情報メディア実験A

物理エンジンを使ったアプリケーション開発

筑波大学情報学群
情報メディア創成学類
藤澤誠

実験の目的

物理シミュレーションエンジンを利用した
アプリケーションが作れるようになる

- 教員による説明回+講義ページの説明を理解して練習問題を解く(実装する)
- 物理エンジンを用いた簡単なアプリケーションを作成

実験スケジュール

- 毎週 水3,4限＆金5,6限
- テーマ内スケジュール
  1. ガイダンス＆事前知識(C++)説明：**4/23**
  2. 物理シミュレーションとは?：**4/25**, 30, 5/2
  3. 物理エンジンとは？ ：**5/9**, 14, 16
  4. 剛体間の衝突判定,衝突応答：**5/21**, 23, 28, 30
  5. 剛体間リンク：**6/4**, 6, 11, 13
  6. 3Dモデル読み込みと弾性体：**6/18**, 20, 25, 27
  7. アプリケーション開発：**7/2, 4**, 9, 11, 16, 18, 23
  8. 成果発表会：7/25

赤太字は説明回，水曜:TA,金曜:教員が基本的に常駐(説明回を除く)
5/7(水)は火曜授業日
7/30(水)はレポート作成回 (レポート提出締切:8/6(水) 17:00)

教室とWebページ

- 教室
  TWINS,manaba等を参照

- Webページ
  https://fujis.github.io/iml_physics/

  講義資料はTeamsも使って共有予定

  チーム名：GC41103_情報メディア実験A_物理エンジンを用いたアプリケーション開発_2025
  チームコード：hy9j2r4

自宅環境での開発

基本的に計算機室のPCで実験を進めるが，実験説明ページ及びサンプルコードは**Windows+Visual Studio 2022以降**を想定しており，自宅でその環境を用意すれば課題を進めることは可能．ただし，Mac環境でもg++とgmakeで動くことは確認している(Mac用のMakefileは同梱している)．

- Windows環境
  無料版のVisual Studio Comminityをインストールしてもらえれば説明ページに書かれていることは問題なく実行可能です(VS2022Comminityで動作確認済み)
  → 詳しいインストール方法はTeamsの「ファイル」にある「**Visual Studio Comminityインストール方法.pdf**」参照

環境構築

基本的に計算機室のPCで実験を進めるが，実験説明ページ及びサンプルコードは**Windows+Visual Studio 2022以降**を想定しており，自宅でその環境を用意すれば課題を進めることは可能．ただし，Mac環境でもg++とgmakeで動くことは確認している(Mac用のMakefileは同梱している).

- Mac環境
  実験ページに書かれている手順でHomeblew, glfw, glewをインストール後，ターミナルでMakefileのあるフォルダに移動して，makeでビルド/make runで実行.

サンプルコード,MakefileはUbuntuなどLinux環境でも動作すると思うが，こちらでは動作確認はしていない.

C++の基本

- 本実習で使う物理エンジン(bullet physics)はC++で書かれたライブラリ(Pythonラッパあり)
  - C言語の拡張としてのC++の基本
  - メモリの確保と解放
  - オブジェクト指向の基本

ライブラリを使うための最低限の機能だけを教えます.

Hello World!(C言語版)

List.1

```
#include <stdio.h>
int main(void)
{
  /* Hello の画面出力World */
  printf("Hello␣World!\n");
  return 0;
}
```

Hello World!(C++版)

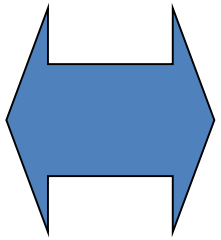
List.2

```
#include <iostream>
//名前空間の設定
using namespace std;
int main(void)
{
  // Hello の画面出力World
  cout << "Hello␣World!" << endl;
  return 0;
}
```

標準入出力

- iostream

標準出力 : cout
標準入力 : cin
標準エラー出力 : cerr



printf
scanf
fprintf(stderr, ･･･)

- 使い方

```
float a;
int b;
cin >> a;
cin >> b;
cout << "a = " << a << endl;
cout << "b = " << b << " = 0x" << hex << b << endl;
```

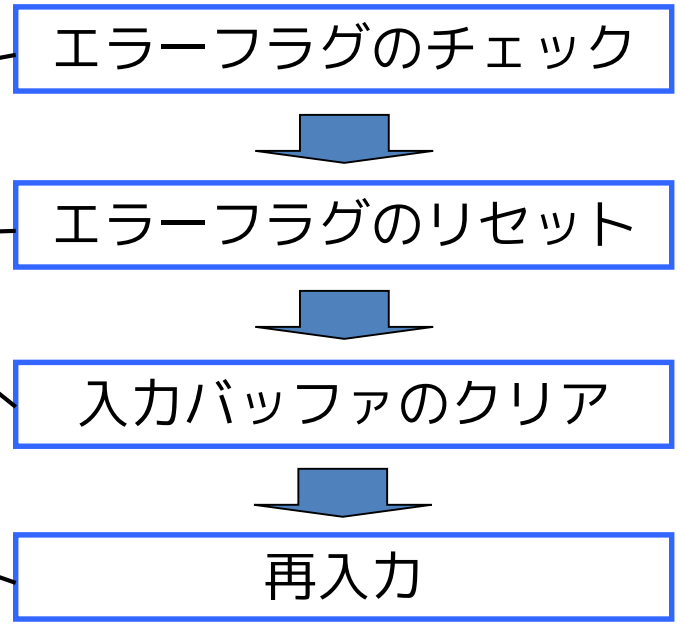
>>, <<の本来の使い方はシフト演算
→演算子のオーバーロード

cinのエラー処理

- エラー処理

  cinは内部にエラーフラグ変数を持つ

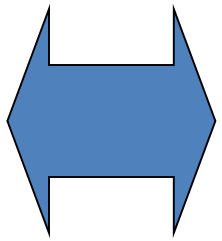
```
int x;
cin >> x;
while(!cin){
  cin.clear();
  cin.ignore(INT_MAX, '\n');
  cout << "再入力 : ";
  cin >> x;
}
```



ファイル入出力 1

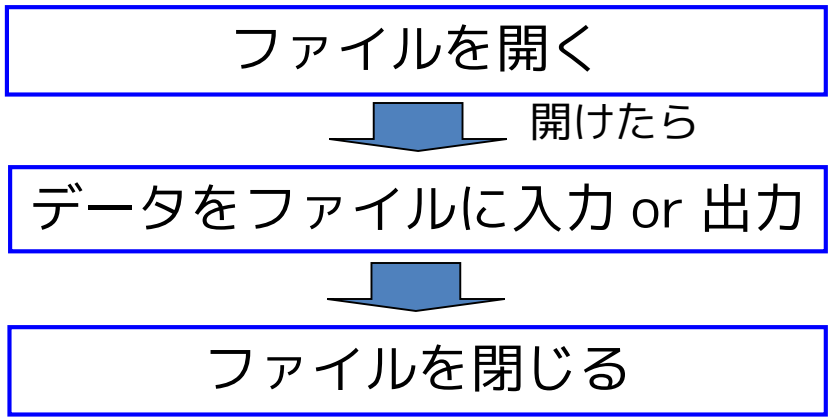
- fstream (ifstream, ofstream)

ファイル出力 : ofstream
ファイル入力 : ifstream
ファイル入出力 : fstream



fprintf
fscanf
FILE

- ファイル入出力の手順



ファイル入出力2

```
ofstream fo;
fo.open("output.txt");
if(fo){
   fo << "データの出力" << endl;
   fo.close();
}
else{
   //エラー処理
}
```

変数の宣言

List.3

```
#include <iostream>
using namespace std;
int main(void)
{
   //変数の宣言
   for(int i = 0; i < 10; ++i){
      double a = (double)i*i;
      cout << a << endl;
   }
   return 0;
}
```

関数のデフォルト引数

List.4

```
#include <iostream>
using namespace std;
//デフォルト引数
void def_test(int d = 0)
{
  cout << "d = " << d << endl;
}
int main(void)
{
  def_test();
  def_test(1);
  return 0;
}
```

関数のオーバロード

List.5

```
#include <iostream>
using namespace std;
//関数オーバロード
int func(void)
{
  int d;
  cout << "Input␣Integer␣Number␣:␣";
  cin >> d;
  return d;
}
void func(int d)
{
  cout << d << endl;
}
int main(void)
{
  func(func());
  return 0;
}
```

引数の参照渡し

List.6

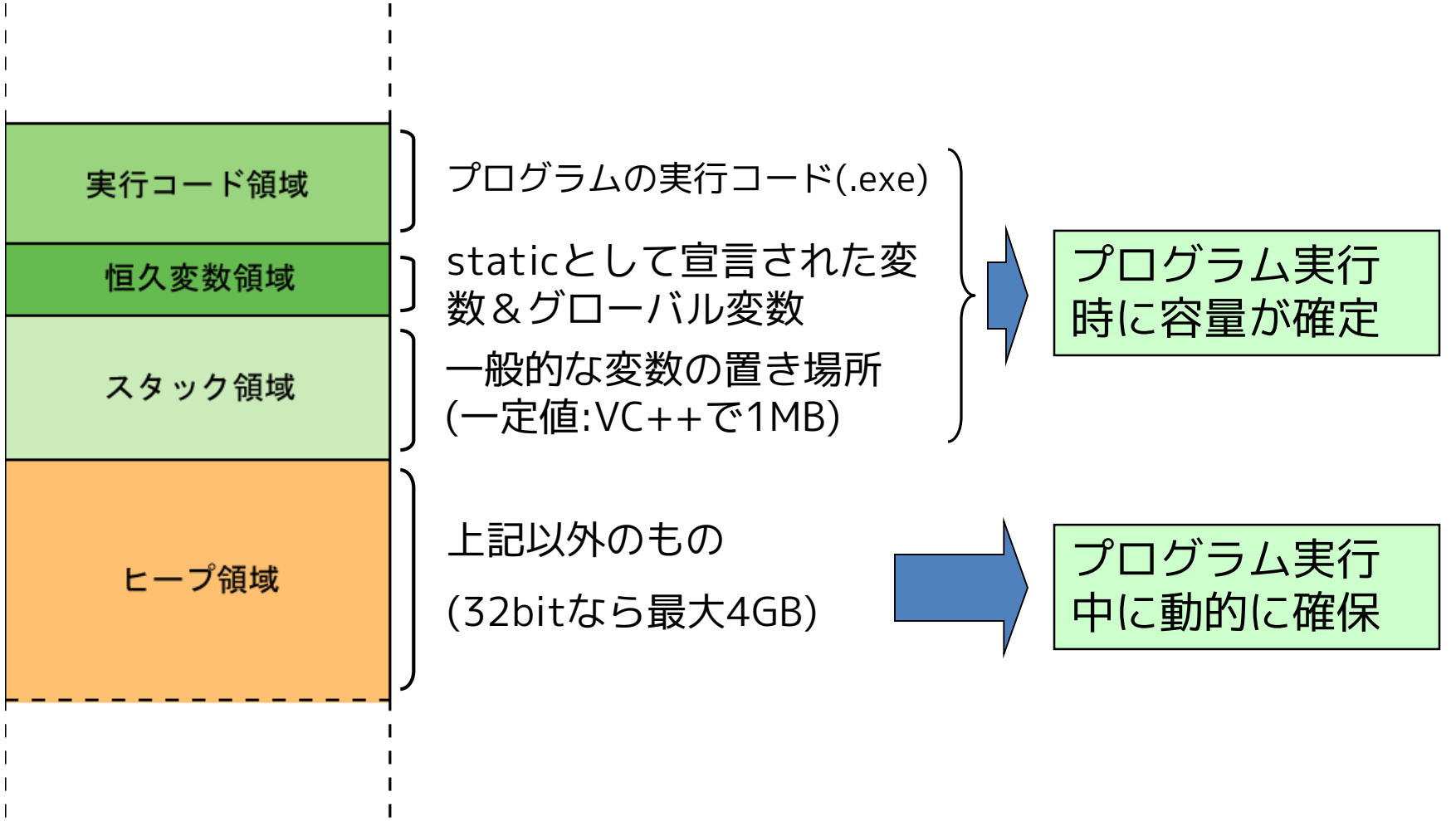
```
void swap(int *a, int *b)
{
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
```

List.7

```
void swap(int &a, int &b)
{
  int c;
  c = a;
  a = b;
  b = c;
}
```

メモリについて

- メモリマップ

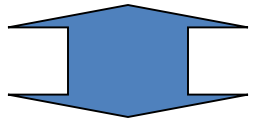


スタック領域

- スタック領域は何のためにあるのか？

  レジスタの数を超えた変数が使用されたときに用いられる

  - スタックの利点：割り当てや開放の手間がないので高速
  - スタックの欠点：スタックのサイズはあまり大きくできない



  - ヒープの利点：サイズ制限なし(ただしメモリ容量による)
  - ヒープの欠点：OSによるメモリ管理コードのため低速

メモリの動的確保

- malloc,free (C言語)

```
void *malloc(size_t size)
void *calloc(size_t n, size_t size)

int *p;
p = (int*)malloc(10*sizeof(int));
--------pを使った処理 ---------
free(p);
```

- new,delete (C++)

```
int *p;
p = new int[10];
--------pを使った処理 ---------
delete [] p;
```

確保した領域は
必ず開放する

(STLのvectorやbulletの
bt*Arrayの場合は自動開放)

番外:STLによる動的配列

- STL(標準テンプレートライブラリ)による動的配列

```
#include <vector>
using namespace std;
void main(void){
    vector<int> x;
    x.resize(5, 0); // 配列サイズ変更(size:5)して，値0で初期化
    x.push_back(10); // 配列に値を追加(size:6)

    // 反復
    for(int i = 0; i != x.size(); ++i) cout << x[i] << endl;
    // イテレータを使った反復
    for(vector<int>::iterator i = x.begin(); i != x.end(); ++i)
        cout << *i << endl;
}
```

オブジェクト指向とは

- オブジェクト指向

「関連するデータの集合と，それに対する手続き(メソッド)を「オブジェクト」と呼ばれる一つのまとまりとして管理し，その組み合わせによってソフトウェアを構築する手法」

(IT用語辞典e-Wordsより)

- カプセル化(Encapsulation)
- 継承(Inheritance)
- 多態性(Polymorphism)

構造体の復習

List.8

```
#include <iostream>
using namespace std;
typedef struct Vec3
{
  float x, y, z;
} Vec3;
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```

はじめてのクラス

List.9

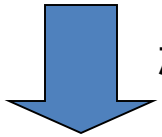
```
#include <iostream>
using namespace std;
class Vec3
{
  float x, y, z;
};
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```

アクセスコントロール１

- private,public,protected

List.9のコードはコンパイルエラーとなる



なぜか？

アクセスコントロール

- publicメンバ：外部からアクセス可能
- protectedメンバ：継承先からアクセス可能
- privateメンバ：クラス内からのみアクセス可能

デフォルトのアクセスコントロール

- 構造体：public
- クラス：private

アクセスコントロール2

List.9

```
#include <iostream>
using namespace std;
class Vec3
{
public:
  float x, y, z;
};
int main(void)
{
  Vec3 v;
  v.x = 1;
  v.y = 2;
  v.z = 3;

  return 0;
}
```

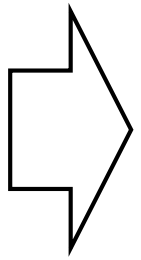
メンバ関数

- 関数をクラスのメンバに

List.10

メンバ関数

```
class Vec3
{
public:
  float x, y, z;
  void input(float x0, float y0, float z0);
  void output(void);
};
void Vec3::input(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
}
void Vec3::output(void)
{
  cout << "v␣=␣(" << x << ",␣";
  cout << y << ",␣";
  cout << z << ")" << endl;
}
```



メンバ関数
の呼び出し

```
int main(void)
{
  Vec3 v;
  v.input(1.0f, 2.0f, 3.0f);
  v.output();
  return 0;
}
```

カプセル化

- カプセル化

  アクセスコントロールでクラス内部の詳細をブラックボックス化

メンバ変数はpublicにしない

```
class Vec3
{
private:
  float x, y, z;
public:
  void input(float x0, float y0, float z0);
  void output(void) const;
};
```

コンストラクタとデストラクタ

- コンストラクタ(Constructor)

  オブジェクトを生成したときに呼ばれ，主に初期化を行う

```
public:
    クラス名 (引数)
```

返値なし，引数あり

オーバーロードにより複数存在可

- デストラクタ(Destructor)

  オブジェクトを破棄したときに呼ばれ，主に後処理を行う

```
public:
    ~クラス名 ()
```

返値なし，引数なし

複数存在不可

コンストラクタとデストラクタ

List.12

```
class Vec3
{
private:
  float x, y, z;
public:
  //コンストラクタとデストラクタ
  Vec3()
  {
    x = 0.0f; y = 0.0f; z = 0.0f;
  }
  Vec3(float x0, float y0, float z0){
  {
    x = x0; y = y0; z = z0;
  }
  ~Vec3(){}
};
```

デフォルトコンストラクタ

インスタンス

List.13

```
int main(void)
{
  Vec3 *vp;
  vp = new Vec3(1.0f, 2.0f, 3.0f);
  vp->output();
  delete vp;
  return 0;
}
```

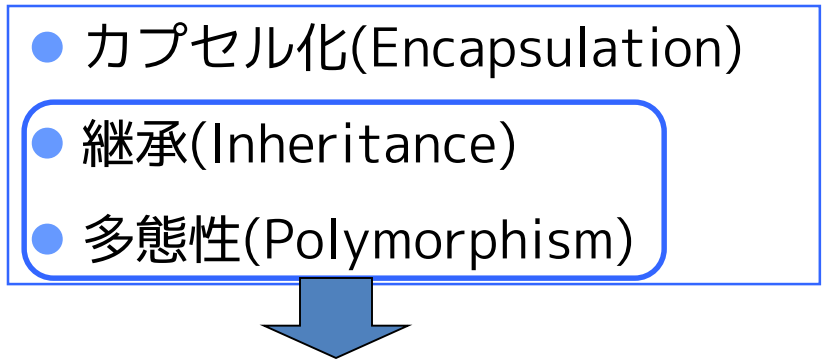
まとめ

- C++で拡張された機能

  オーバーロード，参照渡し，newとdelete

- オブジェクト指向：クラス

  アクセスコントロール，コンストラクタとデストラクタ，カプセル化



興味のある人はWebページにある資料「オブジェクト指向言語 C++ 後編」を読んでみてください．