Template

Template

- Introduce a universal rule to type definition
  - No need to implement function for each type
  - Can design universal functions
- Define the type inside <> used in runtime
  - Swapper＜int＞, Swapper＜Circle＞
- Standard library of template exists
  - STL (Standard Template Library)

Sample code

```
template <class T> // T: universal notation of type
class Swapper {
private:
    T tmp;
public:
    void swap (T& val0, T& val1) { //swap val0 and val1
        tmp = val0;
        val0 = val1;
        val1 = tmp;
    }
};
```

Usage of template

```
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);
}
```

Standard library (vector）

- Template class for managing a list structure
- Operate arbitrary types of one dimensional array
- Supply add / delete operations of instances
- Size of array is automatically adjusted
- Iterative processes can by efficiently coded

Healthy data class without template

```
class HealthGroupManager {
private:
    Health *data[100];// 100 students can be operated at MAX
    int numStudents = 0; // number of registered students

public:
    // Register (addition) of a new data
    void setStudentData (char *n, float h, float w) {
    data[numStudents++]  = new Health (n, h, w);
  }
};
```

with a template of vector

```
class HealthGroupManager {
private:
    std::vector<Health *> data; // can operate arbitrary number
public:
    void setStudentData (char *n, float h, float w) {
    data.push_back(new Health (n, h, w));
   }
    Health* getData (int n) {  // get n-th data
        return data.at (n);
    }
    int getNumStudents () { // total number of registration
        return data.size ();
    }
    void removeAllData () { // remove all registered data
        data.clear();
    }
};
```

Use of iterator

```
float HealthGroupManager::getAverageBMI () { // without vector
    float sumBMI = 0.0;
    for (int i = 0; i < numStudents; i++)
        sumBMI += data[i]->getBMI();
    return sumBMI / (float) numStudents;
}
float HealthGroupManager::getAverageBMI () { // with vector
    float sumBMI = 0.0;
    std::vector<Health*>::iterator it; // define iterator
    for (it = data.begin(); it != data.end(); it++)
        sumBMI += it->getBMI();
    return sumBMI / (float) data.size();
}
```

・ Iterator also can be used in erasing or inserting instances

Exception handling

Usage of exception handler

- Makes error analysis easy
- Region of code is enclose with try {…}, if it possibly causes errors or exceptions
  - catch statement receives messages sent by throw statement followed by accompanying processes

Structure of Exception handling

```
try {
    // statement that produces exception
    // with messages sent by throw operator
}  catch ( received exception message ) {
    // execute statement depending
    // on the type of exception
}
```

Sample code

```
Point p;
try {
        p.setPosition(1, −3);// this function
                                                // throws exception
} catch (const char *errMsg) {
// Display the result of exception handle
    std::cout << ”Error in Point  : ”  << errMsg;
    return(0);
}
```

Sample code for throwing an error

```
class           Point {
private:
    int x, y; // 2D coordinates is define in x>=0 && y>=0
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;
        }
    }
    …
```

Making my exception handler class

```
#include <exception>
#include <string>
// inherit the class of std::exception
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;
    }
};
```

Throw/catch an exception class

```
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);
}
```