第 2 回 Windows アプリ講座

1．文字を表示する

さて、ウィンドウを作ったらその画面に文字を表示してみましょう。早速、ソースコードを乗せます。ウィンドウプロシージャのみを掲載しますが、ほかの部分は前回と同じと考えてください。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//ウィンドウプロシージャ
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
	HDC hdc;		//デバイスコンテキストハンドル
	PAINTSTRUCT ps; //ペイント構造体
	LPCTSTR lpString = TEXT("文字の表示");		//描画する文字列
	switch(msg)
	{
	case WM_CLOSE:
		PostQuitMessage(0);
		break;

	case WM_PAINT:		//画面更新のメッセージ
		hdc = BeginPaint(hWnd, &ps);		//描画の開始
		SetTextColor(hdc, RGB(255, 0, 0));		//文字の色を設定する
		TextOut(hdc, 10, 10, lpString, lstrlen(lpString));
		EndPaint(hWnd, &ps);		//描画の終了
		break;

	default:
		return (DefWindowProc(hWnd, msg, wParam, lParam));
	}

	return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

WM_PAINT とは画面の更新を要求されたときに受け取るメッセージです。このメッセージはウィンドウサイズを変えたり移動したりなどしたときに発行されます。つまり、ウィンドウズアプリケーションは常に画面を描画するのではなく、必要なときだけ画面を描画しているのです。

HDC はデバイスコンテキストハンドルです。デバイスコンテキストとは、大雑把に言うと、描画する媒体の情報を表すものです。今回はウィンドウに描画を行いますが、デバイスコンテキストをプリンタにすると、プリンタに描画、つまり印刷が出来ます。

画面の更新には BeginPaint 関数を使います。この関数は更新リージョン（平たく言うと更新される領域）を自動で求めてくれる関数です。そして、この関数は求められた更新リージョンをデバイスコンテキストハンドルとして返します。

文字の描画には TextOut 関数を使います。引数には描画先のデバイスコンテキストハンドル、描画する座標、文字列へのポインタと文字数を渡します。

すべての描画が終わったら EndPaint 関数を呼び出し、描画が終わったことを知らせてやります。以上で文字の描画は完了。

2.メッセージボックスを使う

さまざまなソフトを使っているとエラーメッセージや確認ダイアログなどメッセージボックスが出てきます。では、メッセージボックスを使ってみましょう。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//ウィンドウプロシージャ
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
	switch(msg)
	{
	case WM_CLOSE:
		switch(MessageBox(hWnd, TEXT("終了しますか？"), TEXT("終了ダイアログ"), MB_YESNO |
                                                                                MB_ICONQUESTION)
		)
		{
		case IDYES:
			PostQuitMessage(0);
			break;
		}
		break;

	default:
		return (DefWindowProc(hWnd, msg, wParam, lParam));
	}

	return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

ウィンドウを閉じようとしたときにメッセージボックスを出すようにしました。メッセージボックスを使うには MessageBox 関数を使います。引数には親ウィンドウへのウィンドウハンドル、メッセージボックス内のメッセージ、メッセージボックスをタイトルとメッセージボックスのスタイルを指定して渡します。

メッセージボックスのスタイルとは、メッセージボックス内のアイコンやボタンの種類等を指します。今回はボタンが「はい」と「いいえ」のタイプ（MB_YESNO）とアイコンがクエスチョンマーク(MB_ICONQUESTION)のものを選択しました。スタイルを複数指定する場合には or 演算子でスタイルの識別子をつないでいきます。

MessageBox 関数は選択されたボタンに対応した値を返します。MB_YESNO で作られたメッセージボックスの場合、「はい」を選択したら IDYES が、「いいえ」を選択したときは IDNO が返されます。メッセージボックスで使われる識別子についてはたくさん種類があるので各自で調べてください。

3.ウィンドウサイズを固定してみる

ウィンドウサイズが固定されたアプリケーションってありますよね。それと同じことをしてみましょう。ソースはウィンドウ作成部のみ載せます。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//ウィンドウの作成
BOOL InitInstance(HINSTANCE hInst, int nCmdShow)
{
	HWND hWnd;	//ウィンドウハンドル

	//ウィンドウの作成
	hWnd = CreateWindow(szClassName,	//使用するウィンドウクラス
		TEXT("真っ白ウィンドウ"),		//タイトルバーの名前
```

```
                WS_OVERLAPPEDWINDOW ^ WS_MAXIMIZEBOX ^ WS_SIZEBOX,  //ウィンドウスタイル
                CW_USEDEFAULT,                                      //ウィンドウの x 座標
                CW_USEDEFAULT,                                      //ウィンドウの y 座標
                CW_USEDEFAULT,                                      //ウィンドウの幅
                CW_USEDEFAULT,                                      //ウィンドウの高さ
                NULL,                                   //オーナーウィンドウハンドル
                NULL,                                   //メニューハンドルか小ウィンドウの ID
                hInst,                                  //アプリケーションのインスタンスハンドル
                NULL);                                  //ウィンドウ作成データ

        if (!hWnd)return FALSE;             //ウィンドウの作成に失敗

        ShowWindow(hWnd, nCmdShow);         //ウィンドウを可視化する
        UpdateWindow(hWnd);                 //ウィンドウを更新する
        return TRUE;
}
```

//

ウィンドウスタイル WS_OVERLAPPEDWINDOW に含まれている WS_MAXIMIZEBOX(最大化ボタン)と WS_SIZEBOX(サイズ変更)を取り除けばいいだけです。ウィンドウスタイルはビット演算で処理するので取り除くときは Ex-or を使います。さて、このままではシステムメニューに最大化とサイズ変更の項目が残ったままです(選択は出来ません)。せっかくなのでこれらのシステムメニューを削除してみましょう。

//

```
//ウィンドウの作成
BOOL InitInstance(HINSTANCE hInst, int nCmdShow)
{
        HWND hWnd;      //ウィンドウハンドル

        //ウィンドウの作成
        hWnd = CreateWindow(szClassName,    //使用するウィンドウクラス
                TEXT("真っ白ウィンドウ"),           //タイトルバーの名前
                WS_OVERLAPPEDWINDOW ^ WS_MAXIMIZEBOX ^ WS_SIZEBOX,
        //ウィンドウスタイル
                CW_USEDEFAULT,                          //ウィンドウの x 座標
                CW_USEDEFAULT,                          //ウィンドウの y 座標
                CW_USEDEFAULT,                          //ウィンドウの幅
                CW_USEDEFAULT,                          //ウィンドウの高さ
                NULL,                                   //オーナーウィンドウハンドル
                NULL,                                   //メニューハンドルか小ウィンドウの ID
                hInst,                                  //アプリケーションのインスタンスハンドル
                NULL);                                  //ウィンドウ作成データ

        if (!hWnd)return FALSE;             //ウィンドウの作成に失敗

        HMENU hSysMenu = GetSystemMenu(hWnd, FALSE);    //システムメニューのメニューハンドルを取得
        DeleteMenu(hSysMenu, SC_MAXIMIZE, MF_BYCOMMAND);       //システムメニュー「最大化」を削除
        DeleteMenu(hSysMenu, SC_SIZE, MF_BYCOMMAND);           //システムメニュー「サイズ変更」を削除
        ShowWindow(hWnd, nCmdShow);             //ウィンドウを可視化する
        UpdateWindow(hWnd);                     //ウィンドウを更新する
        return TRUE;
}
```

//

まずはシステムメニューのメニューハンドルを取得します。それには GetSystemMenu 関数を使います。
引数はウィンドウハンドルと関数の動作フラグを渡します。第2引数が FALSE だと現在のシステムメニューをそのまま残してシステムメニューハンドルを返します。TRUE にするとシステムメニューがリセットされます。そして、

メニューハンドルは返されません。

メニューハンドルが取得できたら次は DeleteMenu 関数でメニューを削除します。第1引数にはメニューハンドルを渡しますが、第2引数は、第3引数によって異なります。第3引数はメニューの削除対称をどのように決めるかを指定します。MF_BYCOMMAND だとメニューの識別子名を使い、MF_BYPOSITION だと上から何番目のメニューを削除するかということになります。今回は MF_BYCOMMAND を使用します。この場合、第2引数には削除対称のメニューの識別子を渡します。「最大化」は SC_MAXIMIZE、「サイズ変更」は SC_SIZE です。

これで、システムメニューの削除が完了しました。実際にメニューが削除されているのを確認してください。

4.時計を作る

時計を作ってみましょう。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <windows.h>
#include <time.h>               //時間処理用のヘッダー

#define IDT_MYTIMER                     0x10000             //タイムアウト時のイベント ID

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam); //ウィンドウプロシージャの宣言

//グローバル変数
LPCTSTR szClassName = TEXT("MainWindow"); //ウィンドウクラス名

//ウィンドウクラスの登録
ATOM InitApp(HINSTANCE hInst)
{
        WNDCLASSEX wc;
                //ウィンドウクラス構造体
        wc.cbSize = sizeof(WNDCLASSEX);                                 //ウィンドウクラス構造体のサイズを設定
        wc.style = CS_HREDRAW | CS_VREDRAW;                             //ウィンドウクラスのスタイルを設定
        wc.lpfnWndProc = WndProc;                                       //ウィンドウプロシージャの設定
        wc.cbWndExtra = 0;                                      //ウィンドウに割り当てられる追加メモリ量
        wc.cbClsExtra = 0;                                      //ウィンドウクラスに割り当てられる追加メモリ量
        wc.hInstance = hInst;                                   //アプリケーションのインスタンスハンドルの設定
        wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);             //アイコンの設定
        wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);           //小アイコンの設定
        wc.hCursor = LoadCursor(NULL, IDC_ARROW);               //カーソルの設定
        wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);     //背景の設定
        wc.lpszMenuName = NULL;                                 //メニューバーの設定
        wc.lpszClassName = szClassName;                         //ウィンドウクラス名の設定
        return (RegisterClassEx(&wc));                          //ウィンドウクラスの登録
}

//ウィンドウの作成
BOOL InitInstance(HINSTANCE hInst, int nCmdShow)
{
        HWND hWnd;      //ウィンドウハンドル

        //ウィンドウの作成
        hWnd = CreateWindow(szClassName,    //使用するウィンドウクラス
                TEXT("真っ白ウィンドウ"),               //タイトルバーの名前
                WS_OVERLAPPEDWINDOW ^ WS_MAXIMIZEBOX ^ WS_SIZEBOX,
        //ウィンドウスタイル
                CW_USEDEFAULT,                                  //ウィンドウの x 座標
```

```
                CW_USEDEFAULT,                          //ウィンドウの y 座標
                200,                                    //ウィンドウの幅
                100,                                    //ウィンドウの高さ
                NULL,                                   //オーナーウィンドウハンドル
                NULL,                                   //メニューハンドルか小ウィンドウの ID
                hInst,                                  //アプリケーションのインスタンスハンドル
                NULL);                                  //ウィンドウ作成データ

        if (!hWnd)return FALSE;             //ウィンドウの作成に失敗

        HMENU hSysMenu = GetSystemMenu(hWnd, FALSE);    //システムメニューのメニューハンドルを取得
        DeleteMenu(hSysMenu, SC_MAXIMIZE, MF_BYCOMMAND);       //システムメニュー「最大化」を削除
        DeleteMenu(hSysMenu, SC_SIZE, MF_BYCOMMAND);           //システムメニュー「サイズ変更」を削除

        ShowWindow(hWnd, nCmdShow);             //ウィンドウを可視化する
        UpdateWindow(hWnd);                             //ウィンドウを更新する
        return TRUE;
}

//時刻文字列の取得関数
void GetTimeString(LPTSTR lpTimeString)
{
        time_t t;
        tm NowTime;

        time(&t);
        localtime_s(&NowTime, &t);
        wsprintf(lpTimeString, TEXT("%d:%d:%d"), NowTime.tm_hour, NowTime.tm_min, NowTime.tm_sec);
}

//ウィンドウプロシージャ
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
        HDC hdc;                    //デバイスコンテキストハンドル
        PAINTSTRUCT ps;             //ペイント構造体
        TCHAR TimeString[256];
        RECT rect;

        switch(msg)
        {
        case WM_CLOSE:
                PostQuitMessage(0);
                break;

        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);            //描画開始
                GetTimeString(TimeString);
                TextOut(hdc, 20, 20, TimeString, lstrlen(TimeString));
                EndPaint(hWnd, &ps);                            //描画終了
                SetTimer(hWnd, IDT_MYTIMER, 1000, NULL);  //タイマーをセット
                break;

        case WM_TIMER:              //タイマーのタイムアウトメッセージ
                KillTimer(hWnd, IDT_MYTIMER);                   //タイマーを破棄
                GetClientRect(hWnd, &rect);                     //クライアント領域を取得
                RedrawWindow(hWnd, &rect, NULL, RDW_ERASE | RDW_INVALIDATE);//ウィンドウの再描画
                break;


        default:
```

```
                return (DefWindowProc(hWnd, msg, wParam, lParam));
        }

        return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst, LPSTR lpCmdLine, int nCmdShow)
{
        if (InitApp(hInstance) == 0)    return -1;
        if (InitInstance(hInstance, nCmdShow) == FALSE)         return -1;

        //メッセージループ
        MSG msg;            //メッセージ構造体

        while(GetMessage(&msg, NULL, 0, 0) != 0)        //メッセージの取得
        {
                DispatchMessage(&msg);                          //メッセージの送信
        }

        return 0;
}
```

//

時計を作るには一秒ごとに画面を書き換えなくてはなりません。そのためにはタイマーを使用して一秒経過するごとに画面を再描画すればいいのです。

タイマーを使用するには SetTimer 関数を使用します。この関数にタイムアウトの時間をミリ秒単位で設定します。タイマーがタイムアウトすると WM_TIMER メッセージが発行されます。そこで、WM_TIMER を受信したときに再描画処理を記述してやれば一秒ごとに画面が更新されます。

5.フォントを変更する

//

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
        HDC hdc;                    //デバイスコンテキストハンドル
        PAINTSTRUCT ps;             //ペイント構造体
        TCHAR TimeString[256];
        RECT rect;
        HFONT hOldFont;

        switch(msg)
        {
        case WM_CREATE:
                g_hTimeFont = CreateFont(40,                    //フォントの高さ
                        0,                                      //フォントの幅
                        0,                                      //文字送り方向の傾き
                        0,                                      //ベースラインの傾き
                        FW_NORMAL,                              //フォントの太さ
                        FALSE,                                  //斜体フラグ
                        FALSE,                                  //下線フラグ
                        FALSE,                                  //取り消し線フラグ
                        ANSI_CHARSET,                           //文字セット識別子
                        OUT_DEFAULT_PRECIS,                     //出力精度
                        CLIP_DEFAULT_PRECIS,                    //クリッピング精度
                        DEFAULT_QUALITY,                        //出力品質
                        FF_DONTCARE | DEFAULT_PITCH,            //ピッチとファミリ
                        TEXT("MS ゴシック")                     //フォント名
```

```
                                        );
                        break;

        case WM_CLOSE:
                        PostQuitMessage(0);
                        break;

        case WM_PAINT:
                        hdc = BeginPaint(hWnd, &ps);            //描画開始
                        GetTimeString(TimeString);
                        hOldFont = (HFONT)SelectObject(hdc, (HFONT)g_hTimeFont);
                        TextOut(hdc, 10, 10, TimeString, lstrlen(TimeString));
                        SelectObject(hdc, (HFONT)hOldFont);
                        EndPaint(hWnd, &ps);                    //描画終了
                        SetTimer(hWnd, IDT_MYTIMER, 1000, NULL);  //タイマーをセット
                        break;

        case WM_TIMER:          //タイマーのタイムアウトメッセージ
                        KillTimer(hWnd, IDT_MYTIMER);           //タイマーを破棄
                        GetClientRect(hWnd, &rect);             //クライアント領域を取得
                        RedrawWindow(hWnd, &rect, NULL, RDW_ERASE | RDW_INVALIDATE);    //ウィンドウの再
描画
                        break;


        default:
                        return (DefWindowProc(hWnd, msg, wParam, lParam));
        }

        return 0;
}
```

//

フォントを変更するには論理フォントを作成しなければなりません。論理フォントの作成には CreateFont 関数を使います。論理フォントが作成できたらフォントハンドルが帰されるので、それを HFONT 型の変数に代入しておき、いつでも使えるようにしておきましょう。

さて、フォントが作成できたら次は使用するフォントの設定です。フォントの設定には SelectObject 関数を使います。この関数に使いたいフォントハンドルを渡せばそのフォントを使用して文字を描画することが出来ます。