Conditional Compilation

Conditional compilation places the designators: #ifdef, #elif, #else, #ifndef and #endif around blocks of code, so #define statements can be used as compiler flags. I use it to introduce, and test, new blocks of code inside of existing blocks. Set the #define flags at the beginning of the program then the compiler will follow the logic path of your choice. It is a good way to test one block of code against another, to test if it is faster or if it generates errors.

I used conditional compilation to trap a bug, which occurred quite rarely, while I was working on Ecology . The bug was caused by a deletion from one of the linked meshes in the system. I wrote three different delete routines, then inserted conditional compilation commands to choose which of them to use. These deletion routines were called in a number of spots throughout the code, so I installed #ifdef #endif markers in many places. But, after all that work, and by setting and resetting those conditional flags, I was able to narrow my bug down to a few lines of code. Once I really looked at those lines of code, scribbled a few rough drawings of the changes to links during the deletion routine, I found my error. Conditional compilation helped me debug the code with only a little added work.

I have also used conditional compilation in preparation for adding a menu driven user interface. Since I write the code, I use I rarely create a user interface. I change a few variable's values, recompile, and run the new code using the edit – compile - run sequence as my user interface. However, I can turn large chunks of code on or off by using conditional compilation. Those chunks of code can then be rewritten as functions, called through a menu structure. Conditional compilation lets me test each module to prepare for adding a graphic user interface. Graphical user interfaces do make life easier for users of the application but they entail many extra lines of code. However, with a little judicious copying and pasting, a decent switch structure can be built to hold the menu pointers to the functions.

Conditional compilation can help you maintain a functioning system, while you are actively building and adding code. When you are ready to test the new modules, you turn them on with your #defines to begin. If you define them finely enough, you can test each module individually as you work. Conditional compilation, combined with commenting out broad swathes of code with /* */ pairs, will help you test your program quickly. That is the main reason I use // for my inline comments. It also determines my use of compilers. // is not strict C, but was added for C++ so using gcc may not work. If you compile with -wall set you will see those warnings. Switching your compiler from gcc to g++ will make some warnings will vanish. Compiling for no errors is obligatory, compiling for no warnings is even better. This explains why my make files so often call for g++ instead of gcc. Crossing the line between strict C and C++ is often as simple as those // comment lines.

http://ee.hawaii.edu/~tep/EE160/Book/chap3/subsection2.1.3.4.html

http://ee.hawaii.edu/~tep/EE160/Book/PDF/Chapter10.pdf

I selected the following chunk of code from threeD.c, a 3D graphics engine. We begin by defining PERSPECTIVE and WIRE. A scene can be viewed in either orthogonal, or perspective, projection. Perspective lets us see things more naturally, while orthogonal projection allows us to measure things more accurately. WIRE changes the color of the pen outlining each face in the object. If we comment out #define WIRE the outlines disappear. More accurately, the pen is no longer black but transparent. Look after the 2D banner in the code below. #ifndef means if WIRE is not defined, perform the following commands until we reach the #endif conditional statement. A blank pen is created in this case.

The next conditional is #ifndef LINE, which implements hidden surfaces, instead of the alternative wire frame objects. Next we have an example of nesting. #ifdef PERSPECTIVE controls how images are projected from 3D to 2D graphics. If PERSPECTIVE is defined, each vertex has every X and Y coordinate divided by the Z coordinate. If PERSPECTIVE hasn't been defined, we ignore the Z coordinate and display the image as X, Y lines. The flow of control starts with #ifdef PERSPECTIVE, takes either the first or the #else path, then ends at #endif, where program flow continues. However, you are still inside the #ifndef LINE path.

Once we are done with the #ifdef PERSPECTIVE clause, our next conditional is a #else, which matches the #ifndef LINE. #else covers the case where we want to display our objects using wire frames instead of faces. Project from 3D to 2D graphics by ignoring the Z coordinate, displaying our object using orthogonal projection. #endif closes the #ifndef LINE clause. Program flow continues with the following function, SelectObject().

Various paths through the code can be picked and tested quickly, using only a few #define settings at the top of a program listing. Very convenient and compact. Imagine how you could write large parts of your code inside a conditional compilation framework to test each path. When you are ready, replace the conditional commands with menu commands. Factor each conditional clause into a function then call those functions with your menus. The menu can run from a command-line or from a pull down menu of a GUI.

Conditional control commands are also good for creating a parallel testing apparatus. During your edit – compile – test cycle remove the comment // from the #define TEST setting. TEST controls the testing path through your code. Each major function should have a TEST section to exercise each of the function's parts. That way, you get test reports during your normal work cycle. Make sure to add all of the test code as you create libraries of functions. They won't take much space and are sure to help your sanity one day in the future.

```
#define PERSPECTIVE
#define WIRE        // display black frames around each shaded face
//#define LINE      // wire frame instead of shaded faces

…

////////////////////////////////////////////////////////////////////////
//  2D display Section  ////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

#ifndef WIRE     // don't display outline of faces
   hpen = CreatePen(PS_NULL, 1, RGB(255, 255, 255)); // draw blank outlines
#endif

#ifndef LINE     // display shaded faces
   for (int i=0; i<total; i++)    // shading, display loop
      {
         // display faces with this color brush
         // fill brush with color of this face
      hbrush = CreateSolidBrush( RGB((int) vFace[i].color.x,
                                     (int) vFace[i].color.y,
                                     (int) vFace[i].color.z) );
      hpenOld = SelectObject(bkHDC, hpen); // Select new pen and brush
      hbrushOld = SelectObject(bkHDC, hbrush);
      for (int j=0; j<vFace[i].vertices; j++) // project face
         {
#ifdef PERSPECTIVE
         // use a perspective projection
         if (vFace[i].vertex[j].z == 0.0)
            vFace[i].vertex[j].z = 0.000001;  // divide by non-zero
         vert[j].x = (int)((w*vFace[i].vertex[j].x/vFace[i].vertex[j].z) + w);
         vert[j].y = (int)((w*vFace[i].vertex[j].y/vFace[i].vertex[j].z) + w);
#else
         // an orthogonal projection instead
         vert[j].x = (int) vFace[i].vertex[j].x;
         vert[j].y = (int) vFace[i].vertex[j].y;
#endif
         // end of PERSPECTIVE projection
         }
      Polygon(bkHDC, vert, vFace[i].vertices);      // draw face
      SelectObject(bkHDC, hbrushOld);
      DeleteObject(hbrush);  // create GDI in loop, destroy GDI within loop!!
      }                       // prevent memory leaks
#else                        // #define LINE start
   // display wireframe instead
   for (int k=0; k<total; k++)
   {
   MoveToEx(bkHDC, (int) vFace[k].vertex[0].x + d.x,
                   (int) vFace[k].vertex[0].y + d.y, NULL);
   for (int j=1; j<vFace[k].vertices; j++)
      LineTo(bkHDC, (int) vFace[k].vertex[j].x + d.x,
                    (int) vFace[k].vertex[j].y + d.y);
      LineTo(bkHDC, (int) vFace[k].vertex[0].x + d.x,
                    (int) vFace[k].vertex[0].y + d.y);
   }
```

```
#endif     // end of wireframe
           // end of #define LINE code

SelectObject(bkHDC, hpenOld);
DeleteObject(hpen);
      // prevent memory leaks
BitBlt(hdc, 0, 0, WINDOWX, WINDOWY, bkHDC, 0, 0, SRCCOPY);
ReleaseDC(hwnd, hdc);
}          // end of forever loop
```