C++プログラミングI

- 第9回：構造体 (struct)
- 担当：二瓶芙巳雄

構造体とは

- ユーザ定義型：ユーザが作成できるデータ型
- 任意の型の要素をまとめられる
  - 配列は同じ型の要素を並べたもの
- 要素ごとに名前をつけ、その名前で要素を指定
  - 配列は自動でつけられた番号で要素を指定
- `struct`：構造体を定義するときの予約語
- `MyType` は定義した構造体名
  - ※構造体名は何でもOK．その構造体の用途をよく表すものを名付けましょう．
  - 構造体名は単語の1文字目を大文字にする習慣がある
- 構造体の要素を**データメンバ**と呼ぶ

```cpp
#include <iostream>

// 構造体定義の例
struct MyType {
  int    x;
  double y;
  bool   z;
};

int main() {
  // MyType 型の変数 a, b を宣言
  MyType a;
  MyType b;

  // a が持つデータメンバ x にアクセス
  a.x = 10;
  std::cout << a.x << "";
}
```

構造体定義の注意

- 閉じ波括弧（`}`）後のセミコロン（`;`）は必須

```
1  struct Point {
2    int    x;
3    double y;
4  }; // <- セミコロン必須！
```

- 定義と同時に変数宣言ができるため

```
1  struct T {
2    int    x;
3    double y;
4  } a;
```

- `struct T {…} a;`
- 赤い部分を型名と考えれば、`int x;`と同じ形式

構造体変数の基本

- データメンバのアクセスには **ドット演算子** `.` を使う
- 構造体変数は代入が可能
- `cin` / `cout` でデータメンバを入出力する場合は，個別メンバの指定が必要

```
% ./a.out
John, 36
Jane, 35
```

```
1  struct Staff {
2    std::string name;
3    int age;
4  };
5
6  int main() {
7    Staff x, y;
8    x.name = "John";
9    x.age = 36;
10
11   y = x;  // 変数の代入
12   std::cout << y.name <<  ", "
13             << y.age <<  "\n";
14
15   y = {"Jane", 35}; // 値リストの代入
16   std::cout << y.name <<  ", "
17             << y.age <<"\n";
18 }
```

構造体の配列

- 配列は同じ型の要素の並び
  - 構造体を並べても良い
- 配列内のメンバへのアクセス
  1. 先に添字で配列の要素を指定（`list[i]`）
  2. ドット演算子で構造体のメンバを指定（`list[i].name`）

```
% ./a.out
i==0: John, 36
i==1: Jane, 35
i==2: Joe, 24
```

```cpp
struct Staff { std::string name; int age; };

int main() {
  Staff x, y;
  x.name = "John";
  x.age = 36;
  y = {"Jane", 35}; // 値リストの代入

  std::vector<Staff> list(3);
  list[0] = x;
  list[1] = y;
  list[2] = {"Joe", 24};

  for (int i = 0; i < list.size(); i++) {
    std::cout << "i==" << i << ": "
              << list[i].name << ", "
              << list[i].age << "\n";
  }
}
```

構造体変数の初期化

- 変数宣言と同時に初期化指定が可能
- 波括弧内をカンマで区切って要素を指定
  - 配列の初期化と類似
- 範囲 `for` 文の変数(`e`)をリファレンス `&` で指定している点に注意
  - ※ このプログラムにおける `auto` は `Employee`

```
1  struct Employee{
2    string name;
3    int age, salary;
4  };
5
6  int main() {
7    Employee x {"John", 38, 300};
8    Employee y {x};
9
10   vector<Employee> d {
11     {"Robert", 23, 220},
12     {"David",  17, 180}
13   };
14   vector<Employee> s {x, y};
15
16   for (const auto& e : s)
17     std::cout << e.name <<": "<< e.salary<<"\n";
18 }
```

データメンバのデフォルト値

- 組み込み型の局所変数の初期値に注意
- 初期値を指定しないと値は不定
- 構造体変数も同じ扱い
- 定義時にデフォルト値を指定できる

```
1  struct Type1 {
2    string name {"none"};
3    int x{10};
4    int y{};
5  };
6
7  int main() {
8    Type1 a;
9    std::cout << a.name << ": "
10             << a.x << " "
11             << a.y << "\n";
12 }
```

```
% ./a.out
none: 10 0
```

構造体や配列を要素に持つ構造体

- 構造体データメンバの型は任意
  - 基本データ型、他の構造体、配列
- 初期化や要素指定も組み合わせで行う
  - 配列や構造体をメンバに持つならば `{{...}}` の形
  - メンバ指定も `.` と `[]` が並ぶ

```
% ./a.out
a c
```

```
1  struct Point   {
2    string nm;
3    int x{}, y{};
4  };
5  struct Triangle{
6    string nm;
7    Point A, B, C;
8  };
9  struct Hexagon {
10   string nm;
11   std::vector<Point> pt;
12 };
13
14 int main() {
15   Triangle t { "T", {"a",0,0}, {"b",3,0}, {"c",2,2} };
16   Hexagon h {
17     "H1",
18     {
19       {"a",0,0}, {"b",3,0}, {"c",4,2},
20       {"d",4,6}, {"e",3,6}, {"f",0,4}
21     }
22   };
23   std::cout << t.A.nm <<" "<< h.pt[2].nm <<"\n";
24 }
```

構造体の役割：コードの冗長さを解決

- ポケモンごとに「名前，体力，攻撃力，防御力」をプログラム上で表現したい

```cpp
int main(){
  std::string pikachu_name{ "pikataro" };
  int pikachu_hp {35};
  int pikachu_atk {55};
  int pikachu_def {40};

  std::string evee_name{ "evejiro" };
  int evee_hp {55};
  int evee_atk {55};
  int evee_def {50};

  std::cout << "pikachu (hp): " << pikachu_hp << "\n";
  std::cout << "evee (hp): " << evee_hp << "\n";
  return 0;
}
```

- ポケモンの数×4（`name`，`hp`，`atk`，`def`）の変数
  - 将来的に管理できなくなることは明らか
- 構造体を使用して書き改めることが可能

```cpp
struct Pokemon {
  std::string name;
  int hp, atk, def;
};

int main(){
  Pokemon pikachu { "pikataro", 35, 55, 40 };
  Pokemon evee { "evejiro", 55, 55, 50 };

  std::cout << "pikachu (hp): " << pikachu.hp << "\n";
  std::cout << "evee (hp): " << evee.hp << "\n";
  return 0;
}
```

- 構造体として `Pokemon` を自作
- `pikachu.hp` で，構造体 `Pokemon` の変数 `pikachu` の `hp` を参照

構造体の配列

- ポケモンは複数匹手持ちにできる
  - このような事象を表現するのに配列は最適

```
% ./a.out
lead pokemon: pikataro
0: pikataro
1: evejiro
2: mewsaburo
```

```cpp
struct Pokemon {
  std::string name;
  int hp, atk, def;
};

int main(){
  Pokemon pikachu { "pikataro", 35, 55, 40 };
  Pokemon evee { "evejiro", 55, 55, 50 };
  Pokemon mew { "mewsaburo", 100, 100, 100 };

  std::vector<Pokemon> team {
    pikachu, evee, mew
  };

  std::cout << "lead pokemon: " team[0].name << "\n";

  for( int i = 0; i < team.size(); i++ ) {
    std::cout << i << ": " << team[i].name << "\n";
  }

  // 範囲for文も使用可能
  // for( const Pokemon& poke : team ) {
  //   std::cout << poke.name << "\n";
  // }
  return 0;
}
```

構造体を有する構造体

- ポケモンは「わざ（Move)」を覚えられる
  - 「わざ」はタイプとダメージ量で定義される

```
% ./a.out
thunder: Thunderbolt 90
pika move: Thunderbolt 90
pikataro Thunderbolt
evejiro Tackle
```

```cpp
struct Move {
  std::string name;
  int damage;
};

struct Pokemon {
  std::string name;
  int hp, atk, def;
  Move move;
};

int main(){
  Move thunder { "Thunderbolt", 90 };
  Pokemon pikachu { "pikataro", 35, 55, 40, thunder };
  Pokemon evee { "evejiro", 55, 55, 50, { "Tackle", 40 } };

  std::cout << "thunder: " << thunder.name << " "
                           << thunder.damage << "\n";
  std::cout << "pika move: " << pikachu.move.name << " "
                             << pikachu.move.damage << "\n";

  std::vector<Pokemon> team { pikachu, evee };
  for( int i = 0; i < team.size(); i++ )
    std::cout << team[i].name << " " << team[i].move.name << "\n";
}
```

構造体の配列を有する構造体

- ポケモンは「わざ（Move)」を**複数**覚えられる

```
% ./a.out
name: pikataro
Thunderbolt 90
Tackle 40
name: evejiro
Tackle 40
Bite 60
```

```cpp
struct Move {
  std::string name;
  int damage;
};

struct Pokemon {
  std::string name;
  int hp, atk, def;
  std::vector<Move> moves;
};

int main(){
  Move thunder { "Thunderbolt", 90 };
  Move tackle { "Tackle", 40 };
  Pokemon pikachu { "pikataro", 35, 55, 40, { thunder, tackle } };
  Pokemon evee { "evejiro", 55, 55, 50, { tackle, { "Bite", 60 } } };

  std::vector<Pokemon> team { pikachu, evee };
  for( int i = 0; i < team.size(); i++ ) {
    std::cout << "name: " << team[i].name << "\n";

    for( int j = 0; j < team[i].moves.size(); j++ ) {
      std::cout << team[i].moves[j].name << " "
                << team[i].moves[j].damage << "\n";
    }
  }
}
```

値渡し引数と戻り値

- 構造体の関数への引数は**値渡し**（実引数の値が仮引数にコピーされる）
- 戻り値も全要素のコピーとなる
- 関数の戻り値は一つだけ．複数の値をまとめた構造体を戻り値にすれば，複数の値を返却可能

```
% ./a.out
input x, y: 3 5
Point: (3, 5)
Point: (10, 20)
```

```
1  struct Point { int x{}, y{}; };
2
3  Point input() {
4    std::cout << "input x, y: ";
5    Point p;
6    std::cin >> p.x >> p.y;
7    return p;
8  }
9
10 void output(Point g) {
11   std::cout << "Point: ("
12             << g.x <<", "<< g.y <<")\n";
13 }
14
15 int main() {
16   Point a { input() };
17   output( a );
18   output( {10, 20} );
19 }
```

良くない例

- 10001個の要素をもつ構造体を100000回、足している。→ 10001個のデータコピーが100000回発生

```
1  struct Data { string n; vector<double> d; };
2
3  double sum(Data x) {
4    double s {};
5    for (auto e: x.d)
6      s += e;
7    return s;
8  }
9
10 int main() {
11   Data a { "abc", vector<double>(10000,1.5) };
12   double x {};
13   for (int i = 0; i < 100000; i++)
14     x += sum(a);
15   cout << x <<"\n";
16 }
```

構造体の
リファレンス引数

- リファレンス引数の受け渡し時のオーバヘッドは小さい
- 実引数の更新あり：`void f(T& x) {...}`
- 実引数の更新なし：`void f(const T& x) {...}`

```
% ./a.out
X,3,5
```

```
struct Point {
  string name;
  int x{}, y{};
};

void update(Point& p) {
  p.x += 2;
  p.y += 3;
}

void print(const Point& p) {
  cout << p.name << "," << p.x << "," << p.y << "\n";
}

int main() {
  Point a = {"X", 1, 2};
  update( a );
  print( a );
}
```

メンバの取り出し

- 構造化バインディング(c++17)

```
% ./a.out
John 10
John 10
Johnson 10
```

```cpp
struct Staff {
  string first_name;
  int total_year;
};

int main() {
  // ドット演算子 . でアクセスする場合
  Staff john {"John", 10};
  std::cout << john.first_name <<" "
            << john.total_year <<"\n";

  // 構造化バインディングによるデータメンバの値のコピー
  auto [n, y] {john};
  std::cout << n <<" "<< y <<"\n";

  // 構造化バインディングによるデータメンバのリファレンス
  auto& [nm, ty] {john};
  nm += "son";
  std::cout << john.first_name <<" "
            << john.total_year <<"\n";
}
```

メンバ取り出しの応用

- 構造体を返す関数 → 複数の値を返すことが可能
- 範囲 `for` 文をシンプルに表現

```
% ./a.out
input: alice 2
staff: alice 2
(John, 5)
(Jane, 8)
(James, 3)
```

```cpp
struct Staff {
  string first_name;
  int total_year;
};

Staff get() {
  std::cout << "input: ";
  string n; int y;
  std::cin >> n >> y;
  return {n, y};
}

int main() {
  auto [a, b] { get() }; // 関数の結果
  std::cout << "staff: " << a <<" "<< b <<"\n";

  std::vector<Staff> list {
    {"John",5},
    {"Jane",8},
    {"James",3}
  };
  for (auto [n, y] : list)
    std::cout <<"("<< n <<", "<< y <<")\n";
}
```

std::pair

- 任意の二つのデータを組み合わせる
- `<utility>` ヘッダファイルをインクルード
- `std::pair` は名前空間 `std` に定義済み
  - 構造体の定義は不要
  - `first` と `second` がデータメンバ
  - 構造化バインディングの使用で簡潔に記述
- ※ `std::pair` はデータメンバを2つしか持てない構造体と言い換えられる
  - 二つのデータメンバを有するだけの構造体の定義をさぼりたいときに使う

```cpp
#include <utility>

int main() {
  std::pair<int,double> p1 {10, 1.5};
  std::cout << p1.first <<" "
            << p1.second <<"\n";

  std::pair p2 {20, 2.5};
  auto [i, d] {p2};
  std::cout << i <<" "<< d <<"\n";
}
```

```
% ./a.out
10 1.5
20 2.5
```

`std::pair`の使用例

- `std::pair`を返す関数
  - 成功の有無
  - 成功時の結果

```
% ./a.out
10 20
(10, 20)
% ./a.out
hello c++
error!
```

```cpp
struct Point {
  int x{}, y{};
};

auto input() {
  Point p;
  // 入力に成功していれば true, 失敗していれば false
  bool f {cin >> p.x >> p.y};

  // first: 入力の成功 or 失敗
  // second: 成功していた場合のPoint
  return std::pair{ f, p };
}

int main() {
  if (auto [f, p] { input() }; !f) cout << "error\n";
  else cout << "(" << p.x <<", "<< p.y <<")\n";

  // ↑ の二行は ↓ と等価
  // auto [f, p] { input() };
  // if (!f) cout << "error\n";
  // else cout << "(" << p.x <<", "<< p.y <<")\n";
}
```

構造体の `cin` / `cout` 対応

- 構造体変数をそのまま `cout` / `cin` できたら便利 → 特別な関数を作って設定する
  - 第一引数の型：入力の場合は `istream&`，出力の場合は `ostream&`
  - 第二引数の型：対応させたい構造体
  - 戻り値の型：第一引数の型．`return` には第一引数の変数を指定．
  - ※ パターンなので覚えましょう
- C++では既存の演算子の機能を拡張できる

```
% ./a.out
(Alice, 22)
Bob 31
(Bob, 31)
```

```cpp
struct Staff {
  string name;
  int age{};
};

// 構造体を std::cin などに対応させるための関数定義
// 第二引数はconstをつけない
std::istream& operator>>(std::istream& in, Staff& s) {
  return in >> s.name >> s.age;
}

// 構造体を std::cout などに対応させるための関数定義
// 第二引数はconstをつけるとよい
std::ostream& operator<<(std::ostream& out, const Staff& s) {
  return out << "(" << s.name << ", " << s.age << ")";
}

int main() {
  Staff staff { "Alice", 22 };
  std::cout << staff << "\n";

  std::cin >> staff;
  std::cout << staff << "\n";
}
```

補足：`operator>>` と `operator<<` の戻り値について

- `std::ostream` の変数は，`<<` 演算子によって内部の状態が変わる
  - 例）`std::cout << "hello"` と書くと，`std::cout` の内部状態が更新される
- `operator<<` の関数では，状態が変わった後の `std::ostream` の変数を返却すればOK

```
1  // ①以下のような表現でもよいし，
2  std::ostream& operator<<(std::ostream& out, const Staff& s) {
3    return out << "(" << s.name << ", " << s.age << ")";
4  }
5
6  // ②以下のような表現でもよい
7  std::ostream& operator<<(std::ostream& out, const Staff& s) {
8    out << "(" << s.name;
9    out << ", " << s.age;
10   out << ")";
11   return out;
12 }
```

構造体変数の比較演算子

- 構造体変数はそのままでは比較できない
- 比較演算子を関数により設定する
  - 二つのconstリファレンス引数
  - `bool` を戻り値

```
% ./a.out
NOT equal.
p is greater than q.
```

```cpp
struct Point { int x{}, y{}; };

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.x<b.x || (a.x==b.x && a.y<b.y);
}

int main() {
  Point p{0, 0}, q{1, 0};
  if( p == q ) std::cout << "equal.\n";
  else std::cout << "NOT equal.\n";

  if( p < q ) std::cout << "p is greater than q.\n";
  else std::cout << "p is NOT greater than q.\n";
}
```

比較演算子の特徴

- 比較演算子は6種類が揃っている方が良い
- 小なり(`<`)だけで他の演算も表せる
- 演算子を設定する場合もこれに従うべき
- `c++20`では自動生成する機能が追加された

演算子	代替の計算
`a==b`	`!(a<b) && !(b<a)`
`a!=b`	`(a<b) || (b<a)`
`a<=b`	`!(b<a)`
`a>b`	`b<a`
`a>=b`	`!(a<b)`

比較演算子の実装の具体例

- まずは `operator<` を定義する．その後は比較演算子 `<` を使用して，ほかの比較演算子も実装する．
  - ※ `operator<` には構造体の変数の比較のための条件を書くが，それ以外はコピペで作ってよい
- `==` と `!=` は演算が複雑なので `<` 演算子とは別に作成してもよいかもしれない

```cpp
struct Student { std::string name; int id; };

bool operator<(const Student& a, const Student& b) { return a.id < b.id; }
bool operator>(const Student& a, const Student& b) { return b < a; }
bool operator==(const Student& a, const Student& b) { return !(a<b) && !(b<a); }

int main() {
  Student x {"x", 0}, y {"y", 1};

  cout << std::boolalpha << (x < y) << "\n"; // true
  cout << std::boolalpha << (x > y) << "\n"; // false
  cout << std::boolalpha << (x == y) << "\n"; // false
}
```

演算子	代替の計算
`a==b`	`!(a<b) && !(b<a)`
`a!=b`	`(a<b) || (b<a)`
`a<=b`	`!(b<a)`
`a>b`	`b<a`
`a>=b`	`!(a<b)`