C 语言用 char 数据类型来表示一个 8 位 ANSI 字符，默认情况下，在源代码中声明一个字符串时，C 编译器会把字符串中的字符转换成由 8 位 char 数据类型构成的一个数组：

```
char c = 'A';
char szBuffer[100] = "Hello world";
```

Microsoft 的 C/C++编译器定义了一个内建的数据类型 wchar_t，它表示一个 16 为的 Unicode（UTF-16）字符。通过在字符串前加一个 L 来通知编译器该字符串应当编译为一个 Unicode 字符串。

```
wchar_t c = L'A';
wchar_t szBuffer[100] = L"Hello world";
```

微软又将这些定义为自己的数据类型，在 WinNT.h 中。

```
typedef char CHAR;//8-bit
typedef wchar_t WCHAR;//16-bit

//8-bit
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR;

//16-bit
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR *PCWSTR;
```

还有下面的宏和类型，使用这些宏和类型，无论是 ANSI 还是 Unicode 字符，都能通过编译，而且使用 Windows 数据类型，有利于增强代码的可读性。

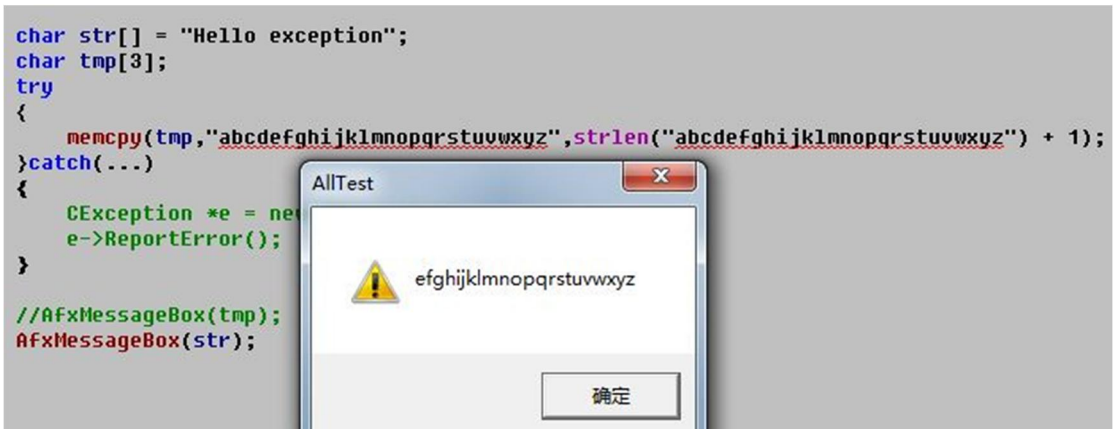
```
#ifdef UNICODE

typedef WCHAR TCHAR,*PTCHAR,PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) L##quote

#else

typedef CHAR TCHAR,*PTCHAR,PTSTR;
typedef CONST CHAR *PCTSTR;
#define __TEXT(quote) quote

#endif

#define TEXT(quote) __TEXT(quote)
```

从 Windows NT 开始，Windows 的所有版本都完全用 Unicode 来构建，也就是说，所有核心函数（创建窗口，显示文本，字符串处理等等）都需要 Unicode 字符串。调用 Windows 函数时，如果向它传入一个 ANSI 字符串（由单字节字符组成的一个字符串），那么函数首先会把字符串转换为 Unicode，再把结果传给操作系统。如果希望函数返回 ANSI 字符串，那么

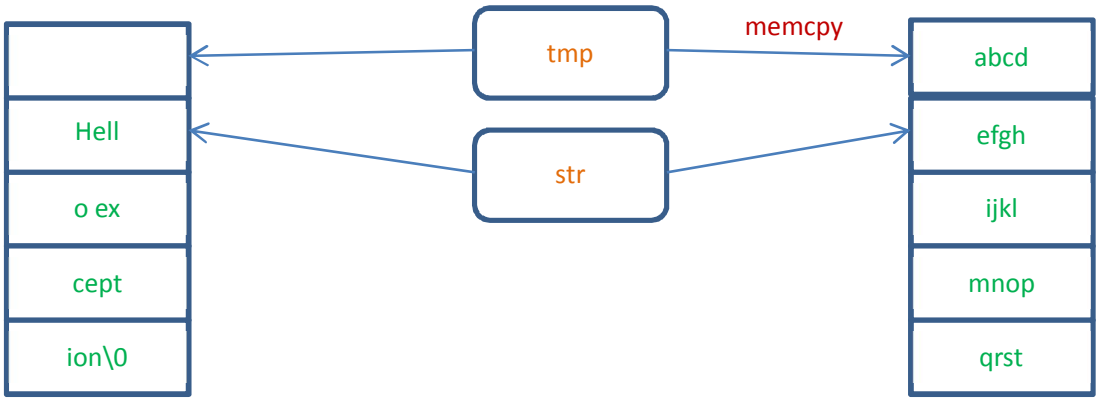
操作系统会先把 Unicode 字符串转换为 ANSI 字符串，再把结果返回给我们的应用程序。所有这些转换都是在幕后进行的。执行这些字符串转换，系统会产生时间和内存上的开销。

C 运行库的 Unicode 函数和 ANSI 函数（新的安全字符串函数）：

在这些新函数发布以前，任何修改字符串的函数都会存在一个安全隐患：如果目标字符串缓冲区不够大，无法容纳生成的字符串，就会导致内存中的数据被破坏。



str 在栈中的高地址，当 memcpy 的操作导致 tem 溢出的时候就会覆盖掉高地址处的内存。



于是这些新的安全函数便应运而生。现在每个函数（如_tcscpy 或_tcscat）都有一个对应的新版本的函数。前面的名称相同，但最后添加了一个_s 后缀，代表 secure。所有这些新函数都有一个共同特征。在将一个可写的缓冲区作为参数传递时，必须同时提供它的大小。这个值应该是一个字符数，通过对缓冲区使用_countof 宏，很容易计算这个值。

所有这些后缀为_s 的安全函数的首要任务是验证传给它们的参数值。要检查的项目包括指针不为 NULL，整数在有效范围内，枚举值是有效的，而且缓冲区足以容纳结果数据。如果这些检查中的任何一项失败，函数都会设置局部于线程的 C 运行时变量 errno。然后，返回一个 errno_t 值来指出成功或失败。然而，这些函数并不实际返回。如果是一次调试版构建（debug），会显示一个 Debug Assertion Failed 对话框，然后终止应用程序。如果是发行版（release），则直接自动终止程序的运行。

C 运行时实际上允许我们提供自己的函数，这样这些函数在检测到一个无效的参数时，就可以调用我们的函数。在这个函数中，我们可以进一步对异常进行处理。为了实现此功能我们必须定义好一个函数，原型如下：

```
void _invalid_parameter(
    const wchar_t * expression,
```

```
        const wchar_t * function,
        const wchar_t * file,
        unsigned int line,
        uintptr_t pReserved
    );
```

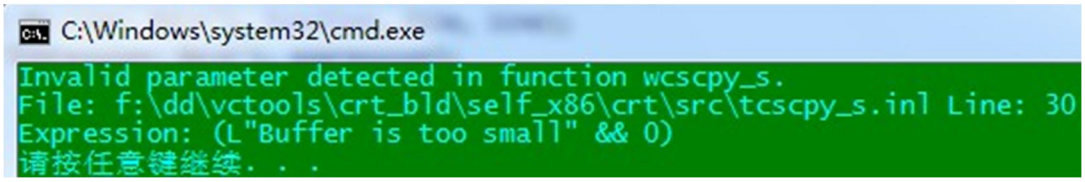
function，file，line，expression 分别描述了出现了错误的函数名称，源代码文件，源代码行数，和代码中出现的错误描述。我们使用_set_invalid_parameter_handler 来注册这个处理程序。我们还要在程序开头的地方调用_CrtSetReportMode(_CRT_ASSERT,0)，禁用掉 Debug Assertion Failed 对话框，不然这个错误对话框依然会出现。这样我们就可以在程序中检查这个 errno_t 的返回值来判断函数执行是否成功。可能的返回值在 errno.h 中有定义。

下面看一个例子。

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <crtdbg.h>
#include <tchar.h>
#include <StrSafe.h>

void _invalid_parameter(
        const wchar_t * expression,
        const wchar_t * function,
        const wchar_t * file,
        unsigned int line,
        uintptr_t pReserved)
{
    _tprintf(_T("Invalid parameter detected in function %s.\n")
        _T("File: %s Line: %d\n"), function, file, line);
    _tprintf(_T("Expression: %s\n"), expression);
}


void _tmain()
{
    _invalid_parameter_handler oldHandle;
    TCHAR szStrFirst[5] = {_T('1'),_T('2'),_T('3'),_T('4'),'\0'};

    _CrtSetReportMode(_CRT_ASSERT,0);

    oldHandle = _set_invalid_parameter_handler(_invalid_parameter);

    if (S_OK == _tcscpy_s(szStrFirst,_countof(szStrFirst),_T("01234")))
    {
        _tprintf("copy is ok!");
    }
```

}

程序的运行结果如图所示：



程序执行出错后的处理已经被我们的函数处理掉了。

C 运行库还新增了一些函数，用于在执行字符串处理时提供更多控制。例如，我们可以控制填充符，或者指定如何进行截断。C 运行库同时提供了函数的 ANSI 和 Unicode 版本。这些函数的名字如下：

StringCchCat，StringCchCatEx，StringCchCopy，StringCchCopyEx，StringCchPrintf，StringCchCopyEx，StringCchPrintf， StringCchPrintfEx。

可以看出，在所有方法的名称中，都含有一个“Cch”这表示 Count of characters，即字符数：通常使用_countof 宏来获取此值。另外还有一系列名称中含有“Cb”的函数，比如 StringCbCat(Ex)， StringCbCopy(Ex)和 StringCbPrintf(Ex)。这些函数要求用字节数来指定大小，而不是字符数：通常使用 sizeof 操作符来获取此值。所有这些函数都返回一个 HRESULT。

不同于安全（后缀为_s）的函数，当缓冲区太小的时候，这些函数会执行截断。为了判断是否发生这种情况，我们可以检测是否返回了 STRSAFE_E_INSUFFICIENT_BUFFER。在这种情况下，源缓冲区可以装入目标可写缓冲区中的那一部分会被复制，而且最后一个可用的字符会被设为'\0'。所以，在前面的例子中，如果用 StringCchCopy 来替代_tcscpy_s，那么 szStrFirst 将包含字符串“0123”。注意，“截断”这个功能可能是也可能不是我们希望的，具体取决于我们想达到什么目标。使用这些函数的 Ex 版本，我们可以决定是否执行开销比较大的填充操作（尤其是目标缓冲区很大的时候），以及用什么字节值来作为填充符使用。

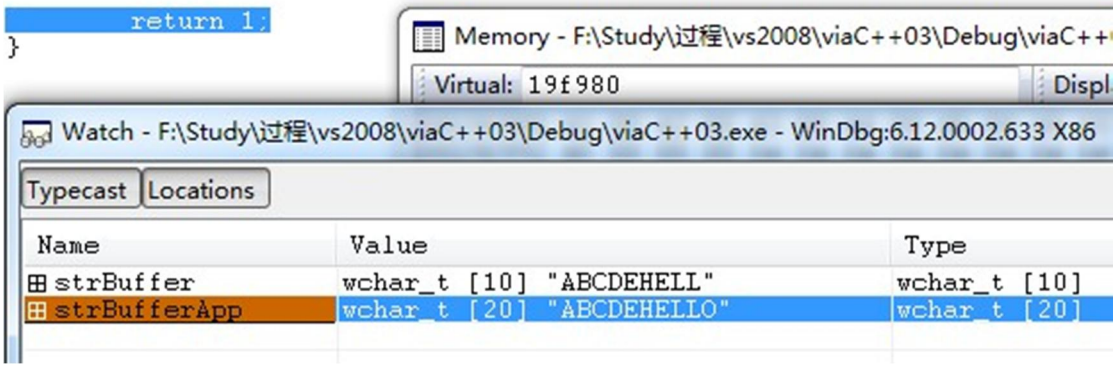
下面用个简单的例子来演示下。

代码如下：

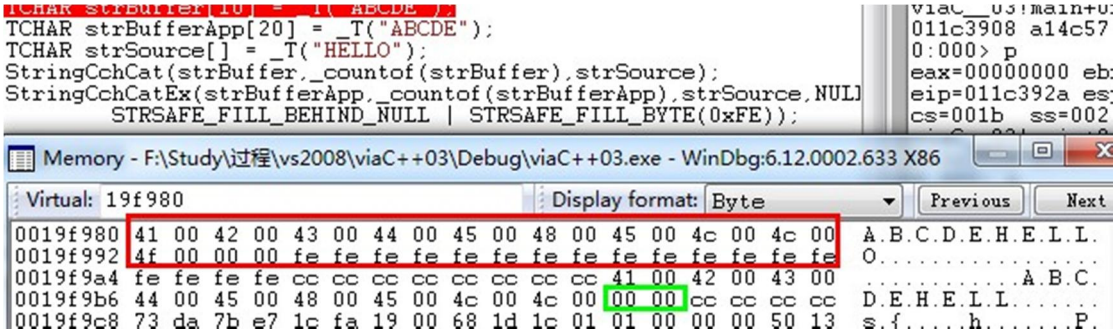
```
#include<stdlib.h>
#include<tchar.h>
#include<strsafe.h>


int main()
{
    TCHAR strBuffer[10] = _T("ABCDE");
    TCHAR strBufferApp[20] = _T("ABCDE");
    TCHAR strSource[] = _T("HELLO");
    StringCchCat(strBuffer,_countof(strBuffer),strSource);
    StringCchCatEx(strBufferApp,_countof(strBufferApp),strSource,NULL,NULL,
        STRSAFE_FILL_BEHIND_NULL | STRSAFE_FILL_BYTE(0xFE));//函数运行成功后用0xfe填充剩余空间

    return 1;
}
```

开发工具是 vs2008，调试工具是 windbg。

```
int main()
{
        TCHAR strBuffer[10] = _T("ABCDE");
        TCHAR strBufferApp[20] = _T("ABCDE");
        TCHAR strSource[] = _T("HELLO");
        StringCchCat(strBuffer,_countof(strBuffer),strSource);
        StringCchCatEx(strBufferApp,_countof(strBufferApp),strSource,NU
                STRSAFE_FILL_BEHIND_NULL | STRSAFE_FILL_BYTE(0xFE));

        return 1;
}
```



运行结果图。两个操作都完成了，下面看看内存处的情况。



如图所示，strBufferApp 在字符串连接完成后'\0'的后面都用 0xfe 来填充了，而 strBuffer 的结尾被设置了'\0'结束符，防止了缓冲区过小而导致的溢出问题。

Windows 同样提供了各种字符串处理函数。其中许多函数（比如 lstrcat 和 lstrcpy）已经不赞成使用了，因为它们无法检测缓冲区溢出问题。而在 ShlwApi.h 中定义了大量方便好用的字符串函数。

我们经常要比较字符串以进行相等性测试或者排序。为此，最理想的函数是 CompareString(Ex)和 CompareStringOrdinal。对于需要以符合用户语言习惯的方式向用户显示字符串，请用 CompareString(Ex)进行比较。

```
int CompareString(          LCID Locale,
    DWORD dwCmpFlags,
    LPCTSTR lpString1,
    int cchCount1,
    LPCTSTR lpString2,
    int cchCount2
);
```

它的第一个参数指定了一个区域设置 ID（locale ID，LCID），这是一个 32 位值，用来标识一种语言。函数使用这个 LCID 来比较两个字符串，具体的做法是检查字符在 LCID 所标识的语言中的含义。以符合当地语言习惯的方式来比较，得到的结果对最终用户来说更有意义。不过，这种比较比基于序数的比较（ordinal comparison）慢。我们可以调用 Windows 函数

GetThreadLocale 来得到主调线程的 LCID。其余 4 个参数指定了两个字符串及其各自的字符长度（字符数，而不是字节数）。如果为 cchCount1 参数传入负数，函数会假设 lpString1 字符串是以 0 来结尾的，并计算字符串的长度；同样的道理适用于后两个参数。如果需要更高级的语言选项，那么应该考虑 CompareStringEx 函数。

为了比较程序内部所用的字符串（如路径名，注册表项/值，XML 元素/属性等），应该使用 CompareStringOrdinal。由于这个函数执行的是码位（code-point）比较，不考虑区域设置，所以速度很快。

它们的返回值有别于 C 运行库的*cmp 字符串比较函数的返回值。0 表明函数调用失败，1 表明 lpString1 小于 lpString2，2 表明 lpString1 等于 lpString2，3 表明 lpString1 大于 lpString2。

代码未经严格测试，开发环境 vs2008。功能很简单，顺便练练 sdk 程序。