

プログラミング基礎 第6回

藤江 真也
2021年5月28日

本日の小ネタ①

■ インデントを揃えよう

- C言語のプログラムでは、ブロックが入れ子になる（ブロックの中に別のブロックが入る）
- 内側のブロックでは、外側のブロックに対して字下げをする

■ よい例

```
#include <stdio.h>
```

```
int main (void) {  
    int x = 30;  
    if (x % 2 == 1) {  
        printf("%4dは奇数\n", x);  
    } else {  
        printf("%4dは偶数\n", x);  
    }  
    return 0;  
}
```

■ わるい例①

```
#include <stdio.h>
```

```
int main (void) {  
    int x = 30;  
    if (x % 2 == 1) {  
        printf("%4dは奇数\n", x);  
    } else {  
        printf("%4dは偶数\n", x);  
    }  
    return 0;  
}
```

8

準備

準備

- ファイル(0528.tgz)をダウンロード

```
$ wget http://sites.fujielab.org/ip/files/0528.tgz
```

- ダウンロードしたファイルを展開

```
$ tar zxvf 0528.tgz
```

- 展開されたディレクトリに移動

```
$ cd 0528
```

- keisan.cなどのファイルがあることを確認

```
$ ls
```

講義の進め方

- サンプルプログラムを実行
 - 資料内にはファイル名のみを書きます
- プログラム内のコメントを見ながら説明
 - /* */ で囲まれている部分
- スライドで補足説明
 - スライド(講義資料)は手元で確認できるようにしておくとい

ファイルからの入力


■ 二つの数で計算をするプログラム

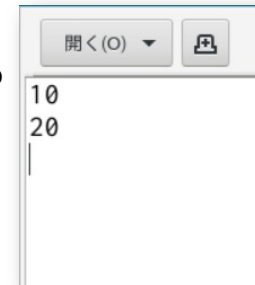
```
#include <stdio.h>
```

```
int main(void) {  
    int x, y;  
  
    printf("Xを入力してください: ");  
    scanf("%d", &x);  
    printf("Yを入力してください: ");  
    scanf("%d", &y);  
  
    printf("X + Y = %d\n", x + y);  
    printf("X - Y = %d\n", x - y);  
    printf("X * Y = %d\n", x * y);  
    printf("X / Y = %d ... %d\n", x / y, x % y);  
  
    return 0;  
}
```

keisan.c

テキストファイルを作成

- gedit で  を押す
- 1行目に10, 2行目に20を入力
- 「保存」を押して保存する
 - ファイル名は何でもよいが、ここでは「value.txt」とする
 - プログラムと同じディレクトリに保存すること



ターミナルで確認

■ ls コマンドでファイルがあるか確認

```
$ ls  
keisan.c  value.txt
```

■ cat コマンドでファイルの内容を確認

```
$ cat value.txt  
10  
20
```

テキストファイルを読み込ませる

■ keisanプログラムに, value.txtを読み込ませる

```
$ ./keisan < value.txt  
Xを入力してください: Yを入力してください: X + Y = 30  
X - Y = -10  
X * Y = 200  
X / Y = 0 ... 10
```

- \$ 実行コマンド < テキストファイル
で, 実行コマンドの実行中の
キーボードからの入力の代わりに,
テキストファイルの内容が入力される
- 入力された文字は画面に表示されない点に注意

本日の小ネタ②

■ ターミナルとエディタを同時に使おう

- gedit はバックグラウンドで起動しよう
 - 最初の起動時に & を付けるとバックグラウンド起動になる

```
$ gedit &
```

- ファイルは「上書き保存」し, gedit 自体は起動したままに
- gedit を起動したままターミナルが使えるので
→ プログラム変更 → コンパイル → 実行 →
がやりやすい.

練習

1. テキストファイルの中の数値を変えて, 計算結果が正しく出るか確認しよう
2. テキストファイルの中の数値の数を, 1つや3つに変えて動作がどう変わるか確認しよう
3. テキストファイルの中に数値以外の文字を入れたらどうなるか確認しよう

繰り返し

思い出そう

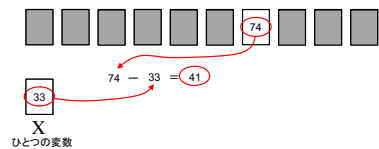
■ 第2回のときの例題

例題 2.1 数列を用いた計算

- 準備
 - 1~100の数字をランダムに10個書き出す
 - 例) 14, 31, 90, 33, 55, 1, 74, 8, 56, 65
 - <http://www.google.co.jp/> で「乱数」と検索すると生成できます
- 準備した数字に対して次の値を計算する方法を考える
 - 最大値, 合計値
- ただし, 以下の条件を守る
 - 計算用の変数 (Xと呼ぶ) を1つだけ使える
 - 同時に見える数字はXの中身と, 10個の数字の中の1つのみ (複数の数値を同時に見ることはできない)

変数Xの中身は書き換えられる
ただし, 今見ている数字か, 今見ている数字とXの内容で計算した結果に限る

10個の数字



最大値を求めるにはどうすればいいか？

■ 数値が2つの場合

max2.c

```
#include <stdio.h>

int main(void) {
    int x = 10, y = 20;

    printf("最大値は%dです\n", x);
    printf("最大値は%dです\n", y);

    return 0;
}
```

最大値を求めるにはどうすればいいか？

■ 数値が3つの場合

max3a.c

```
#include <stdio.h>

int main(void) {
    int x = 10, y = 20, z = 30;

    printf("最大値は%dです\n", x);
    printf("最大値は%dです\n", y);
    printf("最大値は%dです\n", z);

    return 0;
}
```

最大値を求めるにはどうすればいいか？

■ 数値が3つの場合(その2)

max3b.c

```
#include <stdio.h>

int main(void) {
    int x = 10, y = 20, z = 30, m;

    m = x;
    {
        m = y;
    }
    {
        m = z;
    }

    return 0;
}
```

数字の数だけ同じような処理を
書き続けるのはよくない

繰り返し



準備

- 3つの値を読み込んで最大値を求めるプログラムを書こう
- 3つの数値を書いたテキストファイルを読み込ませて正しく最大値が出てくるか見てみよう

maxin.c

```
#include <stdio.h>
```

```
int main(void) {  
    int x, m;
```

```
    scanf("%d", &x);
```

```
    m = x;
```

```
    scanf("%d", &x);
```

```
    if (x > m)
```

```
        m = x;
```

```
    scanf("%d", &x);
```

```
    if (x > m)
```

```
        m = x;
```

```
    printf("最大値は%dです\n", m);
```

```
    return 0;
```

```
}
```

本当にまったく同じことを
2回実行しているの
でなんとかしたい

文にはどんなものがあるか？

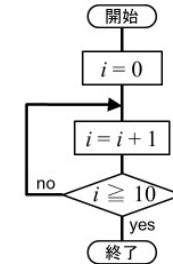
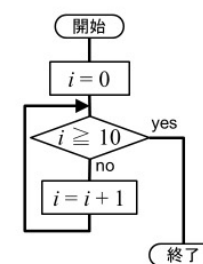
- 変数宣言文 (variable declaration statement)
- 代入文 (assignment statement)
- 関数呼び出し文 (function calling statement)
- if文 (if statement)
- switch文 (switch statement)
- for文 (for statement)
- while文 (while statement)

などなど

フローチャートだと...

フローチャート: 反復構造①

- 反復(ループ)は、判断記号かループ記号で行う
- while文で書ける構造 (前判定型)
- do-while文で書ける構造 (後判定型)



例1 1: /* 100を超える最初の7の倍数を求める */

```
2:
3: #include <stdio.h>
4:
5: int main (void)
6: {
7:     int n, m;
8:
9:     m = 7;
10:    n = 0;
11:
12:    while (n <= 100) {
13:        n = n + m;
14:    }
15:
16:    printf("100を初めて超える7の倍数は、%3dです。¥n", n);
17:
18:    return 0;
19: }
```

while.c

while文

条件式(condition expression)

```
while(n < 100) {
    n = n + m;
}
```

ブロック

- while文では、条件式が成立する限り次に続くブロックが繰り返し実行される
(if文は条件式が成立するときに1度だけブロックが実行される)

```
while(n <= 100) {
    n = n + m;
}
```

1回目 (n:0, m:7) —
条件式: 0 <= 100 ... 真
ブロック: n = 7; (0 + 7)

2回目 (n:7, m:7) —
条件式: 7 <= 100 ... 真
ブロック: n = 14; (7 + 7)

3回目 (n:14, m:7) —
条件式: 14 <= 100 ... 真
ブロック: n = 21; (14 + 7)

15回目 (n:98, m:7) —
条件式: 98 <= 100 ... 真
ブロック: n = 105; (98 + 7)

16回目 (n:105, m:7) —
条件式: 105 <= 100 ... 偽
ブロック: 実行されない

n が105の状態に次
処理が移る

代入演算子

op.c

n = n + a; n = n - a;
n = n * a; n = n / a; n = n % a;
のように、ある変数に演算を加えて、
同じ変数に代入する(上書きする)ということがよく行われる
ため、次のような書き方が出来る

```
n = n + a; ⇔ n += a;
n = n - a; ⇔ n -= a;
n = n * a; ⇔ n *= a;
n = n / a; ⇔ n /= a;
n = n % a; ⇔ n %= a;
```


インクリメント／デクリメント演算子

`n = n + 1; n = n - 1;`

1を足したり引いたりする操作はさらに頻繁に行われるので、特殊な単項演算子(unary operator)が存在する

■ インクリメント演算子(increment operator)

➤ ++

`n = n + 1; ⇔ ++n; または n++;`

■ デクリメント演算子(decrement operator)

➤ --

`n = n - 1; ⇔ --n; または n--;`

参考 インクリメント／デクリメント演算子の前置と後置

- ++n, n++は条件式の中でも使える が意味は異なる

例えば以下のとおり

<code>if (++n < 100) {</code>	<code>if (n++ < 100) {</code>
<code> </code>	<code> </code>
<code>}</code>	<code>}</code>

前置の場合は、
条件式が評価される前に
インクリメントされる

後置の場合は、
条件式が評価された後に
インクリメントされる

do-while文

```
do {  
  文1;  
  文2;  
  ...  
} while(条件式);
```

ブロック

条件式

- do-while文では、ブロックを実行した後、条件式を確認して真であれば繰り返す
- while文との違いは条件式の確認とブロックの実行の順番だけ
 - while文: 先に条件式を確認する
 - do-while文: 先にブロックを実行する

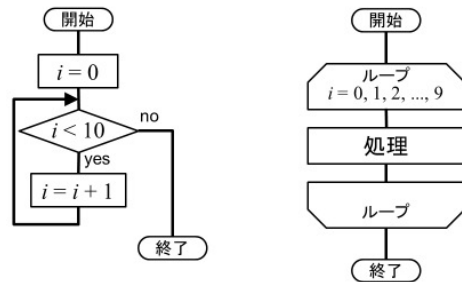
文にはどんなものがあるか？

- 変数宣言文(variable declaration statement)
- 代入文(assignment statement)
- 関数呼び出し文(function calling statement)
- if文(if statement)
- switch文(switch statement)
- for文(for statement)
- while文(while statement)
- などなど

フローチャートだと...

フローチャート: 反復構造②

■ for文で書ける構造(前判定型)



情報処理・プログラミング基礎 第2回

13

例2

for.c

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     int n, m;
5:
6:     m = 7;
7:     printf ("100までの%dの倍数\n", m);
8:
9:     for (n = m; n <= 100; n += m) {
10:        printf ("%d\n", n);
11:    }
12:
13:    return 0;
14: }
```

for文

初期化式 (initialization)

条件式 (condition)

再初期化式 (afterthought)

```
for (n = 1; n <= 100; ++n) {
    ...
}
```

■ for文は以下のような処理をする

1. 初期化式を実行する
2. 条件式を評価し、偽だったら終了する
3. ブロックを実行する
4. 再初期化式を実行する
5. 2に戻る

for文とwhile文

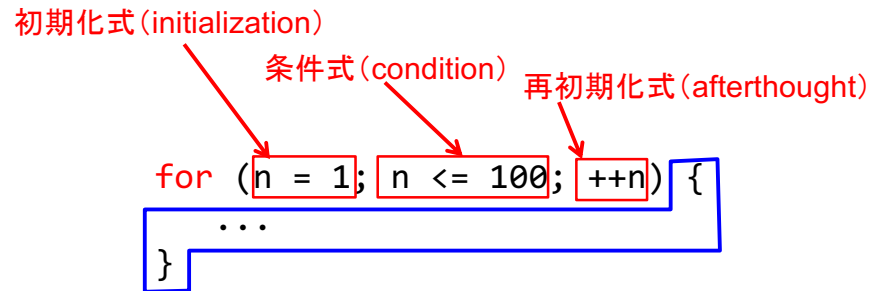
```
for (n = 1; n <= 100; ++n) {
    ...
}
```



```
n = 1;
while (n <= 100) {
    ...
    ++n;
}
```

for文と同じ処理はwhile文だけでも書ける
が、カウンタを利用したループはよく使われる
ため、for文もよく使われる

for文での式の省略



■ 3つの式は省略可能

- 初期化式や再初期化式を省略した場合は単に実行されない
- 条件式を省略した場合は...?

無限ループ (infinite loop) 0 は 偽 (FALSE) 0 以外は 真 (TRUE)

```
while (1) {  
    ...  
}
```

```
for (;;) {  
    ...  
}
```

inf.c

■ 上記のようにすると、ブロックの実行が永遠に繰り返される

- ロボットのように電源投入後は永遠に同じことを繰り返せばよいようなプログラムではこのようなループが頻繁に使われる
- ブロック内で **break;** を実行すると、ループを抜け出すことが出来る

ブロックの省略

- if文, while文, for文のブロックは、ブロックの中が1文である場合に限り、ブロックではなく文で書くことが出来る

