[prog2] Programming C++ (C6) Exercise Guide (Ex10)

11/09, Thursday 3rd period.

Ex10 について

- メモリ管理

new, delete演算子 （概要については【Ex06】で済み）

- 低レベルデータ構造

•後に続く「クラス設計」のための前準備

•ライブラリに依存しないプログラミング手法

⇒ STLライブラリがどのように動いているか？

⇒ 抽象的に（汎用性の高い）プログラムを実装する手段は？

Ex11/12はSTLコンテナの
内部構造に関する演習
↓
その前の準備段階

ハードウェア側に寄ったプログラミング

⇒シンボル（変数名）のコーディングから

メモリ空間上のアドレスを意識したコーディングへ

関数ポインタ

- 変数にはアドレスというものがあった（ProgC）
- 関数にもアドレスがある

- ソースコードはコンパイルによって機械語へ
- プログラム実行時に、機械語で書かれたソースコード（の各命令）はメモリのどこかに格納される
- 関数の本体も、メモリのどこかに格納されているはず

関数ポインタの宣言

- 関数ポインタは、「ある引数の型、ある戻り値の型」の関数のアドレスを保持することができる

//引数void, 戻り値 void型の関数ポインタ func

- void (*func)(void);

//引数string, int, 戻り値 int型の関数ポインタ p

- int (*p)(string, int);

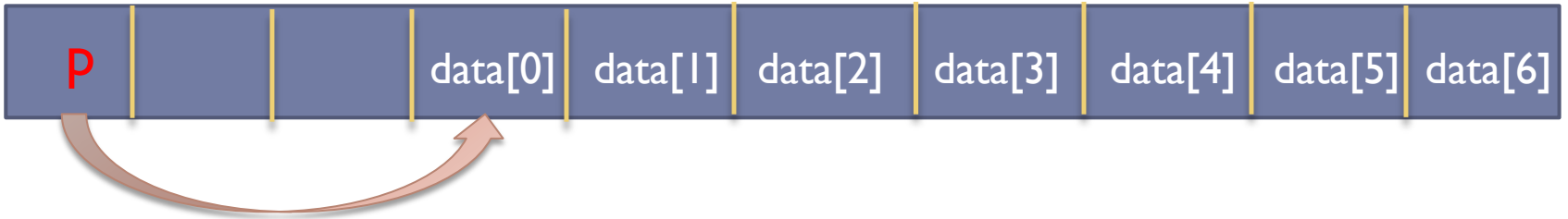
//引数 vector<int>, 戻り値Student_info型の関数ポインタq

- Student_info (*q)(vector<int>);

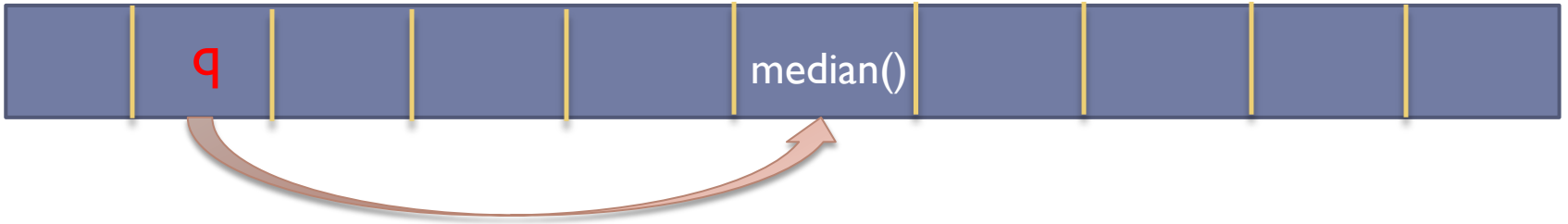
- 関数ポインタの宣言は、通常の関数呼び出しと紛らわしいので、(*p) のように変数名を ( ) でくくることを推奨する

関数ポインタのイメージ

```
int *p;
int data[10];
p = data;     cout << *p ;  // p が指す値（int型整数 data[0]）を出力
```




```
double median(vector<int>);
double (*q)(vector<int>);
q = median;
(*q)();    // qが指す値（関数double median(vector<int>)）を実行する
```



関数ポインタの利用（例題１）

- 引数、戻り値の型が共にvoid型の関数ポインタ p

p = hello; で 関数hello()のアドレスを渡して (*p)();とすると、関数hello() が実行される

p = glass; は、glassの戻り値の型が異なる為、p にアドレスを保持できずエラーになる

⇒ char *q(void); なら保持できる

ex10-test.cc

```
1  #include <iostream>
2
3  using namespace std;
4
5  void hello(void);
6  void morning(void);
7  void thank(void);
8  void count10(void);
9  char glass(void);
10
11 int main(){
12   void (*p)(void); // 戻り値void型、引数void型の関数のポインタ p
13
14   //関数helloのアドレスを代入（※渡す関数を変えて試してみること）
15   p = hello;
16
17   (*p)(); //pが指す関数を実行する
18
19   return 0;
20 }
21
22
23 void hello(void){
24   cout << "Hello, World!" << endl;
25 }
26 void morning(void){
27   cout << "Good morning." << endl;
28 }
29 void thank(void){
30   cout << "Thank you!" << endl;
31 }
32 void count10(void){
33   for(int i=1; i<= 10; i++){
34   cout << i << endl;
35   }
36 }
37 char glass(void){//型が違うので、コンパイルが通らないはず
38   return 'w';
39 }
```

関数ポインタの配列（問題１）

- 関数ポインタ　⇒　void (*func)(void); のような形
- 関数ポインタの配列　⇒ void (**func)(void);　のように書ける

- 問題1では、関数ポインタ配列の宣言時に（定数）初期化

2つの引数が共にint型、戻り値がint型の関数たちの
アドレスをまとめて保持する

- int (*functions[])(int, int) = { /* ここを埋める */ };

↑初期化するだけなら、関数名を
順に並べて書けば良い

functions[0](a,b);
functions[1](a,b);　…　の形で関数呼び出しができる
⇒　添え字の部分を変数にして、条件に応じて値を変えれば
同じような関数たちの呼び出しをひとまとめに統一できる

main関数の引数（C言語と同じ）

※argv は
「ポインタ配列」。
argv自体はアドレスしか
持っておらず、
実体は別の領域にある

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

プログラムに渡された
引数の個数
(実行ファイル自身も含む)

argc 個の プログラムに
渡された引数の中身が
順に入っている

※ただし、引数は全て
文字列リテラルとして扱う

- 例

$./a.out 10 apple 6 -5 book

argc = 6

argv[0]= "./a.out" , argv[1]= "10" , argv[2]= "apple"

argv[3]= "6" , argv[4]= "-5" , argv[5]= "book"

ファイルストリームの利用(要 <fstream>)

■ファイル入出力(両方対応)
fstream という型で宣言する(2段階式)

```
fstream file;
file.open("input.txt", ios::in);
```

Cのfopenにかなり近い書き方ができる。

```
FILE *fp;   fp=fopen("input.txt","r");
```

ios::in　読み込みモード
ios::out　書き込みモード
…等、fopenにもある多様なモードが使える

ファイルストリームの利用(要 <fstream>)

■ファイル入力(ファイルオープン)　ifstream ■

ifstream 変数名(ハンドル) ([ファイル名])

例) ifstream infile("input.txt");

・Cでいえば、FILE *fp;　fp=fopen("input.txt", "r"); に対応

fstream系も、クラス構造をしている。

デフォルトコンストラクタや引数1個、2個のコンストラクタがあり、上記の例は引数1個(開くファイル名)を与えている。

ファイルストリームの利用(要 <fstream>)

■ファイル出力(ファイルオープン)　ofstream ■

ofstream 変数名(ハンドル) ([ファイル名])

例) ofstream outfile("output.txt");

・Cでいえば、FILE *fp;　fp=fopen("output.txt","w"); に対応

ファイルが消えてしまうのは困る(続けて追記したい)場合

ofstream outfile("output.txt" , ios::app);

第2引数にモード指定をするコンストラクタを使えば良い

ファイルストリームの利用(要 <fstream>)

ファイルストリームを通した読み込み、書き込みの方法
⇒ cin や cout をファイルストリームに差し替えるだけ

```
ifstream infile("input");
while(infile >> data){        //  cin の代わりに infile
…
}

ofstream outfile("out");
for(int i=0; i<datasize; i++){
outfile <<  data;             //  cout の代わりに outfile
}
```

ファイルストリームの利用(要 <fstream>)

■ファイルクローズ（要らなくなったらファイルを解放）

・infile という変数名でファイルを開いているなら

⇒ infile.close()

■ファイルオープン成功可否の判定

```
if( !infile.is_open() ){ // ファイルオープン成功でないならば
cerr << “File open failed.” << endl;
return -1;
}
```

標準エラーストリームへの出力
（C言語でいえば、stderr ）
cout とは 上手に使い分けよう

・ if(!infile){…} でも動作する模様

new, delete 演算子（Ex06より再来）

- C++におけるメモリ領域の動的確保＆解放
- malloc, free との違いは 実行時についでに

コンストラクタ，デストラクタもコールする

■1次元のint型配列の動的確保と解放

```
int *data;
data = new int[datasize];
delete []data;
```

■ 2次元配列確保の場合も、アルゴリズムはmallocのときと同じ（解放は逆順）

```
int **data;
data = new int*[ny];            //まず1次元分のポインタを確保
for(i=0;i<ny;i++) data[i] = new int[nx];    //確保した各ポインタに対して
```