OpenFoamのためのC/C++

第1回　メモリ管理

田中昭雄

目的

この勉強会の資料があれば、
OpenFoamカスタマイズ時にC/C++で迷わない

予定

- 第1回　メモリ管理
- 第2回　OpenFOAMで勉強するクラス
- 第3回　OpenFOAMで勉強するテンプレート
- 第4回　OpenFOAMカスタマイズ
- 第5回　未定
- 第6回　未定

今回のテーマ

C++におけるメモリ管理について理解する

- メモリ管理にまつわるバグを直すは大変
- バグを出さない事が肝要
- バグを出さないために正しい理解

今回の前提

- C言語で
  - 配列を使ったことがある
  - 構造体を使ったことがある
  - 関数を使ったことがある
- クラスという言葉を聞いたことがある
- C++ベースで解説していきます

Agenda

- メモリリーク, 32bit / 64bit
- new/malloc, delete/freeの違い
- メモリ確保/解放
- C++におけるスマートポインタ

Agenda

- メモリリーク，32bit / 64bit
- new/malloc，delete/freeの違い
- メモリ確保/解放
- C++におけるスマートポインタ

本章を読まなくてもプログラム書けます

メモリリーク：
　バグの1つ

32bit / 64bit：
　アプリケーション作成の設定
　利用可能なメモリ量が変わります

メモリ管理

malloc / freeとnew / delete

malloc / freeの利用例：

```
int* p = (int*)malloc(sizeof(int));
*p = 100;
std::cout  << "content of p is " << *p << "¥n";
free(p);
```

```
Content of p is 100
```

new / freeの利用例：

```
int* p = new int(500);
std::cout << "content of p is " << *p << "¥n";
delete p;
```

```
Content of p is 500
```

大量データを利用したい場合にメモリ管理

メモリ領域の違い

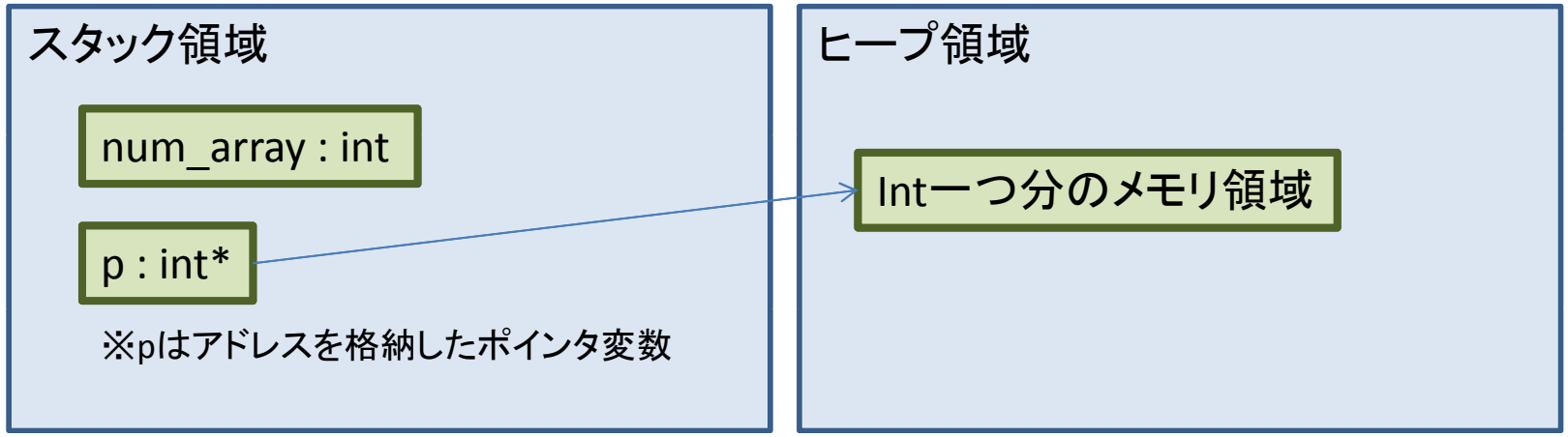
- スタック
  - 関数内の作業用一時変数（ローカル変数用）
  - 高速・容量少ない（数MB程度）
  - 確保/解放は自動
- ヒープ
  - 大規模データ取り扱い用
  - 大容量
  - 確保/解放はプログラマ責任

メモリ領域の違い

ローカル変数とnew/mallocで確保した変数の違い

例 :

```
int num_array = 1000;
for(int i = 0; i < num_array; ++i)
{
    int* p = new int(i);
    std::cout << i << "th content is " << *p << "¥n";
    delete p;
}
```



メモリリーク

メモリを使っていないつもりだけど使っている状態

メモリリークの例：

```
int num_array = 1000;
for(int i = 0; i < num_array; ++i)
{
    int* p = new int(i);
    std::cout << i << "th content is " << *p << "¥n";
}
```

```
0 th content is 0
1 th content is 1

…

999 th content is 999
```

解説：

forループ内のpはローカル変数
ループ毎にpの値（アドレス）が更新
freeされるべきアドレスはループ毎に分からなくなってしまう
⇒もう再利用も解放できない。int1000個分のメモリが無駄に利用されている

メモリリーク

メモリを使っていないつもりだけど使っている状態

- **原因は解放忘れ**
- **アプリケーション（プロセス）が利用可能な最大メモリ量は決まっている**
- **最大メモリ量を超えるとクラッシュ（計算途中でも関係ない）**

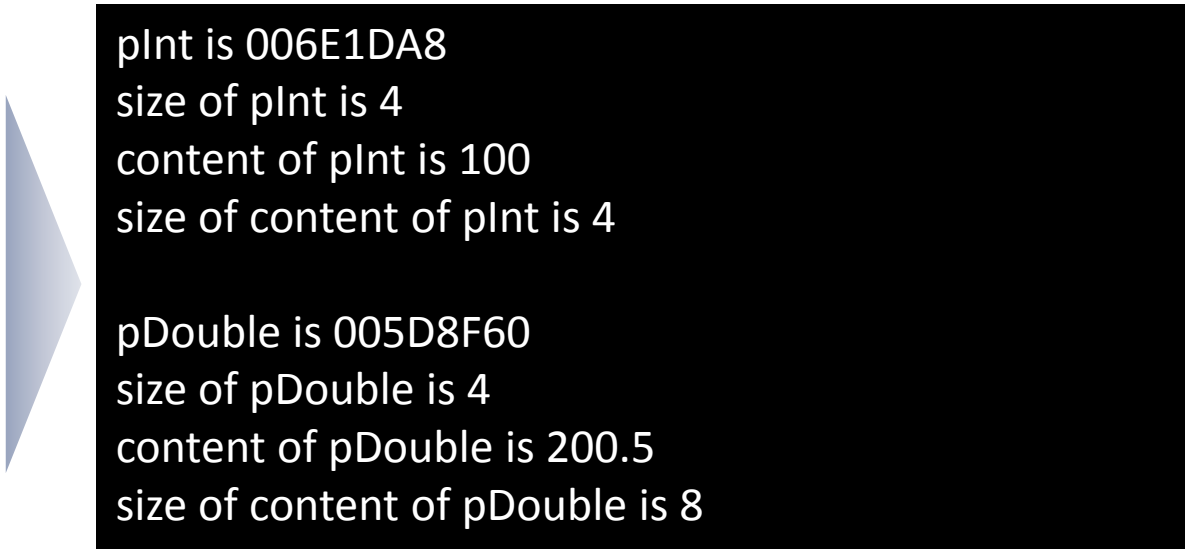
ポインタ変数を覗いてみる

ポインタ変数の値、サイズはどうなっているのか

```
int* pInt = new int(100);
double* pDouble = new double(200.5);

std::cout << "pInt is " << pInt << "¥n";
std::cout << "size of pInt is " << sizeof(pInt) << "¥n";
std::cout << "content of pInt is " << *pInt << "¥n";
std::cout << "size of content of pInt is " << sizeof(*pInt) << "¥n";

std::cout << "pDouble is " << pDouble << "¥n";
std::cout << "size of pDouble is " << sizeof(pDouble) << "¥n";
std::cout << "content of pDouble is " << *pDouble << "¥n";
std::cout << "size of content of pDouble is " << sizeof(*pDouble) << "¥n";

delete pInt;
delete pDouble;
```

ポインタ変数を覗いてみる

ポインタ変数の値、サイズはどうなっているのか



ポインタ変数の値は変な値（アドレスを表現）
ポインタ変数のサイズはint用、double用でも4byte(32bit)
ポインタ変数の中身のサイズはint / doubleで異なる

ポインタ変数を覗いてみる

- **ポインタ変数の中身は。。。？**
  アドレス
  （変数が格納されている郵便番号のようなもの）

- **ポインタ変数のサイズは。。。？**
  ビルド設定によって異なります
  （32bit or 64bit）

32bit / 64bitアプリ

アドレスの長さ（ポインタ変数のサイズ）の違い

- **64bitは32bitよりアドレスが長い**
  - **32bitアプリの場合、ポインタ変数長は32bit**
  - **64bitアプリの場合、ポインタ変数長は64bit**

- **ポインタ変数のサイズの違いが、利用可能なメモリ量の違いを生む**

郵便番号の長さと同じ仕組み

他のプログラミング言語

メモリの自動解放してくれる言語がある

- .NET（C#など）
- Java
- Ruby
- …

実行環境が確保したメモリを監視
使われなくなったらメモリを自動解放してくれる
（ガーベッジコレクション）

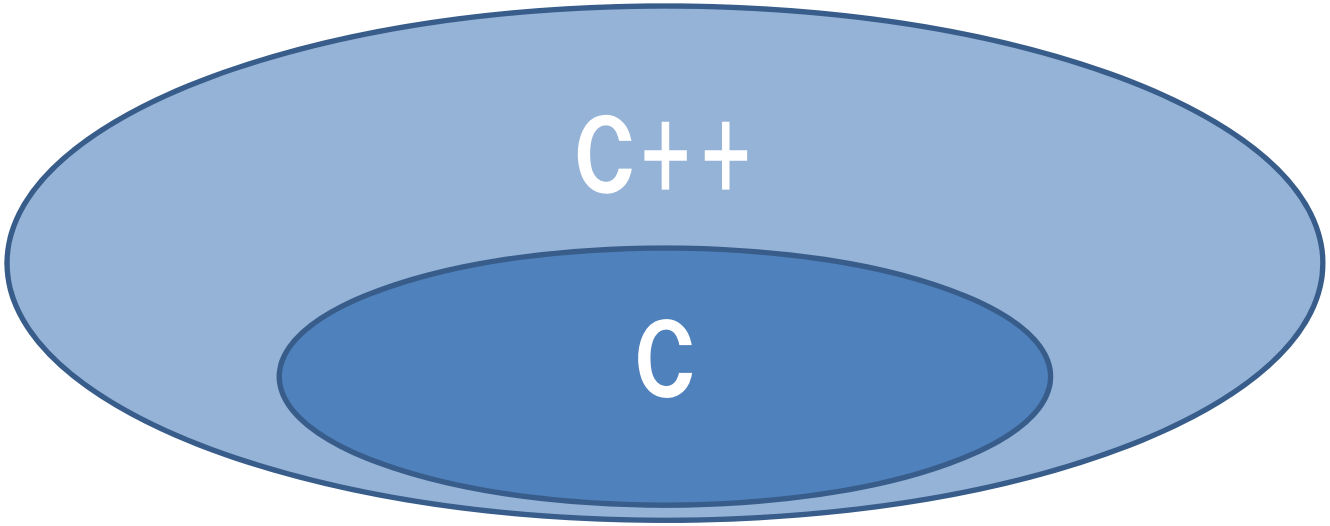
Agenda

- メモリリーク, 32bit / 64bit
- new/malloc, delete/freeの違い
- メモリ確保/解放
- C++におけるスマートポインタ

CとC++の違い

C++はCより色々なことができる

- **C仕様はC++仕様のサブセット**
- **ファイル拡張子.c / .cpp**
- **今回はコンストラクタとデストラクタのみ説明**



クラス / 構造体

理解しやすいように変数などをまとめる

3次元ベクトル構造体例:

```
structVector3D
{
 double x, y, z;

 Vector3D()
 {
  std::cout << “Making Vector3D¥n”;
  x = 10,  y = 10,  z = 10;
 };

 ~Vector3D()
 {
   std::cout << “Deleting Vector3D¥n";
 };
};
```

3次元ベクトルクラス例:

```
class Vector3D
{
public:
 double x, y, z;

 Vector3D()
 {
  std::cout << “Making Vector3D¥n”;
  x = 10,  y = 10,  z = 10;
 };

 ~Vector3D()
 {
   std::cout << “Deleting Vector3D¥n";
 };
};
```

public(アクセス修飾子のひとつ)については次回説明予定

コンストラクタ / デストラクタ

クラス/構造体の初期化・終了処理のための関数

3次元ベクトルクラス例：

```
class Vector3D
{
public:
  double x, y, z;

  Vector3D()
  {
    std::cout << “Making Vector3D¥n”;
    x = 10,  y = 10,  z = 10;
  };

  ~Vector3D()
  {
     std::cout << “Deleting Vector3D¥n";
  };
};
```

コンストラクタ：

クラス名と同じメンバ関数

メモリ確保時に自動実行

この例では、メモリ確保時に

x, y, zを10に設定

デストラクタ：

クラス名の前に~がついたメンバ関数

メモリ解放時に自動実行

この例では、メモリ解放時に

“deleting Vector3D”を表示

C++の新機能（Cでは利用不可）

コンストラクタ / デストラクタ

クラス/構造体の初期化・終了処理のための関数

3次元ベクトルクラス例：

```
Vector3D vec;

std::cout
    << "(x, y, z) = “
    << vec.x << ", " << vec.y << ", " << vec.z << "¥n";
```

```
Making Vector3D
(x, y, z) = 10, 10, 10
Deleting  Vector3D
```

解説：

vecがVector3Dクラスのローカル変数としてメモリ確保（コンストラクタ実行）
vecのx, y, zを表示（コンストラクタで10が設定）
プログラム終了時にローカル変数vecのメモリ解放（デストラクタ実行）

new/malloc, delete/freeの違い

初期化・終了処理を呼び出す / 呼び出さない

newとdelete：

```
Vector3D* vec = new Vector3D();
std::cout
    << "(x, y, z) = "
    << vec->x << “, “ << vec->y << ”, “ << vec->z << "¥n";
delete vec;
```

```
Making Vector3D
(x, y, z) = 10, 10, 10
Deleting  Vector3D
```

mallocとfree

```
Vector3D* vec = (Vector3D*)malloc(sizeof(Vector3D));
std::cout
    << "(x, y, z) = "
    << vec->x << ", " << vec->y << ", " << vec->z << "¥n";
free(vec);
```

```
(x, y, z) = -6.27744e+066, -
6.27744e+066, -6.27744e+066
```

※
x, y, zは初期化されていないため
実際の表示は環境によって異なります

Agenda

- メモリリーク, 32bit / 64bit
- new/malloc, delete/freeの違い
- **メモリ確保/解放**
- C++におけるスマートポインタ

1つの変数のメモリ確保/解放

newとdelete:

```
Vector3D* vec = new Vector3D();
std::cout
    << "(x, y, z) = "
    << vec->x << “, “ << vec->y << ”, “ << vec->z << "¥n";
delete vec;
```

```
Making Vector3D
(x, y, z) = 10, 10, 10
Deleting  Vector3D
```

mallocとfree

```
Vector3D* vec = (Vector3D*)malloc(sizeof(Vector3D));
vec->x = 10;
vec->y = 10;
vec->z = 10;
std::cout
    << "(x, y, z) = "
    << vec->x << ", " << vec->y << ", " << vec->z << "¥n";
free(vec);
```

```
(x, y, z) = 10, 10, 10
```

※mallocを利用する場合は、個別に初期化必要。コンストラクタ実行されないため。

配列のメモリ確保/解放

変数1つの場合と異なる命令が必要

newとdelete:

```
Vector3D* vec = new Vector3D[10];
std::cout
    << "(x, y, z) = "
    << vec[7].x << “, “ << vec[7].y << ”, “ << vec[7].z << "¥n";
delete [] vec;
```

※10個分のコンストラクタ / デストラクタが実行されます

```
Making Vector3D
…
(x, y, z) = 10, 10, 10
Deleting  Vector3D
…
```

mallocとfree

```
Vector3D* vec = (Vector3D*)malloc(sizeof(Vector3D) * 10);
vec[0].x = 10;
vec[0].y = 10;
vec[0].z = 10;
std::cout
    << "(x, y, z) = "
    << vec[7].x << ", " << vec[7].y << ", " << vec[7].z << "¥n";
free(vec);
```

```
(x, y, z) = -6.27744e+066, -6.27744e+066, -6.27744e+066
```

※
最初の配列要素のx, y, zは初期化しているが、インデックスが7の配列要素は初期化されていないので
おかしな値が表示されます

配列のメモリ確保/解放

連続したアドレスとしてメモリ確保

```
Vector3D* vec = new Vector3D[10];
```

10個分のVector3Dのメモリがnewによって確保
配列要素毎にコンストラクタが実行
8番目の配列要素を使いたい場合はvec[7]
最終要素のインデックスは9

(ex) vec[7].x = 100

```
for(int i = 0; i < 10; ++i)
{
  std::cout << vec[i].x << “, “ << vec[i].y << “, “ << vec[i].z << “¥n”;
}
```

```
delete [] vec;
```

vecとして確保された配列（10個分のVector3D）をdeleteによって解放
配列要素毎にデストラクタが実行

配列のメモリ確保/解放

ありがちな間違い

```
delete vec;
```

配列のメモリが正しく解放されません
場合によってはクラッシュ、場合によってはメモリ状態がおかしいまま計算続行
（計算処理には誤りがないのに結果がおかしい、状態）

```
Vector3D* vec = new Vector3D[10];
vec[10].x = 100;
```

何に利用されているか分からないメモリの状態を変更しています
場合によってはクラッシュ、場合によってはメモリ状態がおかしいまま計算続行
（計算処理には誤りがないのに結果がおかしい、状態）

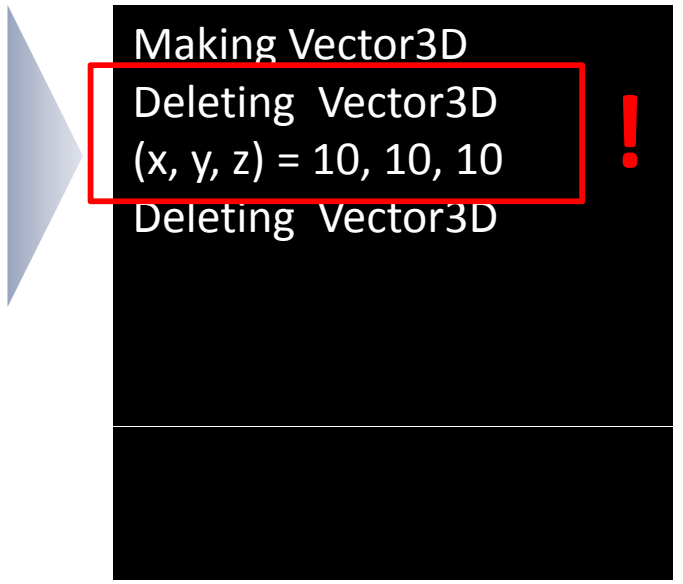
脆弱性に分類

関数にデータを渡したい

関数の引数に対する正しい理解が必要

典型的な誤り例：

```
void assign100IntoX(Vector3D vecInFunc)
{
  vecInFunc.x = 100;
};

int main()
{
  Vector3D vec;
  assign100IntoX (vec);
  std::cout << vec.x << ", " << vec.y << ", " << vec.z << "¥n";
  return 0;
}
```



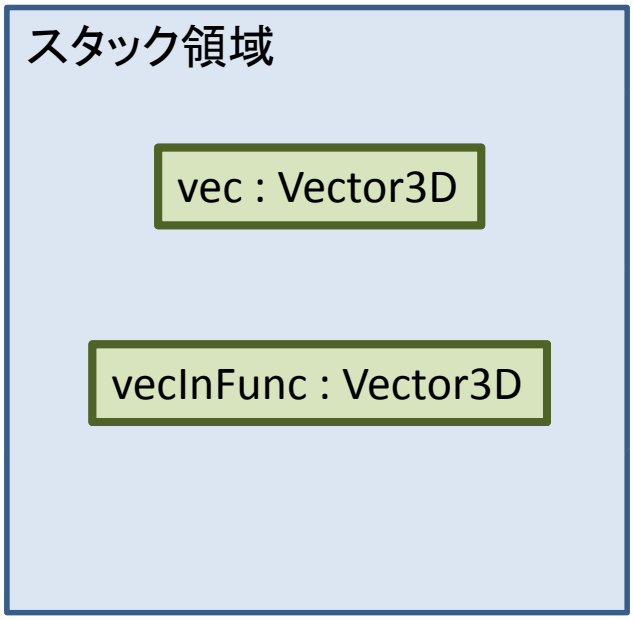
この結果を理解できない、ということは
関数の引数を正しく理解していません

関数の引数は、
関数内のローカル変数かつその値は呼び出し元のコピー

関数にデータを渡したい

関数の引数に対する正しい理解が必要

```
void assign100IntoX(Vector3D vecInFunc)
{
  vecInFunc.x = 100;
};

int main()
{
  Vector3D vec;
  assign100IntoX (vec);
  std::cout << vec.x << ", " << vec.y << ", " << vec.z << "¥n";
  return 0;
}
```



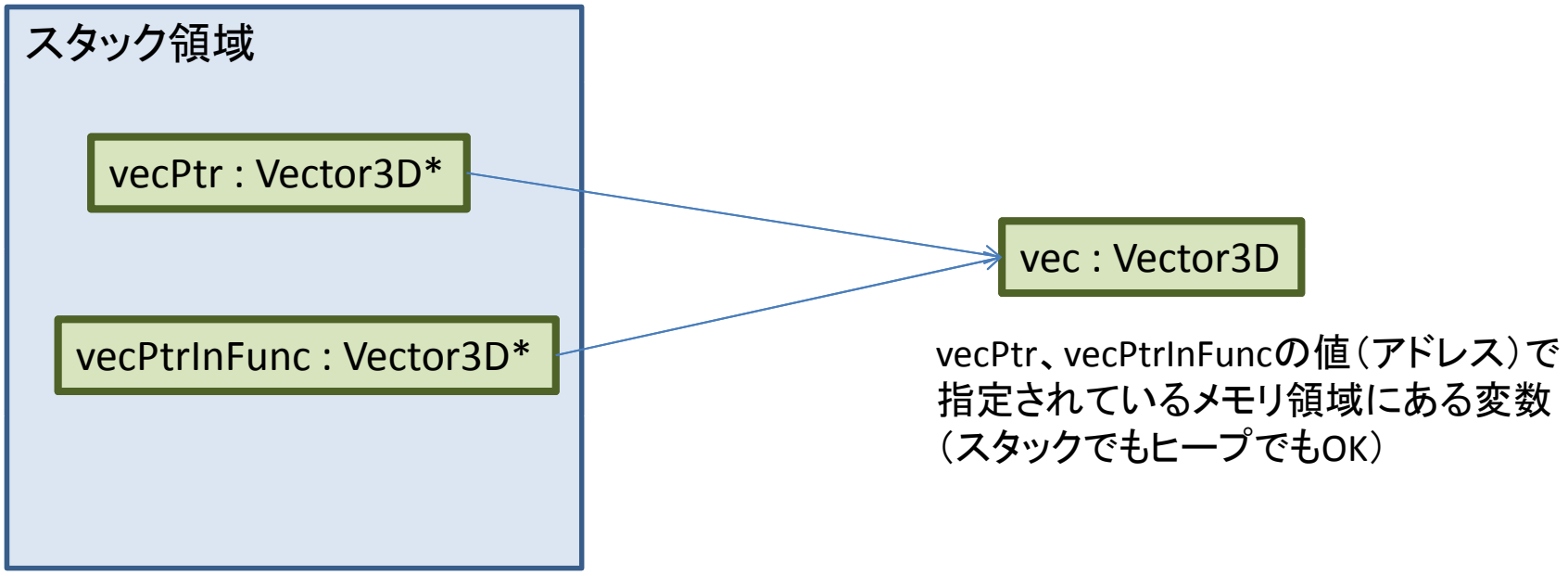
ローカル変数として
vecとvecInFuncは独立して存在

解説

①vecがローカル変数として確保
②vecInFuncがローカル変数として確保
③vecをvecInFuncへコピー
④vecInFuncのxを変更
⑤assign100IntoX内のローカル変数であるvecInFuncを削除（関数実行終了に伴い）

関数にデータを渡したい

アドレスを利用して解決



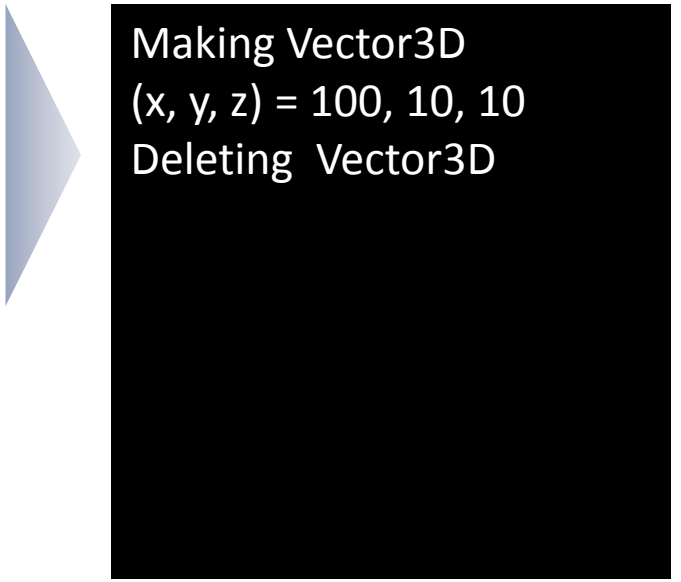
関数引数としてアドレスを渡す
アドレスであれば、コピーであっても問題なし

関数にデータを渡したい

ポインタ変数 or 参照を利用

関数の引数にポインタ変数を利用した例：

```
void assign100IntoXWithPointer(Vector3D* vecPtrInFunc)
{
  vecPtrInFunc->x = 100;
};

int main()
{
  Vector3D vec;
  assign100IntoXWithPointer (&vec);
  std::cout << vec.x << ", " << vec.y << ", " << vec.z << "¥n";
  return 0;
}
```



ローカル変数などのアドレス（=ポインタ変数の値）は
変数名の最初に“&”をつけるとアドレス取得

ポインタ変数がコピー = アドレスがコピーされて関数に渡される

関数にデータを渡したい

ポインタ変数 or 参照を利用

関数の引数に参照を利用した例：

```
void assign100IntoXWithReference(Vector3D& vecRefInFunc)
{
  vecRefInFunc.x = 100;
};

int main()
{
  Vector3D vec;
 assign100IntoXWithReference (vec);
  std::cout << vec.x << ", " << vec.y << ", " << vec.z << "¥n";
  return 0;
}
```

```
Making Vector3D
(x, y, z) = 100, 10, 10
Deleting  Vector3D
```

アドレス取得の”&”と同じ書き方と同じなので混同に注意

効果はポインタ変数と同じ
ただし書き方が異なる

Agenda

- メモリリーク, 32bit / 64bit
- new/malloc, delete/freeの違い
- メモリ確保/解放
- C++におけるスマートポインタ

メモリ管理はC/C++の大きなテーマの一つ

- **うっかりミスで重いバグ修正**
- **プログラマは千差万別**

スマートポインタとは、いい感じに自動deleteしてくれるクラス
（属人的である技量に期待する部分を減らす努力）

基本的な考え方

コンストラクタで確保 / デストラクタで解放

3次元ベクトルクラス例 :

```
class Vector3DPtr
{
public:
    Vector3D* Ptr;

    Vector3DPtr()
    {
        Ptr = new Vector3D();
    };

    ~Vector3DPtr()
    {
        delete Ptr;
    };
};
```

```
int main()
{
   Vector3DPtr vecPtr;
   vecPtr.Ptr->x = 100;
   std::cout << vecPtr.Ptr->x << “¥n”;
   return 0;
}
```

※
実際に実行するためには
#include <iostream>
の記述が必要です。

デストラクタで、メンバ変数で指定されたメモリ領域が解放

std::auto_ptr

標準ライブラリから利用できるスマートポインタ

3次元ベクトルクラス例：

```
#include <memory>
…
int main()
{
  std::auto_ptr<Vector3D> aptrVec(new Vector3D());
  aptrVec->x = 100;
  std::cout
    << aptrVec->x << ", "
    << aptrVec->y << ", "
    << aptrVec->z << "¥n";
  return 0;
}
```

```
Making Vector3D
(x, y, z) = 100, 10, 10
Deleting  Vector3D
```

その他

- Boostライブラリ
  - scoped_ptr / scoped_array
  - shared_ptr / shared_array

課題

- STLファイルを読み込んで、全ての頂点座標をcsvファイルとして出力

- 必要な事
  - ファイル入出力
    (fopen/fclose, ifstream/ofstream
  - 文字列の一致判断
    (strcmp, std::string::compare)
  - 読み込んだ文字列を数値に変換 (atof)
  - 文字列の分割(strtok, std::string::substr)
  - STLファイルフォーマット
    (hiramine.com STLフォーマットで検索)
    http://www.hiramine.com/programming/3dmodelfileformat/stlfileformat.html

参考資料

- C++学習
  - 独習C++
    http://www.amazon.co.jp/gp/product/4798103187/

- 仮想メモリ・仮想アドレス
  http://software.fujitsu.com/jp/manual/manualfiles/M080099/J2UZ9570/03Z2A/tun07/tun00083.htm