### C++によるオブジェクト指向演習　　　　　　 手塚太郎

### 1 オブジェクト指向の概略

本演習ではプログラミング言語C++を用いてオブジェクト指向プログラミングを修得することを目的とする。

**一部の節の見出しには右上に「†（ダガー）」が付けられている。**これらは発展的な内容であるので、時間が足りない場合は飛ばしてよい。後から戻ってきてやってみても良い。また、問題の終わりにある記号「■」はそこまでが問題の範囲であることを示している。

オブジェクト指向（Object-oriented）とは、実世界における「もの」に相当する 「オブジェクト」を構成単位としてプログラムを記述していく枠組みである。本演習で取り扱うC++はオブジェクト指向言語 （Object-oriented programming language）の一つである。C++の基本的な式、制御構造、演算子などの構文はC言語に類似しているが、オブジェクト指向に基づいている点が大きく異なる。

オブジェクト指向はプログラムの可読性や再利用可能性を高めるために提案され、現代のほとんどのプログラミング言語（Java、Ruby、Python 等々）に取り込まれている考え方である。オブジェクト指向は大規模なプログラムをクラスと呼ばれるモジュールに分解して構築していくため、大人数で分業してプログラミングを行う際に特に効果を発揮する。また、フレームワークのように拡張可能なプログラムを作る際にも重要となる。

オブジェクト指向における主要な概念として、**「クラスとインスタンス」「フィールドとメソッド」「継承」**が挙げられる。これらはそれぞれ数学における「集合と要素」「関数」「部分集合」の概念に対応する。

**クラス(class)**はC言語における型や構造体を発展させたものであり、**インスタンス(instance)**とはクラスから生成される具体的なデータ、すなわち変数である。実世界になぞらえるなら、クラスが鋳型、インスタンスが製品といった関係にある。英語で "for instance" といえば「たとえば」という意味になるが、インスタンスは英語で「具体的」という意味であり、クラスは（学校におけるクラスのように）「集まり」というような意味である。クラスは鋳型であるが、特定の鋳型によって作られた（具体的な）インスタンスたちは共通の性質を持った集まりとなるため、それが「クラス」という名前の由来であるとも捉えられる。

プログラム中でクラスを定義する際、そのクラスに属するインスタンスからのみ実行できる固有の関数を記述しておく。これを**メンバ関数(member function)**または**メソッド(method)**と呼ぶ。メソッドはCの関数と違ってどこからでも呼ぶことができるわけではなく、常にインスタンスを表す変数やクラスを通してしか呼べない 。一方、クラスにおいて定義され、インスタンスが属性として値を持つことができる変数を**メンバ(member)** あるいは**フィールド (field)**あるいは**プロパティ(property)**と呼ぶ。これはCにおける構造体のメンバに対応する。

実世界に存在する「もの」は通常、属性と機能を持っている。たとえば自動車であれば「車体の色」「排気量」といった属性に加えて、「加速する」「減速する」という機能がある。プログラミングにこの考え方を持ち込み、属性（フィールド）や機能（メソッド） を持つクラスを多数定義し、それらを連携させる形でプログラムを書いていくのがオブジェクト指向言語の特徴である。

**継承(inheritance)**とはひとつのクラスのフィールドとメソッドを継承させつつ、新たなフィールドとメソッドを付け加えた新しいクラスを作成することを指す 。すなわち最初に一般的なクラス（基本クラス、スーパークラス）を作っておき、それに様々な機能を付け加えることでより個別的なクラス（派生クラス、サブクラス）を作り出す。 基本クラスと派生クラスの関係はいわば「自動車」と「トラック」の関係に類似している。「自動車」に具体的な属性と機能を付け加えて得られるのが「トラック」であるが、トラックは自動車の持つ属性と機能をすべて継承している。また、生物の進化にも類似している。たとえば陸上動物は皆、肺呼吸の能力（機能）を持つが、鳥類はそれを継承しつつ、 飛ぶ能力（機能）も持っている。

これらの概念によってプログラムの書きやすさや読みやすさ、再利用性が高まっているのがオブジェクト指向言語の利点である。本演習ではC++プログラミングを通してオブジェクト指向の考え方に習熟することを目的としている。

#### 2 C++におけるソースファイルとコンパイル

C++のプログラムはCと同様、以下に示す手順で実行できる。

1. ソースファイルをEmacs等のエディタで作成する。
2. ソースファイルをシェル上でg++コマンドでコンパイルし、実行可能ファイル を作成する。
3. 実行可能ファイルを実行する。

C++のソースファイルには拡張子".cpp"をつけるのが慣例である。C言語やRubyのプログラム作成と同様に、EmacsやMacのsublime、WindowsのMeadowなどのテキストエディタを用いて作成すればよい。

g++によるコンパイル時にエラーが出た場合には、ソースコードのどの行でエラーが生じたかが表示されているので、その行を見て確認すること。

本授業で使用するサンプルプログラムはGitHubにアップロードしている。

https://github.com/tarotez/cppintro

また、以下の場所にも置いている。

/home/tezuka.taro.gm/lec/cpp/

たとえば以下のようにコピーしてくることができる。コマンドを入力する際、スペースの位置に注意すること。mkdirはディレクトリを作成するコマンド(mkdir)、cpはファイルをコピーするコマンド(copy)、cdはカレントディレクトリを移動するコマンド(change directory)である。

mkdir cpp

cp /home/tezuka.taro.gm/lec/cpp/\* cpp

cd cpp

簡単なC++プログラムsample1.cppを以下に示す。

/\* sample 1 \*/

#include <iostream>

using namespace std;

int main()

{

string name;

cout << "お名前は？\n";

cin >> name;

cout << "こんにちは、" << name << "さん！\n";

}

　#include <iostream> は標準入力（キーボード入力）std::cin、標準出力（画面出力）std::cout、文字列を格納するクラスであるstd::stringなどを使えるようにするためのinclude文である。

using namespace stdはcoutとcinとstringが含まれるstdという**名前空間（name space）**を使うことを示し、 std::coutやstd::inやstd::stringと書かずに済ませるためのものである。名前空間とは互いに関連するクラスや関数をまとめたグループを表す。stdはstandardの略であり、標準入出力や文字列をまとめた名前空間を表している。「::」は**スコープ解決演算子(scope resolution operator)**と呼ばれ、助詞の「の」に相当するものと考えればよい。

配列を用いて文字列を実現するC言語と異なり、C++では文字列を扱うクラスstringが用意されている。 Cで使用する配列からなる文字列では==によって文字列の比較を行えないが、stringクラスのインスタンスでは可能である。stringクラスが持つプロパティやメソッドの一覧はウェブページなどで参照できる。「c++ string API」で検索するとよい。APIとはApplication Programming Interfaceの略であり、アプリケーションプログラムを作成する際に利用できる機能の一覧という意味である。Stringに限らず、多くのクラスに関してそれが持つプロパティやメソッドの一覧はウェブ上で公開されている。

　cout << "What's your name?\n"; における「<<」は**挿入演算子(insertion operator)**と呼ばれ、右側の文字列をcout（標準出力）に流し込んでいる。また、cin >> nameにおける「>>」は**抽出演算子(extraction operator)**と呼ばれ、cin（標準入力）を文字列（string型の変数）nameに流し込んでいる。挿入演算子と抽出演算子の左側に来るのは**ストリーム(stream)と**呼ばれる文字やデータの流れを表す変数でなくてはならない。標準出力coutや標準入力cinはストリームの例である。文字列を表す変数を挿入演算子や抽出演算子の左側に置くことはできない。

　sample1.cppをコンパイルし、実行せよ。ただし行頭の「%」はコマンドプロンプトであり、入力しなくてもよい。

% g++ sample1.cpp

% ./a.out

#### 3 クラスとインスタンス

**クラス（class）**とはデータ（属性）とそれに対する処理（操作）をひとまとめにして定義した型である。関連するデータ処理機能をひとつにまとめたモジュールとして捉えることもできる。属性は**フィールド（field）**、**プロパティ（property**）、または**メンバ（member）**と呼ばれる。処理は**メソッド（method）**または**メンバ関数（member function）**と呼ばれる。クラスはCにおける構造体にメソッドが追加されたものと考えればよい。属性をメンバと呼ぶのは構造体における呼び方に由来している。

プログラムの実行時にはクラスからオブジェクトを生成し、生成したオブジェクトがメソッドを実行する。生成したオブジェクトのことをクラスに対して**インスタンス（instance）**と呼ぶ。つまり、クラスとはインスタンスを生成する雛形（鋳型）のようなものである。実際、構造体が型になるのと同様、クラスも型の一種である。インスタンスを生成し、処理を実行するC++ プログラムを以下に示す。なお、Cと同様、1行単位のコメントは 「//」の後に書き、何行にも渡るコメントは「/\*」と「\*/」で囲う。

　以下のsamplecppはpersonというクラスを定義し、そのインスタンスからなる配列friendsを使う例である。

/\* sample 2 \*/

#include <iostream>

using namespace std;

class person{

public:

string name;

};

int main()

{

person friends[3];

for(int i=0; i<3; i++){

cout << i + 1 << "番目の名前は？";

cin >> friends[i].name;

}

cout << "\n名前リスト:\n";

for(int i=0; i<3; i++){

cout << friends[i].name << "\n";

}

}

C++ではキーワード（予約語）classによりクラスを定義し、その後にフィールドとメソッドを定義する。メソッドはプロトタイプのみを書いておくことが可能である。プロトタイプは定義を別の所で行うという宣言であり、それを使ったコード例はのちに示す。

public:という行の後に書いたフィールドやメソッドのみ、他のクラスから見ることができる。public:という行よりも前に定義したフィールドやメソッドはprivateとなり、そのクラス自身（およびそのフレンド関数）でしか参照できない。これは**カプセル化（encapsulation）**という考え方に基づいた場合、クラス内部でしか使われないフィールドやメソッドは外から（たとえばそのクラスを使う他の開発者から）見えないようにした方が良いためである。

上記のコードではpersonは人物を表すクラスであり、stringクラスのインスタンスであるnameというフィールドを持つ。フィールドは構造体におけるメンバに対応する。構造体のメンバをchar\* name;と定義した場合、char\*が型でnameが変数名である。オブジェクト指向でフィールドをstring name;と定義した時、stringがクラスでnameがインスタンス名である。

personクラスにはメソッドが定義されていないので、public:という行が入っている以外は構造体の定義とまったく同じであることに注意せよ。実際、C++では構造体にもメソッドを定義できるため、両者の違いは構造体の場合はメンバ（フィールド）とメソッドへのアクセスがデフォルトでpublic（公開）、クラスの場合はフィールドとメソッドへのアクセスがデフォルトでprivate（非公開）という点だけである。

main関数ではpersonクラスの配列であるfriendsを宣言している。C++ではインスタンス変数の宣言を行うと自動的にインスタンスが生成され、その変数に格納される。friends[0]、friends[1]、ならびにfriends[2]がそれぞれpersonクラスのインスタンスである。

**【基本課題1】**

personクラスに新たなフィールドemail（メールアドレス）をpublicとして追加し、main関数内でnameに続いて登録されるようにせよ。また、出力時に「名前：メールアドレス」というフォーマットですべての名前とメールアドレスが表示されるようにせよ。■

#### 4 メソッド （メンバ関数）

メソッドの定義はクラス定義の中に書くこともできるが、そうでない場合、クラス定義の中で**プロトタイプ（prototype）**のみを記述し、クラス定義の外で「クラス名::メソッド名」で始まるブロックを使って定義する。「::」はスコープ解決演算子である。

プロトタイプはCでも使われ、関数の実際の定義に先立ってその引数と戻り値の型をあらかじめ示しておくことを意味する。コンパイラはプログラムを上から読んでいくので、関数の引数と戻り値の型さえ分かっていれば、まだ定義されていない関数についても型の不一致が生じていないかを確認できる。不一致があった場合はコンパイルエラーを返して停止する。

以下のコードsample3.cppではaddressbookというクラスを定義し、そこにpersonクラスの配列friendsを格納している。

/\* sample 3 \*/

#include <iostream>

using namespace std;

class person{

public:

string address;

string name;

};

class addressbook{

person friends[100];

int numberOfFriends;

public:

addressbook();

void listFriends();

void addFriend(string name);

};

// constructor

addressbook::addressbook(){

numberOfFriends = 0;

}

void addressbook::addFriend(string name){

friends[numberOfFriends].name = name;

numberOfFriends++;

}

void addressbook::listFriends(){

for(int i = 0; i < numberOfFriends; i++){

cout << friends[i].name << "\n";

}

}

int main()

{

addressbook abook;

string name;

while(1){

cout << "住所録に登録する名前を入力してください（終了するにはquitと入力してください）: ";

cin >> name;

// cout << "name.length() = " << name.length() << "\n";

if(name == "quit"){break;}

abook.addFriend(name);

}

cout << "\n名前リスト:\n";

abook.listFriends();

}

上記プログラムではmain関数の中でaddressbookクラスのインスタンスabookを生成し、メソッドaddFriendを使って新たな要素をfriends配列に格納し、最終的にlistFriendsメソッドでその中身を出力させている。

listFriendsメソッドとaddFriendメソッドはaddressBookクラスの定義の中ではプロトタイプのみを記述している。これらのメソッドの実際の定義はクラス定義の外でaddressbook::listFriendsならびにaddressbook::addFriendに続いて書いている。

Cと同様、C++における各変数は**スコープ（scope）**を持つ。スコープとは、その変数を単純名（名前のみ）で参照できる範囲である。インスタンス変数のスコープは宣言されたクラスであり、メソッド引数のスコープは宣言されたメソッドである。ローカル変数のスコープは、それが宣言された位置から始まり、それが宣言されたブロック（中括弧で囲まれた範囲）の終りまでとなる。上記プログラムではfriendsとnumberOfFriendsは**インスタンス変数（instance variable）**であり、addFriendメソッドで使われているname は**メソッド変数（method variable）**である。nameに格納されているデータはaddFriendメソッドの実行が終わった時に消えるが、friendsとnumberOfFriendsに格納されているデータはabook インスタンスが残っている間は消えない。

#### 5 オーバーロード

関数に与える引数の型の並びは**シグネチャ（signature）**と呼ばれる。実行環境は関数名とシグネチャを見てどのメソッドを呼ぶかを判定する。シグネチャが異なるが名称が同じである関数を複数定義することを**オーバーロード（overload）**と呼ぶ。シグネチャが異なれば、同一クラスに対して同じ名前を持つメソッドを多重に定義することができる。たとえば次の2つの関数はオーバーロードの関係にある。前者ではstringクラスのインスタンスnameを受け付け、後者ではchar型へのポインタnameを受け付けている。これによって同じ名前の関数で2つの種類のデータに対応することができる。

void addFriend(string name) { ... }

void addFriend(char\* name) { ... }

以下は絶対値を求める関数absをint型とdouble型の両方について定義する例である。

int abs(int x){ ... }

double abs(double x){ ... }

上記の例の場合、引数の数は同じであるが、引数の数がそれぞれの定義で異なってもよい。

**【基本課題2】**

sample2.cppのmain関数の中でaddressbookにnameとaddressの両方が登録されるようにせよ。これにはaddFriendメソッドに対するオーバーロードを使い、addressbook::addFriend(string name, string address)のように2つの引数を取るようにしたものを加えればよい。追加されたメソッドについて、addressbookクラスの定義の中でプロトタイプ宣言も行うこと。名前が同じであるがシグネチャが異なる メソッドが複数あった場合、それぞれについてプロトタイプ宣言が必要である。■

**【基本課題3】**

addressbookクラスにメソッドfind(string query)を追加し、引数として与えたquery（クエリ）を使ってfriendsの各要素のnameフィールドを検索し、戻り値としてその氏名を持つ人物の情報がpersonクラスのインスタンスとして返されるようにせよ。これを実装するにはlistFriendsを参考にするとよい。findメソッドをmain関数の中で使用し、ユーザが入力した氏名に対してその人物の住所が出力されるようにせよ。クエリに一致する要素が見つからなかった場合もfindメソッドはpersonクラスのインスタンスを返さなくてはならないので、これにはたとえばperson no\_one; return no\_one;のようにインスタンスを生成して返せばよい。 ■

#### 6 コンストラクタとデコンストラクタ

クラスと同じ名称を持つメソッドは**コンストラクタ（constructor）**と呼ばれる。コンストラクタはインスタンス生成時に自動的に実行されるため、初期化に用いる。この場合、addressbook::addressbookとそれに続くブロック（中括弧に囲まれた部分）がコンストラクタの定義であり、ここでnumberOfFriendsを0に初期化している。コンストラクタは初期化を行うのが目的であるため、戻り値は返さない。ゆえにも他のメソッドと違い、メソッド名の前に戻り値の型を書く必要はない。つまりvoid addressbook::addressbookでなくaddressbook:addressbookだけでよい。

**デコンストラクタ（deconstructor）**はコンストラクタの逆で、インスタンスを破棄する時に行われる処理を記述する。デコンストラクタはクラス名の前に「~」を付けた関数として定義される。たとえばaddressbookクラスの場合、~addressbookというメソッドを定義すればデコンストラクタになる。デコンストラクタはたとえばインスタンスが利用しているメモリ領域をfreeによって解放するのに使える。

　メソッドの定義が数行で収まる場合はクラス定義の中で書いてしまうことがあるが、この書き方はコンストラクタとデコンストラクタの定義で多用される。

　コンストラクタは関数であるので、引数を付けられる。たとえばインスタンスのプロパティに初期値を与えて生成することができる。たとえば住所録クラスaddressbookのコンストラクタがint型の変数を引数として持つ時、int型の変数numberを使い、以下のようにabookというインスタンスを生成できる。

addressbook abook(number);

#### 7 newとdelete

Cではmallocを使ってメモリを割り当て、freeを使ってメモリの解放を行った。C++ではこれに加え、newを使ったメモリの割り当て、deleteを使ったメモリの解放が可能である。

Cではたとえば以下のように書くとpersonクラスへのポインタfriendsに対して100個分の要素を格納できるメモリ領域の先頭アドレスが割り当てられる。

friends = (person\*)malloc(sizeof(person)\*maximumFriends);

これをC++ではnew演算子を使って以下のように簡潔に書くことができる。これによって関数sizeofや(person\*)によるキャストが不要になる。

friends = new person [maximumFriends];

newの戻り値はポインタであることに注意する。personクラスへのポインタだけが必要な場合は以下のように書ける。

one\_person = new person;

また、free(one\_person)に代わって以下のように書くことができる。

delete one\_person;

friendsの場合、配列の先頭アドレスであり、そこから始まるメモリ領域全体を解放する場合は以下のように書く。

delete[] friends;

以下のサンプルコード（new\_instance.cpp）は簡単なチケット発券システムであり、newを使ってオブジェクトへのポインタを生成する例を示している。

/\* new\_instance.cpp \*/

#include <iostream>

using namespace std;

class person{

public:

string name;

};

class ticket{

public:

int id;

person\* user;

ticket(){user = new person;}

};

int main()

{

int ticketNum;

cout << "チケットを何枚購入しますか？ ";

cin >> ticketNum;

cout << "\n";

ticket\* tickets = new ticket [ticketNum];

for(int i = 0; i < ticketNum; i++){

tickets[i].id = i+1;

cout << i+1 << "人目の利用者の名前を入力してください: ";

cin >> tickets[i].user->name;

}

cout << "\nチケット利用者一覧：\n";

for(int i = 0; i < ticketNum; i++){

cout << " " << tickets[i].id << " : " << tickets[i].user->name << "\n";

}

cout << "\n";

}

　上記のコードではticketクラスがpersonクラスのインスタンスへのポインタであるuserをメンバとして持っている。ticketクラスのインスタンスの生成時にはuserクラスのインスタンスも生成されていなければならないため、コンストラクタの中でuser = new person; と書いて生成させている。

**【基本課題4】**

addressbookクラスにおいてfriendsを配列ではなくpersonクラスへのポインタとして定義し、コンストラクタの中でnewを使ってfriendsが使用するメモリ領域（たとえば要素を100個格納できる領域）を確保するようにせよ。デコンストラクタを追加し、delete[] friends;と書くことで領域を解放するようにせよ。デコンストラクタの定義をクラス定義の外側に書く場合、クラス定義の中にプロトタイプ宣言も必要になることにも注意すること。なお、クラスへのポインタは構造体へのポインタとまったく同じように使うことができる。**■**

#### 8 継承

継承とはあるクラスのフィールドとメソッドに加え、さらなるフィールドとメソッドを持たせたより具体的なクラスを作ることを言う。以下のプログラムinheritance.cppはクラスticketを**基本クラス（base class）**  
として、派生クラス（**derived class**）であるtransTicketを作成して使用する例である。

ticketは任意の種類のチケットの情報を格納するためのクラスであるが、transTicketクラスはその中で特に交通機関のチケットの情報を格納するためのクラスである。つまりticketよりも具体的になっており、より多くの属性と機能が追加される。

class transTicket : ticketと書くことでtransTicketは基本クラスticketの派生クラスとなる。「:」の後にpublicと書くことで基本クラス の持つすべてのpublicなフィールドとメソッドが他クラスから見える形で（つまりpublicな形で）派生クラスに継承される。「:」の後にprivateと書けば、基本クラスのpublicなフィールドとメソッドは派生クラスでprivate（非公開）な形で継承されるが、派生クラスのメソッド定義では使用することができる。

クラスtransTicketの定義では独自のフィールドであるorigin（出発地）とdestination（到着地）  
を定義している。また、コンストラクタも新たに定義している。

/\* inheritance.cpp \*/

#include <iostream>

using namespace std;

class person{

public:

string name;

};

class ticket{

public:

int id;

person\* user;

ticket(){user = new person;}

};

class transTicket : public ticket {

public:

string origin;

string destination;

transTicket(): ticket(){};

};

int main()

{

int ticketNum;

cout << "チケットを何枚購入しますか？ ";

cin >> ticketNum;

cout << "\n";

transTicket\* tickets = new transTicket [ticketNum];

for(int i = 0; i < ticketNum; i++){

tickets[i].id = i+1;

cout << i+1 << "人目の利用者の名前を入力してください: ";

cin >> tickets[i].user->name;

cout << " 出発地を入力してください: ";

cin >> tickets[i].origin;

cout << " 到着地を入力してください: ";

cin >> tickets[i].destination;

}

cout << "\nチケット利用者一覧：\n";

for(int i = 0; i < ticketNum; i++){

transTicket t = tickets[i];

cout << " " << t.id << " : " << t.user->name << " : " << t.origin << " => " << t.destination << "\n";

}

cout << "\n";

}

クラスtransTicketの定義の中にはメンバperson\* userが存在しないにも関わらず、main関数の中でt.user->nameのように使えていることに注意せよ。これはtransTicketクラスがticketクラスを継承しているためである。

transTicket(): ticket(){}という行はコンストラクタの継承である。transTicketのインスタンスが生成される時、ticketクラスのコンストラクタも呼ばれることを示している。これによってtransTicketクラスのインスタンスの生成時にuser = new person;という初期化が行われるようになる。

基本クラスで引数付きのコンストラクタ（たとえばticket(string s)）が定義されている場合、transTicket(string s): ticket(s){};と書くと、transTicketのインスタンスの生成時に引数として与えられたsがticketクラスのコンストラクタにも渡される。この書き方は基本クラスのコンストラクタが派生クラスのコンストラクタよりも少ない引数を持つ場合に特に重要である。

継承を用いると、類似した機能を持つクラスの共通部分をまとめて基本クラスで定義しておき、それぞれ異なる機能だけをその派生クラスに定義できる。基本クラスでは派生クラスよりも一般的な機能（多くの派生クラスで使われる機能）だけが実装されている。そのため、他のプロジェクトでも利用できることが多い。これはソースコードの再利用性が高まることを意味し、オブジェクト指向のメリットのひとつである。

基本クラスはスーパークラス、派生クラスはサブクラスとも呼ばれる。数学の基礎となる集合論では要素aと集合Aという関係（a∈A）に加え、サブセット（部分集合）BとスーパーセットAという関係（B⊂A）がある。要素がインスタンス、集合がクラス、サブセットが派生クラス（サブクラス）、スーパーセットが基本クラス（スーパークラス）に対応する。数学のあらゆる概念は集合を使って記述できる。これは集合という概念の強力さを表しており、集合に対応するクラスに基づくオブジェクト指向が広く使用できることもそれから来ていると考えられる。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **数学** | 集合 | 要素 | 部分集合 | スーパーセット |
| **オブジェクト指向** | クラス | インスタンス | サブクラス | スーパークラス |

派生クラスではそれに対する直接の基本クラスだけでなく、その基本クラスを派生クラスとする基本クラス、つまり直接的な親だけでなくその親やそのまた親からもフィールドとメソッドが引き継がれる。基本クラスのさらに基本クラスを**間接基本クラス（indirect base class）**と呼ぶ。

**【基本課題5】**

transTicketクラスを継承したtrainTicketクラスを作成せよ。trainTicketクラスのコンストラクタを作る際、同じ引数を持つコンストラクタは2つ作れないことに注意すること。たとえば引数を持たないコンストラクタもひとつのクラスにつきひとつしか定義できない。■

**【基本課題6】**

int型のフィールドhourとminuteを持つhmクラスを作成せよ。また、trainTicketクラスがhmクラスのインスタンスdepartureTime（出発時刻）とarrivalTime（到着時刻）をフィールドとして持つようにせよ。また、main関数にて出発地と到着時刻を入力させるようにせよ。scanf("%d:%d", &hour, &minute);と書くと、時と分をコロンで区切った「15:30」のような入力を受け付けられる。この場合、15が（main関数のローカル変数である）hourに入り、30が（main関数のローカル変数である）minuteに入る。これらをさらにticket[i].departureTimeやticket[i].arrivalTimeのフィールドであるhourとminuteに代入すればよい。最終的にmain関数の中でチケット利用者一覧の中で出発時刻と到着時刻も表示させよ。なお、scanfは#include <iostream>では読み込まれないため、Cの場合と同様、冒頭に#include <stdio.h>を付け加える必要がある。■

**【発展課題1】**

前問で実装したhmクラスの中でメソッドinMinutesを実装し、戻り値として分単位での時間、すなわちhour \* 60 + minuteを返すようにせよ。また、hmクラスにint型の変数timeInMinutesを引数とするコンストラクタを作成し、timeInMinutesを60で割った商をhourに、余りをminuteに格納するようにせよ。すなわちhm t(125);というようにインスタンスtを生成した時、hourに2が、minuteに5が格納されるようにせよ。■

オブジェクト指向におけるプログラムの挙動は「オブジェクト間でのメッセージパッシング」と説明されることがあるが、これをソースコードレベルで見ると、「クラスAのメソッドの定義の中でクラスBのメソッドを呼び出す」ことがメッセージパッシングに相当する。上記の課題ではtrainTicketクラスのインスタンスがhmクラスのインスタンスのinMinutesメソッドを呼び出していることがメッセージパッシングに相当する。

#### 9 アクセス指定子

メンバ定義の中で使われるキーワードpublicやprivateは**アクセス指定子(access modifier)**と呼ばれる。C++ではもうひとつ、protected（被保護）と呼ばれるアクセス指定子が存在する。クラス定義の中でprotected:の後に続くメンバ（フィールドとメソッド）はそのクラスとそこから派生したクラスからは見ることができるが、それ以外のクラスからは見ることができない。

**【基本課題7】**

personクラスのフィールドnameをprotectedにしたプログラムを作成せよ。これによってｍain関数の中でnameにアクセスできなくなるため、代わりにpersonクラスにvoid setName(string n){name = n;}とstring getName(){return name;}というpublicなメソッドを実装し、main関数ではそれを使ってnameにアクセスするようにせよ。このようにprivateあるいはpublicなフィールドにアクセスするためのメソッドを**セッター(setter)**ならびに**ゲッター(getter)**と呼ぶ。セッターは値の設定用であり、ゲッターは値の取得用である。■

#### 10 API

　API(Application Programming Interface)とはカプセル化されたモジュール（オブジェクト指向の場合はクラス）がどのようなフィールドとメソッドを開発者に対して提供しているのかを示す情報である。インタフェースとは界面（境界面）の意味であり、自分以外のクラスとの接触の方法を表している。カプセル化によって実装とインタフェースが分離されているため、実装を知らなくてもAPIを知っていれば利用できる。

　たとえば「C++ string」で検索すると、stringクラスのAPIが見つかり、それがどのようなフィールドとメソッドを持っているかが分かる。

　オブジェクト指向に基づく開発では既存のクラスを最大限活用することが一般的である。C++プログラミングをより効果的に行うためにはこの演習で学んだ構文に加え、C++で開発されたどのようなクラスが存在し、それらがどのようなAPIを持っているのかを知っておく必要がある。もちろん既存のクラスの数は膨大であるので、開発したいプログラムの性質に応じて関連すると思われるクラスのAPIを検索して調べれば良い。

11 参照渡し

　C++では値渡しとポインタ渡し（アドレス渡し）に加え、**参照渡し**と呼ばれる引数の渡し方がある。関数定義で使う場合は以下のようになる。

　　　値渡し： void func(int a)

　　　ポインタ渡し： void func(int\* a) あるいは void func(int \*a)

　　　参照渡し： void func(int& a) あるいは void func(int &a)

　参照渡しは一種の略記法であるとも言える。以下の2つのプログラムargument\_pointer.cppとargument\_ref.cppは同一の挙動を持つが、前者はポインタ渡しを使っており、後者は参照渡しを使っている。

/\* argument\_pointer.cpp \*/

#include <iostream>

using namespace std;

void func(int \*x){

\*x \*= 2;

\*x += 3;

}

int main(){

int a = 10;

func(&a);

cout << a << "\n";

}

/\* argument\_ref.cpp \*/

#include <iostream>

using namespace std;

void func(int &x){

x \*= 2;

x += 3;

}

int main(){

int a = 10;

func(a);

cout << a << "\n";

}

　参照渡しの場合、ポインタ渡しのようにfuncには実引数aのアドレスが伝わっているのだが、funcの関数定義内で仮引数を使ってxと書くだけで（実引数aのアドレスではなく）そのアドレスにある値（つまりaの値）  
を取得できる。つまりポインタ渡しと違い、関数内で毎回\*xと書く必要がなくなるので、記述が楽になる。

　また、funcを呼び出す際にもfunc(&a)のようにaのアドレスを求めてから渡すのではなく、func(a)と書くだけでよい。これだけでfuncには実引数aのアドレスが伝わる。

関数呼び出しでオブジェクトを渡した場合、ポインタ渡しではオブジェクトのコピーが生成され、呼び出された関数の中で使用されるが、参照渡しでは生成されず、本来のオブジェクトが参照されるという違いがある。呼び出された関数から戻る時、ポインタ渡しでは（ローカル変数であるコピーが破棄されるため）デコンストラクタが呼ばれるが、参照渡しでは（そもそもコピーは作られていないため）デコンストラクタは呼ばれない。このためデコンストラクタによって意図せずメモリが解放されることが防げる。

値渡しとポインタ渡しをまとめて値渡しと呼ぶこともある。これはポインタ渡しではアドレスを入れる変数（ポインタ変数）の値が渡されていると解釈できるためである。

なお、Javaの場合、そもそもポインタは使用されないが、関数にインスタンス変数を渡す場合、&を付けなくても参照渡しが使われる。

**【基本課題8】**

引数xとyを置換するプログラムswapをポインタ渡しではなく参照渡しを使った形で実装せよ。■

#### 12 参照の返し

参照渡しは関数呼び出しに参照を使うことであるが、参照の返しでは関数の戻り値として参照を使う。関数やメソッドが参照を返すようにすると、それを式の左辺に置いて、値を代入すると行った使い方ができる。たとえば以下の2つのプログラムreturn\_pointer.cppとreturn\_ref.cppは同じ挙動をする。

/\* return\_pointer.cpp \*/

#include <iostream>

using namespace std;

class cls{

public:

int x;

int\* f(){return &x;};

};

int main(){

cls a;

\*(a.f()) = 200;

cout << "a.x = " << a.x << "\n";

};

/\* return\_ref.cpp \*/

#include <iostream>

using namespace std;

class cls{

public:

int x;

int& f(){return x;};

};

int main(){

cls a;

a.f() = 200;

cout << "a.x = " << a.x << "\n";

};

　fがポインタを返す場合、a.f()はアドレスであるので、そのアドレスに値を格納するには\*(a.f()) = 200と書く必要がある。一方、fが参照を返す場合、a.f() = 200と書くだけでa.f()が返すアドレスに値を格納できる。

**【発展課題2】**

棚が3つあり、ひとつの棚に本を一冊だけ置ける本棚を表すクラスbookshelfを作成せよ。 bookshelfクラスには3つのstringクラスのフィールドtop, middle, bottomを持たせよ。また、bookshelfクラスにメソッドstring& getShelf (string destination)を実装し、destinationの値が"すぐ読む", "あとで読む", "積ん読" のいずれであるかに応じ、フィールドtop, middle, bottomのいずれかへの参照を返すようにせよ。 main関数の中でbookshelfクラスのインスタンスbsを生成した上で、ユーザに書籍の名称を尋ね。それをいつ読むか（つまり「すぐ読む」／「あとで読む」／「積ん読」のどれであるか）を尋ねよ。それに合わせて書籍名をbsのtop, middle, bottomのいずれかに格納するプログラムを実装せよ。また、bookshelfクラスにshowShelves()メソッドを実装し、新しく本を置くたびに現在のtop, middle, bottomの内容を表示させよ。まだ書籍が置かれていない棚については空文字列を表示させる必要がある。これにはコンストラクタでtop, middle, bottomの初期値を""に設定すればよい。■

#### 13 仮想関数によるポリモーフィズム

ひとつの基本クラスAから2種類の派生クラスBとCを作成した場合、それらのインスタンスはまとめてリストや配列に格納できる。基本クラスAのメソッドmをBとCで異なった形で定義することで、実行時に配列の要素ごとに異なる挙動を行わせられる。このようにひとつの変数にどのようなクラスのインスタンスが格納されているかに応じ、同じメソッドでも異なる挙動を行わせることを**ポリモーフィズム（polymorophism）**または**多態性**と呼ぶ。

以下のpolymorphism.cppではpersonクラスを継承したmemberクラスとnonmemberクラスを定義している。また、参加者リストを管理するため、personクラスのインスタンスへのポインタをデータとして持つリストを作成している。リストの各ノードはpar\_nodeクラスのインスタンスである。

/\* polymorphism.cpp \*/

#include <iostream>

using namespace std;

class person{

protected:

string name;

public:

person(){};

void setName(string n){name = n;};

virtual void requestData(){};

virtual void showData(){};

};

class par\_node{

public:

person\* participant;

par\_node\* next;

};

class member : public person{

int memberid;

public:

void requestData(){

cout << " 会員番号を入力してください。\n ";

cin >> memberid;

};

void showData(){

cout << " " << name << "(会員) 会員番号:" << memberid << "\n";

};

};

class nonmember : public person {

string email;

string phone;

public:

void requestData(){

cout << " メールアドレスを入力してください。\n ";

cin >> email;

cout << " 電話番号を入力してください。\n ";

cin >> phone;

};

void showData(){

cout << " " << name << "(非会員) " << email << " / " << phone << "\n";

};

};

int main()

{

int i = 0;

string name, mtype;

par\_node\* p = new par\_node;

par\_node\* q = p;

while(1){

cout << "名前を入力してください。（終了する場合はquitと入力してください）\n";

cin >> name;

if(name == "quit"){break;}

cout << " 会員ですか？(y/n) \n ";

cin >> mtype;

if(mtype == "y"){

q->participant = new member;

}else{

q->participant = new nonmember;

}

q->participant->setName(name);

q->participant->requestData();

q->next = new par\_node;

q = q->next;

i++;

}

cout << "\n参加者一覧：\n";

q = p;

while(i > 0){

q->participant->showData();

q = q->next;

i--;

}

cout << "\n";

}

　polymorphism.cppではデータとしてpersonクラスへのポインタparticipantを持つノードを繋げてリストを作っている。より一般的に、arbdata（任意データ）というクラスを作成することで、それを継承する任意のクラスのインスタンスを格納可能なリストを作成できる。Cの配列やリストでは同一の型のデータしか格納できなかったが（すなわちint型の変数を繋げたリスト、あるいはchar型の変数を繋げたリスト等しか作れなかったが）、C++では継承を使うことでひとつのリストの中に多様なデータを格納できるようになる。

personクラスでメソッドrequestDataとshowDataを定義する際、キーワードvirtualが付けられている。これらは**仮想関数(virtual function)**であり、派生クラスで再定義できる。このような再定義（上書き）を**オーバーライド (override)** と呼ぶ。ここではmemberとnonmemberで異なる形でrequestDataとshowDataをオーバーライドしている。

ポインタq->participantはpersonクラスへのポインタである。polymorphism.cppではここにmemberクラスやnonmemberクラスのインスタンスのアドレスを格納している。このように、基本クラスへのポインタが派生クラスのインスタンスのアドレスを指すことは可能である。逆に派生クラスへのポインタで基本クラスのインスタンスのアドレスを指すことは一般に（キャストなどを使わない限り）可能ではない。

　\*(q->participant)がmemberとnonmemberのどちらのクラスのインスタンスになるかは実行時まで決まらず、またそれによってq->participant->requestData()とq->participant->showData()の実行結果が異なる。これを**実行時ポリモーフィズム(run-time polymorphism)**または**実行時バインディング(run-time binding)**呼ぶ。

**【基本課題9】**

学生会員を表すstmemberクラスをmemberクラスの派生クラスとして定義し、grade（学年）をフィールドとして追加せよ。main関数を書き換え、会員に対して学生会員であるかを尋ね、そうである場合はさらに学年を尋ねて格納するようにせよ。■

#### 14 抽象クラス

　派生クラスで定義されるためだけに存在し、基本クラスで定義されていない関数を**純粋仮想関数(pure virtual function)**と呼ぶ。クラス定義の中で以下のように==0と書くと、funcは純粋仮想関数となる。

virtual void func(int x) = 0;

純粋仮想関数をひとつ以上持つクラスを**抽象クラス（abstract class）**と呼ぶ。抽象クラスのインスタンスを作ることはできず、その派生クラスですべての純粋仮想関数をオーバーライドによって具体的に定義した場合にはじめてそのインスタンスを生成できる。

純粋仮想関数は**インタフェース(interface)**を定義するために使用される。インタフェースとはクラスなどのモジュールが外部に対して提供するサービスの入出力の型を示すものであり、実装を問わないものである。

|  |  |  |  |
| --- | --- | --- | --- |
| **純粋仮想関数** | インタフェース | 引数と戻り値の型のみを指定 | Whatの記述 |
| **通常の関数** | 実装 | 実際の計算方法を記述 | Howの記述 |

　インタフェースがどのように使われるかは、ライブラリとフレームワークについて述べる際に説明する。

#### 15 ライブラリとフレームワーク

インタフェースは**ライブラリ(library)**や**フレームワーク（framework）**の構築などで利用される。ライブラリは互いに類似する機能をまとめたクラスの集まりであり、フレームワークとは様々な具体的アプリケーションを作るための共通の枠組みを提供するクラスの集まりであるたとえば電子商取引システムを作る場合、どのような属性を持った商品を扱うことになるかはシステムによって異なるが、ユーザ管理や商品管理、ショッピングカートによる商品の選択や支払い手続きなどは共通である。そのため共通の部分だけを抽象クラスで作っておき、それを継承して具体的な派生クラスを作ることでシステムを完成させられるようにしておくことをフレームワークと呼ぶ。

ここでは具体的な例として、機械学習の機能を提供するライブラリを考える。訓練データにおける観測値をもとに未知の値を予測するのが回帰(regression)である。線形回帰、多項式回帰、ガウシアンプロセス回帰など多数の方法が提案されており、その数は増え続けている。ただ、いずれの手法においても回帰は二つの段階からなる。まず、訓練データを使ってパラメータを学習させる段階（訓練あるいはfit）、続いて学習されたパラメータとテストデータを使って予測(predict)を行う段階である。回帰を行うためのモデルをインスタンス変数regで表した場合、これらの2段階は以下のように表すことができる。ただしtrain\_xは訓練データにおける独立変数の値を並べた行列、train\_yは訓練データにおける従属変数の値を並べたベクトル、test\_xはテストデータにおける独立変数の値を並べた行列、est\_yは従属変数の予測値を並べたベクトルである。なお、パラメータはrがメンバとして持っており、以下のコードでは直接アクセスしていない。

r.fit(train\_x, train\_y);

est\_y = r.predict(test\_x);

どの手法を使って回帰を行われるかに関わらず、同じメソッド名（fitとpredict）で回帰を行えれば、ユーザにとって便利である。また、機械学習では様々な手法を比較することが求められるが、どの手法でもfitを使って訓練、predictを使って予測が行える場合、たとえばforループで多数の手法を比較できる。

ひとりの開発者がライブラリ全体を開発しているのであればそのような統一性が生まれる可能性があるが、実際は多くの開発者が協力してライブラリを作ることが多い。回帰の各手法をそれぞれ別の開発者が実装する場合、どの手法でも訓練にはfit、予測にはpredictという関数名が使われることが望ましい。そのような統一性を実現するための仕組みとして、純粋抽象 関数が使用できる。すなわちregressorという基本クラスをあらかじめ作っておき、そこに以下のように純粋仮想関数fitとpredictを用意しておく。ただしmatrixは行列を表すクラス、vectorはベクトルを表すクラスである。

class regressor{

public:

virtual void fit(matrix x, vector y) = 0;

virtual vector predict(matrix x) = 0;

}

さらに、開発者グループの中で全体の設計を決める際、回帰の手法を実装する場合はregressorクラスを継承することに決めておく。たとえばある開発者が以下のように線形回帰のためのクラスlinearRegressorの実装を始めたとする。

class linearRegressor : public regressor{ ... }

するとこのクラスはfitとpredictを実装しなければコンパイルが通らない。開発者は実装を要求された関数名（fitとpredict）や関連するコメントを見て、fitで訓練を、predictで予測を実装すればよいということを知り、そのように実装する。これによってすべての回帰の手法が統一された入出力方式（すなわちインタフェース）を持つようになる。

#### 16 デザインパターン - Factory Methodパターン -

前節の機械学習のライブラリでは回帰モデルを表す基本クラス（インタフェース）regressorを継承した多数の具体的な回帰モデルが派生クラスとして作られる。たとえばlinearRegressor、polyRegressor、gpRegressorがそれぞれregressorを継承した派生クラスとする。以下のようにすると、線形回帰のためのインスタンス変数r1、多項式回帰のためのインスタンス変数r2、ガウシアンプロセス回帰のためのインスタンス変数r3が作られる。

linearRegressor r1;

polyRegressor r2;

gpRegressor r3;

しかしこのコードでは各インスタンス変数がどのクラスのインスタンスであるかが決まってしまっているために、たとえばユーザが実行時に回帰の種類を切り替える場合、ユーザからの入力に応じてr1, r2, r3の間で切り替えを行わなくてはならず、switch-case文などが必要になり、コードが煩雑になる。また、forループを使ってすべての手法を比較することも容易でない。

そこで単にregressorという抽象クラス（すなわちインタフェース）へのポインタrpを作成し、実行時にそこに具体的な派生クラスのインスタンスのアドレスを格納するようにする。具体的な派生クラスのインスタンスの生成には詳細を後述するメソッドregressor\* regressorCreator::create(string regressorType)を使用する。regressorTypeはstringクラスのインスタンスであり、回帰モデルの名称を入れる。これによって以下のように回帰モデルへのポインタrpを生成できる。

string regressorType;

regressorCreator rc;

cout << "どの回帰モデルを使いますか？" << endl;

cin >> regressorType;

regressor\* rp = rc.create(regressorType);

Aを基本クラス、Bをその派生クラスとした時（すなわちBがAを継承している時）、Aの型を持つインスタンス変数にBのインスタンスを格納することは可能であるため、regressorクラスのインスタンス変数へのポインタrpには派生クラスのインスタンスのアドレスを格納できる。以下のように\*rpにfitとpredictを実行させることで訓練と予測が行える。

rp->fit(train\_x, train\_y);

est\_y = rp->predict(test\_x);

regressorCreatorクラスは以下のように定義できる。throw new noRegressorExceptionは例外を投げるという処理であり、のちの節で後述するが、regressorTypeで指定された種類の回帰が実装されていなかった場合という例外的な状況に対応している。

class regressorCreator {

public:

static regressor\* create(string regressorType){

if(regressorType == "linear"){

return new linearRegressor;

}else if(regressorType == "poly"){

return new polyRegressor;

}else if(regressorType == "gp"){

return new gpRegressor;

}else{

throw new noRegressorException;

}

}

};

regressorCreatorのように派生クラスのインスタンスを生成するためのクラスを用意することで、あるインスタンス変数が属するクラスを実行時に決められるようにするテクニックを**Factory Methodパターン**と呼ぶ。Factoryとは「工場」という意味であり、regressorCreatorクラスのcreateメソッド（メンバ関数）が派生クラスのインスタンスを生成する工場の役割を果たしていることから来ている名称である。

Factory Methodパターンは**デザインパターン(design pattern)**のひとつである。デザインパターンはオブジェクト指向においてプログラム全体の見通しを良くし、拡張性や再利用可能性を高めるためのテクニックの総称であり、多数提案されている。デザインパターンはオブジェクト指向言語の言語仕様の一部というわけではないが、良いコードに頻繁に現れる共通構造である。多くの開発者がデザインパターンを使用してコードを書いているため、それを知っていることで他の開発者が書いたソースコードを読みやすくなる。

#### 17 デザインパターン - Strategyパターン -

前述の機械学習のライブラリでは回帰を行うためのクラスregressorを定義し、具体的なアルゴリズムは派生クラスで実装した。regressorクラスの変数rにどのような派生クラスのインスタンスを入れるかにより、実際に使われるアルゴリズムを切り替えることができた。

このように何らかの処理（たとえば「回帰する」）に対し、それを行う装置（たとえば「回帰モデル」）を考え、それを基本クラスとし、それの派生クラスのインスタンスに処理を行わせるという流れはオブジェクト指向言語におけるプログラミングで頻繁に使われる。これはデザインパターンのひとつであり、**Strategyパターン**と呼ぶ。Strategyとは「戦略」という意味であるが、ひとつの目的を達成するために存在する様々な手法（アルゴリズム）を指す。Strategyパターンは「アルゴリズムのオブジェクト化」と説明されることもある。回帰のアルゴリズムを関数として定義するだけではなく、それを行うためのクラスを定義し、オブジェクトを生成させて処理を行わせるところに特徴がある。

例として、あるプログラムが出力する様々なメッセージを機械翻訳によって別の言語に翻訳することを考える。機械翻訳には様々なアルゴリズムが提案されており、 ある機械翻訳器でうまくいかなかった場合、別の機械翻訳器に切り替えて翻訳させたいこともある。また、新しいアルゴリズムが提案されることもあるため、アルゴリズムの追加も容易であってほしい。そこで機械翻訳器に相当するtranslatorという抽象クラスを定義し、そこで純粋抽象関数translateを定義する。その派生クラスとしてnnTranslator（ニューラルネットワークによる翻訳器）やhmmTranslator（隠れマルコフモデルによる翻訳器）などを定義する。

nnTranslator t;

language = "Japanese";

（中略）

cout << t.translate(message, language);

プログラム中の様々な場所でt.translateは現れるが、最初にtを定義する際、どの派生クラスのインスタンスと設定するかによって翻訳手法を一括して切り替えることができ、コードのメンテナンスが容易である。また、tの値を上書きすることで実行時に機械翻訳の手法を切り替えることも可能である。新しい機械翻訳アルゴリズムを追加する場合には、translatorを継承する新たな派生クラスを定義すればよい。

さらに別の例として、ソートが挙げられる。ソートにはバブルソートやクイックソート、マージソートやヒープソートなど様々なアルゴリズムが存在し、計算時間や必要メモリの要請に応じて切り替えて使いたいことがある。そこでソート装置を表すsorterクラスを抽象クラスとし、そこで純粋仮想関数sortを定義する。たとえばその派生クラスmergeSorterにおいてsortを具体的に実装し、以下のように使用できる。

mergeSorter s;

（中略）

sorted\_array = sorter.s(original\_array);

実行時、変数sをどの派生クラスのインスタンスにするかによってソート手法を切り替えられる。

#### 18 デザインパターン - Commandパターン -

Strategyパターンではアルゴリズムを実行する装置を考えることで、「アルゴリズムのオブジェクト化」を行った。Commandパターンでは「ユーザからのコマンド（命令）のオブジェクト化」を行う。

たとえばオブジェクト指向言語でGUIのユーザインタフェースを作成する状況を考える。メニューバー上のボタンをひとつクリックすると、プルダウンメニューが現れる。文書作成ソフトであれば「新規作成」「開く」「保存する」「名前をつけて保存する」といった個々の項目をメニューアイテムと呼ぶ。これらは表示されている文字列とクリックした時の効果が異なるだけであるので、ひとつのクラスから作られた多数のインスタンスとして管理するのが自然である。そのためメニューアイテムを表すクラスをmenuItemとする。

各メニューアイテムには何らかのコマンドを対応づけたい。コマンドを抽象化したものがcommandクラスである。menuItemのインスタンスにその派生クラスのインスタンスへのポインタを格納させることで対応付けできる。以下のcommand\_pattern.cppではcommandクラスの派生クラスとしてcreateNewFileCommandとopenFileCommandを実装し、それらのインスタンスへのポインタをmenuItemクラスのインスタンスm[0]とm[1]のプロパティcmdに格納している。

/\* command\_pattern.cpp \*/

class command {

public:

virtual void execute() = 0;

};

class createNewFileCommand : public command {

void execute(){

// ここにファイル新規作成の動作を書く。

};

};

class openFileCommand : public command {

void execute(){

// ここにファイルを開く動作を書く。

};

};

class menuItem {

command\* cmd;

public:

menuItem(){};

menuItem(command\* c){cmd = c;};

void onClick(){

cmd->execute();

};

};

int main(){

menuItem\* m = new menuItem [2];

m[0] = menuItem(new createNewFileCommand);

m[1] = menuItem(new openFileCommand);

m[0].onClick();

m[1].onClick();

}

Commandパターンを使うことのメリットとして、ユーザインタフェースとロジック（計算処理）が分離されることが挙げられる。menuItemの中にはcmd.execute()と書かれるだけであり、ボタンが押された時のロジックは記述しなくてよい。実際のロジックについてはcommandクラスの派生クラスの中で書けばよい。すなわちmenuItemはユーザインタフェース、commandの派生クラスはロジック、という分離が行えている。これはユーザインタフェース開発者とロジック開発者の間で分業が行えることを意味する。

同様の理由でユーザインタフェースだけを提供するツールキットを実装する際にCommandパターンは利用される。ツールキットを利用したアプリケーションプログラムではメニューアイテムがクリックされた時に何らかのコマンドが実行されるが、コマンドの内容、ならびにメニューアイテムとコマンドの対応付けはアプリケーションプログラムを実装する際に記述すればよく、ツールキット自体においてはcommandクラスの派生クラスを定義する必要がない。Commandパターンは当然、menuItemに限らず、ボタンやアイコンなど、ユーザの入力を受け付ける任意の要素に対して使用できる。

commandクラスのインスタンスに過去の状態をプロパティとして保持させることで、「元に戻す」が容易に実装できるというメリットもある。commandクラスのインスタンスの系列を配列として残しておけば、何段階でも元に戻すことが可能になる。

#### 19 多重継承

　ひとつのクラスが2つ以上の基本クラスを継承することを**多重継承（multiple inheritance）**と呼ぶ。以下のmulti\_inherit.cppにおけるsmartwatchクラスはwatchクラスとmessengerクラスを共に継承している。

/\* multi\_inherit.cpp \*/

#include <iostream>

using namespace std;

class watch{

public:

int hour, minute;

watch(int h, int m){hour = h; minute = m;};

};

class messenger{

public:

string message;

messenger(string msg){message = msg;}

};

class smartwatch : public watch, public messenger{

public:

smartwatch(int h, int m, string msg) : watch(h,m), messenger(msg){};

};

int main(){

smartwatch sw(15, 30, "Hello user!");

cout << sw.message << " (It's now " << sw.hour << ":" << sw.minute << ")\n";

};

　上記のプログラムではclass smartwatch : public watch, public messenger と書くことでwatchクラスとmessengerクラスを継承したsmartwatchクラスを定義している。また、コンストラクタsmartwatch(int h, int m, string msg) : watch(h,m), messenger(msg){}ではsmartwatchに与えられた引数h, m, msgのうちhとmをwatchに渡してそのコンストラクタが実行され、またmsgがmessengerに渡されそのコンストラクタが実行されるようにしている。

2つのルートでひとつの間接基本クラスを継承した場合、どちらの経路で得られた間接基本クラスを優先したら良いのかがコンパイラに分からず、曖昧性が生じる。このため**仮想基本クラス(virtual base class)**という仕組みがある。派生クラスにおいて基本クラスを継承する際、「:」の後（すなわち基本クラスのアクセス指定子の前）に以下のようにキーワードvirtualを付けると、間接的な派生クラスに基本クラスのコピーが2つ以上含まれることを防げる。

class derived : virtual public base {...}

**【基本課題10】**

stringクラスのフィールドtitleを持たせたmovie（映画）クラス、stringクラスのフィールドcode（コード）を持たせたcopyprotection（コピー保護）クラスを作成せよ。それらをいずれも継承したprotectedMovie（コピー保護された映画）クラスを作成せよ。protectedMovieクラスのインスタンスがtitleとcodeの両方を持つことをmain関数において利用することで示せ。■

#### 20 ファイル入出力

C++におけるファイル入出力には冒頭でfstreamをインクルードし、cinとcoutの代わりにfinとfoutを使うことで行える。

/\* file\_io.cpp \*/

#include <iostream>

#include <fstream>

#include <math.h>

using namespace std;

int main(){

// ファイル書き出し

ofstream fout("test.txt");

if(!fout){cout << "出力ファイルが開けません"; return 1;}

for(int i = 0; i < 5; i++){

fout << i << "\n";

}

fout.close();

// ファイル読み込み

ifstream fin("test.txt");

if(!fin){cout << "入力ファイルが開けません"; return 1;}

string str;

int i;

while(!fin.eof()){

fin >> str;

cout << "line " << i << ": " << str << "\n";

i++;

}

fin.close();

return 0;

}

　上記のコードではwhile(!fin.eof())において、ファイルの末尾に到達するとメソッドfin.eof()はfalseを返すので、whileループから抜けられる。eofはend of fileの略である。

**【基本課題11】**

ユーザにファイル名を尋ね、そのファイル名の最後に「.backup」を付けた複製を作成するプログラムを実装せよ。■

**【発展課題3】**

上記のfile\_io.cppでは出力の最後に4が2回出てくる。これはtest.txtの最後に改行があるため、eofに到達する前に1行あることになるが、fin >> strにおいて何も読み込めないため、strの中身が書き換わらず、4が繰り返されている。4の出力が繰り返されないようにプログラムを書き直せ。■

#### 21 例外処理

プログラムを記述する際、通常処理と例外的な処理を分けて記述したいことが多い。例外的な状況が起きた場合にはその対応箇所への移動が必要である。これを可能にするのが例外処理である。

例外処理は例外を送出する部分と送出された例外を捕捉して決められた処理を実行する部分からなる。tryから始まる中括弧の中で例外が送出された場合、catch(...)から始まる捕捉部分に処理が移動する。

以下のexception.cppではex\_account\_too\_longというクラスを定義し、例外的な状況においてはそのインスタンスex2longを生成し、throw ex2long;によってthrowさせている。その場合、catch(ex\_account\_too\_long e)以下のブロックが実行される。変数eにはthrowされたex2longが格納されている。

/\* exception.cpp \*/

#include <iostream>

using namespace std;

class ex\_account\_too\_long{

public:

string account;

ex\_account\_too\_long(string str){account = str;};

string show(){return "アカウント '" + account + "' は長すぎます。";};

};

int main(){

string account;

int accountlenlim = 6;

cout << "希望アカウントを入力してください（英数字6文字以内）: ";

cin >> account;

try{

if(account.length() > accountlenlim){

ex\_account\_too\_long ex2long(account);

throw ex2long;

}else{

cout << "\n" << account << "さん、ようこそ！\n\n";

}

}catch(ex\_account\_too\_long e){

cout << "\nException: " << e.show() << "\n\n";

}

}

どの例外を投げられるかが指定されていない関数の中で例外が発生し、関数の中にcatch文が存在しない場合、実行はその関数の呼び出し元に移る。その中でcatch文がなければさらに呼び出し元の関数に戻り、 mainでもcatchされなければプログラムは例外を表示させて終了する。

関数定義における関数名の後にthrow EXCEPTION\_NAMEと書くことでその関数が投げられる（すなわち呼び出し元に送れる）例外をEXCEPTION\_NAMEに限定できる。その関数からそれ以外の例外が投げられた場合、プログラムはその時点で終了する。

たとえば以下のように定義される関数の中でex\_account\_too\_long以外の例外が発生した場合、プログラムは終了する。

int signup(char a) throw ex\_account\_too\_long{...}

関数がthrowする例外を書いておく理由は、その関数を使ってプログラミングする際、その例外に対するcatch文を用意しなくてはならないことを示すためである。

**【基本課題12】**

アカウントが2文字以下だった場合に発生させる例外のクラスを作成し、上記exception.cppの中でthrowさせ、アカウントが長すぎた場合と同様の対応を行わせよ。■

#### 22 演算子オーバーロード

関数のオーバーロードと同様に、演算子のオーバーロードも可能である。たとえばC++では "hello" + " " + "world" のように書くことでstringクラスのインスタンスを繋げていくことができるが、これは+演算子をstringクラスでオーバーロードすることで可能になっている。数値の足し算と文字列の連結は全く異なる挙動であるが、直感的であるため使いやすい。このようにオーバーロードによって同じ名前の演算子や関数をクラスに応じた目的で使うことができるようになり、クラスごとに異なる演算子名や関数名をおぼえなくて済むようになる。

　Cでは画面表示にはprintf、ファイル書き込みにはfprintfという別の名前の関数があるが、C++ではいずれも>>出力演算子（挿入演算子）によって実現できる。これも>>に対するオーバーロードによって実現されている。

以下のoverload\_vec3d.cppは「+」演算子をオーバーロードし、3次元ベクトルの間で和を定義する例である。

/\* overload\_vec3d.cpp \*/

#include <iostream>

#include <string>

#include <sstream>

using namespace std;

string to\_str(int num){

stringstream ss;

ss << num;

return ss.str();

}

class vec3d {

public:

int d1, d2, d3;

vec3d(int i1, int i2, int i3){d1 = i1; d2 = i2; d3 = i3;};

vec3d(){};

string show(){return "(" + to\_str(d1) + "," + to\_str(d2) + "," + to\_str(d3) +\

")";};

};

// operator overload for +

vec3d operator+(vec3d u, vec3d v){

vec3d w;

w.d1 = u.d1 + v.d1;

w.d2 = u.d2 + v.d2;

w.d3 = u.d3 + v.d3;

return w;

}

int main()

{

vec3d a(2,3,5), b(7,5,1);

cout << "a = " << a.show() << "\n";

cout << "b = " << b.show() << "\n";

cout << "a + b = " << (a+b).show() << "\n";

}

　上記のコードにおいてvector operator+(vector u, vector v)から始まるブロックが演算子オーバーロードであり、引数uとvについて成分同士を足し合わせたベクトルwを作成し、戻り値として返すようにしている。

**【発展課題4】（比較的容易）**

stringクラスを継承したcase\_insクラスを定義せよ。これについて==演算子のオーバーロードを行い、大文字と小文字の違いを無視した場合に一致する場合にtrueを返し、そうでない場合にはfalseを返すようにせよ。char型の変数の値を数値とみなした時、小文字と大文字の差（たとえば 'a' - 'A'）は32であることを利用できる。main関数の中でこの関数を利用し，正しく判定できていることを確認せよ。■

#### 23 テンプレート

C++では様々なクラスが定義されるが、それらに共通して使われる操作も存在する。たとえばソートという操作は数値に対しても文字列に対しても行われる。対象となる型（クラス）は異なるが、アルゴリズムは共通である。

このように多様なクラスに対してアルゴリズムだけを抜き出し、定義することを可能にする仕組みが**テンプレート(template)**である。様々な型のデータに適用でいる汎用操作を定義する関数を**汎用関数(generic function)**または**テンプレート関数(template function)**呼ぶ。汎用関数は型が指定されると自動的にその型に対応した関数を生成する。すなわち任意の型へのオーバーロードが自動で行えることを意味する。ただしオーバーロードと違い、型に応じて処理を変えるということはできず、どの型に対しても同じ処理が行われることになる。

以下は任意のクラスのインスタンスをスワップできるようにするための関数arb\_swap.cppである。

/\* arb\_swap.cpp \*/

#include <iostream>

using namespace std;

template <class X> void swap\_generic(X &a, X &b){

X temp;

temp = a;

a = b;

b = temp;

}

int main(){

int i = 12, j = 53;

string x = "Hello", y = "Good night";

cout << "置換前：i = " << i << ", j = " << j << endl;

cout << "置換前：x = " << x << ", y = " << y << endl;

swap\_generic(i, j);

swap\_generic(x, y);

cout << "\n";

cout << "置換後：i = " << i << ", j = " << j << endl;

cout << "置換後：x = " << x << ", y = " << y << endl;

return 0;

}

関数swap\_genericは汎用関数であり、任意のクラスXのインスタンスを引数として受け取れる。Xは型を表す変数である。YやZといった別の変数名を使っても良い。main関数ではint型とstringクラスについてswap\_genericを使った置換を行っている。Cで関数swapを作る場合、引数の型を指定しなくてはならないため、型ごとにswapを定義しなくてはならないが、汎用関数を使うとそのような手間が省ける。なお、endlはend of lineの略であり、"\n"の代わりに行末コードを入れるための定数である。

汎用関数と同様、汎用クラスも定義できる。汎用クラスはその定義の中で任意のクラスのインスタンスを表すメンバを持ち、それを操作できる。定義には汎用関数と同様にキーワードtemplateを使う。

以下のgeneric\_max.cppでは汎用クラスmyArrayの定義においてXをクラスを表す変数としている。そのメンバとしてXクラスの変数data、Xクラスのインスタンスを戻り値とするメソッドmaxを定義している。main関数では乱数を使いint型の配列とdouble型の配列を作成し、それぞれインスタンスforIntとforDoubleのメンバdataに格納している。

/\* generic\_max.cpp \*/

#include <iostream>

#include <random>

#define N 4

#define RMAX 10

using namespace std;

// defines a generic class

template <class X> class myArray{

public:

X data[N];

X max();

};

// defines a generic function

template <class X> X myArray<X>::max(){

X largest = data[0];

for(int i = 1; i < sizeof(data)/sizeof(data[0]); i++){

if(largest < data[i]){

largest = data[i];

}

}

return largest;

}

int main(){

random\_device rnd;

mt19937 mt(rnd());

uniform\_real\_distribution<> rnd\_real(0, 1);

myArray<int> forInt;

myArray<double> forDouble;

cout << endl << "forInt.data = ";

for (int i = 0; i < N; ++i) {

forInt.data[i] = rnd() % RMAX;

cout << forInt.data[i] << " ";

}

cout << endl << "forDouble.data = ";

for (int i = 0; i < N; ++i) {

forDouble.data[i] = rnd\_real(mt);

cout << forDouble.data[i] << " ";

}

cout << endl << endl << "forInt.max() = " << forInt.max() << endl;

cout << "forDouble.max() = " << forDouble.max() << endl << endl;

}

**【発展課題5】**

テンプレート（汎用関数）を利用し、比較演算子「<」で比較可能な任意の型の配列から最大の要素を見つける関数を作成せよ。■

#### 20 フレンド関数

　オブジェクト指向ではカプセル化が推奨されるため、クラスは多数の非公開メンバ(private member)を持つ。一般に対しては非公開でも、一部の関数からはアクセスできるようにしたいことがある。これを実現するための仕組みが**フレンド関数(friend function)**である。

どの関数をフレンド関数とするかは、アクセスされるメンバを持つクラスの側であらかじめ指定しておかなくてはならない。

/\* friend.cpp \*/

#include <iostream>

using namespace std;

class locked{

string content;

string correct\_code;

public:

friend string access(locked l, string key);

locked(){content = "インストールキーは○○○です。"; correct\_code = "FFF888";};

};

string access(locked l, string c){

if(c == l.correct\_code){

return l.content;

}else{

return "";

}

};

int main(){

string code;

locked lckd;

cout << "コードを入力してください。\n";

cin >> code;

cout << access(lckd, code) << "\n";

}

　friend.cppではfriendキーワードを使い、クラスlockedに対して関数accessをフレンド関数として定めている。もしfriendから始まる行を取り除くと、関数accessの定義の中でクラスlockedの非公開メンバであるcontentとcodeにアクセスできないため、コンパイルエラーが生じる。

**【発展課題6】（比較的容易）**

ユーザの位置を表すクラスuser\_locと観光地の位置を表すクラスsite\_locを定義し、それぞれに緯度longitudeと経度latitudeをprivateなフィールドとして持たせよ。引数としてuser\_locとsite\_locを受け取り、その間の距離をkm単位で近似して返す関数distanceをフレンド関数として定義し、main関数の中で利用せよ。user\_locとsite\_locのインスタンスが持つ経緯度はmain関数の中でコンストラクタを使って設定するとよい。2地点はたとえば東京駅とつくば駅など、適当に決めること。実際の経緯度は「東京駅 経緯度」で検索すると調べられる。有名な場所であればWikipediaに記載がある。距離は後述のように緯度差と経度差から三平方の定理で求まるユークリッド距離で近似してよい。

関数distanceは引数としてuser\_locクラスのインスタンスとsite\_locクラスのインスタンスを受け取るため、たとえばクラスuser\_locの定義をクラスsite\_locの定義より先に書く場合、user\_locのクラス定義の中でfriend double distance(user\_loc u, site\_loc s);と書くと、まだsite\_locが定義されていないため、コンパイルエラーになってしまう。この問題を解決するために使われるのが**前方宣言(forward declaration)**である。すなわち最初にclass site\_loc;とだけ書き、続いてクラス user\_locの定義、その後にクラスsite\_locの定義を書くようにすると、コンパイル可能になる。このようにクラスsite\_locを実際に定義する前にそのようなクラスが存在することだけをclass site\_loc;と書いておくことを前方宣言と呼ぶ。これはプロトタイプ宣言の発展と考えることもできる。

アプリケーションを東京近郊（緯度35.68度）で利用するものとし、地球は赤道の長さが40,000kmの球体であると仮定する。この場合、2地点間の距離は緯度差dlonと経度差dlatから以下で近似できる。ただしsqrt, pow, cosを使うのに#include <math.h>が必要である。

sqrt(pow(dlon \* 40000 / 360, 2) + pow(dlat \* 40000 \* cos(35.68 \* 3.1416 / 180) / 360, 2)); ■

#### 25 実行時型情報

実行時に変数の型を調べるために関数typeidが使用できる。クラスは型の一種であるので、typeidを使うことであるオブジェクトがどのクラスのインスタンスであるかを調べられる。typeidの戻り値はtype\_infoクラスのオブジェクトへの参照であり、nameというメンバ関数を持つ。また、type\_infoクラスについて==と!=がオーバーロードされているので、比較が可能である。たとえば以下のtypeid.cppでは変数instの型（クラス）を表示させている。

/\* typeid.cpp \*/

#include <iostream>

#include <typeinfo>

using namespace std;

class myclass{

int a, b, c;

};

int main(){

myclass inst;

cout << "typeid = " << typeid(inst).name() << endl;

}

#### 25 constとmutable

メンバ変数の値を書き換えず、参照するだけのメンバ関数を定義するのに使うのが予約語constである。メンバ関数の宣言の後にconstを付けると、そのメンバ関数の中では同じクラスのメンバ変数の値を変更できなくなる。constはconstant（定数）の略である。以下のrectangle\_area.cppはconstメンバ関数としてget\_area()を持つ。この中でメンバ変数areaの値を書き換えているため、コンパイルエラーになる。

/\* rectangle\_area.cpp \*/

#include <iostream>

using namespace std;

class rectangle{

double width, height, area;

public:

int get\_area() const {area = width \* height; return area;};

rectangle(double x, double y){width = x; height = y;};

};

int main(){

rectangle rect(20, 30);

cout << "area = " << rect.get\_area() << endl;

}

const指定によってメンバ関数がメンバ変数の値を誤って書き換えてしまうことを防げる。また、constが付いているか否かを見ることでメンバ関数の役割も分かりやすくなる。一部のメンバ変数のみconstメンバ関数からも変更できるようにしたいことがある。この場合はメンバ変数名の宣言の前にmutableを付ける。

　一方、関数定義の際、仮引数の前にconstを付けるとその関数の中で実引数の値が変更されない。これは仮引数がポインタや参照である場合に意味を持つ。仮引数としてポインタや参照を渡した場合、関数の中でポインタや参照が指しているメモリ領域にアクセスできてしまうが、constを付けることで値を変更してしまうことを避けられる。

**【発展課題7】（比較的容易）**

mutableを使うことで上記のrectangle\_area.cppをコンパイルできるようにせよ。■

#### 26 staticクラスメンバ

メンバは通常、オブジェクト（インスタンス）ごとに異なる値を持つ。あるクラスに属するすべてのオブジェクトで共通の値を持たせるのに使うのが予約語staticである。メンバ変数名の前にstaticと書くと、その値はそのクラスの全オブジェクトで共通になる。すなわち値がオブジェクトに対して定まっているのではなく、クラスに対して定まっているとも捉えられる。メンバ関数の前にstaticと書くと、それは staticなメンバ関数となり、インスタンスを生成しなくても実行できる。すなわちmyclass::mymethod()のように書くことで実行できる。

一部の言語（たとえばJava）ではstaticなメンバ変数やメンバ関数を**クラスプロパティ(class property)**や**クラスメソッド(class method)**と呼ぶ。