WideStudio
プログラミング・ガイド

目 次

第1章　イベントプロシージャ編

1.1　オブジェクトにアクセスするには

イベントプロシージャにおいて、オブジェクトにアクセスすることは、最も基本的な事柄です。いろいろな場合における、オブジェクトへのアクセス方法を説明します。

- プロシージャ関数に渡される引数によるアクセス

  プロシージャ関数に渡される引数によるアクセスは、イベントプロシージャが貼られているクライアントオブジェクトにアクセスする場合に利用します。

- オブジェクト管理を利用したアクセス

  イベントプロシージャが貼られているクライアントオブジェクト以外の他のオブジェクトにアクセスする場合に利用します。

- 外部変数による直接アクセス

  イベントプロシージャが貼られているクライアントオブジェクト以外の他のオブジェクトにアクセスする場合に利用します。イベントプロシージャの移植性が低下します。

1.1.1　プロシージャ関数に渡される引数によるアクセス

イベントプロシージャに渡される引数を利用すると、イベントプロシージャが貼られているクライアントオブジェクトにアクセスすることができます。

```
void event_procedure(WSCbase* object){
  //オブジェクトへアクセス
  object->setProperty(WSNlabelString,"HELLO WORLD");
}
```

object ポインタがそのクライアントに対する WSCbase* ポインタです。WSCbase クラスの API を利用できますが、派生クラスの API を利用したい場合は、クラスポインタの取得の節を参照下さい。

1.1.2　オブジェクト管理を利用したアクセス

オブジェクト管理インスタンス (ロードモジュールにつき、一つ存在) に対して要求すると、アクセスしたいオブジェクトを取得することができます。外部変数アクセス (下記参照) の場合と異なり、コンパイル時にシンボルのリンクをを伴いません。したがって、画面の構成によらない柔軟なプログラムを行うことができます。

オブジェクト管理クラス	インスタンス取得関数
WSCbaseList	WSCbaseList* WSGIappObjectList()

アクセスしたいオブジェクトの取得は、次の様に行います。

```
#include <WSCbaseList.h> //WSGIappObjectList() にアクセスする...
...
void event_procedure(WSCbase* object){

  //オブジェクト管理による WSCbase ポインタの取得（その１）
  char* class_name = "WSCvlabel"; //ラベルクラス
  char* obj_name   = "newvlab_001";     //newvlab_001 という名称
  WSCbase* obj = WSGIappObjectList()->getInstance(class_name,obj_name);

  //WSCbase* クラスポインタによるラベルに対するアクセス
  obj->setProperty(WSNlabelString,"HELLO WORLD");

  //オブジェクト管理による WSCbase ポインタの取得（その２）
  char* class_name2 = "WSCbase";  //どのクラスかを特定しない
  char* obj_name2   = "newvlab_002";     //newvlab_002 という名称
  WSCbase* obj2 = WSGIappObjectList()->getInstance(class_name2,obj_name2);

  //WSCbase* クラスポインタによるラベルに対するアクセス
  obj2->setProperty(WSNlabelString,"HELLO WORLD");
}
```

obj もしくは obj2 がアクセスしたいオブジェクトです。クラス名称とオブジェクト名称を引数にします。もしクラス名を特定したくない場合は、”WSCbase” を与えてください。この場合はすべてのオブジェクトが検索対象となります。

1.1.3 外部変数による直接アクセス

オブジェクトを外部変数参照定義すると、外部変数としてアクセスできます。外部変数定義は、アプリケーションビルダユーザーズガイドの［外部変数として可能なオブジェクトとするには］の節を参照下さい。

```
#include <WSCvlabel.h> //WSCvlabel クラスを直接触るので、インクルード。
...
void event_procedure(WSCbase* object){
  //外部変数の extern 宣言
  extern WSCvlabel* newvlab_001;

  //WSCvlabel* クラスの newvlab_001 に対するアクセス
  newvlab_001->setProperty(WSNlabelString,"HELLO WORLD");
}
```

1.2 オブジェクトのプロパティ値を取得/設定するには

イベントプロシージャにおいて、オブジェクトのプロパティに対してアクセスすることができます。次の API を利用します。

アクセス関数	機能
getProperty()	プロパティ値の取得
setProperty()	プロパティ値の設定

1.2.1 プロパティ値の取得

オブジェクトのプロパティを取得するには、WSCbase クラスのメンバ関数である getProperty() を利用します。

```
void event_procedure(WSCbase* object){

  //WSNx（X 座標）プロパティの値の文字列による取得
  WSCstring x = object->getProperty(WSNx);
  printf("x=%s\n",(char*)x);

  //WSNy（Y 座標）プロパティの値の取得
  short y =  object->getProperty(WSNy);

}
```

WSNx の例では、文字列で値を取得しています。文字列クラス WSCstring は、内部で文字列領域の管理を時動的に行うので、プログラマによる管理の心配はいりません。

WSNy の例では、数値型で受け取っています。関数 getProperty() は、WSCvariant 型で値を返しますが、WSCvariant 型は、それぞれの型に自動的にキャストされます。したがって、型変換を WSCvariant 型が行うので、プログラマが気にするころはありません。

次の例は、整数型を、文字列型に変換するものを示します。

```
void cbop(WSCbase* object){

  //WSNx（X 座標）プロパティの値の文字列による取得
  WSCstring x = object->getProperty(WSNx);
  printf("x=%s\n",(char*)x);

  //WSNy（Y 座標）プロパティの値の取得
  short y =  object->getProperty(WSNy);

  //整数型を文字列に変換
  WSCvariant stry = y;
  printf("y=%s\n",(char*)stry);
  //VARIANT 型を浮動小数点型に変換
```

```
  printf("y=%fl\n",(double)stry);
}
```

プロパティ値を、文字列で取得する場合における注意点ですが、char* で直接取得する行為はやってはいけません。

char* 文字列を取得したい場合は、下記の例のように一旦、WSCstring で取得してから、char* 文字列を取得してください。一旦、WSCstring で受け取ることで、char* 文字列領域が WSCstring 内に確保され、参照可能となります。

悪い例のように、直接、char* で取得すると、getProperty が返した WSCvariant が保持されること無く、開放されるため、取得した、char* がすぐに無効となってしまいます。この無効領域となってしまった char* にアクセスすると、メモリフォルトを引き起こします。

```
void event_procedure(WSCbase* object){

  //WSNlabelString 表示文字列プロパティの文字列による取得
  //やってはいけない例
  char* string = object->getProperty(WSNlabelString);

  //WSNlabelString 表示文字列プロパティの文字列による取得
  //良い例
  WSCstring string1
  string1 = object->getProperty(WSNlabelString);
  char* str = (char*)string1;

}
```

1.2.2 プロパティ値の設定

オブジェクトのプロパティを設定するには、WSCbase クラスのメンバ関数である setProperty() を利用します。次の例は、

```
void event_procedure(WSCbase* object){

  //WSNx（X 座標）プロパティの文字列による設定
  char* x="100";
  object->setProperty(WSNx,x);

  //WSNy（Y 座標）プロパティの設定
  short y=100;
  object->setProperty(WSNy,y);

}
```

WSNx の例では、文字列で値を設定しています。WSNy の例では、整数型で値を設定しています。関数 setProperty() は、WSCvariant 型を引数にします。したがって、いろいろな型を自動的

にキャストして、受け付けることができます。プログラマは、型を気にせずアクセスできます。

1.2.3 プロパティ値の描画への反映

通常、イベントプロシージャの実行直後に反映されますが、変更後、プロパティの値を直ちに反映させたい場合、update()、draw()、redraw() を呼び出します。

ウィンドウシステムによっては、(例えば、X11 システムの場合など) 描画を行った際、すぐにウィンドウシステムに反映されない場合があります。そのような場合は、WSDappDev クラスの update() を呼び出してください。

```
#include <WSDappDev.h>

void event_procedure(WSCbase* object){

  obj1->getProperty(WSNlabelString,"テキスト");
  obj1->update();  //直ちにオブジェクトを更新。
  WSGIappDev()->update(); //ウィンドウシステムへの描画要求を反映させます。

  obj2->getProperty(WSNlabelString,"テキスト");
  obj2->update();  //直ちに描画される
  WSGIappDev()->update(); //ウィンドウシステムへの描画要求を反映させます。
```

1.3 特定のオブジェクトクラスのポインタを取得するには

特定のクラスの API にアクセスする場合、そのクラスのポインタでなければなりません。従ってなんらかの形で、そのクラス型のポインタを取得していなければなりません。ここではそのクラス型のポインタの取得の方法を説明します。

- WSCbase クラスの cast() メンバ関数による取得

クラスポインタ取得 API	機能
void* WSCbase::cast(char* className)	指定されたクラス名のポインタを返値します。

WSCbase::cast() メンバ関数によるクラスポインタの取得を説明します。object が WSCvtoggle クラスのオブジェクトであり、トグルの選択状態を調べる、WSCvtoggle のメンバ関数 setStatus() にアクセスしたい場合の例です。

C++言語では、通常、派生元へのキャストはできますが、派生元から、派生クラスへのキャストはサポートされません。WSCbase クラスの cast() 関数は、その機能を補います。

```
#include <WSCvtoggle.h> //WSCvtoggle クラスを直接アクセスするので。
...
```

```
void cbop(WSCbase* object){
  //WSCvtoggle クラスの getStatus() 関数に対するアクセス(1)
  WSCvtoggle* tgl = (WSCvtoggle*)object->cast("WSCvtoggle");

  if (tgl == NULL){
    //object は、WSCvtoggle クラスのインスタンスではなく、キャストに失敗。
  }else{
    //object は、WSCvtoggle クラスのインスタンスであり、キャストに成功。
    WSCbool status = tgl->getStatus();
  }
}
```

cast() 関数は、インスタンスが持つ、全ての継承クラスのポインタを取得することができます。
しかし、継承継承に含まれない型を指定すると NULL が返されます。この機能をうまく利用すると、次のイベントプロシージャの様にクラスの判別に応用することが出来ます。

```
#include <WSCvbtn.h> //WSCvbtn クラスに直接アクセスするので。
#include <WSCvtoggle.h> //WSCvtoggle クラスに直接アクセスするので。
...

void cbop(WSCbase* object){
  //WSCvbtn クラスを継承しているクラスのオブジェクトの判別
  WSCvbtn* btn = (WSCvbtn*)object->cast("WSCvbtn");

  //WSCvtoggle クラスを継承しているクラスのオブジェクトの判別
  WSCvtoggle* toggle = (WSCvtoggle*)object->cast("WSCvtoggle");

  if (btn == NULL){
    //WSCvbtn クラスと全く関係のないクラスのオブジェクトである!
  }else{
    //WSCvbtn クラスであるかまたは、WSCvbtn クラスを継承している!
  }
  if (toggle == NULL){
    //WSCvtoggle クラスと全く関係のないクラスのオブジェクトである!
  }else{
    //WSCvtoggle クラスであるかまたは、WSCvtoggle クラスを継承している!
  }


}
```

異なるクラスの複数のオブジェクト/クラスにまたがって利用するイベントプロシージャにおいて、クラスを判別するのに便利です。

1.4 特定のオブジェクトクラスのメソッドを実行するには

特定のクラスのメソッドにアクセスする場合、そのクラスのポインタでなければなりません。従ってなんらかの形で、そのクラス型のポインタを取得していなければなりません。［特定のオブジェクトクラスのポインタを取得するには］の節でクラスポインタ取得を行って下さい。

ここではWSClist クラスの addItem() メソッドのアクセスの例を挙げます。WSClist クラスは、文字列を一覧表示します。

```
#include <WSClist.h> //WSClist クラスに直接アクセスするので。
  ...

  void some_function(...){
    //WSClist クラスの list001 インスタンスの取得
    WSCbase* object;
    object = (WSCbase*)WSGIappObjectList()->getInstance("WSClist","list001");

    //WSClist クラスのポインタの取得と addItem API の実行
    WSClist* list = (WSClist*)object->cast("WSClist");
    list->addItem("サンプル文字列",0)
```

まずオブジェクト管理から WSClist クラスの list001 の名称を持つオブジェクトを取得しています。WSCbase ポインタで返値されるので、それを WSClist ポインタにしてから、addItem() メンバ関数にアクセスしています。また、ソースの先頭には、［クラス名］.h (この場合は WSClist.h) をインクルードしておきます。

1.5 親オブジェクトを取得するには

1.5.1 親オブジェクトを取得するには

親オブジェクトを取得するには、メンバ関数 getParent() を利用します。

```
void event_procedure(WSCbase* object){
  //親オブジェクトを取得
  WSCbase* parent = object->getParent();
  //親オブジェクトにアクセス
  parent->setVisible(False);
}
```

object が配置されているの親オブジェクトを取得して、表示属性を不可にしているところです。

1.5.2 親アプリケーションウィンドウを取得するには

親アプリケーションウィンドウを取得するには、メンバ関数 getParentWindow() を利用します。

```
void event_procedure(WSCbase* object){
  //親アプリケーションウィンドウを取得
  WSCbase* win = object->getParent();
  //親アプリケーションウィンドウにアクセス
  win->setVisible(False);
}
```

object が配置されているの親アプリケーションウィンドウを取得して、表示属性を不可にしているところです。

1.6 オブジェクトに配置されている子オブジェクトを取得するには

子オブジェクトを取得する関数には、次のようなメソッドを利用します。

子取得メンバ関数	機能
WSCbase* getChildInstance(char*)	オブジェクト名称指定による取得
WSClistData getChildren()	配置される子オブジェクトの取得
long getAllChildren(WSClistData &)	配置される全ての子オブジェクトの取得

1.6.1 特定の子オブジェクトを取得するには

特定の子オブジェクトを取得するには、メンバ関数 getChildInstance() を利用します。

```
void event_procedure(WSCbase* object){
  //特定の子オブジェクトを名称を指定して取得
  WSCbase* child = object->getChildInstance("newpbtn001");
  if (child != NULL){
    //特定の子オブジェクトにアクセス
    child->setVisible(True);
  }
}
```

object が配置しているの子オブジェクトを名称指定で取得しているところです。名称指定で子オブジェクトを取得する場合、再帰的 (子オブジェクトに配置されている子オブジェクトまで) に検索します。もし指定した名称のオブジェクトが見つからなかった場合、NULL が返されます。

1.6.2 子オブジェクトを取得するには

オブジェクトに配置されている子オブジェクトを取得するには、メンバ関数 getChildren() を利用します。

```
void event_procedure(WSCbase* object){
  //子オブジェクトリストの取得
  WSClistData children = object->getChildren();
```

```
  //子オブジェクトの個数
  long num = children.getNum();
  long i;
  for(i=0; i<num; i++){
    //子オブジェクトの取得
    WSCbase* child = (WSCbase*)children[i];
//  WSCbase* child = (WSCbase*)children.getData(i); //でも良い
    //子オブジェクトにアクセス
    child->setVisible(False);
  }
}
```

object に配置されているの子オブジェクトを取得しているところです。子オブジェクトは WSClistData(リストデータオブジェクト) に格納されて返値されます。個々のオブジェクトは、そのリストデータから、メンバ関数 getData(i) か、配列演算子 [i] で取得し、WSCbase* にキャストして取得して下さい。

1.6.3 全ての子オブジェクトを取得するには

存在する子オブジェクト全てを取得するには、メンバ関数 getAllChildren() を利用します。

```
void event_procedure(WSCbase* object){
  //子オブジェクトを格納するリストデータの宣言
  WSClistData children;
  //子オブジェクトを取得
  object->getAllChildren(children);
  //子オブジェクトの個数
  long num = children.getNum();
  long i;
  for(i=0; i<num; i++){
    //子オブジェクトの取得
    WSCbase* child = (WSCbase*)children[i];
    //子オブジェクトにアクセス
    child->setVisible(True);
  }
}
```

object に配置されているの全ての子オブジェクトを取得しているところです。getChildren() と異なるところは、再帰的に全ての子オブジェクトを取得するところです。

1.6.4 親アプリケーションウィンドウに配置された子オブジェクトを取得するには

存在する子オブジェクト全てを取得するには、メンバ関数 getAllChildren() を利用しますが、それを、親アプリケーションウィンドウで行います。

```
void cbop(WSCbase* object){
  //親アプリケーションウィンドウの取得
  WSCbase* win = object->getParentWindow();
  //子オブジェクトを格納するリストデータの宣言
  WSClistData children;
  //親アプリケーションウィンドウに配置された子オブジェクトを取得
  win->getAllChildren(children);
  //子オブジェクトの個数
  long num = children.getNum();
  long i;
  for(i=0; i<num; i++){
    //子オブジェクトの取得
    WSCbase* child = (WSCbase*)children[i];
    //子オブジェクトにアクセス
    child->setVisible(False);
  }
}
```

親アプリケーションウィンドウに配置されているの全ての子オブジェクトを取得しているところです。

1.7 オブジェクトに貼られているイベントプロシージャを実行するには

オブジェクトに設定されているイベントプロシージャを実行するには、メンバ関数 execProcedure() を利用します。

EP 実行メンバ関数	機能
execProcedure(char*)	イベントプロシージャ名称指定による実行
execProcedure(long)	トリガ指定による実行

1.7.1 オブジェクトに貼られているイベントプロシージャを名称指定で実行するには

オブジェクトに設定されているイベントプロシージャを名称を指定して実行するには、メンバ関数 execProcedure(char*) を利用します。

```
void event_procedure(WSCbase* object){
```

```
  //"設定動作"という名称のイベントプロシージャを実行
  object->execProcedure("設定動作");
}
```

object に設定されているイベントプロシージャで指定された名称のものが存在した場合、そのイベントプロシージャが実行されます。存在しなかった場合は何もしません。

1.7.2 オブジェクトに貼られているイベントプロシージャをトリガ指定で実行するには

オブジェクトに設定されているイベントプロシージャをトリガ指定で実行するには、メンバ関数 execProcedure(long) を利用します。

```
void event_procedure(WSCbase* object){
  // WSEV_ACTIVATE トリガで設定されているイベントプロシージャを実行
  object->execProcedure(WSEV_ACTIVATE);
}
```

object に設定されているイベントプロシージャで指定されたトリガで設定されたものが存在した場合、そのイベントプロシージャが実行されます。存在しなかった場合は何もしません。

1.8 オブジェクトの描画制御を行うには

オブジェクトの描画制御を行うには、次のようなメンバ関数を利用します。

描画制御メンバ関数	機能
setAbsoluteDraw(Boolean)	強制描画フラグを立てる
draw()	通常描画
redraw()	一度クリアして描画
cdraw()	子も含めて再描画
clear()	クリア
update()	プロパティ変更後であれば redraw() する

1.8.1 オブジェクトの描画更新を行うには

通常のプロパティ更新は、イベントプロシージャ実行直後に更新されますが、ユーザが強制的に描画更新を行うこともできます。

```
void event_procedure(WSCbase* object){
  //プロパティの変更
  object->setProperty(WSNlabelString,"設定動作");
  //更新
  object->update();
}
```

update 関数は、プロパティが変更されて更新が必要な場合のみ更新処理を行います。

1.8.2 オブジェクトを再描画させるには

オブジェクトに再描画させるには、次のようなケースがあります。それぞれのケースにおいて再描画方法を示します。

- クリア描画 (イベント発生あり)

  通常の再描画ならば、一度クリアして描画させる redraw() を使います。redraw() はクリアして、EXPOSE イベントを発生させます。もし、他のオブジェクトで重なっているものがあれば、そのオブジェクトも再描画されます。redraw() はイベントを発生させる関係上、多量のオブジェクトが存在するアプリケーションウィンドウでは、パフォーマンスが悪い場合があります。

```
object->redraw();
```

- 描画 (上書き)

  EXPOSE イベントを発生させずに強制的に描画 (上書き) させる場合は、強制描画フラグをたてて draw() を使います。draw() は、再描画の必要のない場合、実行されないので、強制的に描画を行いたい場合は次のように行います。

```
object->setAbsoluteDraw(True);
object->draw();
```

- 単純描画 (上書き)

  EXPOSE イベントを発生させずに単に描画させる場合は、draw() を使います。強制描画フラグが立っていない場合は、EXPOSE イベントが発生した EXPOSE 領域のみ描画します。draw() は、再描画の必要のない場合、実行されないのでパフォーマンスの向上が図れます。

```
object->draw();
```

- クリア描画 (イベント発生なし)

  EXPOSE イベントを発生させずに一度クリアして描画させる場合は、clear() と draw() を使います。もし、他のオブジェクトで重なっているものがあれば、重なっている部分は欠けてしまうので、そのような場合は、クリア描画 (イベント発生あり) を行ってください。

```
object->clear();
object->setAbsoluteDraw(True);
object->draw();
```

1.9 オブジェクトの表示位置を移動するには

オブジェクトの表示位置を移動するには、次の手順に従って、一度消去を行ってください。

```
void event_procedure(WSCbase* object){
  //消去
  object->clear();
  //非表示
  object->setVisible(False);
  //表示位置の移動
  object->setProperty(WSNx,100);
  object->setProperty(WSNy,100);
  //表示
  object->setVisible(True);
}
```

ウィンドウ資源を持たないオブジェクトの座標をそのまま動かすと、前の残像が残ってしまい、一見オブジェクトが二つ有るように見えてしまうことがあります。そのような現象を防ぐために、一度 clear() 関数を呼び出して、非表示状態にしてから、座標を変更して下さい。
なお、ウィンドウ資源を持つのもは、自動的にクリアされます。

1.10 オブジェクトの生成/破棄を行うには

1.10.1 オブジェクトの生成を行うには

オブジェクトを新しく生成するには、メンバ関数 getNewInstance() を利用します。

```
  char*     class_name = "WSCvlabel";
  char*     obj_name   = "vlabel001";
  WSCbase*  parent  //新しいオブジェクトを配置させたい親オブジェクト

  //新オブジェクトを取得
  WSCbase* object = WSCbase::getNewInstance(class_name,parent,obj_name);
  object->initialize(); //とにかく生成後は initialize() を実行。
  object->clear();

  object->setProperty(WSNx,100);
  object->setProperty(WSNy,100);
  object->setProperty(WSNwidth,100);
  object->setProperty(WSNheight,100);
  object->setVisible(True);                    //設定が終了したので表示
```

getNewInstance() で新しいオブジェクトの取得後、他のどのメンバ関数よりも前に initialize() を呼び出して、初期化してください。

1.10.2 オブジェクトの破棄を行うには

オブジェクトを破棄するには、WSGFdestroyWindow 関数を利用します。

```
//破棄
WSGFdestroyWindow(object); //破棄したいオブジェクト
```

object には破棄したいオブジェクトを指定します。二度破棄したり、使用中のオブジェクトを破棄したりすると、修復不可能なメモリエラーとなるので注意しましょう。破棄した後は、そのオブジェクトにアクセス出来ませんので、ポインタなどでそのオブジェクトを覚えている場合など、アクセスすることのないよう注意してください。

1.11 タイマを利用するには

タイマオブジェクトを利用すると、一定期間後の処理や、一定間隔のインターバルでの処理を行うことができます。

タイマクラス	インスタンスアクセス関数
WSDtimer	WSGIappTimer()

1.12 一定期間後の処理を行うには

一定期間後の処理を行うには、まず、タイマから起動され処理を行う関数を用意します。次に、タイマオブジェクトに対して、トリガ起動としてその実行関数を登録します。

タイマ登録メンバ関数	機能
addTriggerProc()	トリガ起動登録関数
delTriggerProc()	トリガ起動削除関数

```
#include <WSDtimer.h>
//タイマから起動される処理関数のサンプル
void triggerHandler(unsigned char clock,void* data){
   //clock は 250ms 毎にカウントアップされるカウンタ
   //data は、タイマ登録する時に渡されたデータ

   //ここでタイマ処理を行う...
}

void event_procedure(WSCbase* obj){
  //ここで指定されたデータが、タイマ処理関数に引き渡される。
  void*  data = (void*)1234;
  //一定期間後に１回起動するタイマ関数の登録  //1000ms 後に起動
  long id = WSGIappTimer()->addTriggerProc( triggerHandler,WS1000MS,data );
  //タイマを途中で止める場合は
  WSGIappTimer()->delTriggerProc( id );
}
```

triggerHandler() 関数には、タイマから起動されて実行したい処理を記述します。triggerHahdler() 関数に、データを与えたい場合、addTriggerProc() の第 3 引数に void* で渡します。addTriggerProc() の返り値は、タイマ処理の ID が返ってきます。もし、そのタイマ処理を止めたい場合は、delTriggerProc() でその ID を指定します。

また、イベントプロシージャの場合とは異なり、タイマ処理関数の場合はオブジェクトの更新処理が呼ばれないので、オブジェクトのプロパティ値などを変更する場合は、update() を呼んで描画更新させてください。

```
//タイマから起動される処理関数のサンプル
void timerHandler(unsigned char clock,void* data){
   WSCbase* object = (WSCbase*)data;
   object->setProperty(WSNlabelString,"時間です");
   object->update(); //オブジェクトの描画更新
}
```

1.13 一定期間毎に処理を行うには

一定期間毎に処理を行うには、まずタイマから起動され処理を行う関数を用意します。次に、タイマオブジェクトに対して、タイマ起動としてその実行関数を登録します。

タイマ登録メンバ関数	機能
addTimerProc()	タイマ起動登録関数
delTimerProc()	タイマ起動削除関数

```
#include <WSDtimer.h>
//タイマから起動される処理関数のサンプル
void timerHandler(unsigned char clock,void* data){
   //clock は 250ms 毎にカウントアップされるカウンタ
   //data は、タイマ登録する時に渡されたデータ

   //ここでタイマ処理を行う...
}

void event_procedure(WSCbase* obj){
  //ここで指定されたデータが、タイマ処理関数に引き渡される。
  void*  data = (void*)1234;
  //一定期間毎に起動するタイマ関数の登録 //500ms 毎に起動
  long id = WSGIappTimer()->addTimerProc( timerHandler,WS500MS,data );
  //タイマを途中で止める場合は
  WSGIappTimer()->delTimerProc( id );
}
```

timerHandler() 関数には、タイマから起動されて実行したい処理を記述します。timerHahdler() 関数に、データを与えたい場合、addTimerProc() の第 3 引数に void* で渡します。addTimer-

Proc() の返り値は、タイマ処理の ID が返ってきます。もし、そのタイマ処理を止めたい場合は、delTimerProc() でその ID を指定します。

また、イベントプロシージャの場合とは異なり、タイマ処理関数の場合はオブジェクトの更新処理が呼ばれないので、オブジェクトのプロパティ値などを変更する場合は、update() を呼んで描画更新させてください。なお、指定できるタイマの間隔は、WS250MS、WS500MS、WS750MS、WS1000MS、WS1250MS、... などの 250ms 間隔の値です。

```
//タイマから起動される処理関数のサンプル
void timerHandler(unsigned char clock,void* data){
   WSCbase* object = (WSCbase*)data;
   object->setProperty(WSNlabelString,"時間です");
   object->update(); //オブジェクトの描画更新
}
```

1.14 グローバルキーフックを利用するには

アプリケーションに入力される全てのキーボードイベントを事前にチェックしたい場合、グローバルキーフックを利用します。グローバルキーフックは、オブジェクトにキーイベントが配られる前に、横取りして調べることができます。グローバルキーフックは WSDkeyboard クラスのグローバルインスタンスに設定します。

キーボードクラス	インスタンスアクセス関数
WSDkeyboard	WSGIappKeyboard()

```
#include <WSDkeyboard.h>
//グローバルキーフックルーチンのサンプル
WSCbool keyhandler(long keycode,WSCbool onoff){
  // keycode : キーコードが渡される
  // onoff   : キーが押されたとき=True, 放されたとき=False
  if (keycode == WSK_F1){
    キーが F1 キーだったら特定処理 ....
    //もしフックしたキーを捨てるならば
    return False; //復帰値=False はキーイベントを捨てる（オブジェクトに配らない）
  }else if (keycode == WSK_F2){
    キーが F2 キーだったら別の特定処理 ....
    //もしフックしたキーを捨てずにオブジェクトに配るならば
    return True; //復帰値=True はキーイベントをオブジェクトに配る...
  }
  return True //復帰値=True はキーイベントをオブジェクトに配る...
}
void event_procedure(WSCbase* obj){
  //グローバルキーフックルーチンの登録
  WSGIappKeyboard()->setGlobalKeyHook( keyhandler );
}
```

keyhandler() 関数は、ユーザがキーボードイベントを横取りして、特別に処理をするために用意するグローバルキーフックルーチンです。WSGIappKeyboard() の setGlobalKeyHook 関数を実行して、キーボードオブジェクトに登録します。キーフックルーチンの登録は、通常、初期化トリガで貼られたイベントプロシージャで行います。キーシンボルの定義は、WSkeysym.h に存在しますので、そちらを御参照下さい。

1.15 インプットフィールドでキー入力を選別するには

インプットフィールドでキー入力を選別するには、キーフックトリガでイベントプロシージャを作成します。

```
//キーフックのイベントプロシージャのサンプル
//WSEV_KEY_HOOK トリガで設定します。
#include <WSDkeyboard.h>
...

void hookop(WSCbase* object){
  //何か処理を記述して下さい
  long key = WSGIappKeyboard()->getKey();
  if ( (key >= WSK_0    && key <= WSK_9   ) ||
       (key >= WSK_KP_0 && key <= WSK_KP_9) ||
        key == WSK_plus      || key == WSK_minus  ||
        key == WSK_BackSpace || key == WSK_Delete || key == WSK_Insert ||
        key == WSK_space     || key == WSK_Up     || key == WSK_Down   ||
        key == WSK_Left      || key == WSK_Right  || key == WSK_Return ){
    //数字キーは処理を通す
    return;
  }
  //それ以外は捨てる
  WSGIappKeyboard()->setKey(0);
}
```

このサンプルは、数字入力のみを行いたいインプットフィールドを実現します。キーボードオブジェクトに対して、getKey() 関数により現在入力されつつあるキーを選別して、いらないキーの場合、 setKey(0) を実行して、入力キーコードを塗りつぶし、入力がなかったように装います。

1.16 オブジェクトに新たなイベントプロシージャを設定するには

オブジェクトに新たなイベントプロシージャを設定するには、メンバ関数 addProcedure() を利用します。

イベントプロシージャ追加メンバ関数	機能
addProcedure(WSCprocedure*)	新たなイベントプロシージャの追加

イベントプロシージャオブジェクトを次の様に生成します。(A) で、イベントプロシージャ名称、トリガを指定してイベントプロシージャオブジェクトを生成します。次に (B) で、実行関数とその関数名称を指定します。setFunction() で指定します。(C) で作成したイベントプロシージャオブジェクトを GUI オブジェクトに設定します。

```
void _new_event_procedure(WSCbase*){
//新規追加されるイベントプロシージャ関数
}

void event_procedure(WSCbase* object){
  //"新しい EP"という EP 名称のイベントプロシージャの作成
  //トリガ WSEV_MOUSE_IN (MOUSE_IN トリガ)
  WSCprocedure* ep = new WSCprocedure("新しい EP",WSEV_MOUSE_IN); //(A)
  ep->setFunction(_new_event_procedure,"_new_event_procedure");    //(B)
  object->addProcedure(ep);                                        //(C)
}
```

1.17 配列化したオブジェクトにアクセスするには

配列化したオブジェクトにアクセスすることができます。オブジェクトを配列化するには、アプリケーションビルダーユーザーズガイドの［オブジェクトを配列として定義するには］を参照ください。配列化したオブジェクトには次の様にアクセスします。

```
#include <WSCvlabel.h>

//アクセスする配列の extern 宣言
extern WSCvlabel** labelarray;

void event_procedure(WSCbase* object){
  labelarray[0]->setProperty(WSNlabelString,"Label No. 0");
  labelarray[1]->setProperty(WSNlabelString,"Label No. 1");
   ...
}
```

1.18 EXIT トリガによる終了イベントプロシージャでダイアログを表示するには

ウィンドウを閉じてアプリケーションを終了する場合、データの保存を行ったり、終了するか否かをダイアログを表示したい場合があります。そのような場合、WSCwindow / WSCmainWindow クラスの EXIT トリガでイベントプロシージャを使うと便利です。WSCwindow/WSCmainWindow クラスは、ウィンドウが不可視状態になった場合に、EXIT トリガをあげ、アプリケーションが終了する前に、イベントプロシージャを実行する機能を持っています。

まず、WSCwindow もしくは WSCmainWindow クラスのプロパティ WSNexit を True にします。このプロパティは、アプリケーション中のウィンドウで特にメインで用いられるものに設定すると良いでしょう。そしてそのウィンドウに対して、EXIT トリガでイベントプロシージャを張り付けます。

次のような機能を持つイベントプロシージャを作ってみます。

- 終了するか否かのダイアログを表示。
- 「OK」が選択された場合は、処理を行って終了。
- 「NO」が選択された場合は、処理を行わず終了。
- 「CANCEL」が選択された場合は、処理を行わず終了もしない。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-----------------------------------------------------------
//Function for the event procedure
//-----------------------------------------------------------
#include <WSCmessageDialog.h>
#include <WSDtimer.h>

//オブジェクトを表示しなおすタイマープロシージャ
void delayproc(unsigned char,void* ptr){
  WSCbase* object = (WSCbase*)ptr;
  object->setVisible(True);
}
//EXIT イベントプロシージャ本体
//終了時にダイアログを表示する。
void exit_ep(WSCbase* object){
  if (object->getVisible() != False){
    return;
  }
  WSCmessageDialog* msg = WSGIappMessageDialog(); //A
  msg->setProperty(WSNwidth,500);
  msg->setProperty(WSNno,True);
  msg->setProperty(WSNdefaultPosition,True);
  msg->setProperty(WSNlabelString,
    "Exit and save data?\n If you do not want to save and exit,push NO...");
  //ダイアログの表示
  long ret = msg->popup(); //B

  if (ret == WS_DIALOG_OK){ //OK ボタンがおされた場合 C
    //saving some data ...
```

```
    exit(0);
  }else
  if (ret == WS_DIALOG_NO){ //NO ボタンがおされた場合 D
    exit(0);
  }else
  if (ret == WS_DIALOG_CANCEL){ //CANCEL ボタンが押された場合 E
    WSGIappTimer()->addTriggerProc(delayproc,WS250MS,object);
  }
}
static WSCfunctionRegister  op("exit_ep",(void*)exit_ep);
```

A で、メッセージダイアログインスタンスを取得し、B でメッセージダイアログを表示します。

C、D、E でダイアログの結果を判定し、OK ボタンがおされたならば C、NO ボタンがおされたならば D、CANCEL ボタンがおされたならば E となります。

E の終了せずに再び、表示しなおす場合、タイマーを使って、少しタイミングを送らせる必要があります。これは、ウィンドウシステムに対し、既に画面終了イベントが発生しているため、そのイベントの処理が確実に処理されてから、表示を行わなければならないからです。

［終了確認ダイアログ］

1.19 マウスのボタンを判定するには

イベントプロシージャ等で、押されているマウスのボタンを判定したい場合があります。下記の例のようにグローバルマウスインスタンスからの情報取得によって、現在押されているマウスのボタンが判定することガできます。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSDmouse.h> //A

void btn_ep(WSCbase* object){
  long status = WSGIappMouse()->getStatus(); //B
  if (status & WS_MOUSE_BTN1){ //C
    //左マウスボタンが押されている場合の処理
```

```
  }
  if (status & WS_MOUSE_BTN2){ //D
    //中マウスボタンが押されている場合の処理
  }
  if (status & WS_MOUSE_BTN3){ //E
    //右マウスボタンが押されている場合の処理
  }

}
static WSCfunctionRegister  op("exit_ep",(void*)exit_ep);
```

まず、A で、グローバルマウスインスタンスにアクセスするため、WSDmouse.h を include しておきます。B で、マウスの押下状態を取得します。C,D,E で、それぞれ、調べたいボタンの判定を行います。== (比較) ではなく、& (アンド) で判定するのは、複数のボタンが同時に押されている場合があるからです。

第2章　オブジェクト編

2.1　サンプルイベントプロシージャ・ラベル編

2.1.1　マウスで選択可能なラベルにするには

イベントプロシージャにおいて、最も基本的なマウスの動作に反応するイベントプロシージャのサンプルです。ラベル上でマウスがクリックされるたびに数字がカウントアップされるイベントプロシージャを作成してみましょう。

```
#include <WSDmouse.h>
//WSEV_MOUSE_PRESS(MOUSE-PRESS) トリガ、でラベルに設定します。
void cbop(WSCbase* object){

  //(0) マウスボタン１以外はリターン
  if ( (WSGIappMouse()->getMouseStatus() & WS_MOUSE_BTN1) == 0){
    return;
  }

  //(A)WSNuserValue プロパティに覚えている数字を取得
  long value = object->getProperty(WSNuserValue);
  //(B) その数字をカウントアップ
  value++;
  //(C) その数字の表示
  object->setProperty(WSNlabelString,value);
  //(D)WSNuserValue プロパティにカウントアップした数字を覚えておく
  object->setProperty(WSNuserValue,value);
}
```

まず、このイベントプロシージャは、WSCvbtn,WSCvlabel など、WSNlabelString プロパティを持つオブジェクトに、MOUSE-PRESS トリガで設定します。マウスボタンがオブジェクト上でマウスがクリックされるたびに、この関数は起動されます。

イベントプロシージャが起動されると、(0) で、マウスボタンの判定をおこない、マウスボタン１以外のボタンだったらリターンします。マウスボタンの判定の参考にしてください。

次に (A) で WSNuserValue プロパティに覚えている数字を取得します。WSNuserValue プロパティは初期値は０で、ユーザが値を自由に指定して、保持することができます。ここで、オブジェクトに覚えさせているのは、多数のオブジェクトに貼られたとき、それぞれのラベルの数字が交じらない様にするためです。逆にスタティック変数などに保持するように作成すると、すべてのラベルで一つのカウンタを利用した表示が行えます。

次に (B) でその数字をカウントアップし、(C) で表示します。(D) で表示した数字を WSNuserValue プロパティに保持し、次のイベントプロシージャ起動に備えます。もし、トリガ、MOUSE-IN で張られた場合は、マウスの出入りした回数を表示するでしょう。

2.1.2 マウスで選択可能なラベルにするには

次に、選択可能なラベルを作成してみましょう。ラベルが選択されている様に見せるために、色を変化させてみましょう。今度は、WSNuserValue プロパティの変わりに set/getUserData() 関数を用いて選択状態を保持してみます。

```
//WSEV_MOUSE_PRESS(MOUSE-PRESS) トリガでラベルに設定します。
void cbop(WSCbase* object){
  //(A) getUserData() で覚えている値を取得
  long value =  (long)object->getUserData("STATUS");
  //(B) value が０だったら選択状態に、１だったら元の表示状態に戻す
  if (value == 0){
    //(C) 元の色名称（文字列）を WSNuserString で覚えておく
    WSCvariant color = object->getProperty(WSNbackColor);
    object->setProperty(WSNuserString,color);
    //(D) 選択色の表示状態にする
    object->setProperty(WSNbackColor,"slategray4");
    //(E) 状態を覚えておく
    value = 1;
    object->setUserData("STATUS",(void*)value);
  }else{
    //(F) 元の色を取り出す
    WSCvariant color =  object->getProperty(WSNuserString);
    //(G) 元の色の表示状態にする
    object->setProperty(WSNbackColor,color);
    //(H) 状態を覚えておく
    value = 0;
    object->setUserData("STATUS",(void*)value);
  }
}
```

このイベントプロシージャは、WSCvbtn,WSCvlabel など、WSNlabelString プロパティを持つオブジェクトに、MOUSE-PRESS トリガで設定します。

イベントプロシージャが起動されると、(A) で setUserData() 関数で覚えている数字を取得します。set/getUserData() 関数は初期値は０で、ユーザが値を自由に指定して、保持することができます。好きな名称を割り当てて、任意の値を void* で保持しておくことが出来ます。

次に (B) でその値を判別し、 (C) で元の色を、次回、表示を元に戻すためにプロパティ WSNuserString で保持して、覚えておきます。次に (D) で選択色での表示状態にし、(E) で再び選択状態を保持しなおします。

色を元に戻すフェーズは、(F) で保持しておいた元の表示色を取得して、(G) でその色を反映します。(H) で再び選択状態を保持します。

2.1.3 マウスでハイライトするラベルにするには

マウスのイン・アウトでハイライトするイベントプロシージャを作ってみましょう。オブジェクト上にマウスが入ってくるとハイライトし、マウスが出ていくともとに戻るイベントプロシージャになります。

ここで重要なことは、マウスが入った場合に発生するトリガと、マウスが出た場合に発生するトリガにそれぞれイベントプロシージャを設定するイベントプロシージャを作成するという点にあります。

すなわち初期化でイベントプロシージャが起動され、その中で新たなイベントプロシージャを設定することで、複数のイベントプロシージャを一つのイベントプロシージャの設定で実現が可能となります。

```
//サブ EP1 //マウスイン
void subop1(WSCbase* object){
    //(A) 元の色名称を WSNuserString プロパティに覚えておく
    WSCvariant color = object->getProperty(WSNbackColor);
    object->setProperty(WSNuserString,color);
    //(B) 選択色の表示状態にする
    object->setProperty(WSNbackColor,"slategray4");
}
//サブ EP2 //マウスアウト
void subop2(WSCbase* object){
    //(C) 元の色名称を取り出す
     WSCvariant color = object->getProperty(WSNuserString);
    //(D) 元の色の表示状態にする
    object->setProperty(WSNbackColor,color);
}
//WSEV_INIT(INITIALIZE) トリガでラベルに設定します。
void cbop(WSCbase* object){
   //初期化トリガが発生して実行されると
   //サブ EP を、それぞれ新しく設定します。
   //(E) イベントプロシージャの新規設定　//サブ EP1
   //OP 名称="ハイライト OP1"　　起動トリガ=WSEV_MOUSE_IN　　関数=subop1
   WSCprocedure* ac1 = new WSCprocedure("ハイライト OP1",WSEV_MOUSE_IN);
   ac1->setFunction(subop1,"subop1");
   object->addProcedure(ac1);
   //(F) イベントプロシージャの新規設定　//サブ EP2
   //OP 名称="ハイライト OP2"　　起動トリガ=WSEV_MOUSE_OUT　　関数=subop2
   WSCprocedure* ac2 = new WSCprocedure("ハイライト OP2",WSEV_MOUSE_OUT);
   ac2->setFunction(subop2,"subop2");
```

```
   object->addProcedure(ac2);
}
```

サブ EP1 は、マウスインのトリガで起動し、ラベルの背景表示色をハイライト色にします。サブ EP2 は、マウスアウトのトリガで起動し、ラベルの背景表示色を元の色に戻します。イベントプロシージャ本体は、初期化トリガでオブジェクトに設定されており、(E)、(F) でサブ EP1,2 をオブジェクトに設定しています。以後、マウスがインアウトすると、新しくプログラムの中から設定されたサブ EP が実行されるようになります。

2.1.4 マウスで選択可能なグループ化されたラベルにするには

同じエリア上に配置されるラベルを、マウスで選択可能なグループ化されたラベルにするイベントプロシージャを実装するために、マウスで選択されたラベルを陥没状態にして、そのエリアにどのラベルが選択されているかを保持するようにします。

```
//WSEV_MOUSE_PRESS(MOUSE-PRESS) トリガでラベルに設定します。
void cbop(WSCbase* object){
  //(A)WSNuserValue プロパティに設定されている値を ID として使用する
  long val = object->getProperty(WSNuserValue);
  //(B) 選択された状態 (陥没状態) にする
  object->setProperty(WSNshadowType,WS_SHADOW_IN);
  //(C) 親に覚えている選択されていたオブジェクトを取得する
  WSCbase* parent = object->getParent();
  WSCbase* target = (WSCbase*)parent->getUserData("SelectedItem");
  //(D) その選択されていたオブジェクトの表示状態を元 (突出状態) に戻す
  if (target != NULL){
    target->setProperty(WSNshadowType,WS_SHADOW_OUT);
  }

  if (target == object){
    //(E) 選択しているオブジェクトをもう一度つついた場合
    //二度目の選択の場合は、覚えていた自分を 0 にして忘れる
    parent->setUserData("GroupValue",(void*)0);
    parent->setUserData("SelectedItem",(void*)0);
  }else{
    //(F) 一度目の選択の場合は、自分を親に登録。
    parent->setUserData("GroupValue",(void*)val);      //ID
    parent->setUserData("SelectedItem",(void*)object); //WSCbase* ポインタ
  }
}
```

グループ化するラベル達に、それぞれ識別するための ID が必要になりますが、WSNuserValue プロパティに設定されている値を ID として使用することにします。

(A) で WSNuserValue プロパティに設定されている ID を取得します。

(B) で選択されたことを示すため、陥没枠表示にします。

(C) で選択されていたオブジェクトを取得します。どのオブジェクトでも良いのですが、この例の場合、グループ化するラベル達に共通な親オブジェクトに覚えておきます。

(D) で選択されていたオブジェクトの表示状態をもとの突出状態に戻します。

(E) で選択されていたオブジェクト (target) と、新たに選択されたオブジェクト (object) が等しい場合、すなわち、２度選択された状態になった場合、解除します。親に覚えていた値とオブジェクトをリセットします。

(F) では、新たに選択されたオブジェクトと値を親に覚え直します。

2.2 サンプルイベントプロシージャ・インプットフィールド編

2.2.1 リターンキーで特定のイベントプロシージャを実行するには

イベントプロシージャにおいて、特定のイベントプロシージャを起動することをしてみましょう。ここでは、リターンキーを入力で、”入力確定”なる名称を持つイベントプロシージャを起動する例をあげます。

```
//WSEV_KEY_HOOK トリガで WSCvifield インスタンスに設定します。
#include <WSDkeyboard.h>
void cbop(WSCbase* object){
  //(A) 入力されつつあるキーを取得。
  long key = WSGIappKeyboard()->getKey();
  //(B) キーがリターンキーであれば
  if (key == WSK_Return){
    //"入力確定"なる名称をもつイベントプロシージャを実行。
    object->execProcedure("入力確定");
  }
}
```

(A) でキーボードグローバルインスタンスから、入力されつつあるキーを取得します。

(B) でリターンキーかどうか判別し、もしそうだったら (C) で、”入力確定”なる EP 名称をもつ EP を実行させます。 この入力確定実行イベントプロシージャは、リターンキーによる入力確定動作を行いたい場合によく用いられます。

2.2.2 初期時入力で前回入力文字列をクリアするには

新たらにフォーカスがあたって入力を開始する場合や新たらにマウスでクリックされて入力を開始する場合に前回入力文字列をクリアするイベントプロシージャを作成してみましょう。大きな流れとしては次の様になります。

- (1) フォーカスが新たにあたった場合、クリアフラグを立てます。
- (2) マウスがクリックされた場合、クリアフラグを立てます。

- (3) キー入力された場合、クリアフラグがたっていたら、クリアします。
- (1),(2),(3) のサブイベントプロシージャを張り付けて初期化します。

```
#include <WSDkeyboard.h>

//フォーカスがあたっていたインプットフィールドを保持する変数
static WSCbase* _focus_if = NULL;

//WSEV_FOCUS_CH トリガで起動するサブ EP
static void _focus_ch_(WSCbase* object){
  //(A) フォーカスが移動して来たのかを調査
  if (_focus_if != object && object->getFocus() != False){
    //(B) 他のオブジェクトからフォーカスが移動してきた場合、
    //クリアするタイミング。そのことをフラグに覚えておく
    object->setUserData("CLEAR TIMING",(void*)1);
    //(C) 新たにフォーカスがあたったのは、自分。
    _focus_if = object;
  }
}

//WSEV_MOUSE_PRESS トリガで起動するサブ EP
static void _btn_press_(WSCbase* object){
  //(D) マウスでクリックされたら
  //クリアするタイミング。そのことをフラグに覚えておく
  object->setUserData("CLEAR TIMING",(void*)1);
  object->setProperty(WSNcursorPos,0);
  //(E) 新たにフォーカスがあたったのは、自分。
  _focus_if = object;
}

//WSEV_KEY_PRESS トリガで起動するサブ EP
static void _key_hook_(WSCbase* object){
  //(F) キー入力時に、クリアタイミングかどうか調査
  long fl =(long)object->getUserData("CLEAR TIMING");
  if (fl == 1){
    long key = WSGIappKeyboard()->getKey();
    //(G) クリアタイミングで、移動キーでなければ
    if (key != WSK_Tab &&
        key != WSK_Up &&
        key != WSK_Down &&
        key != WSK_Left &&
        key != WSK_Right ){
      //(H) クリア。
```

```
      object->setProperty(WSNlabelString,"");
    }else{
      return;
    }
  }
  //(I) クリアタイミングフラグを落とす。
  object->setUserData("CLEAR TIMING",(void*)0);
}

//前回入力文字列クリア本体
//WSEV_INITIALIZE トリガでインプットフィールドに設定します。
void ifdclr(WSCbase* object){
  //フォーカスがあたったときに起動するサブ EP を設定
  WSCprocedure* ac1 = new WSCprocedure("ac1",WSEV_FOCUS_CH);
  ac1->setFunction(_focus_ch_,"_focus_ch_");
  object->addProcedure(ac1);

  //マウスがクリックされたときに起動するサブ EP を設定
  WSCprocedure* ac2 = new WSCprocedure("ac2",WSEV_MOUSE_PRESS);
  ac2->setFunction(_btn_press_,"_btn_press_");
  object->addProcedure(ac2);

  //キー入力されたときに起動するサブ EP を設定
  WSCprocedure* ac3 = new WSCprocedure("ac3",WSEV_KEY_HOOK);
  ac3->setFunction(_key_hook_,"_key_hook_");
  object->addProcedure(ac3);
}
```

まずフォーカス関連のサブ EP について説明します。フォーカスが新たにあたったことを知るために、前回フォーカスがあたっていたオブジェクトを静的変数で保持しておきます。

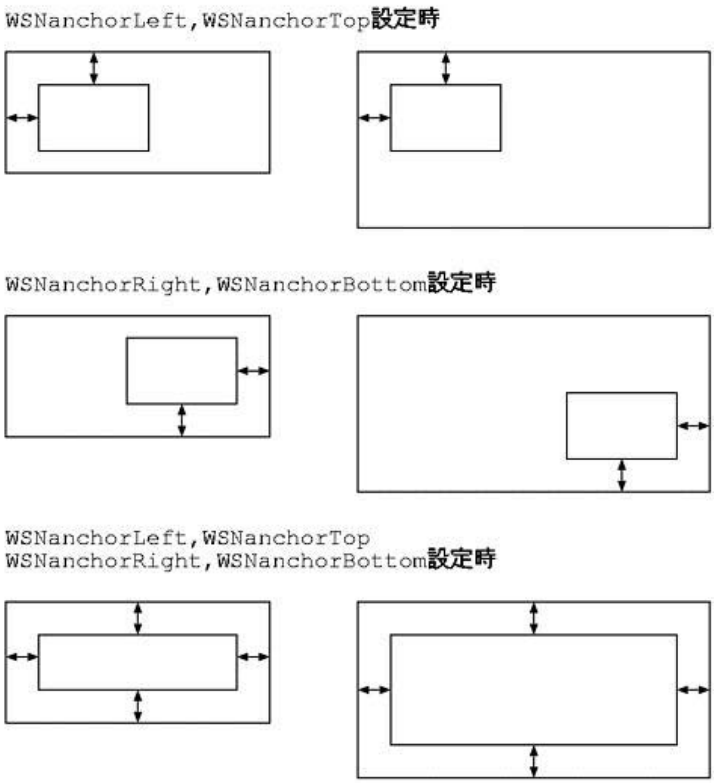
(A) では、ます、保持しておいたオブジェクトと自分が異なるかを調べます。異なれば新たにフォーカスがあたったことを意味し、(B) でクリアフラグを立てます。次に (C) で自分をその静的変数に保持します。

次にマウスのクリック関連のサブＥＰについて説明します。(D) でクリアフラグを立てて入力カーソルを先頭に設定します。次に (E) で自分をフォーカス保持の静的変数に保持します。

次にキー入力関連のサブ EP について説明します。キー入力関連のサブ EP では、クリアフラグがたっている場合、文字列をクリアして、そのクリアフラグを落とします。まず、(F) でクリアフラグが立っているか調べます。(G) では移動キーで、クリアされては悲しいので、そのキーを判別して、(H) でクリアし、(I) でクリアフラグを落とします。

2.3 アンカーによる自動サイズ調整

WideStudio のサイズを持つオブジェクトの多くは、アンカーによる自動サイズ調整の機能を持っています。アンカーは、配置されている自分の親フォームのサイズに追従して一定の距離を保ち、自分のサイズを調整します。

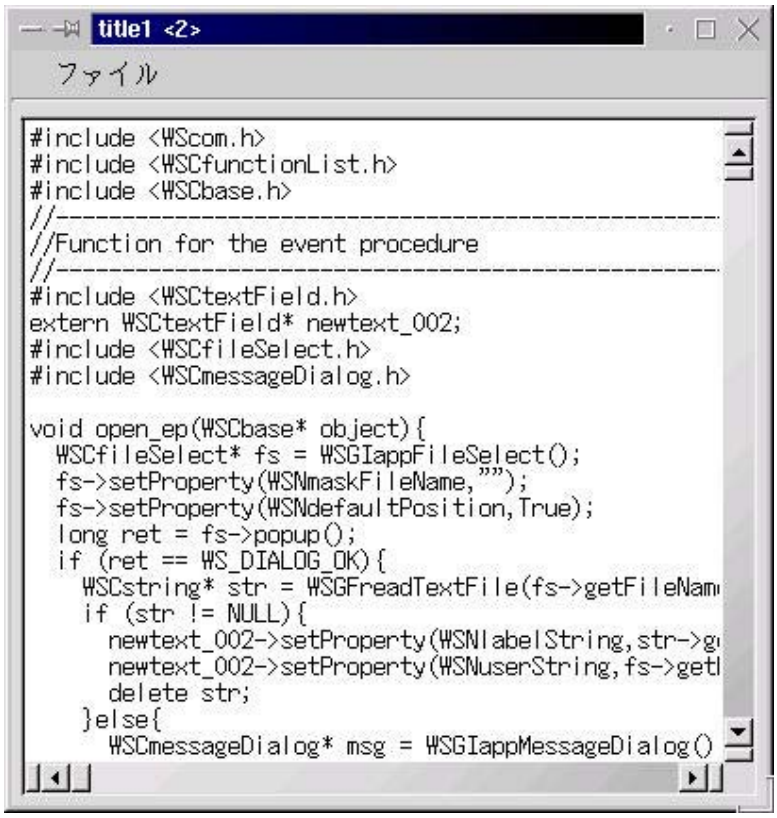


［各アンカー設定時の動作］

2.4 プルダウンメニューとメニューエリア

2.4.1 メニューエリアとは

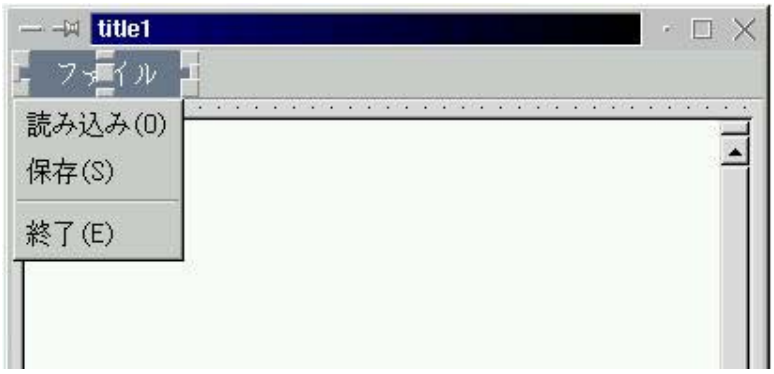
メニューエリア WSCmenuArea クラスは、メニューを配置するためのフォームです。このフォームは、配置されたウィンドウの上部に一定の領域を保ち、メニューを表示するためのスペースを確保します。ウィンドウがサイズ変更されても自動的にサイズ調整されます。



［メニューエリア (WSCmenuArea) を使ったサンプル］

2.4.2 プルダウンメニューを使ってみよう

プルダウンメニューのあるサンプルを一つ作ってみましょう。



［ファイルメニューのあるサンプルアプリケーション］

図のように、「ファイル」メニューがあり、次のようなメニュー項目があるものとします。

- 読み込み
- 保存
- 終了

一つアプリケーションウィンドウを作成し、オブジェクトボックスの Forms からメニューエリア (WSCmenuArea) をドラッグアンドドロップして配置します。次にオブジェクトボックスの Commands から PLD と書いたアイコン (WSCpulldownMenu クラス) を配置したメニューエリア上にドラッグアンドドロップします。プロパティを次の様に設定します。

- 表示文字列： ファイル
- メニュー項目： 読み込み (O):読み込み EP:o, 保存 (S):保存 EP:s,SP, 終了 (E):終了 EP:e

メニュー項目プロパティで、設定する項目は、１項目毎にカンマで区切ります。
1 項目は次のような書式で成り立っています。途中、,SP, をはさむと、セパレータが表示されます。

項目の表示文字列:実行イベントプロシージャ名:ショートカットキー,...

次に、項目を選択された場合に実行するイベントプロシージャです。上記の例では、例えば、「読み込み (O)」が選択された場合、「読み込み EP」という名称のイベントプロシージャが実行されます。実行したいイベントプロシージャをトリガ NONE で「読み込み EP」という名称で張り付けます。
それぞれのメニュー項目に、ID を割り振ることもできます。一つのイベントプロシージャで、選択されたメニュー項目を判別して、処理を行う場合に便利です。
ID を割り振る場合は、次のような書式で指定します。プロシージャからは、getValue() メソッドにより、選択されたメニュー項目の ID を取得できます.

項目の表示文字列:実行イベントプロシージャ名:ショートカットキー:`ID`**,...**

2.4.3 プルダウンメニュー使用時における注意事項

同じフォーム上にプルダウンメニューと他の WSCv 系のオブジェクトの併用はできません。必ず、プルダウンメニューを使用する場合は、プルダウンメニューだけを配置するフォームを用意し、他のクラスのオブジェクトを配置せずにプルダウンメニューを配置してください。

2.5 リスト

2.5.1 メソッドによるリストのデータ表示制御

リストに項目表示データを追加するには、addItem メソッドを用います。addItem により、追加する文字列、追加する位置を指定します。追加位置を省略すると、リストの末尾に追加されます。リストに項目を追加するサンプルです。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
```

```
#include <WSClist.h>
extern WSClist* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストの末尾に項目を追加。
  newlist_001->addItem("item1");
  newlist_001->addItem("item2");
  newlist_001->addItem("item3");
  newlist_001->addItem("item4");
  //リストに位置指定で項目を追加。
  newlist_001->addItem("item0",0);//0 は先頭
  newlist_001->addItem("item5",-1);//-1 は末尾

  //リストを変更した場合、最後に一度、更新。
  newlist_001->updateList();
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.5.2 プロパティからのリストデータの設定

比較的項目数が少ない場合、プロパティから、一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_NONE に指定します。次に、WSNdata プロパティに改行コードで区切ったデータを設定します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSClist.h>
extern WSClist* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにプロパティ経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_NONE);
  newlist_001->setProperty(WSNdata,"item0\nitem1\nitem2\nitem3\nitem4");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

また次のように、アイコンファイルも同時に指定すると、項目毎にアイコンも指定することができます。プロパティ WSNuseIcon を True に設定して、アイコンを指定する場合は、カンマで区

切って指定します。アイコンファイルは省略すると、プロパティ WSCiconPixmap に指定されたものが使用されます。
書式：

アイコンファイル, 項目文字列\n アイコンファイル, 項目文字列\n....

```
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにプロパティ経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_NONE);
  newlist_001->setProperty(WSNdata,
     "$(WSDIR)/sys/pixmaps/bi16.xpm,item1\nitem2\nitem3");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.5.3 ファイルからのリストデータの設定

ファイル名を指定して、ファイルから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_FILE に指定します。次に、WSNdataSourceName プロパティにファイル名を指定します。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-----------------------------------------------------------
//Function for the event procedure
//-----------------------------------------------------------
#include <WSClist.h>
extern WSClist* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにファイル経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_FILE);
  newlist_001->setProperty(WSNdataSourceName,"data.txt");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);

//data.txt の内容
$(WSDIR)/sys/pixmaps/bi16.xpm,item1
item2
item3
```

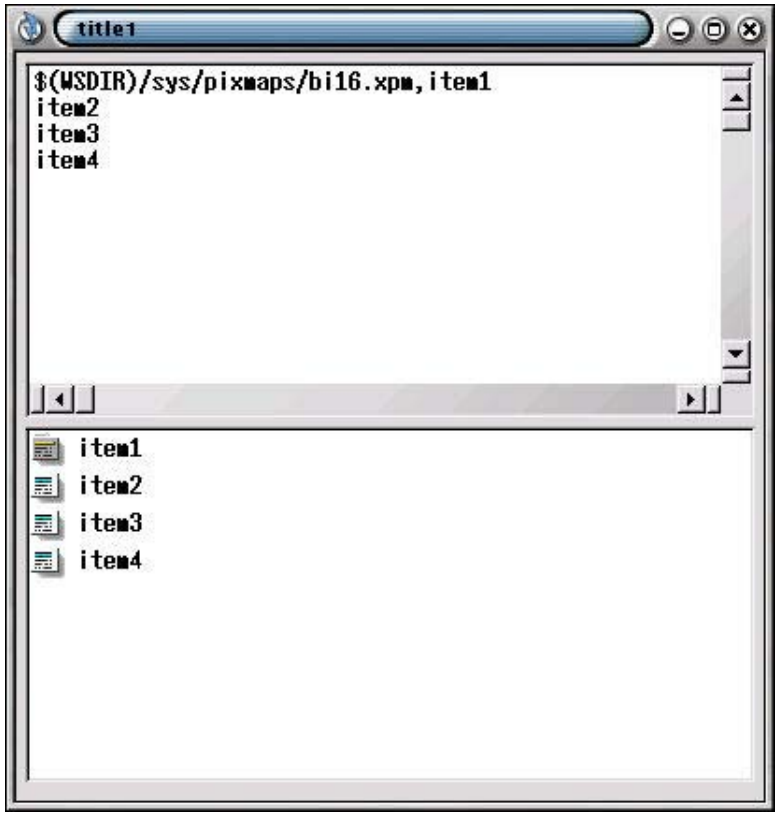
```
item4
$(WSDIR)/sys/pixmaps/bi16.xpm,item5
item6
item7
item8
```

2.5.4 インスタンスからのリストのデータ表示

インスタンス名を指定して、そのインスタンスのデータソース対象プロパティから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_INSTANCE に指定します。次に、WSNdataSourceName プロパティにインスタンス名を指定します。下記の例では、WSCtextField のインスタンス newtext_000 を指定しています。データの形式は、ファイル指定の場合と変わりません。newtext_000 に入力されている文字列がリストに表示されます。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSClist.h>
extern WSClist* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにインスタンス経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_INSTANCE);
  newlist_001->setProperty(WSNdataSourceName,"newtext_000");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

次の図では、上部のテキスト入力から入力した文字列が、下部のリストに表示されています。

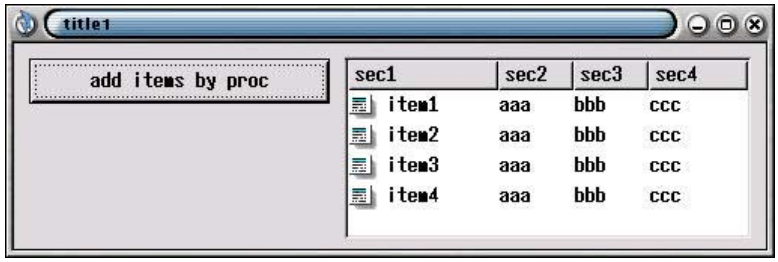


［インスタンスからの項目設定］

2.6 詳細リスト

2.6.1 メソッドによるリストのデータ表示制御

詳細リスト (WSCverbList) または、WSClist のリスト種別を詳細リストに設定したものに項目表示データを追加するには、リストの場合と全く同じように addItem メソッドを用います。addItem により、追加する文字列、追加する位置を指定します。追加位置を省略すると、リストの末尾に追加されます。リストの場合と異なる部分は、各項目のカンマで区切られた項目を設定することです。



［詳細リスト表示の例］

図の様に横 4 セクションのリストとする場合、プロパティ WSNbarValue に、50,100,150 を設定します。このプロパティは、タイトルのセパレータの位置を指定します。このプロパティによって、セクションの数が決まりますので注意してください。プロパティ WSNtitleString には、sec1,sec2,sec3,sec4 を設定します。

詳細リストに項目を追加するサンプルです。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCverbList.h>
extern WSCverbList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストの末尾に項目を追加。
  newlist_001->addItem("item1,aaa,bbb,ccc");
  newlist_001->addItem("item2,aaa,bbb,ccc");
  newlist_001->addItem("item3,aaa,bbb,ccc");
  newlist_001->addItem("item4,aaa,bbb,ccc");

  //リストを変更した場合、最後に一度、更新。
  newlist_001->updateList();
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.6.2 プロパティからの詳細リストデータの設定

比較的項目数が少ない場合、プロパティから、一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_NONE に指定します。次に、WSNdata プロパティに改行コードで区切ったデータを設定します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCverbList.h>
extern WSCverbList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにプロパティ経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_NONE);
  newlist_001->setProeprty(WSNdata,
     "item1,aaa,bbb,ccc\nitem2,aaa,bbb,ccc\nitem3,aaa,bbb,ccc");
```

```
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

また次のように、アイコンファイルも同時に指定すると、項目毎にアイコンも指定することができます。プロパティ WSNuseIcon を True に設定して、アイコンを指定する場合は、カンマで区切って指定します。アイコンファイルは省略すると、プロパティ WSCiconPixmap に指定されたものが使用されます。

書式：

アイコンファイル, **文字列**, **文字列**, **文字列**, . . . \n **アイコンファイル**, **文字列**, **文字列**, . . . \n. . . .

```
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにプロパティ経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_NONE);
  newlist_001->setProperty(WSNdata,
     "$(WSDIR)/sys/pixmaps/bi16.xpm,item1,aaa,bbb,ccc\nitem2,aaa,bbb,ccc\nitem3,aaa,bbb,ccc");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.6.3 ファイルからの詳細リストデータの設定

ファイル名を指定して、ファイルから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_FILE に指定します。次に、WSNdataSourceName プロパティにファイル名を指定します。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCverbList.h>
extern WSCverbList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにファイル経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_FILE);
  newlist_001->setProperty(WSNdataSourceName,"data.txt");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

```
//data.txt の内容
$(WSDIR)/sys/pixmaps/bi16.xpm,item1,aaa,bbb,ccc
item2,aaa,bbb,ccc
item3,aaa,bbb,ccc
item4,aaa,bbb,ccc
$(WSDIR)/sys/pixmaps/bi16.xpm,item5,aaa,bbb,ccc
item6,aaa,bbb,ccc
item7,aaa,bbb,ccc
item8,aaa,bbb,ccc
```

2.6.4 インスタンスからの詳細リストのデータ表示

インスタンス名を指定して、インスタンスから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_INSTANCE に指定します。次に、WSNdataSourceName プロパティにインスタンス名を指定します。下記の例では、WSCtextField のインスタンス newtext_000 を指定しています。データの形式は、ファイル指定の場合と変わりません。newtext_000 に入力されている文字列がリストに表示されます。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。この場合、入力フィールドのインスタンス newtext_000 に入力されている文字列が詳細リストの項目として表示されます。

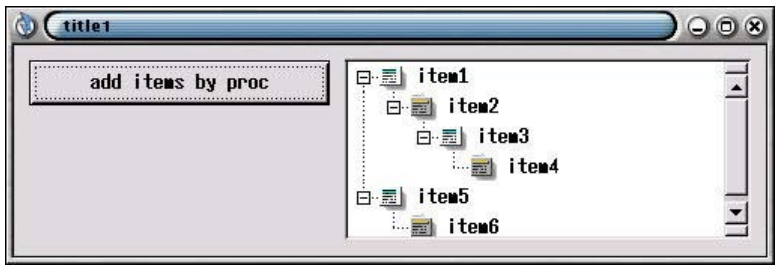
```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCverbList.h>
extern WSCverbList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにファイル経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_INSTANCE);
  newlist_001->setProperty(WSNdataSourceName,"newtext_000");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.7 ツリーリスト

2.7.1 メソッドによるリストのデータ表示制御

ツリーリスト (WSCtreeList) または、WSClist のリスト種別をツリーリストに設定したものに項目表示データを追加するには、リストの場合と全く同じように addItem メソッドを用います。addItem により、追加する文字列、追加する位置を指定します。追加位置を省略すると、リストの末尾に追加されます。また、リスト項目のネスト階層は、setItemValue メソッドで設定することができます。引数には、設定したい項目の位置、値種別に WS_INDENT_LEVEL、そして階層を指定します。何も設定しない場合は第 0 階層となり、最上位の状態になります。

```
setItemValue(pos,WS_INDENT_LEVEL,level);
pos = 0,1,2,...,-1(最後尾)
level = 0(top),1,2,3...
```



［ツリーリスト］

図の様にツリーリストに項目を追加するサンプルです。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCtreeList.h>
extern WSCtreeList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストの末尾に項目を追加。
  newlist_001->addItem("item1");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,0);
  newlist_001->addItem("item2");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,1);
  newlist_001->addItem("item3");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,2);
  newlist_001->addItem("item4");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,3);
```

```
  newlist_001->addItem("item5");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,0);
  newlist_001->addItem("item6");
  newlist_001->setItemValue(-1,WS_INDENT_LEVEL,1);

  //リストを変更した場合、最後に一度、更新。
  newlist_001->updateList();
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

ツリーリストにおいて、注意すべき点は、項目間には親子関係のような特別な関係が全く無く、単なるリスト表示に対して、各項目がインデントを指定された状態で表示されると言うことです。したがって、最上位の項目を削除したからといって、それ以降の階層の項目が無くなるといったことはありません。また、前項目とのインデントの差は、+1 までです。+1 以上差がある場合は、自動的に +1 となるように補正されますのでご注意ください。

2.7.2 プロパティからのツリーリストデータの設定

比較的項目数が少ない場合、プロパティから、一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_NONE に指定します。次に、WSNdata プロパティに下記に示すような書式で設定します。

データ書式：(プロパティ WSNuseIcon が True の場合)

```
icon_filename,indent_level,1=open/0=close,項目文字列\n...
```

データ書式：(プロパティ WSNuseIcon が False の場合)

```
indent_level,1=open/0=close,項目文字列\n...
```

アイコン指定において、アイコンを省略した場合、プロパティ WSNiconPixmap で指定したアイコンが用いられます。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCtreeList.h>
extern WSCtreeList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにプロパティ経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_NONE);
  newlist_001->setProeprty(WSNdata,",0,1,item1\n,1,1,item2\n,2,1,item3");
```

```
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.7.3 ファイルからのツリーリストデータの設定

ファイル名を指定して、ファイルから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_FILE に指定します。次に、WSNdataSourceName プロパティにファイル名を指定します。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCtreeList.h>
extern WSCtreeList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにファイル経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_FILE);
  newlist_001->setProperty(WSNdataSourceName,"data.txt");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);

//data.txt の内容
$(WSDIR)/sys/pixmaps/bi16.xpm,0,1,item1
1,1,item2
2,1,item3
3,1,item4
$(WSDIR)/sys/pixmaps/bi16.xpm,0,1,item5
1,1,item6
2,1,item7
3,1,item8
```

2.7.4 インスタンスからのツリーリストのデータ表示

インスタンス名を指定して、インスタンスから一括して項目を設定することができます。この場合、まず、WSNdataSource プロパティを WS_DATA_SOURCE_INSTANCE に指定します。次に、WSNdataSourceName プロパティにインスタンス名を指定します。下記の例では、WSCtextField

のインスタンス newtext_000 を指定しています。データの形式は、ファイル指定の場合と変わりません。newtext_000 に入力されている文字列がリストに表示されます。ビルダーからのプロパティ設定、プログラムからの設定で動作します。下記はプログラムでの設定の例です。この場合、入力フィールドのインスタンス newtext_000 に入力されている文字列がツリーリストの項目として表示されます。

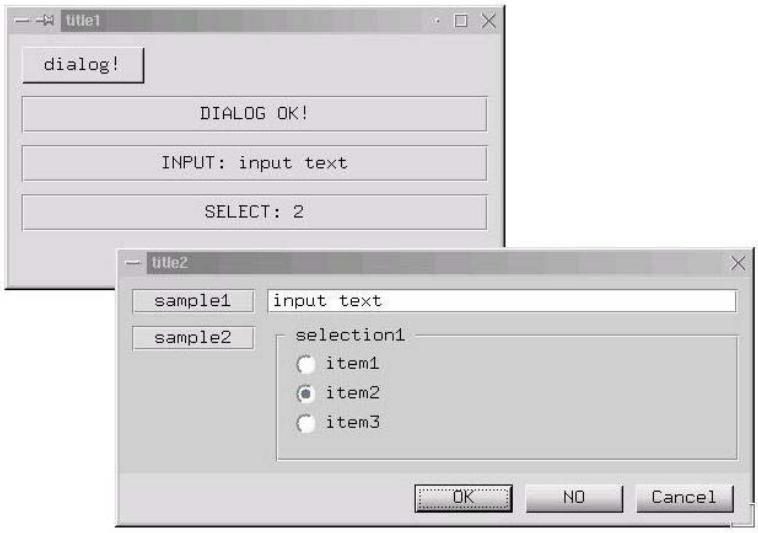
```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCtreeList.h>
extern WSCtreeList* newlist_001;
void btnep1(WSCbase* object){
  //リストの項目を全て削除。
  newlist_001->delAll();
  //リストにファイル経由でデータを設定
  newlist_001->setProperty(WSNdataSource,WS_DATA_SOURCE_INSTANCE);
  newlist_001->setProperty(WSNdataSourceName,"newtext_000");
}
static WSCfunctionRegister  op("btnep1",(void*)btnep1);
```

2.8 ユーザダイアログ

2.8.1 簡単なユーザダイアログの作成

WSCdialog を利用して簡単なユーザダイアログを作成してみましょう。次の様な機能をもっているものとします。

- あるボタンを押すと、ダイアログが表示されます。
- ダイアログには、インプットフィールドとラジオボタングループがあります。
- ダイアログ終了時に、正しく入力が行われているかチェックをします。
- ダイアログの入力結果をラベルに表示します。



［ユーザダイアログを使ったサンプル］

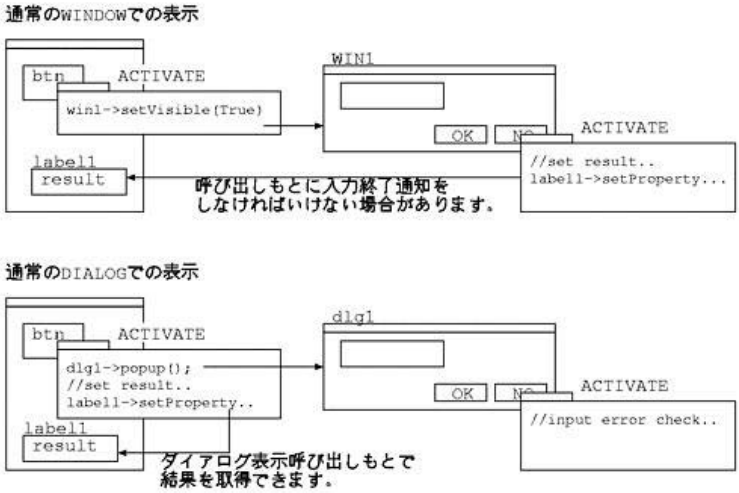
このサンプルは、ws/sampes/share/dialog/newproject.prj で提供されています

2.8.2 ユーザダイアログのポップアップ制御

ws/sampes/share/dialog/newproject.prj で提供されるサンプルをもとにユーザダイアログのポップアップ制御をみていきましょう。ダイアログは、入力、もしくは表示を行うための専用ウィンドウとしてよく用いられます。したがって、通常のウィンドウとして作成すると都合の悪い場合があります。たとえば、複数の場所 (イベントプロシージャ) から、同じダイアログを呼び出したいとします。

この場合、通常のウィンドウとして実装していると、ダイアログの入力が終った場合の通知を受けるのが複雑になります。 しかしながら、ダイアログで実装している場合では、ダイアログ入力終了が、popup メソッドによる完了復帰で同期がとられるので、簡単にイベントプロシージャから、ダイアログ入力の制御が行えます。

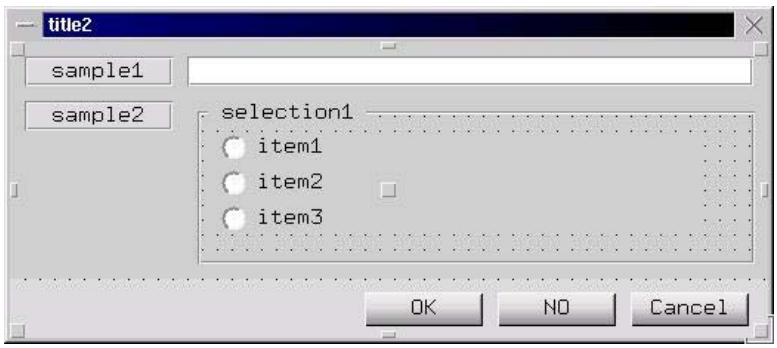
図は、表示されたウィンドウが結果を返す場合のウィンドウとダイアログの違いを示しています。



［ウィンドウとユーザダイアログを使った場合の違い］

簡単なダイアログの処理手順を順を追って説明しましょう。まず、ダイアログの作成です。WSCdialog クラスのインスタンスを一つ作成します。ダイアログ上に各々のオブジェクトを配置します。サンプルでは、下記のオブジェクトを配位しています。

- WSCvifield* newvifi_003
- WSCradioGroup* newradi_006



［簡単なユーザダイアログの例］

次に、ダイアログの OK,NO,CANCEL ボタンが押された場合の処理を記述します。ダイアログに ACITVATE トリガで、イベントプロシージャを張り付けます。このイベントプロシージャは、正しくダイアログに入力されたかどうかチェックし、されていなければ、メッセージダイアログでエラーを表示します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCdialog.h>
```

```
#include <WSCvifield.h>
#include <WSCradioGroup.h>
#include <WSCmessageDialog.h>
extern WSCvifield* newvifi_003;
extern WSCradioGroup* newradi_006;

void dialogep(WSCbase* object){
  WSCdialog* dialog = (WSCdialog*)object->cast("WSCdialog");
  if (dialog == NULL){ //A
    return;
  }
  if (dialog->getStatus() != WS_DIALOG_OK){ //B
    object->setVisible(False);
    return;
  }

  WSCstring str;
  str = newvifi_003->getProperty(WSNlabelString);
  if (!strcmp((char*)str,"")){                          //C
    WSCmessageDialog* msg = WSGIappMessageDialog();
    msg->setProperty(WSNdefaultPosition,True);
    msg->setProperty(WSNwidth,500);
    msg->setProperty(WSNlabelString,"Please input some string to the input field.");
    msg->popup();                       //D
    return;
  }
  long val = newradi_006->getProperty(WSNvalue);
  if (val == 0){                       //E
    WSCmessageDialog* msg = WSGIappMessageDialog();
    msg->setProperty(WSNdefaultPosition,True);
    msg->setProperty(WSNwidth,500);
    msg->setProperty(WSNlabelString,"Please select a item of the radio group.");
    msg->popup();                       //F
    return;
  }
  object->setVisible(False);            //E
}
static WSCfunctionRegister  op("dialogep",(void*)dialogep);
```

まずイベントプロシージャを張り付けたダイアログのクラスポインタを取得します。なぜかというと、ダイアログクラス固有の getStatus メソッドにアクセスして、押されたボタンの種類を調べるためです。

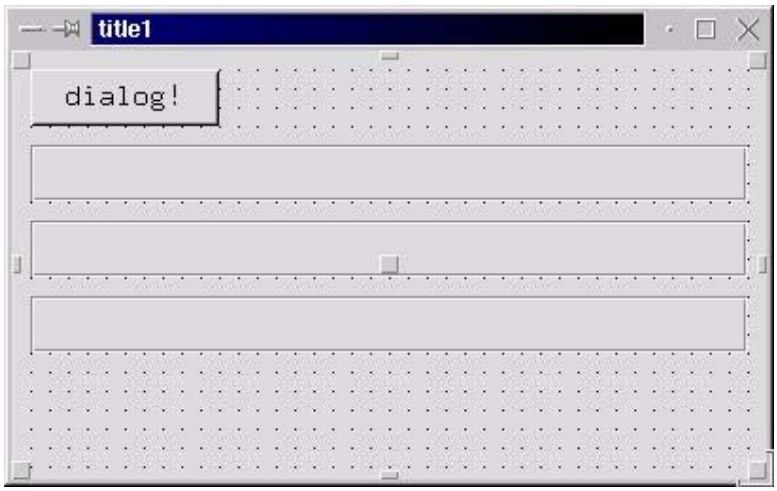
A で、ダイアログクラスでなかった場合、イベントプロシージャを終了します。B では、押され

たボタンをチェックします。OK ボタン以外は、ダイアログをそのまま不可視化します。

C で newvifi_003 に入力されているかチェックします。入力されていない場合は、D でメッセージダイアログを表示して、終了します。E で newradi_006 が選択されているかチェックします。選択されていない場合は、F でメッセージダイアログを表示して、終了します。

次に E で入力が正しい場合には、ダイアログを不可視化します。このとき、このダイアログを不可視化することにより、このダイアログを呼び出している popup メソッドが復帰します。したがって、不可視化しない場合、ダイアログが終了しません。

次はダイアログを呼び出しているイベントプロシージャの例です。



［ユーザダイアログを呼び出す画面の例］

[dialog!] と表示されたボタンを押すと、ダイアログが表示され、入力した結果が、3 つのラベルに表示されます。1 つ目のラベルには、ダイアログの OK、NO、CANCEL ボタンがおされた結果、2 つ目のラベルには、newvifi_003 の入力された結果、3 つ目のラベルには、newradi_006 の選択された結果を表示します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-----------------------------------------------------------
//Function for the event procedure
//-----------------------------------------------------------
#include <WSCdialog.h>
#include <WSCvifield.h>
#include <WSCradioGroup.h>
#include <WSCvlabel.h>
extern WSCdialog* newdial_001;
extern WSCvifield* newvifi_003;
extern WSCradioGroup* newradi_006;
extern WSCvlabel* newvlab_007;
extern WSCvlabel* newvlab_010;
extern WSCvlabel* newvlab_011;
```

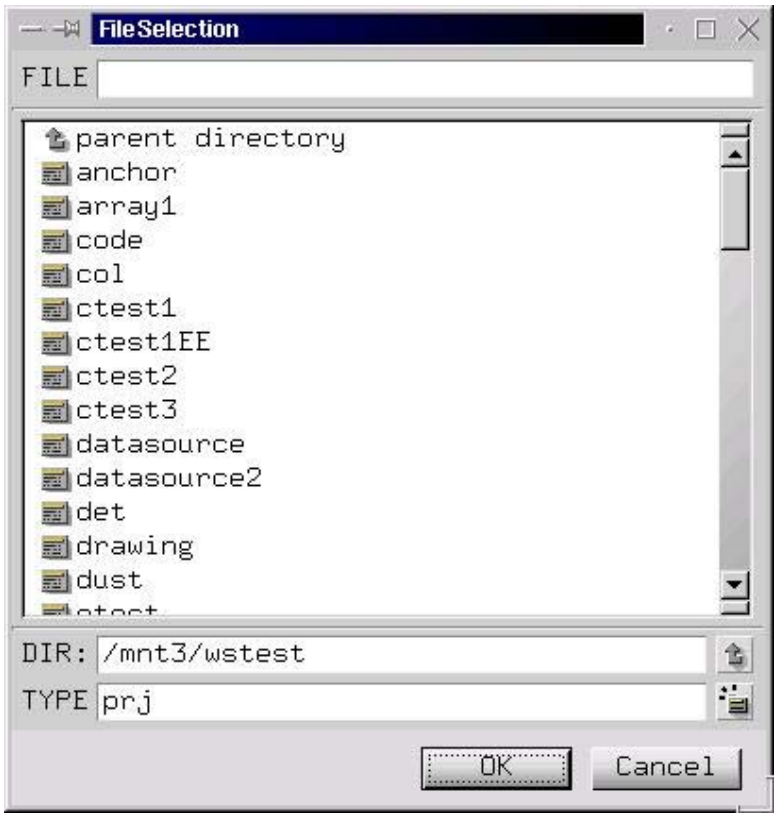
```
void btnep(WSCbase* object){
  long val = newdial_001->popup();
  if (val == WS_DIALOG_OK){
     newvlab_007->setProperty(WSNlabelString,"DIALOG OK!");
  }else
  if (val == WS_DIALOG_NO){
     newvlab_007->setProperty(WSNlabelString,"DIALOG NO!");
  }else
  if (val == WS_DIALOG_CANCEL){
     newvlab_007->setProperty(WSNlabelString,"DIALOG CANCEL!");
  }
  WSCstring tmp;
  tmp = newvifi_003->getProperty(WSNlabelString);
  WSCstring tmp2;
  tmp2 << "INPUT: " << tmp;
  newvlab_010->setProperty(WSNlabelString,tmp2);

  val = newradi_006->getProperty(WSNvalue);
  tmp2 = "SELECT: ";
  tmp2 << val;
  newvlab_011->setProperty(WSNlabelString,tmp2);
}
static WSCfunctionRegister  op("btnep",(void*)btnep);
```

2.9 ファイル選択ダイアログ

2.9.1 ファイル選択ダイアログの表示

ファイル選択ダイアログの使い方について触れてみましょう。ファイル選択ダイアログの様なダイアログに関しては、便利な表示メソッド popup が用意されています。



［ファイル選択ダイアログ］

popup メソッドは、単にダイアログを表示させるだけでなく、オペレータが OK やキャンセルなどを選択した結果を関数復帰で返します。したがって、関数を呼ぶだけで、ダイアログの表示から、ダイアログの終了した時点の選択結果の取得を行うことができます。ファイル選択ダイアログやメッセージダイアログに関して実際のサンプルプログラムを見ていきましょう。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
#include <WSCfileSelect.h> //(A)
#include <WSCmessageDialog.h> //(B)
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
void btnep2(WSCbase* object){
  //アプリケーションに 1 つあらかじめ用意されています。
  //ファイル選択ダイアログのインスタンスにアクセスします。
  WSCfileSelect* fs = WSGIappFileSelect(); //(C)
  fs->setProperty(WSNmaskFileName,"cpp");  //(D)
  fs->setProperty(WSNdefaultPosition,True); //(E)
  long ret = fs->popup();                  //(F)
```

```
  //アプリケーションに 1 つあらかじめ用意されている
  //メッセージダイアログのインスタンスを取得する。
  WSCmessageDialog* msg = WSGIappMessageDialog(); //(G)
  msg->setProperty(WSNwidth,500);                 //(H)
  msg->setProperty(WSNheight,120);                //(I)
  msg->setProperty(WSNdefaultPosition,True);      //(J)

  if (ret == WS_DIALOG_OK){           //(K)
    WSCstring  str;
    str << fs->getFileName() << " が選択されました。";
    msg->setProperty(WSNlabelString,str);
    msg->popup();
  }else if (ret == WS_DIALOG_NO){     //(L)
    msg->setProperty(WSNlabelString,"選択されませんでした。");
    msg->popup();
  }else if (ret == WS_DIALOG_CANCEL){
    msg->setProperty(WSNlabelString,"選択は取り消されました。");
    msg->popup();                     //(M)
  }
}
```

(A),(B) でファイル選択ダイアログやメッセージダイアログにアクセスするので、ヘッダーファイルをインクルードしておきます。

(C) でグローバルインスタンスアクセス関数を使用して、ファイル選択ダイアログのグローバルインスタンスにアクセスします。ファイル選択ダイアログなどのクラスには、あらかじめいつでも利用できるようにグローバルなインスタンスが 1 つ用意されています。そして、グローバルインスタンスアクセス関数によってアクセスできるようになっています。

(D),(E) でプロパティを設定し、(F) で、ダイアログを表示状態にします。実際に選択が行われると、popup メソッドが復帰して来ます。

ファイル選択ダイアログの選択結果をメッセージダイアログを使って表示してみるとします。(G),(H),(I),(J) で、メッセージダイアログのグローバルインスタンスを取得して、プロパティを設定します。

(K) では、選択されたファイルをメッセージ文字列として表示します。(L) では、ファイル選択が無かった主旨をメッセージ文字列として表示します。(M) では、キャンセルされた主旨をメッセージ文字列として表示します。

2.10 スクロールドフォーム

2.10.1 仮想スクロール機能を使用するには

比較的大きいスクロールウィンドウ領域を設定した場合、ウィンドウ資源を大量に消費する場合があります。そのような場合に、実際にはウィンドウ領域をもたないで仮想的にスクロールさせる仮想スクロール機能があります。仮想スクロールモードでは、次のような特徴があります。

- 大きいスクロール領域を設定しても、ウィンドウ資源を必要としない。

また、仮想スクロールモードでは、次のような短所があります。

- ウィンドウ資源を使用した通常のスクロールに比べ、描画パフォーマンスが悪い。
- WSCform 系のウィンドウ資源をもつものは仮想スクロールできない。

WSCform など、ウィンドウ資源をもつオブジェクトクラスは、仮想スクロールモードでスクロールしません。WSCv で始まるウィンドウ資源を持たないクラスが使用できます。

2.11 セパレーテッドフォーム

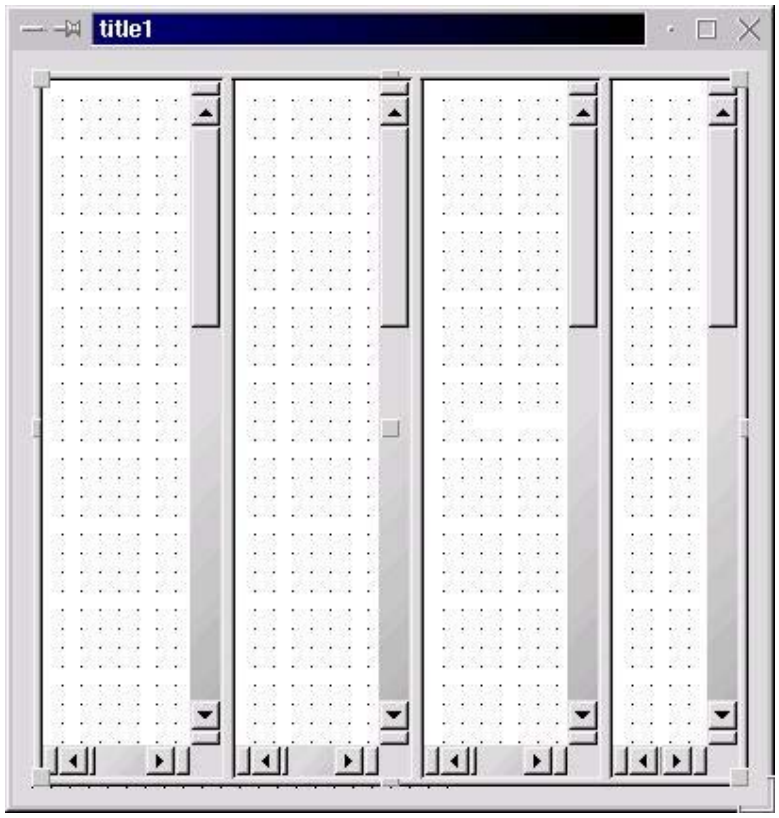
セパレーテッドフォームは、マウスで移動することのできるセパレータを持った、複数に区分された領域をもつフォームです。セパレータを移動することにより、各領域の大きさを変更することができます。

2.12 セパレーテッドフォームの設定方法

まず、区分される領域が縦に区分されるか、横に区分されるかを決めます。プロパティ WSNorientation に縦か横を設定します。次に、プロパティWSNbarValue に、セパレータバーの位置を設定します。

例えば、プロパティWSNorientation を縦に設定して、縦のセパレータバーが３つある場合、すなわち、領域が４つある場合、それぞれのバーの位置の設定が、左から、100,200,300 ドットの所にあるとすると、WSNbarValue を”100,200,300”と設定します。

そしてそれぞれの、領域に、スクロールドフォームや、リストを配置します。



［スクロールドフォームを配置した、バー位置 100,200,300 に設定したセパレーテッドフォーム］

2.13 ドローイングエリア

2.13.1 ドローイングエリアで図形を描画するには

ドローイングエリア (WSCvdrawingArea クラス) は、画面領域に自由に図形を描画することが可能です。 ドローイングエリアには、描画用のメソッドが用意されており、EXPOSE イベント (露出イベント) で、これらのメソッドを用いて描画します。次に示すサンプルプログラムは、ドローイングエリアでの基本的な描画方法を示します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCvdrawingArea.h>
#include <WSCvslider.h>

void drawep(WSCbase* object){
  //drawing_a is same as newvdra_000...
  //You can get it extern WSCvdrawingArea* newvdra000; also.
```

```
  WSCvdrawingArea* drawing_a =
         (WSCvdrawingArea*)object->cast("WSCvdrawingArea"); //(A)
  if (drawing_a == NULL){ //(B)
    return;
  }

  drawing_a->setForeColor("\#ff0000"); //(C)
  drawing_a->drawLine(0,0,100,100);  //(D)

}
static WSCfunctionRegister  op("drawep",(void*)drawep);
```

まず、ドローイングエリアのメソッドにアクセスするために、(A) に示す様に、ドローイングエリアクラス (WSCvdrawingArea) のポインタを取得します。WSCbase クラスのポインタのままですと、ドローイングエリアのメソッドにアクセスできないからです。

イベントプロシージャが間違って他のクラスに張られる場合もあるので、ドローイングエリアクラスかどうか (B) で判別します。ポインタの取得結果が NULL でなければ、ドローイングエリアクラスです。

(C) で描画する時の色を指定しています。(D) では、座標 (0,0) から座標 (100,100) へ線を描画しています。

ドローイングエリアには、線を描画するメソッドの他に、次のようなメソッドがあります。

- 矩形の描画
- 円、円弧、楕円の描画
- 多角形の描画
- イメージの描画

2.13.2 ドローイングエリアでイメージを描画するには

イメージを表示したい場合は、drawImage メソッド、もしくは drawStretchedImage メソッドを用います。drawStretchedImage メソッドは、drawImage メソッドがそのままの大きさで表示するのにたいし、与えられたサイズにイメージを拡大縮小させて表示します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//------------------------------------------------------------
//Function for the event procedure
//------------------------------------------------------------
#include <WSCvdrawingArea.h>
#include <WSCvslider.h>
```

```
void drawep(WSCbase* object){
  //drawing_a is same as newvdra_000...
  //You can get it extern WSCvdrawingArea* newvdra000; also.
  WSCvdrawingArea* drawing_a =
         (WSCvdrawingArea*)object->cast("WSCvdrawingArea");
  if (drawing_a == NULL){
    return;
  }
  WSCushort w = drawing_a->getProperty(WSNwidth);
  WSCushort h = drawing_a->getProperty(WSNheight);
  drawing_a->drawStretchedImage(0,0,w,h,"001.jpg");  //(A)

}
static WSCfunctionRegister  op("drawep",(void*)drawep);
```

A では、ドローイングエリアのサイズにあわせてイメージを表示しています。

2.14 インデックスドフォーム

インデックスドフォームは、インデックスタブで表示を切替えることのできるフォームです。インデックスドフォームの扱い方は、まず、プロパティWSNmenuItems を設定することから始まります。例えばインデックスタブを、タブ 1、タブ 2、タブ 3 とする場合には、カンマで区切って次のように設定します。

プロパティ WSNmenuItems
タブ 1, **タブ** 2, **タブ** 3

次に、それぞれのタブが選択された場合の表示内容の編集は、それぞれのタブを選択してから行います。

どのタブが現在選択されているかは、プロパティ WSNvalue で知ることが来ます。上記の場合、タブ 1 が 0、タブ 2 が 1、タブ 3 が 2 です。左から順に、0,1,2,3... となります。また、このプロパティ WSNvalue を指定すると、そのタブが選択表示された状態にすることができます。

2.15 バルーンヘルプ

2.15.1 バルーンヘルプを表示するには

オブジェクトボックスの Non GUI にあるバルーンヘルプオブジェクト (WSCvballoonHelp) をどこかのウィンドウ、もしくはフォーム上に配置します。そして、バルーンヘルプを表示したいインスタンスの名称を確認して、そのバルーンヘルプインスタンスのプロパティ WSNclient にそのインスタンス名称指定します。次に、プロパティ WSNlabelString に表示したいバルーンヘルプ文字列を設定します。

2.16 タイマー

2.16.1 タイマーを使用するには

オブジェクトボックスの Non GUI にあるタイマークラス (WSCvtimer) をどこかのウィンドウ、もしくはフォームに配置します。次に、プロパティWSNinterval に起動間隔時間をミリ秒単位で設定します。タイマーが実行されると、時間が経過した後、ACTIVATE イベントが起動されます。ACITVATE トリガでイベントプロシージャを張り付けてください。

タイマーの種類に 2 種類あります。1 つは、時間が経過したのち、一度だけ ACTIVATE イベントを起動するもの、もう 1 つは時間が経過する度に何度でも起動するものです。プロパティ WSNcont を True にすると、継続して起動します。また、プロパティ WSNrunning で、実行中のタイマーを止めたり、開始したりすることができます。デフォルトでは、プロパティ WSNrunning は False になっており、タイマーは止まっています。タイマーを動かす時は、実行を開始したい時に WSNrunning を True にしてください。プロパティ WSNcont が False の場合は、一度、イベントを起動した後、タイマー停止状態になります。

2.17 ウィザードダイアログ

ウィザードダイアログは、対話形式のダイアログを作成するのに利用します。

ウィザードダイアログは内部にインデックスタブを表示していないインデックスドフォームを一つ抱えていて、「¡戻る」「次へ¿」ボタンを操作することで、順番に表示が切り替わります。

ウィザードダイアログの扱い方は、まず、プロパティWSNmenuItems に切り替わり画面数を設定することから始まります。

プロパティ `WSNmenuItems:` **切り替わり画面数を設定。**

例えば、設定画面が 5 つある対話形式ダイアログを作成する場合は、

プロパティ WSNmenuItems を 5 と設定します。

次に、ダイアログボタンの設定です。 例えば、通常「¡戻る」「次へ¿」と表示して、最後の画面で、「¡戻る」「完了」と表示する場合は、プロパティWSNlabelString を次のように設定します。

プロパティ `WSNlabelString:` **<戻る , 次へ>, 完了**

どの画面が現在選択されているかは、プロパティ WSNvalue で知ることが来ます。上記の場合、最初の画面が 0、その次がが 1,2,... と続きます。このプロパティ WSNvalue を指定すると、その対応する画面が表示された状態にすることができます。それぞれの画面を編集する場合は、このプロパティを設定しながら、編集します。

プロパティ `WSNvalue:` **画面を指定**

(注意) 編集時は、このプロパティを操作しながら、それぞれの画面を編集を行います。

2.18 オフセット指定による描画制御

2.18.1 オフセット変数による XY 座標の操作

WSCvXXX 系のクラスは、setXOffsetPtr(short*), setYOffsetPtr(short*) を利用して、表示位置の調整ができます。

初期化トリガにて、以下のようにします。

```
extern short xoffset; //A
extern short yoffset; //A
void initep(WSCbase* object){
  object->setXOffsetPt(&xoffset); //B
  object->setYOffsetPt(&yoffset); //B
}
```

まず、どこかに、グローバル変数として xoffset,yoffset を定義しておきます。A では、そのグローバル変数を外部アクセス宣言します。B で、その変数のポインタをオフセットとして設定します。設定された以後、インスタンスは、変数の内容を加算した座表に、表示されるようになります。

複数のオブジェクトに同じオフセット変数を設定し、そのオフセット変数の内容を変更し、表示更新を行うことで、複数のオブジェクトを一度に操作することができます。

2.18.2 スケールオフセットによるサイズの操作

WSCvXXX 系のクラスは、setScaleOffsetPtr(double*) を利用して、表示サイズの調整ができます。

初期化トリガにて、以下のようにします。

```
extern double scaleoffset; //A
void initep(WSCbase* object){
  object->setScaleOffsetPt(&scaleoffset); //B
}
```

A では、いずれかのソースに宣言されているであろう、グローバル変数等を外部アクセス宣言します。B で、その変数のポインタを設定します。設定された以後、インスタンスは、変数の内容を乗算したサイズで、表示されるようになります。

また、複数のオブジェクトに同じオフセット変数を設定し、そのオフセット変数の内容を変更し、表示更新を行うことで、複数のオブジェクトを一度に操作することができます。

2.19 メモリーデバイスを利用したイメージの作成と表示

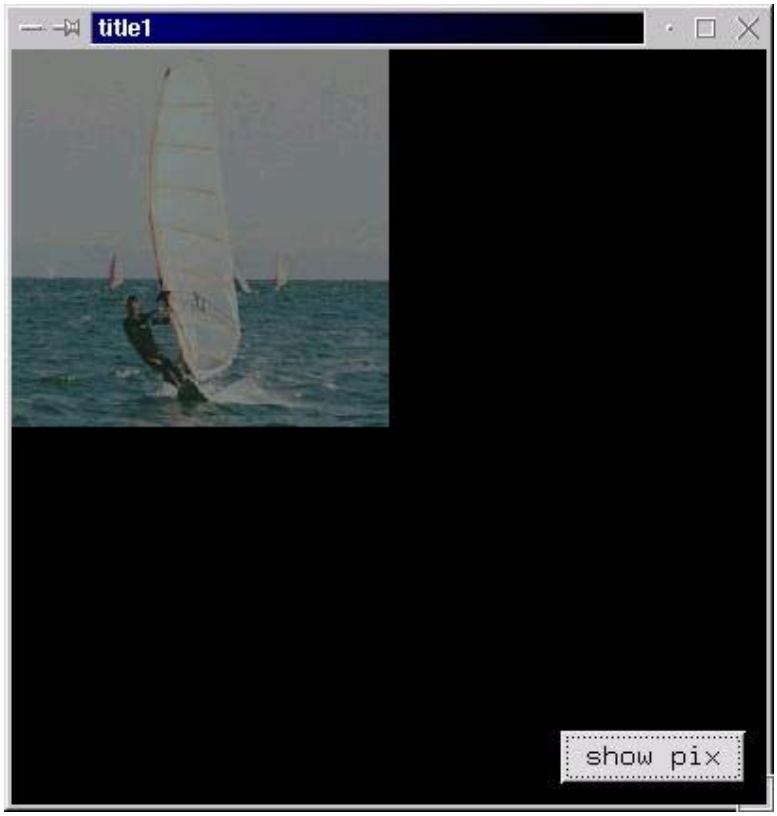
2.19.1 メモリーデバイスの作成と表示

メモリデバイスを利用するとイメージ画像の直接操作を行うことができます。

- メモリデバイスへの図形の描画
- メモリデバイスへのイメージ画像の描画

- メモリデバイスの直接参照と、直接操作
- メモリデバイスのウィンドウへの表示

ws/sampes/share/memdev/newproject.prj で提供されているサンプルを例に、メモリデバイスの使い方について見ていきましょう。このサンプルは、ボタンを押すと 001.jpg なるイメージ画像を読み込んで、だんだん浮かび上がって来るように描画します。



[段々と浮かび上がって来るイメージ画像の表示]

このサンプルの処理の概要は次の通りです。

- メモリデバイスを 2 つ作成
- メモリデバイス 1 に 001.jpg を描画
- 1. メモリデバイス 2 にメモリデバイスから輝度を変えて転送
- 2. メモリデバイス 2 の内容をウィンドウに転送
- 輝度あげながら、1,2 を繰り返し

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSDappDev.h>
#include <WSCcolorSet.h>
#include <WSCimageSet.h>
#include <WSCmainWindow.h>
extern WSCmainWindow* newwin000;

#include <WSDmwindowDev.h>
WSDmwindowDev* mdev = NULL;
WSDmwindowDev* mdev2 = NULL;

void btnep(WSCbase* object){
  WSDdev* dev = newwin000->getdev(); //A

  if (mdev == NULL){ //B
    mdev = WSDmwindowDev::getNewInstance();
    mdev2 = WSDmwindowDev::getNewInstance();
  }

  mdev->createPixmap(200,200); //C
  mdev->beginDraw(0,0,200,200);  //D
  WSDimage* image = WSGIappImageSet()->getImage("001.jpg"); //E
  mdev->drawStretchedImage(0,0,200,200,image); //F
  mdev->endDraw(); //G

  mdev2->createPixmap(200,200);  //H

  mdev->initBuffer();           //I
  mdev2->initBuffer();          //J
```

```
  long i,x,y;
  for(i=0;i<100; i++){
    for(x=0; x<200; x++){
      for(y=0; y<200; y++){
        WSCuchar r,g,b;
        mdev->getBufferRGB(x,y,&r,&g,&b); //K
        r = (WSCushort)((double)(r*i)/100); //L
        g = (WSCushort)((double)(g*i)/100); //L
        b = (WSCushort)((double)(b*i)/100); //L
        mdev2->setBufferRGB(x,y,r,g,b);     //M
      }
    }
    mdev2->putBufferToPixmap();  //N
    mdev2->copyToWindow(dev,0,0,200,200,0,0); //P
  }
}
static WSCfunctionRegister  op("btnep",(void*)btnep);
```

A で描画先のウィンドウのデバイスを取得します。

B では、最初にボタンがおされた場合に、メモリデバイスを作成します。メモリデバイスは、new 演算子での作成は出来ません。メモリデバイスは、ウィンドウシステム異存ですので、getNewInstance メソッドがウィンドウシステムに適合したインスタンスを作成します。

次は，メモリデバイス 1 へのイメージ画層 001.jpg の描画です。メモリデバイスへ描画を行う前に、C のように createPixmap メソッドでメモリデバイスの大きさを指定して初期化を行います。

次にメモリデバイスへ描画です。描画を行う場合は，まず、D のように beginDraw() メソッドを呼び出します。E で、イメージ管理グローバルインスタンスからイメージを取得します。F では、取得したイメージを表示しています。描画がおわったら、endDraw() メソッド呼び出します。

H でメモリデバイス 2 も初期化しておきます。I,J で直接操作のためのバッファを初期化します。このとき、内部のイメージのデータがバッファに転送されます。K で、メモリデバイス 1 の RGB 値を取得します。L で、取得した RGB 値の輝度を調節し、M で、メモリデバイス 1 にセットします。

N では、バッファに設定された内容を内部のイメージに転送しなおし，反映します．P で、反映したイメージデータをウィンドウに転送し、表示しています。100 段階の輝度計算のループを行い、輝度をあげながら 100 回、イメージを表示します。

2.20 TCP/IP を使ったネットワーク通信

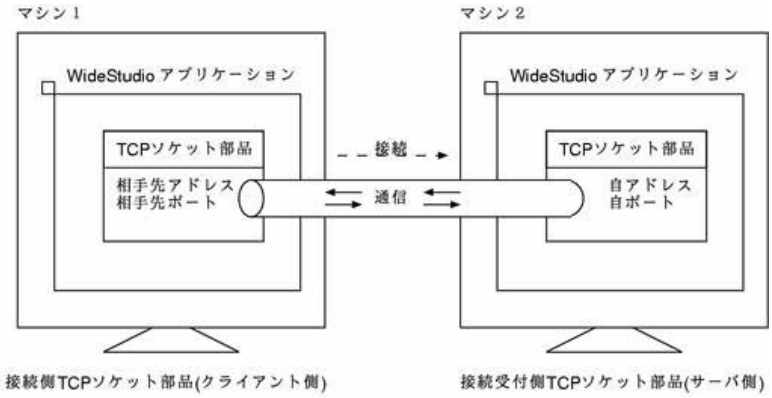
2.20.1 TCP ソケットを使ったネットワーク通信をするには

TCP ネットワーク通信は、クライアント、サーバ接続指向の通信を行います。サーバ側ソケット WSCvssocket クラスは、クライアント側ソケット WSCvcsocket からの接続を受け付けます。

通常、TCP ソケットを C/C++ 言語上で取り扱う場合、accept や、listen、connect 等を利用し

て接続をおこないますが、WideStudio では、TCP ソケットの接続に関する処理は部品内部で自動的に行い、部品の利用者には隠蔽されていて、それらに関する処理は記述する必要がありません。

TCP ソケット部品は、IP アドレスや、PORT のプロパティを持ち、設定するだけで機能するように出来ており、ソケット部品の利用者は単に、TCP ソケット部品に対し、データの送受信を行います。TCP ソケット部品には、接続を行う側 (クライアント側) 接続を受け付ける側 (サーバ側) とがあります。

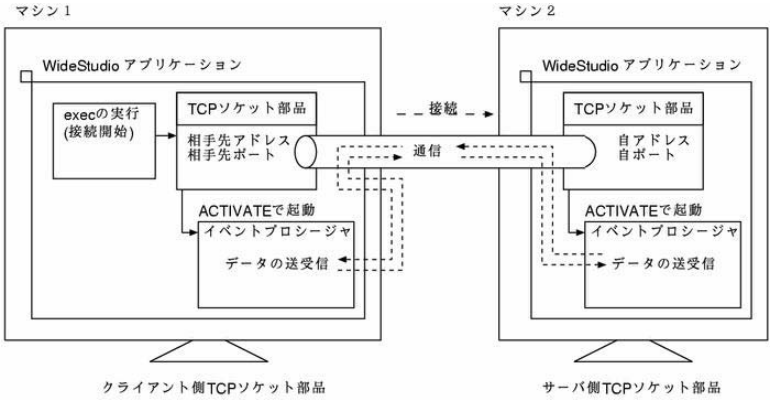


クライアント側とサーバ側では、主に接続を行う際の動作が異なります。クライアント側はネットワーク上のある特定のアドレスとポートに存在するサーバーに接続しに行くのに対して、サーバ側は、クライアントから接続されるのを待ちます。

従って、TCP 通信を行う場合、WSCvcsocket (クライアント側) は必ず、接続待機している WSCvssocket (サーバ側) に接続しなければなりません。

クライアント側におけるプロパティの設定は、WSNip に接続先サーバの TCP/IP アドレスを指定し、WSNport に相手先サーバのソケットのポートを指定します。サーバ側におけるプロパティの設定は、WSNport に受け付け用のソケットのポートを指定し、WSNrunning をオンに設定します。WSNip には通常、特に設定を行いませんが、もし同じマシンに複数のアドレスが存在し、そのアドレスのうち受け付けるアドレスを特定する場合のみ、WSNip を指定します。

クライアント側で WSCvcsocket::exec メソッドを実行することにより、サーバー側の WSCvssocket へ接続が行われます。接続が行われると、クライアント側、サーバー側双方で ACITVATE イベントが発生し、その起動されるイベントプロシージャにおいて通信を行います。



接続に成功した際、ACTIVATE で起動されるイベントプロシージャのクライアント側データ送受信のサンプルです。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCvcsocket.h>

void com_ep(WSCbase* object){
  //do something...
  WSCvcsocket* sock = (WSCvcsocket*)object->cast("WSCvcsocket");
  char buffer[128];
  sprintf(buffer,"test!!! %d",cnt);
  cnt++;

  //send data;
  long send_len = sock->write((WSCuchar*)buffer,128);
  if (send_len == 128){
    //success! do something..
  }else{
    //error!
    return;
  }

  //receive data;
  buffer[0] = 0;
  long recv_len = sock->read((WSCuchar*)buffer,128);
  if (recv_len == 128){
    //success! do something..
  }else{
    //error!
    return;
  }
}
static WSCfunctionRegister  op("com_ep",(void*)com_ep);
```

次は接続確立時に ACTIVATE で起動されるサーバー側のデータ送受信プロシージャのサンプルです。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
```

```
//-------------------------------------------------------------
#include <WSCvssocket.h>

void com_ep(WSCbase* object){
  //do something...
  WSCvssocket* obj = (WSCvssocket*)object->cast("WSCvssocket");
  char buffer[128];

  //receive data:
  //クライアントからのデータが buffer に格納
  obj->read((WSCuchar*)buffer,128);

  //send data:
  //クライアントへののデータを buffer に格納して送信。
  strcpy(buffer,"send data...");
  obj->write((WSCuchar*)buffer,128);
}
static WSCfunctionRegister  op("com_ep",(void*)com_ep);
```

次は接続確立してデータの送受信をさせる、クライアント側のイベントプロシージャのサンプルです。

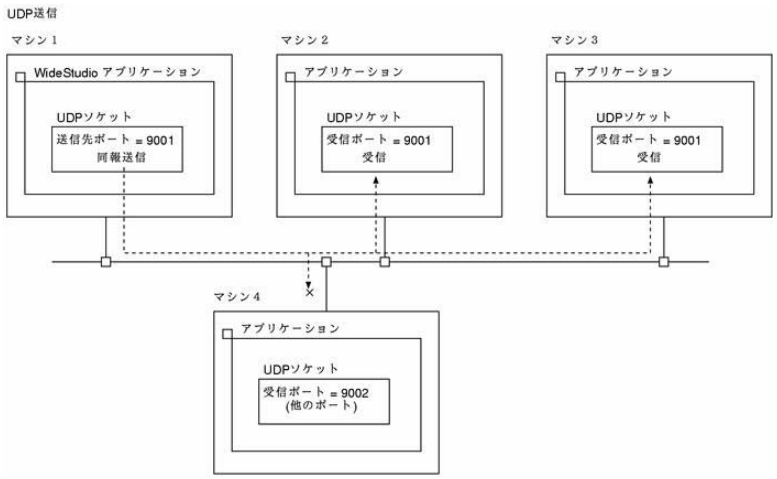
```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCvcsocket.h>
extern WSCvcsocket* newvcso_000;

void btnop(WSCbase* object){
  //接続開始、接続後、データ送受信を行わせます。
  long ret = newvcso_000->exec();
  if (ret != WS_NO_ERR){ //接続失敗
    return;
  }
}
static WSCfunctionRegister  op("btnop",(void*)btnop);
```

2.20.2 UDP ソケットを使った同報ネットワーク通信をするには

UDP の場合、特定の相手と接続することなく、不特定多数 (ブロードキャストアドレス:通常は、xxx.xxx.xxx.255) に対してデータ送信することができます。送信時に指定されたポートでデータを

待ち受けているものがデータを受け取ります。



送信側におけるプロパティの設定は、WSNport に相手先のソケットのポートを指定します。また、WSNip にブロードキャストアドレス、通常は、255.255.255.255、を指定しますが、255.255.255.255 では送信できないシステムでは、送信先ネットワークアドレスの末桁を 255 にしたものを指定します。たとえば、10.20.30.XX に存在する不特定のマシンに送信する場合、10.20.30.255 を指定します。

受信側におけるプロパティの設定は、WSNport に受け付け用のソケットのポートを指定し、WSNrunning をオンに設定します。WSNip には通常、特に設定を行いませんが、もし同じマシンに複数のアドレスが存在し、そのアドレスのうち受け付けるアドレスを特定する場合のみ、WSNip を指定します。

送信側は、接続を行わない分、TCP の場合に比べて単純です。単に、WSCvudpsocket::write を呼び出し、データを送信します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//------------------------------------------------------------
//Function for the event procedure
//------------------------------------------------------------
#include <WSCvudpsocket.h>
extern WSCvudpsocket* newvudp_000;

void btnop(WSCbase* object){
  static long cnt = 0;
  WSCuchar buffer[64];
  //buffer にデータを格納し、送信
  strcpy(buffer,"send data..");
  long ret = newvudp_000->write(buffer,64);
  if (ret < 64){
    //送信失敗
  }else{
```

```
    //送信成功
  }
}
static WSCfunctionRegister  op("btnop",(void*)btnop);
```

受信側は、TCP のサーバー側と同じように、ACTIVATE イベントで起動するイベントプロシージャにおいて受信します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCvudpsocket.h>
extern WSCvudpsocket* newvudp_000;

void recvop(WSCbase* object){
  WSCuchar buffer[64];
  //データの受信
  newvudp_000->read(buffer,64);
}
static WSCfunctionRegister  op("recvop",(void*)recvop);
```

2.21 データーベースクラスを利用したデータベースアクセス

2.21.1 ODBC を通じたデータベースアクセス

WSCvdb クラスを利用すると ODBC インターフェースを通じてデータベースにアクセスすることができます。

ODBC にアクセスするには、プロパティ WSNtype に WS_DB_ODBC を指定し、WSCvdb::open 関数に DSN 、ユーザー名、パスワードを指定して実行します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  long ret = newvdb_000->open("dn","user","passwd");
```

```
  if (ret == WS_NO_ERR){
    //接続。
  }else{
    //接続失敗、エラーメッセージを取得。
    char buffer[1024];
    newvdb_000->getErrorMsg(buffer,1024);
  }
}
```

ODBC にアクセスするには、プロパティ WSNhostname に DSN、WSNusername にユーザ名、WSNpassword にパスワードを指定し、引数無しで WSCvdb::open 関数を実行します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//------------------------------------------------------------
//Function for the event procedure
//------------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  long ret = newvdb_000->open();
  if (ret == WS_NO_ERR){
    //接続。
  }else{
    //接続失敗、エラーメッセージを取得。
    char buffer[1024];
    newvdb_000->getErrorMsg(buffer,1024);
  }
}
```

2.21.2 PostgreSQL インターフェースを通じたデータベースアクセス

WSCvdb クラスを利用すると PostgreSQL インターフェースを通じて直接 PostgreSQL データベースにアクセスすることができます。

PostgreSQL インターフェースを通してアクセスする場合は、プロパティWSNtype に WS_DB_POSTGRES を指定し、WSCvdb::open 関数にホスト名、ユーザー名、パスワード、データーベース名、ポート番号を文字列で指定して実行します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//------------------------------------------------------------
```

```
//Function for the event procedure
//-------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  long ret = newvdb_000->open("dn","user","passwd","dbname","5432");
  if (ret == WS_NO_ERR){
    //接続。
  }else{
    //接続失敗、エラーメッセージを取得。
    char buffer[1024];
    newvdb_000->getErrorMsg(buffer,1024);
  }
}
```

PostgreSQL にアクセスするには、プロパティ WSNhostname にデータベースの存在するホスト名、WSNusername にユーザ名、WSNpassword にパスワード、WSNdbname にデータベース名、WSNport にポート番号を指定し、引数無しで WSCvdb::open 関数を実行します。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  long ret = newvdb_000->open();
  if (ret == WS_NO_ERR){
    //接続。
  }else{
    //接続失敗、エラーメッセージを取得。
    char buffer[1024];
    newvdb_000->getErrorMsg(buffer,1024);
  }
}
```

2.21.3 テーブルの作成

WSCvdb::open を実行してデータベースへの接続が成功した場合、SQL を発行してデータベースを操作することができます。次の例は、データベース上に shinamono というテーブルを一つ作

成する例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  char buf1[512];
  strcpy(buf1, "create table shinamono(code int, hinmei char(20), nedan float)");
  newvdb_000->sqlExecute(buf1);

  if (ret == WS_NO_ERR){
    //成功
  }else{
    //接続失敗、エラーメッセージを取得。
    char buffer[1024];
    newvdb_000->getErrorMsg(buffer,1024);
  }
}
```

2.21.4 テーブルへのデータの格納

WSCvdb::open を実行してデータベースへの接続が成功し、操作可能なテーブルが存在する場合、SQL を発行してテーブルにデータを確報することができます。次の例は、データベース上に shinamono というテーブルに品物のデータを格納する例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-------------------------------------------------------------
//Function for the event procedure
//-------------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  newvdb_000->beginTran();
  strcpy(buf1, "insert into shinamono values(1, 'みかん', 100)");
  newvdb_000->sqlExecute(buf1);
```

```
  strcpy(buf1, "insert into shinamono values(2, 'りんご', 200)");
  newvdb_000->sqlExecute(buf1);
  strcpy(buf1, "insert into shinamono values(3, 'バナナ', 300)");
  newvdb_000->sqlExecute(buf1);
  strcpy(buf1, "insert into shinamono values(4, 'メロン', 0)");
  newvdb_000->sqlExecute(buf1);

  newvdb_000->commitTran();

}
```

2.21.5 テーブル上のデータの参照

WSCvdb::open を実行してデータベースへの接続が成功し、参照可能なテーブルが存在する場合、SQL を発行してテーブル上のデータを確報することができます。次の例は、データベース上に shinamono というテーブルから品物のデータを参照し、そしてデータを更新する例です。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//------------------------------------------------------------
//Function for the event procedure
//------------------------------------------------------------
#include <WSCvdb.h>
extern WSCvdb* newvdb_000;

void db_ep(WSCbase* object){
  newvdb_000->beginTran();
  WSCdbRecord rs(newvdb_000);
  if(rs.open("select * from shinamono order by code") == WS_NO_ERR) {
    while (!rs.isEOF()) {
      rs.getColValue("code", &var);
      int code = (int)var;
      cout << "code:" << (int)var << " ";
      rs.getColValue("hinmei", &var);
      cout << "hinmei:" << (char*)var << " ";
      rs.getColValue("nedan", &var);
      char buf[80];
      double nedan = (float)var + 10;

      sprintf(buf, "%f", (float)var);
      cout << "nedan:" << buf << "\n";
```

```
      if(nedan != 0) {
        sprintf(buf1, "update shinamono set nedan = %f where code = %d", nedan, code);
      } else {
        sprintf(buf1, "delete from shinamono where code = %d", code);
      }
      newvdb_000->sqlExecute(buf1);
      rs.moveNext();
    }
  }
  rs.close();
  newvdb_000->commitTran();
}
```

第3章　クラス編

3.1　メンバオブジェクトにアクセスするには

3.1.1　クラスイベントプロシージャ中でのメンバオブジェクトにアクセス

メンバオブジェクトにアクセスするには、まずそのオブジェクトが、メンバとして定義される必要があります (クラスアプリケーションウィンドウ編の［オブジェクトをメンバ変数にするには］の節を参照下さい)。

次の例は、あるクラス (sample) のイベントプロシージャの例です。

sample* の base ポインタが数値入力クラスのオブジェクト本体です。メンバのオブジェクト (例えば、newvlab000) は、base-¿newvlab000 という具合にアクセスします。

```
#include <sample.h>
void sample::event_procedure(WSCbase* object){
  sample* base = (sample*)object->getUserData(WS_BASE_CLASS);
  //何か処理を記述して下さい

  //メンバのオブジェクトにアクセス...
  WSCvariant val = base->newvlab000->getProperty(WSNuserValue);

   ...
}
```

メンバのオブジェクト newvlab000 に base-¿ でアクセスしている様子が分かります。

3.1.2　メンバ関数中でのメンバオブジェクトにアクセス

クラスイベントプロシージャ中でのメンバオブジェクトにアクセスするには、base-¿ が付きましたが、メンバ関数内 (プロパティハンドラも含みます) では、その必要はありません。

次の例は、同じくあるクラス (sample) のメンバ関数である method1() でのメンバオブジェクトのアクセスです。

```
//サンプルメソッド
void sample::method1(long data){
  newvlab000->setProperty(WSNuserValue,data);
  ...
}
```

第4章　オンラインストア編

4.1　オンラインでアプリケーションウィンドウを読み込むには

4.1.1　オンラインでアプリケーションウィンドウを読み込むには

オンラインでアプリケーションウィンドウを読み込むには次のグローバル関数を使用します。

読み込み関数	機能
WSGFloadWindow(prm1,prm2,prm3,prm4)	ストアアプリケーションウィンドウの読み込み
char* prm1	ストア属性、 ”FILE” = FILE データの読み込み
char* prm2	ストアデータ名、 prm1 = ”FILE” ならば、ファイル名
WSCbase** prm3	アプリケーションウィンドウ返値ポインタ、 ロードしたアプリケーションウィンドウを返値
WSCbase* prm4	部分アプリケーションウィンドウの場合の親の指定
long 返値	WS_NO_ERR = 正常 / 他 = 異常

オンラインでアプリケーションウィンドウを読み込むには次の様に行います。

```
#include <WSCconductor.h>
...

  WSCbase* window   = NULL; //返り値（ロードしたアプリケーションウィンドウ）を格納
  char*    stype    = "FILE";           //FILE 指定
  char*    fname    = "newpic001.oof";  //FILE 名称
  char*    path     = "/usr1/win/data"; //DIR  指定

  //読み先ディレクトリを指定。
  WSGIconductor()->setSerializePath(path);

  //読み込み
  long ret = WSGFloadWindow(stype,fname,&window,NULL);
  if (ret == WS_NO_ERR){
    //読み込み成功!
    window->setVisible(True); //表示してみる。
  }
```

stype には、ストアタイプ名 (ストア属性) を指定します。現在 FILE が用意されています。将来的には、プロセス同士に共有したり、ネットワークでつながったりと、FILE 以外のフィールドを

拡張していく予定です。path には、読み込み先のディレクトリをコンダクタ・グローバルインスタンスに指定します。デフォルトはカレントディレクトリになっています。

fname には、ストアデータ名称 (ファイル名称) を指定して、WSGFloadWindow() 関数で読み込みます。読み込みが成功すると、アプリケーションウィンドウは使用可能となります。ロードの行為を複数回行うと複数生成されますので注意しましょう。

WSGFloadWindow() 関数がエラーになる場合は、ファイルの存在するディレクトリが正しいか、アクセス権は正しいかなどを確認してみて下さい。

4.1.2 オンラインで部分アプリケーションウィンドウを読み込むには

オンラインで部分アプリケーションウィンドウを読み込むには次の様に行います。

```
#include <WSCconductor.h>
...

  WSCbase* window   = NULL;  //返り値（ロードしたアプリケーションウィンドウ）を格納
  WSCbase* parent   = newwin000; //部分アプリケーションウィンドウを抱える親オブジェクト
  char*     stype   = "FILE";           //FILE 指定
  char*     fname   = "newpic001.oof";  //FILE 名称
  char*     path    = "/usr1/pic/data"; //DIR  指定

  //読み先ディレクトリを指定。
  WSGIconductor()->setSerializePath(path);

  //読み込み
  long ret = WSGFloadWindow(stype,fname,&window,parent);
  if (ret == WS_NO_ERR){
    //読み込み成功!
    window->setVisible(True); //表示してみる。
  }
```

stype には、ストアタイプ名 (ストア属性) を指定します。現在 FILE が用意されています。path には、読み込み先のディレクトリをコンダクタ・グローバルインスタンスに指定します。デフォルトはカレントディレクトリになっています。

fname には、ストアデータ名称 (ファイル名称) を指定し、parent には、部分アプリケーションウィンドウを配置する親を指定します。親には、アプリケーションウィンドウやエリア、スクロールエリアなど、オブジェクト配置機能のあるマネージャオブジェクトを指定してください。読み込みが成功すると、アプリケーションウィンドウは使用可能となります。ロードの行為を複数回行うと複数生成されますので注意しましょう。

WSGFloadWindow() 関数がエラーになる場合は、ファイルの存在するディレクトリが正しいか、アクセス権は正しいかなどを確認してみて下さい。

4.2 オンラインでアプリケーションウィンドウを破棄するには

4.2.1 オンラインでアプリケーションウィンドウを破棄するには

オンラインでアプリケーションウィンドウを破棄するには次の様に行います。

```
//破棄
WSGFdestroyWindow(window); //破棄したいアプリケーションウィンドウ
```

window には破棄したいアプリケーションウィンドウを指定します。二度破棄したり、使用中のアプリケーションウィンドウを破棄したりすると、修復不可能なメモリエラーとなるので注意しましょう。破棄した後は、そのアプリケーションウィンドウにアクセス出来ませんので、ポインタなどでアプリケーションウィンドウを覚えている場合などは注意してください。

4.2.2 オンラインで部分アプリケーションウィンドウを破棄するには

オンラインで部分アプリケーションウィンドウを破棄するには次の様に行います。

```
//破棄
WSGFdestroyWindow(object); //破棄したい部分アプリケーションウィンドウ
```

object には破棄したい部分アプリケーションウィンドウを指定します。二度破棄したり、使用中の部分アプリケーションウィンドウを破棄したりすると、修復不可能なメモリエラーとなるので注意しましょう。破棄した後は、その部分アプリケーションウィンドウにアクセス出来ませんので、ポインタなどで部分アプリケーションウィンドウを覚えている場合などはアクセスすることのないよう注意してください。

第5章　リモートインスタンス編

5.1　リモートインスタンスにアクセスするには

5.1.1　リモートインスタンスにアクセスするには

オブジェクト管理インスタンス (ロードモジュールにつき、一つ存在) に対して要求すると、アクセスしたいリモートインスタンスを取得することができます。

オブジェクト管理クラス	インスタンス取得関数
WSCbaseList	WSCbaseList* WSGIappObjectList()

アクセスしたいオブジェクトを取得は、次の様に行います。

```
#include <WSCbaseList.h> //WSGIappObjectList() にアクセスする...
#include <WSCRbase.h>    //仮想リモートインスタンスクラスを使用
...
void event_procedure(WSCbase* object){

  //オブジェクト管理による WSCRbase ポインタの取得
  char* obj_name   = "newvlab_001";     //newvlab_001 という名称のリモートインスタンス
  WSCRbase* rinstance = WSGIappObjectList()->getRemoteInstance(obj_name);

  //仮想リモートインスタンスによるリモートインスタンスに対するアクセス
  rinstance->setProperty(WSNlabelString,"HELLO WORLD");
```

rinstance がリモートインスタンスにアクセスするための仮想リモートインスタンスです。リモートインスタンスのオブジェクト名称を引数にします。通常のインスタンス (オブジェクト) にアクセスするのと同じように、仮想リモートインスタンスを通して、リモートインスタンスにアクセスします。

5.1.2　リモートインスタンスをキャストするには

オブジェクト管理を通して得られた仮想リモートインスタンスは通常のオブジェクトと同じようにオリジナルのクラスにキャストして使用することが出来ます。オリジナルクラスに存在するメソッドを利用する場合にキャストします。次の例は、WSClist::addItem() を呼び出すため、WSCRbase 型の仮想リモートインスタンスから、WSCRlist 型の仮想リモートインスタンスにキャストしています。

```
#include <WSCbaseList.h> //WSGIappObjectList() にアクセスする...
#include <WSCRlist.h>    //仮想リモートインスタンスクラスを使用
...
void event_procedure(WSCbase* object){

  //オブジェクト管理による WSCRbase ポインタの取得
  char* obj_name   = "newlist_001";     //newlist_001 という名称のリモートインスタ
ンス
  WSCRbase* rinstance = WSGIappObjectList()->getRemoteInstance(obj_name);

  //仮想リモートインスタンスをオリジナルのクラス WSClist に対応する
  //仮想リモートクラス WSCRlist にキャスト。
  WSCRlist* rlist = (WSCRlist*)rinstance->cast("WSCRlist");
  if (rlist == NULL){
    //WSCRlist クラスではない。
    return;
  }

  //WSClist クラスのメソッドを WSCRlist 仮想リモートインスタンスクラスを
  //通して呼び出す。
  rlist->addItem("item..");

}
```

第6章　サンプルプログラム編

6.1　サンプル1 (Hello World)

ここでは、一番最初にすべき、プロジェクトの作成、アプリケーションウィンドウの作成、イベントプロシージャの作成を行います。ウィンドウに表示されるボタンを押すと、HelloWorld と表示される、初歩的なサンプルです。

サンプルのソースコードは、ws/samples/EUCJP/hello/hello.prj, ws/samples/SJIS/hello/hello.prj, です。アプリケーションビルダで読み込んでコンパイルしてみてください。

- プロジェクトの作成

  ［プロジェクト］メニューの［新規プロジェクト］を選び、プロジェクト名を hello 、プロジェクト種別を［通常のアプリケーション］で作成します。

- アプリケーションウィンドウの作成

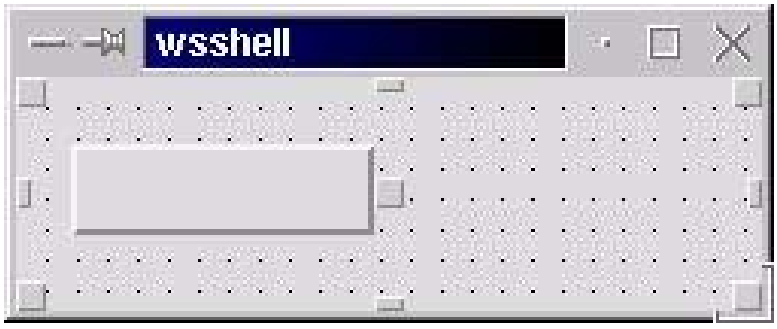
  ［ファイル］メニューの［新規ウィンドウ］を選び、通常のウィンドウ、プロジェクトに登録、newwin000 で作成します。

- オブジェクトの配置

  作成された、アプリケーションウィンドウに、ボタンオブジェクト (WSCvbtn) を配置します。［表示］メニューの［オブジェクト］を選び、オブジェクトボックスを表示します。

  オブジェクトボックスの［Commands］タブを選択し、左上に存在する、WSCvbtn オブジェクト (BTN アイコン) をドラッグして、アプリケーションウィンドウ上にドロップします。

  次に、プロパティ等を設定してアプリケーションウィンドウの体裁を整えます。



［アプリケーションウィンドウの様子］

- イベントプロシージャの作成

  配置されたボタンオブジェクトを選択して、［編集］メニューの［プロシージャ編集／プロシージャ新規作成］を選び、プロシージャを作成します。

ここでは、起動関数名に btn_proc としておきましょう。起動トリガは、WSEV_ACTIVATE を指定します。

次にひな型作成ボタンをクリックして、ひな型を作成します。

- イベントプロシージャの編集

  イベントプロシージャが作成できたら、次は、関数の記述です。イベントプロシージャ new_ep を選択した状態で、［編集］メニューの［プロシージャ編集／プロシージャ編集］を選び、エディタを立ち上げます。

  ここでは、1 回目のクリックで、ボタンに Hello World が表示され、2 回目のクリックで、アプリケーションが終了するプログラムを書いてみます。

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//-----------------------------------------------------------
//Function for the event procedure
//-----------------------------------------------------------
void btn_proc(WSCbase* object){
  //do something...
  static long cnt = 0;
  if (cnt == 0){
    object->setProperty(WSNlabelString,"Hello World.");
    cnt++;
  }else{
    exit(0);
  }
}
static WSCfunctionRegister  op("btn_proc",(void*)btn_proc);
```

- プロジェクトの保存

  ［プロジェクト］メニューの［プロジェクト保存］を選び、プロジェクトを保存します。

- プロジェクトのビルド

  ［ビルド］メニューの［ビルドオール］を選び、プロジェクトをビルドします。無事ビルドし終えたら、実行してみましょう。

［アプリケーション実行の様子］

6.2 サンプル 2 (いろいろな部品 1)

ここでは、いろいろな部品を使ってみます。

サンプルのソースコードは、ws/samples/sample/EUCJP/sample.prj, ws/samples/sample/SJIS/sample.prj です。アプリケーションビルダで読み込んでコンパイルしてみてください。

このサンプルでは、次のようなことをおこないます。

- サンプルダイアログの表示
- スライダによる値の指定
- テキストの入力
- コンボボックスによるテキストの入力または、テキストの選択
- オプションメニューによる値の選択
- リストオブジェクトでの文字列の表示

［アプリケーションウィンドウの様子］

- ダイアログの表示

  「Dialog」と表示されているボタンには、WSEV_ACTIVATE イベントで、btn1_ep() イベントプロシージャが張り付けてあります。このプロシージャは、ボタンが押下されると起動され、メッセージダイアログ (newmess_002) をポップアップさせ、ダイアログの返り値を判断して、ラベル (newvlab_001) に結果を表示します。

- スライダの値

  スライダ 1(newvsli_000) には、WSEV_VALUE_CH イベントで slider1_ep() イベントプロシージャが張り付けてあります。このプロシージャは、スライドされると起動され、スライダの値を、ラベル (newvlab_002) に表示します。

- テキスト入力

  「Input text」と表示されているボタンには、WSEV_ACTIVATE イベントで、btn2_ep() イベントプロシージャが張り付けてあります。このプロシージャは、ボタンが押下されると起動され、テキスト入力フィールド (newvifi_004) に入力されたテキストを、隣のラベル (newvlab_005) に表示します。

- スライダの値 2

  スライダ 2(newvsli_006) には、WSEV_VALUE_CH イベントで slider2_ep() イベントプロシージャが張り付けてあります。このプロシージャは、スライドされると起動され、スライダの値を、バー型メータ (newvmet_008) に表示します。

- コンボボックス

  コンボボックスによる、テキスト入力と、テキスト選択のデモンストレーションです。

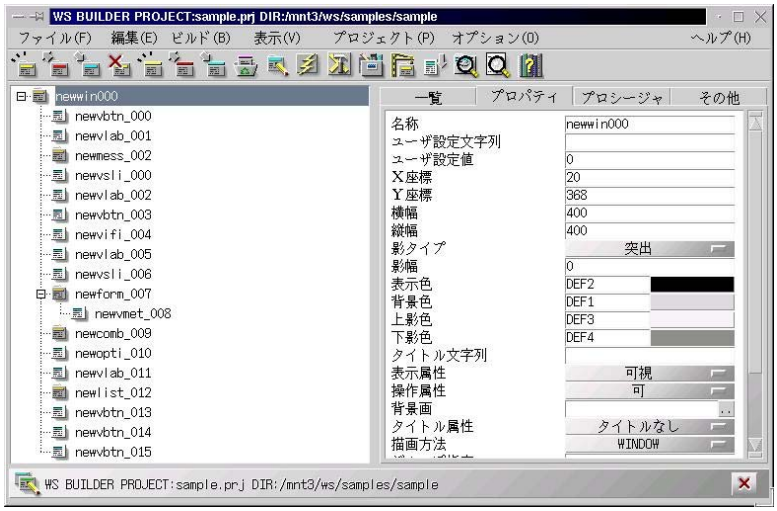
- オプションメニュー

  「Choose」と表示されたオプションメニューには、WSEV_VALUE_CH イベントで opt1_ep() イベントプロシージャが張り付けてあります。このプロシージャは、オプションメニューで値が選択された場合に起動され、隣のラベル (newvlab_011) にその選択された値を表示します。

- リスト

  「Add」と表示されているボタンには、WSEV_ACTIVATE イベントで、btn3_ep() イベントプロシージャが張り付けてあります。このプロシージャは、リスト (newlist_012) に文字列を一行付け加えます。

  「Clear」と表示されているボタンには、WSEV_ACTIVATE イベントで、btn4_ep() イベントプロシージャが張り付けてあります。このプロシージャは、リスト (newlist_012) に表示されている文字列を消去します。



［アプリケーションのオブジェクトの様子］

6.3 サンプル 3 (ラベル)

ラベルを使ったイベントプロシージャのデモンストレーションです。

サンプルのソースコードは、ws/samples/EUCJP/labelwork/labelwork.prj, ws/samples/SJIS/labelwork/labelwork.prj です。アプリケーションビルダで読み込んでコンパイルしてみてください。

このサンプルでは、次のようなことをおこないます。

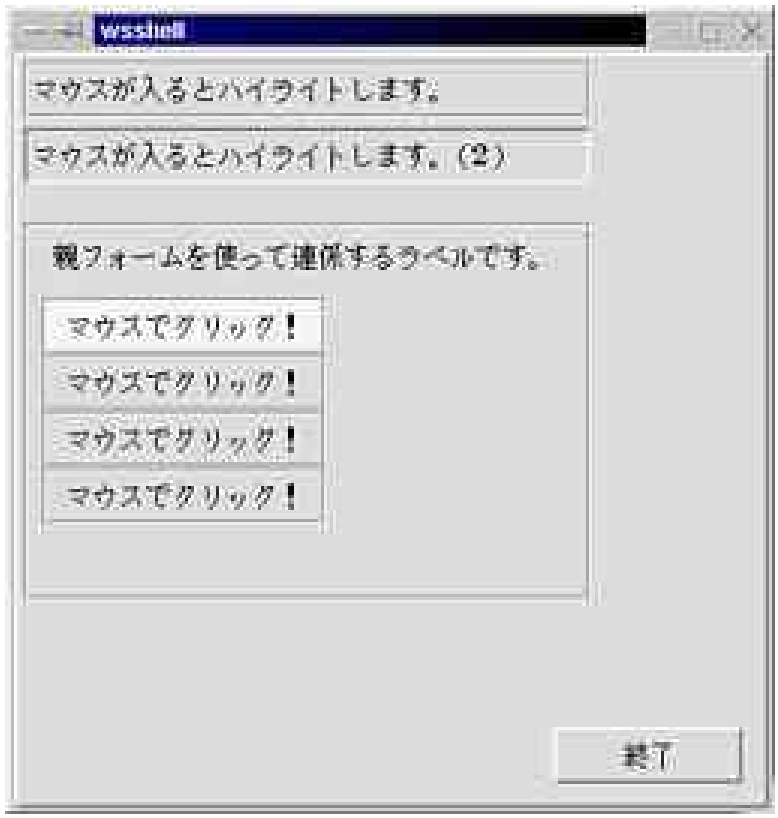
- ラベルのハイライト表示

  通常のイベントプロシージャによるもの。

- ラベルのハイライト表示 (一括設定)

  初期化イベントプロシージャによる一括設定のもの。

- 親フォームを使った連係



［アプリケーションウィンドウの様子］

- ラベルのハイライト表示

  「マウスが入るとハイライトします。」と表示されているラベルには、WSEV_MOUSE_IN イベントと、WSEV_MOUSE_OUT イベントでそれぞれ、イベントプロシージャが張り付けてあります。

  WSEV_MOUSE_IN イベントのプロシージャは、背景色プロパティWSNbackColor の値をプロパティ WSNuserString に退避して、ハイライト色に設定します。

  WSEV_MOUSE_OUT イベントのプロシージャは、退避しておいた背景色をプロパティ WSNback-Color に戻します。

- ラベルのハイライト表示 (一括設定)

  「マウスが入るとハイライトします。(2)」と表示されているラベルには、WSEV_MOUSE_IN イベントと、WSEV_MOUSE_OUT イベントで起動するイベントプロシージャを張り付けるイベントプロシージャを WSEV_INITIALIZE イベントで張り付けています。

  この場合のように、イベントプロシージャをまとめると、複数のイベントプロシージャが一括で設定されますので、たくさんのオブジェクトに張り付ける場合など、イベントプロシージャを張り付ける手間が省けます。

- 親フォームを使った連係

  「マウスでクリック！」と表示されているラベルには、WSEV_MOUSE_PRESS イベントで起動するイベントプロシージャがはりつけられています。

このイベントプロシージャは、親フォームに対して、選択されたオブジェクトを覚え込ませることで、複数のラベルが協調して、選択表示できるようにしています。

6.4 サンプル 4 (電卓)

ボタンの配列を使った電卓のデモンストレーションです。

サンプルのソースコードは、ws/samples/EUCJP/wscalc/wscalc.prj, ws/samples/SJIS/wscalc/wscalc.prj です。アプリケーションビルダで読み込んでコンパイルしてみてください。

このサンプルでは、次のようなことをおこないます。

- 四則演算電卓のデモンストレーション

  (注意) デモのため、動作仕様はあまり厳密ではありません。

［アプリケーションウィンドウの様子］

- 数字のボタン

  数字を表示しているラベルに数字を追加します。

- 演算子ボタン

  何の演算を行うかを数字を表示しているラベルに覚え込ませます。次の段階で「=」ボタン、演算子ボタンが押されたときに、その記憶しておいた演算情報で、演算し結果を表示します。

- その他

  リサイズイベントプロシージャは、サイズ変更されたと気に、数字ボタン、演算子ボタン、数字表示ラベル等のサイズを補正し、正しい位置に設定します。