```
// c_applic.cpp
//          The "application" part of my program. This is concerned with
//          initial creation of windows and the capture of any command-line
//          or other options.
//
//     Copyright (C)/©/¸ Codemist Ltd, 1995-2002

/*
 * This code may be used and modified, and redistributed in binary
 * or source form, subject to the "CCL Public License", which should
 * accompany it. This license is a variant on the BSD license, and thus
 * permits use of code derived from this in either open and commercial
 * projects: but it does require that updates to this code be made
 * available back to the originators of the package.
 * Before merging other code in with this or linking this code
 * with other packages or libraries please check that the license terms
 * of the other material are compatible with those of this.
 */


/* Signature: 2253e25b 30-Sep-2002 */



#include "cwin.hpp"

extern "C" {
extern int Lstop(int a, int b);
}

//
// Windows enters my code at WinMain, but with MFC I arrive via all sorts
// of jolly initialisation code.  To support old code I read the command
// line that invoked me, and parse it into words which I store in argv[],
// much as per the regular C startup process.
//

char programName[64];
char *cwin_full_program_name;
static int argc;
static char **argv;

static int set_up_argv()
// This sets up argc and argv[] as expected for a regular C application.
// It arranges that argv[0] is an unqualified name, forced into upper case.
// Ie argv[0] does not have a path-name on the front of it or any ".EXE"
// suffix after it.
// I return a flag that indicates whether "--" was found among the arguments.
// (and now I will return the same flag if "-f" was among the arguments)
{
    int doubledashFound = 0;
    int i = 0, c, len = 0;
    char *w = GetCommandLine();
// The string I obtained there may be in UniCode, but I will suppose that it
// does not contain any funny characters.  In particular I will demand
// that this module is compiled with Unicode mapped onto ordinary 8-bit
// chars.
    char *w1 = w, *argbuf;
// I scan the command line once to assess its length. I treat any item
// that STARTS with a double-quote mark as running on until the following
// double quote (even if there are blanks in the way), and in that case I
// remove the quotes before passing on to the user's program. Something
// like this appears to be essential for dealing with Windows-95 with
// its long filenames: it puts the first item on the command line (the name
// of the program that is being executed) in double quotes in case it is long
// and in case it has embedded whitespace, and unless I strip those quotes
// later on bits of code crash. Note now the subtle distinction between
//            -W "yy"
```



```
// and        -W"yy"
// where -W is some option that user code tries to interpret. In the first
// case the quotes are gobbled up and removed here, in the second they
// remain for the user to see. Note that
//            -W"yy is here"
// returns three words, two of which have embedded quote marks. It is probably
// impossible to guarantee to win all cases here!
    do
    {   while ((c = *w++) == ' ' || c == '\t');// Blank at start of an item
        if (c == 0) break;
        i++;                                 // Count of words
        if (c == '"')                        // items in double quotes?
        {   c = *w++;
            while (c != 0 && c != '"') c = *w++, len++;
            if (c == '"') c = *w++;
        }
        else while (c != 0 && c != ' ' && c != '\t') c = *w++, len++;
    } while (c != 0);
// Now I can allocate space for the argument vector and a copy of the data.
// I grab a little more space than I am going to use as a matter of caution.
    argv = (char **)malloc((i+1)*sizeof(char *));
    argbuf = (char *)malloc(i+len);
    argc = 0;
    if (argv==NULL || argbuf==NULL) return 0;
// Re-scan the command line copying characters into buffers
    w = w1;
    do
    {   while ((c = *w++) == ' ' || c == '\t');
        if (c == 0) break;
        argv[argc++] = argbuf;
        if (c == '"')   // I strip the quotes while I tokenise
        {   c = *w++;
            while (c != 0 && c != '"') *argbuf++ = c, c = *w++;
            if (c == '"') c = *w++;
        }
        else while (c != 0 && c != ' ' && c != '\t') *argbuf++ = c, c = *w++;
        *argbuf++ = 0;
        if (argv[argc-1][0] == '-' &&
            (argv[argc-1][1] == '-' ||
             argv[argc-1][1] == 'f' ||
             argv[argc-1][1] == 'F')) doubledashFound = 1;
    } while (c != 0);
// Put a NULL pointer at the end of argv[], just to be safe
    argv[argc] = NULL;
// Now I want to trim argv[0] so that even if it started with a full
// path or with an extension (eg. "\bin\csl.exe") it is passed on trimmed
// down to just its root (eg. "csl" in the above case).  This string will
// be left in programName too.
    w = w1 = argv[0];
    cwin_full_program_name = NULL;
    while ((c = *w++) != 0)
    {   if (c == '\\') w1 = w;
// I take the view that if argv[0] contains a ":" character then it can be
// presumed to be a fully rooted file name, including a drive specification.
// In such cases I will use it when I want the full name of the executable
// I am running. Well I will also require in that case that it should end
// in a ".exe" suffix. This final test is certainly needed under Windows
// NT 4.0 when one launches CSL from a command line but specifying
// a drive to find it on. What I want is the text
//     "D:\xxxx.exe"
        else if (c == ':' && *w=='\\')
        {   i = strlen(w)-4;
            if (i > 0 && w[i]=='.' &&
                tolower(w[i+1])=='e' &&
                tolower(w[i+2])=='x' &&
                tolower(w[i+3])=='e')
                    cwin_full_program_name = argv[0];
        }
```

```
    }
    if (*w1 == 0) w1 = "CWIN";  // Final char of argv[0] was '\': use default
    w = programName;
    while ((c = *w1++) != 0 && c != '.') *w++ = toupper(c);
    *w = 0;
    argv[0] = programName;
    if (cwin_full_program_name == NULL)
// Now I would like to get a full path to the program name... The
// SearchPath function looks first in the directory from which the
// application was fetched, and provided that the ".exe" extension
// I specify here is correct the file really ought to be located!
    {   int nameLength = SearchPath(NULL, programName, ".EXE", 0, argbuf, &w);
// There is one critically important case where "SearchPath" will fail here,
// and that is when the program has been started from a debugger and the
// real name of the program is just not available.  In that case tough
// luck, you will have to make resources available by some means NOT
// dependent on the program name or the directory it lives in.  Maybe in some
// cases with DOS extenders the program will appear to have been loaded from
// a directory distinct from the one that the obvious ".EXE" file lives in.
// In those cases I had better hope that argv[0] gave me a completely
// rooted file name.
        cwin_full_program_name = (char *)malloc(nameLength+1);
        if (cwin_full_program_name == NULL)
            cwin_full_program_name = "cwin.exe";
        else
        {   if (SearchPath(NULL, programName, ".EXE",
                           nameLength+1, cwin_full_program_name, &w) == 0)
                cwin_full_program_name = "cwin.exe";
        }
    }
    return doubledashFound;
}


CTheApp theApp;

CString mainWindowClass;

UINT clipboardformat;

BOOL CTheApp::InitInstance()
{
// I find the explanations about m_nCmdShow and GetStartupInfo jolly
// confusing! However the code as given here will be tested with CSL launched
// from a command line with
//       start csl ...
//       start /min csl ...
//       start /max csl ...
// and   start csl -- logfile.log
//
// The last of these ought to start CSL minimised even if /min or
// /max is given as well.
//
// Initial testing is on NT. Next I will need to try Windows 95. The
// issue of win32s (on Windows 3.1x) will be gently ignored now.
    int nShow = m_nCmdShow;
#ifdef __WATCOM_CPLUSPLUS__
#if __WATCOM_CPLUSPLUS__ > 1060
// Oh calamity. The following 3 lines seem necessary to make the initial
// window behave properly, but when I inserted them I had reports that the
// code would not link on a system with just the previous version of
// Watcom C. While I investigate this I will try to disable this facility
// unless I have version 11.0 installed, and maybe this will allow us to
// keep moving forward.
    STARTUPINFO su;
    GetStartupInfo(&su);
    if (su.dwFlags & 1) nShow = su.wShowWindow;
#endif
```



```
#endif

// I will grab information out of the registry as soon as the application
// is started.

    SetRegistryKey("Codemist");   // Use registry rather than ".ini" file
    int left   = GetProfileInt("MainWindow", "ScreenLeft",   -1000000);
    int width  = GetProfileInt("MainWindow", "ScreenWidth",  -1000000);
    int top    = GetProfileInt("MainWindow", "ScreenTop",    -1000000);
    int height = GetProfileInt("MainWindow", "ScreenHeight", -1000000);
    int fsize  = GetProfileInt("MainWindow", "FontSize",     0);
    int fweight= GetProfileInt("MainWindow", "FontWeight",   0);
    int linel  = GetProfileInt("MainWindow", "LineLength",   -1000000);
    CString fname(GetProfileString("MainWindow", "FontName", "Courier New"));

    int doubledashFound = set_up_argv();

    mainWindow = NULL;

    mainWindowClass = ::AfxRegisterWndClass(CS_DBLCLKS,
                          LoadStandardCursor(IDC_ARROW),
                          (HBRUSH)(COLOR_WINDOW+1),
                          LoadIcon("CWIN"));
/*
 * I introduce a private clipboard format here. It will be just like
 * simple text (to start with) except that certain control characters
 * will be used to separate off the places that prompt strings occus. This
 * can be exploited in PASTE operations so that prompts issued by this
 * system do not get re-entered when previous input is copies and pasted.
 */
    clipboardformat = RegisterClipboardFormat("Codemist Text");
/*
 * clipboardformat is left zero if anything went wrong, so in such cases
 * I must not attempt to use it.
 */

    if ((mainWindow = new CMainWindow()) == NULL) return FALSE;
    m_pMainWnd = mainWindow;
    CClientDC dc(mainWindow);
    mainWindow->windowFonts.InitFont(&dc, (LPCTSTR)fname, fweight, fsize);
// If there was "--" given as an argument I start off with the window
// minimized. This is because I then expect an application to use this
// flag to enable output to a file rather than to the screen.
    WINDOWPLACEMENT wp;
    mainWindow->GetWindowPlacement(&wp);
// Here I will see where the window is about to be placed, and adjust its
// width (and maybe its left hand side) in an attempt to make the client
// are just big enough for 80 columns. I am a bit unhappy about the
// calculation here using system metrics, and have added in one more
// CXBORDER as a fudge to bring experimental reality on MY system into
// line. Hope it is OK on other systems and configurations too.
// The "+5" on the end is an attempt to leave room for a caret to the right
// of the last sensible character on a line...
    if (left!=-1000000 && width!=-1000000 &&
        top!=-1000000 && height!=-1000000 &&
        linel!=-1000000)
    {   mainWindow->SetWindowPos(NULL, left, top, width, height,
            SWP_NOACTIVATE | SWP_NOREDRAW | SWP_NOZORDER);
        cwin_linelength = linel;
    }
    else
    {   int screenWidth = GetSystemMetrics(SM_CXSCREEN);
        RECT *wr = &wp.rcNormalPosition;
        int left = wr->left;
        int cwidth = 80*mainWindow->windowFonts.HCourier.across['X'] +
                     3*GetSystemMetrics(SM_CXBORDER) +
                     2*GetSystemMetrics(SM_CXFRAME) +
                     GetSystemMetrics(SM_CXVSCROLL) + 5;
```

```
// Try to get the whole window onto the screen.
        if (left + cwidth > screenWidth)
        {   left = screenWidth - cwidth;
            if (left < 0) left = 0;
        }
        mainWindow->SetWindowPos(NULL,
            left, wr->top,
            cwidth, (wr->bottom - wr->top),
            SWP_NOACTIVATE | SWP_NOREDRAW | SWP_NOZORDER);
        cwin_linelength = 80;
        WriteProfileInt("MainWindow", "ScreenLeft",   left);
        WriteProfileInt("MainWindow", "ScreenWidth",  cwidth);
        WriteProfileInt("MainWindow", "ScreenTop",    wr->top);
        WriteProfileInt("MainWindow", "ScreenHeight", wr->bottom-wr->top);
        WriteProfileInt("MainWindow", "LineLength",   cwin_linelength);
    }
    mainWindow->ShowWindow(doubledashFound ? SW_SHOWMINNOACTIVE : nShow);
    mainWindow->UpdateWindow();

    return TRUE;
}

class CenteredDialogBox : public CDialog
{
public:
    void CreateAndDisplay(HGLOBAL h);
};

void CenteredDialogBox::CreateAndDisplay(HGLOBAL h)
{
    InitModalIndirect(h);
    DoModal();
}

// In-store dialog-box templates need some of their strings in 16-bit
// Unicode form.  This code stretches out a simple string. It also round
// the length of the data written to a multiple of 8 bytes, which seems to
// be an unpublished (?) requirement for the dialog box template structures.

static LPWORD WidenString(LPWORD p, char *q)
{
    int n = 0;
    while ((*p++ = *q++) != 0) n++;
    if (n & 1) *p++ = 0;
    return p;
}

// The following function fills in details about one control within a
// dialog box template.

static LPWORD PlantDlgItem(LPWORD p3, int x, int y, int cx, int cy,
                           int id, DWORD style, int type, char *text)
{
    LPDLGITEMTEMPLATE p2 = (LPDLGITEMTEMPLATE)p3;
    p2->x = x; p2->y = y; p2->cx = cx, p2->cy = cy;
    p2->id = id;
    p2->style = style;
    p3 = (LPWORD)(p2 + 1);
    *p3++ = 0xffff;
    *p3++ = type;
    int n = 1;
    while ((*p3++ = *text++) != 0) n++;
    if (n & 1) *p3++ = 0;
    *p3++ = 0;
    return p3;
}

// I make the "ABOUT" dialog box from an in-memory template, and
```



```
// this makes it possible to make the text that is included depend on
// strings that the user can potentially reconfigure.  The strings put here
// are somewhat generic.   Note also that if a user hits the HELP menu
// during system start-up before the regular user code at main() has been
// entered than the messages shown here will appear, even though later on
// the user's properly selected messages will be the ones that show up. I
// think that on balance I almost count that to be a positive advantage! It
// means that the "CWIN" information and credits are at least just about
// available to all users!

char about_box_title[32]       = "About CWIN";
char about_box_description[32] = "The CWIN window driver";
char about_box_rights_1[32]    = "Copyright Codemist Ltd.";
char about_box_rights_2[32]    = "A C Norman      1994-6";

void CTheApp::OnAbout()
{
    HGLOBAL h = GlobalAlloc(GMEM_ZEROINIT, 1024);
    if (!h) return;
    LPDLGTEMPLATE p1 = (LPDLGTEMPLATE)GlobalLock(h);
    WORD *p0 = (WORD *)p1;
    p1->style = WS_POPUP | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME;
    p1->cdit = 5;
    p1->cx = 167; p1->cy = 86;
    LONG units = ::GetDialogBaseUnits();
    int dlgx = units & 0xffff, dlgy = (units >> 16) & 0xffff;
    p1->x = ((4*mainWindow->clientWidth)/dlgx - p1->cx)/2;
    p1->y = ((8*mainWindow->clientHeight)/dlgy - p1->cy)/2;
    LPWORD p2 = (LPWORD)(p1 + 1);
    *p2++ = 0;        // no menu
    *p2++ = 0;        // a predefined box class
    p2 = WidenString(p2, about_box_title);
    p2 = PlantDlgItem(p2, 0, 4, 167, 8, -1,
        WS_CHILD | WS_VISIBLE | SS_CENTER, 0x0082, about_box_description);
    p2 = PlantDlgItem(p2, 0, 45, 167, 8, -1,
        WS_CHILD | WS_VISIBLE | SS_CENTER, 0x0082, about_box_rights_1);
    p2 = PlantDlgItem(p2, 0, 53, 167, 8, -1,
        WS_CHILD | WS_VISIBLE | SS_CENTER, 0x0082, about_box_rights_2);
    p2 = PlantDlgItem(p2, 74, 22, 0, 0, -1,
        WS_CHILD | WS_VISIBLE | SS_ICON, 0x0082, "CWIN");
    p2 = PlantDlgItem(p2, 66, 65, 32, 14, IDOK,
        WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, 0x0080, "OK");
    GlobalUnlock(h);
    CenteredDialogBox dlg;
    dlg.CreateAndDisplay(h);
    GlobalFree(h);
}

// When I want to pop up a box that says "Press OK to exit" I make the
// structure that defines the dialog box here in memory rather than putting
// it into my resource file.  The reason for taking this step is that it
// allows to to keep the resource file as spartan and simple as possible.
// It also provides a place for me to ensure that the dialog box is central
// in the area that my window occupies.

static void DoFinishBox()
{
    HGLOBAL h = GlobalAlloc(GMEM_ZEROINIT, 1024);
    if (!h) return;
    LPDLGTEMPLATE p1 = (LPDLGTEMPLATE)GlobalLock(h);
    WORD *p0 = (WORD *)p1; //DEBUG
    p1->style = WS_POPUP | WS_CAPTION | WS_SYSMENU | DS_MODALFRAME;
    p1->cdit = 2;
    p1->cx = 95; p1->cy = 52;
// I want the box to appear in the centre of where my window is. This
// causes extra fun because of the special coordinate system used with
// dialog boxes - I have to convert units.
    LONG units = ::GetDialogBaseUnits();
```

```
    int dlgx = units & 0xffff, dlgy = (units >> 16) & 0xffff;
    p1->x = ((4*theApp.mainWindow->clientWidth)/dlgx - p1->cx)/2;
    p1->y = ((8*theApp.mainWindow->clientHeight)/dlgy - p1->cy)/2;
    LPWORD p2 = (LPWORD)(p1 + 1);
    *p2++ = 0;       // no menu
    *p2++ = 0;       // a predefined box class
    *p2++ = 0;       // no title
    p2 = PlantDlgItem(p2, 1, 10, 94, 12, -1,
        WS_CHILD | WS_VISIBLE | SS_CENTER, 0x0082, "Press OK to exit");
    p2 = PlantDlgItem(p2, 28, 23, 40, 14, IDOK,
        WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, 0x0080, "OK");
    GlobalUnlock(h);
    CenteredDialogBox dlg;
    dlg.CreateAndDisplay(h);
    GlobalFree(h);
}


// Here I override the Run member of my application so that I take control
// of the way in which the underlying window system is polled. This is
// somewhat delicate!  The code here has to be a close enough shadow of
// what MFC does that I do not cause conflict.

int cwin_pause_at_end;

int CTheApp::Run()       // Main running routine until application exits
{
// If I had not managed to open a main window then I should give up.
    if (m_pMainWnd == NULL
//        && AfxOleGetUserCtrl()     // Embedding or Automation invocation?
        ) AfxPostQuitMessage(0);
// The message handler needs to access m_msgCur which is a private member
// of the class, so I can not use just (theApp.m_msgCur) to get at it. To
// work around the problem I just dump a reference to it in the variable
// msgPtr.
//+++ This was OK with Visual C++ version 5, but by version 7 (in
//+++ Visual Studio .NET) the m_msgCur member no longer exists, and so this
//+++ code needs to be rebuilt! Oh dear, especially since the way it
//+++ was breaking private visibility is an indication that the original
//+++ version was something of a crock!     ACN September 2002
    msgPtr = &m_msgCur;
// Now the real fun!  I call cwin_main() which fires up my application code
// Remember that cwin_main() should either return to me or do a cwin_exit()
// and it should NOT call exit().
    cwin_pause_at_end = FALSE;
    int returnCode;
//    try {
    returnCode = cwin_main(argc, argv);
//    }
//    catch (int rc) { returnCode = rc; }
    if (cwin_pause_at_end)
    {   cwin_maximize();
        cwin_ensure_screen();
        DoFinishBox();
    }
    if (m_pMainWnd != NULL) delete m_pMainWnd;
    m_pMainWnd = mainWindow = NULL;
    int returnCode1 = ExitInstance();
    if (returnCode1 > returnCode) returnCode = returnCode1;
    if (m_pMainWnd != NULL) m_pMainWnd->SendMessage(WM_CLOSE);
// Although I go to some trouble to collect a return code here I find such
// things somewhat unhelpful under Windows, and so at the last minute I
// check away the information and return zero.
    return 0;              // returnCode;
}

//void cwin_exit(int r)
//{
```



```
//    throw r;
//}

BEGIN_MESSAGE_MAP(CTheApp, CWinApp)

    ON_COMMAND(IDM_HELPCONTENTS,   OnHelpContents)
    ON_COMMAND(IDM_HELPSEARCH,     OnHelpSearch)
    ON_COMMAND(IDM_HELP_ON_HELP,   OnHelpUsing)
    ON_COMMAND(IDM_ABOUT,          OnAbout)
#ifdef DEMOVERSION
    ON_COMMAND(IDM_EUPRICES,       OnEUPrices)
    ON_COMMAND(IDM_WORLDPRICES,    OnWorldPrices)
    ON_COMMAND(IDM_ORDERFORM,      OnOrderform)
#endif
    ON_COMMAND_RANGE(IDM_DYNAMIC_ITEMS, IDM_LAST_DYNAMIC, OnDynamic)

END_MESSAGE_MAP()



// At various times I will want to go back and poll the window manager
// to ensure that mouse activity is responded to, the screen is re-drawn and
// other applications get a share of the CPU. To do that I will arrange that
// 'cwin_poll_window_manager()' is called from time to time in the middle of
// whatever else I am doing.  This grabs a message from the window manager
// and dispatches it to whatever handler is relevant.

static void timer_processing()
{
    SYSTEMTIME t1;
    GetSystemTime(&t1);
//-        if (t1.wHour != lastFlushTime.wHour ||
//-            (t1.wMinute - lastFlushTime.wMinute)*60 +
//-            (t1.wSecond - lastFlushTime.wSecond) > 3)
//-            cwin_almost_ensure_screen();
    if (!theApp.mainWindow->leftSetByUser)
    {
// Here I arrange to update the title-bar clock about once every 5 secs. It
// seems that every second it too frequent, especially since it often flashes
// the title-bar while re-drawing it.  But 10 seconds is too long and lets
// the user feel things may be stuck.
// If the user explicitly sets any value in the left part of the title bar
// then this action is disabled. I do not set titleUpdateTime at the start
// of a run but that does not matter - it will fall into line within a few
// seconds whatever its initial value (however junky) is.
        if (theApp.mainWindow->titleUpdateTime.wHour != t1.wHour ||
            theApp.mainWindow->titleUpdateTime.wMinute != t1.wMinute ||
            theApp.mainWindow->titleUpdateTime.wSecond/5 != t1.wSecond/5)
        {   theApp.mainWindow->titleUpdateTime = t1;
            theApp.mainWindow->titleUpdateTime.wSecond =
                5*(theApp.mainWindow->titleUpdateTime.wSecond/5);
            theApp.mainWindow->cwin_display_date();
        }
    }
}

// This task will busy-wait in its idle-state watching time go by and
// occasionally updating the screen etc.

BOOL CTheApp::OnIdle(LONG lCount)
{
    BOOL r = CWinApp::OnIdle(lCount);
    if (!r) timer_processing();
    return r;
}


void cwin_poll_window_manager(void)
```

```
{
// If the application has registered an idle-time handler then that gets
// invoked until it has finished or until a real message arrives whenever
// I call cwin_poll_window_manager().  I also process ALL the window messages
// that have stacked up and only go back to the user when otherwise idle.
// Note that for my application I always want a stream of idle processing
// to be going on, so that timer-related activity can be handled.
    LONG Idle = 0;
// Now I do any "idle tasks" that have been registered.
    while (!::PeekMessage(theApp.msgPtr, NULL, 0, 0, PM_NOREMOVE))
    {   if (!theApp.OnIdle(Idle++)) return;
    }
// I only drop through if there is a message waiting for me, so that
// PumpMessage will not block.
    do
    {   if (!theApp.PumpMessage())
        {   Lstop(0, 1);  // the "1" here actually means "0"!
            return;
        }
// If the user selects CLOSE from the system menu it causes PumpMessage to
// return FALSE, so in that case I close things down.
    } while (::PeekMessage(theApp.msgPtr, NULL, 0, 0, PM_NOREMOVE));
}

void cwin_minimize()
{
    WINDOWPLACEMENT wp;
    if (!::GetWindowPlacement(theApp.mainWindow->m_hWnd, &wp)) return;
    wp.showCmd = SW_SHOWMINIMIZED;
    ::SetWindowPlacement(theApp.mainWindow->m_hWnd, &wp);
}

void cwin_maximize()
{
    WINDOWPLACEMENT wp;
    if (!::GetWindowPlacement(theApp.mainWindow->m_hWnd, &wp)) return;
    wp.showCmd = SW_RESTORE;
    ::SetWindowPlacement(theApp.mainWindow->m_hWnd, &wp);
}

void CTheApp::OnHelpContents()              // Start on contents page.
{
    WinHelp(0L, HELP_CONTENTS);
}

void CTheApp::OnHelpSearch()
{
    WinHelp((DWORD)"", HELP_PARTIALKEY);    // Search through keywords.
}

void CTheApp::OnDynamic(unsigned int commandId)
{
    char hah[100];
//    sprintf(hah, "Help message %d %s\n", commandId,
//          dynamic_files[commandId-IDM_DYNAMIC_ITEMS]);
//    DisplayMsg(hah);
    ::WinHelp(mainWindow->m_hWnd,
              dynamic_files[commandId-IDM_DYNAMIC_ITEMS],
              HELP_CONTENTS,
              0);
}

void CTheApp::cwin_set_help_file(const char *key, const char *path)
{
    char key1[8];
    int i;
    if (key == NULL)
    {   WriteProfileInt("HelpItems", "HowMany", 0);
```



```
        for (i=0; i<dynamicCount; i++)
        {   sprintf(key1, "T%.3d", i);
            WriteProfileString("HelpItems", key1, NULL);
            sprintf(key1, "P%.3d", i);
            WriteProfileString("HelpItems", key1, NULL);
        }
        return;
    }
    for (i=0; i<dynamicCount; i++)
    {   if (strcmp(key, dynamic[i]) == 0) break;
    }
    if (i == dynamicCount) // not found
    {   if (path == NULL) return;
        else
        {   dynamic[dynamicCount] = key;
            dynamic_files[dynamicCount++] = path;
        }
    }
    else
    {   if (path == NULL)
        {   dynamicCount--;
            for (;i<dynamicCount; i++)
            {   dynamic[i] = dynamic[i+1];
                dynamic_files[i] = dynamic_files[i+1];
            }
        }
        else dynamic_files[i] = path;
    }
    WriteProfileInt("HelpItems", "HowMany", dynamicCount);
    for (i=0; i<dynamicCount; i++)
    {   sprintf(key1, "T%.3d", i);
        WriteProfileString("HelpItems", key1, dynamic[i]);
        sprintf(key1, "P%.3d", i);
        WriteProfileString("HelpItems", key1, dynamic_files[i]);
    }
    if (path == NULL)
    {   sprintf(key1, "T%.3d", i);
        WriteProfileString("HelpItems", key1, NULL);
        sprintf(key1, "P%.3d", i);
        WriteProfileString("HelpItems", key1, NULL);
    }
}

void cwin_set_help_file(const char *key, const char *path)
{
    theApp.cwin_set_help_file(key, path);
}


// DisplayMsg is used a bit like fprintf(stderr, ...) but ONLY for debugging.
// It pops up a modal dialog box each time it is called.  This is easy to
// code, but a bit clumsy in the way it disturbs the screen.

void
#ifdef _MSC_VER
     __cdecl
#endif
     DisplayMsg(char *msg, ...)
{
    char buffer[256];
    va_list a;
    va_start(a, msg);
    vsprintf(buffer, msg, a);
    va_end(a);
    AfxMessageBox(buffer);
}

// End of c_applic.cpp
```