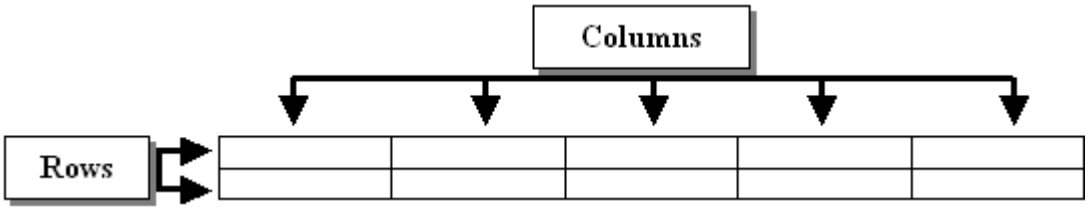
The String Grid Control

Overview of Grids

A grid is a technique of using columns and rows to represent data in a visual format. There is no strict rule as to what a grid control is used for. It can be used to simply display a series of values by categories. It can also be used as a time sheet, which will be the basis of the next exercise. Sometimes it is used as a calendar, similar to the MonthCalendar control and as we will see later on.

To organize its content, a grid is made of vertical and horizontal lines. These lines are used as separators. They create vertical entities called columns and horizontal sections called rows:



The intersection of a column and a row is called a cell. The cell is the most important entity and the most used aspect of a grid. It holds the actual values of the grid. The cells of a grid can be used to display data or they can be used to receive data from the user. This means that data of a grid is entered or stored in cells.

Because the role of a grid is unpredictable, the most top cell of each column can be used to display a label. By nature, a column specifies a category of value. Therefore, the label of a column signifies the category of values of that column:

Date	Time	Holder	Flavor	Scoops	Ingredient
6/21/2003	10:12 AM	Cup	Vanilla	1	None
6/21/2003	10:32 AM	Cup	Chololat	1	Peanuts
6/22/2003	3:22 PM	Cone	Cocoa Crea	2	None
6/22/2003	12:54 PM	Bowl	Chololat	1	M & M
6/22/2003	12:58 PM	Cup	Chololat	2	Peanuts
6/22/2003	1:14 PM	Cup	Cocoa Crea	1	None
6/22/2003	1:20 PM	Cone	Vanilla	3	M & M
6/22/2003	1:26 PM	Bowl	Vanilla	3	None
6/22/2003	2:12 PM	Cone	Vanilla	2	Peanuts
6/22/2003	2:22 PM	Cone	Chololat	2	M & M

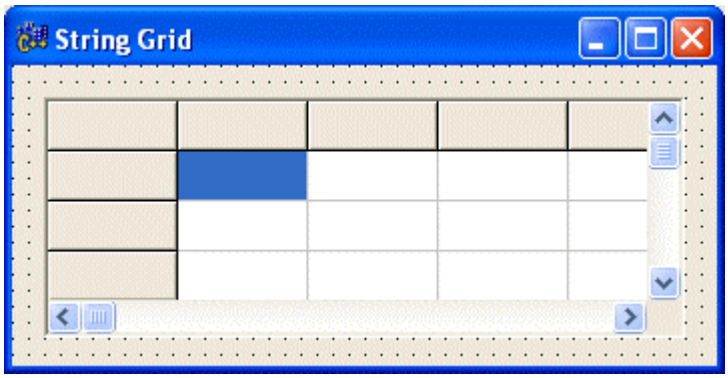
To create a series of values for each category, you use a row of data. A row is also called a record. To make a record explicit, the most left row can display a label. The easiest and most basic label consists of a number. In this case, rows can be labeled from top to bottom as 1, 2, 3, 4, etc.

In most cases, each cell is in fact an Edit control and its content is an **AnsiString**. This means that a cell can contain a natural number, a floating-point variable, or a string.

String Grid Properties

To create a grid of data, the Visual Component Library (VCL) provides various controls. One of these controls is called StringGrid and is implemented by the **TStringGrid** class.

To create a grid, you can add a StringGrid button  . Therefore, you can click this button and position it on a form or another control container:



Like many other controls, a grid is represented with a 3-D effect that raises its borders. This effect is controlled by the **BorderStyle** property. If you do not want to display borders on the control, set the **BorderStyle** property to **bsNone**:

A grid is made of vertical divisions called columns and horizontal divisions called rows. Two of the most visual characteristics of a StringGrid control are its number of columns and its number of rows. These two values are set using the **ColCount** and the **RowCount** properties. The values are integer type and should be >= 0. If you set either property to a negative value, it would be set to 1. If you do not want to display columns, set the **ColCount** to 0. In the same way, if you do not want to display rows, set the **RowCount** value to 0.

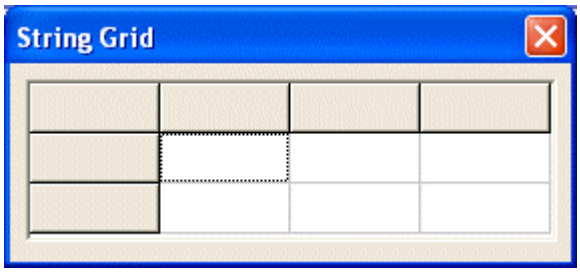
By default, if a grid contains more columns than its width can show, it would display a vertical scroll bar. In the same way, if there are more rows than the control's height can accommodate, it would be equipped with a horizontal scroll bar. The ability to display scroll bars is controlled by the **ScrollBars** property. You can use it to display only the vertical scroll bar (**ssVertical**), only the horizontal scroll bar (**ssHorizontal**), both scroll bars (**ssBoth**), or no scroll bar (**ssNone**) at all.

Like most other list-based controls, a grid is used to display data and, in some applications, you may want the users to enter or use values of the grid control. To guide the users with the values in the grid, you can display explicit text in the fixed columns and fixed rows. The top and left cells are qualified as fixed and, by default, they are the most top and the left cells respectively. Besides these ranges of cells, you can add a fixed row of cells and a fixed column of rows.

If you want to display only one fixed row, it must be the most top range. This characteristic is controlled by the **FixedRows** property and, by default, is set to 1. If you want to display an additional range of fixed cells

on top, change the value of **FixedRows**. In the same way, the number of fixed columns on the left side of the object is controlled by the **FixedCols** property. Setting either of these values to 0 would hide the fixed column or row:

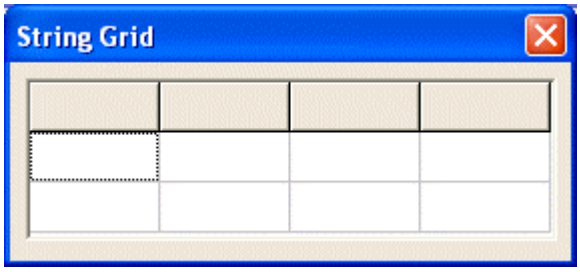
FixedCols=1; FixedRows=1



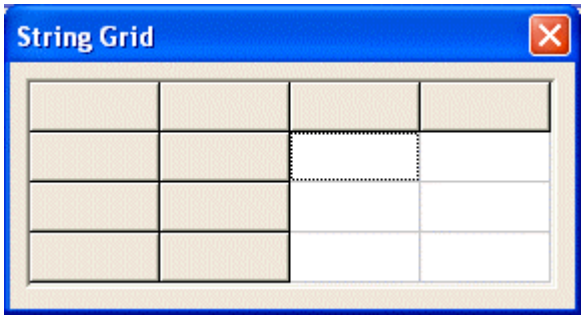
FixedCols=0; FixedRows=0

FixedCols=1; FixedRows=0

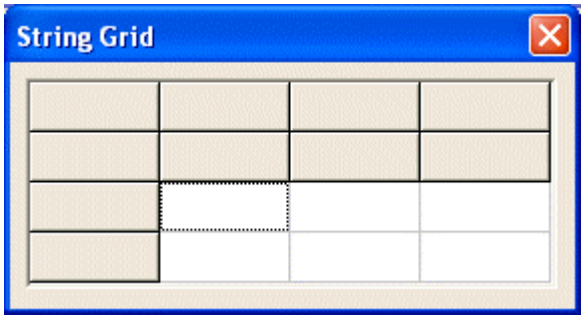
FixedCols=0; FixedRows=1



FixedCols=2; FixedRows=1



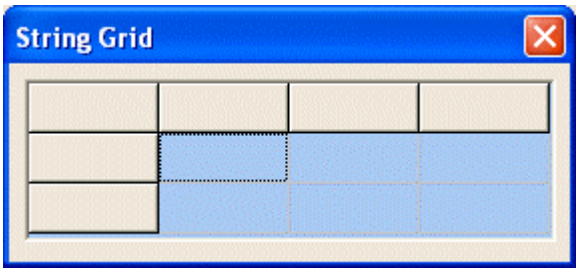
FixedCols=1; FixedRows=2



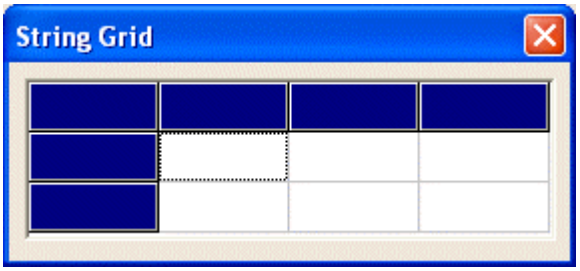
Cells Properties

The intersection of a column and a row is called a cell. To distinguish cells that hold indicative values and those that hold usable or modifiable values, cells are divided in two categories distinguished by two colors. The cells on the top section and those on the left, when displaying a different color than the cells in the middle-center section of the control, are called fixed cells.

To guide the user with the values on the grid, the cells on top and those on the left display the same color as the form, known in the Control Panel as the Button Color. The cells that display usable and modifiable values have a background color known in Control Panel as Window Color. To change the background color of cells that display values, use the **Color** property of the Object Inspector. Here is a grid with the **clSkyBlue** color:



To change the colors of the fixed columns and rows, change the color value of the **FixedColor** property. Here is a grid with the **clNavy FixedColor**:



You can also change these colors programmatically. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   TStringGrid *StatRates = new TStringGrid(this);
   StatRates->Parent = this;

   StatRates->Color = TColor(RGB(255, 230, 204));
   StatRates->FixedColor = TColor(RGB(255, 128, 0));
}
//---------------------------------------------------------------------------
```

To distinguish cells, they are separated by vertical and horizontal lines known as grid lines. By default, the grid lines have a width of 1 integer. To display a wider line, change the value of the **GridLineWidth**. A reasonable value should be less than 10. If you do not want to display grid lines, set the **GridLineWidth** to 0.

In order to access all of the cells that are part of a column, you should know the column's index number. The most left column, which is sometimes the fixed column, unless the **FixedCols** value is set to 0, has an index of 0. The second column from left has an index of 1, etc. In the same way, rows are presented by an index. The most top row has an index of 0; the second row from top has an index of 1, etc.

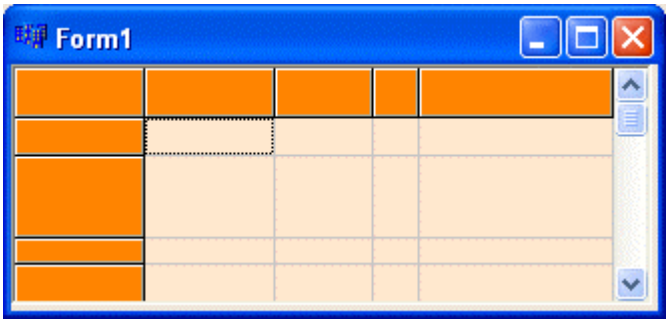
By default, all columns have a width of 64 pixels. At design or run time, you can control this by changing the value of the **DefaultColWidth** property. If you want to control the widths of individual columns, at run time, call the **TStringGrid::ColWidths** property and specify the index of the column you need. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   StringGrid1->Color = TColor(RGB(255, 230, 204));
   StringGrid1->FixedColor = TColor(RGB(255, 128, 0));
```

```
    StringGrid1->ColWidths[2]  = 48;
    StringGrid1->ColWidths[3]  = 22;
    StringGrid1->ColWidths[4]  = 96;
}
//---------------------------------------------------------------------------
```

By default, all rows have a height of 24 pixels. At design or run time, you can control this by changing the value of the **DefaultRowHeight** property. If you want to control the height of individual rows, at run time, call the **TStringGrid::ColHeights** property and specify the index of the row you want access to. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    StringGrid1->Color = TColor(RGB(255, 230, 204));
    StringGrid1->FixedColor = TColor(RGB(255, 128, 0));

    StringGrid1->ColWidths[2]  = 48;
    StringGrid1->ColWidths[3]  = 22;
    StringGrid1->ColWidths[4]  = 96;
    StringGrid1->RowHeights[1] = 18;
    StringGrid1->RowHeights[2] = 40;
    StringGrid1->RowHeights[3] = 12;
}
//---------------------------------------------------------------------------
```



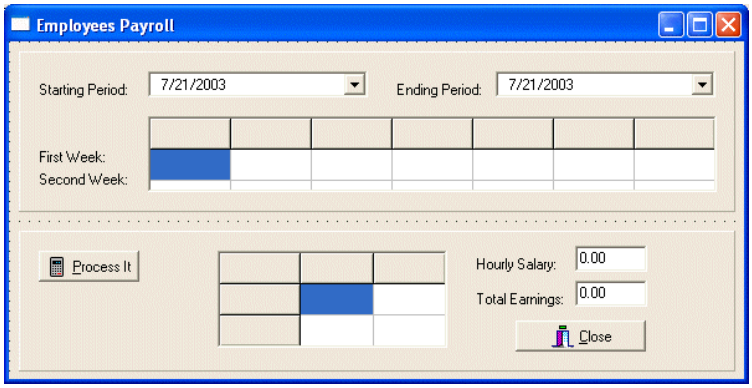
The columns of the grid are stored in a collection or array called **Cols**. By specifying the index of a column, you can change the label of the column header. In the same way, the rows are grouped in a collection called **Rows**. This allows you to change the text of a row header based on its index. The cells of a grid are stored in a two-dimensional array called **Cells**.

The **TStringGrid** class provides **Options** that allow you to customize the behavior of the StringGrid control. For example, if you want to allow the user to move the position of a column, set the **goColMoving** option to true. If you want users to be able to move rows, set the **goRowMoving** option to true.

In order to display cells with their default control appearance, the **DefaultDrawing** property must be set to true, which is the default. If you want to further customize the appearance of cells, you may have to draw them at run time. In this case, you would set the **DefaultDrawing** value to false.

Practical Learning: Controlling a StringGrid Properties

1. Open the Payroll1 application you started previously
2. Continue designing the form as follows:



Label		
Caption: First Week		
Label		
Caption: Second Week		
StringGrid		
Hint: Enter hours worked for each day in the appropriate cell		
ColCount: 7	FixedCols: 0	Height: 78
Name: grdTimeSheet	RowCount: 3	Width: 458
Options	GoEditing: true	GoTabs: true
Panel		
BitBtn		
Glyph C:\Program Files\Common Files\Borland Shared\Images\Buttons\calculat.bmp		
Caption: &Process It	Default: true	Name: btnProcessIt
StringGrid		
ColCount: 3	Height: 78	Name: grdEarnings
RowCount: 3	Width: 185	
Label		
Caption: Hourly Salary:		
Edit		
Name: edtHourlySalary	Width: 58	
Label		
Caption: Total Earnings:		
Edit		
Name: edtTotalEarnings	Width: 58	
BitBtn		
Kind: bkClose		

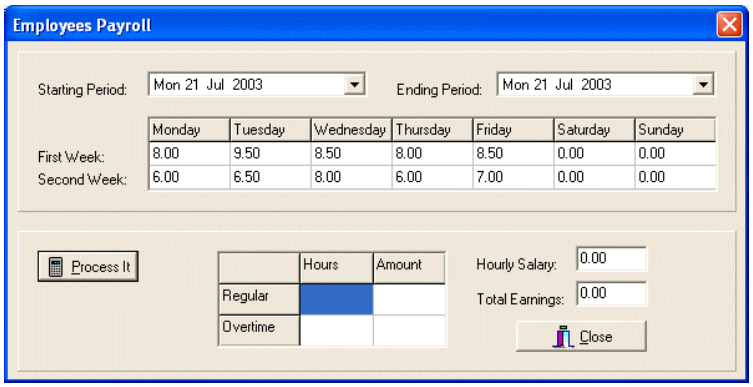
3. Save All

4. Double-click an empty area on the form to access its **OnCreate()** event
5. To customize the StringGrid's cells, add the following code:

```
//---------------------------------------------------------------------------
void __fastcall TfrmMain::FormCreate(TObject *Sender)
{
   dteStartPeriod->CalColors->BackColor = TColor(RGB(230, 245, 255));
   dteStartPeriod->CalColors->MonthBackColor = TColor(RGB(212, 235, 255));
   dteStartPeriod->CalColors->TextColor = clBlue;
   dteStartPeriod->CalColors->TitleBackColor = TColor(RGB(0, 0, 160));
   dteStartPeriod->CalColors->TitleTextColor = clWhite;
   dteStartPeriod->CalColors->TrailingTextColor = TColor(RGB(190, 125, 255));

   dteStartPeriod->Format = "ddd d MMM yyyy";

   dteEndPeriod->CalColors->BackColor = TColor(RGB(255, 236, 218));
   dteEndPeriod->CalColors->MonthBackColor = TColor(RGB(255, 235, 214));
   dteEndPeriod->CalColors->TextColor = TColor(RGB(102, 52, 0));
   dteEndPeriod->CalColors->TitleBackColor = TColor(RGB(176, 88, 0));
   dteEndPeriod->CalColors->TitleTextColor = TColor(RGB(255, 238, 220));
   dteEndPeriod->CalColors->TrailingTextColor = TColor(RGB(230, 115, 0));

   dteEndPeriod->Format = "ddd d MMM yyyy";

   grdTimeSheet->Cells[0][0] = "Monday";
   grdTimeSheet->Cells[1][0] = "Tuesday";
   grdTimeSheet->Cells[2][0] = "Wednesday";
   grdTimeSheet->Cells[3][0] = "Thursday";
   grdTimeSheet->Cells[4][0] = "Friday";
   grdTimeSheet->Cells[5][0] = "Saturday";
   grdTimeSheet->Cells[6][0] = "Sunday";

   grdTimeSheet->RowHeights[0] = 18;

   grdTimeSheet->Cells[0][1] = "0.00";
   grdTimeSheet->Cells[1][1] = "0.00";
   grdTimeSheet->Cells[2][1] = "0.00";
   grdTimeSheet->Cells[3][1] = "0.00";
   grdTimeSheet->Cells[4][1] = "0.00";
   grdTimeSheet->Cells[5][1] = "0.00";
   grdTimeSheet->Cells[6][1] = "0.00";

   grdTimeSheet->RowHeights[1] = 18;

   grdTimeSheet->Cells[0][2] = "0.00";
   grdTimeSheet->Cells[1][2] = "0.00";
   grdTimeSheet->Cells[2][2] = "0.00";
   grdTimeSheet->Cells[3][2] = "0.00";
   grdTimeSheet->Cells[4][2] = "0.00";
   grdTimeSheet->Cells[5][2] = "0.00";
   grdTimeSheet->Cells[6][2] = "0.00";

   grdTimeSheet->RowHeights[2] = 18;

   grdEarnings->Cells[0][1] = "Regular";
   grdEarnings->Cells[0][2] = "Overtime";
   grdEarnings->Cells[1][0] = "Hours";
   grdEarnings->Cells[2][0] = "Amount";
}
```

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

6. Save All and Test the application:



7. Close the form and return to Bcb

StringGrid Methods

To programmatically create a StringGrid control, declare a pointer to **TStringGrid** class using the **new** operator. Use its default constructor to specify the container of the control as parent. Here is an example:

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

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   TStringGrid *StatRates = new TStringGrid(this);
   StatRates->Parent = this;
}
//---------------------------------------------------------------------------
```

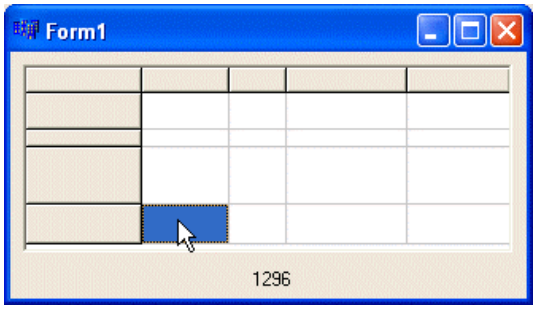
After creating the control, you can set its properties to the values of your choice. After using the control, you can get rid of it using the delete operator, or you can trust its parent to do it for you.

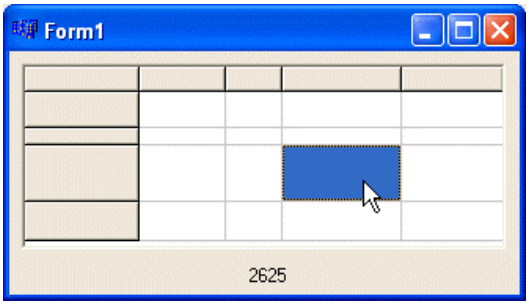
If at any time a cell is selected, you can get the rectangular dimension of that cell using the **CellRect()** method. Its syntax is:

TRect __fastcall CellRect(int *ACol*, int *ARow*);

The arguments, *ACol* and *ARow*, represent the column index and the row index of the cell that has focus. This method returns the **TRect** rectangle of the cell. You can call this method when the user clicks a cell in the StringGrid control. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
   StringGrid1->ColWidths[0] = 72;
   StringGrid1->ColWidths[1] = 54;
   StringGrid1->ColWidths[2] = 35;
   StringGrid1->ColWidths[3] = 75;

   StringGrid1->RowHeights[0]= 15;
   StringGrid1->RowHeights[1]= 22;
   StringGrid1->RowHeights[2]= 10;
   StringGrid1->RowHeights[3]= 35;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1Click(TObject *Sender)
{
   TRect Recto = StringGrid1->CellRect(StringGrid1->Col, StringGrid1->Row);
   int Area = Recto.Width() * Recto.Height();
   Label1->Caption = Area;
}
//---------------------------------------------------------------------------
```





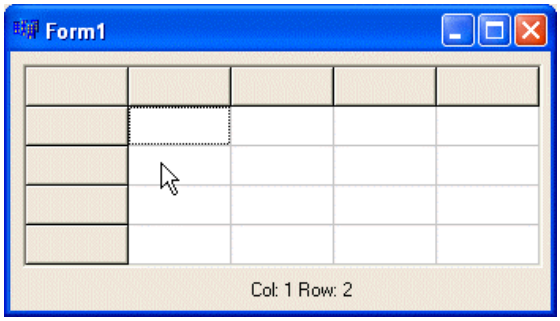
If the mouse is positioned or passing somewhere on or over the StringGrid control and you want to know on what cell the mouse is, you can use the **MouseToCell()** method. Its syntax is:

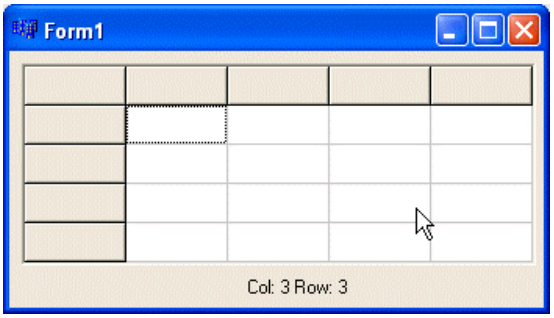
```
void __fastcall MouseToCell(int X, int Y, int &ACol, int &ARow);
```

This method is usually used on a Mouse event such as **OnMouseDown()**, **OnMouseMove()**, and **OnMouseUp()** as these events provide the mouse coordinates. The **MouseToCell()** method retrieves the horizontal and vertical coordinates of the mouse, translates that position to the column and row indexes of the cell under the mouse and return those values. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1MouseMove(TObject *Sender,
      TShiftState Shift, int X, int Y)
{
   int x, y;

   StringGrid1->MouseToCell(X, Y, x, y);
   Label1->Caption = "Col: " + AnsiString(x) + " Row: " + AnsiString(y);
}
```

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





In the same way, sometimes when the user clicks a cell, you may want to find out what cell was clicked. To get this information, you can call the **GridCoord()** method. Its syntax is:

```
struct TGridCoord

{
  int X;
  int Y;
};

TGridCoord __fastcall MouseCoord(int X, int Y);
```

This method also is usually used in a mouse event. It takes as arguments the mouse position and returns a **TPoint**-like object, called **TGridCoord**. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1MouseDown(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{
   TGridCoord GC = StringGrid1->MouseCoord(X, Y);

   Label1->Caption = "Col: " + AnsiString(GC.X) + " Row: " + AnsiString(GC.Y);
}
//---------------------------------------------------------------------------
```

StringGrid Events

As a descendant of **TWinControl**, the **TStringGrid** class inherits all the usual events that are common to Windows control. It fires the **OnClick** event when the user clicks anywhere on the control. It sends an **OnDblClick** event when the user double-click any cell. It uses all mouse events (**OnMouseDown**, **OnMouseMove**, **OnMouseUp**, **OnMouseWheelDown**, and **OnMouseWheelUp**) as well as keyboard events (**OnKeyDown**, **OnKeyPress**, and **OnKeyUp**). Besides the regular control events, the StringGrid control fires events that are proper to its functionality.

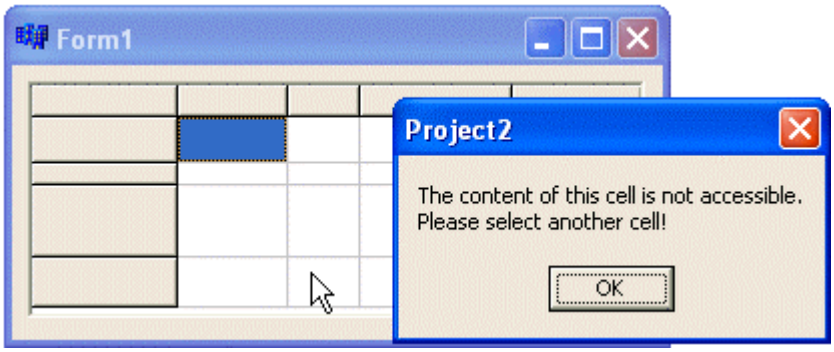
Just before the user selects the content of a cell, the control fires the **OnSelectCell()** event. Its syntax is:

```
void __fastcall OnSelectCell(TObject *Sender, int ACol, int ARow, bool &CanSelect)
```

The *ACol* and *ARow* arguments represent the cell that is about to be selected. The *CanSelect* argument allows you to specify whether the user is allowed to select the content of the cell. This event allows you to

decide whether the cell can be selected or not. The following event is used to let the user know that a certain cell cannot be accessed:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1SelectCell(TObject *Sender, int ACol,
      int ARow, bool &CanSelect)
{
   if( ACol == 2 && ARow == 4 )
   {
      ShowMessage("The content of this cell is not accessible.\n"
                  "Please select another cell!");
      return;
   }
}
//---------------------------------------------------------------------------
```



If the user has selected a cell and wants to edit its content, you can find out the content of such a cell using the **OnGetEdtText()** event. This even fires as soon as the user has selected text included in a cell but just before the user has had a chance to edit it. This means that you can determine whether the user is allowed to change the contents of a particular cell. When this even fires, it communicates the grid coordinates of the cell that was clicked, allowing you to retrieve the content of that cell and do what you want. Here is an example:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1GetEditText(TObject *Sender, int ACol,
      int ARow, AnsiString &Value)
{
   AnsiString Content = StringGrid1->Cells[ACol][ARow];
   Label1->Caption = Content;
}
//---------------------------------------------------------------------------
```

On the other hand, when the user has changed the content of a cell, the StringGrid control fires an **OnSetEditText()** event. This is a good place to validate, accept, or reject the changes that the user has performed. This event also provides you with the grid coordinates of the cell whose contents the user has modified.

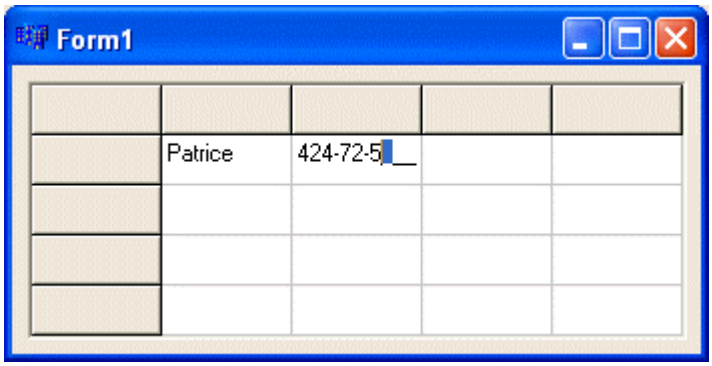
To better control what type of text the user is allowed to enter in a cell, in all cells of a particular row, or in all cells of a particular column, you can use the **OnGetEditMask()** event. Its syntax is:

void __fastcall OnGetEditMask(TObject **Sender*, int *ACol*, int *ARow*, AnsiString &*Value*)

The *ACol* and the *ARow* parameters represent the grid indexes of the cell. The *Value* is a string of the same type used for the **EditMask** property of the MaskEdit control. This event is used to set the **EditMask**

needed for a particular cell. The following event restricts only US Social Security Numbers in all cells of the second column:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1GetEditMask(TObject *Sender, int ACol,
      int ARow, AnsiString &Value)
{
   if( StringGrid1->Col == 2 )
      Value = "000-00-0000";
}
//---------------------------------------------------------------------------
```



If you had allowed the user to move the columns, whenever a user has performed this operation, the StringGrid control would fire the **OnColumnMoved()** event. Its syntax is:

```
void __fastcall TStringGrid(TObject* Sender, long FromIndex, long ToIndex);
```

This event is a good place to decide what to do, if there is anything to do, when the user has moved a column. In the same way, if you had allowed the user to move rows, the StringGrid control sends an **OnRowMoved()** event immediately after the user has moved a row.

If you had let the compiler know that you would set the appearance of cells yourself, which would have been communicated by setting the **DefaultDrawing** property to false, you can use the **OnDrawCell()** event to perform this customization. The syntax of this event is:

```
void __fastcall StringGridDrawCell(TObject *Sender, int ACol, int ARow,
                                   TRect &Rect, TGridDrawState State)
```

The cell whose characteristics need to be set is Cell[*ACol*][*ARow*]. This means that you can locate any cell in the grid and set its properties as you like and as possible. For example, you can change the individual background color of a cell. The following code changes the background color of Cell[3][2] to blue:

```
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1DrawCell(TObject *Sender, int ACol,
      int ARow, TRect &Rect, TGridDrawState State)
{
   if( ACol == 3 && ARow == 2 )
   {
      StringGrid1->Canvas->Brush->Color = clBlue;
      StringGrid1->Canvas->FillRect(Rect);
   }
}
//---------------------------------------------------------------------------
```

In the same way, you can change the text color of any cell of your choice independently of the other cells.

The *Rect* parameter is the location and dimension of the cell whose characteristics you want to change.

The *State* argument is a member of the **TGridDrawState** set which is defined as follows:

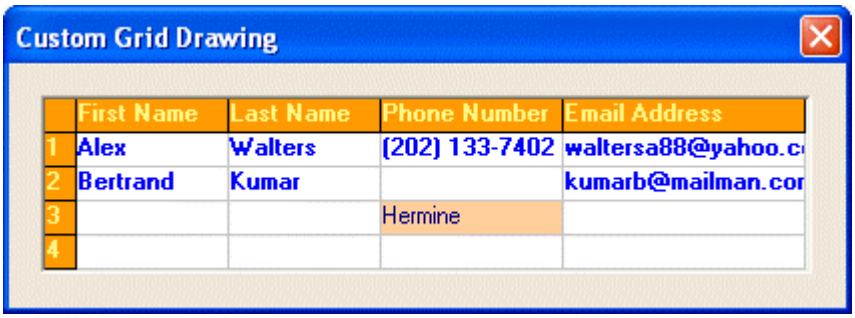
```
enum Grids__3 { gdSelected, gdFocused, gdFixed };
typedef Set<Grids_3, gdSelected, gdFixed> TGridDrawState;
```

This set allows you to examine the state of a particular cell. Because this value is a set, a particular cell can have more than one of these values. If a cell is selected, which by default gives it a background color different than the others, then its *State* contains the **gdSelected** value. If a cell has focus, which could mean that the user has just clicked it, sometimes to edit, the cell has the **gdFocused** value. Note that a cell can be selected and have focus, which means it would have both **gdSelected** and **gdFocused**. If a cell is a fixed cell as we described previously, then the cell has the **gdFixed** value.

Here is an example of using the **OnDrawCell()** event to customize the appearance of a StringGrid object:

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

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
   StringGrid1->DefaultDrawing = False;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::StringGrid1DrawCell(TObject *Sender, int ACol,
      int ARow, TRect &Rect, TGridDrawState State)
{
   if( State.Contains(gdFixed) )
   {
      StringGrid1->Canvas->Brush->Color = static_cast<TColor>(RGB(255, 155, 0));
      StringGrid1->Canvas->Font->Style = TFontStyles() << fsBold;
      StringGrid1->Canvas->Font->Color = static_cast<TColor>(RGB(250, 245, 135));

      StringGrid1->Canvas->Rectangle(Rect);
   }
   else if( State.Contains(gdSelected) )
   {
      StringGrid1->Canvas->Brush->Color = static_cast<TColor>(RGB(255, 205, 155));
      StringGrid1->Canvas->Font->Style = TFontStyles() >> fsBold;
      StringGrid1->Canvas->Font->Color = clNavy;
      StringGrid1->Canvas->FillRect(Rect);
   }
   else
   {
      StringGrid1->Canvas->Brush->Color = clWhite;
      StringGrid1->Canvas->Font->Color  = clBlue;
      StringGrid1->Canvas->FillRect(Rect);
   }
```

```
    StringGrid1->ColWidths[0] = 15;
    StringGrid1->ColWidths[1] = 75;
    StringGrid1->ColWidths[2] = 75;
    StringGrid1->ColWidths[3] = 90;
    StringGrid1->ColWidths[4] = 120;

    StringGrid1->RowHeights[0] = 16;
    StringGrid1->RowHeights[1] = 16;
    StringGrid1->RowHeights[2] = 16;
    StringGrid1->RowHeights[3] = 16;
    StringGrid1->RowHeights[4] = 16;

    AnsiString text = StringGrid1->Cells[ACol][ARow];
    StringGrid1->Canvas->TextRect(Rect, Rect.Left, Rect.Top, text);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    StringGrid1->Cells[0][1] = "1";
    StringGrid1->Cells[0][2] = "2";
    StringGrid1->Cells[0][3] = "3";
    StringGrid1->Cells[0][4] = "4";

    StringGrid1->Cells[1][0] = "First Name";
    StringGrid1->Cells[2][0] = "Last Name";
    StringGrid1->Cells[3][0] = "Phone Number";
    StringGrid1->Cells[4][0] = "Email Address";

    StringGrid1->Cells[1][1] = "Alex";
    StringGrid1->Cells[2][1] = "Walters";
    StringGrid1->Cells[3][1] = "(202) 133-7402";
    StringGrid1->Cells[4][1] = "waltersa88@yahoo.com";
    StringGrid1->Cells[1][2] = "Bertrand";
    StringGrid1->Cells[2][2] = "Kumar";
    StringGrid1->Cells[4][2] = "kumarb@mailman.com";
    StringGrid1->Cells[3][3] = "Hermine";
}
//---------------------------------------------------------------------------
```



Practical Learning: Using String Grid Events

1. In the **private** section of the form, declare two variables and a method as follows:

```
private:
    double HoursWeek1, HoursWeek2;
    double __fastcall EvaluateTime(AnsiString StrTime); // User declarations
```

2. In the top section of the source file of the form, include the StrUtils header file:

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

#include "Main.h"
//---------------------------------------------------------------------------
```

3. Implement the above method as follows:

```
//---------------------------------------------------------------------------
double __fastcall TfrmMain::EvaluateTime(AnsiString StrTime)
{
   //TODO: Add your source code here
   double dValue;
   AnsiString StrValue = AnsiReplaceStr(StrTime, " ", "");

   if( StrValue.IsEmpty() )
      return 0.00;
   else
      return StrToFloat(StrValue);
}
//---------------------------------------------------------------------------
```

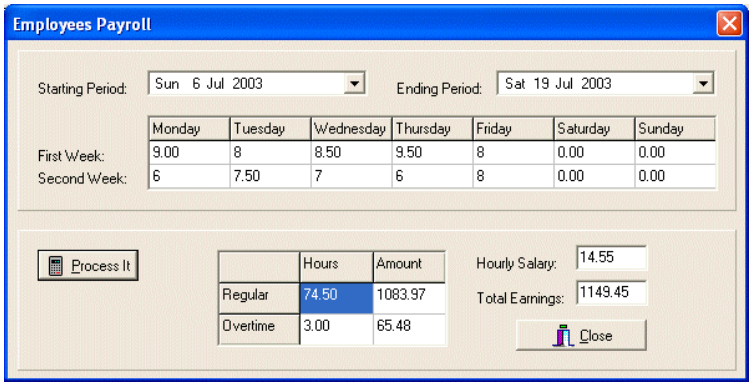
4. To use of the StringGrid events, on the form, click the top StringGrid control
5. In the Object Inspector, click the Events tab and double-click the event side of **OnSetEditText**
6. Implement it as follows:

```
//---------------------------------------------------------------------------
void __fastcall TfrmMain::grdTimeSheetSetEditText(TObject *Sender, int ACol,
     int ARow, const AnsiString Value)
{
   double Monday1    = EvaluateTime(grdTimeSheet->Cells[0][1]);
   double Tuesday1   = EvaluateTime(grdTimeSheet->Cells[1][1]);
   double Wednesday1 = EvaluateTime(grdTimeSheet->Cells[2][1]);
   double Thursday1  = EvaluateTime(grdTimeSheet->Cells[3][1]);
   double Friday1    = EvaluateTime(grdTimeSheet->Cells[4][1]);
   double Saturday1  = EvaluateTime(grdTimeSheet->Cells[5][1]);
   double Sunday1    = EvaluateTime(grdTimeSheet->Cells[6][1]);

   HoursWeek1 = Monday1 + Tuesday1 + Wednesday1 +
            Thursday1 + Friday1 + Saturday1 + Sunday1;

   double Monday2    = EvaluateTime(grdTimeSheet->Cells[0][2]);
   double Tuesday2   = EvaluateTime(grdTimeSheet->Cells[1][2]);
   double Wednesday2 = EvaluateTime(grdTimeSheet->Cells[2][2]);
   double Thursday2  = EvaluateTime(grdTimeSheet->Cells[3][2]);
   double Friday2    = EvaluateTime(grdTimeSheet->Cells[4][2]);
   double Saturday2  = EvaluateTime(grdTimeSheet->Cells[5][2]);
   double Sunday2    = EvaluateTime(grdTimeSheet->Cells[6][2]);

   HoursWeek2 = Monday2 + Tuesday2 + Wednesday2 +
            Thursday2 + Friday2 + Saturday2 + Sunday2;
}
//---------------------------------------------------------------------------
```

7. On the form, double-click the Process It button and implement its **OnClick** event as follows:

```
//---------------------------------------------------------------------------
void __fastcall TfrmMain::btnProcessItClick(TObject *Sender)
{
   double RegHours1, RegHours2, OvtHours1, OvtHours2;
   double RegAmount1, RegAmount2, OvtAmount1, OvtAmount2;
   double RegularHours, OvertimeHours;
   double RegularAmount, OvertimeAmount, TotalEarnings;

   double HourlySalary = StrToFloat(edtHourlySalary->Text);
   double OvtSalary    = HourlySalary * 1.5;

   if( HoursWeek1 < 40 )
   {
      RegHours1  = HoursWeek1;
      RegAmount1 = HourlySalary * RegHours1;
      OvtHours1  = 0.00;
      OvtAmount1 = 0.00;
   }
   else if( HoursWeek1 >= 40 )
   {
      RegHours1  = 40;
      RegAmount1 = HourlySalary * 40;
      OvtHours1  = HoursWeek1 - 40;
      OvtAmount1 = OvtHours1 * OvtSalary;
   }

   if( HoursWeek2 < 40 )
   {
      RegHours2  = HoursWeek2;
      RegAmount2 = HourlySalary * RegHours2;
      OvtHours2  = 0.00;
      OvtAmount2 = 0.00;
   }
   else if( HoursWeek2 >= 40 )
   {
      RegHours2  = 40;
      RegAmount2 = HourlySalary * 40;
      OvtHours2  = HoursWeek2 - 40;
      OvtAmount2 = OvtHours2 * OvtSalary;
   }

   RegularHours   = RegHours1  + RegHours2;
   OvertimeHours  = OvtHours1  + OvtHours2;
   RegularAmount  = RegAmount1 + RegAmount2;
   OvertimeAmount = OvtAmount1 + OvtAmount2;
   TotalEarnings  = RegularAmount + OvertimeAmount;

   grdEarnings->Cells[1][1] = FloatToStrF(RegularHours, ffFixed, 6, 2);
   grdEarnings->Cells[1][2] = FloatToStrF(OvertimeHours, ffFixed, 6, 2);
   grdEarnings->Cells[2][1] = FloatToStrF(RegularAmount, ffFixed, 6, 2);
   grdEarnings->Cells[2][2] = FloatToStrF(OvertimeAmount, ffFixed, 6, 2);
   edtTotalEarnings->Text   = FloatToStrF(TotalEarnings, ffFixed, 6, 2);
}
//---------------------------------------------------------------------------
```

8. Save All and test the application. Here is an example:



9. Close the form and return to Bcb