3. C++ (3) 「コンストラクタ, デストラクタ, 代入」

♣ C++ のクラスを理解するために, コンストラクタ, デストラクタ, 代入の仕組みについて, 少し詳しく紹介する. この仕組みを良く理解していないと, 記憶制御に関してドツボにはまり, C++ が使えなくなってしまう恐れがあるので, 少し難しいが, 良く勉強して欲しい.

3.1 コンストラクタとデストラクタの起動

3.1.1 コンストラクタとデストラクタ

- 大原則

  原則として, クラスから生成される全ての変数および値に対し, 生成される時には何らかのコンストラクタが起動され, 消滅する時にはデストラクタが起動される.

  `stack s;` 等の宣言時は言うまでもなく, 関数の引数や返り値の受渡しを行う際や, `y=(a+b)*c;` の `a+b` や `(a+b)*c` 等の中間結果として生成される値に対しても, 何らかのコンストラクタが起動されている. また, これらの変数が消滅する際には, 必ずデストラクタが起動される.

  `int`, `double`, `char` などの組み込み型では, コンストラクタ/デストラクタで特に処理は行われないので, 実際には何も起こらない. しかし, クラスを自分で定義した場合には, コンストラクタ/デストラクタ内で記憶の割当てや解放等, 様々な処理が行われるため, どのタイミングでコンストラクタ/デストラクタが起動されるかを正確に把握することが要求される.

- コンストラクタ/デストラクタの種類
  - コンストラクタには, 次の 3 種類が存在する
    1. デフォルトコンストラクタ
    2. 引数を持つコンストラクタ
    3. コピーコンストラクタ … (こいつが曲者!)
  - デストラクタは, デフォルトのデストラクタ 1 種類のみ

3.1.2 コンストラクタの起動

- デフォルトコンストラクタの起動

  引数を持たないコンストラクタは**デフォルトコンストラクタ (default constructor)** と呼ばれ, 次のような場合に起動される.

  ※ クラスの例として, 1 章で定義した `Complex` を用いる

  1. 単体の宣言

     ```
     Complex a;
     ```

     と宣言すると, 制御がその宣言文に達した時点で, `a` のデフォルトコンストラクタが起動される. グローバル変数として宣言した場合には, `main` 本体の実行前にコンストラクタが起動される.

2. 配列の宣言

```
Complex a[10];
```

と宣言すると, a の要素 a[0]˜ a[9] のそれぞれに対してデフォルトコンストラクタが起動される (計 10 回!). ちなみにこれは vector でも同様. vector<Complex> a(10); でデフォルトコンストラクタは計 10 回起動される.

3. クラスのメンバー

あるクラスのデフォルトコンストラクタが起動されると, そのクラスの各メンバーについてデフォルトコンストラクタが起動される.

例えば, Complex のデフォルトコンストラクタが起動されると, そのメンバ real と imag のデフォルトコンストラクタが起動される (real と imag は組込み型のメンバーなので, この場合はコンストラクタは起動されないが).

クラス a がメンバ b と c を持ち, さらに b は b1, b2 というメンバを, c は c1, c2 というメンバを持つクラスであるとする. a に対するデフォルトコンストラクタが起動されると, b, c のデフォルトコンストラクタが起動されるが, その結果, b1, b2, c1, c2 のデフォルトコンストラクタが起動されることになる. このように, デフォルトコンストラクタの起動は下位のクラスへ次々と連動するが, これを**デフォルトコンストラクタの連鎖 (chaining)** という.

4. new による動的割当

```
Complex *a = new Complex;
```

によって Complex の領域が割り当てられた直後に Complex のデフォルトコンストラクタが起動される. 配列を動的に割り当てた場合は, 領域割り当て後, 各要素に対してデフォルトコンストラクタが起動される.

- 引数を持つコンストラクタ

  3.3 節で詳説する.

- コピーコンストラクタ

  3.3 節で詳説する.

3.1.3 デストラクタの起動

デストラクタは, 変数が消滅する際に起動される. 変数がいつ消滅するかは C 言語とほぼ同じである. C++ では, デストラクタが呼ばれるタイミングを把握するために, 変数がいつ消滅するかを正確に知っておくことが要求される.

1. automatic 変数

```
int main(void) {
  Complex a, b, c;
  ...
  return 0;
}
```

のように宣言される a, b, c のような「普通の変数」は **automatic 変数**と呼ばれる. C++ の automatic 変数は, 宣言されたところで生成され, プログラムの実行がその変数の「有効範囲」の外に出る時に消滅する. 「有効範囲」は正式には**スコープ (scope)** と呼ばれ, 簡単に言えば「その変数が宣言された {...} の中」となる. 例えば, 関数のローカル変数はその関数が return する時に消滅する. また, if 文, for 文, while 文で使う {...} も例外ではなく,

```
for (int i=0; i<10; i++) {
  Complex x;
  …
}
```

のように宣言された x に対しては, コンストラクタとデストラクタが計 10 回づつ起動される.

2. 動的変数

   new で割り当てた変数に対しては, delete で削除する時にデストラクタが起動される.

3. 式中の中間結果

   式中で生成される中間結果は, その式の評価が終ると消滅するので, そのタイミングでデストラクタが起動される. 例えば,

```
Complex a, b, c, d;
a = (b+c)*d;
```

   の代入文を実行する際に, b+c や (b+c)*d の中間結果を保持する一時的変数が作られる. (この際には何らかのコンストラクタが起動される.) これらの一時変数は代入文の実行終了とともに消滅するが, その際にデストラクタが起動される. (詳細は 3.3 節やその演習で再登場する.)

課題 3.1 [List 3.1] は, aaa, bbb, ccc と言う 3 つのクラスについて, コンストラクタ, デストラクタが呼ばれる毎にその旨を表示するようにしたものである. このプログラムを実行し, 結果を分析せよ. (実行結果の各行に対して, そのコンストラクタ, デストラクタがプログラム中のどこでどのように呼ばれたか説明せよ.)

[List 3.1]

```
 1:  #include <iostream>
 2:
 3:  class aaa の定義 [List 3.2]
 4:  class bbb の定義 [List 3.3]
 5:  class ccc の定義 [List 3.4]
 6:
 7:  int main(void) {
 8:    aaa a;
 9:    bbb x[3];
10:    aaa *p;
11:    p = new aaa;
12:    a = aaa(3,4);
13:    for (int i=0; i<3; i++) {
14:      aaa x(7,4);
15:    }
16:    delete p;
17:    ccc c;
18:    return 0;
19:  }
```

[List 3.2]

```
 1:  class aaa {
 2:  public:
 3:    int x;
 4:    int y;
 5:    aaa() {std::cout << "aaa()" << std::endl;}
 6:    aaa(int xx, int yy) {
 7:      x=xx; y=yy;
 8:      std::cout << "aaa(" << xx << "," << yy << ")" << std::endl;
 9:    }
10:    ~aaa() {std::cout << "~aaa()" << std::endl;}
11:  };
```

[List 3.3]

```
 1:  class bbb {
 2:  public:
 3:    int x;
 4:    int y;
 5:    bbb() {std::cout << "bbb()" << std::endl;}
 6:    ~bbb() {std::cout << "~bbb()" << std::endl;}
 7:  };
```

[List 3.4]

```
 1:  class ccc {
 2:  public:
 3:    aaa p;
 4:    bbb q;
 5:    ccc() {std::cout << "ccc()" << std::endl;}
 6:    ~ccc() {std::cout << "~ccc()" << std::endl;}
 7:  };
```

3.1.4 応用: 「トレースオブジェクト」

automatic 変数に対してコンストラクタ, デストラクタが呼ばれる仕組みを利用すると, プログラムのデバッグ用にトレースを行う仕組みが簡単に作れる.

1. まず, 次の [List 3.5] のようなクラス `trace` を作る. (`<string>` を include すること.)

[List 3.5]

```
 1:  class trace {
 2:    private:
 3:      std::string name;
 4:    public:
 5:      trace(const std::string& nm) {  // 文字列引数を持つコンストラクタ
 6:        name = nm;  // 受け取った文字列を記憶
 7:        std::cerr << name << " begin" << std::endl;  // ○○○ begin  と表示
 8:      }
 9:      ~trace() {  // デストラクタ
10:        std::cerr << name << " end" << std::endl;  // ○○○ end と表示
11:      }
12:  };
```

2. 使い方は, 次の例のように, 関数の本体やブロックの先頭で, その関数の名前などを引数として trace 型の automatic 変数を宣言するだけ. 先頭に宣言をおくだけで, その関数やブロックの終了まで表示される.

[List 3.6]

```
 1:  int add(int a, int b) {
 2:    trace t("add");
 3:    return a+b;
 4:  }
 5:
 6:  int main(void) {
 7:    trace t1("main");
 8:    for (int i=0; i<3; i++) {
 9:      trace t2("for-loop");
10:      int c = add(i,i*i);
11:    }
12:    return 0;
13:  }
```

課題 3.2 trace を実際に使用してみよ. [List 3.6] のプログラムを実行し, 予想と結果を比較せよ.

3.2 代入演算

♣ 代入演算は, ユーザが定義したクラスに対しても, 特別な定義や宣言なしに使用できる. しかし, **コンストラクタ内で動的割当を用いている場合等には, デフォルトの代入では大問題が発生する**ことがある. このような場合には, 代入演算をユーザ定義することが必要になる.

3.2.1 デフォルトの代入演算

クラスに対する代入演算は, 定義や宣言なしに使える. この場合, クラスの各メンバに対する代入が行われる.

例えば,

```
class Complex {
  private:
    double real;
    double imag;
  …
};
```

と宣言されたクラス Complex に対して,

```
Complex a, b;
a = Complex(3.0,4.0);
b = a;
```

と書くと, 常識的な予想通り,

```
a.real = 3.0; a.imag = 4.0;
b.real = 3.0; b.imag = 4.0;
```

となる.

クラスのメンバがクラスだった場合は, 下位の各メンバに対する代入が行われる. (**代入の連鎖 (chaining)** と呼ばれる.)

3.2.2 代入演算の定義

C++ では operator 記法を用いて, ユーザが代入演算を定義することができる.

デフォルトと同じ動作をする代入演算を省略せずにきちんと定義してみよう. 内容としては「各メンバーの代入」を書き下すだけでよいが, 記法が少しわかりにくいので, 順を追って説明する.

1. 代入演算は Complex のメンバー関数として定義する. 取り敢えず, 代入関数を assign と書くことにする. a = b; は, a.assign(b);, つまり「右辺の値を左辺の変数にセットする」関数と考える. すると, Complex の assign は次のように書ける.

   ```
   void Complex::assign(const Complex& c) {
     real = c.real;
     imag = c.imag;
   }
   ```

   一般にはデータのサイズが大きくなる可能性があるので, 値渡しではなく参照渡しを用いる. この際に, 右辺の値を書き換えることはないので, const キーワードを付ける. 本体は, 各メンバーに対する代入を並べる (c をメンバー毎にコピーする) だけである.

2. 関数名 assign の代わりに「'=' 演算子」を表す operator= を用いれば, a=b; という記法が可能になる.

   ```
   void Complex::operator=(const Complex& c) {
     real = c.real;
     imag = c.imag;
   }
   ```

3. 代入を行うだけなら void 型でもよいのだが, C や C++ では = 演算は値を持つので, それに対応する必要がある. 例えば, c=(b=a); という書き方が許され, c には, b に代入されたのと同じ値が代入される. ここで (b=a) は Complex 型の値を持っている. すなわち, 代入関数は void 型ではなく, Complex 型を返すのである.

```
Complex& Complex::operator=(const Complex& c) {
  real = c.real;
  imag = c.imag;
  return *this;
}
```

ここでも, クラスのデータが大きい場合のコピーを避けるため, 値そのものではなく参照 Complex& を返していることに注意. さて, 返す値はこのメンバー関数や real, imag の持ち主自身であるが, それは *this と書かれる. this は C++ のキーワードで, メンバー関数の持ち主へのポインタを意味する.

4. 代入関数の本体は以上. メンバー関数であることの宣言を Complex クラスの宣言に追加することを忘れないように.

```
Complex& operator=(const Complex&);
```

課題 3.3 Complex に対して, 代入演算, および左辺に右辺の値を足し込む演算 += を定義し, 動作を確認せよ. (定義の方法は代入とほぼ同様である.)

3.2.3 動的割り当てを行うクラスの代入演算

- 問題点 (1)

  次のように定義されたスタックを考える. (配列の動的割当を行うもので, 基本的に 1 章で演習したものと同じ. dump(std::std::ostream&) はデバッグ用にスタックの内部を表示する関数.

[List 3.7]

```
 1:  class stack {
 2:     private:
 3:        int sp;
 4:        int max;
 5:        int* data;
 6:     public:
 7:        stack(int sz) {sp=0; max=sz; data=new int[max];}
 8:        ~stack() {delete [] data;}
 9:        void push(int d) {assert(sp<=max); data[sp++]=d;}
10:        void pop() {assert(0<sp); --sp;}
11:        bool empty() const {return sp==0;}
12:        int top() const {return data[sp-1];}
13:        int size() const {return sp;}
14:        void dump(std::ostream&) const;
15:  };
16:
17:  void stack::dump(std::ostream& os) const {
18:     os << "max=" << max << ", ";
19:     os << "sp=" << sp << ", ";
20:     os << "data=(";
21:     for (int i=0; i<sp; i++) os << data[i] << " ";
22:     os << ")" << std::endl;
23:  }
```

デフォルトの代入演算を用いて次のようなメインを書いたとする. どのような結果が得られるだろうか?

[List 3.8]

```
 1:  int main(void) {
 2:
 3:    stack s1(5), s2(7);
 4:
 5:    s1.push(1); s1.push(3); s1.push(5);
 6:
 7:    s2 = s1;
 8:
 9:    s1.pop(); s1.pop(); s1.push(300); s1.push(500);
10:
11:    s1.dump(std::cout);
12:    s2.dump(std::cout);
13:
14:    return 0;
15:  }
```

s1 には最終的には (1 300 500) が入ることになる. 7 行目の代入を行う時点で s1 には, (1 3 5) が入っており, これが s2 にコピーされた後は s2 に操作は行われないので, 予想される結果は,

```
max=5, sp=3, data=(1 300 500 )
max=7, sp=3, data=(1 3 5 )
```

である. しかし実際には, プログラムがクラッシュするか, 実行できても

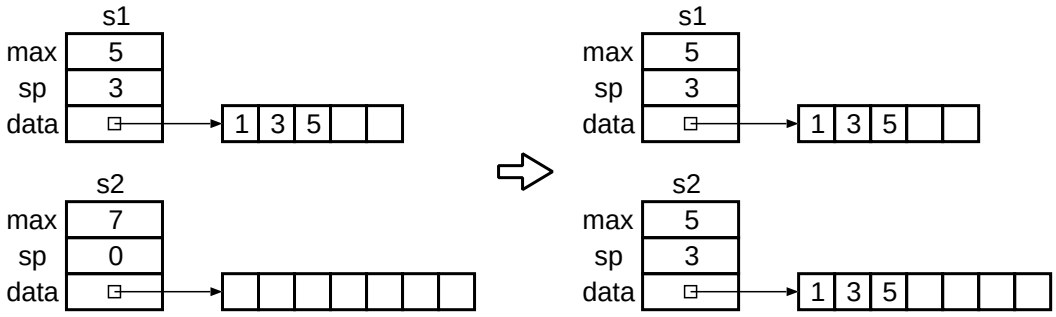
```
max=5, sp=3, data=(1 300 500 )
max=5, sp=3, data=(1 300 500 )
```

となったり, 「既に free されているデータに対して free を行った」というようなエラーメッセージが出力されるなど致命的なエラーが発生することがある.

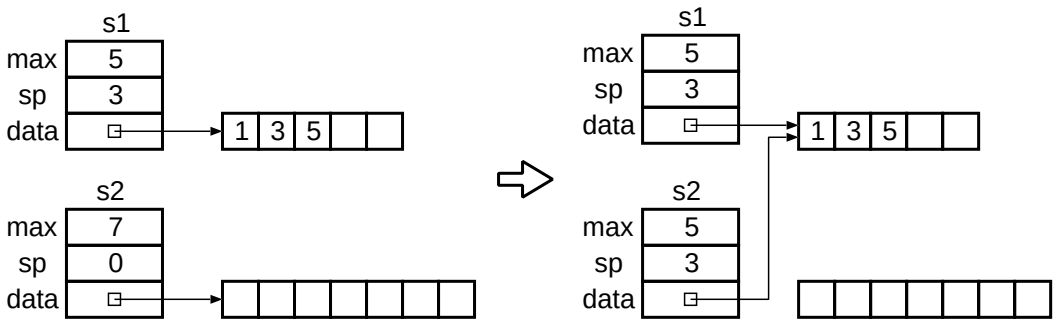
課題 3.4 [List 3.7], [List 3.8] のプログラムを実行し, どのような結果になるか確認せよ.

- 問題点 (1) の原因

  [List 3.8] の 7 行目の代入に対し, 我々は下の図のようになることを期待している.



しかし, 実際にデフォルトの代入演算で行われるのは, **各メンバーのコピーに過ぎない**. max や sp の代入は問題ないが (s2.max が 5 になってしまっていることは少し問題なのだがそれは後で議論するとして), s1.data に s2.data を代入すると, 次の図のようなことが起こる.



つまり, s2.data に代入されるのは s1.data のポインタとしての値であり, 結果 s1.data と s2.data は同じアドレスを指すようになるだけで, 動的配列の複製は行われないのである.

この後, s1 にデータ操作を行うと, s2 のデータが書き変わってしまうように見える「予期せぬバグ」が発生してしまう. (この手のバグの発見は難しい.)

ちなみに, C 言語の構造体でも, 代入ではビットレベルでのコピーしか行われないので, これと同じ問題が起こる.

更に, デストラクタの起動に際しても問題が起こる. main 終了時, s2, s1 の順序でデストラクタが呼ばれる. s2 のデストラクタが呼ばれると, s2 自身とともに, デストラクタ中の delete [] data; によってこの動的配列は解放されてしまう. 次に s1 のデストラクタが呼ばれ, data [] data; が実行された時には, この動的配列は既に存在しないので, 「既に解放されている領域を解放しようとした」というエラーになるわけである.

また, もともと s2.data で指されていた動的配列にはどこからもアクセスできず, 永遠に解放されない (典型的なメモリーリーク). このような領域は OS がしっかりしていれば, プログラムの終了時に解放されるが, ショボい OS だと使えるメモリー量が減ったり, システムが落ちたりする.

また, s2.max が 5 になってしまうのも好ましくない. せっかく大きい領域を確保しているのに, それが使えなくなってしまう. 逆に, s2.max が実際に確保したサイズ以上になってしまうと更にまずいのだが, それについては「問題点 (2)」で述べる.

- 問題点 (1) の解決

  デフォルトの代入演算が使われないように, ユーザが代入演算をきちんと定義するしかない.

  内容的には次の通り.

  1. max の値はコピーしない.
  2. sp の値はコピーする.
  3. data の指す動的配列の値をコピーする.

  プログラムは, 次のようになる.

  [List 3.9]

```
1:  stack& stack::operator=(const stack& s) {
2:    sp = s.sp;
3:    for (int i=0; i<sp; i++) {data[i] = s.data[i];}
4:    return *this;
5:  }
```

  ※ クラスの宣言にメンバー関数のプロトタイプを忘れず追加する.

```
stack& operator=(const stack&);
```

- 問題点 (2)

  動的配列を用いている場合の代入には, 更に「サイズの問題」が生じる.

スタック同士の代入 s1 = s2; において, s2 のに格納されているデータ数が s1 の確保した動的配列の大きさを上回っていた場合に動的配列のコピーを行うと, s1 の確保した領域外にデータが書き込まれ, 思わぬデータの破壊が起こる. これは必ず防止しなければならない.

防止には, これを事前に検出してエラーとするか「再配置」を行う.

再配置とは, 十分なメモリーを確保してそちらにコピーを行い, 古い領域を解放するというものである.

[List 3.10]

```
1:  stack& stack::operator=(const stack& s) {
2:    if (max<s.sp) {  // コピーし切れなかったら
3:      delete [] data;  // 古い配列を解放し
4:      data = new int[max=s.max];  //十分な大きさの配列を割り当てる
5:    }
6:    sp = s.sp;
7:    for (int i=0; i<sp; i++) {data[i] = s.data[i];}
8:    return *this;
9:  }
```

- 問題点 (3)

「自己代入」の問題として知られている.

上記プログラムではかろうじて回避されているのだが, s = s; という代入に対して, データが失われてしまうことが生じるという問題である.

もし, [List 3.10] のプログラムの 2 行目が,

```
2'  if (max<=s.sp) {
```

だったとする. max==s.sp の場合にはサイズがぎりぎりなので, 今後の処理の余裕を見てサイズ s.max で再配置を行いたい, と考えた場合にはこのようなコードが書かれる可能性がある.

ここで, 自己代入 s = s; が行われたとする. 3 行目で data の指す動的配列の解放を行うと, 実は data と s.data は同一なので, s.data の指す動的配列はこの時点で失われてしまい, 7 行目で配列のコピーを行おうとした時にはデータは残っていない. (実際には, 解放された「カス」として残っていることはあるが, それが正しいデータであることは保証されない.)

この問題の解決は簡単. 代入関数に一行「ガード」を加えるだけでよい.

代入元が同じ場合は, 代入の処理を行わなくてもよいので, そのチェックだけ行えば良い. (しかし, たったこれだけを知らない/忘れただけで悲惨な結果になることも…).

[List 3.11]

```
 1:  stack& stack::operator=(const stack& s) {
 2:    if (&s!=this) {
 3:      if (max<s.sp) {
 4:        delete [] data;
 5:        data = new int[max=s.max];
 6:      }
 7:      sp = s.sp;
 8:      for (int i=0; i<sp; i++) {data[i] = s.data[i];}
 9:    }
10:    return *this;
11:  }
```

あるいは, 冒頭に

```
if (&s==this) return *this;
```

とだけ書くだけでも良い.

課題 3.5 stack に対する代入関数を定義し, 課題 3.4 のプログラムがきちんと動くようにせよ.

- 全く別の解決法

  「代入演算を使えなくする」という方法もある.

  そもそも代入演算が必要なのかどうかを考えてみる. 代入演算の利用を想定しないのなら, 動的配列を用いていてもわざわざ代入演算を定義する必要はない.

  ただし, そのクラスを作った本人は知っていて使わなくても, 誰か他の人が知らずにこのクラスの代入演算を使って事故を起こすかも知れない. その予防として, そのクラスの代入演算を使えなくしてしまう. これは, 代入演算をそのクラスの private メンバー関数として宣言することにより行える.

```
  private:
      stack& operator=(const stack&);
```

  private のメンバーは外部からはアクセスできないので, 代入演算を使おうとするとコンパイルエラーとなって拒否されてしまう.

代入演算に関するコメント

1. コンストラクタで動的割当を行う場合には, 代入演算の定義が必要と思ってまず間違いない. 名前などに C スタイルの文字列を用いた場合にも, このような問題は生じる. この他にも様々なケースがあるので, デフォルトの代入で問題が起きないかどうか良く考えること.
2. 代入演算を定義する時は, 再配置処理, 自己代入の処理も忘れないように. 特に, 自己代入の処理は参考書などにも忘れているものが多いらしい. 知っていれば簡単なので, 代入演算を定義する時は, 自己代入のチェックを書くよう習慣付けよう.
3. 代入演算に関しては, クラスのメンバに動的配列や C スタイルの文字列が存在すると何かと面倒が生じる. 特に, 一々再配置を書くのは面倒である. これに対して, vector や string にはあらかじめ代入演算が定義されていて, 内部で再配置も行ってくれるので, デフォルトの代入演算でもあまり問題が生じないことが多い. C++ で vector や string の使用が推奨される理由の一つである.

3.3 コピーコンストラクタ

♣ 少し難しいが, 極めて重要!

3.3.1 コピーコンストラクタとは

- 2 つの `double` 型メンバ `real` と `imag` を持つクラス `Complex` と次の 4 つのコンストラクタを考える.

  1: `Complex() {real = 0.0; imag = 0.0;}`

  2: `Complex(double r, double i) {real = r; imag = i;}`

  3: `Complex(double r) {real = r; imag = 0.0;}`

  4: `Complex(const Complex& c) {real = c.real; imag = c.imag;}`

  1 はデフォルトコンストラクタである.

  2˜ 4 はいずれも「引数を持つコンストラクタ」に見えるが, このうち, 自分と同じ型への const 参照を持つ 4 は, **コピーコンストラクタ (copy constructor)** と呼ばれる「別格」のコンストラクタである. 引数を持つコンストラクタ達の中に埋没して, その存在すら知らない人も多いが, プログラムの重要なところで密かに起動されており, その理解は C++ を使いこなす上で極めて重要な意味を持っている.

3.3.2 コピーコンストラクタの起動

- コピーコンストラクタが起動される場所

  コピーコンストラクタは思わぬところで密かに起動されている. それは次の 2 点. 働きは, 代入演算とよく似ている.

  1. 初期化を伴った宣言

     次の 2 つのプログラムは, 同じに見えて異なる振舞をする.

     [List 3.12]

```
1:  Complex a;
2:  Complex b;
3:  b = a;
```

     [List 3.13]

```
1:  Complex a;
2:  Complex b = a;
```

     いずれも 1 行目では `a` に対してデフォルトコンストラクタが起動される.

     [List 3.12] の 2 行目では `b` に対してデフォルトコンストラクタが起動された後, 3 行目で代入演算が起動される.

     これに対し, [List 3.13] では, 代入演算は起動されない. 2 行目の代入文に見える文は, 実は `b` に対して `a` を引数としたコピーコンストラクタを起動する構文である. すなわち, [List 3.13] は次の [List 3.14] と等価である.

     [List 3.14]

```
1:  Complex a;
2:  Complex b(a);
```

     `Complex b = a;` のような文では, `b` を構築した後すぐに `a` の値が書き込まれるのがわかっている. このような場合には, わざわざデフォルトの初期化を行って直後に上書きするのは無駄である. `Complex` のような小さなデータ構造では大した問題ではないが, 巨大なクラスではこ

のオーバヘッドが無視できない. そこで 1 回のコンストラクタ呼出しで値の設定を行える仕組みが準備されているのである.

この理由は, 後の 2 点にも共通する.

2. 関数呼び出しにおける引数の値渡し

値渡しで関数を呼び出すと, 引数をコピーすることが必要だが, この時のコピーは代入演算ではなく, コピーコンストラクタにより行われる. 例えば,

[List 3.15]

```
Complex operator+(Complex a, Complex b) {
  Complex c;
  c.set_re(a.re() + b.re());
  c.set_im(a.im() + b.im());
  return c;
}
```

において,

```
z = x + y;
```

という呼出しを行うと, 関数の実行に先だって, x の値が a に, y の値が b にコピーされる. このコピーの際に, 一旦デフォルトコンストラクタで a と b を初期化してから代入演算で x と y を書き込むのではなく, コピーコンストラクタで一気にコピーが行われる. これは, 1. の初期化を伴った宣言の時と同じ理由による.

ただし, コピーが行われるのは値渡しの時のみで, 参照渡しの場合はデータのコピーは作られない. 例えば, 加算を次のように定義するとコピーコンストラクタは起動されない. (新たに a や b が作られるということはないので, デフォルトコンストラクタも起動されない)

[List 3.16]

```
Complex operator+(const Complex& a, const Complex& b) {
  Complex c;
  c.set_re(a.re() + b.re());
  c.set_im(a.im() + b.im());
  return c;
}
```

課題 3.6 次の [List 3.17] は, Complex のコンストラクタ, デストラクタ, 代入演算の起動に際してメッセージを出すようにしたものである. 実行結果について, どの呼出しがプログラム中のどの部分で行われたものか説明せよ. (プログラムは, ダウンロードせよ. プログラムを添付する必要はない)

[List 3.17]

```
 1:  #include <iostream>
 2:
 3:  class Complex {
 4:     private:
 5:        double real;
 6:        double imag;
 7:     public:
 8:        Complex() {
 9:           real = 0.0; imag = 0.0;
10:           std::cout << "Complex()" << std::endl;
11:        }
12:        Complex(double r, double i) {
13:           real = r; imag = i;
14:           std::cout << "Complex(" << r << "," << i << ")" << std::endl;
15:        }
16:        Complex(const Complex& c) {
17:           real = c.real; imag = c.imag;
18:           std::cout << "Complex(const Complex&)" << std::endl;
19:        }
20:        ~Complex() {std::cout << "~Complex()" << std::endl;}
21:        Complex& operator=(const Complex& c) {
22:           real = c.real; imag = c.imag;
23:           std::cout << "operator=(const Complex& c)" << std::endl;
24:           return *this;
25:        }
26:        double re() const {return real;}
27:        double im() const {return imag;}
28:        void set_re(double r) {real = r;}
29:        void set_im(double i) {imag = i;}
30:        void print(std::ostream& os) const {os << real << "+" << imag << "i";}
31:  };
32:
33:  std::ostream& operator<<(std::ostream& os, Complex& c) {
34:     c.print(os);
35:     return os;
36:  }
37:
38:  Complex operator+(Complex a, Complex b) {
39:     Complex c;
40:     c.set_re(a.re() + b.re());
41:     c.set_im(a.im() + b.im());
42:     return c;
43:  }
44:
45:  int main(void) {
46:     Complex x(3.14,2.26);
47:     Complex y = x;
48:     Complex z;
```

```
49:     z = x + y;
50:     std::cout << z << std::endl;
51:     return 0;
52:  }
```

3.3.3 デフォルトのコピーコンストラクタ

- コピーコンストラクタの省略

  コピーコンストラクタは必ずしも定義しなくても良い. 定義を省略した場合には, 代入演算と同じように, デフォルトのコピーコンストラクタが仮定される. デフォルトのコピーコンストラクタは, 各メンバに対してそのコピーコンストラクタを起動する (コピーコンストラクタの連鎖 (chaining)).

- デフォルトのコピーコンストラクタの例

  [List 3.17] のように, `double` 型のメンバ `real`, `imag` を持つにクラス `Complex` においてコピーコンストラクタの定義を省略すると, 自動的に,

  ```
  Complex(const Complex& c) {real = c.real; imag = c.imag;}
  ```

  が仮定される.

☆ この事実が, コピーコンストラクタの理解を難しくしたり, 次節のような事故の発生を起こすわけである. C++ の初心者は, コピーコンストラクタの存在すら知らないので, 何が起こっていているのかわからないし, わかってもどうやって解決したらいいかわからないのである.

3.3.4 コピーコンストラクタの定義が必要なケース

- 動的割当を行っている場合

  基本的に, 代入演算と同じ問題 (ポインタのコピーやサイズの問題) が生じる.

  [List 3.18] のプログラムを考える. 関数 `plus(stack s1, stack s2)` は, 2 つのスタック `s1` と `s2` を連結したスタックを返す関数である. (例えば, `s1 = (1 3 5)`, `s2 = (2 4 6)` のときには, `(1 3 5 2 4 6)` を返す.) 値引数と値による返しを使っているので, コピーコンストラクタでコピーが行われるが, この際にポインタは単純にコピーされるだけなので, 課題 3.5 で作成した (代入演算は定義済みの) `stack` ではエラーが生じる.

[List 3.18]

```
 1:  stack plus(stack s1, stack s2) {
 2:    int sz = s1.size() + s2.size();
 3:    stack tmp(sz), s(sz);
 4:    while(!s2.empty()) {tmp.push(s2.top()); s2.pop();}
 5:    while(!s1.empty()) {tmp.push(s1.top()); s1.pop();}
 6:    while(!tmp.empty()) {s.push(tmp.top()); tmp.pop();}
 7:    return s;
 8:  }
 9:
10:  int main(void) {
11:
12:    stack s1(5);
13:    stack s2(5);
```

```
14:    stack s(10);
15:
16:    s1.push(1); s1.push(3);  s1.push(5);
17:    s2.push(2); s2.push(4);  s2.push(6);
18:
19:    s = plus(s1,s2);
20:
21:    s1.dump(std::cout);
22:    s2.dump(std::cout);
23:    s.dump(std::cout);
24:
25:    return 0;
26:  }
```

課題 3.7 [List 3.18] のプログラムでどのようなエラーが出るか確認せよ.

- 解決法

  コピーコンストラクタを自分で定義すれば良い.

  代入演算よりは簡単である. デフォルトコンストラクタと代入演算を合わせたような形になる.

```
stack::stack(const stack& s) {
  max に s.max をコピー;
  data に動的割り当てを行う;
  sp に s.sp をコピー;
  data の指す動的配列に, s.data の指す動的配列をコピー;
}
```

課題 3.8 コピーコンストラクタを追加し, [List 3.18] のプログラムが実行できるようにせよ.

コピーコンストラクタに関するコメント

1. 上の例では, そもそも, 関数呼出の際に, 大きなオブジェクトを値で渡すのが間違っていると言う話がある. 大きなオブジェクトは, 参照で渡すのが自然である. 例題の `stack` のように, `pop` などデータを書き換えてしまう方法でしか内部のデータが読めない場合には値渡しにせざるを得ないが, `friend` 関数にすればデータを書き換えずに内部を参照することもできる.
2. 同様に, 大きなオブジェクトを値で返すインタフェースも望ましくないと言える. ただし, こちらは単純に参照返しにすると, メモリーエラーが起きるので注意. 例えば, [List 3.18] の関数 `stack plus(stack, stack)` を `stack& plus(stack,stack)` として参照返しにすると, 最後の `return s;` で計算されたスタック `s` への参照が返されるが, `s` は automatic 変数のため, この関数の実行が終了する時点で消滅してしまう. 従って, `main` の `s = plus(s1,s2);` で得られる `s` は, 既に解放された領域を参照することになり, メモリーエラーを引き起こす.
3. 動的割当を行っている場合には, 代入演算だけ定義したのでは片手落ちである. 代入演算を定義したその手でコピーコンストラクタも定義すること.
4. 代入演算と同様にコピーコンストラクタをクラスのプライベートメンバー関数として宣言してしまえば, その使用を禁止することができる.
5. デフォルトのコピーコンストラクタは連鎖するので, 代入演算の場合と同様, メンバーに `vector` や `string` を用いておけば, わざわざコピーコンストラクタをユーザ定義する必要はなく, 問題も起こりにくい.

6. コピーコンストラクタの引数は, 必ず参照渡し

```
stack::stack(const stack& s)
```

であって, 値渡し

```
stack::stack(const stack s)
```

でないことに注意!! 値渡しにすると, コピーコンストラクタへの引数渡しの際にまたコピーコンストラクタが起動されるので, コピーコンストラクタの無限再帰呼出しに陥ってしまう (はまった人は多い).

3.3.5 コンストラクタ内でのコンストラクタ呼出しの構文

- コンストラクタ内での初期化はこれまで,

  (a)

```
Complex::Complex() {real = 0.0; imag = 0.0}
```

  のように書いてきたが, これは,

  (b)

```
Complex::Complex() : real(0.0), imag(0.0) {}
```

  という構文を用いて書く方がよい.

  (b) は, 引数を持つコンストラクタを用いてクラス内のメンバを初期化するための構文である. (a) では, 本体 ({ } 内) の実行に先立ってクラスの各メンバのデフォルトコンストラクタが起動され, その後に代入が行われる. これは, 前述の通り, メンバが大きなクラスの場合無駄が多いので, コピーコンストラクタを使って 1 回で初期化したいところである. (b) はデフォルトコンストラクタの変わりに引数を持つコンストラクタを使用することを指示する構文で, クラスの各メンバーは, 本体の実行に先立って指定されたコンストラクタで初期化される.

- コンストラクタの引数を渡すこともできる.

```
Complex::Complex(double r, double i) {real = r; imag = f}
```

⇓

```
Complex::Complex(double r, double i) : real(r), imag(i) {}
```

コメント

初期化の無駄が省けるという理由以外にもエラー処理などの様々な観点から, 本節で紹介した構文が優れており, C++ ではその使用が推奨されている. 本演習を履修した諸君は是非こちらの構文を使って欲しい.

課題 3.9 これまでに作成した Complex, stack の最も新しいバージョンのコンストラクタを **3.3.5** の構文を用いたものに書き直せ. (Complex のプログラムを `3_9c.cpp` に, stack のプログラムを `3_9s.cpp` にそれぞれ添付せよ.)

3.3.6 引数を持つコンストラクタ

引数を持つコンストラクタが呼び出される場所

コピーコンストラクタ以外の引数を持つコンストラクタは, コピーコンストラクタと同様の 3 つの場面 (および 3.3.5 で紹介した引数つきコンストラクタによる初期化の構文) で起動される. ただし, 引数が明示してあるので, コピーコンストラクタよりそれがわかりやすい.

1. 初期化を伴った宣言

次では, 代入ではなく引数付きコンストラクタで初期化が行われる.

```
Complex a = Complex(1.0,2.0);
```

すなわち, 上は次と等価である.

```
Complex a(1.0,2.0);
```

2. 関数呼び出しにおける引数の値渡し

   値渡しの引数に直接引数付きコンストラクタを用いた定数を指定すると, コピーコンストラクタではなく, その引数付きコンストラクタによって引数のコピーが行われる. 前出の,

   [List 3.19]

```
Complex operator+(Complex a, Complex b) {
  Complex c;
  c.set_re(a.re() + b.re());
  c.set_im(a.im() + b.im());
  return c;
}
```

   において,

```
z = x + y;
```

   という呼出しを行うと, x と y の値がコピーコンストラクタでそれぞれ a と b にコピーされたが,

```
z = Complex(1.0,2.0) + Complex(2.3,4.5);
```

   という呼出しを行うと, 右辺の 2 つの値は, 引数付きコンストラクタで一旦値を生成した後にコピーコンストラクタで a と b にコピーするのではなく, 引数付きコンストラクタを使って直接 a と b にコピーされる.

3. 値を返す関数の返り値

   [List 3.15] を次のように書き, return 文中に引数付きコンストラクタを書くと, 返り値はコピーコンストラクタではなく, return 文に書かれた引数付きのコンストラクタでコピーされる.

   [List 3.20]

```
Complex operator+(Complex a, Complex b) {
  double r = a.re() + b.re();
  double i = a.im() + b.im();
  return Complex(r,i);
}
```

Nagisa ISHIURA