オブジェクト指向と ゲームプログラミング

▷ フレームワーク編 － 第０回 Windowsプログラミング ◁

データ型

Windowsプログラミングでは、intやdoubleといった基本的なデータ型のほかに、Microsoft社が定義する独自のデータ型を多く使用します。このようなデータ型は、ヘッダファイル<windows.h>およびその関連ヘッダファイル<windef.h>で定義されていおり、<windows.h>をインクルードすることにより、ほとんどのデータ型が使用できるようになっています。よく使うデータ型は以下のものです。

型	サイズ	値の範囲	備　　考
void/VOID	－	な　し	
LPVOID	32	32ビットのアドレスを格納	void*と同じ
bool	8	true, false	
BOOL	32	TRUE, FALSE	bool型とは互換性なし。実体はint型
char/CHAR	8	-128～127	
BYTE	8	0～255	unsigned charと同じ
LPBYTE	32	32ビットのアドレスを格納	unsigned char*と同じ
wchar_t	16	UNICODE文字型	1文字を16ビットで表すunicode文字を格納
WCHAR	16	UNICODE文字型	wchar_tとほとんど同じ
TCHAR	8/16	UNICODEまたはchar文字	コンパイル時にcharとwchar_t切り替えられる
short	16	-32768～32767	
WORD	16	0～65535	unsigned shortと同じ
LPWORD	32	32ビットのアドレスを格納	unsigend short*およびWORD*と同じ
int/INT	32	-2147483648～2147483647	
UINT	32	0～4294967295	unsigend intを省略した記述方法
long/LONG	32	-2147483648～2147483647	
LPLONG	32	32ビットのアドレスを格納	long*と同じ
DWORD	32	0～4294967295	unsigned longと同じ
LPDWORD	32	32ビットのアドレスを格納	unsigned long*およびDWORD*と同じ
__int64	64	-9223372036854775808 ～ 9223372036854775807	符号あり64ビット整数
DWORDLONG	64	0～18446744073709551615	符号なし64ビット整数
float/ FLOAT	32	1.175494351E-38 ～ 3.402823466E+38	
double	64	2.2250738585072014E-308 ～ 1.7976931348623158E+308	Windowsではlong double型も同等
LPSTR	32	文字列のアドレスを格納	char*と同じ。文字列へのポインタを格納
LPCSTR	32	文字列のアドレスを格納	const char*と同じ。固定文字列へのポインタ
LPTSTR	32	文字列のアドレスを格納	コンパイル時にchar*とwchar_t*を切り替えられる
LPCTSRT	32	文字列のアドレスを格納	LPTSTRの固定文字列版。文字列は変更不可
HANDLE	32	32ビット整数値	リソースを識別する値(ハンドル)を格納

※「サイズ」はデータを格納するのに必要なビット数を表しています。

WinMain関数

C／C++プログラムでは、main関数から実行が開始されますが、WindowsプログラムではWinMain関数から実行が開始されます。一般的なWindowsプログラムにはmain関数がありません。

通常、WinMain関数は、以下のような形式を取ります。

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nShowCmd)
```

プログラムが起動されると、WinMain関数はWindowsから４つのパラメータを受け取ります。それぞれ以下の情報が格納されます。

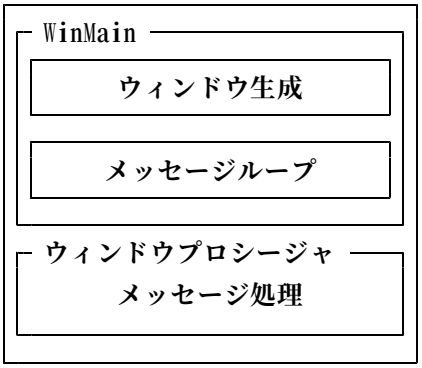
HINSTANCE	hInstance……プログラムを識別するための値(インスタンスハンドル)
HINSTANCE	hPrevInstance…常にNULL。Windows3.1との互換性のために残っています
LPTSTR	lpCmdLine……コマンドラインオプション(/Pなどの起動時のオプション文字列)
int	nShowCmd……メインウィンドウの表示モード(最大化・最小化・通常表示など)

最初の「インスタンスハンドル」は特に重要で、いくつかのAPIで必要になります。

WinMain関数は、WINAPIという呼び出し規約(関数呼び出しのきまり)を使ってコンパイルされなければならないので、戻り値intと関数名WinMainの間に「WINAPI」がついています。WINAPIのかわりに「APIENTRY」や「PASCAL」が使われることもあります。

Windownsプログラムの構成

Windowsプログラムの基本的な構成は、WinMain関数とウィンドウメッセージ(以下、メッセージ)を処理するウィンドウプロシージャという２つの関数から成ります。
WinMain関数では、ウィンドウを生成し、Windowsから送られてくるメッセージを取り出してウィンドウプロシージャに送出するためのメッセージループを確立します。
ウィンドウプロシージャは、プログラマが定義しなければならない関数で、各メッセージをどのように処理するかを記述します。



メッセージ

Windowsは、各アプリケーションが生成したすべてのウィンドウを常に監視しています。キーボードのキーが押されたり、マウスが動かされたりといったイベントが発生すると、該当するウィンドウにメッセージという形でイベントが起きたことを通知します。各ウィンドウは、送出されてきたメッセージを取り出して正しく処理しないと、自身のアプリケーションだけでなく、Windows全体の動作が不安定になってしまうこともあります。

メッセージ	発　生　条　件
WM_ACTIVATEAPP	アプリケーションがアクティブ化または非アクティブ化されようとしている
WM_CLOSE	閉じるボタンまたはAlt+F4でウィンドウを閉じようとしている
WM_COMMAND	メニューが選択された
WM_CREATE	ウィンドウが生成された(CreateWindow(Ex)関数が実行された)
WM_DESTROY	ウィンドウが破棄された(DestroyWindow関数が実行された)
WM_KEYDOWN	キーボードのキーが押された(押し続けられているときにも発生)
WM_KEYUP	キーボードのキーが離された
WM_LBUTTONDOWN	マウスの左ボタンが押された(押し続けても１度しか発生しない)

メッセージ	発　生　条　件
WM_LBUTTONUP	マウスの左ボタンを離した
WM_MOUSEMOVE	マウスカーソルが移動した
WM_MOVE	ウィンドウが移動された
WM_PAINT	クライアント領域の一部に描画が必要になった
WM_QUIT	PostQuitMessage関数を呼び出され、アプリケーションが終了しようとしている
WM_SIZE	ウィンドウのサイズが変更された
WM_TIMER	SetTimer関数で指定した時間が経過した
WM_SYSCOMMAND	最大化ボタンや最小化ボタンが押されたり、システムメニューが選択された

▲おもなメッセージと発生条件

メッセージループ

Windowsアプリケーションは、メッセージを受け取り、それを処理し、再びメッセージが送られてくるのを待つという動作を繰り返します。これをメッセージループといいます。この点がWindowsアプリケーションの最大の特徴です。Windowsアプリケーションでは、必ずメッセージループを確立し、メッセージを処理しなければなりません。

ウィンドウには、Windowsから送られてくるメッセージをためておく「メッセージキュー」という領域があります。ここにためられたメッセージは、取り出され、正しく処理するまでなくなりません。メッセージを処理しないアプリケーションは「応答なし」となり、フリーズ状態と見なされます。

一般的なメッセージループは以下のように構成されています。

```
MSG    msg;  // MSG構造体…メッセージの情報を格納します
ZeroMemory(&msg, sizeof(msg));

// メッセージループ
while(true) {
    const BOOL   ret = GetMessage(&msg, NULL, 0, 0);     // メッセージの取得
    if(ret == 0 || ret == -1)
        break;
    TranslateMessage(&msg);                              // 仮想キーコードの変換
    DispatchMessage(&msg);                               // メッセーの送出
}   // while(true)
```

GetMessage関数は、メッセージキューからメッセージを取り出します。メッセージがないときは、メッセージがくるまでプログラムを休止し、実行権を他のアプリケーションに渡します。この関数は、アプリケーションがメッセージを取り出せる状態になるたびに、継続的に呼び出す必要があります。そうしないと不安定な動作になってしまいます。

メッセージが取り出されたら、２つのAPIを呼び出します。ひとつはTranslateMessage関数です。この関数は、キーボード入力を文字メッセージに変換します(たとえば、WM_KEYDOWNとWM_KEYUPをWM_CHARとWM_DEADCHARに変換)。もうひとつはDispatchMessage関数です。この関数は、メッセージをウィンドウプロシージャに送出します。

メッセージループは、一般的にある条件を満たすまでループを続けるように構成します。メッセージループは、GetMessage関数が終了メッセージWM_QUITを受信した場合、またはエラーが発生してメッセージを取得できないときに終了させます。GetMessage関数の戻り値は、WM_QUITメッセージを受信すると０、エラーが起きた場合には－１が返されるので、これを終了条件にします。

メッセージループを抜けたWindowsアプリケーションは、MSG構造体のwParamメンバをWindowsに返し、アプリケーションを終了させます。

ウィンドウプロシージャ

すべてのウィンドウは、「ウィンドウプロシージャ」と呼ばれる特別な関数を準備しなければなりません。ウィンドウプロシージャは、メッセージループから送出されてくるメッセージを処理するために必要となる関数です。この関数はプログラマが定義しなければなりません。

GetMessage関数により取り出されたメッセージは、DispatchMessage関数によりウィンドウプロシージャへ送出されます。DispatchMessage関数が実行されると、Windowsを介してウィンドウプロシージャが呼び出されます。ウィンドウプロシージャでは、送出されてきたメッセージに対応する処理を行うことになります。

ウィンドウプロシージャのプロトタイプは以下のようになります。

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
```

「LRESULT」は戻り値の型でLONG型と同等です。次の「CALLBACK」は、APIが必要になったときに呼び出すことを表します。通常の関数のようにプログラマが必要に応じて呼び出す関数ではない、ということです。たとえば、DispatchMessage関数、CreateWindow(Ex)関数、DestroyWindow関数などのAPIが必要なときに自動的に呼び出します。

ウィンドウプロシージャが呼び出されると、Windowsから４つの引数が渡されます。それぞれ、

```
HWND   hWnd……メッセージが発生したウィンドウのハンドル
UINT   uMsg……送出されたメッセージの種類
WPARAM wParam…メッセージの追加情報(UINT型)。内容はメッセージによって異なります
LPARAM lParam…メッセージの追加情報(LONG型)。内容はメッセージによって異なります
```

という情報が格納されています。

典型的なウィンドウプロシージャは､以下のように送出されてきたメッセージをswitch文で分岐させ､それぞれにあわせた処理を行います。メッセージは１００種類以上あり、すべてのメッセージを正しく処理しなければなりません。
すべての種類にあわせた処理を記述するのは大変なので、Windowsではデフォルト処理を提供しています。デフォルト処理を利用するための関数がDefWindowProc関数です。ほとんどのメッセージはデフォルト処理に任せるのが一般的です。
デフォルト処理以外の方法でメッセージを処理した場合は、決められた戻り値を返してウィンドウプロシージャから抜けます。

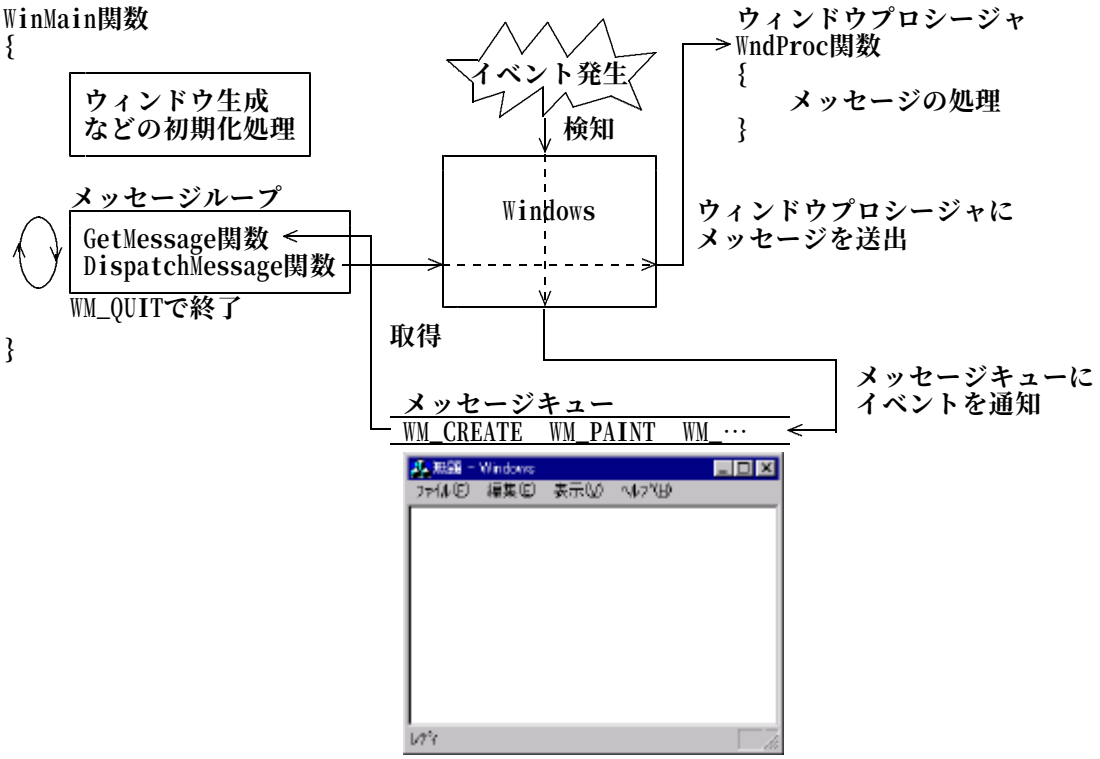
```
// ウィンドウプロシージャ(メッセージを処理する関数)
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
      // キーダウン
      case WM_KEYDOWN:
        ※キーが押されたときの処理(省略)
        return 0;

      // ウィンドウ破棄
      case WM_DESTROY:
        ※ウィンドウが破棄されたときの処理(省略)
        return 0;

      case XXXX:
        ※イベントの処理
        return x;
    }   // switch(uMsg)

    // 上記以外のメッセージはデフォルト処理に任せる
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

Windowsプログラミングは、メッセージに応答しながら処理を行うことからメッセージ駆動型プログラミングと呼ばれています。アプリケーションが独自に動作するのではなく、Windowsから取得したメッセージをトリガ(きっかけ)として初めて動作するためです。

このようなメッセージループ、ウィンドウプロシージャといった複雑な構造になっているのは、Windowsが各アプリケーションを最適にスケジュールし、安定して協調動作させるためです。
旧来のMS-DOS(コンソール)アプリケーションでは、main関数からプログラムの実行が始まり、プログラマが記述した流れにそってプログラムが実行されます。プログラムの中で文字を表示する必要があれば、そこでprintf関数などを実行することによって文字出力が行われます。MS-DOSアプリケーションでは、プログラマが記述したコードのとおりに処理が進んでいきます。

ところがWindowsアプリケーションでは、プログラマが書くコードは、システムから必要なときに呼び出されるようになっています。Windowsの世界では、プログラマはプログラムの流れを取り仕切るのではなく、メッセージを受け取ったときのアプリケーションの動作という局部的なコードを記述するのです。「プログラムの流れはこうしよう」という考え方ではなく、「このメッセージを受け取ったらこうしよう」という考え方になっています。

このようなシステムからのメッセージに応答するようなプログラム構造は、キーボードやマウスからの入力がない限り処理を行わないワープロや表計算といったビジネスアプリケーションと非常に相性が良いのですが、常にリアルタイムで動作しなければならないゲームプログラムと相性が良いとはいえません。



▲Windowsアプリケーションの基本プログラミングモデル