Boostのスマートなポインタを使ってみる

ゼグラム

自己紹介

- HM：ゼグラム（本名：清水 政宏、年齢：４４歳）
- お仕事：北九州市小倉でシステム開発
  ４月以降はたぶん福岡市でお仕事
- 趣味：ガジェット(オモチャ)探し・集め
- 一番長く使った言語：C/C++
- あまり好きではない言語：C/C++
- 好きな言語：Ruby

C/C++の変数とC#の変数の違い
インスタンスがスタック上に確保される値型①

- C#の場合
  - int, double, char, byteなどstring以外の基本型とstruct定義したユーザ型

```
struct UserData {
    public int p1;
    public double p2;
}
int a = 10;
UserData b;    // structなのでインスタンスはスタック
```

C/C++の変数とC#の変数の違い
インスタンスがスタック上に確保される値型②

- C/C++の場合
  - 基本的に全てのローカル変数のインスタンスは、スタック上に確保される

```
class UserData {
public:
    int p1;
    double p2;
};
int a = 10;
UserData b;    // クラスでもインスタンスはスタック
```

C/C++の変数とC#の変数の違い
ヒープ上に確保される変数①

- C#の場合(参照型)
  - stringや、classで定義したユーザ型はヒープ上に作成される

```
class UserData {
    public int p1;
    public double p2;
}
// newでインスタンスをヒープ上に確保し
// スタック上の参照変数にアドレスを代入
UserData a = new UserData();
```

C/C++の変数とC#の変数の違い
ヒープ上に確保される変数②

- C++の場合(ポインタ型)
  - C++言語の場合はstructやclassに関係なく全ての型をnewでヒープ上から確保しポインタ変数に確保したアドレスを入れることで、C#の参照型と同じように使用する

```
struct UserData {
    int p1;
    double p2;
};
UserData* a = new UserData();
```

多態性の実現に対して(C#)

- C#の値型変数では、多態性を実現することはできない。
- C#で多態性を実現するには、参照型変数を使用しなければならない。

多態性の実現に対して(C++)

- C++でも値型ある通常変数では、多態性は実現できない。
- C++で多態性を実現するには、ポインタ変数を使用しなければならない。

C++にはGCが無い

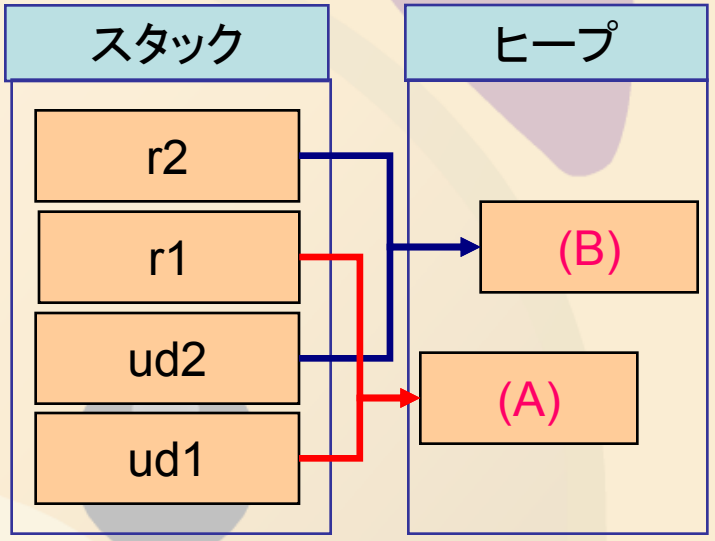
- 昔ながらのC言語との互換性を重視しているC++/Objective-Cには、自動的にメモリ管理を行ってくれるGC(ガベージコレクション)が存在しない。
- C++のnewでヒープから確保したメモリに対しての管理はプログラマが自力で行わなければならない。
- オブジェクト使用言語としてGCが無いのは不便でしょうがない。

基本手的なGCの動作

- GCの内部処理の方式には色々な方法が存在するが、通常のGCはマークアンドスイープと呼ばれる方法をベースにしている。

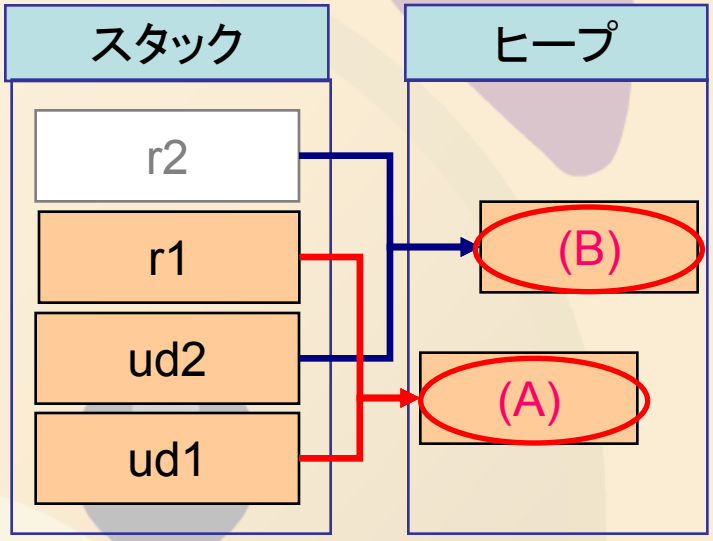
マークアンドスイープの例①

```
class UserData {
    public int d;
}
static vod Main(string[] args) {
    UserData ud1 = new UserData();←(A)
    {
        UserData ud2 = new UserData();←(B)
        {
            UserData r1 = ud1;
            {
                UserData r2 = ud2;
            }
        }
    }
}
```



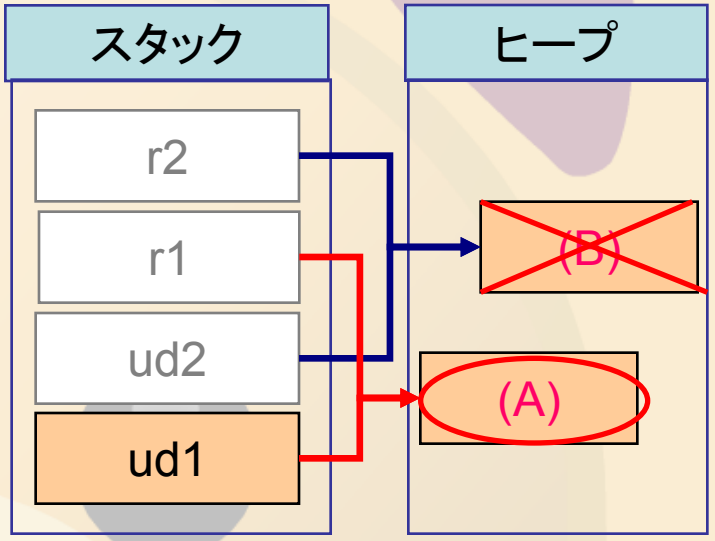
マークアンドスイープの例②

```
class UserData {
    public int d;
}
static vod Main(string[] args) {
    UserData ud1 = new UserData();←(A)
    {
        UserData ud2 = new UserData();←(B)
        {
            UserData r1 = ud1;
            {
                UserData r2 = ud2;
            }← ここでGCが動いたとして
        }
    }
}
```



マークアンドスイープの例③

```
class UserData {
    public int d;
}
static vod Main(string[] args) {
    UserData ud1 = new UserData();←(A)
    {
        UserData ud2 = new UserData();←(B)
        {
            UserData r1 = ud1;
            {
                UserData r2 = ud2;
            }
        }
    }← ここでGCが動いたとして
}
```



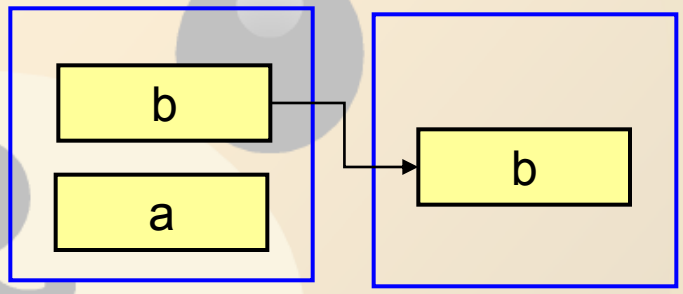
C++でGCを実現できないのはなぜ①

- GCはスタック上に残っている参照変数から繋がっている（使用中）のメモリに対してマークし、残った非使用メモリに対してスイープする。
- C言語では、スタック上格納された変数のタイプが通常の変数なのかポインタ変数なのか判断できない。

C++でGCを実現できないのはなぜ②

- C言語との互換性を重視している C++/Objective-Cでも同じく通常変数とポインタ変数を区別できない。

```
int a = 10;
int* b = new int;
```



  - 変数a, bのどちらがポインタでどちらが通常変数か判断できない

GCについての現在の動向

- AppleのMacOS X 10.5 用のObjective-C 2.0では、GCを導入済みらしい。(但しiPhone用の開発ツールではGCは使えないらしい)
- C++の次期標準規格C++0xでも最初GCを導入する方向で進んでいたが、現在の資料では直接的な導入は見送られた。
- C/C++では、保守的GCとう考え方で作成されたBoehm GCが存在する。このGCはC言語でも利用可能。

スマートポインタとは

- ガベージコレクタと同じメモリ管理機能を、ライブラリと使用できるようにしたもの
- C++では標準ライブラリにauto_ptrと呼ばれるスマートポインタが搭載されている

autp_ptr

- 通常のポインタとをauto_ptrに切り替えて使用すると、deleteを使用しなくても、自動でメモリを解放してくれる。

```
int* ip = new int;
*ip = 10;
delete ip;
```

```
auto_ptr<int> ip(new int);
*ip = 10;
```

ipのライブサイクルが終わると、自動的にdeleteされる

auto_ptrは使えない？①

- auto_ptrを別のauto_ptrに代入すると
- auto_ptr<int> p1(new int);

```
*p1 = 10;
auto_ptr<int> p2 = p1;
cout << "p2: " << *p2 << endl;
cout << "p1: " << *p1 << endl; // ｺﾝﾊﾟｲﾙOK,実行時エラー
```

- コンパイルは所有権が移動し、代入元のインスタンスを参照しようとすると実行時エラーになってしまう。

auto_ptrは使えない？②

- また標準ライブラリのvectorなどのコンテナクラスにも格納することは出来ない

新しいスマートポインタ

- Boostライブラリ登場
  - Boostとは、C++標準化委員会の多くが参加して作成されたオープンソースライブラリ
  - Boostには新しい4つのスマートポインタが存在する「shared_ptr(weak_ptr), scoped_ptr,intrusive_ptr」
  - この内shared_ptr(weak_ptr)はC++0x(次期標準C++規格)にそのまま導入される
    - VC-2008(VC9)でVS2008SP1を適用すると、TR1としてshared_ptr(weak_ptr)使用可能

shared_ptr①

- auto_ptrと同じような使い方が可能

```
shared_ptr<int> pi(new int);
*pi = 10;
```

shared_ptr②

- auto_ptrと違い所有権が無く、他のshared_ptrに代入しても、代入元もそのまま使用できる

```
shared_ptr<int> p1(new int);
shared_ptr<int> p2 = p1;
cout << "p2: " << *p2 << endl;
cout << "p1: " << *p1 << endl;
```

shared_ptr③

- shared_ptrは参照カウンタを使ってメモリ管理を実現している

```
{
    shared_ptr<int> p1(new int);    // 参照カウント=1
    {
        shared_ptr<int> p2 = p1;    // 参照カウント=2
        {
            shared_ptr<int> p3 = p2;  // 参照カウント=3
        }  // ここで参照カウント=2
    } // ここで参照カウント=1
} // ここで参照カウント=0になったのでインスタンスを解放
```

weak_ptr①

- 参照カウンタ方式のメモリ管理を使用した場合、参照先が参照元を参照するような循環参照ではインスタンスを解放できない

```
struct UserData2;
struct UserData1 {
    shared_ptr<UserData2> op;
};
struct UserData2 {
    shared_ptr<UserData1> op;
};
int main() {
    shared_ptr<UserData1> p1(new UserData1());
    shared_ptr<UserData2> p2(new UserData2());
    p1->op = p2;
    p2->op = p1;
}
```

weak_ptr②

- weak_ptrは弱参照ポインタで参照カウントを増加させない

```
struct UserData2;
struct UserData1 {
    shared_ptr<UserData2> op;
};
struct UserData2 {
    weak_ptr<UserData1> op;
};
int main() {
    shared_ptr<UserData1> p1(new UserData1());
    shared_ptr<UserData2> p2(new UserData2());
    p1->op = p2;
    p2->op = p1;  // p1 の参照カウントは増加しない
}
```

scoped_ptr

- scoped_ptrはauto_ptrやshared_ptrと違い、メモリ管理は行わない
- その代わり代入は出来ない

```
{
    scoped_ptr<int> p1(new int);
    *i = 10;
    scoped_ptr<int> p2 = p1;   // コンパイルエラー
}
```

unique_ptr

- 次期標準C++規格のC++0xには、auto_ptrの代わりにunique_ptrが導入される
- C++0xのムーブセマンティックスと右辺値参照の機能を使い、auto_ptrの問題点を少し改良しているらしい。
- C++0xの詳しい説明は：わんくま同盟　東京勉強会 #22の「C++0x 言語の未来を語る」byアキラさんをみましょう。

それぞれのスマートポインタ関係

C++03	Boost	C++0x
auto_ptr		unique_ptr (auto_ptrも互換性のために残される)
	shared_ptr	shared_ptr
	weak_ptr	weak_ptr
	scoped_ptr	