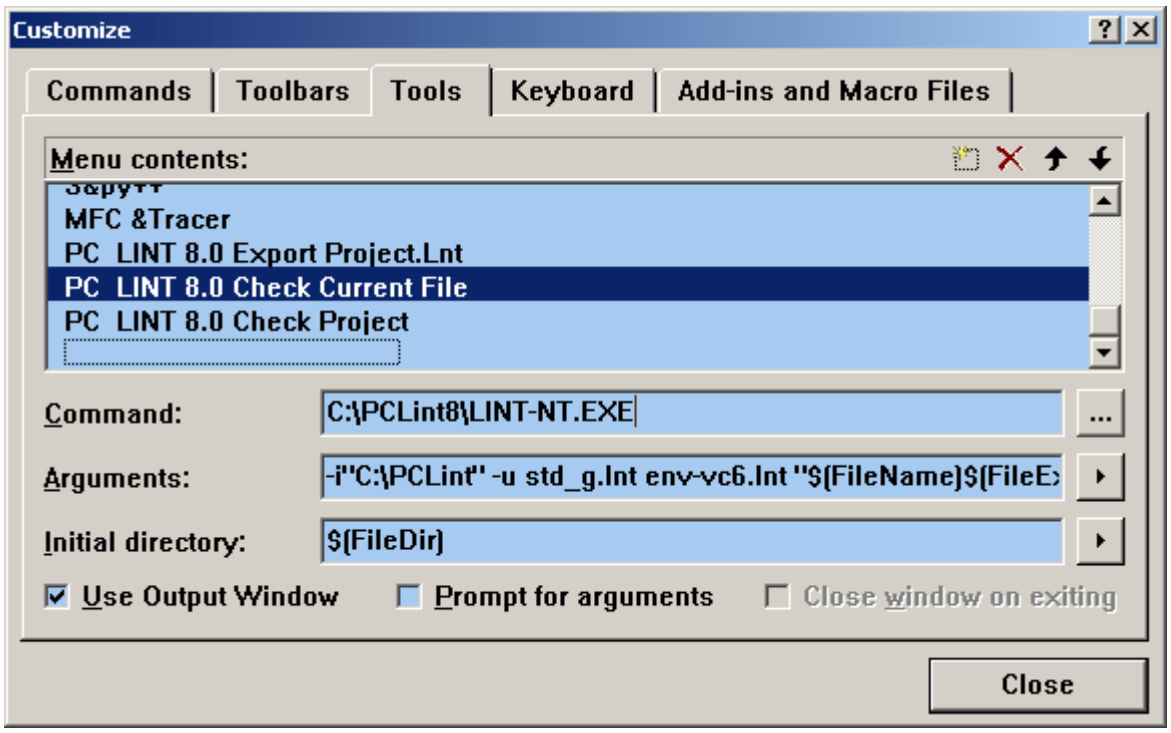

图 4.19 添加检查当前文件文件的定制命令

最后要添加一个检查整个工程的定制命令，检查整个工程的命令行是：

lint-nt.exe +ffn -i"C:\PCLint8" std_g.lnt env-vc6.lnt $(TargetName).lnt>$(TargetName).chk

这个命令的结果就是将整个工程的检查结果输出到与工程同名的.chk 文件中。参数中+ffn 表示 Full File Names，可被用于控制是否使用的完整路径名称表示。

下面就以一个简单的例子介绍一下如何在 VC 集成开发环境中使用 PC-Lint。首先新建一个“Win32 Console Application”类型的简单工程（输出“Hello World”），然后将本文第二章引用的例子代码添加到工程的代码中，最后将这个工程代码所倚赖的包含目录手工添加到配置文件中，因为代码检查要搜索 stdafx.h 这个预编译文件，所以本例要手工添加工程代码所在的目录。本文的例子生成的配置文件是 std_g.lnt，用文本文件打开 std_g.lnt,在文件中添加一行：

-iC:\unzipped\test

“C:\unzipped\test”就是例子工程所在的目录（stdafx.h 就在这个目录）。如果你的工程比较庞大，有很多头文件包含目录，就需要将这些目录一一添加到配置文件。在确保代码输入没有错误之后（有错误页没关系，PC-Lint 会检查出错误），就可以开始代码检查了。例子工程，打开要检查的代码文件，本例是 test.cpp，然后选择“Tools”菜单下的“PC_LINT 8.0 Check Current File”命令，Output 窗口输出对本文件的检查结果，如图 4.20 所示：

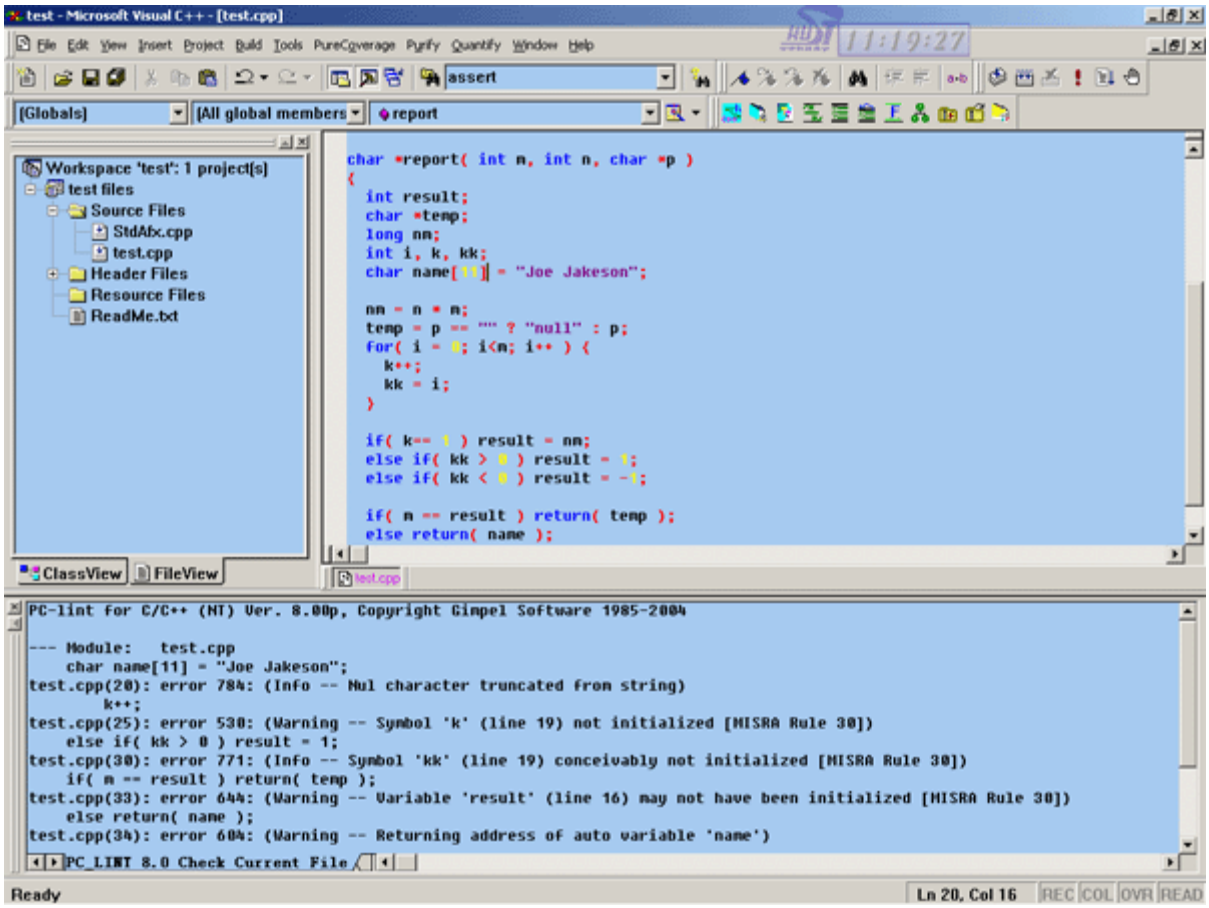

图 4.20 检查结果

4.2.3 PC-Lint 与 source insight 集成

PC-Lint 与 source insight 的集成也是通过添加定制命令实现的，从“Options”菜单中选择“Custom Commands”命令项。点击“Add...”按钮，如图 4.21 所示，在弹出的“Custom Commands”窗口中完成以下输入：
在 Name 栏中输入“PC-lint Check Current File”，原则上这个名称可以随便起，只要你能搞清楚它的含义就可以了；
在 Run 栏中输入“C:\PcLint\lint-nt -u -iC:\PcLint\Lint std_f env-si %f”其中 C:\PcLint 是你 PC-LINT 的安装目录，std_f 表示为 Source Insight 定制的配置文件 std_f.lnt；
在 Output 栏中选择“Iconic Window”、“Capture Output”选项；
在 Control 栏中选择“Save Files First”；
在 Source Links in Output 栏中选择“Parse Links in Output”、“File，then Line”；
在 Pattern 栏中输入“^\([^]*\) \([0-9]+\)”；

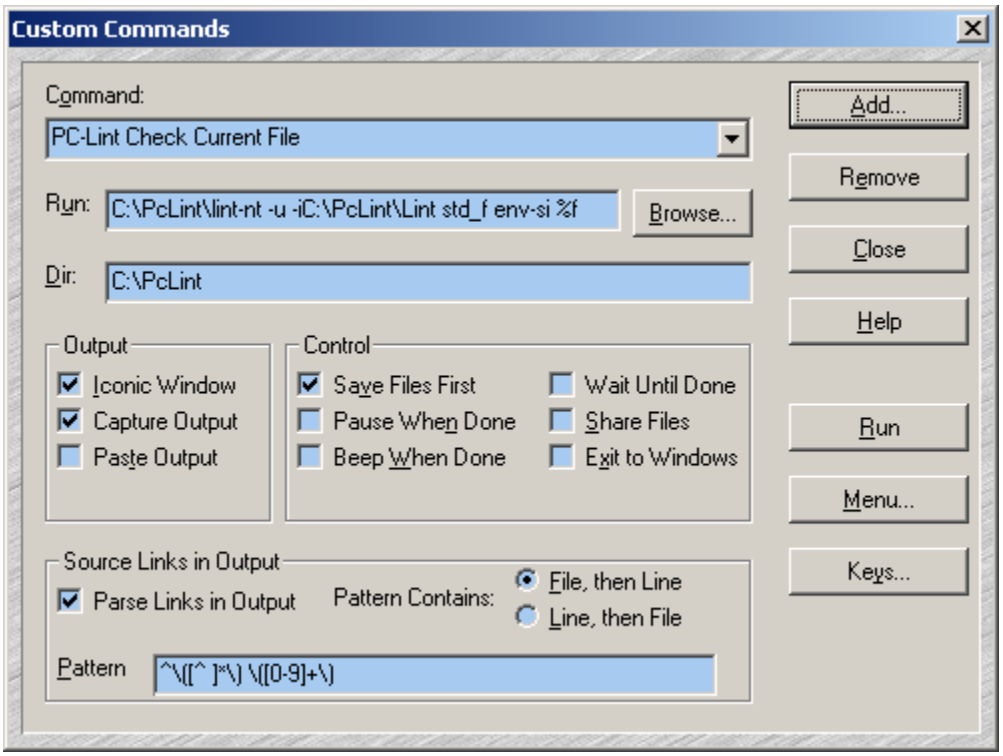

图 4.21 在 Source Insight 中添加定制命令

命令添加完成后就可以点击“Run”按钮就可以对当前文件执行 PC-Lint 检查。为了方便使用，还可以点击“Menu...”按钮将这个定制命令添加到 Source Insight 的菜单中。

4.2.4 PC-Lint 与 UltraEdit 集成

在 UltraEdit 中集成 PC-Lint 的方法和 Source Insight 类似，也是添加一个定制命令菜单，具体实现方法是先单击 UltraEdit 的“高级”菜单中的“工具配置”命令，如图 4.22 所示，在打开的配置窗口中依次输入以下内容：
在“菜单项目名”栏输入“PC-lint Check Current File”；
在“命令行”栏输入以下命令：C:\PCLint\lint-nt –u -iC:\PCLint std env-si %f 其中，C:\PCLint 是 PC-Lint 的安装目录，使用 std.lnt 中的配置，由于 UltraEdit 和 Source Insightde 的检查环境类似，所以借用 env-si 中的环境配置；
在“工作目录”栏输入以下路径：E:\code，这是代码所在目录；
选中“先保存所有文件”选项；
在“命令输出”栏中，选中“输出到列表”和“捕捉输出”两个选项；
点“插入”将命令行插入 UltraEdit 的菜单中；

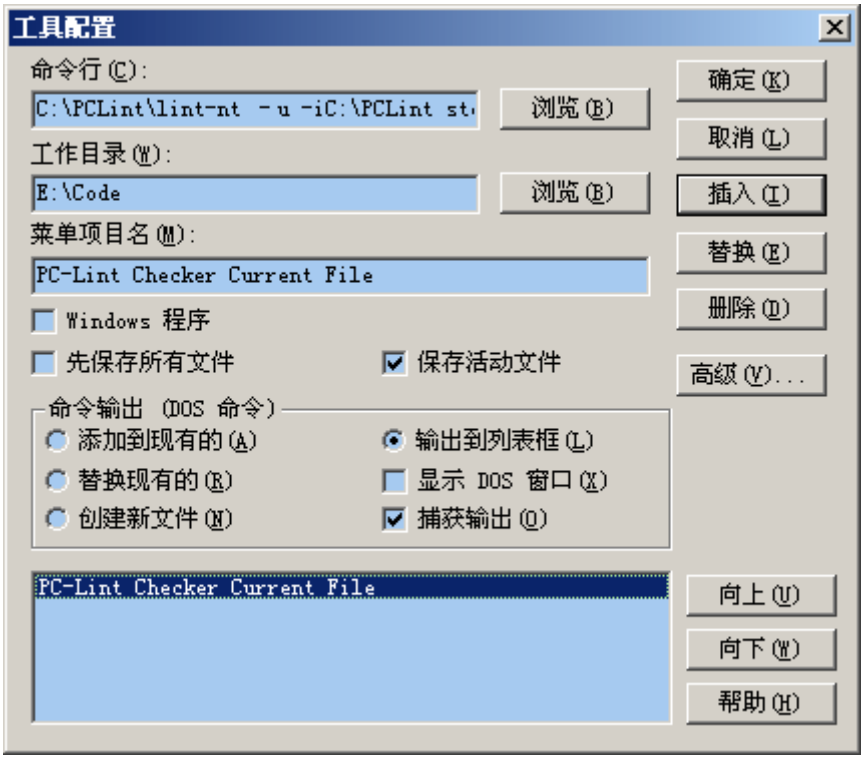

图 4.22 在 UltraEdit 中添加定制命令

此时在 UltraEdit 的“高级”菜单中会增加一个“PC-lint Check Current File”菜单，点击该菜单即可对当前文件执行 PC-lint 检查。

五 总结

软件除错是软件项目开发成本和延误的主要因素，PC-lint 能够帮你在程序动态测试之前发现编码错误，降低软件消除错误的成本。使用 PC-Lint 在代码走读和单元测试之前进行检查，可以提前发现程序隐藏错误，提高代码质量，节省测试时间。另外，使用 PC-lint 的编码规则检查，可以有效地规范软件人员的编码行为。如果能够在软件开发过程中有效地使用 PC-lint 代码检查工具，将大大地提高代码质量，降低软件成本。

参考文献

[1] Gimpel Software. Reference Manual for PC-lint/FlexeLint. July,2001
[2] PC-Lint 选项详解

附录一 PC-Lint 重要文件说明

Msg.txt ： 解释告警的内容。

options.lnt ： 反映全局编译信息显示情况的选项文件，通常需要添加自定选项以使代码检查更为严格。

env-xx.lnt ：讲述如何将 PC-lint 与对应的编辑环境结合起来，xx 是 si 表示是为 Source Insight 配置的检查环境，xx 是 vc6 则表示是为 Visual C++ 6.0 准备的检查环境。

co-xxx.lnt ：选定的编译器与库选项。

std.lnt ：标准配置文件，包含内存模型等全局性东西。

lib-xxx.lnt ：库类型的列表，包括标准 C/C++库，MFC 库，OWL 库等等。

au-xxx.LNT ：C++编程提出过重要建议的作者，选择某作者后，他提出的编程建议方面的选项将被打开。

附录 二　错误信息禁止选项说明

命令格式	说明	代码中的举例
-e#	隐藏某类错误	/*lint -e725 */
-e(#)	隐藏下一表达式中的某类错误	/*lint –e(534) */ printf (“it’s all”);
!e#	隐藏本行中的错误	/*lint !e534*/ printf(“it’s all”);
-esym(#, Symbol)	隐藏有关某符号的错误	/*lint –esym(534, printf)*/ printf (“it’s all”);
-elib(#)	隐藏头文件中的某类错误	/*lint –elib(129) */ #include “r01.h”
-efunc(#, <func>)	隐藏某个函数中的特定错误	/*lint –efunc(534, mchRelAll)*/ unsigned int mchRelAll(mchHoData *pHoData) { printf (“it’s all”); }

附录 三　PC-Lint 检测中的常见错误

错误编码	错误说明	举例
40	变量未声明	

506 固定的 Boolean 值 char c=3;
if(c<300){}
525 缩排格式错误
527 无法执行到的语句 if(a > B)
return TRUE;
else
return FALSE;
return FALSE;
529 变量未引用 检查变量未引用的原因
530 使用未初始化的变量
534 忽略函数返回值
539 缩排格式错误
545 对数组变量使用& char arr[100], *p;
p=&arr;
603 指针未初始化 void print_str(const char *p);
...
char *sz;
print_str(sz);
605 指针能力增强 void write_str(char *lpsz);
...
write_str("string");
613 可能使用了空指针
616 在 switch 语句中未使用 break;
650 比较数值时，常量的范围超过了 if(ch == 0xFF) ...
变量范围
713 把有符号型数值赋给了无符号型
数值
715 变量未引用
725 Indentation 错误
734 在赋值时发生变量越界 int a, b, c;
...
c=a*b;
737 无符号型变/常量和有变量型
变/常量存在于同一个表达式中。
744 在 switch 语句中没有 default
752 本地声明的函数未被使用
762 函数重复声明
774 Boolean 表达式始终返回真/假 char c;
if(c < 300)