C++勉強会

第1回「Hello, world」からクラスまで

Created by T.Miyaji and F.Hirakoba

目次

- C++のHello, world
- C++の型
  - 文字列型とキャスト
  - 参照
- 動的配列とfor文
  - std::vector
- 変数のスコープ
- クラス

C++のHello, world

```cpp
#include <iostream>

int main()
{
  std::cout << "Hello, world" << std::endl;
}
```

C言語との違い

- 標準入出力は、`#include <stdio.h>`ではなく、`#include <iostream>`をインクルードする。
- 関数の引数がないときは、`int main()`のようにカッコ内に`void`と書く必要はない。
- main関数だけは、`return`文を書かなくてもよい。

C++のHello, world

```cpp
#include <iostream>

int main()
{
  std::cout << "Hello, world" << std::endl;
}
```

- `std::cout`は、standard(標準) の console out (コンソール出力)という意味。
- 演算子<< で、コンソールに向けてデータを送る。送られたデータは、バッファに溜まっていく。
- `std::endl`は、end line (改行)という意味。改行して、バッファをフラッシュ（出力）する。

Q1-1 C++で「Hello, world」と表示するプログラムを作ってみよう。

C++プログラムの実行方法

1. C++ソースファイルは、拡張子 **.cpp** で保存する。
2. 端末上で、保存したファイルが置いてあるディレクトリに移動して、以下のコマンドを入力する。
   `g++ (保存ファイル名).cpp -o (実行ファイル名)` もしくは
   `clang++ (保存ファイル名).cpp -o (実行ファイル名)`
3. 端末上で、`./(実行ファイル名)` で実行

端末に `Hello, world` と出力されれば、OK!

C++の型

- C言語の型(整数や小数など)は、C++でも使える。

```cpp
#include <iostream>

int main()
{
  int    x = 0;   // 整数
  double y = 1.2; // 小数
  char   c = 'A'; // 文字

  std::cout << x << '\n' << y << '\n' << c << std::endl;
}
```

C++の型

- C++では文字列型`std::string`が用意されている。

⚠️ 文字列型を使うには、`string`ヘッダを読み込む必要があります。

```
#include <iostream>
#include <string>

int main()
{
  std::string hello = "Hello, world";

  std::cout << hello << std::endl;
}
```

コンパイルして、`Hello, world`と表示されれば、OK!

C++の型

- 異なる型の変数へ値を代入（キャスト）するときは、`static_cast`を使う。

```cpp
int main()
{
  int    x = 0;
  double y = 1.2;
  // int <- double (キャストが必要)
  x = static_cast<int>(y);

  std::cout << x << std::endl;
}
```

文法: `static_cast<(キャストしたい型)>(キャストしたい変数)`

C++の型

- 型を自動判別する`auto`キーワード

```cpp
int main()
{
  auto x = 1;         // int型になる
  auto y = 1.2;       // double型になる
  auto c = 'A';       // char型になる
}
```

⚠️ `auto`キーワードを使ったソースコードのコンパイルが失敗する場合は、コンパイラのバージョンが古い可能性があります。

`g++ (保存ファイル名).cpp -std=c++11 -o (実行ファイル名)` もしくは
`clang++ (保存ファイル名).cpp -std=c++11 -o (実行ファイル名)`としてみてください。

C++の型

- 型を自動判別する`auto`キーワード

```
int main()
{
  auto x = 1;        // int型になる
  auto y = 1.2;      // double型になる
  auto z = func(); // func()の戻り値の型になる
}
```

`auto`キーワードの利点と欠点

利点: `auto z = func()`と書くと、`func()`の戻り値を変更しても受け取り側まで変更する必要はない。

欠点: 型がすぐに判断できない。(利用者が型を意識しないプログラムを作れ、という意見もあります)

C++の型

- 参照（エイリアス）

```cpp
#include <iostream>

int main()
{
  int x = 0;
  int& a = x;
  a = 1;
  std::cout << x << std::endl; // 1と表示される
}
```

参照は、変数に別名を付けることができる。文法:「型& 変数」

上記コードの`a`は、`x`の別名なので、`a`を変更すると`x`も変更される。

C++の型

参照（エイリアス）を使う場面

- 関数内で実引数の値を変更したいとき

```
void func(int& a)
{
  a = 1; // aは実引数xの別名なので、xの値も変更される
}

int main()
{
  int x = 0;
  func(x);
}
```

C++の型

参照（エイリアス）を使う場面

- コピーコストを抑えたいとき

```
void func(Huge& huge)
{
  huge.x = 1; // 構造体hugeのメンバを変更
}

int main()
{
  Huge huge; // 巨大な構造体Hugeを作成
  func(huge);
}
```

Q1-2 日本語「こんにちは」を英語「Hello」に翻訳する関数を作ってみよう

Q1-2のソースコードは、Githubに上げています。

問題の答えは、こちらです。

動的配列とfor文

- 動的配列 `std::vector` は、要素ごとにサイズが変わる。

```cpp
#include <iostream>
#include <vector>
int main()
{
  std::vector<int> x(5);                // 宣言
  std::vector<int> y{1, 2, 3};          // 初期化(サイズは3になる)
  std::cout << y[0] << std::endl;       // アクセス
  y.push_back(4);                       // 末尾へ要素を追加
  y.pop_back();                         // 末尾の要素を削除
}
```

⚠ 動的配列を使うには、`vector` ヘッダを読み込む必要があります。
⚠ コンパイルに失敗する場合は、コンパイラのバージョンが古い可能性があります。
`g++ (保存ファイル名).cpp -std=c++11 -o (実行ファイル名)` もしくは
`clang++ (保存ファイル名).cpp -std=c++11 -o (実行ファイル名)` としてみてください。

動的配列とfor文

- 配列の要素をすべて取得する方法（for文）。

```
#include <iostream>
#include <vector>
int main()
{
  std::vector<int> x{1, 2, 3};
  for(auto i = 0; i != x.size(); ++i)
    std::cout << x[i] << '\n';

  std::cout << std::endl;
}
```

- C++言語では、変数の宣言を先頭でまとめる必要がなくなった。
  for文で作成した`i`はfor文を抜けると破棄される。
- `size()`は、配列の要素数を返す(符号なし整数)。

動的配列とfor文

- 配列の要素をすべて取得する方法（範囲ベースfor文）。

```cpp
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> x{1, 2, 3};
  for(auto& e : x)  std::cout << e << "\n";

  std::cout << std::endl;
}
```

- 文法: `for(型 変数 : 動的配列)`
- 動的配列`x`の要素1つひとつが`e`に代入される。さらに、要素を参照で受け取ることでコピーコストを抑えることもできる。

変数のスコープ

- ある変数が生成されてから、破棄されるまでの範囲のこと。
- スコープが広い = 変数を参照できる区間が広い
- C++には、名前空間（namespace）という概念がある。

```cpp
#include <iostream>
namespace A {
  int x = 0;
}

int main()
{
  int x = 1;
  std::cout << x << std::endl;    // 1と出力される
  std::cout << A::x << std::endl; // 0と出力される
}
```

名前空間は、名前の衝突を防ぐために使用する。
スコープ解決演算子`::`で、名前空間内にアクセスできる。

Q1-3 スコープを意識して、出力結果を予想してみよう

Q1-3の問題は、Githubに上がっています。

答えは、実際に実行してみて確認しよう。

クラス

- データと機能を一体化した型。
- 関数と、その関数内でのみ使用する変数をまとめることができる。

```
#include <iostream>
#include <string>
class Book {
  private:
    std::string title;   // 書名
    std::string author;  // 著者名
    int price;           // 価格
  public:
    void printPrice() { }; // 価格を表示する関数
};
```

- クラスの変数を メンバ変数 といい、関数を メンバ関数 という。
- メンバ変数は、メンバ関数以外でアクセスできないようにする（`private`）。
- メンバ関数は、外部からアクセスできるようにする（`public`）。

クラス

- `private`に設定したものは、外部からアクセスできない。

```
#include <iostream>
#include <string>
class Book {
  private:
    std::string title;    // 書名
    std::string author;   // 著者名
    int price;            // 価格
  public:
    void printPrice() { } // 価格を表示する関数
};

int main()
{
  Book book;          // Bookクラスのオブジェクトを生成
  book.title;         // コンパイルエラー(privateな変数はアクセスできない)
  book.printPrice(); // publicな関数はアクセスできる
}
```

クラス

メンバ変数の初期化方法

```
class Book {
  private:
    std::string title;
    std::string author;
    int price;
  public:
    // コンストラクタ
    Book(std::string t, std::string a, int p)
     : title(t), author(a), price(p) { }
    void printPrice() { }
};
```

- メンバ変数は、コンストラクタ という特殊な関数で初期化する。
- コンストラクタは、クラス名と同じ名前をもつ。戻り値はない。
- メンバ変数の指定順序に従って、コンストラクタで初期化する。（必ず順序を同じにすること！）

クラスの作り方

クラスを作る手順を次のように定めます。

1. 作りたいクラス名に沿ってヘッダファイルを作成する。
2. ヘッダファイルにインクルードガードを作る。
3. クラスの宣言とコンストラクタを実装する。
4. ヘッダファイル名と同名のソースファイルを作成する。
5. クラスを実装する。

以降は、手順1つひとつの詳細について述べます。

クラスの作り方

1. 作りたいクラス名に沿ってヘッダファイルを作成する。

- 宣言するクラス名が`ClassName`のとき、ヘッダファイル名は`class_name.h`とします。
- `string`や`vector`といったライブラリは、ヘッダファイルに記述します。

クラスの作り方

2. ヘッダファイルにインクルードガードを作る。

インクルードガードとは、ヘッダファイルを1回だけ呼ばれるようにするテクニックのことです。

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

// ここにクラスの宣言を書く

#endif
```

- `#ifndef CLASS_NAME_H`とは、`CLASS_NAME_H`が定義されていなければ真となる条件文です。クラス名が`ClassName`のときは、`CLASS_NAME_H`のようにすべて大文字で指定します。
- `#define CLASS_NAME_H`とは、`CLASS_NAME_H`を定義するという文です。これによって2回目以降は、`#ifndef`の条件式が偽となります。
- `#endif`は、`#ifndef`のブロックの終了を表します。

クラスの作り方

3. クラスの宣言とコンストラクタを実装する。

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

class ClassName {
  private:
    int x;
  public:
    ClassName() { }
    explicit ClassName(int x_) : x(x_) { }
    void func();
};

#endif
```

- 引数を取るコンストラクタを作る場合は、**引数を取らないコンストラクタを必ず作成**してください。これはクラスから作ったオブジェクトをデータ構造にもつときに不都合があるからです。
- 引数を取るコンストラクタを作る場合は、`explicit`指定子をつけてください。これは、暗黙的なキャストが行なわれないようにするためです。
- メンバ関数は、ヘッダファイルでは実装しません。

クラスの作り方

4. ヘッダファイル名と同名のソースファイルを作成する。

- 宣言するクラス名が`ClassName`のとき、ヘッダファイル名は`class_name.cpp`とします。

クラスの作り方

5. クラスを実装する。

```
#include "class_name.h"

void ClassName::func()
{
  // 関数の中身
}
```

- ソースファイルでは、実装するクラスの宣言が書かれたヘッダファイルのみ読み込みます。
- メンバ関数は、`クラス名::関数名`で指定します。これは、クラスというスコープ内の関数を実装するためです。

Q1-4 Dogクラスを作ってみよう

問題は、Githubに上がっています。
クラスを作る手順を参考に、ヘッダファイルおよびソースファイルを完成させてください。

- Dogクラスは、名前`name`と年齢`age`をコンストラクタに取るクラスです。
- 以下のようにプロフィールを出力する`profile()`も作ってください。

```
[Name] Taro
[Age]  8
```

`dog.h`と`dog.cpp`を作成したのち、`make run`として、プロフィールが出力されればOKです。
答えは、こちらです。

次回予告

クラスの継承と多態性