

[Prog2] Programming C++ (C6) Exercise Guide (Ex06)

10/23, Monday 3rd period.

Ex06

- ▶ クラスの実体(instance)が生成、破棄されときの挙動
 - コンストラクタ
 - デストラクタ
- ▶ メモリ領域の動的確保・解放
 - new 演算子
 - delete 演算子

コンストラクタ (constructor)

クラスのオブジェクト生成時に行う初期化処理

▶ Ex05では書いていなかったが、実際には勝手にコンパイラが適当なものを生成していた

▶ オブジェクト生成時に、明示的に1回実行

▶ 具体的な初期化処理を書きたければ

コンストラクタを明示的に書けばよい

クラス名 () {
 初期化内容;
}

main関数で、**Test x;** と生成すると、
x.a = 1; と初期化された状態になる

```
1  class Test{  
2  private:  
3      int a;  
4  
5  public:  
6      Test(){  
7          a = 1;  
8      }  
9      int get(){  
10         return a;  
11     }  
12 };  
13  
14 int main(){  
15     Test x;  
16  
17     cout << x.get();  
18     return 0;  
19 }
```

コンストラクタ

今の場合、a は毎回 a=1;で初期化される

⇒ 任意の値で初期化したければ...?

⇒ 引数を受け取るコンストラクタを作る

先程と同様に、クラス名を付けた関数で

```
Test(int val){  
    a= val;  
}
```

として、Test x(3); とすると、
x.a = 3;で初期化されたオブジェクト x が
生成される

```
1  class Test{  
2      private:  
3          int a;  
4  
5      public:  
6          Test(){//引数無し  
7              a = 1;  
8          }  
9          Test(int val){//引数あり  
10             a = val;  
11         }  
12         int get(){  
13             return a;  
14         }  
15     };  
16  
17     int main(){  
18         Test x(3);  
19         Test y;  
20  
21         cout << x.get() << endl;  
22         cout << y.get();  
23         return 0;  
24     }
```

コンストラクタ（異なる引数）

引数の個数が違うが、名前の同じ関数が...

⇒ エラーにはならない

引数無しのもの、ありのものを両方定義すると、
オブジェクト生成時の引数の数によって、
自動的に呼び出すコンストラクタが適切に選ばれる

もちろん、右の場合、
Test z(2, 5); とすると ここでは
引数2個のコンストラクタは定義されていない
⇒ エラーになる

```
1  class Test{
2      private:
3          int a;
4
5      public:
6          Test(){//引数無し
7              a = 1;
8          }
9          Test(int val){//引数あり
10             a = val;
11         }
12         int get(){
13             return a;
14         }
15     };
16
17     int main(){
18         Test x(3);
19         Test y;
20
21         cout << x.get() << endl;
22         cout << y.get();
23         return 0;
24     }
```

デストラクタ (destructor)

- ▶ クラスのオブジェクト破棄時に行う処理
- ▶ 変数のスコープが抜けると、必ず何かしら行われている。何も書かないと、勝手にコンパイラが適当なものを生成していた
- ▶ オブジェクト破棄時に、明示的に、具体的な破棄処理を書きたければデストラクタを明示的に書けばよい

(チルダ)クラス名

```
~クラス名 () {  
    処理内容;  
}
```

```
1  class Test{  
2  private:  
3      int a;  
4  
5  public:  
6      Test() { // 引数無し  
7          a = 1;  
8      }  
9      Test(int val) { // 引数あり  
10         a = val;  
11     }  
12     ~Test() {  
13         cout << "This instance was destructed." << endl;  
14     }  
15     int get() {  
16         return a;  
17     }  
18 };  
19  
20 int main() {  
21     Test x(3), y, z(10);  
22  
23     cout << x.get() << endl;  
24     cout << y.get() << endl;  
25     cout << z.get() << endl;  
26     return 0;  
27 }
```

宣言した関数を抜けて(終了して)、変数が破棄される時に実行される

クラスのメンバー関数を定義の外部に書く

- ・アクセス関数（ゲッター、セッター）
- ・コンストラクタ、デストラクタ

既に、クラス内の関数が増えてきた。

⇒ クラス定義が非常に長くなる

クラス内の関数も、プロトタイプだけを書いて、**実体を外に出す**とスッキリする

定義の外に出したときは、どのクラスのメンバー関数かを明示する必要あり

クラス名::メンバー関数名(){

...

}

ex06-test4.cc

```
1  #include <iostream>
2  using namespace std;
3
4  class Test{
5  private:
6      int a;
7
8  public:
9      Test();
10     Test(int);
11     ~Test();
12     int get();
13 };
14
15 int main(){
16     Test x(3),y,z(10);
17
18     cout << x.get() << endl;
19     cout << y.get() << endl;
20     cout << z.get() << endl;
21     return 0;
22 }
23
24
25 Test::Test(){//引数無し
26     a = 1;
27 }
28 Test::Test(int val){//引数あり
29     a = val;
30 }
31 Test::~~Test(){
32     cout << "This instance was destructed." << endl;
33 }
34 int Test::get(){
35     return a;
36 }
```

C++における動的メモリ領域確保・解放

- ▶ クラス変数も、そのスコープ(生存範囲)は、関数内で宣言されたものについては、その関数が終了するまでである
- ▶ メンバー変数に、画像データといった大きいデータを持たせるような構造のクラスを作るとメモリ消費が増大する(空間計算量が大きくなる)
- ▶ C++の場合、一般に変数の宣言はどこで行っても構わないのでオブジェクト生成のタイミングについては調整できる
- ▶ オブジェクト破棄のタイミングを調整することはできるか？

C++における動的メモリ領域確保・解放

- ▶ **new**演算子と **delete**演算子
- ▶ **C++**におけるメモリ動的確保と解放
- ▶ **C**でいうところの、**malloc()**と **free()** に対応する

- ▶ 変数はポインタ型で宣言しておいて、**new** を使うとメモリ確保と生成ができる。

ポインタ型は **x.get()** のような参照不可
アロー演算子を使って

x->get(); と書くか、

(*x).get(); と書く

- ▶ **delete** をすると、その瞬間に生成していた変数は破棄される

```
1  int main(){
2      Test *x, *y, *z;
3      x = new Test();
4      y = new Test(3);
5      z = new Test(10);
6
7      cout << x->get() << endl;
8      cout << y->get() << endl;
9      cout << z->get() << endl;
10
11     delete x;
12     delete y;
13     delete z;
14
15     return 0;
16 }
```

C++における動的メモリ領域確保・解放

▶ **new**演算子と **delete**演算子と **malloc()** と **free()** の違い

▶ **new** で生成した時、同時に **コンストラクタ**が呼ばれる

⇒ **malloc**にそのような機能は無い

▶ **delete** で生成した時、同時に **デストラクタ**が呼ばれる

⇒ **free** にそのような機能は無い

C++ で malloc や free を使うこと(要include <stdlib.h>)は
できるにはできるが、

malloc したものを **delete** したり、**new** したものを **free** したりしては
いけない

C++における動的メモリ領域確保・解放

- ▶ クラスだけでなく、通常の変数も動的確保が可能
- ▶ 16個のdouble型変数を動的確保したい場合
⇒ **double *a;** としておいて、 **a = new double[16];** と書く

ただし、
配列を開放する場合は
delete []a;

のように、配列名の前に
[] を付けて書く!!

※ **[]** を付け忘れると、
たとえコンパイルに成功したとしても
後々、深刻なバグに繋がる可能性有

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int i;
6      double *a, *sum;
7      a = new double[16]; //16個のdouble型変数の配列
8      sum = new double;  //1個の通常のdouble型変数
9
10     a[0] = 1.0;
11     for(i=1; i<16; i++){ //項a_iを求める
12         a[i] = a[i-1]*0.5;
13     }
14
15     *sum = 0.0;
16     for(i=0; i<16; i++){ //各項の和を逐次計算、ついでにそのときのa_iも表示
17         *sum += a[i];
18         cout << a[i] << endl;
19     }
20
21     delete [ ]a; //この先ではもう使わないので、配列 a を解放
22     cout << "sum: " << *sum << endl; // sum の値を出力
23     delete sum; //もう使わないので sum を解放
24
25     return 0;
26 }
```

C++における動的メモリ領域確保・解放

- ▶ オブジェクトを破棄した後なら、その変数を使いまわして新しいオブジェクトを生成することもできる

- ▶ 特にループ内で使う場合に、**多重解放に注意**

```
Test *obj;    // Testというクラスがあったとして、その実態を obj とする
              // obj を生成する(と同時にデフォルトコンストラクタを呼ぶ)
```

```
obj = new Test();
```

```
while(1){
```

```
if(obj==NULL) obj = new Test();    // obj が NULL なら 新しく生成
```

```
.....
```

```
if(obj->get_val() == 1){
```

```
    delete obj;
```


```
    obj = NULL;
```

```
}
```

```
}
```

// 何等かの条件を満たしたときに

// オブジェクトを破棄する(デストラクタも呼ぶ)



オブジェクトが破棄されたといっても、中身(メモリ上)には残骸が残っていることが多い。例えば、明示的に **NULL** を代入しておくと後々都合がよくなる。