第一章　面向对象编程

C++ Builder 3 的最大特点就是面向对象，在面向对象领域，Borland®一直处于世界领先的地位。C++ Builder 3 的面向对象主要体现在三个方面，一是编程语言最接近 ISO C++ 标准，二是全面支持基于元件的应用程序开发，三是借助于对象库可实现代码重用。

本章从一些基本概念开始，详细介绍 C++ Builder 3 面向对象的编程思想，并初步介绍 VCL 的结构。C++ Builder 3 中的元件很多，尽管这些元件千差万别，但它们都是从几个公共基类继承下来的，因此，这些元件具有某种程度的相似性，为了节省篇幅，本书按照这样的思路编写，即先介绍公共基类，后面将只介绍每个元件特有的特性和方法。

1.1　什么是对象

什么是对象？这得先从什么是类说起。类(Class)是一种数据类型，与一般的数据类型不同的是，类除了包含数据以外，还包含了操纵数据的方法，类把数据和方法封装在一起。在一个类中，可以包含不同类型的数据，这一点与 C++的结构相似。

类具有可继承性，如果要创建一个新的类，只要选择一个基类再加上自己的成员就派生出一个新的类。事实上，C++ Builder 3 中所有的类都是从一个共同的基类继承下来的，基类的非私有成员自动成为派生类的成员。类的继承还具有传递性，例如，假设 T3 继承了 T2，而 T2 又继承了 T1，可以认为 T3 也继承了 T1。

在 C++ Builder 3 中，所有的类都是从 TObject 继承下来的，TObject 是一个抽象类，它的派生类可以对 TObject 的方法包括构造和析构重载，TObject 也被称为默认祖先类。

类只是一种数据类型，要使用类，一般还得声明类的变量(某些特殊的情况下可以直接对类进行操作)。需要指出的是，类的变量和类的实例从严格意义上讲，不完全是一个概念。类的实例(Instance)是一块动态分配的内存区域，一般我们把类的实例称为对象(Object)，同一个类，可以有多个实例存在，每个对象都有自己的数据，但方法是共享的。

类的变量实际上是一个指针，指向类的实例，要访问对象的成员，就是通过类的变量来引用的。同一个类可以有多个变量，多个变量可以引用同一个对象。

下面我们还是通过一个典型的程序示例把类和对象的概念讲清楚，当您使用“File”菜单上的“New Application”命令时，C++ Builder 3 将创建一个新的项目，默认情况下，这个项目中只有一个空白的 Form 和一个单元文件及头文件。

在 C++ Builder 3 中，Form 就是一个非常典型的对象，Form 的直接基类是 TForm，我们先看看头文件中是怎样声明 Form 的：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
```

```
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
    _published:
    private:
    public:
        _fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

可以看出，头文件中声明了一个类叫 TForm1，它的基类是 TForm。由于 Form 现在是空白的，也没有建立任何事件句柄，因此，TForm1 中除了构造外没有其它数据和方法。

如果您对类的概念理解得比较透彻的话，您就知道，TForm1 中实际上是有成员的，因为 TForm1 是从 TForm 继承下来的，TForm 的成员也就成为 TForm1 的成员。

声明了 TForm1 后，头文件中声明了 TForm1 类型的变量 Form1，以后就是用 Form1 来引用这个 Form。注意：在调用 TForm1 的构造之前，也就是说创建 TForm1 的实例之前，Form1 这个变量基本上是没意义的。

我们再看看单元文件的代码是怎样的：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
_fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

```
//---------------------------------------------------------------------------
```

由于现在 Form 是空白的，单元文件中只有 TForm1 的构造函数，您可以在构造函数中加入任何代码，只要不引用 Form 本身，因为在构造函数中，类的实例还没有创建好。

假设您向 Form 中加入一个按钮(TButton)，名称是 Button1，双击此按钮，C++ Builder 3 就自动建立一个处理 OnClick 事件的句柄。这时候，头文件就变成这样：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
    _published:
        TButton *Button1;
        void _fastcall Button1Click(TObject *Sender);
    private:
    public:
        _fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

可以看出，头文件在 TForm1 的_published 部分声明了一个 TButton 类型的变量叫 Button1，同时声明了一个无返回的函数 Button1Click 作为处理 OnClick 事件的句柄。

我们再看看单元文件的代码有什么变化：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
```

```
#pragma resource "*.dfm"
TForm1 *Form1;

//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{

}

//---------------------------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{

}
//---------------------------------------------------------------------------
```

可以看出，单元文件唯一的变化就是给出了事件句柄的实体，目前实体中没有代码，您可以加入任何代码，包括对 Form 的引用。程序示例如下：

```
void__fastcall TForm1::Button1Click(TObject *Sender)
{
    Form1->Color = clGreen;
}
```

1.2　修改元件的名称

一般来说，您最好在设计期用 Object Inspector 修改元件的名称，而且最好在您刚刚把元件加到 Form 上时就改。在设计期修改元件的名称有一个好处，C++ Builder 3 会自动更新源代码中所有引用元件名的地方，也就是说，您不用担心是否有不一致的地方。例如，您可以把 Form 的名称从 Form1 改为 MainForm，头文件会相应变化：

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
```

```
//---------------------------------------------------------------------------
class TMainForm : public TForm
{
    _published:
        TButton *Button1;
        void _fastcall Button1Click(TObject *Sender);
    private:
    public:
        _fastcall TMainForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TMainForm *MainForm;
//---------------------------------------------------------------------------
#endif
```

可以看出，所有原先是 Form1 的地方现在都改为 MainForm，包括类名。

完全可以预料，单元文件的代码也改过来了：

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TMainForm *MainForm;
//---------------------------------------------------------------------------
_fastcall TMainForm::TMainForm(TComponent* Owner)
    : TForm(Owner)
{

}
//---------------------------------------------------------------------------

void _fastcall TMainForm::Button1Click(TObject *Sender)
{
    Form1->Color = clGreen;
}
//---------------------------------------------------------------------------
```

要注意的是，如果您在事件句柄 Button1Click 中写了代码，而且还用老名字引用了 Form，当您修改 Form 的名称时，这部分代码不会自动改变，因为代码是您自己键入的。这时候，您就要手动更改对 Form 的引用，否则编译不能通过。

1.3　对象的作用域问题

在 C++ Builder 3 中要特别注意对象的作用域问题，因为这关系到对象的成员是否可见和可访问。总的原则是，在类的方法中，类的成员都是可见的，例如，假设 Form(类名叫 TForm1)上有一个按钮 Button1，由于 C++ Builder 3 把处理 Form 及 Form 上元件事件的句柄作为 TForm1 的一个方法，因此，在处理按钮的 OnClick 事件的句柄中，您可以任意访问 TForm1 的所有成员，而且不需要用 TForm1 的变量来引用，例如：

```
_fastcall TForm1::Button1Click(TObject *Sender);
{
    Color = clFuchsia;
    Button1->Color = clLime;
}
```

由于 Color 是 TForm 的成员，也就成为 TForm1 的成员，因此，您可以直接对 Color 特性赋值，而不需要这么写(尽管这样写编译器也不认为出错)：

Form1->Color = clFuchsia;

但是，如果要访问 Form 上某个元件的成员，例如 Button1 的 Color 特性，您就必须加上对象限定符，否则编译器就认为您要访问的是 TForm1 的 Color 特性。

也许您要问，既然访问按钮的 Color 特性时要加上 Button1 限定符，为什么不在 Button1 前加 Form1 限定符呢？这是因为，Button1 被声明为 TForm1 的一个数据成员，在类的方法中，类的成员总是可见的、可访问的，因此，不需要加 Form1 限定符。

假设当前项目中有两个 Form，分别是 Form1 和 Form2，如果要在 Form1 中访问 Form2 上的 Button2 元件，该怎样访问呢？

由于 Button2 已超出 Form1 的作用域，因此，在访问 Button2 时您必须加上 Form2 限定符，程序示例如下：

Form2->Button2->Color = clLime;

如果您以为这样就可以了，那就错了，因为 Form1 根本就不知道 Form2 是什么东西，您还需要把 Form2 的头文件包含到 Form1 的单元文件中。

1.4　类成员的可见性

面向对象编程的重要特征之一就是隐藏复杂性，C++ Builder 3 通过_published、private、public、protected 和 automated 等几个关键字使类成员具有不同的可见性，例如：

```
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
    _published:
        TButton *Button1;
        void _fastcall Button1Click(TObject *Sender);
    private:
    public:
        _fastcall TForm1(TComponent* Owner);
};
```

上例中，数据 Button1 和方法 Button1Click 是在_published 部分声明的，而 TForm1 的构造函数是在 public 部分声明的，那么这几个关键字究竟是什么含义呢？

在 public 部分声明的成员是公共的，也就是说，它们虽然是在某个类中声明的，但其它类的实例也可以访问。例如，假设应用程序由两个 Form 构成，相应的单元是 Unit1 和 Unit2，您希望 Unit2 能共享 Unit1 中的整型变量 Count，您可以把 Count 在 TForm1 类中的 public 部分声明，然后把 Unit1 的头文件包含到 Unit2 中。

不过，除非您必须把某个成员在不同类之间共享，一般来说尽量不要把成员声明在类的 public 部分，以防止程序意外地不正确地修改了数据。

在 private 部分声明的成员是私有的，它们只能被同一个类中的方法访问，就好象局部变量一样，对于其它类包括它的派生类，private 部分声明的成员是不可见的，这就是面向对象编程中的数据保护机制，程序员不必知道类实现的细节，只需关心类的接口。

protected 与 private 有些类似，在 protected 部分声明的成员是私有的，不同的是，在 protected 部分声明的成员在它的派生类中是可见的，并且成为派生类的私有成员。

在 protected 部分声明的成员通常是方法，这样既可以在派生类中访问这些方法，又不必知道方法实现的细节。

在_published 部分声明的成员，其可见性与在 public 部分声明的成员的可见性是一样的，它们都是公共的，不同的是，_published 主要用于声明 Form 上的元件和事件句柄，它是由 IDE 自动维护的。

在 automated 部分声明的成员，其可见性与在 public 部分声明的成员的可见性是一样的，它们都是公共的，唯一的区别在于，在 automated 部分声明的方法和特性将生成 OLE 自动化的类型信息。不过，在 C++ Builder 3 中，automated 的作用不大。

1.5 对象的相互赋值

尽管类是一种复杂的数据类型，既有数据又有方法，但是，您完全可以象对一般的变量那样，用赋值号把类的变量赋给另一个类的变量，只要这两个类的类型是一致的或相容的，例如，假设 Form1 和 Form2 都是 TForm 的变量，您可以这么写：

```
Form2 = Form1;
```

因为 Form1 和 Form2 的类型都是 TForm。

您还可以把派生类的变量赋给基类的变量，例如，假设 TMyForm 是这样声明的：

```
class TMyForm : public TForm
{
    _published:
        TButton *Button1;
        TEdit *Edit1;
        TDBGrid *DBGrid1;
        TDatabase *Database1;
    private:
    public:
        virtual _fastcall TMyForm(TComponent* Owner);
};
```

然后分别声明了 TMyForm 和 TForm 的变量：

```
TMyForm *MyForm;
TForm *AForm;
```

由于 TMyForm 是从 TForm 继承下来的，因此您可以这样赋值：

```
AForm = MyForm;
```

不知您注意到没有，每一个事件句柄中都有一个 Sender 参数，它的类型是 TObject，大家知道，VCL 的所有对象都是从 TObject 继承下来的，因此，VCL 的所有对象都能赋值给 Sender 参数，这就是为什么 Sender 参数能代表所有 VCL 对象的原因。

Sender 参数主要用于在运行期判断元件的类型，程序示例如下：

```
if (typeid(*Sender) = = typeid(TEdit))
    DoSomething;
else
    DoSomethingElse;
```

1.6　自己创建一个对象

C++Builder 3 已经在 VCL 中声明了众多的对象，不过，有时候您可能需要自己声明一个类，然后再创建类的实例即对象，因为 VCL 中的对象未必能完全满足您的编程需要。例如，您可以声明一个 TEmployee 类：

```
class TEmployee : public TObject
{
    private:
```

```
        char Name[25];
        char Title[25];
        double HourlyRate;
    public:
        double CalculatePayAmount( );
        TEmployee( );
};
```

类的声明一般放在头文件中，然后您得在要用到这个类的单元文件中声明类的变量：

```
TEmployee *Employee;
```

TEmployee 只是一种数据类型，要使用这个类，您必须创建类的实例：

```
Employee = new TEmployee;
```

new 会自动调用 TEmployee 的构造函数，现在您就可以访问 TEmployee 中的成员了。

前面讲过，对象实际上是一块内存区域，换句话说就是 Windows 的资源。如果您不再需要用到 TEmployee 的对象，您应当及时地删除它，以释放它所占用的资源。

要删除 TEmployee 的对象，您可以调用 delete：

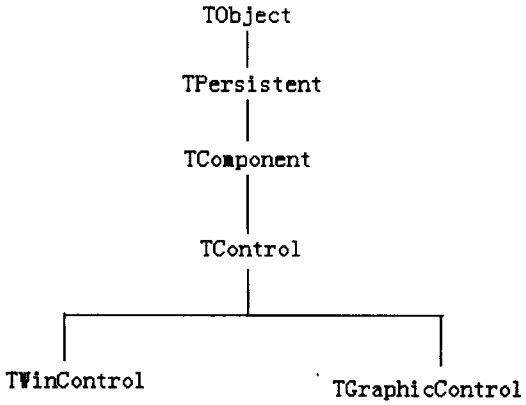
```
delete Employee;
```

不过，您要考虑到这样一种情况，就是如果程序出现异常，程序将非正常退出，这种情况下，程序可能执行不到删除对象的地方。为了保证资源能可靠地得到释放，您就要用到 C++的异常处理机制，把删除对象的代码放到 try..catch 块中。

要说明的是，当您把元件选项板上的元件加到 Form 上时，C++ Builder 3 会自动创建元件的实例，当程序正常退出的时候，C++ Builder 3 会自动删除元件的实例，这一切都用不着您自己编写代码，除非您是自己在运行期动态地把一个元件加到 Form 上。

1.7 VCL 的结构

VCL(Visual Component Library 的缩写)是 C++ Builder 3 的核心，VCL 是完全面向对象的，VCL 中的所有对象都存在着继承与被继承的关系，下图是 VCL 的示意：



从示意图可以看出，最顶层的是 TObject，它是一切对象的基类。

TPersistent 是 TObject 的下一级继承者，它是一个抽象类，主要用于提供对象之间的相互赋值和读写流的能力。

TComponent 又是 TPersistent 的继承者，这个类是 VCL 中所有元件的祖先类，TComponent 定义了所有元件最基本的行为。

尽管所有元件都是从 TComponent 继承下来的，但直接继承的只有几个非可视的元件如 TTimer 和 TDataSource 等，其它元件则是从 TComponent 的下级 TControl 继承下来的，从 TControl 继承下来的元件是可视元件，可视元件也称为控件。TControl 定义了所有控件的基本属性，包括特性、方法和事件等。

TWinControl 和 TGraphicControl 这两个类都是从 TControl 继承下来的，这两个类的区别正如它们取的名字一样，从 TWinControl 继承的主要是按钮、对话框、列表框等有窗口句柄的控件，这些控件占用 Windows 的资源，并且允许用户输入。而从 TGraphicControl 继承的元件例如 TLabel、TSpeedButton 等，没有窗口句柄，不占用 Windows 资源，也不能接受键盘的输入，使用这一类元件的好处在于节约资源。

VCL 的面向对象还体现在它的可扩展性，您可以选择其中一个元件作为基类，派生出一个新的类(元件)，并把新创建的元件加入到 VCL 中。

1. TObject

TObject 是 VCL 中所有对象的基类，它定义了操纵对象的基本方法，其中，有的是类方法，用于返回对象的类型信息，有的是虚拟方法，能够在派生类中重载，有的仅用于 C++ Builder 3 的 IDE 自身调用，而不能被程序调用。TObject 没有数据成员。

请读者注意本书的编写习惯，在介绍一个类时，先介绍类的继承关系，再按字母顺序介绍类的特性、方法和事件，可能会作一些取舍。

AfterConstruction 函数

声明：virtual void _fastcall AfterConstruction( );

应用程序不要直接调用这个函数，因为这个函数是当类的构造函数执行完毕时自动调用的，但您可以重载这个虚拟函数。

BeforeDestruction

声明：virtual void _fastcall BeforeDestruction( );

应用程序不要直接调用这个函数，因为这个函数是在类的构造函数马上就要执行时自动调用的，但您可以重载这个虚拟函数。

ClassName 函数

声明：static ShortString _fastcall ClassName(TClass cls);

ShortString _fastcall ClassName( ){ return ClassName(ClassType( ));}

这个函数返回一个对象实例的[redacted]。注意：您可以用基类的变量来引用派生类的对象实例，用此变量来调用 ClassName 时，返回的是派生类的类名，而不是变量本身的类型。

假设 Form 上有一个按钮(TButton)、标签(TLabel)、复选框(TCheckBox)，当用户单击其中一个任何控件，要使控件的类名显示在 Form 的标题栏上，您当然应当首先为这些控件建立处理 OnClick 事件的句柄，然后这样写代码：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
    Form1->Caption = String(Button1->ClassName( ));
}

void _fastcall TForm1::CheckBox1Click(TObject *Sender)
{
    Form1->Caption = String(CheckBox1->ClassName( ));
}

void _fastcall TForm1::Label1Click(TObject *Sender)
{
    Form1->Caption = String(Label1->ClassName( ));
}
```

ClassNameIs 函数

声明：static bool _fastcall ClassNameIs(TClass cls, const AnsiString string);

这个函数用于判断对象的类型是否是指定的类型，如是，就返回 True。

DefaultHandler 函数

声明：virtual void _fastcall DefaultHandler(void* Message);

如果一个对象处理某个消息，DefaultHandler 能提供对该消息的默认处理。派生类可以重载这个方法，例如，TWinControl 就重载了 DefaultHandler，并且换名为 DefWindowProc。

Dispatch 函数

声明：virtual void _fastcall Dispatch(void *Message);

这个函数用于调用对象的消息处理方法，究竟调用哪个方法，由 Message 参数指定的消息来决定。如果对象中没有找到处理该消息的方法，Dispatch 就从基类中查找，如果一直到最顶层还没有找到，就调用 DefaultHandler。

Message 是个无类型的参数，从语法上讲，您可以传递任何类型的数据，不过，最好是 TMessage 结构，它是这样声明的：

```
struct TMessage
{
    Cardinal Msg;
```

```
    union
    {
        struct
        {
            Word WParamLo;
            Word WParamHi;
            Word LParamLo;
            Word LParamHi;
            Word ResultLo;
            Word ResultHi;
        };
        struct
        {
            int WParam;
            int LParam;
            int Result;
        };
    };
};
```

FreeInstance 函数

声明：virtual void _fastcall FreeInstance( );

这个函数是由析构函数自动调用的，您一般不要直接调用它，除非派生类重载了NewInstance，您也要相应地重载 FreeInstance。

FreeInstance 是与 NewInstance 配对使用的，NewInstance 用于建立一个新的对象实例，而 FreeInstance 用于删除 NewInstance 建立的对象实例。

InheritsFrom 函数

声明：bool _fastcall InheritsFrom(TClass aClass) {return InheritsFrom (ClassType ( ), aClass);}

这个函数用于判断两个对象的继承关系，如果 aClass 参数指定的类是对象的基类，这个函数就返回 True，否则就返回 False。注意这个函数与 ClassNameIs 函数的区别，ClassNameIs 要求类名精确匹配，而 InheritsFrom 函数只要求是继承关系。

NewInstance 函数

声明：virtual TObject* _fastcall NewInstance(TClass cls);

这个函数用于为对象实例分配内存并返回对象实例的指针。注意：类的构造函数会自动调用 NewInstance，应用程序不能直接调用这个函数，但派生类可以重载 NewInstance，相应地，派生类也必须重载 FreeInstance，因为这两个函数是配对使用的。

SafeCallException 函数

声明：virtual HResult _fastcall SafeCallException(TObject *ExceptObject, void *ExceptAddr);

实现 COM 和 OLE 接口的类应当重载这个函数以处理异常。

1.9 TPersistent

TPersistent 提供了对象之间相互赋值和读写流的能力。TPersistent 是一个抽象类，不要直接创建 TPersistent 的对象实例。TPersistent 的大部分方法是从 TObject 继承的，因此，这里只介绍几个 TPersistent 特有的方法。

Assign 函数

声明：virtual void _fastcall Assign(TPersistent* Source);

这个函数用于把 Source 参数指定的对象的数据复制给自己，调用的形式是这样的：

Destination->Assign(Source);

Source 参数的类型是 TPersistent，因此，只要是 TPersistent 的派生类都可以相互赋值，只有一些直接从 TObject 继承下来的类如 TComObject、TAutoObject 等才是不可赋值的。

要注意的是，对象之间的相互赋值是有前提的，那就是两个对象必须是赋值相容的，两个对象的类型要么完全一致，要么具有继承关系。

您也可以用赋值语句在对象之间相互赋值，赋值形式是这样的：

Destination = Source;

但要注意，这与调用 Assign 函数含义不同，赋值语句的含义是用目标变量引用源变量所引用的对象实例，而调用 Assign 则是把源对象的数据复制给目标对象。

AssignTo 函数

声明：virtual void _fastcall AssignTo(TPersistent* Dest);

这个函数与 Assign 恰好相反，用于把自己的数据复制给 Dest 参数指定的对象。

DefineProperties 函数

声明：virtual void _fastcall DefineProperties(TFiler* Filer);

这是个虚拟函数，TPersistent 的派生类重载它是为了把对象的非公开数据写到流例如 Form 文件中，Filer 参数指定要写的流。

1.10 TComponent

TComponent 是 C++ Builder 3 中所有元件的抽象基类，它提供了元件的基本特征：元件可以出现在元件选项板上，可以被加到 Form 上，可以拥有和管理其它元件。

TComponent 是个抽象类，不要直接创建它的实例。尽管 C++ Builder 3 的所有元件都是从 TComponent 继承下来的，但直接继承于 TComponent 的只是少数几个非可视的元件，大部分元件是从 TComponent 的派生类 TControl 继承下来的。

ComObject 特性

声明：_property _di_IUnknown ComObject;

这个只读的特性返回支持 COM 的 VCL 元件的接口引用。如果 VCL 元件不是作为一个 COM 组件的外套，访问这个特性将导致 EComponentError 异常。

ComponentCount 特性

声明：_property int ComponentCount;

这个只读的特性返回元件拥有的子元件的数目，例如，对于 TForm 元件来说，它的 ComponentCount 特性就是 Form 上元件的个数。

元件拥有的子元件放在 Components 特性中，这是个由所有子元件组成的数组，其索引从 0 开始，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
    AnsiString nameString("TButton");
    char compBuf[10];
    TButton * button;
    for(int i=0; i < ComponentCount; i++)
    {
        if (Components[i]->ClassNameIs(nameString))
        {
            //cast the component to a TButton *
            button = (TButton *)Components[i];
            button->Font->Name = "Courier";
            itoa(ComponentCount, compBuf, 10);
            Edit1->Text = AnsiString(compBuf) + AnsiString(" components");
        }
    }
}
```

ComponentIndex 特性

声明：_property int ComponentIndex;

这个只读的特性返回某个子元件在元件数组中的索引，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
    char index[10];
```

```
    char *string = new char[strlen("The index of the button is ")];
    itoa(Button1->ComponentIndex, index, 10);
    strcpy(string, "The index of the button is ");
    strcat(string, index);
    Edit1->Text = string;
    delete string;
}
```

上例把 Form1 上的 Button1 元件的索引值转换成字符串后显示在编辑框 Edit1 中。

Components 特性

声明：_property TComponent* Components[int Index];

这个只读的特性就是由元件所拥有的所有子元件组成的数组，程序示例请参照前面。

ComponentState 特性

声明：_property TComponentState ComponentState;

这个只读的特性返回元件当前的状态。TComponentState 是个集合类型，包含下列元素：csLoading, csReading, csWriting, csDestroying, csDesigning,csAncestor, csUpdating, csFixups。

ComponentStyle 特性

声明：_property TComponentStyle ComponentStyle;

这个只读的特性返回元件的风格，如果返回 csInheritable，表示这个元件是可继承的。

Name 特性

声明：Property Name: TComponentName;

这个特性就是元件的名称。当您把一个元件放到 Form 上时，C++ Builder 3 给这个元件一个默认的名字，如 Button1、Edit1 等，当然您可以另取一个名字。在同一个项目中，元件的名字不能重名，如果不是很有必要，不要在运行期修改元件的名字。

Owner 特性

声明：_property TComponent* Owner;

这个只读的特性返回元件的拥有者。当拥有者被删除时，所有被拥有的元件也将被删除。例如，当 Form 被删除时，Form 上的元件都将被删除。

Tag 特性

声明：_property int Tag;

这个特性通常用于给一组元件加上相异的序号，这样您就可以在运行期识别它们。例如，假设 Form 上有十个按钮，它们有一个共同的事件句柄，为了知道是哪个按钮被按下，您可以事先给这十个按钮编号，然后用 Tag 特性判断是哪个按钮被按下，程序示例如下：

```
void _fastcall TForm1::Button1Click(TObject *Sender)
{
    if (typeid(*Sender) = = typeid(TButton))
        if (((TButton *)Sender)->Tag = = 1)
            ((TButton *)Sender)->Caption = "My tag is 1";
}
```

TComponent 构造函数

声明：_fastcall virtual TComponent(TComponent* AOwner);

这个构造函数做了这么几件事情：创建和初始化元件的实例，指定元件的拥有者(由 AOwner 参数指定)，把 ComponentStyle 特性设为 csInheritable 表示此元件是可继承的。

对于元件来说，一般不需要显式地调用这个构造函数，因为当您把一个元件放到 Form 上时，元件的实例是自动创建的。不过，TComponent 的派生类可以重载这个虚拟函数。

~TComponent 析构函数

声明：_fastcall virtual ~TComponent(void);

析构函数用于删除元件的实例，包括该元件所拥有的元件。不过，一般不需要显式地删除元件的实例，因为当应用程序终止时，会自动删除应用程序所拥有的 Form，进而删除 Form 上的元件。当然，如果元件是在运行期动态引入的，您得用 delete 关键字来删除元件的实例。对于 Form 来说。最好调用 release 来释放它的实例。注意：千万不要在元件的事件句柄中删除元件自己或元件的拥有者，否则会导致意想不到的结果。

FindComponent 函数

声明：TComponent* _fastcall FindComponent(const System:: AnsiString AName);

这个函数从元件所拥有的元件(实际上就是从 Components 数组)中查找一个元件，其名称是 AName 参数(大小写不敏感)。如找到，就返回这个元件。

FreeNotification 函数

声明：void _fastcall FreeNotification(TComponent* AComponent);

这个函数通知 AComponent 参数指定的元件：本元件将要被删除了。例如，元件 A 被赋值给元件 B 的某特性，当元件 A 将要被删除时，就要通知元件 B。不过，如果元件 A 和元件 B 位于同一个 Form 内，通知是自动进行的。如果两个元件不在同一个 Form 内，并且存在依赖关系，才需要调用 FreeNotification。

GetParentComponent 函数

声明：DYNAMIC TComponent* _fastcall GetParentComponent(void);

这是个动态方法，派生类重载它是为了返回一个特定的“父”类型。

HasParent 函数

声明：DYNAMIC bool _fastcall HasParent(void);

这个函数判断元件是否有“父”，“父”通常是元件所在的容器如 Form、Panel 等。

InsertComponent 函数

声明：void _fastcall InsertComponent(TComponent* AComponent);

这个函数向元件的 Components 数组中增加一个 AComponent 参数指定的元件，这个元件要么没有名字，要么与 Components 数组中现有的元件相异。程序示例如下：

```
void MoveToModule(TForm*Source, TDataModule *Dest)
{
    int I;
    TComponent*Temp;
    for (I = Source->ComponentCount - 1; I >= 0; I--)
    {
        Temp = Source->Components[I];
        if (dynamic_cast<TControl *>(Temp) == NULL)
        {
            Source->RemoveComponent(Temp);
            Dest->InsertComponent(Temp);
        }
    }
}
```

MoveToModule 函数的作用就是把 Form 上所有非可视的元件移到数据模块上，函数通过一个类型强制转换来判断元件是否属于可视的控件，如不是，就调用 RemoveComponent 从 Form 上删除这个控件，再调用 InsertComponent 把控件插入到数据模块上。

RemoveComponent 函数

声明：void _fastcall RemoveComponent(TComponent* AComponent);

这个函数从元件的 Components 数组中删除 AComponent 参数指定的元件。注意：在设计期，无论您把一个元件加到 Form 上，还是从 Form 上移走一个元件，InsertComponent 或 RemoveComponent 都是自动调用的。

1.11 TControl

TControl 是从 TComponent 继承下来的，作为 C++ Builder 3 中所有控件的基类。所谓控件，是指那些在运行期可见的元件，例如，TButton 元件在运行期就是个按钮，TEdit 元