もっとC++

C++の機能を使う

C++には楽しい機能がたくさんある

- 可読性の向上
- 短く書ける(ことがある)
- 書いている俺かっこいい(プログラミングのモチベUPに繋がる．意外に大事)

目次

参照

ポインタの機能制限版だけど使いやすい

ふつうのやつ

vectorの出力関数を作りたい

```
void showVec(vector<int> v) {
  for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
  }
}
int main()
{
  vector<int> v = {1, 2, 3, 4};
  showVec(v);
  return 0;
}
```

問題点: 引数に渡す度にvの値がコピーされるのでメモリに優しくない

ポインタ使う

C言語っぽくポインタを使う

```
void showVec(vector<int> *v) {
  for (int i = 0; i < v->size(); i++) {
    cout << (*v)[i] << endl;
  }
}
int main()
{
  vector<int> v = {1, 2, 3, 4};
  showVec(&v);
  return 0;
}
```

- *とか&を書くのが面倒だし間違えがち.

参照を使う

- 「再代入できないポインタ」という理解でOK

```
void showVec(vector<int>& v) {
  for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
  }
}
int main()
{
  vector<int> v = {1, 2, 3, 4};
  showVec(v);
  return 0;
}
```

- 内部的にはポインタ
- ふつうの変数っぽく書けてGood

もっと詳しく1/2

```
int a = 10;
int *x = &a; // aのアドレスを持っている
int& y = a; // 上に同じだがよりすっきりと書ける
```

```
cout << *x << endl; // xが指すアドレスの値を出力
cout << y << endl; // 上に同じだがよりすっきりと書ける
```

もっと詳しく2/2

```
int a = 10;
int *x = &a; // aのアドレスを持っている
int& y = a; // 上に同じだがよりすっきりと書ける
```

```
int b = 8;
x = &b; // bのアドレスを持っている
y = b; // 再代入はできないのでエラー
```

こうして

「別関数に処理を任せる」という目的でポインタを使う必要は無くなったのであった
(完)

いやでもそれがポインタの全てか?
⇒ 動的確保の用途がまだ残ってる!

再代入したいですか?

- 例えば隣接リストを実装したいとき
- ふつうのポインタを使えば良い?

思い出して欲しいmalloc

- 確保したら解放，面倒くさくないですか?
- 実は現代のC++でふつうのポインタは非推奨
  - **スマートポインタ(smart pointer)** という新概念がある
  - 確保したメモリを解放してくれる頭のいいポインタ
  - ふつうのポインタは「生ポインタ(raw pointer)」と呼ばれている

このあたりの話は面白いんだけど詳しくはググってください

演算子オーバーロード

オーバーロードとは

- 同名の関数を複数定義すること
- 引数の個数と型が異なればコンパイラ君が区別できる

```
int calc(int x, int y) {
  return x + y;
}
int calc(double x, double y) {
  return x + y
}
int calc(int x) {
  return 2*x;
}
```

演算子もオーバーロードできる

つまりオリジナルの構造体に演算子が定義できる

座標があります

```
struct Point {
  int x, y;
  Point(int x, int y): x(x), y(y) { }
};
```

補足: コンストラクタ

宣言時に自動で呼ばれる関数

```
クラス名(引数): メンバ(引数), ... {
   何か処理
}
```

足し算と引き算を定義してみる

```
Point operator+(const Point &a, const Point &b) {
  return Point(a.x + b.x, a.y + b.y);
}
Point operator-(const Point &a, const Point &b) {
  return Point(a.x - b.x, a.y - b.y);
}

int main()
{
  Point p1(1, 2), p2(3, 4);
  Point p3(p1 + p2);
  cout << p3.x << ' ' << p3.y << endl;
  return 0;
}
```

引数にconst付けなくても動くが，つけたほうが行儀がよい

見慣れない(?)記法

```
Point p3(p1 + p2)
// Point p3(Point型の変数)と宣言している
```

こんなコンストラクタ書いてない!

コピーコンストラクタ

コンストラクタもオーバーロードできて，その一種

```
struct Point {
  ...
  Point(const &obj) { ...  }
}
```

これを書かないと勝手に「メンバを全部コピー」というコピーコンストラクタを作ってくれる

インクリメントとかもオーバーロードできる

```
struct Point{
  ...
  Point& operator++() {
    this->x++;
    this->y++;
    return *this;
  }
}

int main()
{
  Point p(1, 2);
  ++p;
  cout << p.x << ' ' << p.y << endl;
}
```

補足:this

自分自身の情報を持つ．ただしポインタ．

比較演算子も書こう

```
bool operator==(const Point &a, const Point &b) {
  return a.x == b.x && a.y == b.y;
}
bool operator!=(const Point &a, const Point &b) {
  return !(a == b);
}
bool operator<(const Point &a, const Point &b) {
  if (a.x != b.x) return a.x < b.x;
  else return a.y < b.y;
}
...
```

自分のクラスをcoutしたい

目的

こうしたいときあるよね

```
int main()
{
  Point p(1, 3);
  cout << p << endl; // (1, 3)とか表示したい
  return 0;
}
```

そもそも何の演算子

```
cout << p;
```

- <<はシフト演算子
- coutとpとの二項演算をしている
- cout: C++で最初から用意されたostream型の変数みたいなもの

⇒ ostream型とPoint型の演算子<<を定義すればよさそう

解決

ostreamを宣言します

std::は省いているけど許してほしい

```
struct Point {
  int x;
  int y;
  Point(int x, int y): x(x), y(y) { }
};
ostream& operator<<(ostream& stream, const Point& value) {
  stream << "(" << value.x << ", " << value.y << ")";
  return stream;
}
```

自分だけの演算子を作ろう!
でもやり過ぎは禁物

関数型プログラミング

関数型プログラミング is 何

- 問題を関数の組み合わせで解くプログラミングスタイル
    - ここでの関数は「副作用の無い数学的な関数」を指す.

副作用

- 副作用が無い: どんな場合でも入力が同じなら出力はただ1つに決まること

```
int add (int a, int b) {
  return a + b;
}
```

副作用はこうしておこる:

```
int g = 2;
void add(int a, int b) {
  return a + b + g;
}
```

- 一般に副作用があると，バグの発見が困難になりがち
- でも競プロではよく書く
    - 競プロで書くようなコードはそこまで大規模では無い

C++での関数型プログラミング

- 他の「関数型言語」と呼ばれる部類(Haskellとか)に比べるとできることは限られている.
- JavaScriptやろう(夏にやる)

ラムダ式

ラムダ式 is 何

変な名前だが「名前の無い関数」のこと.

- 小規模な関数を変数として取っておきたい
- 関数に関数を引数として渡したい

などの用途で使われる.

ことはじめ

```
double add(int a, int b)
{
  return a + b;
}
int main()
{
  cout << add(1, 2) << endl;
  return 0;
}
```

ラムダ式を使う

関数もオブジェクトとして扱える!

```
int main()
{
  auto add = [](int a, int b)->double {
    return a + b;
  };
  cout << add(1, 2) << endl;
  return 0;
}
```

文法

とりあえずこれを覚えておけばOK

```
[キャプチャリスト](引数) -> 戻り値の型 { 関数の本体 }
```

戻り値の型が明らか(コンパイラが推測できる)なら省略可

```
[キャプチャリスト](引数) { 関数の本体 }
```

キャプチャリスト is 何 (1)

外部変数を使うための指定を書く場所

```
vector<int> v = {1, 2, 3, 4, 5};
auto showV = []() {
  for (auto e : v) cout << e << ' ';
  cout << endl;
};
```

- vを外部で宣言している．このままだとエラー
  - showVにとっては外部変数が見えない

キャプチャリスト is 何 (2)

```
vector<int> v = {1, 2, 3, 4, 5};
auto showV = [=]() {
  for (auto e : v) cout << e << ' ';
  cout << endl;
};
```

=をつけると「外にある変数を**コピー**して利用」になる

キャプチャリスト is 何 (3)

```
vector<int> v = {1, 2, 3, 4, 5};
auto showV = [&]() {
  for (auto e : v) cout << e << ' ';
  cout << endl;
};
```

&をつけると「外にある変数を**参照**して利用」になる

もっと詳しい文法があるがググって.

ラムダ式応用: ソートに関数を指定

点があります

```
struct Point {
  int x;
  int y;
  Point (int x, int y): x(x), y(y) { }
}
```

ラムダ式応用: ソートに関数を指定

Pointのvectorを作ってソートしたい
「y座標についてソート」にしたい

```
int main()
{
  vector<Point> v;
  v.push_back(Point(1, 2));
  v.push_back(Point(3, 3));
  v.push_back(Point(2, -1));
  sort(v.begin(), v.end(), [](const Point &p, const Point &q) {
    return p.y < q.y;
  });
}
```

sortの3つめの引数に比較関数を入れる

ラムダ式応用: ソートに関数を指定

「2つの要素a,bを比較してa<bであればtrueを，そうでなければfalseを返す関数」を引数にとる.

```
sort(v.begin(), v.end(), [](const Point &p, const Point &q) {
  return p.y < q.y;
});
```

もちろんこうやると降順ソートになる

```
sort(v.begin(), v.end(), [](const Point &p, const Point &q) {
  return p.y > q.y;
});
```

ラムダ式応用: accumulate

accumulateでvectorの全要素の総和が簡潔に書ける.

```
#include <numeric>
...
vector<int> v = {1, 2, 3, 4, 5};
int sum = accumulate(v.begin(), v.end(), 0);
cout << sum << endl;
```

動き: acc = 0から始めて，v[0], v[1], ...,v[4]と足していく

```
acc = 0;
acc = acc + v[0]; -> acc == 1
acc = acc + v[1]; -> acc == 3
acc = acc + v[2]; -> acc == 6
acc = acc + v[3]; -> acc == 10
acc = acc + v[4]; -> acc == 15
```

ラムダ式応用: accumulate

- accumulateは「蓄積する」という意味.
- なので総和だけがaccumulateだけじゃない
- ラムダ式を指定すると総和以外も書ける

```
vector<int> v = {1, 2, 3, 4, 5};
int pro = accumulate(v.begin(), v.end(), 1, [](int acc, int e) {
  return acc * e;
});
cout << pro << endl;
```

ラムダ式応用: accumulate

分散も簡単に書ける

```
vector<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};
double ave = accumulate(v.begin(), v.end(), 0)/v.size();
double var = accumulate(v.begin(), v.end(), 0, [&](double acc, double e) {
  return acc + (e - ave)*(e - ave);
});
```

もっとラムダを使いたい?

次スライド「もっとSTL」へ⇒.