C++勉強会資料

つよつよになろう！

R02I 鵜飼

はじめに

- この資料はプログラミング初心者～中級者に向けたものです
- 本質的な意味と乖離している部分があるかもしれません
- この資料の二次配布はご自由に(責任は負いません)
- やさしいC++の内容に準拠してます(順序が異なる場合があります)
- 参考コードはgithubのリポジトリに置いています(section対応済み)

github:https://github.com/snct-ukai/Cpp_study_2021

目次

- Gitの使い方(基本のみ)
- C++の基本(記法ルール・入出力・コンパイル)
- 変数・型・演算子・配列
- 関係演算子・条件分岐
- 繰り返し(while・for・do~while)
- 関数・オーバーロード・関数テンプレート

目次(続き)

- 列挙型・構造体・メンバ・演算子
- クラスの定義
- スコープ・記憶寿命
- ポインタ・メモリ・参照・演算子
- クラスの機能・継承・仮想関数
- クラスの高度な部分(演算子関数・変換関数…)

目次(続き)

- クラステンプレート
- 例外処理
- 名前空間
- ストリーム
- 挿入/抽出演算子のオーバーライド

目次(続き)

- ファイル入出力
- 静的メンバ(クラス)
- STL
- マニピュレータ
- コマンドライン引数
- 良いプログラムのために(おまけ)

Git

section1

Gitの使い方

Gitとは

- 簡単に言うと、プログラムファイルを作業状況と共に保存できるシステム
- 履歴も保存されるため正しく使いこなせば安全な開発が可能となる
- 複数人で開発時、ブランチというものを作成することで効率よく開発できる
- Gitのサービスの例としてgithubがある
- リポジトリと呼ばれるフォルダのようなものに保存される

Gitの使い方

インストールしよう

windowsではGitはインストールしないと使うことが出来ません

https://gitforwindows.org/

ここから.exeファイルをダウンロードしインストールしましょう

インストーラーでAdd pathのようなオプションが出たらチェックを入れてください

→環境変数が自動で追加されるので楽になります

Gitの使い方

Gitで必要な最低限のコマンド

- clone
- add
- commit
- pull
- push

clone

cloneとは

リポジトリをclone(複製)するコマンド、リポジトリの内容全て(履歴も)がコピーされる

コマンドの例

```
$ git clone <URL>
```

```
ukai@ukai:~/share/programing/cpp$ git clone https://github.com/snct-ukai/Cpp_study_2021
Cloning into 'Cpp_study_2021'...
warning: You appear to have cloned an empty repository.
ukai@ukai:~/share/programing/cpp$
```

add

addとは

編集したファイルを反映させるための準備のコマンド、追加されたファイルは反映の準備状態になる

コマンドの例

```
$ git add <ファイル>
```

```
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$ git add ./section1/sample1.cpp
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$
```

commit

commitとは

変更をリポジトリへ反映させるコマンド

コマンドの例

```
$ git commit –m “<commit メッセージ>”
```

```
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$ git commit -m "add:section1/sample1.cpp"
[master (root-commit) 9fd671f] add:section1/sample1.cpp
 1 file changed, 6 insertions(+)
 create mode 100755 section1/sample1.cpp
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$
```

pull

pullとは

リポジトリに加えられた変更を適用させるコマンド、複数のコンピュータで同一のリポジトリを用いる場合によく使う

コマンドの例

```
$ git pull <URL>
```

```
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$ git pull
Already up to date.
ukai@ukai:~/share/programing/cpp/Cpp_study_2021$
```

push

pushとは

commitされた変更をgithubなどのgitサービスに反映させるとき使うコマンド

前回のpushからcommitされた内容全てが反映される

コマンドの例

```
$git push
```

C++の基本(記法ルール・入出力・コンパイル)

section2

記法ルール

- 処理の終わりにはセミコロン(;)を打つ
- コメント(メモ書き)はスラッシュ２つ(//)の後に改行せずに書く

  複数行にまたがるときは(/*hogehoge*/)のように

  スラッシュ(/)アスタリスク(*)を組み合わせて囲む
- 大文字と小文字は区別される→間違えないように注意
- main関数が必須

入出力

ここでは標準入出力についてのみ触れます、詳しいことは別セクションに…

- コマンドラインへの出力は cout
- コマンドラインからの入力は cin

出力は挿入演算子(<<)を用いる

入力は抽出演算子(>>)を用いる

コンパイル

コンパイルとは

C++などの人間が理解できるコードはコンピュータ(CPU)が理解できない

→理解できる形に変換する必要がある、これがコンパイル

コンパイルをするプログラムをコンパイラという

この資料ではC++のコンパイラの1つであるg++を用いる

コンパイル

g++の使い方

コマンドラインでコンパイルをする

```
$ g++ <ファイル名> [オプション]
```

オプションは無くても良いので説明しません

基本的にLinux環境ではa.out、Windows環境ではa.exeが生成される

変数・型・演算子・配列

section3

変数・型

変数とは

一つだけ値を入れることができる箱のようなもの

変数には型があり、その型と同じデータしか入れることができない

変数を作ることを宣言すると言う

変数の名前は変数名といい、基本的に自由に決めることができる

同じ変数名の変数を宣言することは基本的にはダメ

型

C++には型が複数ある

- int(整数)
- float(単精度浮動小数点数)
- double(倍精度浮動小数点数)
- char(文字)
- bool(true or false)

演算子(二項演算子)

演算子とは四則演算や代入など、プログラムで演算をする際に使う記号のこと

- 加算(+)
- 減算(-)
- 積算(*)
- 除算(/)
- 代入演算(=)

演算子(二項演算子)

演算子を挟んで左側を左辺、右側を右辺と言う

代入演算をする際、左辺に右辺の値が代入される

演算子には優先順位がある

[*]&[/] > [+]&[-] > [=]

優先順位が同じものは左側から演算される

また余りを取得する剰余演算子(%)も存在する(省略)

演算子(単項演算子)

単項演算子は左辺、右辺が存在しない演算子のこと

- ＋1するインクリメント(＋＋)
- -1するデクリメント(--)
- 符号を反転させる-演算子
- ↑に対の関係の+演算子

が存在する

演算子(単項演算子)

単項演算子の内、インクリメントとデクリメントには前置と後置がある

それぞれ前置インクリメント(デクリメント)、後置インクリメント(デクリメント)

と言う

演算子を変数の前に置くことを前置、後ろに置くことを後置と言う

配列

配列とは、変数が複数連結されたもののこと

連結する数を配列の大きさと言う

C++では配列の大きさを指定して宣言する

大きさnの配列のm番目は、コード上ではm-1番目となる

つまり、0~n-1番目までの変数が作られる

配列

配列の宣言は

```
<型名> <変数名>[<大きさ>];
```

n番目の要素を取り出したい場合は

```
<変数名>[n-1];
```

関係演算子・条件分岐

section4

関係演算子

関係演算子とは左辺と右辺を比較する演算子のこと(比較演算子)

左辺と右辺の関係が関係演算子を満たすならばtrue、そうでないならば

falseが値として出力される

比較なので左辺もしくは右辺の変数(または定数)が変わることはない

比較しtrue/falseを判定することを評価すると言う

関係演算子

関係演算子は複数ある

- 小なり(＜)・小なりイコール(＜＝)
- 大なり(＞)・大なりイコール(＞＝)
- 等価(＝＝)
- 非等価(!＝)

関係演算子

等価とは

左辺と右辺が同じ値であることを評価

非等価とは

左辺と右辺が異なる値であることを評価

関係演算子

評価する際に使う単項演算子

- 論理否定演算子(!)

!<変数/評価式>

とすることでtrue/falseが反転する

↓

!(等価評価式) ＝ 非等価評価式

条件分岐

評価した結果に応じて処理を分岐させたい場合に条件分岐を用いる

C++で条件分岐をするにはif-else文を用いる

if文で評価させる式のことを条件式と言う

```
if(<条件式>){}else{}
```

条件式には評価式の他、bool型やint型などが入る

int型などの数字が入る変数は0のときfalse、それ以外はtrueで評価

条件分岐

{}で囲ったものをブロックと言う

条件式が満たされたとき、if文のブロックが、満たされなかったときelse文のブロックが実行される

if文のブロックとelse文のブロックを両方実行することは基本的にはできない

条件分岐

if-else文には

```
else if(<条件式>){}
```

という構文もある

これはif文と組み合わすことで「もしそうでないとき、もし{条件式を満たす}ならば」のような処理が可能となる

条件分岐

if-else文の他にswitch文というものもある

→if(),else if()が複数個ある場合に有効

if文では条件式により分岐したがswitch文では変数を用いて分岐させる

```
switch(<変数>){}
```

条件分岐

switchブロックの中にcaseキーワードと値を組み合わせ分岐先を作成する

→ 変数＝＝指定した値 となれば分岐先が実行される

処理の終わりにはbreak文を置かなければならない

```
case <値>:
//処理
break;
```

条件分岐

switch文でcaseだけではelseにあたる処理がかけない

→defaultキーワードを使う

defaultに書かれた処理はどのcaseにも当てはまらない場合に実行される

switchブロックの一番下に書きbreakはいらない

```
default:
//処理
```

条件分岐

三項演算子

<条件式> ? <trueのとき> : <falseのとき>

true(false)のときに指定できるものは定数または変数のみ

→処理は指定できない(javascriptなど処理を指定できる言語もある)

繰り返し(while・for・do~while)

section5

while

while文とは

```
while(<条件式>){}
```

条件式を満たしている間ブロック内をループする構文

trueを指定すると無限ループになる

for

for文とは繰り返す回数を簡単に指定ができる構文

```
for(int i = <初期値>; <条件式>; <増減式>){}
```

while文と同じで、条件式を満たしている間ブロック内がループされる

言葉で説明してもイメージがしづらいので次のスライドでサンプルコードを示す

for

```
for(int i = 0; i < 100; i++){}
```

これが意味するのは、int型変数iが100未満のとき、ブロック内を実行してiを

インクリメントするということ

iがインクリメントされ続けると100回目が終わったタイミングでi == 100になり、

条件式を満たさなくなるためループが終了する

iの初期値や条件式を変更すれば回数を簡単に指定できる

do~while

do~while文はwhile文と似たような性質を持つ

```
do{}while(<条件式>);
```

異なる点は、while文では始めから条件式を満たしていない場合は1回も実行されないが、do~while文では少なくとも1回はブロックの中が実行される

繰り返し

繰り返し処理のブロック内で使える特殊なキーワード

- continue;

  繰り返し処理の残りをスキップし次のループに移る

- break;

  繰り返し処理をその時点で終了し、繰り返しブロックの下の処理に移る

これらを適切に使うことでより良いプログラムが作成できる

関数・オーバーロード・テンプレート

section6

関数

関数とは

プログラミングにおける関数は、あらかじめ複数の処理をパッケージにしておき、

それを呼び出すことでまとまった処理が実行されるようにできるもの

関数には実行時に渡す引数と、終了時に投げ返される返り値がある

関数(定義)

関数を作ることを定義すると言う

関数定義の例

```
<返り値の型>　<関数名>(<引数リスト>){}
```

関数名は自由に決めることができるが、すでに宣言している変数名や関数名、

またC++で使われているキーワード(doなど、予約語と言う)は基本的にダメ

関数(定義)

関数プロトタイプ

関数の定義時、関数名と関数の処理内容を分けて書くことが出来る

その場合の関数名を定義する部分を関数プロトタイプと言う

呼び出し元よりも下の行で定義している関数は呼び出すことができない

その場合は上の行でプロトタイプを宣言することで呼び出せるようになる

関数(返り値)

返り値とは

その関数を実行することにより得られた演算結果のこと

関数を定義する際は返り値の型を指定する必要がある

何も値を返さない場合はvoid型にする

関数(引数)

引数とは

関数内の処理に渡す値のこと、仮引数と実引数の２つある

仮引数とは関数の定義時に指定する引数のこと

実引数とは関数の呼び出し時に指定する引数のこと

具体的なコードは次に示す

関数(引数)

定義

```
int plus(int a, int b){
        int sum = a + b;
        return sum;
}
```

int aとint bが仮引数に相当する

呼び出し

```
int main(void){
        int x = 10;
        int y = 20;
        int z = plus(x, y);
}
```

int xとint yが実引数に相当する

関数(引数)

デフォルト引数

デフォルト引数とは、関数呼び出し時に引数が指定されなかった場合に使われる引数の値、下のようにすることで指定できる

```
int plus(int x = 10, int y = 20);
```

デフォルト引数を指定しなかった場合、呼び出し時に引数を指定する必要がある

オーバーロード

オーバーロードとは

同じ名前の関数を多重定義すること

挙動が同じなのに引数の型が違うから関数名を変えないといけない…

→これはプログラムを書いていると不都合

オーバーロードすると同じ名前の関数が定義でき、呼び出し時の引数の型から

適した関数をその中から推測し、自動で選んでくれる

オーバーロード

C++ではオーバーロードは標準機能なので一般的な関数の宣言で用いることが出来る

→特別何かを指定する必要はない

引数の型が同じものを２つ以上作るとオーバーロードができなくなりコンパイルエラーの原因となる

関数テンプレート

関数のオーバーロードでは引数の型が違う場合、そちらも定義することにより同じ関数名で使えるようにした

→int と doubleのように２つだけなら問題は無いが複数の型でオーバーロードするとコード量が多くなりメンテナンス性を損ねてしまう

そこで関数のテンプレートを宣言して使いまわそうというのが関数テンプレート

関数テンプレート

```
template<class T>
T hogehoge(T x, T y…){}
```

このコードのTをテンプレート引数と言い、関数呼び出し時に与えられた引数の型に置き換えられる

このようにすることで関数を一つ定義するだけで複数の型に対応できる

列挙型・構造体・メンバ・演算子

section7

列挙型

列挙型とは

識別子を値として格納できる型のこと

識別子とはそれだけで意味を持てる固有名詞のようなもの

識別子は自由に決められるが、全て大文字にすることが多い

識別子を決められるため、どのような値が入る型かを自由に決められる

列挙型(定義)

```
enum <型名>{<識別子1>, <識別子2>, <識別子3>, …}
```

このようにして定義する

定義をした列挙型はint型などの変数のように宣言することができる

値を代入したい場合は識別子を代入演算子を使って代入する

列挙型(定義)

列挙型の定義時、識別子に特定の値を紐付けることが出来る

```
enum Week{SUN = 1, MON = 1, TUE = 3,…}
```

この場合、Week型の変数に0を代入することでSUNを代入する処理と同じになる

曜日はZellarの公式による計算で0~6で表せるのでそういった場合に便利

ツェラーの公式 - Wikipedia

構造体・メンバ

構造体とは

複数の変数をまとめた新しく定義できる型のこと

例としてナンバー(int)と型番(char[])、燃費(int)がある車のようなもの

→車という1つのオブジェクトに3つの情報がまとまっている

構造体の中に存在する変数のことをメンバという

構造体(定義)

```
struct <構造体名>{<型名> <変数名>;…};
```

定義した構造体は、構造体名を使うことで一般的な型のように宣言可能

構造体のメンバに関数も追加できるクラス(後述)を使うことのほうが多い

構造体・メンバ・演算子

構造体を宣言したあとメンバにアクセスをしたい

そのときにはドット演算子(.)を用いる

<変数>.<メンバ識別子>

メンバ識別子は構造体定義時に指定した変数名のこと

ドット演算子を使いメンバにアクセスすると一般的な変数のように代入などの

各種演算が可能となる

クラスの定義

section8

クラス

クラスとは

構造体のメンバに変数だけでなく関数も追加できるオブジェクト

クラス独自の機能や挙動が多い(section11・12にて後述)

メンバへのアクセスを制限するアクセス指定子がある

クラス(定義)

```
class <クラス名>{<メンバ変数・関数>};
```

メンバ変数や関数は一般的な宣言・定義方法で追加ができる

定義したクラスはクラス名を型として宣言できる

クラスにはオブジェクト生成時と破棄時に実行される特別な関数(コンストラクタ・デストラクタ)を持たせることが出来る

ドット演算子を用いてアクセス可能なメンバにアクセスできる

クラス(アクセス指定子)

メンバにアクセス出来る範囲を指定するもののこと

- private(そのクラス内でのみアクセス可能)
- protected(private+継承先クラス(後述)でのみアクセス可能)
- public(protected+オブジェクトからドット演算子でアクセス可能)

の3つある

private < protected < public

クラス定義の例

定義

```
class Car{
private://アクセス指定子(クラス内からのみ)
    int num;
    string cat;//string(文字列)型(後述)
public:
    Car(){//コンストラクタ
        num = 1;
        cat = “H○NDA NB○x”;
    }
    ~Car(){}//デストラクタ
    int getnum(){return num;}
    string getcat(){return cat;}
};
```

使用方法

```
Car car;//Carオブジェクト生成
car.getnum();


car.num;//これはできない(∵private指定)
```

スコープ・記憶寿命

section9

スコープ

スコープとは

変数(オブジェクト)が有効な範囲のこと、宣言した範囲の外側ではその変数(オブジェクト)には基本的にはアクセスができない

for文やif文などによりブロックが入れ子になっている場合に考える事が多い

section2では同じ変数名はダメと伝えたが、スコープ範囲が異なっていれば重複していても問題はない(おすすめはしない)

スコープ

外側のブロックはスコープが広いと言い、内側のブロックになるほどスコープが狭いと言う

異なるスコープで変数名が重複している場合、スコープの狭い方の変数が優先される

外側のスコープの変数を用いたい場合は名前空間(後述)を用いるか諦める

グローバル変数はスコープ解決演算子を先頭につけることでアクセス可能

スコープの例

```
int main(void){
    int x;//①
    while(true){
        int x;//②
        x = 10;//②のxに10を代入
    }
}
```

記憶寿命

変数には寿命があり、そのことを記憶寿命と言う

宣言方法により動的記憶寿命と静的記憶寿命の2つある

動的記憶寿命:

```
<型名> <変数名>;
```

静的記憶寿命:

```
static <型名> <変数名>;
```

動的記憶寿命

動的記憶寿命とは

変数(オブジェクト)が宣言されるコードが実行されるときに生成され、宣言されたブロック内の処理が全て終わったタイミングで寿命が尽きること

→使われる可能性のある間のみ生きている

静的記憶寿命

プログラムが実行されたタイミングで変数(オブジェクト)が生成される

プログラムが終了するまで寿命が尽きることはない

使われる可能性の無いタイミングでも生きていて値が保存されているため、再びブロックを実行するなどのときに使うと便利になる

静的記憶寿命にしたい場合staticキーワードを用いる

記憶寿命

動的記憶寿命

```
int main(){
    if(1 == 1){
        int x;
        x = 1;
    }
}
```

静的記憶寿命

```
int main(){
    if(1 == 1){
        static int x;
        x = 1;
    }
}
```

int型変数xの寿命を赤色で示す

ポインタ・メモリ・参照・演算子

section10

ポインタ

ポインタとは

プログラム中で宣言した変数やオブジェクトはコンピューターのRAMに記憶領域が確保される

記憶領域の場所(住所のようなものアドレスと言う)のことをポインタと言う

ポインタを格納する変数をポインタ変数と言う

メモリ

確保された記憶領域のことをメモリと言う

記憶領域を確保することをメモリの確保、確保したメモリを削除することをメモリの解放と言う

動的記憶寿命や静的記憶寿命ではプログラムで自動的にメモリの確保・解放がされる

メモリ

RAMには予め1バイト毎に16進数でアドレスが割り振られている

一つの変数で使われるバイト数は定められている

変数のメモリが確保された→メモリ上の空いているアドレスに変数を作成

複数バイト使用する変数は隣り合うアドレスを連結して確保する

複数バイト使用する変数を示すアドレスは一番目のアドレスになる

メモリ

主要な型のバイト数

- int：4バイト
- float：4バイト
- double：8バイト
- char：1バイト
- bool：1バイト

メモリ

メモリのイメージ図(int型)

0x0080	0x0081	0x0082	0x0083	0x0084	0x0085	0x0086	0x0087
int型変数							

確保された　　確保されていない(使われていない)

メモリ

配列を確保したときは連続するアドレスを使用する

int型配列[8]の場合(0x0080から0x009Fまでを使用)

0x0080	**0x0084**	**0x0088**	**0x008C**	**0x0090**	**0x0094**	**0x0098**	**0x009C**
int型変数	int型変数	int型変数	int型変数	int型変数	int型変数	int型変数	int型変数
int型配列[8]							

配列を示すアドレスは配列の先頭のアドレス

ポインタとメモリ

ポインタとはメモリ上に存在する変数を示すアドレスのこと

ポインタ変数にポインタを格納することができる

ポインタには変数(オブジェクト)の型に対応した型が存在し、ポインタ変数を宣言する際は何型のポインタを格納するかを指定する必要がある

ポインタ・演算子

宣言されている変数のポインタを取得するとき、ポインタ演算子(&)を用いる

```
&<変数>
```

配列の名前は先頭アドレスを示す→配列の先頭アドレスを取得したい場合は

ポインタ演算子を使わず配列の名前を使えば良い

ポインタ

ポインタ変数の宣言

hogehoge型の変数のポインタを格納するポインタ変数

`hogehoge* <変数名>;` または `hogehoge *<変数名>;`

ポインタ変数にはアドレス演算子で取得したアドレスを代入できる

ポインタ・演算子

ポインタが示す変数の値を取得するとき、間接参照演算子(*)を用いる

`*<ポインタ変数>`

int*型のポインタ変数に間接参照演算子を使った場合、int型として使える

→int型にできる一般的な演算・評価が可能

間接参照演算子+hogehoge*型ポインタ変数→hogehoge型変数

ポインタ・演算子

ポインタは変数だけでなく構造体やクラスにも存在する

ポインタ変数が示すオブジェクトのメンバを取得したい場合はドット演算子を用いることが出来ない

→ドット演算子の代わりにアロー演算子(->)を用いることで取得可能

```
<ポインタ変数> -> <メンバ変数/関数>
```

ポインタ(演算)

配列の要素のポインタではいくつかの演算が可能

- 加算(+m)

  →ポインタ変数がn番目の要素を示しているとき、n+m番目を示す

- インクリメント

  →ポインタ変数が示す要素の1つ次の要素

ポインタ(演算)

- 減算(-m)

  →ポインタ変数がn番目の要素を示しているとき、n-m番目を示す

- デクリメント

  →ポインタ変数が示す要素の1つ前の要素

- ポインタ同士の減算

  →間にある要素数(5番目-3番目なら2)

参照

参照とは

ある変数の記憶領域を共有しアクセスできるようにすること

参照型変数を宣言することで使える

すでに存在する変数にあだ名を付けて使えるようにするといったイメージ

宣言時にどの変数を参照するか初期化してあげる必要がある

参照先は変更できない

参照

宣言方法

```
<型名>& <変数名> = <宣言済み変数>;
```

このとき、型名と宣言済み変数の型は一致していなければならない

参照型変数はそれに対応する型の値を代入することができる

→参照先の変数に代入がされる

```
<参照型変数(hogehoge型)> = <hogehoge型の値>;
```

ポインタ・参照と引数

関数の引数はポインタ変数または参照型変数でなければ値のコピーが入る

→実引数に指定した変数には関数内で変更ができない

関数の引数にポインタ変数または参照型変数を指定することで関数内で実引数の保存されているデータを変更することが出来る

仮引数にポインタ変数を指定することをポインタ渡しと言う

仮引数に参照型変数を指定することを参照渡しと言う

ポインタ渡しの例

定義

```
void swap(int* x, int* y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

呼び出し

```
int main(){
    int a = 10;
    int b = 20;
    cout << a << “¥t”<< b << endl;
    swap(&a, &b);
    cout << a << “¥t“<< b << endl;
}
```

出力結果：

10　　20

20　　10

参照渡しの例

定義

```
void swap(int& x, int& y){
    int tmp = x;
    x = y;
    y = tmp;
}
```

呼び出し

```
int main(){
    int a = 10;
    int b = 20;
    cout << a << “¥t”<< b << endl;
    swap(a, b);
    cout << a << “¥t“<< b << endl;
}
```

出力結果：

10　　20

20　　10

コピーを渡すと…

定義

```
void swap(int x, int y){
    int tmp = x;
    x = y;
    y = tmp;
}
```

呼び出し

```
int main(){
    int a = 10;
    int b = 20;
    cout << a << “¥t”<< b << endl;
    swap(&a, &b);
    cout << a << “¥t“<< b << endl;
}
```

出力結果：

10　　20　　**←入れ替わらない！**

10　　20

ポインタ(参照)渡しとコピー渡し

ポインタ(参照)渡しは関数の中で実引数のメモリを直接編集する

コピー渡しは関数に実引数の値のコピーが渡され、関数内で新しくメモリを確保する→実引数に影響がない

ポインタ(参照)渡しにすると関数実行時に新しくメモリが確保されない

→動作が速くなる

メモリの動的確保

動的記憶寿命や静的記憶寿命の変数はプログラム中で自動でメモリの確保や開放をしてくれる→手動で確保や解放ができない

new演算子(new)とdelete演算子を使うことで手動で行うことが出来る

new演算子を用いると空いているアドレスに変数が生成される

delete演算子を用いると指定したアドレスを解放できる

どちらもポインタ変数を指定したり代入することが多い

メモリの動的確保

new演算子

```
int* pa = new int;
//int型変数の確保

int* parray = new int[8];
//int型配列[8]の確保

int* pb = nullptr;
//特別なアドレスでメモリ上のどこも示さない
pb = new int;//変数宣言と同時でなくても良い
```

delete演算子

```
delete pa;
//配列でない変数のメモリ解放

delete[] parray;
//配列のメモリ解放

delete pb;
```

ポインタ・演算子

ポインタ変数に配列の変数を作成する、各要素は一般的な配列と同じで配列参照演算子([])を用いて取り出すことが出来る

```
int* array = new int[8];
```

```
array[2];
```

配列参照演算子を使うと上の例ではint型になり一般的な演算が可能

クラスの機能・継承・仮想関数

section11

クラスの機能

(復習)

クラスとは複数の変数や関数を1つのオブジェクトとして定義できるもの

クラス内に存在する変数をメンバ変数と言う

クラス内に存在する変数をメンバ関数と言う

定義したクラスはコード内でオブジェクトとして宣言できる

クラスの機能

コンストラクタ

オブジェクトの生成時に自動的に実行される関数

メンバ変数の初期化やその他生成時にさせたい処理を記述する

一般的な関数と同様に引数を持たせることができ、引数の種類や数に応じて

オーバーロードも可能

クラスの機能

デストラクタ

オブジェクトの破棄時に自動的に実行される関数

メンバ変数にポインタがあり、手動でメモリを開放する必要がある場合など、破棄時にしなければならない処理がある場合に記述する

引数を取ることはできないため1つしか定義できない(オーバーロード不可)

コンストラクタ/デストラクタの定義

コンストラクタ

```
class Car{
private:
    int* x;
    string str;
public:
    Car() : x(nullptr), str(“”){}
    //処理が高速化される初期化方法
    Car(unsigned int size, string s){
        x = new int(size);
        str = s;
    }
};
```

デストラクタ

```
class Car{
private:
    int* x;
    string str;
public:
…
    ~Car(){
        delete[] x;
    }
};
```

継承

すでに存在するクラスをベースとして新しくクラスを定義することができる

これを継承と言い、継承して作られたクラスを派生クラス、継承元のクラスを基本クラスと言う

派生クラスには基本クラスのprotectedまたはpublicのメンバが全て含まれる

派生先と継承元で同一名の変数または関数がある場合、一般的なメンバ呼び出し方法では派生先のものが呼び出される

継承

継承方法

```
class <派生クラス名>: <アクセス指定子> <基本クラス>{};
```

アクセス指定子にはprivate/protected/publicのどれかを指定する

基本クラスに指定するアクセス指定子は、基本クラスのメンバのアクセス範囲を狭めるために指定する

継承

派生クラスでの基本クラスのメンバのアクセス範囲表

		基本クラスのメンバのアクセス指定子		
		private	protected	public
基本クラスに指定するアクセス指定子	private	private	private	private
	protected	private	protected	protected
	public	private	protected	public

継承

派生クラスのオブジェクトを生成すると、コンストラクタは基本クラス→派生クラスの順に実行される

派生クラスのオブジェクトを破棄すると、デストラクタは派生クラス→基本クラスの順に実行される

メンバ変数・関数は一般的に派生クラスのものが優先され、派生クラスに存在せず基本クラスに存在するものは基本クラスのものが使われる

継承

あるクラスAを基本クラスとした派生クラスBが存在する、このときクラスBを派生したクラスCを作ることが可能

→クラスCにはクラスAとクラスBの要素が受け継がれている

派生クラスから見て直接継承しているクラス(この例ではB)を直接基本クラス、継承元クラスが継承しているクラス(この例ではA)を間接基本クラスと言う

継承

C++では派生クラスの定義時に基本クラスを複数指定することが可能

これを多重継承と言う

多重継承では継承クラスごとにアクセス指定子を指定し、カンマで区切る

継承元が複数あり、それぞれのメンバの名前が被っている場合、スコープ解決演算子(::)を用いてどのクラスのメンバかを指定する

<クラス名>::<メンバ変数・関数>

継承

多重継承時、各直接基本クラスが同じクラスを継承している場合がある

この場合、間接基本クラスにアクセスしたいときは基本的にはどちらの基本クラスが継承している間接基本クラスのメンバなのかを指定する必要がある

→1つのオブジェクトに同じオブジェクトが複数重なっている状態(菱形継承)

これでは間接基本クラスのメンバを扱いづらい

→直接基本クラスの定義で基本クラスをvirtual指定する(仮想基本クラス)

仮想基本クラス

仮想基本クラスとは継承時にvirtual指定した基本クラスのこと

菱形継承時に間接基本クラスが仮想基本クラスになっている場合、派生クラスでは間接基本クラスを1つしか持たない

菱形継承時に仮想基本クラスを用いることでスコープ解決演算子を用いること無くメンバを指定できるため基本的にはvirtual指定する

コード量が多いためサンプルコードは割愛(github参照)

クラスとポインタ

クラスのポインタ変数にnew演算子を用いて動的にオブジェクトを生成できる

new演算子で生成した場合、delete演算子で破棄する必要がある

基本クラスのポインタ変数には派生クラスのポインタを代入することが可能

基本クラスの参照型変数は派生クラスを参照することが可能

基本クラスのポインタ変数に派生クラスのポインタを代入しアロー演算子でメンバを取得しようとした場合、基本クラスのメンバが優先される

クラスとポインタ

クラスの定義

```
class Base{
private:
    int x;
public:
    int getx(){return x;}
};

class Derived: public Base{
public:
    int getx(){return 10;}
}
```

オブジェクト生成(ポインタを使用)

```
int main(){
    Base* a;
    a = new Derived;
    a -> getx();//Baseクラスのgetx()が実行

    delete a;
}
```

仮想関数

基本クラスのポインタから派生クラスのメンバ関数を呼び出したい場合、関数名が同じだと基本クラスのメンバ関数が優先されてしまう

→ポインタが派生クラスのオブジェクトを示している場合、動作が理解しづらい

基本クラスと派生クラスでメンバ関数の名前が同じ場合、派生クラスが常に実行されてほしい

→基本クラスのメンバ関数を仮想関数(virtual指定)にする

仮想関数

基本クラスのメンバ関数で virtual キーワードを付けることで定義可能

virtual <返り値の型> <関数名>(<引数リスト>){}

virtual指定することにより、ポインタ変数の型が基本クラスの場合でもポインタが示すクラスのメンバ関数が実行されるようになる

基本クラスのポインタ変数にdeleteをした場合、基本クラスのデストラクタが呼ばれてしまうため、基本クラスのデストラクタにはvirtual指定をする

継承・仮想関数

基本クラスは出来る限り抽象的にし、派生クラスで具体化する

→基本クラスに処理が無いメンバ関数を作成し、派生クラスで処理を指定

処理がないメンバ関数はブロック内を空にすれば良いが明示的に処理が無いことを示したい

→純粋仮想関数を用いる

純粋仮想関数

処理を持たない(実体がない)特別な関数

関数名しか無くメンバ関数として呼び出すことは出来ない

派生クラスでオーバーライドすることが必要になる

定義は↓のようにする

```
virtual <返り値の型> <関数名>(<引数リスト>) = 0;
```

仮想関数・オーバーライド

基本クラス

```
class Pet{
    string name;
public:
    string getName(){return name;}
    virtual void play(){
        cout << “play” << endl;
    } //仮想関数
    virtual void bark() = 0; //純粋仮想関数
    virtual ~Pet(){}//デストラクタを仮想関数化
}
```

派生クラス1

```
class Cat : public Pet{
public:
    void play(){
        cout << “with cat tower” << endl;
    }
    //play関数をオーバーライド
    void bark(){
        cout << “nya~” << endl;
    }
    //bark関数をオーバーライド
}
```

仮想関数・オーバーライド

派生クラス2

```
class Dog : public Pet{
public:
    void play(){
        cout << “in dog run” << endl;
    }
    //play関数をオーバーライド
    void bark(){
        cout << “wan” << endl;
    }
    //bark関数をオーバーライド
}
```

オブジェクト生成・メンバ関数呼び出し

```
int main(){
    int* mypet = new Cat;
    mypet -> bark();//出力 : nya~
    delete mypet;

    mypet = new Dog;
    mypet -> play();//出力 : in dog run
    delete mypet;
}
```

仮想関数・オーバーライド

それぞれの派生クラスは同じ基本クラスを継承している、また基本クラスに派生クラスが持っている関数の情報(返り値の型・関数名・引数リスト)がある

→生成するオブジェクトを変えるだけで動作がオーバーライドされ修正が容易

→基本クラスのポインタ変数を用いて派生クラスのオブジェクトを生成することでコード量を削減し、メンテナンス性を向上できる

クラス内の特別なポインタ

クラスには自分自身を示す特別なポインタ、thisポインタが存在する

仮引数とメンバ変数名が被っている場合にメンバ変数を指定するとき、引数に渡されたクラスのポインタと自身のポインタを比較したい場合などに使われる

thisポインタはクラスのポインタなので間接参照演算子を用いてクラスのオブジェクトを参照したり、クラスのポインタ変数に代入が出来る

クラスの高度な部分(演算子・変換関数…)

section12

演算子のオーバーライド

クラスは自分で定義するため演算子の挙動が定義されて無く、そのままでは一般的な演算子を用いて演算をすることができない

演算子を用いたときに動作する関数を定義することで使用できるようにする

定義方法

クラス ＋ 変数・オブジェクトのとき

<返り値> operator<演算子>(<右辺で受け取る変数・オブジェクト>)

演算子のオーバーライド

引数に取るものがオブジェクトでかつ、演算子関数を持っているオブジェクトと型が同じ場合は関数内で引数のprivateメンバにアクセスできる

引数にはオブジェクトだけでなくint型などの変数を取ることも可能

演算結果はオブジェクトのコンストラクタを呼び出すことでコードが簡略化可能

→出来る限り処理が少ないコードを意識

演算子のオーバーライド

引数に取るものは出来る限りポインタ渡しまたは参照渡しを用いる

→メモリが新しく確保されずメモリ効率が良くなる

クラスは変数とは比べ物にならないくらいの大きいオブジェクト、動作の速いクラスを定義することでパフォーマンスを向上させることができる

引数の値を変更することがない場合はconst参照を用いる

const参照とは関数内で値を変更できないようにする参照方法

演算子のオーバーライド

定義例

```
class Vector{
public:
    int x,y;
    Vector(int a, int b){
        x = a;
        y = b;
    }
    Vector operator+(
        const Vector& vec){//const参照
        return Vector(vec.x+x, vec.y+y);
    }
};
```

使用例

```
int main(){
    Vector vec1(10, 20);
    Vector vec2(30, 0);
    vec1 = vec1 + vec2;
    //→ vec1 = vec1.operator+(vec2);
    cout << vec1.x << “¥t”;
    cout << vec1.y << endl;
    //出力 : 40          20
}
```

演算子のオーバーライド

左辺値の演算子関数が実行されるため、左辺値がint型などのC++の標準的な型の場合はオーバーライドできない

→メンバ関数ではなくフレンド関数として演算子関数を定義する

friend <返り値の型> <関数名>(<左辺値>, <右辺値>)

フレンド関数はクラスのメンバ関数ではないためクラスの外で処理を実装するのが普通、クラス内ではプロトタイプ宣言のみ

演算子のオーバーライド

定義

```
class Vector{
    friend Vector operator+(
        int a, const Vector& vec);
}

Vector operator+(int a, const Vector&
vec){
    return Vector(vec.x+a, vec.y);
}
```

使用例

```
int main(){
    Vector vec1(10,20);
    int x = 100;
    vec1 = x + vec1;

    cout << vec1.x << “¥t”;
    cout << vec1.y << endl;
    //出力 : 110        20
}
```

演算子のオーバーライド

演算子関数とは演算子が使われた際に実行される関数

引数はポインタ渡しまたは参照渡しを用いてメモリ効率を向上させる

実引数を変更する必要がないときはconst参照を用いる

返り値を参照にすることでメモリが確保される量を極限まで減らすことが可能

→高速な動作が実現できる

演算子のオーバーライド

メンバ変数に配列の要素があり、クラスに配列参照演算子を使用した場合、メンバ変数内の配列から値を取り出す処理をさせたい

→配列参照演算子をオーバーライドする

配列参照演算子をオーバーライドする場合は返り値の型は配列の各要素の型の参照になる

配列参照演算子の中の値が実引数となる

演算子のオーバーライド

定義

```
class Vector{
int* p;
public:
    Vector(){p = new int[20];}
    int& operator[](int n){
        return p[n];
    }
}
```

使用例

```
int main(){
    Vector vec;
    for(int i = 0; i < 20; i++){
        vec[i] = i;
        //メンバ変数pにある配列の要素にアクセス
    }
}
```

コピーコンストラクタ・代入演算子

すでに存在するオブジェクトの値をコピーした新しいオブジェクトを生成することがある場合は、コピーコンストラクタを定義する

→コピーコンストラクタを定義せず代入演算子を使って代入するとバグの原因となる

代入演算子は浅いコピーをするため深いコピーをするようにオーバーライドする必要がある

浅いコピーと深いコピー

浅いコピーとはクラスのメンバ変数の値をそのままコピーすること

→メンバ変数にポインタ変数がある場合、コピー先にはアドレスがコピーされる

深いコピーとはクラスのメンバ変数が示す値をコピーすること

→ポインタ変数に新しくメモリを確保し、そこにコピー元のアドレスにあるデータを代入していく

深いコピーでないとコピー元とコピー先でデータが同期されてしまう

コピーコンストラクタ

コピーコンストラクタを定義しない場合、生成と同時にコピー元で初期化をしようとすると代入演算子により浅いコピーが実行される

(代入演算子の動作が定義されている場合は演算子関数が実行される)

不具合の原因となるのでオブジェクト生成時にクラスが渡された場合のコンストラクタ(コピーコンストラクタ)を定義する必要がある

コピーコンストラクタ

コピーコンストラクタの定義は一般的なコンストラクタと同じようにすれば良い

```
<クラス名>(<クラスオブジェクト>){}
```

コピーコンストラクタでは、ポインタではない変数はそのままコピーし、ポインタ変数の場合は新しくメモリを確保し、コピー元のデータを新しいメモリに代入する

参考コードはコード量が多いので割愛(github参照)

代入演算子

代入演算子はオーバーライドしていない状態だと浅いコピーをする

→コピーコンストラクタの処理と同じことをする演算子関数を定義

メンバ変数にポインタ変数を持っている場合、使われていないメモリが発生しないようメモリを確保する前にすでに存在するメモリを解放する必要がある

また、コピー元とコピー先が同じ場合、実行しても意味がなく動作が遅くなる原因となるのでアドレスを比較して処理を回避する必要がある

代入演算子

代入演算子のオーバーライドの定義

```
<クラスの参照型> operator=(<クラスのconst参照>){}
```

引数はクラスを取り、変更する必要がないためconst参照を用いる

返り値は他の演算子関数と違いクラスの参照を返す

→オブジェクトのコピーでなくオブジェクトそのものを返すため

コピーコンストラクタと同様参考コードは割愛

変換関数

int型変数にdouble型変数を代入するとint型に型変換される

→このような変換を暗黙の型変換と言う

static_cast<型名>(<変数>)

を用いて明示的に型を変換することもできる

このような変換をクラスに施したい場合、変換先の型毎に変換関数を定義することで可能となる

変換関数

変換関数とは暗黙の型変換や明示的な型変換が行われるときに実行される

特別な関数

```
operator <変換先型>(){return <変換後の値>;}
```

とすることで定義することが可能

返り値は変換先の型で推測が可能なため指定する必要はない

変換関数

定義(Vector => int)

```
#include<cmath>
class Vector{
int x,y;
public:
    operator int(){
        return std::sqr(x*x + y*y);
    }
}
```

使用例

```
int main(){
    int x = 0;
    int y = 0;
    Vector vec(30,40);
    x = vec;//変換関数が実行
    y = satatic_cast<int>(vec);
    //変換関数が実行 この時点でx == y

    cout << x << endl;
    //出力 : 50
}
```

クラステンプレート

section13

クラステンプレート

関数テンプレートは型が違っても同じ処理を施したい場合に用いた

クラステンプレートは処理が同じだがメンバ変数の型が異なる場合などに用いることでクラスの定義を1つで済ませることが可能

→コードを簡略化できる

クラステンプレートは関数テンプレートと同じように仮の型を定義して作成する

クラステンプレート

テンプレート化したクラスのオブジェクトを生成したい場合、メンバ変数の型を指定する必要がある

→指定せずにオブジェクトを生成することはできない

クラスのポインタ変数も指定した型により区別される

→異なる型同士を代入することは出来ない

クラステンプレート

定義

```
template <class T>
class Vector{
T* p;
size_t size;//sizeを表すための自然数変数
public:
    Vector(size_t s = 1){
        size = s;
        p = new T[size];
    }
    T* getP(){return p}
}
```

使用例

```
int main(){
    Vector<int> veci(10);//p=int
    Vector<double> vecd(3);//p=double

    int* ip = veci.getP();
    double* dp = vecd.getP();
    //Vector vec;はできない(型指定必須)
}
```

クラステンプレート

クラステンプレートを用いることでクラスの定義数を減らせることがある

→メンテナンス性が向上

テンプレート化したクラスオブジェクトを生成するときは型を指定する必要がある

メンバ関数をクラス外で処理の定義をする場合、関数テンプレートを使用する必要がある

例外処理

section14

例外処理

例外とは

プログラム中に発生する、処理ができないまたは意図しない処理がされようとしている状態のこと

例外が発生した場合の処理を例外処理といい、適切に処理を定義することで安全なプログラムとなる

例外処理

どういう状態が例外なのかは基本的に自分で定義しなければならない

→例外の判別をプログラムで評価し、例外が発生したことを知らせる

例外が発生したと断定できるとき、throwキーワードを用いることで例外の発生をプログラム全体に知らせることが出来る

```
throw <例外内容/オブジェクト>;
```

今後throwキーワードを使うことを例外を投げると言う

例外処理

例外を投げる際、例外内容は何らかのオブジェクトや文字列で表す

→例外内容を見たとき誰でも理解できるように心がける必要がある

例外オブジェクトは stdexcept として標準で用意されている

stdexceptには例外オブジェクトの基本クラスであるexceptionや、それから

各例外に特化させたクラスが存在する

→適切な例外オブジェクトを選択することが大切

例外処理

例外を投げるとそこでブロック内の処理が停止し例外処理に移る

投げられた例外は決まったブロック内でしか受け取ることが出来ない

例外を受け取るためにtry～catch構文を用いる

try{}catch(＜例外オブジェクトの型＞){}

tryの中で例外が発生し、catchの引数で指定した例外オブジェクトの型と一致した場合のみ例外を受け取ることが出来る

例外処理

例外を受け取ったときの処理はcatchブロックの中に定義する、このときcatchに指定した引数には例外が入っておりコード中で使うことが出来る

例外処理が終わった後に処理を停止させる場合、例外が原因として処理が止まったことを明示的にするため `return -1;` とすることが一般的

コード量が多いためサンプルコードは割愛(github参照)

名前空間

section15

名前空間

名前空間とは

変数やオブジェクトが存在する空間のこと

→都道府県や地区町村と同じようなイメージ

名前空間の指定はスコープ解決演算子(::)を使用する

主に変数や関数の名前が被っていて名前空間が異なる場合に使う

C++に存在する標準ライブラリはstd(standard)名前空間に存在する

名前空間

名前空間はnamespaceキーワードを用いて指定する

名前空間が異なる場合は変数名や関数名を重複させることが可能

名前空間はブロックではあるが処理が分離されることはなく、名前空間外でも

その空間に存在する変数や関数を使用することが可能

```
namespace <名前空間名>{}
```

名前空間は入れ子にすることが可能

名前空間

名前空間のメンバを呼び出す場合は必ずその名前空間で指定しなければならない

同じ名前空間のメンバを多数用いる場合は毎回名前空間を指定することが面倒

→usingキーワードを使い同じ名前空間の指定を省略できる

名前空間

usingキーワードを用いるとそのブロック内では指定した名前空間のメンバは指定なしで使うことが出来る

```
using namespace <名前空間名>;
```

一般的なプログラムでusing namespace std;を記述するのはstd名前空間内のメンバをよく利用するため

メンバ名が被る可能性があるのでusingの使用は推奨されていない

ストリーム

section16

ストリーム

ストリームとはデータの入出力の流れのこと

C++にはストリームを管理するクラスがstd名前空間に存在する

入力はistreamクラス、出力はostreamクラスが基本クラスで、入出力のソースに応じて派生クラスが存在する

→コンソールへの入手力はiostream

ストリーム

iostream(I/O stream)

iostreamはstd名前空間に1つしか存在しないストリームで、入力はcin

、出力はcoutとして存在している

→I/Oとはコンピュータの標準的な入出力ソースのこと

iostreamを使用するにはヘッダをインクルードする必要がある

```
#include <iostream>
```

ストリーム

ストリームクラスは基本的にはistreamとostreamに分かれて定義されている

istreamを継承したクラスを用いてデータを入力したい場合、抽出演算子(>>)を用いる

ostreamを継承したクラスを用いてデータを出力させたい場合挿入演算子(<<)を用いる

ストリーム

ストリームクラスではエラーが発生したか？や正常か？などの状態が取得できるようになっている

→ストリームクラスを使う場合はこのような状態を確認するようアルゴリズムを組み適切な処理が行われるよう気をつける

状態確認はストリームクラスのメンバ関数として存在している

→メンバ関数をドット演算子(アロー演算子)で呼び出す

ストリーム

- good() : 正常であるか(true or false)
- eof() : ストリームの終端に達したか(true or false)
- fail() : エラーが発生したか(true or false)
- bad() : 回復不可能なエラーが発生したか(true or false)
- operator!() : fail()と同じ
- operator bool() : fail()と同じ(C++11以降)

ストリーム

データの読み込みには抽出演算子の他にいくつかの方法がある

- getline(char* str, size_t max_size, char delim(オプション))関数
  第1引数に読み込んだデータを保存するchar配列、第2引数にchar配列の最大サイズ、第3引数に改行文字(“¥n”)以外で区切る場合の文字
  →最大サイズまたは区切り文字まで読み込まれる
- get(char& c)関数
  引数に渡した変数に一文字代入する

ストリーム

ストリームにはデータがどこまで流れているか(流れたか)が保存されている

→同じデータを読み込んでしまうことは無い

eof()でtrueが返ってきた場合は読み込めるデータが無いためそれ以上データを読み込まないようにする処理が必要になる

状態確認の関数を用いて適切な処理をさせる

コード量が多いためサンプルコードは割愛(github参照)

文字列ストリーム

データの入力元や入力先には標準入出力や外部ファイルがあるが文字列も

データの一種であり、ストリームを用いて管理することが可能

→このとき使うストリームを文字列ストリームと言う

文字列ストリームはstringstreamクラスのヘッダファイルを読み込み使用する

```
#inclide<sstream>
```

ヘッダファイルはsstreamだがクラスはstringstreamであることに注意

文字列ストリーム

stringstreamはistreamとostreamの2つを多重継承しているため挿入/抽出演算子を両方使用することが可能

挿入演算子を使いstringstreamオブジェクトにデータを入力し、抽出演算子を用いて文字列データを読み込む

stringstreamを用いることにより文字列を数値にしたり、逆に数値を文字列に変換することが可能

文字列ストリーム

C++11以降では文字列と数値の相互変換の関数はsstreamヘッダファイルにてstd名前空間に定義されている

→文字列ストリームを用いて変換する必要はない

文字列から数値は変換したい方に合わせてstoi()やstof()、stod()関数があり、引数にstringオブジェクトを取り、それぞれ対応する型の値を返す

数値から文字列はto_string(<int , float, double>)関数を用いる

挿入/抽出演算子のオーバーライド

section17

挿入/抽出演算子のオーバーライド

ストリームと変数では直接データの入出力が可能だがクラスではできない

→これでは不便

そこでクラスに挿入/抽出演算子を使用したに動作する演算子関数を定義することでストリームとクラスで直接データの入出力ができるようにする

→直感的なコードになり使いやすいクラスになる

挿入/抽出演算子のオーバーライド

一般的な演算子関数と同じようにoperator句と演算子を組み合わせ定義

左辺値がストリームクラスで右辺値に演算子関数を定義する関数があるため

フレンド関数とする必要がある

挿入演算子は出力時に使用するためostreamを、抽出演算子は入力時に

使用するためistreamを使用する(ストリームの基本クラスの参照を用いる)

抽出演算子関数の第二引数は値を変更するためconst参照は用いない

挿入/抽出演算子のオーバーライド

演算子関数の定義例

```
class Vector{
public:
    friend istream& operator>>(
        istream& s, Vector& v);
    friend ostream& operator<<(
        ostream& s, const Vector& v);
};
```

演算子関数の実装例

```
istream& operator>>(
istream& s, Vector& v){
    s >> v.x >> v.y;
    return s;
}

ostream& operator<<(
ostream& s, const Vector& v){
    s << v.x << “¥t“ << v.y;
    return s;
}
```

挿入/抽出演算子のオーバーライド

挿入演算子で出力したデータはそのまま抽出演算子で入力できるようにすることが望ましい

→区切り文字を”,”にしたりすると取り除く必要が出てきて困ってしまう

ストリームの基本クラスの参照を引数と返り値にすることで、ストリームの派生クラスでも演算子関数が適用されるようになる

適切な演算子関数を定義することでより良いクラス設計となる

ファイル入出力

section18

ファイル入出力

ファイルの入出力をする際はfstreamのヘッダをインクルードする

#include <fstream>

ファイル入力はifstream、ファイル出力はofstreamを用いる(クラス)

ファイルへ入出力をする前にストリームオブジェクトを作成し、ファイル名を指定してオープン(ファイルを編集できるように)する

ファイル入出力

オブジェクト作成時にコンストラクタにファイル名を文字列で渡すことによりファイルをオープンすることが可能

ファイルストリームオブジェクトにはopen関数があり、引数にファイル名の文字列を渡すことでもファイルをオープンできる

オープンした後はファイルをclose関数を使ってクローズする必要がある

→closeされないことを防ぐためデストラクタで実行される

ファイル入力

ファイルから入力する際はifstreamクラスを用いる

ifstreamはistreamの派生クラス

→抽出演算子を用いてデータを取得する

ファイル内の全てのデータを読み取ったか(ファイルの終端まで読み取ったか)は

メンバ関数eof()を実行して評価する

読み取り方法はストリームと同様なので割愛

ファイル出力

ファイルへ出力する際はofstreamを用いる

ofstreamはostreamの派生クラス

→挿入演算子を用いてデータを入力する

出力ファイル名はopen関数またはコンストラクタで指定したものになる

出力方法はストリームと同様なので割愛

ファイル入出力

ファイル入出力を管理するクラスはストリームクラスを派生したもの

挿入/抽出演算子を適切に用いてデータを操作する

ファイルはopen関数またはコンストラクタで指定する

close関数またはオブジェクトを破棄することによりファイルを閉じる

→閉じないと他のプログラムで使用できないだけでなく最悪の場合データ破損する

静的メンバ(クラス)

section19

静的メンバ

クラスのメンバを静的に(static指定)することで静的メンバを定義可能

基本的にはオブジェクトを生成しないとメンバも生成されないが静的メンバはプログラムの実行開始時に生成される

静的メンバはオブジェクトを生成せずにアクセスすることが可能、また通常のメンバ関数で静的メンバを使用することが出来る

→静的メンバ関数で通常のメンバを使用することはできない

静的メンバ

静的メンバはオブジェクト生成時に作られないためそのクラスのオブジェクト共通のものとなる

→オブジェクトを何個作っても静的メンバは実行時にのみただ1つ作られる

静的メンバはオブジェクトではなくクラスの存在そのものに関連付けられる

オブジェクトが1つも存在していなくてもアクセスできる

→スコープ解決演算子を用いてアクセスする

静的メンバ

静的メンバにアクセスする際はドット演算子を使うことも出来るが、静的メンバにアクセスしていることがコードから把握しづらい

→スコープ解決演算子を用いてアクセスすることが望ましい

＜クラス名＞::静的メンバ

静的メンバはクラスのオブジェクトの数を管理したい場合などに使うと有効

参考コードはgithubを参照

STL

section20

STL

STLとは

標準テンプレートライブラリ(Standard Template Library)のことでテンプレートを活用した標準ライブラリのこと

STLにはよく使うデータ構造やアルゴリズムが用意されている

→自分で作るよりも効率がよく、活用することで安全なプログラムが作成できる

std名前空間に存在する

STL

STLに存在するもの

- コンテナ：オブジェクトを保持するデータ構造
- イテレータ：コンテナの要素にアクセスするもの
- アルゴリズム：コンテナの要素に対して行う操作
- 関数オブジェクト：オブジェクトを関数のように扱う(省略)

コンテナ

コンテナはオブジェクトを保持するためのデータ構造

→いくつものオブジェクトをしまっておけるオブジェクトのようなもの

コンテナにはいくつかの種類がある(vector・deque・list・set…)

→vector以外は説明を省略する

vector

C++では一般的な配列は長さが固定されている(固定長配列)

→要素数を拡大することができない

vectorは可変長配列で長さを固定せずに複数要素を持てる配列のこと

vectorはクラスオブジェクトで要素を管理するためのメンバ関数が用意されている

使う場合はヘッダファイル <vector> をインクルードする

vectorのメンバ関数

- push_back：配列の末尾に要素を追加
- pop_back：配列の末尾の要素を削除
- insert：指定した位置に要素を追加
- erase：指定した位置の要素を削除
- clear：配列を削除する

vectorのメンバ関数

- operator[]：指定した位置の要素を返す(配列参照演算子)
- at：指定した位置の要素を返す
- size：配列のサイズを返す
- empty：配列が空ならtrue、空でなければfalseを返す

vector

vectorに存在するoperator[]関数(配列参照演算子)を用いることにより、一般的な配列のような使い方ができる

vectorを使うことでオブジェクトの管理が簡単になり、より高度なプログラムを作成することが出来る

vectorオブジェクトの宣言方法

```
vector<格納したいオブジェクトの型> <変数名>;
```

イテレータ

イテレータとは一般的な配列に対するポインタのようなものでコンテナの要素にアクセスするために用いる

イテレータには型がありvectorオブジェクトごとに対応する型が存在するイテレータが格納できる変数を宣言でき、ポインタ変数のように扱える

vector<オブジェクトの型>::iterator

イテレータ

コンテナにはイテレータのためのメンバ関数が用意されている

- begin：先頭を指すイテレータを返す
- end：末尾の次を指すイテレータを返す

イテレータに間接参照演算子を使うことでそのイテレータに存在するオブジェクトを取得することが出来る → アロー演算子を使うことも可能

ポインタに対する演算も行える

アルゴリズム

アルゴリズムはコンテナの要素に対して操作を行う関数

- find：指定した範囲内で一致する最初のイテレータ(存在しないならend)
- distance：イテレータ間の距離を求める
- reverse：指定した範囲内の要素の順序を反転する
- sort：指定した範囲内の要素を昇順に並び替える

アルゴリズム(find)

find(

＜最初のイテレータ＞，＜検索を終わりたいイテレータ＞，＜検索する値＞)

返り値はイテレータで、指定した範囲内の要素と指定した値を順に評価し、

trueであればそのイテレータを返す

見つからなかった場合は範囲の末尾のイテレータを返す

アルゴリズム

distance(<イテレータ1>, <イテレータ2>)

返り値はint型で、イテレータ1とイテレータ2の間に存在する要素数を返す

イテレータ同士の引き算でも同様の結果を取ることが出来る

アルゴリズム(reverse)

reverse(<最初のイテレータ>, <最後のイテレータ>)

指定した範囲にある要素の順序を反転させる関数

返り値は無し(void)

アルゴリズム(sort)

sort(<最初のイテレータ>, <最後のイテレータ>)

指定した範囲の要素を昇順に並び替える

→昇順：例)1→10, a→z

返り値は無し(void)

reverseと組み合わせることで降順に並び替えることも可能

STL

STLはその名の通りテンプレートを活用したstd名前空間に存在するオブジェクトのライブラリ

テンプレートを使用するため宣言する際は型を指定する必要がある

STLをうまく活用することはC++でより良い開発をするために必要不可欠

ここで説明したことはSTLのほんの一部で実際にはこれの何倍もの量がある

基本的なサンプルコードはgithubを参照

コマンドライン引数

section21

コマンドライン引数

C++で外部からデータを入力する際の方法として入力ストリームを学んだ

→プログラム実行中にデータを入力する方法

実行時にデータを渡したい場合はコマンドライン引数を用いる

コマンドラインとはコマンドプロンプトやコンソール、ターミナルと呼ばれているもの

のことで、コマンドによって操作を受け付けるもののこと

プログラム実行時にコマンドラインでデータを入力することが出来る

コマンドライン引数

コマンドライン引数を用いる際はmain関数の引数に仮引数を指定する

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

argcには文字列が何個あるか、argv[]には文字列のデータが格納される

main関数実行時にはこれらの変数にはデータが入っている状態なので処理に使うことが可能

コマンドライン引数

コマンドライン引数を使う際は、コマンドラインで入力されるデータの個数や値によって以上を起こさないように気をつけなければならない

→コマンドライン引数が複雑になるときは専用のライブラリを見つけて使ったほうが安全

良いプログラムのために

おまけ

良いプログラムのために

配列のサイズなど、何かのサイズを保存する変数にはsize_t型を用いる

→型を見たときにサイズを表していることが明確

関数を定義する際、可能であればconst参照を用いる

→コピーを引数に取るよりも高速な処理になる

演算子関数をオーバーライドする場合は常識的な処理をさせる

→演算子から想像できる処理でなければならない

良いプログラムのために

プログラムに仕様変更を加える際、修正箇所が少なくなるように工夫をする

→同じ処理を関数化、クラス化するなど…

深いコピーが必要かどうかを見極め、適切にコピーコンストラクタなどを定義する

→デフォルトだと浅いコピーでバグの原因なる

ポインタ変数にポインタを入れない場合はnullptrを用いる

→nullptrをdeleteしても何もされないので安全

良いプログラムのために

STLをうまく活用し効率的な開発をこころがける

→標準ライブラリを使いこなせるかが良いプログラムになるかの鍵

名前空間を適切に使う

→複数人での開発の際、識別子の衝突問題を改善できる

usingを乱用しない

→識別子が衝突してしまう可能性が増える

良いプログラムのために

扱う文字コードに注意する

→日本語などアルファベット以外を使う場合はワイド文字が安全

挿入演算子で出力したデータはそのまま抽出演算子で入力できるようにする

→後で自分が困ることになる

静的メンバをうまく使いこなす

→スマートなコードになることがある

良いプログラムのために

処理全てを自力で実装しようとするのはおすすめしない

→ライブラリを使ったほうが確実で安全

バグを探す際は標準出力をうまく活用する

→どこでバグが起きてるのか分かりやすくなる

問題を把握し、解決法を見つけ、その方法を使いこなす力が必要

→これができないと大規模な開発はできない