4. Windows アプリケーション(1) Windows プログラムの基本，リソース

さて，今回からはWindowsアプリケーションの作成を始めます。Windowsのアプリケーションといと，ちょっと難しいように聞こえますが，実際はパターンが決まっていますので，それさえ押さえてしまえば，それほど苦労もなく作ることができます。また，実際に動作を見ながらプログラミングができますので，作ってみると意外と簡単ということもあります。

4.1. コンソールアプリケーションとの違い

まず，Windows アプリケーションとコンソールアプリケーションの違いから，確認してみましょう。コンソールアプリケーションを作るためには，

1) main 関数を書く。
2) ユーザ定義関数とプロトタイプ宣言をする。
3) 必要なら，変数を定義してアルゴリズム（プログラムの流れ）通りにソースを書く。
4) 必要に応じてヘッダファイルをインクルードする。

といった感じでした。では，Windows アプリケーションではどうでしょうか？実は，基本的にはほとんど変わりません。ただし，

1) main 関数の代わりに WinMain 関数を使う。
2) アプリケーションに最低1つインスタンスが必要で，これを最初に作る必要がある。
3) 1つのウィンドウに1つのウィンドウハンドルと呼ばれる識別子が必要である。
4) 必要に応じて，リソーススクリプトを書く必要がある。

という違いがあります。

4.2. Windows プログラム

4.2.1. Windows API

API（Application Program Interface）とはOSがアプリケーションに提供する機能（関数）セットのことで，これまでハードウェアなどを操作するためにたくさんのプログラムを書かなくてはいけなかったものを，OS が肩代わりすることで必要最小限の操作で同様の機能を実現するためのものです。その実体はDLL（Dynamic Link Library）と呼ばれるもので，必要なときに読み込まれる（Dynamic Link）C 言語で書かれたライブラリです。これらの機能はアプリケーションだけでなくOSそのものでも数多く使用しているため，実際にはWindowsを使っている間に何度も目にしている機能が数多くあります。

WindowsアプリケーションはこのWindows APIを呼び出すことで作成できます。呼び出す方法は，直接プログラムに書き込むかMFC（Microsoft Foundation Class）というものを通して行います。MFCはWindows APIを機能ごとにまとめて極単純なコードでWindows APIを使用できるようにしています。そのため，Visual C++でもできるだけMFCを使うように推奨しています。しかし，MFC は汎用性を高めるために無駄な処理も多く行っているため処理が遅い，処理の流れが見えなくなるためWindowsプログラムの仕組みが理解できないなどの欠点も持っています。ここでは，Windowsアプリケーションの作り方をマスターするのが目的ですので，MFC については取り扱わないことにします。

【練習問題 4.1】 Windows API

VC++のヘルプから MessageBox 関数という API の機能を調べなさい。複数選択出きる場合は，「プラットフォームSDK」の項目を参照すること。また，MFC の AfxMessageBox 関数も参照してその違いを比較しなさい。

4.2.2. MessageBox だけを使った Windows プログラム

ではMessageBox APIだけを使ったWindowsプログラムを作ってみましょう。基本的な作り方はコンソールアプリケーションと同じです。ただし，今回はプロジェクトの種類を「Win32 Application」にします。ただ，メッセージを表示するだけではつまらないので，時刻に応じてあいさつを表示するプログラムを作ってみましょう。必要となる API は

MessageBox と GetLocalTime ですので，あらかじめヘルプを参照しておいてください。list4.1 にソースを示します。

```
#include    <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){
    SYSTEMTIME          stSystemTime;
    int                 iRet;

    // 現在の時刻を取得します
    GetLocalTime(&stSystemTime);

    // 時刻に応じてメッセージを表示します
    if((stSystemTime.wHour > 0) && (stSystemTime.wHour < 12)){
        iRet = MessageBox(NULL, "おはようございます", "メッセージ", MB_OK);
    }else if(stSystemTime.wHour < 17){
        iRet = MessageBox(NULL, "こんにちは", "メッセージ", MB_OK);
    }else{
        iRet = MessageBox(NULL, "こんばんわ", "メッセージ", MB_OK);
    }

    return 1;
}
```

list4.1 MessageBox を使ったプログラム

ここで使った SYSTEMTIME は GetLocalTime 関数のパラメータに与える構造体で，MB_OK は MessageBox 関数のパラメータに与える定数です。これらは windows.h に定義されています。

4.2.3.　メッセージ

Windows の特徴である GUI(Graphical User Interface)とマルチタスクはメッセージというものを使うことで実現しています。メッセージとは，様々な動作に対して定義された番号で基本動作に関してはWindows によってあらかじめ決められています。プログラムの中では WM_・・・という名前で定義されており，これらをつかまえることで様々な動作を実現します。その動きを見るためには，Spy++というアプリケーションを使います。起動すると，図 4.1 のような画面が出てきます。では，メモ帳を動かしてその動きを見てみます。ウィンドウ1と表示された中には現在動いているアプリケーションの一覧が表示されます。その中から「003D0372“無題－メモ帳” Notepad」を選択してください(003D0372 のところは環境によって変化します)。この 003D0372 という数値はウィンドウを識別するための値でウィンドウハンドルといいます。「スパイ」－「メッセージ」を選択すると「メッセージオプション」というウィンドウがでてきますので，そのまま「OK」を押してください。すると「メッセージ(ウィンドウ 003D0372)」というウィンドウが開きます。ここにメモ帳に送られるメッセージが全て表示されます。「メモ帳」を動かしてその動作を確認してみてください。

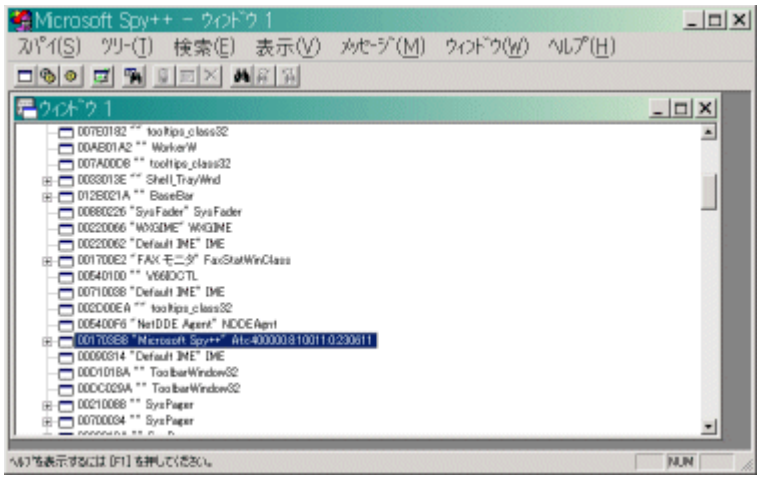

図 4.1　Spy++の起動画面

4.3.　基本的な Windows アプリケーション

では，メッセージを使った Windows アプリケーションの流れを見てみましょう。このプログラムソースはこれから作るプログラムの基本(テンプレート)になります。

4.3.1.　テンプレートとプログラムの流れ

list4.2(http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/4/)に示すソースの中を順に追っていくとこにしましょう。

1)　windows.h を読み込みます。これは，Windows 関連のいろいろな定義をしてあるヘッダなので，必ず読み込むようにしておきましょう。

2)　プリプロセッサ定義 WINNAME はこれから定義する WindowClass の名前になります。これから作るアプリケーショ

```
////////////////////////////////////////////////////////////////
// Win32 Application Template
////////////////////////////////////////////////////////////////
#include    <windows.h>

#define WINNAME "Template"          // ウィンドウクラスの名前

ATOM InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMes, WPARAM wp,
    LPARAM lp);
BOOL QuitMessage(HWND hWnd);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){

    if(!InitApplication(hInstance))
        return FALSE;

    HWND                hWnd;       // メインウィンドウハンドル
    if(NULL == (hWnd = InitInstance(hInstance, nCmdShow)))
        return FALSE;

    // イベントキューからメッセージを取得する
    MSG                 msg;
    do{
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if(msg.message == WM_QUIT)
                break;              // while()ループを抜ける
            TranslateMessage(&msg); // キー入力の取得
            DispatchMessage(&msg);  // メッセージを処理する
        }
        if(msg.message == WM_QUIT)
            break;                  // do～while()ループを抜ける
    }while(WaitMessage());

    return (int)msg.wParam;
}

ATOM InitApplication(HINSTANCE hInstance){
    WNDCLASSEX  wcl;

    // ウィンドウクラスを定義する
    wcl.hInstance = hInstance;      // インスタンスのハンドル
    wcl.lpszClassName = WINNAME;    // ウィンドウクラス名
    wcl.lpfnWndProc = WindowFunc;   // ウィンドウ関数
    wcl.style = 0;                  // デフォルトのスタイル
    wcl.cbSize = sizeof(WNDCLASSEX);// WNDCLASSEX構造体サイズ
    wcl.hIcon = NULL;               // ラージアイコン
    wcl.hIconSm = NULL;             // スモールアイコン
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
                                    // カーソルスタイル
    wcl.lpszMenuName = NULL;        // メニューなし
    wcl.cbClsExtra = 0;             // エキストラなし
    wcl.cbWndExtra = 0;             // 必要な情報なし

    // ウィンドウを白くする
    wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);

    // ウィンドウクラスを登録する
    return RegisterClassEx(&wcl);
}
```

```
HWND InitInstance(HINSTANCE hInstance, int nCmdShow){
    HWND        hWnd;

    // ウィンドウの作成
    hWnd = CreateWindow(
        WINNAME,                    // ウィンドウクラスの名前
        "Template",                 // タイトルバー
        WS_OVERLAPPEDWINDOW,        // ウィンドウスタイル
        CW_USEDEFAULT,              // x座標-Windowsに任せる
        CW_USEDEFAULT,              // y座標-Windowsに任せる
        CW_USEDEFAULT,              // 高さ-WIndowsに任せる
        CW_USEDEFAULT,              // 幅-Windowsに任せる
        HWND_DESKTOP,               // 親Windowなし
        NULL,                       // メニューなし
        hInstance,                  // インスタンスハンドル
        NULL                        // 追加引数なし
    );
    if(NULL == hWnd)
        return NULL;

    // ウィンドウを表示する
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return hWnd;
}

// この関数は，Windowsから呼び出されて，メッセージキューから
// メッセージの引き渡しを受ける
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM wp,
                            LPARAM lp){
    static HINSTANCE        hInstance;

    switch(uMsg){
    case WM_CREATE:                 // ウィンドウの作成
        hInstance = (HINSTANCE)GetWindowLong(hWnd,
                                    GWL_HINSTANCE);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);         // WM_QUITを発行する
        break;
    case WM_CLOSE:                  // ウィンドウの終了処理
        if(QuitMessage(hWnd)){      // 終了の確認
            DestroyWindow(hWnd);    // メインウィンドウに
                                    // WM-DESTROYを発行する
        }
        break;
    default:                        // 上記以外はWindowsに
        return DefWindowProc(hWnd, uMsg, wp, lp);
    }
    return 0;
}

BOOL QuitMessage(HWND hWnd){
    if(MessageBox(hWnd, (LPCSTR)"終了しますか？",
        (LPCSTR)"終了確認",
        MB_YESNO | MB_ICONQUESTION) == IDYES)
        return TRUE;
    return FALSE;
}
```

list4.2　Windows アプリケーションテンプレート

ンの基本的な名前にもなります。

3)　4つのプロトタイプ宣言をしていますが，最初の3つは Windows アプリケーションの基本的な動作を決める関数です。毎回ほとんど変わらないので，これらを使いまわすことで，ソースを書く効率がよくなります。最後の関数は，終了するときにいきなり終了しないようにするための関数です。これも毎回入れておくと便利でしょう。

4)　WinMain 関数です。ここで定義している「HWND hWnd」がこのアプリケーションのメインウィンドウハンドルです。まず InitApplication 関数でこれから作成するウィンドウの定義を行い，InitInstance 関数で hWnd を取得します。ついでに取得できなかったら，終了するという処理もしています。メインウィンドウに与えられるメッセージは「MSG msg」と定義し，以下の do～while 文と while 分で2重の無限ループで処理します。PeekMessage というのは Windows メッセージを取得する関数で，メッセージを受け取ると TRUE を返します。もし，受け取ったメッセージが WM_QUIT というメッセージなら2回 break して無限ループを終了，つまりアプリケーションを終了します。メッセージが WM_QUIT でない場合は，内側のループで囲まれた処理をして，WaitMessage()関数のところで次のメッセージが来るまで停止しています。

5)　InitApplication 関数では，このアプリケーションでメッセージを処理する関数(WindowsFunc)，アイコン，カーソル，メニュー，ウィンドウの色などを szWinName で定義した名前でウィンドウクラスというクラスに定義しています。

6)　InitInstance 関数は，定義されたウィンドウクラスに基づいてウィンドウを作成します。ShowWindow 関数はここで

作成されたウィンドウを表示します。さらに UpdateWindow 関数を呼び出しているのは，WindowFunc 関数で Window が作成されたときにする処理を画面に反映するためです。

7)　さて，いよいよ WindowFunc 関数です。この関数がプログラムの中心になります。まず，Instance を static で定義しています。これは，Instance を必要とする WindowsAPI が結構たくさんあるためです。実際の取得はこのアプリケーションウィンドウが作成されたとき，すなわち WM_CREATE で行います。次に，switch 文が来ます。WindowFunc 関数は PeekMessage 関数で TRUE が返ってきたとき，すなわちこのアプリケーションに何らかのメッセージが来たときに毎回呼び出されます。どのようなメッセージが与えられたかは uMsg に格納されています。アプリケーションで操作に合わせた処理をするためには uMsg によって適切な処理を行えば良い，ということになります。

例えば，このウィンドウが作成された時には WM_CREATE メッセージが通達されます。アプリケーションはこのとき，Window 内に文字を書く必要があれば，この段階で書くことができますし，メモ帳などのような文字入力部分を作成したいのであればこの段階で作成することになります。ここでは GetWindowLong 関数を使って hInstance の取得を行っています。

WM_CLOSE と WM_DESTROY は終了するときの処理です。ウィンドウの右上の×印が押されるとウィンドウに WM_CLOSE メッセージが通知されます。そこで，WindowFunc では WM_CLOSE が来たら終了して良いかの確認を行い（QuitMessage 関数），TRUE が来たら hWnd を破棄（DestroyWindow，つまり WM_DESTROY の送信）を行っています。WM_DESTROY はこのウィンドウが閉じる時の処理で，PostQuitMessage は WM_QUIT を送れという意味です。この WM_QUIT を WinMain が受け取ると，WinMain の無限ループが終了し，プログラムが終了するということです。

また，最後の DefWindowProc 関数は自分で処理する必要がないものは Windows に任せるという関数です。

8)　QuitMessage 関数は，終了して良いかどうかの確認をするための関数です。MessageBox の引数の hWnd は親 Window ハンドルで，このメッセージに応えるまで親ウィンドウを選択できないようにします。MessageBox では YES が押されたら IDYES が返ってくるので TRUE を返し，それ以外（この場合は NO だけですが）は FALSE を返すということです。ですから，この関数の戻値は TRUE（真）と FALSE（偽）を返す型 BOOL という具合になっています。

4.3.2.　Windows での型と変数名

list4.2 を見ていて，変数の名前と型が変だなと思った人もいるでしょう。実は Windows では変数の型が少し拡張されています。また，変数名についてもその変数の型がわかりやすいように型の識別名がつくようになっています。決まりという訳ではないのでこだわる必要はないですが，Help をみる手助けにもなるでしょうから，簡単に触れておきましょう。

表 4.1　Windows での型拡張

型	表記	意味	表記	意味
文字	CHAR	符号あり 8bit 文字	LPSTR	文字列定数
	UCHAR	符号なし 8bit 文字	LPCSTR	文字列
2 値	BOOL	TRUE と FALSE		
整数	SHORT	16bit 符号あり整数	ULONG	32bit 符号なし整数
	USHORT	16bit 符号なし整数	BYTE	8bit 符号なし整数
	INT	符号あり整数	WORD	16bit 符号なし整数
	UINT	符号なし整数	DWORD	32bit 符号なし整数
	LONG	32bit 符号あり整数	LONGLONG	64bit 符号なし整数
Windows	WINAPI	Win32 API	LRESULT	メッセージ処理の戻値
	WPARAM	32bit メッセージ変数	WNDPROC	Window 関数へのポインタ
	LPARAM	32bit メッセージ変数	VOID	任意の変数型

ハンドル系	HANDLE	オブジェクト	HCURSOR	カーソル
	HINSTANCE	インスタンス	HFILE	ファイル
	HWND	ウィンドウ	HICON	アイコン
	HACCEL	アクセラレータ	HMENU	メニュー
	HBRUSH	ブラシ		
ポインタ	LPBOOL	BOOL へのポインタ	LPDWORD	DWORD へのポインタ
	LPINT	INT へのポインタ	LPLONGLONG	LONGLONG へのポインタ
	LPLONG	LONG へのポインタ	LPVOID	VOID へのポインタ
	LPBYTE	BYTE へのポインタ	LPHANDLE	HANDLE へのポインタ
	LPWORD	WORD へのポインタ		

変数名は例えば，WORD 型なら w･･･，DWORD 型なら dw･･･，ハンドル型なら h･･･，ポインタ型なら lp･･･，文字列型なら lpsz･･･といった感じです。

4.4. リソース

Windows プログラムの特徴の一つにリソースがあります。リソースとはプログラムとは独立したデータのことで，アイコン，カーソル，文字列，ビットマップ，メニュー，キーボードアクセラレータ，ダイアログボックスなどがこれにあたります。

4.4.1. アイコンとカーソル

それではプログラムにアイコンとカーソルを追加してみましょう。リソースを書くにはリソースエディタを使います。

1) まず，テンプレートを基にワークスペースを作ります。

2) 「ファイル」ー「新規作成」から新規作成ウィンドウを表示します。「リソーススクリプト」をクリックして選択し，ファイル名を書き込んでから「OK」を押します。すると，FileView の Source Files フォルダの中に，･･･.rc というファイルができているのが確認できます。また，ワークスペースウィンドウに FileView，ClassView の他に ResourceView というのが追加されます。

3) では，ResouceView に切り替えて新しいリソースを作ってみます（アウトプットウィンドウは閉じておいてください）。フォルダの形をしたアイコンで右クリックしてポップアップメニューから「挿入」を選択します。リソースの挿入ダイアログが出てくるので，「Icon」を選択して「新規作成」ボタンをクリックします。作業領域にアイコンを作成するためのリソースエディタが表示されますので，適当なアイコンを作成してみてください。

   作成できたらワークスペースウィンドウを見ると IDI_ICON1 という名前でアイコンができていることがわかります。この識別子（ID）を使ってプログラムからこのアイコンを呼び出すことになります。

4) 同様にしてカーソルも作成してみましょう。アイコンと異なることは，白黒といくつかのグレーしか使用できないことと，ホットスポット（クリック位置を示すもの）を指定できることです。ホットスポットの指定は上の方にある  を押すと指定できます。今度は IDC_CURSOR1 という ID で作成できています。

5) ここで，「ファイル」ー「すべて保存」を選択してから，ファイルのあるフォルダを見ると，今作ったアイコン，カーソルと resource.h というファイルができています。このファイルを開いてみると，

```
#define    IDI_ICON1          101
#define    IDC_CURSOR1        102
```

と書かれているはずです。このようにリソースは実際にはこのような数字で識別されています。

6) さて，実際のプログラムにこれを反映するには元のプログラムに多少の修正が必要です。

   ① resource.h を include する。

   ② InitApplication 関数で，

```
wcl.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
```

```
wcl.hIconSm = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_ICON1));
wcl.hCursor = LoadCursor(hInstance, MAKEINTRESOURCE(IDC_CURSOR1));
```

を追加します。

7)　変更が終わったら実際にビルド，実行して変更を確認してみましょう。

4.4.2.　メニュー

Windows のプログラムで使用されるメニューには上のほうに表示される通常のメニューとマウスの右ボタンを押したときに表示されるポップアップメニューがあります。ここでは使用頻度の高い通常のメニューについてのみ説明します。

メニューはアイコンやカーソルと同じように ID が付けられるだけではなく，メニュー内の個々の項目にも ID が付けられます。この ID は WM_COMMAND メッセージを使ってアプリケーションに通知されます。どのメニューが選択されたかは WM_COMMAND を受け取ったときの WindowFunc の引数 wp の下位ワード（32bit 整数の下位 16bit）に格納されます。では，「ファイル」－「終了」という項目を持つメニューをテンプレートに付けてみましょう。

1)　まず，テンプレートを基にワークスペースを作成して，リソーススクリプトを追加しておきます。

2)　今回はリソースの追加で「Menu」を選択します。すると，IDR_MENU1 という ID が追加されます。

3)　破線で表示された四角形をダブルクリックすると，メニューアイテムプロパティというウィンドウが表示されます。ここでメニューに関する設定を行います。キャプション欄に「ファイル(&F)」と入力してください。アンパサンド（&）をアルファベットや数字の前に付けるとメニューの中では「ファイル(F)」のようにアンダーライン付きで表示されます。このようにしておくとキーボードの「Alt」と「F」を同時に押すことで，このメニューを選択できるようになります。

4)　今度は下に表示された四角をダブルクリックして，今と同じように「終了(&X)」を作ります。ここで終了すると，このメニューアイテムには自動的に ID_MENUITEM40001 という ID がつきます。このままでもプログラム上はさしつかえないですが，あとでわかりやすいように自分で ID を決めておきましょう。今回は IDM_QUIT という名前にしておきます。ID の欄に「IDM_QUIT」と入力してください。

5)　さて，プログラムにこれを反映させましょう。

①　前回と同様に resource.h を include する。

②　InitApplication 関数で，

```
wcl.lpszMenuName = MAKEINTRESOURCE(IDR_MENU1);
```

を追加します。

③　WIndowFunc に，

```
case WM_COMMAND:
    switch(LOWORD(wp)){
    case IDM_QUIT:
        PostMessage(hWnd, WM_CLOSE, 0, 0L);
        break;
    }
    break;
```

を追加します。

6)　変更が終わったら実際にビルド，実行して変更を確認してみましょう。

4.4.3.　ダイアログボックス

多くの Windows プログラムではメインウィンドウの他に入力や設定を行うためのダイアログボックスを使います。ダイアログボックスには2種類あり，それぞれモーダルダイアログとモードレスダイアログと呼ばれます。モーダルダイアログとはファイルを開くときに出てくるダイアログボックスのように OK や Cancel などを押さない限り他の処理ができないものです。一方，モードレスダイアログとは検索ダイアログのように検索中も文章の編集ができるように，他のウィンドウと並行

して動作するダイアログボックスです。ここでは，モーダルダイアログについてのみ触れます。

ダイアログボックスを作るためには DialogBox 関数を，閉じるには EndDialog 関数を使います。また，ダイアログボックス自身やその上に表示されるコントロール（第５回で解説）の処理はウィンドウプロシージャ（WindowsFunc）と同様のダイアログプロシージャで処理します。

まだ，コントロールの説明をしていませんので，今回はただ文字（バージョン情報）を表示するだけのダイアログを表示してみましょう。

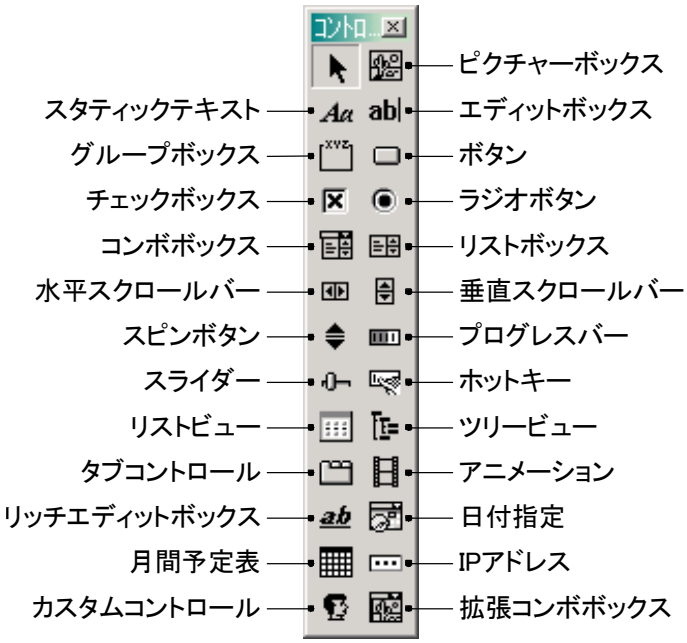

図 4.2　コントロールボックス

1)　まず，テンプレートを基にワークスペースを作成して，リソーススクリプトを追加しておきます。

2)　前回と同様にメニューを作成しておきます。今回はメニュー項目を「ファイル(&F)」－「終了(X)」（IDM_QUIT）と「ファイル(&F)」－「バージョン(&A)」(IDM_ABOUT)の2つにしておきます。

3)　リソースの追加で「Dialog」を選択します。すると，IDD_DIALOG1 という ID が追加されます。また，エディタ画面に図 4.2 のようなコントロールボックスが表示されます。それぞれのボタンは様々なコントロールを示しています。今回はスタティックテキストコントロールだけを使います。ダイアログにはあらかじめ「OK」と「キャンセル」の2つのボタンが張りつけられています。これらにはそれぞれ「IDOK」，「IDCANCEL」の ID があらかじめ付けられています。今回はこれ以外にスタティックテキストで「ダイアログのテスト Ver.1」という文字も追加したりしながら自由に加工してみてください（図 4.3）。

図 4.3　ダイアログボックス

4)　さて，プログラムにこれを反映させましょう。今回は長いので list4.3 に示しました。

```
BOOL QuitMessage(HWND hWnd);
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp,
                         LPARAM lp);
                              ⋮
LRESULT CALLBACK WindowFunc(HWND hWnd, UINT uMsg, WPARAM
wp, LPARAM lp){
                              ⋮

    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDM_QUIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0L);;
            break;
        case IDM_ABOUT:
            DialogBox(hInstance,
                      MAKEINTRESOURCE(IDD_DIALOG1),
                      hWnd,
                      (DLGPROC)DialogProc);
            break;
        }
        break;
```

```
                              ⋮
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMes, WPARAM wp,
                         LPARAM lp){
    switch(uMes){
    case WM_INITDIALOG:
        break;
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDOK:
            EndDialog(hDlg, 0);
            return TRUE;
        case IDCANCEL:
            EndDialog(hDlg, 0);
            return TRUE;
        }
        break;
    }
    return FALSE;
}
```

list4.3　ダイアログの表示

最初のは今回追加するダイアログプロシージャのプロトタイプ宣言です。戻値が BOOL になっている以外はウィンドウプロシージャと同じです。

次にこのダイアログを出すメニュー項目（IDM_ABOUT）を追加します。これは前回と方法は同じです。この中で

ダイアログボックスを呼び出すことになります。モーダルダイアログボックスを呼び出すにはDialogBox関数を使います。引数は，インスタンスハンドル(hInstance)，ダイアログリソース，呼びだし側ウィンドウハンドル(hWnd)，ダイアログプロシージャ名となります。

最後にダイアログプロシージャ本体を追加します。ほとんどウィンドウプロシージャと同じです。ダイアログボックスも一つのウィンドウになるわけですから，ダイアログを作成するとプログラム本体とは別のウィンドウハンドル hDlg が1つ作成されます。

ウィンドウプロシージャと異なる点をまとめると，

① ウィンドウが作成されたときに来るメッセージは WM_INITDIALOG となる。

② DefWindowProc が必要ない。その代わり，処理をしたときは TRUE を，しなかったときは FALSE を返す。

③ ダイアログを閉じるには EndDialog を呼び出す。

というところです。

4.5. ダイアログベースアプリケーション

ダイアログボックスはリソースエディタを使うことで容易にインタフェースを作ることができます。そこで，4.3 で説明した WinMain の最初でモードレスダイアログを作成することで，インターフェースの構築が容易なダイアログベースアプリケーションを作ることができます。list4.4（http://avse.bpe.agr.hokudai.ac.jp/~ici/pc/4/DialogBase.zip，VS6 用は DialogBase6.zip）のソース。基本的な流れは list 4.2 と同じで，①ウィンドウクラスの登録とインスタンスの生成を CreateDialog だけで行う。②WindowFunc の代わりに DialogProc を用いる。③ウィンドウクラスで登録するようなリソースは WM_INITDIALOG で登録する。というところが大きな違いとなります。

```
//=================================================
// ダイアログボックスベースのアプリケーション
// === モードレスダイアログ版 ===
//=================================================
#include    <windows.h>
#include    <windowsx.h>
#include    "resource.h"

BOOL CALLBACK DialogProc(HWND hWnd, UINT uMes, WPARAM wp,
LPARAM lp);
BOOL QuitMessage(HWND hDlg);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow){
    // モードレスダイアログボックスを生成する
    // ダイアログボックスが生成されると処理が戻ります
    static HWND         hDlg;
    hDlg = CreateDialog(hInstance,
        MAKEINTRESOURCE(IDD_DIALOG), NULL,
        (DLGPROC)DialogProc);

    // アクセラレータのロード
//  HACCEL              hAccel; // アクセラレータハンドル
//  hAccel = LoadAccelerators(hInstance,
        MAKEINTATOM(IDR_ACCELERATOR));

    // メッセージループを生成する
    MSG                 msg;
    do{
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if(msg.message == WM_QUIT)
                break;      //while()ループを抜ける
//          if(TranslateAccelerator(hWnd, hAccel, &msg))
//              continue;
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        if(msg.message == WM_QUIT)
            break;          // do～while()ループを抜ける
    }while(WaitMessage());

    return msg.wParam;
}
```

```
BOOL CALLBACK DialogProc(HWND hDlg, UINT uMsg, WPARAM wp,
                         LPARAM lp){
    static HINSTANCE    hInstance;

    switch(uMsg){
    case WM_INITDIALOG:         // ダイアログの生成
        hInstance = (HINSTANCE)GetWindowLong(hDlg,
            GWL_HINSTANCE);
        SetClassLong(hDlg,      // アイコンの設定
            GCL_HICON, (LONG)LoadIcon(hInstance,
            MAKEINTRESOURCE(IDI_ICON)));
        ShowWindow(hDlg, SW_SHOW);  // ウィンドウの表示
        UpdateWindow(hDlg);
        return TRUE;

    case WM_CLOSE:              // ×を押したとき
        if(QuitMessage(hDlg)){  // 終了確認
            DestroyWindow(hDlg);// ダイアログの破棄
            PostQuitMessage(0); // WM_QUITを送る
            return TRUE;
        }
        return TRUE;

    // メニューから呼び出されるメッセージ
    case WM_COMMAND:
        switch(LOWORD(wp)){
        case IDM_QUIT:          // [ファイル]－[終了]
            PostMessage(hDlg, WM_CLOSE, 0, 0L);
            return TRUE;
        }
    }
    return FALSE;               // 何もしないときはFALSE
}

BOOL QuitMessage(HWND hDlg){
    if(MessageBox(hDlg, "終了してもよろしいですか。",
        "終了確認",MB_YESNO | MB_ICONQUESTION) == IDYES)
        return TRUE;
    return FALSE;
}
```

list4.4 ダイアログベースアプリケーション