第3回C++講座

目次

- クラスの基礎
- コンストラクタ、デストラクタ
- クラス作成の指針
- STLのstack、queue
- スタックを使った状態制御
- POD

ポイント

- クラスの定義　→　新しい型を作ること。ただのデータの集まりではなく、動作も定義できる
- メンバ変数　→　オブジェクトごとに保持
- メンバ関数　→　「呼び出したオブジェクト」のメンバ変数を操作する
- アクセス修飾子→メンバの参照範囲の制限
- 変数の代入　→　ビットコピー

- メンバ関数　→　オブジェクトごとに違う値を持つが、同じ処理をする場合に使う

- クラス定義

```
class クラス名
{
アクセス修飾子：
    メンバ変数;
    メンバ関数;
};
```

- コンストラクタ、デストラクタ→変数の初期化と破棄時の処理を定義する

クラスとは？

クラス・構造体の定義 ⇒ 新しい”型”を作ること！

(型：プログラム上で使用される変数の種類(ex:　int, double)　)

プログラム上での振る舞いを定義する必要がある

そのクラスの大きさはいくらか？
そのクラスのメモリ構成はどうなっているか？ ⇒ **メンバ変数**

メンバ変数をどのように操作するか？ ⇒ **メンバ関数**

メンバにどこからアクセスできるか？ ⇒ **アクセス修飾子**

初期化時の処理
破棄時の処理はどのようにするか？ ⇒ **コンストラクタ・デストラクタ**
コピーコンストラクタ

代入処理・不等式等はどのように行なうか？ ⇒ 演算子オーバーロード

共通した性質を抜き出す ⇒ 継承

自分で定義した通りに作用する変数を作ることができる！

クラスに必ず必要なもの

コンストラクタ

・変数の宣言時に必ず実行される。
・どのように初期化するか？
・デフォルトでは何もしないコンストラクタを生成

デストラクタ

・変数の破棄時に必ず実行される。
・どのように後始末をするか？
・デフォルトでは何もしないデストラクタを生成

コピーコンストラクタ

・変数を同じ型で「初期化」する場合に使用
・同じ型の参照を引数にとるコンストラクタ→ClassName(const ClassName&)
・デフォルトではすべてのメンバ変数のコピーコンストラクタを呼び出す
・組み込み型(int など)はビットコピーを行なう

代入演算子

・変数に「代入」処理をする場合に使用される演算子
・デフォルトでは、すべてのメンバ変数の代入演算子を呼び出す
・演算子は関数の一種
・組み込み型はビットコピーを行なう

変数の生成・破棄に関わる処理を定義することができる！

コンストラクタ、デストラクタ

- **コンストラクタとは**

(1)変数の宣言時に必ず実行される

(2)オーバーロードでき、引数が違う コンストラクタを多数定義できる

(3)自分で一つもコンストラクタを定義していない場合、自動で何もしない引数なしのコンストラクタが定義される

(4)返り値を持たない（voidであるという意味ではなく、そもそも返り値自体がない）

(5)コンストラクタは、そのクラスの名前と同じ名前である

```
class Test {

public :

    Test() {}

};
```

- **デストラクタとは**

(1)変数の破棄時に必ず実行される

(2)オーバーロード出来ず、引数を取らないデストラクタしか存在しない

(3)自分で定義しなかった場合、自動で何もしないデストラクタが定義される

(4)コンストラクタ同様返り値を持たない

(5)デストラクタは、そのクラスの名前の前に~をつけた名前である

```
class Test {

public :

    ~Test() {}

};
```

動作の確認

(1) コンストラクタは関数のように多重定義できる

(2) TestCase t; という書き方は、引数を取らないコンストラクタを呼び出している

(3) 引数を取らないコンストラクタが定義されていないと、TestCase t;と書けない

(4) int 型やdouble 型のような組み込み型も、int i(), j(3); double f(), g(3.3); のように初期化できる

(5) デストラクタは一つしか定義できない。
→自動で呼ばれる関数なので、引数を指定することは出来ず、結果として多重定義は出来ない。

```
class TestCase {
public :
    TestCase() {
        std::cout << “Constructor” << std::endl;
    }
    TestCase(int) {
        std::cout << “Constructor:int” << std::endl;
    }
    ~TestCase() {
        std::cout << “Destructor” << std::endl;
    }
};
int main() {
    TestCase t1, t2(3);   // コンストラクタ
    std::cout << “process” << std::endl;
    return 0;   // デストラクタの実行
}
```

動作の確認2

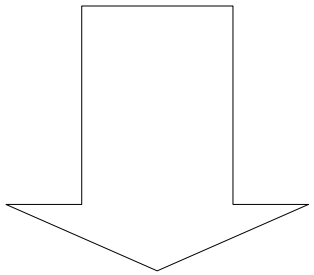
```
class Bunsu {
private:
    int d_bunsi, d_bunbo;
public :
    Bunsu(int bunsi, int bunbo)
      : d_bunsi(bunsi), d_bunbo(bunbo)
    {}
};
int main() {
    Bunsu b1; // NG
    Bunsu b3(3, 2); // OK
    return 0;
}
```

- ポイント

(1)一つでもコンストラクタを定義すると、何もしない引数なしのコンストラクタは自動では作られない

(2)これを利用すれば、初期化されていない分数クラスの変数が存在しないことになる

(3)ただし、後で値を代入したい場合も初期化しなければならなくなる



配列もコンストラクタが必要？

動作の確認3（配列）

```
class ArrayTest {
public :
    ArrayTest() {
        std::cout << “Constructor” << “\n”;
     }
    ~ArrayTest() {
        std::cout << “Destructor” << '\n' ;
    }
};
int main() {
    ArrayTest ar[10];
    return 0;
}
```

- ポイント

(1)配列とは、指定した型の変数を指定した数だけつくるものである

(2)つまり、配列で作成した変数それぞれに対してコンストラクタが実行され、破棄されるときにはそれぞれに対してデストラクタが呼ばれる

(3)初期化しない配列を作成するには、引数を取らないコンストラクタが必要である

(4)初期化処理の記述はルールが複雑なので、配列が必要なら引数なしのコンストラクタを定義するとよい

メンバ変数の初期化と初期化リスト

```
class Test {
private:
    int d_x, d_y;
public :
    Test(int x, int y) : d_x(x), d_y(y)
    {}
};
```

```
class Test2 {
private:
    int d_x, d_y;
public :
    Test2(int x, int y) {d_x = x; d_y = y;}
};
```

- ポイント

(1) 左のクラスのうち、メンバ変数を初期化しているのは上のクラスで、下のクラスはメンバ変数に値を代入しているだけである

(2) : d_x(x), d_y(y)
　の部分を初期化リストと呼び、複数の変数をカンマで区切って初期化する

(3) 初期化リストは、初期化したいメンバ変数のコンストラクタを呼び出すことで初期化する。つまり、引数を複数取るようなコンストラクタで初期化しても問題ない

(4)デフォルトコンストラクタ（引数なしのコンストラクタ）が定義されていないクラスの変数を用いる場合、初期化リストを使わなければならない

(5)クラス変数の初期化は、内部状態がどうであれ、コンストラクタを実行することで初期化されたと見なされる

演習

1. 前回作成したStackやQueueの初期化関数をコンストラクタに置換えてみよ。
   初期化関数を使用する場合とコンストラクタを使用する場合の相違点は何か

2. 右のプログラムの動作を予想せよ。なぜそうなるか。

```
void func(TestCase t) {}
void func_p(TestCase* tp) {}
void func_ref(TestCase& tr) {}

int main() {
    TestCase t1;
    func(t1);
    func_p(&t1);
    func_ref(t1);

    return 0;
}
```

3. 動作の確認2のBunsuクラスに、分子と分母を
   1で初期化するコンストラクタを作成せよ。

4.プログラムの多重起動防止機構に、Mutexを使用する
ものがある。
createMutex関数で文字列固有のMutexを生成し、 同一のMutexがリクエストされるとエラーを返す。 このMutexは、CloseHandle関数でクローズするまで有効である。

プログラム開始時にMutexを取得し、プログラム終了時にMutexを開放すると多重起動防止になる。
このような処理を自動で行なうクラスはどのように作ることができるか。

5.ゲームプログラミングではFPSを一定にする処理が重要になってくる。
ソースコード中に直に処理を書くと、別のゲームを作成するときにコピペが面倒であるから、FPSTimerクラスを作成することにした。必要な機能を考えて作成せよ。

データの集まりとしてのクラス

例　：SaveDataクラス

目的　：ゲームの各種フラグ、描画・音声・入力の設定等をひとまとめにして使いやすくしたい
メリット：引数の簡略化、変数の所在のわかりやすさ、設定変更の容易さ

必要なもの　：ゲーム内のフラグ30個、グラフィックの設定10個、音声設定10個、キー入力設定8個
フラグは全てint型にすることに決めた
設定は大まかな括りごとに分けて分かりやすくする
クラス外から簡単にアクセスできると便利
特に必要な初期化処理も終了処理もない。コピーに特段気を遣う必要もない

```
class SaveData {
public :
    int gameFlags[30];
    int graphicSettings[10];
    int soundSettings[10];
    int keySettings[8];
};
```

使われ方

```
SaveData save_data;
load(“filename”, save_data);
.....
gameStart(save_data);
.....
save(“filename”, save_data);
....
```

機能としてのクラス

例　：Timerクラス

目的　：簡単に時間を計測したい
メリット：使用するときに変数を大量に必要としない。プログラム中でわかりやすい

必要なもの　：start()からend()までの時間を計測するようにしたい
getTime()で計測時間を返すようにしたい
getPastTime()で途中の経過時間も知ることができるようにしたい
変数は、開始時間と終了時間を保持する必要がある。→クラス外に公開する必要はない
初期化、終了、コピー処理は特に必要ない

```
class Timer {
private :
    int d_start, d_end;

public :
    void start();
    void end();
    int getTime();
    int getPastTime();
};
```

使われ方

```
Timer t;
t.start();
....
t.end();
std::cout << “時間：“ << t.getTime() << std::endl;
...
```

＊時間は<ctime>ヘッダのtime(0);関数で取得可能

演習

データの集まりとしてのクラスと,機能としてのクラスを使用する

目的　：リプレイのためにプレイヤーの入力履歴を記録する
今回は標準入力で入力された文字と入力までの時間の記録
メリット：　一つの入力データがまとまっているとわかりやすい

必要なもの　：一つの入力を表わすデータクラスInput
どの文字を入力したのか、何秒かけたのかを記録する

時間を計測するTimerクラス

ヒント
・右のように使いたい
・MAX_INPUTは入力回数
・今回は入力回数分の配列を用意して
入力履歴を記録する

＊Enterキーを押さなくても反応する入力
getch()は、<conio.h>　ヘッダに定義されている。

＊C言語とC++の入出力が混在するのは
本来好ましくない。

```
static const int MAX_INPUT = 5;

int main() {
    Input inputs[MAX_INPUT];
    char c;
    Timer t;
    for(int i = 0; i < MAX_INPUT; ++i) {
        std::cout  << “文字の入力:”;
        t.start();
        c = getch();
        t.end();
        inputs[i].key = c;
        inputs[i].time = t.getTime();
    }
    ....表示
}
```

既存のstack,queueの使用

```
#include <iostream>
#include <stack>

int main()
{
	std::stack<int> s;
	while( true)
	{
		int i = 0;
		std::cin >> i;
		if(i == 0) break;

		s.push(i);
	}

	while( ! s.empty() )
	{
		std::cout << s.top() << " ";
		s.pop();
	}

	return 0;
}
```

<stack>ヘッダに定義されているstackクラス
名前空間：std

関数：

void push(const TYPE &値);	値の挿入
bool empty();	スタックが空かどうか
void pop()	値の破棄(破棄のみ)
TYPE &top();	値の取り出し
size_type size();	スタックの要素数

TYPEは、使用しているときの型
他、== での比較、=でのコピーも可能

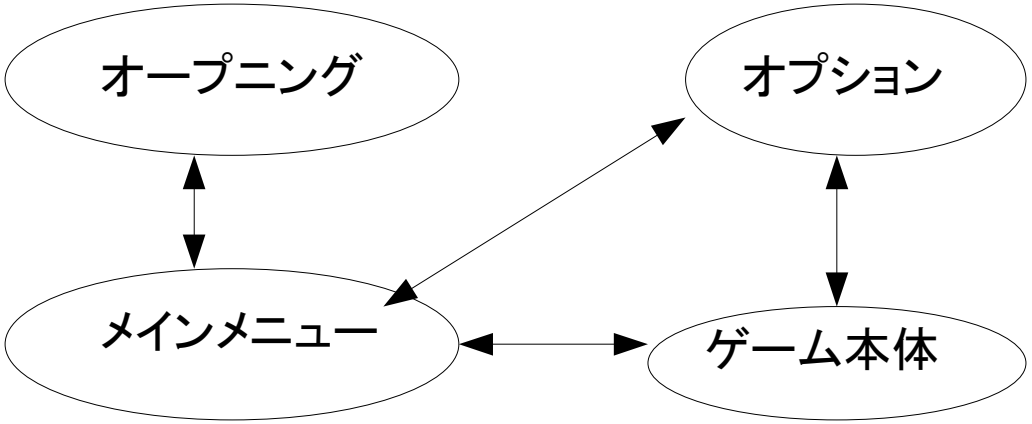
stack<int> と書くと、int用のstackになる。
stack<double>と書くと、double用のstackになる。
ただし、stack<int>型とstack<double>型は別の型！

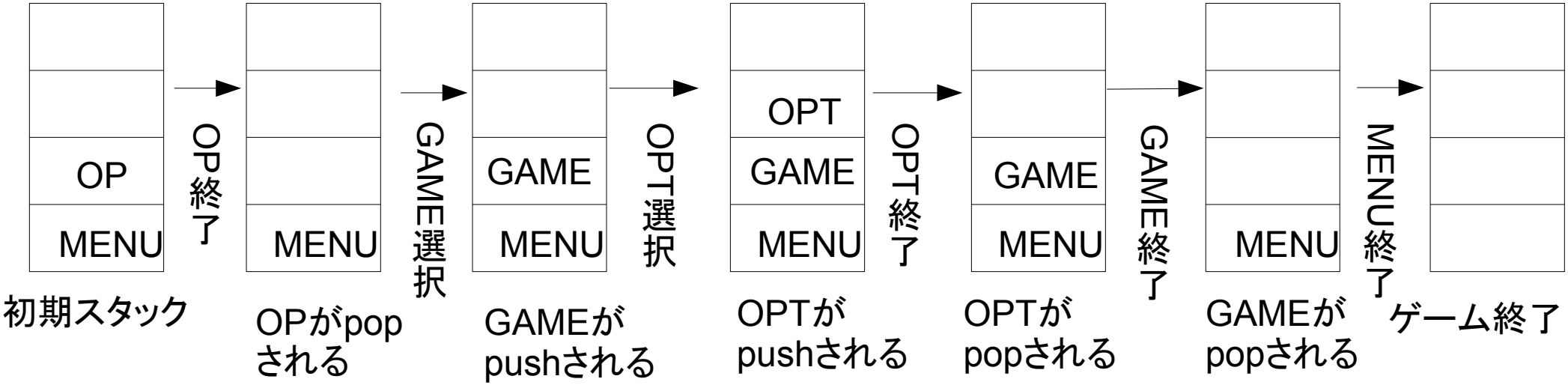
演習
<queue>ヘッダに定義されているqueue クラスを用いてchar型のqueue<char>クラスを作成し、←と同じようなことを試してみる。

stackを使用した状態制御

例:ゲームの状態制御

前提:ゲームの状態は以下の4種類
必ず前の状態に戻ってくる
複雑な状態変化はしない





MENUから飛んできてもGAMEから飛んできても
同一のOPTで処理可能(条件わけが不要)

⇒ 前の状態を気にせずに
プログラミングできる！

演習

stack<int>型の変数 gameState を作成し、ゲームの状態制御デモを行なう

１. ゲームの状態と整数を関連づける

例)

```
static const int OPENING  = 0;        #define OPENING    0
static const int MENU     = 1;   または #define MENU       1
static const int GAME     = 2;        #define GAME       2
...                                   ...
```

２. メイン関数で gameState 変数に初期状態をpushする

３. gameStateが空になるまで実行し続けるループを作成し、switch文で処理を分ける

```
while( ..... ) {
   switch( gameState.top() ) {
       case OPENING : .....
       case MENU : .......
       case GAME : ........
   }
}
```

４. 標準入力を用いて、擬似的に処理を再現する

例) オープニングで0が入力されたらオープニングを終わりにする
メニューで1が入力されたらオプションに移行する
メニューで2が入力されたらゲーム本体に移行する etc

POD(Plane Old Data)

```
class PlainOldData {
public :
    int x, y;
};
class TestClass {
public :
    int x, y;
    TestClass(int a, int b) {x = a; y = b;}
};
int main() {
    PlainOldData pod = {3, 4}; //OK
    TestClass tc = {3, 4};         //NG
    TestClass tc2(3, 4);           //OK
    return 0;
}
```

- C言語の構造体とC++の構造体は別物
- C++の構造体は、基本的にクラスと等価
- C++のクラスや構造体を、C言語での構造体と同じように使用できるデータ構造がPOD
- 以下にクラスや構造体がPODであるための条件を示す

(1)コンストラクタ、デストラクタ、コピーコンストラクタ、代入演算子を自分で定義しない

(2)メンバ変数が全てpublic である

(3)クラスの継承をしていない

(4)仮想関数を一つも持たない

(5)メンバ変数が全て上述の条件を満たしている

＊PODはC言語の構造体と同じ初期化ができる！
＊memcpy等のメモリ操作系の関数はPODにのみ使用すること！
＊C++のクラスに対するメモリ操作は保証されない！