プログラミング言語 D

山本和彦
(株)インターネットイニシアティブ
kazu@iij.ad.jp

D とは何か？

- C, C++ の後継者
  - ネイティブコードを生成する
  - VM は使わない
  - C と同じオブジェクトを生成する
    - C / D 相互にリンク可能
    - gdb が使える
- C の評判の悪い機能を直す
  - マクロ、ポインタ、宣言
  - C と似ているけど互換性はない
    - C++ は C と完全互換にしたため複雑になった
- 最近の評判のよい機能を取り込む
  - C++, C#, Java のいいとこ取り
    - ガベージコレクション
      - 明示的に delete() もできる
    - 単一継承のクラス
    - インターフェイス
    - テンプレート
    - 契約プログラミング
    - 単体テスト

山本による誇大広告

- C よりも安全かつ明瞭
  - D にはマクロがない
  - D にはポインタはあるが、普通は使わなくてよい
  - D では宣言が今風
- C++ よりも簡潔
  - D は C との互換性を諦めている
- Java, C# よりも速い
  - D はネイティブコードを生成する

D 言語の素性

- コンパイラ屋が設計実装した言語
  - DIGITAL MARS 社の Walter Bright 氏

新しい言語って、結局ほとんどは二つに分類できる
ような気がする。
一つは、アカデミックな世界から出てくる
根本的に新しいパラダイムに基づいた言語で、
もう一つは、大会社が作るようなRADとか
Webアプリに目を向けた言語。
でもそろそろ、コンパイラ実装の実経験に裏打ちされた
新しい言語が生まれる時なんだろうね。

Michael

コンパイラ屋の視点の例

- マクロはない
  - プリプロセッサの情報はコンパイラには伝わらない
- C++ のテンプレート
  - 演算子 <, >, >> と区別が付かない

```
a<b,c>d;
a<b<c>>d;
```

- D のテンプレート
  - !() を使う
    - ! は二項演算子には使われていないから

```
a!(b,c)
```

実装

- DMD
  - Digital Mars D compiler?
  - Linux, Windows
- gdc
  - Gnu D Compiler
  - Linux, BSD, Mac, Windows

Hello World

- コード

```
import std.stdio; // マクロはない

void main() {
  writefln("Hello World!");
}
```

- コンパイルと実行

```
% gdc -o hello hello.d
% ./hello
```

特長の紹介

- 配列
- イテレータ
- 宣言
- 引数の参照渡し
- クロージャ
- 構造体
- クラス
- 例外処理
- 契約プログラミング
- 単体テスト
- 条件コンパイル

配列はオブジェクトだ

- 大きさを知っている

```
void foo(int[] ar) { // C だと int size が必要
  writefln("%d", ar.length);
}
```

- 動的配列

```
int[] ar;
ar = [10, 11];        // 配列のように！
ar ~= 12;             // push
→ [10,11,12]

ar[0..2] = 0;         // 一括定義
→ [0,0,12]

int[] x = ar[1..3]; // スライシング
→ [0,12]
```

- length

```
ar[length - 1] // 最後の要素
```

イテレータ

- foreach

```
import std.stdio;

void main(string[] args) { // argc は不要
    foreach (argv; args) { // 型推論
      writefln("%s", argv);
    }
}
```

分りやすい宣言

- D の宣言は左から右へ読めばよい
- クイズ1)

```
int (*a[5])[3]; // C
```

→ a は int の3要素配列へのポインタの5要素配列

```
int[3]*[5] a; // D
```

- クイズ2)

```
int (*sw[])(char); // C
```

→ sw は int を返す関数(引数はchar)へのポインタの配列

```
int function(char)[] sw; // D
```

- 定義と呼び出し

```
sw = [&boo, &foo, &woo];
sw[1]('a');
```

引数の参照渡し

- 引数で値を返したいなら out 修飾子を使う

```
import std.stdio;

void main() {
  int a = 0;
  foo(a);
  writefln("%d", a);
}

void foo(out int b) {
  b = 5;
}
```

クロージャ

```
import std.stdio;

void main() {
  int delegate(int) plus1 = acc(1);
  writefln("%d", plus1(3));
}

int delegate(int) acc(int n) {
  return delegate int(int x) {
    return x + n;
  };
}
```

- delegate は function へ統合される予定

構造体

- 初期値やメンバー関数が書ける

```
struct foo {
  int a = 1;
  int b = 2;
  char[] toString() {
    return format("{a: %d, b: %d}", a, b);
  }
}
void main() {
  foo bar;
  static foo baz = {b:3};

  writefln("%s", bar);
  writefln("%s", baz);
}
```

クラス

- 1つの基底クラスを継承
  - 多重継承はない
- インターフェイス
- 抽象クラス
- インスタンスは間接参照

```
Stack st = new Stack();
```

例外処理

- try, catch, finally

```
void locked_foo() {
  Mutex m = new Mutex;
  lock(m);
  try {
    foo();
  } finally {
    unlock(m);
  }
}
```

- scope exit

```
void locked_foo() {
  Mutex m = new Mutex;
  lock(m);
  scope(exit) unlock(m);
  foo();
}
```

契約プログラミング

- Design by Contract
  - in 事前契約, out 事後契約, invariant クラス不変契約
  - コンパイル・オプション(-frelease)で削除できる

```
class Stack {
private:
  int count = 0;
  int[] array;

  int item_at(int i)
    in {
      assert(i >= 1);
      assert(i <= count);
    } body {
      return array[i - 1];
    }

  invariant {
    assert(count >= 0);
  }
}
```

D の契約プログラミングの弱点

- エラーメッセージが貧弱

  Error: AssertError Failure dbc.d(26)

- Eiffel の old が使えない

```
public:
  int pop()
    in {
      assert(count > 0);
    } out {
      // assert(count == (old count - 1));
    } body {
      count--;
      return array[count];
    }
```

  - せめて in と out のスコープがいっしょならいいのにぃ

単体テスト

- main を呼び出す前に、すべての単体テストを実行

```
class Stack {

  unittest {
    Stack s = new Stack();
    s.push(1);
    assert(s.pop() == 1);
  }
}

% gdc -funittest -o stack stack.d
```

条件コンパイル

- version

```
version (demo) {
  writefln("デモバージョン");
} else {
  writefln("商用バージョン");
}
```

```
% gdc -fversion=demo -o foo foo.d
```

- debug

```
debug(1) {
  writefln("count: " , count);
}
```

```
% gdc -fdebug=1 -o foo foo.d
```