プログラミング入門
第13回講義

～最終回～
よいプログラミングのスタイル

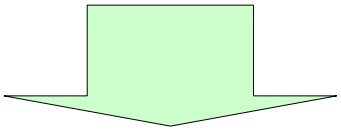
よいプログラムとは何か

動けばよいというわけではない

- プログラムがわかりやすい
  - 見た目がきれい
  - 修正、変更がしやすい
- 効率が良い
  - 実行速度が速い
  - 無駄な変数がない

何故分かり易いプログラムが良いのか？

- 他人にも、自分にも理解しやすい
  ⇒1ヶ月後の自分は他人である
- 誤りが混入しにくい
- 修正（デバッグ）が楽



見た目に美しいプログラムは良いプログラムの基本

分かりやすいプログラムのために

- 分かりやすい変数名
- ソースファイルを見やすく
  - インデント
  - 空白行の使用
  - コメント
  - マクロ
  - 適切な長さに分割（関数化）
- 表現の選択

分かりやすい変数名の工夫

- 変数名は31文字まで使える
  ⇒意味の分かりやすい変数名を使う。
  良い例: `zeiritu, souryou`
  ⇒短いと意味が分からないし、あまり長くても見づらい
  悪い例: `a, b, zeikomigoukeigaku`
- ループのカウンタなどの、特に意味のない変数には、単純な変数名を使う(通常慣例的にi,j,kなどを使用)
  例: `for( i = 0 ; i < n ; i++ )`
- 逆にi,jなどは一般的にループ以外の変数にはほとんど使われないので、使わないようにする。

見やすいソースプログラム

- 一見してプログラムの構造がわかるように、
  - 空白
  - 空行
  - インデント

  などを利用して、ソースプログラムを整形する
- 整形せず、乱雑に書いたプログラムは、誤りが混入しやすい

インデント(1)

- プログラム中で、if文や、for, while ループの中身を、行頭に空白を入れて一段下げて記述すること
- 下げる文字数は2～4文字ぐらいが適当
- emacsのC-modeの場合 M-x indent-region かタブキーでインデントが出来る。
- インデントが崩れているプログラムは「汚い！」と感じる美意識を持とう

インデント(2)

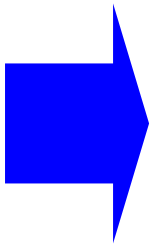
- インデントの方法には何通りかある。いろいろなプログラムを見て自分のスタイルを作ること
- 同じプログラムではインデントスタイルを首尾一貫させること。そうでないと余計見づらい。
- 以下はインデントの代表例である（他にもある）

```
for(i = 0 ; i < 10 ; i++)
{
  do_something();
}
```

```
for(i = 0 ; i < 10 ; i++){
  do_something();
}
```

インデント・空行の例

```
#include <stdio.h>
main(){
int i;
while(1){
printf(“input number”);
scanf("%d",&i);
  if(i<=0) break;
if(i%2 == 0) printf("%dは偶数＼n",
   i);
else{
printf("%dは奇数＼n", i);
}}
}
```



```
#include <stdio.h>
main()
{
  int i;
  while(1)
  {
    printf(“input number”);
    scanf(“%d”,&i);

    if(i<=0) break;
    if(i%2 == 0)
    {
      printf(“%dは偶数＼n”, i);
    }
    else
    {
      printf(“%dは奇数＼n”, i);
    }
  }
}
```

意味の切れ目の空行

コメント

- プログラム中に、注釈として記述する
- 実行に影響はない(コンパイラに無視される)
- /* ではじまり */ で終わる
- コードだけでは分からない、付随情報や他のプログラマに対するメモなどを書く
- コメント行の書き方にも何種類かのスタイルがあるので、色々なプログラムに接して慣れていくと良いだろう

コメントの例

```
/*
 *  flprl.c   flip right and left
 *
 *  pbm形式の白黒画像データを受け取って,
 *  左右反転させた画像を作成
 *
 *  終了コード
 *    0 : 正常終了
 *    1 : 入力データの形式がおかしい
 *    2 : 入力データの画素数が多すぎる
 *    3 : 入力データがおかしいかデータが揃わないうちに
 *        EOFになった
 *    4 : 入力データの画素の値が0／1ではなかった
 *
 */
#include <stdio.h>
(中略)
  /* 最初にP1と書いていないものはデータ形式が違う */
  /* この部分はプログラミング入門では扱わない文字 */
  /* 型を使用しているので、今のところは呪文だと   */
  /* 思っておいてください                         */
   if (getchar() != 'P' || getchar() != '1'){
     fprintf(stderr, "データの形式が違います\n");
     exit(1);
   }
(以後略)
```

プログラムの最初に、名前機能の概略などを書く。
作者、最終更新日時なども入れると良いだろう

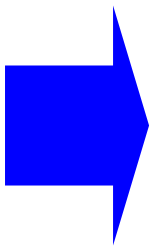
プログラムを見ても分からないようなメモも書いておく

マクロ

- マクロの有効性は以下の二つである
  - プログラムの修正が簡単
  - 定数より意味が分かりやすい

```
#include <stdio.h>
main()
{
  int data[100],i = 0;
  ..
  while(i < 100){
    scanf("%d",&data[i]);
    i++
  }
...
```

何故100なのか分かり難い



```
#include <stdio.h>
#define MAX 100
main()
{
  int data[MAX],i = 0;
  ..
  while(i < MAX){
    scanf("%d",&data[i]);
    i++
  }
...
```

配列を大きくする時はここを変えるだけ

プログラムの長さ

- プログラムは様々なケース（例えばエラーの場合など）を考えれば考えるほど長くなる。
- しかしソースが長くなれば長くなるほど可読性（読み易さ）がなくなる
- プログラムが長くなる場合はある程度まとまった部分を関数として独立させると良い。
- 関数にすることで以下のような利点がある
  - 複数の場所で何度も同じ処理をさせることが出来る
  - 各関数はあまり長くないのでソースの可読性が増す

表現法の選択

1から10までの和を5通りの方法で求める

- 5通りのループの計算結果は、どれも同じである
- なるべく理解しやすい表現を選ぶことが大事(特に、初心者のうちは)
- 1から10までの和を求めるループの他の書き方を考えてみよう

```
#include <stdio.h>
main()
{
  int i, sum;

  /* 方法1 */
  sum = 0;
  for(i = 1 ; i <= 10 ; i++){
    sum += i;
  }

  /* 方法2 */
  sum = 0;
  i = 1;
  while(i <= 10){
    sum += i;
    i++;
  }

  /* 方法3 */
  sum = 0;
  for(i = 1 ; i <= 10 ; sum += i++);

  /* 方法4 */
  sum = 0;
  for(i = 0 ; i < 10 ;){
    sum += ++i;
  }

  /* 方法5 */
  sum = 0;
  i = 0;
  while(i < 10){
    sum += ++i ;
  }
}
```

それぞれ変数sumに1から10までの和が計算される

効率のよい変数の利用

- 変数の利用目的、有効範囲を考えて、無駄な変数の利用はやめる
  - カウンタ変数の再利用
  - 配列を使う必要があるかどうか考える
- ただし、プログラムが分かりにくくなってしまうような使い方はしない
  - 意味を持った名前の変数を別の用途にも使いまわすと、逆に分かりにくくなってしまう。

カウンタ変数の再利用

```
for(i = 0 ; i < n ; i++){
   sum = sum + x[i];
}

ave = (float)sum / n;

for(j = 0 ; j < n ; j++){
   printf("%d:%d\n",j,x[j]-ave);
}
…
```

```
for(i = 0 ; i < n ; i++){
   sum = sum + x[i];
}

ave = (float)sum / n;

for(i = 0 ; i < n ; i++){
  printf("%d:%d\n",i,x[i]-ave);
}
…
```

- このiは、このループ以降では使わないので
- このjは、iを再利用してかまわない

ムダな配列の利用

平均値を求める

```
n = 0
while(1){
  status = scanf("%f",&x[n]);
  if(status == EOF) break;
  sum = sum + x[n];
  n++;
}
average = (float)sum / n;
…
```

```
n = 0
while(1){
  status = scanf("%f",&x);
  if(status == EOF) break;
  sum = sum + x;
  n++;
}
average = (float)sum / n;
…
```

以降でx[n]を参照しないならば、配列変数を使う意味がない

⇒単純変数で置き換えて良い

アルゴリズム（方法）の効率化

- 無駄な計算を減らす(変数をうまく使う)
- ひとつに出来るループは、分けずにまとめる
- ループ回数を考える

ムダな計算を減らす

```
if(b > n*(n+1)*(2*n+1)){
  a = b - n*(n+1)*(2*n+1);
}
else{
  a = n*(n+1)*(2*n+1);
}
….
```

```
c = n*(n+1)*(2*n+1);
if(b > c){
  a = b - c;
}
else{
  a = c;
}
….
```

左は、同じ計算　n*(n+1)*(2*n+1)　を3箇所で行っている!!

if文のどちらを実行しても同じ計算を2回行うことになる

一つ変数を用意して計算しておけば、この計算は1回で済む

ループをひとつに

```
for(i = 0; i < n;
   i++){
   scanf("%d",&x[i]);
}
sum1 = 0;
sum2 = 0;
for(i = 0; i < n;
   i++){
   sum1 += x[i];
   sum2 += x[i]*x[i];
}
….
```

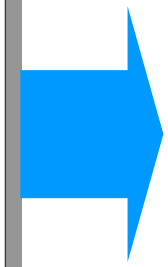
```
sum1 = 0;
sum2 = 0;
for(i = 0; i < n; i++){
   scanf("%d",&x[i]);
   sum1 += x[i];
   sum2 += x[i]*x[i];
}
```

さらに、x[i]をもう使わないなら

```
sum1 = 0;
sum2 = 0;
for(i = 0; i < n; i++){
   scanf("%d",&x);
   sum1 += x;
   sum2 += x*x;
}
```

ループ回数を考える

```
scanf("%d",&x);
for(i = 0; i < N; i++){
  if(x == data[i]){
    found = 1;
  }
}
if (found == 1){
....
```

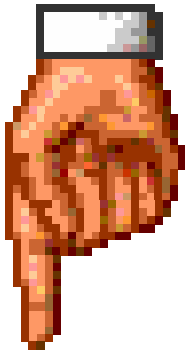


```
scanf("%d",&x);
for(i = 0; i < N; i++){
  if(x == data[i]){
    found = 1;
    break;
  }
}
if (found == 1){
....
```

左例は、変数xと等しいデータが見つかった後もループし続ける。
右例では見つかったらすぐループから脱出する

（Nが1000だとすると、左は必ず1000回ループするが、右の場合のループ回数は最良で1回、最悪でも左と同じ1000回である。）

プログラムの常套手段を覚えよう

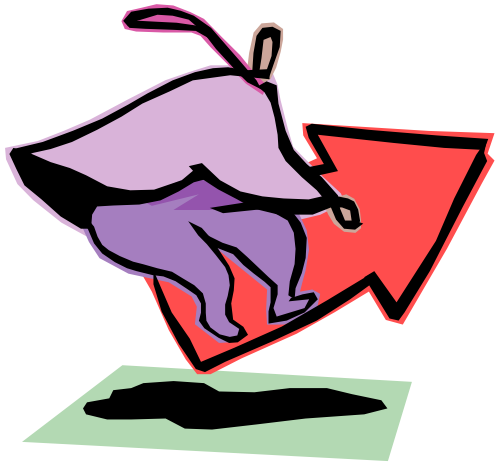


そのためには

さまざまなプログラムを読んでみよう

この講義のポイント

- 分岐
  - ifとswitch-case
  - 条件式
- ループ
  - for, while
  - 無限ループ
- 配列
  - 宣言、配列変数の個数とインデックス（添字）
  - 二次元配列と二重ループ
- 関数
  - プロトタイプ宣言、定義、呼び出し
  - 引数、仮引数



ProgrammingI
文字・文字列
ファイル
ポインタ
構造体

プログラミングIに向けて

- 後期には、プログラミングI（必修）が始まる
  - プログラミング入門の知識を前提に、さらに高度なことを学ぶ。
- 夏休み中に、プログラミング入門の範囲をしっかり復習し、更に時間があればその先を少し予習しておこう。
- プログラミングはそれ自体が「目的」ではなく、何かを作るための「ツール（道具）」であるので、
  - 他の授業では使える事が前提となる
  - 使わないと錆びる！
  - 問題意識を持ち、使うチャンスがあれば積極的に使う