# 第１回 Windows アプリ講座

## １．Wndows アプリケーションを作るには

Windows アプリケーションを作るには Win32API というものを使います。これは、Windows アプリケーションを作るのに欠かせないものです。よって、Windows アプリ講座では Win32API の使い方を中心にしていきます。

## 2.ウィンドウを作る

さて、何はともあれまずはウィンドウを作らなければなりません。そこで、真っ白なウィンドウを作成するソースコードを紹介します。

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <windows.h>

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,			//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;
}


//ウィンドウプロシージャ
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
	switch(msg)
	{
		case WM_CLOSE:
			PostQuitMessage(0);
			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;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

それなりに量がありますね。しかし、このプログラムで実装されるウィンドウの機能は、いわゆるウィンドウズが提供する機能のみ（終了・最大化・最小化など）です。しかし、賢明な我々はプログラムを作るたびにコードを 1 から書くのではなく、使いまわして、必要な箇所だけ書き換えればいいことに気づきます。

さて、今回の講座では、この「何もしない」プログラムの解説を、Windows アプリケーションの仕組みを交えながら解説します。全部資料に書き起こすと朗読会になるのでこちらでは大雑把に済ませてしまいます。必要な部分は随時メモするなどしてください。

# 3.文字セット

さて、アプリケーションに使用される文字セットについて少し説明しておきます。アプリケーションの開発では、マルチバイト文字と Unicode 文字があります。結論を先に述べると、アプリケーション開発には Unicode 文字を

使用しましょう。

さて、マルチバイト文字というのは、半角文字には 1 バイト、全角文字には 2 バイトで表現しています。しかし、Unicode では、すべての文字を 2 バイトで表現します。文字によるバイト数の変動が無いことや、全角文字を直接文字操作可能になります。そういった点で Unicode が使われています。

Uncode では 2 バイトで文字を表現するため char では足りません。そこで、wchar_t という文字型を使用します。wchar_t は 2 バイトの文字型です。

さて、char 型では、文字列を代入するときは””でくくればよかったですが、wchar_t 型の変数に文字列を代入するときは L””を使用します。ところで、WIN32API には TEXT マクロというものが存在します。これは、プログラムの文字セットによって、””と L””を切り替えるマクロです。何らかの都合で文字セットを変更しなければならなかったときにすべての””を L””に変更する手間を省くことが出来ます。さらに、TCHAR というデータ型は文字セットによって char と wchar_t を自動で切り替えてくれます。

# 4.データ型

WIN32API ではさまざまなデータ型が typedef で定義されています。それらを表 4.1 にまとめました。

表 4.1 WIN32API で用いられるデータ型

| 型名 | 意味 | 備考 |
|---|---|---|
| HWND | ウィンドウハンドル | ウィンドウに与えられる識別番号 |
| UINT | 32 ビットの符号無し整数型 | unsigned int と同じ |
| WPARAM | メッセージの付加情報 | |
| LPARAM | メッセージの付加情報 | |
| LPCTSTR | 定数文字列の先頭へのポインタ | 開発時の文字セットによって型が変化 |
| LPTSTR | 文字列の先頭へのポインタ | 開発時の文字セットによって型が変化 |
| TCHAR | 文字型 | 開発時の文字セットによって型が変化 |
| LPCSTR | 定数文字列の先頭へのポインタ | const char*と同じ |
| LPSTR | 文字列の先頭へのポインタ | char*と同じ |
| CHAR | char 型 | |
| LPCWSTR | 定数文字列の先頭へのポインタ | const wchar_t*と同じ |
| LPWSTR | 文字列の先頭へのポインタ | wchar_t*と同じ |
| WCHAR | wchar_t 型 | |
| ATOM | ウィンドウズに登録される情報の識別番号 | |
| HINSTANCE | インスタンスハンドル | アプリケーションの識別番号 |
| WNDCLASSEX | ウィンドウクラス構造体 | ウィンドウの定義を納める |
| BOOL | bool 型 | |
| LRESULT | ウィンドウプロシージャの戻り値 | |
| HRESULT | 関数の戻り値 | |
| MSG | MSG 構造体 | |
| WORD | 16 ビットの符号無し整数 | unsigned short と同じ |
| DWORD | 32 ビットの符号無し整数 | unsigned long と同じ |

まだまだありますが、それは随時必要になったら紹介します。さて、WIN32API 固有のデータ型を全部覚えるのはほぼ無理です。しかし、データ型の名前にはある程度の規則性があります。たとえば、ハンドルを表すデータ型は必ず"H"で始まります。また、何らかのポインタをあらわす変数は"LP"で始まります。

ウィンドウズアプリケーションでは慣れないうちは、これら独特のデータ型に苦戦する場合が多いです。このあたりは慣れるしかありません。もし、これらのデータ型を覚えることがあるとすれば、それはプログラムを組む中で自然の覚えるものなのです。

# 5.ウィンドウを作るまでの流れ

ウィンドウを作成するには、ウィンドウクラスを登録する必要があります。ウィンドウクラスでは、ウィンドウの基礎となる情報を設定していきます。たとえば、タイトルバーに載せるアイコンやウィンドウの背景などです。それらは **WNDCLASSEX** 構造体のメンバに定義されています。すべての設定が終わったら **RegisterClassEx** 関数を使って、**OS** に対してウィンドウクラスを登録します。

ウィンドウクラスの登録が終わったらいよいよウィンドウを作成します。ウィンドウの作成には **CreateWindow** 関数を使います。ウィンドウの形や機能はここで定義します。ウィンドウが無事生成されるとウィンドウハンドルというものが割り当てられます。このウィンドウハンドルはウィンドウを制御するのに必要なものです。

さて、**CreateWindow** 関数で作成されたウィンドウはこの段階では表示されません。そこで、**ShowWindow** 関数でウィンドウを可視化する必要があります。

# 6.ウィンドウズアプリケーションの流れ

ウィンドウズアプリケーションの基本はアプリケーションに対して送られてきたメッセージに対してどのような処理を行うかということになります。たとえば、「閉じる」ボタンを押したとき、アプリケーションには"WM_CLOSE"というメッセージが送られます。そのときに特別な処理をしてからアプリケーションを終了させるならば、**WM_CLOSE** を受信したときの処理を記述する必要があります。このように、ユーザーの動作（イベント）に対して特定の動作を行う形態をイベント駆動型（イベントドリブン）といいます。ウィンドウズアプリケーションはまさにイベントドリブンなのです。

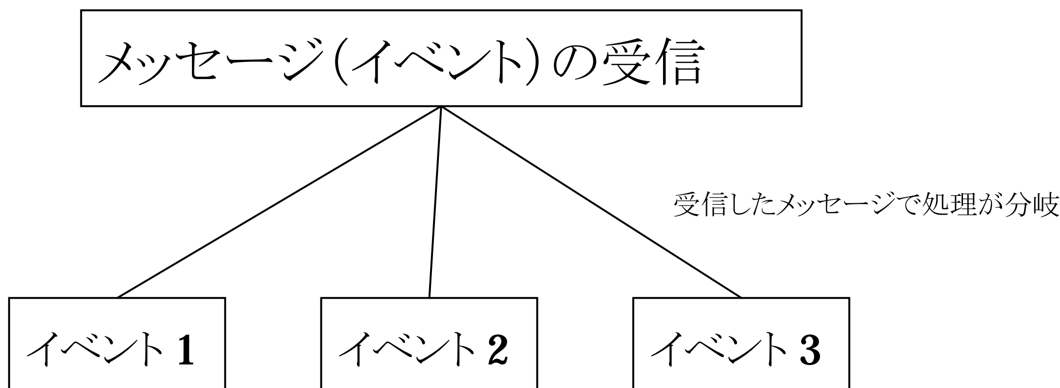


図 **6.1** イベントドリブン

# 7.ウィンドウプロシージャ

実際にメッセージの処理を行うのはウィンドウプロシージャです。プロシージャとは複数の処理をひとまとめにし

たものを言います。ウィンドウプロシージャにはさまざまメッセージに対する処理をすべてウィンドウプロシージャに記述します。

ウィンドウプロシージャは以下のように定義されています。

**LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);**

関数の定義に"**CALLBACK**"とかかれたものをコールバック関数といいます。コールバック関数はプログラマ自身が呼び出すのではなく、関数のポインタだけをプログラムに知らせておき、必要に応じてプログラムが呼び出す関数です。ウィンドウプロシージャもコールバック関数なのです。つまり、ウィンドウプロシージャはプログラマ自身が呼び出すものではないのです。

引数の役割を説明すると、**hwnd** はウィンドウハンドルです。**msg** が受信したメッセージです。ウィンドウプロシージャは **msg** の値によって処理を分岐します。**wParam** は **msg** に対応した追加情報です。たとえば、**WM_KEYDOWN**（キーボードが押された）というメッセージには **wParam** に仮想キーコードが与えられます。**lParam** も **wParam** と同様、追加情報を格納するものです。

さて、受信されるメッセージに対する処理をすべてプログラマが書くとなるとその量は膨大になります。出来るならば、自分が処理したいメッセージに関する処理だけを書きたいものです。そこで、自分が扱うメッセージ以外のものを受信したときは **DefWindowProc** 関数に処理を任せてしまいます。この関数には、すべてのメッセージに対する処理があらかじめ書かれています。自分が必要としないメッセージはすべてこの関数に任せましょう。

# 8.メッセージの取得と送信

メッセージの取得は **GetMessage** 関数で行います。**GetMessage** 関数は **WM_QUIT**（アプリケーションの終了）のメッセージを取得したとき **0** を返し、そうでないときは **0** 以外を返します。メッセージを取得したら **DispatchMessage** 関数でウィンドウプロシージャにメッセージを送信します。つまり、ウィンドウプロシージャは **DispatchMessage** 関数内で呼ばれることになります。