Objective-C/C++の本

haseham

Objective-Cのソースコードを見てみよう

// インターフェイスファイル **MyClass1.h**

```
#import <Foundation/Foundation.h>


// クラス型の宣言
@interface MyClass1 : NSObject // 親クラスがない場合は、NSObject を継承する。
{
      // データメンバの宣言。
        int data_membar1;
        int data_member2;
}

    - (int)MyMethod1; // インスタンスメソッドの宣言。
    + (int)MyMethod2; // クラスメソッドの宣言。

@end
```

// 実装ファイル **MyClass1.m**

```
#import <MyClass1.h>

// クラス型の実装
@implementation MyClass1

    // -------------------
    // イニシャライザの実装
       //
    // ( このメソッドは、allocメソッドによって、
       //        インスタンスが確保された直後に呼び出される。)
    //
    - (id)init
    {

        self = [super init]; // 親クラスのイニシャライザを呼び出す。
                             // ( 戻り値は、インスタンス自身。)


        if ( self ) // 親クラスの初期化処理が失敗したら、nil が返される。
        {
            // メンバの初期化処理を、ここに書く。
        }

        return self; // そのまま返す。( これを子クラスが受け取る。)
    }

    // -------------------
    // デアロケータの実装
       //
       // ( このメソッドは、releaseメソッドによって、
       //      インスタンスが解放される直前に呼び出される。 )

    - (void)dealloc
    {
        // メンバの解放処理を、ここに書く。

        [super dealloc]; // 親クラスの解放処理を呼び出す。
    }

    // -------------------

    - (int)MyMethod1 // インスタンスメソッドの実装。
    {

    }

    // -------------------
```

```
    + (int)MyMethod2 // クラスメソッドの実装。
    {

    }

    // -------------------

@end
```

// 実装ファイル **main.m**

```
#import <Foundation/Foundation.h>
#import <MyClass1.h>


int main()
{

  @autorereasepool
  {

     // インスタンスを生成し、イニシャライザを呼ぶ。
     id obj1 = [[MyClass1 alloc]init];

     [obj1 MyMethod1];      // インスタンスメソッドを呼ぶ。
     [MyClass1 MyMethod1]; // クラスメソッドを呼ぶ。

  }

  return 0;
}
```

メソッドの宣言

【メソッド宣言の書式】

- (戻り値)メソッド名;

- (戻り値)メソッド名:(引数1のデータ型)引数1の名前;

- (戻り値)メソッド名:(引数1のデータ型)引数1の名前
　　引数2のラベル:(引数2のデータ型)引数2の名前;

・1文字目が、「-」のものは、「インスタンスメソッド」。
・1文字目が、「+」のものは、「クラスメソッド」。

・引数1にはラベルがありませんが、
　これは、メソッド名の最後に付ける慣習があるからです。↓

```
- (int)getValueByIndex:(int) index_
                Option:(int) option_;
```

・ラベルは、メソッドを呼び出す際にも書き添える必要があり、
　プログラマは、これを読むことによって、
　それぞれの引数の意味を確認することができるのです。

メソッドの呼び出し

【 メソッドの呼び出しの書式 】

```
[レシーバー メソッド名];

[レシーバー メソッド名:引数1];

[レシーバー メソッド名:引数1
        引数2のラベル:引数2];
```

・「レシーバー」というのは、インスタンスメソッドを
　持っているクラスのインスタンスのことです。

・クラスメソッドの場合は、クラス名を指定します。

・また、レシーバーには、メソッド呼び出しを使うこともできます。↓
(この場合は、戻り値がレシーバーになる。)

```
[[レシーバー　メソッド名] メソッド名];
```

・インスタンス自身へのポインタは、「**self**」です。↓

```
[self MyMethod1];
```

・親クラスへのポインタは、「**super**」です。↓

[super MyMethod1];

・このように、**super**を書いた場合は、
親クラスのメソッドを、子クラス側でオーバライド (上書き) していても、
親クラスのメソッドが呼び出されます。

インスタンスの確保と解放 (alloc release)

```
// インスタンスの確保と解放。↓
MyClass1* p_obj1 = [[MyClass1 alloc] init]; // 確保
[release p_obj1]; // 解放
```

・1行目の alloc メソッドは、
「MyClass1」クラスのインスタンスを生成してから、
「MyClass1」クラスの init メソッド（イニシャライザ）を呼んでいます。

・「イニシャライザ」は、インスタンス内のメンバの初期化を行うメソッドで、
C++でいうところのコンストラクタの役割を果たします。

・2行目の release メソッドは、
「MyClass1」クラスの dealloc メソッド（デアロケータ）を呼んでいます。

・「デアロケータ」は、インスタンス内のメンバの解放を行うメソッドで、
C++でいうところのデストラクタの役割を果たします。

【 インスタンス変数の正体 】

・「Objective-C」では、クラスのインスタンスは、
すべて「スマートポインタ」になっています。

・「スマートポインタ」というのは、
中にインスタンスへのポインタを入れておけば、
使う人がいなくなった時点で、
自動的に解放してくれるというアレです。

・Objective-Cのインスタンス変数は、
インスタンス化された時点では、
内部の参照カウンタが、1 になっています。

・そして、このインスタンス変数を利用している各インスタンスが
ぞれぞれリリースして行き、カウントが 0 になると、
内部で管理している実体が、自動的に解放されます。

・Objective-Cの参照カウントは、「レテインカウント」といい、
retainCountメソッドを呼び出すことで、現在値を取得することができます。↓

```
NSLog( @"p_obj1のレテインカウントは、%d です。", [p_obj1 retainCount] );
```

・レテインカウントは、retainメソッドを呼び出す度に、
1ずつ カウントアップされていきます。↓

```
[p_obj retain]; // これでカウントが +1 された。
```

・つまり、deallocメソッドが呼ばれるタイミングは、
retainメソッドを呼び出した回数 + 1回 だけ、
releaseメソッドが呼ばれた時です。

・ちなみに、「+1回」というのは、インスタンスの生成時に、
alloc メソッドによってカウントアップされたものです。

・retain メソッドを呼び出すタイミングとしては、
別のクラスにも、p_obj を持たせるために、
そのイニシャライザなどへ、 p_obj を渡す時です。

・そして、release メソッドを呼ぶタイミングは、
そのクラスのメンバが解放される時で、
これは、デアロケータの中などです。

・このライフサイクルを、C++風に書くと、
おおよそ、以下のようになります。↓

```
 // -------------------------
// 他のクラスに、メンバとして持たせるクラスの宣言。↓

class Class1{ /* 中略 */ };

// -------------------------
// Class1のインスタンスを生成するクラスの宣言↓

class Class2
{
   Class1* p_obj1;

   Class2()
   {
       p_obj1 = new Class1();
       // 参照カウントが 1 になる。この1は、作成者自身の持ち分です。 )
    }

 ~Class2()
  {
       // delete p_obj1; とはせず、参照カウントを減らす。( release )
  }
};

// -------------------------
// さらにそれを借り受けて共有するクラスの宣言。↓

 class Class3
{
   Class1* p_obj1; // 他のクラスが生成したインスタンス。

   Class3( Class1* p_obj_ )
   {
       p_obj1 = p_obj_;
      // この時に、参照カウントを増やす。( retain )
   }

  ~Class3()
  {
       // ここで参照カウントを減らす。( release )
  }
};
```

自動解放プール (autorelease)

```
#import <Foundation/Foundation.h>

int main()
{
    // 自動解放プールを生成する。
    NSAutoreleasePool* p_pool1 = [[NSAutoreleasePool alloc] init];

    MyClass1* p_obj1 = [[MyClass1 alloc] init]; // インスタンスを生成する。

    [p_obj1  autorelease]; // 自動解放プールに登録しておく。
    // ( ※ 最後に生成されたプールに登録される。)

    [p_pool1 release]; // 自動解放プールを解放する。( p_obj1 も 解放される )

     return 0;
}
```

・インスタンス「p_obj1」を、他のクラスと共有する場合は、
受け渡しの際に、retain メソッドを呼び出して、
レテインカウントをカウントアップさせないといけません。

・Foundationフレームワークのメソッドでも、
autorelease メソッドが呼ばれたインスタンスが返されるようになっています。

(※ copy メソッドや、mutableCopy メソッドで複製されたインスタンスは、
autorelease メソッドが呼ばれていません。)

・プールを生成するのは面倒なので、ふつうは、次のように書きます。↓

```
#import <Foundation/Foundation.h>

int main()
{

  @autoreleasepool
  {
     // ここにコードを書く。
  }

  return 0;
```

```
}
```

ガベージコレクション (finalize drain)

・OSX10.5以降 (Objective-C 2.0) からは、
「ガベージコレクション」が導入されました。

・Java や C# では、インスタンスの解放は、
ガベージコレクションによって、自動的に行われます。

・つまり、delete演算子や releaseメソッドなどを呼ぶ必要がないのです。

・ガベージコレクションを有効にしている場合は、
release メソッド や、autorelease メソッドを呼び出しても、
何もされず、デアロケータも呼ばれません。

・代わりに、解放時には、finalize メソッドが呼ばれます。

(※ finalize メソッドは、いつ呼ばれるか不明なので、
順序や、タイミングに左右されるような処理を書いてはいけません。

※ また、処理は、 スレッドセーフである必要があります。)

・finalize メソッドは、以下のように実装します。↓

```
- (void)finalize
{
    // このクラスのメンバを解放する。

    [super finalize]; // 最後に、親クラスのfinalizeメソッドを呼ぶ。
}
```

・ガベージコレクションを有効にしてビルドしたアプリでは、
自動解放プールを解放する時は、**drain**メソッドを呼びます。↓

```
[p_pool1 drain]; // 自動解放プールを解放する。( p_obj1 も 解放される。)
```

・**drain**メソッドは、ガベージコレクションが無効である場合には、
releaseメソッドと同じ処理を行い、
有効である場合には、不要なインスタンスを探して、自動的に解放します。

・Cocoaアプリケーションでは、ガベージコレクションは有効になっていますが、
コマンドラインアプリケーションで有効にするには、次のように書く必要があります。↓

```objc
#import <Foundation/Foundation.h>

int main()
{
    NSGarbageCollector* p_gc = [NSGarbageCollector defaultCollector];
    [p_gc enable]; // GCを有効にする。
    // 中略
    MyClass1* p_obj1 = [[MyClass1 alloc]init]; // インスタンスを生成する。
    p_obj1 = nil; // nilを代入すると、もう使われなくなったとみなされる。
    [p_gc collectIfNeeded]; // 必要なら解放する。( 重くなりそうな時に呼ぶ。)
    [p_gc collectExhaustively]; // 確実にすべて解放する。( アプリの終了直前に呼ぶ。)
    return 0;
}
```

・p_obj1 は、collectIfNeededメソッドが呼ばれた時点で解放されます。

ブロック構文

```
// クラスの実装

@implementatin MyClass1

- (void)MyMethod1
{
    int (^tasizan)(int,int); // ブロック構文変数を定義する。

   // ブロック構文変数を実装する。
   tasizan = ^( int v1, int v2 )
             {
                 return v1+v2;
             };

    // コマンドラインに、戻り値を出力する。
    NSLog( @"足し算の答えは、%d です。", tasizan );
}

@end
```

・さて、次のメソッドでは、引数がブロック構文になっています。↓

```
- (void)OutputAnswer : (   int (^formula)(int,int)    ) formula_
 {
    // コマンドラインに、戻り値を出力する。
    NSLog( @"数式の答えは、%d です。", formula_ );}
```

```
}
```

・このメソッドを呼び出す場合は、こうなります。↓

```
#import "Foundation/Foundation.h"

int main()
{

   MyClass1* p_obj1 = [[MyClass1 alloc]init]: // インスタンスを生成する。

   // 計算式を入力し、その結果を出力する、というメソッドを呼ぶ。
   [p_obj1 OutputAnswer:^( int v1, int v2 ){return v1+v2;} ];

  [p_obj1 release]; // インスタンスを解放する。

   return 0;
}
```

プロパティ

・**C#**には、「プロパティ」という便利なものがありますが、
Objective-Cのプロパティは、少し違っていて、
C++のアクセッサを自動的に実装してくれるものです。↓

```
// ヘッダファイル側　クラスの宣言

@interface MyClass1
{
   int data_member1; // データメンバの宣言。
}

 @property int MyProperty1; // プロパティの宣言。

@end
```

```
// ソースファイル側　クラスの実装

@implementation MyClass1

 @synthesize MyProperty1 = data_member1; // プロパティの実装は書かない。


@end
```

・コンパイラは、上記のソースコードに基づいて、
下記の**Getter**メソッドと**Setter**メソッドを実装してくれます。↓

```
- (int)MyProperty1;
- (void)setMyProperty1(int)value;
```

・これらの実装を、C/C++ で書くと、こんな感じになります。↓

```
int   MyProperty1(void) const { return data_member1; }
void setMyProperty1( int value ) { data_member1 = value; }
```

・それでは、このプロパティを使ってみましょう。↓

```
#import<Foundation/Foundation.h>

int main()
{

 MyClass1* p_obj1 = [[MyClass1 alloc]init]; // インスタンスを生成する。

 int v1= [p_obj1 MyProperty1]; // Getterメソッドを呼び出す。

 [p_obj1 setMyProperty1:9999]; // Setterメソッドを呼び出す。

 return 0;

}
```

・アクセッサは、片方、または両方を、独自に実装することもできます。

・それから、プロパティには、「属性」を付けることができ、
これによって、実装の内容が変化します。↓

```
getter=Getterの名前
setter=Setterの名前
readonly    ... 読み取り専用。
readwrite   ... 読み書き可能。
nonatomic   ... 複数スレッド間での同期処理を行わない。( ロック処理を行わない )
```

・値の持ち方についての属性は、次のいずれかを指定します。↓

---

assign　... 単純に、値を代入するだけ。基本データ型向け。(デフォルトではこの動作。)

retain　... インスタンスをセットし直す際に、
元のインスタンスの**rerease**メソッドを呼び、
渡されたインスタンスの**retain**メソッドを呼んで、
インスタンスをセットする。

copy　... インスタンスをセットし直す際に、
元のインスタンスの**rerease**メソッドを呼び、
渡されたインスタンスの**copy**メソッドを呼んで、
作成されたコピーをセットする。

・プロパティ属性は、次のようにして指定します。↓

```
@property(readonly,retain) int MyProperty1; // プロパティの宣言。
```

カテゴリ

・「カテゴリ」は、同じ種類のメソッドをグループ化して
わかりやすくまとめるためのものです。

・例えば、次のようなシンプルなクラスがあるとします。↓

```
// 画像を表示するクラスの宣言 (Jpegのみ対応) ... MyImage.h
@interface MyImage
{
}
- (int)LoadJpeg(NNString*) path_;
- (int)SaveJpeg(NNString*) path_;
@end
```

```
// 画像を表示するクラスの実装 (Jpegのみ対応) ... MyImage.mm
@implimentation MyImage
- (int)LoadJpeg(NNString*) path_ {}
- (int)SaveJpeg(NNString*) path_ {}
@end
```

・このクラスでは、今のところ、**Jpeg**画像しか表示できませんが、
近いうちに、**Gif**など、その他のファイル形式の画像も表示させたい、
と考えているとします。

・そういう場合には、このクラスを継承したり、包含したりして、
機能を拡張していけばいいのですが、いまひとつ柔軟性に欠けることがあり、
そんな場合には、このカテゴリを使うという手もあります。↓

```
// 画像を表示するカテゴリの宣言 (Gif対応) ... MyImageClass+Gif.h
```

```
@interface MyImage(Gif)
{
}
- (int)LoadGif(NNString*) path_;
- (int)SaveGif(NNString*) path_;
@end
```

```
// 画像を表示するカテゴリの実装 (Gifのみ対応) ... MyImage+Gif.mm
@implimentation MyImage(Gif)
- (int)LoadGif(NNString*) path_ {}
- (int)SaveGif(NNString*) path_ {}
@end
```

・クラス名は、「MyImage」クラスと同じですが、↑
　クラス名の直後に、カッコで囲んだ中に、
　カテゴリ名が書きそえてあります。

・カテゴリで追加できるのは、メソッドだけです。

【 カテゴリの注意点 】

　・インスタンス変数を宣言できない。
　・プロパティも宣言できない。
　・元クラスのプライベートメソッドと同じ名前を再定義してはいけない。

・カテゴリで追加されたメソッドは、元クラスのメソッドと同じように
　元クラスのインスタンスから呼び出すことができます。↓

```
#import <Foundation/Foundation.h>
#import <MyImage.h>
#import <MyImage+Gif.h>

int main()
{

  MyImage* p_img1 = [[MyImage alloc] init]; // 元クラスのインスタンスを生成する。

  [p_img1 LoadGif:@"c:¥ryuenbu.gif"]; // Gifカテゴリのメソッドが呼び出せる。

  [p_img1 release]; // インスタンスを解放する。

  return 0;
}
```

・何か使いたいカテゴリがある場合は、元クラスのヘッダファイルと一緒に、
そのカテゴリのヘッダファイルをインクルードしておきます。

・すると、そのカテゴリに含まれるメソッドを呼び出すことができます。

・カテゴリは、**NSString** など、システムフレームワークのクラスに
機能を追加して使う場合などで、よく使われているようです。

クラスエクステンション

・「クラスエクステンション」は、名前のないカテゴリのようなものです。

・ソースファイルの外には公開されないため、宣言は書きません。↓

```
// MyImage.h

 // 画像を表示するクラスの宣言 (Jpegのみ対応)
@interface MyImage
{
}
- (int)LoadJpeg(NNString*) path_;
- (int)SaveJpeg(NNString*) path_;
@end
```

```
 // MyImage.mm

// 画像を表示するクラスエクステンションの宣言 (Gif対応)
@implimentation MyImage()
- (int)LoadGif(NNString*) path_ {}
- (int)SaveGif(NNString*) path_ {}
@end

 // 画像を表示するクラスの実装 (Jpegのみ対応)
@implimentation MyImage
- (int)LoadJpeg(NNString*) path_ {}
- (int)SaveJpeg(NNString*) path_ {}
@end
```

・カテゴリでは、システムフレームワークのクラスを

機能拡張することができましたが、
クラスエクステンションではできません。

・その代わりに、インスタンス変数やプロパティを
　メンバとして宣言することができます。

プロトコル

・「プロトコル」は、**C/C++**でいうところの
「インターフェイス」のようなものです。(__interface)

・プロトコルの宣言では、
抽象クラスのように、メソッドの宣言だけを行い、
クラス側で実装をさせます。↓

```
// プロトコルの宣言。

@protocol MyProtocol1
// メソッドの宣言。
- (int)Method1;
- (int)Method2;
@end
```

・さらに、別のプロトコルのメソッド宣言を含めることもでき、
その場合は、**<>**の中に、含めたいプロトコル名を、カンマで区切って書きます。↓

```
// プロトコルの宣言。

@protocol MyProtocol3< MyProtocol1, MyProtocol2 >
   // 中略
@end
```

```
// MyProtocol1 を実装するクラスの宣言。↓
@interface MyClass1<MyProtocol1>
{
}
@end
```

```
// MyProtocol1 を実装するクラスの実装。↓
@implementation MyClass1<MyProtocol1>
// メソッドの実装。
- (int)Method1{}
- (int)Method2{}
@end
```

・複数のプロトコルを実装する場合は、
　カンマで区切って書きます。

・カテゴリで実装する場合は、次のように書きます。↓

```
// MyProtocol1 を実装するカテゴリの宣言。↓
@interface MyClass1(Category1)<MyProtocol1>
{
}
@end
```

・実装側にも、同じように書きます。

・使い方は、**C/C++**のインターフェイスと同じです。↓

```
#import <Foundation/Foundation.h>
```

```
int main()
{

  // MySoundWAVクラスのインスタンスを生成し、
  // そのアドレスを、MySound プロトコルのポインタに代入する。
  //
  MySound* p_snd1 = [[MySoundWAV alloc]init];
  [p_snd1 play];    // WAVEファイルが再生される。
  [p_snd1 release]; // インスタンスを解放する。



  // MySoundMP3クラスのインスタンスを生成し、
  // そのアドレスを、MySound プロトコルのポインタに代入する。
  //
  p_snd1 = [[MySoundMP3 alloc]init];
  [p_snd1 play];    // MP3ファイルが再生される。
  [p_snd1 release]; // インスタンスを解放する。

  return 0;
}
```

・MySoundWAVクラスと、MySoundMP3クラスは、
どちらも同じMySoundプロトコルを実装していますが、
同じ名前のplayメソッドでも、実装が異なっており、
ファイル形式に適した再生処理を実行することができます。

プリプロセッサ (**#import**)

・**Objective-C**では、「#include」の代わりに
「#import」を使います。

・#import は、同じヘッダファイルを一度しか読み込まないため、
インクルードガードをする必要がありません。

例外処理 (@try @catch @finally)

```
@try
{
   // 例外が発生しそうな処理を、ここに書く。
}
@catch( NSException* ex )
{
   // 例外が発生した時の処理を、ここに書く。
}
@finally
{
   // 例外の有無に関係なく、必ず実行したい処理を、ここに書く。
}
```

・例外をスローするには、raiseメソッドを呼びます。↓

```
[[NSException exceptionWithName:@"例外名"
                         reason:@"例外の原因"
                       userInfo:@"説明"]raise];
```

---

id型

・id 型の変数は、あらゆるデータ型の値を代入することができます。

```
// Class1.h

// クラス1の宣言
@interface Class1
// 中略
- (void) output1();
@end
```

```
// Class2.h

// クラス2の宣言
@interface Class2
// 中略
- (void) output2();
@end
```

```
// Class1.m

#import <Foundation/Foundation.h>
#import <Class1.h>

// クラス1の実装
@implementation Class1
// 中略
- (void) output1() { NSLog(@"Class1"); }
@end
```

```
// Class2.m

#import <Foundation/Foundation.h>
#import <Class2.h>

// クラス2の実装
@implementation Class2
// 中略
- (void) output2() { NSLog(@"Class2"); }
@end
```

```
// main.m

#import <Foundation/Foundation.h>
#import <Class1.h>
#import <Class2.h>

int main()
{

   id obj1 = [[Class1 alloc]init]; // インスタンスを生成する。
   id obj2 = [[Class2 alloc]init];// インスタンスを生成する。

  [obj1 output1]; // 「Class1」と出力される。
  [obj2 output2]; // 「Class2」と出力される。

  [obj1 release]; // インスタンスを解放する。
  [obj2 release]; // インスタンスを解放する。

  return 0;
}
```

・ポインタ変数に代入されるアドレスのデータ型が定まっていない場合は、
C/C++では、**void*** 型のポインタを使ったり、
または、親クラスや、インターフェイスのポインタを使って
インスタンスのアドレスを受け取っていました。

・**void***型のポインタに代入した場合は、代入されたインスタンスが、
どのデータ型なのかといった情報が記録されていないため、
そのメソッドを呼び出すことができません。

・その点、親クラスや、インターフェイスのポインタであれば、
メソッドを呼び出すことはできますが、
対応していないデータ型のアドレスを代入することはできません。

・**id**型の変数では、いずれの心配もありません。

・加えて、これはオブジェクト型であるため、
release メソッドや **autorelease** メソッドなどが使えます。

文字列リテラル (**NSString**)

```
// ・文字列リテラルクラスのポインタを、
//    文字列リテラルのアドレスで初期化している。↓

NSString* p_s1 = @"あいうえお";

// ・このように、リテラルの頭に@を付けると、
//  NSStringクラスのインスタンスとみなされる。
```

・全角文字が使えるのは、ソースファイルのエンコードが**UTF-8**の時だけで、
それ以外の場合は、**ASCII**のみとなります。

・**NSString**クラスは、リテラル(定数)専用ですから、
文字列を部分的に修正することはできません。

・修正したい場合は、派生クラスの **NSMutableString**クラスを使います。↓

```
// 文字列変数を、文字列リテラルで初期化する。
NSMutableString* p_ms1 = [NSMutableString stringWithString:@"コアラ"];

// 文字列バッファを作成する。( バッファサイズは、引数で指定する。)
NSMutableString* p_ms2 = [NSMutableString stringWithCapacity:24];

// 文字列変数を、書式文字列で初期化する。
NSMutableString* p_ms3
  = [NSMutableString initWithFormat:@"v1=%d, v2=%d", v1, v2];
```

```
NSLog( @"%@", p_ms3 ); // コマンドラインへ出力する。
```

```
// 文字列リテラルを、文字列リテラルで初期化する。( UTF-8エンコーディング )
NSString* p_s1 = [NSString stringWithUTF8String:@"コアラ"];
```

```
NSString* p_s1 = @"あいうえお"; // 文字列リテラルを宣言する。

// 文字列変数に変換する。
NSMutableString* p_ms1 = [[p_s1 mutableCopy] autorelease];

// 逆に、文字列リテラルに変換する。
NSString* p_s2 = [NSString stringWithString:p_ms1];
```

ポインタ配列

```
#import <Foundation/Foundation.h>

int main()
{
    // --------------------
    // ポインタ配列を作成する。

    // 弱い参照関係のポインタ配列を作成する。
    NSPointerArray* p_a1 =
        [NSPointerArray pointerArrayWithWeakObjects];

    // 強い参照関係のポインタ配列を作成する。( nil を代入できる。)
    NSPointerArray* p_a2 =
        [NSPointerArray pointerArrayWithStrongObjects];

    // --------------------
    // 要素を追加する。

     // 要素となるインスタンスを生成する。
    NSString* p_item1 = [NSString stringWithString:@"りんご"];
    NSString* p_item2 = [NSString stringWithString:@"みかん"];

    // ポインタ配列に、そのポインタを追加する。
     [p_a1 addPointer:p_item1]; // 0:りんご
     [p_a1 addPointer:p_item2]; // 0:りんご  1:みかん
      // ( 要素は、配列の末尾に追加される。この場合は、要素0、1。 )

    // --------------------
    // 要素を挿入する。

      NSString* p_item3 = [NSString stringWithString:@"バナナ"];

    [p_a1 insertPointer:p_item3
                        atIndex:0]; // 0番目に挿入する。( 0:バナナ  1:りんご  2:みかん )
     // --------------------
    // 要素を削除する。

     [p_a1 removePointerAtIndex:0]; // 0番目を削除する。

     [p_a1 compact]; // nil 要素を削除する。

     // --------------------
    // 要素を置き換える。
    [p_a2 replacePointerAtIndex:0
                        withPointer:nil]; // 0番目を、nilに置換 する。
     // --------------------
```

```
    // 要素を取得する。

    NSString* p_item3 = [p_a1 pointerAtIndex:0]; // 0番目を取得する。

    // ちなみに、要素の列挙は、次のようにして foreach文 風に行える。↓
    NSString* p_i = nil;
    for ( p_i in p_a1 )
    {
        NSLog( @"%@", p_i ); // コマンドラインに出力する。
    }
    // ・NSFastEnumerationプロトコルを実装しているため、高速に列挙できる。
    //   (※ nil が格納されていることもあることに注意して、読み取って下さい。)
    // ------------------
    // 通常の配列を作成する。

    NSArray* p_a3= [p_a1 allObjects]; // 配列を作成する。
    // ( ポインタ配列が弱い参照関係のものでも、強い参照関係の配列が作成される。)
    // ( ※オブジェクトへのポインタ以外が格納されている場合は、作成できない。)
    // ------------------
    // 要素数を取得する。

    int item_count = [p_a1 count]; // 要素数を取得する。
    // ------------------

    return 0;
}
```

ハッシュテーブル

```
#import <Foundation/Foundation.h>

int main()
{
    // --------------------
    // ポインタ配列を作成する。

    // 弱い参照関係のポインタ配列を作成する。
    NSPointerArray* p_h1 =
        [NSPointerArray pointerArrayWithWeakObjects];

    // --------------------
    // 要素を追加する。

     // 要素となるインスタンスを生成する。
    NSString* p_item1 = [NSString stringWithString:@"りんご"];
    NSString* p_item2 = [NSString stringWithString:@"みかん"];

    // ポインタ配列に、そのポインタを追加する。
     [p_h1 addObject:p_item1];
     [p_h1 addObject:p_item2];

     // --------------------
    // 要素を削除する。

    [p_h1 removeObject:@"みかん"]; // 「みかん」を削除する。

    [p_h1 removeAllObjects]; // すべての要素を削除する。

    // --------------------
    // 要素数を取得する。

     int item_count = [p_h1 count]; // 要素数を取得する。

    // --------------------
    // 要素が存在するかを判定する。

     BOOL is_entry = [p_h1 containsObject:@"みかん"]; // 要素数を取得する。

     // --------------------
```

```
    // ハッシュテーブルを結合する。

     NSPointerArray* p_h2 =
       [NSPointerArray pointerArrayWithWeakObjects];

     [p_h1 unionHashTable:p_h2]; // p_h2 を削除する。

    // --------------------
    // 別のハッシュテーブルに含まれる要素を削除する。

     [p_h1 minusHashTable:p_h2]; // p_h2 の要素を削除する。

     // --------------------
     // 要素を列挙する。

    for ( NSString* p_i in p_h1 )
    {
        NSLog( @"%@", p_i ); // コマンドラインに出力する。
    }
    // ・NSFastEnumerationプロトコルを実装しているため、高速に列挙できる。
    // --------------------
    // すべての要素をコピーする。

     NSArray* p_a1 = [p_h1 allObjects]; // 配列にコピーする。
     NSSet* p_s1 = [p_h1 setRepresentation]; // セットにコピーする。

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

マップテーブル

```
#import <Foundation/Foundation.h>

int main()
{
    // --------------------
    // ポインタ配列を作成する。

    // 弱い参照関係のポインタ配列を作成する。
    NSMapTable* p_m1 =
        [NSMapTable mapTableWithStringToWithWeakObjects];

    // --------------------
    // 要素を追加する。

     // 要素となるインスタンスを生成する。
    NSString* p_item1 = [NSString stringWithString:@"りんご"];
    NSString* p_item2 = [NSString stringWithString:@"みかん"];

    // キー文字列を添えて追加する。
    [p_m1 setObject:p_item1
                forKey:@"青森"];

    [p_m1 setObject:p_item2
                 forKey:@"和歌山"];

    // ( 登録済みのキーを指定した場合は、上書きされる。)

     // --------------------
    // 要素を削除する。

    [p_m1 removeObjectForKey:@"和歌山"]; // 「みかん」を削除する。

    [p_m1 removeAllObjects]; // すべての要素を削除する。

    // --------------------
    // 要素数を取得する。

     int item_count = [p_m1 count]; // 要素数を取得する。

    // --------------------
```

```
    // キーを指定して、要素を取得する。

     NSString* p_item4= [p_m1 objectForKey:@"青森"]; // 要素を取得する。

    // ------------------
    // キー ( + 要素 ) を列挙する。

   NSEnumerator* p_keys1 = [p_m1 keyEnumerator]; //

   for ( NSString* p_key in p_keys1 )
   {
       // コマンドラインに出力する。
        NSLog( @" key=%@ : value=%@ ",
              p_key, [p_m1 objectForKey:p_key] );
   }
   // ------------------
   // 要素を列挙する。

   NSEnumerator* p_objs1 = [p_m1 objectEnumerator];

   for ( NSString* p_obj in p_objs1 )
   {
       // コマンドラインに出力する。
        NSLog( @"value=%@ ",
              p_key, [p_m1 objectForKey:p_obj] );
   }

   // ------------------
    // ディクショナリーを作成する。( 強い参照関係となるため、nilは許容されない。)

     NSDictionary* p_d1= [p_m1 dictionaryRepresentation]; // 作成する。
    // ------------------

   return 0;
}
```
