ObjectScripting tutorial

This tutorial teaches you how to use the cScript language to add menu items to the Help menu. The tutorial consists of four parts:

Part 1: Adds an item to the Help menu that prints text in the Script page of the Message window. This part takes approximately 10 minutes to complete.

Part 2: Adds the menu item OWL Help to the Help menu and launches the OWL Help file when the menu item is selected. This part takes approximately 10 minutes to complete.

Part 3: Adds two new items, ObjectScripting Help and Standard Template Library Help, to the Help menu using two different methods. Launches the appropriate Help file when a menu item is selected. This part takes approximately 15 minutes to complete.

Part 4: Adds all of the Borland C++ Help files to the Help menu. Launches the appropriate Help file when a menu item is selected. This part takes approximately 20 minutes to complete.

About this tutorial

Each part of this tutorial teaches how to accomplish a unique task, building on knowledge learned in the previous part(s). It is recommended that you follow the tutorial from beginning to end. You are not required to complete the tutorial in one sitting, however. You can choose to complete only one part and return at another time to complete another part or parts.

Using this tutorial

- To move to the next step, click Next.
- To move to the previous step, click Previous.
- To exit at any time, click the close button in the upper right corner of the Help topic.
- When entering sample code, type it exactly as shown noting indentations and curly braces. Press Enter at the end of each line you add.
- To copy the entire sample for each part of the tutorial to the text editor:
- Click Sample Code below the topic title.
- Right-click in the topic and choose Copy.
- Switch to the Borland C++ Edit window.
- Delete the first and the last line of the file.

ObjectScripting Tutorial: Part 1

Next Sample code

This part of the tutorial teaches you how to:

- Start the script file (Step 1)
- Create a local instance of an object (Step 2)
- Create a class (Step 3)
- Load MENUHOOK.SPP (Step 4)
- Declare a method that adds a menu item (Step 5)
- Execute the method (Step 6)
- Run the script file (Step 7)

Sample code for Tutorial Part 1

```
// ObjectScripting example
// Copyright (c) 1996 by Borland International, All Rights Reserved
// STEP1.SPP: Add an item to the Help menu that prints text
//            in the Script page of the Message window.

declare ScriptEngine scriptEngine;

class HelpMenu()
  {
// Load "MENUHOOK.SPP", necessary for adding menu items.
     if(!scriptEngine.IsLoaded("menuhook.spp"))
          {
          scriptEngine.Load("menuhook.spp");
          }
// Declare a method to add a menu item and execute the associated script.
     AddMenu(menu_text, script_text)
     {
     assign_to_view_menu("IDE", menu_text, script_text, menu_text);
     }
  };

// At load time, create a Print Text menu item on the Help menu.
// When Print Text is selected, print "A message from the Help menu!"
// in the Script page of the Message window.

declare x = new HelpMenu();
x.AddMenu("&Help|Print Text", "print(\"A message from the Help menu!\")");
```

Starting the script file

(ObjectScripting Tutorial, Part 1: Step 1 of 7)

Next Sample code

To start a script file in Borland C++,

1. Choose File|New|Text Edit.
2. To name the file, choose File|Save As.
3. In the Save File As dialog box, choose the following directory:
   `C:\BC5\SCRIPT\EXAMPLES`
4. In the File Name box, enter the name `STEP1.SPP`.
5. Click OK.

- You have just started a script file called STEP1.SPP.
- In the next step, you will start adding code to your script file.

Creating a local instance of an object

(ObjectScripting Tutorial, Part 1: Step 2 of 7)

Next Previous Sample code

A ScriptEngine object loads, unloads, executes, maintains modules and keeps error information on scripts. In this step, you will create a local instance of the object.

To create a local instance of a *ScriptEngine* object,

1. Enter the following text in STEP1.SPP:

```
declare ScriptEngine scriptEngine;
```

- You have just created a local instance of a *ScriptEngine* object. Creating the script engine locally provides slightly better performance than importing the symbol of the system-wide instance.
- In the next step, you will create a class called *HelpMenu.*

Creating a class

(ObjectScripting Tutorial, Part 1: Step 3 of 7)

Next Previous Sample code

Use the class keyword to define a cScript class. A class is a collection of properties, methods, and events that affect the behavior of the IDE.

To create a cScript class,

1. Add the following line to your script file:

```
class HelpMenu()
```

- You have just created a class called *HelpMenu.* You will use this class in your script file when you add a menu item to the Help menu.
- In the next step, you will load the file MENUHOOK.SPP.

Loading MENUHOOK.SPP

(ObjectScripting Tutorial, Part 1: Step 4 of 7)

Next Previous Sample code

MENUHOOK.DLL, in the BC5\BIN directory, contains the functionality you need to add menu items to menus, menu items to SpeedMenus, and buttons to the SpeedBar. To use this functionality, you need to load the associated script file, MENUHOOK.SPP.

When you load MENUHOOK.SPP, the following functions become available:

Function	Description
assign_to_view_menu()	Adds a menu item to a menu
remove_view_menu_item()	Removes a menu item that was added with **assign_to_view_menu()**
define_button()	Defines a button that can be added to the SpeedBar

To load MENUHOOK.SPP,

1. Add the following lines to your script file:

```
{
if(!scriptEngine.IsLoaded("menuhook.spp"))
   {
   scriptEngine.Load("menuhook.spp");
   }
```

The first line uses the if keyword with the ! logical operator and the ScriptEngine.IsLoaded method to determine if MENUHOOK.SPP has already been loaded. If it has not been loaded, the ScriptEngine.Load method loads it.

Note: Case is important when loading and running script files. For more information, see: About script initialization and About script function referencing

---

- You have just loaded MENUHOOK.SPP.
- In the next step, you will declare a method that adds a menu item.

Declaring a method that adds a menu item

(ObjectScripting Tutorial, Part 1: Step 5 of 7)

Next Previous Sample code

This step declares a method, **AddMenu(),** that adds a new menu item to the Help menu.

To declare **AddMenu(),**

1. Add the following lines to your script file:

```
AddMenu(menu_text, script_text)
   {
   assign_to_view_menu("IDE", menu_text, script_text, menu_text);
   }
};
```

The first line declares a method called **AddMenu()** with the parameters *menu_text* and *script_text.* **AddMenu()** uses assign_to_view_menu, a function defined in MENUHOOK.SPP, to assign a menu item to a menu.

- You have just declared the **AddMenu()** method.
- In the next step, you will execute the method when the menu item is selected.

Executing the method

(ObjectScripting Tutorial, Part 1: Step 6 of 7)

Next Previous Sample code

To execute **AddMenu()** when the menu item is selected,

1. Add these lines to your script file:

```
declare x = new HelpMenu();
x.AddMenu("&Help|Print Text", "print(\"A message from the Help menu!\")");
```

The declare keyword declares the variable *x*. This line also assigns the variable *x* to a new instance of the *HelpMenu* class.

x.AddMenu displays the string `Print Text` on the Help menu. `A message from the Help menu!` is displayed on the Script page of the Message window when Help|Print Text is selected.

- You have just executed a method that prints a message on the Script page of the Message window.
- In the next step, you will run the script file.

Running the script file

(ObjectScripting Tutorial, Part 1: Step 7 of 7)

Previous Sample code

To run the script file you've been working on,

1. Choose File|Save to save the script file.
2. To run the script, right click in the Edit window and choose Run File.
3. To see the results, go to the Help menu. Note that the menu item Print Text has been appended to the bottom of the Help menu. Click Print Text.
4. Display the Message window by choosing View|Message. Choose the Script tab. Scroll to the bottom of the message display, where the following text is displayed:

   ```
   A message from the Help menu!
   ```

5. To remove the Print Text command from the Help menu, exit Borland C++. When you load Borland C++ again, Print Text will no longer display.

- You have now finished Part 1 of the tutorial.
- In Part 2, you will add the menu item OWL Help to the Help menu and launch the OWL Help file when the menu item is selected.

ObjectScripting Tutorial: Part 2

Next Sample code

This part of the tutorial teaches you how to:

- Import the *IDE* object (Step 1)
- Import a symbol of a system-wide instance of a *ScriptEngine* object (Step 2)
- Declare a method that adds a menu item to the Help menu (Step 3)
- Execute the method (Step 4)

Sample code for Tutorial Part 2

```
// ObjectScripting example
// Copyright (c) 1996 by Borland International, All Rights Reserved
// STEP2.SPP: Add the menu item OWL Help to the Help menu. Launch the
//            OWL Help file when OWL Help is selected.

import IDE;
import scriptEngine;

class HelpMenu()
   {
// Load "MENUHOOK.SPP", necessary for adding menu items.
     if (!scriptEngine.IsLoaded("menuhook.spp"))
          {
          scriptEngine.Load("menuhook.spp");
          }
// Declare a method to add a menu item and execute the associated script.
     AddMenu(menu_text, script_text)
     {
     assign_to_view_menu("IDE", "&Help|" + menu_text, script_text,
menu_text);
      }
   };

// At load time, create an OWL Help menu item on the Help menu.
// When OWL Help is selected, launch the help file OWL.HLP.

declare helpMenu = new HelpMenu();
helpMenu.AddMenu("OWL Help", "IDE.HelpOWLAPI()");
```

Importing the *IDE* object

(ObjectScripting Tutorial, Part 2: Step 1 of 4)

Next Sample code

When you start the Borland C++ IDE, the object *IDE,* in *IDEApplication,* is automatically created as a global object. *IDE* gives you control over the system. All items contained in the Borland C++ IDE menu structure can be accessed through the *IDE* object.

First, start a script file and call it STEP2.SPP. If you don't know how to do this, see Part 1: Step 1.

To import the *IDE* object,

1. Enter the following text in the script file.

```
import IDE;
```

- You have just started a script file and imported the *IDE* object.
- In the next step, you will import a symbol of a system-wide instance of the *ScriptEngine* object.

Importing a symbol of the system-wide instance of an object

(ObjectScripting Tutorial, Part 2: Step 2 of 4)

Next Previous Sample code

A ScriptEngine object loads, unloads, executes, maintains modules and keeps error information on scripts. You can import a symbol of a system-wide instance of the *ScriptEngine* object (note that a local instance can also be created).

To import a symbol of a system-wide instance of a *ScriptEngine* object,

1. Enter the following text in your script:

```
import scriptEngine;
```

- You have just imported a symbol of a system-wide instance of a *ScriptEngine* object. Importing the symbol as system-wide makes the script engine's functionality available to all scripts. In Part 1: Step 2, you created the script engine as a local instance, which slightly increases performance.
- In the next step, you will create a class called *HelpMenu,* load MENUHOOK.SPP, and declare a method that adds a menu item.

Declaring a method that adds a menu item

(ObjectScripting Tutorial, Part 2: Step 3 of 4)

Next Previous Sample code

This step declares a method, **AddMenu(),** that adds a menu item to the Help menu.

As you learned in Part 1, create a class called *HelpMenu,* then load MENUHOOK.SPP. If you need more information, go to Part 1, Step 3 and Step 4.

To declare **AddMenu(),**

1. Add the following lines to your script:

```
AddMenu(menu_text, script_text)
   {
   assign_to_view_menu("IDE", "&Help|" + menu_text, script_text, menu_text);
   }
};
```

The first line declares a method called **AddMenu()** with the arguments *menu_text* and *script_text.* **AddMenu()** uses assign_to_view_menu, a function that is defined in MENUHOOK.SPP, to add a menu item to a menu. In this case, the new menu item is being added to the Help menu of the IDE view.

Note: In Part 1: Step 5, you performed a similar task, but specified the Help menu when the method was loaded. These two examples represent two different ways to use the **assign_to_view_menu** function.

- You have just created *HelpMenu* class, loaded MENUHOOK.SPP, and declared the **AddMenu()** method.
- In the next step, you will execute the method and run the script.

Executing the method

(ObjectScripting Tutorial, Part 2: Step 4 of 4)

Previous Sample code

To execute **AddMenu()** when the associated menu item is selected,

1. Add these lines to your script:

```
declare helpMenu = new HelpMenu();
helpMenu.AddMenu("OWL Help", "IDE.HelpOWLAPI()");
```

   The declare keyword declares the variable *helpMenu.* This line also assigns the variable *helpMenu* to a new instance of the *HelpMenu* class. (Note the difference in case - cScript is a case sensitive language.) *helpMenu.AddMenu* displays OWL Help on the Help menu and launches the associated Help file when Help|OWL Help is selected.

2. Save the script file and run it. For more information, see Part 1: Step 7.

   To see the results, go to the Help menu. Note that the menu item OWL Help is now on the Help menu.

3. Click OWL Help.

   The Contents topic of the OWL Help file is displayed.

4. To remove the OWL Help command from the Help menu, exit Borland C++. When you load Borland C++ again, OWL Help will no longer display.

- You have just learned how to execute a method that launches the OWL Help file when the menu item OWL Help is selected.
- You have now finished Part 2 of the tutorial.
- In Part 3, you will use two different methods to add two new items, ObjectScripting and Standard Template Library, to the Help menu.

ObjectScripting Tutorial: Part 3

Next Sample code

This part of the tutorial teaches you how to:

- Locate the Borland C++ Help directory and store it (Step 1)
- Declare two methods that add menu items to the Help menu (Step 2)
- Assign a menu item to the Help menu (Step 3)
- Declare a function that adds a backslash to the Help directory path name (Step 4)
- Launch a Help file when the associated Help menu item is selected (Step 5)

Sample code for Tutorial Part 3

```
// ObjectScripting example
// Copyright (c) 1996 by Borland International, All Rights Reserved
// STEP3.SPP: Add two menu items to the Help menu. Launch Help files
//            when menu item is selected.

import IDE;
import scriptEngine;

class HelpMenu()

{
// Find the help directory and store it in the static sHelpDir
declare sProgram = new String();
   sProgram.Text = IDE.ModuleName;
   declare breakIndex = sProgram.Index("\\BIN\\", SEARCH_BACKWARD);
   declare sHelpDir = sProgram.SubString(0, breakIndex - 1).Text + "\\
HELP\\";

// Load "MENUHOOK.SPP", necessary for adding menu items.
     if(!scriptEngine.IsLoaded("menuhook.spp"))
          {
          scriptEngine.Load("menuhook.spp");
          }

// Add a menu item under "Help" menu, launch the Help file
// associated with it. The helpFile parameter is the file name
// without the path.
     AddHelpFile(menuText, helpFile)
        {
        AddHelpFileFullPath(menuText, sHelpDir + helpFile);
        }

     AddHelpFileFullPath(menuText, helpFile)
        {
        declare menuCmd =
           "IDE.Help(\"" + AddBackSlash(helpFile) + "\", "
            + "3, " + "\"\"" + ");";
         assign_to_view_menu("IDE", "&Help|" + menuText,
            menuCmd, menuText);
         }

// Important note!
// The command text passed to assign_to_view_menu
// should be IDE:Help("C:\\BC5\\HELP\\...", ...).
// When cScript compiles, it compiles the double backslash
// as a single backslash. This routine adds a backslash to
// the directory path name.

   AddBackSlash(fileName)
      {
      declare origFileName = new String();
      origFileName.Text = fileName;
      declare targetFileName = "";
      declare breakIndex = origFileName.Index("\\");
      while (breakIndex > 0)
```

```
         {
         targetFileName += origFileName.SubString(0, breakIndex - 1).Text
            + "\\\\";
         origFileName = origFileName.SubString(breakIndex);
         breakIndex = origFileName.Index("\\");
         }
      targetFileName += origFileName.Text;
      return targetFileName;
      }
};
// At load time, create two new menu items on the Help menu:
// ObjectScripting for SCRIPT.HLP, and Standard Template
// Library for STL.HLP.
// These two menu items show two different ways to add a help item.
declare helpMenu = new HelpMenu();
helpMenu.AddHelpFile("ObjectScripting", "SCRIPT.HLP");
helpMenu.AddHelpFileFullPath("Standard Template Library",
   "C:\\BC5\\HELP\\STL.HLP");
```

Finding the Help directory

(ObjectScripting Tutorial, Part 3: Step 1 of 5)

Next Sample code

This step shows you how to find the name of the Borland C++ Help directory and store it. You use the stored name when you add the Help file name to the Help menu.

First, start a script file and call it STEP3.SPP. Then, as previously learned:

- Import the *IDE* object (Part 2: Step 1)
- Import a symbol of a system-wide instance of *ScriptEngine* (Part 2: Step 2)
- Create a class called *HelpMenu* (Part 1: Step 3)

To find the name of the Help directory,

1. Add the following lines to your script:

```
{
declare sProgram = new String();
   sProgram.Text = IDE.ModuleName;
   declare breakIndex = sProgram.Index("\\BIN\\", SEARCH_BACKWARD);
```

   The declare keyword declares the variable *sProgram.* This line also assigns the variable *sProgram* to a new instance of the String class. *sProgram.Text* is assigned the value returned by IDE.ModuleName (the name of the currently executing module).

   The next line declares the variable *breakIndex.* This line also assigns *breakIndex* to the string returned by the Index method (the occurrence of the specified substring).

2. Enter the following code to store the path name in *sHelpDir.*

```
   declare sHelpDir = sProgram.SubString(0, breakIndex - 1).Text + "\\
HELP\\";
```

   This line declares the variable *sHelpDir.* It also assigns *sHelpDir* to the value returned by SubString *(breakIndex* using the specified starting and ending positions) plus the value in *sProgram.Text* plus the value `"\\HELP\\"`.

- You have just imported the *IDE* object, imported a symbol of a system-wide instance of *ScriptEngine,* created a class called *HelpMenu,* and added code that will find the name of the Help directory and store it in *sHelpDir.*
- In the next step, you will load MENUHOOK.SPP and declare two methods that add menu items to the Help menu.

Declaring methods that add menu items

(ObjectScripting Tutorial, Part 3: Step 2 of 5)

Next Previous Sample code

This step declares two methods:

- **AddHelpFile()** adds the Help file name to the Help menu and launches the Help file.
- **AddHelpFileFullPath()** uses *sHelpDir* to locate the full path name of the Help directory, then adds the Help file name to the Help menu and launches the Help file.

First, as you learned in Part 1, add code to load MENUHOOK.SPP. (For more information, see Part 1: Step 4; however, in your code, do not include the first curly brace shown in that step.)

To define **AddHelpFile()** and **AddHelpFileFullPath(),**

1. Add the following lines to your script:

```
AddHelpFile(menuText, helpFile)
        {
        AddHelpFileFullPath(menuText, sHelpDir + helpFile);
        }
```

   The first line declares **AddHelpFile(),** passing the arguments *menuText* and *helpFile.* The third line declares **AddHelpFileFullPath(),** passing the arguments *menuText* and *sHelpDir* plus *helpFile.* In both cases, *helpFile* is the Help file name without the path.

2. Add:

```
AddHelpFileFullPath(menuText, helpFile)
```

   Here, the parameter *helpFile* now includes the path.

- You have just declared methods that will add menu items to the Help menu.
- In the next step, you assign a menu item to the Help menu.

Assigning a menu item

(ObjectScripting Tutorial, Part 3: Step 3 of 5)

Next Previous Sample code

This step shows how to use the **assign_to_view_menu** function to assign a new menu item to the Help menu.

Note: When you load MENUHOOK.SPP, the **assign_to_view_menu** function automatically becomes available.

To assign a menu item to the Help menu,

1. Add the following lines to your script:

```
{
declare menuCmd =
"IDE.Help(\"" + AddBackSlash(helpFile) + "\", "
  + "3, " + "\"\"" + ");";
assign_to_view_menu("IDE", "&Help|" + menuText,
  menuCmd, menuText);
}
```

The first statement invokes a Help file using the IDE.Help method. The name of the invoked Help file is assigned to *menuCmd.* The second statement uses the assign_to_view_menu function to assign the new menu item to the Help menu.

- You have just assigned a menu item to the Help menu.
- In the next step, you will declare the **AddBackSlash()** function.

Addding a backslash to the path name

(ObjectScripting Tutorial, Part 3: Step 4 of 5)

Next Previous Sample code

Because cScript compiles a double backslash as a single backslash when it sends a file path to WinHelp, you need to add code that will add a double backslash to your script. This step declares the **AddBackSlash()** function, used in *menuCmd* (defined in the previous step).

To declare **AddBackSlash()**,

1. Add the following lines to your script:

```
AddBackSlash(fileName)
      {
      declare origFileName = new String();
      origFileName.Text = fileName;
      declare targetFileName = "";
      declare breakIndex = origFileName.Index("\\");
      while (breakIndex > 0)
         {
         targetFileName += origFileName.SubString(0, breakIndex - 1).Text
            + "\\\\";
         origFileName = origFileName.SubString(breakIndex);
         breakIndex = origFileName.Index("\\");
         }
      targetFileName += origFileName.Text;
      return targetFileName;
      }
};
```

The first declare statement declares the variable *origFileName.* This line also assigns *origFileName* to a new instance of a String object. In the next line, *origFileName.Text* is assigned to *fileName.*

The next **declare** statement declares the variable *targetFileName.* This line also assigns *targetFileName* an empty value. The last **declare** statement declares the variable *breakIndex.* This line also assigns *breakIndex* to the string returned by the Index method (the occurrence of the specified substring).

The while loop says that while *breakIndex* is greater than zero, give *targetFileName* the current value of *targetFileName* plus the value returned in SubString (the value of *breakIndex* specified by the starting and ending positions) plus `"\\\\"`. Then, assign *origFileName* the value returned by the SubString method (the value of the substring specified by *breakIndex).* The last line of the **while** loop assigns *breakIndex* the value returned by the Index method. If this value is greater than 0, the **while** loop executes again.

When *breakIndex* is equal to 0, *targetFileName* equals *targetFileName* plus the value returned by the Text property. The **return** statement exits the **AddBackSlash()** function, returning the value of *targetFileName.*

- You have just declared the **AddBackSlash()** function used in the **AddHelpFilePath()** method.
- In the next step, you will add the Help file names to the Help menu and run the script.

Executing Help menu methods

(ObjectScripting Tutorial, Part 3: Step 5 of 5)

Previous Sample code

This step shows how to execute **AddHelpFile()** and **AddHelpFileFullPath().**

To execute these methods,

1. Add the following lines to your script file:

```
declare helpMenu = new HelpMenu();
helpMenu.AddHelpFile("ObjectScripting", "SCRIPT.HLP");
helpMenu.AddHelpFileFullPath("Standard Template Library",
   "C:\\BC5\\HELP\\STL.HLP");
```

   The declare keyword declares the variable *helpMenu.* This line also assigns *helpMenu* to a new instance of the *HelpMenu* class. (Note the difference in case - cScript is a case sensitive language.)

   helpMenu.AddHelpFile assigns the value "ObjectScripting" to the parameter *menuText.* SCRIPT.HLP is assigned to the *helpFile* parameter. SCRIPT.HLP is launched when the Help|ObjectScripting menu item is selected.

   helpMenu.AddHelpFileFullPath assigns the value "Standard Template Library" to the parameter *menuText.* STL.HLP (and the full path name) is assigned to the *helpFile* parameter. STL.HLP is launched when the Help|Standard Template Library menu item is selected.

2. Save the script file and run it. For more information, see Part 1: Step 7.

   To see the results, go to the Help menu. Note that the menu items ObjectScripting and Standard Template Library have been appended to the bottom of the Help menu. Click Standard Template Library.

   The Contents topic of the Standard Template Library Help file is displayed.

3. To remove the Help files from the Help menu, exit Borland C++. When you load Borland C++ again, these help files will no longer display on the Help menu.

- You have just assigned new menu items to the Help menu, executed the associated Help files, and run the script file.
- You have now finished Part 3 of the tutorial.
- In Part 4, you will add all the Borland C++ Help files to the Help menu.

ObjectScripting Tutorial: Part 4

Next Sample code

This part of the tutorial teaches you how to:

- Declare a function that adds all Borland C++ Help files to the Help menu (Step 1)
- Execute the function (Step 2)

Sample code for Tutorial Part 4

```
// ObjectScripting example
// Copyright (c) 1996 by Borland International, All Rights Reserved
// STEP4.SPP: Add all menu items to the Help menu. Launch Help file
//            when menu item is selected.

import IDE;
import scriptEngine;

class HelpMenu()

{
// Find the help directory and store it in the static sHelpDir.
declare sProgram = new String();
   sProgram.Text = IDE.ModuleName;
   declare breakIndex = sProgram.Index("\\BIN\\", SEARCH_BACKWARD);
   declare sHelpDir = sProgram.SubString(0, breakIndex - 1).Text + "\\
HELP\\";

// Load "MENUHOOK.SPP", necessary for adding menu items.
     if(!scriptEngine.IsLoaded("menuhook.spp"))
          {
          scriptEngine.Load("menuhook.spp");
          }

// Add a menu item on the Help menu, launch the Help file
// associated with it. The helpFile parameter is the file name
// without the path.
     AddHelpFile(menuText, helpFile)
        {
        AddHelpFileFullPath(menuText, sHelpDir + helpFile);
        }

     AddHelpFileFullPath(menuText, helpFile)
        {
        declare menuCmd =
           "IDE.Help(\"" + AddBackSlash(helpFile) + "\", "
            + "3, " + "\"\"" + ");";
         assign_to_view_menu("IDE", "&Help|" + menuText,
            menuCmd, menuText);
         }

// Important note!
// The command text passed to assign_to_view_menu
// should be IDE:Help("C:\\BC5\\HELP\\...", ...).
// When cScript compiles, it compiles the double backslash
// as a single backslash. This routine adds a backslash to
// the directory path name.

   AddBackSlash(fileName)
      {
      declare origFileName = new String();
      origFileName.Text = fileName;
      declare targetFileName = "";
      declare breakIndex = origFileName.Index("\\");
      while (breakIndex > 0)
```

```
         {
         targetFileName += origFileName.SubString(0, breakIndex - 1).Text
            + "\\\\";
         origFileName = origFileName.SubString(breakIndex);
         breakIndex = origFileName.Index("\\");
         }
      targetFileName += origFileName.Text;
      return targetFileName;
      }
// This is a list of all the Help files in BC5.
// Comment out the ones you don't need.
AddStandardHelpFiles()
   {
   AddHelpFile("Borland C++ Tools", "BCTOOLS.HLP");
   AddHelpFile("Borland C++ User's Guide", "BCW.HLP");
   AddHelpFile("Borland Custom Controls", "BWCC.HLP");
   AddHelpFile("C++ Programmer's Guide", "BCPP.HLP");
   AddHelpFile("Class Library Reference", "CLASSLIB.HLP");
   AddHelpFile("Control 3D", "CTL3D.HLP");
   AddHelpFile("DOS Reference", "BCDOS.HLP");
   AddHelpFile("Error Messages", "BCERRMSG.HLP");
//   AddHelpFile("Formula One Visual Tools", "VTSS.HLP");
//   AddHelpFile("Help Author's Guide", "HCW.HLP");
//   AddHelpFile("Hot Spot Editor", "SHED.HLP");
//   AddHelpFile("HeapWatch32", "HW32.HLP");
//   AddHelpFile("MAPI Programmer's Reference", "MAPI.HLP");
//   AddHelpFile("MAPI Reference", "MM.HLP");
//   AddHelpFile("Message Compiler for NT", "MC.HLP");
//   AddHelpFile("MicroHelp Customer Controls", "VBT300.HLP");
//   AddHelpFile("OCF Reference", "OCF");
   AddHelpFile("ObjectScripting", "SCRIPT.HLP");
//   AddHelpFile("OLE 2 Reference", "OLE.HLP");
//   AddHelpFile("OLE 2.0 Object Viewer", "OLE2VIEW.HLP");
//   AddHelpFile("OLE Knowledge Base", "KBASE.HLP");
//   AddHelpFile("Open GL", "OPENGL.HLP");
   AddHelpFile("OpenHelp", "OPENHELP.HLP");
   AddHelpFile("OWL 5.0 Examples", "OWLEX.HLP");
   AddHelpFile("OWL 5.0 Reference", "OWL50.HLP");
//   AddHelpFile("Remote Procedure Call Reference", "RPC.HLP");
//   AddHelpFile("Resource Compiler for NT", "RC.HLP");
//   AddHelpFile("Resource Localization Manager", "RLMAN.HLP");
//   AddHelpFile("Resource Reference", "RC32.HLP");
   AddHelpFile("Resource Workshop", "WORKSHOP.HLP");
   AddHelpFile("TDWINI.EXE Information", "TDWINI.HLP");
   AddHelpFile("Standard Template Library", "STL.HLP");
   AddHelpFile("Visual Database Tools", "BCVDTREF.HLP");
   AddHelpFile("Windows 16 API", "WIN31WH.HLP");
   AddHelpFile("Windows 32 API", "WIN32.HLP");
   AddHelpFile("Windows 32s Reference", "WIN32S.HLP");
   AddHelpFile("Windows Developer's Guide", "GUIDE.HLP");
   AddHelpFile("Windows Interface Guidelines", "UIGUIDE.HLP");
   AddHelpFile("Windows System Class", "WINSYS.HLP");
//   AddHelpFile("WinSight", "WINSIGHT.HLP");
//   AddHelpFile("WinSpector", "WINSPCTR.HLP");
   }
};
```

```
// At load time, load a list of help files.
// Customize the list by modifying AddStandardHelpFiles() function.
declare helpMenu = new HelpMenu();
helpMenu.AddStandardHelpFiles();
```

Declaring a method

(ObjectScripting Tutorial, Part 4: Step 1 of 2)

Next Sample code

This step declares the **AddStandardHelpFiles()** method that adds all Borland C++ Help files to the Help menu. Because there are 42 Help files, you may want to comment out any Help files you don't think you'll use frequently.

First, start a script file and call it STEP4.SPP. Then, as previously learned:

- Import the *IDE* object (Part 2: Step 1)
- Import a symbol of a system-wide instance of *ScriptEngine* (Part 2: Step 2)
- Create a class called *HelpMenu* (Part 1: Step 3)
- Find the location of the Borland C++ Help directory and store it (Part 3: Step 1)
- Load MENUHOOK.SPP. (Part 1: Step 4). In your code, do not include the first curly brace shown in this step.
- Declare two functions that add menu items to the Help menu (Part 3: Step 2)
- Assign a menu item to the Help menu (Part 3: Step 3)
- Add a backslash to the Help directory path name (Part 3: Step 4). In your code, do not include the final curly brace and semi-colon shown in this step.

To declare the **AddStandardHelpFiles()** method,

1. Add the following lines to your script file:

```
AddStandardHelpFiles()
   {
   AddHelpFile("Borland C++ Tools", "BCTOOLS.HLP");
   AddHelpFile("Borland C++ User's Guide", "BCW.HLP");
   AddHelpFile("Borland Custom Controls", "BWCC.HLP");
   AddHelpFile("C++ Programmer's Guide", "BCPP.HLP");
   AddHelpFile("Class Library Reference", "CLASSLIB.HLP");
   AddHelpFile("Control 3D", "CTL3D.HLP");
   AddHelpFile("DOS Reference", "BCDOS.HLP");
   AddHelpFile("Error Messages", "BCERRMSG.HLP");
   AddHelpFile("Formula One Visual Tools", "VTSS.HLP");
   AddHelpFile("Help Author's Guide", "HCW.HLP");
   AddHelpFile("Hot Spot Editor", "SHED.HLP");
   AddHelpFile("HeapWatch32", "HW32.HLP");
   AddHelpFile("MAPI Programmer's Reference", "MAPI.HLP");
   AddHelpFile("MAPI Reference", "MM.HLP");
   AddHelpFile("Message Compiler for NT", "MC.HLP");
   AddHelpFile("MicroHelp Customer Controls", "VBT300.HLP");
   AddHelpFile("OCF Reference", "OCF");
   AddHelpFile("ObjectScripting", "SCRIPT.HLP");
   AddHelpFile("OLE 2 Reference", "OLE.HLP");
   AddHelpFile("OLE 2.0 Object Viewer", "OLE2VIEW.HLP");
   AddHelpFile("OLE Knowledge Base", "KBASE.HLP");
   AddHelpFile("Open GL", "OPENGL.HLP");
   AddHelpFile("OpenHelp", "OPENHELP.HLP");
   AddHelpFile("OWL 5.0 Examples", "OWLEX.HLP");
   AddHelpFile("OWL 5.0 Reference", "OWL50.HLP");
   AddHelpFile("Remote Procedure Call Reference", "RPC.HLP");
   AddHelpFile("Resource Compiler for NT", "RC.HLP");
   AddHelpFile("Resource Localization Manager", "RLMAN.HLP");
   AddHelpFile("Resource Reference", "RC32.HLP");
   AddHelpFile("Resource Workshop", "WORKSHOP.HLP");
   AddHelpFile("TDWINI.EXE Information", "TDWINI.HLP");
   AddHelpFile("Standard Template Library", "STL.HLP");
   AddHelpFile("Visual Database Tools", "BCVDTREF.HLP");
   AddHelpFile("Windows 16 API", "WIN31WH.HLP");
```

```
    AddHelpFile("Windows 32 API", "WIN32.HLP");
    AddHelpFile("Windows 32s Reference", "WIN32S.HLP");
    AddHelpFile("Windows Developer's Guide", "GUIDE.HLP");
    AddHelpFile("Windows Interface Guidelines", "UIGUIDE.HLP");
    AddHelpFile("Windows System Class", "WINSYS.HLP");
    AddHelpFile("WinSight", "WINSIGHT.HLP");
    AddHelpFile("WinSpector", "WINSPCTR.HLP");
    }
};
```

The **AddStandardHelpFiles()** method uses the **AddHelpFile()** method. Each **AddHelpFile()** method identifies:

- The Help file to display on the Help menu (the *menuText* parameter)
- The filename of the file to launch when the Help menu item is selected (the *helpFile* parameter)

- You have just declared a method that adds all Borland C++ Help files to the Help menu.
- In the next step, you will execute the method and run the script file.

Executing the Help menu method

(ObjectScripting Tutorial, Part 4: Step 2 of 2)

Previous Sample code

This step executes **AddStandardHelpFiles().**

To execute the Help menu method,

1. Add the following lines to your script file.

```
declare helpMenu = new HelpMenu();
helpMenu.AddStandardHelpFiles();
```

   The declare keyword declares the variable *helpMenu.* This line also assigns the variable *helpMenu* to a new instance of the *HelpMenu* class. (Note the difference in case - cScript is a case sensitive language.) *helpMenu.AddStandardHelpFiles* adds all Borland C++ Help files to the Help menu. A Help file is launched when a menu item is selected.

2. Save the script file and run it.

   To see the results, go to the Help menu. Note that many menu items have been appended to the bottom of the Help menu. Choose one.

   The Contents topic of the selected Help file is displayed.

3. To remove the help files from the Help menu, exit Borland C++. When you load Borland C++ again, these help files will no longer display.

- You have now finished Part 4 of the tutorial.
- For more information on ObjectScripting, click the Contents tab and browse through the Help topics. You can also look at the example programs in BC5\SCRIPT\EXAMPLES.

ObjectScripting overview

With ObjectScripting, you can customize the Borland C++ IDE programatically using built-in classes and a scripting language called cScript, a language much like C++. cScript supports classes, late binding, object-specific method overriding, and dynamic variable typing. Using cScript requires C++ or other object-oriented language experience.

Through an object called IDEApplication, which is instantiated when Borland C++ first starts up, you can access most parts of the IDE, including the Editor, the debugger, the keyboard, and the Project Manager. You can customize them to suit you, as well as add your own new features.

The following conceptual information tells you about ObjectScripting:

About running a script

About script loading

About script initialization

About script function referencing

About script debugging

About example scripts

The following tasks get you started with scripting:

Setting scripting options

Executing a script statement

Writing a script

Running a script

Debugging a script

Unloading a script

About running a script

See also

By convention, the source files for scripts have the extension .SPP. When you load a script for the first time, it is compiled into an interpreted tokenized format called pcode. By default, the tokenized file is created with the same name using the extension .SPX in the same directory as the script. The header in the .SPX file contains the original name of the file from which it was generated (the .SPP file) and the date/time stamp of the .SPP file when it was generated. Before executing a .SPP file, the dates are compared to ensure the source file has not changed. If it has, the. SPX file is regenerated.

If the script affects the display (for example, it contains **print** statements), you see something onscreen immediately. If you define new behavior for the IDE, you will see that behavior when you use that part of the IDE. The script remains loaded until you unload it.

You can use the following commands to run scripts:

Command	Description
Script|Run	Opens the Script Run window at the bottom of the IDE desktop, into which you enter a single script command. Executing a single script statement is useful when you are developing and testing a script.
Script|Compile	Compiles the file in the active Edit window. If the compile is successful, the script is loaded into the IDE and runs.
Script|Run File	Compiles, loads, and runs the file in the active Edit window. Use Script|Run File if your script contains a **breakpoint** statement.

Use Script|Commands to open the Script Commands dialog box which displays a list of the available script commands and variables, including classes, functions, and global objects. If an object is an instance of a class, its properties and methods are also displayed.

To run a script command,

1. Double-click a command from the list.
2. Enter the argument, if any, next to the selected command.
3. Click Run.

About script loading

See also Example 1 Example 2

You can load a script in any of the following ways:

- Choose the Script|Modules command. In the Script Modules dialog box, choose the module, or script, you want to load. Click Load. All loaded modules and all modules on your script path are listed in the Script Modules dialog box.
- Enter the name of the script in the Startup Scripts field in the Scripting Options dialog box. For example, enter `test`. To specify multiple scripts, separate script names with spaces.
- Specify a script on the BCW command line with the `-s` switch. The script is loaded after the complete processing of scripts specified in Scripting Options dialog box.
- Script names require no quotation marks.
- If you include script parameters, put the script name and parameters in quotation marks, or put the parameters in parentheses.
- To pass string parameters, enclose the strings in backslash-quotation combinations.
- To start multiple scripts, use the `-s` parameter for each script.
- Modify the source code of STARTUP.SPP (or any of the files that it loads). Note that when you update to a new version of Borland C++, you need to redo the changes to STARTUP.SPP.
- Create a script called PERSONAL.SPP in the Script directory. This script is automatically loaded after STARTUP.SPP finishes processing. PERSONAL.SPP can load other scripts, allowing multiple scripts to be loaded whenever the IDE starts. Using PERSONAL.SPP protects your script from being overwritten by new releases of Borland C++.

Note: To run a loaded script that has either an **_init()** function or a function with the same name as the script, choose the function name from the Script Commands dialog box.

Command-line startup example

```
//Starts three scripts from the BCW command line using the
//-s switch.
//Script3 shows how to start a script from the command line
//with optional parameters. Note that the script name and
//parameters are in quotation marks.
bcw -sScript1 -sScript2 -s"Script3 Param1 Param2"

//MyScript shows how to pass string parameters using
//backslash-quotation combinations.
bcw -sMyScript(\"string\", \"string\")
```

The advantage to starting the script from the command line is that the script will not be affected whenever you update to a new version of Borland C++.

Script startup example

```
//Starts three scripts - "test", "MyScript" and "bar" -
//from  the Startup Scripts field of the Scripting
//Options dialog box.
test MyScript bar
```

The advantage of loading a script from the Startup Scripts field of the Scripting Options dialog box is that script names can be shared by multiple Borland C++ users. Since a script's path is stored as part of the .SPX file, the script directory must be mapped to the same path for all users using the script.

However, every time you install a new version of Borland C++, you have to reenter script names.

About script initialization

See also Example

When you load a module into the IDE, script initialization takes place as follows: global commands are processed first, followed by the **_init()** function, if one exists. If an autocall function exists, it is processed last. Initialization is the order in which script commands and functions are processed.

Order processed	Description
Global commands	Script commands not in a function block.
_init() function	If a module contains an **_init()** function, it runs automatically, immediately after the global commands. If a series of scripts are loaded at the same time, first all the **_init()** functions are processed (left to right).
Autocall function	If a module contains a function with the same name as the file in which it resides (an autocall function), it will execute automatically, immediately after the global commands and the **_init()** function (if any).

The script initialization process lets you implement functionality without changing the STARTUP script, the IDE command line, or the Borland C++ configuration files.

Script initialization example

Assume you have written a script called HELLO.SPP that contains a function called **hello** declared as follows:

```
hello()
  {
    print "Hello World";
  }
```

When you load the script HELLO.SPP for the first time, the message `Hello World` displays in the Script page of the Message window and the **hello()** function stays in memory. If you subsequently choose Script|Run and type `hello()` in the Script Run window and press Enter, the script processor calls the function **hello()** which displays `Hello World` in the Message window.

About script function referencing

See also

When a function is referenced in a script, it is processed as follows:

1. All loaded modules (scripts) are searched for a matching function name. Searches are case sensitive (Test is not the same as test). The search starts with the module most recently loaded. If unsuccessful, the search continues to the next most recently loaded module.
2. If found, the function executes. If the function exists in more than one loaded module, the function located in the most recently loaded module is executed and other instances are ignored.
3. If the function is not found, the IDE checks an internal table constructed by calls to ScriptEngine.SymbolLoad. This table contains a list of scripts and the predefined symbols they contain. If the function is found in the table, the associated module is loaded into the IDE and the script runs.
4. If no matching function is found, the IDE searches the script path defined in the Scripting Options dialog box for a script file name that matches the function name.

- If a matching script file name is found, it is loaded into the IDE and the script runs.
- If no matching script file name is found, the IDE displays a message in the Script page of the Message window indicating that the function was not found.

Note: After the module is loaded, a second search for a function may be successful when the first search was not. For example, assume that a script file is found in the symbol table and gets loaded as a result of a function reference. The first search does not find the function, so the function does not execute. After the module is loaded, however, a second search finds the function in memory and it executes.

About script debugging

See also

You can debug scripts using one of the following techniques:

- Built-in diagnostics
- The breakpoint statement
- The print statement
- The Script|Run command

Built-in diagnostics

To force the cScripting environment to provide diagnostic messages and stop at breakpoints, you need to set the Scripting options Stop at Breakpoint and Diagnostic Messages. Stop at Breakpoint halts execution of a script at a breakpoint statement. Diagnostic Messages displays messages in the Script page of the Message window. For information on setting Scripting options, see the section Setting scripting options.

The breakpoint statement

When you enter a **breakpoint** statement into your script and the Scripting option Stop at Breakpoint is on, script execution halts and the Script Breakpoint Tool is displayed. The Script Breakpoint Tool allows:

- Stepping over or into function calls
- Evaluation of the values of expressions or script variables

The print statement

Use the print statement to display a value. Output from a **print** statement is displayed in the Script page of the Message window. Printed messages are placed into a queue which, when time allows, is moved into the view.

The Script|Run command

The Script|Run command opens the Script Run window at the bottom of the IDE desktop, into which you can enter a single script command. The results of the command are immediately displayed in the IDE, making results immediately available.

About example scripts

See also

Choose Script|Install/Uninstall Examples to load all examples in the BC5\SCRIPT\EXAMPLES directory. This command loads:

- All example scripts and makes them available in the Script Commands dialog box
- The Script Manager, a script that helps you work with the example scripts

To unload example scripts or the Script Manager, choose Script|Install/Uninstall Examples again and restart BCW.

Once the example scripts are loaded, choose the menu item Example Scripts to see a list of all example scripts. Choose the Example Scripts|Script Directory command to display the Script Directory window, where you can load, edit, and unload an example script, as well as edit the Script Manager data file.

Example scripts

The script examples directory contains the following types of scripts and script applets:

Script management

Editing

Coding

Debugging

Project management

Miscellaneous

Support classes and routines

Demonstration

Script management examples

Script	Description
LOADLAST.SPP	Load Last Script. Loads the last-loaded script. Useful for frequently reloading a script under development (before it is assigned to a hot key, menu, or some other quick trigger).
SPPMAN.SPP	Script Manager. Allows you to specify scripts for autoloading. Adds scripts to IDE menus. Displays the Script Directory window.
TEST.SPP	Test Harness. A template for inserting test code.

Editing examples

Script	Description
ALIGNEQ.SPP	Align at Equals. Aligns a block of assignments by positioning the equals operators one space after the longest lvalue in the current block.
APIEXP.SPP	API Expander. Expands current word in editor to the matching Windows API or C RTL signiture. Provides selection list if seed string has multiple matches. If the match is an RTL member, API Expander indicates if the corresponding header file needs to be added to the source file.
COMMENT.SPP	Commenter. Comments the selected block, or removes the comment if the lines are already commented.
EDITSIZE.SPP	Editor Size. Allows easy customization of Edit window size and position without changing default values in STARTUP.SPP. CONFIG.SPP provides a different but more comprehensive approach to positioning IDE windows.
EDONLY.SPP	Edit Only. Temporarily shows only those lines in the current buffer that contain a specified string. Useful for seeing how an identifier is being used, making changes without searching and replacing, isolating strings for spell-checking, etc.
SHIFTBLK.SPP	Shift Block. Shifts the current block right or left a column at a time.
SRCHALL.SPP	Search All. Searches and replaces across files in the current project.
TEMPLATE.SPP	BRIEF Template Support. Causes the IDE to use BRIEF template support. This support is used in all editor emulation.

Coding examples

Script	Description
CODELIB.SPP	Code Library. Displays libraries of code snippets you can insert in the current buffer. You can also edit code library data files, and create library entries from selected text. You can create as many code libraries as you want.
FILEINSR.SPP	File Insert. Inserts a file into the current buffer.
FINDTABS.SPP	Find Tabs. Searches all .C, .H, .CPP, .HPP, and .SPP files in the specified directory and reports all lines that have at least one tab character to the message database. Double-click a message to edit the referenced file. Useful for coding styles that don't use tab characters.
LONGLINE.SPP	Long Line Finder. Searches all .C, .H, .CPP, .HPP, and .SPP files in the specified directory and reports all lines that are longer than a given threshold value to the message database. Double-click a message to edit the referenced file.
OPENHDR.SPP	Open Header. Opens the .H or .HPP file corresponding to the current source file. Optionally creates a header file if one does not exist.
REVISIT.SPP	Code Revisit Tool. Quickly lists occurrences of a configurable "revisit this code" marker in all files in the specified directory.

Debugging examples

Script	Description
EVALTIPS.SPP	Evaluation Tips. When the debugger has a process loaded, evaluates the item under the cursor and displays the result in a mouse tip.
VIEWLOCS.SPP	View Locals. Inspects local variables if the debugger has a process.

Project management examples

Script	Description
LOADPROJ.SPP	Load Project. Opens the last project on startup.
PRJNOTES.SPP	Project Notes. For new projects, creates a notes text file in the project directory and adds it to the project.

Miscellaneous examples

Script	Description
AUTOSAVE.SPP	Autosave. Saves files, environment, desktop, project, and/or messages at the specified interval.
CONFIG.SPP	Configure Windows. Resizes and positions IDE windows as they are created. Also maps keys in the default and classic keyboards for buffer manipulation.
DIRTOOL.SPP	Directory Tool. Creates a new tool called Directory Listing, which takes a file specification and generates a directory listing in the Message window.
DIRVIEW.SPP	File Maintenance. Displays a directory listing and loads the following commands: Command — Description Backspace — Backs up one directory Delete — Deletes selected file or directory Enter — Changes to selected directory or opens selected file Insert — Creates a new file in the current directory Escape — Exits the directory listing. To make these commands the default, add the following to STARTUP.SPP: `scriptEngine.Load("dirview");`
FASTOPEN.SPP	Fast Open. Opens files and projects based on a search path, so you don't have to navigate directories.
KEYASSGN.SPP	Key Assignments. Shows what commands are assigned to a given key sequence.
NETHELP.SPP	Internet Help. Opens an URL with Netscape Navigator by selecting from a list of programming pages, FTP sites, and newsgroups.
SOUND.SPP	Sound Enabler. Plays WAV files on specified IDE events, such as Build Failure.

Support classes and routines examples

Script	Description
FILE.SPP	File Classes. Includes configuration file management.
FOREACH.SPP	For Each. Calls a function for all the nodes of the given type in a project.
MSG.SPP	Message Class. Provides methods to simplify and standardize user messages. Message captions automatically indicate the calling module.
MISC.SPP	Miscellaneous. Miscellaneous script.
SORT.SPP	Sort. Quick sorting routines.

Demonstration examples

Script	Description
AUTO.SPP	Automation. Demonstrates the IDE as an OLE automation controller and server.
CRTL.SPP	CRTL. Demonstrates script access to the CRTL by writing to a file.
INTNATL.SPP	International. Demonstrates the use of FormatString for localization of strings in scripts.
MODLIST.SPP	Module List. Demonstrates how to handle events from other objects to maintain the contents of a list. Implements some of the functionality provided by the Script Modules dialog box.
MLIST.SPP	Multi-select list window. Demonstrates a simple multiple-selection list window. Also shows how to position a popup window in the list.
LIST.SPP	List Window. Demonstrates a simple sorted list window.
POPUP.SPP	SpeedMenu. Demonstrates a simple SpeedMenu.

Script Directory window

See also

To display the Script Directory window, choose Example Scripts|Script Directory.

Note: To display the Example Scripts menu bar item, choose Script|Install/Uninstall Examples.

The Script Directory window consists of four columns of information:

Column	Description
Script Name	The name of script file in the BC5\SCRIPT\ EXAMPLES directory
Description	A brief description of what the script does
Autoload Status	Indicates whether the script is automatically loaded when BCW is started up
Load Status	Indicates whether the script is currently loaded

Click a script to display the Script Directory SpeedMenu. Commands on the SpeedMenu let you load, edit, and unload script files; edit the Script Manager script file; cancel the SpeedMenu; and close the directory.

Command	Description
Load	Loads the selected script file
Edit	Loads the selected script file into an Edit window
Unload	Unloads the selected script file
Edit Script Manager Data File	Loads the Script Manager data file, SPPMAN.DAT, into an Edit window
Cancel	Cancels the SpeedMenu
Close Directory	Closes the Script Directory window

Setting scripting options

See also

To set options for the scripting environment,

1. Choose Options|Environment|Scripting. The Scripting Options dialog box is displayed.
2. Set the following options:

Option	Description
Stop at Breakpoint	Stops the script when the keyword breakpoint appears. Loads the script debugger's Breakpoint Tool.
Diagnostic Messages	Specifies whether or not to display all script processor messages in the Script page of the Message window. By default, this option is off.
Startup Scripts	Specifies the script to load and execute as part of the IDE startup procedure. (Borland C++ always tries to load STARTUP.SPP from the SCRIPT subdirectory or any path you specify for scripts.) Use spaces to separate multiple script names. You can specify script parameters by enclosing the script name and its arguments in quotation marks. For example, `MyStartup DisplayCurProj "Ascript Param1"`
Script Path	Specifies the path to search when loading a script.

During a load, every entry on the path is searched for a file with the .SPX extension. If that fails, the same directories is searched a second time for files with the .SPP extension. Starting the path with `.;` causes the current directory to be searched first.

Executing a script statement

See also

To execute a script **print** statement and view it in the Script page of the Message window,

1. Choose View|Message. Click the Script tab to open the Message window Script page, where the output of all script **print** statements is directed.

   To start with a blank page, delete the existing messages by right clicking in the Script page and choosing Delete All.
2. Choose Options|Environment|Scripting and click Diagnostic Messages to send all scripting messages to the Script page.
3. Choose Script|Run. The Script Run window opens at the bottom of the IDE desktop.
4. Enter the following statement:

   ```
   print "Hello World";
   ```
5. Press Enter.

   `Hello World` is displayed at the end of the Message window.

   If you made an error entering the statement, error messages appear in the Script page of the Message window.

To execute a script **print** statement and display the output in a message box, instead of in the Message window, see: Displaying output in a message box.

Displaying output in a message box

See also

To display output in a message box, instead of in the Script page of the Message window,

1. Choose Options|Environment|Scripting and click Diagnostic Messages to send all scripting messages to the Script page.
2. Choose Script|Run. The Script Run window opens at the bottom of the IDE desktop.
3. Enter the following statement:

   ```
   IDE.Message("Hello World");
   ```

   The method IDEApplication.Message displays output in a message box instead of in the Message window.
4. Press Enter.

   `Hello World` is displayed in an information dialog box.

   If you made an error entering the statement, error messages appear in the Script page of the Message window.
5. Click OK to close the message box.

Writing a script

See also

Scripts are simply ASCII text files. You can use any text editor to write a script, then save it to a file with an .SPP extension. Header files for scripts typically have the extension .H. (Header files are used to define constants and provide for conditional compilation.)

Follow these steps to write a simple script:

1. Choose Options|Environment|Scripting.
2. Add your script directory to the Script Path so the IDE can find your scripts. For example, if your path already contains `.;C:\BC5\SCRIPT`, it would look like this after you add a directory called C:\MYSCRPTS:

   ```
   .;C:\BC5\SCRIPT;C:\MYSCRPTS
   ```

   Do not insert any spaces before your path name. Doing so will stop the search at the previous path.
3. While you're in the Scripting options dialog box, click Diagnostic Messages to send scripting error messages and **print** statement output to the Script page of the Message window.
4. Press Enter to exit the Scripting Options dialog box.
5. Choose View|Message and click the Script tab to open the Message window Script page.

   To start with a blank page, delete the existing messages by right clicking in the Script page and choosing Delete All.
6. Choose File|New|Text Edit to open a new file in the IDE editor. Enter the following script:

   ```
   import IDE;     //Use the IDE object and any of its methods
   hello()
   {
     IDE.Message ("Hello World");
   }
   ```
7. Choose File|Save and save the file with an .SPP extension in a directory of your choice (for example, C:\MYSCRPTS\HELLO.SPP).

Running a script

See also

To run the script you just created,

1. Choose Script|Run File.

   Script|Run File compiles the script, runs it, and loads it into the IDE.

2. `Hello World` is displayed in a message box.

   If you made an error entering the statement, error messages appear in the Script page of the Message window.

3. Click OK to close the message box.

Debugging a script

See also

To add a debug statement to a file and use the script debugger,

1. Choose View|Message. Click the Script tab to open the Message window Script page, where the output of all script **print** statements is directed.

   To start with a clear page, delete the existing messages by right clicking in the Script page and choosing Delete All.
2. Choose Options|Environment|Scripting and click Stop at Breakpoint to stop the script at the **breakpoint** statement and open the Script Breakpoint Tool.
3. Click Diagnostic Messages to send all scripting messages to the Script page.
4. Choose File|Open and open the Hello World file.

   The file should look like this:

```
import IDE;     //Use the IDE object and any of its methods
hello()
{
   IDE.Message ("Hello World");
}
```

5. Embed the keyword breakpoint in your source code before the line that starts with *IDE.Message.* The file should now look like this:

```
import IDE;     //Use the IDE object and any of its methods
hello()
{
   breakpoint;
   IDE.Message ("Hello World");
}
```

6. Choose File|Save.
7. Choose Script|Run File.
8. The Script Breakpoint Tool is displayed.

- Click Step Over to execute the call to *IDE.Message* without stepping into and executing it. Note that nothing happens since you are stepping over the line that displays output. Press Run to run the script to the end.
- Click Step Into to step into and execute *IDE.Message.* `Hello Word` is displayed in a message box. Press OK to close the message box.
- Click Run to continue full-speed execution of the script until the next **breakpoint** statement is encountered, or the script ends.
- Click Abort to cancel script execution and close the Script Breakpoint Tool.
- To immediately execute a line of code, enter the code into the Statement(s) edit box and click Execute. This lets you test code before you add it to your script.

Script Breakpoint Tool

See also

The Script Breakpoint Tool is a script debugging tool that lets you step through cScript statements and evaluate the values of expressions or script variables. The Tool is displayed when a breakpoint statement is encountered in an executing script.

Note: To display the Script Breakpoint Tool, the Stop at Breakpoint option in the Scripting Options dialog must be on.

Script function calls can either be stepped over or into, and the value of any variable visible within the context of the actively executing script can be evaluated. The name of the running script is displayed as well as the next statement to be executed and its line number.

When the Script Breakpoint Tool is active, output from print statements in the script itself continue to be sent to the Script page of the Message window. However, you can enter a **print** statement in the Immediate Mode Statement(s) edit box whose output is displayed in the Output box.

Script Breakpoint Tool options

Immediate Mode Statement(s)	The cScript statement to execute. Immediate Mode statements are executed in the context of the active script, as if the statement entered were actually in the script before the next line of the script about to be executed. Variables must be within scope in that context to be available for evaluation. In-scope variables can be both read and their values changed, although caution must be taken in changing them. Any function available to script at the time of execution can also be called, whether a local cScript function, an IDE object method, or an external library function from an active dynamic library. Care must be taken to ensure that the method is appropriate in a given context. If the statement is a print statement, its output is displayed in the Output box. In this way, an Immediate Mode Statement can be used to inspect the current value of script variables.
Execute	Executes the cScript statement in the Immediate Mode Statement(s) edit box.
Output	Displays the results of the statement executed in the Immediate Mode Statement(s) edit box. Output is displayed only if the statement is valid or if you have pressed the Execute button.
Run	Continues full-speed execution of the script until the next **breakpoint** statement is encountered or the script ends.
Abort	Stops script execution and closes the Script Breakpoint Tool.
Step Over	When the next executable statement is a call to a cScript function, executes the function call without stepping into and executing the function's statements.
Step Into	When the next executable statement is a call to a cScript function, steps into and executes the function's statements.
Help	Displays this Help screen.

Unloading a script

See also

Scripts are not unloaded automatically. To unload a script,

1. Choose Script|Modules.
2. In the Script Modules dialog box, choose the name of the script you want to unload.
3. Click Unload.

Scripts can also be unloaded by using unload.

When a script unloads, it looks for a function in the script called **~()** (the name of the function is simply a tilde). If this function is found, it is executed as part of the script unloading process and acts as a destructor for the script.

About cScript

The cScript language is a late-bound, object-oriented language that supports syntax and constructs familiar to the C++ developer - you declare classes and provide them with properties and member functions.

cScript offers C++ programmers a familiar environment for customizing the IDE. It has many of the same constructs as C++ and on the surface looks and feels like C++.

But under the hood the two languages are very different: They address two separate problem domains, the early-bound environment versus late-bound, and as a result there are some major semantic differences.

For more information, see:

About late-bound languages

See also

cScript is a late-bound, object-oriented language, which is roughly analogous to being an interpreted language. This gives cScript programs more flexibility than early-bound programs, such as those written in C++. In C++, everything about a program is known at compile time. The types of the variables, the return types and number of parameters to functions, the classes that will be used as well as all their properties and behaviors are all known when the program is compiled.

cScript is very different. While the syntax looks very similar to C++, you cannot declare a variable's type at compile time. Variables are generic and can hold any type of data needed at run time. In fact, the same variable can hold different types of data as the program executes.

Just as in C++, you create classes with properties and methods and create objects which are instances of those classes. But in cScript, you are free to override the methods for a given object (not the class, just the object itself) at run time with a new implementation of the method or a method "borrowed" from another object.

This means that an object of one class can use the methods of an object of another class without having to know anything about the second object at compile time. Existing objects can have their functionality extended without the need for the source code to the object's class, and without recompiling.

The benefits of late-binding

Late-binding provide important practical benefits. Let's say that you want to create a program to extend the functionality of the Borland C++ IDE. For example, you want to create a script that automatically saves changed source files to a central repository on the network as well as in your project directory. You want to add this functionality to the IDE and have it behave like a built-in feature.

The Borland C++ IDE is represented by a cScript object called *IDE* of the cScript class *IDEApplication.* If the object IDE was instead created from a C++ class, you would have to alter that C++ class and add your repository methods to it directly, through multiple inheritance, function pointers, or through some other mechanism. Then you would need to recompile the source for the class to create the extended object IDE. In cScript, you do not need to touch the definition of *IDEApplication* (the class) at all. You can use cScript to attach your repository methods to the IDE object at run time. There are no changes to the *IDEApplication* class and no recompilation is necessary.

So late-binding means that you can alter and extend the behavior of objects without having to know the details of how they are implemented, without having access to the source code, and without having to recompile.

Differences between cScript and C++

See also

cScript differs from C++ in the following ways:

- All class members are public. There is no way to make members private or protected as part of their declaration. You can use on statements to make members inaccessible.
- cScript programs have no **main()** or **WinMain()** function.
- Globally scoped statements are allowed and will be executed when the script is run.
- Executable statements are allowed within a class definition, and in conjunction with optional initialization arguments passed when the class is instantiated, constitute the class's constructor. There is no constructor function per se in cScript.
- The implementation of a class's methods are defined within the class. That is, the definition (not just the declaration) of a member function must always occur in the class declaration.
- Arrays are objects in cScript. When deallocating an array with the delete command, the square brackets are not needed.
- Functions may have varying numbers of parameters. cScript truncates or pads argument lists as necessary.
- Compound logical expressions do not short circuit. For example, in the expression `if(TRUE || Foo())...`, the function `Foo()` will always be called even though the constant **TRUE** insures that the expression will always evaluate to true.
- cScript does not have the following C++ features (this is not a complete list):
- Type checking (but there are type conversions with some operations). See cScript and types for more information.
- Type casting
- Multiple inheritance
- C++-style exceptions
- Class constructor functions
- Function overloading
- Character arrays (cScript directly supports strings)
- Default arguments to functions
- Templates
- Default parameters in method declarations
- Pointers
- Direct memory access
- Function declarations that support default parameters
- Enums
- Unions
- Structs or typedefs
- Bitfields
- Operator overloading
- The **const** keyword (except in DLL imports)
- The **static** keyword
- Global scope resolution. You can access globally scoped variables, using the module function
- The **#if** preprocessor directive (#ifdef is supported)
- The following operators: `->  *  ->*  .*`

cScript objects

See also

All objects in the IDE are exposed through the global object called *IDE,* in the class IDEApplication. This object is created in STARTUP.SPP, a script that is automatically executed when Borland C++ is started. You use *IDEApplication* to access many parts of the IDE. Additional classes provide access to the debugger, the search engine, the Editor, and the Keyboard Manager. Classes are also provided to create and manage list windows and pop-up menus.

In the Borland C++ IDE, all user commands are directly mapped to corresponding scripts. Every IDE window that uses the keyboard API has each keystroke mapped to a script. All main menu commands have a mapping to a script. These scripts, supplied by Borland, provide standard behavior that you can use to customize your environment. If you want to modify the behavior of the IDE, you can write scripts that interact with the exposed IDE components.

cScript and types

See also

cScript is not an explicitly typed language and does not allow you to declare variables with C++ base types. When the parser encounters an unknown identifier, it makes it a new variable (unless the identifier is immediately followed by an open parenthesis, which might indicate it's a function). New variables created this way are local to the current scope.

The only declarators you can use are **declare**, **import**, and **export**, which are not types but declarators that indicate a new variable. Declaring variables discusses **declare**, **import**, and **export**.

Identifiers do have types, but the type of an identifier is determined by its value. For example, *x* in the following code is an integer because it is assigned an integer:

```
declare x = 25;
```

x can become any other cScript native type, depending on what is assigned to it. In the following example, *x* is of type *IDEApplication* because an object of that class is assigned to it:

```
declare MyIDE = new IDEApplication;
x = MyIDE;
```

Use the intrinsic function typeid to determine the type of an identifier.

Type conversions

When you use operators with variables of different types, the simple conversion rule with binary operators (such as + and /) is that the operand on the left determines the type of the expression. For example,

```
declare x = 4;
declare y = 4.0;
print x/3; // output is 1
print y/3; // output is 1.333333
```

The rule becomes more complicated with conversions between strings and numbers because cScript does some interpretation.

- When converting from a number to a string, cScript represents digits as numeric strings (3 becomes "3").
- When converting from a string to a number, the string is converted to a number if the string can be interpreted as a number. If the string evaluates to anything but a number, it is converted to zero ("33" becomes 33, "33abc" also becomes 33, but "abc33" becomes 0). For example,

```
declare x = 10;
print "String" + x;  // prints "String10"
print x + "String";  // prints the result of 10 + 0 which is "10"
```

- If an object is converted to a string, it becomes the string "[OBJECT]". For example,

```
declare a = new IDEApplication;  // create a new
                                 // IDEApplication object
declare b = "Hello";             // create a new string variable
                                 // add the object to the string
                                 // converting the object to a string
declare c = b + a;
print c;                         // prints "Hello[OBJECT]"
```

Language elements

Choose one of the following sections for more information about cScript language elements:

Comments

Identifiers

Declaring variables

Statements

Operators

Strings

Arrays

Prototyping

Flow control statements

Classes

Event handling

Built-in functions

Reserved identifiers

Comments

See also

cScript supports C++ comment syntax, including:

- `// This is a comment to the end of the physical line`
- `/* This is a comment to the closing */`

Nested comments are permitted in cScript.

Identifiers

See also

Identifier names are made up of letters, digits and underscores (_). The first character of an identifier name cannot be a digit. Identifier names can be up to 64 characters in length.

cScript is case-sensitive. Therefore foo, Foo, and FOO are three different identifiers. Keywords, operators, and intrinsic function names are also case-sensitive.

Declaring variables

See also Example

A cScript source file (an .SPP file) is a module. A variable declared or used for the first time at the module level is global to that module, and a variable declared or used for the first time inside a block is local to that block.

Because you don't have to declare variables as you do in C++, it's easy to mistakenly use a global variable in a function or class when you intend it to be local. It's safest to use declare with variables that you intend to be local.

For an example, see: Local variable example.

Variables created at the module level (not in a function, method, class, control structure, or block) are global variables of the module. They are not normally accessible to other modules. To access a variable defined in module A from module B, three things must occur:

- Both module A and module B must be loaded.
- The variable must be declared export in Module A, at module scope.
- Module B must contain an import statement for the variable, at module scope.

Example

```
// This is an example of declaring a local variable
declare X = 2;          // Module scope X
declare Y = 4;          // Module scope Y

Func1(X){               // Parameter (local variable) X
  Y = "hello";          // modifies global Y.
}
Func2(X){               // Parameter (local variable) X
  declare Y = "hello"; // New local variable Y created
                        // and set to "hello".
}
```

Example

```
//This example shows how to declare a variable export in Module A
//and import in Module B.
Module A
  declare varOne;     //A global variable accessible only in Module A.
  export varTwo;      //A variable accessible outside Module A.

Module B
  import varOne;      //Trying to link with exported varOne. Will fail
                      //unless some other module exports varOne.
  import varTwo;      //Trying to link with varTwo in Module A.

  varOne = 33;        //Causes the run-time warning "Cannot locate
                      //external variable varOne".
  varTwo = 33;        //Changes the value of varTwo in Module A to 33.
```

Statements

See also

As in C++, statements must terminate with a semicolon. You can group multiple statements by surrounding them with braces. Variables declared within braces are local to those braces and go out of scope when the closing brace is reached.

You can chain expressions with the comma operator.

Strings

See also

cScript strings (note the lowercase "s") work much the same as C++ strings. A string is a series of characters delimited by quotation marks. In cScript, a string's length is limited to 4096. cScript automatically keeps track of the ends of strings; appending \0 (**NULL**) is unnecessary.

Unlike C or C++, you cannot access each character of the array independently using an offset of [] operators, as a string is not a pointer to memory. Internally, the variable assigned to the string represents the entire group of characters as a string. To access characters independently of each other, use a String object.

Because strings are stored as an entire group of characters you can:

- Add text together
- Check for equality, inequality, greater than, and less than

String formatting characters

cScript recognizes many C++ formatting characters within strings such as new line (\n) and horizontal tab (\t).

Besides the alphanumeric and other printable characters, you can designate hexadecimal and octal escape sequences much as you can in C++. These escape sequences are interpreted as ASCII characters, allowing you to use characters outside the printable range (ASCII decimal 20-126).

The format of a hexadecimal escape sequence is \x<hexnum>, where <hexnum> is up to 2 hexadecimal digits (0-F). For example, the string "R3" can be written as "\x523" or "\x52\x33".

Octals are a backslash followed by up to three octal digits (\ooo). For example, "R3" in octal could be written "\1223" or "\122\063".

Prototyping

See also

Forward referencing for functions and methods is not supported. Because scripts are interpreted in a single pass at run time, classes and the methods in them, must be defined before they can be used.

cScript does not provide a function prototype mechanism. This is because when the parser sees a function call, it needs to know the implementation at that time. At compile time, however, a C++ compiler only needs to be able to match the name, number of parameters, the types of the parameters, and the return values, but does not really need to know anything internally about the function.

Parameter counting and type conversions are performed at run time. cScript will pad (with NULLS) or truncate the argument list as necessary at run time to ensure that the correct number of arguments is available on the stack.

Flow control statements

See also

The following flow control statements work in cScript as they do in C++:

break	continue
do	else
if	for
return	while

The behavior of switch is slightly different. Because cScript is not a compiled language, the expression is checked against each case exactly as if evaluating an **if**-**else if** construct. This means that the cases need not be constants - they may be any expression (including function calls). It also means that if a default case is desired, it must be the last case.

For an example, see: Switch example

Example

```
// Switch example
switch( someNumber ) {
   case 3:             //Execution continues to case bar()
   case MyFunc():
      DoSomeStuff();
                       // No break. Even if this case executes,
                       // the next case is still evaluated.
   case W.Y.Z:
      DoSomethingElse();
      break;           // If this case executes, switch ends here.
   case 42:
      DoItAll();
   default:
   // Anything not matching previous cases comes through here
}
```

Pass by reference

See also

Parameters passed to methods and functions are passed by value unless explicitly made to be passed by reference. (Passing by value does not allow changes to the value of the caller's variable, while passing by reference does.) For example,

```
PassByValueFunction(aValueParameter){
  aValueParameter = 100; // Value of aValueParameter changed to
                         // 100. Caller's value unmodified.
}

PassByReferenceFunction(&aReferenceParameter){
  aReferenceParameter = 100; // Value of aReferenceParameter
}                            // changed to 100. Caller's value
                             // also updated
```

If you want to pass a variable by value in a pass-by-reference parameter, put it in parentheses. For example,

```
x = 10;
PassByReferenceFunction((x));
print x;   // Prints 10
PassByReferenceFunction(x);
print x;   // Prints 100
```

Built-in functions

See also

The cScript language provides the following built-in functions:

Function	Description
attach	Links a method of an instance of one class to a method of an instance of another class.
call	Directly invokes a closure.
detach	Detaches a method instance of one class from a method instance of the same or another class when the two were previously linked using **attach.**
FormatString	Formats strings at run time.
initialized	Indicates if a variable has ever been initialized.
load	Opens and parses the specified script.
module()	Gets access to any loaded module.
pass	Used in an **on** handler to invoke the original function that is being overridden.
print	Prints the specified expression in the Script page of the Message window.
reload	Does an **unload** followed by a **load.**
run	Loads and runs the module indicated.
select	Creates a special global variable, **selection,**that refers to the selected variable.
typeid	Gets run-time identification of variables or the resulting value of expressions.
unload	Unloads the specified module.
yield	Forces cScript to check if the abort (Esc) key has been pressed.

Reserved identifiers

See also

cScript reserves identifier names starting with two underscores as internal to the language. The following identifiers cannot be used in your scripts:

__break
__const
__cdecl
__error
__pascal
__refc
__rundebug
__runimmediate
__stack
__stdcall
__warn
event
Factory
false
FALSE
library
method
NULL
object
property
system
true
TRUE

cScript and DLLs

See also

Because all needed functionality is not directly available through the language or exposed by an object in the system, cScript allows you to access a function in a DLL directly through cScript by using code similar to the following:

```
// expose DLL entry points
import "foo.dll" {
 int __pascal FooFunc(short, char, unsigned, long);
 void DoIt();
}
// directly access the DLL calls
if (FooFunc(1, "hello there",2,3))
  print "FooFunc() succeeded";
else
  DoIt();
```

This DLL call uses the data type keywords **short**, **char**, **unsigned**, and **long**. Other data type keywords available for use in DLL calls are **void**, **int**, **bool**, and **const**.

This form of the import command allows you to declare a prototype for the external DLL functions, including their return types and arguments.

Unlike normal cScript functions, variable numbers of arguments are not supported when using functions from DLLs. You can pass dummy integer arguments for enums and pointers, since cScript does not support these types. There is no support for passing structs.

Note: When possible, declaring arguments of DLL calls with the **const** modifier will improve performance.

cScript supports the calling conventions **__cdecl**, **__pascal**, and **__stdcall**.

cScript and OLE2

See also

cScript to OLE2 interaction

If an automatable object has been exposed in the OLE2 registry, its functionality may be accessed from cScript by using the special *OleObject* class. For example,

```
// Creates an object with all the methods of
// Microsoft Word BASIC
wordBasic = new OleObject("word.basic");
// Call the Word BASIC function AppInfo() to find out
// what version of Word is installed
print wordBasic.AppInfo(2);     // Returns "7.0" for Word version 7.0
```

OLE2 to cScript Interaction

The IDE registers the automation name *BorlandIDE.Application* with the OLE2 registry during initialization. From any automation controller, the IDE's functionality may be accessed by creating a *BorlandIDE.Application* object and using it. For example, from a *Visual* dBASE program you could do the following:

```
* Visual dBASE syntax:
BorCppIDE = NEW OleAutoClient("BorlandIDE.Application")
BorCppIDE.ProjectOpenProject("foo.ide")
IF(BorCppIDE.ProjectBuildAll())
  BorCppIDE.FileSend("success notification")
ELSE
  BorCppIDE.FileSend("failure notification")
ENDIF
```

Arrays

See also

cScript supports two types of arrays:

- Bounded arrays
- Associative arrays

Bounded arrays

See also

cScript bounded arrays are similar to C++ arrays and are declared with a size specifier. Run-time warnings occur if you attempt to access a bounded array out of bounds. Bounded arrays use a zero-based index; that is, the first element of an array is element 0 and the last element is element *size* - 1.

You can declare a bounded array by using either of the following syntax variations:

```
x = new array [10];
array x[10];
```

Access is then as you would expect:

```
x[0] = 5;
x[1] = "a string";
x[2] = Foo;
x[3] = x[2];
```

You can also declare and initialize a bounded array using the following:

```
z[] = {"one", "two", x};  //Note the use of braces, {},
                          //rather than brackets, [].
```

In this case, *one*, *two,* and the value of *x* are the values in the array, and the array indexes start at 0 and go to 2. For example,

```
print z[0];   //Prints one
print z[1];   //Prints two
print z[2];   //Prints the value of x
```

You cannot initialize variables in an array initialization list: You must initialize them elsewhere. For example, you cannot define an array as follows:

```
z = {x=1, y=3, slogan="No more woe"} //Illegal syntax
```

In this array definition, assignments to *x*, *y*, and *slogan* must be elsewhere in your code.

Note: Be careful not to unintentionally overwrite an existing array with a new one during initialization. The following example declares an array "a", but then overwrites it with the elements *1, red* and *2.*

```
declare array a[10];  // declares an array with 10 elements
// The next line destroys the array "a" and declares
// a new array with three initialized elements
a = {1, "red", 2};
```

You can assign values beyond array bounds. Such an assignment does not increase the size of the bounded array to match the new index, but rather declares an associative array that is attached to the original bounded array. You can use any value as the new index.

For an example, see: Associative array attached to bounded array example.

Note: You cannot use a negative number to index into an array. Doing so causes a run-time warning.

Example

```
// This script generates no errors or warnings. It declares
// and initializes a bounded array with 4 elements (0 - 3)
declare a = new array[4];
a[0] = "Able";
a[1] = "Baker";
a[2] = "Charlie";
a[3] = "Delta";
print a[0];  // prints Able
print a[1];  // prints Baker
print a[2];  // prints Charlie
print a[3];  // prints Delta

// The following lines seem to add an element to the bounded array
// on the fly, but are actually declaring an associative array and
// appending it to the bounded array. Since the index is contiguous
// with the indices of the bounded array, the new element can be
// used as if it were part of the bounded array.
a[4] = "Edward";
print a[0];  // prints Able
print a[1];  // prints Baker
print a[2];  // prints Charlie
print a[3];  // prints Delta
print a[4];  // prints Edward

// The following lines add an element to the associative array.
// The new element's index is not contiguous with the existing
// elements. Note: adding element a[6] does NOT declare element a[5].
a[6] = "Frank";
print a[0];  // prints Able
print a[1];  // prints Baker
print a[2];  // prints Charlie
print a[3];  // prints Delta
print a[4];  // prints Edward
print a[5];  // prints [UNINITIALIZED]
print a[6];  // prints Frank

// The following lines add an element to the associative array using
// a string as an index. Adding this element has no effect on the rest
// of the array.
a["Bob"] = "Robert";
print a[0];      // prints Able
print a[1];      // prints Baker
print a[2];      // prints Charlie
print a[3];      // prints Delta
print a[4];      // prints Edward
print a[5];      // prints [UNINITIALIZED]
print a[6];      // prints Frank
print a["Bob"];  // prints Robert

return;
```

Associative arrays

See also

You declare associative arrays without a size specifier and access them on demand. They grow as required. Associative arrays are typically sparse and do not perform as well as bounded arrays.

To declare a new associative array, use one of the following syntax variations:

```
z = new array[];
array z[];
```

Associative arrays can take string as their indexes as well as numbers. Typically, the index of an associative array element is something which is related to the data the element holds. For example:

```
History = new array[];
History["President"] = "Bill Clinton"
History["Vice President"] = "Al Gore"
History[1776] = "U.S. Independence"
History[1789] = "U.S. Constitution"
```

You also declare an associative array when you make assignments beyond the bounds of a bounded array. For more information, see the section Bounded arrays.

Note: You cannot use a negative number to index into an array. Doing so causes a run-time warning.

Classes

See also

cScript supports single inheritance. There is no support for overloaded methods (member functions). In addition, there is no hiding of members: all properties (member data) and methods are public and virtual. You can override an instance of a class (an object) with on and pass, and you can bind objects' events (function calls) together in an event handling chain using **attach**. For more information, see the section Event handling.

Defining methods

All methods must be defined entirely in the class definition. A class definition may be nested in another class definition. The name of that nested class exists in the scope of the outer class, and is thus protected from accidental collision with identifier names in the module and global scopes. You can instantiate a nested class with the following syntax:

```
// Class Inner is nested in class Outer
class Outer {
   class Inner {}
}
declare Outer outerInstance;
declare innerInstance = new Inner from outerInstance;
```

Modifying the behavior of methods and properties

You can modify the behavior of methods in script classes:

- Derive a new class from the script class, overriding the methods whose behavior you want to change. Use this technique when you want to provide new behavior for a collection of objects.
- Override an instance of a class by using an on handler or attach to hook one of the object's methods. Use this technique when you want to tweak the behavior of a particular instance of a class.

You can also modify the behavior of properties in script classes:

- Derive a new class from the script class, overriding the properties whose values you want to change. Use this technique when you want to provide new behavior data values for a collection of objects.
- Override an instance of a property by using getters and setters. Use this technique when you want to tweak the behavior of a particular instance of a class.

Declaring a class

See also Example 1 Example 2 Example 3

There are no constructors in cScript as there are in C++. (Defining a method with the same name as the class, as you do in C++, does not make it a constructor.) Instead, code embedded in the class declaration that is not part of a method declaration is executed for each object instantiated from the class, and is therefore treated as constructor code. For this reason, constructor arguments must be defined in the class declaration.

Member functions must be defined entirely in the class declaration. You cannot declare a member function in a class and then define it later in the program.

There are destructors in cScript, and they work as they do in C++. (Defining a method that starts with a tilde (~) and has the same name as the class makes it a destructor.) Destructors are called when the object is being destroyed.

Example

```
// The following class is declared without parameters
class noParams{
  declare aMember;
  declare anotherMember;
  // Constructor code is any code outside of class methods
  Func1();                      // call to a module-level method
  for (declare y=1; y<10; y++) // more constructor code
     print "hello";

  // Here is the destructor:
  ~noParams(){
     print "A noParams has been destroyed.";
  }
  // More constructor code.
  print "noParams construction completed";
};

x = new noParams;  // declare instance, run constructor code
x = 0;             // calls destructor
```

Example

```
// The following class is declared with parameters
class Base(parmOne, parmTwo)
  print "parmOne=", parmOne;
  print "parmTwo=", parmTwo;
  declare X = parmOne;
  declare Y = parmTwo;
  MethodOne() {
    X = X + Y;
  }
  AnotherMethod() {
  }
}

// aParm and cParm are passed through to
// Base as ParmOne and parmTwo.
class Derived(aParm, bParm, cParm): Base(aParm, cParm) {
  declare Z = bParm;
};

declare d;
d = new Derived (1,2,3);
print "d.X = ", d.X;
print "d.Y = ", d.Y;
print "d.Z = ", d.Z;
```

Example

```
//The following class is inherited from the class Base
// aParm and cParm are passed through to Base
class Derived(aParm, bParm, cParm): Base(aParm, cParm) {
  declare Z = bParm;
};
```

Note: Initialization arguments must be explicitly passed to the base class. They must also be stated in the derived class parameter list because that is the list referenced when a derived class object is instantiated.

Creating instances of cScript classes

See also

Objects in script are created in one of two ways (assuming an already defined class Foo):

```
x = new Foo();
```

or

```
Foo x();
```

As with any declaration, you can use the declare and export keywords when you create objects. For example,

```
declare x = new Foo();
export Foo y();
```

cScript has automatic garbage collection. When an object goes out of scope, it is deallocated. Objects can be explicitly deallocated using the delete command. For example,

```
declare x = new Foo(); // allocate new object
delete x;              // explicitly delete object
```

Because cScript is untyped, you can destroy an object by assigning it another value. For example, cScript does not complain when you assign 0 to the object *x* as follows:

```
declare x = new MyClass();  // create an object of class MyClass
x = 0;                      // object overwritten and replaced with 0
```

Note: The object is only actually destroyed if the reassigned variable is the only reference to that object. If there are additional references to the object, the object will continue to exist when one of its reference variables is reassigned.

Discovering class and array members

You can use ?? and iterate to discover the contents of classes and associative arrays.

- With ?? you can test if a particular property exists in an object or if a particular index exists in an array.
- With **iterate**, you can see all members of a class or array.

Closures

See also

Closures let you obtain a reference to a method or property without invoking it. They are analogous to function pointers in C++.

Closures are powerful features of cScript. You can pass a closure as a function argument, for example. Since it represents a member of a class instance (an object), it carries a this pointer for that object with it and has all of the object's context information.

Use the closure operator (**:>**) in the following situations:

- To bind a class instance (an object) with one of its methods in a single reference.
- To assign a closure to a variable and use that variable anywhere you would use the closure.
- To dynamically expand or change the interface or behavior of a particular object, in ways not specified in the class of which the object is an instance.
- To declare arrays of closures to use like arrays of function pointers. The functions need not do anything unless they happen to be defined. Calling an undefined closure is not an error - nothing happens because there's nothing to call.
- In on handlers and attach and detach statements to handle a method call.
- In both cases, you can use pass to call the original method (if any) from within the attached method, and control the parameters passed to the original method. Overriding or adding an object method using an **on** handler or an **attach** statement affects only the one object instance. It does not affect the class, or any existing or new objects instantiated from that class. Only when **on** handlers are defined within a class definition itself using the **this** reference do all objects of that type inherit that event handling behavior.

Event handling

See also

cScript uses an event handling model to override class behavior. Given an instance of a class, you can modify its behavior by hooking a specified method and supplying an alternative implementation. You can use either an **on** handler or **attach** and **detach** to accomplish this. Go to one of the following for more information:

On handlers

Attach and detach

On handlers

See also

You can use an on handler to hook method call events for an instance of a class and override, or enhance, its functionality. You need not call the hooked method inside the **on** handler: Any code in the **on** handler will be executed instead of the hooked method. If you want to invoke the original method, use pass. If the hooked method returns a value, that or any other value can be returned by assigning the return value of **pass** to a local variable, including a return statement in the event handler.

In the **on** handler header, you use the closure operator (`:>`) to bind a class instance (an object) with a method of the object as a closure reference. For example,

```
Declare AClass MyObject;  // or MyObject = new Aclass;
// Given this instance of class AClass, you
// can intercept one of its methods.
on MyObject:>Method1(parm1){
  // Programmer may provide some preprocessing here.
  // Programmer may delegate to original implementation
  // or get original return value with pass().
  Declare rv = pass(parm1); // call MyObject.Method1(parm1)
  // Programmer may provide some postprocessing here.
  return rv;
}
```

Note: To be bound to an existing object method, the number of parameters in the **on** handler definition must match the hooked method. Once invoked, however, **pass** will call the hooked event regardless of how many arguments it passes. As with all function calls, cScript will ensure that the proper number of arguments are passed, truncating or padding as needed.

While inside an **on** handler, keep the following in mind:

- You aren't actually in a method of the object. Simple function calls resolve to their global counterparts, not to the object's methods. If you want to call the method *bar* from the *Method1* **on** handler, you must explicitly denote the object. For example,

```
on MyObject:>Method1(){
  MyObject.bar();
}
```

- Another way to explicitly denote a method of this object is to use the shorthand *dot notation*, which relies on the fact that, in an **on** handler, the dot is a shortcut for the controlling object. For example (given an object *MyObject* that has methods *Method1* and *bar*):

```
on MyObject:>Method1(){
  .bar();
}
```

Attach and detach

See also Example

If you want to make dynamic changes to class instances, you can set up dynamic **on** handlers using the closure operator with attach and detach. An on handler is not dynamic, but stays in effect once established as long as the module in which it is defined remains loaded and as long as the object exists.

Attached closures are used to set up a linkage between any member (method or property) of an instance of one class with any member from an instance of another class.

Example

```
// attach and detach example
x = new Foo();        // Create an instance of Foo called x
                      // and assume Color() is a method.
x.Color();            // Call x.Color().
y = new Bar();        // Create an instance of Bar called y
                      // and assume Notify() is a method.
y.Notify();           // Call y.Notify().
attach y:>Notify to x:>Color; // When x.Color() is called,
                      // instead call y.Notify().
x.Colo r();           // Call y.Notify().
// NOTE: In y.Notify() a pass() will
// now delegate back to x.Color().
detach y:>Notify from x:>Color; // unlink the two objects
x.Color();                     // Call x.Color().
```

Accessing cScript properties

See also

You can use **on** handlers to control what happens when users get (read) or set (write) the values of properties. These two types of **on** handlers are called **getters** and **setters**. This feature allows you to execute some code when a property is accessed instead of having to implement the property as a method.

Using getters

Using setters

Using getters

See also Example

You can use a **getter:**

- To restrict access to a property
- To execute related methods or modifying related properties
- To perform computations on a value before returning it

The syntax for a **getter** is:

```
on object:>property{
  [optional pre-processing statement(s)]
  return [pass()|SomeValue];
}
```

Since no value is passed to the on handler, no parameter is needed. You need a **return** statement because a **getter** is always invoked when the object's property is used in a statement that needs to obtain its current value. When you access the property (for example, on the right side of an assignment operator or as an argument in a print statement), the on-read property event handler is called and its statements are executed.

Example

```
// The following getter hides the property Hidden1:
import IDE;               //Import IDE, an IDEApplication object
class MyClass () {
 declare Hidden1 = "Hidden: can't see this one";
 declare Public1 = "Public: can see this one";

// Getter
 on this:>Hidden1 {
   return NULL;
 }
} // End MyClass declaration

getter() {
 declare MyClass myobj;
 IDE.Message (myobj.Hidden1);
      //Prints nothing
 IDE.Message (myobj.Public1);
      //Prints "Public: can see this one"
}
```

Using setters

See also Example

You can use a **setter:**

- To restrict values of a property to a certain range
- To limit access to a property (or even make it read-only)
- To execute related methods or modify related properties
- To perform computations on a value before setting it
- To convert user-supplied data to an internal format

The syntax for a setter is:

```
on ClassInstance:>property(parameter){
 [optional pre-processing statement(s)]
 [pass(parameter | SomeValue);]
 [optional post-processing statement(s)]
}
```

Unlike the **getter** syntax, parentheses and a parameter are required for the **setter** to obtain the value intended to be assigned to the hooked object property. If you want the handler to be able to set the property (rather than simply block write access to it), you need a pass statement that sets the property's value. When you try to set the property (for example, when the property is used on the left side of the assignment operator `object.property = 1`), the on handler code executes.

Example

```
// In the following example, the setter uses the value set
// in radius to calculate and set the values of circumference
/  and area. It then passes the user's value on to radius.

import IDE;    //Import IDE, an IDEApplication object
declare PI = 3.141592654;

class Circle(rad) {
 declare radius = rad;
 declare circumference;
 declare area;

 // Setter
 on this:>radius(x) {
 if (x > 0) {
   circumference = PI * 2 * x;
   area = PI * x * x;
   pass(x);
  }
  else
   IDE.Message("Error: Radius must be greater than zero.");
 }

 // Methods
}

 ShowProperties() {
   IDE.Message("radius = " + radius +
               ",     circumference = "
               + circumference +
               ",     area = " + area);
 }
}     // End of Circle class declaration

declare Circle obj(1);    //Initialize radius to 1.
obj.ShowProperties();

//Call the IDEApplication method SimpleDialog to prompt
//the user for input and get a value for radius.
declare radius = IDE.SimpleDialog("Enter a radius", "10");

obj.radius = 0 + radius; //Convert string to integer
obj.ShowProperties();
```

Adding menu items and buttons to the IDE

See also

Through cScript, you can add menu items to a view's SpeedMenu or to menus on the main IDE menu, and define buttons that can be added to the IDE SpeedBar. This functionality is contained in the file MENUHOOK.DLL, located in the Borland C++ BIN directory. A script called MENUHOOK.SPP is provided in the Borland C++ SCRIPT directory to enable these capabilities through cScript.

To use its functions, MENUHOOK.SPP must be loaded using the load command or through the Script Modules dialog box. To automatically load MENUHOOK.SPP each time you start the IDE, add the following line to STARTUP.SPP:

```
scriptEngine.Load("menuhook");  // load the MenuHook functions
```

MENUHOOK functions

The following table lists the MENUHOOK functions:

Function	Description
assign_to_view_menu()	Adds a menu item to a menu
remove_view_menu_item()	Removes a menu item from a menu
define_button()	Defines a button that can be added to the SpeedBar

assign_to_view_menu

See also Example

Creates a new menu item on a SpeedMenu or on a main IDE menu.

Syntax

```
int assign_to_view_menu(string view_type, string menu_text,
                        string script_text, string hint_text);
```

view_type	Defines the type of view to attach this menu item to. Supported values are: IDE, Editor, and Project. Passing IDE creates a new menu bar item on the main IDE menu. The other values attach the menu item to the SpeedMenus of views of the given type.
menu_text	The words that will appear on the menu item. If you include an ampersand (&) in the string, the character following the ampersand will be underlined and will be the selection character for the menu item. Menu items can be nested by putting a pipe (|) between the words of the menu items.
script_text	cScript statement(s) to be executed when the menu item is selected.
hint_text	The text to display in the status bar when the menu item is highlighted.

Return value

1 if the menu item is successfully added, 0 otherwise

Description

menu_text should be unique for the menu. Built-in menu items cannot be replaced using this function. Defining a menu item with *view_text* identical to that of a menu item previously defined with **assign_to_view_menu()** will replace the original menu item with the new one.

A one-level submenu can be created by specifying a *menu_text* with a pipe (|) character between the menu text and the menu item text. By using the same menu text to the left of the pipe with different menu item text to the right of the pipe in several calls to **assign_to_view_menu()**, you can create a submenu with several menu items.

Editor or project views that are visible when **assign_to_view_menu()** is executed will not have their menus updated. By adding calls to **assign_to_view_menu()** in STARTUP.SPP, you can customize the IDE's menu system from the time it starts up, and assure that all views will have the customized menus.

Menu items can be removed from SpeedMenus using remove_view_menu_item(). Menus created on the IDE menu bar cannot be removed without exiting the IDE.

Example

```
// Loads the MENUHOOK functions if not loaded in STARTUP.SPP
load("menuhook");

// This function call adds a single menu item
// to the editor view's SpeedMenu.
assign_to_view_menu("Editor", "&Click Me",
                    "IDE.Message(\"I'm clicked!\");",
                    "Click this menu item to see a message");

// These function calls add a submenu to the project
// view's SpeedMenu with three menu items.
assign_to_view_menu("Project", "Ne&w Menu | &First Item",
                    "IDE.Message(\"Clicked the first item\");",
                    "This is the first submenu item");
assign_to_view_menu("Project", "Ne&w Menu | &Second Item",
                    "IDE.Message(\"Clicked the second item\");",
                    "This is the first submenu item");
assign_to_view_menu("Project", "Ne&w Menu | &Third Item",
                    "IDE.Message(\"Clicked the third item\");",
                    "This is the first submenu item");

// These function calls add a menu pad to the
// main IDE menu bar with three menu items.
assign_to_view_menu("IDE", "E&xample | &First Item",
                    "IDE.Message(\"Clicked the first item\");",
                    "This is the first submenu item");
assign_to_view_menu("IDE", "E&xample | &Second Item",
                    "IDE.Message(\"Clicked the second item\");",
                    "This is the first submenu item");
assign_to_view_menu("IDE", "E&xample | &Third Item",
                    "IDE.Message(\"Clicked the third item\");",
                    "This is the first submenu item");
```

remove_view_menu_item

See also Example

Removes a menu item from the specified view's SpeedMenu.

Syntax

```
int remove_view_menu_item(string view_type, string menu_text);
```

view_type	Defines the type of view the menu item is attached to. Supported values are: Editor and Project.
menu_text	The words that appear on the menu item to be removed. This includes the ampersand (&) denoting the selection character, if any. If a menu and menu item were defined using a pipe (|) in *menu_text* in the call to assign_to_view_menu() that created the menu/menu item, then the exact same text, including the pipe, are required in this function as well.

Return value

1 if the menu item is successfully removed, 0 otherwise

Description

menu_text must exactly match the string used in the *menu_text* argument in **assign_to_view_menu()**.

Menus and menu items created with **assign_to_view_menu( )** can be removed. Menus and menu items on the IDE menu bar can also be removed.

When removing menus with multiple menu items, **remove_view_menu_item()** must be called for each item in the menu.

Example

```
// These function calls remove the SpeedMenu menus and menu
// items created with the assign_to_view_menu() example.

// Removes menu item from the editor view's SpeedMenu
remove_view_menu_item("Editor", "&Click Me");

// Removes the submenu from the project view's SpeedMenu
remove_view_menu_item("Project", "Ne&w Menu | &First Item");
remove_view_menu_item("Project", "Ne&w Menu | &Second Item");
remove_view_menu_item("Project", "Ne&w Menu | &Third Item");
```

define_button

See also Example

Defines a new SpeedBar button.

Syntax

```
int define_button(string button_name, string script_text,
                  string hint_text, string tooltip_text,
                  int button_index);
```

button_name	Defines a name for this button. The name should not conflict with any of the built-in button names. Multiple buttons with the same name can be defined.
script_text	cScript statement(s) to be executed when the button is selected.
hint_text	The text to display in the status bar when the mouse pointer rests on the button.
tooltip_text	The tip text to display when the mouse pointer rests on the button.
button_index	The index of the glyph to show for the button. MENUHOOK.DLL contains a built-in set of 38 glyphs (numbered 0 through 37) that can be used for buttons.

Return value

1 if the button successfully added, 0 otherwise

Description

Defining a button with **define_button()** adds the button to the Available Buttons list in the Options|Environment|SpeedBar|Customize dialog box. Use this dialog to add the button to the button bar.

User-created button definitions are automatically saved to the IDE configuration file when the IDE shuts down, and reloaded when the IDE starts.

Example

```
// Creates a new button definition and adds it to the
// Available Buttons list so it can be added to the
// SpeedBar.
define_button("Example Button",
  "IDE.Message(\"You pressed the example button\");",
  "This is the example button",
  "Example Button", 4);
```

Keywords and functions

{button A,JI(``,`cScr_Keywords_a')}{button B,JI(``,`cScr_Keywords_b')}{button C,JI(``,`cScr_Keywords_c')}{button D,JI(``,`cScr_Keywords_d')}{button E,JI(``,`cScr_Keywords_e')}{button F,JI(``,`cScr_Keywords_f')}{button I,JI(``,`cScr_Keywords_i')}{button L,JI(``,`cScr_Keywords_l')}{button M,JI(``,`cScr_Keywords_m')}{button N,JI(``,`cScr_Keywords_n')}{button O,JI(``,`cScr_Keywords_o')}{button P,JI(``,`cScr_Keywords_p')}{button R,JI(``,`cScr_Keywords_r')}{button S,JI(``,`cScr_Keywords_s')}{button T,JI(``,`cScr_Keywords_t')}{button U,JI(``,`cScr_Keywords_u')}{button V,JI(``,`cScr_Keywords_v')}{button W,JI(``,`cScr_Keywords_w')}{button Y,JI(``,`cScr_Keywords_y')}

Keywords and functions are reserved for use in the cScript language and cannot be used as names of variables, methods, or classes or as any other identifier names.

The following list shows the cScript keywords and functions in alphabetical order. It does not include words from the list of reserved identifiers or preprocessor directives.

A

array

attach

B

break

breakpoint

C

call

case

char

class

continue

D

declare

default

delete

detach

do

E

else

export

F

for

FormatString

from

I

if

import

int

initialized

iterate

L
load
long

M
module
module()

N
new

O
of
on

P
pass
print

R
reload
return
run

S
select
selection
short
super
switch

T
this
to
typeid

U
unload
unsigned

V
void

W
while
with

Y
yield

array

See also Keywords Example

Declares an array.

Syntax 1

```
declare array_var = new array[ [size] ];
declare array array_var[ [size] ];
```

size The number of elements in the bounded array. If *size* is omitted, the array is associative.

Syntax 2

```
array_var[ [size] ] = {element1[, element2[, ...]] };
```

size An array created with this syntax always takes the number of elements in the declaration list. *size* is ignored.

element1... Creates a bounded array with contents *element1, element2,* and so on. Element numbering starts at 0 and continues to *size* - 1. The number of elements determines the size of the array and overrides *size* if it is specified.

Description

In cScript, you can create two types of arrays, bounded and associative:

- Bounded arrays are similar to C++ arrays. As in C++, they use a zero-based index. (The first element is 0 and the last is *size* -1.) If you create an array with a list of elements, as in Syntax 2, it is a bounded array and its size is the number of elements.
- Associative arrays are grown as needed. If you assign more members to a bounded array than its *size*, the rest of the array becomes an associative array.

Arrays can contain data of any cScript type, including objects and other arrays. An array with other arrays as elements is multidimensional. Elements of the contained arrays are accessed using additional sets of square brackets as shown in the example.

Example

```
// Creates a bounded array of 10 elements
declare myArray;
myArray = new array[10];
myArray[1] = "Hello";
myArray[2] = "World";
print myArray[0], myArray[1];      // prints "Hello World"

// Creates an associative array
declare myAssocArray;
myAssocArray = new array[]         // no size declared
myAssocArray["Element1"] = "One";
myAssocArray["Element2"] = "Two";
print myAssocArray["Element2"]     // prints "Two"

// Creates a multidimensional array
declare array multiArray[] = {{1,2,3}, myArray, myAssocArray};
print multiArray[0][2], multiArray[1][0], multiArray[2]["Element2"];
// Prints: 3 Hello Two
```

attach

See also Keywords Example

Links a method of an instance of one class to a method of an instance of another class.

Syntax

```
attach ClassInst1:>method1 to ClassInst2:>method2
```

Description

To make dynamic changes to class instances, you can set up dynamic function call event handlers (also called on handlers) using the closure operator with **attach**. This technique allows you to supply an alternative implementation for an instance method.

In other words, you can override an object's method and provide an alternate implementation of that method at runtime, without affecting the class from which the object was instantiated. The override remains in effect for the lifetime of the object or until the link is broken using detach.

This binding is on a per-instance basis unless you use the **attach** statement in the class definition with the this reference in place of a specific instance name.

Example

```
// Attaches a method belonging to a String object (myStr1)
// to an EditStyle object (myStyle);
declare myStr1, myStr2, myStyle;
myStr1 = new String("HELLO WORLD");
myStyle = new EditStyle("Example");

// Attaches myStr1's Lower() method to myStyle
attach myStr1:>Lower to myStyle:>Lower;

// Calls Lower() from myStyle
myStr2 = myStyle.Lower();
print myStr2.Text;               // prints "hello world"

// Detaches Lower() from myStyle
detach myStr1:>Lower from myStyle:>Lower;
myStr2 = myStyle.Lower();
print myStr2.Text;              // prints [UNINITIALIZED]
```

break

See also Keywords

Passes control to the first statement following the innermost enclosing brace.

Syntax

```
break;
```

Description

Use **break** within a:

- do loop
- while loop
- for loop
- iterate loop
- switch construct

The implementation of **break** in cScript is identical to the implementation in C++.

breakpoint

Keywords

Stops the program and passes control to the script debugger Breakpoint Tool.

Syntax

```
breakpoint;
```

Description

If the Breakpoint Tool is not active, **breakpoint** is ignored.

call

Keywords Example

Directly invokes a closure.

Syntax

```
call ClosureName(argumentList);
```

ClosureName	The name of the closure.
argumentList	The arguments for the method or property being invoked.

Description

The closure is invoked using the same arguments as the method normally uses. There is no method for obtaining a return value when calling through closures. If the method returns a value, it will be ignored.

Example

```
// Shows creating a closure and assigning it to a
// variable, then calling the closure directly.

Class MyClass {
  method1(p1, p2)
  {
    print p1, p2;
  }
};

declare MyClass instance;
declare closure = instance:>method1; // declare the closure
call closure("Hello", "world");      // output is Hello world
```

case

See also Keywords

Determines which statements to execute in a switch statement.

Syntax

```
switch ( switch_expression ){
  case expression :
    [statement1;]
    [statement2;]
    ...
    [break;]
  [default :
    [statement1;]
    [statement2;]
    ...]
}
```

switch_expression	Any valid cScript expression, including a function call. Unlike C++, the *switch_expression* is evaluated for each **case** in a top-down fashion until a match is found or no more **case** statements remain.
expression	Any valid cScript expression, including a function call.
statement	One or more statements to execute.

Description

A **case** statement is the branch condition of a **switch** statement. If the value of the *expression* following **case** matches the value of *switch_expression*, the statements up to the next break or the end of the **switch** execute.

Note: Because cScript is a late-bound language, *expression* does not have to be a literal as in C++, nor does the *expression* have to be of integral type. Otherwise, **case** behaves exactly as it does in C++.

class

See also Keywords Example 1 Example 2

Defines a cScript class.

Syntax

```
class className [(initializationList)]
   [:baseClassName[(initExpressionList)]]
   {memberList}[;]
```

className	The name of the class. *className* can be any name unique within its scope
initializationList	The initial constructor values for the class, if any.
baseClassName	The class that this class derives from (optional). **of** is a synonym for the **:** separator preceding this identifier.
initExpressionList	The initialization for the class instance.
memberList	Declarations of the class's properties, methods, and events.

Description

A class declaration in cScript is similar to a class declaration in C++, with a few key differences.

For example, defining a method with the same name as the class, as you do in C++, does not make it a constructor. Instead, executable statements embedded in the class declaration that are not part of a method declaration is considered constructor code. For this reason, initialization parameters must be defined in the class declaration. The base class is always initialized first, before the child class.

Only one base class can be initialized in a derived class declaration because cScript does not support multiple inheritance. Where a class is defined as being derived from a base class and the base class requires initialization values, they must be passed to the base class through the derived class's declaration. The base class initializer is essentially an implicit constructor call, and as such, expressions are allowed for its arguments.

When instantiated, the number and type of initializers is not checked (function overloading is not supported in cScript). Arguments are padded and/or truncated the same as they are with functions.

Methods must be defined entirely in the class declaration. You can't just declare a member function in a class and then define it later in the program. All properties and methods of the class are public.

Destructors in cScript work as they do in C++. Defining a method that starts with a tilde (~) and has the same name as the class makes it a destructor. Destructors are called when the object is being destroyed. Destructors may not have parameters.

Where inheritance is used, the access method for base class members is the same as for those of the derived class. However, if a derived class member has the same name as one of the base class, you must use super to clearly specify the reference.

Note: You cannot instantiate a class as part of its declaration as in traditional C structs, so a semicolon is optional at the end of the declaration.

Example

```
//The following class is declared without parameters:
class noParams{
  declare aMember;
  declare anotherMember;
  Func1();          // constructor code
  for (y = 1; y < 10; y++) // more constructor code
   print "hello";
  ~noParams(){
   print "A noParams has been destroyed.";
  }
}
// The following class is declared with parameters:
class Base(parmOne, parmTwo){
  declare X = parmOne;  // a member variable
  declare Y = parmTwo;  // a member variable
  MethodOne(){
   X = X + Y;
  }
  AnotherMethod(){
  }
}
```

Example

```
// The following class is inherited from the class Base:
// aParm and cParm are passed through to
// Base as parmOne and parmTwo.
class Derived(aParm, bParm, cParm) : Base(aParm, cParm) {
  declare Z = bParm;
}
// example using the Derived class:
declare obj = new Derived(1, 2, 3)  // 1&3 passed to Base
                                    // Base constructed before
                                    // Derived
```

continue

See also Keywords

Passes control to the end of the innermost enclosing brace, allowing the loop to skip intervening statements and re-evaluate the loop condition immediately.

Syntax

```
continue;
```

Description

Use **continue** within a:

- do loop
- while loop
- for loop
- iterate loop

The implementation of **continue** in cScript is identical to the implementation in C++.

declare

Keywords Example

Declares a variable and ensures that it is local to the current scope and does not override a variable from an enclosing scope.

Syntax

```
declare identifier [optional identifier_syntax][, identifier…];
```

identifier The variable being declared.

identifier_syntax The variable's default values. *identifier_syntax* is optional.

Description

The scope of a variable is the block in which it is first used and any blocks nested in that block. While in a nested block, it is possible that a variable you think you are using for the first time has already been used in the enclosing block. What happens in that case is that you override the enclosing block's variable value (and possibly its type as well) with what you mistakenly think is a local variable.

To ensure that this doesn't happen, use **declare** with any variables that are local to a block. Although not needed, **declare** can also be used in conjunction with the export and import declarators. Note that you can declare multiple basic variables, objects, and arrays in a single statement, but you cannot mix them in the same statement.

See array and new for specifics on declaring arrays and class objects.

Example

```
// Examples of declare
declare x;
declare x = 1;
declare x, y, z;
declare x = 1, y, z = 2;
```

default

See also Keywords

Provides statements to process in a switch statement when none of the case conditions apply.

Syntax

```
switch ( switch_expression ){
  case expression :
    [statement_list;]
    ...
    [break;]
  [default :
    [statement_list;]
    ...]
}
```

switch_expression	Any valid cScript expression, including a function call. Unlike C++, the *switch_expression* is evaluated for each **case** in a top-down fashion until a match is found or no more **case** statements remain.
expression	Any valid cScript expression, including a function call.
statement_list	A list of statements to execute.

Description

default is optional. If you include a **default** statement, it must be the last condition in the **switch**. If you do not include a **default** statement and none of the **case** conditions apply, none of the statements in the **switch** are executed. The behavior of **default** in cScript is the same as C++.

delete

Keywords

Deallocates an object and causes the object destructor, if any, to be called.

Syntax 1

```
delete object_name;
```

object_name The name of the object to delete.

Syntax 2

```
delete array_name;
```

array_name The name of the array to delete. Deleting an array does not require square brackets in the **delete** command, as it does in C++.

Description

Unlike C++, cScript has automatic garbage collection. Therefore, objects are automatically deleted when there are no longer any references to them, or when they go out of scope. Use **delete** only when you need to explicitly deallocate an object before the references to that object have been destroyed.

detach

See also Keywords Example

Detaches a method instance of one class from a method instance of the same or another class when the two were previously linked using attach.

Syntax

```
detach ClassInst1:>method1 from ClassInst2:>method2
```

Description

To make dynamic changes to class instances, you can set up dynamic function call event handlers (also called on handlers) using the closure operator with attach. This technique allows you to supply an alternative implementation for an instance method.

In other words, you can override an object's method and provide an alternate implementation of that method at runtime without affecting the class from which the object was instantiated. The override remains in effect for the lifetime of the object or until the link is broken using **detach.**

Example

```
// Attaches a method belonging to a String object (myStr1)
// to an EditStyle object (myStyle);
declare myStr1, myStr2, myStyle;
myStr1 = new String("HELLO WORLD");
myStyle = new EditStyle("Example");

// Attaches myStr1's Lower() method to myStyle
attach myStr1:>Lower to myStyle:>Lower;

// Calls Lower() from myStyle
myStr2 = myStyle.Lower();
print myStr2.Text;               // prints "hello world"

// Detaches Lower() from myStyle
detach myStr1:>Lower from myStyle:>Lower;
myStr2 = myStyle.Lower();
print myStr2.Text;              // prints [UNINITIALIZED]
```

do

See also Keywords

Executes the specified statement until the value of the specified condition becomes **FALSE**.

Syntax

```
do statement while ( condition );
```

statement The statement to be executed. *statement* executes repeatedly as long as the value of *condition* remains **TRUE**.

condition Either **TRUE** or **FALSE.** When **FALSE,** *statement* stops executing.

Description

The behavior of **do** in cScript is the same as C++. break terminates loop execution, while continue evaluates *condition* immediately without executing any intervening statements.

Note: Because *condition* is tested after *statement* is executed, the loop executes at least once.

export

See also Keywords Example

Provides access to a variable across modules.

Syntax

```
export variable_name;
```

variable_name The name of the variable to export.

Description

Declare the variable as **export** in the module that declares it and import in another module that needs access to it.

Variables created at the module level (not in a function, method, class, or control structure) are global variables of the module, but are not accessible to any other modules. To access module scope variables defined in module A from module B, three things must occur:

- Both module A and module B must be loaded.
- The module scope (global) variable must be declared **export** in Module A.
- Module B must contain an **import** statement for the variable.

Example

```
// Example of export and import
// FILE1.SPP
export myExVar;  // export variable for use in other modules
myLocal = 10;
myExVar = 10;

// FILE2.SPP
import myExVar;  // import variable exported by another module
print myLocal;   // prints [UNINITIALIZED]
print myExVar;   // prints 10
```

for

See also Keywords

Executes the specified statement as long as the condition is **TRUE.**

Syntax

```
for ( [initialization] ; [condition] ; [expression] ) statement
```

initialization Initializes variables for the loop. *initialization* can be an expression or a declaration. Variables are initialized before the first iteration of the loop.

condition Must evaulate to either **TRUE** or **FALSE.** When **FALSE,** *statement* stops executing.

expression The expression to evaluate after each iteration of the loop. *expression* usually increments or decrements the initialization variable in some way.

statement The statement to be executed. *statement* executes repeatedly as long as the value of *condition* remains **TRUE**.

Description

The behavior of **for** in cScript is the same as C++. *statement* executes repeatedly as long as *condition* is **TRUE.** The scope of any identifier declared within the **for** loop extends to the end of the script module.

Note: Because *condition* is tested before *statement* is executed, the loop may never execute.

The cScript **for** statement works the same as a C++ **for** statement.

All the parameters are optional. If c*ondition* is left out, it is assumed to be always **TRUE**. break will cause loop execution to be terminated, while continue will cause the condition to be evaluated immediately without executing any intervening statements.

FormatString

Keywords

Formats strings at run time.

Syntax

```
FormatString("formatString" [, expression1 [, expression2…]]);
```

formatString	Literal text, placeholders for values, or a combination of the two. A placeholder is in the format of "%*n*", where *n* is the number representing the place of the expression in the list following the format string.
expression	Any valid cScript expression (literals, variables, function calls, etc.). Note that numeric values are automatically converted to strings.

Return value

The string created by combining the *formatString* and the variable list.

Description

Use **FormatString** to build strings at runtime using a formatting string and a list of cScript expressions.

For example:

```
declare str = "Hello";
declare value = 10;
print FormatString("str = %1, value = %2", str, value);
// the string "str = Hello, value = 10" is printed
```

In the above example, the value of *str*, the first variable in the list, was substituted for %1 in the output string. Likewise, the value of *value*, the second variable in the list, was substituted for %2 in the output string.

The number of variables in the variable list must match the number of placeholders in *formatString*.

from

See also Keywords Example

Used in a detach statement or when instantiating nested classes.

Syntax 1

```
innerObject = new Inner from ClassInstance;
```

Inner The nested class.

ClassInstance Instance of the enclosing class.

Syntax 2

```
detach ClassInst1:>method1 from ClassInst2:>method2;
```

Example

```
// Attaches a method belonging to a String object (myStr1)
// to an EditStyle object (myStyle);
declare myStr1, myStr2, myStyle;
myStr1 = new String("HELLO WORLD");
myStyle = new EditStyle("Example");

// Attaches myStr1's Lower() method to myStyle
attach myStr1:>Lower to myStyle:>Lower;

// Calls Lower() from myStyle
myStr2 = myStyle.Lower();
print myStr2.Text;               // prints "hello world"

// Detaches Lower() from myStyle
detach myStr1:>Lower from myStyle:>Lower;
myStr2 = myStyle.Lower();
print myStr2.Text;              // prints [UNINITIALIZED]
```

if

See also Keywords

Implements a conditional statement. **if** works exactly as it does in C++.

Syntax 1

```
if ( condition )  statement;
```

condition Must evaulate to either **TRUE** or **FALSE.** When **FALSE,** *statement* stops executing.

statement The statement to be executed. *statement* executes repeatedly as long as the value of *condition* remains **TRUE**.

Syntax 2

```
if ( condition )  statement;
   else  statement2;
```

condition Must evaulate to either **TRUE** or **FALSE.** When **TRUE,** *statement* executes. When **FALSE,** *statement2* executes.

statement The statement to execute. *statement* executes repeatedly as long as the value of *condition* remains **TRUE**.

else An optional keyword. If you use nested **if** statements, any **else** statement is associated with the closest preceding **if** unless you force association with br aces.

statement2 The second statement to execute. *statement2* executes when the value of *condition* is **FALSE**. *statement2* can be another **if** statement.

Description

Use **if** to implement a conditional statement.

You can declare variables in the condition expression. For example:

```
if (int val = func(arg))
```

is valid syntax. The variable *val* is in scope for the **if** statement and extends to an **else** block when it exists.

The condition statement must convert to a bool type. Otherwise, the condition is ill-formed.

When `<condition>` evaluates to **TRUE,** `<statement1>` executes.

If `<condition>` is **FALSE,** `<statement2>` executes.

The **else** keyword is optional, but no statements can come between an **if** statement and an **else.**

import

See also Keywords Syntax 1 Example Syntax 2 Example

Allows access to a variable across modules.

Syntax 1

```
import variableName;
```

variableName The name of the variable to import.

Syntax 1 Description

Declare the variable as export in the module that declares it and **import** in another module that needs access to it.

Variables created at the module level (not in a function, method, class, or control structure) have module scope. They are not accessible to any other modules. To access a variable defined in module A from module B, three things must occur:

- Both module A and module B must be loaded.
- The variable must be declared **export** in Module A.
- Module B must contain an **import** statement for the variable.

import is also used to make functionality contained within a Windows DLL file available from within cScript.

Syntax 2

```
import "DLL_Name" {functionPrototypes}
```

DLL_Name	The name of the DLL you wish to use. The path can be included if necessary.
functionPrototypes	Each external function must be prototyped according to general C++ prototype conventions. DLL calls use the data type keywords **char**, **short**, **int**, **unsigned**, **long**, **bool**, **void** and **const**.

Syntax 2 Description

Makes functions contained in external DLLs available to cScript.

Unlike normal cScript functions, variable numbers of arguments are not supported when using functions from DLLs. You can pass **int** arguments for **enums**, and **long** for **pointers**, since cScript does not support these types. There is no support for passing **structs**.

cScript supports the calling conventions, **__cdecl**, **__pascal**, and **__stdcall**.

Example

```
// Example of export and import
// FILE1.SPP
export myExVar;  // export variable for use in other modules
myLocal = 10;
myExVar = 10;

// FILE2.SPP
import myExVar;  // import variable exported by another module
print myLocal;   // prints [UNINITIALIZED]
print myExVar;   // prints 10
```

Example

```
// This example exposes DLL entrypoints using import
import "foo.dll" {
 int __pascal FooFunc(short, char, unsigned, long);
 void DoIt();
}
// directly access the DLL calls
if (FooFunc(1, "hello there", 2, 3))
  print "FooFunc() succeeded";
else
  DoIt();
```

initialized

See also Keywords Example

Indicates if a variable has ever been initialized.

Syntax

```
initialized(x);
```

x The name of the variable to check.

Return values

TRUE if the value has ever been initialized, **FALSE** otherwise

Description

initialized is an intrinsic function that provides a means for determining the state of a variable before using it. Using an uninitialized variable is almost never as dangerous as in C++, but is also usually not what was intended.

initialized is particularly useful in determining the state of arguments passed to functions on call, and in class instantiation, and can also be used to prevent unintended divide by zero errors because of an uninitialized divisor.

Example

```
// Example of initialized
declare x, y;  // declares variables,
               // but does not initialize them!
x = 10;        // initialized!
print initialized(x);  // returns TRUE
print initialized(y);  // returns FALSE
```

iterate

See also Keywords Example

Use an **iterate** loop to cycle through the members of a class object or an associative array in first to last order.

Syntax

```
iterate(outputvar; object[;keyvar]) [statement];
```

outputvar A variable to hold a copy of the contents of the array or class data member

object The array or class object to iterate

keyvar Variable to hold the index or key into the array, or class object data member name

statement The statement to be executed.

Description

iterate is a loop structure that allows some action, such as printing, to be performed on each member of the array or property of a class object.

You can use continue and break to control execution inside the loop. Like a for loop, curly braces ({}) must be used to enclose multiple loop statements.

iterate can also be used to determine the number of properties in an object or the number of elements in an array.

Example

```
//Prints all the members of associative array z
//using the variable x
iterate( x; z ) {
 print x;
}

//Prints all the members and the key values of
//associative array z using the variable x
iterate( x; z; k ) {
 print "Key = " + k + "Value = " + x;
}
```

load

See also Keywords Example

Opens and parses the specified script file.

Syntax

```
moduleHandle = load(fileName);
```

fileName The name of the script file to open and parse.

Return value

A module handle (module object reference) if successful, or **NULL** if not.

Description

Once the script file is opened and parsed, **load** executes the file using run. Although classes and functions defined in a module come into existence when the module is loaded, variables declared in the module are not defined, nor are any other statements executed, until the script is run.

If there is an **_init** function, the module executes that code first. If there is a function with the same name as the module, that function is then executed.

Example

```
//Loads and runs a script file
declare myModule;
myModule = load("demo.spp");  // loads module and gets a handle
if (myModule) {               // if loaded
  run(myModule);              // run the module
unload(myModule);             // unloads the module
}
```

module command

Keywords

Provides an alternative internal name, or alias, for a module.

Syntax

```
module ["newName"];
```

newName The module's alternative name.

Description

After being parsed, every script file loaded into the IDE is assigned a module name. The name defaults to the file name without its path or file extension. This name may be used by other modules to explicitly access functionality in the module.

You can alter a module's name by embedding the following anywhere in the file:

```
module "newname";
```

module function

Keywords

Gets access to any loaded module.

Syntax

```
module (["moduleName"]);
```

moduleName The name of the module to get.

Return value

If *moduleName* is not specified, returns one of the following:

- A reference to an object
- The module handle associated with the named module
- The module handle associated with the current module

If *moduleName* is specified and no matching module is available or no parameter is entered, it returns **NULL.**

Description

Use **module** to get access to any loaded module. If you use it with the current module, *moduleName* has the same value as this used at the module level.

One use for this function is to access a globally scoped variable from a local scope. For example:

```
// Modtest.spp
declare x = 1;
declare ModRef = this;
local x = 2;
print (module()).x; // prints 1
print ModRef.x;     // prints 1
```

new

See also Keywords

Creates a new object or array.

Syntax 1

```
objectname = new className[([initializerList])]
             [from outerClassName[([initializerList])]]
```

initializerList The list of objects used for initializing this class.

Syntax 2

```
arrayname = new array [[arraySize]];
```

arraySize The size of the array.

Description

Use **new** as an alternate syntax for creating new class objects or arrays. For more information, see class, array, and declare.

Unlike C++, cScript does not distinguish between static and dynamic memory allocation. The difference between the standard declaration syntax and that using **new** is syntactic only.

cScript has automatic garbage collection. Therefore, objects created with **new**, or otherwise, are automatically deleted when there are no longer references to them (that is, when these objects and any variables that reference them go out of scope). Use delete only when you need to explicitly deallocate an object before the references to that object have been destroyed.

of

See also Keywords

A synonym for the colon (:) separator used when defining a class that derives from a base class.

Syntax

```
class classname [ (initialization_list) ]
   [of baseClass[ (initialization_list) ]] { member_list }
```

initializationList	The initial constructor values for the class, if any.
member_list	Declarations of the class's properties, methods, and events.

on

See also Keywords Example

Sets up one of the following:

- A dynamic object method call event handler, also called an on handler (syntax 1)
- An object read-property getter (syntax 2)
- A write-property setter (syntax 3)

Syntax 1

```
on ClassInstance:>{xe ">"}Method([argumentList]){
  [pre-processing statement(s)]
  [pass([argumentList]);]
  [post-processing statement(s)]
  [return [value];]
}
```

Syntax 1 Description

This syntax is used for an object method call event handler. This form of dynamic event handling allows processing to occur both before and after the optional call, through pass to the hooked method. It also allows alternate values to be both passed to the hooked method and returned by the event handler.

Note: In order to be bound to an existing object method, the number of parameters in an **on** handler definition must match the hooked method. Once invoked, pass will call the hooked method regardless of how many arguments it passes. As with all function calls, cScript will ensure that the proper number of arguments are passed, truncating or padding as needed.

Syntax 2

```
on ClassInstance:>property{
  [pre-processing statement(s)]
  return [pass() | value];
}
```

Syntax 2 Description

This syntax is used for a property **getter** and would be triggered by any subsequent statement that references that object's property for read access, such as on the right hand side of an assignment statement.

This form of the syntax allows **pass** to return the actual value, or, alternatively, any specified value.

Syntax 3

```
on ClassInstance:>property(parameter){
  [pre-processing statement(s)]
  [pass(parameter | value);]
  [post-processing statement(s)]
}
```

Syntax 3 Description

This syntax is used for a property **setter.** The setter is triggered when the object's property is used as an lvalue, such as on the left hand side of an assignment statement. The value to be assigned to the property is what is passed to the **setter** as its parameter. The value passed in **pass** sets the value of the property.

Description

Use **on** handlers (also referred to as object method call event handlers) to create new methods, or redefine existing methods, on an object of a given class.

Unlike **attach,** methods overriden with **on** cannot be detached. To call the original method from within the overriden version with the same name, invoke the **pass** function. **on** handlers can be defined to

control both read and write access to an object's properties.

Note: If the global reference variable selection has been set using select, its reference will not be affected, but is superseded with the with block.

Example

```
import editor;
// Create a new Debugger object called debug
declare debug = new Debugger();

// Create a new method called RunToCurrent()
// on the object debug (not the class!)
on debug:>RunToCurrent()
{
   declare fileName = editor.TopBuffer.FullName;
   declare row = editor.TopBuffer.TopView.Position.Row;
   .RunToFileLine(fileName, row);
}
```

pass

Keywords

Used in an on handler to invoke the original function that is being overridden.

Syntax

```
varname = pass([param1[,param2[,...]]);
```

print

See also Keywords

Prints the specified expression in the Script page of the Message window.

Syntax

```
print [expression_list];
```

expression_list The list of expressions to print.

Description

print takes any string, expression, or variable as a parameter. To concatenate expressions, separate them with commas. For example:

```
print "hello world";
print "the number is", x;
print "My name is", name, "and I'm", years, "years old";
```

A space is printed for each comma in the expression list. If no expressions are passed, **print** does nothing.

- An uninitialized value outputs [UNINITIALIZED].
- A variable initialized to NULL outputs [NULL].
- An object outputs [OBJECT].

reload

Keywords

Does an unload followed by a load.

Syntax

```
reload (moduleName);
```

moduleName The name of the module to unload and load.

Return value

A module handle if successful or **NULL** if not

Description

reload searches the module list for a matching module. If found, **reload** removes it and then loads it again. If it does not find a module to unload, it simply loads the module for the first time.

Note: If when reloaded, the module references global objects, these references continue to refer to the older objects. (The module is not destroyed, but is stored to maintain these references.) Global module values that are not part of an object are destroyed and then reloaded.

return

Keywords Example

Exits from the current function, on handler, or module, optionally returning a value.

Syntax

```
return [ expression ];
```

Description

A module, by default, returns **TRUE** if successfully run. However, an explicit **return** statement can be provided to return a customized return value, or simply to terminate execution prior to the end of the script.

Example

```
//Example of return
sqr(x)
{
  return (x*x);
}
```

run

Keywords Example

Loads and runs the module indicated.

Syntax

```
run (moduleName)
```

moduleName The name of the module to load and run.

Return value

By default, **run** returns **TRUE** if successful or **FALSE** if not. If the module has a global return statement, **run** returns that value if the module successfully runs and then displays a warning that the standard return value for **run** has been overridden.

Description

run runs the module if it is already loaded. The module remains loaded until explicitly unloaded using unload.

Example

```
//Loads and runs a script file
declare myModule;
myModule = load("demo.spp");  // loads module and gets a handle
if (myModule) {               // if loaded
  run(myModule);              // run the module
unload(myModule);             // unloads the module
}
```

select

Keywords Example

Creates a special global variable, selection,that refers to the selected variable.

Syntax

```
select objectName;
```

objectName The name of the object to select.

Description

You can call **select** on any variable that is loaded in any script. Doing so sets **selection** to reference that variable for all scripts in the session. You then have access to that variable from any script by using the alias **selection** as the name of the variable. Variables so selected can also be referenced using the shorthand dot (".") notation.

Because the variable is global to all loaded scripts, only one **selection** can be active in an IDE session at a time. If you call **select** and there is already a **selection**, you override the current **selection** with your new one.

Example

```
// Example of select and selection
// SELECT1.SPP
class C0 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
class C1 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
declare C0 obj1("One", "Two", "Three");
declare C1 obj2(1, 2, 3);

// Select the first object
select obj1;

// Iterate across the selected object
// using selection, then dot notation.
iterate(iterator; selection; key)
   print typeid(selection), "property", key, "=", iterator;

iterate(iterator; . ; key)
   print typeid(.), "property", key, "=", iterator;

// Note that the dot within the with
// block refers to its own local selection.
with(obj2)
   iterate(iterator; . ; key)
      print typeid(.), "property", key, "=", iterator;

// But the global selection has not changed.
print .v1;
print selection.v2;
print ". and selection still refer to", typeid(.);
```

selection

Keywords Example

Defines a special global reference variable created by calling select on a variable.

Syntax

```
selection.Member
```

Member The class member for which the global variable is created.

Description

Once the **selection** has been made, you can use **selection** in any way that you normally use the variable it refers to. You can access members of the referenced object with **selection**.*member*. The dot (".") shorthand syntax can also be used instead of selection outside a with or iterate block or an on handler. In those situations, the dot has local context and refers to the controlling variable for that block (usually an object).

Because the selection variable is global to all loaded scripts, only one **selection** can be active in an IDE session at a time.

Example

```
// Example of select and selection
// SELECT1.SPP
class C0 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
class C1 (p1, p2, p3) {
   declare v1 = p1;
   declare v2 = p2;
   declare v3 = p3;
}
declare C0 obj1("One", "Two", "Three");
declare C1 obj2(1, 2, 3);

// Select the first object
select obj1;

// Iterate across the selected object
// using selection, then dot notation.
iterate(iterator; selection; key)
   print typeid(selection), "property", key, "=", iterator;

iterate(iterator; . ; key)
   print typeid(.), "property", key, "=", iterator;

// Note that the dot within the with
// block refers to its own local selection.
with(obj2)
   iterate(iterator; . ; key)
      print typeid(.), "property", key, "=", iterator;

// But the global selection has not changed.
print .v1;
print selection.v2;
print ". and selection still refer to", typeid(.);
```

super

See also Keywords

Provides access to a member of the base class with the same name as a member of a derived class.

Syntax

```
objectName.super[.super...].member
```

objectName The name of the object to access.

Description

Base class members can be directly accessed without using **super** where the member name is unique within the class definition.

cScript does not support function overloading or the :: operator. However, you can use **super** to get access to overridden class members as follows:

```
class C1 {
  declare x = "C1";
  Method1() {
   print x;
  }
 }
class C2:C1 {
  Method1() {
   print "C2 derived from ", x;
  }
}
MyObj = new C2;
MyObj.Method1();       //Prints C2 derived from C1
MyObj.super.Method1(); //Prints C1
```

If a base class is itself a derived class and you want to access one of its overridden members, use **super.super** (and so on for further access up the inheritance hierarchy). For example:

```
class C3:C2 {
  Method1() {
   print "C3 derived from C2";
  }
}
MyObj3 = new C3;
MyObj3.Method1();              //Prints C3 derived from C2
MyObj3.super.Method1();        //Prints C2 derived from C1
dMyObj3.super.super.Method1(); //Prints C1
class C3:C2 {
  Method1() {
    print "C3 derived from C2";
  }
}
MyObj3 = new C3;
MyObj3.Method1();             //Prints C3 derived from C2
MyObj3.super.Method1();       //Prints C2 derived from C1
MyObj3.super.super.Method1(); //Prints C1
```

switch

See also Keywords

Chooses one of several alternatives.

Syntax

```
switch (switch_expression){
  case expression :
    [statement1;]
    [statement2;]
    ...
    [break;]
  [default :
    [statement1;]
    [statement2;]
    ...]
}
```

switch_expression	Any valid cScript expression, including a function call. Unlike C++, the *switch_expression* is evaluated for each **case** in a top-down fashion until a match is found or no more **case** statements remain.
expression	Any valid cScript expression, including a function call.
statement	The statement to execute.

Description

The value of the *switch_expression* is checked against the value of each case *expression* until a match is found or until either default or the end of the **switch** statement is reached.

As in C++, all statements but **case** or **default** following the matching **case** are executed until break or the end of the **switch** statement is reached. If no **case** *expression* matches *switch_expression,* the statements following **default**, if any, are executed.

If you insert a **default** case, it must be the last case.

Note: If you don't use **break** as the last statement in the case that executes, all remaining statements (except **case** or **default)** in the **switch** execute until either a **break** is encountered or the end of the **switch** is reached. If you do use a **break** that executes, the **switch** statement ends there.

this

See also Keywords Example

Provides an object reference within a class definition.

Syntax

```
this:>method1() {}
```

Description

The cScript **this** keyword is analogous to the C++ **this** pointer. It is used to provide an object reference within a class definition. **this** is primarily needed to define closures used in event handlers that will apply to all instances of that class.

For example, given the class definition:

```
class MyClass {
  method1() {}
  on this:>method1() {}
}
```

all objects of that class will have a default method call event on handler defined (rather than on a per-instance basis as when the on handler is defined outside of the class).

When **this** is used outside of a class definition, it refers to the current module object since a script module can actually be treated as an object. You can use it to create an event handler for a global function.

For example:

```
DoNothing (){}                        //Globally scoped function
on this:>DoNothing() {                // method of current object
 print "Did something else first";
 pass();
}
```

Note: Calls to module scope functions for which an event handler has been defined will only trigger the handler when they are called in the same way as defined in the **on** handler. For example:

```
this.DoNothing();  // Triggers the event handler
DoNothing();       // Does not trigger an event
```

Example

The following example shows how to use **this** in a class definition in conjunction with **on** handlers or **attach** to bind a method across all instances of that class.

Event handlers normally provide a binding to a specific object or instance, and not to all instances of a class.You can bind an event handler to a class when you want to do either of the following:

- Ensure that some default processing occurs as the very first action regardless of how many other event handlers are subsequently chained to a method of a specific class
- Use more complex pre-processing and post-processing of method calls

```
class C0 {
   declare property1 = 0;
   GetProperty1() {
      return property1;
   }
   on this:>GetProperty1() {  // Increments property1 before call
      property1++;
      return pass();
   }
}

declare C0 myObj;
print myObj.property1;        // Prints 0
print myObj.GetProperty1();   // Prints 1
```

typeid

Keywords

Gets run-time identification of variables or the resulting value of expressions.

Syntax

```
typeid(name_expn);
```

name_expn Any legal variable name or expression

Return value

A string representing the type. Possible return values are

[ARRAY]

classname

[CLOSURE]

[INTEGER]

[NULL]

[REAL]

[STRING]

[UNINITIALIZED]

Description

If the variable or expression value is a built-in type, **typeid** returns the type in brackets [ ]. If it is an object, **typeid** returns the class name. If the expression is a function or method, **typeid**() indicates the type of the return value of the function.

unload

Keywords Example

Unloads the specified module.

Syntax

```
unload (moduleName);
```

moduleName The name of the module to unload.

Return value

TRUE if successful, otherwise **FALSE**

Description

unload searches the module list for a matching module. If found, **unload** removes it, causing all functions, classes, and local variables that were defined in the module to become invalid. However, if an object within the script is referenced from another active script (for example, where a function in the unloaded script returned a reference to an object), that object will not be destroyed.

Example

```
//Loads and runs a script file
declare myModule;
myModule = load("demo.spp");  // loads module and gets a handle
if (myModule) {               // if loaded
  run(myModule);              // run the module
unload(myModule);             // unloads the module
}
```

while

See also Keywords Example

Repeats one or more statements until *condition* is **FALSE**.

Syntax

```
while [( condition )] [{statement_list}]
```

condition Either **TRUE** or **FALSE.** When **FALSE,** *statement_list* stops executing.

statement_list The list of statements to execute.

Description

If no condition is specified, the **while** clause is equivalent to **while**(**TRUE**). Because the test takes place before any statements execute, if *condition* evaluates to **FALSE** on the first pass, the loop does not execute.

break will cause loop execution to be terminated, while continue will cause the *condition* to be evaluated immediately without executing any intervening statements.

Example

```
// Example of while loop
i = 0
while (p[i] < 50) {
  p[i] += 10;
  i += 1;
}
```

with

Keywords

Creates a shorthand reference to a variable.

Syntax

```
with (variable){member_list}
```

variable	The variable being referenced.
member_list	Declarations of properties, methods, and events.

Description

with is particularly useful when the variable is a deeply nested object.

For example, assume an object *z,* which is contained within an object *y,* which is contained within an object *x.* Access to *z*'s members can be cumbersome. For example:

```
x.y.z.DoSomething();
x.y.z.DoSomethingElse();
x.y.z.NowDoThis();
```

You can decrease syntactical complexity by assigning *x.y.z* to another variable. For example:

```
p = x.y.z;              // Assignment lookup
p.DoSomething();        // 1 lookup
p.DoSomethingElse();    // 1 lookup
p.NowDoThis();          // 1 lookup
```

If you use **with**, referencing can be made even simpler:

```
with (x.y.z){           // 1 lookup
  .DoSomething();       // No lookup
  .DoSomethingElse();   // No lookup
  .NowDoThis();         // No lookup
}
```

Scoping of **with** statements in functions is handled as you would expect: the scope is local to the current function and the correct member gets called. For example:

```
WFunc1(){
  with (x.y.z){
   .DoSomething();
  }
}

WFunc2(){
  with (MyClass){
   Wfunc1();            // WFunc1 calls x.y.z.DoSomething()
   .Func2();            // This call is to MyClass.Func2()
  }
}
```

Note: Using the dot operator in a **with** block refers to the current **with** assignment. If the global reference variable selection has been set using select, its reference will not be affected, but is superseded with the **with** block.

yield

Keywords

Forces cScript to check if the abort (Esc) key has been pressed.

Syntax

```
yield;
```

Return value

None

Description

Imbedding **yield** in a time consuming process, such as a loop that executes many times, provides a way to break out of the process, if desired.

About cScript operators

See also

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. cScript uses many of the C++ operators. For the most part, these operators have the same precedence, associativity, and functionality as in C++.

Because cScript has no structs, unions, or references to memory locations, the following C++ operators do not exist in cScript:

`->  *  ->*  .*`

For the same reason, the & operator can be used only to declare function parameters as pass-by-reference parameters (not to dereference variables).

Additionally, cScript does not provide the following C++ operators:

`::  sizeof  const_cast  reinterpret_cast`

cScript does provide two new operators:

`:>` The closure operator, typically used in on statements to override functions

`??` The in operator, used to test members of arrays and classes

Depending on context, the same operator can have more than one meaning. For example, the minus (-) can be interpreted as:

- subtraction (x - y)
- a unary negative (-y)

Note: No spaces are allowed in compound operators (such as `:>`). Spaces change the meaning of the operator and will generate an error.

For information on cScript operators, see:

Operator precedence

Binary

Arithmetic

Assignment

Bitwise

Comma

Conditional

Logical

Reference

Relational

Enclosing

Object-oriented

Unary

Punctuators

lvalues and rvalues

Operator precedence

Operators

Operators on the same line in the table below have equal precedence.

Operators											Associativity
()	[]										left to right
.											left to right
:>	??										left to right
!	~	+	-	++	--	&					right to left
*	/	%									left to right
+	-										left to right
<<	>>										left to right
<	<=	>	>=								left to right
==	!=										left to right
&											left to right
^											left to right
|											left to right
&&											left to right
||											left to right
?:											left to right
=	*=	/=	%=	+=	-=	&=	^=	|=	<<=	>>=	right to left
,											left to right

Binary operators

See also Operators

The binary cScript operators are as follows:

Arithmetic	+	Binary plus (add)
	-	Binary minus (subtract)
	*	Multiply
	/	Divide
	%	Remainder (modulus)
Bitwise	<<	Shift left
	>>	Shift right
	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
	|	Bitwise inclusive OR
Logical	&&	Logical AND
	||	Logical OR
Assignment	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)

	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	|=	Assign bitwise OR
Relational	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
	==	Equal to
	!=	Not equal to
Conditional	? :	Actually a ternary operator
	a ? x : y	"if a then x else y"
Comma	,	Evaluate

Arithmetic operators

Operators

The arithmetic operators are:

```
+   -   *   /   %   ++   --
```

Syntax

```
+ expression
- expression
expression1 + expression2
expression1 - expression2
expression1 * expression2
expression1 / expression2
expression1 % expression2
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

Description

Use the arithmetic operators to perform mathematical computations. *expression1* determines the type of the result when variables of different types are used.

Operator	Description
+ (unary expression)	Assigns a positive value to *expression*
1 (unary expression)	Assigns a negative value to *expression*
+ (addition)	Adds all data typs
- (subtraction)	Subtracts data types
* (multiplication)	Multiplies data types
/ (division)	Divides data types
% (modulus operator)	Returns the remainder of integer division
++ (increment)	Adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.
-- (decrement)	Subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

Assignment operators

See also Operators

The assignment operators are:

```
=     *=      /=      %=     +=     -=
<<=   >>=     &=      ^=     |=
```

Syntax

```
unary-expr  assignment-op  assignment-expr
```

Description

The **=** operator is the only simple assignment operator, the others are compound assignment operators.

In the expression `E1 = E2`, *E1* must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op = E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue *E1* is evaluated only once. The expression's value is *E1* after the expression evaluates.

For example, the following two expressions are equivalent:

```
x += y;
x = x + y;
```

Any assignment can change the cScript native type of the value on the left of the assignment, depending on the type of the value assigned.

See cScript and types for more information.

Note: Do not separate compound operators with spaces. For example, do not enter:

```
+<space>=
```

This generates errors.

Bitwise operators

Operators

Use bitwise operators to modify individual bits of a number rather than the whole number.

Syntax

```
AND-expression & equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
~expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Operator	What it does
&	Bitwise AND: compares two bits and generates a 1 result if both bits are 1; otherwise, it returns 0.
|	Bitwise inclusive OR: compares two bits and generates a 1 result if either or both bits are 1; otherwise, it returns 0.
^	Bitwise exclusive OR: compares two bits and generates a 1 result if the bits are complementary; otherwise, it returns 0.
~	Bitwise complement: inverts each bit. (~ is also used to create destructors.)
>>	Bitwise shift right: moves the bits to the right, discards the far right bit and assigns the leftmost bit to 0.
<<	Bitwise shift left: moves the bits to the left, it discards the far left bit and assigns the rightmost bit to 0.

Both operands in a bitwise expression must be of an integral type.

Bit value		Results of &, ^, | operations		
E1	**E2**	**E1 & E2**	**E1 ^ E2**	**E1 | E2**
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Comma (,) punctuator and operator

See also Operators

A comma acts as a punctuator and operator. It is used as follows:

- Separates elements in a function argument list
- Acts as an operator in comma expressions

Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

Syntax

```
expression , assignment-expression
```

Description

If the left operand *E1* is evaluated as a void expression, then the right operand *E2* is evaluated to give the result and type of the comma expression. By recursion, the expression:

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each *Ex,* with the value and type of *En* giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. The following example calls `func` with three arguments: `i, 5,` and `k.`

```
func(i, (j = 1, j + 4), k);
```

Conditional (?:) operator

See also Operators Example

The conditional operator (?:) is a ternary operator used as a shorthand for **if-else** statements.

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Description

This operator allows you to use a shorthand for:

```
if (expression)
   statement1;
else
   statement2;
```

In the expression:

```
E1 ? E2 : E3
```

E1 evaluates first. If its value is nonzero (**TRUE**), then *E2* evaluates and *E3* is ignored. If *E1* evaluates to zero (**FALSE**), then *E3* evaluates and *E2* is ignored. The result of the statement is the value of either *E2* or *E3*, depending upon which evaluates.

Example

```
//if-else statement:
if (x < y)
   z = x;
else
   z = y;

//Equivalent:
z = (x < y) ? x : y;
```

Logical operators

Operators

Use logical operators to evaluate an expression to **TRUE** or **FALSE.**

Syntax

```
logical-AND-expr  && inclusive-OR-expression
logical-OR-expr   || logical-AND-expression
! expression
```

Operator	Description
&&	Logical AND returns **TRUE** (1) only if both expressions evaluate to a nonzero value; otherwise it returns **FALSE** (0). Unlike C++, if the first expression evaluates to **FALSE**, the second expression is still evaluated.
||	Logical OR returns **TRUE** (1) if either of the expressions evaluates to a nonzero value; otherwise it returns **FALSE** (0). Unlike C++, if the first expression evaluates to **TRUE**, the second expression is still evaluated.
!	Logical negation returns **TRUE** (1) if the entire expression evaluates to a nonzero value; otherwise it returns **FALSE** (0). The expression *!E* is equivalent to `(0 == E)`.

Reference operator

Operators Example

Passes arguments in a function definition header by reference.

Syntax

```
methodName(&parameter[,…]){statementList}
```

Description

In cScript as in C++, the default function calling convention is to pass by value. The reference operator can be applied to parameters in a function definition header to pass the argument by reference instead.

cScript reference types created with the & operator, create aliases for objects and let you pass arguments to functions by reference.

When a variable *x* is passed by reference to a function, the matching formal argument in the function receives an alias for *x,* (similar to an address pointer in C++). Any changes to this alias in the function body are reflected in the value of *x.*

When a variable *x* is passed by value to a function, the matching formal argument in the function receives a copy of *x.* Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can return a value that could be used later to change *x,* but the function cannot directly alter a parameter passed by value.

Note: The reference operator is only valid when used in function definitions as applied to one or more of its parameters. The address of operator is not supported in cScript as it is in C++, where it can be used to obtain the address of (create a pointer to) a variable.

Example

```
// Example of reference operator
func1 (i){i=5;}
func2 (&Ir){i=5;}
// It is a reference variable
    ...
sum = 3;
func1(sum);     // sum passed by value
print sum;      // Prints 3
func2(sum);     // sum passed by reference
print sum;      // Prints 5
```

sum, passed by reference to *func2,* has its value changed when the function exits. *func1,* on the other hand, gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by *func1*.

Relational operators

Operators

Relational operators test equality or inequality of expressions.

Syntax

```
equality-expression   == relational-expression
equality-expression   != relational-expression
relational-expression <  shift-expression
relational-expression >  shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
```

Description

If the statement evaluates to **TRUE** it returns a nonzero value; otherwise, it returns **FALSE** (0).

Operator	Description
==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Enclosing operators

See also Operators

The enclosing operators are:

`[ ]` (brackets), `[[ ]]` (double-brackets), `( )` (parentheses), and `{ }` (braces)

Syntax

```
(expression-list)
function (arg-expression-list)
array-name [expression]
{statement-list}
compound-statement {statement-list}
OLEObject.indexedProperty[[expression]]
```

Operator	Description
[ ]	Array subscript operator. Indicates single and multidimensional array subscripts.
[[ ]]	OLE index operator. Indicates the index of an indexed OLE property.
( )	Parentheses operator. Groups expressions, isolates conditional expressions, or indicates function calls and function parameters.
{ }	Braces. Starts and ends compound statements and indicates a code block.

Array subscript ([ ]) operator

See also Operators

The array subscript operator ([ ]) indicates single and multidimensional array subscripts.

Syntax

```
[expression-list]
```

Description

Use the array subscript operator to declare an array or to access individual array components.

For example:

```
declare myArray = new array [10];
myArray[0] = 5;
myArray[1] = "Cheers";
declare array multiArray[] = {myArray};
print multiArray[0][1]; // prints "Cheers"
```

OLE index ([[ ]]) operator

See also Operators

The OLE index operator ([[ ]]) indicates an OLE object's indexed property index.

Syntax

```
[[expression]]
```

Description

Use double-brackets to access individual indexed entries of an OLE object's indexed property:

```
// create an OLEObject of class OLEGeneric
// which contains an indexed property called foo
declare myObj = new OleObject("OLEGeneric");
print myObj.foo[[3]]; // print the third element of foo
```

Note: This operator is only to be used for accessing elements of an OLE indexed property.

Parentheses ( ) operator

See also Operators

Use the parentheses operator () to:

- Group expressions
- Isolate conditional expressions
- Indicate function calls and function parameters

Syntax 1

```
(expression-list)
```

Description

Syntax 1 groups expressions or isolates conditional expressions.

Syntax 2

```
postfix-expression (arg-expression-list)
```

arg-expression-list A comma-delimited list of expressions of any type representing the actual (real) function arguments.

Description

Syntax 2 describes a call to the function given by the postfix expression. The value of the function call expression, if it has a value, is determined by the **return** statement in the function definition.

Object-oriented operators

See also Operators

The cScript object-oriented operators are:

Operator	Description
:>	Closure operator. Binds a class instance and a method as a single closure reference.
??	In operator. Tests for the existence of a class object property or array index.
.	Member selector. Access a class object member.

In addition, there is a colon (:) punctuator:

:	Refers to a base class in a derived class declaration.

Closure (:>) operator

See also Operators Example

Binds a class instance with a class member.

Syntax 1 (on handler)

```
on ClassInstance:>Method{[code_to_replace_method_code]}
```

Syntax 2 (attach)

```
attach ClassInst1:>method1 to ClassInst2:>method2;
```

Syntax 3 (detach)

```
detach ClassInst1:>method1 from ClassInst2:>method2;
```

Syntax 4 (getter)

```
on ClassInstance:>property{
  // your code here
  return [pass()|SomeValue];
}
```

Syntax 5 (setter)

```
on ClassInstance:>property(parm){
 // your code here
 [pass(SomeValue);]
}
```

Syntax 6 (closure variable)

```
declare closureVar = classInstance:>methodName;
```

Note: A closure variable as declared above can subsequently be used wherever a closure is needed. For example, an alternative to the **attach** statement (Syntax 2) using closure variables would be:

```
declare closureVar1 = classInst1:>method1;
declare closureVar2 = classInst2:>method2;
attach closureVar1 to closureVar2;
```

Description

Use the closure operator (:>) in an on handler, an attach statement, or a detach statement to bind a class instance with a class member as a single closure reference.

Example

```
// Example of closure
import scriptEngine;
import IDE;
...
modList = new ListWindow(50, 5, 100, 300, "Module List",
          TRUE, FALSE, loadedModules);

// Hook the Accept event in order to do nothing.
// Default behavior is to put the list away.
on modList:>Accept(){}
```

Member (.) selector operator

See also Operators Example

Use the member selector operator (.) to access class members.

Syntax

```
class-instance.class-member
```

Description

Suppose that the object *a* is of class A and *b* is a property declared in A. The expression:

```
a.b
```

represents the property *b* in *a*.

Note: Although the precedence of the . operator is the same as C++ in most respects, one place where it is not is in cScript native function calls that do not use parentheses. For example, `print module "MyModule".Data1` does not print the *Data1* member of *MyModule.* To print this reference, you must use parentheses with the **module** function, as follows:

```
print module ("MyModule").Data1
```

Example

```
// Example of member selector (.) operator
class myClass {
   i = 0;
}
s = new myClass();
s.i = 3;              // assign 3 to the i property of myClass s
```

In (??) operator

Operators

Use the in operator (??) to test for the existence of an object property or for an array index.

Syntax 1

```
string-expression | "string" ?? objectname |arrayname
```

Syntax 2

```
integer-expression | integer ?? arrayname
```

Description

Use a quoted string, or an expression that evaluates to a string, to test for the existence of an object property or an associative array index.

Use an integer, or an integer expression, to test for the existence of an index value in an indexed array. For example:

```
class MyClass {
  declare property1 = 0;
  declare property2 = 1;
}

declare MyClass instance;
if ("property1" ?? instance)
  print "property1 is a property of instance.";

declare array a1[];
a1[0] = "a";
a1["Hello"] = 1;
if (0 ?? a1)
  print "Array a1 has an index 0.";
if ("Hello" ?? a1)
  print "Array a1 has an index \"Hello\".";
```

Unary operators

Operators

Syntax

```
unary-operator unary-expression
```

Description

cScript provides the following unary operators:

++ Increment

-- Decrement

+ Plus

- Minus

! Logical negation

~ Bitwise complement

Increment and decrement operators

Operators

The increment and decrement operators are ++ and --. They can be used either to change the value of the operand expression before it is evaluated (pre) or change the value of the whole expression after it is evaluated (post). The increment or decrement value is appropriate to the type of the operand.

Syntax 1 (pre)

```
postfix-expression ++       (postincrement)
postfix-expression --       (postdecrement)
```

Description

The value of the whole expression is the value of the postfix expression before the increment or decrement is applied. After the postfix expression is evaluated, the operand is incremented or decremented by 1.

Syntax 2 (post)

```
++ unary-expression         (preincrement)
-- unary-expression         (predecrement)
```

unary-expression The operand, which must be a modifiable lvalue.

Description

The operand is incremented or decremented by 1 before the expression is evaluated. The value of the whole expression is the incremented or decremented value of the operand.

Plus and minus operators

See also Operators

The plus (+) and minus (-) operators can operate in either a unary or binary fashion on any type of variable.

Syntax 1 (Unary)

```
+ unary-expression
- unary-expression
```

+ unary-expression	Value of the operand after any required integral promotions.
- unary-expression	Negative of the value of the operand after any required integral promotions.

Syntax 2 (Binary)

```
expression1 + expression2
expression1 - expression2
```

expression1	Determines the type of the result.
expression2	Is converted if necessary to a type matching *expression1*, and then the operation is carried out.

Multiplicative operators

See also Operators

There are three multiplicative operators:

*	(multiplication)
/	(division)
%	(modulus or remainder)

Syntax

```
multiplicative-expr * unary-expression
multiplicative-expr / unary-expression
multiplicative-expr % unary-expression
```

Description

The usual type conversions are made on the operands.

`(op1 * op2)`	Product of the two operands
`(op1 / op2)`	Quotient of the two operands (op1 divided by op2)
`(op1 % op2)`	Remainder of the two operands (op1 divided by op2)

For / and %, *op2* must be a nonzero value. If *op2* is zero, the operation results in an error. Note that division of a number by a string can result in this divide by zero error.

When *op1* is an integer, the quotient must be an integer. If the actual quotient would not be an integer, the following rules are used to determine its value:

1. If *op1* and *op2* have the same sign, *op1* / *op2* is the largest integer less than the true quotient, and *op1* % *op2* has the sign of *op1.*
2. If *op1* and *op2* have opposite signs, *op1* / *op2* is the smallest integer greater than the true quotient, and *op1* % *op2* has the sign of *op1.*

Note: Rounding is always toward zero.

Punctuators

Operators

The cScript punctuators (also known as separators) are:

( ) Parentheses

{ } Braces

, Comma

; Semicolon

: Colon

= Equal sign

Pound sign

Most of these punctuators also function as operators.

Braces ({ }) punctuator

Operators

Braces ({ }) indicate the start and end of a compound statement.

Semicolon (;) punctuator

Operators

The semicolon (;) is a statement terminator.

Any legal cScript expression (including the empty expression) followed by ; is interpreted as a statement. The expression is evaluated and its value is discarded. If the statement has no side effects, cScript can ignore it. Semicolons are often used to create an empty statement.

Colon (:) punctuator

See also Operators

Use the colon when declaring a child class or a class with a label.

Syntax 1

```
class childClass:parentClass
```

Use this version to indicate the parent class when declaring a child class. For an example of this syntax, see class.

Syntax 2

```
case expression:
```

Use this version to indicate the end of a case expression. For example:

```
switch (a) {
   case 1:
      print "One";
      break;
   case 2:
      print "Two";
      break;
   default: print "None of the above!";
}
```

Equal sign (=) punctuator

Operators

The equal sign (=) separates variable declarations from initialization lists and determines the type of the variable.

Syntax

```
array x[] = { 1, 2, 3, 4, 5 } ;
x = 5;
```

Description

In cScript, declarations of any type can appear (with some restrictions) at any point within the code. In a cScript function argument list, the equal sign indicates the default value for a parameter:

```
MyFunc(i = 0){...} //Parameter i has default value of zero
```

The equal sign is also used as the assignment operator.

Pound sign (#) operator

See also Operators

The pound sign (#) indicates a preprocessor directive when it occurs as the first non-whitespace character on a line. It signifies a compiler action not necessarily associated with code generation.

lvalues and rvalues

Operators

lvalues

An lvalue is an identifier or expression that can be accessed as an object and legally changed in memory. A constant, for example, is not an lvalue. A variable, array member, or property is an lvalue.

Historically, the l stood for left, meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Only modifiable lvalues can legally stand on the left of an assignment statement.

For example, if *a* and *b* are variables, they are both modifiable values and assignments. The following are legal:

```
a = 1
b = a + b
```

rvalues

An rvalue (short for "right value") is an expression that can be assigned to an lvalue. It is the "right side" of an assignment expression. While an lvalue can also be an rvalue, the opposite is not the case. For example, the following expression cannot be an lvalue:

```
a + b
```

`a + b = a` is illegal because the expression on the left is not related to an object that can be accessed and legally changed in memory.

However, `a = a + b` is legal, because *a* is a variable (an lvalue) and *a* + *b* is an expression that can be evaluated and assigned to a variable (an rvalue).

Preprocessor directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program.

The cScript preprocessor, unlike a C++ preprocessor, supports preprocessor directives in the expansion side of a macro definition. It detects the following preprocessor directives and parses the tokens embedded in them:

#define	#ifndef
#else	#include
#endif	#undef
#ifdef	#warn

Any line with a leading # is considered as a preprocessor directive unless the # is part of a string literal, is in a character constant, or is embedded in a comment. The initial # can be preceded or followed by one or more spaces (excluding new lines).

#define

See also Directives Example

Defines a macro.

Syntax

```
#define macro_identifier <token_sequence>
```

macro_identifier	The identifier for the macro. Each occurrence of *macro_identifier* in your source code following the **#define** is replaced with *token_sequence* (with some exceptions). Such replacements are known as macro expansions.
token_sequence	The sequence to replace *macro_identifier* with. The token sequence is sometimes called the body of the macro. If *token_sequence* is empty, the macro identifier is removed wherever it occurs in the source code.

Description

The **#define** directive defines a macro. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters. Unlike C++ preprocessors, cScript allows you to continue a line with a backslash (\). You cannot use cScript keywords as macros.

After each individual macro expansion, the preprocessor scans the newly expanded text to see if there are further macro identifiers that are subject to replacement (nested macros).

cScript imposes these restrictions on macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro is not expanded during its own expansion (so **`#define A A`** won't expand indefinitely).

Examples

```
// Examples of #define
#define HI "Have a nice day!"
#define empty
#define NIL ""
#define GETSTD #include <stdio.h>
```

#ifdef, #ifndef, #else, and #endif

Directives

Tests whether an identifier is currently defined or not.

Syntax

```
#ifdef/#ifndef identifier [logical-operator identifier [...]]
<section-1>
[#else
<final-section>]
#endif
<next-section>
```

Description

Assume that **#ifdef** tests **TRUE** for the defined condition; so the line:

```
#ifdef identifier
```

means that if *identifier* is defined, include the code up to the next **#else** or **#endif**. If *identifier* is not defined, ignore that code and skip to the next **#else** or **#endif**.

The line:

```
#else
```

means that if *identifier* is not defined, include the code up to the next **#endif**.

The line:

```
#ifndef
```

tests **TRUE** for the not-defined condition; so:

```
#ifndef identifier
```

means that if *identifier* is not defined, include the code up to the next **#else** or **#endif**. If *identifier* is defined, ignore that code.

In this case, **#else** means that if *identifier* is defined, include the code up to the next **#endif**.

In the **TRUE** case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with *next-section*. In the **FALSE** *case*, control passes to the next **#else** line (if any), which is used as an alternative condition for which the previous test proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#ifdef or #ifndef** must be matched with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#ifdef** or **#ifndef** can be matched with its correct **#endif**.

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous #define command has been processed for that identifier and is still in force. You can combine identifiers with logical operators.

An identifier defined as **NULL** is considered to be defined.

cScript supports conditional compilation by replacing the lines that are not to be compiled as a result of the directives with blank lines. All conditional compilation directives must be completed in the source or include file in which they begin.

#include

See also Directives Example

Pulls other cScript files into the source code.

Syntax 1

```
#include <file_name>
```

Syntax 2

```
#include "file_name"
```

Syntax 3

```
#include macro_identifier
```

Description

The **#include** syntax has three formats:

- The first and second formats imply that no macro expansion will be attempted; in other words, *file_name* is never scanned for macro identifiers. *file_name* must be a valid file name with an optional path name and path delimiters.
- The third format does not allow **<** or " to appear as the first non-whitespace character following **#include**. A macro definition that expands the macro identifier into a valid delimited file name with either of the *<file_name>* or *"file_name"* formats must follow the **#include**.

The preprocessor removes the **#include** line and replaces it with the entire text of the cScript source file at that point in the source code. The source code itself is not changed, but the compiler processes the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *file_name*, only that directory will be searched.

Unlike the C++ **#include,** there is no difference between the *<file_name>* and *"file_name"* formats. With both versions, the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the script directories in the order in which they are defined in the Options|Environment|Scripting|Script Path dialog box. If the file is not located in any of the default directories, an error message is issued.

Example

This **#include** statement causes the preprocessor to look for MYINCLUD.H in the current directory, then in default directories.

```
#include "myinclud.h"
```

or

```
#include <myinclud.h>
```

After expansion, this **#include** statement causes the preprocessor to look in C:\BC5\SCRIPT\INCLUDE\MYSTUFF.H. Note that you must use double backslashes in the **#define** statement.

```
#define myinclud "C:\\BC5\\SCRIPT\\INCLUDE\\MYSTUFF.H"
#include  myinclud
/* macro expansion */
```

#undef

Undefines a macro.

Syntax

```
#undef macro_identifier
```

Description

#undef detaches any previous token sequence from the macro identifier; the macro definition is forgotten, and the macro identifier is undefined. No macro expansion occurs within **#undef** lines.

The state of being defined or undefined is an important property of an identifier, regardless of the actual definition. The #ifdef and #ifndef conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier is undefined, it can be redefined with #define, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

Example

```
// Example of #undef
#define BLOCK_SIZE 512
  .
  .
  .
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be an illegal "unknown" identifier */
  .
  .
  .
#define BLOCK_SIZE 128   /* redefinition */
```

#warn

Directives

Sets the warning level.

Syntax

```
#warn warning_level
```

warning_level Ranges from 0 (suppress all warnings) to 3 (show all warnings).

Description

For example, the following statement causes all warnings to be shown when the script is compiled:

```
#warn 3
```

Macros with parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

There can be no whitespace between the macro identifier and the **(**. The optional *arg_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional *actual_arg_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg_list* of the #define line. There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual_arg_list*.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Note: The similarities between function and macro calls can obscure their differences. A macro call can give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Nesting parentheses and commas

The *actual_arg_list* can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

Using the backslash (\) for line continuation

A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

Manifest constants

The following constants are used in cScript and in the scripting classes. The constants in this list jump to the class member topics where they are used. All constants are reserved words.

A

ABORT
ARGUMENT_DIAGNOSTICS
ASSIGN_EXPLICIT
ASSIGN_IMPLICIT_KEYPAD
ASSIGN_IMPLICIT_SHIFT
ASSIGN_IMPLICIT_MODIFIER

B

BACKWARD_RIP
BRIEF_RE
BRIEF_RE_BACK_MAX
BRIEF_RE_BACK_MIN
BRIEF_RE_FORWARD_MAX
BRIEF_RE_FORWARD_MIN
BRIEF_RE_SAME_MAX
BRIEF_RE_SAME_MIN

C

CANCEL
COLUMN_BLOCK

D

DOWN

E

ERROR
EXCLUSIVE_BLOCK

F

FALSE
FATAL
FULL_DIAGNOSTICS

I

IDE_RE
INCLUDE_ALPHA_CHARS
INCLUDE_LOWERCASE_ALPHA_CHARS
INCLUDE_NUMERIC_CHARS
INCLUDE_SPECIAL_CHARS
INCLUDE_UPPERCASE_ALPHA_CHARS
INCLUSIVE_BLOCK
INFORMATION
INVALID_BLOCK
INVERT_LEGAL_CHARS

L

LANGUAGE_DIAGNOSTICS
LEFT
LIBRARY_MODULE
LINE_BLOCK

M

MEMBER_DIAGNOSTICS
METHOD_DIAGNOSTICS
MODULE_DIAGNOSTICS

N
NO_DIAGNOSTICS

O
OBJECT_DIAGNOSTICS
OK

R
RETRY
RIGHT

S
SCRIPT_MODULE
SEARCH_BACKWARD
SEARCH_FORWARD
SKIP_LEFT
SKIP_NONWHITE
SKIP_NONWORD
SKIP_RIGHT
SKIP_STREAM
SKIP_WHITE
SKIP_WORD
SW_MAXIMIZE
SW_MINIMIZE
SW_NORMAL

T
TE_APPLICATION
TE_DLL
TE_DOS16
TE_DOSCOM
TE_DOSOVERLAY
TE_EASYWIN
TE_IMPORTLIB
TE_LINKAGE_DYNAMIC
TE_MM_COMPACT
TE_MM_TINY
TE_MM_SMALL
TE_MM_MEDIUM
TE_MM_LARGE
TE_MM_HUGE
TE_NT_FSCONSOLE
TE_NT_GUI
TE_NT_WINCONSOLE
TE_STATICLIB
TE_STDLIB_BGI
TE_STDLIB_BIDS
TE_STDLIB_BWCC
TE_STDLIB_C0F
TE_STDLIB_CODEGUARD
TE_STDLIB_CTL3D
TE_STDLIB_EMU
TE_STDLIB_MATH
TE_STDLIB_NOEH
TE_STDLIB_OCF
TE_STDLIB_OLE2
TE_STDLIB_OWL
TE_STDLIB_RTL
TE_STDLIB_TVISION

U

V

W

FALSE/false
A constant meaning false or zero.

TRUE/true
A constant meaning true or not zero.

BufferOptions class

See also Description

This class is one of the editor classes. *BufferOptions* objects hold data controlling the characteristics of edit buffers.

Syntax

```
BufferOptions()
```

Properties	Access
`bool CreateBackup`	Read-write
`bool CursorThroughTabs`	Read-write
`bool HorizontalScrollBar`	Read-write
`bool InsertMode`	Read-write
`int LeftGutterWidth`	Read-write
`int Margin`	Read-write
`bool OverwriteBlocks`	Read-write
`bool PersistentBlocks`	Read-write
`bool PreserveLineEnds`	Read-write
`bool SyntaxHighlight`	Read-write
`string TabRack`	Read-write
`string TokenFileName`	Read-write
`bool UseTabCharacter`	Read-write
`bool VerticalScrollBar`	Read-write

Methods

```
void Copy(BufferOptions source)
```

Events

None

BufferOptions class description

BufferOptions class

This class holds buffer options settings, such as scroll bars, right margin setting, tab rack, syntax highlighting, cursor shape, gutter width, block style and tabbing modes.

An instance of this class exists as a member of the global editor options accessible via Ediitor.Options. This class controls the settings of all edit buffers. Any change to this object changes the settings of all edit buffers. The properties are initialized during construction to match the global defaults.

You can instantiate a member of this class to store buffer options. They are not applied to any edit buffers until you copy them into *Editor.Options,* at which point the settings affect all edit buffers.

For example, in a *BufferOptions* object, you can store a set of options that you want to apply to a buffer when it is activated (such as tab stops, syntax highlighting and color). Applying these values to *Editor.Options* sets the buffer options for the new buffer and all other edit buffers as well.

CreateBackup property

See also BufferOptions class

Automatically creates a backup of the source file loaded in the active Edit window when you choose File|Save. The backup file has the extension .BAK.

Access

Read-write

Type expected

```
boolCreateBackup
```

Description

In the IDE, *CreateBackup* is set with the Create Backup option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|File.

CursorThroughTabs property

See also BufferOptions class

Causes the cursor to move uniformly through the line as you press arrow keys for horizontal movement.

Access

Read-write

Type expected

```
bool CursorThroughTabs
```

Description

When *CursorThroughTabs* is **FALSE,** the cursor jumps several columns when moved over a tab.This setting has no effect unless tabs are set with the TabRack property.

In the IDE, *CursorThroughTabs* is set with the Cursor Through Tabs option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

HorizontalScrollBar property

See also BufferOptions class

Set to **TRUE** to display a horizontal scroll bar in the active Edit window. Set to **FALSE** to hide the horizontal scroll bar.

Access

Read-write

Type expected

```
bool HorizontalScrollBar
```

Description

In the IDE, *HorizontalScrollBar* is set with the Horizontal Scroll Bar option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Display.

InsertMode property

See also BufferOptions class

Sets or clears text insert mode.

Access

Read-write

Type expected

```
bool InsertMode
```

Description

Set to **TRUE** to put the buffer in Insert mode. This pushes the existing text to the right as you type.

Set to **FALSE** to put the buffer in Overwrite mode. This writes over the existing text.

In the IDE, *InsertMode* is set with the Insert Mode option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

LeftGutterWidth property

See also BufferOptions class

The width of the Edit window's left gutter.

Access

Read-write

Type expected

```
int LeftGutterWidth
```

Description

The gutter width represents pixels. It is a positive decimal measurement (for example 16). The default setting is 32.

In the IDE, *GutterWidth* is set with the Gutter Width option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Display.

Margin property

See also BufferOptions class

The column number to use for the Edit window's right margin.

Access

Read-write

Type expected

```
int Margin
```

Description

Valid entries are from 1 to 1024.

In the IDE, *Margin* is set with the Right Margin option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Display.

OverwriteBlocks property

See also BufferOptions class

Deletes selected text as you type.

Access

Read-write

Type expected

```
bool OverwriteBlocks
```

Description

Works in conjunction with Persistent Blocks to delete selected text as you type. If you mark a block of text and type a letter, the letter you type replaces the entire marked block.

If you press...	Overwrite blocks will...
DEL or Backspace	Clear the entire block of selected text
Any key or choose Edit|Paste	Replace the entire block of selected text

- When this property is **FALSE** and *PersistentBlocks* is **TRUE,** text entered in a marked block is added at the insertion point.

In the IDE, *OverwriteBlocks* is set with the OverwriteBlocks option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

PersistentBlocks property

See also BufferOptions class

Allows marked blocks to remain selected until they are deleted or unmarked, or until another block is selected.

Access

Read-write

Type expected

```
bool PersistentBlocks
```

Description

When *PersistentBlocks* is **FALSE** and you move the cursor after a block is selected, the text does not stay selected.

In the IDE, *PersistentBlocks* is set with the Persistent Blocks option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

PreserveLineEnds property

See also BufferOptions class

Saves files with their original line ends. When *PerserveLineEnds* is **FALSE,** files are saved with the Borland C++ default value for line ends.

Access

Read-write

Type expected

```
bool PreserveLineEnds
```

Description

Use this option to specify how the line ends are written when a file is saved: you can use the Borland C++ default value, or you can write the original line end of the file.

Line ends usually consist one of the following combination of characters:

LF

CR

LF CR

CR LF (Borland C++ default)

where *LF* = Line Feed (ASCII value 10) and *CR* = Carriage Return (ASCII value 13).

In the IDE, *PerseveLineEnds* is set with the Perserve Line Ends option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|File.

SyntaxHighlight property

BufferOptions class

Indicates if the editor displays code with syntax highlighting.

Access

Read-write

Type expected

```
bool SyntaxHighlight
```

Description

You can specify your own keywords, functions, or other language elements that you want highlighted. These elements are stored in token (.TOK) files. Use TokenFileName to open the .TOK file.

In the IDE, *SyntaxHighlighting* is set with the Use Syntax Highlighting option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Syntax Highlighting.

TabRack property

See also BufferOptions class

The buffer's tab settings.

Access

Read-write

Type expected

```
string TabRack
```

Description

The tab settings are indicated as a space-delimited sequence of tab stops in ascending order. For example, "3 7 12" sets tab stops at 3", 7" and 12".

In the IDE, *TabRack* is set with the Tab Stops option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

TokenFileName property

BufferOptions class

The name of the token file (.TOK) to use for syntax highlighting.

Access

Read-write

Type expected

```
string TokenFileName
```

Description

In the IDE, *TokenFileName* is set with the Syntax Extensions option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Syntax Highlighting.

UseTabCharacter property

See also BufferOptions class

If **TRUE,** inserts a true tab character (ASCII 9) when you press Tab. If **FALSE,** replaces tabs with spaces.

Access

Read-write

Type expected

```
bool UseTabCharacter
```

Description

TabRack determines the number of spaces used to replace a tab.

In the IDE, *UseTabCharacter* is set with the Use Tab Character option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

VerticalScrollBar property

See also BufferOptions class

Set this property to **TRUE** to display a vertical scroll bar in the active Edit window. Set to **FALSE** to hide the vertical scroll bar.

Access

Read-write

Type expected

```
bool VerticalScrollBar
```

Description

In the IDE, *VerticalScrollBar* is set with the Vertical Scroll Bar option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Display.

Copy method

BufferOptions class

Copies the values from the source *BufferOptions* object into this *BufferOptions* object.

Types expected

```
void Copy(BufferOptions source)
```

source The name of the buffer to copy from.

Return value

None

Debugger class

Description

Debugger class members let you debug a cScript program. You can set breakpoints, single step through code, and inspect variables.

Syntax

```
Debugger()
```

Properties	Access
bool HasProcess	Read-only

Methods

```
bool AddBreakpoint()
bool AddBreakpointFileLine(string fileName, int lineNum)
bool AddWatch(string symbolName)
bool Animate()
bool Attach(string processID)
bool BreakpointOptions()
string Evaluate(string symbol)
bool EvaluateWindow(string symbol)
bool FindExecutionPoint()
bool Inspect(string symbol)
bool InstructionStepInto()
bool InstructionStepOver()
bool IsRunnable(int processID)
bool Load(string exeName)
bool PauseProgram()
bool Reset()
bool Run()
bool RunToAddress(string addr)
bool RunToFileLine(string fileName, int lineNum)
bool StatementStepInto()
bool StatementStepOver()
bool TerminateProgram()
bool ToggleBreakpoint(string fileName, int lineNum)
bool ViewBreakpoint()
bool ViewCallStack()
bool ViewCpu([address])
bool ViewCpuFileLine(string fileName, int lineNum)
bool ViewProcess()
bool ViewWatch()
```

Events

```
void DebugeeAboutToRun()
void DebugeeCreated()
void DebugeeStopped()
void DebugeeTerminated()
```

Debugger class description

Debugger class

No matter how careful you are when you code, your script is likely to have errors (bugs) that prevent it from running the way you intended. Debugging is the process of locating and fixing errors that prevent your script from operating correctly.

The *Debugger* class lets you:

- Add breakpoints to your script file
- Add a watch on a symbol name
- Watch your script's execution in slow motion
- Evaluate expressions
- Inspect symbols
- Step over and step into function calls
- Pause, reset and run the current process
- View the call stack
- View the CPU register

HasProcess property

Debugger class

TRUE when the debugger has a process loaded, **FALSE,** otherwise.

Access

Read-only

Type expected

```
bool HasProcess
```

AddBreakpoint method

See also Debugger class

Opens the Add Breakpoint dialog.

Types expected

```
bool AddBreakpoint()
```

Return value

TRUE if successful, **FALSE,** otherwise

AddBreakpointFileLine method

See also Debugger class

Adds a breakpoint on the specified line of the specified file.

Types expected

```
bool AddBreakpointFileLine(string fileName, int lineNum)
```

fileName The name of the file to add the breakpoint to.

lineNum The number of the line on which to add the breakpoint.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If the arguments are **NULL**, *AddBreakpointFileLine* opens the Add Breakpoint dialog.

AddWatch method

See also Debugger class

Adds a watch on the specified *symbolName*.

Types expected

```
bool AddWatch(string symbolName)
```

symbolName The name of the symbol on which to place the watch.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If *symbolName* is **NULL**, *AddWatch* opens the Add Watch dialog.

Animate method

Debugger class

Lets you watch your script execute in "slow motion."

Types expected

```
bool Animate()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

Animate performs a continuous series of StatementStepInto commands.

To interrupt animation, invoke one of the following *Debugger* methods either by menu selections or by keystrokes tied to the script:

Run

RunToAddress

RunToFileLine

PauseProgram

Reset

TerminateProgram

FindExecutionPoint

Attach method

Debugger class

Invokes the debugger for the currently executing process.

Types expected

```
bool Attach(string processID)
```

processID The process to debug.

Return value

TRUE if successful, **FALSE,** otherwise

BreakpointOptions method

See also Debugger class

Opens the Breakpoint Condition/Action Options dialog.

Types expected

```
bool BreakpointOptions()
```

Return value

TRUE if successful, **FALSE,** otherwise

Evaluate method

Debugger class

Evaluates the given expression, such as a global or local variable or an arithmetic expression.

Types expected

```
string Evaluate(string expression)
```

expression The expression to evaulate.

Return value

The result of the evaluation

EvaluateWindow method

See also Debugger class

Opens the Evaluator window.

Types expected

```
bool EvaluateWindow(string expression)
```

expression The expression to evaluate.

Return value

TRUE if successful, **FALSE,** otherwise

Description

When *EvaluateWindow* opens the Evaluator window, *expression* is pasted into the Expression field of the window.

FindExecutionPoint method

See also Debugger class

Displays the current execution point.

Types expected

```
bool FindExecutionPoint()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

The current execution point is indicated by the EIP register. If the current execution point is in source, the execution point is shown in an Edit window. (The appropriate source file is opened if necessary.)

If the current execution point is at an address which has no source associated with it, the execution point is shown in a CPU view. (One is opened if necessary.)

Inspect method

Debugger class

Opens an inspector for the specified *symbol*.

Types expected

```
bool Inspect(string symbol, EditView view, int row, int column)
```

symobl	The symbol to inspect.
view	The view on which to place the Inspector window.
row	The number of the row at which to place the top of the Inspector window.
column	The number of the column at which to place the left side of the Inspector window.

Return value

TRUE if successful, **FALSE,** otherwise

InstructionStepInto method

Debugger class

Executes the next instruction, stepping into any function calls.

Types expected

```
bool InstructionStepInto()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *InstructionStepInto* first loads the executable for the current project.

InstructionStepOver method

Debugger class

Executes the next instruction, running any functions called at full speed.

Types expected

```
bool InstructionStepOver()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *InstructionStepOver* first loads the executable for the current project.

IsRunnable method

Debugger class

Indicates if the specified process can be run or single stepped.

Types expected

```
bool IsRunnable(int processID)
```

processID The process you wish to query. If that process is not runnable or does not exist, the current process is used.

Return value

TRUE if the EXE is runnable or can be single stepped; **FALSE,** otherwise

Load method

See also Debugger class

Loads the specified executable into the debugger.

Types expected

```
bool Load(string exeName)
```

exeName The name of the executable to load. If *exeName* is **NULL**, *Load* opens the Load Program dialog.

Return value

TRUE if successful, **FALSE,** otherwise

Description

Upon loading, the process runs to the starting point specified in the Options|Environment|Debugger| Debugger Behavior dialog.

PauseProgram method

Debugger class

Pauses the current process.

Types expected

```
bool PauseProgram()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

PauseProgram has an effect only if the current process is running or is animated.

Reset method

See also Debugger class

Reset the current process to its starting point.

Types expected

```
bool Reset()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

The starting point is specified in the Options|Environment|Debugger|Debugger Behavior dialog.

Run method

Debugger class

Causes the debugger to run the current process.

Types expected

```
bool Run()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If no process is loaded, *Run* first loads the executable associated with the current project.

RunToAddress method

Debugger class

Runs the current process until the instruction at the given *address* is encountered.

Types expected

```
bool RunToAddress(string address)
```

address The address at which to stop execution. *address* must be given as a hexidecimal value (i.e. it must begin with "0x").

Return value

TRUE if successful, **FALSE,** otherwise

Description

If no process is loaded, *Run* first loads the executable associated with the current project.

RunToFileLine method

Debugger class

Runs the current process until the source at the specified line in the specified file is encountered.

Types expected

```
bool RunToFileLine(string fileName, int lineNum)
```

fileName The name of the file to execute.

lineNum The number of the line at which to halt execution.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If no process is loaded, *RunToFileLine* will first load the executable associated with the current project.

StatementStepInto method

Debugger class

Executes the next source statement and steps through the source of any function calls.

Types expected

```
bool StatementStepInto()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *StatementSetpInto* first loads the executable for the current project.

StatementStepOver method

Debugger class

Executes the next source statement and does not step into any functions called, but rather runs them at full speed.

Types expected

```
bool StatementStepOver()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *StatementStepOver* first loads the executable for the current project.

TerminateProgram method

Debugger class

Terminates the current process.

Types expected

```
bool TerminateProgram()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If no process is loaded, *TerminateProgram* has no effect.

ToggleBreakpoint method

See also Debugger class

If no breakpoint exists, *ToggleBreakpoint* adds a breakpoint on the specified line of the specified file. If a breakoint exists, *ToggleBreakpoint* deletes it.

Types expected

```
bool ToggleBreakpoint(string fileName, int lineNum)
```

fileName The name of the file to add the breakpoint to.

lineNum The number of the line on which to add the breakpoint.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If the arguments are **NULL**, *ToggleBreakpoint* opens the Add Breakpoint dialog.

ViewBreakpoint method

See also Debugger class

Opens the Breakpoints window.

Types expected

```
bool ViewBreakpoint()
```

Return value

TRUE if successful, **FALSE,** otherwise

ViewCallStack method

See also Debugger class

Opens the Call Stack window.

Types expected

```
bool ViewCallStack()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

ViewCallStack works only if a process is loaded.

ViewCpu method

See also Debugger class

Opens or selects the CPU window.

Types expected

```
bool ViewCpu([address])
```

address The address at which to open the CPU window. address is optional. If it is not specified, the view opens for the current address.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If the Allow Multiple CPU Views option is checked in the Debugger Behavior dialog, *ViewCpu* always opens a new CPU window. If the option is not checked, *ViewCpu* only opens a new CPU window if one is not already open.

ViewCpu works only if a process is loaded.

ViewCpuFileLine method

See also Debugger

Opens or selects the CPU window.

Types expected

```
bool ViewCpu(string fileName, int lineNum)
```

fileName The name of the file to view in the CPU window.

lineNum The number of the line to view in the CPU window.

Return value

TRUE if successful, **FALSE,** otherwise

Description

If the Allow Multiple CPU Views option is checked in the Debugger Behavior dialog, *ViewCpuFileLine* always opens a new CPU window. If the option is not checked, *ViewCpuFileLine* opens a new CPU window only if one is not already open.

After opening or selecting a CPU window, the Disassembly pane is scrolled so that the disassembled code for the specified line of the specified file is visible.

If the parameters are **NULL** or if the line doesn't generate code, the window displays an error message. *ViewCpuFileLine* works only if a process is loaded.

ViewProcess method

See also Debugger

Opens the Process window.

Types expected

```
bool ViewProcess()
```

Return value

TRUE if successful, **FALSE,** otherwise

ViewWatch method

See also Debugger class

Opens the Watches window.

Types expected

```
bool ViewWatch()
```

Return value

TRUE if successful, **FALSE,** otherwise

DebugeeAboutToRun event

Debugger

Raised just before a process is run.

Types expected

```
void DebugeeAboutToRun()
```

Return value

None

DebugeeCreated event

Debugger

Raised when a new process is loaded into the debugger.

Types expected

```
void DebugeeCreated()
```

Return value

None

DebugeeStopped event

Debugger

Raised when a process stops.

Types expected

```
void DebugeeStopped()
```

Return value

None

Description

A process can stop for any number of reasons:

- Upon normal termination
- After a step
- When a breakpoint is hit
- When an exception occurs
- When the user pauses, resets, or terminates a running application

DebugeeTerminated event

Debugger

Raised when a process is terminated.

Types expected

```
void DebugeeTerminated()
```

Return value

None

EditBlock class

See also Description

This class is one of the editor classes. *EditBlock* class members provide area-marking features for an edit buffer or view.

Syntax

```
EditBlock(EditBuffer);
EditBlock(EditView);
```

Properties	Access
`bool IsValid`	Read-only
`int EndingColumn`	Read-only
`int EndingRow`	Read-only
`bool Hide`	Read-write
`int Size`	Read-write
`int StartingColumn`	Read-only
`int StartingRow`	Read-only
`int Style`	Read-write
`string Text`	Read-only

Methods

```
void Begin()
void Copy([bool useClipboard, bool append])
void Cut([bool useClipboard, bool append])
bool Delete()
void End ()
bool Extend(int newRow, int newCol)
bool ExtendPageDown()
bool ExtendPageUp()
bool ExtendReal(int newRow, int newColumn)
bool ExtendRelative(int deltaRow, int deltaColumn)
void Indent(int magnitude)
void LowerCase()
bool Print()
void Reset()
void Restore()
void Save()
bool SaveToFile([string fileName])
void ToggleCase()
void UpperCase()
```

Events

None

EditBlock class description

EditBlock class

EditBlock objects let you mark areas of text. Because *EditBlock* members exist in both the EditView and the EditBuffer, *EditView* and *EditBuffer* support different marked areas in different views on the same *EditBuffer*.

Although multiple *EditBlock*s can exist in script for an individual *EditBuffer* or *EditView*, they are mapped to the same internal representation of the *EditBlock*. Therefore, manipulations on one will affect the others.

Use of the following *EditBlock* members will cause the *EditPosition* for the owner to be updated appropriately:

- Extend
- ExtendPageDown
- ExtendPageUp
- ExtendReal
- ExtendRelative

IsValid property

EditBlock class

Is **TRUE** if the block is valid. Becomes **FALSE** in any of the following cases:

- The owning *EditBuffer* or *EditView* is destroyed.
- A destructive operation, such as delete or cut, occurrs on the block.
- The ending point is not greater than the starting point.

Access

Read-only

Type expected

```
bool IsValid
```

EndingColumn property

EditBlock class

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method.

Access

Read-only

Type expected

```
int EndingColumn
```

EndingRow property

EditBlock class

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method.

Access

Read-only

Type expected

```
int EndingRow
```

Hide property

EditBlock class

Visually disables the block without modifying its coordinates.

Access

Read-write

Type expected

```
bool Hide
```

Size property

EditBlock class

If the area is not valid, the value is zero; otherwise, the value is the number of characters contained in the marked area. A newline (CR/LF) counts as one character.

Access

Read-write

Type expected

```
int Size
```

StartingColumn property

EditBlock class

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method.

Access

Read-only

Type expected

```
int StartingColumn
```

StartingRow property

EditBlock class

Initialized to the current position in the *EditView* or *EditBuffer* upon construction. May be changed by a call to an external method.

Access

Read-only

Type expected

```
int StartingRow
```

Style property

EditBlock class

Sets the style of the *EditBlock*.

Access

Read-write

Type expected

```
int Style
```

Description

Style can be set to one of the following values:

INCLUSIVE_BLOCK

EXCLUSIVE_BLOCK

COLUMN_BLOCK

LINE_BLOCK

INVALID_BLOCK

An *EditBlock* is initially set to the *Style* EXCLUSIVE_BLOCK. It is also set to this style after a *Reset* is called.

If an *EditBlock* has a *Style* of INVALID_BLOCK, it was retained after the *EditBuffer* or *EditView* to which it was attached was destroyed.

Text property

EditBlock class

If the marked block is valid, *Text* returns the marked text. If it is invalid, *Text* returns the empty string.

Access

Read-only

Type expected

```
string Text
```

Begin method

EditBlock class

Resets the StartingRow and StartingColumn values to the current location in the owning *EditBuffer* or *EditView.*

Type expected

```
void Begin()
```

Return value

None

Copy method

EditBlock class

Copies the contents of the marked block to the Windows Clipboard.

Types expected

```
void Copy([bool append])
```

append Defaults to **FALSE**. If **TRUE,** the contents of the marked block are appended to the Clipboard.

Return value

None

Cut method

EditBlock class

Cuts the contents of the marked block to the Windows Clipboard and invalidates the marked block.

Types expected

```
void Cut([bool append])
```

append Defaults to **FALSE**. If **TRUE,** the contents of the marked block are appended to the Clipboard.

Return value

None

Delete method

EditBlock class

Deletes the current block if it is valid. The cursor position is restored to the position it occupied prior to the delete.

Types expected

```
bool Delete()
```

Return value

TRUE if characters were deleted; **FALSE,** otherwise

End method

EditBlock class

Resets the EndingRow and EndingColumn values to the current location in the owning *EditBuffer* or *EditView.*

Types expected

```
void End()
```

Return value

None

Extend method

EditBlock class

Extends an existing *EditBlock* to encompass the text delimited by *newRow* and *newCol*.

Types expected

```
bool Extend(int newRow, int newCol)
```

newRow The row to extend the block to. Text delimited by this row is included in the block.

newCol The column to extend the block to. Text delimited by this column is included in the block.

Return value

TRUE if the *Extend* successfully completes; **FALSE,** otherwise

ExtendPageDown method

See also EditBlock class

Updates the starting or ending points of the existing mark to extend the mark to the specified location.

Types expected

```
bool ExtendPageDown()
```

Return value

TRUE if the cursor move is successful; **FALSE,** otherwise

Description

ExtendPageDown causes the position in the owning EditBuffer or EditView to be updated to the new location. *ExtendPageDown* only works if the block is associated with an *EditView.* It is ignored if the block is associated with an *EditBuffer*.

ExtendPageUp method

See also EditBlock class

Updates the starting or ending points of the existing mark to extend the mark to the specified location.

Types expected

```
bool ExtendPageUp()
```

Return value

TRUE if the cursor move is successful

Description

ExtendPageUp causes the position in the owning EditBuffer or EditView to be updated to the new location. *ExtendPageUp* only works if the block is associated with an *EditView.* It is ignored if the block is associated with an *EditBuffer*.

ExtendReal method

See also EditBlock class

Updates the starting or ending points of the existing mark to extend the mark to the specified location.

Types expected

```
bool ExtendReal(int newRow, int newColumn)
```

newRow The row to extend the block to. Text delimited by this row is included in the block.

newCol The column to extend the block to. Text delimited by this column is included in the block.

Return value

TRUE if the cursor move is successful

Description

ExtendReal causes the position in the owning EditBuffer or EditView to be updated to the new location.

ExtendRelative method

See also EditBlock class

Updates the starting or ending points of the existing mark to extend the mark to the specified relative location.

Types expected

```
bool ExtendRelative(int deltaRow, int deltaColumn)
```

deltaRow The row to extend the block from. Text delimited by this row is included in the block.

newCol The column to extend the block from. Text delimited by this column is included in the block.

Return value

TRUE if the cursor move is successful

Description

ExtendRelative causes the position in the owning EditBuffer or EditView to be updated to the new location.

Indent method

EditBlock class

Moves the contents of the block.

Types expected

```
void Indent(int magnitude)
```

magnitude The number of columns to move the block. Negative values move the block to the left, positive values move it to the right.

Return value

None

LowerCase method

EditBlock class

Converts all alphabetic characters enclosed within the *EditBlock* to lowercase.

Types expected

```
void LowerCase()
```

Return value

None

Print method

EditBlock class

Prints the current block.

Types expected

```
bool Print()
```

Return value

TRUE if the print was successful, **FALSE** if there is no marked block or if the print failed.

Reset method

EditBlock class

Unmarks the block. Implicitly invoked by the constructor.

Types expected

```
void Reset()
```

Return value

None

Description

Reset also resets the Style to EXCLUSIVE_BLOCK and the starting and ending points to the current position in the owning *EditBuffer* or *EditView*.

Restore method

EditBlock class

Restores a block from an internal stack. The block must have been saved with Save.

Types expected

```
void Restore()
```

Return value

None

Save method

EditBlock class

Preserves the block attributes on an internal stack for future restoration using Restore.

Types expected

```
void Save()
```

Return value

None

SaveToFile method

EditBlock class

Causes the contents of the marked block to be saved.

Types expected

```
bool SaveToFile([string fileName])
```

fileName The name of the file to save the block to. If *fileName* is not supplied, the user will be prompted for one.

Return value

TRUE if the save was successful or **FALSE** if it wasn't.

ToggleCase method

EditBlock class

Converts all the uppercase alphabetic characters in the *EditBlock* to lowercase, and the lowercase characters to uppercase.

Types expected

```
void ToggleCase()
```

Return value

None

UpperCase method

EditBlock class

Converts all the lowercase alphabetic characters in the *EditBlock* to uppercase.

Types expected

```
void UpperCase()
```

Return value

None

EditBuffer class

See also Description

This class is one of the editor classes. An edit buffer is associated with one file and any number of edit views.

Syntax

```
EditBuffer(string fileName [, bool private, bool readOnly])
```

fileName The name of the file associated with the edit buffer.

private Implies that the buffer is a hidden system buffer. Undo information is not retained, and the *EditBuffer is* never attachable to an *EditView.* The file attached to the buffer cannot be viewed in the IDE until the private buffer is destroyed. When a private *EditBuffer* is no longer needed, you should always explicitly destroy it with *EditBuffer.Destroy*

The default value of *private* is **FALSE**.

readOnly Marks the buffer as read-only. The default value is **FALSE**. Associating a read-only file with the *EditBuffer* does not make the *EditBuffer* read-only.

Properties	Access
`EditBlock Block`	Read-only
`TimeStamp CurrentDate`	Read-only
`string Directory`	Read-only
`string Drive`	Read-only
`string Extension`	Read-only
`string FileName`	Read-only
`string FullName`	Read-only
`TimeStamp InitialDate`	Read-only
`bool IsModified`	Read-only
`bool IsPrivate`	Read-only
`bool IsReadOnly`	Read-only
`bool IsValid`	Read-only
`EditPosition Position`	Read-only
`EditView TopView`	Read-only

Methods

```
void ApplyStyle(EditStyle styleToApply)
EditBlock BlockCreate()
string Describe()
bool Destroy()
EditBuffer NextBuffer(bool privateToo)
EditView NextView(EditView)
EditPosition PositionCreate()
bool Print()
EditBuffer PriorBuffer(bool privateToo)
bool Rename(string newName)
int Save([string newName])
```

Events

```
void AttemptToModifyReadOnlyBuffer()
void AttemptToWriteReadOnlyFile()
```

```
void HasBeenModified()
```

EditBuffer class description

EditBuffer class

An *EditBuffer* is a representation of the contents of a file. An EditView is used to provide a visual representation of the *EditBuffer*. The same *EditBuffer* can be displayed simultaneously in different *EditView*s (for example, two edit windows can be open on the same file). *EditBuffer* objects provide functionality for a file being edited that is independent of the number of views associated with the buffer.

Edit buffers:

- Use the NextView method to to traverse the list of views containing the same *EditBuffer*.
- Maintain access to a list of bookmarks (position markers which track text edits).
- Can be queried for their time and date stamps.
- Have a Position member through which manipulation of the underlying *EditBuffer* is performed. Typically this member will be used when manipulating an *EditBuffer* through script.
- Can be specified as read-only.
- Can be created as private or system buffers. System buffers are not visible in the IDE or listed in the buffer list.

A single *EditBuffer* object exists internally for each file loaded into the buffer. If you create additional representations for an edit buffer, they are attached to the existing *EditBuffer* object. Any changes to one of these representations changes the others, since they refer to the same object. All representations inherit the IsReadOnly and IsPrivate attributes of the original, because these properties are set only when the object is first created.

You can make buffers private to provide raw data storage for script usage. No undo information is maintained for private buffers, nor are they attachable to an *EditView*. Private *EditBuffer* objects should be explicitly destroyed when no longer needed using the Destroy method. Otherwise, they remain in memory for the duration of the IDE session.

Block property

EditBuffer class

Contains a reference to the hidden EditBlock.

Access

Read-only

Type expected

```
EditBlock Block
```

CurrentDate property

EditBuffer class

Originally set to the same value as InitialDate but is updated when the buffer's contents are altered.

Access

Read-only

Type expected

```
TimeStamp CurrentDate
```

Directory property

EditBuffer class

NULL if the *EditBuffer* is invalid; otherwise, indicates the directory path in uppercase letters.

Access

Read-only

Type expected

```
string Directory
```

Drive property

EditBuffer class

NULL if the *EditBuffer* is invalid; otherwise, indicates the drive in uppercase with the associated colon (:).

Access

Read-only

Type expected

```
string Drive
```

Extension property

EditBuffer class

NULL if the *EditBuffer* is invalid; otherwise, indicates the file extension in uppercase including the period (.), if any.

Access

Read-only

Type expected

```
string Extension
```

FileName property

EditBuffer class

NULL if the *EditBuffer* is invalid; otherwise, indicates the file name in uppercase.

Access

Read-only

Type expected

```
string FileName
```

FullName property

EditBuffer class

The name of the *EditBuffer* or **NULL** if the *EditBuffer* is invalid.

Access

Read-only

Type expected

```
string FullName
```

InitialDate property

EditBuffer class

The date on which the file was first created.

Access

Read-only

Type expected

```
TimeStamp InitialDate
```

Description

If the buffer was initialized from a disk file, *InitialDate* reflects the file's age. If the file does not reside on disk, *InitialDate* holds the time at which the buffer was created. It is a read-only property.

IsModified property

EditBuffer class

Indicates if the buffer was changed since it was last opened or saved, whichever occurred most recently.

Access

Read-only

Type expected

```
bool IsModified
```

IsPrivate property

EditBuffer class

TRUE if the buffer was created with the *private* parameter set to **TRUE; FALSE,** otherwise.

Access

Read-only

Type expected

```
bool IsPrivate
```

IsReadOnly property

EditBuffer class

TRUE if the buffer was created with the *readOnly* parameter set to **TRUE; FALSE** otherwise.

Access

Read-only

Type expected

```
bool IsReadOnly
```

IsValid property

EditBuffer class

FALSE if the *EditBuffer* is destroyed, otherwise, **TRUE**.

Access

Read-only

Type expected

```
bool IsValid
```

Position property

EditBuffer class

Provides access to the EditPosition instance for this *EditBuffer*.

Access

Read-only

Type expected

```
EditPosition Position
```

TopView property

EditBuffer class

The topmost *EditView* that contains this *EditBuffer.* **NULL** if no view is associated with the buffer.

Access

Read-only

Type expected

```
EditView TopView
```

ApplyStyle method

EditBuffer class

Updates the EditOptions.BufferOptions property with the contents of *styleToApply*.

Types expected

```
void ApplyStyle(EditStyle styleToApply)
```

styleToApply The EditStyle object to apply.

Return value

None

BlockCreate method

EditBuffer class

Creates an edit block for the *EditBuffer*.

Types expected

```
EditBlock BlockCreate()
```

Return value

The edit block.

Describe method

EditBuffer class

Invoked during buffer list creation by an Editor object. Returns a text description of the buffer, as in:

FOO.CPP(modified)

BAR.CPP

Types expected

```
string Describe()
```

Return value

None

Destroy method

EditBuffer class

Removes the buffer from the IDE's buffer list and does not save any changes.

Types expected

```
bool Destroy()
```

Return value

TRUE if the buffer was actually destroyed, or **FALSE** if views relying on it still exist

Description

When private *EditBuffer* objects are longer needed, you should always explicitly destroy them.

NextBuffer method

See also EditBuffer class

Finds the next edit buffer in the buffer list.

Types expected

```
EditBuffer NextBuffer(bool privateToo)
```

privateToo **TRUE** if private buffers are to be included in the buffer list, **FALSE** otherwise.

Return value

The edit buffer found or **NULL** if none is found

Description

The buffer list is circular, so if a buffer exists, it will be found. However, if all buffers are private and if *privateToo* is set to **FALSE,** no buffer will be found.

NextView method

EditBuffer class

Returns the next *EditView* containing this *EditBuffer*.

Types expected

```
EditView NextView(EditView next)
```

next The view to use in getting the next associated view for this edit buffer. Start traversing the view list by passing the value of TopView to this method.

Return value

None

Description

An *EditBuffer* is a representation of the contents of a file. An *EditView* is used to provide a visual representation of the *EditBuffer*. The same *EditBuffer* can be displayed simultaneously to the user in different *EditView*s (for example, two edit windows can be open on the same file). This method enables you to cycle through all the *EditViews* representing this *EditBuffer*.

PositionCreate method

EditBuffer class

Creates an *EditPosition* object.

Types expected

```
EditPosition PositionCreate()
```

Return value

None

Print method

EditBuffer class

Prints this buffer.

Types expected

```
bool Print()
```

Return value

TRUE if the print was successful or **FALSE** if the print failed.

PriorBuffer method

See also EditBuffer class

Finds the previous edit buffer in the buffer list.

Types expected

```
EditBuffer PriorBuffer(bool privateToo)
```

privateToo **TRUE** if private buffers are to be included in the buffer list, **FALSE** otherwise.

Return value

The edit buffer found or **NULL** if none is found

Description

The buffer list is circular, so if a buffer exists, it will be found. However, if all buffers are private and if *privateToo* is set to **FALSE,** no buffer will be found.

Rename method

EditBuffer

Changes the *EditBuffer* name.

Types expected

```
bool Rename(string newName)
```

newName The new name of the buffer.

Return value

TRUE if the operation succeeded or **FALSE** if it failed

Description

Rename fails when an *EditBuffer* with the new name is already in the buffer list. If a file with the new name already exists on disk, it is overwritten when this buffer is saved.

Save method

EditBuffer class

Writes the file associated with the buffer to disk.

Types expected

```
int Save([string newName])
```

newName The new name of the file.

Return value

The number of bytes written or 0 if the save was unsuccessful

Description

Saves the file whether it was modified or not. *Save* uses the current name of the file or *newName* if it is specified.

AttemptToModifyReadOnlyBuffer event

EditBuffer class

Triggered when an attempt is made to modify a read-only buffer.

Note: For the *EditBuffer* to be read-only, it must be created with the *readOnly* parameter set to **TRUE.** Creating an *EditBuffer* from a read-only file does not create a read-only buffer.

Types expected

```
void AttemptToModifyReadOnlyBuffer()
```

Return value

None

AttemptToWriteReadOnlyFile event

EditBuffer class

Triggered when an attempt is made to write the contents of an *EditBuffer* to a read-only file. The buffer may or may not have been created as read-only.

Types expected

```
void AttemptToWriteReadOnlyBuffer()
```

Return value

None

HasBeenModified event

EditBuffer class

Triggered when a buffer has been modified for the first time.

Types expected

```
void HasBeenModified()
```

Return value

None

EditOptions class

See also Description

This class is one of the editor classes. *EditOptions* class members hold editor characteristics of a global nature.

Syntax

```
EditOptions()
```

Properties	Access
`string BackupPath`	Read-write
`int BlockIndent`	Read-write
`BufferOptions BufferOptions`	Read-only
`string MirrorPath`	Read-write
`string OriginalPath`	Read-write
`string SyntaxHighlightTypes`	Read-write
`bool UseBRIEFCursorShapes`	Read-write
`bool UseBRIEFRegularExpression`	Read-write

Methods

None

Events

None

EditOptions class description

EditOptions class

The *EditOptions* object holds editor characteristics of a global nature, such as:

- Whether to create backups
- The destination paths for backups
- The insert/overtype setting
- The optimal fill setting
- Handling of blocks cut or copied from the buffer (scrap manipulation)
- The default regular expression language

Property values are initialized from global defaults during construction.

BackupPath property

See also EditOptions class

Contains the path where the editor stores back ups.

Access

Read-write

Type expected

```
string BackupPath
```

Description

In the IDE, *BackupPath* is set with the BackupPath option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|File.

BlockIndent property

See also EditOptions class

Indents or outdents a block of characters.

Access

Read-write

Type expected

```
int BlockIndent
```

Description

BlockIndent indicates the number of characters to indent or outdent a block of characters. The value must be between 1 and 16.

In the IDE, *BlockIndent* is set with the Block Indent option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

BufferOptions property

EditOptions class

Holds the buffer options settings for all edit buffers.

Access

Read-only

Type expected

```
BufferOptions BufferOptions
```

MirrorPath property

See also EditOptions class

Holds the path where the editor stores mirror copies of files.

Access

Read-write

Type expected

```
string MirrorPath
```

Description

In the IDE, *MirrorPath* is set with the Mirror Path option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|File.

OriginalPath property

See also EditOptions class

Holds the path where the editor stores the original files.

Access

Read-write

Type expected

```
string OriginalPath
```

Description

In the IDE, *OriginalPath* is set with the Original Path option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|File.

SyntaxHighlightTypes property

EditOptions class Example

Holds the file extensions, or file names, of the file types for which syntax highlighting is to be enabled in the editor.

Access

Read-write

Type expected

```
string SyntaxHighlightTypes
```

Description

Wild cards are permitted. Separate multiple names/extensions with a semicolon.

In the IDE, *SyntaxHighlightTypes* is set with the Syntax Extensions option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Syntax Highlighting.

SyntaxHighlightTypes example

```
//Example of SyntaxHighlightTypes
JavaSyntaxHighlight( yes ) {
  if ( yes ) {
    IDE.Editor.Options.BufferOptions.TokenFileName = "java.tok";
    // enable syntax highlighting for .java files
    IDE.Editor.Options.SyntaxHighlightTypes = "*.java";
  }
  else {
    IDE.Editor.Options.BufferOptions.TokenFileName = ""; // C++
    // enable syntax highlighting for standard C++ files
    IDE.Editor.Options.SyntaxHighlightTypes =
      "*.cpp;*.c;*.h;*.hpp;*.rh;*.rc";
  }
  //-- redraw with new option settings --
  declare EditStyle es;
  IDE.Editor.ApplyStyle( es );
}
```

UseBRIEFCursorShapes property

See also EditOptions class

When **TRUE**, the editor uses the default cursor shapes that Brief provides for insert mode and overtype mode.

Access

Read-write

Type expected

```
bool UseBRIEFCursorShapes
```

Description

In the IDE, *UseBRIEFCursorShapes* is set with the BRIEF Cursor Shapes option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Display.

UseBRIEFRegularExpression property

See also EditOptions class

When **TRUE**, complex search and search/replace operations can be performed using the Brief regular expression syntax.

Access

Read-write

Type expected

```
bool UseBRIEFRegularExpression
```

Description

In the IDE, *UseBRIEFRegularExpressions* is set with the BRIEF Regular Expressions option of the Environment Options dialog. To display this dialog box, choose Options|Environment|Editor|Options.

Editor class

See also Description

This class provides access to the IDE's internal editor. *Editor* is associated with other classes which provide the editor with its functionality.

Syntax

```
Editor()
```

Properties	Access
`EditStyle FirstStyle`	Read-only
`EditOptions Options`	Read-only
`SearchOptions SearchOptions`	Read-only
`EditBuffer TopBuffer`	Read-only
`EditView TopView`	Read-only

Methods

```
void ApplyStyle(EditStyle newOptions)
void BufferList()
BufferOptions BufferOptionsCreate()
bool BufferRedo(EditBuffer buffer)
bool BufferUndo(EditBuffer buffer)
EditBuffer EditBufferCreate(string fileName [, bool private, bool readOnly])
EditOptions EditOptionsCreate()
EditStyle EditStyleCreate(string styleName[,EditStyle toInheritFrom])
EditWindow EditWindowCreate(EditBuffer buffer)
string GetClipboard()
int GetClipboardToken()
EditWindow GetWindow([bool getLast])
bool IsFileLoaded(string filename)
EditStyle StyleGetNext(EditStyle)
bool ViewRedo(EditView view)
bool ViewUndo(EditView view)
```

Events

```
void BufferCreated(EditBuffer buffer)
void MouseBlockCreated()
void MouseLeftDown()
void MouseLeftUp()
string MouseTipRequested(EditView theView, int line, int column)
void OptionsChanged(EditorOptions newOptions)
void OptionsChanging(EditorOptions newOptions)
void ViewActivated(EditView view)
void ViewCreated(EditView newView)
void ViewDestroyed(EditView deadView)
```

Editor class description

See also Editor class

The IDE instantiates an *Editor* object, which maintains undo and redo data and has methods allowing access to the list of all buffers and edit windows. Editors have a member of type *EditOptions* that controls global editor characteristics.

Although multiple instances of *Editor* objects may be created in script, they all refer to the same instance of a single C++ object internal to the IDE. Modification of one *Editor* object's options will be reflected in all *Editor* objects.

Manipulating the Editor

The Editor's functionality is accessible at a low enough level that you can mimic in script the behavior of popular editors (such as BRIEF, Epsilon, vi, and WordStar). The Editor itself is accessed through an object instantiated from the Editor class. Because the IDE instantiates an *Editor* object itself, any *Editor* objects you instantiate point to this internal IDE object; therefore, modifications in one *Editor* object's options are reflected in all *Editor* objects.

Further editor access is provided through the following classes:

BufferOptions	Controls characteristics of the *EditBuffer*, such as margin, tab rack, syntax highlighting, and bookmarks.
EditBlock	Cut, copy, delete, dimensions, and style.
EditBuffer	Access status, save, describe, time/date stamp.
EditOptions	Holds characteristics of a global nature, such as the insert/overtype setting, optimal fill, and scrap settings (how to handle blocks cut or copied from Editor buffers).
EditPosition	Location-dependent operations in a view or buffer: cursor movement, text rip, search, insert.
EditStyle	Provide named styles that overrride settings in a buffer or the entire editor.
EditView	Access to buffer, visual cursor manipulations, zoom.
EditWindow	Pane control, access to views.

FirstStyle property

Editor class

Contains the first style in the list of editor styles.

Access

Read-only

Type expected

```
EditStyle FirstStyle
```

Description

FirstStyle is usually used with the StyleGetNext method. At least one EditStyle must exist for this property to contain a valid value.

Options property

Editor class

Holds the buffer options settings.

Access

Read-only

Type expected

```
EditOptions Options
```

Description

Options holds the options settings for all edit buffers. Changing an option in this property affects all edit buffers.

SearchOptions property

Editor class

Provides access to the instance of SearchOptions associated with this editor.

Access

Read-only

Type expected

```
SearchOptions SearchOptions
```

TopBuffer property

Editor class

The current edit buffer.

Access

Read-only

Type expected

`EditBuffer TopBuffer`

TopView property

Editor class

The current view.

Access

Read-only

Type expected

`EditView TopView`

Description

TopView provides a quick way to get at the top view associated with the current edit buffer. When you create a script which operates on the current view, obtain *TopView* from the editor as outline below:

```
//Import the instance of the IDE's editor
import editor;
PrintCurrentLineAneRow()
{
  //Get the current view's EditPosition object
  declare ep=editor.TopView.Position;
  print "Row=",ep.Row,"Column=",ep.Column
}
```

ApplyStyle method

Editor class

Updates the edit options.

Types expected

```
void ApplyStyle(EditStyle newOptions)
```

newOptions The options for the EditStyle object.

Return value

None

BufferList method

Editor class

A text description of the buffer list.

Types expected

```
void BufferList()
```

Return value

None

Description

The description returned in *BufferList* comes from the EditBuffer.Describe method.

BufferOptionsCreate method

Editor class

Creates a new instance of the BufferOptions class.

Types expected

```
BufferOptions BufferOptionsCreate()
```

Return value

A *BufferOptions* object

BufferRedo method

Editor class

Reapplies the last operation on the buffer or view regardless of whether the operation was performed on the EditBuffer, the EditView, an EditBlock, or an EditPosition.

Types expected

```
bool BufferRedo(EditBuffer buffer)
```

buffer The name of the buffer or view to reapply the operation to.

Return value

TRUE if there are more operations to redo, or **FALSE** if there are not

BufferUndo method

Editor class

Undoes the last operation on the buffer or view regardless of whether the operation was performed on the EditBuffer, the EditView, an EditBlock, or an EditPosition.

Types expected

```
bool BufferUndo(EditBuffer buffer)
```

buffer The name of the buffer or view from which to undo the operation.

Return value

TRUE if there are more operations to undo or **FALSE** if there are not

EditBufferCreate method

Editor class

Creates an edit buffer.

Types expected

```
EditBuffer EditBufferCreate(string fileName [, bool
           private, bool readOnly])
```

fileName The name of the file associated with the edit buffer.

private Implies that the buffer is a hidden system buffer. Undo information is not retained, and the *EditBuffer is* never attachable to an *EditView.* The default value is **FALSE**.

readOnly Marks the buffer as read-only. The default value is **FALSE**. Associating a read-only file with the *EditBuffer* does not make the *EditBuffer* read-only.

Return value

The edit buffer created, or **NULL** if none could be created

EditOptionsCreate method

Editor class

Creates a new instance of the EditOptions class.

Types expected

```
EditOptions EditOptionsCreate()
```

Return value

An *EditOptions* object

EditStyleCreate method

Editor class

Creates an edit style.

Types expected

```
EditStyle EditStyleCreate(string styleName[,EditStyle
          toInheritFrom])
```

styleName The name of the style to create.

toInheritFrom The name of the EditStyle object to inherit from.

Return value

The edit style created, or **NULL** if none could be created

EditWindowCreate method

Editor class

Creates an edit window.

Types expected

```
EditWindow EditWindowCreate(EditBuffer buffer)
```

buffer The name of the buffer to associate with this edit window.

Return value

The edit window created, or **NULL** if none could be created

GetClipboard method

Editor class

Returns the contents of the Windows Clipboard in a string.

Types expected

```
string GetClipboard()
```

GetClipboardToken method

Editor class

Returns the memory address of the Windows Clipboard contents.

Types expected

```
int GetClipboardToken()
```

GetWindow method

Editor class

Returns an *EditWindow*.

Types expected

```
EditWindow GetWindow([bool getLast])
```

getLast The name of the window to get.

- If *getLast* is **FALSE**, *GetWindow* returns the top level window.
- If it is **TRUE**, *GetWindow* returns the last *EditWindow* in the Z-order.

getLast defaults to **FALSE**.

Return value

None

IsFileLoaded method

Editor class

Verifies if the specified file is loaded.

Types expected

```
bool IsFileLoaded(string fileName)
```

fileName The name of the file to check for.

Return value

TRUE if a buffer by that name exists, or **FALSE** if one doesn't.

StyleGetNext method

Editor class

Gets the next style in the list of editor styles.

Types expected

```
EditStyle StyleGetNext(EditStyle)
```

Return value

The editor style that was found, or **NULL** if no editor style is found.

Description

Use with FirstStyle to access the circularly linked list representing all the editor styles. At least one EditStyle must exist for this property to contain a valid value.

ViewRedo method

Editor class

Reapplies the last operation that was undone on the buffer or view regardless of whether the operation was performed on the EditBuffer, the EditView, an EditBlock, or an EditPosition.

Types expected

```
bool ViewRedo(EditView view)
```

view The name of the buffer or view to reapply the operation to.

Return value

TRUE if there are more operations to redo, or **FALSE** if there are not

ViewUndo method

Editor class

Undoes the last operation on the buffer or view regardless of whether the operation was performed on the EditBuffer, the EditView, an EditBlock, or an EditPosition.

Types expected

```
bool ViewUndo(EditView view)
```

view The name of the buffer or view from which to undo the operation.

Return value

TRUE if there are more operations to undo, or **FALSE** if there are not

BufferCreated event

Editor class

Triggered when a new *EditBuffer* is created. The default action is to do nothing.

Types expected

```
void BufferCreated(EditBuffer buffer)
```

buffer The name of the buffer to create.

Return value

None

MouseBlockCreated event

Editor class

Triggered when the user selects a block with the mouse in the top view.

Types expected

```
void MouseBlockCreated()
```

Return value

None

MouseLeftDown event

Editor class

Triggered when the mouse left button is pressed in an Edit window.

Types expected

```
void MouseLeftDown()
```

Return value

None

MouseLeftUp event

Editor class

Triggered when the mouse left button is released in an Edit window.

Types expected

```
void MouseLeftUp()
```

Return value

None

MouseTipRequested event

Editor class

Raised when the mouse has remained idle over an editor window for a period of time.

Types expected

```
string MouseTipRequested(EditView theView, int line,
                         int column)
```

theView The *EditView* object describing the edit window that contains the idle mouse.

line, column The position in the edit buffer of the character the cursor is on.

Return value

If this routine returns a string, it displays the string to the user as a help hint. The default implementation returns a NULL.

OptionsChanged event

Editor class

Raised when the OptionsChanging event handler has completed and the global values have been changed.

Types expected

```
void OptionsChanged(EditorOptions newOptions)
```

newOptions The new global editor options to apply.

Return value

None

Description

OptionsChanged notifies a script that needs to update the global options that those options have changed.

OptionsChanging event

See also Editor class

Raised when leaving one of the editor MPD pages with accept.

Types expected

```
void OptionsChanging(EditorOptions newOptions)
```

newOptions The new global editor options.

Return value

None

Description

OptionsChanging contains a copy of the new values for the global editor options. An event handler may examine these values and determine if any of the values need to be overridden with any values from *newOptions*.

ViewActivated event

Editor class

Triggered when the EditView represented by *view* is activated. There is no default action for this event.

Types expected

```
void ViewActivated(EditView view)
```

view The name of the view to activate.

Return value

None

ViewCreated event

Editor class

Triggered when the EditView represented by *newView* is created. There is no default action for this event.

Types expected

```
void ViewCreated(EditView newView)
```

newView The name of the view to activate.

Return value

None

ViewDestroyed event

Editor class

Triggered when the *EditView* represented by *deadView* is destroyed. There is no default action for this event.

Types expected

```
void ViewDestroyed(EditView deadView)
```

deadView The name of the view to destroy.

Return value

None

EditPosition class

See also Description

This is one of the editor classes. *EditPosition* class members provide positioning functionality related to the active location in an *EditView* or *EditBuffer.*

Syntax

```
EditPosition(EditBuffer)
EditPosition(EditView)
```

Properties	Access
`int Character`	Read-only
`int Column`	Read-only
`bool IsSpecialCharacter`	Read-only
`bool IsWhiteSpace`	Read-only
`bool IsWordCharacter`	Read-only
`int LastRow`	Read-only
`int Row`	Read-only
`SearchOptions SearchOptions`	Read-only

Methods

```
void Align(int magnitude)
bool BackspaceDelete([int howMany])
bool Delete([int howMany])
int DistanceToTab(int direction)
bool GotoLine(int lineNumber)
void InsertBlock(EditBlock block)
void InsertCharacter(int characterToInsert)
void InsertFile(string fileName)
void InsertScrap()
void InsertText(string text)
bool Move([int row, int col])
bool MoveBOL()
bool MoveCursor(moveMask)
bool MoveEOF()
bool MoveEOL()
bool MoveReal([int row, int col])
bool MoveRelative([int deltaRow, int deltaCol])
string Read([int numberOfChars])
bool Replace([string pat, string rep, bool case, bool useRE, bool dir, int
  reFlavor, bool global, EditBlock block])
bool ReplaceAgain()
void Restore()
string RipText(string legalChars [,int ripFlags])
void Save()
int Search([string pat, bool case, bool useRE, bool dir, int reFlavor,
  EditBlock block])
int SearchAgain()
void Tab(int magnitude)
```

Events
None

EditPosition class description

EditPosition class

An *EditPosition* object is the point at which operations occur within the EditBuffer. One *EditPosition* object exists for each *EditBuffer* and each EditView. In the *EditView,* the cursor location visually represents the *EditPosition*'s current location.

Since each *EditView* can have its own *EditPosition* object, you can have multiple *EditView*s at multiple locations. Additionally, the *EditBuffer*'s *EditPosition* object maintains its own location information.

Character property

EditPosition class

Integer value of the character at this position or one of the following values:

VIRTUAL_TAB

VIRTUAL_PAST_EOF

VIRTUAL_PAST_EOL

Access

Read-only

Type expected

```
int Character
```

Column property

EditPosition class

The current column position in the buffer. To change, use one of the following *EditPosition* methods:

Move

MoveBOL

MoveCursor

MoveEOF

MoveEOL

MoveReal

MoveRelative

Access

Read-only

Type expected

```
int Column
```

IsSpecialCharacter property

EditPosition class

TRUE if the character at the current edit position is not an alphanumeric or whitespace character; **FALSE** otherwise.

Access

Read-only

Type expected

```
bool IsSpecialCharacter
```

IsWhiteSpace property

EditPosition class

TRUE if the character at the current edit position is a Tab or Space; **FALSE,** otherwise.

Access

Read-only

Type expected

```
bool IsWhiteSpace
```

IsWordCharacter property

EditPosition class

TRUE if the character at the current edit position is an alphabetic character, numeric character or underscore. Otherwise, **FALSE**.

Access

Read-only

Type expected

```
bool IsWordCharacter
```

LastRow property

EditPosition class

The line number of the last line in the edit buffer.

Access

Read-only

Type expected

```
int LastRow
```

Row property

EditPosition class

The current row position in the buffer. To change, use one of the following *EditPosition* methods:

Move
MoveBOL
MoveCursor
MoveEOF
MoveEOL
MoveReal
MoveRelative

Access

Read-only

Type expected

```
int Row
```

SearchOptions property

EditPosition class

Contains an instance of the SearchOptions class, the options currently in place for searching.

Access

Read-only

Type expected

```
SearchOptions SearchOptions
```

Align method

EditPosition class Example

Positions the insertion point on the current line, aligning it with columns calculated from prior lines in the file.

Types expected

```
void Align(int magnitude)
```

magnitude If positive, enough characters are inserted to align the character position as follows:

- Starting with the column defined by the current character position on the current line, the character is aligned with the first character after the first white space on the previous line after the column position.
- If the previous line is too short to calculate a position on the current line, previous lines are scanned until finding one that is long enough to calculate a column position.

If negative, the column position is moved to the left.

Return value

None

Align example

Assume that two lines of code contain the text "Leaning over the console, she stuck out her hand and said," and "'Hello there, buddy.'" The cursor (**^**) is in column 2 on the current line.

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy"
  ^
```

Calling `Align(1)` results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy."
       ^
```

Calling `Align(1)` again results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you, buddy."
              ^
```

Calling `Align(1)` again results in:

```
Leaning over the console, she stuck out her hand and said,
"How are you  buddy."
                 ^
```

Calling `Align(-1)` results in:

```
Leaning over the console, she stuck out her hand and said,
"Hello there, buddy."
              ^
```

BackspaceDelete method

EditPosition class

Deletes characters to the left of the current position.

Types expected

```
bool BackspaceDelete([int howMany])
```

howMany The number of characters to delete. The default is 1.

Return value

TRUE if any characters are deleted; **FALSE** if there are no characters to the left

Delete method

EditPosition class

Deletes characters to the right of the current position.

Types expected

```
bool Delete([int howMany])
```

howMany	The number of characters to delete. The default is 1.

Return value

TRUE if any characters are deleted; **FALSE** if there are no characters to the right

DistanceToTab method

EditPosition class

Retrieves the number of character positions between the current cursor position and the next/previous tab stop.

Types expected

```
int DistanceToTab(int direction)
```

direction	Either SEARCH_FORWARD or SEARCH_BACKWARD. SEARCH_FORWARD is the default.

Return value

Number of character positions between the current cursor position and the next/previous tab stop.

GotoLine method

EditPosition class

Moves the cursor to the specified line, without changing column position.

Types expected

```
bool GotoLine(int lineNumber)
```

lineNumber	The number of the line to change to. If *lineNumber* is not specified, the user is prompted for a line number.

Return value

TRUE if the move was successful, **FALSE,** otherwise

InsertBlock method

EditPosition class

Inserts the last marked block at the current cursor position.

Types expected

```
void InsertBlock(EditBlock block)
```

block Restricts the search to the indicated block.

Return value

None

InsertCharacter method

EditPosition class

Inserts a character at the current cursor position.

Types expected

```
void InsertCharacter(int characterToInsert)
```

characterToInsert The integer value of the character to insert.

Return value

None

InsertFile method

EditPosition class

Inserts the contents of the specified file at the current cursor position.

Types expected

```
void InsertFile(string fileName)
```

fileName The name of the file to insert.

Return value

None

InsertScrap method

EditPosition class

Insert text in the Windows Clipboard at the current cursor position.

Types expected

```
void InsertScrap()
```

Return value

None

InsertText method

EditPosition class

Inserts the specified string at the current cursor position.

Types expected

```
void InsertText(string text)
```

text The string to insert.

Return value

None

Move method

EditPosition class

Moves the cursor to the specified row and column.

Types expected

```
bool Move([int row, int col])
```

row The number of the row to move to.

col The number of the column to move to.

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

Move attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

MoveBOL method

EditPosition class

Moves the cursor to the first character on the current line.

Types expected

```
bool MoveBOL()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveBOL attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

MoveCursor method

EditPosition class

Moves the current position forward or backward in the buffer.

Types expected

```
bool MoveCursor(moveMask)
```

moveMask The position to move the cursor to. The value of *moveMask* can be built from the one of the following:

SKIP_WORD (default)

SKIP_NONWORD

SKIP_WHITE

SKIP_NONWHITE

SKIP_SPECIAL

SKIP_NONSPECIAL.

These masks can be combined with SKIP_LEFT (default) or SKIP_RIGHT. SKIP_STREAM can also be used with any of these combinations if line ends are ignored.

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveCursor attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

MoveEOF method

EditPosition class

Moves the current position to the last character in the file.

Types expected

```
bool MoveEOF()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveEOF attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

MoveEOL method

EditPosition class

Moves the current position to the last character on the line.

Types expected

```
bool MoveEOL()
```

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveEOL attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

MoveReal method

EditPosition class

The position assumes that the file is unedited. If edits have been made to the file, the move is relative to the original, unedited file.

Types expected

```
bool MoveReal([int row, int col])
```

row The number of the row to move the cursor to. *row* is relative to the line numbers in the original, unedited file.

col The number of the column to move the cursor to. *column* is relative to the column numbers in the original, unedited file.

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveReal attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

For example, assume that the original, unedited file is a two-line file with the word ONE on the first line and the word TWO on the second line. The user subsequently inserts 100 lines of text after line 1. `MoveReal(2,1)` moves the cursor to the "T" in "TWO".

MoveRelative method

EditPosition class

Moves the cursor the specified number of rows and columns from the current row and column position.

Types expected

```
bool MoveRelative([int deltaRow, int deltaCol])
```

deltaRow The number of rows to move the cursor. *deltaRow* is relative to the current row number.

deltaCol The number of columns to move the cursor. *deltaCol* is relative to the current column number.

Return value

The return value, **TRUE** or **FALSE**, indicates whether the position actually changed.

Description

MoveRelative attempts to position:

- The column at 0 or less
- The column at 1025 or more
- The row at 0 or less

MaxLineNumber + 1 or more are invalid. *MaxLineNumber* depends on the computer's capacity.

Read method

EditPosition class

Reads the specified number of characters.

Types expected

```
string Read([int numberOfChars])
```

numberOfChars The number of characters to read from the current cursor position. If omitted, it reads to the end of the line.

Return value

A string containing the characters read

Replace method

See also EditPosition class

Searches EditBuffer in the indicated direction for the search expression. The expression is replaced with the specified expression.

Types expected

```
bool Replace([string pat, string rep, bool case,
              bool useRE, bool dir, int reFlavor, bool
              global, EditBlock block])
```

pat The string to search for.

rep The string to replace with.

case Indicates if the case of *pat* is significant in the search.

useRE Indicates whether or not to interpret *pat* as a regular expression string.

dir One of the following:

SEARCH_FORWARD (default)

SEARCH_BACKWARD

reFlavor The type of regular expression being used; it may be one of the following:

IDE_RE (default)

BRIEF_RE

BRIEF_RE_FORWARD_MIN

BRIEF_RE_SAME_MIN

BRIEF_RE_BACK_MIN

BRIEF_RE_FORWARD_MAX

BRIEF_RE_SAME_MAX

BRIEF_RE_BACK_MAX

block If given, restricts the search to the indicated block.

Return value

TRUE if the replace operation was successful, **FALSE,** otherwise.

ReplaceAgain method

EditPosition class

Repeats the most recently performed Replace operation.

Types expected

```
bool ReplaceAgain()
```

Return value

TRUE if the replace operation was successful, **FALSE,** otherwise.

Restore method

EditPosition class

Restores the cursor position to the position saved by the last call to the Save method.

Types expected

```
void Restore()
```

Return value

None

RipText method

EditPosition class

Performs an edit rip operation. This routine can rip an entire line.

Types expected

```
string RipText(string legalChars [,int ripFlags])
```

legalChars Determines the legal characters to include in the edit rip. If *legalChars* is omitted:

INCLUDE_ALPHA_CHARS

INCLUDE_NUMERIC_CHARS

INCLUDE_SPECIAL_CHARS

are all automatically added to the *ripFlags* argument, making any character beween ASCII decimal 32 and 128 a legal character. A rip can be halted by specifying a character in *legalChars* then using INVERT_LEGAL_CHARS as the *ripFlags* parameter.

ripFlags A mask built by combining any or all of the following values:

Value	Description
BACKWARD_RIP	Rip from left to right.
INVERT_LEGAL_CHARS	Interpret the *legalChars* string as the inverse of the string you wish to use for *legalChars*. In other words, specify `t` to mean any ASCII value between 1 and 255 except `t`.
INCLUDE_LOWERCASE_ALPHA_CHARS	Append the characters `abcdefghijklmnopqrstuvwxyz` to the *legalChars* string.
INCLUDE_UPPERCASE_ALPHA_CHARS	Append the characters `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to the *legalChars* string.
INCLUDE_ALPHA_CHARS	Append both uppercase and lowercase alpha characters to the *legalChars* string.
INCLUDE_NUMERIC_CHARS	Append the characters `1234567890` to the *legalChars* string.
INCLUDE_SPECIAL_CHARS	Append the characters `` `-=[]\;',./~!@#$%^&*()_+{}|:"<>? `` to the *legalChars* string.

Return value

The string copied from the edit buffer

Save method

EditPosition class

Save the current cursor position. Use Restore to later restore the cursor to this position.

Types expected

```
void Save()
```

Return value

None

Search method

EditPosition class

Searches the edit buffer for the search expression.

Types expected

```
int Search(string pat [, bool case, bool useRE,
           bool dir, int reFlavor, EditBlock block])
```

pat The string to search for.

case Indicates if the case of *pat* is significant in the search.

useRE Indicates whether or not to interpret *pat* as a regular expression.

dir One of the following:

SEARCH_FORWARD (default)

SEARCH_BACKWARD

reFlavor The type of regular expression in use; it may be one of the following:

IDE_RE (default)

BRIEF_RE

BRIEF_RE_FORWARD_MIN // same as BRIEF_RE

BRIEF_RE_SAME_MIN

BRIEF_RE_BACK_MIN

BRIEF_RE_FORWARD_MAX

BRIEF_RE_SAME_MAX

BRIEF_RE_BACK_MAX

block If given, restricts the search to the indicated block.

Note: If *case*, *useRE,* or *reFlavor* is not supplied, the value is determined by querying the Editor object.

Return value

The size (in characters matched) of the match

SearchAgain method

EditPosition class

Repeats the most recently performed Search operation.

Types expected

```
int SearchAgain()
```

Return value

The number of matches found

Tab method

EditPosition class

Moves the current cursor location to the next or previous tab stop.

Types expected

```
void Tab(int magnitude)
```

magnitude If positive, moves the cursor to the next tab stop. If negative, moves to the previous tab stop.

Return value

None

EditStyle class

See also Description

This class is one of the editor classes. *EditStyle* applies styles that overrride settings for a buffer or for the entire editor.

Syntax

```
EditStyle(string styleName[,EditStyle styleToInitializeFrom])
```

styleName	The name of the style to create.
styleToInitializeFrom	The name of the style to initialize from.

Properties	Access
`EditOptions EditMode`	Read-write
`int Identifier`	Read-only
`string Name`	Read-write

Methods

None

Events

None

EditStyle class description

EditStyle class

EditStyle objects provide a mechanism to collect EditOptions, name them, and apply them across buffers, across the entire Editor, or both. You can store all your preferred settings for the editor in an *EditStyle* object and apply them to an editor all at once.

EditStyle objects contain:

- An *EditOptions* member
- A name
- An internal filter that indicates the characteristics that the style controls

EditStyles are implicitly persistent. The list of available styles may be traversed from the *Editor* object.

EditMode property

EditStyle class

Contains an EditOptions object that defines the options for the style.

Access

Read-write

Type expected

```
EditOptions EditMode
```

Identifier property

EditStyle class

Identifies styles with a unique integer.

Access

Read-only

Type expected

```
int Identifier
```

Name property

EditStyle class

A unique name for this *EditStyle,* taken from the *styleName* parameter.

Access

Read-write

Type expected

```
string Name
```

EditView class

See also Description

This class is one of the editor classes. *EditView* class members provide the visual representation of the *EditBuffer.*

- Each edit view has only one edit buffer.
- Each edit view is in an edit window.

Syntax

```
EditView (EditWindow parent[, EditBuffer buffer])
```

parent The edit window.

buffer The currently active buffer. If *buffer* is omitted, the parent's currently active *EditBuffer* is used.

Properties	Access
`EditBlock Block`	Read-write
`int BottomRow`	Read-only
`EditBuffer Buffer`	Read-only
`int Identifier`	Read-only
`bool IsValid`	Read-only
`bool IsZoomed`	Read-write
`int LastEditColumn`	Read-only
`int LastEditRow`	Read-only
`int LeftColumn`	Read-only
`EditView Next`	Read-only
`EditPosition Position`	Read-only
`EditView Prior`	Read-only
`int RightColumn`	Read-only
`int TopRow`	Read-only
`EditWindow Window`	Read-only

Methods

```
EditBuffer Attach(EditBuffer buffer)
bool BookmarkGoto(int bookmarkIDorPrevRef)
int BookmarkRecord(int bookmarkIDorPrevRef)
void Center([int row, int col])
void MoveCursorToView()
void MoveViewToCursor()
void PageDown()
void PageUp()
void Paint()
int Scroll(int deltaRow[, int deltaCol])
void SetTopLeft(int topRow, int leftCol)
```

Events

None

EditView class description

EditView class

EditView objects provide an editing window for the current buffer. The frame of an *EditView* is an EditWindow. Each view has a direct relationship to an EditBuffer. During creation, the *EditView*'s Position member is initialized from the *EditBuffer*'s Position member.

Edit views:

- Have methods that traverse their sibling views.
- Can be queried to find the associated *EditWindow* or *EditBuffer.*
- Have a *Position* member that manipulates the underlying *EditBuffer*. Typically this member is used by scripts and primitives tied to the user interface.

The underlying *EditBuffer* object owns the list of bookmarks (position markers that track text edits). Use EditView.BookmarkRecord and EditView.BookmarkGoto to provide access to those bookmarks. A common list of bookmarks is maintained for the same buffer regardless of the view being used.

Block property

EditView class

Provides access to the instance of the *EditBlock* class attached to this *EditView*.

Access

Read-write

Type expected

```
EditBlock Block
```

BottomRow property

EditView class

Row number displayed at the last line in the view.

Access

Read-only

Type expected

```
int BottomRow
```

Buffer property

EditView class

Returns the *EditBuffer* to which the view is attached.

Access

Read-only

Type expected

```
EditBuffer Buffer
```

Identifier property

EditView class

A unique identifier for each view.

Access

Read-only

Type expected

```
int Identifier
```

IsValid property

EditView class

TRUE if the view is valid, **FALSE** if it is not.

Access

Read-only

Type expected

```
bool IsValid
```

Description

The view will be invalidated if it is destroyed by the user.

IsZoomed property

EditView class

Zooms the view.

Access

Read-write

Type expected

```
bool IsZoomed
```

Description

A zoomed *EditView* expands to occupy the entire *EditWindow* client space. If an *EditView* is zoomed in an *EditWindow*, you cannot manipulate sibling views by creating, resizing or deleting them.

LastEditColumn property

EditView class

Identifies the position of the most recent edit.

Access

Read-only

Type expected

```
int LastEditColumn
```

Description

LastEditColumn works in conjunction with LastEditRow to identify the character position of the most recent edit. An edit modifies the contents of the buffer and occurs as a character or block insertion or deletion.

LastEditRow property

EditView class

Identifies the position of the most recent edit.

Access

Read-only

Type expected

```
int LastEditRow
```

Description

LastEditRow works in conjunction with LastEditColumn to identify the character position of the most recent edit. An edit modifies the contents of the buffer and occurs as a character or block insertion or deletion.

LeftColumn property

EditView class

Column number displayed at the left edge of the view.

Access

Read-only

Type expected

```
int LeftColumn
```

Next property

EditView class

The next *EditView* embedded in the same window.

Access

Read-only

Type expected

```
EditView Next
```

Position property

EditView class

Provides access to the instance of the *EditPosition* class attached to this *EditView*.

Access

Read-only

Type expected

```
EditPosition Position
```

Prior property

EditView class

The previous *EditView* embedded in the same window.

Access

Read-only

Type expected

```
EditView Prior
```

RightColumn property

EditView class

Column number displayed at the right edge of the view.

Access

Read-only

Type expected

```
int RightColumn
```

TopRow property

EditView class

Row number displayed at the first line in the view.

Access

Read-only

Type expected

```
int TopRow
```

Window property

EditView class

Returns the window in which this view is embedded.

Access

Read-only

Type expected

```
EditWindow Window
```

Attach method

EditView class

Attaches the view to a new EditBuffer.

Types expected

```
EditBuffer Attach(EditBuffer buffer)
```

buffer The name of the buffer to attach to.

Return value

The previous edit buffer

Description

Attach replaces the currently attached edit buffer. When a view is created, it is associated with an *EditBuffer*. The purpose of the view is to provide a visual representation of the edit buffer to which it is attached.

For example, to display a current view in a different edit buffer, use *Attach* to switch its associated buffer to another edit buffer.

BookmarkGoto method

EditView class

Updates the EditBuffer position with the value from the specified marker.

Types expected

```
bool BookmarkGoto(int bookmarkIDorPrevRef)
```

bookmarkIDorPrevRef Either an index (range 0-19) to the list of bookmarks in the buffer or a reference to a bookmark that was returned from a previous call to BookmarkRecord.

Return value

TRUE if the marker is valid, **FALSE** otherwise

BookmarkRecord method

EditView class

Returns a value suitable for passing to BookmarkGoto. Returns zero if there was an error.

Types expected

```
int BookmarkRecord(int bookmarkIDorPrevRef)
```

bookmarkIDorPrevRef Either an index (range 0-19) to the list of bookmarks in the buffer or a reference to a bookmark that was returned from a previous call to *BookmarkRecord*.

Return value

None

Description

Use *BookmarkRecord* to store a known location in a buffer. The bookmark moves with edit inserts and deletes.

For example, if you insert a bookmark using `BookMarkRecord(1)` at the `a` in `are` in the following line, you could move around and then return to that location with *BookmarkGoto(1)*:

```
hello how are you?
```

If the word `how` were deleted, you would still return to the `a` in `are`.

Center method

EditView class

Scrolls the *EditView* as necessary to center the character in the view window.

Types expected

```
void Center([int row, int col])
```

row The number of the row to center the character to. A 0 does not change the row number.

col The number of the column to center the character to. A 0 does not change the column number.

Return value

None

Description

Center centers the character at the specified position vertically or horizontally or both. If the character is already centered, nothing happens.

MoveCursorToView method

EditView class

Ensures that the cursor is visible in the view by altering the cursor's position, if necessary.

Types expected

```
void MoveCursorToView()
```

Return value

None

MoveViewToCursor method

EditView class

Ensures that the cursor is visible in the view by altering the view's coordinates, if necessary.

Types expected

```
void MoveViewToCursor()
```

Return value

None

PageDown method

EditView class

Advances the row position by the number of visible rows in the *EditView*.

Types expected

```
void PageDown()
```

Return value

None

PageUp method

EditView class

Moves the cursor toward the top of the buffer by the number of lines in the visible rows in the *EditView*.

Types expected

```
void PageUp()
```

Return value

None

Paint method

EditView class

Forces a screen refresh. During normal script execution, screen updates are suppressed.

Types expected

```
void Paint()
```

Return value

None

Scroll method

EditView class

Scrolls in the direction indicated and returns the number of lines actually scrolled.

Types expected

```
int Scroll(int deltaRow[, int deltaCol])
```

deltaRow The direction in which to scroll.

- A value less than 0 means scroll up by the specified number of lines.
- A value greater than 0 means scroll down by the specified number of lines.

deltaCol The magnitude of the scroll.

- A value less than 0 means scroll left by the specified number of columns.
- A value greater than 0 means scroll down by the specified number of columns.

Return value

Number of lines and columns scrolled

SetTopLeft method

EditView class

Attempts to position the character at the specified position in the upper left corner of the *EditView*. Might fail if the position is outside the window's bounds.

Types expected

```
void SetTopLeft(int topRow, int leftCol)
```

topRow The row number of the upper left corner of the *EditView.* A 0 ignores the position request and sets only the column number.

leftCol The column number of the upper left corner of the *EditView.* A 0 ignores the position request and sets only the row number.

Note: A zero in both parameters causes the method to be ignored altogether.

Return value

None

EditWindow class

See also Description

This class is one of the editor classes. *EditWindow* class members provide control of editor views.

Syntax

```
EditWindow(EditBuffer buffer)
```

buffer The name of the *EditBuffer* to create.

Properties	Access
`int Identifier`	Read-only
`bool IsHidden`	Read-write
`bool IsValid`	Read-only
`EditWindow Next`	Read-only
`EditWindow Prior`	Read-only
`string Title`	Read-write
`EditView View`	Read-only

Methods

```
void Activate()
void Close()
void Paint()
EditView ViewActivate(int direction[, EditView srcView])
EditView ViewCreate(int direction[, EditView srcView])
bool ViewDelete(int direction[, EditView srcView])
EditView ViewExists(int direction[, EditView srcView])
void ViewSlide(int direction[, int magnitude, EditView srcView])
```

Events

None

EditWindow class description

EditWindow class

EditWindow objects manage window panes (also known as views). An *EditWindow* can contain one or more views in which each *EditView* represents different buffers.

Creation of an *EditWindow* does not cause a window to appear; it provides an object to which a view may be attached. As soon as the first view is attached to an *EditWindow*, it can be displayed.

Views can be zoomed, in which case they expand to fill the client area of their *EditWindow.* A zoomed view hides all sibling views. Sibling views are those embedded in the same *EditWindow.* As long as an *EditWindow* contains a zoomed view, views can't be created, destroyed or resized.

*EditWindow*s can be hidden and unhidden to allow the user to free screen space and preserve the view layout.

Identifier property

EditWindow class

Identifies views with a unique value.

Access

Read-write

Type expected

```
int Identifier
```

IsHidden property

EditWindow class

Indicates if the current *EditWindow* is hidden.

Access

Read-write

Type expected

```
bool IsHidden
```

IsValid property

EditWindow class

TRUE if the current *EditWindow* is ready for edit operations, **FALSE** if the window is not available (for example, it is closed).

Access

Read-only

Type expected

```
bool IsValid
```

Next property

EditWindow class

Indicates the next *EditWindow*, if any.

Access

Read-only

Type expected

```
EditWindow Next
```

Prior property

EditWindow class

Indicates the previous *EditWindow*, if any.

Access

Read-only

Type expected

```
EditWindow Prior
```

Title property

EditWindow class

Indicates the title of the current *EditWindow*.

Access

Read-write

Type expected

```
string Title
```

View property

EditWindow class

Indicates the current *EditView*.

Access

Read-only

Type expected

```
EditView View
```

Activate method

EditWindow class

Brings this window to the top and gives it focus.

Types expected

```
void Activate()
```

Return value

None

Close method

EditWindow class

Closes the current window.

Types expected

```
void Close()
```

Return value

None

Paint method

EditWindow class

Forces a screen refresh. During normal script execution screen updates are suppressed.

Types expected

```
void Paint()
```

Return value

None

ViewActivate method

EditWindow class

Makes an existing view the current, active view.

Types expected

```
EditView ViewActivate(int direction[, EditView srcView])
```

direction Relative to the current EditView in an *EditWindow*. If *direction* is 0, and a *srcView* is specified, the specified *srcView* is activated. *direction* can be one of the following values:

UP

DOWN

LEFT

RIGHT

srcView The view to activate. If omitted, the *EditWindow*'s current *EditView* is activated.

Return value

The newly activated view or NULL if no view exists

ViewCreate method

EditWindow class

Creates an *EditView*.

Types expected

```
EditView ViewCreate(int direction[, EditView srcView])
```

direction Relative to the existing EditView(s) in an *EditWindow.* Ignored for the first view. *direction* can be one of the following values:

UP

DOWN

LEFT

RIGHT

srcView The view to create. If omitted, the *EditWindow*'s current *EditView* is used. By default, the newly created *EditView* is not activated.

Return value

The new *EditView* or **NULL** if creation failed

ViewDelete method

EditWindow class

Deletes the view in the *direction* relative to the *srcView*, if any.

Types expected

```
bool ViewDelete(int direction[, EditView srcView])
```

direction Relative to the existing EditView(s) in an *EditWindow* and ignored for the first view. Can be one of the following values:

UP

DOWN

LEFT

RIGHT

srcView The view to delete. If omitted, the *EditWindow*'s current *EditView* is deleted. The target view (if any) is then removed from the *EditWindow. srcView* is then resized to occupy the space previously held by the target view.

Return value

TRUE if the view was deleted, **FALSE** otherwise

ViewExists method

EditWindow class

Gets a reference to an adjoining EditView, if the adjoining *EditView* exists.

Types expected

```
EditView ViewExists(int direction[, EditView srcView])
```

direction Relative to the current *EditView* in an *EditWindow*. Can be one of the following values:

UP

DOWN

LEFT

RIGHT

srcView If omitted, the *EditWindow*'s current *EditView* is used.

Return value

The *EditView* or NULL if the *EditView* does not exist

ViewSlide method

EditWindow class

Moves the view in the direction indicated.

Types expected

```
void ViewSlide(int direction[, int magnitude,
               EditView srcView])
```

direction Relative to the existing EditView in an *EditWindow*. Can be one of the following values:

UP

DOWN

LEFT

RIGHT

magnitude The direction (+ or -) and amount to move

srcView If omitted, the *EditWindow*'s current *EditView* is used.

Return value

None

IDEApplication class

Description

This class represents the Borland C++ Integrated Development Environment (IDE). An *IDEApplication* object called *IDE* is instantiated when Borland C++ starts up. You typically use this class to determine how to use or extend this IDE object.

Syntax

```
IDEApplication()
```

Properties	Access
string Application	Read-only
string Caption	Read-write
string CurrentDirectory	Read-only
string CurrentProjectNode	Read-only
string DefaultFilePath	Read-write
Editor Editor	Read-only
string FullName	Read-only
int Height	Read-write
int IdleTime	Read-only
int IdleTimeout	Read-write
int LoadTime	Read-only
string KeyboardAssignmentFile	Read-write
KeyboardManager KeyboardManager	Read-only
int Left	Read-write
string ModuleName	Read-only
string Name	Read-only
string Parent	Read-only
bool RaiseDialogCreatedEvent	Read-write
string StatusBar	Read-write
int Top	Read-write
bool UseCurrentWindowForSourceTracking	Read-write
int Version	Read-only
bool Visible	Read-write
int Width	Read-write

Methods

```
void AddToCredits()
bool CloseWindow()
bool DebugAddBreakpoint()
bool DebugAddWatch()
bool DebugAnimate()
bool DebugAttach()
bool DebugBreakpointOptions()
string DebugEvaluate()
bool DebugInspect()
bool DebugInstructionStepInto()
```

```
bool DebugInstructionStepOver()
bool DebugLoad()
bool DebugPauseProcess()
bool DebugResetThisProcess()
bool DebugRun()
bool DebugRunTo()
bool DebugSourceAtExecutionPoint()
bool DebugStatementStepInto()
bool DebugStatementStepOver()
bool DebugTerminateProcess()
int DirectionDialog(string prompt)
string DirectoryDialog(string prompt, string initialValue)
void DisplayCredits()
bool DoFileOpen(string filename, string toolName [, ProjectNode node])
bool EditBufferList()
bool EditCopy()
bool EditCut()
bool EditPaste()
bool EditRedo()
bool EditSelectAll()
bool EditUndo()
void EndWaitCursor()
void EnterContextHelpMode()
void ExpandWindow()
bool FileClose()
string FileDialog(string prompt, string initialValue)
bool FileExit( [int IDEReturn] )
bool FileNew([string toolName, string fileName])
bool FileOpen([string name, string toolName])
bool FilePrint(bool suppressDialog)
bool FilePrinterSetup()
bool FileSave()
bool FileSaveAll()
bool FileSaveAs([string newName])
bool FileSend()
int GetRegionBottom(string RegionName)
int GetRegionLeft(string RegionName)
int GetRegionRight(string RegionName)
int GetRegionTop(string RegionName)
bool GetWindowState()
void Help(string helpFile, int command, string  helpTopic)
bool HelpAbout()
bool HelpContents()
bool HelpKeyboard()
bool HelpKeywordSearch([string keyword])
bool HelpOWLAPI()
```

```
bool HelpUsingHelp()
bool HelpWindowsAPI()
string KeyPressDialog(string prompt, string default)
string[ ] ListDialog(string prompt, bool multiSelect, bool sorted, string
  [ ] initialValues)
void Menu()
bool Message(string text, int severity)
int MessageCreate(string destinationTab, string toolName, int messageType,
  int parentMessage, string filename, int lineNumber, int columnNumber,
  string text, string helpFileName, int helpContextId)
bool NextWindow(bool priorWindow)
bool OptionsEnvironment()
bool OptionsProject()
bool OptionsSave()
bool OptionsStyleSheets()
bool OptionsTools()
bool ProjectAppExpert()
bool ProjectBuildAll([bool suppressOkay, string nodeName])
bool ProjectCloseProject()
bool ProjectCompile([string nodeName])
bool ProjectGenerateMakefile([string nodeName])
bool ProjectMakeAll([bool suppressOkay, string nodeName])
bool ProjectManagerInitialize()
bool ProjectNewProject([string pName])
bool ProjectNewTarget( [string nTarget, int targetType, int platform, int
  libraryMask, int modelOrMode] )
bool ProjectOpenProject([string pName])
void Quit()
bool SaveMessages(string tabName, string fileName)
bool ScriptCommands()
bool ScriptCompileFile(string fileName)
bool ScriptModules()
bool ScriptRun([string command])
bool ScriptRunFile([string filename])
bool SearchBrowseSymbol([string sName])
bool SearchFind([string pat])
bool SearchLocateSymbol([string sName])
bool SearchNextMessage()
bool SearchPreviousMessage()
bool SearchReplace([string pat, string rep])
bool SearchSearchAgain()
bool SetRegion(string RegionName, int left, int top, int right, int bottom)
bool SetWindowState(int desiredState)
string SimpleDialog(string prompt, string initialValue [, int maxNumChars])
void SpeedMenu()
void StartWaitCursor()
```

string StatusBarDialog(string prompt, string initialValue [, int maxNumChars])
bool StopBackgroundTask()
bool Tool([string toolName, string commandstring])
void Undo()
bool ViewActivate(int direction)
bool ViewBreakpoint()
bool ViewCallStack()
bool ViewClasses()
bool ViewClassExpert()
bool ViewCpu()
bool ViewGlobals()
bool ViewMessage([string tabName])
bool ViewProcess()
bool ViewProject()
bool ViewSlide(int direction [, int amount])
bool ViewWatch()
bool WindowArrangeIcons()
bool WindowCascade()
bool WindowCloseAll([string typeName])
bool WindowMinimizeAll([string typeName])
bool WindowRestoreAll([string typeName])
bool WindowTileHorizontal()
bool WindowTileVertical()
string YesNoDialog(string prompt, string default)

Events

void BuildComplete(bool status, string inputPath, string OutputPath)
void BuildStarted()
void DialogCreated(string dialogName, int dialogHandle)
void Exiting()
void HelpRequested(string filename, int command, int data)
void Idle()
void KeyboardAssignmentsChanged(string newFilename)
void KeyboardAssignmentsChanging(string newFilename)
void MakeComplete(bool status, string inputPath, string outputPath)
void MakeStarted()
void ProjectClosed(string projectFileName)
void ProjectOpened(string projectFileName)
void SecondElapsed()
void Started(bool VeryFirstTime)
void SubsytemActivated(string systemName)
bool TransferOutputExists(TransferOutput output)
void TranslateComplete(bool status, string inputPath, string outputPath)

IDEApplication class description

See also IDEApplication class

When you start the Borland C++ IDE, the object *IDE,* in *IDEApplication,* is automatically created as a global object. IDE gives you control over the system. All items contained in menu commands can be accessed through the IDE object.

The *IDE* object is registered as a Windows automation server, so any automation controller can programatically run the full IDE.

To view *IDEApplciation* member functions that correspond to menu commands, go to the topic: IDEApplication Function Groups

IDEApplication function groups

This table shows the main function groups, according to the menu they correspond to:

Group	Description
Debug	Corresponds to the Debug menu. Use these functions to load the debugger, run it, set breakpoints, add watches, and inspect variables.
Edit	Corresponds to the Edit menu. Use these functions to undo, redo, cut, copy, paste and select text in an edit window.
File	Corresponds to the File menu. Use these functions to create, open, close, save and print files.
Help	Corresponds to the Help menu. Use these functions to display the Help contents, perform keyword searches, get help about the keyboard and get help about using help.
Options	Corresponds to the Options menu. Use these functions to set options for the project and the working environment, to customize the Tools menu and to create and edit style sheets.
Project	Corresponds to the Project menu. Use these functions to open and close a project, compile a file, build the project or rebuild the entire project.
Search	Corresponds to the Search command. Uses these functions to search for text, replace text and search for symbols.
Script	Corresponds to the Script command. Use these functions to load, run and compile script files.
View	Corresponds to the View menu. Use these commands to display the Project window, Message window, the Classes window, the Globals window, the CPU window, the Processes window, the Watches window, the Breakpoint window and the Stack window.
Window	Corresponds to the Window menu. Use these commands to arrange editor windows, close windows, minimize and maximize windows and restore them.

Application property

IDEApplication class

Contains the *IDEApplication* object's internal name.

Access

Read-only

Type expected

```
string Application
```

Description

The internal name is used by Windows. Its presence is required by Microsoft guidelines for automation servers. It serves as a starting place for an automation controller, like Word or Excel.

Caption property

IDEApplication class

Gets and sets the caption of the Borland C++ IDE main window.

Access

Read-write

Type expected

```
string Caption
```

CurrentDirectory property

IDEApplication class

The application's current directory.

Access

Read-only

Type expected

```
string CurrentDirectory
```

Description

Whenever a project file is opened, the value of *CurrentDirectory* changes to the directory containing the project file.

CurrentProjectNode property

See also IDEApplication class

The name of the node currently selected in the Project window.

Access

Read-only

Type expected

```
string CurrentProjectNode
```

Description

If the Project window is closed, or if multiple nodes are selected in the Project window, *CurrentProjectNode* contains an empty string ("").

DefaultFilePath property

IDEApplication class

The default file path for the Borland C++ IDE.

Access

Read-only

Type expected

```
string DefaultFilePath
```

Editor property

IDEApplication class

An instance of the Borland C++ IDE editor.

Access

Read-only

Type expected

```
Editor Editor
```

FullName property

IDEApplication class

Contains the string, "Borland C++ for Windows, vers. 5.02".

Access

Read-only

Type expected

```
string FullName
```

Height property

IDEApplication class

The height of the Borland C++ IDE main window.

Access

Read-only

Type expected

```
int Height
```

IdleTime property

IDEApplication class

The number of seconds since the last user-generated event.

Access

Read-only

Type expected

```
int IdleTime
```

IdleTimeout property

IDEApplication class

The number of seconds the IDE must remain idle before an idle event will be generated.

Access

Read-write

Type expected

```
int IdleTimeout
```

Description

IdleTimeOut defaults to 180 (3 minutes).

LoadTime property

IDEApplication class

The number of milliseconds it takes for the IDE to load.

Access

Read-only

Type expected

```
int LoadTime
```

Description

LoadTime reflects time through the processing of the startup script. Thereafter it remains fixed.

KeyboardAssignmentFile property

See also IDEApplication class

The name of the keyboard file (.KBD) most recently selected from the Options|Environment|Editor dialog.

Access

Read-write

Type expected

```
string KeyboardAssignmentFile
```

KeyboardManager property

IDEApplication class

An instance of the Borland C++ IDE keyboard manager.

Access

Read-only

Type expected

```
KeyboardManager KeyboardManager
```

Left property

IDEApplication class

The left coordinate of the IDE main window.

Access

Read-write

Type expected

```
int Left
```

ModuleName property

IDEApplication class

The module name of the running application, including its path. For example:

```
c:\bc5\bin\bcw.exe
```

Access

Read-only

Type expected

```
string ModuleName
```

Name property

IDEApplication class

The name of the Borland C++ IDE, BCW.

Access

Read-only

Type expected

```
string Name
```

Parent property

IDEApplication class

A value required by Windows.

Access

Read-only

Type expected

```
string Parent
```

Description

Parent is required by Microsoft conventions.

RaiseDialogCreatedEvent property

IDEApplication class

Initialized to **FALSE**. Setting it to **TRUE** causes the DialogCreated event to be raised whenever a new dialog is created.

Access

Read-write

Type expected

```
bool RaiseDialogCreatedEvent
```

StatusBar property

IDEApplication class

Gets or sets the text displayed in the IDE's status bar.

Access

Read-write

Type expected

```
bool StatusBar
```

Top property

IDEApplication class

The top coordinate of the IDE main window.

Access

Read-write

Type expected

```
int Top
```

UseCurrentWindowForSourceTracking property

IDEApplication class

If **TRUE,** the IDE replaces the contents of the active Edit window whenever a new file is loaded. If **FALSE,** the IDE opens a new Edit window.

Access

Read-write

Type expected

```
bool UseCurrentWindowForSourceTracking
```

Version property

IDEApplication class

The value 502 for Borland C++ version 5.02.

Access

Read-only

Type expected

```
int Version
```

Visible property

IDEApplication class

If **TRUE**, makes the IDE visible to the user. If **FALSE**, the IDE is not visible on the screen.

Access

Read-write

Type expected

```
bool Visible
```

Width property

IDEApplication class

The width of the IDE main window.

Access

Read-write

Type expected

```
int Width
```

AddToCredits method

IDEApplication class

Adds a name to the list of developer credits in the About dialog box.

Types expected

```
void AddToCredits()
```

Return value

None

Description

AddToCredits adds the new name to the end of the existing list.

Note: To display developer credits, choose Help|About and press Alt-I.

CloseWindow method

IDEApplication class

Closes the currently selected IDE child window.

Types expected

```
bool CloseWindow()
```

Return value

TRUE if the window closed, **FALSE** if unable to close the window

DebugAddBreakpoint method

See also IDEApplication class

Opens the Add Breakpoint dialog.

Types expected

```
bool DebugAddBreakpoint()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugAddBreakpoint corresponds to the Debug|Add Breakpoint command.

DebugAddWatch method

See also IDEApplication class

Adds a watch on the current symbol.

Types expected

```
bool DebugAddWatch()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

When you call *DebugAddWatch* from an active Edit window, the Add Watch dialog box contains selected text, or if no text is selected, it contains the word at the cursor.

After you add the watch, the Watches window is displayed.

DebugAddWatch corresponds to the Debug|Add Watch command.

DebugAnimate method

IDEApplication class

Lets you watch your program's execution in "slow motion."

Types expected

```
bool DebugAnimate()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugAnimate performs a continuous series of StatementStepInto commands.

To interrupt animation, invoke one of the following *Debugger* methods either by menu selections or by keystrokes tied to the script:

Run

RunToAddress

RunToFileLine

PauseProgram

Reset

TerminateProgram

FindExecutionPoint

DebugAttach method

See also IDEApplication class

Invokes the debugger for the currently executing process.

Types expected

```
bool DebugAttach()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

Use *DebugAttach* to begin a debugging session on a process that is already running. This is useful when you know approximately when the problem occurs during program execution, but you are not sure of the corresponding location in the program source code.

DebugAttach opens the Attach to Program dialog box.

DebugBreakpointOptions method

See also IDEApplication class

Opens the Breakpoint Condition/Action Options dialog.

Types expected

```
bool DebugBreakpointOptions()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugBreakpointOptions corresponds to the Debug|Breakpoint Options command.

DebugEvaluate method

IDEApplication class

Evaluates the current expression, such as a global or local variable or an arithmetic expression.

Types expected

```
string DebugEvaluate()
```

Return value

The result of the evaluation

DebugInspect method

See also IDEApplication class

Opens the Inspect Expression dialog box for the current symbol.

Types expected

```
bool DebugInspect()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugInspect has effect only when the integrated debugger is paused in a program you are debugging.

DebugInspect corresponds to the Debug|Inspect command.

DebugInstructionStepInto method

IDEApplication class

Executes the next instruction, stepping into any function calls.

Types expected

```
bool DebugInstructionStepInto()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *DebugInstructionStepInto* first loads the executable for the current project.

DebugInstructionStepOver method

IDEApplication class

Executes the next instruction, running any functions called at full speed.

Types expected

```
bool DebugInstructionStepOver()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *DebugInstructionStepOver* first loads the executable for the current project.

DebugLoad method

See also IDEApplication class

Loads the current executable into the debugger.

Types expected

```
bool DebugLoad()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

Upon loading, the process is run to the starting point as specified in the Options|Environment|Debugger| Debugger Behavior dialog.

If the parameter is **NULL**, this method opens the Load Program dialog.

DebugPauseProcess method

See also IDEApplication class

Causes the debugger to pause the current process.

Types expected

```
bool DebugPauseProcess()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugPauseProcess has an effect only if the current process is running or is animated. It corresponds to the Debug|Pause Process command.

DebugResetThisProcess method

See also IDEApplication class

Resets the current process to its starting point as specified in the Options|Environment|Debugger| Debugger Behavior dialog.

Types expected

```
bool DebugResetThisProcess()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugResetThisProcess corresponds to the Debug|Reset This Process command.

DebugRun method

See also IDEApplication class

Causes the debugger to run the current process.

Types expected

```
bool DebugRun()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If no process is loaded, *DebugRun* first loads the executable associated with the current project.

DebugRun corresponds to the Debug|Run command.

DebugRunTo method

IDEApplication class

Causes the debugger to run the current process.

Types expected

```
bool DebugRunTo()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If *DebugRunTo* is called while working with an EditView, the current process runs until the source at the current line in the current file is encountered.

If the current object is not an *EditView, DebugRunTo* runs the current process until the instruction at the current address is encountered.

If no process is loaded, *DebugRunTo* first loads the executable associated with the current project.

DebugSourceAtExecutionPoint method

See also IDEApplication class

Displays the source code at the current execution point.

Types expected

```
bool DebugSourceAtExecutionPoint()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

The current execution point is indicated by the EIP register. If the current execution point is in source code, the execution point is shown in an Edit window. (The appropriate source file is opened if necessary.)

If the current execution point is at an address that has no source associated with it, the execution point is shown in a CPU view. (One is opened if necessary.)

DebugSourceAtExecutionPoint corresponds to the Debug|Source At Execution Point command.

DebugStatementStepInto method

IDEApplication class

Executes the next source statement and steps through the source of any function calls.

Types expected

```
bool DebugStatementStepInto()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *DebugStatementSetpInto* first loads the executable for the current project.

DebugStatementStepOver method

IDEApplication class

Executes the next source statement and does not step into any functions called, but runs them at full speed.

Types expected

```
bool DebugStatementStepOver()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

If a process is not loaded, *DebugStatementStepOver* first loads the executable for the current project.

DebugTerminateProcess method

See also IDEApplication class

Terminates the current process.

Types expected

```
bool DebugTerminateProcess()
```

Return value

TRUE if successful, **FALSE,** otherwise

Description

DebugTerminateProcess:

- Stops the current debugging sesson
- Releases memory your program has allocated and some of the memory used by the debugger
- Closes any open files that your program was using

If no process is loaded, *DebugTerminateProcess* has no effect.

DebugTerminateProcess corresponds to the Debug|Terminate Process command.

DirectionDialog method

IDEApplication class

Invokes a dialog that allows the user to specify a direction.

Types expected

```
int DirectionDialog(string prompt)
```

prompt The value to place in the caption of the dialog.

Return value

One of the following values: CANCEL, RIGHT, LEFT, UP, DOWN

DirectoryDialog method

IDEApplication class

Invokes a directory-browsing dialog box that lets the user choose a directory.

Types expected

```
string DirectoryDialog(string prompt, string initialValue)
```

prompt	The value to place in the caption of the dialog.
initialValue	The directory in which to start browsing.

Return value

If successful, this method returns a fully qualified directory name. If the user cancels, it returns the empty string ("").

DisplayCredits method

IDEApplication class

Displays the list of developer credits in the About dialog box.

Types expected

```
void DisplayCredits()
```

Return value

None

Description

To display developer credits, choose Help|About and press Alt-I.

DoFileOpen method

IDEApplication class

Opens the specified file.

Types expected

```
bool DoFileOpen(string fileName, string toolName [,ProjectNode node])
```

fileName	The name of the file to open. If the specified file does not exist, it is created.
toolName	The name of the tool to be associated with the file to open. Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can run a DOS program with the Windows IDE transfer. If *toolName* is not provided, a default is used.
node	The *node* argument is passed if the file is to be associated with a specific node in the project.

Return value

TRUE is successful, **FALSE,** otherwise

Description

DoFileOpen is used internally by the FileOpen method to open files.

EditBufferList method

See also IDEApplication class

Displays the Buffer List dialog.

Types expected

```
bool EditBufferList()
```

Return value

TRUE if the buffer list was successfully edited, **FALSE** if no edit buffers exist

Description

The Buffer List displays a list of buffers. If a file has been changed since it was last saved, the label **(modified)** appears after the filename.

Use *EditBufferList* to replace the contents of an Edit window without closing the original file. If the file you replace is not loaded in another Edit window, it is hidden. You can then later use the buffer list to load the hidden buffer into an Edit window.

EditBufferList corresponds to the Edit|Buffer List command.

EditCopy method

See also IDEApplication class

Copies selected text from the current edit buffer to the Windows Clipboard.

Types expected

```
bool EditCopy()
```

Return value

TRUE if the topmost window is an EditView with a valid marked block, **FALSE,** otherwise

Description

EditCopy leaves the selected text intact. To paste the copied text into any other document or somewhere else in the same document, use EditPaste.

EditCopy is only available if an Edit window is currently active and text has been marked for selection.

EditCopy corresponds to the Edit|Copy command.

EditCut method

See also IDEApplication class

Copies selected text from the current edit buffer to the Clipboard and deletes the selected text.

Types expected

```
bool EditCut()
```

Return value

TRUE if the topmost window is an EditView with a valid marked block, **FALSE,** otherwise

Description

EditCut removes the selected text from the Edit window. To paste the cut text into any other document or somewhere else in the same document, use EditPaste.

EditCut is only available if an Edit window is currently active and text has been marked for selection.

You can paste the cut text as many times as you want until you choose *EditCut* again or EditCopy.

EditCut corresponds to the Edit|Cut command.

EditPaste method

See also IDEApplication class

Copies selected text from the Clipboard to the current edit position in the current edit buffer.

Types expected

```
bool EditPaste()
```

Return value

TRUE if the topmost window is an EditView with a valid marked block, **FALSE,** otherwise

Description

EditPaste inserts the contents of the Clipboard into the current window at the cursor position.

EditPaste is available only if an Edit or Resource Editor window is currently active and there is something to paste.

EditPaste corresponds to the Edit|Paste command.

EditRedo method

See also IDEApplication class

Reapplies the operation that was undone with the last EditUndo.

Types expected

```
bool EditRedo()
```

Return value

TRUE if the operation was successful, **FALSE,** otherwise

Description

EditRedo only has an effect immediately after an *EditUndo* or another *EditRedo.*

A series of *EditRedo* calls reverses the effects of a series of *EditUndo* calls.

EditRedo is available only if an Edit window is currently active and there is something to redo.

EditRedo corresponds to the Edit|Redo command.

EditSelectAll method

See also IDEApplication class

Selects all the text in the current edit buffer.

Types expected

```
bool EditSelectAll()
```

Return value

TRUE if the select was successful, **FALSE,** otherwise

Description

EditSelectAll selects the entire contents of the active Edit window.

You can then use EditCopy or EditCut to copy it to the Clipboard, or perform any other editing action.

EditSelectAll is available only if an Edit or Resource Editor window is currently active.

EditSelectAll corresponds to the Edit|Select All command.

EditUndo method

See also IDEApplication class

Undoes the last edit operation.

Types expected

```
bool EditUndo()
```

Return value

TRUE if the operation was successful, **FALSE,** otherwise

Description

EditUndo restores the file in the current window to the way it was before your most recent edit or cursor movement.

EditUndo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to a prior position.

If you undo a block operation, your file will appear as it was before you executed the block operation.

EditUndo will not change an option setting that affects more than one window or reverse any toggle setting that has a global effect; for example, Ins/Ovr.

EditUndo is available only if an Edit window is currently active and there is something to undo.

EditUndo corresponds to the Edit|Undo command.

EndWaitCursor method

IDEApplication class

Stops the display of the Windows wait cursor (by default, an hourglass).

Types expected

```
void EndWaitCursor()
```

Return value

None

EnterContextHelpMode method

IDEApplication class

Puts the IDE in help context mode.

Types expected

```
void EnterContextHelpMode()
```

Return value

None

Description

After *EnterContextHelpMode* is called, the next click of the mouse generates a help event for whatever the mouse pointer is on.

ExpandWindow method

IDEApplication class

Increases the size of the currently selected window to its maximum view managed size, defined by calls to SetRegion.

Types expected

```
void ExpandWindow()
```

Return value

None

Description

After the window has been expanded with *ExpandWindow,* there is no way to decrease its size.

FileClose method

See also IDEApplication class

Closes the file that is currently open and selected.

Types expected

```
bool FileClose()
```

Return value

TRUE if the file was successfully closed, **FALSE,** otherwise

Description

If the project window is active, this command unloads the current project and closes the project tree including all project nodes.

FileClose corresponds to the File|Close command.

FileDialog method

See also IDEApplication class

Invokes an Open a File dialog box and lets the user choose a file.

Types expected

```
string FileDialog(string prompt, string initialValue)
```

prompt The value to place in the caption of the dialog.

initialValue The value to initialize the edit field with.

Return value

Returns a fully qualified file name if successful. If the user cancels, the method returns the empty string ("").

FileExit method

See also IDEApplication class

Closes the application after first ensuring that all files are saved.

Types expected

```
bool FileExit( [int IDEReturn] )
```

IDEReturn The return value of the IDE application when it exits. By default, this value is 0.

Return value

TRUE if the application was closed, **FALSE,** otherwise

Description

FileExit corresponds to the File|Exit command.

FileNew method

See also IDEApplication class

Creates a new file with the extension .CPP.

Types expected

```
bool FileNew([string toolName, string fileName])
```

toolName The name of the tool to associate with the file to open. Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can run a DOS program with the Windows IDE transfer. If *toolName* is not provided, a default is used.

fileName The name of the new file.

Return value

TRUE if the file was created, **FALSE,** otherwise

Description

FileNew opens a blank Edit window and loads a file with the default name NONAMExx.CPP (where xx stands for a number). It automatically makes the new Edit window active. NONAME files are used as a temporary edit buffer and the Borland C++ IDE prompts you to supply a new name when saved. If you load a file into an active Edit window that contains an empty NONAME file, the contents of the Edit window is replaced.

FileNew corresponds to the File|New|Text Edit command.

FileOpen method

See also IDEApplication class

Opens a file. Internally, this method uses DoFileOpen.

Types expected

```
bool FileOpen([string name, string toolName])
```

name The name of the file to open. If the specified file doesn't exist, the user is prompted for a file name.

toolName The name of the tool to associate with the file being opened. Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can run a DOS program with the Windows IDE transfer. If *toolName* is not provided, a default is used.

Return value

TRUE if the file was opened, **FALSE,** otherwise

Description

FileOpen displays the Open a File dialog box that lets you select a file to load into the Borland C++ IDE. Use this command to open a project (.PRJ or IDE), source file (.C or CPP), resource (.RC), script (.SPP or SPX), or any other type of file. The IDE automatically loads the file into the default viewer.

FileOpen corresponds to the File|Open command.

FilePrint method

See also IDEApplication class

Prints the contents of the active edit window.

Types expected

```
bool FilePrint(bool suppressDialog)
```

suppressDialog If set to **TRUE**, *FilePrint* does not display the Printer Options dialog prior to performing the print operation but reuses the last print options specified.

Return value

TRUE if the print operation was successful, **FALSE,** otherwise

Description

FilePrint corresponds to the File|Print command.

FilePrinterSetup method

See also IDEApplication class

Displays the Printer Setup dialog box.

Types expected

```
bool FilePrinterSetup()
```

Return value

TRUE if the dialog sets the options or **FALSE** if the user exits with Cancel

Description

FilePrinterSetup displays the system Printer Setup dialog box where you select which printer you want to use for printing with the Borland C++ IDE. *FilePrinterSetup* does not have an effect if no printer is detected.

FilePrinterSetup corresponds to the File|Printer Setup command.

FileSave method

See also IDEApplication class

Saves the file in the active Edit window.

Types expected

```
bool FileSave()
```

Return value

TRUE if the file was saved, **FALSE,** otherwise

Description

If the file in the active Edit window has as a default name (such as NONAME00.CPP), *FileSave* opens the Save File As dialog box so you can rename the file as well as save it in a different directory or on a different drive.

If you use an existing file name to name the file, the IDE asks if you want to overwrite the existing file.

FileSave corresponds to the File|Save command.

FileSaveAll method

See also IDEApplication class

Saves all open editor files.

Types expected

```
bool FileSaveAll()
```

Return value

TRUE if all files were saved, **FALSE** if a file could not be saved

Description

FileSaveAll works just like FileSave except that it saves the contents of all modified files loaded into an Edit window, not just the file in the active Edit window.

FileSaveAll corresponds to the File|Save All command.

FileSaveAs method

See also IDEApplication class

Displays the standard File Save As dialog box so the user can save the currently active editor file.

Types expected

```
bool FileSaveAs([string newName])
```

newName The new name of the file. If supplied, *FileSaveAs* attempts to save the file under that name in the current directory.

Return value

TRUE if the file was saved, **FALSE,** otherwise

Description

FileSaveAs displays the Save File As dialog box, where you can save the file in the active Edit window under a different name, in a different directory, or on a different drive.

You can enter the new file name, including the drive and directory.

All windows containing this file are updated with the new name.

If you choose an existing file name, the Borland C++ IDE asks if you want to overwrite the existing file.

FileSaveAs corresponds to the File|Save As command.

FileSend method

See also IDEApplication class

Instructs the Windows MAPI to send files to another MAPI client.

Types expected

```
bool FileSend()
```

Return value

TRUE if the file was sent, **FALSE,** otherwise

Description

FileSend has an effect only if you have a mail message service (MAPI) installed on your system.

FileSend corresponds to the File|Send command.

GetRegionBottom method

See also IDEApplication class

Gets the bottom value of the specified region.

Types expected

```
int GetRegionBottom(string RegionName)
```

RegionName The name of the region to examine. Valid region names are:

Breakpoint	CPU
Debugger	Editor
Evaluator	Event Log
Inspector	Message
Processes	Project
Stack	Thread Count
Watches	

Return value

The bottom value of the specified region in display units (0 - 10000) or -1 if no such region exists.

Description

GetRegionBottom can be used with SetRegion to position a window.

GetRegionLeft method

See also IDEApplication class

Gets the left value of the specified region.

Types expected

```
int GetRegionLeft(string RegionName)
```

RegionName The name of the region to examine. Valid region names are:

Breakpoint	CPU
Debugger	Editor
Evaluator	Event Log
Inspector	Message
Processes	Project
Stack	Thread Count
Watches	

Return value

The left value of the specified region in display units (0 - 10000) or -1 if no such region exists

Description

GetRegionLeft can be used with SetRegion to position a window.

GetRegionRight method

See also IDEApplication class

Gets the right value of the specified region.

Types expected

```
int GetRegionRight(string RegionName)
```

RegionName The name of the region to examine. Valid region names are:

Breakpoint	CPU
Debugger	Editor
Evaluator	Event Log
Inspector	Message
Processes	Project
Stack	Thread Count
Watches	

Return value

The right value of the specified region in display units (0 - 10000) or -1 if no such region exists

Description

GetRegionRight can be used with SetRegion to position a window.

GetRegionTop method

See also IDEApplication class

Gets the top value of the specified region.

Types expected

```
int GetRegionTop(string RegionName)
```

RegionName The name of the region to examine. Valid region names are:

Breakpoint	CPU
Debugger	Editor
Evaluator	Event Log
Inspector	Message
Processes	Project
Stack	Thread Count
Watches	

Return value

The top value of the specified region in display units (0 - 10000) or -1 if no such region exists

Description

GetRegionTop can be used with SetRegion to position a window.

GetWindowState method

IDEApplication class

Retrieves the state of the currently focused window.

Types expected

```
bool GetWindowState()
```

Return value

One of the following:

SW_NORMAL

SW_MINIMIZE

SW_MAXIMIZE

Help method

IDEApplication class

Invokes the Windows Help system with the specified Help file and context ID.

Types expected

```
void Help (string helpFile, int helpCommand, string helpTopic)
```

helpFile	The name (with optional path) of the Windows Help file to open.
helpCommand	A constant representing a command passed to the Windows Help engine. The *helpCommand* constants begin with HELP_ and are defined in the C++ header file WINUSER.H. See the *Windows API Reference* for details on these constants.
helpTopic	The name of the Help topic to display.

Return value

None

HelpAbout method

See also IDEApplication class

Displays the Help About dialog box.

Types expected

```
bool HelpAbout()
```

Return value

TRUE if the dialog box displays, **FALSE,** otherwise

Description

HelpAbout corresponds to the Help|About command.

HelpContents method

See also IDEApplication class

Displays the default Help contents screen. For Windows 95 Help systems, this window is the Help Topics Contents page.

Types expected

```
bool HelpContents()
```

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpContents corresponds to the Help|Contents command.

HelpKeyboard method

See also IDEApplication class

Displays a Help window describing how to map the keyboard in the IDE.

Types expected

```
bool HelpKeyboard()
```

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpKeyboard corresponds to the Help|Keyboard command.

HelpKeywordSearch method

See also IDEApplication class

Displays the Help Topics Index page with the specified keyword selected.

Types expected

```
bool HelpKeywordSearch([string keyword])
```

keyword The entry selected in the Help Topics Index page.

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpKeyboardSearch corresponds to the Help|Keyboard Search command.

HelpOWLAPI method

See also IDEApplication class

Displays the Help Contents page for the ObjectWindows Library Help.

Types expected

```
bool HelpOWLAPI()
```

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpOWLAPI corresponds to the Help|OWL API command.

HelpUsingHelp method

See also IDEApplication class

Displays a Help window describing how to use Help.

Types expected

```
bool HelpUsingHelp()
```

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpUsingHelp corresponds to the Help|Using Help command.

HelpWindowsAPI method

See also IDEApplication class

Displays the Help Contents page for the Microsoft Windows API Help.

Types expected

```
bool HelpWindowsAPI()
```

Return value

TRUE if the Help window can be displayed, **FALSE,** otherwise

Description

HelpWindowsAPI corresponds to the Help|Windows API command.

KeyPressDialog method

IDEApplication class

Displays a dialog and records the keys pressed.

Types expected

```
string KeyPressDialog(string prompt, string default)
```

prompt The string to display in the caption of the dialog.

default The value to display as a default. If *default* is empty, no value is displayed.

Return value

The key pressed by the user or the empty string ("") if the user presses Esc or Cancel.

Description

KeyPressDialog records the keys pressed in a mnemonic format suitable for using with key assignments.

ListDialog method

IDEApplication class

Dsplays a modal list dialog.

Types expected

```
string[ ] ListDialog(string prompt, bool multiSelect,
                     bool sorted, string [ ] initialValues)
```

prompt	The value to place in the caption of the dialog.
multiSelect	Indicates if multiple selection of items in the list is allowed.
sorted	Indicates how the list is to be sorted.
initialValues	The strings to display in the dialog.

Return value

An array containing the strings that were selected

Menu method

IDEApplication class

Activates the main menu.

Types expected

```
void Menu()
```

Return value

None

Message method

IDEApplication class

Displays messages to the user in a message box.

Types expected

```
bool Message(string text, int severity)
```

text	The message to display.
severity	One of the following values: INFORMATION, WARNING, ERROR. The value specified also determines the text for the caption.

Return value

TRUE if the message box successfully opened, **FALSE,** otherwise.

Description

The message box contains the following buttons: CANCEL, ABORT, RETRY, and OK.

MessageCreate method

See also IDEApplication class

Adds messages to the Message window.

Types expected

```
int MessageCreate(string destinationTab, string toolName,
   int messageType, int parentMessage, string filename,
   int lineNumber, int columnNumber, string text,
   string helpFileName, int helpContextId)
```

destinationTab	The name of the tab on the page of the Message window on which this message should appear. The default supported values for this parameter are Buildtime, Runtime, and Script. If a non-existent tab name is given, a new tab will be created.
toolName	The name of the tool to be associated with the file to open. Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can also use the tool name: AddOn. You can run a DOS program with the Windows IDE transfer. If *toolName* is not provided, a default is used.
messageType	The severity to be associated with the message. The values supported are: INFORMATION - default WARNING ERROR FATAL
parentMessage	The message that this message should be stored under. A value of 0 creates a new top-level message.
fileName	Provides navigation for the message. When the message is selected, the user will be taken to this file.
lineNumber	Provides navigation for the message. When the message is selected, the user will be taken to this line in the specified file.
columnNumber	Provides navigation for the message. When the message is selected, the user will be taken to this column in the specified line of the specified file.
helpFile	Specifies where the user can find Windows Help for the message. When set to a valid value, the specified *helpContext* in this file will display.
helpContext	Specifies where the user can find Windows Help for the message. When set to a valid value, this help topic in the specified help file will display.

Return value

The message ID of the generated message

NextWindow method

IDEApplication class

Advances focus and activation to the next MDI child window from the currently selected window.

Types expected

```
bool NextWindow(bool priorWindow)
```

priorWindow If **TRUE**, focus and activation go to the previous window. *priorWindow* defaults to **FALSE**.

Return value

TRUE if focus changes to another window, **FALSE,** otherwise

OptionsEnvironment method

See also IDEApplication class

Displays the Environment Options dialog box where you set IDE options.

Types expected

```
bool OptionsEnvironment()
```

Return value

TRUE if the dialog box can be displayed, **FALSE,** otherwise

Description

OptionsEnvironment corresponds to the Options|Environment command.

OptionsProject method

See also IDEApplication class

Displays the Project Options dialog box where you set project options.

Types expected

```
bool OptionsProject()
```

Return value

TRUE if the dialog box can be displayed, **FALSE,** otherwise

Description

OptionsProject corresponds to the Options|Project command.

OptionsSave method

See also IDEApplication class

Opens the Options Save dialog box, where you save the contents of the project and the desktop, the messages in the Message window, and the Environment settings.

Types expected

```
bool OptionsSave()
```

Return value

TRUE if the dialog can be opened, **FALSE** if it cannot

Description

OptionsSave corresponds to the Options|Save command.

OptionsStyleSheets method

See also IDEApplication class

Displays the Style Sheets dialog box where you specify default compile and run-time option settings associated with a project.

Types expected

```
bool OptionsStyleSheets()
```

Return value

TRUE if the dialog box can be opened, **FALSE,** otherwise

Description

Style sheets are predefined sets of options that can be associated with a node.

OptionsStyleSheets corresponds to the Options|Style Sheets command.

OptionsTools method

See also IDEApplication class

Displays the Tools dialog box where you install, delete or modify the tools listed on the Tool menu.

Types expected

```
bool OptionsTools()
```

Return value

TRUE if the dialog box can be opened, **FALSE,** otherwise

Description

The Tool menu lets you run programming tools of your choice without leaving the Borland C++ IDE.

OptionsTools corresponds to the Options|Tools command.

ProjectAppExpert method

See also IDEApplication class

Starts the AppExpert.

Types expected

```
bool ProjectAppExpert()
```

Return value

TRUE if AppExpert was successfully started, **FALSE,** otherwise

ProjectBuildAll method

See also IDEApplication class

Builds all the files in the current project, regardless of whether they are out of date.

Types expected

```
bool ProjectBuildAll([bool suppressOkay, string nodeName])
```

suppressOkay	Builds the project without requiring the user to respond with OK to continue.
nodeName	The node to build.

Return value

TRUE if the build was successful, **FALSE,** otherwise

Description

ProjectBuildAll:

1. Deletes the appropriate precompiled header (.CSM) file, if it exists.
2. Deletes any cached autodependency information in the project.
3. Does a rebuild of the node.

If you abort a *ProjectBuildAll* by pressing Esc or choosing Cancel, or if you get errors that stop the build, you must explicitly select the nodes to be rebuilt.

ProjectBuildAll corresponds to the Project|Build All command.

ProjectCloseProject method

See also IDEApplication class

Closes the current project.

Types expected

```
bool ProjectCloseProject()
```

Return value

TRUE if the project was successfully closed, **FALSE,** otherwise

Description

ProjectCloseProject unloads your current project including all project files (nodes) and closes the project tree window, if it is open.

ProjectCloseProject corresponds to the Project|Close Project command.

ProjectCompile method

See also IDEApplication class

Compiles the current project.

Types expected

```
bool ProjectCompile([string nodeName])
```

nodeName The name of the node to compile. Compilation depends on the type of node:

- A .CPP node causes the C++ compiler to be called.
- A .RC node causes resource compiler to be called.
- An .EXE node causes the linker to be called.
- A .LIB node causes the librarian to be called.
- An .SPP node causes the cScript compiler to be called.

Return value

TRUE if the project was successfully closed, **FALSE,** otherwise

Description

ProjectCompile corresponds to the Project|Compile command.

ProjectGenerateMakefile method

See also IDEApplication class

Generates a make file for the current project.

Types expected

```
bool ProjectGenerateMakefile([string nodeName])
```

nodeName If specified, the generated makefile contains only the commands necessary to build that node. Otherwise, commands are generated to build the entire project.

Return value

TRUE if the makefile was successfully generated, **FALSE,** otherwise

Description

ProjectGenerateMakefile generates a makefile for the current project. It gathers information from the currently loaded project and produces a makefile named <projectfilename>.MAK. You cannot convert makefiles to project files.

The IDE displays the new makefile in an Edit window.

ProjectGenerateMakefile corresponds to the Project|Generate Makefile command.

ProjectMakeAll method

See also IDEApplication class

Makes all targets for the current project, rebuilding only those files that are out of date.

Types expected

```
bool ProjectMakeAll([bool suppressOkay, string nodeName])
```

suppressOkay Makes the project without requiring the user to respond with OK to continue.

nodeName Makes only the specified node.

Return value

TRUE if the targets were successfully made, **FALSE,** otherwise

Description

ProjectMakeAll MAKEs all targets. It checks file dates and times to see if they have been updated. If so, *ProjectMakeAll* rebuilds those files, then moves up the project tree and checks the next nodes' file dates and times. *ProjectMakeAll* checks all the nodes in a project and builds all of the out-of-date files.

The .EXE file name is fully spelled out in the project tree for target names. If no project is loaded, the .EXE name is derived from the name of the file in the Edit window.

ProjectMakeAll corresponds to the Project|Make All command.

ProjectManagerInitialize method

IDEApplication class

ProjectManagerInitialize is called once during IDE initialization to ensure that the IDE Project Manager is in a stable state prior to the occurrence of any major events, such as the opening of files or creation of new targets.

Types expected

```
bool ProjectManagerInitialize()
```

Return value

TRUE if the Project Manager has successfully intialized, **FALSE,** otherwise

ProjectNewProject method

IDEApplication class

Creates a new project.

Types expected

```
bool ProjectNewProject([string pName])
```

pName If specified, the project is created with *pName* as its name. Otherwise, the user is prompted for a project name.

Return value

TRUE if the project was successfully created, **FALSE,** otherwise.

Note: Returns FALSE when no parameter is specified.

ProjectNewTarget method

See also IDEApplication class

Creates a new target for the specified node.

Types expected

```
bool ProjectNewTarget ( [string nTarget, int targetType,
     int platform, int libraryMask, int modelOrMode] )
```

nTarget The name of the node.

targetType One of the following target values:

TE_APPLICATION (default)	TE_DLL
TE_DOSCOM	TE_EASYWIN
TE_IMPORTLIB	TE_STATICLIB
TE_WINHELP	

platform One of the following platform values:

TE_WIN32 (default)

TE_DOS16

TE_DOSOVERLAY

TE_WIN16

libraryMask Indicates which libraries to link. One or more of the following values:

TE_STDLIBS (default: same as TE_STDLIB_BIDS | TE_STDLIB_RTL | TE_STDLIB_EMU)	TE_STDLIB_BIDS
TE_STDLIB_BWCC	TE_STDLIB_BGI
TE_STDLIB_C0F	TE_STDLIB_CODEGUARD
TE_STDLIB_CTL3D	TE_STDLIB_EMU
TE_STDLIB_MATH	TE_STDLIB_NOEH
TE_STDLIB_OCF	TE_STDLIB_OLE2
TE_STDLIB_OWL	TE_STDLIB_RTL
TE_STDLIB_TVISION	TE_STDLIB_VBX

modelOrMode One of the following values:

TE_NT_GUI (default if platform is TE_WIN32)	TE_MM_LARGE (default if platform is not TE_WIN32)
TE_MM_TINY	TE_MM_SMALL
TE_MM_MEDIUM	TE_MM_COMPACT
TE_MM_HUGE	TE_NT_WINCONSOLE
TE_NT_FSCONSOLE	

Return value

TRUE if the target was successfully created, **FALSE,** otherwise

Description

The new node is added to the current project and placed at the bottom of the project tree. This is created as a stand alone target. You can move it or make it a child of another node in the project tree by using the Alt+UpArrow/Alt+DownArrow, or Alt+LeftArrow/Alt+RightArrow keys.

ProjectNewTarget corresponds to the Project|New Target command.

ProjectOpenProject method

See also IDEApplication class

Displays the Open a Project dialog box, where you select and load an existing project file.

Types expected

```
bool ProjectOpenProject([string pName])
```

pName If specified, *ProjectOpenProject* opens the project. If not, it displays the Open a Project dialog box and prompts the user for a project name.

Return value

TRUE if the project opened, **FALSE,** otherwise

Description

You can load and use projects from previous versions of Borland C++ (.PRJ files for example). If you load an old Borland C++ project, it is converted to the new project format.

ProjectOpenProject corresponds to the Project|Open Project command.

Quit method

IDEApplication class

Shuts down the IDE and exits, without saving files or prompting the user.

Types expected

```
void Quit()
```

Return value

None

Description

To exit and prompt the user to save changes, use FileExit.

SaveMessages method

See also IDEApplication class

Saves the contents of the specified Message window tab page to the specified file.

Types expected

```
bool SaveMessages(string tabName, string fileName)
```

tabName One of the following values:

Buildtime

Runtime

Script

Return value

TRUE if the messages are saved, **FALSE,** otherwise

ScriptCommands method

See also IDEApplication class

Displays the Script Commands dialog.

Types expected

```
bool ScriptCommands()
```

Return value

TRUE if the users chooses a command and clicks Run, **FALSE,** otherwise

Description

The Script Commands dialog lists the currently available script commands and variables, including classes, functions, and global objects. If an object is an instance of a class, its properties and methods are also displayed.

ScriptCommands corresponds to the Script|Commands command.

ScriptCompileFile method

See also IDEApplication class

Compiles the specified script file.

Types expected

```
bool ScriptCompileFile(string fileName)
```

fileName The name of the script file to compile.

Return value

TRUE if the compile was successful, **FALSE,** otherwise

Description

ScriptCompileFile corresponds to the Script|Compile File command.

ScriptModules method

See also IDEApplication class

Displays the Script Modules dialog box. The dialog box lists the modules loaded (.SPP or .SPX files) and modules in the Script Path.

Types expected

```
bool ScriptModules()
```

Return value

TRUE if a module is selected, **FALSE,** otherwise

Description

ScriptModules corresponds to the Script|Modules command.

ScriptRun method

See also IDEApplication class

Executes the specified script command.

Types expected

```
bool ScriptRun(string command)
```

command The script command to execute. If no *command* is given, the Script Run window is displayed.

Return value

TRUE if the command is executed, **FALSE,** otherwise

Description

ScriptRun corresponds to the Script|Run command.

ScriptRunFile method

See also IDEApplication class

Executes the specified script file.

Types expected

```
bool ScriptRunFile([string fileName])
```

fileName The name of the script file to execute. If no *fileName* is given, *ScriptRunFile* attempts to execute the commands in the current EditView.

Return value

TRUE if a file is executed or an *EditView* is found, **FALSE,** otherwise

Description

ScriptRunFile corresponds to the Script|Run File command.

SearchBrowseSymbol method

See also IDEApplication class

Searches for the specified symbol.

Types expected

```
bool SearchBrowseSymbol([string sName])
```

sName The name of the symbol to search for. If *sName* is not provided, the Browse Symbol dialog box is displayed. If *sName* is not provided and an edit window is active, the Browse Symbol dialog box contains the word at the cursor.

Return value

TRUE if the symbol is found, **FALSE,** otherwise

Description

SearchBrowseSymbol corresponds to the Search|Browse Symbol command.

SearchFind method

See also IDEApplication class

Searches the EditBuffer for the specified pattern.

Types expected

```
bool SearchFind([string pat])
```

pat The string to search for. If *pat* is found, the cursor is moved to the occurance of *pat*. The pattern can be a simple string or a search expression.

Return value

TRUE if the expression is found, **FALSE,** otherwise

Description

If the Edit window is active, *SearchFind* searches the Edit window for *pat*. If the Message window is active, Search Find searches the Message window.

SearchFind corresponds to the Search|Find command.

SearchLocateSymbol method

See also IDEApplication class

Searches through the current target of the current project for the specified symbol.

Types expected

```
bool SearchLocateSymbol([string sName])
```

sName The name of the symbol to search for. If *sName* is not provided, the user will be prompted for it.

Return value

TRUE if the expression is found, **FALSE,** otherwise

Description

SearchLocateSymbol uses the Browser's symbol information to locate a symbol's definition.

On success, *SearchLocateSymbol* opens the source file and line where the symbol name *sName* is defined. If *sName* is **NULL,** *SearchLocateSymbol* rips the current word out of the editor and searches for that symbol. *SearchLocateSymbol* works only with globally defined symbols

For a function symbol, *SearchLocateSymbol* locates the line where the function begins. For a class or typedef symbol, it locates the line where the typedef or class is defined. For a variable, it locates the line where the variable is defined.

SearchLocateSymbol corresponds to the Search|Locate Symbol command.

SearchNextMessage method

See also IDEApplication class

Displays an active Edit window and places the cursor on the line in your source code that generated the next error or warning.

Types expected

```
bool SearchNextMessage()
```

Return value

TRUE if the next message is displayed, or **FALSE** if there is no message to display

Description

SearchNextMessage works only if a Message window is displayed and another message exists.

SearchNextMessage corresponds to the Search|Next Message command.

SearchPreviousMessage method

See also IDEApplication class

Displays an active Edit window and places the cursor on the line in your source code that generated the previous error or warning.

Types expected

```
bool SearchPreviousMessage()
```

Return value

TRUE if the source line is found, **FALSE,** otherwise

Description

SearchPreviousMessage works only if a Message window is displayed and a previous message exists.

SearchPreviousMessage corresponds to the Search|Previous Message command.

SearchReplace method

See also IDEApplication class

Searches the EditBuffer for the specified pattern and replaces it with the specified string.

Types expected

```
bool SearchReplace([string pat, string rep])
```

pat The string to search for. The pattern can be a simple string or a search expression.

rep The string to replace the found string with.

Return value

TRUE if the text is found, **FALSE,** otherwise

Description

If *pat* or *rep* is not specified, *SearchReplace* opens the Replace Text dialog box and prompts the user for input.

SearchReplace corresponds to the Search|Replace command.

SearchSearchAgain method

See also IDEApplication class

Repeats the last SearchFind or SearchReplace.

Types expected

```
bool SearchSearchAgain()
```

Return value

TRUE if the text is found, **FALSE,** otherwise

Description

SearchSearchAgain corresponds to the Search|Search Again command.

SetRegion method

See also IDEApplication class

Determines how windows tile and cascade on the IDE desktop and their initial position when they are created.

Types expected

```
bool SetRegion(string RegionName, int left, int top, int right, int bottom)
```

RegionName The name of the region to examine. Valid region names are:

Breakpoint	CPU
Debugger	Editor
Evaluator	Event Log
Inspector	Message
Processes	Project
Stack	Thread Count
Watches	

left, top, right, bottom The dimensions of the window in display units of 1-9999.

Return value

TRUE if the region was successfully set, **FALSE** otherwise

Description

SetRegion is used with the following *IDEApplication* class methods:

GetRegionBottom

GetRegionTop

GetRegionLeft

GetRegionRight

These methods change the area where windows are placed when tiled and cascaded.

For example, the default configuration of the IDE is to have all Editor windows in the upper two-thirds of the screen when you tile, and the Message window and the Project window in the lower one-third. You could change this default with the script statement

```
IDE.SetRegion("Editor", 1, 1, 5000, 5000);
```

After executing this statement, the editors are in the upper left quarter of the IDE desktop after tiling. Look at STARTUP.SPP for other examples.

SetWindowState method

IDEApplication class

Changes the style of the currently focused window.

Types expected

```
bool SetWindowState(int desiredState)
```

desiredState The style to change the window to. One of the following values:

SW_MINIMIZE

SW_MAXIMIZE

SW_RESTORE

Return value

TRUE if the state was successfully set, **FALSE,** otherwise

SimpleDialog method

IDEApplication class

Invokes a simple dialog containing a single text field, an OK button, and a Cancel button.

Types expected

```
string SimpleDialog(string prompt, string initialValue
       [,int maxNumChars])
```

prompt	The caption of the dialog.
initialValue	The value that initializes the edit field.
maxNumChars	The maximum number of characters allowed in the edit field.

Return value

The value in the edit field if the user clicks OK or presses Enter, or the empty string ("") if the user clicks Cancel.

SpeedMenu method

IDEApplication class

Activates the SpeedMenu for the current subsystem.

Types expected

```
void SpeedMenu()
```

Return value

None

StartWaitCursor method

IDEApplication class

Displays the Windows wait cursor (by default, the hourglass).

Types expected

```
void StartWaitCursor()
```

Return value

None

StatusBarDialog method

IDEApplication class

Displays a dialog on top of the status bar.

Types expected

```
string StatusBarDialog(string prompt, string initialValue
       [, int maxNumChars])
```

prompt	The caption of the dialog.
initialValue	The value that initializes the edit field.
maxNumChars	The maximum number of characters allowed in the edit field.

Return value

The value in the edit field if the user clicks OK or presses Enter, or the empty string ("") if the user clicks Cancel.

StopBackgroundTask method

IDEApplication class

Terminates the background task of a compile, link, make or build when the task is in asynchronous compile mode.

Types expected

```
void StopBackgroundTask()
```

Return value

None

Tool method

IDEApplication class

Runs the specified tool specified using the specified command string.

Types expected

```
bool Tool([string toolName, string commandString])
```

toolName	The name of the tool to be associated with the file to open. Tools can be standalone programs (like GREP, Turbo Debugger, or an alternate editor), or they can be translators that are used for each file (or node) in a project. You can run a DOS program with the Windows IDE transfer. If *toolName* is not provided, a default is used.
commandString	The name of the command to run.

Return value

TRUE if the tool successfully ran, **FALSE,** otherwise

Description

If no parameters are specified, *Tool* displays a dialog box prompting the user for a tool.

Undo method

See also IDEApplication class

Undoes the last edit operation.

Types expected

```
void Undo()
```

Return value

None

Description

Undo does the same thing as EditUndo. *Undo* is included for compliance with Microsoft conventions.

ViewActivate method

IDEApplication class

Activates the IDE pane that is adjacent to the currently selected pane.

Types expected

```
bool ViewActivate(int direction)
```

direction The direction of the adjacent pane to activate, relative to the current pane. The supported values are:

UP

DOWN

LEFT

RIGHT

Return value

TRUE if there was a valid current pane and the method was able to activate an adjacent pane in the direction indicated by direction, **FALSE,** otherwise

ViewBreakpoint method

See also IDEApplication class

Opens the Breakpoints window.

Types expected

```
bool ViewBreakpoint()
```

Return value

TRUE if breakpoints can be found, **FALSE,** otherwise

Description

ViewBreakpoint corresponds to the View|Breakpoint command.

ViewCallStack method

See also IDEApplication class

Opens the Call Stack window.

Types expected

```
bool ViewCallStack()
```

Return value

TRUE if the Call Stack window can be displayed, **FALSE,** otherwise

Description

ViewCallStack corresponds to the View|Call Stack command.

ViewClasses method

See also IDEApplication class

Opens the Browsing Objects window, which displays all the classes in your application.

Types expected

```
bool ViewClasses()
```

Return value

TRUE if the Browsing Objects window can be displayed, **FALSE,** otherwise

Description

ViewClasses corresponds to the View|Classes command.

ViewClassExpert method

See also IDEApplication class

Displays the ClassExpert window, where you can add and manage classes in an AppExpert application.

Types expected

```
bool ViewClassExpert()
```

Return value

TRUE if the Class Expert can be run or **FALSE** if it cannot be run (for example, because the current target was not generated with the AppExpert)

Description

ViewClassEpxert does not work unless the current target is an AppExpert target.

ViewClassEpxert corresponds to the View|Class Expert command.

ViewCpu method

See also IDEApplication class

Opens or selects the CPU window.

Types expected

```
bool ViewCpu()
```

Return value

TRUE if the CPU window can be displayed, **FALSE,** otherwise

Description

ViewCpu corresponds to the View|CPU command.

ViewGlobals method

See also IDEApplication class

Opens the Browsing Globals window, which lists every variable in the program in the current Edit window or the first file in the current project.

Types expected

```
bool ViewGlobals()
```

Return value

TRUE if the Browse Globals window can be displayed, **FALSE,** otherwise

Description

If the program has not been compiled, the IDE must first compile it before invoking the Browser.

ViewGlobals corresponds to the View|Globals command.

ViewMessage method

See also IDEApplication class

Displays the specified page of the Message window.

Types expected

```
bool ViewMessage([string tabName])
```

tabName The name of the Message window page to select. If *tabName* is not found, the currently selected tab remains unchanged. *tabName* can be set to one of the following values:

Buildtime

Runtime

Script

tabName can also be the name of a user-defined tab.

Return value

TRUE if the Message window can be displayed or **FALSE** if it cannot. If *tabName* is not found, the method returns **FALSE** even if the Message window is successfully displayed.

Description

ViewMessage corresponds to the View|Message command.

ViewProcess method

See also IDEApplication class

Opens the Process window.

Types expected

```
bool ViewProcess()
```

Return value

TRUE if the Process window can be displayed, **FALSE,** otherwise

Description

ViewProcess corresponds to the View|Process command.

ViewSlide method

IDEApplication class

Moves the border of the currently selected IDE pane the number of specified characters in the specified direction.

Types expected

```
bool ViewSlide(int direction [, int amount])
```

direction The direction in which to move the border of the currently selected IDE pane. *direction* can be one of:

UP

DOWN

LEFT

RIGHT

amount The number of characters to move the currently selected IDE pane. The size of a character is determined by the number of pixels high and wide a character is in the font used by the pane. If *amount* is not given, the border moves until the user presses the *Enter* or *Esc* keys.

Return value

TRUE if there is a valid current IDE pane, and it was successfully moved, **FALSE,** otherwise

ViewProject method

See also IDEApplication class

Displays the Project window for the currently open project.

Types expected

```
bool ViewProject()
```

Return value

TRUE if the Project window can be displayed, **FALSE,** otherwise

Description

ViewProject corresponds to the View|Project command.

ViewWatch method

See also IDEApplication class

Displays the Watches window for the current program.

Types expected

```
bool ViewWatch()
```

Return value

TRUE if the Watches window can be displayed, **FALSE,** otherwise

Description

ViewWatch corresponds to the View|Watch command.

WindowArrangeIcons method

See also IDEApplication class

Rearranges any minimized window's icons on the desktop. The rearranged icons are evenly spaced, beginning at the lower left corner of the desktop.

Types expected

```
bool WindowArrangeIcons()
```

Return value

TRUE if there are icons to rearrange, **FALSE,** otherwise

Description

WindowArrange corresponds to the Window|Arrange Icons command.

WindowCascade method

See also IDEApplication class

Stacks all open windows and overlaps them, making all windows the same size and showing only part of each underlying window.

Types expected

```
bool WindowCascade()
```

Return value

TRUE if there are windows to cascade, **FALSE,** otherwise

Description

WindowCascade corresponds to the Window|Cascade command.

WindowCloseAll method

See also IDEApplication class

Closes all windows of the specified type.

Types expected

```
bool WindowCloseAll([string typeName])
```

typeName The type of window to close. *typeName* can be one of the following values:

Browser

Debugger

Editor

If *typeName* is not specified, *WindowCloseAll* closes all open windows.

Return value

TRUE if all windows successfully close, **FALSE,** otherwise

Description

WindowCloseAll corresponds to the Window|Close All command.

WindowMinimizeAll method

See also IDEApplication class

Minimizes all windows of the specified type.

Types expected

```
bool WindowMinimizeAll([string typeName])
```

typeName The type of window to minimize. *typeName* can be one of the following values:

Browser

Debugger

Editor

If *typeName* is not specified, *WindowMinimizeAll* minimizes all open window.

Types expected

```
bool WindowMinimizeAll([string typeName])
```

Return value

TRUE if all windows successfully minimize, **FALSE,** otherwise

Description

WindowMinimizeAll corresponds to the Window|Minimize All command.

WindowRestoreAll method

See also IDEApplication class

Restores all minimized windows of the specified type.

Types expected

```
bool WindowRestoreAll([string typeName])
```

typeName The type of minimized window to restore. *typeName* can be one of the following values:

Browser

Debugger

Editor

If *typeName* is not specified, *WindowRestoreAll* restores all minimized window.

Return value

TRUE if all windows successfully restore or **FALSE** if at least one does not

Description

WindowRestoreAll corresponds to the Window|Restore All command.

WindowTileHorizontal method

See also IDEApplication class

Stacks all open windows horizontally.

Types expected

```
bool WindowTileHorizontal()
```

Return value

TRUE if all windows successfully tile, **FALSE,** otherwise

Description

WindowTileHorizontal corresponds to the Window|Tile Horizontal command.

WindowTileVertical method

See also IDEApplication class

Stacks all open windows vertically.

Types expected

```
bool WindowTileVertical()
```

Return value

TRUE if all windows successfully tile, **FALSE,** otherwise

Description

WindowTileVertical corresponds to the Window|Tile Vertical command.

YesNoDialog method

IDEApplication class

Displays a dialog box that prompts the user for a yes or no response.

Types expected

```
string YesNoDialog(string prompt, string default)
```

prompt The prompt that displays in the dialog box

default The button that is to be selected by default. Valid values are Yes and No.

Return value

Yes or No

BuildComplete event

IDEApplication class

Raised at the end of a build.

Types expected

```
void BuildComplete(bool status, string inputPath, string outputPath)
```

status	Indicates if the build was successful. *status* is **TRUE** if successful, **FALSE** if there were errors.
inputPath	The source directory.
outputPath	The directory where files created as a result of the build are created.

Return value

None

BuildStarted event

IDEApplication class

Raised at the beginning of a build.

Types expected

```
void BuildComplete()
```

Return value

None

DialogCreated event

IDEApplication class

Raised as new dialogs are presented to the user.

Types expected

```
void DialogCreated(string dialogName, int dialogHandle)
```

dialogName The name of the dialog's caption.

dialogHandle An environment-specific identifier used by the system when referring to the dialog. For Microsoft Windows the *dialogHandle* is the HWND of the dialog. This value is supplied in case you need your script to interact directly with the system.

Return value

None

Description

Use *DialogCreated* in conjunction with the KeyboardManager.SendKeys method to simulate user entries to dialogs and drive the dialog.

DialogCreated is only raised if the property RaiseDialogCreatedEvent is set to **TRUE**.

Exiting event

IDEApplication class

Raised as the IDE is closing. Default action is to do nothing.

Types expected

```
void Exiting()
```

Return value

None

HelpRequested event

IDEApplication class

Raised when one of the *IDEApplication* class Help methods is invoked:

Types expected

```
void HelpRequested(string fileName, int command, int data)
```

fileName	The name (with optional path) of the Windows Help file to open.
Command	A constant representing a command passed to the Windows Help engine. The c*ommand* constants begin with HELP_ and are defined in the C++ header file WINUSER.H. See the *Windows API Reference* for details on these constants.
data	The data to display.

Return value

None

Description

HelpRequested is raised when one of the following *IDEApplication* class methods are inoked:

EnterContextHelpMode

Help

HelpAbout

HelpContents

HelpKeyboard

HelpKeywordSearch

HelpOWLAPI

HelpUsingHelp

HelpWindowsAPI

HelpRequested passes the appropriate parameters to the Windows Help engine. Default action is to do nothing.

Idle event

IDEApplication class

Raised when the number of seconds specified by IdleTimeout has elapsed without a significant event occurring (like a user event). Default action is to do nothing.

Types expected

```
void Idle()
```

Return value

None

KeyboardAssignmentsChanging event

See also IDEApplication class

Raised when the user exits the Options|Environment|Editor dialog after having modified the keyboard file (.KBD) option.

Types expected

```
void KeyboardAssignmentsChanging(string newFileName)
```

newFileName The name of the new keyboard (.KBD) file.

Return value

None

KeyboardAssignmentsChanged event

See also IDEApplication class

Raised after the keyboard file name (.KBD) is changed in the Options|Environment|Editor dialog.

Types expected

```
void KeyboardAssignmentsChanged(string newFileName)
```

newFileName The name of the new keyboard (.KDB) file.

Return value

None

MakeComplete event

IDEApplication class

Raised at the end of a make.

Types expected

```
void MakeComplete(bool status, string inputPath, string outputPath)
```

status	Indicates if the make was successful. *status* is **TRUE** if the make was successful, **FALSE** if there were errors.
inputPath	The source directory.
outputPath	The directory where files created as a result of the make are created.

Return value

None

MakeStarted event

IDEApplication class

Raised at the beginning of a make.

Types expected

```
void MakeComplete()
```

Return value

None

ProjectClosed event

IDEApplication class

Raised when a project file has been successfully closed.

Types expected

```
void ProjectClosed(string projectFileName)
```

projectFileName The absolute name of the project file.

Return value

None

Description

Since the IDE always has a project open (even if it is the default project: BCWDEF.IDE), *ProjectClosed* will always precede the ProjectOpened that it corresponds to.

ProjectOpened event

IDEApplication class

Raised when a project file has been successfully opened.

Types expected

```
void ProjectOpened(string projectFileName)
```

projectFileName The absolute name of the project file.

Return value

None

SecondElapsed event

IDEApplication class

Raised once every second. Default action is to do nothing.

Types expected

```
void SecondElapsed()
```

Return value

None

Started event

IDEApplication class

Raised after the IDE has been loaded and initialized and all startup scripts have been processed.

Types expected

```
void Started(bool VeryFirstTime)
```

VeryFirstTime	Indicates whether this is the first time the IDE has been loaded on a particular machine. Its value is determined by the presence or absence of the default configuration file (BCCONFIG.BCW). This file is created the first time you run the IDE and should be present only if the IDE has been run previously.

Return value

None

SubsytemActivated event

IDEApplication class

Raised when the active subsystem is changed (usually in response to the user clicking on another window type).

Types expected

```
void SubsytemActivated(string systemName)
```

systemName The name of the subsystem acquiring focus. Default action is to do nothing.

Return value

None

TransferOutputExists event

IDEApplication class

Raised when a transfer tool has created output that needs processing (usually in a Make sequence). Default action is to do nothing.

Types expected

```
bool TransferOutputExists(TransferOutput output)
```

output The data that needs to be processed by the transfer tool.

Return value

FALSE if no error occurred, **TRUE** if there was an error parsing the data supplied by output.

TranslateComplete event

IDEApplication class

Raised at the end of a translation.

Types expected

```
void TranslateComplete(bool status, string inputPath,
     string outputPath)
```

status	Indicates if the translation was successful. *status* is **TRUE** if the translation was successful, **FALSE** if there were errors.
inputPath	The source directory.
outputPath	The directory where files created as a result of the translation are created.

Return value

None

Keyboard class

Description

This class works with the KeyboardManager class to manage keyboards assigned to various IDE components, such as the Editor and the Project View.

Syntax

```
Keyboard([bool transparent])
```

transparent Allows keystrokes with no assignment in this keyboard to be passed to the next keyboard on the current keyboard stack. This value defaults to **FALSE** if not supplied.

Properties	Access
`int Assignments`	Read-only
`string DefaultAssignment`	Read-write

Methods

```
void Assign(string KeySequence, string CommandName, int ImplicitAssignments)
void AssignTypeables(string CommandName)
void Copy(Keyboard SourceKeyboard)
int CountAssignments(string CommandName)
string GetCommand(string KeySequence )
string GetKeySequence(string CommandName [,int whichOne])
bool HasUniqueMapping(string KeySequence)
void Unassign(string KeySequence)
```

Events

None

Keyboard class description

Keyboard class

Keyboard objects administer key assignments and can be:

- Assigned to IDE components
- Pushed and popped from the keyboard manager's keyboard stack
- Queried on individual key assignments

KeyboardManager manipulates *Keyboard* objects.

Assignments property

Keyboard class

Indicates the number of key assignments contained in this keyboard.

Access

Read-only

Type expected

```
int Assignments
```

DefaultAssignment property

Keyboard class

Establishes the command to execute if no other commands are assigned to a keystroke. It returns an empty string ("") if no assignment exists.

Access

Read-write

Type expected

```
string DefaultAssignment
```

Assign method

Keyboard class Example

Assigns a script to a keystroke.

Types expected

```
void Assign (string KeySequence, string CommandName, int
             ImplicitAssignments)
```

KeySequence A mnemonic key name made up of a key description, such as `<a>`. Key descriptions can be augmented with any (or all) of the following: Shift, Ctrl, Alt, and Keypad.

CommandName The script to be executed when the key is pressed, for example, `editor.MarkWord(TRUE);`

implicitAssignments One or more of the following values:

Value	Definition
ASSIGN_EXPLICIT (default)	No implicit assignments should be created.
ASSIGN_IMPLICIT_KEYPAD	When an assignment is made to a sequence that has a numeric keypad (Keypad) equivalent, such as Page Up, a second assignment is implicitly made for the equivalent. Assignments are made to both the shifted and non-shifted versions at the same time, but only if the implicit assignment doesn't overwrite an existing explicit assignment.
ASSIGN_IMPLICIT_SHIFT	<a> == <A>
ASSIGN_IMPLICIT_MODIFIER	<Ctrl-k><Ctlr-b> == <Ctrl-k><b>

Return value

None

Description

Keys that do not map to a single character have names associated with them. Keys in this category are: Enter, Backspace, Tab, Home, End, Page Up, Page Down, Left, Right, Up, Down, Insert, Delete, Escape, Space, Print Screen, Center, Pause, CapsLock, Scroll Lock, and Num Lock.

Modifiers and names are separated by a dash (`-`). For example `<Ctrl-Enter>` is valid.

To assign the dash character in a key sequence, use the keyname `<Minus>`. Use the keyname `<Plus>` for the `+` character.

The *Assign* method has no effect on the default keyboard, which is returned from a call to KeyboardManager.GetKeyboard.

Example

```
// This example creates an explicit assignment to <Home>.
// It creates an implicit assignment to <Keypad-Home>.

Assign("<Home>","ToStart();",ASSIGN_IMPLICIT_KEYPAD);

// Explicit assignment to <Keypad-End>.
Assign("<Keypad-End>","ToEnd();");

// Explicit assignment to <End>
Assign("<End>","ToEnd(TRUE);",ASSIGN_IMPLICIT_KEYPAD);

// Implicit assignment to <Keypad-End> thwarted due to
// existence of explicit assignment to <Keypad-End>.
```

AssignTypeables method

Keyboard class

Assigns a script to the predefined typeable characters.

Types expected

```
void AssignTypeables(string CommandName)
```

CommandName	The command to assign and any parameters to the command.

Return value

None

Description

The *AssignTypeables* method has no effect on the default keyboard, which is returned from a call to KeyboardManager.GetKeyboard.

Predefined typeable characters

~	!	@	#	$	%	^	&	'*	'
(	')	'_	+	`	1	2	3	4	5
6	7	8	9	0	-	=	Q	W	E
R	T	Y	U	I	O	P	{	}	|
q	w	e	r	t	y	u	i	o	p
[	]	\\	A	S	D	F	G	H	J
K	L	:	"	'	a	s	d	f	g
h	j	k	l	;	\	Z	X	C	V
B	N	M	<	>	?	z	x	c	v
b	n	m	,	.	/				

Other keys include: Enter, Delete and Backspace.

Copy method

Keyboard class

Copies all assignments made from *SourceKeyboard* into this keyboard, replacing any that already exist.

Types expected

```
void Copy(Keyboard SourceKeyboard)
```

Keyboard	The name of the keyboard to copy assignments into.
SourceKeyboard	The name of the keyboard from which assignments are to be copied.

Return value

None

Description

The *Copy* method has no effect on the default keyboard, which is returned from a call to KeyboardManager.GetKeyboard.

CountAssignments method

Keyboard class

Returns the number of key assignments tied to the specified command.

Types expected

```
int CountAssignments(string CommandName)
```

CommandName The name of the command in which to count key assignments.

Return value

None

GetCommand method

Keyboard class

Returns the command assigned to the specified key code. *GetCommand* returns the empty string ("") if no script has been assigned.

Types expected

```
string GetCommand (string KeySequence )
```

KeySequence The name of the key sequence to check for an assigned command.

Return value

None

GetKeySequence method

Keyboard class

Returns the key sequence tied to the specified command.

Types expected

```
string GetKeySequence(string CommandName [,int whichOne])
```

CommandName	The name of the command to check for a key sequence.
whichOne	Finds nth occurrence of that assignment. If less than 1 or omitted, *whichOne* is assumed to be 1.

Return value

None

HasUniqueMapping method

Keyboard class

Determines if a key:

- Has no mapping
- Maps directly to a command
- Is the non-terminating key of a multikey assignment

Types expected

```
bool HasUniqueMapping(string KeySequence)
```

KeySequence The name of the key sequence to check for mapping assignments.

Return value

TRUE if a key either has no mapping or maps directly to a command. **FALSE** if the key is a non-terminating key of a multikey assignment.

For example, WordStar `<Ctrl-K>` would be **FALSE** since `<Ctrl-K>` signifies the beginning of a multikey assignment, such as `<Ctrl-K><Ctrl-B>` or `<Ctrl-K><Ctrl-K>`.

Unassign method

Keyboard class

Restores a key assignment.

Types expected

```
void Unassign(string KeySequence)
```

KeySequence The name of the key sequence to restore.

Return value

None

Description

The *Unassign* method has no effect on the default keyboard, which is returned from a call to KeyboardManager.GetKeyboard.

KeyboardManager class

Description

This class works with the Keyboard class to manage keyboards assigned to various IDE components, such as the Editor and the Project view.

Syntax

```
KeyBoardManager()
```

Properties	Access
`bool AreKeysWaiting`	Read-only
`Record CurrentPlayback`	Read-only
`Record CurrentRecord`	Read-write
`int KeyboardFlags`	Read-only
`int KeysProcessed`	Read-only
`int LastKeyProcessed`	Read-only
`Record Recording`	Read-only
`string ScriptAbortKey`	Read-write

Methods

```
string CodeToKey( int KeyCode )
void Flush()
Keyboard GetKeyboard( [string ComponentName] )
int KeyToCode( string KeyName )
void PausePlayback()
int Playback( [Record RecordObject] )
Keyboard Pop( string ComponentName )
bool ProcessKeyboardAssignments( string fileName, bool unassign )
void ProcessPendingKeystrokes()
void Push( Keyboard keyboard, string ComponentName, bool transparent )
int ReadChar( void )
void ResumePlayback()
bool ResumeRecord( Record RecordObject )
bool SendKeys(string keyStream)
bool StartRecord( Record RecordObject )
void StopRecord( )
```

Events

None

KeyboardManager class description

KeyboardManager class

You access keyboard features through a keyboard manager, implemented by the global *KeyboardManager* object. The keyboard manager manipulates Keyboard objects (instantiations of the class *Keyboard*).

KeyboardManager manages individual component keyboards, such as that of the Editor and the Project view. This implementation allows support of BRIEF functionality through script simulation without predefined classes for each of the individual IDE components. Each component has a defineable keyboard. The desktop has a keyboard assignment that acts as a global assignment. If a key is not found in the local keyboard, the desktop keyboard is searched. If the key assignment is not in the desktop's keyboard, the default internal mapping is used.

The keyboard manager operates on the assumption of a set context. A derived class is used in a call to *SetContext*() to specify the current object to be used as a local scope. Since different macros may mean different things to different components, this mechanism provides a simple, straightforward approach to localizing functionality. For example, classes *A* and *B* both have a member function called *Search*(). If class *A* is the current context, class *A*'s *Search*() member is called. The same goes with class *B*. If no context is set, then a global *Search*() function is accessed.

The *IDE* object contains a *ReadOnly* member that holds the value of the *KeyboardManager*. New script instances may be created; however, they will all reference the same internal data and changes to one will be reflected in all.

AreKeysWaiting property

KeyboardManager class

TRUE if any keys are waiting to be processed, **FALSE** otherwise.

Access

Read-only

Type expected

```
bool AreKeysWaiting
```

CurrentPlayback property

KeyboardManager class

Plays back the current keystroke assignment. *CurrentPlayback* is only valid while the *Playback* method is active.

Access

Read-only

Type expected

```
Record CurrentPlayback
```

CurrentRecord property

KeyboardManager class

Contains a reference to the Record object associated with this *KeyboardManager*.

Access

Read-write

Type expected

```
Record CurrentRecord
```

KeyboardFlags property

KeyboardManager class

Returns a value whose bits indicate the state of Num Lock, Caps, Ctrl, Alt, and so on.

Access

Read-only

Type expected

```
int KeyboardFlags
```

Description

The mask values returned are:

0x03 - Shift pressed

0x04 - Ctrl pressed

0x08 - Alt pressed

0x10 - Scroll Lock on

0x20 - Num Lock on

0x40 - Caps Lock on

KeysProcessed property

KeyboardManager class

The total number of keys processed by any keyboard since the IDE was loaded.

Access

Read-only

Type expected

```
int KeysProcessed
```

LastKeyProcessed property

KeyboardManager class

The keycode of the last key that was processed by any keyboard.

Access

Read-only

Type expected

```
int LastKeyProcessed
```

Recording property

See also KeyboardManager class

TRUE if a keys are currently being recorded, **FALSE** otherwise.

Access

Read-only

Type expected

```
Record Recording
```

Description

Only valid while in a StartRecord. Becomes invalid when StopRecord is called.

Note: The return value matches Brief's *inq_kbd_flags*().

ScriptAbortKey property

See also KeyboardManager class

Contains the key sequence of the key which, when pressed, causes the currently running script to abort.

Access

Read-write

Type expected

```
string ScriptAbortKey
```

Description

The default value for *ScriptAbortKey* is `<Escape>`, except when Epsilon emulation is enabled in which case the default is `<Ctrl-G>`.

Key sequence

The key sequence is a mnemonic key name made up of a key description, such as `<a>`. Key descriptions can be augmented with any (or all) of the following: Shift, Ctrl, Alt, and Keypad.

To assign the dash character in a key sequence, use the keyname `<Minus>`. Use the keyname `<Plus>` for the `+` character.

Key mapping

Keys that do not map to a single character have names associated with them. Keys in this category are: Enter, Backspace, Tab, Home, End, Page Up, Page Down, Left, Right, Up, Down, Insert, Delete, Escape, Space, Print Screen, Center, Pause, Caps Lock, Scroll Lock, and Num Lock.

Modifiers and names are separated by a dash (`-`). For example:

```
<Ctrl-Enter>
```

CodeToKey method

KeyboardManager class

Accepts the integer key code representation.

Types expected

```
string CodeToKey(int KeyCode)
```

KeyCode An integer representation of a keystroke.

Return value

The textual description of the key. It matches the Brief key naming conventions for *inq_assignment* and *assign_to_key*.

Flush method

KeyboardManager class

Removes all waiting keystrokes from the IDE message queue.

Types expected

```
void Flush()
```

Return value

None

GetKeyboard method

KeyboardManager

This method finds the keyboard currently assigned to the IDE subsystem.

Types expected

```
Keyboard GetKeyboard ( [string ComponentName] )
```

ComponentName The name of the IDE subsystem. To return the internal mapping, specify Default. Note that the default mapping cannot be remapped. If *ComponentName* is omitted, the method gets the current keyboard. Valid subsystems are:

Browser	Editor
ClassManager	Message
Default	Project
Desktop	

Return value

The keyboard currently assigned to an IDE subsystem, or **NULL** if the subsystem is invalid

KeyToCode method

KeyboardManager class

Converts the name of a key into its integer key code equivalent.

Types expected

```
int KeyToCode (string KeyName)
```

keyName The textual name of a key.

Return value

The integer keycode of the key

Description

KeyToCode accepts single keystroke entries such as `<F>` and `<Ctrl-B>`, but not multikey sequences such as `Ctrl+K Ctrl+B`.

PausePlayback method

See also KeyboardManager class

Pauses the playback of a Record object.

Types expected

```
void PausePlayback()
```

Return value

None

Description

For *PausePlayback* to work, the play back must have been initiated with the Playback member. To resume playback, call ResumePlayback.

Playback method

See also KeyboardManager class

Replays the series of keystrokes assigned to a Record object. If no *Record* object is specified, the last recording is replayed.

Types expected

```
int Playback ( [Record RecordObject] )
```

RecordObject The name of the *Record* object from which keys are to be replayed.

Return value

One of the following values:

0	No sequence to play back
1	Playback successful
-1	Sequence is paused or being remembered
-2	Error loading disk file (macros will handle this)
-3	Canceled by user with ScriptAbortKey

Pop method

KeyboardManager class

Restores the previously assigned keyboard mapping after a call to Push.

Types expected

```
Keyboard Pop(string ComponentName)
```

ComponentName	The name of the IDE subsystem whose keyboard you want to restore. Valid IDE subsystem names are:

Browser	Editor
ClassManager	Message
Default	Project
Desktop	

Return value

The keyboard that was restored or **NULL,** which indicates that no additional keyboard mappings were applied and the default keyboard desktop mapping is active

ProcessKeyboardAssignments method

KeyboardManager class

Converts a .KBD file into a .KBP file.

Types expected

```
bool ProcessKeyboardAssignments (string fileName, bool unassign)
```

fileName The name of the .KBD formatted file. Includes the path to the file.

unassign Specifies if the file contents should be used to unassign keys defined in the .KBD file. If TRUE, defined keys will be unassigned. If FALSE, defined keys will be assigned.

Return value

TRUE if a .KBP file is loaded, **FALSE** otherwise.

Description

ProcessKeyboardAssignments converts a .KBD file into a .KBP file, which is a preprocessed version of the .KBD file. If the .KBP file exists and is newer than the .KBD file, the .KBP file will be used without creating another .KBP file.

ProcessPendingKeystrokes method

KeyboardManager class

Fine-tunes the behavior of SendKeys.

Types expected

```
void ProcessPendingKeystrokes()
```

Return value

None

Description

If one or more calls to *SendKeys* indicated that key processing was to be delayed, these keystrokes are not processed until *ProcessPendingKeystrokes* is called or until the script completes execution.

Push method

KeyboardManager class

Pushes a keyboard on the keyboard stack, making the new keyboard mapping current. A subsequent *Pop* operation restores the previously assigned keyboard mapping.

Types expected

```
void Push ( Keyboard keyboard, string ComponentName, bool transparent )
```

keyboard	The name of the keybard to push onto the stack.
ComponentName	The name of the IDE subsystem whose keyboard is to be pushed onto the stack. Valid IDE subsystem names are: Browser Editor ClassManager Message Default Project Desktop
transparent	Determines the run-time behavior of keystrokes not found in the keyboard. If *transparent* is set, the next keyboard on the stack is searched. Otherwise, the key is ignored.

Return value

None

ReadChar method

KeyboardManager class

Reads the key that was pressed.

Types expected

```
int ReadChar ( void )
```

Return value

This method returns either -1 (no key is waiting) or the scan value for the key that was pressed. The high-order byte is the scan code, and the low-order byte is the ASCII value.

Description

ReadChar manages two queues, a local queue for Push and the queue for the standard Windows messaging system. It first checks the local queue for any waiting keys. If no keys are available in the local queue, it checks the Windows message queue.

ResumePlayback method

See also KeyboardManager class

Resumes the playback of a Record object.

Types expected

```
void ResumePlayback()
```

Return value

None

Description

For *ResumePlayback* to work, the playback must be initiated with the Playback member after suspending the recording with a call to PausePlayback.

ResumeRecord method

See also KeyboardManager class

Initiates record mode on a Record object.

Types expected

```
bool ResumeRecord ( Record RecordObject )
```

RecordObject The name of the *Record* object to continue recording.

Return value

TRUE if is able to resume recording, **FALSE** otherwise

Description

New keystrokes are appended to the end of the record buffer. The Recording member is updated.

SendKeys method

KeyboardManager class Example

Simulates the pressing of the keys indicated in the *keyStream* parameter.

Types expected

```
bool SendKeys(string keyStream[, bool suppressImmediateProcessing])
```

keystream	A series of key presses. The limit on the number of characters in Windows 95 is 715. There is no limit in Windows NT.
suppressImmediateProcessing	The default behavior is to process the keys immediately, before the next line of script is processed. If you include this parameter and set it to **TRUE**, *SendKeys* delays processing of the keys until ProcessPendingKeystrokes is called or until the script completes execution.

Return value

TRUE if *keyStream* has valid syntax and can be interpreted or **FALSE** if *keyStream* could not be turned into a series of key presses

Description

SendKeys takes a key or series of keys as its parameter.

Simple displayable keys are just a string of characters that are the same as the keycaps. For example, the following is valid:

```
SendKeys("hello world");
```

There are two separate keyboard parsers: one for processing key assignments and the other for processing *SendKeys.* These processors accept different formats for the same keys. For example, `<Alt-a>` is the same as `%a`.

It is possible, though not probable, to accidentally send a key sequence to another application besides Borland C++ with *SendKeys.* This can occur if *SendKeys* is executed while BCW.EXE is not active. For example, if *SendKeys* is called by a timer event or while a user is in the process of task switching, the key sequence could be sent to another application.

Alt, Shift and Ctrl keys

Keys that do not have simple displayable counterparts, like `Alt+S`, have a special syntax.

The following table shows how to indicate `Alt+keyname`, `Shift+keyname` and `Ctrl+keyname`:

Key	Description	Example
Alt key modifier	Preface the key name with the percent character (%).	Alt+s is `%s`.
Shift key modifier	Either preface the key name with the plus character (+) or capitalize it.	Shift+s is either +s or S
Ctrl key modifier	Preface the key name with the carat character (^).	Ctrl+s is `^s`.

Note: The *SendKeys* parameter is case sensitive. `^s` is `Ctrl+S`, but `^S` is `Ctrl+Shift+S`.

%, + and ^ keys

To indicate the `%`, `+` and `^` key, precede the key name with a backslash (\) as below. To indicate:

- `%`, use `+\\%`

- `^`, use `+\\^`
- `+`, use `+\\+`

Non-displaying keys

To simulate non-displaying keys, use a key mnemonic and enclose it in braces ({ }).

For example, to simulate the key sequence `Alt+s 1 + 2 [Enter]`, use the following syntax:

```
SendKeys("%s1\+2{VK_RETURN}");
```

SendKeys example

```
//Example of SendKeys

x = new KeyboardManager();

/* Sends Ctrl+S and processes it immediately
x.SendKeys("^S");

/* Sends Ctrl+S and processes it immediately
z.SendKeys("^S", FALSE);

/* Sends Ctrl+S and delays processing
x.SendKeys("...", TRUE);

/* Processes the delayed keystrokes
x.ProcessPendingKeystrokes();
```

Table of key mnemonics

VK_ADD	VK_F12	VK_NUMPAD2
VK_BACK	VK_F13	VK_NUMPAD3
VK_CAPITAL	VK_F14	VK_NUMPAD4
VK_CANCEL	VK_F15	VK_NUMPAD5
VK_CLEAR	VK_F16	VK_NUMPAD6
VK_CONTROL	VK_F17	VK_NUMPAD7
VK_DECIMAL	VK_F18	VK_NUMPAD8
VK_DELETE	VK_F19	VK_NUMPAD9
VK_DIVIDE	VK_F20	VK_PAUSE
VK_DOWN	VK_F21	VK_PRINT
VK_END	VK_F22	VK_PRIOR
VK_ESCAPE	VK_F23	VK_RBUTTON
VK_EXECUTE	VK_F24	VK_RETURN
VK_F1	VK_HELP	VK_RIGHT
VK_F2	VK_HOME	VK_SCROLL
VK_F3	VK_INSERT	VK_SELECT
VK_F4	VK_LBUTTON	VK_SEPARATOR
VK_F5	VK_LEFT	VK_SHIFT
VK_F6	VK_MBUTTON	VK_SNAPSHOT
VK_F7	VK_MENU	VK_SPACE
VK_F8	VK_MULTIPLY	VK_SUBTRACT
VK_F9	VK_NUMLOCK	VK_TAB
VK_F10	VK_NUMPAD0	VK_NEXT
VK_F11	VK_NUMPAD1	VK_UP

StartRecord method

See also KeyboardManager class

Begins storing keystroke sequences in a Record object. Updates the Recording member.

Types expected

```
bool StartRecord ( Record RecordObject )
```

RecordObject The name of the *Record* object in which keys are to be recorded.

Return value

TRUE if the key sequence is stored, **FALSE** otherwise

Description

StartRecord replaces any key sequences already stored in the *Record* object.

You can record to only one *Record* object at a time. If you attempt a *StartRecord* before calling a matching StopRecord for a previous recording, the *StartRecord* fails.

StopRecord method

See also KeyboardManager class

Halts recording keystrokes previously started with StartRecord.

Types expected

```
void StopRecord ( )
```

Return value

None

Description

StopRecord updates the CurrentRecord member and updates the Recording member to **FALSE.**

ListWindow class

Description

The *ListWindow* class implements and manages list windows.

Syntax

```
ListWindow(int Top, int Left, int Height, int Width,
           string Caption, bool MultipleSelect, bool Sorted,
           string[] InitialValues)
```

Top, *Left*, *Height*, *Width*	Initial coordinates of the list.
Caption	Text to be displayed in the list title.
MultipleSelect	Determines whether the list will support multiple selections.
Sorted	Determines whether new additions to the list are put in their sorted order.
InitialValues	An array of strings specifying the initial contents of the list.

Properties	Access
`string Caption`	Read-write
`int Count`	Read-only
`int CurrentIndex`	Read-only
`[]Data`	Read-only
`int Height`	Read-write
`bool Hidden`	Read-write
`bool MultiSelect`	Read-only
`bool Sorted`	Read-only
`int Width`	Read-write

Methods

```
void Add(string newEl, int offset)
void Clear()
void Close()
void Execute()
int FindString(string toFind)
string GetString(int offset)
void Insert()
bool Remove(int offset)
```

Events

```
void Accept()
void Cancel()
void Closed()
void Delete()
bool KeyPressed(string keyName)
void LeftClick(int xPos, int yPos)
void Move()
void RightClick(int xPos, int yPos)
```

ListWindow class description

ListWindow class

ListWindow objects create a list window. A list window is a list view that displays a list of selectable items. *ListWindow* objects control:

- The size and position of the list
- The contents of the list
- The number of items in the list
- Finding and getting strings in the list
- Opening and closing the list

Caption property

ListWindow class

The title of the list window.

Access

Read-write

Type expected

```
string Caption
```

Count property

ListWindow class

The number of elements in the list.

Access

Read-only

Type expected

```
int Count
```

CurrentIndex property

ListWindow class

Contains the zero-based index of the currently highlighted list element, or -1 if nothing is selected.

Access

Read-only

Type expected

```
int CurrentIndex
```

Data property

ListWindow class

Contains an array of strings that represent the contents of the list.

Access

Read-only

Type expected

```
[]Data
```

Height property

ListWindow class

Contains the height of the list window in pixels.

Access

Read-write

Type expected

```
int Height
```

Hidden property

ListWindow class

Determines whether the list window can be removed from the display.

Access

Read-write

Type expected

```
bool Hidden
```

Description

Hidden only has meaning after the Execute method has been called and before the list window is closed.

MultiSelect property

ListWindow class

If **TRUE**, allows multiple selections from the list. If **FALSE**, only a single selection can be made.

Access

Read-only

Type expected

```
bool MultiSelect
```

Sorted property

ListWindow class

If **TRUE**, the elements in the list are sorted as new elements are added. If **FALSE**, elements appear at the offset given in the call to the Add method.

Access

Read-only

Type expected

```
bool Sorted
```

Width property

ListWindow class

Contains the width of the list window in pixels.

Access

Read-write

Type expected

```
int Width
```

Add method

ListWindow class

Adds the string *newEl* to the list at the position designated by *offset*.

Types expected

```
void Add(string newEl, int offset)
```

newEl	The string to add to the list.
offset	The position to add the string to. *offset* is zero-based. *offset* should not be higher than Count, or else the new element will not appear in the list. *offset* is ignored if the list is sorted.

Return value

None

Description

Add only has an effect after the ListWindow has been opened using the Execute method.

Clear method

ListWindow class

Removes all elements from the list.

Types expected

```
void Clear()
```

Return value

None

Close method

ListWindow class

Removes the *ListWindow* from the screen.

Types expected

```
void Close()
```

Return value

None

Execute method

ListWindow class

Creates and displays the *ListWindow.*

Types expected

```
void Execute()
```

Return value

None

FindString method

ListWindow class

Finds the specified string.

Types expected

```
int FindString(string toFind)
```

stringToFind The string to find.

Return value

The one-based offset of the string or zero if not found.

Description

FindString only has an effect after the *ListWindow* has been opened using the Execute method.

GetString method

ListWindow class

Returns a string.

Types expected

```
string GetString(int offset)
```

offset The location of the string to get.

Return value

The string at the specified offset or "" if the offset is illegal.

Insert method

ListWindow class

Invoked when the user presses Insert. The default action is to do nothing.

Types expected

```
void Insert()
```

Return value

None

Remove method

ListWindow class

Removes the element from the specified offset.

Types expected

```
bool Remove(int offset)
```

offset The position to remove the string from. *offset* is zero-based.

Return value

TRUE if the element was removed, **FALSE** otherwise.

Description

Remove only has an effect after the *ListWindow* has been opened using the Execute method.

Accept event

ListWindow class

Raised when the user presses Enter or double-clicks on a list element. Default action is to close the list.

Types expected

```
void Accept()
```

Return value

None

Cancel event

ListWindow class

Raised when the user presses Escape. Default action is to close the list.

Types expected

```
void Cancel()
```

Return value

None

Closed event

ListWindow class

Raised when the *ListWindow* is destroyed.

Types expected

```
void Closed()
```

Return value

None

Delete event

ListWindow class

Raised when the user presses Delete. Default action is to do nothing.

Types expected

```
void Delete()
```

Return value

None

KeyPressed event

ListWindow class

Raised when the user presses a key other than Delete, Insert, Accept, or Cancel.

Types expected

```
bool KeyPressed(string keyName)
```

keyName Indicates a key in the standard key format `(<a>` or `<Ctrl-a>)`.

Return value

TRUE indicates that the script has processed the key and that no further processing is desired; **FALSE** indicates that normal processing should take place.

LeftClick event

ListWindow class

Raised when the user left-clicks the *ListWindow.*

Types expected

```
void LeftClick(int xPos, int yPos)
```

xPos The x-position of the mouse at the time of the left-click.

yPos The y-position of the mouse at the time of the left-click.

Return value

None

Move event

ListWindow class

Raised when the selection in the list is changed. Default action is to do nothing.

Types expected

```
void Move()
```

Return value

None

RightClick event

ListWindow class

Raised when the user right-clicks the *ListWindow.*

Types expected

```
void RightClick(int xPos, int yPos)
```

xPos The x-position of the mouse at the time of the right-click.

yPos The y-position of the mouse at the time of the right-click.

Return value

None

PopupMenu class

Description

The *PopupMenu* class manages pop-up menus. In the Borland C++ IDE, pop-up menus are known as SpeedMenus.

Syntax

```
PopupMenu(int Top, int Left, string [] InitialValues)
```

Top, *Left*	Initial coordinates of the pop-up menu.
InitialValues	An array of strings specifying the initial contents of the pop-up menu.

Properties	Access
`[] Data`	Read-only

Methods

```
void Append(string newChoice)
int FindString(string toFind)
string GetString(int offset)
bool Remove(int offset)
string Track()
```

Events

None

PopupMenu class description

PopupMenu class

PopupMenu objects create create a pop-up menu. A pop-up menu pops up and displays a list of menu choices. *PopupMenu* objects control:

- The size and position of the pop-up menu
- The contents of the pop-up menu
- The number of items in the pop-up menu
- Finding and getting strings in the pop-up menu
- Opening and closing the pop-up menu

Data property

PopupMenu class

Contains an array of strings that specifies the choices that will be offered on the pop-up menu.

Access

Read-only

Type expected

```
[] Data
```

Append method

PopupMenu class

Appends a new choice to the pop-up menu.

Types expected

```
void Append(string newChoice)
```

newChoice The name of the new menu choice.

Return value

None

FindString method

PopupMenu class

Looks for the specified menu choice.

Types expected

```
int FindString(string toFind)
```

toFind The name of the string to find.

Return value

The one-based offset of the string found or zero if not found

GetString method

PopupMenu class

Returns a menu choice.

Types expected

```
string GetString(int offset)
```

offset The location of the string to get. *offset* is zero-based.

Return value

The string at the specified offset or "" if the offset is illegal

Remove method

PopupMenu class

Removes the specified menu choice.

Types expected

```
bool Remove(int offset)
```

offset The location of the menu choice to remove. *offset* is zero-based.

Return value

TRUE if the element is removed, **FALSE,** otherwise

Track method

PopupMenu class

Displays the pop-up menu to the user and tracks responses.

Types expected

```
string Track()
```

Return value

The string selected or the empty string ("") if the user cancels the menu

ProjectNode class

Description

Manages the nodes of a project.

Syntax

```
ProjectNode(nodeName, EditView associatedView)
```

nodeName A string indicating the full name of the node (as in MyProg.exe). If no name is specified, *ProjectNode* uses the top level IDE node.

associatedView *associatedView* is optional. If given and if the specified view is associated with a particular project node, the node that represents the *EditView* is used and *nodeName* is ignored. To associate an *EditView* with a *ProjectNode,* create the *EditView.* (To create an *EditView,* double-click the node or press Enter in the Project window.)

Properties	Access
`[] ChildNodes`	Read-only
`string IncludePath`	Read-only
`string InputName`	Read-only
`bool IsValid`	Read-only
`string LibraryPath`	Read-only
`string Name`	Read-only
`bool OutOfDate`	Read-write
`string OutputName`	Read-only
`string SourcePath`	Read-only
`string Type`	Read-only

Methods

```
bool Add(string nodeName [, string type])
bool Build(bool suppressUI)
bool Make(bool suppressUI)
void MakePreview()
bool Remove([string nodeName])
bool Translate(bool suppressUI)
```

Events

```
void Built(bool status)
void Made(bool status)
void Translated(bool status)
```

ProjectNode class description

ProjectNode class

Each node has its own *ProjectNode* class instance. *ProjectNode* class members:

- Display child nodes
- Indicate the node's source, input, output, and library source paths
- Indicate if a specified node is valid
- Indicate the type of node
- Add nodes to and removes nodes from a project
- Build or make a node
- Translate a node

ChildNodes property

ProjectNode class

Indicates all the child nodes of the current node.

Access

Read-only

Type expected

```
[] ChildNodes
```

Description

ChildNodes consists of an array of strings containing the InputNames of the child nodes.

IncludePath property

ProjectNode class

Indicates the path to use for include files for the currently loaded project.

Access

Read-only

Type expected

```
string IncludePath
```

InputName property

ProjectNode class

The node's relative path name of the input file including extension, as in Myfile.cpp or SOURCE\MYFILE.CPP.

Access

Read-only

Type expected

```
string InputName
```

IsValid property

ProjectNode class

Indicates whether a node is valid.

Note: A node becomes invalid if the project file it is associated with is closed or if the node is deleted.

Access

Read-only

Type expected

```
bool IsValid
```

LibraryPath property

ProjectNode class

Indicates the path to use for libraries for the currently loaded project.

Access

Read-only

Type expected

```
string LibraryPath
```

Name property

ProjectNode class

Indicates the node's relative path name with an extension, as in Myfile.cpp or SOURCE\MYFILE.CPP.

Access

Read-only

Type expected

```
string Name
```

OutOfDate property

ProjectNode class

Can be checked, or set, to determine the date of a node.

Access

Read-write

Type expected

```
bool OutOfDate
```

Description

OutOfDate is used by the Make engine to determine if a node needs to be rebuilt.

OutputName property

ProjectNode class

Indicates the relative path name of the output file including extension, as in MYFILE.CPP or Source\Myfile.cpp.

Access

Read-only

Type expected

```
string OutputName
```

Description

You can always generate the absolute file name by prepending the result of IDEApplication.CurrentDirectory to *InputName*, as in:

```
absName = IDE.CurrentDirectory + node.InputName;
```

SourcePath property

ProjectNode class

Indicates the path where the source files for the currently loaded project reside.

Access

Read-only

Type expected

```
string SourcePath
```

Type property

ProjectNode class

Indicates the type of node (.CPP, .H, SourcePool, .LIB, and so on).

Access

Read-only

Type expected

```
string Type
```

Description

When the node is invalid, *Type* contains the empty string ("").

Add method

ProjectNode class

Adds a node to this project node.

Types expected

```
bool Add(string nodeName [, string type])
```

nodeName	The name of the node to add.
type	The type of the node, such as .CPP, .DEF or .RC. If *type* is omitted, it is derived from the *nodeName*.

Return value

TRUE if the node is added, **FALSE,** otherwise

Build method

See also ProjectNode class

Causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node.

Types expected

```
bool Build(bool suppressUI)
```

suppressUI If **TRUE**, the build status dialog will not be displayed during the build process.

Return value

TRUE if the node is built successfully, **FALSE,** otherwise.

Make method

See also ProjectNode class

Causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node if the node's OutOfDate property is **TRUE**.

Types expected

```
bool Make(bool suppressUI)
```

suppressUI If **TRUE**, the build status dialog will not be displayed during the build process.

Return value

TRUE if the node is made successfully, **FALSE,** otherwise

MakePreview method

ProjectNode class

Provides information about what files will be processed if you Make or Build this node.

Types expected

```
void MakePreview()
```

Return value

None

Description

MakePreview performs the same dependency checks as a make and generates a report to the Message window listing the nodes that need to be rebuilt to keep the project up to date.

Remove method

ProjectNode class

Removes the node from the project, if the name of the node is specified.

Types expected

```
bool Remove([string nodeName])
```

nodeName The name of the node to remove from the project. If *nodeName* is not specified, *Remove* removes this node from the project.

Return value

TRUE if the node is removed, **FALSE,** otherwise

Translate method

See also ProjectNode class

Causes the node to be built, made, or translated by the IDE's Make engine according to the rules of the node.

Types expected

```
bool Translate(bool suppressUI)
```

suppressUI If **TRUE**, the build status dialog will not be displayed during the build process.

Return value

TRUE if the node is translated successfully, **FALSE,** otherwise

Built event

See also ProjectNode class

Raised after a build has been performed on the node. The event's default behavior is to do nothing.

Types expected

```
void Built(bool status)
```

status Describes the result of the build. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors.

Return value

None

Made event

See also ProjectNode class

Raised after a make has been performed on the node. The event's default behavior is to do nothing.

Types expected

```
void Made(bool status)
```

status Describes the result of the build. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors.

Return value

None

Translated event

See also ProjectNode class

Raised after a translate has been performed on the node. The event's default behavior is to do nothing.

Types expected

```
void Translated(bool status)
```

status Describes the result of the build. *status* is set to **TRUE** if the build completed successfully or with warnings, and **FALSE** if there were errors.

Return value

None

Description

When the user performs a make or a build, the *ProjectNode* object receives a *Translated* event before it receives the *Built* or *Made* event.

Record class

Description

Creates an empty *Record* object into which keystrokes are saved.

Syntax

```
Record([string RecordName])
```

RecordName The name of the *Record* object. If *RecordName* is not specified, a default name is automatically assigned ("Record1", "Record2", and so on).

Properties	Access
`bool IsPaused`	Read-only
`bool IsRecording`	Read-only
`int KeyCount`	Read-only
`string Name`	Read-write

Methods

```
void Append( int KeyCode )
string GetCommand( int offset )
int GetKeyCode( int offset )
Record Next( void )
```

Events

None

Record class description

Record class

Because *Record* objects can be built programatically (outside the context of the keyboard manager), recordings can be saved to disk and restored, and keyboard sequences can be simulated through script.

KeyboardManager can also be used for key recording. KeyboardManager.StartRecord and KeyboardManager.StopRecord members populate a *Record* object with key sequences. An unlimited number of *Record* objects can be named and iterated.

IsPaused property

Record class

TRUE when *KeyboardManager.PauseRecording* is called in order to allow users to enter keystrokes that will not become part of the recording. **FALSE** otherwise.

Access

Read-only

Type expected

```
bool IsPaused
```

IsRecording property

Record class

TRUE when the *KeyboardManager* begins storing keystrokes to the *Record* object in response to a call to KeyboardManager.StartRecord. **FALSE** otherwise.

Access

Read-only

Type expected

```
bool IsRecording
```

KeyCount property

Record class

The number of keystrokes stored in this *Record* object.

Access

Read-only

Type expected

```
int KeyCount
```

Name property

Record class

The name of the *Record* object. This is a read-write property.

Type expected

```
string Name
```

Append method

Record class

Appends a keycode to the record buffer.

Types expected

```
void Append(int KeyCode)
```

keyCode The keycode to append.

Return value

None

Description

Append allows empty *Record* objects to be built programatically or added to through a script.

GetCommand method

Record class

Returns information describing a key stored in the *Record* object.

Types expected

```
string GetCommand(int offSet)
```

offSet The offset to examine.

Return value

Because the meanings of the stored keystrokes can be altered by the execution of the recording, the information returned is transitory. For example, if the recording switches to another subsystem with a different key map, the stored keystrokes would be different than expected. The return values reflect the values as of the last run.

Description

GetCommand is intended to be used after a *Record* object has been executed.

Note: Keys are stored in the order which they are recorded. The first key in the recording is at offset 0.

GetKeyCode method

Record class

This method returns information describing a key stored in the *Record* object.

Note: Keys are stored in the order which they are recorded. The first key in the recording is at offset 0.

Types expected

```
int GetKeyCode(int offset)
```

offSet The offset to examine.

Return value

The keystroke of the key at the specified offset, or zero if the offset is illegal.

Next method

Record class

As *Record* objects are created, they are automatically linked together. This method provides a mechanism for iterating the recordings.

Types expected

```
Record Next(void)
```

Return value

The next *Record* object or **NULL** indicating the end of the list

ScriptEngine class

Description

A *ScriptEngine* object is responsible for carrying out the action in script files.

Syntax

```
ScriptEngine()
```

Properties	Access
`bool AppendToLog`	Read-write
`int DiagnosticMessageMask`	Read-write
`bool DiagnosticMessages`	Read-write
`string LogFileName`	Read-write
`bool Logging`	Read-write
`string ScriptPath`	Read-write
`string StartupDirectory`	Read-only

Methods

```
int Execute(string commandLine, bool temporary)
string Execute(string commandLine, bool temporary)
bool IsAClass(string className)
bool IsAFunction(string functionName)
bool IsAMethod(string className, string methodName)
bool IsAProperty(string className, string propertyName)
bool IsLoaded(string scriptFileName)
bool Load(string scriptFileName)
[] Modules(bool libraryOnly)
bool Reset(int resetWhat)
void SymbolLoad(string fileName, string symbols)
bool Unload(string scriptFileName)
```

Events

```
void Loaded(string scriptFileName)
void Unloaded(string scriptFileName)
```

ScriptEngine class description

ScriptEngine class

A *ScriptEngine* object loads, unloads, executes, maintains modules and keeps error information on scripts. A *ScriptEngine* object can be created in any script; however, a system wide instance exists.

To create a local instance of a *ScriptEngine* object, use the following syntax:

```
declare ScriptEngine scriptEngine;
```

Once this statement is in your script file, you can use the *ScriptEngine* object as in the following example:

```
Function()
{
  scriptEngine.Load("ascript");
}
```

To reuse the system wide instance, include the following statement in your script file:

```
import scriptEngine;
```

Note: Declaring the script engine locally provides slightly better performance than importing it.

AppendToLog property

ScriptEngine class

Determines whether the next message logged to the log file name should replace an existing log file (if one exists) before performing the write.

Access

Read-write

Type expected

```
bool AppendToLog
```

Description

AppendToLog is used when Logging is on. Once the write is completed, *AppendToLog* is set to **TRUE**, causing subsequent messages to be appended to the log.

DiagnosticMessageMask property

ScriptEngine class

Controls which types of diagnostic messages to record.

Access

Read-write

Type expected

```
int DiagnosticMessageMask
```

Description

DiagnosticMessageMask can be any combination of:

OBJECT_DIAGNOSTICS
METHOD_DIAGNOSTICS
MEMBER_DIAGNOSTICS
ARGUMENT_DIAGNOSTICS
LANGUAGE_DIAGNOSTICS
MODULE_DIAGNOSTICS
FULL_DIAGNOSTICS
NO_DIAGNOSTICS

DiagnosticMessages property

ScriptEngine class

Controls whether diagnostic messages should be recorded in the Message window.

Access

Read-write

Type expected

```
bool DiagnosticMessages
```

LogFileName property

ScriptEngine class

The name of the log file. Defaults to \SCRIPT.LOG.

Access

Read-write

Type expected

```
string LogFileName
```

Logging property

ScriptEngine class

If **TRUE**, script messages will be stored in the log file.

Access

Read-write

Type expected

```
bool Logging
```

ScriptPath property

ScriptEngine class

Holds a string containing the names of the one or more directories to search for script files. Each directory path is separated from the others by a semicolon (;).

Access

Read-write

Type expected

```
string ScriptPath
```

StartupDirectory property

ScriptEngine class

The name of the directory in which the file STARTUP.SPX was found during initialization.

Access

Read-only

Type expected

```
string StartupDirectory
```

Execute method

ScriptEngine class

Executes the specified command.

Types expected

```
int Execute(string commandLine, bool temporary)
string Execute(string commandLine, bool temporary)
```

commandLine The command to execute. *commandLine* must be a valid cScript command.

temporary If *temporary* is **TRUE**, the command is run within a new context and must use **import** to access global variables declared in another module. Any global variables it creates will be used for the purposes of the command and then discarded.

If *temporary* is **FALSE** (the default), the command is executed with the scope of Immediate mode and has automatic access to globals from other modules. In this case, any variables created by the command continue to exist after the command has run and can be accessed from Immediate mode.

Return value

The value appropriate to whatever *commandLine* evaluates to. If that value is an object, it is converted to a string.

IsAClass method

ScriptEngine class

Determines if cScript has seen the class declaration for the specified class.

Types expected

```
bool IsAClass(string className)
```

className The name of the class. *IsAClass* searches for declaration of this class.

Return value

TRUE if instances of the class can be constructed, **FALSE,** otherwise

IsAFunction method

ScriptEngine class

Determines if cScript has seen the function declaration for the specified function.

Types expected

```
bool IsAFunction(string functionName)
```

functionName The name of the function. *IsAClass* searches for declaration of this function.

Return value

TRUE if the function can be called, **FALSE,** otherwise

IsAMethod method

ScriptEngine class

Determines if the specified class has as a method with the specified name.

Types expected

```
bool IsAMethod(string className, string methodName)
```

className	The name of the class. *IsAClass* searches in this class for the method specified in *methodName.*
methodName	The name of the method to search for.

Return value

TRUE if the method is a member of the class, **FALSE,** otherwise

IsAProperty method

ScriptEngine class

Determines if the specified class has as a property with the specified name.

Types expected

```
bool IsAProperty(string className, string propertyName)
```

className	The name of the class. *IsAClass* searches in this class for the property specified in *propertyName.*
propertyName	The name of the property to search for.

Return value

TRUE if the property is a member of the class, **FALSE,** otherwise

IsLoaded method

ScriptEngine class

Determines whether the specified script file has been loaded, or if the file (either the source or the binary) can be found in the ScriptPath.

Types expected

```
bool IsLoaded(string scriptFileName)
```

scriptFileName The name of the script file to load.

Return value

TRUE if the file is loaded or can be loaded, **FALSE,** otherwise

Load method

ScriptEngine class

Loads the specified script file. If not already loaded, the file (either the source or the binary) is searched for using the ScriptPath.

Types expected

```
bool Load(string scriptFileName)
```

scriptFileName The name of the script file to load.

Return value

TRUE if the script was located and loaded, **FALSE** if the script file was not found

Description

If the script file to be loaded has already been loaded into memory, *Load* performs an in-place Reset. (The module's position in the module chain is not affected, but all its variables are restored to their original state.)

The **on** handlers are disconnected or reconnected. All variables local to the module are released and reset. Any code at the module level scope is executed again.

Modules method

ScriptEngine class

Finds all the loaded modules.

Types expected

```
[] Modules(bool ModuleName)
```

ModuleName	One of the following:	
	SCRIPT_MODULES	For all modules
	LIBRARY_MODULES	For only library modules

Return value

An array of strings containing the names of the loaded modules

Reset method

ScriptEngine class

Resets the script session by discarding all modules that match the specified value. If no value is supplied, the method does nothing.

Types expected

```
bool Reset(int resetWhat)
```

resetWhat The module to reset. Can be either LIBRARY_MODULE or SCRIPT_MODULE

Return value

TRUE if the session is reset, **FALSE,** otherwise

SymbolLoad method

ScriptEngine class

Provides hints about where the definition of a given symbol might be. For example:

```
SymbolLoad("ScriptFile", "Foo, Bar, jump")
```

Types expected

```
void SymbolLoad(string fileName, string symbols)
```

fileName A script file that should be loaded if the lookup for any of the listed symbols fails.

symbols A comma delimited string of the symbols which may be resolved by loading *fileName.*

Return value

None

Description

At run time when the Script Engine tries to find a class, function, method, or global variable that it doesn't know about, it consults an internal table constructed by calls to this method.

Unload method

ScriptEngine class

Tries to unload the specified script file. Future references from other scripts to variables, functions or classes defined in the unloaded script file are no longer valid.

Types expected

```
bool Unload(string scriptFileName)
```

scriptFileName The name of the script file to unload.

Return value

FALSE when the script file is not found to have been loaded, **TRUE,** otherwise

Loaded event

ScriptEngine class

Raised whenever a new script module is successfully loaded.

Types expected

```
void Loaded(string scriptFileName)
```

scriptFileName The name of the script file that was loaded.

Return value

None

Unloaded event

ScriptEngine class

Raised when when a module has been unloaded.

Types expected

```
void Unloaded(string scriptFileName)
```

scriptFileName The name of the script file that was unloaded.

Return value

None

SearchOptions class

Description

The *SearchOptions* class members search for text and error locations in your script file.

Syntax

```
SearchOptions()
```

Properties	Access
`bool CaseSensitive`	Read-write
`bool FromCursor`	Read-write
`bool GoForward`	Read-write
`bool PromptOnReplace`	Read-write
`bool RegularExpression`	Read-write
`bool ReplaceAll`	Read-write
`string ReplaceText`	Read-write
`string SearchReplaceText`	Read-write
`string SearchText`	Read-write
`bool WholeFile`	Read-write
`bool WordBoundary`	Read-write

Methods

```
void Copy(SearchOptions optionsToCopyFrom)
```

Events

None

SearchOptions class description

SearchOptions class

SearchOptions class members search and replace occurrences of text strings. *SearchOptions* class members allow:

- Case sensitive searching
- Searching from the current cursor position
- Searching forward or backward in the file
- Confirmation before text replacements
- Use of regular expressions in the search
- Replacement of all matching text
- Searching and replacing in the same operation

CaseSensitive property

SearchOptions class

If **TRUE**, a case-sensitive search is performed.

Access

Read-write

Type expected

```
bool CaseSensitive
```

FromCursor property

SearchOptions class

If **TRUE**, the search is made from the current cursor position.

Access

Read-write

Type expected

```
bool FromCursor
```

GoForward property

SearchOptions class

If **TRUE**, the search is "forward" towards the end of the file.

Access

Read-write

Type expected

```
bool GoForward
```

PromptOnReplace property

SearchOptions class

If **TRUE**, you are prompted (before a replacement is made) to confirm each instance where the SearchReplaceText will be replaced by the ReplaceText.

Access

Read-write

Type expected

```
bool PromptOnReplace
```

RegularExpression property

SearchOptions class

If **TRUE**, regular expressions are used in matching the SearchText or SearchReplaceText with the text to be searched.

Access

Read-write

Type expected

```
bool RegularExpression
```

ReplaceAll property

SearchOptions class

If **TRUE**, all text which matches the SearchReplaceText is replaced with the ReplaceText without any prompting for confirmation.

Access

Read-write

Type expected

```
bool ReplaceAll
```

ReplaceText property

SearchOptions class

Contains text which replaces instances of the SearchReplaceText string(s) found in the text being searched.

Access

Read-write

Type expected

```
string ReplaceText
```

SearchReplaceText property

SearchOptions class

Contains the text to search for in a search and replace operation (not a search-only operation).

Access

Read-write

Type expected

```
string SearchReplaceText
```

SearchText property

SearchOptions class

Contains the text to search for in a search operation (not a search and replace operation).

Access

Read-write

Type expected

```
string SearchText
```

WholeFile property

SearchOptions class

If **TRUE**, the whole file is searched for SearchText or SearchReplaceText, regardless of the cursor position.

Access

Read-write

Type expected

```
bool WholeFile
```

WordBoundary property

SearchOptions class

If **TRUE**, a match between SearchText or SearchReplaceText and the text being searched only occurs if the characters in *SearchText* make up an entire word (that is, they are surrounded by whitespace) and are not embedded in a larger word.

Access

Read-write

Type expected

```
bool WordBoundary
```

Copy method

SearchOptions class

Creates a copy of the current *SearchOptions*.

Types expected

```
void Copy(SearchOptions optionsToCopyFrom)
```

optionsToCopyFrom The options to copy.

Return value

None

StackFrame class

Description

StackFrame class members display information about the call stack.

Syntax

```
StackFrame(int howFarBack)
```

howFarBack The number of stack frames to go back through.

- If *howFarBack* is 0, the stack frame for this call is retrieved.
- If *howFarBack* is 1, the stack passed to this function's caller is retrieved, and so on.

When *howFarBack* is less than the depth of the stack, the object is not valid.

Properties	**Access**
`int ArgActual`	Read-only
`int ArgPadding`	Read-only
`string Caller`	Read-write
`bool IsValid`	Read-only

Methods

```
StackElement GetParm(int parmNumber)
string IngType(int arg)
bool SetParm(int parmNumber, newValue)
```

Events

None

StackFrame class description

StackFrame class

StackFrame class members display information about the call stack, the sequence of function calls that brought your script program to its current state. It deciphers all active functions and their argument values and displays them in a readable format.

The most recently called function displays at the top of the list, followed by its caller and the previous caller to that. The list continues to the first function in the calling sequence, which displays at the bottom of the list.

StackFrame class members:

- Return the number of arguments that were actually passed to a method.
- Indicate the number of objects cScript had to pad or truncate from the original call stack.
- Indicate the name of the method owning the stack frame.
- Indicate if the stack frame is valid.
- Return the object at a specified stack frame offset.

ArgActual property

StackFrame class

Indicates the number of objects on the cScript stack belonging to this call frame.

Access

Read-only

Type expected

```
int ArgActual
```

Description

ArgActual is the number of arguments that were actually passed to a method. cScript either pads or truncates arguments as necessary, so it must keep track of the number actually passed.

For example, if you have a call in your code to

```
MyMethod("hi");
```

its declaration shows the following:

```
MyMethod(first, second, third, fourth){
  print first, second, third, fourth;
}
```

If you were to insert `x = new StackFrame(0);` into the call to *MyMethod*, the value of *x.ArgActual* would be 1 since only one argument is passed.

ArgPadding property

StackFrame class

Indicates the number of objects cScript had to pad or truncate from the original call stack to resolve any discrepancy between the number of arguments in the declaration and the number of arguments in the call.

Access

Read-only

Type expected

```
int ArgPadding
```

Caller property

StackFrame class

Indicates the name of the method owning the stack frame.

Access

Read-only

Type expected

```
string Caller
```

Description

Caller contains the empty string ("") if the call is a top level one. When the value is set, it is reflected in subsequent *StackFrame* calls until the current stack frame is popped off, at which point the value of *Caller* is reset to its original value.

IsValid property

StackFrame class

FALSE if the object was constructed with an invalid stack frame depth or if the stack frame has gone out of scope. It is **TRUE** otherwise.

Access

Read-only

Type expected

```
bool IsValid
```

InqType method

StackFrame class

Returns the type of argument.

Types expected

```
string InqType(int arg)
```

arg The specified argument.

Return value

A descriptor for the argument specified. If *arg* is greater than or equal to ArgActual, "Out of range" is returned.

GetParm method

StackFrame class

Returns the object at the specified stack frame offset.

Types expected

```
StackElement GetParm(int parmNumber)
```

parmNumber The number of the parameter to return.

Return value

The object at the specified stack frame offset.

SetParm method

StackFrame class

Sets the value of the object at the specified stack frame offset.

Types expected

```
bool SetParm (int parmNumber, newValue)
```

parmNumber The value of the object to change.

newValue The object's new value.

Return value

TRUE when the value was successfully changed. **FALSE** if the *StackFrame* is currently invalid or if *parmNumber* is not within the range of arguments specified for the *StackFrame.*

String class

Description

The *String* object manipulates text.

Syntax

```
String(string theText)
```

theText The text to declare as a text object.

Properties	Access
`int Character`	Read-write
`int Integer`	Read-write
`bool IsAlphaNumeric`	Read-only
`int Length`	Read-only
`string Text`	Read-write

Methods

```
String Compress()
bool Contains(string charactersToLookFor, int mask)
int Index(string substr[, int direction])
String Lower()
String SubString(int startPos[, int length])
String Trim([bool fromLeft])
String Upper()
```

Events

None

String class description

String class

The *String* class manipulates text characters independently of each other. To store and manipulate text characters as a group, delcare the text as a string (note the lower case "s").

Class members can be used to:

- Get the first character of the text.
- Get the numeric equivalent of the beginning of the text. (Useful for converting text to numeric values.)
- Test if the first character is alphanumeric.
- Return the length of the text.
- Return the text object as a string versus a String.
- Convert multiple whitespace characters to one whitespace character.
- Find any number of characters within the text.
- Return the offset of a particular character.
- Lower case the text.
- Return a portion of the text as a String.
- Remove whitespace from either the right or left of the text.
- Upper case the text.

Character property

String class

Indicates the integer value of character 0 of the string.

Access

Read-write

Type expected

```
int Character
```

Description

When the value of *Character* is set, it changes the whole string to the new value.

For example, if you start with a string *Str* containing the text "FOO", the value of *Str.Text* is "FOO" and the value of *Str.Character* is 'F'. If you then set the value of *Str* with `Str.Character = 'X'`, the value of *Str.Text* is now "X" and not "XOO".

Integer property

String class

Indicates the numerical equivalent of the character string that this object represents, or zero if the string does not contain numerals.

Access

Read-write

Type expected

```
int Integer
```

IsAlphaNumeric property

String class

TRUE if the text of the *String* is made up entirely of alphanumeric characters (determined by checking the system's current locale). **FALSE,** otherwise.

Access

Read-only

Type expected

```
bool IsAlphaNumeric
```

Length property

String class

Calculates length of the string (equivalent to *strlen*). Does not include the **NULL.**

Access

Read-only

Type expected

```
int Length
```

Text property

String class

The character string that this object represents.

Access

Read-write

Type expected

```
string Text
```

Compress method

String class

Compresses a strong.

Types expected

```
String Compress()
```

Return value

A new string consisting of *String* with whitespace removed.

Contains method

String class

Searches *String* for the specified characters.

Types expected

```
bool Contains(string charactersToLookFor, [int mask])
```

charactersToLookFor The characters to search for in *String.*

mask Any of the following constants:

Constant	Description
BACKWARD_RIP	Rip from left to right.
INVERT_LEGAL_CHARS	Interpret the string as the inverse of the string you wish to use. In other words, specify "t" to mean any ASCII value between 1 and 255 except ‘t’.
INCLUDE_LOWERCASE_ALPHA_CHARS	Append the characters `abcdefghijklmnopqrstuvwxyz` to the string.
INCLUDE_UPPERCASE_ALPHA_CHARS	Append the characters `ABCDEFGHIJKLMNOPQRSTUVWXYZ` to the string
INCLUDE_ALPHA_CHARS	Append both uppercase and lowercase alpha characters to the string.
INCLUDE_NUMERIC_CHARS	Append the characters `1234567890` to the string.
INCLUDE_SPECIAL_CHARS	Append the characters `` `-=[]\;',./~!@#$%^&*()_+{}|:"<>? `` to the string.

Return value

TRUE if the string contains one of the characters specified, **FALSE,** otherwise

Index method

String class

Scans the string for an embedded occurrence of the specified substring.

Index does not accept regular expressions.

Types expected

```
int Index(string substr[, int direction])
```

substr	The string to search for.
direction	The direction to search in. One of:
	SEARCH_FORWARD (default)
	SEARCH_BACKWARD

Return value

0 if *substr* is not found or, if found, the one based offset + 1 of the substring

Lower method

String class

Translates *String* to lowercase.

Types expected

```
String Lower()
```

Return value

A new string consisting of *String* in lower case text.

SubString method

String class

This method returns a new string consisting of the specified substring.

Types expected

```
String SubString(int startPos[, int length])
```

startPos	The starting point of the substring in the string.
length	The number of characters in the substring. Defaults to MAX_EDITOR_LINE_LEN (1024). If *length* is not specified, *SubString* continues to the end of the string.

Return value

A new string consisting of the specified substring.

Trim method

String class

Trims whitespace from *String.*

Types expected

```
String Trim([bool fromLeft])
```

fromLeft If **TRUE,** trims leading whitespace. If **FALSE,** trims trailing whitespace.

Return value

A new string consisting of *String* without trailing or leading whitespaces (depending on *fromLeft* selection).

Upper method

String class

Translates *String* to upper case.

Types expected

```
String Upper()
```

Return value

A new string consisting of *String* in upper case text.

TimeStamp class

TimeStamp indicates the current time. It initializes to the system time at the time of construction.

Syntax

```
TimeStamp()
```

Properties	Access
`int Day`	Read-write
`int Hour`	Read-write
`int Hundredth`	Read-write
`int Millisecond`	Read-write
`int Minute`	Read-write
`int Month`	Read-write
`int Second`	Read-write
`int Year`	Read-write

Methods

```
int Compare(TimeStamp tstamp)
string DayName()
string MonthName()
```

Events

None

Day property

TimeStamp class

Indicates the current day in the range of 0 (Sunday) to 6 (Saturday).

Access

Read-write

Type expected

```
int Day
```

Hour property

TimeStamp class

Indicates the current hour in the range of 0 (Midnight) to 23 (11:00 PM).

Access

Read-write

Type expected

```
int Hour
```

Hundredth property

TimeStamp class

Indicates the current hundredth of an hour in the range of 0 to 99.

Access

Read-write

Type expected

```
int Hundredth
```

Millisecond property

TimeStamp class

Indicates the number of milliseconds after the current second in the range of 0 to 999.

Access

Read-write

Type expected

```
int Millisecond
```

Minute property

TimeStamp class

Indicates the number of minutes after the current hour in the range of 0 to 59.

Access

Read-write

Type expected

```
int Minute
```

Month property

TimeStamp class

Indicates the current month of the year in the range of 0 (January) to 11 (December).

Access

Read-write

Type expected

```
int Month
```

Second property

TimeStamp class

Indicates the number of seconds after the current minute in the range of 0 to 59.

Access

Read-write

Type expected

```
int Second
```

Year property

TimeStamp class

Indicates the current year.

Access

Read-write

Type expected

```
int Year
```

Compare method

TimeStamp class

Compares the time properties of the calling *TimeStamp* object with those of the *tstamp* argument.

Types expected

```
int Compare(TimeStamp tstamp)
```

tstamp The properties to compare *TimeStamp* to.

Return value

-1 if the calling *TimeStamp* is newer than *tstamp*, 0 if the calling *TimeStamp* is the same age as *tstamp*, and 1 if the calling *TimeStamp* is older than *tstamp*.

DayName method

TimeStamp class

Returns the name of the current day of the week.

Types expected

```
string DayName()
```

Return value

Monday, Tuesday, and so on

MonthName method

TimeStamp class

Returns the name of the current month.

Types expected

```
string MonthName()
```

Return value

January, February, and so on

TransferOutput class

Description

Internally created by the IDE after processing a transfer tool, *TransferOutput* is passed to the IDE event TransferOutputExists.

Syntax

```
TransferOutput()
```

Properties	Access
`int MessageId`	Read-only
`string Provider`	Read-only

Methods

```
string ReadLine()
```

Events

None

TransferOutput class description

TransferOutput class

An object of type *TransferOutput* is internally created by the IDE whenever a transfer operation is performed.

When the IDE starts a transfer, it outputs a message to the Message window saying "Transferring to *ToolName*..."

When a transfer happens, the IDE captures all its output and stores it in an internal buffer. The contents of this buffer may be accessed by using TransferOutput.ReadLine.. This method returns the next line of text until the stream is exhausted, at which point it returns **NULL**.

The IDE contains built-in processing for tools it commonly transfers to. These tools include TASM and Grep. The script sample files FILTSTUB.SPP and FILTERS.SPP show uses of this class in action.

MessageId property

TransferOutput class

The owning message stored to the message system.

Access

Read-only

Type expected

```
int MessageId
```

Description

MessageID is intended to be used as the *parentMessage* parameter of IDE.MessageCreate. The messages produced by the transfer can be grouped with the transfer message rather than at the same level.

Provider property

TransferOutput class

Indicates the name of the tool that was spawned by the transfer; for example, COMMAND.COM.

Access

Read-only

Type expected

```
string Provider
```

ReadLine method

TransferOutput class

Reads the next line of text that was produced by the transfer.

Types expected

```
string ReadLine()
```

Return value

The next line of text that was produced by the transfer. If the line is empty, returns the empty string (""). If there is no more input to read, it returns **NULL**.

Description

When a transfer happens, the IDE captures all its output and stores it in an internal buffer. The contents of this buffer may be accessed by repeatedly calling *ReadLine*, which returns the next line of text until the stream has been exhausted, at which point it returns **NULL**.

ObjectScripting error messages

{button Symbols,JI(`',`serrorsxref_symbols')} {button a,JI(`',`serrorsxref_A')} {button b,JI(`',`serrorsxref_B')} {button c,JI(`',`serrorsxref_C')} {button d,JI(`',`serrorsxref_D')} {button e,JI(`',`serrorsxref_E')} {button f,JI(`',`serrorsxref_F')} {button g,JI(`',`serrorsxref_G')} {button i,JI(`',`serrorsxref_I')} {button l,JI(`',`serrorsxref_L')} {button m,JI(`',`serrorsxref_M')} {button n,JI(`',`serrorsxref_N')} {button o,JI(`',`serrorsxref_O')} {button p,JI(`',`serrorsxref_P')} {button r,JI(`',`serrorsxref_R')} {button s,JI(`',`serrorsxref_S')} {button t,JI(`',`serrorsxref_T')} {button u,JI(`',`serrorsxref_U')} {button w,JI(`',`serrorsxref_W')}

Symbols

‘)’ expected

‘ .’ expected

‘ const’ is valid only for variables

#ELSE without prior #IFDEF

#ENDIF expected

#ENDIF without prior #IFDEF

#ERROR: *error message*

A

A private buffer may not be attached to an EditView

A STEP value of 0 will result in an endless loop

An EditBuffer with a conflicting value for ‘IsPrivate’ already exists on EditView, argument 2 ignored

An EditBuffer with a conflicting value for ‘ReadOnly’ already exists, argument 3 ignored

An unrecognized Script Message has been encountered

Argument number is NULL -- converted to integer

Argument number to method methodName is not a valid parameter

Argument number to the className constructor is invalid

Attempt to access empty string from method methodName of class className

Attempt to access empty string from property propertyName of class className

Attempt to access invalid object of type name

Attempt to access non-existent element number

Attempt to coerce non-integral string string

Attempt to create a new object failed

Attempt to create an object of unknown type: ‘type’

Attempt to dispatch to unknown command ID

Attempt to invoke unrecognized tool toolName

Attempt to modify non-modifiable keyboard

Attempt to pop the last keyboard off the component’s stack

Attempt to set value of manifest constant name ignored

Attempt to use negative array index value

Attempt to write to the read-only property propertyName of class className

Attempted divide by zero

Attempted method/property access using unallocated object

Attempted modulo by zero

Auto loading file name

B

Bad token file

Break or continue has no enclosing loop

C

Call to non-existent method ‘name’
Cannot locate external variable ‘variable’
Cannot perform member selection on class-name ‘name’
Command parsing failed for: command
‘__const’ is valid only for variables
Constant objects must be initialized when declared
Created a new name

D

Default parameter must be last
Destroyed a name
Destructor name must match class name
Destructors are only allowed at module and class scope
Dispatch of member failed
Dispatch of method failed
Dispatch of object failed

E

ELSE without preceding IF
ELSE without prior #IFDEF
End of comment encountered outside of a comment block
ENDIF expected
ENDIF without prior #IFDEF
ERROR: *error message*
Exception fault detected. Script terminated
‘ export’ allowed only at module scope
Expression requires a numeric value
Expression requires an lvalue

F

Failed to load new factory from name
Failed writing number bytes to file name. Disk full?
File name cannot be read
File name failed to parse
File name is unrecognizable as a token file

G

Got ‘tokenA’ when ‘tokenB’ was expected
Got 'token' when an external type was expected

I

Identifier exceeds maximum length
Identifier expected
Identifier ‘name’ already defined
Identifier ‘name’ may clash with a global; use ‘declare’ to disambiguate
Identifier ‘name’ reserved for future use
Illegal character ‘char’ encountered
Illegal opcode encountered
Illegal or duplicate function name ‘name’

Import and export are mutually exclusive
Improperly formatted character constant: 'constant'
Invalid function definition
Invalid or reserved identifier ‘name’
Iterator object required
Iterator variable required

L

Label expected
Loaded new factory from name
Loading file name

M

Macro 'macro' must have less than 16 arguments
Macro expansion text for 'macro' is too large
Member selection (‘.’) not allowed in object declarations
Method name accepts number parameters, none supplied
Missing ‘(‘
Missing ‘)’
Missing ')' in macro: '*macro*'
Missing term in expression
Module level code is too large
Module name not found
Module name removed

N

No Editor windows are currently active
No Factory DLL name specified
No initial elements were passed to the List constructor, the list starts empty
No windows currently exist
number is too narrow for a List, width adjusted to newNumber
number is too short for a List, height adjusted to newNumber

O

Object expected
Only ‘=’ is allowed here
Overwrite of pass-by-value parameter

P

Parameter to LOAD must be a string
Preprocessor nesting level too deep
Processing of module name has aborted

R

Recursive macro expansion of 'macro' is too deep
Reference to non-existent property
Return value is NULL -- converted to integer
Returning an unrecognized value for window style
Returning from a module will override the standard return value of ‘run’

S

Skipping WITH block since object in NULL
Stack Overflow
Statement can only be used at module scope
String expected
Successfully parsed module
Symbol already defined
SymbolLoad table was unable to resolve symbol symbolName from module moduleName
Syntax error
Syntax error

T

The argument passed to the EditPosition constructor was not an EditBuffer nor an EditView
The command is too large for the input buffer
The component name has no registered keyboards
The Debugger is not available to process the name method
The Debugger is not available to process the name property
The debugger method methodName failed
The desired coordinate for left is larger than desired coordinate for right
The desired coordinate for top is larger than desired coordinate for bottom
The displayed list contains no members
The keyboard manager cannot playback while it is recording
The keyboard manager has nothing to playback
The keyboard manager is not currently paused
The keyboard manager is not currently playing back
The keyboard manager is not currently recording
The method methodName failed
The name object is no longer valid
The offset number is out of range for the list
The offset number is out of range for the menu
The parameter EditView passed to Attach() is currently parented elsewhere
The PushBack stack has been overrun
The region type ‘regionName’ is unrecognized
The string string was not found in the list
The string string was not found in the menu
The toolName tool encountered errors
The toolName tool encountered fatal errors
The TransferOutput constructor cannot open its input stream
The value number for argument argNumber is out of range for method methodName of class className
The value number is out of range for method methodName of class className
The value number is out of range for property propertyName of class className
‘ token’ is not a valid macro name
Token ‘token’ is unknown
Too many defaults
Too many expressions in array initialization

U

Unable to create menu
Unable to determine type of currently active window
Unable to iterate in object ‘object’
Unable to locate file name
Unable to locate project node for projectName
Unable to open file name
Unable to open include file: *filename*
Unable to open script: *filename*
Unexpected ‘,’
Unexpected ‘.’
Unexpected ‘*token*’
Unexpected end of file due to an unterminated comment
Unexpected end of file
Unexpected end of line
Unexpected end of statement
Unexpected operator ‘operator’
Unhandled script exception caught
Uninitialized value on right hand side of assignment
Unknown base class
Unknown preprocessor directive: *directive*
Unknown token
Unloaded file name
Unmatched right brace ‘}’
Unterminated string: macro
User Break detected, script aborting
User error
User warning

W

Writing file name
Wrong number of arguments in call to macro: 'macro'

Syntax error ObjectScripting message

This is a general error indicating that the statement does not follow proper cScript syntax. Common causes for this error are misspelled keywords, missing operators, or no semicolon (;) at the end of the statement.

Got '*tokenA*' when '*tokenB*' was expected ObjectScripting message

The *tokenA* (keyword, operator, variable, or punctuation) that was encountered was not what the compiler expected to see here to make a syntactically correct statement. What should have appeared at this point was *tokenB*.

Unexpected '*token*' ObjectScripting message

The compiler encountered a *token* (keyword, operator, variable, or punctuation) that doesn't make sense in this context.

Unknown token ObjectScripting message

The compiler does not recognize the encountered token or its purpose.

Statement can only be used at module scope

ObjectScripting message

This statement can only be used at the outermost scope within a file—it cannot be used inside of any nested constructs.

String expected ObjectScripting message

The compiler was expecting a string, but instead another data type was used.

Symbol already defined ObjectScripting message

An attempt was made to **define** a symbol that has already been defined in this module.

Default parameter must be last ObjectScripting message

Any default parameters defined in a function's argument list must be at the end of the argument list.

Too many defaults

ObjectScripting message

Too many default parameters have been defined in the function's argument list.

ELSE without preceding IF ObjectScripting message

The compiler encountered an **else** without a preceding **if** to begin the conditional block.

Missing ‘(‘ ObjectScripting message

An opening parenthesis is missing from an expression.

Missing ‘)’ ObjectScripting message

A closing parenthesis is missing from an expression.

Identifier expected ObjectScripting message

The compiler expected an identifier but encountered something else.

Missing term in expression ObjectScripting message

A required part of this expression is missing.

‘.’ expected ObjectScripting message

This statement requires a dot (.) operator, which was not provided.

Too many expressions in array initialization

ObjectScripting message

There are too many expressions that make up the initialization statement for an array. Simplify the expressions by breaking them up into separate statements and making the result the array initializer.

Label expected ObjectScripting message

The compiler expected a label, but encountered something else.

‘export’ allowed only at module scope

ObjectScripting message

The **export** keyword cannot be used inside of a function or class definition.

‘__const’ is valid only for variables ObjectScripting message

Variables in external DLL function prototypes should use the keyword ‘const’ to indicate that the argument cannot be changed by the external function. ‘__const’ is a separate keyword used for cScript variables.

Got *token* when an external type was expected ObjectScripting message

You specified a type, other than a cScript type, as the return value from an external DLL function call. This is not permitted.

Unmatched right brace ‘}’ ObjectScripting message

A closing brace (}) was encountered without a matching opening brace ({).

Iterator variable required

ObjectScripting message

An output variable is required in the **iterate** statement.

Iterator object required ObjectScripting message

An object reference is required in the **iterate** statement.

Unknown base class ObjectScripting message

The base class for this class definition has not been defined or cannot be found.

Destructor name must match class name ObjectScripting message

The name of the destructor function in this class definition (the function beginning with a tilde (~)) must be the same as the class name.

Destructors are only allowed at module and class scope ObjectScripting message

The destructor for a class must be defined within that class, and the destructor for a module must be defined at the outermost scope within that module.

Only ‘=’ is allowed here ObjectScripting message

The assignment operator (=) is required in this statement.

Member selection (‘.’) not allowed in object declarations ObjectScripting message

You may not access the properties/methods/events of an object at the same time you **declare** it. Declare the object first, then access its members.

Import and export are mutually exclusive ObjectScripting message

You cannot use the **import** and **export** keywords together in the same declaration statement.

Preprocessor nesting level too deep ObjectScripting message

Too many **#ifdef** blocks contained within each other.

#ELSE without prior #IFDEF ObjectScripting message

You cannot have an **#else** without an **#ifdef** preceding it to start a conditional block.

#ENDIF without prior #IFDEF ObjectScripting message

You cannot have an **#endif** without an **#ifdef** preceding it to start a conditional block.

#ENDIF expected ObjectScripting message

You must use an **#endif** to terminate a conditional block started with **#ifdef**.

Unable to open script: *filename*

ObjectScripting message

The requested script file was not found or is locked by another application.

Unknown preprocessor directive: *directive* ObjectScripting message

The given preprocessor directive is not recognized by cScript.

Unable to open include file: *filename* ObjectScripting message

The requested include file was not found or is locked by another application.

Missing ')' in macro: '*macro*' ObjectScripting message

Unmatched parentheses in the macro definition.

#ERROR: *error message* ObjectScripting message

User-defined error caused by the **#error** directive.

Macro '*macro*' must have less than 16 arguments ObjectScripting message

A macro may not have more than sixteen arguments in its argument list.

Macro expansion text for '*macro*' is too large

ObjectScripting message

The physical size of the macro definition exceeds cScript limitations.

Wrong number of arguments in call to macro: '*macro*'

ObjectScripting message

The number of arguments in the call to the macro did not match its definition.

Module level code is too large ObjectScripting message

The script file is too large to be loaded.

Unterminated string: *macro* ObjectScripting message

A string is not properly terminated. Usually this implies a missing double-quote (").

Improperly formatted character constant: '*constant*'

ObjectScripting message

The syntax used to format a character constant is not correct.

Unexpected end of file due to an unterminated comment ObjectScripting message

A comment, begun with " /* " was not terminated with " */ ".

End of comment encountered outside of a comment block

ObjectScripting message

A " */ " comment terminator was found without a " /* " to begin the comment before it.

Recursive macro expansion of '*macro*' is too deep

ObjectScripting message

A macro, which expands recursively, called itself too many times.

Bad token file ObjectScripting message

The compiled cScript script file (.spx) is corrupted or missing.

Illegal character '*char*' encountered ObjectScripting message

An illegal character was encountered in this statement while the compiler was tokenizing the script file.

Identifier exceeds maximum length ObjectScripting message

An identifier name cannot be longer than 64 characters.

‘*token*’ is not a valid macro name ObjectScripting message

The *token* you attempted to use as a macro is not a macro.

Unexpected end of statement ObjectScripting message

The parser encountered the end of the statement (usually a semicolon) before a complete cScript statement had been parsed.

Unexpected end of line ObjectScripting message

The parser encountered the end of the line before a complete cScript statement had been parsed.

Unexpected end of file ObjectScripting message

The parser encountered the end of the file before a complete cScript statement had been parsed.

Unexpected ‘,’ ObjectScripting message

A comma (,) was encountered in a statement in a place where a comma does not belong.

Unexpected ‘.’ ObjectScripting message

A dot (.) was encountered in a statement in a place where a dot does not belong.

Unexpected operator ‘*operator*’ ObjectScripting message

An operator was encountered in a statement in a place where an operator does not belong.

Token ‘*token*’ is unknown ObjectScripting message

The parser encountered a token which is not part of cScript, and was not previously defined by the user.

Syntax error ObjectScripting message

The parser encountered a statement that does not conform to cScript language syntax.

‘)’ expected ObjectScripting message

An opening parenthesis does not have a matching closing parenthesis.

Invalid function definition

ObjectScripting message

The function definition is syntactically or logically incorrect.

Object expected ObjectScripting message

The parser expected an object reference in the statement, but found something else.

Illegal or duplicate function name '*name*' ObjectScripting message

The parser encountered an illegal function name, such as a name beginning with a number, or the same function name was defined twice.

Invalid or reserved identifier '*name*' ObjectScripting message

The parser encountered an illegal identifier name, such as a name beginning with a number, or you attempted to use a cScript reserved word as an identifier name.

Identifier ‘*name*’ already defined ObjectScripting message

The identifier *name* has already been defined in the current scope.

Cannot perform member selection on class-name '*name*'

ObjectScripting message

An attempt was made to perform an operation on a class, instead of an object instantiated from the class.

Identifier ‘*name*’ reserved for future use ObjectScripting message

The identifier *name* has been reserved for possible use in future versions of cScript.

Identifier ‘*name*’ may clash with a global; use ‘declare’ to disambiguate

ObjectScripting message

A local variable has the same name as an existing global variable. You should either use **export**/**import** to access the global, or **declare** to create a new local variable.

Constant objects must be initialized when declared

ObjectScripting message

Any object declared **const** must be initialized with a valid value when first declared.

Break or continue has no enclosing loop ObjectScripting message

The keywords **break** and **continue** can only be used in the cScript looping statements: **while**, **do..while**, **for**, and **iterate**. In addition, **break** is valid in a switch statement.

Expression requires an lvalue ObjectScripting message

When an expression includes an operator that alters the value of an operand, such as assignment (=) or incrementation (++), that operand must be a variable or class instance property rather than a constant or an expression.

A STEP value of 0 will result in an endless loop ObjectScripting message

The *condition* value of this loop does not change and therefore the loop will never terminate. If this is desired, use a **while** loop with a **break** statement to terminate the loop.

Attempted divide by zero

ObjectScripting message

Zero cannot be the divisor of an expression.

Attempted modulo by zero ObjectScripting message

Zero cannot be the divisor of an expression.

Parameter to LOAD must be a string ObjectScripting message

The **load** command requires a string, the name of a script module, as its parameter.

Illegal opcode encountered

ObjectScripting message

This error indicates a problem in the script engine. Please report this error to Borland Customer Support.

Exception fault detected. Script terminated.

ObjectScripting message

An error has occurred in a system underlying cScript.

Stack Overflow ObjectScripting message

The number of objects pushed onto the stack exceeds the stack's capacity.

Uninitialized value on right hand side of assignment

ObjectScripting message

One or more variables in an expression on the right side of an assignment statement is uninitialized or NULL.

Attempt to use negative array index value ObjectScripting message

Array index values must be positive integers or strings.

Call to non-existent method '*name*' ObjectScripting message

The method *name* does not exist or cannot be found.

Attempted method/property access using unallocated object ObjectScripting message

You attempted to call a method or access a property of an object that does not exist.

Reference to non-existent property ObjectScripting message

The property you attempted to access does not exist for this object.

Attempt to create an object of unknown type: '*type*'

ObjectScripting message

The *type* used to create this object does not exist. Check the spelling of the class name.

Skipping WITH block since object in NULL ObjectScripting message

Skipping the **with** block that deals with the object because the object is NULL and has no defined properties or methods..

Overwrite of pass-by-value parameter

ObjectScripting message

The value of a parameter, passed by value to a function, was overwritten with a new value.

Cannot locate external variable '*variable*' ObjectScripting message

You attempted to **import** a variable that was not **export**ed from another module.

User warning ObjectScripting message

A user defined warning message.

User error ObjectScripting message

A user defined error message.

Returning from a module will override the standard return value of ‘run’

ObjectScripting message

A module, by default, returns true when completing its run. You are overriding that return value with another value or no value.

Unable to iterate in object '*object*' ObjectScripting message

The object may not have any members to iterate through.

Expression requires a numeric value

ObjectScripting message

This expression requires an integer or floating point value--the value provided was of a different type.

Loaded new factory from *name*

ObjectScripting message

Factory was successfully loaded from the given source.

Module *name* removed ObjectScripting message

The given module was successfully removed from memory.

Failed to load new factory from *name*

ObjectScripting message

The factory was not successfully loaded from the given source.

Attempt to create a new *object* failed ObjectScripting message

object could not be created.

File *name* is unrecognizable as a token file ObjectScripting message

The given file is not a cScript token file.

File *name* cannot be read

ObjectScripting message

The file *name* is locked by another application, corrupted, or missing.

Unable to open file *name*

ObjectScripting message

The file *name* is locked by another application, corrupted, or missing.

Method *name* accepts *number* parameters, none supplied

ObjectScripting message

This method is expecting up to *number* parameters to be passed to it. No parameters were given.

Failed writing *number* bytes to file *name*. Disk full?

ObjectScripting message

The file *name* is locked by another application, corrupted, or missing, or the disk is full.

The command is too large for the input buffer

ObjectScripting message

The length of the command line is too large. Try breaking it up into multiple statements.

Module *name* not found ObjectScripting message

The module *name* is not in the current directory or the directorie(s) designated in the script path.

Unable to locate file *name*

ObjectScripting message

The file *name* is not in the current directory or the directorie(s) designated in the script path.

The Debugger is not available to process the *name* method

ObjectScripting message

Cannot load the debugger for this method.

The Debugger is not available to process the *name* property message

ObjectScripting

Cannot load the debugger for this property.

The TransferOutput constructor cannot open its input stream

ObjectScripting message

A new *TransferOutput* object was unable to open its input stream and will be unable to transfer data.

Unable to create menu ObjectScripting message

The requested menu cannot be created.

An unrecognized Script Message has been encountered ObjectScripting message

The script runtime engine has received a message that it does not know how to process.

Processing of module *name* has aborted ObjectScripting message

The script module *name* has aborted prior to reaching the end of the script.

No Factory DLL name specified

ObjectScripting message

An attempt was made to access functions in a DLL without first specifying the DLL name.

Attempt to set value of manifest constant *name* ignored ObjectScripting message

The value of a manifest constant cannot be changed.

Unhandled script exception caught ObjectScripting message

The script generated an exception which was not handled within the script.

Loading file *name*

ObjectScripting message

The file *name* is loading into memory.

Unloaded file *name*

ObjectScripting message

The file *name* is unloaded from memory.

Successfully parsed *module* ObjectScripting message

The file *module* was successfully parsed.

Auto loading file *name* ObjectScripting message

The file *name* is loading into memory automatically.

Writing file *name* ObjectScripting message

The file *name* is writing to disk.

File *name* failed to parse

ObjectScripting message

The file *name* has an error and failed to parse.

The displayed list contains no members

ObjectScripting message

You have defined a list and asked it to display, but the list is empty.

Created a new *name* ObjectScripting message

Successfully created a new object of type *name*.

Destroyed a *name*

ObjectScripting message

Successfully destroyed an object of type *name*.

Dispatch of object failed

ObjectScripting message

The object could not be dispatched to its destination.

Attempt to coerce non-integral string *string*

ObjectScripting message

Attempted to do a calculation with an array index that is a string instead of an integer.

Attempt to access non-existent element *number* ObjectScripting message

An attempt was made to access an element of an array that does not exist.

Attempt to access invalid object of type *name*

ObjectScripting message

You attempted to use an object of type *name* that does not exist.

The string *string* was not found in the menu

ObjectScripting message

The given *string* is not a menu item in the given menu

The string *string* was not found in the list ObjectScripting message

The given *string* is not an item in the given list

The *name* object is no longer valid ObjectScripting message

You attempted to use an object *name* that no longer exist.

Unable to locate project node for *projectName*

ObjectScripting message

You attempted to use a project node for *projectName* that does not exist.

An EditBuffer with a conflicting value for 'IsPrivate' already exists on *EditView*, argument 2 ignored ObjectScripting message

Once an *EditBuffer* has been created with *private* set to **FALSE** and is attached to an *EditView*, you cannot create an *EditBuffer* for the same *fileName* with *private* set to **TRUE**, and vice-versa.

An EditBuffer with a conflicting value for 'ReadOnly' already exists, argument 3 ignored ObjectScripting message

Once an *EditBuffer* has been created with *readOnly* set to **FALSE**, you cannot create an *EditBuffer* for the same *fileName* with *readOnly* set to **TRUE**, and vice-versa.

No windows currently exist ObjectScripting message

You attempted an operation involving open windows, but there are currently no windows open in the IDE.

Attempt to invoke unrecognized tool *toolName*

ObjectScripting message

You attempted to use a tool that is not installed.

No Editor windows are currently active

ObjectScripting message

You attempted an operation involving Editor windows, but there are currently no Editor windows open.

Attempt to modify non-modifiable keyboard

ObjectScripting message

You attempted to modify the operation of the keyboard, but the current keyboard definition cannot be modified.

The keyboard manager is not currently paused

ObjectScripting message

This operation requires that the keyboard playback be paused. Invoke the *KeyboardManager*.*PausePlayback*( ) method.

The keyboard manager is not currently playing back

ObjectScripting message

This operation requires that keyboard playback be activated. Invoke the *KeyboardManager*.*Playback*( ) method.

The keyboard manager has nothing to playback

ObjectScripting message

You must record something before you can play it back.

The keyboard manager cannot playback while it is recording message

ObjectScripting

You must stop recording before you can playback keystrokes.

The keyboard manager is not currently recording ObjectScripting message

This operation requires that the keyboard manager be in recording mode. Invoke the *KeyboardManager*.*StartRecord*( ) method.

Dispatch of member failed

ObjectScripting message

The member could not be dispatched to its destination.

The value *number* is out of range for property *propertyName* of class *className*

ObjectScripting message

The value that was assigned to property *propertyName* is not within the range of values the property can accept.

Attempt to access empty string from property *propertyName* of class *className*

ObjectScripting message

The property attempted to read the value of an empty string.

Attempt to write to the read-only property *propertyName* of class *className*

ObjectScripting message

You cannot write a value to a read-only property, only read its value.

SymbolLoad table was unable to resolve symbol *symbolName* from module *moduleName* ObjectScripting message

A symbol is used in the module *moduleName* that is not in cScript's symbol table.

Returning an unrecognized value for window style

ObjectScripting message

The window style being returned by the method is not one recognized by cScript.

The *toolName* tool encountered errors

ObjectScripting message

The tool *toolName* returned an error code.

The *toolName* tool encountered fatal errors

ObjectScripting message

The tool *toolName* returned a fatal error code.

The method *methodName* failed

ObjectScripting message

The method *methodName* returned an error code.

Unable to determine type of currently active window

ObjectScripting message

cScript is unable to determine which type of window is currently active.

A private buffer may not be attached to an EditView

ObjectScripting message

An *EditBuffer* created with the *private* parameter set to **TRUE** may not be attached to an *EditView* object.

Attempt to access empty string from method *methodName* of class *className*

ObjectScripting message

The method attempted to read the value of an empty string.

Dispatch of method failed ObjectScripting message

The method could not be dispatched to its destination.

The value *number* is out of range for method *methodName* of class *className*

ObjectScripting message

A value that was passed to method *methodName* is not within the range of values the method can accept.

The debugger method *methodName* failed ObjectScripting message

The debugger method *methodName* returned an error code.

Attempt to pop the last keyboard off the component's stack

ObjectScripting message

An attempt was made to unload all the keyboard definitions. This is not allowed.

The PushBack stack has been overrun

ObjectScripting message

The number of objects pushed onto the PushBack stack exceeds the stack's capacity.

Attempt to dispatch to unknown command id

ObjectScripting message

An attempt was made to dispatch a method to a command ID that is not known to cScript.

The region type ‘*regionName*’ is unrecognized ObjectScripting message

The *regionName* argument is not a region name recognized by cScript.

The desired coordinate for left is larger than desired coordinate for right

ObjectScripting message

The left side of a screen object must have a lower coordinate value than its right coordinate.

The desired coordinate for top is larger than desired coordinate for bottom

ObjectScripting message

The top coordinate of a screen object must have a lower value than its bottom coordinate.

The parameter EditView passed to Attach() is currently parented elsewhere

ObjectScripting message

An *EditView* cannot be attached to more than one *EditBuffer* at the same time.

Argument *number* to method *methodName* is not a valid *parameter* ObjectScripting message

The value that was passed to the method does not have the right type or value for the given argument.

Argument *number* to the *className* constructor is invalid

ObjectScripting message

The value that was passed to the constructor for this class does not have the right type or value for the given argument.

Argument *number* is NULL -- converted to integer ObjectScripting message

The **NULL** value that was passed to the method or function was converted to an integer (zero) to match the type the method or function expects.

Return value is NULL -- converted to integer

ObjectScripting message

The **NULL** value that was returned by the method or function was converted to an integer (zero).

The value *number* for argument *argNumber* is out of range for method *methodName* of class *className*

ObjectScripting message

A value that was passed to argument *argNumber* of method *methodName* is not within the range of values the argument can accept.

The component *name* has no registered keyboards

ObjectScripting message

The component must have at least one registered keyboard.

number is too narrow for a List, width adjusted to *newNumber* ObjectScripting message

The value given for the width of the list was too small. The value has been adjusted up to the minimum value.

number is too short for a List, height adjusted to *newNumber* ObjectScripting message

The value given for the height of the list was too small. The value has been adjusted up to the minimum value.

No initial elements were passed to the List constructor, the list starts empty

ObjectScripting message

An empty list was created since no values were passed to put in it.

The offset *number* is out of range for the menu

ObjectScripting message

The offset given for the menu selection is our of range of the values available in the menu.

The offset *number* is out of range for the list

ObjectScripting message

The offset given for the list selection is our of range of the values available in the list.

The argument passed to the EditPosition constructor was not an EditBuffer nor an EditView ObjectScripting message

EditPosition's constructor requires a valid *EditBuffer* or *EditView* as an argument.

User Break detected, script aborting ObjectScripting message

You pressed the Esc key, or another key, that caused script processing to abort.

Command parsing failed for: *command*

ObjectScripting message

The command *command* failed to parse.

watch definition

A watch monitors the state of a variable. Whenever your program pauses, the debugger evaluates all watched variables and updates the Watches window with their values.

transfer definition

The term used when another application is spawned from within the IDE. Command-line tools such as the WinHelp compiler or a DOS box (COMMAND.COM) are commonly invoked from the IDE during a build process.

SKIP_LEFT
A constant meaning skip to the left, towards the beginning of the line.

SKIP_NONSPECIAL
A constant meaning skip to next non-special character. That is, the next character which *is* either alphanumeric or whitespace.

SKIP_NONWHITE
A constant meaning skip to next non-whitespace character. That is, the next character which is *not* a space, tab, or newline character.

SKIP_NONWORD
A constant meaning skip to next non-alphanumeric character.

SKIP_RIGHT
A constant meaning skip to the right, towards the end of the line.

SKIP_SPECIAL
A constant meaning skip to next special character. That is, the next character which is *not* an alphanumeric or whitespace character.

SKIP_STREAM
A constant meaning ignore line ends when skipping. When used with **SKIP_WHITE**, and the cursor is at the beginning or end of a line, the cursor will continue to move until it reaches a non-white character, non-EOL character, BOF or EOF.

SKIP_WHITE
A constant meaning skip to next whitespace (space, tab, newline) character.

SKIP_WORD
A constant meaning skip to next alphanumeric character.

Edit rip definition

Edit rip is the process by which the editor detects the current cursor position and copies characters from the edit buffer. The IDE editor uses edit rip to find relevant Help topics when the user press F1 in an Edit window.

Translator

A translator is a program that creates one file type from another. For example, the C++ compiler is a translator that creates .OBJ files from .CPP files; the linker is a translator that creates .EXE files from .OBJ, .LIB, .DEF, and .RES files.

Search expression

A search expression is a set of characters that the search engine will attempt to match against the text contained in an edit buffer. A search expression can be either a literal string or a regular expression.

- In a literal string, there are no operators: Each character is treated literally.
- In a regular expression, certain characters have special meanings: They are operators that govern the search.

A regular expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.

Regular expressions have two formats:

- IDE
- Brief

Brief regular search expression symbols

The following table describes the symbols that can be used in a Brief search expression:

Symbol	Description
`?`	Any single character except a newline
`*`	Zero or more characters (except newlines)
`\t`	Tab character
`\n`	Newline character
`\c`	Position cursor after the match
`\\`	Literal backslash
`< or %`	Beginning of line
`> or $`	End of line
`@`	Zero or more of the last expression
`+`	One or more of the last expression
`|`	Either the last or the next expression
`{}`	Define a group of expressions
`[ ]`	Any one of the characters inside `[ ]`
`[~ ]`	Any character except those in `[~ ]`
`[a-z]`	Any character between a and z, inclusive

In replacement text, `\t`, `\n`, and `\c` are allowed, as well as

`\<n>`	Substitute text matched by `<n>`th group (0 <= n <= 9)

IDE search expression symbols

The following table describes the symbols that can be used in an IDE search expression:

Symbol	Description
`^`	A circumflex at the start of the expression matches the start of a line.
`$`	A dollar sign at the end of the expression matches the end of a line.
`.`	A period matches any single character.
`*`	An asterisk after a string matches any number of occurrences of that string followed by any characters, including zero characters. For example, `bo*` matches `bot`, `boo`, and `bo`.
`+`	A plus sign after a string matches any number of occurrences of that string followed by any characters, except zero characters. For example, `bo+` matches `bot` and `boo`, but not `b` or `bo`.
`{ }`	Characters or expressions in braces are grouped to allow for controlling evaluation of a search pattern or referring to grouped text by number.
`[ ]`	Characters in brackets match any single character that appears in the brackets, but no others. For example `[bot]` matches `b, o,` or `t.`
`[^]`	A circumflex at the start of the string in brackets means **NOT**. For example, `[^bot]` matches any characters except `b, o,` or `t.`
`[-]`	A hyphen in brackets signifies a range of characters. For example, `[b-o]` matches any character from `b` through `o.`
`\`	A backslash before a special character indicates that the character is to be interpreted literally. For example, `\^` matches `^` and does not indicate the start of a line.