

[Login](#) | [Register](#)

- [Visual C++ / C++ »](#)
 - [Sample Chapter](#)
 - [Security](#)
 - [C++ »](#)
 - [Algorithms & Formulas »](#)
 - [General](#)
 - [Checksum Algorithms](#)
 - [Combinations](#)
 - [Compression/Decompression](#)
 - [Factorials](#)
 - [Hash Tables](#)
 - [Linked Lists](#)
 - [Mathematics](#)
 - [Searching](#)
 - [Sorting](#)
 - [String Algorithms](#)
 - [Portability Issues](#)
 - [C++ & MFC »](#)
 - [General](#)
 - [Array Handling](#)
 - [Binary Trees](#)
 - [Bits and Bytes](#)
 - [Buffer & Memory Manipulation](#)
 - [Callbacks](#)
 - [Classes and Class Use](#)
 - [Collections](#)
 - [Compression](#)
 - [Drag and Drop](#)
 - [Events](#)
 - [Exceptions](#)
 - [External Links](#)
 - [File I/O](#)
 - [Function Calling](#)
 - [Linked Lists](#)
 - [Memory Tracking](#)
 - [Object Oriented Programming \(OOP\)](#)
 - [Open FAQ](#)
 - [Parsing](#)
 - [Patterns](#)
 - [Pointers](#)
 - [Portability](#)
 - [RTTI](#)
 - [Serialization](#)
 - [Singletons](#)

- [Standard Template Library \(STL\)](#)
 - [Templates](#)
 - [Tutorials](#)
- [Date & Time »](#)
 - [General](#)
 - [Date Controls](#)
 - [Time Routines](#)
- [C++/CLI »](#)
 - [.NET Framework Classes](#)
 - [General](#)
 - [ASP/ASP.NET](#)
 - [Boxing and UnBoxing](#)
 - [Components](#)
 - [Garbage Collection and Finalizers](#)
 - [Interop](#)
 - [Moving from Unmanaged](#)
 - [Processes & Threads](#)
 - [Templates](#)
 - [Visual Studio .NET 2003](#)
- [String Programming »](#)
 - [General](#)
 - [CString Alternatives](#)
 - [CString Extensions](#)
 - [CString Manipulation](#)
 - [Open FAQ](#)
 - [Regular Expressions](#)
 - [String Arrays](#)
 - [String Conversions](#)
 - [.NET](#)
- [COM-based Technologies »](#)
 - [ATL & WTL Programming »](#)
 - [General](#)
 - [ATL](#)
 - [Active Scripting](#)
 - [ActiveX Controls](#)
 - [Database](#)
 - [Debugging](#)
 - [External links](#)
 - [Graphics Support](#)
 - [Misc.](#)
 - [Performance](#)
 - [Printing](#)
 - [Tutorials](#)
 - [Utilities](#)
 - [Windows Template Library \(WTL\)](#)
 - [ActiveX Programming »](#)
 - [General](#)
 - [Active Scripting](#)
 - [ActiveX Controls](#)
 - [ActiveX Documents](#)
 - [Apartments & Threading](#)
 - [Error Handling](#)
 - [External links](#)
 - [General COM/DCOM](#)
 - [Misc.](#)
 - [Registry](#)
 - [Security](#)

- [Structured Storage](#)
 - [Tutorials](#)
 - [Wrappers](#)
- [COM+ »](#)
 - [General](#)
 - [COM Interop](#)
 - [Managed Code / .NET](#)
 - [SOAP and Web Services](#)
- [Shell Programming »](#)
 - [General](#)
 - [Open FAQ](#)
 - [Shortcuts](#)
 - [Tray Icons](#)
 - [Previous Section Manager](#)
- [Controls »](#)
 - [Property Sheet »](#)
 - [Open FAQ](#)
 - [Property Sheet Buttons](#)
 - [Sizing](#)
 - [Wizards](#)
 - [Button Control »](#)
 - [Advanced Buttons](#)
 - [Bitmap Buttons](#)
 - [Flat Buttons](#)
 - [Menus](#)
 - [Non-Rectangular buttons](#)
 - [Windows XP](#)
 - [ComboBox »](#)
 - [Colour Pickers](#)
 - [DropDown](#)
 - [Font selection combos](#)
 - [Multicolumn combos](#)
 - [Special Effects](#)
 - [Tooltips](#)
 - [Edit Control »](#)
 - [Background & Color](#)
 - [Editors](#)
 - [Keyboard](#)
 - [Masked Edit Controls](#)
 - [Passwords and Security](#)
 - [Spin Controls](#)
 - [Transparent](#)
 - [ImageList Control »](#)
 - [Open FAQ](#)
 - [ListBox Control »](#)
 - [Checkboxes](#)
 - [Color Listboxes](#)
 - [Drag & Drop](#)
 - [LEDs](#)
 - [ListView Control »](#)
 - [Advanced](#)
 - [Background color and image](#)
 - [Checkboxes](#)
 - [Columns](#)
 - [Custom Drawing](#)
 - [Data](#)
 - [Deleting](#)

- [Drag & Drop](#)
- [Editing items and subitem](#)
- [FilterBar](#)
- [Grid lines](#)
- [Header Control](#)
- [Introduction](#)
- [Miscellaneous](#)
- [Navigation](#)
- [New ListView control \(IE 4.0\)](#)
- [Printing](#)
- [Property Lists](#)
- [Reports](#)
- [Scrollbars](#)
- [Selection](#)
- [Sorting](#)
- [Tooltip & Titletip](#)
- [Using images](#)
- [Views](#)
- [Menu »](#)
 - [Alternative menu](#)
 - [Bitmapped menus](#)
 - [Dockable menus](#)
 - [Message and Command Routing](#)
 - [Miscellaneous](#)
 - [XML](#)
 - [XP-Style Menus](#)
- [Other Controls »](#)
 - [Bitmap Buttons](#)
 - [Charting and analogue controls](#)
 - [Check Box Controls](#)
 - [Clocks & Timers](#)
 - [Cool Controls](#)
 - [Date Selection Controls etc.](#)
 - [Digital Controls](#)
 - [Extending/Subclassing Techniques](#)
 - [File and Directory choosers](#)
 - [Grid Control](#)
 - [Group Box](#)
 - [HyperLink controls](#)
 - [Internet & Web Related](#)
 - [Lists, Trees and Combos](#)
 - [Minimize Button](#)
 - [Progress Controls](#)
 - [Resizing](#)
 - [Scroll Bars](#)
 - [Slider Controls](#)
 - [Spin Control](#)
 - [System Tray](#)
 - [Tab Controls](#)
 - [Tooltip controls](#)
 - [Charting and analogue controls](#)
 - [Extending/Subclassing Techniques](#)
- [Rich Edit Control »](#)
 - [Conversions](#)
 - [Editors and Editing](#)
 - [Syntax Hilighting](#)
 - [Windowless](#)

- [Static Control »](#)
 - [Bevel lines](#)
 - [Data display controls](#)
 - [Scrolling Text](#)
- [Status Bar »](#)
 - [Advanced](#)
 - [System Tray](#)
- [Toolbar »](#)
 - [Customizing Toolbars](#)
 - [Docking](#)
 - [Flat Toolbar](#)
 - [Miscellaneous](#)
 - [Placing Controls in Toolbars](#)
 - [Placing Controls in Toolbars](#)
- [Treeview Control »](#)
 - [Classes](#)
 - [Directory Browsers](#)
 - [Drag & Drop](#)
 - [Editing labels](#)
 - [Expand - Collapse](#)
 - [Misc - Advanced](#)
 - [Multiview](#)
 - [New Listview control \(IE 4.0\)](#)
 - [Searching](#)
 - [Tree traversal](#)
 - [Using images](#)
- [Data »](#)
 - [Database »](#)
 - [ADO](#)
 - [ADO.NET](#)
 - [ATL](#)
 - [DAO](#)
 - [Dynamic Data Access](#)
 - [Microsoft Access](#)
 - [Microsoft Excel](#)
 - [Misc.](#)
 - [Object Oriented](#)
 - [ODBC](#)
 - [OLE DB](#)
 - [Oracle](#)
 - [SQL Server](#)
 - [Stored Procedures](#)
 - [XML](#)
 - [Miscellaneous »](#)
 - [File Information](#)
 - [INI Files](#)
 - [Values](#)
 - [XML](#)
- [Frameworks »](#)
 - [UI & Printing Frameworks »](#)
 - [Component Libraries](#)
 - [Outlook Controls](#)
 - [Reporting and Report Writing](#)
 - [Skins](#)
 - [Reporting and Report Writing](#)
 - [Windowing Techniques and Classes](#)
- [Graphics & Multimedia »](#)

- [Bitmaps & Palettes »](#)
 - [Capturing](#)
 - [Compressing](#)
 - [Displaying and Sizing](#)
 - [External Links](#)
 - [Icons](#)
 - [Image Manipulation](#)
 - [Merging](#)
 - [Other formats...](#)
 - [Palettes and Color Tables](#)
 - [Special Effects](#)
 - [Using regions](#)
 - [Viewers](#)
 - [Views & MDI clients](#)
- [DirectX »](#)
 - [DirectDraw](#)
 - [DirectInput](#)
 - [DirectShow](#)
 - [DirectX 8](#)
- [GDI »](#)
 - [Capturing Images](#)
 - [Fills](#)
 - [Font Handling and Detection](#)
 - [GDI+](#)
 - [Icons and Cursors](#)
 - [Lines](#)
- [Multimedia »](#)
 - [Audio](#)
 - [Desktop Effects](#)
 - [Graphics](#)
 - [Imaging](#)
 - [Timers](#)
 - [Twain](#)
 - [Video](#)
- [OpenGL »](#)
 - [Game Programming](#)
 - [Printing](#)
 - [Texture Mapping](#)
- [Internet & Networking »](#)
 - [Internet Protocols »](#)
 - [ActiveX](#)
 - [Browser Control](#)
 - [Chat Programs](#)
 - [DHTML](#)
 - [Dial-up](#)
 - [DNS](#)
 - [Email](#)
 - [File Transfer](#)
 - [FTP](#)
 - [General Internet](#)
 - [HTML](#)
 - [HTTP](#)
 - [Instant Messaging](#)
 - [Internet Protocol \(IP\)](#)
 - [Network](#)
 - [Secure Socket Layer \(SSL\)](#)
 - [Security](#)

- [Streaming Media](#)
- [Web Services](#)
- [XML](#)
- [IE Programming »](#)
 - [Displaying Information](#)
 - [Security](#)
 - [Voice](#)
- [ISAPI »](#)
 - [Cookies](#)
 - [Data/Databases](#)
 - [Extensions](#)
 - [Filters](#)
 - [Related CODEGURU sections](#)
- [Network Protocols »](#)
 - [Active Directory](#)
 - [Basic Network Operations](#)
 - [Dial-up Networking](#)
 - [Games](#)
 - [IPX](#)
 - [Messaging](#)
 - [Named Pipes](#)
 - [Network Information](#)
 - [Remote Administration](#)
 - [Remote Invocation](#)
 - [Serial Communications](#)
 - [TCP/IP](#)
 - [Winsock Solutions](#)
- [Miscellaneous »](#)
 - [Miscellaneous »](#)
 - [Application Control](#)
 - [Assembly Language](#)
 - [CD-ROM](#)
 - [Compiler and Pre-Compiler](#)
 - [Console Apps](#)
 - [CryptoAPI](#)
 - [Drag and Drop](#)
 - [File and Directory Naming](#)
 - [File I/O](#)
 - [Flicker Free Drawing](#)
 - [Fonts](#)
 - [Graphics](#)
 - [ImageList](#)
 - [Intellisense](#)
 - [Interfacing to other languages](#)
 - [Internet Explorer](#)
 - [IPC Techniques](#)
 - [Keyboard](#)
 - [Log and Trace Files](#)
 - [MAPI](#)
 - [Math](#)
 - [Memory](#)
 - [Message Box Handling](#)
 - [MFC Help](#)
 - [Microsoft Office / Outlook](#)
 - [Microsoft Shell \(MSH\)](#)
 - [Mouse and Cursor Handling](#)
 - [MSN](#)

- [Multi-Lingual Support](#)
- [Plug-ins / Add-ins](#)
- [Power Management API](#)
- [Screen Savers](#)
- [Splash Screens](#)
- [Stack](#)
- [System](#)
- [System Tray](#)
- [Task Scheduler](#)
- [Templatized Classes](#)
- [Threads/Processes](#)
- [Timers](#)
- [Titlebar](#)
- [Tools](#)
- [UML](#)
- [Version Info](#)
- [Video](#)
- [Visual InterDev](#)
- [Windows Messaging](#)
- [Samples »](#)
 - [Basic Programming](#)
 - [Code Highlighting](#)
 - [Games](#)
 - [System Information](#)
 - [Testing & Debugging](#)
- [Visual Studio »](#)
 - [Debugging »](#)
 - [Debuggers](#)
 - [Handling Crashes](#)
 - [Logging](#)
 - [Memory Issues](#)
 - [Reverse Engineering](#)
 - [Tracing](#)
 - [Add-ins & Macros »](#)
 - [Add-Ins](#)
 - [Bookmarks](#)
 - [Browser Windows](#)
 - [Code Generation](#)
 - [Code Navigation](#)
 - [Code Reformatting](#)
 - [Code Template Add-in](#)
 - [Comment Creators](#)
 - [Custom AppWizards](#)
 - [Debugging](#)
 - [File Creation](#)
 - [File Opening](#)
 - [Text Operations](#)
 - [Version Control](#)
 - [Visual Studio .NET](#)
 - [Window Management](#)
 - [Editor Tips »](#)
 - [Customizing the IDE](#)
 - [Debugging](#)
 - [Syntax Highlighting](#)
- [Windows Programming »](#)
 - [Vista](#)
 - [CE »](#)

- [Bitmaps and the GDI](#)
- [COM](#)
- [Controls](#)
- [Database](#)
- [Embedded Visual C++](#)
- [Files](#)
- [Game Programming](#)
- [Memory Management](#)
- [Networking](#)
- [Pocket PC](#)
- [Registry](#)
- [Samples](#)
- [Shell and Related Programming](#)
- [SmartPhone](#)
- [Clipboard »](#)
 - [External Links](#)
 - [Previous Section Manager](#)
- [DLL »](#)
 - [Hooking](#)
 - [Import/Export issues](#)
 - [Interfacing to Visual Basic applications](#)
 - [Mixed DLLs](#)
 - [Tips](#)
 - [Interfacing to Visual Basic applications](#)
- [File & Folder »](#)
 - [Browser Functions & Dialogs](#)
 - [Controls & Dialogs](#)
 - [File Information](#)
 - [File I/O](#)
 - [Folder/Directory Maintenance](#)
 - [INI files](#)
 - [Installers](#)
 - [Shell API Functions](#)
- [Help Systems »](#)
 - [HTML](#)
 - [MSDN](#)
- [Printing »](#)
 - [Open FAQ](#)
 - [Print Preview](#)
- [Win32 »](#)
 - [Cursors](#)
 - [MessageBox](#)
 - [Security](#)
 - [Tutorials](#)
 - [Versioning](#)
- [System »](#)
 - [CD-ROM](#)
 - [Control Panel](#)
 - [Device Driver Development](#)
 - [Hardware Information](#)
 - [Keyboard](#)
 - [Logging](#)
 - [Message Handling](#)
 - [Misc](#)
 - [MMC Snapins](#)
 - [NT Services](#)
 - [Performance Statistics](#)

- [Processes / Modules](#)
- [Registry](#)
- [Resource Detection/Management](#)
- [Security](#)
- [Services](#)
- [Shared Memory](#)
- [System Information](#)
- [Taskbar](#)
- [Task Manager](#)
- [Threading](#)
- [Timers](#)
- [VxD](#)
- [Windows & Dialogs »](#)
 - [Console »](#)
 - [Redirection](#)
 - [Dialog »](#)
 - [Animation](#)
 - [Background](#)
 - [Bitmaps/Images](#)
 - [Browse Namespace](#)
 - [Colors](#)
 - [CommandUI](#)
 - [Common Dialogs](#)
 - [Credits Dialog](#)
 - [DDX/DDV](#)
 - [Dialog-based applications](#)
 - [Dialog Bars](#)
 - [Dialog for selecting folders](#)
 - [Dialog Layout Manager](#)
 - [Dialog Template Resources](#)
 - [Displaying in a window](#)
 - [Display Performance](#)
 - [Font Handling](#)
 - [Help](#)
 - [Image Preview](#)
 - [Menu Handling](#)
 - [Miscellaneous](#)
 - [Modeless Dialogs](#)
 - [Non-Rectangular Dialogs](#)
 - [Options Dialogs](#)
 - [Progress Indicators](#)
 - [Resizable Dialogs](#)
 - [Scrolling](#)
 - [Splash Screens](#)
 - [Splitter Windows withing Dialogs](#)
 - [Threads](#)
 - [Tip of the Day](#)
 - [Title Bar](#)
 - [Toolbars and Statusbars](#)
 - [Tooltips for Dialog Controls](#)
 - [Tutorials](#)
 - [Win32](#)
 - [Wizards](#)
 - [Splitter Windows withing Dialogs](#)
 - [Docking Window »](#)
 - [Open FAQ](#)
 - [Doc/View »](#)

- [Background](#)
 - [CHtmlView](#)
 - [Command Routing](#)
 - [Control Views](#)
 - [DDX/DDV](#)
 - [File Open/Save](#)
 - [Help](#)
 - [Message Maps](#)
 - [Misc](#)
 - [MRU \(Most Recent Used file list\)](#)
 - [Printing](#)
 - [Scrolling](#)
 - [SDI](#)
 - [Splitter Views](#)
 - [Tabs](#)
 - [Title Bar](#)
 - [View Management](#)
 - [Window Positions/Sizes](#)
- [Splitter »](#)
 - [Open FAQ](#)
 - [Sizing](#)
 - [Toolbars](#)
 - [Tutorials](#)
- [Standard Template Library](#)
- [.NET / C# »](#)
 - [Sample Chapter](#)
 - [.NET »](#)
 - [Silverlight](#)
 - [WCF](#)
 - [WPF](#)
 - [Windows Workflow](#)
 - [Data & Databases »](#)
 - [DataGrid](#)
 - [Sorting and Iterating](#)
 - [Web Grid](#)
 - [Debugging »](#)
 - [Debugging](#)
 - [Logging](#)
 - [State Management](#)
 - [Techniques](#)
 - [Tracing](#)
 - [Framework »](#)
 - [Licensing](#)
 - [Microsoft Namespace](#)
 - [System Namespace](#)
 - [General »](#)
 - [Arithmetic](#)
 - [Assemblies](#)
 - [Code-Behind](#)
 - [COM/COM+](#)
 - [Common Type System \(CTS\)](#)
 - [Debugging and Error Handling](#)
 - [Events and Delegates](#)
 - [Graphics](#)
 - [IL](#)
 - [Internet](#)
 - [Keyboard](#)

- [Macros](#)
- [.NET Framework Classes](#)
- [.NET My Services](#)
- [Patterns](#)
- [Performance](#)
- [Rotor](#)
- [System Information](#)
- [Threads](#)
- [Tips & Tricks](#)
- [Tools and 3rd Party](#)
- [Visual Studio .NET Add-Ins](#)
- [Debugging and Error Handling](#)
- [JScript .NET »](#)
 - [Essam Ahmed on JScript .NET](#)
- [Managed C++ »](#)
 - [Windows Services](#)
- [Net Security »](#)
 - [Authentication](#)
 - [Encryption](#)
 - [PGP](#)
- [VS Add-Ins »](#)
 - [UML and Design](#)
 - [Visual Studio Add-ins](#)
 - [Visual Studio .NET 2003 Add-ins](#)
- [General ASP.NET »](#)
 - [Controls](#)
 - [Email](#)
 - [Files](#)
 - [Miscellaneous](#)
 - [Scripting](#)
 - [Themes](#)
 - [Tutorials](#)
 - [Web Forms](#)
- [Azure »](#)
- [C# »](#)
 - [.NET 3.0 / .NET 3.5](#)
 - [LINQ](#)
 - [Basic Syntax »](#)
 - [Anand C# Tutorials](#)
 - [Attributes](#)
 - [Component Development](#)
 - [Controls](#)
 - [Enumerations](#)
 - [Error Handling](#)
 - [Indexers](#)
 - [Inheritance](#)
 - [Interfaces](#)
 - [Operators](#)
 - [Reflection](#)
 - [Remoting](#)
 - [Serialization](#)
 - [Threading](#)
 - [Unmanaged Code](#)
 - [Unsafe](#)
 - [Collections](#)
 - [Controls »](#)
 - [Custom](#)

- [Data Grid](#)
 - [Property Grid](#)
 - [Rich Text](#)
 - [Tree View](#)
 - [Tutorials](#)
 - [Wizards](#)
- [Data & I/O »](#)
 - [Caching and Performance](#)
 - [Searching](#)
 - [Streaming](#)
 - [Tutorials](#)
 - [XML](#)
- [Date & Time »](#)
 - [Formatting](#)
 - [Time Routines](#)
- [Delegates »](#)
 - [Event Handling](#)
- [Graphics & Multimedia »](#)
 - [Charts and Graphing](#)
 - [Customizing User Interfaces](#)
 - [Drawing](#)
 - [Mouse](#)
 - [Screen Captures](#)
 - [Sound](#)
 - [Threading](#)
 - [Wav Files](#)
 - [Customizing User Interfaces](#)
- [Internet »](#)
 - [Desktop Applications](#)
 - [Mail](#)
- [Miscellaneous »](#)
 - [COM](#)
 - [Design/Techniques](#)
 - [DLLs and Executables](#)
 - [E-Mail](#)
 - [Graphics and Images](#)
 - [Icons](#)
 - [Mathematics](#)
 - [Reflection](#)
 - [Sample Programs](#)
 - [Security](#)
 - [User Interface](#)
- [Network & Systems »](#)
 - [Configuration Files / INIs](#)
 - [Database](#)
 - [Directory Services](#)
 - [HTTP](#)
 - [Internet/Web](#)
 - [IP](#)
 - [Remoting](#)
 - [Sockets](#)
 - [Windows Services](#)
 - [WMI](#)
- [Web Services »](#)
 - [Security](#)
 - [Tutorials](#)
- [Windows 8/RT »](#)

- [XAML »](#)
- [Visual Basic »](#)
 - [Sample Chapter](#)
 - [.NET 3.0 \(VB\)](#)
 - [LINQ \(VB\)](#)
 - [Internet »](#)
 - [ASP .NET](#)
 - [Database](#)
 - [HTML](#)
 - [SMTP / eMail](#)
 - [Web Services](#)
 - [Indexing](#)
 - [VB Controls »](#)
 - [VB Other Controls »](#)
 - [OCX Controls](#)
 - [VB ListView](#)
 - [VB ListBox](#)
 - [VB ComboBox](#)
 - [VB ActiveX »](#)
 - [ActiveX](#)
 - [.NET Controls »](#)
 - [DataGrid Control](#)
 - [Listview](#)
 - [VB Files »](#)
 - [Directory](#)
 - [Drive](#)
 - [VB Shell](#)
 - [General »](#)
 - [VB Multimedia »](#)
 - [MP3s](#)
 - [VB Graphics »](#)
 - [Animation](#)
 - [File Formats](#)
 - [GDI](#)
 - [Transparency](#)
 - [Database »](#)
 - [ADO](#)
 - [ADO.NET](#)
 - [Database Access](#)
 - [DataSets](#)
 - [Microsoft Access](#)
 - [SQL Server](#)
 - [Stored Procedures](#)
 - [Strongly Typed Data](#)
 - [DataGrid](#)
 - [Forms & Controls »](#)
 - [Backgrounds](#)
 - [Icons](#)
 - [ListBox](#)
 - [ListView Controls](#)
 - [Resizing](#)
 - [IDE & Language »](#)
 - [Attributes](#)
 - [Error Handling](#)
 - [IDE \(including Visual Studio .NET\)](#)
 - [IDE & Language](#)
 - [Indexers](#)
















- [Miscellaneous](#)
 - [IDE \(including Visual Studio .NET\)](#)
- [Misc »](#)
 - [Algorithms](#)
 - [Console Applications](#)
 - [Date](#)
 - [Debugging and Tracing](#)
 - [Encryption](#)
 - [Games and Fun](#)
 - [Icons](#)
 - [Multi-Language Support](#)
 - [Numbers](#)
 - [OOP](#)
 - [PDF](#)
 - [Plugins](#)
 - [Printing](#)
 - [Samples](#)
 - [Text](#)
 - [Tips](#)
 - [User Defined Types \(UDTs\)](#)
- [System »](#)
 - [Directories and Files](#)
 - [Keyboard](#)
 - [Network](#)
 - [Services](#)
 - [Win32](#)
- [Mobile/Wireless »](#)
 - [Mobile Internet Toolkit](#)
 - [Pocket PC](#)
- [Windows 8/RT »](#)
- [Others »](#)
 - [Windows Mobile Dev Center](#)
 - [Internet of Things \(IoT\)](#)
 - [CodeGuru Blogs](#)
 - [Codeguru Live Chats](#)
 - [Azure Activities](#)
 - [Videos](#)
 - [Whitepapers](#)
 - [Slideshows](#)
- [Forums](#)
- [Submit an Article](#)
- [Newsletter](#)

Stay up-to-date with our free Microsoft Tech Update Newsletter

Subscribe
- [Close](#)
- [Slideshows](#)
- [Sponsored](#)

[codeguru](#)
[Visual C++ / C++](#)
[C++](#)
[C++ & MFC](#)
[Standard Template Library \(STL\)](#)

[Read More in Standard Template Library \(STL\) »](#)

- [Post a comment](#)
- [Email Article](#)
- [Print Article](#)
- Share Articles
 -  [Digg](#)
 -  [del.icio.us](#)
 -  [Newsvine](#)
 -  [Facebook](#)
 -  [Google](#)
 -  [LinkedIn](#)
 -  [MySpace](#)
 -  [Reddit](#)
 -  [Slashdot](#)
 -  [StumbleUpon](#)
 -  [Technorati](#)
 -  [Twitter](#)
 -  [Windows Live](#)
 -  [Yahoo Buzz](#)
 -  [FriendFeed](#)
-
- [Page 1](#)
- [Page 2](#)
- [Page 3](#)
- [Page 4](#)

A TR1 Tutorial: Smart Pointers

- 0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50
4.75
5.00

Posted by [Marius Bancila](#) on **July 16th, 2008**

-



WEBINAR: On-demand webcast How to Boost Database Development Productivity on Linux, Docker, and Kubernetes with Microsoft SQL Server 2017 **REGISTER >**

Until TR1, the only smart pointer available in the standard library was `auto_ptr`; that presents some major disadvantages because of its exclusive ownership model. To address these issues, the new standard offers two new implementations, `shared_ptr` and `weak_ptr`, both defined in header `<memory>`.

Class `std::auto_ptr`

As mentioned earlier, `auto_ptr` is based on an exclusive ownership model; each means that two pointers (of this type) cannot point to the same resource. Copying or assigning makes the resource changing its owner, with the source giving the ownership to the destination.

```

1. #include <memory>
2. #include <iostream>
3.
4. class foo
5. {
6. public:
7.     void print() {std::cout << "foo::print" << std::endl;}
8. };
9.
10. int main()
11. {
12.     std::auto_ptr<foo> ap1(new foo);
13.     ap1->print();
14.     std::cout << "ap1 pointer: " << ap1.get() << std::endl;
15.
16.     std::auto_ptr<foo> ap2(ap1);
17.     ap2->print();
18.     std::cout << "ap1 pointer: " << ap1.get() << std::endl;
19.     std::cout << "ap2 pointer: " << ap2.get() << std::endl;
20.
21.     return 0;
22. }
```

The output is:

```

1. foo::print
2. ap1 pointer: 0033A790
3. foo::print
4. ap1 pointer: 00000000
5. ap2 pointer: 0033A790
```

The exact value of the wrapped pointer (0033A790) is not important. The issue here is that, after creating and initializing object `ap2`, `ap1` gave up the ownership of the resource, and its wrapper pointer became `NULL`.

The major problems introduced by `auto_ptr` are:

- Copying and assigning changes the owner of a resource, modifying not only the destination but also the source, which is not intuitive.
- It cannot be used in STL containers because the constraint that a container's elements must be copy constructable and assignable does not apply to this class.

What's New in TR1?

Two new smart pointers were added to the standard template library:

- **shared_ptr**: Based on a reference counter model, with the counter incremented each time a new shared pointer object points to the resource, and decremented when the object's destructor executes; when the counter gets to 0, the resource is released. This pointer is copy constructable and assignable; this makes it usable in STL containers. Moreover, the shared pointer works with polymorphic types and incomplete types. Its major drawback is the impossibility to detect cyclic dependencies, in which case the resources never get released (for example, a tree with nodes having (shared) pointers to children but also to the parent, in which case the parent and the children are referencing each other, in a cycle). To fix this issue, a second smart pointer was created:
- **weak_ptr**: Points to a resource referred by a shared pointer, but does not participate in reference counting. When the counter gets to 0, the resource is released, regardless the number of weak pointers referring it; all these pointers are marked as invalid.

The next example shows a similar implementation to the first example, replacing `auto_ptr` with `shared_ptr`.

```

1. int main()
2. {
3.     std::tr1::shared_ptr<foo> sp1(new foo);
4.     sp1->print();
5.     std::cout << "sp1 pointer: " << sp1.get() << std::endl;
6.
7.     std::tr1::shared_ptr<foo> sp2(sp1);
8.     sp2->print();
9.     std::cout << "sp1 pointer: " << sp1.get() << std::endl;
10.    std::cout << "sp2 pointer: " << sp2.get() << std::endl;
11.
12.    std::cout << "counter sp1: " << sp1.use_count() << std::endl;
13.    std::cout << "counter sp2: " << sp2.use_count() << std::endl;
14.
15.    return 0;
16. }

1. foo::print

```

```

2. sp1 pointer: 0033A790
3. foo::print
4. sp1 pointer: 0033A790
5. sp2 pointer: 0033A790
6. counter sp1: 2
7. counter sp2: 2

```

As you can see, when sp2 is created, sp1 does not give up the ownership, changing its wrapped pointer to NULL; it only increments the reference counter. When the two shared pointer objects get out of scope, the last one that is destroyed will release the resource.

A TR1 Tutorial: Smart Pointers

- 0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
4.25
4.50
4.75
5.00

Posted by [Marius Bancila](#) on **July 16th, 2008**

•

-
- 
-

WEBINAR: On-demand webcast How to Boost Database Development Productivity on Linux, Docker, and Kubernetes with Microsoft SQL Server 2017 **REGISTER >**

Class `std::tr1::shared_ptr`

A `shared_ptr` object has the ownership of an object if:

- It was constructed with a pointer to that resource
- It was constructed from a `shared_ptr` object that owns that resource

- It was constructed from a `weak_ptr` object that points to that resource
- Ownership of that resource was assigned to it, either with `shared_ptr::operator=` or by calling the member function `shared_ptr::reset()`.

All the shared pointer objects that share the ownership of the same resource also shared a control block, containing the number of `shared_ptr` objects that own the resource, the number of `weak_ptr` objects that point to the resource, and the deleter (a function used to release the resource), if it has one. An empty `shared_ptr` object does not own any resources and has no control block. On the other hand, a `shared_ptr` that was initialized with a NULL pointer has a control block; this means it is not an empty shared pointer. When the reference counter to a resource becomes 0 (regardless the number of weak pointer still referring the object), the resource is released, either by deleting it or by passing its address to the deleter.

Creating a shared_ptr



Data Center Expansion

[Download Now](#)

There are several constructors available for a `shared_ptr`:

```
1. shared_ptr();
2. template<class Other>
3.     explicit shared_ptr(Other*);
4. template<class Other, class D>
5.     shared_ptr(Other*, D);
6. shared_ptr(const shared_ptr&);
7. template<class Other>
8.     shared_ptr(const shared_ptr<Other>&);
9. template<class Other>
10.    shared_ptr(const weak_ptr<Other>&);
11. template<class Other>
12.    shared_ptr(const std::auto_ptr<Other>&);
```

You basically can create a new `shared_ptr` from:

- A pointer to any type T (including const T), having the possibility of specifying a deleter for the pointed resource
- Another `shared_ptr` object
- A `weak_ptr` object
- An `auto_ptr` object

The next sample shows a `shared_ptr` created from an `auto_ptr` object. The auto pointer gives up the ownership of the resource, resetting its wrapped pointer to NULL.

```
1. int main()
2. {
3.     std::auto_ptr<foo> ap1(new foo);
4.     ap1->print();
5.     std::cout << "ap1 pointer: " << ap1.get() << std::endl;
6.
7.     std::tr1::shared_ptr<foo> sp1(ap1);
8.     sp1->print();
9.     std::cout << "ap1 pointer: " << ap1.get() << std::endl;
10.    std::cout << "sp1 pointer: " << sp1.get() << std::endl;
11.
12.    return 0;
13. }
```

The output is:

```
1. foo::print
2. ap1 pointer: 0033A790
3. foo::print
4. ap1 pointer: 00000000
```

5. spl pointer: 0033A790

I was saying earlier that, when a `shared_ptr` object is created, you can specify a special function called `deleter`, used to release the resource. If no such function is provided, the resource is simply deleted by calling `operator delete`.

Consider, for instance, a case when the creation and deletion of a resource should be logged somewhere. For class `foo` defined at the beginning at the article I created a helper class, that creates and destroys instances, but also logs these events.

```

1. class foo_handler
2. {
3. public:
4.     static foo* alloc()
5.     {
6.         foo* f = new foo;
7.         ::OutputDebugString(_T("a new foo was created\n"));
8.         return f;
9.     }
10.
11.     static void free(foo* f)
12.     {
13.         delete f;
14.         ::OutputDebugString(_T("foo destroyed\n"));
15.     }
16. };

```

Each time a new object is created or destroyed, a message is printed in the output window (for simplicity, you will ignore the copy construction or assignment). Function `foo_handler::free` can be provided as a `delete` to the `shared_ptr` constructor. As a result, when the resource is deleted a message is printed in the output window (you have to run in debugger to see it).

```

1. int main()
2. {
3.     std::tr1::shared_ptr<foo> ptr(
4.         foo_handler::alloc(),
5.         &foo_handler::free);
6.
7.     ptr->print();
8.
9.     return 0;
10. }

```

Running in debugger and looking into the output window, you can see:

```

1. a new foo was created
2. foo destroyed

```

Function `get_deleter` from header `<memory>` returns a pointer to the `deleter` of a `shared_ptr`, if one was provided, or 0 otherwise. The next sample shows how to get the `deleter` of the shared pointer created earlier.

```

1. typedef void (*deleter)(foo*);
2.
3. deleter* del = std::tr1::get_deleter<deleter>(ptr);
4.
5. std::cout << "get_deleter(ptr) != 0 == " << std::boolalpha
6.     << (del != 0) << std::endl;

```

The output is:

```

1. get_deleter(ptr) != 0 == true

```

Operators -> and * and function get

Class `shared_ptr` overloads operators `->` and `*`, the first returning a pointer to the resource and the second a reference to the value of the resource, so that accessing the internal wrapped pointer is not necessary.

```

1. template<class Ty>
2.     class shared_ptr {
3. public:
4.     Ty *get() const;
5.     Ty& operator*() const;
6.     Ty *operator->() const;
7. };

```

Function `get()` returns the wrapped pointer to the resource (basically identical to `operator->` and available for compatibility with `auto_ptr`).

```

1.     std::tr1::shared_ptr<foo> sp(new foo);
2.     foo* f = sp.get();
3.
4.     if(f) f->print();

```

Conditional operator

Class `shared_ptr` defines a `bool` operator that allows shared pointers to be used in boolean expressions. With `auto_ptr`, that is not possible; you have to use function `get()` to access the internal pointer and check it against `NULL`.

```

1. void is_empty(std::tr1::shared_ptr<std::string> ptr)
2. {
3.     if(ptr)
4.     {
5.         std::cout << "not empty" << std::endl;
6.     }
7.     else
8.     {
9.         std::cout << "is empty" << std::endl;
10.    }
11. }
12.
13. int main()
14. {
15.     std::tr1::shared_ptr<std::string> sp1;
16.     std::tr1::shared_ptr<std::string> sp2(new std::string("demo"));
17.
18.     is_empty(sp1);
19.     is_empty(sp2);
20.
21.     return 0;
22. }

```

The output is:

```

1. is empty
2. not empty

```

Swap and assignment

Method `swap()` and the function with the same name from header `<memory>` exchange the content of the shared pointers.

```

1. int main()
2. {
3.     std::tr1::shared_ptr<std::string> sp1;
4.     std::tr1::shared_ptr<std::string> sp2(new std::string("demo"));
5.
6.     is_empty(sp1);
7.     is_empty(sp2);
8.

```

```

9.     sp1.swap(sp2);
10.
11.     is_empty(sp1);
12.     is_empty(sp2);
13.
14.     return 0;
15. }

```

The output is:

```

1. is empty
2. not empty
3.
4. not empty
5. is empty

```

On the other hand, `operator=` is overloaded so that a shared pointer can be assigned from another `shared_ptr` OR `auto_ptr`.

```

1. template<class Ty>
2.     class shared_ptr
3.     {
4.     public:
5.         shared_ptr& operator=(const shared_ptr&);
6.         template<class Other>
7.             shared_ptr& operator=(const shared_ptr<Other>&);
8.         template<class Other>
9.             shared_ptr& operator=(auto_ptr<Other>&);
10.    }

```

The next sample shows an example of using `operator=`.

```

1. int main()
2. {
3.     std::tr1::shared_ptr<int> sp1(new int(1));
4.     std::cout << "sp1 = " << *sp1 << std::endl;
5.
6.     std::tr1::shared_ptr<int> sp2(new int(2));
7.     std::cout << "sp2 = " << *sp2 << std::endl;
8.
9.     sp1 = sp2;
10.    std::cout << "sp1 = " << *sp1 << std::endl;
11.
12.    return 0;
13. }

```

Methods `unique` and `use_count`

Method `use_count()` returns the number of references to the shared resource (pointed by the current shared pointer object). Method `unique()` indicates whether another shared pointer shares the ownership of the same resource or not (basically, it's identical to `1 == use_count()`).

```

1. int main()
2. {
3.     std::tr1::shared_ptr<std::string>
4.         sp1(new std::string("maris bancila"));
5.
6.     std::cout << "unique : " << std::boolalpha << sp1.unique()
7.         << std::endl;
8.     std::cout << "counter : " << sp1.use_count() << std::endl;
9.
10.    std::tr1::shared_ptr<std::string> sp2(sp1);
11.
12.    std::cout << "unique : " << std::boolalpha << sp1.unique()
13.        << std::endl;
14.    std::cout << "counter : " << sp1.use_count() << std::endl;
15.
16.    return 0;

```

```
17. }
```

The output is:

```
1. unique : true
2. counter : 1
3.
4. unique : false
5. counter : 2
```

Resetting

Function `reset()` decrements the shared reference counter. It then transforms the shared pointer to an empty `shared_ptr`.

```
1. int main()
2. {
3.     // a shared_ptr owns the resource, counter is 1
4.     std::tr1::shared_ptr<foo> sp1(new foo);
5.     std::cout << "counter sp1: " << sp1.use_count() << std::endl;
6.
7.     // a second shared_ptr owns the resource, shared counter is 2
8.     std::tr1::shared_ptr<foo> sp2(sp1);
9.     std::cout << "counter sp1: " << sp1.use_count() << std::endl;
10.    std::cout << "counter sp2: " << sp2.use_count() << std::endl;
11.
12.    // first shared_ptr is reset, the counter decremented
13.    // and the object becomes empty (no control block anymore)
14.    sp1.reset();
15.    std::cout << "counter sp1: " << sp1.use_count() << std::endl;
16.    std::cout << "counter sp2: " << sp2.use_count() << std::endl;
17.
18.    return 0;
19. }
```

The output is:

```
1. counter sp1: 1
2. counter sp1: 2
3. counter sp2: 2
4. counter sp1: 0
5. counter sp2: 1
```

A TR1 Tutorial: Smart Pointers

- 0.25
- 0.50
- 0.75
- 1.00
- 1.25
- 1.50
- 1.75
- 2.00
- 2.25
- 2.50
- 2.75
- 3.00
- 3.25
- 3.50
- 3.75
- 4.00
- 4.25
- 4.50
- 4.75
- 5.00

Posted by [Marius Bancila](#) on **July 16th, 2008**

•



WEBINAR: On-demand webcast How to Boost Database Development Productivity on Linux, Docker, and Kubernetes with Microsoft SQL Server 2017 [**REGISTER >**](#)

shared_ptr in STL containers

Unlike `auto_ptr` that cannot be used in STL containers, `shared_ptr` can be used because it is copy constructable and assignable. The following sample shows a vector of `shared_ptr` to `int`; a transformation is applied on the elements of the vector, doubling the value of the pointed objects.

```
1. std::tr1::shared_ptr<int> double_it(const
2.   std::tr1::shared_ptr<int>& sp)
3. {
4.   *sp *= 2;
5.   return sp;
6. }
7.
8. int main()
9. {
10.   std::vector<std::tr1::shared_ptr<int>> numbers;
11.
12.   numbers.push_back(std::tr1::shared_ptr<int>(new int(1)));
13.   numbers.push_back(std::tr1::shared_ptr<int>(new int(2)));
14.   numbers.push_back(std::tr1::shared_ptr<int>(new int(3)));
15.
```

```

16.     std::cout << "initially" << std::endl;
17.     for(std::vector<std::tr1::shared_ptr<int>>::const_iterator
18.         it = numbers.begin();
19.         it != numbers.end();
20.         ++it)
21.     {
22.         std::cout << *(*it) << " (counter = " << (*it).use_count()
23.             << ")" << std::endl;
24.     }
25.
26.     std::transform(numbers.begin(), numbers.end(), numbers.begin(),
27.         double_it);
28.
29.     std::cout << "after transformation" << std::endl;
30.     for(std::vector<std::tr1::shared_ptr<int>>::const_iterator it =
31.         numbers.begin();
32.         it != numbers.end();
33.         ++it)
34.     {
35.         std::cout << *(*it) << " (counter = " << (*it).use_count()
36.             << ")" << std::endl;
37.     }
38.
39.     return 0;
40. }

```

Notes for the preceding code:

- `*(*it)` or `**it` means two dereferences: the first to get to the `shared_ptr` object from the iterator, and the second to get to the `int` object from the `shared_ptr`;
- The program shows the reference counter to show that calling function `double_it()` does not affect it, even though this function returns a `shared_ptr` by value.



Data Center Expansion

[Download Now](#)



Data Center Expansion

[Download Now](#)

The output is:

```

1. initially
2. 1 (counter = 1)
3. 2 (counter = 1)
4. 3 (counter = 1)
5. after transformation
6. 2 (counter = 1)
7. 4 (counter = 1)
8. 6 (counter = 1)

```

shared_ptr with class hierarchies

`shared_ptr` can work with class hierarchies, so that `shared<D>` is convertible to `shared`, where `D` is a class (or struct) derived from `B`. The following class hierarchy is used to demonstrate the concept.

```

1. class Item
2. {
3.     std::string title_;
4. public:
5.     Item(const std::string& title): title_(title) {}
6.     virtual ~Item() {}
7.
8.     virtual std::string Description() const = 0;
9.     std::string Title() const {return title_;}
10. };
11.
12. class Book : public Item
13. {

```

```

14.     int pages_;
15. public:
16.     Book(const std::string& title, int pages):Item(title),
17.         pages_(pages) {}
18.
19.     virtual std::string Description() const {return "Book: " +
20.                                                Title();}
21.     int Pages() const {return pages_;}
22. };
23.
24. class DVD : public Item
25. {
26.     int tracks_;
27. public:
28.     DVD(const std::string& title, int tracks):Item(title),
29.         tracks_(tracks) {}
30.
31.     virtual std::string Description() const {return "DVD: " +
32.                                                Title();}
33.     int Tracks() const {return tracks_;}
34. };

```

Having those classes, I will create a vector of `shared_ptr<Item>`, but add objects of type `Book` and `DVD`.

```

1. int main()
2. {
3.     std::vector<std::tr1::shared_ptr<Item>> items;
4.
5.     items.push_back(std::tr1::shared_ptr<Book>
6.                     (new Book("Effective STL", 400)));
7.     items.push_back(std::tr1::shared_ptr<DVD>
8.                     (new DVD("Left of the Middle", 14)));
9.
10.    for(std::vector<std::tr1::shared_ptr<Item>>::const_iterator
11.        it = items.begin();
12.        it != items.end();
13.        ++it)
14.    {
15.        std::cout << (*it)->Description() << std::endl;
16.    }
17.
18.    return 0;
19. }

```

The output is:

```

1. Book: Effective STL
2. DVD: Left of the Middle

```

Cast operators

To convert back, from `shared_ptr` to `shared_ptr<D>`, where `D` is a class (or structure) derived from `B`, you can use the cast function `std::tr1::dynamic_pointer_cast`.

```

1. template<class T, class U>
2.     shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);

```

This function does not throw any exception. If the case can be successfully performed, it returns a `shared_ptr<T>` that shares the ownership of the resource with the initial object (the reference counter is incremented); otherwise, it returns an empty `shared_ptr`.

```

1. int main()
2. {
3.     std::tr1::shared_ptr<Item> spi(new DVD("Left of the Middle",
4.                                           14));
5.     std::cout << "spi counter: " << spi.use_count() << std::endl;
6.
7.     std::tr1::shared_ptr<Book> spb =

```

```

8.     std::tr1::dynamic_pointer_cast<Book>(spi);
9.     if(spb)
10.    {
11.        std::cout << spb->Title() << ", " << spb->Pages()
12.            << " pages" << std::endl;
13.    }
14.
15.    std::tr1::shared_ptr<DVD> spd =
16.        std::tr1::dynamic_pointer_cast<DVD>(spi);
17.    if(spd)
18.    {
19.        std::cout << spd->Title() << ", " << spd->Tracks()
20.            << " tracks" << std::endl;
21.    }
22.
23.    std::cout << "spi counter: " << spi.use_count() << std::endl;
24.    std::cout << "spb counter: " << spb.use_count() << std::endl;
25.    std::cout << "spd counter: " << spd.use_count() << std::endl;
26.
27.    return 0;
28. }

```

The output is:

```

1. spi counter: 1
2. Left of the Middle, 14 tracks
3. spi counter: 2
4. spb counter: 0
5. spd counter: 2

```

A second cast function is `std::tr1::static_pointer_cast`. It returns an empty `shared_ptr` if the original object is empty, or a `shared_ptr<T>` object that owns the resource that is owned by the original object. The expression `static_cast<T*>(r.get())` must be valid.

```
1. static_cast<T*>(r.get());
```

This function does not throw and, if successful, increments the reference counter.

```

1. template<class T, class U>
2.     shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);

```

In the next sample, a vector holds `shared_ptr` to void. The first element is statically cast to `shared_ptr<char>`. The cast is valid as long as the source is not empty, regardless of whether the types are compatible or not.

```

1. int main()
2. {
3.     std::vector<std::tr1::shared_ptr<void>> items;
4.
5.     std::tr1::shared_ptr<char> sp1(new char('A'));
6.     std::tr1::shared_ptr<short> sp2(new short(66));
7.
8.     std::cout << "after creating the shared pointer" << std::endl;
9.     std::cout << "    sp1 counter: " << sp1.use_count() << std::endl;
10.
11.    items.push_back(sp1);
12.    items.push_back(sp2);
13.
14.    std::cout << "after adding to the vector" << std::endl;
15.    std::cout << "    sp1 counter: " << sp1.use_count() << std::endl;
16.
17.    std::tr1::shared_ptr<char> spc =
18.        std::tr1::static_pointer_cast<char>(*(items.begin()));
19.    if(spc)
20.    {
21.        std::cout << *spc << std::endl;
22.    }
23.
24.    std::cout << "after casting" << std::endl;
25.    std::cout << "    sp1 counter: " << sp1.use_count() << std::endl;

```

```

26.     std::cout << "   spc counter: " << spc.use_count() << std::endl;
27.
28.     return 0;
29. }

```

The output is:

```

1. after creating the shared pointer
2.   spl counter: 1
3. after adding to the vector
4.   spl counter: 2
5. A
6. after casting
7.   spl counter: 3
8.   spc counter: 3

```

If I switch the order of adding the elements to the vector, the program will print letter 'B' instead (its ASCII decimal code is 66).

A third casting function is `std::tr1::const_pointer_cast` that returns an empty `shared_ptr` if `const_cast<T*>(sp.get())` returns a NULL pointer. Otherwise, it returns a `shared_ptr<T>` object that owns the same resource as the source.

```

1. template<class T, class U>
2.     shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);

```

The function does not throw and, if successful, the reference counter for the resource is incremented.

The following sample yields an error:

```

1. std::tr1::shared_ptr<const int> csp(new int(5));
2. std::cout << *csp << std::endl;
3. *csp += 10;

1. error C3892: 'csp' : you cannot assign to a variable that is const

```

To modify the value of the pointer object the `const` specifier must be removed. This is shown below.

```

1. int main()
2. {
3.     std::tr1::shared_ptr<const int> csp(new int(5));
4.     std::cout << "csp counter: " << csp.use_count() << std::endl;
5.
6.     std::tr1::shared_ptr<int> sp =
7.         std::tr1::const_pointer_cast<int>(csp);
8.     *sp += 10;
9.
10.    std::cout << *csp << std::endl;
11.    std::cout << *sp << std::endl;
12.
13.    std::cout << "csp counter: " << csp.use_count() << std::endl;
14.    std::cout << "sp counter: " << sp.use_count() << std::endl;
15.
16.    return 0;
17. }

```

The output is:

```

1. csp counter: 1
2. 15
3. 15
4. csp counter: 2
5. sp counter: 2

```

A TR1 Tutorial: Smart Pointers

- 0.25
- 0.50
- 0.75
- 1.00
- 1.25
- 1.50
- 1.75
- 2.00
- 2.25
- 2.50
- 2.75
- 3.00
- 3.25
- 3.50
- 3.75
- 4.00
- 4.25
- 4.50
- 4.75
- 5.00

Posted by [Marius Bancila](#) on **July 16th, 2008**

•



WEBINAR: On-demand webcast How to Boost Database Development Productivity on Linux, Docker, and Kubernetes with Microsoft SQL Server 2017 [REGISTER >](#)

Class `std::tr1::weak_ptr`

The major weakness of `shared_ptr` is that it cannot detect cyclic dependencies. In this case, the reference counter is incremented more than it should actually be, so that the resources are no longer released when the shared pointer objects go out of scope. To fix this problem, a second smart pointer was created, `weak_ptr`, that points to a resource owned by a `shared_ptr` but does not affect the reference counter; it is a "weak reference." When the last `shared_ptr` that owns the resource referred by a `weak_ptr`, the resource is released and the weak pointer is marked as invalid. To check whether a `weak_ptr` is valid or not, you can use function `expired()` that returns true if the pointer was marked as invalid.

Even though function `get()` (that provides direct access to the wrapped pointer) is available, it's not recommended to use it even in single-threaded applications. The safe alternative is function `lock()` that returns a `shared_ptr` sharing the resource pointed by the weak pointer.

```
1. void show(const std::tr1::weak_ptr<int>& wp)
2. {
```

```

3.     std::tr1::shared_ptr<int> sp = wp.lock();
4.     std::cout << *sp << std::endl;
5. }
6.
7. int main()
8. {
9.     std::tr1::weak_ptr<int> wp;
10.    {
11.        std::tr1::shared_ptr<int> sp(new int(44));
12.        wp = sp;
13.
14.        show(wp);
15.    }
16.
17.    std::cout << "expired : " << std::boolalpha << wp.expired()
18.              << std::endl;
19.
20.    return 0;
21. }

```



Data Center Expansion

[Download Now](#)

The output is:

1. 44
2. expired : true

Example of using shared_ptr and weak_ptr together

I was saying at the beginning of this article that a typical example for a cyclic dependency is a tree implementation when the children have a "reference" to the parent. If only `shared_ptr` was used, both for references to the children and the parent, the reference counters would be incremented more than necessary and resources would not longer be deleted.

The following sample shows such a tree, but uses a `weak_ptr` to solve the cyclic dependency.

```

1. class Node
2. {
3.     std::string value_;
4.     std::tr1::shared_ptr<Node> left_;
5.     std::tr1::shared_ptr<Node> right_;
6.     std::tr1::weak_ptr<Node> parent_;
7.
8. public:
9.     Node(const std::string value): value_(value){}
10.
11.     std::string Value() const {return value_;}
12.     std::tr1::shared_ptr<Node> Left() const {return left_;}
13.     std::tr1::shared_ptr<Node> Right() const {return right_;}
14.     std::tr1::weak_ptr<Node> Parent() const {return parent_;}
15.
16.     void SetParent(std::tr1::shared_ptr<Node> node)
17.     {
18.         parent_.reset();
19.         parent_ = node;
20.     }
21.
22.     void SetLeft(std::tr1::shared_ptr<Node> node)
23.     {
24.         left_.reset();
25.         left_ = node;
26.     }
27.
28.     void SetRight(std::tr1::shared_ptr<Node> node)
29.     {
30.         right_.reset();

```

```

31.     right_ = node;
32. }
33. };
34.
35. std::string path(const std::tr1::shared_ptr<Node>& item)
36. {
37.     std::tr1::weak_ptr<Node> wparent = item->Parent();
38.     std::tr1::shared_ptr<Node> sparent = wparent.lock();
39.
40.     if(sparent)
41.     {
42.         return path(sparent) + "\\ " + item->Value();
43.     }
44.
45.     return item->Value();
46. }
47.
48. int main()
49. {
50.     std::tr1::shared_ptr<Node> root(new Node("C:"));
51.
52.     std::tr1::shared_ptr<Node> child1(new Node("dir1"));
53.     std::tr1::shared_ptr<Node> child2(new Node("dir2"));
54.
55.     root->SetLeft(child1);
56.     child1->SetParent(root);
57.
58.     root->SetRight(child2);
59.     child2->SetParent(root);
60.
61.     std::tr1::shared_ptr<Node> child11(new Node("dir11"));
62.
63.     child1->SetLeft(child11);
64.     child11->SetParent(child1);
65.
66.     std::cout << "path: " << path(child11) << std::endl;
67.
68.     return 0;
69. }

```

The output is:

```
1. c:\dir1\dir11
```

Conclusions

If `auto_ptr` was the only smart pointer available in STL until recently (but inadequately implemented), the new classes, `shared_ptr` and `weak_ptr`, are truly smart pointers. `shared_ptr` is based on reference counting (unlike `auto_ptr`, which is based on exclusive ownership) and is copy constructable and assignable; these features make it usable in STL containers. It works with polymorphic types and incomplete types, but lacks the possibility of detecting cyclic dependencies. `weak_ptr` is used to solve this problem, "weakly" referring a resource owned by a `shared_ptr`, without affecting the reference counter.

- [Page 1](#)
- [Page 2](#)
- [Page 3](#)
- [Page 4](#)

About the Author

Marius Bancila

Marius Bancila is a Microsoft MVP for VC++. He works as a software developer for a Norwegian-based company. He is mainly focused on building desktop applications with MFC and VC#. He keeps a blog at www.mariusbancila.ro/blog, focused on Windows programming. He is the co-founder of codexpert.ro, a community for Romanian C++/VC++ programmers.

Comments

- There are no comments yet. Be the first to comment!

Leave a Comment

- Your email address will not be published. All fields are required.

- Name

- Email

- Title

- Comment



[Privacy & Terms](#)



-

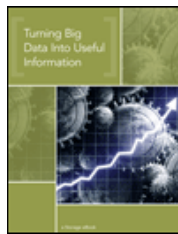
Top White Papers and Webcasts



The Challenges and Rewards of Big Data

As all sorts of data becomes available for storage, analysis and retrieval - so called 'Big Data' - there are potentially huge benefits, but equally huge

challenges...



Turning Big Data into Useful Information

The agile organization needs knowledge to act on, quickly and effectively. Though many organizations are clamouring for "Big Data", not nearly as many know

what to do with it...



The Challenges of Cloud Integration

Cloud-based integration solutions can be confusing. Adding to the confusion are the multiple ways IT departments can deliver such integration...

Most Popular Programming Stories

- [Today](#)
- [This Week](#)
- [All-Time](#)
- There have been no articles posted today.
- [1 Creating a .NET Transparent Panel](#)
- [2 IoT Development Platform: PlatformIO](#)
- [3 Using the Visual Studio Code Refactoring Tools](#)
- [1 Creating a .NET Transparent Panel](#)
- [2 IoT Development Platform: PlatformIO](#)
- [3 Using the Visual Studio Code Refactoring Tools](#)
- [4 Code Parallelization in C#](#)
- [5 IoT Development Platform: Intel IoT Platform](#)

More for Developers

- [Site Map](#)
- [News](#)
- [Windows Mobile](#)
- [Videos](#)
- [Discussions](#)
- [Blog](#)

RSS Feeds

- [All](#)
- [C#/.NET](#)
- [Win Mobile](#)
- [C++](#)
- [VB](#)

CodeGuru

- [About](#)
- [Contact](#)
- [FAQs](#)
- [List of Gurus](#)
- [Sitemap](#)

Topics

- [Visual C++ / C++](#)
- [.NET / C#](#)
- [Visual Basic](#)
- [Article Submission](#)
- [Video Submission](#)

Legal

- [Terms of Service](#)
- [Licensing and Permissions](#)
- [Privacy Policy](#)
- [Advertise](#)



Copyright 2017 QuinStreet Inc. All Rights Reserved.



Thanks for your registration, follow us on our social networks to keep up-to-date

