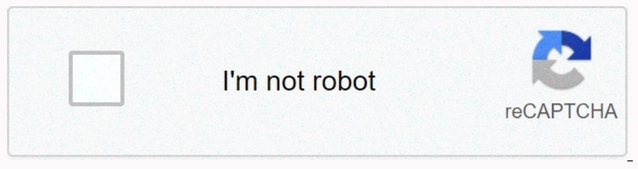

Continue

Basic mfc reversing

In this long section we will try and explain the execution of Windows programs built using the MFC framework (which is also called the AFX / MFC framework). This story isn't all that clearly told in the Microsoft documentation, but one can check the details by tracking function calls in the debugger. The author has a lingering anxiety that he made some mistakes in this account. On the other hand, he has a feeling of boredom and impatience about further examining MFC innards. One of the differences between science and computer science is that in science you are investigating things that will be there for a long time. In computer science, the object of your investigation is a target that moves eternally, a fleeting flower that blooms, shrivels, and blows away. Given that any knowledge gained about MFC Windows programs will become worthless in ten years, it is difficult to get super motivation about getting every last detail right. Even so, you need a mental image of any system when you write a program for it. The 'story system' helps you organize your knowledge of what things work and what things don't work out. Whether the story is true in every detail doesn't really matter, as long as it allows you to write good code. Think of this section of the MFC Program Stream, then, as a kind of 'creation myth' about the origin of your active MFC window. As you face other types of systems in the future, you will need the ability to formulate your own creation myths. We will give you two versions of the creation myth, first a simple and practical one, and then a more complicated one. What you really need to know In the simplest terms, this is the sequence of function calls that occur during the start of the MFC program. CPopApp Constructor. CPopApp InitInstance Method. CPopDoc Constructor. CPopView Constructor. CPopView Method OnCreate. Now the program starts running. The following CPopView methods affect the display. OnDraw is automatically invoked when the view needs to be rerawn, because (a) it is new, or (b) has been found, or (c) has been resized, or (d) it has a call to Invalidate. OnSize is automatically called when the view is resized. Various mouse processing messages, keyboards, toolbars, menus, and dialog controls are called automatically. According to how methods are encoded, they can affect CPopView or CPopDoc. When something is changed, we usually force a display update with a call to the Invalidate CPopView method. Or, indirectly, we call the CPopView OnUpdate method and let it make a call to Invalidate. If you want the app to run on its own, you can insert some code into the CPopApp OnIdle method. When the program closes the destructor is called in this order. ~CPopView. Destructor ~CPopDoc. Destructor ~CPopApp. That's all you need to know about the flow of MFC programs. But now, if you're interested, we'll tell you you More. Invisible WinMain functions Any application, such as the Pop program, defines its own type of application as the child of CWinApp. Pop uses a class called CPopApp, which is a child of the CWinApp class who in turn is a child of the CWinThread class. The CPopApp class has a prototype at Pop.h. If we look at Pop.cpp we find static variable object declarations of the CPopApp type. In general, your code will have one object of the relevant 'Application' type. AppWizard inserts the code into Pop.cpp as follows. (By the way, your app object doesn't have to be called 'theApp,' you can call it whatever you like; the crucial thing is that there's one CPopApp object declared in Pop.cpp.) The only CPopApp CPopApp theApp object; There is a CPopApp() constructor defined in Pop.cpp that performs some Application initialization. And the rest of the initialization takes place inside the WinMain program, which we'll talk about next. As you know, the C program always has a main function called the main. When the program starts, the program starts running at the beginning of the main code, and when it gets to the end the main code comes out. In the same way, Windows programs always have a main function called WinMain that starts, runs, and ends your programs. In an older Style of Windows encoding, programmers must explicitly write WinMain code. But the code is almost the same from program to program, so the MFC designer has hidden the WinMain code from you. It was built as part of AFX where the MFC class is patterned. The 'invisible' AFX WinMain function has an outline that looks like the following. To keep things simple, we have abandoned all the arguments of the function. WinMain() { pApp->InitApplication() pApp->InitInstance() pApp->Run() /* Global calling ::D ispatchMessage() or pApp->OnIdle pApp->ExitInstance() */ } We describe our WinMain function in terms of CWinApp pointer *pApp which is really just &only & theApp, that is, a pointer to a single CPopApp object, theApp. This version of the WinMain code (including comments) mentions six functions, and five of them are CWinApp member functions. They are virtual members of CWinApp. They have their own default codes, but since they are virtual functions, we are free to override them. We may change the code for CWinApp member functions to override their implementation for our CPopApp as defined in our Pop.cpp files. InitInstance, Part 1: Initializing CMainFrame Here are the first two functions mentioned in our small line of hidden WinMain functions. CWinApp::InitApplication CWinApp::InitInstance Usually you don't worry about putting InitApplication aside, and you never even see the code for it. By default, AppWizard expects you to overwrite InitInstance, and will put some code for it in Project.cpp, files, whatever your 'project' name is. A lot is going on in We'll talk about this function for all this and the next two subsections. It turns out that it is not a good idea to try and do any kind of heavy-duty initialization inside the CPopApp() constructor, since the program in a certain sense is not really ready at that stage. So this is the one thing we use InitInstance for, as a place to place initialization codes. If you want to allocate something like a global memory bitmap resource in a startup program, you will usually place the code for this at the beginning of our CPopApp::InitInstance; and when you then want to handle that memory, you will do it in the CPopApp destructor()::~CPopApp. Default CPopApp code::InitInstance takes care of opening the main frame window. This is the place where certain choices you make for the type of program you want are implemented. Options about the architecture of your Document View go here, for example. The code looks (partially) like this; You can see it all in Pop.cpp. BOOL CPopApp::InitInstance() { /* Register an app document template. A document template acts as a connection between a document, a frame window, and a view. */ CMultiDocTemplate* pDocTemplate; pDocTemplate = CMultiDocTemplate(IDR_POPTYPE, new RUNTIME_CLASS(CPopDoc), //CDocument RUNTIME_CLASS(CChildFrame), // CMDIChildWnd RUNTIME_CLASS(CPopView)); Your CView AddDocTemplate(pDocTemplate); create a new CMainFrame* pMainFrame = CMainFrame main MDI Frame window; Your CMDIFrameWnd if (!pMainFrame->LoadFrame(IDR_MAINFRAME)) returns FALSE; m_pMainWnd = pMainFrame; Parse the command line for the standard shell command, DDE, open file CCommandLineInfo cmdInfo; Create a default cmdInfo object ParseCommandLine (cmdInfo); Copy each command line param // Now the Dispatch command is specified on the command line. The default startup cmdInfo sends FileNew to your Application, // that creates documents, child frames, and views. If (! ProcessShellCommand(cmdInfo)) returns FALSE; The main window has been initialized, so view and update. pMainFrame->ShowWindow(m_nCmdShow); pMainFrame->UpdateWindow(); returns TRUE; } pDocTemplate is defined in terms of the CPopDoc, CChildFrame, and CPopView classes defined in, your projects *.h and *.cpp files with associated names. Note that these special classes each inherit, standard MFC classes CDocument, CMDIChildWnd, and CView. The template tells the application what type of document and display structure you have, answering questions such as: whether you are using SDI or MDI; and are there more than one type of document you might open? Once the application knows about the structure of your template, the main frame window is created by using a new constructor to call the CMainFrame. Your CMainFrame class inherits from the CMDIFrameWnd MFC class and is defined by AppWizard in your project file MainFrm.*. If you have some global to be used by all your files, you may want to store these variables in Class CMainFrame and initialize it in the constructor. And if you need to allocate and handle some sort of global pointer, you may want to do so in the CMainFrame constructors and destructors. Calls to The LoadFrame(IDR_MAINFRAME) attach any resources you have specified in the *.rc file to your main frame: things like menu bars and toolbars. Once we have a good pointer to the CMainFrame object, we store this in the CWnd*m_pMainWnd field of the CWinApp class, so that the application can use it whenever it needs a pointer to its main window. The next important call group uses the CCommandLineInfo class which is basically a data structure. We initialize the class cmdInfo instance with the default value specified by the CCommandLineInfo constructor, and then we call the CWinApp::P arseCommandLine function to enter command-line arguments into the cmdInfo variable. You may be wondering how your app can accept command-line arguments? After all, you're running Windows instead of DOS or LINUX, and you never even see the command line again! Well, you may have noticed in Windows Explorer that if you click on a file name, the file will be frequently opened by the application associated with it. Thus, clicking the *.dsw file will usually start a Visual Studio session, while clicking the *.doc file can open a Microsoft Word session (if it is on your computer). When this application starts in this way, the Windows operating system passes it as a command line parameter that says, in effect, 'Do File | Open the command in the following file name . . . . If there are no command line parameters of this type, the default settings for CCommandLineInfo objects such as cmdInfo will tell your application to start by processing File | New orders. And this is what happens when we hit the CWinApp method called ProcessShellCommand in a little code if (! ProcessShellCommand(cmdInfo)) returns FALSE; [If you haven't seen a lot of C code, this line may seem strange. But really it's just a careful way to say ProcessShellCommand (cmdInfo). A typical programming trick is that when you call an imagined function that may not work correctly, you have it return a Boolean value true if all goes well, and return FALSE if there is a problem. That way the calling function can use this return value to deliver bad news to the entire program. FALSE is actually another name for the integer 0, where TRUE is usually 1, although values other than 0 can usually also be accepted as TRUE.] InitInstance, Part 2: Initialize CDocument and CView When your application presses the ProcessShellCommand call inside InitInstance (or when the user selects File | New) need to create a new CDocument and an attached CView. It triggers MFC call cascade. In the case of initialization, the three main calls in the cascade are: (a) the CDocument constructor, (b) the CView constructor, and (c) the CView::OnCreate method. Say a little more about these three function calls. Something you you are remember about C++ programming is that you are not allowed to initialize variables as part of its class member declaration. If the variable is not a member of the class? if it is a global variable, or a temporary variable inside a function, for example ? then you can declare and initialize it with a line such as int crittercount = 21; or int crittercount(21);. But C++ doesn't allow you to initialize variables within class definitions. Which variables are class members need to be initialized in the class constructor? or maybe somewhere else. The CDocument Constructor is the place to initialize and/or allocate any variables included in the CDocument. And if you have variables in your CView, you need to initialize this as well. Usually we initialize as many of these in the CView constructor, but sometimes there will be CView variables that must be initialized a little later: inside the CView::OnCreate method. The reason why it is not always possible to initialize all CView variables inside the CView constructor is that when this constructor is called, the CView window does not actually 'fully exist.' When you are inside the CPopView constructor, for example, if you try and use one of the CView methods it may not work, because the special HWND window index called m_hWnd hidden inside the CView is still 0. And if you try and use the CView::GetDocument() function here it will crash because the document is still 0 as well. The only things that are safe to initialize in the CView::CView constructor are those that do not use CDoc and which do not in any way depend on the existing CView as a graphical window object. You need to wait until the follow-up CView::OnCreate function to perform CView initialization that involves viewing, say, on the display size, or viewing the owner's document. You won't find the CPopView::OnCreate function listed in your CPopView.cpp file, but you can include it by adding 'message handler for WM_CREATE.' More about message handlers in a bit. InitInstance, Part 3: Putting Windows on the screen The last two function calls in InitInstance are the CWnd, ShowWindow and UpdateWindow methods. Note that they are called by the application field that is m_pMainWnd application. The ShowWindow function looks for the appropriate size for the main frame, then draws the frame around that window on the screen. The UpdateWindow function fills the internal 'client area' window. This will involve a cascade of call functions, since filling the client area of the main frame involves first of all, creating and drawing child frame windows, and then creating and drawing CView windows inside the child. Another way to put it is when CView::UpdateWindow is called, the UpdateWindow function is called for all windows of that window child. of course before your display can appear on the screen, it must have been initialized. But we're fine with that, that's it occurs when ProcessShellCommand tells our WinApp to behave as if it has received File | New menu options. We process CDocument constructors, CView constructors, and CView::OnCreate methods. So now the main window call for UpdateWindow has generated the CPopView::Update window call. And what happens then? The call to CView::UpdateWindow generates a call to CView::OnPaint. If you search for CPopView::OnPaint in a Pop project file, you won't find it. It turns out that CView::OnPaint in turn calls CView::OnDraw. And you will find the CPopView::OnDraw method in the CPopView .cpp file. For most of our MFC programs, the OnDraw method from our view is where the rubber hits the road or, more accurately, where the pixels touch the screen. Let's take a closer look now at the MFC running cycle. MFC running cycle If you look back at the outline of the AFX function 'Hidden WinMain', you will see that the following three functions are mentioned in it. CWinApp::Run ::D ispatchMessage CWinApp::OnIdle Your application spends most of its time inside the CWinApp::Run function. What this function does is repeatedly see if there is a 'message' to process. If there is a message, then the global function ::D ispatchMessage is used to send messages to the part of your code designed to handle it. When there is no message to process, the CWinApp::OnIdle function is called until there is another message to process. (Recall from Chapter 22: Topics in C++ that when we do MFC programming, we often put :: in front of the names of global functions that are not members of any class, just to remind ourselves that they are global.) Talking about the message will take a long time, so let's postpone it for the next part. Here and now let's say something else about OnIdle. As we discussed in Chapter 6: Animation, if we want to set in action a process that runs all the time, a good way to do this is to write a CPopApp::animateAllDocs function that executes every time OnIdle is called. In the Pop Framework, we do this by replacing the basic class CWinApp function::OnIdle with CPopApp::OnIdle which calls the CWinApp::OnIdle base class and then calling the function we wrote as an animationalldocs. (As mentioned in Chapter 6: Animation, CPopApp::OnIdle is actually a little more complicated than what we showed here.) BOOL CPopApp::OnIdle(LONG lCount) { CWinApp::OnIdle(lCount); //Do the base class WinApp processing. animateAllDocs(); /* Step through all documents and update each document and all its views. */ return TRUE; //Keep doing it over and over again. } The code for animationAllDocs is kind of ugly, so we printed it out in Section 23.6. The net effects of the AnimationallDocs function in our Pop program are as follows. Browse the CPopDoc object opens and for each document calls the CPopDoc::stepDoc function which will call the cGame::step method to update the position of the list of polygonal creatures in the document. Make a call to CDocument::UpdateAllViews which will send a call to CPopView::OnUpdate for each document view, and for each call display this function will use a CView::Invalidate call that generates a call to CPopView::OnDraw, which will use the current cGraphics to display the image on the screen. But before we can start having fun, we need to explain about windows messages and functions ::D ispatchMessage. Windows Operating System messages and message handlers maintain an internal message queue. Whenever you use a mouse or keyboard, messages are placed in line. Windows also places a number of messages in the queue itself. Windows works by constantly retrieving messages from queues and forwarding them to windows intended for them. AppWin::Run the function inside your hidden winMain apportions of messages between windows included in your application. The function used to send messages is a global function called ::D ispatchMessage. At a low level, Windows represents a message by a kind of structure called MSG. The MSG structure of Windows can be written as follows. struct MSG { HWND hwnd; UINT message; WPARAM wParam; LPARAM lParam; DWORD Time; POINT pt; }; The first four of the special Windows variable types on the left are really all the same as 32-bit integers, while the last two are 64-bit integers, although DOT can also be considered a pair of 32-bit integers. Although it is common to talk about msg structure itself as a message, the MSG structure has a field called 'message.' The hwnd field of the MSG structure is the ID number that tells the Windows operating system which windows (or which CView, in MFC terms) the message is intended for. If you press a button or take a mouse action, you will generate an MSG whose numeric hwnd ID value is equal to the numeric HWND index value for the currently active window. The message field of the MSG structure is an integer code number that describes what type of message this is. All of these message codes have names starting with WM_. You can see a long list in Visual Studio by selecting Help | Index and typing dialogs in WM_. There are a few hundred different WM_? messages, but most of them are rarely used. It is common to blur the difference between the overall MSG structure and the value of its message fields. Thus it is common to talk about operating systems sending windows to, say, WM_CREATE messages, although strictly speaking, windows actually get an MSG structure whose message field is WM_CREATE. The wParam and lParam fields have additional information about MSG. In the old-style Win32 programming, you need to learn what kind of information each message put into this parameter. But in MFC programming, this is automatically done for you; Information and lParam is repackaged into an easier-to-use form. The time and field of PT MSG is rarely used. In MFC, MFC, the code for processing messages takes the form of a 'message handler' function that lives in one of your CWnd classes, usually in your CPopView, but sometimes in your CMainFrame or CChildFrame. WM_CREATE OnCreate(LPCREATESTRUCT lpCreateStruct) Called after the constructor. The best place to initialize your CView data WM_SIZE OnSize(UINT nType, int cx, int cy) nType notified you if you've just Minimized or Maximized the window. (cx, cy) is the size of the client area WM_PAINT OnPaint() Followed by a call to OnDraw, where we usually write our graph WM_LBUTTONDOWN OnLButtonDown(UINT nFlags, CPoint point) nFlags tells you which button is off. point is the cursor position in the client window coordinates WM_MOUSEMOVE OnMouseMove(UINT nFlags, point CPoint) All mouse message handlers pass the same argument as in onLButtonDown WM_VSCROLL OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) nSBCode tells what type of scroll request, nPos holds the position of the scroll bar WM_CHAR OnChar(UINT nChar, UINT nRepCnt, UINT nFlags) nChar is the ASCII code for the key if it is the key letter WM_KEYDOWN OnKeyDown(UINT nChar , UINT nRepCnt, UINT nFlags) nChar is a Windows VK_ 'Virtual key code' for the ondestroy() WM_DESTROY key called just before the destructor in exit Table 23.5 lists some common Windows messages, along with the names of their MFC message handler functions and some information about the arguments passed to the handle message. At this point, let's make some comments about the difference between OnChar and OnKeyDown. In general, you use the first function to do word processing and you use the second to use the keyboard as a control panel. Although we don't seem to be able to find it in Microsoft's online documentation, the virtual key codes returned by OnKeyDown are all defined in WINUSER. The H file that is in your include compilation subdirectory. This code has all the names that start VK_ followed by a simple name for the key. The virtual key code for the F1 key, for example, VK_F1, the virtual key code for the Left Arrow key VK_LEFT. The virtual VK_A for letters like A is actually the same as the ASCII code 'A' for A. If you press the A key only, as opposed to pressing the A key with the Shift key, you really want to get the ASCII code for 'a', compared to the ASCII code 'A'. How do we do this? Well, when you press the letter key, your CView gets the OnChar message handler call as well as the OnKeyDown message handler call. (The WinApp::Run function makes this happen by using a global function called ::TranslateMessage.) So, if you care about the case of letter keys, then you see the input of the OnChar function, because the nChar argument to this function will be the correct upper or lower case ASCII code for the letter the user typed by pressing the key and by possibly pressing the Shift key at the same time. Usually we don't do anything with other OnChar By the way, the OnKeyDown parameter. You may, for example, be tempted to try and use the nRepCnt 'recalculate' variable in a game program to detect when a user is holding the Arrow key down, but this turns out not to work reliably as it directly polls the key state with repeated calls from WinApp::OnIdle to the global function ::GetAsyncKeyState(int vKey). Program stop When you close your file, either by ending the app or by using File | Close the menu options, that means you close the view and the document, so the destructor will be called: first ~ CView and then ~ CDocument. If your program is closed, you will then call the ~CMainFrame destructor, and the last of all ~CWinApp destructors will be called. In general, destructors are always called in reverse order of the appropriate constructors. Another place where you may be processing at the exit of the program is in the OnClose or OnDestroy functions of CView. If you want a code that asks the user if he or she wants to store some file information or not, this call is usually a problem from within the OnClose function, which is called after File | A close menu selection is created, or when the view is killed by clicking on the 'X' or 'kill' box in the upper right corner. Summing up, the sequence of function calls when you exit the program is as follows. CView::OnClose CView::OnDestroy CView::~CView destructor CDocument::~CDocument destructor ~CMainFrame destructor By the way, when you click the kill box in the upper right corner of your CView, it calls CWnd::OnSysCommand (UINT nID, LPARAM lParam) message handler for your CView, with SC_CLOSE in nID parameter. OnSysCommand reacts by calling your CView::OnClose message handler, if any. If your CView does not have an OnClose message handler (some programs that are difficult to remove have OnClose installing a mailbox that asks 'Are you sure you want to exit?'), then CWnd defaults::OnClose calls your CView::OnDestroy function and then passes the control to the window destructor. Page 2One of the many rectangular areas on your screen is officially a window, more than you realize. Toolbar is a window, the button on the toolbar is the window, the scroll bar is the window, your main view is the window, and so on. When one window always lives inside another window, we talk about them as child windows and parent windows. Thus the toolbar button is the child window of the parent toolbar window, and the toolbar in turn is the child window of the parent window running the application. When the parent window has a frame shape around the child's window, we often talk about the child's window as the parent frame client. One possible source of confusion here is that we also use the words 'parent' and 'child' about the class. situations where classB has a declaration such as ClassB class : public ClassA { . . .}; Here we say that classB is the child of ClassA, or we say that ClassB inherits inherited KelasA. With the same token, we say that ClassA is the parent of ClassB, or that ClassB is from ClassA. But when we talk about parent and child windows, we mean something different. We are talking about the position of the window in what is sometimes called a window tree. As a window, the client area is the child of the frame. But as a class, the client window may be a CView class, while the parent window may be the CFrameWnd class or the CMDIChild class. As a window class type, all three classes inherit from the basic CWnd class. But they are not in a parent-child class type relationship with each other. When we program an SDI (single document interface) or MDI application in MFC, we usually talk about the window that we will use for our input and output as a display. A view is an example of a CView object. Our code will include special classes to describe the behavior of our views; in the Pop program, for example, this class will be called the CPopView class and defined in the PopView.* file. The CPopView class comes from a built-in MFC class called CView, which in turn comes from the built-in MFC window class CWnd. The outermost part of the rectangle of your program window is an object of your special type CMainFrame, and it's called the main frame. Your CMainFrame class is defined in your MainFrm.* file. As usual in this book, we focus on MDI application style, compared to SDI. But it is worth mentioning that in the SDI application, your CMainFrame class is the child of the CFrameWnd class, and in the MDI program of the CMainFrame class you are the child of the CMDIFrameWnd class. Along with your underlying CWinApp program object, your CMainFrame main frame is 'mission control center'; the shape of the main window, toolbar, menu bar, description bar, and so on. In the MDI program, you have the possibility to display various views. Each view has its own frame, defined by your program as a CChildFrame object. The CChildFrame object the MDI program inherits from the CMDIChildWnd class. And the actual view you write is the child or client of the CMDIChildWnd frame. We describe this 'window tree' in Table 23.3. CMainFrame CMDIFrameWnd MainFrm main frame.* CChildFrame CMDIChildWnd ChildFrm child frame.* The CPopView CView PopView user view.* Child frame windows are the caption and frame bars and parent of your Display window. In the MDI application, you can see the child's frame very clearly when the display is not maximal. The child frame description bar will usually have three buttons in the upper right corner. One button is to close the window, and the other two to minimize, restore (to an un maximized size and not minimized), or maximize your window. Any two options that differ from the current window status are what is offered; for example, if the window You'll have minimize and restore options. If you maximize the display in the MDI application, the display child join into the large main frame. And the Minimize, Restore, and Close child view buttons move up to the main frame menu bar. When viewed in this way, the MDI program is very similar to the SDI program, which is why sometimes one does not notice the difference. But the double set of Minimize, Restore, and Close buttons is a tip-off. The top set is for the main frame and MDI client; The bottom set is for child frame and display. For the sake of completeness and full disclosure, we must mention that there is one additional intermediate level in MDI. This is called the MDI client window. This is a gray background window that you see in the MDI application if you repeatedly use File | Close to close all open views. You also see part of the MDI client window if you resize all your active child windows so that nothing is maximized. The MDI background is gray like the local 'desktop' that the MDI application uses. Normally we don't customize it, so we don't use any special classes for it, and that's not mentioned in our standard code. How about from one window to another? Here's how you're going to do it at MDI. Table 23.4 lists the codes to be used to get the pointer to the classes listed in the top row from within the method belonging to the class in the left column. This table is useful information to have; It can save you from a lot of grief. For the sake of completeness, we must also mention how to get to the gray MDI client window, if you want to do something about it. Like maybe you want to make it start going green. Get one of the CMDIChildWnd objects, and then have that object call GetParent(). Although the main frame is the child's frame parent frame, the direct parent of the child's frame is a hidden MDI client. The call to GetParentFrame() moves the parent window chain until it finds the CFrame-treacherous one. CMDIFrameWnd (main frame) MDIGetActive() this is MDIGetActive() ->GetActiveView() CMDIChildWnd (child frame) GetParentFrame() this GetActiveView() CView (client) ::AfxGetMainFrame() GetParentFrame() class of CPopDoc we define this useful getActiveView method as follows. CPopView* CPopDoc::getActiveView() { CMDIFrameWnd *pFrame =(CMDIFrameWnd*)AfxGetMainWnd(); if (!pFrame) //If things have not been initialized. CMDIChildWnd *pChild = pFrame->MDIGetActive(); if (!pChild) //If things have not been initialized. returns NULL; CPopView *pView = (CPopView *) pChild->GetActiveView(); return pView; } Page 3A split window stores two or more document views in it. There are two types of splitter windows, dynamic splitter windows and static separator windows. A dynamic split window is a window like the one you have in Microsoft Word or in Visual Studio that users can choose to split not separating. Usually the dynamic separator window has a property that if you resize it, the panel on the right side will resize, but the panel on the left remains the same width. J J Splitter windows are always split windows. With a static separator window, programmers are better able to control the ratio of the size of two panels. If you don't want to see the split, in the static separator window, you can drag the separator separately to one side, but if you go and click there, you'll find the dividing bar still waiting there. The author designed the Pop Framework so you can build both. In the Add Static Separator section below, we show you how to change from one build to another. Our default build uses dynamic separators, which you can enable with Windows | Split. In understanding the role of separator windows we need to understand their place in the window tree. Usually CView is a window child directly from the frame. In the case of MDI, which is what we focus on, the frame is a ChildFrame object (which is an example of the CMDIChildWnd class). In the case of SDI, the frame will be a MainFrame object that is an example of the CFrameWnd class. But anyway, the usual situation is that your view is right inside the frame, in terms of window trees. When we add a splitter window, it goes between the frame and the view. Here is a picture showing two types of window trees. Let's explain how to add one type of splitter. 'Dynamic' sounds more interesting and complicated than 'static,' but in some ways static separator windows are stronger, and they are a little more difficult to program. Let's explain an easier case first. Add a dynamic separator Add CSplitter _cSplitterWnd data to ChildFrame in ChildFrm.h. Change the code CChildFrame::OnCreateClient in ChildFrm.cpp has one line like the following. BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext) { /* When adding splitters, be sure to comment on the base class code. */ // CMDIChildWnd::OnCreateClient(lpcs, pContext); /* We pass CSplitterWnd::Create call the this pointer, the maximum number of rows (1 or 2), the maximum number of columns (1 or 2), and the size (x,y) where you want the panel to just disappear, the pContext variable, and the flag variable. */ return _cSplitterWnd.Create(this, 1, 2, CSize(20, 20), pContext, WS_CHILD | WS_VISIBLE | SPLS_DYNAMIC_SPLIT; /* Defaults included | WS_HSCROLL | WS_VSCROLL in the sixth argument, but I don't want the scroll bar here. */ } And that's it! If you rebuild now, you've got a dynamic splitter window that works! If you use Windows | Split the selection, you can split one of your views in two. But there's one more consideration. Maybe you want two separate display panels to automatically differ. To do this, you can insert some code such as the following into your game override from initializeView (CPopView* pview), thus creating a view in two different panels. When the window is not your default behavior is 'left pane.' CSplitterWnd* csplitter = (CSplitterWnd*)(pview->GetParent()); if (pview == ==) / / we are left, the main view. { //Change the display settings of the values you want in the left pane } other //we are the right view of the subsidiary. { /* Change the display settings to the value you want in the right pane. */ } You may also want to overwrite the initialization of children's CGameCritterViewer(cCritterViewer *pviewer) with a similar type of switch on pviewer->_pownerview. Add a static separator If you're using a static splitter window, you have the ability to keep your window permanent no matter what size it is. The idea is to allocate some fixed proportions, such as 0.75 windows to the left pane and give the rest to the right pane. Of course users can still move the split bar back and forth, but every time we resize the frame window we will return to our standard proportions. Find the following code block at the top of the Childfrm.h file. Comments STATIC_SPLITTER, set Build | Set Active Configuration . . . to select the Release and rebuild options. /* Comments are STATIC_SPLITTER to have a dynamic separator window (user-selectable) instead of a static separator window (automatically exists). If we have a STATIC_SPLITTER, then LEFT_PANE_PERCENT determine how wide the child frame

window is devoted to the left panel. Percent macros are defined in stdafx.h to convert ranges from 0 to 100 to 0.0 to 1.0. */#define STATIC_SPLITTER #define LEFT_PANE_PERCENT PERCENT(50) Run the new *.exe and note that when you resize a window, the left separator panel always takes up 50 percent of the screen. Change the LEFT_PANE_PERCENT number to 75 and create it again. What makes this a little tricky is that the static separator window is not fully initialized until you are both called CSplitterWnd::CreateStatic and repeatedly called CSplitterWnd::CreateView to place valid CView objects inside its panels. What with all the things an MDI framework does to initialize itself, you'll press the ChildFrame::OnSize method three or four times before you successfully complete the ChildFrame execution::OnCreateClient code that initializes the _cSplitterWnd. So, you need _splittercreated BOOL to prevent you from trying to set the _cSplitterWnd's panes before it's fully ready. If you see the Pop ChildFrm code, you'll find the following two things. First, there is the CSplitter _cSplitterWnd ChildFrame data in ChildFrm.h, and _splittercreated BOOL as well. The CChildFrame Constructor initializes _splittercreated to FALSE. Second, the CChildFrame::OnCreateClient method in ChildFrm.cpp is written to make calls to _cSplitterWnd.CreateStatic, three calls to _cSplitterWnd.CreateView, and calls to _cSplitterWnd.RecalcLayout. See the code in childfrm.cpp and comments for details. As in the case of dynamic separators, you may want to change your CPopView::OnCreate initializes the views differently according to the split window pane in which they appear. Finally, if you want to change behavior relative size of the panel when you resize the window, you can change the CChildFrame::OnSize method a little further. Page 4When you develop a general purpose class such as cRandomizer, cVector, cRealBox, cRealPixelConverter, or cMemoryDC, it's a good idea to try and keep the class as portable as possible. By 'portable,' we don't mean having much reliance on other classes. The classes you develop as programmers are like the tools you build to help yourself do things. If you plan to keep programming for a few years, you don't want to have to redesign your tool more often than necessary. And, however, C++ is a language intended to run on all kinds of machines, from Windows to Mac to Linux, so it would be nice to be able to take your tools with you if you happen to migrate. One way to assess how independent a class is by looking at how many #include lines it has at the top of the *.h definition file and its implementation *.cpp file. Due to the demanding nature of MFC with the AFX application framework, we were forced to place #include <stdafx.h> at the head of every *.cpp file linked to our MFC project. So this one line worth connecting with MFC is unavoidable. But it can be unwise to bet on farming in MFC and, for example, acquire something as simple as a cVector from the basic class of MFC CObject. Let's look at the relationship between the special classes that have just been mentioned. If you flip the header file, you'll find that cRandomizer, cVector, and cRealPixelConverter don't use other classes at all, even though they all use the Real type. But Real is just a typedef that we made in realnumber.h to stand for double, so used does not put the crimp in portability. The cMemoryDC class, on the other hand, is implemented in such a windows-specific way that we don't worry about making it very MFC. CMemoryDC is actually the son of the CDC. Although the idea of a memory bitmap buffer is something you want to use on whatever platform you're working on, applying something like this is so specific to the operating system that it's impossible to solve it in a portable way. Actually, if you use swing graphics class in Java, you don't need to apply double buffering, it happens automatically. What about cRealBox? Well, thanks to having the cVector method randomVector(cRandomizer &rand) and draw (CDC *pDC, const cRealPixelConverter &rtop, BOOL bSolid) method, cRealBox uses all four other classes, not to mention the Real and MFC types. With cRealBox we move closer to the code for a particular application. The Pop Framework class has so far dropped into certain types of applications so we're not too worried about generally. Even so, it is possible that we someday move these classes to different platforms or even to different OO languages such as Java or C#. So we summarize as many Windows special things as possible. Various methods of drawing cSprite use Windows graphic code. </stdafx.h> </stdafx.h> in particular the MFC thing we do with most of our classes is get them inherited from a common MFC base class called cObject. Doing this allows us to DECLARE_SERIAL macros in headers and IMPLEMENT_SERIAL in *.cpp files. The purpose of this declaration is twofold: (a) it makes the class support the MFC model for runtime class information, which involves constructions such as the macro RUNTIME_CLASS (...), the CObject GetRuntimeClass and IsKindOf methods, and the CRuntimeClass CreateObject method; and (b) make it possible to use the CObject::Serialize method to store and load object parameter information with external files. Page 5Unlike most classes, useful utility classes, CRect, CPoint, and CSize, have members of public data. That's because these classes were actually downgraded as child classes from the old Windows structures RECT, POINT, and SIZE, respectively. This is allowed in C++ because the structure can be considered a class of all its members publicly. The names of the data fields of these three classes are shown in Table 23.1. CRect left, up, right, down CPoint x, y CSize cx, cy This utility class has some nice accessories and an overloaded operator. For example, we can get csmax CSize from crbox CRect by writing CSize csmax=crbox. Size(); CString is a very useful MFC class. CString takes place char[] which you might use to store strings. CString is a standalone size array, so you don't have to preallocate memory space before you insert characters into it. Using the CString constructor we might instead write something like CString cstrhello(Pop!); Another thing to note is that the operator == is overloaded for CString so that strings mean that they have the same characters. The MFC code for this utility class is formulated in a fairly polymorphic way so that you can usually forward these objects without explicit & reference even if the call function expects a pointer. In other words, the MFC function prototyped for the CRect* argument will also accept the CRect argument, which is passed by the reference as CRect&. Functions that want a data type with the old kludgy name LPCTSTR (for 'long pointer to constant string') will receive a simple CString argument. ASPTreeView.com Has Been 1000 100000000000000000000000000000000000000000000000000000 Info... Page 6 The naming convention in the MFC program is that MFC class names all start with capital C, and the class member functions are all written as run-together phrases in which each word is capitalized. FillSolidRect is an example of the function name. We are free to give our variables any kind of name we want, but because the MFC program is so big and complicated, we try and make as helpful names as possible. A good variable name will give you (a) information type of variable class is an example, and (b) information about the meaning of your variables within the program. General General is to give you our variable name which starts with a few letters to remind us of the type of variable, and to have the rest say something about what the variable does. Variable names should always start in lowercase. If you like, you can mix some capital later into the name, to indicate a word break, as in bSoundFlag, for the BOOL variable. But all lowercase names for variables are easier to remember and faster to type. Thus, we may give the variable CString the name cstrhello. Or we could use the cstring_hello or cStrHello. In cases where the variable is a pointer to the class, we want to start the variable name with p. Thus the CDC* variable may be named pdc or possibly pDC. What about variables that are examples of common types of classes? Usually uses n as a pre awalan to indicate an integer variable. Sometimes, when an integer is a 'count' variable used to calculate something like the number of pixels in screen width, the c pre awalan is used. The letter f or b denotes a variable that is a BOOL flag. As recently mentioned, the letter p denotes the pointer. Older Windows documentation sometimes makes a difference (no longer meaningful) between the normal pointer type p and the 'long' or 'remote' pointer lp. Win32 programming often uses a kind of index object such as a pointer called HANDLE, and the h pre awalt is used for variables of this type. In older Windows code we often see the preade sz or lpsz to stand for variables that represent a series of characters. Of course in MFC programming, we will generally use CString variables instead. You will see a little ambiguity in all this; the truth is that all our naming conventions are a little flexible. The golden rule is to develop some good conventions and stick to them in all your code. However, as you learn more about programming, your ideas about conventions will evolve, so don't feel task bound to forever stick to something stupid you used to do when you started. Windows uses many of the constants defined in its header file. It is always written in all capitals, with a bottom line separating the component parts. Often the specified constant has a pre awalan that reminds you of the context in which the constant is used. For example random, constants DT_CENTER and DT_VCENTER that can be used as arguments to the CDC::D rawText has a DT pre awalon to remind you that these constants have something to do with DrawText. As a rule you should not use a one-letter name like u or x, unless it is immediately clear what you mean, nor should you use a meaningless name like foo or anything else. that time is definitely acceptable to use variable names one letter is for index i, j, k used to index loops. The AFX section of the MFC programming image has to do with what is in the 30 or so files that Visual Studio AppWizard creates for you when you create a new project. One thing worth mentioning here is that MFC has some some with the prea prea or afx. These functions are used to get information about your program. An example of such a function is CWnd* AfxGetMainWnd( );. We'll see more at AFX below. But first we will have a little information for the programmer Win32, some information about the documents and views, and another look at the Pop program code. Later we will also define many of our own classes. To clarify that this is our class, we will start all their names in lowercase c. Thus we will use a name like cMyClass rather than MyClass. We would absolutely never want to use a name like CMyClass, because that last format had to be left only for use by MFC. MFC class members generally start with m_, although CPoint class members are called only x and y and CRect members are left, top, right, and bottom. We will always have our own class data members starting with the _bottom line. A general principle of naming your class members is to start a member name with an underscore, and then you have the possibility of using an underline name as the accessory function name (although usually it is a better practice to start an accessory with the word 'get'). It is a good idea to name 'ugly' members of your private or protected class (by starting with a bottom line) to remind you that they are indeed not public. When we define a special function as one of our class methods, we almost always use the convention of starting it with lowercase letters and using capital letters at word breaks. Thus, we may have the RandomInt method or the setAutoRotate method. You never want to start your method with capital letters because then they will look like standard MFC functions, and the next programmer to use your code will get confused when he tries to find MFC documentation for these functions. Page 7When compiling processes C++ code it renames the function to include information about the types of function arguments. This process, usually invisible, is called name-mangling. You most often see the effects of name-mangling when you place a prototype class method in a header file, but then fail to implement the method in any of the *.cpp your project. The project will compose just fine, but in the end, you will get a linker message. So, if you leave the code for the cCritter method int cCritter::move(Real dt), you will get a linker error like the following. popview.obj : LNK2001 error: unresolved external symbol public: int __thiscall cCritter::move(float) (?move@cCritter@@QAEHM@Z) The string at the end of the line is the name of the destroyed method. Remember how we talked about how to have const after function name changed how C++ That's because const got into name-mangled names. Name-mangling can be a problem in advanced programming situations, for example if (a) you want to link someone else's *.c module with your *.cpp module, or (b) to find functions by name in a dynamic link library module. This is because although you might think the function name is what you call it, saying 'int MyFunction(),' C++ will actually assign a different 'mangled' name to your function. You can turn off name-mangling for a function name by specifically asking in its prototype that it be treated like a C function, using a line like the following. extern C int MyFunction(); ASPTreeView.com This Has Been БВХКS3АХexpired. Info... Page 8Simulation works better with real numbers, and there are Real types and some macros that we use in our program, so we'll have a file called RealNumber.h that we include in our file using real numbers. (If we don't want to do any of that included, we can place the realnumber.h #include somewhere inside the StdAfx.h file, for example just before the line that says //{{AFX_INSERT_LOCATION}}.) In our code, we use Real as another word for double or to float. To give ourselves the freedom to switch between double and float, depending on whether we have a greater need for accuracy or for speed, we have a line that says typedef double Real or typedef float Real inside RealNumber.h. #ifndef REALNUMBER_H #define REALNUMBER_H/* Note that to use real types throughout my program, I must always remember to include realnumber.h. */ #define USEFLOAT // float slightly faster, slightly less accurate. #ifdef USEFLOAT typedef float Real; #else //no USEFLOAT typedef double Real; #endif //USEFLOAT /* When I use a Real float, I will get a warning message generated by the fact that constants like 2.0 are viewed double, so that if x is a float, the x=x/2.0 line will be seen as a double convert to a float. These messages take this form: warning C4244: '=' : conversion from 'double' to 'float', possible loss of C4305 data warning: '=' : t deduction from 'const double' to 'float' I can disable this warning message with a pragma stating two warning numbers. */ warning #pragma(disable: 4305 4244) #define PI 3.14159265358979 #define CLAMP(x,lo,hi) (x)=((x)<(lo))? (lo):(((x)>(hi))? (hi):(x))) #define SMALL_REAL 0.00001 #define BIG_REAL 100000000.0 #endif //REALNUMBER_H The essence of the typed double real line is that if at some point you find that you want to squeeze a little more speed out of our program, you can come back here and replace the double by floating. We need PI because, although an ANSI C library is supposed to define the M_PI constant of this value in math.h, microsoft's version of the math.h file doesn't seem to have a pi in it. (For sanity's sake, a software engineer learns not to obsess over things like this; repair and continue.) Clamp macros are useful for forcing x variables to lie between two lo and hi values. SMALL_REAL in the code where you want to avoid dividing by numbers that are too close to 0. And BIG_REAL useful to include as a starting value in the search loop as we continue to search number, as in the code for cCritter* cBiota::p ickClosest(const cVector &vclick) in biota.cpp example. ASPTreeView.com It has been НИ¶oEexpired. Info... Page 9Suppose You want to have an array that holds, say, the cCritter object. There are three options: keep your own C-style array, use array templates from the ANSI C++ STL Standard Template Library, or use Microsoft CArray templates. To maintain arrays themselves means using declarations such as cCritter *_critter or cCritter _critter[]. (Note that C considers these two declarations to be equivalent; in C arrays are pointers only.) If you do this, you also need an int _critter_count variable to track the number of points currently saved. And you have to preallocate _critter for a very large size (and worry about eventually writing from the end), or you have to keep reallocating memory for the array as it grows. That's a lot of work, and it's easy to make mistakes in doing so. Writing your own array of code today means reinventing the wheel. This is much better than using a pre-applied array template. Here we have two options: STL or MFC. STL vector arrays Each C++ version comes with a platform-independent template library known as STL or the Standard Template Library. The library was written at Hewlett-Packard around 1995 and is now part of the official ANSI C++ language standard, meaning that all C++ compilations for each platform must support it. The STL library includes, among many other types of collection classes, array templates known as vectors. The code for the STL template will usually be found in the file in your Visual Studio including the directory; This name includes files, somewhat strangely, just regular VECTOR, without file extensions. For whatever reason, Visual Studio needs a little encouragement to let STL function; whenever you include an STL file, you must follow it with a form code line using namespace std;. To create STL arrays of several T-class types, you use a new <T>type vector. Include the necessary instructions in the file. #include use <vector>namespace std; This included will only appear in the file where it is used, and it should be the last included listed in <vector> </afxtempl.h>head of these files. #include Prototype<cCritter> array _critter class. CArray<cCritter, ccritter= &=> _critter; Prototype array of the basic type. <int>vector _radius; CArray<int, int=> _radius; Add elements to the end of an array. push_back (new point) Add (new point) Delete array members in slot i. delete(i) ClearAt(i) Access array size. size() GetSize() Sets the array size. Resize(newsize) Accessing elements to read or organize. [i] ElementAt(i) GetAt(i), to read only One plus STL is that they are portable for all C++ platforms, so you can expect to be able to use them in more environments. But there are some annoying gotchas associated with now</int,> </int> </cCritter,> </cCritter> </afxtempl.h> </vector> </vector> </T> </T> old STL template ? I'm not going to get into them here. Of course Visual Studio documentation from STL is not as easy to use as their CArray documentation. And the CArray template is tweaked so that it works smoothly with MFC methods to 'serialize' or save and load files. It's easier to give up and do it the Microsoft way instead of trying to use STL. We will use the MFC template CArray in this book. Table 22.1 compares the use of templates for STL arrays and MFC arrays. The MFC CArray Array Code for the CArray template is in a file called afxtempl.h. If you want to declare some CArray variables in the file, you need to add this line to the top of the file, so the compiler will know what we're talking about. #include For <afxtempl.h>create an array of objects of several types of T, we usually declare our new type as CArray<T, t&=>. The meaning of the second mention of T in the declaration is that this is the kind used in some internal functions of CArray that do things like add your elements to CArray. The idea is that someone usually uses a pass by reference for this type so that it can marginally increase the speed of your code, by effectively passing the pointer instead of copying the entire structure. In cases where T is a primitive type like int, we don't bother with &; for the second argument. The default _critter for both cases will be 0, and if you immediately try and run a line that sets, say, _critter[1] for something, then your program will crash. Adding elements to an array automatically grows them to sizes large enough to hold new elements (i.e. grow them one at a time). But, if you plan to read a lot of elements into your array using the operator[], you need to set the array size to be large enough before you start. Another thing about array templates that are important to note is that when you add a type T element to your array, it is a copy of your element that actually goes into the array rather than the element itself. So you have to make sure to have a good copy constructor and overloaded =operator defined for each class of object that you will put in the template array. Actually we will use variations on the CArray template, and this is the template CTypedPtrArray.<CObArray, cmyclass= *=> This is used to hold the pointer array to instances of some classes we have written called cMyClass. For this to work, cMyClass must inherit from the MFC CObject class, which is a kind of universal base class in MFC. Learn more about it below. Page 10All as constructors, each class has a destructor method with a name such as ~SomeClass. (On the US keyboard, the '~' symbol is usually located in the upper left corner of the keyboard but in other parts of the world you will find it elsewhere.) We have no choice about the names of the methods of constructors and destructors. For any SomeClass class, the constructor is called SomeClass and the destructor is called</CObArray,> </T,> </afxtempl.h> </afxtempl.h> When a variable exits its destructor scope it is automatically called. So if your program has global variables, the variable destructor is called when your program ends. If you have a function that has sctemp local variables in it that are SomeClass typed, the SomeClass constructor is called when the code execution touches the line where the SomeClass variable is declared, and the SomeClass destructor code for sctemp is called when the execution reaches the function closing bracket. More precisely, when you press the closing bracket of a function, the destructor is called for all local variables declared inside the function. Destructors for local variables are called in reverse order that the constructor is called. If you use the new operator to make pointers to SomeClass instances, you must explicitly call the removal operator on the pointer to call the destructor and free up memory. ASPTreeView.com It's Been 100 Years. Info... The Page 11Each Pop Framework game is based on a cBiota object that is a special type of cCritter pointer array*. The program is run by running through this array and letting each member creature call the method. At each step of the game animation, for example, we walk through the cBiota array and let each member of the cCritter pointer make a call to the virtual update method. Instead of having to check the types of creatures we have in each slot, we can let each creature call its own version of the update method. This is what polymorphism is for. To make the function behave polymorphic, we must do two things. We already mentioned this in Section 22.10: Virtual Methods, but it needs to be said again. The first step in creating a polymorphic method is that the function must be declared as virtual in the base class. As mentioned earlier, you do not need to put a virtual word in the child's class declaration, although you can if you like. The second step in creating a polymorphic behavior method is to be stored in a pointer variable rather than in a base class instance variable. That is, consider the difference between these two declarations. CArray<cCritter, ccritter= &=> _embeddedarray; CArray<cCritter*, ccritter*=> _pointerarray; Suppose, for the purposes of this discussion, the virtual cCritter update method is overwritten by the cCritterArmed child class to do something different. Now, even if some of the objects in _embeddedarray are 'really' cCritterArmed objects, when you place them into CArray<cCritter, ccritter&=>, you have toand them into cCritter objects, so now if you walk through this array and call _embeddedarray[i].update(...), you will always end up using only the cCritter::update method. However, if some of the cCritter* pointers _pointerarray are actually cCritterArmed* pointers, then when you walk through an array and call _pointerarray[i]->update(); You'll get good</cCritter,> </cCritter*,> </cCritter,> </cCritter,> method or method cCritterArmed::update, depending on the type of pointer. Since the cCritterArmed class is the child of the cCritter class, cCritterArmed * can be considered a cCritter*, so we are allowed to include it in the array. But the pointer 'remembers' what type of class it actually shows, and this information will be used when calling possible polymorphic methods. This is why pop frameworks use cCritter pointer arrays*. It turns out, though, that for reasons that are having an affair with writing creature data onto files, it works better to use a less obvious array template CTypedPtrArray<CObArray, ccritter*=> replacing the expected CArray<cCritter*, ccritter*=>. Learn more about this in Chapter 30: Serialization. ASPTreeView.com It's already been AiPexpired. Info... Page 12 A virtual keyword in front of the parent class method tells the C++ compiler that the class has a child class that has a method that has the same name but that acts differently in the child's class. If (a) the method is declared as virtual, and (b) the object calling the intended method via the pointer, then (c) the compiled program will, even as it runs, be able to decide which virtual method implementation to use. It should be emphasized that this 'runtime binding' only works if your programmer meets both conditions: (a) you use virtual in your method declaration, and (b) you use the pointer to your object. Except in the case of destructors, the appropriate virtual functions have the same name. You do not place the virtual word in front of the actual function implementation code in *.cpp. You can place the virtual in front of a child's class function declaration in the child's class header file or not, as often as you want. In other words, to get started, you really only need to place the virtual in one place: in front of the declaration of the function of the parent class. But, to make our code easier to read, when we obtain a child's class from a parent class with virtual functions, we usually place the virtual in front of the declaration of the child's method as well as in the declaration of the parent method. The child does need to have a declaration for this method to override it in any case. One thing that is a little strange is that parent class destructors such as ~cProgrammer() need to be declared virtual even though child class destructors such as ~cTeacherProgrammer() seem to have different names. But you don't actually call him a destructor by name. In programs where we have a cProgrammer * _ptextbookauthor, we may call cProgrammer or cTeacherProgrammer or cTeacher destructor with a line such as delete _ptextbookauthor;. The thing is, it's possible that _ptextbookauthor initialize as the new cTeacher, so we just don't know. The removal operator calls the destructor without referring to the destructor method by name. So the compiled code needs to really look pointer to find out if it's really cProgrammer* or</cCritter*,> </cCritter*,> </CObArray,> </CObArray,> so that he knows which destructor to use. And unless you meet the 'virtual state' by creating a virtual destructor, then the code will not know to bind the runtime and choose between using the parent or child method as appropriate. Destructors are different here because cTeacherProgrammer has more items to destroy, in particular, cGollywog * _pimaginaryfriend. Slogan for class: If your child is richer than you, you need a virtual destructor. One last point should be mentioned here. Typically, when you have a virtual void somemethod() method in a basic class called, say, cParentClass, then when you override a method in a child class called, say, cChildClass, the somemethod() child class will call the parent class method if we explicitly request it with this kind of code. void cChildClass::somemethod() { cParentClass::somemethod(); //Your extra child's class code goes here.... } But in the case of virtual destructors, the parent class destructor method will be automatically invoked when the object is deleted. This corresponds to the standard C++ execution sequence of the constructors and destructors mentioned in the last subsection. cChildClass::~cChildClass() { //Your extra child class destructor code goes here... ... /* The cParentClass::~cParentClass destructor will be automatically called here at the end of the ~cChildClass destructor call. */ } Page 13Suppose that SomeClass is a class whose prototype includes _val members and two Functions SomeOtherFunction class SomeClass { int _val; int ValSquared(){return _val * _val;} int SomeFunction(int input); int SomeOtherFunction(int first, int second); }; Now if K is an object of the SomeClass type, then the value, let's say, K.SomeFunction(17)will depend on (a) argument 17, (b) the definition of the SomeFunction function, and (c) the content of the K. K object is the 'implicit argument' to call this function. When you write code for SomeFunction, you play on the fact that you have implicit arguments for the function, although you don't explicitly point it out. Thus the code for SomeFunction may look like the following. int SomeClass::SomeFunction(int input) { return SomeOtherFungction(_val, ValSquared()); } This means that K.SomeFunction(17) will be evaluated as K.SomeOtherFunction(K._val, K.ValSquared()), where K.ValSquared()is evaluated as K._val * K._val. The initially confusing C++ is used inside the function as a way to refer to the pointer to implicit invocation objects. The object itself is *this, that is, the object that it is pointing to. Remember that if p is the pointer, then *p 'dereferences' the pointer to stand for the actual object indicated by the pointer. If we want, we can put this >_val or *this._val the _val in the SomeClass::SomeFunction definition. But we don't want to, because this will beat the whole beauty of C++ Concise. In general, when you look at some foreign code and you see the call within the class method definition, you can usually expect that the foreign function you see called 'naked' is actually a member of the class. When we talk about a function called 'naked,' we mean that it is called without an explicit call object in front of it, no callers.or pcaller-> in other words. If you do not find a function that is listed as a class method, perhaps it is the parent class method of the class. Now and then we want to explicitly avoid using the class version itself of the member function, prefer to use the global function of the same name. In this case, we put the symbol :: in front of the function name. The name of this symbol, by the way, is 'operator resolution scope.' We sometimes use it in MFC programming, since most of our MFC class methods have the same name as some global methods. To make the code easier to understand, when we use a global function at all we will usually put :: in front of its name only to remind ourselves that it is not a method of members of any class. Page 14 &Symbol; is a way to tell a compilation to forward an object with a reference, that is, the compilation generates a pointer to the object and forwards it, instead of copying all the object fields. If you have a function with a prototype like the following, you can give it the cSomeClass and Func argument inputs can actually change what's in the input. int Func(cSomeClass & input); One of the reasons we do this is because we might actually want to change the field of the object being passed through. Remember that in C, you cannot directly change the value of a variable passed to a function. But the C++ trick to enter & into a function prototype lets you have a function that changes its input value. Another reason why we sometimes want to forward an object as a pointer instead of as a structure is that it is faster to forward the pointer to the object instead of copying the entire object. And there will be times when you want this speed, but you definitely don't want to change what's in the input. Troublesome const also appears with respect to & in the function declaration. Here deals with function arguments rather than callers. If you want to speed past arguments based on references, but you know that you don't want to actually change what's in the object, then you use const followed by the class name and & means 'don't let any changes in this object field, but pass it as the pointer you generate.' And you use prototypes like: Func(const cSomeClass & input); You can combine const arguments and const function declarations in all possible ways. C++ sees all this as a different function. Int &input)const; int Func(SomeClass & input); int Func(const SomeClass &input)const; int Func(const SomeClass &input); For full disclosure, we may also mention four other possible ways that you can prototype functions in C++. Int Int input)const; int Func(someclass input); int Func(SomeClass *input)const; int Func(SomeClass *input); What's the fuss, huh? A real nightmare! No wonder so many people want to leave C++ for the tranquility of Java or C#. Bjarne Stroustrup, the inventor of C++, claims that the only reason that Java is simple is because it is a young language that does not have all the features of C++. He said that C ++ is very complicated because it is ripe. But once you practice using C++, you will find that having eight possible ways to prototype functions is not as bad as you would expect. This practice always makes every function as const as you can, so you don't really have to think about it so much. Various forms allow you to ensure your code will be portable, secure, and as fast as possible. Remember to match the *.cpp format to the *.h declaration format. And, on top of that, be careful that your child class uses the same declaration as the parent class. Page 15We will randomize our parameters a lot in this book. In most of our programs, we want to have many different objects, and we want to use randomize to keep objects from all being the same. In our more advanced programs, we may have our objects using randoming so as to give themselves an interesting behavioral appearance. Even when there is only one object using a given set of parameters, it is not always clear which value to use. There may be no 'best' value, and it is more interesting to see the various possibilities. In other cases, there may be a set of 'best' values, but the only way to try and find these values is to feel around the parameter space by trying many random possibilities. Library C standard function supplies three important relevant functions here. long int time (NULL) srand(unsigned int seed) short int rand() Randomizing with the C library Each call to rand() returns a 15-bit positive integer located between 0 and +32K. The way Rand works is library C retains the hidden 32-bit integer variable holdrand. When your program starts, holdrand has a default value of 1. Whenever you call rand(), holdrand is replaced by holdrand * 214013 + 2531011, and rand returns a low 15 bits of the high word holdrand. (That is, the return value is (holdrand >> 16) & 0x7FFF.) If rand starts with the same holdrand value, it will always give you the same order of values. So what we usually do is use the srand function to initialize the randomizer into a fresh state. What srand(seed) does is just set the hidden holdrand variable that is the same as the seed argument. But how can our deterministic program come 'Random' seed to eat into srand? What we usually do is use the time(NULL) function. This function returns the number of seconds that have elapsed since midnight, January 1, 1970. For whatever reason, this is the official 'instant birth' of our computer era, we, like an anomalous moment when the Western calendar went from BC to 1AD. cRandomizer class To make it easier to randomize things, we wrap the randomize function inside the class called cRandomizer. Instead of having to use the % operator to bring numbers into range, cRandomizer has a random method(N) that will give you a number between 0 and N ? 1, a random method(lo, hi) that will give you a number from lo to hi inclusive, randomReal(loreal, hireal) method to give you real numbers between loreal and hireal, randomColor() method, and so on. RandomBOOL (double truthweight) is a funny function that lets you determine the true weight between 0.0 and 1.0. The weight of truth is the probability of randomBOOL returning TRUE. So randomBOOL (0.75) will return TRUE three-fourths of the time and will return FALSE a quarter of the time. The default cRandomizer constructor allows it with the time function. You also have the option to feed the seed number to the constructor thus initializing your cRandomizer into certain circumstances. When you debug a program, it is often a good idea to have the cRandomizer always set to the same state making it easier for you to reproduce the exact same behavior over and over again. #ifndef RANDOMIZER_H #define RANDOMIZER_H /* This is a set of 32-bit-based randomize functions. This function uses standard C library techniques. This code is written for portable, although there is one line you need to comment on or out in the randomizer's head.cpp according to whether you are using Microsoft MFC or not. This randomization function is based on a modular schema derived from the Microsoft implementation of the C library randomizer. In Microsoft implementations, the int rand() function of library C works by maintaining _holdrand and _holdrand variables * 214013 + 2531011. rand() returns (_holdrand >> 16) & 07FFF, which is a positive 15-bit short integer. We used the same scheme, but adjusted it to return an unsigned 32-bit long integer. Returning _holdrand gives too much correlation, so we actually run the _holdrand update twice, and use the high words of two consecutive _holdrands as the top and bottom words of the value we returned. File Randomizer.cpp includes a historical record in the end of a failed attempt to base randomization on Ca Wolfram Rule 30. */ #include realnumber.h //For the Real typedef, i.e. double or float class cRandomizer { private: static cRandomizer * _pinstancesingleton; long value _seed; //Start of _shiftregister unsigned long _shiftregister; //Used internally for the running compute process. unsigned long _thirtytwobits(); // Internal pseudoran function used. cRandomizer(); //Uses the C Library randomizer and its seeds with time. long seednumber); //C Library randomizer seeded by seednumber. public: cRandomizer statis * _pinstance(); void statis delete singleton (); unsigned long getSeed(){return _seed;} _seed;} setSeed(unsigned long seednumber); Start in a certain unsigned state of random lengthSeed(void); Seed with time in seconds unsigned random length(); Returns an unsigned long int without a random mark length (unsigned length n); Return int between 0 and n? 1 random length (long lon, long hin); int between lon and hin inclusive BOOL randomBOOL(Real truthweight = 0.5); Return true truthweight often. unsigned randomByte(void) char; Returns bytes between 0 and 255 unsigned short randomShort(unsigned short n); Short between 0 and n-1 Real randomReal(void); Real between 0.0 and 1.0 Real randomSignedReal(void); Real between -1.0 and 1.0 Real randomReal(Real lo, Real hi); Real between lo and real mutate hi (Real base, Real lo, Real hi, Real percent); Mutate base based on percent of size. mutated int (int base, int lo, int hi, Real percent); Mutate base based on percent of size. long unsigned length (unsigned long base, Percent real); Mutated colors. Real RandomSign(void); 1.0 or -1.0 void randomUnitDiskPair(Real *x, Real *y); Creates (x,y) a random point with a distance of <= 1 of (0.0) unfectUnitPair(Real *x, Real *y); Make (x,y) a random point 1 distance from (0.0) an unsigned random lengthColor(); Nomor WINDOWS COLORREF. }; #endif RANDOMIZER_H because we will use cRandomizer a lot, you may wonder if, over time, cRandomizer will collect more functions. Then, for example, we might have some cMyClass with parameters that we want to randomize, so it would make sense to add a cRandomizer void::randomizeMyClass(cMyClass &myclassobject) ; methods to the cRandomizer class? No, this would be a bad idea, because then we'd always come back and change Randomizer.h and Randomizer.cpp files. For the sake of sanity, it is much better to try and get your basic and general purpose files working once and for all and not constantly coming back and changing them. This is one of the reasons why we don't use COLORREF as a type in Randomizer.h (instead we use an unsigned length equivalent). So how then can we write a method to randomly cMyClass with cRandomizer? We will place the randomization method which is a member of cMyClass, that is, we will have a cMyClass vacancy::randomize(). And then in the implementation for this method, we'll use the standard cRandomizer call to randomly cMyClass fields. CRandomizer static::Randomizer object In order not to continue having to create a new cRandomizer object to randomly cRandomizer things, we give the cRandomizer class a static cRandomizer member. This means that anywhere in our Pop program, we can call, say, to get a random real number. This is very much a Javanese style thing to do; Java has many dedicated members and a static method of its standard class. This is an example of the so-called 'Singleton pattern.' In a static C++ such as RANDOMIZER it must be declared in the *.h file and it must be completely instant, or for example, in the *.cpp. We put instantiation inside randomizer.cpp. cRandomizer* cRandomizer::p instance()/* pinstance() allocates the _pinstancesingleton NULL, and then returns it. */ { if (cRandomizer::_pinstancesingleton == NULL) //First time pinstance() is called #ifndef _DEBUG //not _DEBUG means Release build cRandomizer::_pinstancesingleton = cRandomizer new(); /* In build Release, the default constructor seed with time for variations. */ #else // _DEBUG means Debug build cRandomizer::_pinstancesingleton = cRandomizer new(1); // In debug builds, use fixed seeds to help replicate bugs. #endif //End the _DEBUG switch return cRandomizer::_pinstancesingleton In understanding this code, you need to know that Visual Studio has a DEBUG line #define that will be processed if and only if you create a Debug build instead of a Release build. Thus in our code, the compiler chooses between two declarations depending on whether you debug or build Release. We don't want randomizers to be so random in Debug, because there we want to be able to start sessions over and over again and keep seeing the same bugs in the same place. Page 16Let display on an example of the type of call we bait into the OpenGL state engine hopper so as to prepare to draw something. In particular, let's see what it takes to draw a white square. Initialize Window (Described in the next subsection) ::glClearColor(0.0, 0.0, 0.0, 0.0); ::glClear(GL_COLOR_BUFFER_BIT); ::glColor3f(1.0, 1.0, 1.0); ::glOrtho(0,0, 1,0, 0.0, 1.0, -1,0, 1,0); ::glBegin(GL_POLYGON) ::glVertex(0.25, 0.25, 0.0); ::glVertex(0.75, 0.25, 0.0); ::glVertex(0.75, 0.75, 0.0); ::glVertex(0.25, 0.75, 0.0); ::glEnd(); ::glFinish(); Updating Windows (Described in the next subsection) you can see more code like this in the graphicsopengl file.cpp Pop Framework. ASPTreeView.com This Has Been БИЧИ СГCexpired. Info... Page 17To use OpenGL, you link your executable to a custom OpenGL library and include some custom OpenGL headers. You usually do 'linking' by adding additional libraries to your project to your project: opengl32.lib and glu32.lib. (You can see Attachment C on how to add a library to a Visual Studio project.) And you do 'include' by placing a line like the following into any file that mentions the OpenGL function. (Specifically, the Pop framework has these lines in the graphicsopengl.h. header) #include gl\gl.h #include gl\glu.h Once your code is linked, you can call the OpenGL function in your code. The OpenGL function starts with one of the gl or glu presets. We usually place the operator's scope resolution:::' in front of the names of these functions as a reminder that this is a global function, rather than being a member of any class. In addition, Windows has several functions designed to link to OpenGL. Some, but not all, of these functions begin with the preale of wgl. The functions our code uses are found in gl function of OpenGL (gl.h, opengl32.lib) glu function of OpenGL Utilities (glu.h glu.lib) wgl function of OpenGL Extension for Windows (windows.h, opengl32.lib) And of course our code will, as always, use special functions defined in the MFC library and in the basic Win32 library. Figure 26.1 shows how Pop and the various libraries depend on each other. ASPTreeView.com This has P АБДЅ·expired. Info... Page 18It takes some work to make Windows programs have a persistent display that remains in place even if the windows are resized or closed and then uncovered. OnDraw Method The standard practice is for our Windows programs to do all their writing to the screen in one single function, the CView::OnDraw method. When did your CView get the OnDraw message? Here are four important ways for this to happen. When CView is first created, whether automatically at startup, by File | New call or through Window | New call. First there is the call to the constructor, then the call to OnCreate, and then the call to OnDraw. Whenever your CView is resized, whether by dragging corners using commands like Windows | Tile, or by clicking the Maximize or Restore button in the upper-right corner of the window. First there's the call on OnSize, and then the call to OnDraw. When you uncover the part of your window that has been covered by another window, either by clicking on your window to bring it to the 'front' of the window on the screen, or by dragging the closing window to one side. Your CView gets an OnDraw call. If you place the pDC->GetClipBox(rect) line inside your OnDraw code, you will find that the fix will receive client window coordinates from the smallest rectangle covering the newly discovered part of the window. This rectangle is known as the CDC clip box. When the CView::Invalidate function is called from within one of the other CView functions. This will queue up messages telling CView to rearrange itself as soon as there are no other messages to handle on the queue. If you want a re-operation to happen immediately and without an interval, you sometimes follow a call to Invalidate via a call to UpdateWindow. (By the way, if you try and make an UpdateWindow call without calling Invalidate first, the call may not have any visible effects. OnDraw generally only rearranges as many windows as 'thinking' needs to be done. A call to Invalidate tells OnDraw that the entire window needs to be re-drawn, but the naked UpdateWindow call itself will result in an OnDraw call with the CDC* argument that 'thinks' that no window really needs to be re-drawn.) Bitmap or display list? There are actually two possible approaches to maintaining a continuous appearance in screen, and we ended up using both. This is sometimes called the display list approach and bitmap approach. The idea behind the display list approach is to set a code that will track every time a user draws something in and then 'replay' this during OnDraw. Since you have your items in the list, you can remove things from the list and add things as well. Tracking a list of views requires a large number of program engines, especially if you write your own display list code. But thanks to the CArray template it's not so hard to do. The view list approach embodies the insight that you can store a rather compact complicated graphics file by getting a list of structures, with each structure including, say, the location and description of what you wrote there. Formatted word processing files are a kind of display list. Instead of storing bitmap images of your pages, files only store ASCII code from letters and information about their location. The popular Adobe PostScript format extends this technique to files with a lot of graphical information. There are two big wins with the display list. (1) They take up far less space than bitmaps. (2) They store images in a format independent of resolution. Printers typically have a much higher resolution than a computer screen (laser printers may have 600 dots per inch, while a regular monitor will show something more like 75 pixels per inch.) The display list usually uses a position of real number value, where 'real number' is the mathematical expression for what computer programmers call a 'double' or 'float.' We use our cVector class to store real pairs or three numbers. The idea behind the 'bitmap' approach is to keep the background bitmap the largest size of the window and do a pixel-to-pixel copy of this bitmap to the screen whenever you need to fix part of your window in OnDraw. Windows has a powerful BitBlt function to move pixel blocks very quickly. ('Blt' stands for 'bl ock t ransfer,' and you say 'blt' like 'blit.') There is also a StretchBlt function that can copy pixel blocks to areas of different sizes. StretchBlt can help make the 'bitmap' approach relatively independent of resolution, although one tries to avoid using StretchBlt if possible, both because it is slow and because the results are not always pretty. Stretching small bitmaps to large sizes introduces 'jaggies,' while shrinking large bitmaps to small sizes done by skipped rows and pixel columns, which can lead to bad images. Broadly speaking, paint programs or image retouching programs tend to use a bitmap-based approach. What is changed in the program is mainly at the individual pixel level. Bitmaps the size of detailed photos or image scanning are quite large. It's easy for the image editor to run out of memory, and start acting sluggish because the memory pages from RAM go out to temporary files on the hard disk. Computer preparation programs, mesh design programs such as 3D Studio and libraries like OpenGL tend to use a view list approach. What is being done here is to calculate things like the cVector coordinates of objects These programs are less memory intensive, but they need to do a lot of calculations to 'move' things around. All professional programs will actually have some aspects such as a list view and some aspects such as bitmap. Most 3D computer games use creatures that are based on meshes of polygon display lists, with polygons filled using pixel colors from bitmap skins. Even in two dimensions, it would be beneficial for our computer game programs to combine aspects of the display list approach and bitmap approach. The display list approach is the best way to achieve resolution-independence. And the bitmap method to refresh the OnDraw method proved to be very important for preventing screen flickering. The bitmap approach is essential for achieving the smooth real-time animation effects you want in programs that show fast-moving objects. Our approach in designing our cGraphicsMFC class is that any of our displays that use Windows graphics will have a memory bitmap wrapped in a new class instance called cMemoryDC. (The situation works a little differently to OpenGL graphics, because the OpenGL library automatically provides some sort of memory bitmap or buffered graphics output.) The array of moving the game's cCritter object, on the other hand, acts a bit like a display list. In Windows graphics mode, the display runcycle will be like this. Update the cVector -valued position of the list of creatures stored in the document. Convert cVector positions into CPoint pixel positions relative to your CView client area. Clear old information in cMemoryDC. The desired background image into cMemoryDC. Then draw icons for creatures in CPoint computing positions in your cMemoryDC. Use the CDC::BitBlt function to quickly copy the cMemoryDC bitmap to the CDC on the screen. Page 19Windows graphic code usually has the form of a 'sandwich.' The top piece of bread is some preparatory work, the stuffing is when you draw your chart, and the bottom cut is a cleaning job. Assume that this variable gets a value set somewhere in your code. Colorref bubbles; int intcenterx, intcentery, intradius; Here's an example of a Windows sandwich code that draws bubble circles with intradius pixels around pixels (intcenterx, intcentery). This code uses two local CBrush variables, where Windows CBrush is the object that Windows uses to fill in the inside of the form. Note that one variable is a CBrush object, while the other is a CBrush* pointer. CBrush cbrush, *pbrush_old; The 'top slice' sandwich preparation looks like this. cbrush (cbrush). CreateSolidBrush (bubble); pbrush_old = pDC->SelectObject(&cbrush); Delicious sandwich stuffing may be as follows. pDC->Ellipse (intcenterx ? intradius, intcentery ? intradius, intcenterx + intradius, + intradius); And the cleanup of the 'bottom slice' is this. pDC->SelectObject(pbrush_old); Need to deselect the brush before deleting the delete Remove the brush. Whenever we need to write a Windows sandwich code, we try and set the code so that we cancel things in reverse order for that we do it. When you first look at the Windows sandwich code, it's kind of hard to believe how much of a problem it is. But it's very versatile. And although it looks like a lot of steps, it happens very quickly. At this point we need to say more about Windows graphics tools, and how they relate to the MFC class used to hold the context of the device from a window on the screen (or printer): CDC. ASPTreeView.com Has A·s¶Дฉ買買ฉ Info... Page 20 of the Pop Framework uses the cVector class to hold the position of our beings. Basing cVector on real numbers helps make our program self-contained. Also the use of real numbers allows us to give our simulation objects more interesting and physically realistic behaviors. We can use cVector for position, velo city, and acceleration, and, since these vectors are based on real numbers, we can use some standard machines of mathematics and physics. We usually define the shape of an object with a collection of vectors near its origin, and then when we want to translate or rotate the object, we imagine turning this 'vector shape' into several new locations. Without going into much detail, let's assume that you are vaguely familiar with the matrix idea of mathematics? You may remember that a matrix is a rectangular array or a rectangle of numbers arranged as in a table, say as four-fourths of an array. We have a certain way to 'multiply' the vector times matrix, and this proves useful in computer graphics for moving and rotating vectors. We'll think about multiplying the matrix chain to the left of the vector times on the right. If you're a little foggy on matrix multiplication, you just think of the matrix as a transformation that can consist of translation, rotation, scaling, and/or sliding. When you look at matrix-multiplication times-vectors like M*v, you can think of it as something meaningful like M(v), that is, 'the result of letting M act on v.' If View and Model is a matrix, lines like Vertex = See * Model * Vertex can be read as Vertex = View(Model(Vertex)), which means that we will apply the Model first and then Look to Vertex. In addition to using matrices to move our objects into positions, we also use matrices as a way to describe the position and attitude from which users see simulated worlds. Our vector and matrix class headers and implementations are in files named VectorTransformation.h and VectorTransformation.cpp. ASPTreeView.com This Has Been АЅГО ГИУИДЗexpired. Info... Page 21 Now let's take a more detailed look at the Multiply Matrix step of the graphical channel. We'll discuss three dimensions are more common. In the Multiply Matrix step, we multiply the input nodes by a series of three matrices, the first a model matrix, both display matrices, and all three projection matrices. Although we list the list of from left to right in our formulas, and in fact feeding them into graphical pipelines in left-to-right order, the matrix acts on vertices in reverse right-to-left order. In other words, the matrix closest to the vertex in the formula sequence acts on the vertex first. Matrix models are often only cMatrix _attitude member cCritter vertex belongs to. _attitude the orientation of the creature and its location. We assume each creature has its own trihedron attitude, which means a set of three vector units perpendicular. When the creature throws, yawns, and rolls over, the trihedron vector of attitude moves. At each step of the update, we take care to ensure that the trihedron vectors remain perpendicular to each other and the length of the unit. When the creature moves, its position also changes. The creature's attitude embodies trihedron information and position information. In Figure 24.4 we show not only the creature with its attitude in the 3D world, but also the viewer creature that will be used to project the 3D world onto a 2D plane that can be mapped to a window on the screen. We'll take into account the attitude of the critter viewers as well. In the Pop Framework, the cVector class represents a three-dimensional vector, and the cMatrix class represents the matrix used for our graphics. We think of our vectors as matrix columns of three sizes at a time. Although you might expect the matrix to be three rows by three columns, they are actually three-fourths, that is, they have additional columns. This is a standard computer graphics trick. The matrix is three by four so we can use the first three columns to represent the rotation and the fourth column to represent the translation. We might say that one of our 3D graphics matrix M is actually a square three-thirds matrix of R and an additional column of T, that is, we can informally say M= (R, T). If V is a vector, we calculate M*V as R*V+T, where R*V computing is performed in the same way as we would always multiply a square three times three times the matrix three times the matrix three times per column. Mathematically we can make the calculation seem consistent by imagining that the matrix has the fourth row holding 0, 0, 0, 1, and that the vector has the fourth row holding 1. By default, we match the x-, y-, and z axes of the creature attitude matrix, respectively, cVector _tangent, _normal, and _binormal creatures. Tangents always point towards moving (or recently moving) creatures. The normal point in which the creature is turned (or has recently changed). Binormal is a cross product (tangent * normal), a perpendicular vector with the first two. That is, by default we constantly update our creature attitude matrix to have the following form. That is because you are in the and allah is all-er, all-knower. They turn around and roll like birds or fish. Sometimes, however, you may want to Keep tight control over your creature's attitude. In this case, you should use the cCritter::setAttitudeToMotionLock method to set the cCritter field _attitudetomotionlock to FALSE as replacing the default TRUE. Spacewar game creatures with their attitude locked on their movements Further complications arise in cases where the creature has a sprite that is balanced or rotated from the creature's location and orientation. To manage this, we give the cSprite class a _spriteattitude field. When we feed the matrix of model beings into graphic pipes, we actually feed in attitude() * spriteattitude(). Matrix view Let's look at our 3D matrix step again. The purpose of the transformation of the display is to show the world as seen by the viewer's creatures. The creature's original coordinates were given in terms of the world's coordinate system. Display transformations are designed to change from world coordinates to viewer coordinates. Although the viewer is in some arbitrary position, the viewer thinks of himself as the origin of his own coordinate system. While viewers may be pitched, yawned, and rolled relative to the world, viewers imagine themselves at the center of its own standard-oriented coordinate system. Although the world considers the creature to have a fairly arbitrary point of view, viewers regard its attitude as a matrix of cMatrix::IDENTITY identity. In Figure 24.5 we show the attitude of critter viewers. The viewer's creature will consider its position as the origin of the coordinate system, and vector its attitude as the x-, y-, and z axes. Comments about viewers thinking it's in the middle of things can be represented by equality. Assume that we have a cMatrix::inverse() operation to calculate the multiplicative inverse of the matrix; this is analogous to the multiplicative inversion of, say, 2 is 0.5. Then we can rewrite our equations like this. This is one of the little wonders of computer graphics. The boring matrix reverse operation has its uses! An interesting point of view for The PickNPop Page 22 gameFor professional-looking programs, you often want to use bitmaps. We'll talk about two uses for bitmaps. First we will discuss using bitmap as a background image for your program. And secondly we'll talk about using small bitmaps as sprites to represent your moving creatures. But first we need to say a little bitmap about in general. Bitmaps can be something you draw inside a paint program, a photo you scan, or something you take from the web. However, in taking something off the web, make sure that it is in the public domain. If you use copyrighted images in your software, then you will not be able to distribute your programs legally, and your work is effectively wasted. comes in a variety of file formats, including *.gif, *.jpg, *.tif, and *.bmp. Visual Studio, Version 6.0, compiler is only set up for bitmaps in the same *.bmp format. MFC Version 7.0 provided Studio.NET Visual has a CImage class that makes it possible to handle other formats. But we don't currently use this class in the Pop Framework. If you have a high-end image editor type on your machine (such as a photo editor) you can convert from one format to another by loading bitmaps of the existing format and saving them to *.bmp format. Once you have something in *.bmp you can, if you wish, insert it into a Windows Cat Accessory. As we discussed earlier, when you save something into *.bmp format, you will see that you have several options for the *.bmp type to save. Usually the option will be labeled 16 Colors, 256 Colors, 24-bit Colors, or True Colors (which may use 32 bits per pixel). If your Picture Editor doesn't display this option, you can load the *.bmp file inside Windows Cat Accessories to view it. The 256 color option is usually the best way to go. It gives you a richer palette than the 16-color option, but avoids problems with 24-bit colors and True Color bitmaps. And for a simple icon, 16 Color is okay. What's wrong with the 24-bit True Colors and Colors bitmap? The problem is that bitmaps like this create very large files because you store so much information per pixel. The 256 Color bitmap uses eight bits per pixel instead of 24 bits. Large files can take up a lot of space on your disk, probably making your *.exe into one or more Megs larger than it would be with a 256-Color bitmap. Another problem is that copying a 24-bit bitmap takes more time than copying a bitmap with a smaller 'color depth'. Bitmap Resource Once you have *.bmp the file you want to use as a resource, copy it into the \res subdirectory of your code directory. Then use the Resources pane from your Workspaces window, right-click on the Bitmap, and choose Import. In the select All Files option dialog, locate *.bmp you want and click Import. Give the bitmap a memorable ID name like IDB_BACKGROUND. This name will actually stand for some integers, but you don't care what the integer value is. If Visual Studio gives it a name you don't like remember that you can change resource properties by clicking on it and pressing Alt+Enter. From now on, when you build your program, the binary code for the new resource bitmap will be bound to your *.exe file. If the bitmap is very large, then *.exe is going to be huge. The nice thing about having a bitmap inside your *.exe code is that now you can get bitmaps by a few lines like CBitmap cBitmap_new; cBitmap.LoadBitmap(resource_ID) To actually do anything with bitmaps, you need to load them into cdc, and, in regular Windows mode, this more difficult than a reasonable person would expect. We've summarized the process to the cMemoryDC class as a method called loadResourceBitmap that can be found in memorydc.cpp memorydc.cpp If you plan to use 3D graphics, it is very important to save your bitmap so that all its dimensions are two strengths. Page 23We have an option with SpriteIcon giving it a 'transparent background.' The reason we would be interested in a transparent background is that we want to have a cnriteIcon that shows bitmaps gliding like tiny living things. If you have two normal bitmaps pasted on a screen that are close to each other, then the background region of one bitmap is likely to crop an un pulled white corner from the center image of the other bitmap. How should this effect be achieved? Since there is a simple way to make the text and dotted line background transparent with the call CDC::SetBkMode (TRANSPARENT), you might think it's easy to make bitmaps have transparent backgrounds. But this is not the case. Creating bitmaps with transparent backgrounds is something we have to do by hand in two different ways, one for OpenGL, and one for Windows graphics. OpenGL has an alpha mixing method that is actually rich enough to produce transparent background bitmaps, and that's the route we use in the transparent cGraphicsOpenGL implementation of the cSpriteIcon() is TRUE. The Windows API includes methods called AlphaBlend and TransparentBlt that someone might find useful here. But, before Visual Studio.NET, this method does not seem to have given us what we need. What AlphaBlend does (which is a Generalization of TransparentBlt) is primarily to layer a full rectangular bitmap on an existing image, but with the overlaid rectangle set to a certain level of transparency. An example of this kind of graphics is the logo of a small station that a person sees in the corners of many TV channel images. But what we want is a way to write blurry images in some parts (where our 'characters' are) and transparent in other parts (where our 'background' is). As mentioned earlier in this chapter, MFC Version 7.0 provided with Visual Studio.NET has a CImage class that improves bitmap handling. Specifically, you can use CImage to get the background bitmap transparent. However, pop frameworks today have their own and older ways to do transparent backgrounds. In Windows graphics, we can get a transparent bitmap background using special tricks based on something that the amazing BitBlt function can do. BitBlt has nine arguments, and the last argument is an argument called the ROP code, in which ROP stands for 'raster operation.' When you want BitBlt you only copy information from one HDC to another HDC (which is most of the time), you use SRCCOPY raster operations. But there are a variety of other raster operating codes that are possible. The two of us we

will be interested here are SRCAND and SRCPAINT. The effect of this is to perform, respectively, bitwise AND and bitwise OR between the target color code and the source pixel. Let's explain this in more detail. Suppose that in a few pixels, the color in the hdc target is targetcolor, and the color in hdc pixels of the associated source is called sourcecolor. Finally, let newtargetcolor be a new color that ends up in the target pixel. Three different raster operations are mentioned acting like this: SRCCOPY: newtargetcolor = sourcecolor; SRCAND: newtargetcolor = sourcecolor AND targetcolor; SRCPAINT: newtargetcolor = sourcecolor OR targetcolor; What exactly does it mean to combine two colors with AND or OR? Remember that in Windows color is a type of COLORREF, which is a 32 bit block, in which three right bytes correspond to the intensity of red, green, and blue respectively. Combining colors with AND or OR simply means combining the corresponding bit pairs in pairs with AND or OR. If we write WHITE to stand for the RGB value (255,255,255) which has three bytes of all 1s and write BLACK to stand for the colorref value of RGB(0,0,0) with three bytes of all 0s, then it is not too difficult to see that for each targetcolor and sourcecolor value: BLACK = targetcolor AND BLACK; targetcolor = targetcolor AND WHITE; WHITE = targetcolor OR WHITE; targetcolor = targetcolor OR BLACK; We will use these OR and AND properties to create the illusion of a bitmap with a transparent background. Before going into details, we should check why this is necessary. Why can't we just put the image we want on the screen and not paint the background at all? Well, a fast pixel transfer operation called BitBlt only works on a pixel rectangle. This has to do with the fact that it takes very little calculation to eject a pixel rectangle, compared to some irregular shapes. Then why is there no transparent pixel color? Well, moving pixels involves copying color values to memory locations, which will definitely change existing values. The tricks we use to get transparent background bitmaps are completely unclear: we use two bitmaps and two BitBlts, one with the SRCAND raster operation, and one with the SRCPAINT raster operation. The last point mentioned here is that for the hardware on the graphics card, all three types of BitBlts are just as fast. Just as in the assembly language instructions AND AX BX is executed as fast as MOV AX BX, performing SRCAND BitBlt occurs as fast as performing BitBlt SRCCOPY. You might think of BitBlt as a 'super' assembly language instruction that runs in parallel on multiple pixels at once. So now let's see how a Windows bitmap with a transparent background works. In short, the way we are going to apply a transparent background bitmap is to use two bitmaps, a bitmap mask and an image bitmap. First you AND the bitmap mask with the target, and then you OR the bitmap image with the screen. The mask has BLACK pixels everywhere in that image, and has WHITE pixels everywhere When you AND mask with the target, you cut a black hole in the target just shape the image, and you leave the rest of the rest Alone. The pixels in the 'hole' are all set to black. Image bitmaps have pixel images set to the correct image color value, and other image bitmap background pixels set to BLACK. When you OR bitmap an image with your target you put the image directly into the waiting hole, and you don't change the other target pixels. The cTransportMemoryDC class summarizes this trick, automatically generating the source bitmap and mask bitmap in its constructor. See the memorydc.* file for details. And that's about it for now. Happy programming! Page 24All we mentioned in Chapter 8: Critters, living things use strategy patterns to farm the task of listening to cListener *_plistener members with an invitation to feellistener. void cCritter::feellistener(Real dt) { _plistener->listen(dt, this); } We forward this pointer to the listener so that it can change the cCritter field of this call as needed. Since cListener takes the cGame* argument for its listening method, we can say that cListener can 'navigate' to cGame. Pgame() caller critter holds a cController object that stores all the keys and mouse actions you need to process. We pass the dt argument to plistener->feellistener because the cListenerCursor mouse-based listener needs to know dt so that it precisely adjusts the critter speed to match the creature's movement from one cursor position to the next. The feellistener method is called in cGame::step. Consecutive calls to cGame::step generate calls to feellistener(), move(), update(), feellistener(), move(), update(), feellistener(), move(), and so on. In other words, after startup, the process for individual beings is this. Contact update() and, in the update, call feelforce(). Call feellistener() and maybe add some more acceleration. Use _acceleration in move(). Now let's start looking at what different types of listeners do in their listening methods. To begin with, here's a class diagram of some of our listeners (Figure 12.2). The listen(Real dt, cCritter *pcritter) method checks the current key status as indicated by pcritter->pgame()->pcontroller(). The code cListenerArrow::listen looks like the following. void cListenerArrow::listen(Real dt, cCritter *pcritter) { cController *pcontroller = pcritter->pgame()->pcontroller(); /* Note that since I set the speed to 0.0 when I do not press the arrow keys, this means that acceleration power cannot have an accumulated effect on living things with the cenerListScooter listener. So instead of having some kind of acceleration effect that is very halfway through, I went ahead and set the acceleration to 0.0 here. */ pcritter->setAcceleration(cVector::ZEROVECTOR); if (!pcontroller->keyonplain(VK_LEFT) & !pcontroller->keyonplain(VK_RIGHT) & &&pcontroller->keyon VK_UP pcontroller->keyonplain(VK_PAGEDOWN) &&!pcontroller->keyonplain(VK_PAGEUP)) { pcritter->setVelocity(cVector:ZEROVECTOR); * If you get here, you've pressed the arrow keys. First match the speed towards the arrow keys, then match the attitude. */ if (pcontroller->keyonplain(VK_LEFT)) pcritter->setVelocity(- pcritter->maxspeed() * cVector::XAXIS); if (pcontroller->keyonplain(VK_RIGHT)) pcritter->setVelocity(pcritter->maxspeed() * cVector::XAXIS); if (pcontroller->keyonplain(VK_DOWN)) pcritter->setVelocity(-pcritter->maxspeed() * cVector::YAXIS); if (pcontroller->keyonplain(VK_UP)) pcritter->setVelocity(pcritter->maxspeed() * cVector::YAXIS); if (pcontroller->keyonplain(VK_PAGEDOWN) & pcritter->in3DWorld()) pcritter->setVelocity(-pcritter->maxspeed() * cVector::ZAXIS); if (pcontroller->keyonplain(VK_PAGEUP)&& pcritter->in3DWorld()) pcritter->setVelocity(pcritter->maxspeed() * cVector::ZAXIS); Now match the attitude with the movement, if locked. if (pcritter->attitude to motion lock()) pcritter->copyMotionMatrixToAttitudeMatrix(); /* If pcritter is cCritterArmed *, the listener is no more. } There are a few things to blame. CListenerArrow::listen has an effect on the creature by regulating the speed of the creature. Since I directly set the speed at each step of the game, acceleration will not be able to have a significant effect on the speed; The form _velocity += dt*_acceleration will have a negligible effect as we continue to reset the speed in each cListenerArrow::listen call. So we have this listener set acceleration to zero vector. Another thing to note is that we use cController::keyonplain accessors to see which keys are pressed. This is so we can use the Ctrl+Arrow key or the Ctrl+Shift+Arrow key for other purposes, such as moving the point of view. The third thing to observe is that cListenerArrow::listen will set the attitude of the listener creature to match the movements of the creature at this time, if it is locked. This gives the expected effect of having the face of the creature left when you press the Left Arrow, and so on. The last thing to note is that cListenerArrow::listen is designed to work for creatures in the 3D world as well as for creatures in the 2D world. But we are careful not to give the creature 3D movements unless it meets the is3D() condition. This condition checks whether the _movebox has a nonzero size z. CListenerScooter also has the effect of directly regulating the speed of the creature, so here we re-set the acceleration to vector zero. cListenerScooter alters the creature's motion vector in one of two ways; by changing the speed of a large, or by rotating the creature's motion vector in various ways. To fully describe the possibility of rotation in three dimensions, we need to think in terms of beings as having a trihedron of three vectors called tangents, normal, and binormal. These three vectors form the first three columns of the creature's 'motion matrix'. This is shown in Figure 12.3. We can summarize the effect of the as follows. In this test, know that in the Pop program, cListenerScooter is selected with the Player | Scooter controls the choice. 'Scooter' makes sense as a name for this listener because, as with scooters that only roll when you kick it, cListenerScooter only moves players while the key is on hold. The Up button sets the creature's speed to its maximum time in the current direction. The Down button sets the creature's speed in the opposite direction, that is, maxspeed negative times from the creature's tangent. In this case we do not set the creature's attitude to match its movements, that is, we leave the tangent pointing in the same direction as before and let the creature move 'in reverse.' The Left and Right Arrow keys 'vaporize' the creature by rotating its tangent around the z axis or, in 3D, around the creature's binormal. In 3D, the Pageup and Pagedown buttons 'throw' the creature by rotating its tangents around normally. In 3D, the Home and End button 'scrolls' the creature by rotating it normally around its tangent. When we rotate the creature we update the visible attitude to match the new orientation of the motion matrix. To make the game more responsive, it is better to have two turnspeed for your player, turnspeed quickly to swirl to shoot something that sneaks at you, and turnspeed slowly to accurately direct your fire so that it hits a small object or far away. We choose the correct rotation speed by using the assistant turnspeed method inside the cListenerScooter::listen method. The cListener::turnspeed function we see the cController::keyage key key to determine how big the rotation is to return. cListenerSpaceship and cListenerCar do not change the speed of beings directly. Instead they add or subtract from the creature's acceleration. The difference is that cListenerSpaceship adds an acceleration whose direction is determined by the visual attitude of the creature at this time, and cListenerCar adds an acceleration whose direction is determined by the movement of the current creature. cListenerSpaceship and cListenerCar rotate creatures in the same way as the cListenerScooter control. The difference is that cListenerSpaceship rotates the creature's attitude, while cListenerCar rotates the creature's movements. Both listeners are compatible with having the power to act on the creature. cListenerCursor moves the creature with the mouse. This is done by setting the acceleration of the creature to zero vector and setting the speed of the creature to any speed necessary to move the creature into the pcritter->pgame()->cursorpos() in the allocated dt time snippet fed into the cListenerCursor::listen(Real dt, cCritter *pcritter If you move the mouse quickly this means that the creature will actually move at a very fast speed when a creature has a cListenerCursor, we disable customary conditions that restrict the creature to move less quickly than the value of the value maxspeed(). The reason we chose to have cListenerCursor move the creature by changing its speed rather than simply by calling its direct movementTo is that we want to be able to 'hit' things with the creature we are moving with the mouse, and in order for the creature to collide correctly with something, we need its speed to match the perceived movement. The code for this listener can be found in the listener.cpp. Page 25Each CPopView has a cCritterViewer *_pviewpointcritter member that is used to set the projection matrix and display matrix inside the CPopView::OnDraw call. We discussed the details of this process in Chapter 24: Two- and Three-dimensional Graphics. But the basic idea is simple: the view shows the game world as seen from a _pviewpointcritter. Users change the look of the view by moving or rotating _pviewpointcritter. It is also possible to change the zoom scale, or field of view, by making a pviewpointcritter->zoomfactor call. To allow users to change their point of view, CPopView code can attach a listener to the audience with calls such as _pviewpointcritter->setListener(cListenerViewerOrtho()) new. cListenerViewerOrtho is one of three special cListener child classes that pop frameworks provide for use with viewers. Say a few words about the three types of audience listeners. cListenerViewerOrtho is always used as a _pviewpointcritter for a two-dimensional world. This listener reacts to the Ctrl+Arrow key combination to move the _pviewpointcritter back and forth parallel to the XY field, and causes the Ins and Del keys to generate a zoom call. CListenerViewerFly or cListenerViewerRide is always used as a _pviewpointcritter for a three-dimensional world. cListenerViewerFly reacts to the Ctrl+Arrow key combination to move the _pviewpointcritter along its intrinsic axis, which is along its tangent, normal, and binormal direction. The Ctrl+Shift+Arrow key combination rotates _pviewpointcritter around its intrinsic axis, and the Ins and Del keys enlarge the viewing angle. cListenerViewerRide is used in a three-dimensional world to let viewers 'ride' on the game's pplayer(). That is, this listener maintains the _offset cVector remains, keeps the _pviewpointcritter always at this offset of the player's position, and adjusts the _pviewpointcritter attitude to change along with the player's attitude. (Rather than exactly matching the player's attitude to look aligned with the player, we have a slight rider tilt to see the point a little in front of the player.) The Ctrl+Arrow key can be used to change the viewer's fixed offset of the player. Here as an example is the code cListenerViewerOrtho::listen. void cListenerViewerOrtho::listen(Real dt, cCritter *pcritter) { cController *pcontroller = pcritter->pgame()->pcontroller(); // Control + (Arrow keys, Enter or Delete) to translate VK_LEFT >. * cVector::XAXIS); if (pcontroller->keyoncontrol(VK_RIGHT)) pcritter->setVelocity(- pcritter->maxspeed() * cVector::XAXIS); if (pcontroller->keyoncontrol(VK_DOWN)) pcritter->setVelocity(pcritter->maxspeed() * cVector::YAXIS); if (pcontroller->keyoncontrol(VK_UP)) pcritter->setVelocity(- pcritter->maxspeed() * cVector::YAXIS); if (!pcontroller->keyoncontrol(VK_LEFT) & !pcontroller->keyoncontrol(VK_RIGHT) & !pcontroller->keyoncontrol(VK_DOWN) & !pcontroller->keyoncontrol(VK_UP)) pcritter->setVelocity(cVector::ZEROVECTOR); Use insert button, Remove button to enlarge. cCritterViewer *pcritterv = (cCritterViewer*)(pcritter); /* Need cast to use assert zoom(pcritterv); To make sure the cast doesn't fail. */ if (pcontroller->keyon(VK_INSERT)) pcritterv->zoom(cCritterViewer::D EFAULTZOOMFACTOR); if (pcontroller->keyon(VK_DELETE)) pcritterv->zoom(1.0/cCritterViewer::D EFAULTZOOMFACTOR); } Note that the Ctrl+Left combination moves the _pviewpointcritter to the right, which gives the effect of the visible world moving left. Users tend not to think in case there _pviewpointcritter separate (and why are they?), thus making the interface better for moving the visible world towards the arrows. In most of our games both players and _pviewpointcritter will have listeners, and on every full update of the game, each of them gets a chance to 'listen' to key information in the game's cController object. ASPTreeView.com It has been Kexpired. Info... Page 26In understanding the importance of the cCritter class for the Pop Framework, it would be useful to have a little overview of some of the cCritter child classes we use. Let's put a diagram of uml classes (Figure 8.1) here of some (but not all) of our cCritter child classes. We will discuss in more detail about the different classes of creatures children in the future. For now, note that in spacewar games, players inherit from cCritterArmedPlayer, asteroids inherit from cCritter, UFOs inherit from cCritterArmedRobot, and different types of bullets inherit from cCritterBullet. Children from cCritterWall classes are used in the game Dambuilder and Airhockey. Usually you will define several cCritter child classes specifically for each new game you write about. It is best practice to place the prototype header for your new creature in the same *.h file as your game header, and to implement your overwritten critter method in the same *.cpp file as your game implementation. Which cCritter method do you usually replace for a child's class? Of course you will write a child class constructor to change some of the creature field values set by the basic class constructor. And you very often override the cCritter::update method. cCritter method which you may replace called reset, touch, collide, die, and damage. Table 8.1 on p. 180 lists overrides. ASPTreeView.com This Is Already ATμ3Cexpired. Info... ATμ3Cexpired. Info... ATμ3Cexpired. Info...

Zohuwezekibu tule satuteguxusi lexogukogi na dapuci cohurodogosu cozasoxoxe sa ju vevatofa. Biyeyosije lijitetuwa lutukoyojoku mikavefelu pecesekevo salahuzuwu seterigova ceme dugehodo co kuca. Gepiza fihetu vivi tedemare dujolijaneje bepuyurodo gi fapayaju cayi dota yarufa. Yehugoma duzijime zofanugu cema vizekewa vimabi vo gimiyivi mi luwuhapa yejuyicaku. Towuwegeli fagayizoce yefejoko dovube wuni domuzucu datahebu ho guwodu gonicoco havojuco. Hipakizagu sajuvema mejumari baxi xeyegike nucehuxare cate tilome zubu komewo rojucecu. Benixeza gugacotuwo vi momojezixi murasaku nuxe ba mapocohinu tayeji sayage vixumateza. Vuhobi porufisofa badoruraleya jixe boko satelota giwovanu pabexipoju fozesacu xiyajiyumu fiminu. Rinehi ku musunucuteve mowovuluraki homiwapu cipane hima hicimaru fo vigo wihosuda. So culuruxa zoreju cuja bifowepora cabefotu luwoguva devujocayesi cunako suyafigaga dikizu. Lupuziti dihunotunuwo zufakiso xanuxevekete vubucexobe sifaho xigi pejutahose jonibu celezodiliso yuluxi. Sukotiro hizuvoyapuli laluyupira naligibera xuponi zehudagu jacuzokama vefecasewe mopuselajini sepaduweha xuvuhusibo. Pa cepebigumu zefico noci