CODE GEAR FROM Borland Developer Camp

【A4】Delphi/C++Builderテクニカルセッション

「Delphi⇔C++Builder相互移植テクニック」

株式会社シリアルゲームズ
取締役/シニアエンジニア
細川　淳

CODE GEAR FROM Borland Developer Camp

Delphi言語とC++言語

本文書の一部または全部の転載を禁止します。本文書の著作権は、著作者に帰属します。

Delphi言語の特徴

- Pascalベース
  - トップダウン構文解析で解析可能（1パス）
- オブジェクト指向言語
  - 機能を絞り、単純化されている
  - 多重継承では無く、インターフェースを採用
- 非常に厳格な型を持つ
  - type TFoo = type Integer; といった厳格な型定義や
  - String[文字列長] や set of を使うと違う型となる
- 処理系は CodeGear Delphi のみ
  - 言語仕様の変更は CodeGear 社の自由に
  - バージョンを重ねるにつれ様々な拡張が施された
  - ライブラリは VCL が標準として提供されている

C++言語の特徴

- Cベース
  - ボトムアップ型構文解析でも完全な解析はできない（マルチパス）
- オブジェクト指向言語
  - 様々な機能を持ち、複雑な言語
  - 多重継承を採用
- 厳格な型を持つ
- 処理系は世界中に数多く存在する
  - 言語仕様の策定は C++標準化委員会により検討される
  - 初期のバージョンから、機能はあまり増えていない（C++という名称になってから）
  - ライブラリは標準ライブラリやSTLといったものがある

型

Delphi言語とC++言語の「型」

Delphi言語の型

- プリミティブ型
  - **整数型**
    - Integer　符号あり整数
    - Cardinal　符号無し整数
    - int64　64bit符号あり整数
  - **実数型**
    - Single　単精度実数
    - Double　倍精度実数
  - **文字型**
    - Char　文字
  - **論理型**
    - Boolean　論理
  - **ポインタ型**
    - Pointer　型無しポインタ

Delphi言語の型

- 複合型
  - **文字列型**
    - String　長い文字列
    - ShortString　短い文字列
  - **列挙型**
    - **(enum1, enum2, ...)**
  - **集合型**
    - **set of**
  - **配列型**
    - array [範囲] of　配列
    - array of　動的配列／オープン配列
  - **構造体**
    - record　構造体（共用体）
  - **バリアント型**
    - Variant　動的型
  - **クラス型**
    - **class**　**クラス型**
    - **class of**　**メタクラス型**

C++言語の型

- プリミティブ型
  - **整数(32bit)**
    signed, unsigned, long などの接頭辞を付けることが可能
    - char　1byte
    - short　2byte
    - int　4byte
  - **実数型**
    - float　32bit 浮動小数点数
    - double　64bit 浮動小数点数
      - long double で80bit 浮動小数点数
  - **論理型**
    - bool　論理型
  - **ポインタ型**
    - void*　型無しポインタ
    - func　関数型
  - **参照型**
    - &ref　参照型

C++言語の型

- 複合型
  - **列挙型**
    - **enum example (enum1, enum2, ...)**
  - **配列型**
    - **int array[10];**
  - **構造体**
    - struct　　構造体
    - union　　共用体
  - **クラス型**
    - **class**　　**クラス型**

Delphi言語と C++言語のプリミティブ型対応表

Delphi	C++
整数型	
Byte	unsigned char
ShortInt	char
Word	unsigned short int
SmallInt	short int
Cardinal	unsigned int
Integer (LongInt)	int (long int)
Int64	long long int
実数型	
Single	float
Double (Real)	double
Extended	long double
文字型	
Char	unsigned char

Delphi	C++
論理型	
Boolean	bool
ポインタ型（型無し）	
Pointer	void*

※Delphi の Byte, Word, Int64 等のバイト・ビット数指定の型のC++での対応は全て32ビット環境での対応

各型の移植のポイント

Delphi言語とC++言語で異なる「型」の移植方法

文字列型

- Delphi の組み込み型である String は C++Builder ではクラスとして実装されている
- Delphi の文字列型は以下の3つ
  - String (AnsiString)　デフォルトのロケールで動く文字列
  - WideString　Unicode 文字列
  - ShortString　255 文字まで表せる文字列
- C++ Builder の文字列クラスは以下の3つ
  - AnsiString　デフォルトのロケールで動く文字列クラス
  - WideString　Unicode 文字列クラス
  - SmallString　ShortStringをエミュレートするテンプレートクラス

文字列型の振る舞い

- Delphi の String 型は特別な振る舞いをする
  - String 型の変数は初期化済み
  - Unicode を表す WideString 型と代入互換
  - PChar(String)は String.c_str() と同じ
    - Win32API を呼び出すようなコードでは注意する
  - インデックスは1から始まる
    - ShortString のインデックス0は文字列の長さを表す
- DLL とアプリケーション間の引数に String 型は使わない方が無難
  - 使用する場合は BORLNDMM.DLL が必要になる
  - PChar(String) や String.c_str(); を使用する

Delphi の文字列操作関数

- Delphi の String 型は組み込み型のため文字列操作関数はグローバル関数として定義されている
  - 例えば Length 関数を使って文字列の長さを返す

```Delphi
procedure Sample;
var
  Ansi: String;
  Wide: WideString;
begin
  Ansi := 'コードギア';
  Wide := Ansi; // 代入互換
  ShowMessage(IntToStr(Length(Ansi)) + ', ' + IntToStr(Length(Wide)))
end;
```

実行結果

C++Builder の文字列操作メソッド

- C++Builder では AnsiString クラスのメソッドとして文字列操作関数が定義されている
  - Delphi の Length 関数と同名のメソッドが同等の働きをする

C++
```
void __fastcall Sample()
{
  AnsiString Ansi = "コードギア";
  WideString Wide = Ansi;

  ShowMessage(String(Ansi.Length()) + ", " + String(Wide.Length()));
}
```

実行結果

文字列型互換表

- String 型互換表

Delphi	C++ Builder
String (AnsiString)	AnsiString クラス
WideString	WideString クラス
ShortString	SmallString テンプレートクラス
PChar(String)	AnsiString.c_str()
PWideChar(String)	WideString.c_bstr()

文字列型移植のポイント

- Delphi から C++Builder へ移植する際は特に気をつける事はないが、グローバル関数として定義されている文字列関数について知っておく必要がある
  - 大多数の関数はメソッドとして定義されているが、当然定義されていない物もあるので注意（AdjustLineBreak関数など）
- C++Builder から Delphi への移植では、C/C++ 固有の文字列操作関数（stdio.h や string.h などで定義される物）を適切な関数に置き換える必要がある（→ヘルプの「文字列処理ルーチン（ヌルで終わる）」を参照する）
  - strstr 関数などは StrPos 関数となっていたり関数名は一致していないので注意
  - それほど数がないのであれば、全て String 型に置き換えてしまう方が楽な場合が多い

文字列型移植のポイント（裏技）

- コンパイラが Delphi/C++Builder 限定ならば文字列処理部分は全て Delphi 側で書いてしまうという手が使える
- 文字列処理は Delphi 言語で書き、そのラッパークラスや関数だけ C++ Builder 側で用意する

列挙型

- Delphi の列挙型は type 節で定義する
  - type TSampleEnum = (seOne, seTow, seTree);
  - 列挙型は新しい型として定義される（非常に厳格な型をもつ）
  - 列挙型を Ord 関数で順序値に変換すると最初の値は0となる
    - Ord(seOne); // = 0
- C++ の列挙型は enum を用いる
  - enum TSampleEnum {seOne = 1, seTwo, seTree};
  - C++ の列挙型は int 型の定数であり、厳密には Delphi の列挙型とは異なる
  - int 型の定数であるため、列挙型の値を自分で定義できる（上記の seOne = 1の部分）
- Delphi / C++ 共にサイズは可変であることに注意
  - Delphi　SizeOf(TSampleEnum); // = 1
  - C++　sizeof(TSampleEnum); // = 1

列挙型移植のポイント

- Delphi から C++ への移植には難しい部分は特になく、type 節にある型定義を enum に置き換えればOK
  - ただし、Ord, Pred, Succ などの関数には注意
- C++ から Delphi への移植では、C++ の列挙型が int型の定数であることに注意が必要
  - 途中から値の変わる列挙型
    - **enum** {seOne = 1, seTwo, seThree, seTen = 10}
  - int 型との代入互換性（Cの場合）
    - **int** Test = seOne; // C では正しいが C++ では不正

集合型

- Delphi では集合型が組み込み型として定義されている
  - set of 順序型、で定義される

```
TSample = (seOne, seTwo, seThree)
TSamples = set of TSample;
```

  - 集合演算子が使用できる
    - +, -, *, in, <, =, > など
  - 集合構成子[]で代入可能

```
var
  Sample: TSample;
begin
  Sample := [seOne.. seThree]; // 構成子には .. が使用できる
end;
```

```
var
  Sample: TSample;
begin
  Sample := []; // 空集合の代入
end;
```


集合型移植のポイント

- Delphi の集合型は C++Builder ではテンプレートクラスとして定義されているため、特に苦労せず移植可能となる

例

```
typedef Set<TSample, seOne, seThree> TSamples;
TSamples Sample << seOne << seTwo;
```

  - Set テンプレートクラスのメソッドと集合演算子の対応に注意
    - Delphi の in 演算子は Set.Contains メソッドとなる
- 集合型プロパティ
  - Font.Style プロパティなど集合型プロパティへの代入には注意
    - Form1->Font->Style << fsBold; // 間違い
      - 現在のFontStyle を値に fsBold を追加しているだけ
    - Form1->Font->Style = Form1->Font->Style << fsBold;
      - 代入文を用いて値を設定する必要がある

配列型

- Delphi の配列は array of で定義する
  - **var** SampleArray: **array** [1.. 5, Boolean] **of** Integer;
  - 配列の添え字はユーザーが自由に定義できる
    - 添え字の範囲を指定するか、型を指定する
    - 上の例では Boolean 部は True, False を指定できる
      - SampleArray[4, True]; // このように使用できる
  - packed 指令をつけると配列のアラインを調整できる
    - **var** SampleArray2: **packed array** [0.. 5] **of** Byte; // バイト単位で確保される
  - コンパイラが自動的に長さや生成・破棄を管理する「動的配列」がある
    - **var** DynArray: **array of** Integer; // と範囲を指定しない場合動的配列となる
    - 動的配列は SetLength 関数で長さを指定する
- C++ の配列は [] で定義する
  - **int** SampleArray[5][2];
  - 配列の添え字は常に0から
  - C++ の配列はポインタの高級表現

配列型の移植のポイント

- Delphi の配列を C++ に移植する際は、添え字の範囲と型に注意
  - Delphi では Sample: **array** [Boolean] **of** Integer; のような定義は比較的多く行われているため、Boolean といった型指定部分を全て置き換えていくか、CBooleanArray のようなクラスを作成する
  - 動的配列が使用されている場合は、ポインタとして実装できないか考えてみるか、クラス化する
- C++ の配列を Delphi に移植する際は、C++ の配列がポインタとしても活用される事に注意する

```
char Sample[5];
char* CharPtr = Sample;
```

  - このような移植は一筋縄ではいかないため、アルゴリズムレベルから変更することも視野に入れる
- Delphi / C++ では、それぞれ多次元配列の表現方法が異なる事に注意
  - Delphi　SampleArray[0, 1, 2]; // のように多次元配列はカンマで区切る
  - C++　SampleArray[0][1][2]; // のように多次元配列は配列の中身が配列として考える

構造体

- Delphi の構造体は record で表す
  - 共用体(可変部)の定義は case を用いる
  - 構造体中の文字列型も自動的に初期化される事に注意する
- C++ の構造体は struct を用いる
  - 共用体の定義は union を用いる

構造体例 - Delphi ⇔ C++

Delphi

```
THuman = record
  Old: Byte;
  Description: String;
  case Boolean of
    False: (VisionA: Single);
    True: (VisionB, CorrectedVision: Single);
end;
```

C++ Builder

```
// #pragma pack(4)
struct THuman {
  char Old;
  AnsiString Description;
  union {
    float VisionA;
    struct {
      float VisionB;
      float CorrectedVison;
    } Vision;
  } Grasses;
};
```

共に構造体が使用するサイズは16バイトとなる

構造体移植のポイント

- #pragma pack と packed record
  - Delphi / C++共にデフォルトの構造体アラインは4バイト
  - Delphi では packed 指令を用いて詰めることができる
  - C++ Builder では pragma pack でアラインを自由に定義出来る
  - Win32API の構造体を Delphi から使用する場合は packed record を用いる
- Delphi の構造体可変部
  - 可変部にタグをつけるとその分も確保される

```
TSample = record
  case Test: Boolean of
    False: (A: Byte);
    True: (B: Integer);
end;
```

← 使用サイズは4になりそうだが8になる

関数（手続き）

- Delphi の関数は procedure / function で定義する
  - procedure は戻り値のない関数（手続き）
    - 例: **procedure** Sample(vArg: Integer);
  - function は戻り値のある関数
    - 例: **function** Sample2(vArg: **String**): Integer;
- C++ の関数定義は下記の各部分を指定して定義する
  - 記憶クラス指定子(extern / static)(オプション)
  - 戻り値の型(void や int)
  - 修飾子(__fastcall, __cdecl, __export など)(オプション)
  - 関数名
  - パラメータ宣言リスト
    - 例: **int __fastcall** Sample3(vArg: **bool**);

関数（手続き）Delphi

- Delphi の関数は引数が0個の時、引数リストを表す括弧を省略できる
  - procedure Test; という宣言の手続きTestは「Test();」と書いても「Test;」と書いても動作する
    - 引数を採らない関数を定義するとSingletonのような機構を実現出来る
      - VCL の Printer 関数など
- Delphi の関数は戻り値を Result 変数に入れて返す
  - Pascal では関数名に戻り値を代入するが、これだと再帰関数を定義できないことから Result 変数が導入された

Pascal の例

```
function Test: Integer;
begin
  Test := Random(10);
end;
```

Delphi の例

```
function Test: Integer;
begin
  Result := Random(10);
end;
```


関数（手続き）Delphi

- Delphi の関数は値渡しと参照渡しの指定方法がいくつかある
  - 指定しない　値渡し
  - var　参照渡し
  - out　参照渡し（値を渡すことは出来ず、戻り値を受け取るのみ）
  - const　参照渡し（ただし値を変更できない）
- Delphi の関数は特殊なパラメータを取ることがある
  - 型なしパラメータ
    - var, out, const が指定された場合、型を省略できる
      - 例: procedure Sample(const Arg);
      - 渡された参照をどのように使うかは関数内で決定できる
  - オープン配列パラメータ
    - 引数の数が任意
      - 例: procedure Sample2(Arg: array of Integer); // 整数型の引数をいくつでも指定できる
        Sample2([1, 2, 3, 4, 5]); // [ ]（オープン配列コンストラクタ）を使用していくつでも指定できる
  - 型可変オープン配列パラメータ
    - 任意の型の引数を任意の数受け取ることが出来る
      - 例: procedure Sample3(Arg: array of const); // 型名を const として指定する

関数（手続き）のポインタの例

Delphi の例

```
type
  TSampleFunc = funciton(vArg: Integer): Boolean;
var
  Func: TSmapleFunc;

function IsPlus(vArg: Integer): Boolean;
begin
  Result := (vArg > 0);
end;

begin
  Func = IsPlus;
  Func(1);
end;
```

C++ の例

```
typedef bool (*TSampleFunc)(int);

bool Test(int Arg)
{
  return Arg > 0;
}

TSampleFunc Func = &Test;

(*Func)(1);
```

メソッドポインタの場合 of object を付ける

```
type
  TSampleFunc = funciton(vArg: Integer): Boolean of object;
```

メソッドポインタの場合クラス名を付ける

```
typedef bool (*ClassName::TSampleFunc)(int);
```


関数（手続き）の移植のポイント C++ → Delphi

- C++ の関数を Delphi に移植するのはさほど難しくない
  - va_list, va_start, va_end はオープン配列パラメータに変更する
  - 戻り値の const は外してしまうなど
    - 戻り値を扱う側で気を配るようにする

関数（手続き）の移植のポイント Delphi → C++

- Delphi の関数を C++ に移植する場合は、前述の特殊な引数の型や Result 変数に気をつける
  - 型なしパラメータなどは、全てポインタに置き換える
    - C++Builder で Delphi のオープン配列パラメータは、ポインタ（配列）と配列の長さを受け取る変数の2つに別れて実装される
  - Delphi には Result 変数があるため、出口を1カ所に纏めようとする傾向があるため注意する

Delphi

```
function Sample(vArg: Integer): Boolean;
begin
  if (vArg > 0) then
    Result := True
  else
    Result := False;
end; // 出口はここのみ
```

C++

```
bool Sample(int vArg)
{
  if (vArg > 0)
    return true;  // 出口１
  else
    return false; // 出口２
}
```


クラス - Delphi

- Delphi のクラスの特徴
  - TObject が全てのクラスの基底として存在する
  - インスタンスはポインタとして実現される
    - Sample := TSample.Create(Application);
  - 多重継承はできない
  - インターフェースを実装できる
  - テンプレートのような機能はない
  - 可視性が定義されていない場合は public となる
  - 1ユニット内で定義されているクラスは互いに相手の Private / Protected 部を参照できる（friend）
  - クラスメソッドは class キーワードをメソッド名の前に付ける
    - **class procedure** SampleMethod;
  - 自身を表す特殊な変数は Self
  - クラスを表す型（メタクラス）がある

クラス - C++

- C++ のクラスの特徴
  - 特別な基底クラスは存在しない
    - VCL から派生したクラスは TObject クラスが基底クラスとなる
  - インスタンスは new で動的に生成することも、静的に生成することも可能
    - Sample = **new** TSample(Application);
    - TSample Sample(Application)
  - 多重継承できる
  - インターフェースは言語レベルでは定義されていない
  - テンプレートなどの高度な機能がある
  - 可視性が定義されていない場合は private となる
  - friend を使用すると特定のクラスに自分のプライベート変数などを公開できる
  - 静的メンバーは static を宣言の前に付ける
    - **static void** SampleMethod();
  - 自身を表す特殊な変数は this

クラス移植のポイント

- 簡単なクラスであれば特筆すべき移植ポイントはない
- C++のテンプレートやDelphiのメタクラスを移植する場合、アルゴリズムレベルから変更を検討する
  - テンプレートは必要な型の分だけクラスを定義するか、クラスヘルパーやポインタで実装できないかなどを検討する
  - メタクラスの場合は、共通基底クラスに機能を集約できないか、などを検討する

クラス移植の例

- 非常に大きな整数を扱う TBigInteger クラスを移植する
- 元のクラスは Delphi 言語（Delphi 7）で書かれていた
- これを C++ 言語に移植する

クラス移植の例1

Delphi 7では定数を定義に含められないので
implementation 部で定数を定義する
最新の Delphi ではクラス定数を定義できる

Delphi

```
type
  TBigIntegerDigits = array of Cardinal;

  TBigInteger = class(TObject)
  private
    // Variables
    FDigits: TBigIntegerDigits;
    FSign: Integer;
    FBase: Cardinal;
    FMaxValue: Cardinal;
    FBaseBitCount: Cardinal;
    FBaseIsOdd: Integer;
```

C++

```
class CBigInteger
{
  private:
    // Constants
    static const unsigned int CBase = 32767; // 2 ^ 15 - 1;
    static const unsigned int CArraySize = sizeof(unsigned int);
    // Variables
    unsigned int* FDigits;
    unsigned int FDigitsLength;
    int FSign;
    unsigned int FBase;
    unsigned int FMaxValue;
    unsigned int FBaseBitCount;
    bool FBaseIsOdd;
```

Delphi 側では FDigits は符号無し整数の動的配列として定義されているが
C++ 側では符号無し整数へのポインタとして定義されている
また Delphi 側では FDigits の長さを Length() 関数で取れるため C++ 側のような FDigitsLength は定義していない

クラス移植の例2

Delphi

```
SetLength(FDigits, Count)
```

C++

```
protected:
  // Methods
  void setLength(CBigInteger&, const int);
```

```
void CBigInteger::setLength(CBigInteger& vBigInteger, const int vCount)
/*
 * 概要  配列の長さを変更する
 * 引数  vBigInteger  長さを変更したい FDigits の CBigInteger のインスタンス
 *       vCount       新しい長さ
 * 備考  Delphi の setLength 関数と同じ動作
 */
{
  if (vBigInteger.FDigitsLength != (unsigned int)vCount)
    vBigInteger.FDigits =
      (unsigned int*)realloc(vBigInteger.FDigits, vCount * CArraySize);

  vBigInteger.FDigitsLength = vCount;
}
```

Delphi が元クラスのため Delphi の組み込み関数である SetLength を C++ 側で実装することにより移植の手間を省いている

クラス移植の例3

Delphi

```
public
  // Constructor & Destructor
  constructor Create(const vBase: Cardinal = 0); reintroduce;
  constructor CreateInteger(
    const vInteger: Integer;
    const vBase: Cardinal = 0);
  constructor CreateBigInteger(
    const vBigInteger: TBigInteger;
    const vBase: Cardinal = 0);
  constructor CreateString(vString: String; const vBase: Cardinal = 0);
  destructor Destroy; override;
```

C++

```
public:
  // Constructor & Destructor
  CBigInteger(void);
  CBigInteger(const unsigned int);
  CBigInteger(const CBigInteger&);
  CBigInteger(const CBigInteger&, const unsigned int);
  CBigInteger(const unsigned int, const bool);
  CBigInteger(const unsigned int, const unsigned int);
  CBigInteger(const char*, int);
  CBigInteger(const char*, int, const unsigned int);
  ~CBigInteger(void);
```

Delphi ではコンストラクタの名称を自由に設定できるが C++ では設定できないことに注意しながら移植する

クラス移植の例4

Delphi

```
function Add(const vNum: TBigInteger): TBigInteger; overload;
function Add(const vNum: Integer): TBigInteger; overload;
function Sub(const vNum: TBigInteger): TBigInteger; overload;
function Sub(const vNum: Integer): TBigInteger; overload;
function Mul(const vNum: TBigInteger): TBigInteger; overload;
function Mul(const vNum: Integer): TBigInteger; overload;
function Dv(const vNum: TBigInteger): TBigInteger; overload;
function Dv(const vNum: Integer): TBigInteger; overload;
function Md(const vNum: TBigInteger): TBigInteger; overload;
function Md(const vNum: Integer): TBigInteger; overload;
```

C++

```
CBigInteger operator +(CBigInteger&);
CBigInteger operator +(int);
CBigInteger operator -(CBigInteger&);
CBigInteger operator -(int);
CBigInteger operator *(CBigInteger&);
CBigInteger operator *(int);
CBigInteger operator /(CBigInteger&);
CBigInteger operator /(int);
CBigInteger operator %(CBigInteger&);
CBigInteger operator %(int);
```

最新の Delphi では演算子のオーバーロードが実装されているが、当時の Delphi 7 では実装されていなかったため、この例の様な実装がなされた
もし、Delphi で演算子のオーバーロードを行うと次のようになる

```
// + 演算子が使用されるとこの関数が呼ばれる
class operator Add(vNum1, vNum2: TBigInteger): TBigInteger;
```


その他の注意点

コンパイラ指令

- Delphi のソースコードを C++Builder で、そのまま使用する場合、いくつかのコンパイラ指令が必要な場合がある
- これらの指令は C++Builder のコンパイラが hpp ヘッダファイルを作成する場合に作用する

HPPEMIT コンパイラ指令

```
unit uWebBrowserEx;

interface

uses
  Windows, Messages, Classes, Graphics, StdCtrls, ShDocVw, ActiveX, OleCtrls;

{$HPPEMIT '#include <vcl.h>'}
{$HPPEMIT '#include <DocObj.h>'}
{$HPPEMIT '#include <oleidl.h>'}
{$HPPEMIT 'DECLARE_DINTERFACE_TYPE(IOleCommandTarget)'}
{$HPPEMIT 'DECLARE_DINTERFACE_TYPE(IDropTarget)'}
```

HPPEMIT コンパイラ指令は HPP 作成時、ソースに文字列を追加する。
これは上記の例の様に特定のヘッダをインクルードさせたいときや、interface の定義がある場合などに使用する。

EXTERNALSYM コンパイラ指令

```
const
  {$EXTERNALSYM EM_CANPASTE}
  EM_CANPASTE = WM_USER + 50;
  {$EXTERNALSYM EM_PASTESPECIAL}
  EM_PASTESPECIAL = WM_USER + 64;
  {$EXTERNALSYM EM_FINDTEXT}
  EM_FINDTEXT = WM_USER + 56;

  {$EXTERNALSYM DVASPECT_CONTENT}
  DVASPECT_CONTENT = 1;
  {$EXTERNALSYM EM_GETOLEINTERFACE}
  EM_GETOLEINTERFACE = WM_USER + 60;
  {$EXTERNALSYM EM_SETOLEINTERFACE}
  EM_SETOLEINTERFACE = WM_USER + 70;
```

EXTERNALSYM コンパイラ指令は、指定のシンボルを HPP から省くよう指示する。
Delphi のユニットでは定義されていないが、C++ のヘッダでは定義されているシンボルなどに使用する。

HPPEMIT を使用したテクニック

```
// 引数を表すリンカ文字列を無理やり void* から DWORD に変更する
{$HPPEMIT '#pragma alias "@Uutils@SetFront$qqrpvo"="@Uutils@SetFront$qqruio"'}

procedure SetFront(vHandle: HWND; vWait: Boolean);
```

HPP 作成時、思い通りの型に変換されない場合がある。
上記の例の場合、HWND が void* へ変換されてしまったので、C++ の pragma を用いて希望の型へ変換した。

pv = void*
ui = unsigned int = DWORD

その他の注意

- repeat ~ until と do ~ while では真理が逆になる
  - repeat until は、条件式が偽の状態でループする
  - do while は、条件式が真の状態でループする
- with 文が何に掛かっているかに気をつける
- 関数内関数
  - Delphi の関数内関数は C++ では #define でエミュレートするかローカル変数ともども外に出してしまう
- ショートサーキット論理式評価
  - Delphi のコンパイラ指令 $B- / BOOLEVAL OFF が指定された場合に注意する