第 1 章　C++/CLI 托管编程

C++/CLI 是为.NET 平台特别设计的，它提供了既优雅又强大的新语法支持。VS.NET 通过新的编译模式确保.NET 框架 CLI 的一致性和可验证性，提供了 C++本机环境和 C++/CLI 环境的无缝合并，把.NET 的全部强大功能带给 C++，同时也把 C++强大的功能带给.NET。程序员可以非常自然地在.NET 环境下开发 C++应用程序。用托管代码编写 C++程序，不需要重新学习新的编程语言，只需要了解一些简单的规则而已。

.NET 框架提供的公共语言运行库的功能很庞大。本章主要通过控制台程序介绍 C++/CLI 基本语法，为了与后续 Windows 窗体应用程序衔接，编写了几个简单窗体应用程序。

1.1　.NET 框架

.NET 是微软公司于 2000 年推出的面向互联网时代构筑的新一代平台，.NET 框架（.NET Framework）简化了在高度分布式环境中开发应用程序的工作。一整套基于.NET 框架设计的工具组件，被集成到 Visual Studio .NET（简称 VS.NET）开发环境中，包括 Visual Basic .NET、Visual C++ .NET、Visual C# .NET、Visual J# .NET、ASP.NET 等多种开发工具。Visual C++ .NET（简称 VC.NET）是在 C++基础上产生的，它的基本语法与 C++相同，并具有很多新的特性，增强了 C++的性能，使用起来也更加方便。

VS.NET 是一套完整的开发工具，各种语言都使用相同的集成开发环境（IDE），开发人员可以轻松地创建用混合语言设计的解决方案。

.NET 框架有两个主要的组件：公共语言运行库（CLR，Common Language Runtime）和.NET 框架类库（FCL，Framework Class Library）。公共语言运行库在执行时管理代码，提供核心服务（如内存管理、线程管理和远程处理），强制实施严格的类型安全以及可确保安全性和健壮性的其他形式的代码准确性。类库是综合性的、面向对象的、可重用类型的集合，可用来开发应用程序。

（1）公共语言运行库

CLR 是一个公共语言运行库。在.NET 中，编译生成不是最终的机器码，而是一种中间语言（MSIL，Microsoft Interlanguage）代码。运行时由 CLR 的实时编译器（JIT，Just In Time）“翻译”成特定的机器代码，然后执行。

CLR 支持多种语言，实现各种语言互操作。因此，CLR 提供了公共类型系统（CTS，Common Type System），对数据类型提供定义、管理和使用的严格说明；提供了代码必须遵守的公共语言规范（CLS，Common Language Standard），它包括许多应用程序所需要的基本语言功能。

通过 CLR 运行的代码称为托管（Managed）代码。在 CLR 控制之外的代码称为非托管（UnManaged）代码，通常称为本地代码。

通常所说的 C++，是 ISO/ANSI 标准的，为执行普通的应用程序——非托管的 C++而设计。而 C++/CLI 标准是专门为编写 CLR 托管程序而设计的，它是对 C++的一种扩展。CLI 的意思是通用语言结构（Common Language Infrastructure）。

所有 VB.NET 和 C#.NET 代码都是托管代码。但 VC.NET 提供了两种代码的互操作，允许程序员在项目中的不同程序模块之间，甚至同一个文件中，混合使用托管（C++/CLI）代码和本地（C++）代码，而不同语言编写的组件也可以进行交互。这体现了 VC.NET 功能的强大。

（2）.NET 框架类库

.NET 框架类库 FCL 是一个与公共语言运行库紧密集成、可重用的类的集合。该类库包括类、接口、结构和枚举，它们可以加速和优化开发过程并提供对系统功能的访问。.NET 框架类库使用 CTS 数据和 CLS 规范，是生成.NET 应用程序、组件、控件的基础。.NET 框架类库包含了将近 100 个命名空间。本书对涉及的命名空间会做简单说明。

1.2 简单控制台程序

在一个 VC.NET 应用程序中，既可以使用非托管代码，也可以使用托管代码。本节通过简单的控制台程序，初步认识托管代码和非托管代码的区别。

1.2.1 一个简单程序

【例 1-1】 在 VC.NET 的 IDE 中，建立一个 CLR 空项目，然后添加一个 C++文件（cpp），输入以下代码。

```
//托管代码使用的命名空间
using namespace System;
//非托管代码使用的头文件和命名空间
#include "iostream"
using namespace std;
//托管代码，计算正方形的周长和面积
void writeSquare()
{
    Double len;
    Console::Write("length of a side=");
    len=Double::Parse(Console::ReadLine());
    Console::WriteLine("girth of square={0:f}", 4*len);
    Console::WriteLine("area of square={0:f}", len*len);
}
//非托管代码，计算圆的周长和面积
void coutCircle()
{
    double r;
    cout<<"radius=";
    cin>>r;
    cout<<"girth of circle="<<2*3.14*r<<endl;
    cout<<"area of circle="<<3.14*r*r<<endl;
}
int main()
{
    writeSquare();
    coutCircle();
    return 0;
}
```

从上述程序中，很容易比较托管代码与非托管代码的区别。

（1）使用头文件和命名空间

托管代码中使用的各种数据类型，以及表示控制台应用程序的标准输入流、输出流和错误流的 Console 类都在 System 命名空间中声明，因此，需要语句

```
using namespace System;
```

而语句

```
#include "iostream"
using namespace std;
```

声明了非托管代码中输入、输出需要的头文件和命名空间。

如果在 IDE 中建立 CLR 控制台程序，则系统自动生成程序框架如下：

```
//CLR 控制台应用程序.cpp：主项目文件
#include "stdafx.h"
using namespace System;
int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    return 0;
}
```

程序员可以在这个框架上添加和修改代码。注意，这时的

```
#include "stdafx.h"
```

指令不能删除，并且要放在起始位置。

（2）数据类型说明

在函数 writeSquare 中，变量 len 被说明为 Double 类型。数据类型是 System 命名空间定义的结构。.NET 框架把 C++的类型关键字定义为对应结构的同义词。所以，在函数 coutCircle 中定义变量 r 类型为 double，它是 Double 的同义词。

（3）控制台数据输入

在非托管代码中，iostream 类预定义的标准输入、输出流对象 cin、cout 用于控制台程序的数据交互。而托管代码中，.NET 框架类库 System::Object::Console 类定义了控制台应用程序的标准输入流、输出流和错误流。

上述程序中，用 Console::ReadLine()方法获取输入的一行字符串，然后用 Double::Parse 方法把输入串转换为浮点数形式。Console 类的 Read 有以下几种形式。

- 从标准输入流读取下一个字符：

  int Read()

- 从标准输入流读取下一行字符：

  String^ ReadLine()

- 获取用户按下的字符或功能键，并显示在控制台窗口中。返回一个 ConsoleKeyInfo 对象，描述标准键常数和对应于按下的控制台键的 Unicode 字符：

  ConsoleKeyInfo ReadKey()

- 获取用户按下的字符或功能键，可以选择是否显示在控制台窗口中：

  ConsoleKeyInfo ReadKey(bool intercept)

（4）控制台数据输出

Console 类的 Write 方法和 WriteLine 方法用于输出数据，前者表示输出数据不换行，后者表示输出数据后插入行终止符。

输出方法有很多个重载版本，可以用于输出一个表达式项，也可以用于输出多个表达式项。若输出多个表达式项，则需要用格式字符串参数指定输出格式。语法形式如下：

Write(<字符串>, <输出对象列表>);

WriteLine(<字符串>, <输出对象列表>);

其中，<字符串>可以包含直接输出的字符串和用一对花括号“{ }”相括的格式串。例如，在以下代码中：

```
Console::WriteLine("girth of square={0:f}", 4*len);
```

字符串"girth of square={0:f}"中的"girth of square ="是直接输出的字符串，"{0:f}"称为格式串，第 1 个数字 0 表示该格式用于逗号之后的第一个输出对象，冒号之后的"f"表示用定点格式输出。如果不指示格式符，例如只用"{0}"，则系统自动选择最节省的输出格式。格式串之间用逗号分隔。<字符串>之后是<输出对象列表>，输出对象可以是表达式或具有输出值的对象名。

Console 类还有一些常用的方法，见表 1.1。这些方法的具体例程，请读者参阅 MSDN 资源。

表 1.1　Console 类其他常用方法

名　　称	说　　明
Beep	通过控制台扬声器播放提示音
Beep(Int32, Int32)	通过控制台扬声器播放具有指定频率和持续时间的提示音
Clear	清除控制台缓冲区和相应的控制台窗口的显示信息
ResetColor	将控制台的前景色和背景色设置为默认值
SetBufferSize	将屏幕缓冲区的高度和宽度设置为指定值
SetCursorPosition	设置光标位置
SetWindowPosition	设置控制台窗口相对于屏幕缓冲区的位置
SetWindowSize	将控制台窗口的高度和宽度设置为指定值

1.2.2　格式串

格式设置是指将类、结构或枚举值的实例转换为字符串的表示形式，或者把字符串形式的数据还原为原始数据类型。.NET 提供了丰富的格式设置支持，以满足开发人员的各种要求。

格式串用于设置各输出项的输出格式。各种数据类（例如，Int32 等）的 ToString 方法、Console 类和 StreamWriter 类的 Write 和 WriteLine 方法、String::Format 方法及 StringBuilder::AppendFormat 方法都支持格式串。

一个格式串可以包含多个用花括号相括的格式项。格式项的语法形式为：

{ <索引> [, <对齐>] [: <格式说明>] }

① <索引>

“索引”参数说明符是一个从 0 开始的数字，标识输出参数列表中对应项的序号。参数说明符为 0 表示输出列表的第一个对象，参数说明符为 1 表示列表的第二个对象，其余类推。

通过指定相同的“索引”参数，多个格式项可以引用对象列表中的同一个元素。例如：

```
Console::WriteLine("{0:X} {0:E} {0:N}", object);
```

分别将同一个 object 数值设置为十六进制数、科学记数法和数字格式输出。

每个格式项都可以引用列表中的任一对象。例如，如果有三个输出对象，则可以通过指定类似于"{1} {0} {2}"的复合格式字符串来设置第二个、第一个和第三个对象的格式。未被格式项引用的对象会被忽略。如果“索引”参数说明符的值超出对象列表范围的项，将导致运行时的异常。

② <对齐>

可选的“对齐”参数是一个带符号的整数，指示格式的字段宽度。

若“对齐”值小于设置了格式的字符串的长度，则“对齐”参数被忽略；

若“对齐”值为正数，则字段中设置格式的数据为右对齐；

若“对齐”值为负数，则字段中设置格式的数据为左对齐；

若需要填充，则使用空白。

③ <格式说明>

可选的“格式说明”参数是适合正在设置格式的对象类型的格式字符串。数值型的“格式说明”参数可以附带示数精度指示。表 1.2 中列出了三种标准格式说明符的形式。在后续的例程中，将会应用这些格式说明符输出各种类型的数据。实际上，.NET 还提供各种自定义格式和复合格式字符串，有需要的读者请查阅相关资源。

若不指定“格式说明”参数，则由系统根据对象类型选择输出格式。

若输出多个表达式的值，而不需要提示字符串，可以用这样的形式：

```
Console::WriteLine(" {0},{1},{2}", x, y, z);
```

此时，x、y、z 的值以逗号分隔输出，输出格式由 x、y、z 的类型决定。

如果 Write 方法和 WriteLine 方法只用于输出一个表达式的值，则可以省略格式串。例如：

```
Console::WriteLine(4*len);
```

表 1.2　标准格式说明符

格式说明符	说　明	示　例
标准数字格式字符串		
C 或 c	货币值	−123.456("C")→￥−123.46
D 或 d	十进制整型	1234("D")→1234 −1234("D6")→−001234
E 或 e	科学指数计数法	1052.0329112756("E")→1.052033E+003 1052.0329112756("e2")→−1.05e+003
F 或 f	定点	1234.567("F")→1234.57 −1234.56("F4")→−1234.5600
G 或 g	常规，用最紧凑方式表示数据	−123.456("G")→−123.456 123.4546("G4")→123.5 −1.234567890e−25("G6")→−1.23457E−25
N 或 n	带分隔符输出，默认精度	1234.567("N")→1,234.57
P 或 p	百分比，结果乘以 100 并显示百分比符号，默认精度	1("P")→100.00 % −0.39678("P1")→−39.7 %
R 或 r	数字与对应字符串转换	"123456789.12345678"("R")→123456789.12345678 −1234567890.12345678("R")→"−1234567890.1234567"
X 或 x	十六进制数	255("X")→FF −1("X4")→00FF
枚举格式字符串		
G 或 g	如有可能，将枚举项显示为字符串值，否则显示当前实例的整数值	
F 或 f		
D 或 d	以尽可能短的表示形式将枚举项显示为整数值	
X 或 x	将枚举项显示为十六进制值	
标准日期和时间格式字符串		
d	短日期模式	2016-7-3 16:01:41→2016-7-3
D	长日期模式	2016-7-3 16:01:41→2016 年 7 月 3 日
f	完整日期/时间模式（短时间）	2016-7-3 16:01:41→2016 年 7 月 3 日 16:01
F	完整日期/时间模式（长时间）	2016-7-3 16:01:41→2016 年 7 月 3 日 16:01:41
g	常规日期/时间模式（短时间）	2016-7-3 16:01:41→2016-7-3 16:01
标准日期和时间格式字符串		
G	常规日期/时间模式（长时间）	2016-7-3 16:01:41→2016-7-3 16:01:41
M 或 m	月/日模式	2016-7-3 16:01:41→7 月 3 日
O 或 o	日期/时间模式	2016-7-3 16:01:41→2016-7-3T16:01:41.2187500+08:00
R 或 r	RFC1123 规范模式	2016-7-3 16:01:41→Sun, 03 Jul 2016 16:01:41 GMT
S	可排序日期/时间模式	2016-7-3 16:01:41→2016-7-3T16:01:41

续表

格式说明符	说　明	示　例
标准日期和时间格式字符串		
t	短时间模式	2016-7-3 16:01:41→16:01
T	长时间模式	2016-7-3 16:01:41→16:01:41
u	通用的可排序日期/时间模式	2016-7-3 16:01:41→2016-7-3 16:01:41Z
U	通用完整日期/时间模式	2016-7-3 16:01:41→2016 年 7 月 3 日 8:01:41
Y 或 y	年月模式	2016-7-3 16:01:41→2016 年 7 月

注：① 表中圆括号内表示使用的输出格式说明符。

② 表中箭头“→”左边为要输出的数据，右边为按特定格式的输出结果。

1.3　C++/CLI 数据

在 C++的数据表示中，有基本数据类型、结构类型、指针类型等。这些数据类型在 C++/CLI 中都有对应的定义，并扩展了一系列的功能。

System 命名空间包含基本类和基类、模板类，这些类定义常用的值和引用数据类型、事件和事件处理程序、接口、属性和异常处理。这个命名空间包括应用程序使用的基础数据类型的类，例如：Object（继承层次结构的根）、Byte、Char、Array、Int32 和 String 等。

1.3.1　基本数据类型

.NET 框架内置了 C++基本数据类型的关键字，并把这些关键字预定义为 System 命名空间中托管类型的别名，当使用.NET Framework 类型编写代码时，可以应用 C++相应的关键字。

表 1.3 列出了 C++和 C++/CLI 类型标识符的对照，以及数据类型的取值范围。

表 1.3　C++与 C++/CLI 基本类型标识符的对照

C++类型	C++/CLI 类型	字　节	说　明
bool	Boolean	1	表示逻辑值 true 或 false
signed char	SByte	1	有符号整数，取值范围-128～+127
unsigned char	Byte	1	无符号整数，取值范围 0（Byte::MinValue）～255（Byte::MaxValue）
wchar_t	Char	2	16 位 Unicode 字符
Decimal	Decimal	16	128 位十进制数值
Short signed short	Int16	2	有符号整数，取值范围-32768～+32767
unsigned short	UInt16	2	无符号整数，取值范围 0～65535。不符合 CLS
int signed int long signed long	Int32	4	有符号整数，取值范围-2 147 483 648（Int32::MinValue）～2 147 483 647（Int32::MaxValue）
unsigned int unsigned long	UInt32	4	无符号整数，取值范围 0～4 294 967 295。不符合 CLS
_int64 signed _int64	Int64	8	有符号整数， 取值范围-9 223 372 036 854 775 808～+9 223 372 036 854 775 807
unsigned _int64	UInt64	8	无符号整数，取值范围 0～18 446 744 073 709 551 615。不符合 CLS

续表

C++类型	C++/CLI 类型	字　节	说　明
float	Single	4	单精度浮点数，取值范围-3.402823e38～+3.402823e38
double	Double	8	双精度浮点数，取值范围-1.79769313486232e308～+1.79769313486232e308
void	Void		为不返回值的方法（函数）指定返回值类型

C++/CLI 数据类型与 C++数据类型运算方式一致，并且重载了 Object 类的 Parse 方法，把数值数据转换成对应字符串。

Parse 方法将数字的字符串表示形式转换为等效指定类型的数值，语法形式如下：

<type> Parse(String^ s)

其中，<type>为 System 命名空间定义的数据类型。此方法可以用类或实例形式调用，例如：

```
Int16 t;
Double p1,p2;
t=Int16::Parse("618");
p1=Double::Parse("3.1415");
p2=p1.Parse("3.1415");
```

ToString 方法将实例的数值转换为字符串表示形式：

String^ ToString()

String^ ToString(String^ format)

其中，format 参数是指定格式的字符串。

【例 1-2】　数值与字符串的转换。程序运行结果如图 1.1 所示。

图 1.1　数值与字符串的转换

```
using namespace System;
int main()
{
    Int16 t;
    Double p1,p2;
    t=Int16::Parse("618");              //把字符串转换成整数
    p1=Double::Parse("3.1415");         //把字符串转换成浮点数
    p2=p1.Parse("3.1415");
    Console::WriteLine("t={0:d}\np1={1:f5}\np2={2:f5}",t,p1,p2);
    String ^s1=p1.ToString();           //把浮点数转换成对应字符串
    Console::WriteLine(s1);
    String ^s2=p2.ToString("e");        //按格式转换浮点数
    Console::WriteLine(s2);
    Console::Read();                    //停留在控制台显示，按任意键时结束程序
}
```

【例 1-3】　格式化输出负整数和浮点数。程序运行结果如图 1.2 所示。

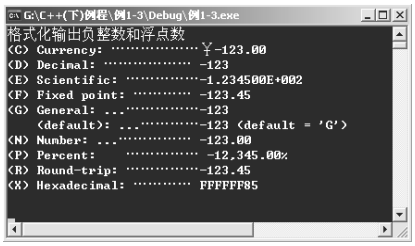

图 1.2　格式化输出负整数和浮点数

```
using namespace System;
int main()
{
    Console::Clear();
    Console::WriteLine("格式化输出负整数和浮点数");
    Console::WriteLine(
        "(C) Currency: ……{0:C}\n" +
        "(D) Decimal: ……{0:D}\n" +
        "(E) Scientific: ……{1:E}\n" +
        "(F) Fixed point: ……{1:F}\n" +
        "(G) General: ...……...{0:G}\n" +
```

```
            "      (default): ...……{0} (default='G')\n" +
            "(N) Number: ...……..{0:N}\n" +
            "(P) Percent:……....{1:P}\n" +
            "(R) Round-trip: ……{1:R}\n" +
            "(X) Hexadecimal: ……{0:X}\n",
            -123, -123.45f);
        Console::Read();
    }
```

1.3.2 枚举类型

C++/CLI 的枚举类型使用一个带空格的关键字 enum class 声明。例如：

```
enum class Color {Yellow=1, Blue, Green};
```

声明一个枚举类型后，可以建立它的实例（变量）。托管枚举数据与C++枚举数据在使用上有所区别：

托管枚举值需要通过域运算符或枚举名来访问。

托管枚举值可以通过默认 G 格式或 F 格式输出枚举定义的字符串，通过 D 格式或 X 格式输出枚举序值。

托管枚举类型数据不能像 C++枚举类型数据那样直接与整数或其他简单类型数据进行运算。若要用托管枚举类型数据与其他类型数据进行运算，必须使用类型转换。

【例 1-4】　格式化枚举值。程序运行结果如图 1.3 所示。

```
using namespace System;
//声明托管枚举类型
enum class Colors
{
    Red, Green, Blue, Yellow
};
int main()
{
    //声明枚举变量
    Colors myColor1=Colors::Blue;
    Colors myColor2=Colors(1);
    Console::WriteLine("格式化输出枚举值：");
    Console::WriteLine(
        "(G) General: ……..{0:G}\n" +
        "      (default): ……...{1:G} (default='G')\n" +
        "(F) Flags: ……..{0:F} \n" +
        "(D) Decimal number: ……{1:D}\n" +
        "(X) Hexadecimal: ……{2:X}\n",
        myColor1,myColor2,Colors::Yellow);
    Console::Read();
}
```

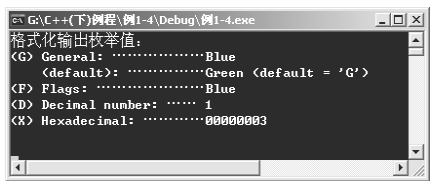

图 1.3　格式化输出枚举值

建立 Windows 窗体应用程序时，需要通过控件来显示数据。文本类的控件都有 Text 属性，它的类型是 String。当把一个 String 类型值赋给控件的 Text 属性时，这些文本数据就在控件上显示出来了。

验证使用控件显示文本，需要创建 Visual C++ Windows 窗体应用程序，并在窗体上建立显示数据用的文本框 TextBox 控件和执行代码的命令按钮 Button 控件。

有关 Windows 程序的基本界面设计详细操作请参阅第 3 章。

【例 1-5】　在文本框中显示枚举数据。使用 String::Format 可以把枚举数据显示在文本控件上。创建 Windows 应用程序后，在窗体上建立一个文本框控件 textBox1，把它的 MultiLine 属性设为 true；再建立一个按钮控件 button1，并编写 button1_Click 鼠标单击事件函数。程序运行结果如图 1.4 所示。

```
enum class Colors
{
    Red, Green, Blue, Yellow
};
private: System::Void button1_Click(System::Object^
                    sender, System::EventArgs^ e)
{
    Colors myColor1=Colors::Blue;
    Colors myColor2=Colors(1);
    String ^s;
    //生成格式串
    s=String::Format(
        "(G) General: ……{0:G}\r\n" +
        "      (default): ……...{1:G} (default='G') \r\n" +
        "(F) Flags: ……..{0:F}\r\n" +
        "(D) Decimal number: ……{1:D}\r\n" +
        "(X) Hexadecimal: ……{2:X}\r\n",
        myColor1,myColor2,Colors::Yellow);
    textBox1->Text=s;                    //把格式串赋给文本框的 Text 属性
}
```

图 1.4　在文本框中显示枚举数据

1.3.3　字符与字符串

字符和字符串是程序中常用的数据。在.NET 中，System 命名空间的 Char 结构和 String 类分别定义了字符和字符串，它们还可以与 C++的字符、字符串进行转换。

1．字符

.NET 使用 Char 结构表示 Unicode 字符。Unicode 称为统一码、万国码，是一种国际标准字符集，它为世界各种语言绝大多数已知的字符集定义了唯一的 16 位二进制编码，以满足计算机跨语言、跨平台进行文本转换、处理的要求。Unicode 字符包括基本 ASCII 字符集、各种其他字母、重音符、货币符号、小数、音调符号以及数学和技术符号。

Char 字符可以分为大写字母、小写字母、控制字符、货币符号、标点符号和数学符号等。字符的表示形式与 C++一致，反斜杠"\"表示为特殊意义的转义字符。

Char 结构重载了基本算术运算和逻辑运算符，可以像 C++的 char 类型数据一样方便地进行操作，以及在运算中做强类型转换。表 1.4 中列出了用于判断字符类别和数据转换的主要方法。

表 1.4　Char 的主要方法

方　法	说　明
IsControl	判断指定字符是否属于控制字符类别
IsDigit	判断指定字符是否属于十进制数字类别
IsLetter	判断指定字符是否属于字母类别
IsLower	判断指定字符是否属于小写字母类别
IsNumber	判断指定字符是否属于数字类别
IsPunctuation	判断指定字符是否属于标点符号类别
IsSeparator	判断指定字符是否属于分隔符类别
IsSymbol	判断指定字符是否属于符号字符类别
IsUpper	判断指定字符是否属于大写字母类别
IsWhiteSpace	判断指定字符是否属于空白类别
Parse	将指定字符串的值转换为它的等效 Unicode 字符

续表

方　法	说　明
ToLower	将指定字符转换为它的小写符号
ToString	将指定字符转换为其等效的字符串表示
ToUpper	将指定字符转换为它的大写符号

【例 1-6】　用控制台程序测试字符操作。

```
using namespace System;
int main()
{
    Char chA='A';
    Char ch1='1';
    String^ str="test string";
    Console::WriteLine(chA.CompareTo('B'));              //输出：-1
    Console::WriteLine(chA.Equals('A'));                 //输出：true
    Console::WriteLine(Char::GetNumericValue(ch1));      //输出：1
    Console::WriteLine(Char::IsControl('\t'));           //输出：true
    Console::WriteLine(Char::IsDigit(ch1));              //输出：true
    Console::WriteLine(Char::IsLetter(','));             //输出：false
    Console::WriteLine(Char::IsLower('u'));              //输出：true
    Console::WriteLine(Char::IsNumber(ch1));             //输出：true
    Console::WriteLine(Char::IsPunctuation('.'));        //输出：true
    Console::WriteLine(Char::IsSeparator(str, 4));       //输出：true
    Console::WriteLine(Char::IsSymbol('+'));             //输出：true
    Console::WriteLine(Char::IsWhiteSpace(str, 4));      //输出：true
    Console::WriteLine(Char::Parse("S"));                //输出：S
    Console::WriteLine(Char::ToLower('M'));              //输出：m
    Console::WriteLine("x");                             //输出：x
    Console::Read();
}
```

【例 1-7】　用 Windows 窗体应用程序中的文本框中显示字符操作结果。测试以下程序需要在窗体上建立 4 个文本框和一个按钮，并编写按钮单击事件函数。

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    //以下操作将在文本框中显示字符 K
    Char ch1='A';
    textBox1->Text=(Char(ch1+10)).ToString();
    //以下操作将在文本框中显示整数 76
    Char ch2='B';
    textBox2->Text=(ch2+10).ToString();
    //以下操作将在文本框中显示逻辑值 true
    Char ch3='C';
    textBox3->Text=(ch3.IsLetter(ch3)).ToString();
    //以下操作将在文本框中显示整数 68
    Int16 t;
    t=Char::Parse("D");
    textBox4->Text=t.ToString();
}
```

2．字符串

C++/CLI 字符串是 System 命名空间定义的 String 类，其对象是 Char 对象的队列。String 类提供了不同形式的构造函数、公有方法和运算符重载函数，可以很方便地进行 String 对象的操作。System::String 类的操作方法与 C++的 String 类很相似。表 1.5 中列出了 String 类的主要方法，这

些方法的原型，读者可以查阅 MSDN 资料。一些方法如比较、连接、复制等，已经重载了运算符操作。

表 1.5　String 的主要方法

方　法	说　明
Compare	比较两个指定的 String 对象。已重载“>”、“<”、“==”运算符
Concat	连接 String 的一个或多个对象，已重载“+”运算符
Contains	判断指定的 String 对象是否出现在此字符串中
Copy	创建具有相同值的 String 的新对象，已重载“=”运算符
Equals	判断两个 String 对象是否具有相同的值，已重载“==”运算符
Format	将指定的 String 中的每个格式项替换为相应对象的值的文本等效项
Insert	在指定索引位置插入一个指定的字符串
Remove	从对象中删除指定个数的字符
Replace	将指定字符或串替换为其他指定的字符或串
Substring	检索子字符串
ToLower	将 String 对象转换为小写形式副本
ToString	将指定实例的值转换为 String 对象
ToUpper	将 String 对象转换为大写形式的副本

String 的 Format 方法把输出项替换为等效格式化的文本，用控件的 Text 属性显示数据时，可以达到类似控制台的数据格式化输出的效果。

【例 1-8】　在文本框中显示格式化数据。本例将整数和浮点数按格式转换成字符串。本程序在 Windows 窗体应用程序的窗体上建立一个命令按钮 button1 和一个文本框控件 textBox1，用命令按钮的单击事件函数 button1_Click 在文本框 textBox1 上显示字符串。

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    String ^s;
    s=String::Format("formatstring:{0,5:d}\t" + "{0:x}\t" + "{1:f2}", 123, 45.3456);
    textBox1->Text=s;
}
```

上述 Format 方法中，格式串由"+"连接三个子串运算符而成。第一个子串除了自身的字符外，还嵌入了格式变量{0,5:d}，把整数 123 转换成长度为 5 的十进制数字符串；第二个子串中的格式变量{0:x}把整数 123 转换成十六进制形式字符串；第三个格式串中的格式变量{1:f2}把浮点数 45.3456 转换成精度为 2 的定点数形式字符串。程序运行结果如图 1.5 所示。

【例 1-9】　在文本框中格式化显示多行数据。本例把文本框的 Multiline 属性设为 true，以便显示多行文本。代码中的"\r\n"用于显示换行。程序运行结果如图 1.6 所示。

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    String ^s;
    s=String::Format(
        "(C) Currency:……{0:C}\r\n" +
        "(D) Decimal:……{0:D}\r\n" +
        "(E) Scientific:……..…..{1:E}\r\n" +
        "(F) Fixed Point:…..……{1:F}\r\n" +
        "(G) General:……..…{0:G}\r\n" +
        "default='G':……...{0}\r\n" +
        "(N) Number:……{0:N}\r\n" +
        "(P) Percent:……..…..{1:P}\r\n" +
        "(R) Round-trip:…..……{1:R}\r\n" +
```

```
            "(X) Hexadecimal:……{0:X}\r\n",
            -123 , -123.45f
        );
        textBox1->Text=s;
    }
```

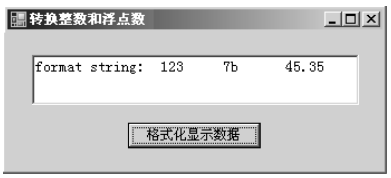

图 1.5　在文本框中显示格式化数据

图 1.6　在文本框中格式化显示多行数据

3．字符串转换

在 C++中，习惯地用 ASCII 码的 char* 类型或 basic_string 类来表示字符串。但如果在文本控件中利用 Text 属性显示字符串，则要求 Text 属性的类型为 System::String。String 类的构造函数及 Runtime::InteropServices 命名空间的相应方法可以实现不同类型字符串之间的转换。下面通过代码进行简单说明。

① 将 char*字符串转换成 String 字符串

```
char *orig="Hello, World!";
String ^systemstring=gcnew String(orig);
systemstring += "(System::String)";
```

② 将 string 字符串转换成 System::String 字符串

```
string orig("Hello, World!");
String ^systemstring=gcnew String(orig.c_str());
systemstring += "(System::String)";
```

经过转换之后，systemstring 就可以向控件的 Text 属性赋值了。例如：

```
textBox1->Text= systemstring;
```

注意，若在 Windows 程序中使用 C++的 String 类定义字符串，则需要包含 string 头文件和使用 std 命名空间：

```
#include<string>
using namespace std;
```

③ 将 String 字符串转换成 char* 和 string 字符串

```
#include <iostream>
#include <string>
using namespace std;
using namespace System;
using namespace Runtime::InteropServices;
int main()
{
    String ^ s=gcnew String("Hello, World!");
    //转换成 char*字符串
    char * chars=(char*)(Marshal::StringToHGlobalAnsi(s)).ToPointer();
    out<<chars<<endl;
    //转换成 string 字符串
    string ss=chars;
    cout <<ss<<endl;
}
```

System::Runtime::InteropServices 命名空间提供托管代码与非托管代码的交互操作。

Marshal::StringToHGlobalAnsi(s)把 String 类型字符串 s 的内容复制到非托管内存中，并转换为 ANSI 格式。 ToPointer 方法把内存实例转换为 char*指针。 这些详细资料可以从 MSDN 的 Visual C++编程指南中查找。

1.3.4 日期时间

DateTime 是 System 命名空间定义的结构，表示值范围在公元 0001 年 1 月 1 日午夜 12:00:00 到公元 9999 年 12 月 31 日晚上 11:59:59 之间的日期和时间。时间值以 100 毫微秒为单位（该单位称为刻度）进行计量。DateTime 类有不同形式的构造函数，见表 1.6。表 1.7 中列出了 DateTime 的常用属性。

表 1.6 DateTime 的主要构造函数

名　　称	说　　明
DateTime (Int64)	将 DateTime 结构的新实例初始化为指定的刻度数
DateTime (Int32, Int32, Int32)	将 DateTime 结构的新实例初始化为指定的年、月和日
DateTime (Int32, Int32, Int32, Int32, Int32, Int32)	将 DateTime 结构的新实例初始化为指定的年、月、日、时、分和秒
DateTime (Int32, Int32, Int32, Int32, Int32, Int32, Int32)	将 DateTime 结构的新实例初始化为指定的年、月、日、时、分、秒和毫秒

表 1.7 DateTime 的常用属性

名　　称	说　　明
Date	日期部分，格式为 00:00:00
Day	表示的日期为该月中的第几天
DayOfWeek	表示的日期是星期几
Hour	表示日期的小时部分
Millisecond	表示日期的毫秒部分
Minute	表示日期的分钟部分
Month	表示日期的月份部分
Now	获取一个 DateTime 对象，该对象设置为此计算机中的当前日期和时间
Second	表示日期的秒部分
Year	获取此实例所表示日期的年份部分

DateTime 结构重载了必要的算术运算符，还提供了判断是否为闰年的方法：

bool DateTime::IsLeapYear(int year);

其中，year 参数是 4 位有效年份的整数。

【例 1-10】 DateTime 的属性和判断闰年的方法。

```
System::DateTime moment=System::DateTime::Now;   //说明当前时间对象
int year=moment.Year;                            //获取年
bool b=moment.IsLeapYear(moment.Year);           //判断是否为闰年
int month=moment.Month;                          //获取月
int day=moment.Day;                              //获取日
System::DayOfWeek dw=moment.DayOfWeek;           //获取星期几的枚举值
int hour=moment.Hour;                            //获取时
int minute=moment.Minute;                        //获取分
int second=moment.Second;                        //获取秒
```

```
int millisecond=moment.Millisecond;                    //获取毫秒
```

【例 1-11】　格式化输出当前的日期和时间。程序运行结果如图 1.7 所示。类似地，可以使用 String::Format 方法在文本控件上格式化地显示日期和时间数据。

```
using namespace System;
int main()
{
    DateTime thisDate=DateTime::Now;
    Console::Clear();
    Console::WriteLine("格式化输出当前日期时间：");
    Console::WriteLine(
        "(d) Short date: ……....{0:d}\n" +
        "(D) Long date: ……....{0:D}\n" +
        "(t) Short time: ……{0:t}\n" +
        "(T) Long time: ……....{0:T}\n" +
        "(f) Full date/short time: …….{0:f}\n" +
        "(F) Full date/long time:…….{0:F}\n" +
        "(g) General date/short time: ..{0:g}\n" +
        "(G) General date/long time: ..{0:G}\n" +
        "       (default): ……...{0} (default='G')\n" +
        "(M) Month: ……..{0:M}\n" +
        "(R) RFC1123: ……...{0:R}\n" +
        "(s) Sortable: ……...{0:s}\n" +
        "(u) Universal sortable:……...{0:u} (invariant)\n" +
        "(U) Universal full date/time:.{0:U}\n" +
        "(Y) Year: ……..{0:Y}\n",
        thisDate);
    Console::Read();
}
```

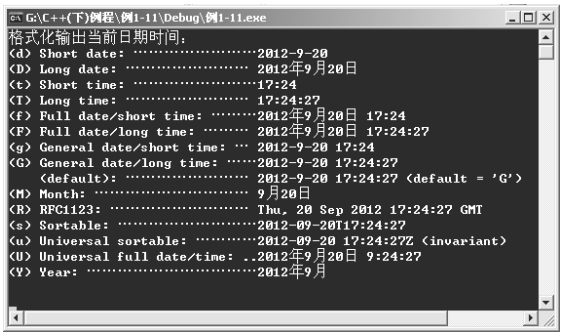

图 1.7　格式化输出当前日期和时间

1.3.5　数组

System::Array 类是支持 C++/CLI 语言数组 array 实现的基类，它提供数组的长度属性，以及创建、操作、搜索和排序数组的方法。用户应当使用由语言提供的 array 构造数组。

1．一维数组

array 是一个类模板。声明托管数组的语法形式为：

array<type> ^ arrayNme=gcnew array<type> (size) ;

其中，type 是数组元素类型，size 是数组元素的数量，即数组长度。

例如，声明一个长度为 5 的整型数组：

```
array <int> ^ myArray1=gcnew array<int>(5);
```

其中，关键字 array 以及尖括号中的数组元素类型 int 声明了数组对象。“^”表示 myArray 是一个数组的句柄（详见 1.4 节）。

关键字 gcnew 用于分配内存，这类似于 C++中的动态数组。

声明托管数组的同时可以提供初始化值列表，和 C++中的数组一样，初始化值列表中的元素数量不能大于数组的长度。如果初始化值列表的元素数量小于数组元素数量，则剩余元素简单地保存为默认值。例如：

```
array <int> ^ myArray2=gcnew array<int>(5) {1, 2, 3};
```

也可以使用初始化值列表确定数组长度。以下语句声明 myArray3 长度为 5：

```
array <int> ^ myArray3={1, 2, 3, 4, 5};
```

2．高维矩阵数组

多维矩阵数组指的是数组的每行必须具有相同的列数。这种数组存放在一个连续的存储区段中，具有较快的访问速度。高维数组的声明形式为：

```
array<type, rank> ^ arrayNme=gcnew array<type, rank> (size) ;
```

其中，rank 指定数组的维数，最多可以有 32 维；size 参数用逗号分隔，用于指定各维的元素数量。

例如，以下代码声明一个 3 行 4 列的二维整型数组：

```
array<int, 2> ^ myArray4=gcnew array<int, 2> (3,4);
```

其中，尖括号中的整数 2 表示 myArray4 是一个二维（行、列）数组，圆括号中的参数表示第一维（索引值为 0）有 3 行，第二维（索引值为 1）有 4 列。

托管的矩阵数组元素访问形式与本地 C++语法形式不同，元素的行列索引值位于同一个方括号内，中间用逗号分隔。数组 myArray4 的元素表示为：

```
myArray4[0,0]    myArray4[0,1]    myArray4[0,2]    myArray4[0,3]
myArray4[1,0]    myArray4[1,1]    myArray4[1,2]    myArray4[1,3]
myArray4[2,0]    myArray4[2,1]    myArray4[2,2]    myArray4[2,3]
```

以下都是二维矩阵数组的声明：

```
//声明一个 2 行 2 列的二维字符串数组
array< String^, 2 > ^ myArray5=gcnew array< String^, 2 >
{ { "Apple", "Banana"}, { "Cat", "Dog"} };
//声明一个 3 行 2 列的二维浮点型数组
array< double, 2 > ^ myArray6=gcnew array< double, 2 > (3, 2)
{ { 1.1, 2.2 }, { 3.3, 4.4 } };
```

3．不规则数组

不规则数组以一维数组的形式维护，它的每个元素表示一个一维数组。不规则数组需要独立地为每行元素分配内存，因此各行元素的长度可以不相同。

以下语句声明一个长度为 3 的一维数组，它的元素是一维整型数组：

```
array< array<int>^ > ^irregAry=gcnew array< array<int>^ > (3);
```

从语法分析可以看到外层尖括号：

```
array< > ^irregAry
```

说明 irregAry 是一个数组。内层尖括号：

```
array<int>^
```

说明数组是元素类型，每个元素都是一维整型数组。

gcnew 为 irregAry 分配了 3 个元素，每个元素是指向一维数组的句柄。要使用数组，还要用 gcnew 每行元素分配内存，例如：

```
irregAry[0]=gcnew array<int> (4);
irregAry[1]=gcnew array<int> (2);
irregAry[2]=gcnew array<int> (3);
```

现在，irregAry 的每个元素都已经分配了内存，可以存放整型数据了。

另外，可以使用初始化值表的形式来声明一个不规则数组。例如：

```
array< array<int>^ > ^irregAry=
{     gcnew array<int> { 1, 2, 3, 4 },
```

```
        gcnew array<int> { 5, 6 },
        gcnew array<int> { 7, 8, 9 }
    };
```

不规则数组元素也可以用下标方式访问，但行、列的索引值分别置于两个方括号内。例如：

```
    irregAry[1][2]=irregAry[1][0] + 10;
```

4．边界检查

托管数组具有边界检查功能。当程序试图访问一个数组定义界外的元素时，将出现运行异常，防止了访问内存的未知副作用。例如，有以下代码：

```
    using namespace System;
    int main()
    {
        array< int >^ myArray=gcnew array<int>(3);
        myArray[4]=999;              //错误访问数组范围之外的元素
    }
```

可以编译通过，但运行后将出现运行错误对话框，进入调试将出现异常信息框。

5．托管数组作为函数参数

托管数组元素作为函数参数的情形与本地数组的一样，可以按值传递，也可以用引用类型传递。

函数参数是托管数组名时，总是以引用方式传递。托管数组传递的优点是不用传递数组长度，其原因是，System::Array 类已经定义了数组的长度属性 Length，并且具有界限检查功能。

值得注意的是，当一个函数的形式参数是托管数组时，它不能像本地数组那样声明为 const。

【例 1-12】　显示二维矩阵数组和不规则数组元素。程序运行结果如图 1.8 所示。

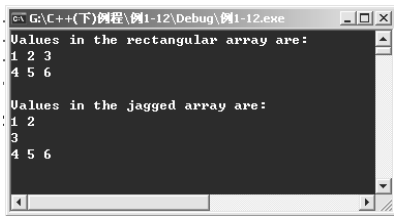

图 1.8　显示数组元素

```
    using namespace System;
    void outputArray(array<int, 2>^);     //显示二维矩阵数组
    void outputArray(array<array<int>^>^);
                                          //显示二维不规则数组
    int main()
    {
        array<int,2> ^aryOne={{1,2,3},{4,5,6}};
                                          //声明二维矩阵数组
        array<array<int>^>^aryTwo=   //声明二维不规则数组
        {
            gcnew array<int> { 1,2 },
            gcnew array<int> { 3 },
            gcnew array<int> { 4, 5, 6 }
        };
        outputArray(aryOne);
        Console::WriteLine();
        outputArray(aryTwo);
        Console::Read();
    }
    void outputArray(array<int,2>^ arr)
    {
        Console::WriteLine("Values in the rectangular array are:");
        for(int row=0; row<arr->GetLength(0);row++)
        {
            for(int colum=0; colum<arr->GetLength(1); colum++)
                Console::Write("{0}",arr[row,colum]);
            Console::WriteLine();
        }
    }
    void outputArray(array< array<int>^>^ arr)
    {
        Console::WriteLine("Values in the jagged array are:");
```

```
        for(int row=0; row<arr->Length; row++)
        {
            for(int colum=0; colum<arr[row] ->Length; colum++)
                Console::Write("{0}",arr[row][colum]);
            Console::WriteLine();
        }
    }
```

System::Array 类中，Length 属性为数组的长度，它表示 gcnew 分配的元素数量。由于矩阵数组存放在连续的存储区间中，因此 Length 属性值表示全部元素的总数量。例如：

```
aryOne->Length
```

的值等于 6。在函数 void outputArray(array<int, 2>^)中，嵌套循环为了分别获取数组行、列的长度值，使用了 GetLength 方法：

```
arr->GetLength(0)          //获取行（0 维）长度
arr->GetLength(1)          //获取列（1 维）长度
```

而对于不规则数组，每行元素是分别分配内存的，可以通过 Length 属性得到行、列的长度。在函数 outputArray(array< array<int>^>^ arr)中：

```
arr->Length                //获取行的长度
arr[row] ->Length          //获取当前行的列长度
```

System::Array 类还有其他的属性，以及复制、查找、排序、逆置等方法，这些属性、方法都适用于 array 数组，读者可以参考 MSDN 相关内容。

1.4 句柄

C++的指针有很强大的功能。但本地指针不能应用于托管对象，原因是，托管对象存储在托管内存中，受到垃圾收集器的支配。垃圾收集器的主要功能有两个：一是观察内存中的对象，自动回收垃圾对象占有的内存；二是高效地整理内存，使其形成连续的内存区域。

本地指针手工使用 new 分配内存和 delete 释放内存，一旦内存被整理，指针所指对象的数据就遭到破坏。因此，托管代码需要一种可以与垃圾收集器同步的“跟踪指针”，这就是句柄。如果句柄所指向的数据被垃圾收集器移动，句柄会自动更新。句柄用符号“^”表示（发音为“hat”）。

1.4.1 句柄操作

句柄是指向托管内存对象的引用。句柄使用运算符 gcnew 申请动态内存，没有指向的句柄用空值 nullptr 表示。访问句柄所指的对象，使用“*”和“->”运算符。

例如：

```
int ^ ptr=nullptr;
ptr=gcnew int;
*ptr=10;                   //访问 ptr 所指对象
```

又如：

```
Pen^ bluePen=gcnew Pen(Brushes::Blue);
bluePen->Width=8.0F;       //访问 bluePen 所指对象的成员
```

这些操作与本地 C++很相似，但句柄的操作有以下限制。

① 不能用句柄直接创建动态数组。若有

```
ptr=gcnew int[10];         //已有声明 int^ptr
ptr[i]=x;
```

将导致编译错误。建立托管数组要使用 1.3.5 节所介绍的方法，建立 array 类的实例。

② 不能对句柄执行指针运算。例如，++、--、逻辑（如==）运算，对句柄都是没有意义的。

③ 不能创建指向函数的句柄。

④ 不能创建指向句柄的句柄。

⑤ 句柄是类型安全的，因此不能像 void *指针那样建立 void^句柄。

⑥ 在 C++的指针操作中，可以用“&”运算符取操作数的内存地址。句柄为托管类型提供一个类似的运算符“%”，返回托管类对象的句柄。“%”不能返回本地内存中的对象句柄。例如，int 变量不是托管类对象，以下代码将导致编译错误：

```
int ^intPtr=nullptr;
int x=10;
intPtr=%x;              //错误
```

在托管代码中，显示一个句柄的值，结果是它所指向对象的值。

【例 1-13】 建立一个 System::DateTime 的对象和指向它的句柄，用不同形式输出同一个数据。程序运行结果如图 1.9 所示。

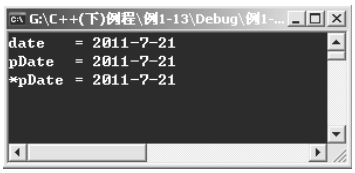

图 1.9 用不同的形式输出同一个数据

```
using namespace System;
int main()
{
    DateTime date=DateTime(2011,7,21);
    DateTime ^pDate=nullptr;
    pDate=%date;
    Console::WriteLine("date={0:d}\npDate={1:d}\n*pDate={2:d}",date,pDate,*pDate);
    Console::Read();
}
```

“%”运算符除了可以用于 System 提供的类构造的对象外，也可以用于用户定义的托管类对象。托管类定义见 1.5 节。

【例 1-14】 本例声明了一个托管类 test 和 test 类句柄 pTest，句柄指向 test 类对象 programTest。程序通过对象和句柄调用成员函数输出对象的值，运行结果如图 1.10 所示。

图 1.10 输出对象的值

```
using namespace System;
ref class test              //声明托管类
{
    public:
        test() {}
        test(String^ sub, int score)
        {
            Subject=sub; testScore=score;
        }
        void printTestScore()
        {
            Console::WriteLine("Subject: {0}\tTest score:{1}",Subject,testScore);
        }
        property String^ Subject;
        property int testScore;
};
int main()
{
    DateTime testDay=DateTime(2011, 6, 15);          //声明 System::DateTime 对象
    test programTest("program", 85);                 //声明托管类对象
    DateTime ^pDate=%testDay;                        //声明 DateTime 类句柄，指向对象 testDay
    test ^pTest=%programTest;                        //声明 test 类句柄，指向对象 programTest
    Console::WriteLine("Output 1");
    Console::WriteLine("Test date: {0:d}", testDay); //输出对象数据
    programTest.printTestScore();                    //调用对象成员函数
    Console::WriteLine("Output 2");
    Console::WriteLine("Test date: {0:d}", *pDate);  //输出句柄所指对象数据
    pTest->printTestScore();                         //调用句柄所指对象成员函数
    Console::Read();
}
```