

C++程序设计（2）

典型程序设计实例

正整数分解质因数

- 将一个正整数分解质因数。
 - 例如：输入90，打印出 $90=2*3*3*5$

```
void divideFactor(int n){
    for (int i = 2; i <= n; i++) { //iterate over all integers
        while (n != i) {
            if (n%i == 0){
                cout<<i<<"*";
                n = n/i;
            }
            else //If n cannot be divided by i
                break;
        }
    }
    cout<<n<<endl;
}
```

辗转相除法求最大公约数

- 辗转相除法基于如下原理：两个整数的最大公约数等于其中较小的数和两数的差的最大公约数。

```
void gcd(int a, int b){  
    int remainder;  
    while (b != 0){  
        remainder = a%b;           //get the remainder  
        a = b;                     //swap the integers  
        b = remainder;  
    }  
    cout << a << endl;  
}
```

冒泡排序

- 排序过程中总是小数往前放，大数往后放，相当于气泡往上升，所以称作冒泡排序

```
void bubbleSort (int *a, int count){
    int temp;
    for (int i = 1; i < count; i++){
        for(int j = count - 1; j >= i; j--){
            //The smallest will be bubbled to first
            if (a[j] < a[j-1]){
                //If in wrong order, swap them
                temp = a[j-1];
                a[j-1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

基本错误

语法错误

- 1、变量没有区分大小写
例如：变量X 与x 不同；
- 2、关键字写错
例如：void 写成了viod等，cout写成count等
- 3、把英文符号写成了中文符号
例如：常见的错误包括单引号、双引号、分号等
- 4、混淆了“=”与“==”的区别
- 5、复合语句没有使用{ }
- 6、把“x>5 && x<9”写成“5<x<9”

编译错误（1）

- **关键字： `newline in constant`**
- 直译：在常量中出现了换行。
- 错误分析：
 1. 字符串常量、字符常量中是否有换行。
 2. 在这句语句中，某个字符串常量的尾部是否漏掉了双引号。
 3. 在这语句中，某个字符创常量中是否出现了双引号字符“`"`”，但是没有使用转义符“`\`”。
 4. 在这句语句中，某个字符常量的尾部是否漏掉了单引号。
 5. 是否在某句语句的尾部，或语句的中间误输入了一个单引号或双引号。

编译错误（2）

- **关键字：too many characters in constant**
- 直译：字符常量中的字符太多了。
- 错误分析：
 1. 单引号表示字符型常量。一般的，单引号中必须有，也只能有一个字符（使用转义符时，转义符所表示的字符当作一个字符看待），如果单引号中的字符数多于4个，就会引发这个错误。
 2. 如果语句中某个字符常量缺少右边的单引号，也会引发这个错误，例如：`if (x == 'x || x == 'y') { ... }`。
 3. 值得注意的是，如果单引号中的字符数是2-4个，编译不报错，输出结果是这几个字母的ASC码作为一个整数（int，4B）整体看待的数字。
 4. 两个单引号之间不加任何内容会引发如下错误: empty character constant。

编译错误（3）

- **关键字：** **unknown character '0x##'**
- 直译：未知字符 ‘0x##’。
- 错误分析：
 - 0x##是字符ASC码的16进制表示法。这里说的未知字符，通常是指全角符号、字母、数字，或者直接输入了汉字。如果全角字符和汉字用双引号包含起来，则成为字符串常量的一部分，是不会引发这个错误的。

编译错误（4）

- **关键字：illegal digit '#' for base '8'**
- 直译：在八进制中出现了非法的数字‘#’（这个数字#通常是8或者9）。
- 错误分析：
 - 如果某个数字常量以“0”开头（单纯的数字0除外），那么编译器会认为这是一个8进制数字。例如：“089”、“078”、“093”都是非法的，而“071”是合法的，等同于十进制中的“57”。

编译错误（5）

- **关键字： 'xxxx' : undeclared identifier**
- 直译：标识符“xxxx”未定义。
- 标识符：
 - 标识符是程序中出现的除关键字之外的词，通常由字母、数字和下划线组成，不能以数字开头，不能与关键字重复，并且区分大小写。变量名、函数名、类名、常量名等等，都是标识符。
 - 所有的标识符都必须先定义，后使用。
- 错误原因：
 1. 如果“xxxx”是一个变量名，那么通常是程序员忘记了定义这个变量，或者拼写错误、大小写错误所引起的，所以，首先检查变量名是否正确。（关联：变量，变量定义）
 2. 如果“xxxx”是一个函数名，那就怀疑函数名是否没有定义。另外一种情况，没有在调用之前对函数原形进行申明

编译错误（5）

3. 如果“xxxx”是一个库函数的函数名，比如“sqrt”、“fabs”，那么看看你在cpp文件已开始是否包含了这些库函数所在的头文件（.h文件）。
4. 如果“xxxx”是一个类名，那么表示这个类没有定义，可能性依然是：根本没有定义这个类，或者拼写错误，或者大小写错误，或者缺少头文件，或者类的使用在申明之前。
5. C++的作用域也会成为引发这个错误的陷阱。在花括号之内变量，是不能在这个花括号之外使用的。类、函数、if、do(while)、for所引起的花括号都遵循这个规则。
6. 前面某句语句的错误也可能导致编译器误认为这一句有错。如果你前面的变量定义语句有错误，编译器在后面的编译中会认为该变量从来没有定义过，以致后面所有使用这个变量的语句都报这个错误。如果函数申明语句有错误，那么将会引发同样的问题。

编译错误（6）

- **关键字： 'xxxx' : redefinition**
- 直译：“xxxx”重复申明。
- 错误原因：
 - 变量“xxxx”在同一作用域中定义了多次。检查“xxxx”的每一次定义，只保留一个，或者更改变量名。

编译错误（7）

- **关键字： syntax error : missing ';' before (identifier) 'xxxx'**

- 直译：在（标志符）“xxxx”前缺少分号。

- 错误原因：

这是VC6的编译期最常见的误报，当出现这个错误时，往往所指的语句并没有错误，而是它的上一句语句发生了错误。

1. 上一句语句的末尾真的缺少分号。那么补上就可以了。
2. 上一句语句不完整，或者有明显的语法错误，或者根本不能算上一句语句（有时候是无意中按到键盘所致）。
3. 如果发现发生错误的语句是cpp文件的第一行语句，在本文件中检查没有错误，而且这个文件使用双引号包含了某个头文件，那么检查这个头文件，在这个头文件的尾部可能有错误。

链接错误（1）

- **关键字：** **unresolved external symbol _main**
- 直译：未解决的外部符号：_main。
- 错误原因：
 - 缺少main函数。看看main的拼写或大小写是否正确。

链接错误（2）

- **关键字： `_main already defined in xxxx.obj`**
- 直译： `_main`已经存在于xxxx.obj中了。
- 错误原因：
 - 直接的原因是该程序中有多个（不止一个） `main`函数。这个错误通常不是你在同一个文件中包含有两个 `main`函数，而是在一个 `project`（项目）中包含了多个 `cpp`文件，而每个 `cpp`文件中都有一个 `main`函数。
 - `Workspace`和 `Project`的关系
 - 每一个程序都是一个 `Project`（项目），一个 `Project`可以编译为一个应用程序（*.exe），或者一个动态链接库（*.dll）。
 - 每个 `Project`下面可以包含多个 `.cpp`文件，`.h`文件，以及其他资源文件。在这些文件中，只能有一个 `main`函数。
 - `Workspace`（工作区）是 `Project`的集合

一些错误编程观念

程序编出来，运行正确就行了

- 运行正确的程序并不一定是好程序，程序员时刻要牢记的一条就是自己写的程序不仅是给自己看的，要让别人也能轻易地看懂。很遗憾，许多的编程新手不能清晰地驾驭软件的结构，对头文件和实现文件的概念含糊不清，写出来的程序可读性很差。

模块化程序设计

- C程序采用模块化的编程思想，需合理地将一个很大的软件划分为一系列功能独立的部分合作完成系统的需求，在模块的划分上主要依据功能。模块由头文件和实现文件组成，对头文件和实现文件的正确使用方法是：
 - 规则1 头文件(.h)中是对于该模块接口的声明，接口包括该模块提供给其它模块调用的外部函数及外部全局变量，对这些变量和函数都需在.h文件中冠以extern关键字声明；
 - 规则2 模块内的函数和全局变量需在.c文件开头冠以static关键字声明；
 - 规则3 不要在.h文件中定义变量；

数组和指针（1）

- 很多人认为数组名就是指针，而实际上数组名和指针有很大区别，在使用时要进行正确区分。

1. 数组名指代一种数据结构，这种数据结构就是数组；

例如：

```
char str[10];
```

```
char *pStr = str;
```

```
cout << sizeof(str) << endl;
```

```
cout << sizeof(pStr) << endl;
```

输出结果为：

10

4

这说明数组名str指代数据结构char[10]。

数组和指针（2）

2. 数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；

```
char str[10];
```

```
char *pStr = str;
```

```
str++; //编译出错，提示str不是左值
```

```
pStr++; //编译正确
```

3. 指向数组的指针则是另外一种变量类型（在WIN32平台下，长度为4），仅仅意味着数组的存放地址；

数组和指针（3）

4. 数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；很遗憾，在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

```
void arrayTest(char str[])
{
    cout << sizeof(str) << endl;           //输出指针长度
    str++; //编译正确
}
int main(int argc, char* argv[])
{
    char str1[10] = "I Love U";
    arrayTest(str1);
    return 0;
}
```

整形变量长度（1）

- 整形变量是不是32位这个问题不仅与具体的CPU架构有关，而且与编译器有关。在嵌入式系统的编程中，一般整数的位数等于CPU字长，常用的嵌入式CPU芯片的字长为8、16、32，因而整形变量的长度可能是8、16、32。在未来64位平台下，整形变量的长度可达到64位。
- 长整形变量的长度一般为CPU字长的2倍。

整形变量长度（2）

- 比较

- ```
typedef struct tagTypeExample
{
 unsigned short x;
 unsigned int y;
}TypeExample;
```
- ```
# define unsigned short UINT16 //16位无符号整数
#define unsigned int UINT32 //32位无符号整数
typedef struct tagTypeExample
{
    UINT16 x;
    UINT32 y;
}TypeExample;
```

malloc的用法（1）

- 不要为了用malloc而用malloc，malloc不是目的，而是手段

```
/* xx.c: xx模块实现文件 */
int *pInt;
/* xx模块的初始化函数 */
xx_intial()
{
    pInt = ( int * ) malloc
( sizeof( int ) );
    ...
}
/* xx模块的其他函数（仅为举例*/
xx_otherFunction()
{
    *Int = 10;
    ...
}
```

- 修改，不使用malloc

```
/* xx.c:xx模块实现文件 */
int example;
/* xx模块的初始化函数 */
xx_intial()
{
    ...
}
/* xx模块的其他函数：仅为举例
*/
xx_otherFunction()
{
    example = 10;
    ...
}
```

上述程序完全不具备动态申请内存的性质

malloc的用法（2）

1. 需要采用malloc的情形

- 不知道有多少数据要来，来了的又走了。譬如正在处理一个报文队列，收到的报文都存入该队列，处理完队列头的报文后你需要取出队列头的元素，不知道有多少报文来（因而你不知道应该用多大的报文数组），这些来的报文处理完后都要走（释放），这种情况适合用malloc和free。
- 数组慢慢的长大：文本编辑程序
是不是应该这样定义数据结构并在用户每输入一个字符的情况下malloc一个CharQueue空间呢？

```
typedef struct tagCharQueue
{
    char ch;
    struct tagCharQueue *next;
}CharQueue;
```

malloc的用法（3）

- 上述做法不好，这将使每个字符占据“1+指针长度”的开销。
- 正确的做法：

```
typedef struct tagCharQueue
{
    char str[100];
    struct tagCharQueue *next;
}CharQueue;
```

让字符以100为单位慢慢地走，当输入字符数达到100的整数倍时，申请一片CharQueue空间

malloc的用法（4）

- 2. malloc与free要成对出现
- 3. free后一定要置指针为NULL，防止其成为“野”指针

malloc的用法（5）

- `char * func(void)`
- `{`
- `char *p;`
- `p = (char *)malloc(...);`
- `if(p!=NULL)`
- `...; /* 一系列针对p的操作 */`
- `return p;`
- `}`
- `/*在某处调用func(), 用完func中动态申请的内存后将其free*/`
- `char *q = func();`
- `...`
- `free(q);`
- `/* 在调用处申请内存，并传入func函数 */`
- `char *p=malloc(...);`
- `if(p!=NULL)`
- `{`
- `func(p);`
- `...`
- `free(p);`
- `p=NULL;`
- `}`
- `/* 函数func则接收参数p */`
- `void func(char *p)`
- `{`
- `... /* 一系列针对p的操作 */`
- `}`

值传递改变函数值？

- 数组越界

- 值传递的确改变参数的内容？

```
int n = 9;
```

```
char a[10];
```

```
example ( n, a ); //调用函数example(int n, char *pStr)
```

```
printf ( “%d” , n ); //输出结果不是9
```

- 上面的例子其实不是什么编译器出错，而是在函数 **example** 内对字符串 **a** 的访问越界！当在函数 **example** 内对 **a** 的访问越界后，再进行写操作时，就有可能操作到了 **n** 所在的内存空间，于是改变了 **n** 的值。

数组越界例子 (1)

```
void example1()
{
    char string[10];
    char* str1 = "0123456789";
    strcpy ( string, str1 );
}

void example 2(char* str1)
{
    char string[10];
    if( strlen( str1 ) <= 10 )
    {
        strcpy( string, str1 );
    }
}
```


数组越界例子（2）

```
void example3()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

Strcpy实现的比较

- `void strcpy(char *strDest, char *strSrc)`
- `char * strcpy(char *strDest, const char *strSrc)`
- {
 - (1) 程序要强大：为了实现链式操作，将目的地址返回，函数返回类型改为`char *`
 - (2) 程序要可读：源字符串指针参数加`const`限制，表明为输入参数
 - (3) 程序要健壮：验证`strDest`和`strSrc`非空
- `while (*strDest++ != '\0');`
- `while (*strSrc++ != '\0');`
- `return address;`
- }

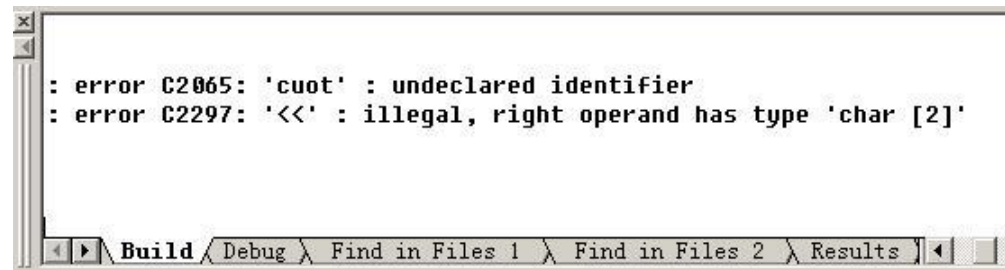
VC++的基本调试方法

Visual C++的基本调试方法

- 在开发程序的过程中，经常需要查找程序中的错误。这就需要利用调试工具来帮助读者进行程序的调试。当然目前有许多调试工具，而集成在Visual C++中的调试工具以其强大的功能，深受用户欢迎。

错误类型

- 一个程序编写完成后往往会存在错误。有些错误在编译连接阶段可以由编译系统发现并指出，称为语法错误。一般来说，当用户对应用程序进行编译（单击菜单【Build】|【Compile】项或按下快捷键【Ctrl+F7】）后产生的错误就是语法错误，常见的语法错误有拼写错误和参数设置错误。例如，当读者将C++的输出函数cout函数写成cuot后，编译系统给出如下图所示错误提示。



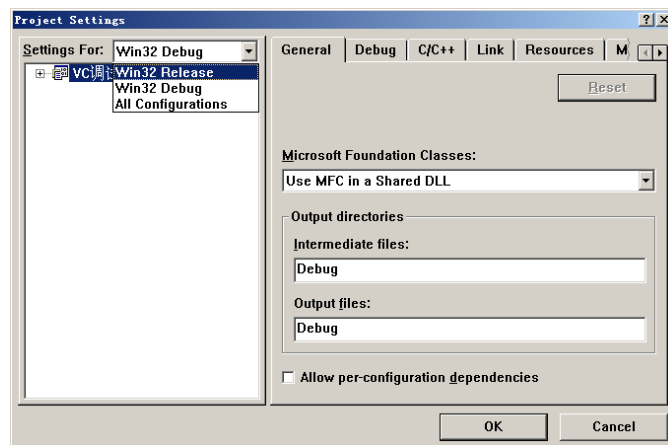
```
: error C2065: 'cuot' : undeclared identifier
: error C2297: '<<' : illegal, right operand has type 'char [2]'
```

The screenshot shows a standard Windows-style error dialog box with a title bar and a list of error messages. The messages are displayed in a monospaced font. At the bottom, there is a navigation bar with buttons for 'Build', 'Debug', 'Find in Files 1', 'Find in Files 2', and 'Results'.

- 另一类错误是逻辑错误，该类错误是没有语法错误的情况下，发生过得让程序无法正确运行的错误。在具体编程中，当修改完语法错误生成了可执行程序后，并不意味着程序已经正确。读者经常会发现程序运行的结果与预期的结果相去甚远。有时甚至在程序执行过程中程序终止或发生死机。这种错误称为就是逻辑错误，或者称为运行错误。这些现象是因为算法设计不当或编程实现时的疏忽造成的。通俗的说这种错误就是**BUG**。
- 所谓调试就是指在发现了程序存在运行错误以后，寻找错误的原因和位置并排除错误。这一工作是非常困难的。**Visual C++**的有两个基本调试功能：找出**BUG**发生的地方和分析如何修改**BUG**。该调试工具允许读者每次只执行一行程序代码。这样可以更方便地找到**BUG**发生的地方。

建立调试环境

- 在Visual C++中每当建立一个工程（Project）时，Visual C++都会自动建立两个版本：Release版本和Debug版本。正如其字面意思所说的，Release版本是当程序完成后，准备发行时用来编译的版本。而Debug版本是用在开发过程中进行调试时所用的版本。
- 在新建立的工程中，读者所看到的是Debug版本。如果选择Release版本，可以单击【Project】|【Settings】命令。



设置断点

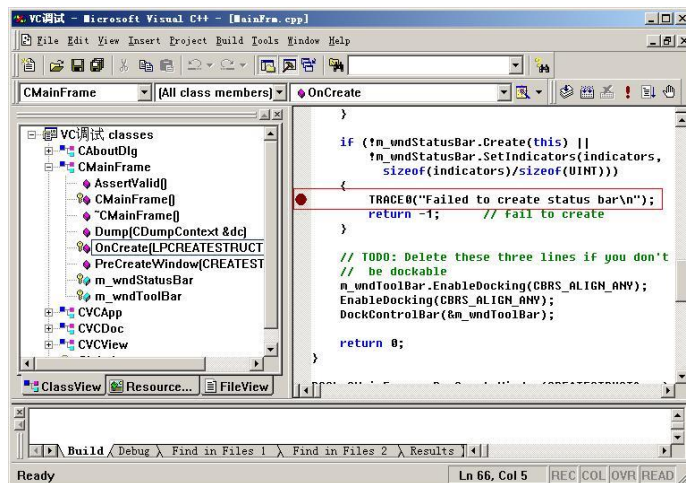
- 设置了调试环境后，读者再来看一下调试的一般过程。所谓调试，就是在程序的运行过程的某一阶段观测程序的状态，而在一般情况下程序是连续运行的，所以必须使程序在某一地点停下来。所以读者进行调试所做的第一项工作就是设立断点。其次，再运行程序，当程序在设立断点处停下来时，再利用各种工具观察程序的状态。程序在断点停下来后，有时需要按用户的要求控制程序的运行，以进一步观测程序的流向。所以下面依次来介绍断点的设置，如何控制程序的运行以及各种观察工具的利用。

ActiveX控件的事件

- 在Visual C++中，用户可以设置多种类型的断点，可以根据断点起作用的方式把这些断点分为如下3类：
- 与位置有关的断点。
- 与逻辑条件有关的断点。
- 与WINDOWS消息有关的断点。

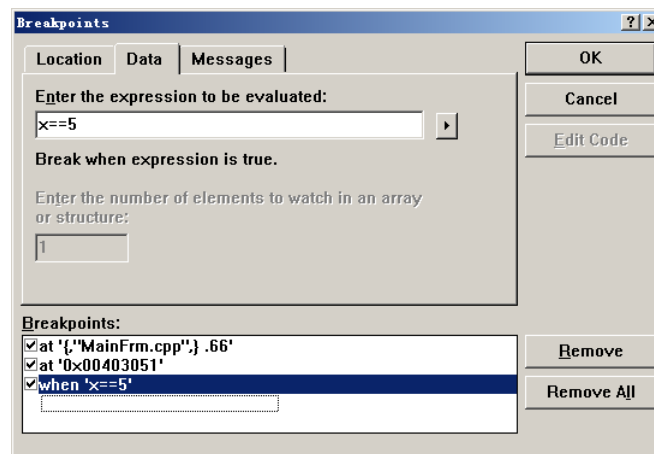
位置断点

- 读者只要把光标移到要设断点的位置。当然这一行必须包含一条有效语句的。然后按工具条上的按钮（Insert/Remove BreakPoint）或按下快捷键【F9】。此时读者将会在屏幕上看到在这一行的左边出现一个红色的圆点，表示这一行设立了一个断点。



逻辑条件触发断点

- 上面所讲的断点主要是由于其位置发挥作用的，即当程序运行到设立断点的地方时程序将会停下来。但有时需要设立只与逻辑条件有关的断点，而与位置无关。读者可以在【Breakpoint】对话框中的【Data】选项卡中输入逻辑条件。

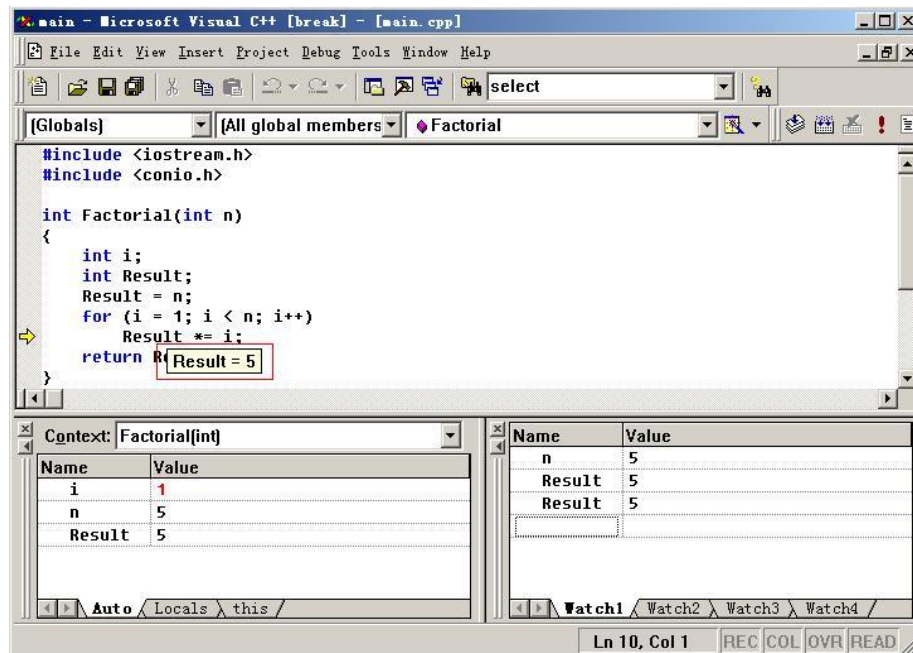


使用查看工具

- 调试过程中最重要的是要观察程序在运行过程中的状态。这样用户才能找出程序的错误之处。这里所说的状态包括各变量的值、寄存存中的值、内存中的值和堆栈中的值等。

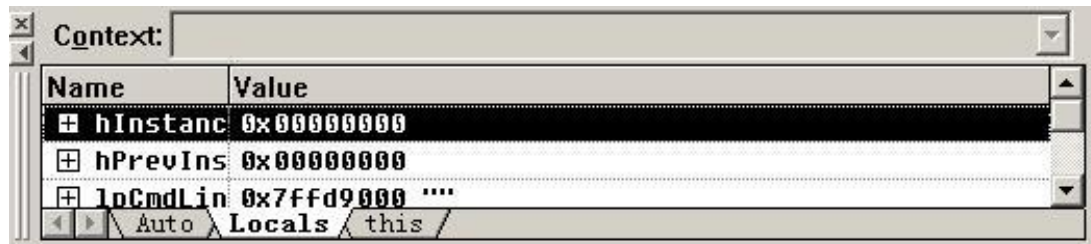
调试信息泡泡

- 当程序在断点停下来后，要观察一个变量或表达式的值的最容易的方法是利用调试信息泡泡。要看一个变量的值，只需在源程序窗口中，将鼠标放到该变量上，将会看到一个信息泡泡弹出。其中显示出该变量的值。要查看一个表达式的值，先选中该表达式，然后将鼠标放到选中的表达式上。同样会看到一个信息泡泡弹出以显示该表达式的值。



变量窗口

- 单击【View】|【Debug Window】|【Variables】命令，则变量窗口将出现在屏幕上。其中显示着变量名及其对应的值。读者将会看到在变量观察窗口的下部有3个标签：【Auto】、【Local】和【This】。选中不同的标签，不同类型的变量将会显示在该窗口中。



观察窗口

- 单击【VIEW】|【Debug Window】|【Watch】命令，则观察窗口将出现在屏幕上，如图20-18所示。在下图的观察窗口中双击Name栏的某一空行，输入要查看的变量名或表达式。回车后读者将会看到对应的值。观察窗口可有多页，分别对应于标签Watch1、Watch2、Watch3等。

快速查看变量对话框

- 在快速查看变量对话框中，读者可以像利用观察窗口一样来查看变量或表达式的值。还可以利用它来改变运行过程中的变量。在【Debug】菜单，选择【Quick Watch】命令。这时屏幕上将会出现【Quick Watch】对话框，如图20-19所示。
- 在上图中的【Expression】编辑框中输入变量名例如a。按回车后在【Current Value】格中将出现变量名及其当前对应的值，如图20-20所示。

