C++プログラミングI

第9回 構造体

成蹊大学理工学部

構造体とは

- ユーザ定義型：後から追加できるデータ型
- 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 MyType { // 構造体定義の例
  int    x;
  double y;
  bool   z;
};
```

- `struct` は予約語, `MyType` は定義した型名
- 構造体の要素をデータメンバと呼ぶ
- 構造体名は単語の 1 文字目を大文字にする習慣がある

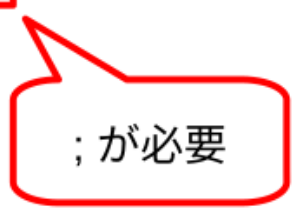
構造体定義の注意

- 閉じ波括弧（'}'）後のセミコロン（';'）は必須
- 定義と同時に変数宣言ができるため

```
struct T {
  int    x;
  double y;
} a ;
```

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

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



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 には個別メンバの指定が必要

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

int main() {
  Staff x, y;
  x.name = "John";
  x.age = 36;

  y = x;  // 変数の代入
  std::cout << y.name <<", "<< y.age <<"\n"

  y = {"Jane", 35}; // 値リストの代入
  std::cout << y.name <<", "<< y.age <<"\n"
}
```

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し (`list[i]`)、
  2. ドット演算子で構造体のメンバを指定 (`list[i].name`)

```
std::vector<Staff> list(3);
list[0] = x;
list[1] = y;
list[2] = {"Joe", 24};
for (int i = 0; i < 3; i++) {
  cout<< 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) の初期値に注意

```
struct Employee{ string name;int age, salary;};

Employee x {"John", 38, 300};
Employee y {x};
vector<Employee> d {{"Robert", 23, 220},
                    {"David",  17, 180}};
vector<Employee> s {x, y};

for (const auto& e : s)
   std::cout << e.name <<": "<< e.salary<<"\n";
```

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

```
struct Type1 {
    string name {"none"};
    int x{10};
    int y{};
};

Type1 a;    // none:  10 0
std::cout << a.name <<": "
          << a.x <<" "<< a.y <<"\n";
```

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

```
struct Point   { string nm;int x{}, y{}; };
struct Triangle{ string nm;Point A, B, C; };
struct Hexagon { string nm;vector<Point> pt;};

Triangle t {"T",{"a",0,0},{"b",3,0},{"c",2,2}};
Hexagon h {"H1",
           {{"a",0,0},{"b",3,0},{"c",4,2}
            {"d",4,6},{"e",3,6},{"f",0,4}} };
cout << t.A.nm <<" "<< h.pt[2].nm <<"\n";
```

値渡し引数と戻り値

- 構造体の関数への引数は値渡し
  - 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

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

Point input() {
    Point p;
    cin >> p.x >> p.y;
    return p;
}

void output(Point g) {
    cout << g.x <<" "<< g.y <<"\n";
}

Point a { input() };
output(a);
output({10, 20});
```

良くない例

- 値渡しでデータがコピーされる様子を想像せよ

```
struct Data { string n; vector<double> d; };

double sum(Data x) {
    double s {};
    for (auto e: x.d)
        s += e;
    return s;
}

int main() {
    Data a { "abc", vector<double>(10000,1.5) };
    double x {};
    for (int i = 0; i < 100000; i++)
        x += sum(a);
    cout << x <<"\n";
}
```

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

```
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 );
}
```

- 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) {...}`

メンバの取り出し

- 構造化バインディング (c++17)

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

Staff john {"John", 10};
std::cout << john.first_name <<" "
          << john.total_year <<"\n";

auto [n,y] {john}; // メンバ値のコピー
std::cout << n <<" "<< y <<"\n";

auto& [nm,ty] {john}; // メンバのリファレンス
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
- 範囲 for 文

```
Staff get()
{
   string n; int y;
   cin >> n >> y;
   return {n, y};
}

...
auto [a, b] {get()}; // 関数の結果
std::cout << 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
- 構造体の定義は不要
  - first と second がメンバ
  - 構造化バインディングの使用で簡潔に記述

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

std::pair p2 {10, 1.5};
auto [i, d] {p2};
std::cout << i <<" "<< d <<"\n";
```

std::pairの使用例

- std::pairを返す関数
  - 成功の有無
  - 成功時の結果

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

auto input() {
   Point p;
   bool f {cin >> p.x >> p.y};
   return std::pair{f,p};
}

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

構造体の cin/cout 対応

- 関数を作って設定する
  - 関数名を特殊な名前にする
  - 引数と戻り値の型を istream&または ostream&とする
  - 入出力の式をそのまま return に指定する
- 3 種のストリームに使用できる

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

std::istream&
operator>>(std::istream& in, Staff& s)
{
   return in >> s.name >> s.age;
}

std::ostream&
operator<<(std::ostream& out, const Staff& s)
{
   return out <<"("<<s.name<<", "<<s.age<<")";
}
```

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

```
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);
}
```

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

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)

- c++20 では自動生成する機能が追加された