2I プログラム設計　後期期末まとめ　レポート

番号　　6　　氏名　糸川倫太朗　.

例外処理：

なにか問題が起こったら例外と呼ばれるオブジェクトを throw する。エラー発生の後処理は例外オブジェクトを投げられた側で catch して処理をする。この時一致する型が存在しなかった場合は catch しない。

例外が発生した時点で例外処理に飛ばされる。

クラス内では例外のチェックと throw のみを記述。クラスの利用側で例外の処理(catch)を記述する。

std::exception はすべての例外オブジェクトの基本クラス。

```
#include <stdexcept>
```

catch を乱立させるのもイケてないが std::exception を用いればポリモフィズムを利用してまとめることが可能。

名前空間：(1)

使いたい識別子(変数名、関数名、クラス名など)が既に使われていて衝突が起こったときの回避策。ネストをすることが可能なスコープを定義する。(国/都道府県/市区町村/市名)的な。

ある名前空間内の変数や関数、クラスを使う場合はスコープ解決演算子を用いる。

using ディレクティブを用いるとそのブロック内(for / if / while { ここの中 })のスコープで名前空間が適用される。

string：(5)

文字列を簡単に扱える。STL の特徴を持つ。vector<char>が近い。

ストリーム：(2)

データの流れのこと。入ってくるデータのことを入力ストリームといい、出ていくストリームを出力ストリームという。

ストリームからの入力：抽出演算子(operator>>())、ストリームへの出力：挿入演算子(operator<<())

ストリームの状態確認

.good()	正常であるか？
.eof()	ストリームの終端に達したか？
.fail()	エラーが発生したか？
.bad()	回復不可能なエラーが発生したか？

ストリームの種類

ストリーム名	入力ストリーム	出力ストリーム
基本クラス	istream	ostream
iostream	cin	cout, cerr, clog
fstream	ofstream	ifstream
sstream	stringstream	

抽出演算子・挿入演算子オーバーロード：(3)

返り値の型はストリームの基本クラスの参照型にし、friend 関数として定義する。ストリームで弄るデータを istream は参照、ostream は const 参照を用いる。返り値は stream の参照型。

区切り文字：

区切り文字には空白(‘ ‘)、水平タブ(‘¥t’)、垂直タブ(‘¥v’)、改行(‘¥n’)、書式送り(‘¥f’)、復帰(‘¥r’)。

operator>>を使うとその先の型へ自動で合わせてくれる。

文字の一行読み込み：

.getline()	(char[], sizeof(char[]))	(char[], sizeof(char[]), ‘¥区切り’)
getline()	(istream, string)	

文字の一文字読み込み：

.get(char) <- 1 文字入力

.put(char) <- 1 文字出力

書式の指定：(4)

.width(int) <- 出力幅

.fill(char) <- 出力幅を char で満たす

.precision(int) <- 浮動小数点の精度

.setf(fmtflags) <- 書式設定

マニピュレーター：(4)

```
#include <iomanip>
```

fmtflags

dec	整数の入出力を 10 進数で行う。
hex	整数の入出力を 16 進数で行う。
oct	整数の入出力を 8 進数で行う。
showbase	整数の出力時に先頭に基数を表す出力(8 進数だと 0, 16 進数だと 0x)を追加する。
scientific	浮動小数点数の出力表記を科学表記(1234567e+01)で行う。
fixed	浮動小数点数の出力表記を固定表記(12345670)で行う。
showpoint	浮動小数点数の出力のときに必ず小数点を必ず出力する。
showpos	負でない数の出力時に+記号を出力する。
internal	中央揃え。埋め文字は出力の内側に追加する。出力の内側に追加する適切な場所がなければ right と等価。
left	左寄せ。埋め文字を出力の右側に追加する。
right	右寄せ。埋め文字を出力の左側に追加する。
adjustfield	left | internal | right
basefield	oct | dec | hex
floatfield	scientific | fixed

stream のメンバ関数とマニピュレータの対応表

stream のメンバ関数	マニピュレータ
width	setw
fill	setfill
precision	setprecision
setf	各 fmtflags

書式設定でもマニピュレータでも幅の設定は operator<<でリセットされる。

静的メンバ：(6)

クラスオブジェクトのメンバに static 指定したもの。寿命はクラスに関連付けられる(オブジェクト非依存)。クラスのメンバなのでメンバ関数から呼び出すことが可能。ただし、静的メンバ関数から通常のメンバへはアクセスできない。通常のメンバ関数で静的メンバを用いることは可能。使用する際は global スコープでの初期化が必要。

コマンドライン引数：(5)

CUI で与える命令を表す文字列、main 関数に引数を持たして使用する。

```
int main(int argc, char* argv[]) {}
```

argc にはコマンドライン引数 argv には各要素で与えられたコマンドライン引数で与えられた C 文字列が格納される。

re-taro:~/ProgrammingDesign$ ./a.out in.txt out.txt

a.out などもコマンドライン引数として含む。

デザインパターン：

過去のソフトウェア設計射が発見し編み出した設計ノウハウを蓄積し、名前をつけ再利用しやすいように特定の規約に沿ってカタログ化したもの。オブジェクト指向では GoF のデザインパターンが有名。

Singleton：(6)

あるクラスに対してインスタンスが一つしか存在しないことを保証し、それにアクセスするためのグローバルな方法を提供する。->リソース管理のプログラムに有効。(メモリが改善される)

static Factory Method：

オブジェクトを生成する静的メンバ関数を持つクラスを作成し、引数に enum class を渡すことでオブジェクトを生成するデザインパターンのこと。クラス感の結合度が下がるためクラスの変更に対して修正箇所が減る。基本クラスのインターフェイスが重要になる。

名前空間(1)

```
namespace paper {
  namespace A4 {
    void size() {cout << "A4" << endl;}
  }
  namespace A5 {
    void size() { cout << "A5" << endl;}
  }
}
int main() {
  using namespace paper;
  A4::size();
  {
    using namespace A5;
    size();
  }
}
```

fstream(2)

```
int main(void) {
  int n;
  float w;
  float total = 0.0f;
  ifstream in;
  in.open("data.txt");
  while(!in.eof()) {
    in >> n >> w;
    total += static_cast<float>(n*w);
  }
  ofstream out("output.txt");
  out << total;
}
```

入出力ストリームのオーバーロード(3)

```
class Point {
private:
  int x, y, z;
public:
  friend istream& operator>>(istream& in, Point& p) {
    in >> p.x >> p.y >> p.z;
    return in;
  }
  friend ostream& operator<<(ostream& out, const Point& p) {
    out << p.x << ' ' <<p.y << ' ' << p.z;
    return out;
  }
}
int main(void) {
  Point p;
  cout << "入力:";
  cin >> p;
  cout << "出力:" << p << endl;
}
```

書式設定(4)

```
int main(void) {
  cout << setw(4) << setfill('+');
  for (int i = 0; i < 3; i++)
    cout << i;
  cout << endl;
  for (int i = 0; i < 3; i++)
    cout << setw(4) << setfill('+') << i;
  cout << endl;
  cout.setf(ios::showbase);
  cout.setf(ios::hex, ios::basefield);
  cout << 256 << endl;
}
```

コマンドライン引数(5)

```
int main(int argc, char* argv[]) {
  for (int i = 1; i < argc; i++) {
    string str(argv[i]);
    reverse(str.begin(), str.end());
    cout << str << endl;
  }
}
```

Singleton(6)

TextureManager.h

```
#include <string>
#include <unordered_map>
#include <SFML/Graphics.hpp>
class TextureManager {
  std::unordered_map<std::string, sf::Texture> textures;
  TextureManager();
  ~TextureManager();
public:
  sf::Texture* get(const std::string& filename);
  static TextureManager* getInstance();
  TextureManager(const TextureManager&) = delete;
  TextureManager& operator=(const TextureManager&) = delete;
  TextureManager(const TextureManager&&) = delete;
  TextureManager& operator=(const TextureManager&&) = delete;
};
```

TextureManager.cpp

```
TextureManager* TextureManager::getInstance() {
  static TextureManager instance;
  return &instance;
}
sf::Texture* TextureManager::get(const std::string& filename) {
  auto it = textures.find(filename);
  if (it == textures.end())
    textures[filename].loadFromFile(filename);
  return &textures[filename];
}
```

Ball.cpp

```
void Ball::init() {
  /*****************************************************************/
  texture = TextureManager::getInstance()->get("bomb.png");
  /*****************************************************************/
}
```

良いプログラムのために：

- プログラムの異常終了や強制終了はあってはならない->運用システムのコードの 7 割はエラー処理
- using の多用は避ける->どの名前空間かの混乱を避ける
- ヘッダファイルでのグローバルスコープでの using は避ける->そのヘッダファイルインクルードした先全体で影響を受ける
- 扱う文字コードに注意し、シングルバイト文字なのかワイドバイト文字なのかを適切に使う ->wchar_t を使うと多言語文字も扱える シングルバイト->1byte マルチバイト->2byte~ ワイドバイト->16bit
- 静的メンバへのアクセスはクラス名::静的メンバとしたほうが良い->オブジェクト.静的メンバだと静的メンバであることがわからない
- 独立性の高いクラスを設計するべし->分散並行開発がしやすくなる
- 自分の能力に過信するな->大体の場合先人が定番のデザインやライブラリを用意してくれてるからそれを使え
- 良い開発者は、開発の本質を把握し、それらを解決する方法を選択し、選択した方法を活用する力が必須