

# 第3回C++講座

- コンストラクタとデストラクタについて
- 初期化子リスト
- コンストラクタと暗黙の型変換

# クラスに必ず必要な関数

- コンストラクタ

変数の宣言時に必ず実行される。デフォルトでは何もしないコンストラクタが生成される

- デストラクタ

変数の破棄時に必ず実行される。デフォルトでは何もしないデストラクタが生成される

- コピーコンストラクタ

変数を同じ型の変数で初期化する場合に使用されるコンストラクタの一種。デフォルトでは、全てのメンバ変数のコピーコンストラクタを呼び出す。組み込み型の変数はビットコピーをしようになっている(例: Point p; Point q = p;)

- 代入演算子

変数に同じ型の変数を「代入」する場合に使用される演算子。デフォルトでは、全てのメンバ変数の代入演算子を呼び出す。組み込み型の変数はビットコピーをしようになっている

# 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関数
- (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)戻り値を持たない(voidであるという意味ではなく、そもそも戻り値自体がない)
- (5)コンストラクタは、そのクラスの名前と同じ名前関数である

```
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関数
- (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戻り値を持たない
- (5)デストラクタは、そのクラスの名前の前に~をつけた名前関数である

```
class Test {  
public :  
    ~Test() {}  
};
```

# 動作の確認

- ポイント: (2)を訂正

(1) コンストラクタは多重定義でき、どのコンストラクタが呼ばれるかは、関数の多重定義の時のルールに従う

(2) 今までの、`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(デフォルトコンストラクタ)

(3) つまり、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;` のように使うことは出来ない

(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);` のように使用できるが、これはクラスの書き方に合わせるためのもので、厳密にはコンストラクタではない。

(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、結果として関数の多重定義は出来ない。

```
class TestCase {
public :
    TestCase() {
        std::cout << "Constructor" << '\n';
    }
    TestCase(int) {
        std::cout << "Constructor:int" << '\n';
    }
    ~TestCase() {
        std::cout << "Destructor" << '\n';
    }
};

int main() {
    TestCase t1, t2(3);

    std::cout << "process" << std::endl;

    return 0;
}
```

# クラスの変数をメンバに持つクラス

- 後述の初期化子リストを使用しない場合、メンバ変数は全てデフォルトコンストラクタで初期化される
- int 型などの組み込み型は、今までの場合同様、デフォルトコンストラクタでは何の値も格納されない不定値をとる
- デフォルトコンストラクタが定義されていないクラスの変数は、初期化子リストで初期化されなければならない

# メンバ変数の初期化と初期化子リスト

```
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) の部分を初期化子リスト(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) 初期化子リストは、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、初期化子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

# 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、何も引数なしのコンストラクタは自動では作られない
- (2)引数を取るコンストラクタしか定義していない場合、変数を宣言するときに必ず引数を渡さなければならない
- (3)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
- (4)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
- (5)このクラスの場合、デフォルトコンストラクタでは分子、分母共に1で初期化するようにつくる、などの改善案がある

# 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(デフォルトコンストラクタ or 規定のコンストラクタ)が必要である

# コンストラクタと暗黙の型変換

```
class FPSTimer {
public :
    FPSTimer(double fps) {}
};

class Bunsu {
public :
    Bunsu(int bunsu) {}
    Bunsu(int bunsu, int bunbo) {}
};

int main() {
    FPSTimer t = 60.25;
    Bunsu b;
    b = 3;
    return 0;
}
```

## • ポイント

- (1)引数を一つしか取らないコンストラクタの場合、  
FPSTimer t(60.25); と、  
FPSTimer t = 60.25;  
のような書き方は等価
- (2)b = 3; は、まず、3 を Bunsu(int bunsu) を使用してBunsu型に暗黙の型変換して、その後b に代入している
- (3)つまり、一時変数の生成とコンストラクタの実行、変数のコピーと一時変数の破棄と、多くの動作をしている
- (4)暗黙の型変換によって予期しない動作をする場合もあるので注意する(関数の引数の場合など)
- (5)基本的に、暗黙の型変換に頼ったコードは書かない方が良い。暗黙の型変換を抑止するためのキーワードに explicit がある。

# PlainOldData(POD)

- C言語の構造体とC++の構造体は根本的に異なる
- C++の構造体は、基本的にクラスと等価である
- 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である
- 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と一般のクラスの相違点

```
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;  
}
```

## • ポイント

- (1)C++のクラスや構造体では、初期化はコンストラクタで行なう
- (2)PODでなければ、C言語での配列や構造体で使った初期化の方法が使えない
- (3)C++のクラスや構造体のうち、PODであるものはC言語のときと同様に使用できる
- (4)クラスの機能は多岐に渡り、その分メモリ上での構造も複雑化している。C言語で保証されていた構造体のデータの並びも、クラスでは正しいとは限らない
- (5)memcpy 等のメモリコピー系の関数を使用する場合はPODにのみ使用すること。代入演算子、コピーコンストラクタ等を無視するという点で非常に危険である

# 初期化処理のまとめ

- 文法上の初期化とは、変数の宣言と同時に値を格納することである (`int a = 3;`)
- メンバ変数の初期化は初期化子リストで行なわれ、省略した場合、メンバ変数ごとにデフォルトコンストラクタが呼び出される
- クラス、構造体の文法上の初期化とは、コンストラクタを実行することである
- 明示的に初期化を行なわない場合、クラス変数はデフォルトコンストラクタで初期化される
- 「何もしないコンストラクタ」に代表されるように、クラスの変数は、メンバ変数が初期化されていなくても、文法上は初期化したといえる
- 初期化といえは、一般に変数の宣言と同時に、その変数を使用可能な状態にすることを意味する

# コンストラクタ、デストラクタの使用例

- 第1回で配布したクラス使用例で、init や set関数の代わりにコンストラクタを使用することができる
- 一般的な用途として、初期化処理をコンストラクタに任せることができ、デフォルトコンストラクタでは、規定値で初期化することもできる(分数クラスを宣言したら、勝手に分子・分母が1で初期化されるようにできる)
- ゲーム作成の際、絵や音楽などのリソースは、使ったら明示的に破棄しなければならない。これをデストラクタに任せれば破棄し忘れが無くなる
- コンストラクタ、デストラクタは、何もしない場合も書く習慣をつけておいた方がよい

# 演習

- 前回作成したPoint クラスについて、座標を任意の値で初期化するコンストラクタと、座標を(0, 0)で初期化するデフォルトコンストラクタを定義せよ
- Pointクラスの配列を作成せよ。二つのコンストラクタのうち、デフォルトコンストラクタのみを消去した場合どうなるか
- 前回示した100個限定のスタックについて、どのようなコンストラクタが考えられるか。また、コンストラクタの処理内容を一つの関数にまとめられないか考察してみよ
- FPSTimer クラスがある。このクラスはdouble 型を一つ引数にとるコンストラクタを持っている。また、FPSTimerを引数に取る void draw(FPSTimer fps); という関数がある。この関数を、draw(3.3); と使用することはできるか。理由も考えよ。
- クラスBunsuは、分子と分母を表わすint 型の変数を二つ持つ。このクラスは二つのint 型の引数をとるコンストラクタで分子と分母を初期化し、引数を取らないコンストラクタでは分子と分母を1に初期化する。このようなBunsuクラスを作成せよ
- クラスResourceは、使用する前にinit(Resource\*)関数を呼び出し、使用後はdestroy(Resource\*)関数を呼び出さなくてはならない。このクラスを保持するResourceHolderクラスを作成し、initとdestroyの呼び出し忘れを防ぐことのできるクラスを作成せよ