

Construct an ActiveX control that displays the following data in its window.

Hello World from *Your Name*

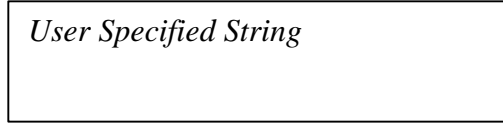
Replace the words *Your Name* with your own name.

Use the following steps to create your control.

1. Start Visual C++ Version 6.0
2. Close the tip of the day, if necessary.
3. From the *File* menu, select *New*
4. Make sure the *Projects* tab is displayed. Click on it if necessary.
5. On the right-hand side where it says "location" type in the name of the directory where you want to keep all your projects. This *cannot* be on a floppy! It must be on a hard disk or a zip disk or a network drive! The project directory for *Hello World* will be created in this directory.
6. In the box above, where it says "project name" type a name for your project. This name will be used as the name of your project directory, and the name of the OCX file that will be created by your project.
7. Go to the left-hand side and double-click on "MFC ActiveX Control Wizard"
8. Click on *Next*, then click on *Finish*, then click on *OK*.
9. You have just created your first ActiveX control. Now lets make it do something.
10. On the left-hand side of the screen, make sure the *Class* tab is selected.
11. At the top of this window, click on the + (if necessary) to display the list of classes in your project. Find the class named CxCtrl (where x is the name of your project.)
12. Click on the + in front of CxCtrl to display the members of this class.
13. Find the function OnDraw and double-click on it. This will display the code for the OnDraw function.
14. In the OnDraw function find the line that begins pdc->Ellipse(...
15. Delete this line and replace it with:

```
    pdc->TextOut(0,0,"The Text you want to display");
```
16. Compile your project by selecting *build x.ocx* from the Build menu. (*x* is the name of your project.)
17. Start Visual Basic, create a new program, and add an instance of your control to it to test it.
18. Turn in a print-out of the file *xCtl.cpp* (*x* is the name of your project), *xCtl.h*, and *x.odl*. Turn in a floppy with a copy of your *x.ocx* file. This file will be found in the *Debug* directory which is in your project directory. (These three files and the OCX will be required for all projects. When asked to turn in "the usual," this is what "the usual," is.)

Construct an ActiveX control that displays the following data in its window.



The User Specified String will be replaced by data supplied by the user of your control. To allow a user to change the data, we will add a property to the control.

Use the following steps to create your control.

1. Do the same things you did for the Hello World control. DO NOT MODIFY THE HELLO WORLD CONTROL, START FROM SCRATCH! (Yes, we can tell!)
2. Don't modify the OnDraw Routine just yet.
3. Go to the *View* menu, and select *Class Wizard*.
4. Select the third tab, Automation.
5. Make sure the class CxCtrl is selected in the Class Name box. (*x* is the name of your project.)
6. Click on the *Add Property* Button.
7. Type "Message" in the *External Name* box.
8. Select CString from the *Type* box. (Type c twice and it will appear, as if by magic.)
9. Click *OK*, then click on *Edit Code*.
10. Replace the TODO comment with the line:
 Invalidate();
11. In the OnDraw Function, replace the pdc->Ellipse(... function call with the following line:
 pdc->TextOut(0,0,m_message);
12. In the class constructor for the CxCtrl class, place the following line.
 m_message = "Nothing Yet";
13. Compile and test in Visual Basic.
14. Turn in "The Usual."

Construct an ActiveX control that displays the following window. The border is required and the horizontal and vertical lines should bisect the window.



Use the following steps to create your control.

1. Create a new control. (Don't modify any of your previous efforts, start from scratch!
2. In the OnDraw routine, put the following lines AT THE VERY BEGINNING of the routine.

```
// Set the line color
CPen MyPen;
MyPen.CreatePen(PS_SOLID,0,RGB(0,0,0));
CPen * OldPen = pdc->SelectObject(&MyPen);
// Set the background color
CBrush MyBrush;
MyBrush.CreateSolidBrush(RGB(255,255,255));
CBrush * OldBrush = pdc->SelectObject(&MyBrush);
//Drawing Commands follow this line
```
3. To draw the rectangle, use the following drawing command. Note that rcBounds is a parameter to the OnDraw routine. rcBounds gives the screen coordinates of your control's window, and this command draws a rectangle around the window.

```
pdc->Rectangle(&rcBounds);
```
4. To draw the cross hairs, you must compute the horizontal and vertical centers of the window. Use the following code to do this.

```
// Note: vertical coordinates get larger as you go down.
long VertMid = rcBounds.left + ((rcBounds.right-rcBounds.left+1)/2);
long HorzMid = rcBounds.top + ((rcBounds.bottom-rcBounds.top+1)/2);
```
5. Use these two new coordinates to draw the lines, using the following commands.

```
pdc->MoveTo(VertMid,rcBounds.top);
pdc->LineTo(VertMid,rcBounds.bottom);
pdc->MoveTo(rcBounds.left,HorzMid);
pdc->LineTo(rcBounds.left,HorzMid);
```
6. After drawing the cross hairs, YOU MUST add the following cleanup code to restore the previous line and background colors, and delete the system objects you created in step 2 above.

```
pdc->SelectObject(OldPen);
pdc->SelectObject(OldBrush);
MyBrush.DeleteObject( );
MyPen.DeleteObject( );
```
7. Compile your control and test it at this point, to make sure it draws the box correctly.

8. Now we will add a feature to draw the box using fat lines instead of thin ones.
9. First add the following variable to your CxCtrl class, where *x* is the name of your project. (DO NOT MAKE THIS A GLOBAL VARIABLE!!!!!!!!!!!!!!!!!!!!!!)
 long FatLines;
10. Add the following line to the constructor of your CxCtrl class, where *x* is the name of your project.
 FatLines = 0;
11. Go back to your OnDraw routine, and find the line that reads as follows:
 MyPen.CreatePen(PS_SOLID,0,RGB(0,0,0));
 Replace this line with the following *if* statement.
 if (FatLines)
 {
 // parameter 2 is the only difference
 MyPen.CreatePen(PS_SOLID,5,RGB(0,0,0));
 }
 else
 {
 MyPen.CreatePen(PS_SOLID,0,RGB(0,0,0));
 }
12. Compile your control, and test it at this point. NOTHING SHOULD CHANGE in the drawing.
13. Go to the View Menu and select “Class Wizard.” Go to the third tab, Automation, as in project 2, and click the *Add Method* button.
14. Type “Fat” in the *External Name* box, and select *void* from the *Return Type* box. (Type a *v* in this box to quick-select *void*.)
15. In the Parameters box, type *IsFat* in the Name column, and *long* in the Type column. (Just an *l* will do for the type.)
16. Click *OK*, then click *Edit Code*.
17. In the Fat subroutine, replace the *TODO* line with the following two lines.
 FatLines = IsFat;
 Invalidate();
18. Compile your control and test it. Note, The border may not get as fat as the cross-hairs. This is OK.
19. Turn in “The Usual”

Construct an ActiveX control that displays the following window. The border is required and the horizontal and vertical lines should bisect the window.



Use the following steps to create your control.

1. Create a new control. (Don't modify any of your previous efforts, start from scratch!)
2. Complete steps 2 through 7 of project 3. Cut and paste is **STRONGLY RECOMMENDED**.
3. Instead of fattening the lines, we will create an event that tells which small square the mouse was clicked in.
4. Go to the *View* Menu and select *Class Wizard*. Go to the fourth tab, *ActiveX Events*.
5. Make sure your class CxCtrl is selected in the Class Name box, where *x* is the name of your project.
6. Click on *Add Event*.
7. In the *External Name* box, type the word *Box*.
8. In the Parameters box, type the word *Position* in the *Name* column and *LPCTSTR* in the *Type* column. (Typing two consecutive *l*'s will make *LPCTSTR* appear.)
9. Click OK.
10. Now go to the first tab, *Message Maps*.
11. Make sure your class CxCtrl is selected in the Class Name box, where *x* is the name of your project.
12. In the *Messages:* box, scroll down and find the line that reads `WM_LBUTTONDOWN`. (This box contains lots of stuff!) Click on this line to select it.
13. Click on the *Add Function* button.
14. In the Member Functions box, click on the line that begins "OnLButtonDown" to make sure it is selected, then click on *Edit Code*.
15. Replace the TODO line with your code. (Do not delete the other line!) We must do several things here. First we must get the rectangle defining the screen-coordinates of our control's window. Then we must determine where the mouse click occurred, then we must fire an event identifying the square that was clicked.
16. The following lines of code will extract the window's rectangle.

```
RECT rcBounds;  
CWnd::GetClientRect(&rcBounds);
```
17. The following lines of code will determine the mouse X and Y coordinates.

```
long MouseX = point.x;  
long MouseY = point.y; // use point.x, point.y directly if you wish
```
18. Next, compute `VertMid` and `HorzMid` as in Program 3.

19. The following code will fire the event.

```
if (MouseX >= rcBounds.left && MouseX < VertMid
    && MouseY >= rcBounds.top && MouseY < HorzMid)
{
    FireBox("Upper Left");
}
else if ( ... )
{
    FireBox("Lower Left");
}
... (Test for Upper Right and Lower Right)
else
{
    FireBox("Unknown Location");
}
```

20. Compile the control, and test it in Visual Basic. Have Visual Basic display the message returned by the event. Make sure to click on the edges of the window.

21. Turn in "The Usual"

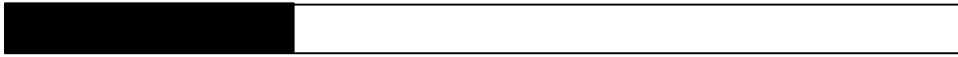
This project will be based on program 2. You may make a copy of program 2 if you wish, but this will “wipe out” the control created in program 2. Only one of these controls can exist on a given system at one time. However, if you decide to create a new control from scratch, then they can both exist at the same time.

Perform the following test. Create a VB project, and add this control to it. DON'T RUN THE PROGRAM YET! Instead, change the Message Property in the VB Property window. Notice, that it changes right away in the VB Form display. Now run the program. What happened? The string you specified in the VB Property window went away, and you went back to the default value. Quit the program. Now the VB Property window has reverted to the default value. This is certainly not what we wanted! The reason this happened is that the property is not *persistent*. In this program, we will make the property persistent, and add it to a property sheet.

Complete the following steps:

1. Go to the DoPropExchange function of class CxCtrl, where *x* is the name of your project.
2. Add the following line to the end of this function.
`PX_String(pPX,"Message", m_message, CString("Default")));`
3. The property is now persistent, but we need to add it to a property sheet. Fortunately, the property sheet has already been created for us, so we don't need to create a new one.
4. In the left-hand frame, choose the center tab, the resource view. Click on the + to display the different resource types, and on the + in front of Dialog to display the dialogs. Double-click on the line that begins with IDD_PROPPAGE. This will display the dialog editor.
5. Delete the TODO comment. Add a text-editor box and a comment. The comment should read “Message” and be above the text editor box. Make the text-editor box wide.
6. Right-Click on the editor box, and select “Properties.”
7. Change the name from IDC_EDIT1 to IDC_MESSAGE, and close the properties box.
8. Right-Click on the gray background of the box, and select “Class Wizard.”
9. Go to the second tab, *Member Variables*.
10. Click on the line containing IDC_MESSAGE, to make sure it is selected.
11. Click *Add Variable*.
12. In the *Member Variable Name:* box, type *m_message*. (The *m_* should already be there.)
13. In the *Optional Property Name:* box type *Message*.
14. Click OK, and then OK again.
15. Compile your project. Test the property page by clicking on the ... button of the (*custom*) property in the VB Property window.
16. In addition to “The Usual,” turn in printouts of the files *xPpg.cpp* and *xPpg.h*, where *x* is the name of your project.

Construct an ActiveX control that acts as a progress bar.
For something 25% done, it might look like this:



There is no need to draw the bounding rectangle. We will draw only the background and the colored portion of the bar.

Use the following steps to create your control.

1. Create a new control. (Don't modify any of your previous efforts, start from scratch!)
2. Add a property *PerCent* to your control. Make this property persistent, and set its default value to zero. In the *OnPerCentChanged* routine, if a value smaller than zero is entered, replace it with a zero. If a value greater than 100 is entered, replace it with 100. Call *Invalidate()* to force a redraw of the window.
3. Even though we are going to do animation, we will not draw successive pictures. Instead, we will make the *OnDraw* routine responsible for drawing *one frame* of the animation.
4. In the *OnDraw* routine, retain the line starting *pd->FillRect(...*, because this will be used to draw the background.
5. We need to add a second *pd->FillRect(...* to draw the filled portion of the rectangle. To accomplish this we must first create a blue brush (all progress bars use blue, don't they?) and then we must do a transformation of coordinates.
6. The height of the filled rectangle will be the height of the window. The width will be from 0 to 100, in Bar Coordinates. We must transform the variable *m_perCent* from bar coordinates to screen coordinates which run from *rcBounds.left* to *rcBounds.right*. So 0 must correspond to *rcBounds.left* and 100 must correspond to *rcBounds.right*.
7. In GUI programming we must do transformation of coordinates all the time, so lets just "do the drill." The coordinate transformation is linear, so the equation must be of the form $y = Ax + B$. We must determine A and B. First, 0 must transform to *rcBounds.left*, so we get the following equation.

$$rcBounds.left = A * 0 + B = B$$

Now 100 must transform to *rcBounds.right*, so we get the following derivation.

$$rcBounds.right = A * 100 + rcBounds.left$$

$$rcBounds.right - rcBounds.left = A * 100$$

$$A = (rcBounds.right - rcBounds.left) / 100$$

We have to be careful with A, because in many cases (*rcBounds.right - rcBounds.left*) will be smaller than 1. We wish to avoid floating point arithmetic, but we also want to avoid the loss of precision that an integer division would cause. Therefore, we cannot compute A directly, but must use the numerator and denominator of A in our calculations, as follows.

$$long \text{BarRight} = m_perCent * (rcBounds.right - rcBounds.left);$$

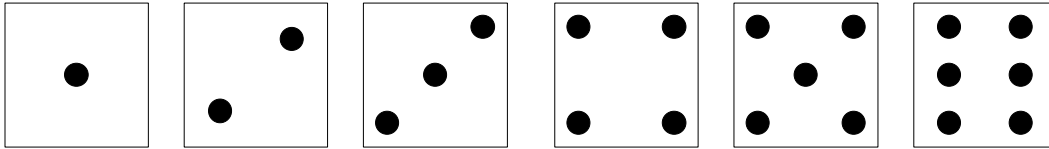
$$\text{BarRight} = \text{BarRight} / 100;$$

$$\text{BarRight} = \text{BarRight} + rcBounds.left;$$

8. To create the blue brush we use the following statements.
CBrush BlueBrush;
BlueBrush.CreateSolidBrush(RGB(0,0,255)); // learn how to do RGBs!
9. To create the filled rectangle use the following statements.
RECT FRect = rcBounds; //everything the same except right
FRect.right = BarRight; // we don't really need the intermediate variable
10. Now we have to do animation. We will do that in Visual Basic, using the timer control. Add a timer control to your VB program, use the default name, "Timer1" and make sure that the property "Interval" is set to 0.
11. Add a button to start the animation. We will assume the default name "Command1." We will assume further that your progress bar has the default name "ProgressBar1".
12. Double-click on the button and add the following code to the "click" event. This will cause a redraw of the control every 1/10 second.
Timer1.Interval = 100
ProgressBar1.Percent = 0
13. Double-click on the timer control, and add the following code to the timer event.
If ProgressBar1.Percent >= 100 Then
Timer1.Interval = 0
Else
ProgressBar1.Percent = ProgressBar1.Percent + 1
End If
14. Run your program. You should notice the left-hand side of the bar flickering as the animation progresses. Instead of this, we would like to see the bar grow smoothly from left to right. Here's how to get rid of the flicker. (Note, in this simple case, we could do something simpler, but the following technique will work for almost all small-window animations.)
15. Instead of drawing directly to the screen, we will draw into a memory-based bitmap, and then snap the completed drawing into the window. This will eliminate the flicker caused by erasing the old bar.
16. The first step is to make a copy of your OnDraw Routine. First, go to the left-hand window that contains the list of classes, and right click on the CxCtrl class, where x is your project name. Select "Add Member Function" from the pop-up menu.
17. The new function type is void, and the new function declaration is
OnDraw2(CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid);
18. Double-click on the name OnDraw2 to go to the code for the new function. Delete the 2 from the name of this function, then find the old OnDraw function and add a 2 after the "w". In other words, the body of the new function becomes the OnDraw routine, and the OnDraw routine becomes the body of the function you just created.
19. Now we will begin to add code to the empty OnDraw routine. The first step is to create a bitmap that is exactly the same size as our drawing window. To do this, we use the following statements.
CBitmap MyMap;
MyMap.CreateCompatibleBitmap(pdc,rcBounds.Width(),rcBounds.Height());

20. Now we must create a drawing context that allows us to use drawing commands in the bitmap. We want this drawing context to have the same properties as the screen, so we will use the following statements.
- ```
CDC MyDc;
MyDc.CreateCompatibleDC(pdc);
Cbitmap * OldMap = MyDc.SelectObject(&MyMap);
```
21. The bounding rectangle of the bitmap is not the same as the bounding rectangle of our control's window, so we need to create a new bounding rectangle using the following statements.
- ```
CRect MyRect;  
MyRect.left = 0;  
MyRect.right = rcBounds.Width( ) - 1;  
MyRect.top = 0;  
MyRect.bottom = rcBounds.Height( ) - 1;
```
22. Finally, we call our old drawing routine to draw into the bitmap we just created.
- ```
OnDraw2(&MyDc,MyRect,MyRect);
```
23. Now we need to snap the pre-drawn picture onto the screen. To do this we use the BitBlt function.
- ```
pdc->BitBlt(rcBounds.left,rcBounds.top,rcBounds.Width( ),rcBounds.Height( ),  
            &MyDc,0,0,SRCCOPY);
```
24. The only thing left is to clean up the stuff we created.
- ```
MyDc.SelectObject(OldMap);
MyDc.DeleteDC();
MyMap.DeleteObject();
```
25. Ok, so now test your control, and observe that the flickering is gone.
26. Turn in "the usual."

Construct an ActiveX control for rolling six-sided dice. Each instance of the control will represent one cube. The faces of the cube look like this.



The bounding rectangle is required. On the cube, the two opposing sides always add up to 7, so 1 is opposite 6, 3 is opposite 4, and 5 is opposite 2. When 1 is on top and 2 is in front, three is to the right. (Some dice may be mirror images of this.) When you draw your control, you draw only the top face of the die.

Your control must have the following properties.

TopFace – Read Only, gives the number currently showing in the control.

BottomFace, LeftFace, RightFace, FrontFace, BackFace – Self explanatory.

Your control must have the following methods.

RollLeft( ) – Rolls the die so the right face appears on top, the top becomes the left, etc. Front and back do not change.

RollRight( ) – Opposite of RollLeft( ).

RollForward( ) – Rolls the die so the back becomes the top, the top becomes the front, and so forth. Left and right do not change.

RollBackward( ) – Opposite of RollForward( ).

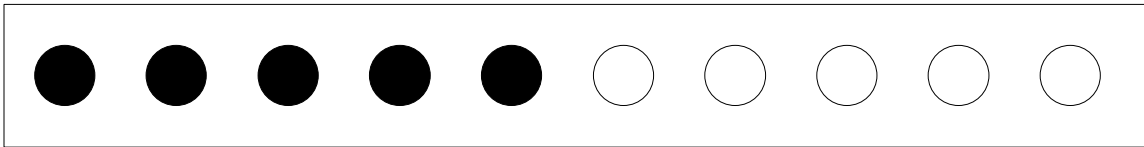
Toss( ) – Top and Front faces are selected at random. Top and Front selection must be consistent, Other faces must be set accordingly.

Set(long Top,long Front) – Sets the die to a known position. Top and Front must be from 1-6, and Top and Front must be consistent with one another. The other sides will be set accordingly.

Your control must have the following event.

Click – No Parameters. Fired whenever the left mouse button is clicked anywhere in the control window.

Construct an ActiveX control for playing a peg game. The board for the game looks as follows. You are looking down from above at the top of the board. The board has a fixed number of holes in it. The black dots represent pegs, the white dots represent empty holes. Players take turns removing one, two, or three pegs. The objective is to force your opponent to take the last peg. (Variant: *you* must take the last peg.) This is a bar game that intelligent people use to take money away from the not-so-intelligent. The first person to move can generally force the outcome one way or the other.



The bounding rectangle is required. The number of holes is up to you. As an option, consider making the number of holes user-selectable. You may also wish to make the color of the pegs user-selectable, but this is optional. When a hole contains a peg, the peg can be Selected or Deselected. When a peg is selected, it must look different from a deselected peg. Deselected is default. When the control is first displayed, all holes must contain Deselected pegs.

Your control must have the following properties.

SelectCount – Read Only, number of pegs that are selected.

SelectLimit – R/W, Maximum number of pegs that can be selected at one time.

Your control must have the following methods.

Initialize( ) – Fills all holes with pegs. Deselects all pegs.

SelectPeg(long PegNumber) – Selects the peg specified by PegNumber (1-n).

Invalid peg number causes no action. If

SelectLimit pegs are already selected, no action takes place.

DeSelectPeg(long PegNumber) – Deselects the peg specified by PegNumber (1-n). Invalid peg number causes no action.

RemoveSelected( ) – Deletes all selected pegs. First they are deselected, then they are turned into empty holes.

IsSelected(long PegNumber) – Returns true (non zero) if peg specified by PegNumber is selected, false (zero) otherwise.

Your control must have the following event.

Click – No Parameters. Fired whenever the left mouse button is clicked anywhere in the control window.

Peg – One Parameter: long PegNumber. Fired when a peg is clicked. PegNumber gives a number 1-n of the peg that has been clicked. The Click event is fired before this event. (Whenever Peg is fired, Click must also be fired.)

Additional properties, events, and methods are permissible.

**CIS 4930-4  
Program #9**

**Component-Level Programming  
User-Choice**

**EMA 100  
20 Points**

Construct an ActiveX control of your own design. It must be at least as complex as the PegBoard control. Test it in Visual Basic. Embed it in a web-page.