テンプレート

テンプレートとは？

- 型の定義に任意性を持たせる事ができる
  - 型毎に関数を実装する必要が無くなる
  - 汎用的な関数を設計できる
- 実際に使用する型は，オブジェクトを生成する際に，<> 内に宣言する
  - 例: Swapper<int>, Swapper<Circle>
- 標準的に使用するテンプレートのライブラリが存在する
  - STL (Standard Template Library) と呼ばれる

テンプレートの実装例

```
template <class T> // T は型を一般化した表現
class Swapper {
private:
    T tmp;
public:
    void swap (T& val0, T& val1) { //二つの変数値の入替
        tmp = val0;
        val0 = val1;
        val1 = tmp;
    }
};
```

テンプレートの使用例

```
int main () {
    int a = 1, b = 2;
    Swapper<int> swapInt;
    swapInt.swap (a, b); // a == 2, b == 1
    Circle circ0 (1,2,3), circ1(4,5,6);
    Swapper<Circle> swapCircle;
    swapCircle.swap (circ0, circ1);
}
```

テンプレートの代表例（vector）

- リスト構造を扱うテンプレートのクラス
- 任意の型の一次元データ配列を操作する
- データの追加と削除が簡単
- 配列のサイズは自動的に増減される
- 繰り返し処理が効率的に記述できる

健康データの集合の場合（再掲）

```
class HealthGroupManager {
private:
    Health *data[100]; // 最大数 100人分の健康データ
    int numStudents = 0; // 登録済みの学生数

public:
    // データの追加登録
    void setStudentData (char *n, float h, float w) {
    data[numStudents++]  = new Health (n, h, w);
  }
};
```

vectorを使用した場合

```
class HealthGroupManager {
private:
    std::vector<Health *> data; // 任意の数の健康データ
public:
    void setStudentData (char *n, float h, float w) {
    data.push_back(new Health (n, h, w));
  }
    Health* getData (int n) {   // n番目のデータの参照
        return data.at (n);
    }
    int getNumStudents () { // 全登録数（numStudentsは不要となる）
        return data.size ();
    }
    void removeAllData () { // 全データを削除
        data.clear();
    }
};
```

反復子

```
float HealthGroupManager::getAverageBMI () { //vector用いない場合
        float sumBMI = 0.0;
        for (int i = 0; i < numStudents; i++)
                sumBMI += data[i]->getBMI();
        return sumBMI / (float) numStudents;
}
float HealthGroupManager::getAverageBMI () { // vector 用いる場合
        float sumBMI = 0.0;
        std::vector<Health*>::iterator it; // 反復子の宣言
        for (it = data.begin(); it != data.end(); it++)
                sumBMI += it->getBMI();
        return sumBMI / (float) data.size();
}
```

・要素の削除(erase)、挿入(insert)の操作等にも、反復子を用いる

例外処理

例外処理の使い方

- 例外処理の機能を組込むと，実行時エラーの解析が容易になる
- エラーや例外が発生する可能性のあるコード部分を try {...} で囲む
  - その中で throw 文によって投げられたメッセージの値を catch 文が受け取る
  - その後に続く処理を記述できる

例外処理機構の構造

```
try {
    // 例外を発生するコード部分（throw によって例外メッセージを送信）
} catch ( 例外メッセージ ) {
    // 例外の種類に依存する処理を実行するコード
}
```

例外処理の実装例

```
Point p;
try {
        p.setPosition(1, -3);
} catch (const char *errMsg) {
// 例外処理の結果を表示
    std::cout << "Error in Point  : " <<
errMsg; return(0);
}
```

エラーを報告する機能の設計例

```
class       Point {
private:
    int x, y; // 2次元座標は，第1象限でのみ定義される
public:
    Point() { x = y = 0; }
    void setPosition(int _x, int _y) throw(const char *) {
        if (_x < 0 || _y < 0) {
            throw "Negative value NOT permitted";
        }
        else {
            x = _x; y = _y;
        }
    }
    ...
```

例外処理のクラスの設計例

```
#include <exception>
#include <string>
// std::exception クラスを継承して作成する
class          PointException : public std::exception {
private:
    std::string e_msg; // エラーメッセージ

public:
    PointException(const std::string& msg) : e_msg(msg) {};
    void print() const {
        std::cerr << "Error in Point class : " << e_msg;
    }
};
```

実装方法の変更

```
class        Point {
    void setPosition(int _x, int _y) throw(PointException) {
        if (_x < 0 || _y < 0) {
            throw PointException("Negative value NOT permitted");
        }
...
```

```
Point p;
try {
        p.setPosition(1, -3);
} catch (PointException pe) {
    pe.print(); return(0);
}
```