情報処理概論

第３回　Fortran の基本１

情報基盤研究開発センター　谷本 輝夫

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

主なシェル(UNIX)のコマンド

コマンド	動作
pwd	現在位置（カレントディレクトリ）の表示
ls	ファイル一覧の表示
less　ファイル名	ファイルの内容表示
cd　移動先	カレントディレクトリの変更（移動）
mkdir　ディレクトリ名	新規ディレクトリの作成
mv　移動元　移動先	ファイルの移動, ファイル名の変更
cp　コピー元　コピー先	ファイルのコピー
rm　ファイル名	ファイルの削除
emacs　ファイル名	ファイルの新規作成,もしくは編集

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

Fortranプログラムのコンパイル gfortran

- 使い方：gfortran ソースファイル –o 実行ファイル

- 例： test.f90 をコンパイルして，その翻訳結果の機械語プログラムを testという名前のファイルに保存する

```
$ gfortran test.f90 –o test
```

プログラムの実行

- コンパイルによって得られた実行ファイルの名前をコマンドとして利用する
  - 例：カレントディレクトリの実行ファイル test の実行

```
$ ./test
```

- UNIXのコマンドと区別するために実行ファイルの前に必ず ./ を付ける
  - ./ の . はカレントディレクトリの意味

エラーメッセージ

プログラムを作成・入力 →

コンパイルをするとエラーが発生

↓

```
program hello
  write(*, *) 'Hello, Fortran"
  write(*, *) 'Let's enjoy!'
stop
end program
```

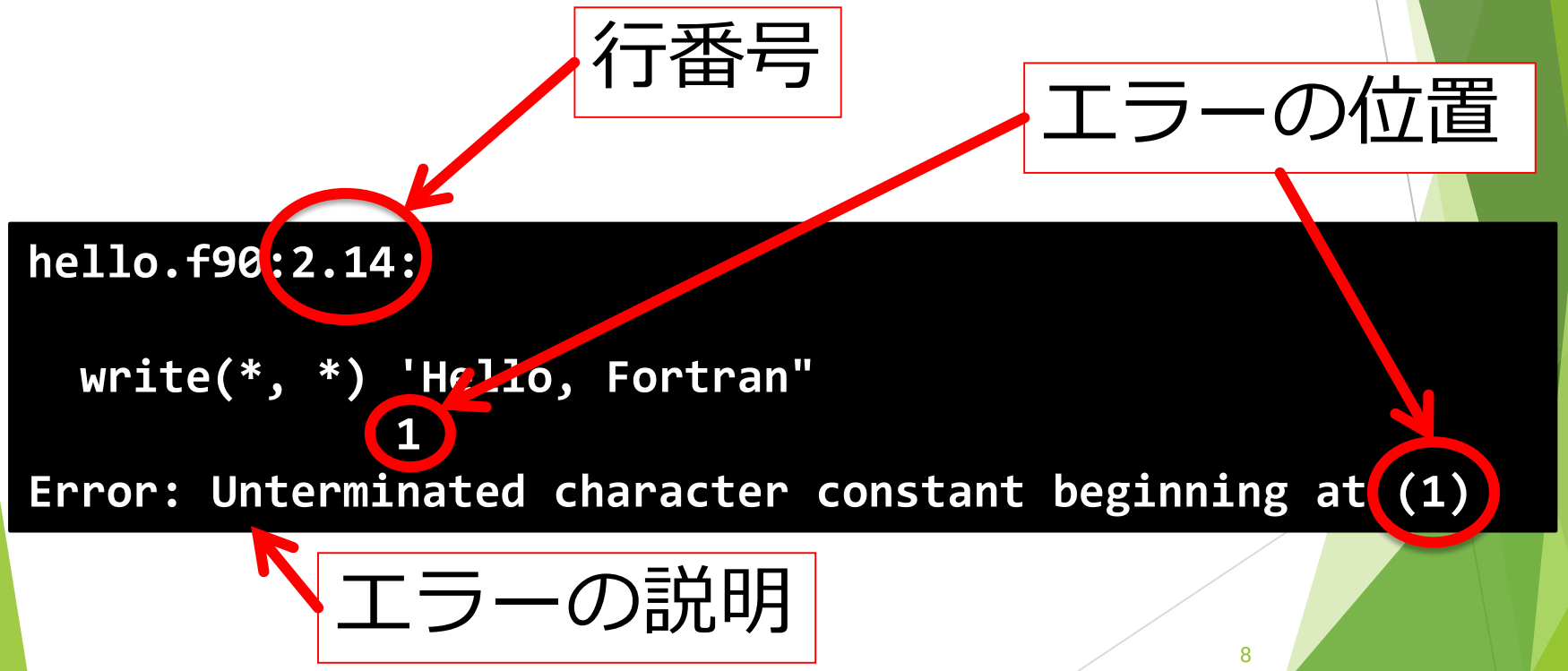
```
$ gfortran hello.f90 -o hello
hello.f90:2.14:

  write(*, *) 'Hello, Fortran"
              1
Error: Unterminated character constant beginning at (1)
hello.f90:3.19:

  write(*, *) 'Let's enjoy!'
                    1
Error: Syntax error in WRITE statement at (1)
```

エラーが発生したら

- エラーメッセージから、エラーに関する情報を読み取る



間違いを探して修正

- 修正箇所の探し方
  - まず、エラーの発生箇所
  - 見つからなければ、その周辺
  - それでも見つからなければ、発生行に関係しそうな場所
- 修正の順序
  - 上の行から順に修正
  - 一つの間違いを修正するだけで、
    後続の複数のエラーが解決する場合が多い
- 修正してコンパイル
  - エラーメッセージが出なくなったら、実行

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

Fortranプログラムの基本形

- 基本構造

```
program プログラムの名前

  プログラムの本体

end program

関数やサブルーチン　（７月頃に紹介）
```

プログラムの本体
＝　主プログラム

- 先ほどのプログラム：

```
program hello
  write(*, *) ‘Hello, Fortran’
stop
end program
```

Fortranプログラムの決まり事

- プログラムの名前は自分の好みでつけてよい
  - 名前に使える文字： 英数字と　_
- プログラムは基本的に上から下に１行ずつ実行される
- ただし，繰り返しや条件分岐等（再来週紹介）によって上に戻ったり，行が飛ばされたりする
- & を使って1行分の内容を複数行に分けて書いてもよい
  - ただし、単語の途中で分けることはできない。

正しい例）

```
program &
test
  write(*, *) 'Hello, Fortran'
stop
end program
```

間違いの例）

```
prog&
ram test
  write(*, *) 'Hello, Fortran'
stop
end program
```

Fortranプログラムの決まり事（続き）

- 大文字と小文字は区別しない.

```
program test  =  Program Test  =  pRoGrAm TeSt
```

- 空白は１個でも複数個でも同じ

```
program test  =  program  test  =  program     test
```

- 空行は無視される

```
program test

  write(*, *) 'Hello, Fortran'
stop
end program
```

=

```
program test
  write(*, *) 'Hello, Fortran'
stop
end program
```

前回のプログラムの意味

- Hello, Fortran と表示する　だけ　のプログラム

```
program hello
  write(*, *) 'Hello, Fortran'
stop
end program
```

- write：文字列や計算結果を表示
  - 後で説明
- stop：その場所でプログラムの実行を停止

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

画面への表示
write

- 利用方法：write （出力先, 書式） 表示内容
  - 例）
```
write(*, *) 'Hello, Fortran'
```
- 出力先：出力する "装置" の番号.
  - ＊ を指定すると、画面に表示
  - ＊ 以外の”装置”については6月に説明予定
- 書式：表示の形式
  - 表示の位置合わせが必要な場合に指定する
  - ＊ を指定すると，表示形式の指定無し
  - ”書式” の書き方については、来週説明予定

文字列の表示

- 文字列：　"　または　'　で囲まれた文字の並び
- 文字列の表示例（どちらも同じ意味）

```
write(*,*) 'Hello.'
```

```
write(*,*) "Hello."
```

- ' や " を表示したいときは？
  ⇒　同じ記号を並べて書く.

プログラム中の表記

```
write(*,*) 'I''m a student'
```

表示結果

```
I'm a student
```

数値の表示

- 使用例

プログラム中の表記	表示結果
`write(*,*) 100`	`100`
`write(*,*) 10 + 20`	`30`

- "数値として" 表示
  - 計算式の場合は、計算結果を表示
  - 文字列の表示との違い

`write(*,*) '100'`	`100`
`write(*,*) '10 + 20'`	`10 + 20`

文字列や数値を並べて表示

- 使用例

プログラム中の表記

```
write(*,*) "Answer 1 is ", 100+200, &
           " , Answer 2 is ", 1000+2000
```

表示結果

```
Answer 1 is 300 , Answer 2 is 3000
```

- , で区切って，数値や文字列を列挙

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

四則演算

- 加算、減算：　+ , -
- 乗算：　＊
- 除算：　/
- べき乗：　＊＊
  - 例）2^3 の計算・表示：

```
write(*,*) 2**3
```

数学の数式との違い

- 括弧は何重にも利用可能
  - 何重になっても、すべて　（　）
  - 例：

```
write(*,*) (((4 + 5) * (10 - (3 - 1)**2)) + &
            (6 - 2) * 2)/2
```

- 乗算記号は省略できない
  - 間違いの例：

```
write(*,*) (10 + 20)(40 + 50)
```

  - 正しい例：

```
write(*,*) (10 + 20) * (40 + 50)
```

今回の内容

- 前回の復習：ファイル操作
- プログラム開発の流れ
- プログラムの基本形
- 画面への文字、数値の出力
- 四則演算
- 整数と実数の違い

プログラム中の実数と整数

- プログラムの中で、整数と実数は区別して扱われる
  - 例）　以下の２つのプログラムは出力結果が違う

```
write(*, *) 10/3
```

```
write(*, *) 10.0/3.0
```

- さらに、実数には精度に応じて複数の種類がある

数値の "型"と、その表し方

- 整数型：　整数
  表し方 ⇒　小数点を付けず、数字のみで表記

```
10
```

- 単精度実数型：　有効桁数が短い実数
  表し方　⇒　小数点と数字のみで表記

```
10.0
```

- 倍精度実数型：　有効桁数が長い実数
  表し方　⇒　末尾に D0 を付ける

```
10D0
```

以下の出力結果がどうなるか予測してみる

```
write(*, *) 10/3 , 10.0/3, 10D0/3
```

型の混在

- 前ページの出力結果

```
3           3.333333           3.33333333333333
```

- 型が違う数値同士の計算結果の型は，
  精度の高い方が選択される．
- ただし，計算式全体ではなく，個別の四則演算毎に選択
  - 例

プログラム中の表記	表示結果
`write(*,*) 1/4*(0.4D0+0.6D0)`	`0.0`
`write(*,*) 1D0/4D0*(0.4D0+0.6D0)`	`0.250000000`

実数の丸め誤差

- 計算機で扱える実数の桁数には限りがある
  ⇒　割り切れない場合には切り捨て等で調整
- 丸め誤差
  ＝　実数計算時の切り捨て等によって生じる計算誤差
- 誤差は計算の順番によって変わる
  - 例）

プログラム中の表記	表示結果
`write(*,*) 1000000.0+0.53+0.005`	`1000000.5`
`write(*,*) 0.53+0.005+1000000.0`	`1000000.6`

計算機の中のデータ

- 全て2進数で処理
  - 0 と 1 で数値を表す方式
  - 電気や磁気での表現が簡単なのでコンピュータでの処理が容易
  - 例えば、電圧が高い = 1、低い = 0
- ホントに２進数で計算が出来る？

２進数	１０進数
0	0
1	1
１０	2
１１	3
１０１０	１０

2進数の計算

0+0=0	0+1=1
1+0=1	1+1=10

0×0=0	0×1=0
1×0=0	1×1=1

組み合わせが少ないので計算回路の実装が簡単

- 計算例：

```
  101     111          010          111
 +100    +010         ×110         ×110
-----   ----         -----        -----
 1001    1001          000          000
                       010          111
                      010          111
                     ------       ------
                     001100       101010
```

bit と byte

- 1bit　＝　２進数の１桁
  - コンピュータにおける情報の最小単位
  - １か０
- 1byte　＝　８bit
  - コンピュータの記憶場所（番地）の単位
  - メモリやハードディスクの容量，ファイルの大きさなどは全て byte 単位で表記
  - なぜ 8 bit?
    - 半角英数字 1文字分に必要な情報量
    - アルファベットと数字全部に番号付け可能（28=256）
    - 本当は 7bit で十分だが，2のべき乗の方が 2進数で扱いやすいため 8bit を単位とした

整数以外のデータは？

- 文字
  - それぞれの文字に番号を付けて管理
    - ‘A’ は 1 番、’B’ は 2 番のように
    - 実際（多くの文字コード）では、’A’は 65 番、’B’ は 66 番、’a’ は 97 番
- 画像
  - 光の3原色(RGB)に分解して，それぞれの色の強さを整数で記録
    - Webでは #FFFFFF と言った表記を使う。
    - RGB各1バイトの色の強さで、#FFFFFF は白，#FF0000 だと赤
  - それを圧縮したもの（ .jpg　.gif　.png ）
- 音
  - 周波数に分解して、それぞれの周波数の強さを整数で記録
    - 「フーリエ変換」を利用
  - 音の波形の信号レベルをそのまま値にして保持（.wav）
    - それを圧縮する（ .mp3　.wma　.aac ）

2 進数の実数

- やはり 2 のべき乗で表現できる。

2進数	10進数
0.0	0
0.1	0.5(=1 x 2^{-1})
0.01	0.25(=1 x 2^{-2})
0.11	0.75=(1x2^{-1} + 1x2^{-2})
0.1011	0.8125 (=1x2^{-1}+1x2^{-2}+1x2^{-4})

- 通常、32桁、もしくは 64桁の 2進数を使用

小数点の位置

- 64桁（もしくは32桁）のうち何桁を小数点以下にするか？
  - 4桁　絶対値が 2^{-4} (=0.0625)より小さい数が扱えない
  - 40桁　絶対値が 2^{24} (=4194304) より大きい数が扱えない
  - 半分　絶対値が 2^{-32} (=約 10^{-10}) ～ 2^{32} (=約 4×10^{9})

- 問題によって過不足がある

浮動小数点

- 小数点の場所を可変にして、その場所の情報も一緒に記録
- 実数を $a \times 2^x$ の形に変換し，a と x を格納
  - ただし 0.0 ≦ a ＜ 1.0
  - 最初の1桁は符号　（0： 正， 1: 負）

- a を仮数部，x を指数部と呼ぶ
- 単精度（全体で32桁）では，仮数部 23 桁，指数部 8 桁
  - 2-127 ～ 2128 の数を表現可能
- 倍精度（全体で64桁）では，仮数部 52 桁，指数部 11 桁
  - 2^{-1023} ～ 2^{1024} の数を表現可能

実数の精度

- 2進数の桁数が大きいほど計算誤差が少ない
  - 精度が高い
- 計算機で用いる実数
  - 単精度：　2進数 32桁
  - 倍精度：　2進数 64桁
- 倍精度の方が計算に要する時間が少し長いが，ほとんどの科学技術シミュレーションでは倍精度を利用
- さらに高精度の計算のため 4倍精度（128桁）で計算する場合もある

演習 3

- p.7 の間違えているプログラム
  - 実行し、エラーを表示させてみる
  - エラー表示にしたがって、バグも見つけ、修正する
- 整数、単精度実数、倍精度実数で同じ計算をして結果が変わる例を考え、実際にプログラムを作って確認する