C 言語の初歩

C 言語で学ぶ Fortran の初歩

兵庫県立大学 シミュレーション学研究科
大野

1. コンパイラ

コンピュータに、何か仕事をさせたい。そのためには、コンピュータに指示をしなければならない。その指示のあつまりが、プログラムである。

では、何語でプログラムを書けばよいのか？ 英語？ 日本語？ フランス語？ コンピュータは、日本語も英語もフランス語も理解できない。理解できるのは機械語であり、たぶん 100010001111000 など。これは、おそらく常人（つねびと）には理解できない。

人間(一般人)が理解できる形でプログラムを書くのは不可能なのか？ プログラミング言語を使えば可能である。ただし、人間が書いたプログラムを機械語に直す翻訳機(コンパイラ)が必要である。

プログラミング言語といっても、いろいろな種類がある。たとえば、C/C++、Fortran 77, Fortran90、Java、etc。それぞれの言語は、それぞれの言語用のコンパイラが必要である(Fortran のコンパイラで、C 言語で書かれたプログラムはコンパイルできない)[1]。

2. プログラミング言語

プログラミング言語は人間が理解可能であるが、言語というくらいなので文法がある。英語やフランス語のように、文法を覚えないとプログラムが書けない。コンピュータは融通が利かないので、すこしでも間違うと、コンパイルできない。英会話は、少しくらい文法や単語を間違っても、アメリカ人やイギリス人は理解してくれる。

ここまでをまとめると、プログラミング言語の文法に従い、プログラム(ソースコード[2]と呼ぶ)を書き、それをコンパイルするとコンピュータが実行できる実行ファイルができる。

コンピュータの基礎では、C 言語の基礎を学ぶ。C 言語の学習に最低限必要なものは、

- テキストエディタ(Windows ならサクラエディタ、秀丸エディタ、MIFES、TeraPad、Linux なら vi、emacs、nedit、gedit など)
- C コンパイラ

---

[1] コンパイルが必要ないプログラミング言語(インタプリタ言語と呼ばれる。コンパイラが必要な言語は、コンパイラ言語と呼ばれる)も存在する。たとえば、Perl や昔の BASIC。これらの言語は、実行時に機械語に翻訳されながら実行される。そのため、一般的には、コンパイラ言語で書かれたものより遅い。Java は、その中間かもしれない。興味がある人は調べてみよう。

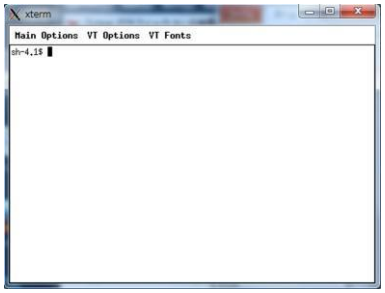
[2] テキストファイル（文字コードのみで記述されている）である。テキストファイルとは、メモ帳などによって開くことができるファイルである。Word の docx ファイルは文章であるがテキストファイルではない。

である。ここでは、Windows + Cygwin で C 言語を学ぶ。Cygwin とは、Windows 上で動作する Linux と考えればよく、フリ”で入手可能（家のパソコンにも無料でインストールできる）である。また、C/C++コンパイラ、Fortran コンパイラ、テキストエディタもインストールすることができる。

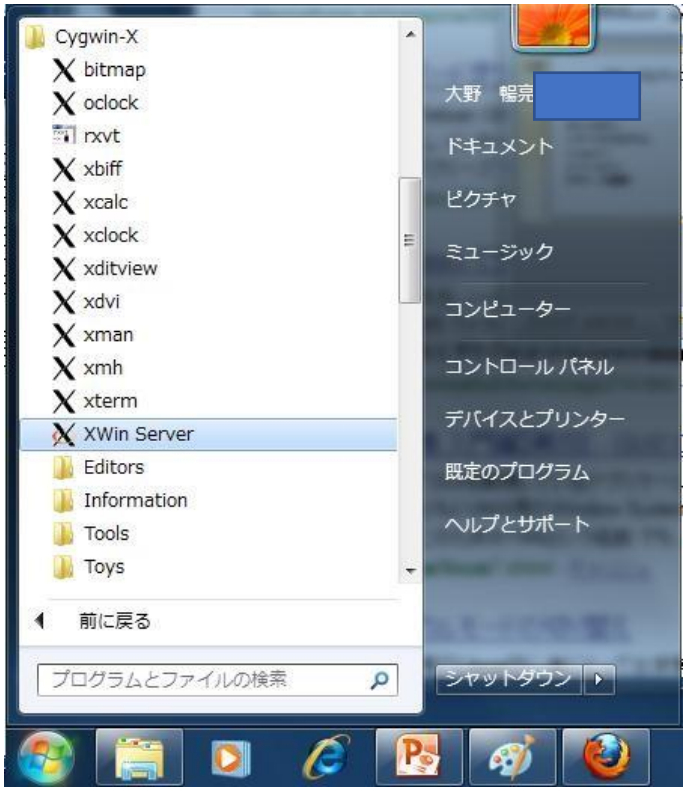
3. Cygwin でのソースコード入力、コンパイル、実行

実習室の PC で、Cygwin を起動してみる。

スタートボタン
→ すべてのプログラム
→ Cygwin-X
→Xwin server
で X サーバを起動



たぶん↑xterm も起動する。
起動しないときは、
スタートボタン
→ すべてのプログラム
→ Cygwin-X
→ Xterm

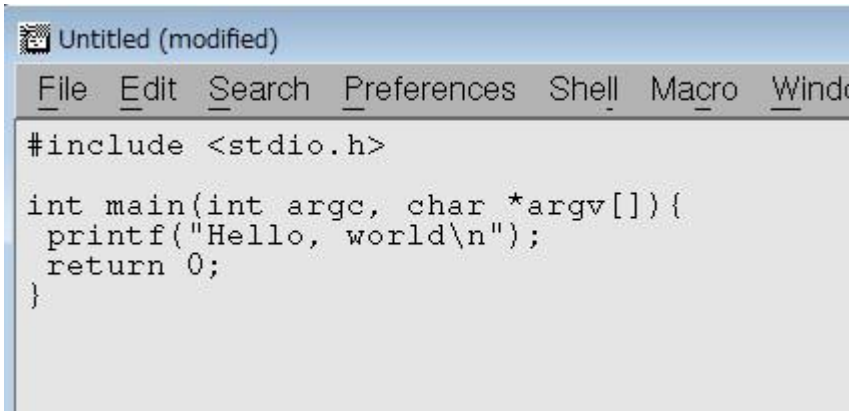


xterm の窓の中で

nedit␣&⏎

と入力すると、nedit というエディタが起動する。&をつける意味は、Linux でコマンド（プログラム）を実行すると、そのコマンドが終了するまで、次のコマンドが実行できない。しかし、コマンドの最後に&をつけて実行すると、そのコマンドはバックグラウンドで実行されるので、すぐに次のコマンドを実行できる。nedit に&をつけてバックグラウンドで実行することで、nedit を終了することなく、つまりソースコードを編集しつつ、そのソースコードをコンパイルをすることができる。

nedit で次のプログラムを入力してみよう。



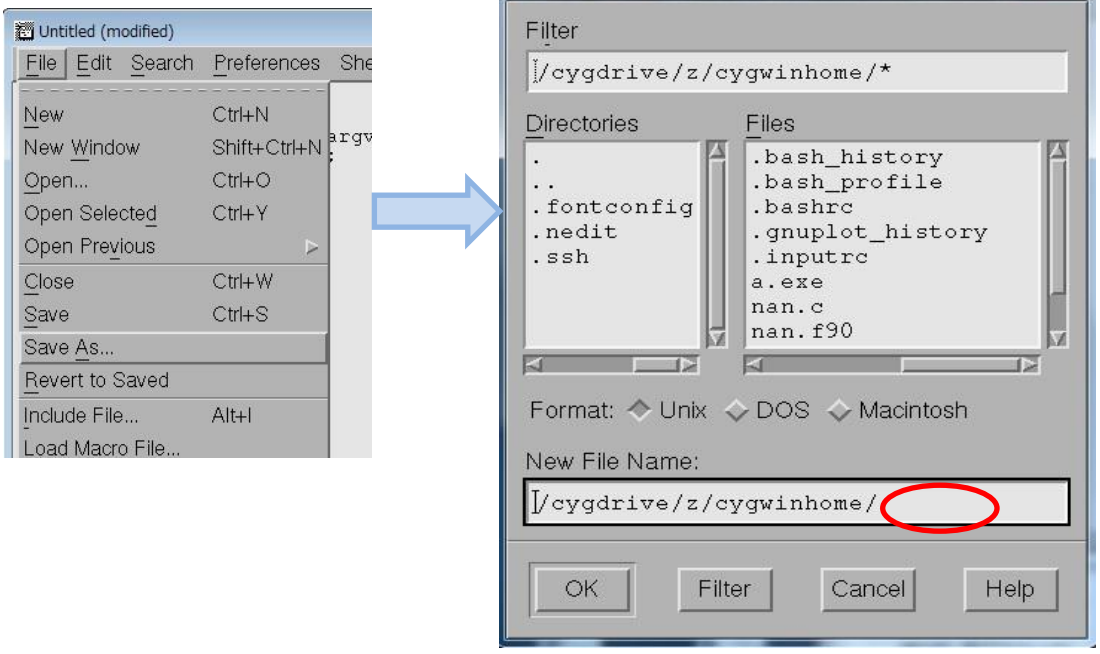

```
#include␣<stdio.h>⏎
⏎
int␣main(int␣argc,␣char␣*argv[ ])⏎
{⏎
 printf("Hello,␣world¥n");⏎
  return␣0;⏎
}⏎
```

ただし、

⏎　：改行を表す,

⏎　␣:スペースを表すエディタの中では、¥はバックスラッシュになる

入力が終了したら、hello.c として保存する(File → Save As)。



hello.c をいよいよコンパイルする。cygwin には、gcc というフリーの C コンパイラがインストールされているので、それを使用する。

xterm で、

gcc␣-o␣hello␣hello.c⏎

とすると、hello.c がコンパイルされ、hello.exe という実行ファイルが出来る。

これを実行するには

./hello.exe⏎

とする。カレントディレクトリを表す ./ をつけなければいけないことに注意。普通の Linux では、hello.exe ではなく、hello という実行ファイルができる。

単に

gcc␣hello.c⏎

とすると、a.exe(普通の Linux では a.out)という実行ファイルができる。

4. C 言語のプログラムの基本と書き方の注意点

```
#include␣<stdio.h>⏎
int␣main(int␣argc, char␣*argv[])⏎
{⏎
printf("Hello,  world¥n");⏎
return␣0;⏎
}⏎
```

hello.c

main(){....}を main 関数と呼ぶ。C 言語や C++言語では、main 関数を上から下に向かって実行し、main 関数が終了したら、プログラムが終了する。

書き方の注意点としては、

- 大文字と小文字を区別し、半角文字で入力すること。全角文字は使わないこと。
- 文の最後に、「;」 (セミコロン)を入れること。
- Python と違い改行やスペースは、比較的自由にできる。例えば先ほどのメイン関数は、

```
int main(int argc,                    char *argv[ ]){
                printf(“hello, world¥n”);          return 0;}
```

と書いても文法的に間違いではない。 （でも、見にくいので、やめよう）

ただし、語句の途中での改行やスペースはだめ。ダメな例

```
int main(int argc, char *argv[ ]){
                printf(“hello,
world¥n”); return 0;}
```

練習問題 1.

hello.c について、下記を実行せよ。

1. 「Hello, World」 の部分を色々変えて、コンパイル・実行してみよう。
2. printf をもっと入れてみよう。
3. Hello, world¥n の¥n は、改行を表す。¥n を消して、コンパイル・実行してみよう

printf("…")が、" "で囲まれた部分を表示する機能を持つことが容易に想像できると思う。printf のような一つのまとまった機能をもつものを C 言語では関数と呼ぶ。(C でいう関数を命令と呼ぶ言語もある)。printf 以外にも、さまざまな機能を持った関数が、標準で用意されている。関数の呼び出し方は、 関数名(引数); printf の例では、printf が関数名、"hello, world¥n"が引数である。main 関数以外にも自分で関数を作成することもできる

main 関数以外の関数は、使用する前に宣言をしなければならない。では、 printf は、どこで宣言されているのか？ stdio.h の中である。stdio.h の中で printf の宣言が書かれており(printf 以外にも数多くの関数の宣言が書かれている)、#include <stdio.h>をソースコードの頭に書くことで、ソースコード内で printf が使用可能になる (stdio.h のようなファイルをヘッダファイルと呼ぶ)。

stdio.h の他にも、stdlib.h や math.h などのヘッダファイルがあり、それぞれ多くの関数の宣言が書かれている。C 言語で標準で用意されている関数をプログラムで使用する場合、その関数が宣言されているヘッダファイルをインクルードする必要がある。

5. 変数とデータ型

変数とは、コンピュータに、何か記憶させたい時に使う記憶の箱。いろいろなデータ型があり、記憶させるデータの種類によって、使い分けなければならない。例えば、

char 型（1 バイト）	文字や文字列（配列使用時）を記憶させることができる。整数 -128 ～127 を記憶させることもできる。

int 型（4 バイト）	整数を記憶させることができる。2009 とか -558000 とか。小数点がつく数値は記憶できない。
float 型(4 バイト、単精度)	実数を記憶させることができる。5.5 とか 3.14159 など。float 型は 6 桁くらい、double 型は、15 桁くらいまで記憶できる。
double 型（8 バイト、倍精度）	

```
1
2
3
4
5
6
7
8
9
10
```

```
#include <stdio.h>⏎
⏎
int main(int argc, char *argv[ ])⏎
{⏎
int a;⏎
double b;⏎
a = 1; b = 3.14159;⏎
printf("a = %d, b = %f¥n", a, b);⏎
return 0;⏎
}⏎
```

var1.c

5,6 行目の int a や double b は、変数の宣言であり、関数の上のほうでしておく必要がある[3]。a, b は、変数の名前であり、宣言したのち、型にあったデータを記憶させることができる。たとえば、a には整数、b には実数を記憶させるとこができる。

変数のネーミングはアルファベット・数字・アンダーバー(“_”)等を使い自由にできる。単純な a や b などより、分かりやすい、意味のある名前にしたほうが、プログラムのミスを減らせる。

例：double PI = 3.14159; /* π 、円周率 */

ただし、数字が先頭に来ることは許されない。

例：double 10g;

つぎに、7 行目の a=1; や b=3.14159;であるが、これは＝の右にある数字が、a や b の中にセットされる。**a = 1** は、「a が 1 に等しい」という意味ではなく、「a に 1 を代入する」という意味。

8 行目の printf では、a と b の中身を表示させている。int 型は%d、 double 型や float 型は%f で表示できる。

```
printf("a = %d, b = %f¥n", a, b);
```

[3] 正確には、ブロックの上のほうで宣言する。ブロックについては触れないが、興味のある人は調べること。

練習問題 2

var1.c につき、下記を実行せよ。

1. a, b にいろいろな数値を代入してみよう
2. a = 3.14159;
3. b = 5; etc.…

float と double の精度の違いにも気を付けよう。次の例は、同じ数字を float と double に代入してどのような違いが出るか確認するプログラムである。

```
#include <stdio.h>

int main(int argc, char*argv[ ])
{
  double pid = 3.14159265358979;
  float pif = 3.14159265358979;
  printf("pid = %.14f¥npif = %.14f¥n", pid, pif);
  return 0;
}
```

実行結果

```
$ ./a.exe
 pid = 3.14159265358979
 pif = 3.14159274101257
```

float 型の精度では、上記の π をすべて記憶できていないことがわかる。

6. 計算

計算用の記号

+	足し算
-	引き算
*	掛け算。×の代わり
/	割り算。÷の代わり
%	余り。5%4 を計算すると答えは 1.。整数のみの計算のみに有効。
( )	括弧。何重になっても( )を使い、中括弧{ }は使用しない。

練習問題 3 calc1.c の 5 行目の式の右辺をいろいろ変えて実際に計算してみる。

1. a = 5%2+5;
2. a = (1+(2+4)*3)*2;
3. a = 5/2;

```
1  #include <stdio.h>
2   int main(int argc, char *argv[])
3  {
4    int a;
5   a = 5*2+1;
6    printf(“a=%d¥n”,a);
7    return 0;
8  }
```

calc1.c

練習問題 4

calc2.c の 5 行目の式の右辺をいろいろ変えて実際に計算してみる。

1. a = 5%2+5;
2. a = 5.0/2.0;
3. a = 5/2;

etc….

```
1  #include <stdio.h>
2   int main(int argc, char *argv[])
3  {
4    double a;
5   a = 5.0*2.2+1.5;
6    printf(“a=%f¥n”,a);
7    return 0;
8  }
```

calc2.c

2 番と 3 番の結果の違いに注意せよ。整数のみの計算の場合、割り算をしても小数点が出てくる計算はしてくれない。

また、整数、実数などの計算が混在する場合は、大きいほうに変換されたのちに計算される

- 例 1：5*2.0 → 5.0*2.0 (整数 5 が実数 5.0 に変換される)

- 例 2：5*2+1.5 → 10+1.5 → 10.0+1.5 →11.5 明示的に型変換することもできる
- 例：(double)5→5.0, (int)5.3→5

実数(5.0 など)は、基本的に double 型として扱われる。float 型として扱ってほしいときは、5.0f または(float)5.0 などとする。

- 例：float a; a=3.14159 → 3.14159f → a = 3.14159f
- Fortran では、5.0 と書いたら、単精度の 5.0 とみなされる

プログラム作成段階では、計算結果の表示がおかしいときがある。次の例を見てみる、

```
#include <stdio.h>

int main(int argc, char *argv[]){
 float v, m, n;
 v = 1.0; m = 0.0;
 n = v/m;
 printf("%f¥n", n);
 return 0;
}
```

1.0 を 0.0 で割った結果を、n に代入している。これ printf で表示するとどうなるか？

```
$ ./a.exe
 inf
```

もう一つのサンプル。

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

int main(int argc, char*argv[]){
double a, b;
b=-1.0;
a = sqrt(b);
printf("%f¥n", a);
return 0;
}
```

sqrt 関数は、平方根を求める関数。-1.0 の平方根を求めようとすると

```
$ ./a.exe
  nan
```

inf (無限大)や nan(not a number)という表示が出たら、上記のようなバグを疑うこと。ちなみに 0.0/0.0 でも nan がでる。

次に、文字変数のサンプルプログラムを示す。

```
1  #include␣ <stdio.h>⏎
2  ⏎
3  int␣main(int argc, char *argv[]){⏎
4  char␣a;⏎
5  a = 'A';⏎
6  printf("a = %c(ascii␣code = %d) n", a, a);⏎
7  return␣0;⏎
8  }⏎
```

var2.c

5 行目の、文字の代入式に注意。シングルクォーテーションで、文字を囲む。また、変数 a は、実は数字(-128～127)を記憶している。

練習問題 5

var2.c の 5 行目の右辺をいろいろ変えてみる。

1. a = 65;
2. a = ‘B‘;
3. a = ‘a‘ + 1;

   etc

半角文字には、ascii コード(背番号のようなもの)がつけられており、char 型は、その番号を記憶しているに過ぎない。(「背番号 1 のすごい奴」を調べてみよう。)

ここで、printf の使い方について、少し詳しく説明する。

- %c, %d, %f で、1 文字、整数、実数を表示する
- %2d や%10.2f などの指定も可能
  - %2d は、整数表示のため、あらかじめ 2 桁用意する。たとえば、%2d で 1 を表示すると ␣1 となる。
  - %10.2f は、実数を 10 桁（小数点含む）で表示そのうち小数点以下を 2 桁表示する(整数部分が 7 桁となる)。%10.5f とすると、小数点以下が 5 桁、整数部分が 4 桁。
- %e で指数形式で実数を表示することも可能
  - 0.0543: %f の場合 0.0543, %e の場合 5.43E-2

7. 計算 2

次のプログラムの結果はどうなるだろうか？

```
1  #include␣<stdio.h>⏎
2  ⏎
3  int␣main(int␣argc, char␣*argv[ ])⏎
4  {⏎
5  double␣a = 1.0;⏎
6  double␣b;⏎
7  a+1.0;⏎
8  printf(“a=%f\n”,a); ⏎
9  b=a+5.0;⏎
10 printf("a=%f, b=%f¥n",a,b);⏎
11 return␣0;⏎
12 }⏎
```

実行結果は、 a=1.000000

a=1.000000,b=6.000000

となる。9 行目で b=a+5.0 という式は、a に 5.0 足した数値を b に代入するという意味で、変数 a 自体に 5.0 がたされるわけではない。7 行目の a+1.0 という式も同じ。

```
1  #include␣<stdio.h>⏎
2  ⏎
3  int␣main(int argc, char *argv[ ])⏎
4  {⏎
5   double␣a = 1.0;  double␣b;
6   b=a+5.0;
7   printf(“a=%f, b=%f¥n",a,b);⏎
8   a=a+2.0;
9   printf(“a=%f, b=%f¥n",a,b);⏎
10  return␣0;⏎
11  }⏎
```

実行結果は、

a=1.000000,b=6.000000

a=3.000000,b=6.000000

7 行目で b=a+5.0 を実行したあと、a の値を変化させても、b の値は変化しない。

#b=a+5.0 という式で、b を定義しているわけではない。

#a=a+2.0 は、a に 2.0 を足した数値を改めて a に代入している。この手の式はよく使う。a+2 と a が等しいという意味ではない。

計算記号の省略形

- a=a+b → a += b
- a=a−b → a -= b

インクリメント、デクリメント演算子

- i++, ++i → i=i+1
- i--, --i → i=i-1
- 厳密には、++や--が変数の前にくるか後ろにくるかで、動作が変わるので注意。単独で i++や ++i とする場合は結果は同じ。

8. キーボードからの入力

まず、次のプログラムを見よう

```
#include <stdio.h>⏎
⏎
int␣main(int argc, char *argv[])⏎
const␣double␣PI = 3.14159265358979;⏎
double␣radius = 5.1;⏎
double␣area, circum;⏎
circum = 2.0*PI*radius;⏎
area = PI*radius*radius;⏎
printf(“Enshu = %f, Menseki=%f ¥n",circum,area);⏎
return␣0;⏎
}⏎
```

circ_area.c

これは、円周と円の面積を求めるプログラムである。しかし、異なる半径の円の面積・円周を求めるとき、radius を変更後、再コンパイルが必要で、実用的なプログラムとは言えない。プログラム実行後に、半径を入力するようにしたい。このような時、scanf 関数(stdio.h をインクルード)を使うとよい。下記に、scanf のサンプルプログラムを載せる。(const 修飾子をつけた変数は、内容を変更できない。変更しようとすると、コンパイル時にエラーまたは警告が出る)

```
#include␣<stdio.h>
int␣main(int␣argc, char␣*argv[]){
 int␣i;
 float␣f;
 double␣d;
 scanf("%d",&i);
 scanf("%f",&f);
 scanf("%lf",&d);
 printf("i=%d␣f=%f␣d=%f¥n",i,f,d);
 return␣0;
}
```

変数の前に&がつくことに注意
double 型は%f ではなく%lf(エルエフ)

実行すると

$ ./a.exe⏎

5⏎

1.0⏎ キーボードからの入力

2.0⏎

i=5 f=1.000000 d=2.000000

scanf("%d",&i); scanf("%f",&f);
scanf("%lf",&d); を
scanf("%d%f%lf",%i,&f,&d);と 1 行で書くことも可能。

少なくともプログラムが完成するまでは、scanf でとりこんだ数値を printf で表示するようにすること。

練習問題 6
circ_area.c に関して、半径(radius)の値をキーボードから入力するように修正せよ。

9. 条件分岐 1(if 文)

「もし(if)、天気が晴れなら釣り、雨なら映画」のように、条件により処理を変えることができる。

基本形
if(条件式){ 処理A }
else { 処理 B }
条件式が真ならば A を実行、偽ならば B を実行する。

else 以下は省略可能。 → if(条件式){ 処理Ａ }

条件式で使える比較演算子は、下記のとおりである。

	意味		意味
A > B	A は B より大きい	A <= B	A は B 以下 (≦)
A >= B	A は B 以上（≧）	A == B	A と B は等しい(※)
A < B	A は B より小さい	A != B	A と B は異なる （≠）

※A=B と書くと、B をＡに代入という意味になるので注意

if(a+2 > 100)のように、if のかっこの中に、計算式を入れてもよい。

if の使用例

if(a>b) a が b より大きいならば

if(a%2==0) a が偶数ならば。(2 で割ると余りがゼロ)

if(a%2==1) a が奇数ならば。(2 で割ると余りが 1)

練習問題 7

下記のプログラムを入力し、いろいろな整数を入力して、動作を確認せよ。また、条件式やメッセージを変えて実行せよ。

```
#include␣<stdio.h>⏎
⏎
int␣main(int␣argc, char␣*argv[ ])⏎
{⏎
 int␣i; ⏎
 printf("input a number: "); ⏎
 scanf("%d", &i); ⏎
 if(i > 5){⏎
    printf("It is bigger than 5¥n"); ⏎
 } else {⏎
    printf("It is not bigger than 5¥n"); ⏎
 } ⏎
  return 0; ⏎
} ⏎
```

if 文は、分岐を多数書くこともできる。

「もし(if)晴れたら釣り、晴れなかった場合、もし曇りならハイキング、どちらでもない場合、家で寝てる」のように、2 分岐以上も可能である

基本形

if(条件式 1){ 処理 A

} else if (条件式 2) { 処理 B

} else { 処理 C

}

else if をたくさん入れることもできる。

```
if(天気は晴れ){
        釣り
} else if (天気は曇り){
        ハイキング
} else {
        家で寝てる
}
```

多分岐のサンプルプログラム

```
#include <stdio.h>


int main(int argc, char *argv[])
{
  int a, b;
  printf("input a number(a): ");
  scanf("%d", &a);
  printf("input a number(b): ");
  scanf("%d", &b);
  if(a > b){
     printf("a is bigger than b¥n");
  } else if(a == b){
     printf("a is equal to b¥n");
  } else {
     printf("a is smaller than b¥n");
  }
}
```

comp_int.c

実行例

```
$ ./a.exe
input a number(a): 5
input a number(b): 5
a is equal to b
$ ./a.exe
input a number(a): 6
input a number(b): 3
a is bigger than b
```

下記のような条件を考える

1. もし晴れたら、あるいは雲ったら、…..
2. もし晴れて、かつ雲ひとつなかったら、…
3. もし晴れなかったら

AND や OR を使うと、簡単に表現できそうである。

条件式で使用できる論理演算子

A ｜｜ B	A あるいは B (A か B どちらか一方でも正しければ)	OR
A && B	A かつ B(A と B 両方正しければ) (&や｜ではなく&&, ｜｜なことに注意[4])	AND
!A	否定を表す(A が正しくなければ)	NOT

これらの論理演算子を使うと、先述した条件式は、次のように表現できる。

1. if(晴れ ｜｜ 曇り){……}
2. if (晴れ && 雲ひとつない){……}
3. if(!晴れ){……} また、if(a>b && b<c && a>c ..) のように 3 個以上並べることもできる。

練習問題 8 下記プログラムを実行すると、どのメッセージが表示されるか？

```
#include <stdio.h>

int main(int argc, char *argv[]){
 int a, b, c;
 a = 10; b = 8; c = 4;
 if(a>b || b < c) printf("No.1 is true¥n");
 if(a>b && b < c) printf("No.2 is true¥n");
 if(!(a>b)) printf("No.3 is true¥n");
 return 0;
}
```

条件式を()で囲むこともできる。計算の時のように、優先される 。

Supplement 1

数字の 0 は偽とみなされる。その他の数字は真とみなされる。

サンプルプログラム

[4] &や| と書くと、ビット演算子になってしまうので注意すること。コンパイルは通る可能性がある。

```
#include <stdio.h>
 int main(int argc, char *argv[ ])
{
  int i;
  for(i = -10;i<11;i++)
     if(i) printf("%d:true¥n",i);
           else printf("%d:false¥n",i);
  return 0;
}
```

```
実行結果
-10:true
-9:true
 ….
-1:true
0:false
1:true
 ….
10:true
```

for は、のちほど登場する。

Supplement 2

ある変数の範囲が ある区間の範囲に入っているか、判定する場合を考える。例えば、変数 r が、0.1 から 0.2 に入っているかどうか確かめたいとき、

if(0.1 < r <0.2) { …. }

と書きたくなると思うが、Python とちがい、これは間違いである。

```
#include <stdio.h>

int main(int argc, char *argv[ ])
{
 double r=0.0;
  if( 0.1 < r < 0.2){
   printf("0.1 < r < 0.2 is true¥n");
 }
  return 0;
}
```

上のプログラムを実行すると

```
$ ./a.exe
0.1 < r < 0.2 is true
```

となる。これは、まず 0.1 < r が評価され、r=0.0 なので、偽である。この結果は、0 と置き換わり、次に 0 < 0.2 が評価される。これは真なので、printf が実行されてしま結

果になる。コンパイルはされるが、作成した人の意図通りには動かない。正しくは、論理演算子を使い

if(0.1 < r && r < 0.2) { …}

と書く。

Supplement 3

if(i<5) {printf("i<5¥n");} を

if(i<5) printf("i<5¥n");

と書くのは間違いではない。1 文しかない時は、中括弧は書かなくてもよい。しかし、

```
if(i<5) {
  printf("I is less");
  printf("than 5¥n");
}
```

を

```
if(i<5)  printf("I is less ");
  printf("than 5¥n");
```

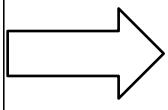
と書くのは間違い。なぜか考えてみよう。この規則は、後で出てくる for や while などでも適用される。初めのうちは、めんどうくさがらずに中括弧を書いていれば、間違いはない。

10. 条件分岐 2(switch)

if の他に、switch 文で、多分岐させることができる。基本形は、

```
switch(式){⏎
  case␣定数式1:⏎
  処理1
  break; ⏎

  case␣定数式2:⏎
  処理2
  break; ⏎
…..
  default: ⏎
  処理D⏎
  break;⏎
 }⏎
```



```
例：
switch(a){⏎
 case␣1:⏎
  printf(“a=1¥n”);⏎
 break; ⏎

 case␣2:⏎
  printf(“a=2¥n”);⏎
 break; ⏎
…..

 default:⏎
   printf(“default¥n”)⏎
 break;⏎
}⏎
```

式が定数式に等しいか判断して分岐する。式が定数式 1 と等しければ処理 1、定数式 2 に等しければ処理 2, ,,,, どれとも等しくない時は、処理Dを実行する、ただし、式および定数式は、整数や文字でなければならない(if 文では、浮動小数点なども使用できる)

練習問題 9

下記プログラムの if 文を switch で書き換えよ。

```
#include␣<stdio.h>⏎
⏎
int␣main(int␣argc, char␣*argv[ ])⏎
{⏎
int choice;⏎
⏎
printf("Input a number (1-3): ");⏎
scanf("%d",&choice);⏎
⏎
if(choice == 1){⏎
   printf("one¥n");⏎
} else if(choice == 2){⏎
   printf("two¥n");⏎
} else if(choice == 3){⏎
   printf("three¥n");⏎
} else {⏎
printf("1,2 or 3 is required¥n");⏎
}⏎
⏎
return␣0;⏎
}⏎
```

switch123.

Supplement 1

switch で break がないとどうなるのだろうか？ 次のプログラムで試してみよう。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int choice;
  for(choice=1;choice<5;choice++){
   switch(choice){
    case 1:
```

```
        printf("one¥n");
      case 2:
        printf("two¥n");
      case 3:
        printf("three¥n");
     default:
       printf("default¥n");
   }
  }
  return 0;
}
```

実行結果

```
$ ./a.exe one
two
three
default two
three
default three
default default
```

※break がないと、選択されたものから下がすべて実行されることが分かる。

11. 繰り返し実行 1(for)

シミュレ”ションでは、何らかの処理を繰り返し実行することが必ず必要になる。hello.c を考える。Hello World と 10 回表示させるには、どうしたらよいだろうか？ もっとも単純な答えは、printf を 10 個並べることであろう。しかし、 1 万回表示させたいときは、どうしたらよいだろうか？ 上の方法だと、1 万行 printf をコピーアンドペーストすることになり、まったくばかげている。このような時は、for を使うとよい。

基本形

```
for(初期処理; 継続条件; 処理){
        繰り返したい内容
}
```

1. 整数型の変数 i を用意
2. for の初期処理で、i に 0 を代入

3. 継続条件 (i<10) が成立していれば、{}で囲んだ部分、つまり printf を実行、 i≧10 なら終わる。
4. for の下の}まできたら、処理 i++ を実行。3 に戻る具体的には、下記のように書く

```
int i;  for(i=0;i<10;i++){
    printf("Hello, World¥n");
 }
```

Hello,World を 10 回表示するプログラムは、下記のようになる。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
int i;
for(i=0;i<10;i++){
 printf("i=%d:Hello, World¥n",i);
}
return 0;
}
```

hello10.c

練習問題 10

1. 100 回表示するプログラムに直してみよう
2. if を使用し、i が偶数のときだけ printf を実行するようにしてみよう

以下に for 文の例を書く

```
for(i=0;i<10;i=i+2){
  .....
}
```

i は 0 から始まり、2 ずつ繰り上がる。i が 10 以上になったら、終了

```
for(i=10; i>0; i--){
  .....
}
```

i は 10 から始まり、1 ずつ小さくなる。i が 0 になったら終了

```
for(i=0; i<10; i++){  ....
for(k=0; k<3; k++){   .....
 }
 ....
}
```

上のように for 文は、入れ子にすることができる

練習問題 11
かけ算九九の表を表示するプログラムを作成せよ（ヒント：for を入れ子にする）

for 文の途中で繰り返しを終了させたり、特定の回だけ処理を回避することもできる。

- break
  - ループから抜け出る
- continue
  - ループのその回の処理(continue より下にある部分)をスキップする break や continue は、if と一緒に用いられることが多い。break と continue の働きを図示すると

```
for(i=0;i<10;i++){
   処理 1;
   if(条件) continue;
   処理 2;
}
```

条件が成立すると、continue より下(処理 2)を行わない

```
for(i=0;i<10;i++){
   処理;
    if(条件) break;
}
```

条件成立でループ終了

break, continue のサンプルプログラムとその実行結果は、

```
#include <stdio.h>

int main(int argc, char *argv[])
{
int i;
for(i=0;i<10;i++){
   if(i==5) break;
  printf("i = %d¥n", i);
 }
 return 0;
}
```

実行結果

```
$ ./a.exe
i = 0
i = 1
i = 2
i = 3
i = 4
```

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int i;
 for(i=0;i<10;i++){
  if(i<5) continue;
   printf("i = %d¥n", i);
  }
return 0;
}
```

実行結果

```
$ ./a.exe
 i = 5
 i = 6
 i = 7
 i = 8
 i = 9
```

Advanced

for(初期処理; 継続条件; 処理){

　　繰り返したい内容

}

1. 初期条件や継続条件、処理は、「,」(カンマ)で区切ることで複数書くことができる
2. 継続条件のところで、if 文の時に勉強した論理演算子なども使用できるサンプルプログラム

```
#include <stdio.h>
 int main(int argc, char *argv[ ])
{
  int i, j;
  for(i=0, j=1; i < 10 && j < 10; i++,j++){
     printf("i=%d j=%d¥n",i,j);
 }
 return 0;
}
```

12. 繰り返し実行 2(while, do while)

while や do-while 文を使用すると、継続条件のみを指定して、繰り返し実行させることもできる。

基本形は、

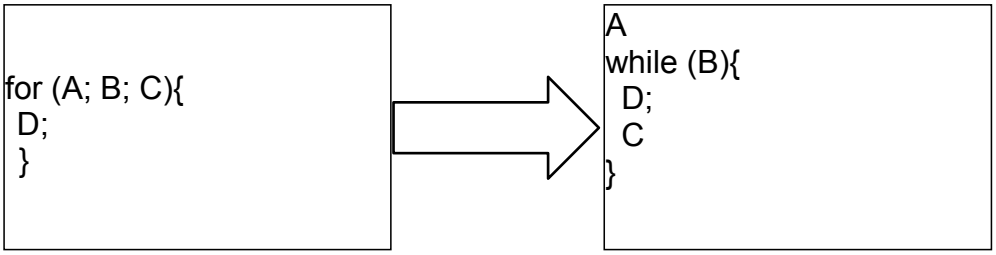
while (継続条件) {　処理 ;

}

例

```
int i=0;
 while (i<10) {
   printf("i=%d:hello, world¥n",i) ;
    i++;
}
```

i++がないとループから抜けられない。なぜか考えよう。for 文は while 文で書き直すことができる。



do - while 文は、while 文の変形版で、継続条件をあとから調べるという点が、 while 文と少し違う。

```
while (A){
  B;
}
```
A を調べる
B を実行
A を調べる
…..

```
do{
  B;
}while (A);
```
- B を実行
- A を調べる
- B を実行

…..

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 int i=10;
 while(i<10)
 {
    printf("hello¥n");
 }
 return 0;
}
```

実行結果

$ ./a.exe

```
#include <stdio.h>

int  main(int  argc,  char*argv[ ])
{
 int i=10;
 do {
   printf("hello¥n");
 } while (i<10);
  return 0;
}
```

実行結果

$ ./a.exe
Hello

練習問題 12 次の動作をするプログラムを作成せよ

1. 摂氏温度(temperature in Celsius)をキーボードから入力すると、その温度を華氏温度(temperature in Fahrenheit)に変換して表示する。
2. 1. を摂氏 300 度を上回る温度が入力されるまで繰り返す。
3. 絶対零度を下回る温度が入力された場合、何も表示しない。

- 華氏温度(Fahrenheit) = 1.8×摂氏温度(Celsius) + 32.0
- 絶対零度 -273.15℃
- Hint
  - while, if, break, scanf

13. ファイル入出力

シミュレーションでは、実行時にデータを読み込んだり、結果をファイルに書き出したりする。

リダイレクト

今まで作成したプログラムは、

- 標準出力(ディスプレイ)に何か出力し、
- 標準入力(キーボード)から何か入力した。(printf:標準出力, scanf:標準入力)

OS のリダイレクトと言う機能で、簡単にファイルへ出力、ファイルから入力に切り替えることができる。Linux でも、Windows の DOS プロンプトでも使える。

例

- $ ./a.exe > out.txt
  - 出力を out.txt にする（ディスプレイに表示せず、out.txt というファイルに出力する)。out.txt と言うファイルが存在しない場合は、新たに作成される。すでに out.txt が存在する場合は、上書きされる。
- $ ./a.exe < in.txt
  - 入力を in.txt にする(キーボードからでなく、in.txt と言うファイルから入力する)
- $ ./a.exe < in.txt > out.txt

次のプログラムと radius.txt というファイルを例に、リダイレクトの実行結果を示す。

```
#include␣<stdio.h>⏎
⏎
int␣main(int argc, char *argv[])
{⏎
const␣double␣PI = 3.14159265358979;⏎
 double␣radius, area;⏎
 scanf("%lf", &radius); ⏎
 printf("radius = %f¥n",radius); ⏎
 area = PI*radius*radius;⏎
 printf("Menseki=%f¥n", area);⏎
 return␣0;⏎
}⏎
```

```
5.0⏎
```

radius.txt

実行例 1

$./a.exe < radius.txt⏎

radius = 5.000000
Menseki=78.539816

実行例 2

$ ./a.exe > result.txt⏎
2.0⏎
$ cat␣result.txt⏎ [5]
radius = 2.000000
Menseki=12.566371

実行例 3

$ ./a.exe < radius.txt > result.txt⏎
$ cat␣result.txt
radius = 5.000000
Menseki=78.539816

実行例 4

$ ls␣-l > lslog.txt⏎
$ cat␣lslog.txt⏎
合計 3
-rw-r--r-- 1 ohno None 0 Jun 2 15:33 lslog.txt
-rw-r--r-- 1 ohno None 199 Apr 15 13:21 prac1.c
-rw-r--r-- 1 ohno None 344 Apr 15 13:28 prac2.c
-rw-r--r-- 1 ohno None 705 Apr 15 16:30 prac3.c

リダイレクトで、出力を既に存在しているファイルにすると、上書きされるが、「>>」を使ってリダイレクトすると、ファイルの末尾に結果が追加される実行例

$ ls␣-l > lslog.txt⏎
$ ls␣-l >> lslog.txt⏎
$ cat␣lslog.txt ⏎
合計 3
-rw-r--r-- 1 ohno None 0 Jun 2 15:39 lslog.txt
-rw-r--r-- 1 ohno None 199 Apr 15 13:21 prac1.c
-rw-r--r-- 1 ohno None 344 Apr 15 13:28 prac2.c

---

[5] cat コマンドで、テキストファイルの内容を表示できる。

-rw-r--r-- 1 ohno None 705 Apr 15 16:30 prac3.c
合計 4
-rw-r--r-- 1 ohno None 203 Jun 2 15:39 lslog.txt
-rw-r--r-- 1 ohno None 199 Apr 15 13:21 prac1.c
-rw-r--r-- 1 ohno None 344 Apr 15 13:28 prac2.c
-rw-r--r-- 1 ohno None 705 Apr 15 16:30 prac3.c
追加書き込みにしたため、2 度の ls の結果が記録されている。

Advanced

>&により、標準エラー出力もリダイレクトできる。

実行例

$ ls⏎

lslog.txt prac1.c prac2.c prac3.c

$ ls␣ohno⏎

ls: ohno にアクセスできません: No such file or directory ←エラーを起こした

$ ls␣ohno > error.txt⏎

ls: ohno にアクセスできません: No such file or directory ←標準エラー出力をリダイレクトしても、画面に表示される

$ ls␣ohno >& error.txt⏎

$ cat␣error.txt⏎

ls: ohno にアクセスできません: No such file or directory

(>&で標準エラー出力もリダイレクトできることを確認できた)

ファイル入出力

C 言語には、ファイルを指定して、データを読み込んだり、書き出したりする方法がある。

ファイル指定の決まり文句は、

1. FILE *fp;
2. fp = fopen(“data.txt“, “w“);
3. fclose(fp);

1.ファイルを開くための変数の宣言

2. data.txt というファイルをオープンし、変数 fp に関連付ける。ファイル名のところには、ディレクトリもかける。例:fopen(“../data/data.txt“, “w“)。“w“の意味するところは、書き込み(write)であるが、w の他に a や r の場合もある。

w ： 書き込み(write)

ファイルが存在しない場合：生成される

ファイルが存在する場合：上書きされる

a ：追加書き込み(append)

r ：読み出し(read)

3. ファイルを閉じる

ファイルがオープンされていると、printf, scanf の仲間の fprintf や facanf で、オープンされているファイルからデータを書き出したり、読み込むことができる。fprintf、fscanf の使い方は、printf, scanf とほぼ同じ。

次のプログラムを見てみよう。

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5   FILE␣*fp; int i;
6   fp = fopen(“hello.txt”, “w”);
7   for(i=0;i<10;i++){
8    if(i%2==0) fprintf(fp, “i=%d:Hello, World¥n”, i);
9   }
10  fclose(fp);
11  return 0;
   }
```

7 行目で、hello.txt を書き出し専用にオープンし、fp に関連付ける。8 行目で hello.txt に対して、書き出しを行う。fp でファイルを区別している。printf で画面に文字を表示するように、hello.txt に文字を出力する次に、読み込みの例を示したプログラムを見てみよう。

```
 1 #include <stdio.h>
 2
 3 int main(int argc, char *argv[])
 4 {
 5  FILE *fp;
 6  int i;
 7  double f;
 8 fp=fopen("data.txt", "r");
 9 for(i=0;i<5;i++){
10     fscanf(fp, "%lf", &f);
11     printf("%d: %f¥n", f);
12  }
13 fclose(fp);
14 return 0;
15 }
```

8 行目で data.txt を読み出し用にオープンし、fp に関連付ける。data.txt から、実数の数値を読み込む。scanf でキーボードから数値を入力するように、data.txt から数値を入力する

上の例では、ファイルは 1 つしかオープンしていないが、複数のファイルを同時にオープンすることもできる。その場合、FILE 型の変数でファイルを区別する。

練習問題 13

次の動作をするプログラムを作成せよ

1. 摂氏温度(temperature in Celsius)を cel.txt から読み込み、その温度を華氏温度(temperature in Fahrenheit)に変換して fah.txt に書き出す。
2. 1. を 300 度を上回る温度が入力されるまで繰り返す。

cel.txt の内容は、

```
50.5⏎
23.4⏎
65.1⏎
23.1⏎
350.0⏎
```

Supplement 1

fprintf, fscanf の補足

- ファイルポインタ(FILE *fp の fp)の代わりに、以下も使うことができる
- stdout
  - 標準出力(ディスプレイ)
- stdin
  - 標準入力(キーボード)
- stderr
  - 標準エラー出力(ディスプレイ)

```
#include <stdio.h>

int main(int argc,char *argv[])
{
double c,f;
  fprintf(stdout, "Enter a temperature (Celsius): ");
  fscanf(stdin, "%lf", &c);
  f=1.8*c+32.0;
  fprintf(stdout, "%f (Celsius) is %f (Fahrenheit)¥n",c,f);
 return 0;
}
```

このプログラムは、キーボードから数値（摂氏温度）を入力し、ディスプレイに華氏温度に換算した数値を表示する。

Supplement 2

fopen は、ファイルオープンに失敗すると、NULL を返すので、ファイルオープンは、次のようにするとよい。

```
if((fp=fopen("data.txt", "r"))==NULL){
 fprintf(stderr, "file open error¥n");
 exit(1);
}
….
fclose(fp);
```

これは、次と同じ意味である。

```
fp=fopen("data.txt", "r");
 if(fp==NULL){
 fprintf(stderr, "file open error¥n");
 exit(1);
}
```

もし、ファイルオープンに失敗すると、fp に NULL が代入される。そこで、if 文により fp が NULL かどうか調べて、もし NULL(失敗した)であれば、「file open error」というメッセージを標準エラー出力に出力して、プログラムを終了する(exit 関数が呼ばれると、そこでプログラムが終了する。exit 関数を使うには、stdlib.h をインクルードすることが必要)

14. 配列

int i や double cel と宣言したのでは、変数は一つしか用意されない。同じ変数を複数用意したい場合がある。たとえば、クラスの生徒（40 人）のテストの点数を記憶する変数がほしい。

int score0, score1, ….. ,score39;

学年全体(360 人)なら？ この方法で宣言するのは非効率的である。

int score[40];

こう書くと、一気に int 型の変数が 40 個できる。int score[360]とすれば、一気に 360 個確保できる。

配列

int a[5]を考えてみる。このような宣言をすると、指定したデータ型（この場合 int）の変数が、[ ]の中の数字で指定した数(この場合 5 個)だけ、用意される。また、メモリの中では連続している。イメージは、

a[0]	a[1]	a[2]	a[3]	a[4]

使い方：

a[0]～a[4] を普通の int 型の変数として、使うことができる。(1 からでなく 0 からということに注意) たとえば、

a[0] = 80; a[4] = 50;

printf("a[1]=%d¥n",a[1]);

普通の変数と同様に、宣言と同時に値を代入することもできる。

int a[5] = {1, 2, 3, 4, 5};

double b[3] ={1.2, 4.55, 6.332};

など、中かっこ {} を使い、カンマ , で区切る。

例

int a[5] = {1,2,3,4,5};

double b[3] ={1.2, 4.55, 6.332};

printf("a[2]=%d, b[1]=%f¥n", a[2], b[1]);
→ a[2]=3, b[1]=4.550000

文字配列
代入したい文字列の数+1 以上のサイズにする
TEST と言う 4 文字の文字列を char 型の配列に代入したい時は、4 文字+1 の 5 文字分のサイズの配列を宣言し、TEST に続き、¥0 を代入する例
char s[5]={'T', 'E', 'S', 'T', '¥0'}
※ ¥0 は、文字列の終端を表す文字配列のみ、特別な代入の仕方がある
char s[5] = "TEST"
(自動的に¥0 が最後に付加される)
また、
strcpy という関数（string.h のインクルードが必要）を使うと、容易に文字列の代入ができる(終端に¥0 も代入される) 例
char s[5];
strcpy(s, "TEST");
※s[0] = 'T'; s[1]= 'E'; s[2]= 'S'; s[3]= 'T'; s[4]= '¥0';と同じ

fprintf, printf, scanf, fscanf などを使い文字配列を表示したりするときは、%s を使う(1 文字の時は%c)
char carray[64];
scanf("%s", carray)
printf("%s¥n", carray);

scanf の 2 つ目の引数に、&がついていないことに注意。変数の前に&をつけると、変数のメモリ中のアドレスをあらわす。配列の名前は、配列の先頭要素のアドレスを表すので、&をつける必要はない。つけたい場合は、&carray[0]とする

練習問題 14
下記の動作をするプログラムを作成せよ
ファイル score.txt から、テストの点数(整数型)を 10 人分(学生番号 1～10 の順に並んでいる)読み込み平均点(浮動小数点型)を計算し、平均点と平均点以上の点数を取っている学生番号を pass.txt に書き出す。
score.txt の中身

75↵
45↵
63↵
55↵
43↵
98↵
87↵
75↵
68↵
78↵

15. 多次元配列

2 次元配列 int a[4][5]を宣言すると、下のようなイメージで、変数が用意される

a[0][0]	**a[0][1]**	**a[0][2]**	**a[0][3]**	**a[0][4]**
a[1][0]	**a[1][1]**	**a[1][2]**	**a[1][3]**	**a[1][4]**
a[2][0]	**a[2][1]**	**a[2][2]**	**a[2][3]**	**a[2][4]**
a[3][0]	**a[3][1]**	**a[3][2]**	**a[3][3]**	**a[3][4]**

使い方は、1 次元のときと同じ。例：a[0][0]=1; m=a[2][1]。3 次元、4 次元配列など、3 次元以上の配列も作れる

多次元配列も、宣言と同時に値を代入するこができる。例

int a[2][3] = { 1, 2, 3, 4, 5, 6}; ⇒ a[0][0] = 1; a[0][1] = 2; a[0][2] = 3; a[1][0] = 4; a[1][1] = 5; a[1][2] = 6; //後ろの添え字が優先される

{ }を入れ子にするとわかりやすい

int a[2][3] = { {1, 2, 3}, {4, 5, 6} };

- どちらでもよいが、後者の方がわかりやすい
- 3 次元以上では、{}を入れ子にしても、わかりにくい

変数を用いて、配列を宣言することはできない。

```
int NUM = 10;
int score[NUM];   /* 間違い */
```

ただし、const 修飾がされていれば可能(例外あり)

```
const int NUM = 10;
int score[NUM]; /* int score[10]と同じ*/
```

(配列の要素数に限らないが)生の数字をソースコードに入れる(ハードコーディング)のではなく、わかりやすい名前(変数が保持する値の意味)を付けた変数など使うと間違いが少なくなる。例えば、練習問題 14 の解答例を下記のようにすると、意味が分かりやすくなる。

```
#include <stdio.h>

int main(int argc, char *argv[ ]){
 const int NUM_STUDENTS = 10;
 int i;
 int score[NUM_STUDENTS]; double ave;
 FILE *fp;

 fp = fopen("score.txt", "r");
 ave = 0.0;

 for(i=0;i< NUM_STUDENTS;i++){
   fscanf(fp, "%d", &score[i]);
   ave += score[i];
 }

 fclose(fp);
 ave /= NUM_STUDENTS;

 fp = fopen("pass.txt", "w");
 fprintf(fp, "average = %f¥n",ave);
/* printf("average = %f¥n",ave); */

 for(i=0;i< NUM_STUDENTS;i++){
   if(score[i] >= ave) fprintf(fp, "%d¥n", i+1);
 /* if(score[i] >= ave) printf("*"); else printf(" ");
     printf("%2d:%d¥n",i+1,score[i]);*/
 }
 fclose(fp);
 return 0;
}
```

pass.c

こうすると、生徒 20 人バージョンも、一カ所直すだけですぐに作成することができる。1 次元配列の配列名は、配列の先頭アドレスを表した。

Example：char carray[64] carray == &carray[0]

2 次元の場合

int a[2][3];

a[0] ⇒ &a[0][0], a[1] ⇒ &a[1][0]

を意味する。

練習問題 15

scoredata.txt は、 1 クラス(10 人)ぶんの名前、英語、数学、科学の点数が下記の要領で記録されている

生徒の名前 [Tab] 英語 [Tab] 数学 [Tab] 科学⏎

scoredata.txt を読み込み、英語、数学、科学すべての点数(整数)が、それぞれの教科のクラスの平均点以上獲得した生徒の名前を result.txt に出力せよ。

```
Seiko   85      96      80⏎
Akina   77      45      86⏎
Noriko  88      55      85⏎
Ken     75      75      78⏎
Masashi         86      90      90⏎
Nobunaga        75      45      58⏎
Cha     74      87      65⏎
Tomoyo          45      36      45⏎
John    74      45      87⏎
George  84      45      94⏎
```

scoredata.txt

Supplement 1:sizeof 演算子

変数のサイズ（何バイトなのか？）や、配列の要素数を確認できる

```
int a[4];
printf("sizeof(char) = %d¥n",sizeof(char)); /* 1 */
printf("sizeof(int) = %d¥n",sizeof(int)); /* 4：処理系により異なる*/
printf("sizeof(float) = %d¥n",sizeof(float)); /* 4：処理系により異なる */
printf("sizeof(double) = %d¥n",sizeof(double)); /* 8：処理系により異なる */
printf("sizeof(a) = %d¥n",sizeof(a)); /* 16(4×4)：処理系により異なる */
printf("num of elements of a =%d¥n",sizeof(a)/sizeof(int)); /* 4 */
```

Supplement 2

C 言語では、ハードコーディングを避けるために、const をつけた変数の他に、#define を使う場合も多い。

```
#include <stdio.h>

#define NUM_STUDENTS 10

int main(int argc, char *argv[])
{
 int i;
 int score[NUM_STUDENTS];
  double ave;
  FILE *fp;
  fp = fopen("score.txt", "r");
  ave = 0.0;
  for(i=0;i< NUM_STUDENTS;i++){
   fscanf(fp, "%d", &score[i]);   ave += score[i];
  }
  fclose(fp);
 ave /= NUM_STUDENTS;
 fp = fopen("pass.txt", "w");
 fprintf(fp, "average = %f¥n",ave);
  for(i=0;i< NUM_STUDENTS;i++){
  if(score[i] >= ave) fprintf(fp, "%d¥n", i+1);
 }
  fclose(fp);
  return 0;
}
```

上の例では、コンパイル時に（厳密にはコンパイル前に）NUM_STUDENTS がすべて 10 に置き換わる。NUM_STUDENTS と 10 の間に = がないことや、最後にセミコロンがないことに注意。

16. 関数

プログラムを作る時、まとまった処理ごとに関数を作って（特に何度も同じ処理が必要な場合）、それを呼び出すようにすると、間違いが少ないし、すっきりする。main 関数が 1 万行あるようなプログラムは作らないように！

関数の書き方は、基本的に main 関数と同じ。ただし、main 関数のように自動的に実行されるわけではなく、明示的に呼び出さないと実行されない。

例 1

仮引数、戻り値、いずれもない場合。

```
#include <stdio.h>

void hello_world2( )
{
  int i;
  for(i=0;i<2;i++){
    printf("Hello, World¥n");
  }
  return;
}

int main(int argc, char *argv[ ])
{
  int i;
  for(i=0;i<10;i++){
    hello_world2( );
  }
return 0;
}
```

Main 関数内の下線のところで、hello_world2 が呼び出される。 関数の名前の後に()がついていることに注意。

hello_wrold2 を実行した後、元の場所に戻る

(正確には「;」のうしろ)

main 関数と hello_world2 関数内に、同じ名前の変数 i があるが、別々の関数内にある変数は互いに一切影響を及ぼさない。(名前と型は同じだが、全くの別物)

関数の定義法（戻り値、引数なしの場合）

void 関数の名前( ) {

処理

}

関数のネーミングは、変数の時と同じ。ただし、C が用意している標準の関数と同じ名前は使わない方が良い

例 2

仮引数あり、戻り値なしの場合

```
#include <stdio.h>

void star(int␣num)
{
 int i;
 for(i=0;i<num;i++){
   putchar('*');
 }
 putchar('¥n');
 return;
}

int main(int argc, char *argv[])
{
 int i;
 for(i=0;i<10;i++){
   star(i);
 }
 return 0;
}
```

関数の仮引数は、型と名前をかく
変数の宣言と同じ 。複数の引数を取る場合は、仮引数を「,」カンマ で区切って書く。
例
void star(int num, char c)

i=3
のとき star(i) は star(3) として呼ばれ値がコピーされる。star 関数は、 num に 3 が代入された状態で、実行される。
また、関数を呼び出すときは、引数の型だけが 一致していればよい。
main 内でわざわざ num という int 型の変数を使用する必要はない。
例: int i=1; star(i)

整数を直接書いてもよい。
例：star(0)
なお、仮引数 num であるが、 star 関数以外の関数で num と言う変数があっても、無関係である。

ちなみに、putchar 関数は、1 文字出力する関数で、stdio.h のインクルードが必要。

例 3

引数に渡した変数の内容

```
#include <stdio.h>

void star(int num)
{
 int i;
 for(i=0;i<num;i++){
  putchar('*');
 }
 putchar('¥n');
 num = 100;
 return;
}

int main(int argc, char *argv[ ]){
 int i=5;
 star(i);
 printf("i=%d ¥n", i);
 return 0;
}
```

実行結果

```
$ ./a.exe
*****
i=5
$
```

呼び出した側の関数の変数の内容（この場合 i ）を、呼び出された関数で変更することはできない。（ num は 100 に変わっている。） 。

※言語によっては変わるものもある

繰り返しになるが、star の仮引数は num であるが、呼び出す側では、num という名前の変数を用意する必要はない。 **型さえ 一致していればよい**

例 4

仮引数、戻り値、いずれもある場合

```
#include <stdio.h>

int sum(int a, int b)
{
 int c;
 c = a + b;
 return␣c;
}

int main(int argc, char *argv[])
{
 int i, j, s;
 printf("Enter two integers: ");
 scanf("%d%d",&i,&j);
 s = sum(i, j);
 printf("%d + %d = %d ¥n", i, j, s);
 return 0;
}
```

戻り値

関数は、一つだけ値を返すことができる。戻り値がある関数をつくるときは、関数の名前の前に、 void にかわり、**戻り値の型**をかく。（点線で囲んでいる）

また、返す値は、

return␣ に続き、変数なり数値などを直接書く。（実線で囲んでいる）

例 5

グローバル変数

```
#include <stdio.h>

char a;

void star(int num)
{
 int i;
 for(i=0;i<num;i++){
  putchar(a);
 }
 putchar(¥n');
 return;
}

void hello()
{
 int a=10;
 printf("a = %d¥n",a);
}

int main(int argc, char *argv[ ])
{
 a = '*';
 star(10);
 hello();
 return 0;
}
```

関数内で宣言された変数は局所変数(local variable)と呼ばれ、他の関数からはアクセスできない。

関数外で宣言された変数は大域変数(global variable)と呼ばれ、関数間で共通に使用できる。（点線で囲んでいる）

局所変数にグローバル変数と同じ名前の変数がある場合は、局所変数が優先される。実線で囲んでいる）

戻り値がない場合、return は省略可能

実行結果

$ ./a.exe **********

a = 10

例 6

静的変数

```
#include <stdio.h>

void count()
{
 static int i = 0;
 i++;
 printf("count has been called for %d times¥n", i);
}

int main(int argc, char *argv[])
{
 int i;
 for(i=0;i<10;i++){
   count();
 }

 return 0;
}
```

static をつけた変数は、プログラムが終了するまで値が保持される。

宣言と同時の初期化は、初めに呼び出されたとき、1 回だけ実行される。

宣言と同時に行わない初期化

static int i;

i=0;

こう書くと毎度初期化される。

実行結果

```
$ ./a.exe
count has been called for 1 times
count has been called for 2 times
count has been called for 3 times
count has been called for 4 times
count has been called for 5 times
count has been called for 6 times
count has been called for 7 times
count has been called for 8 times
count has been called for 9 times
count has been called for 10 times
```

例 8

プロトタイプ宣言

```
#include <stdio.h>

int main(int artc, char *argv[]){
 int i;
 double d;
 i=5;
 d = half(i);
 printf("%d/2 = %f¥n", i, d);
 return 0;
}

double half(int i){
 double d;
 d = i/2.0;
 return d;
}
```

```
$ gcc example8.c

func6.c:12:8: error: conflicting types for
'half'

func6.c:7:6: note: previous implicit

declaration of 'half' was here
```

関数は、使用する前に定義を書いておかなければならない。定義が出てくる前に呼ぶと、コンパイラは戻り値を int 型とみなすので、実際の戻り値が int でないとコンパイラによってはコンパイル時にエラーが出る。 コンパイルが通っても、動作はおかしなものになる場合がある

プロトタイプ宣言（**Function Prototype**）

プロトタイプ宣言を行うことで、定義の前に関数を呼ぶことができる。

```
#include <stdio.h>

double half(int i);

int main(int artc, char *argv[]){
 int i;
 double d;
 i=5;
 d = half(i);
 printf("%d/2 = %f¥n", i, d);
 return 0;
}

double half(int i){
 double d;
 d = i/2.0;
 return d;
}
```

プロトタイプ宣言（Function Prototype）

実線で囲んだ部分がプロトタイプ宣言。関数の戻り値、関数の名前、仮引数をかく。

下線部のところ(+ セミコロン)をかくと思えばよい。

コンパイルは通る。

C 言語の書籍では、標準関数の説明に、プロトタイプを含めた下記の●の様なものが書かれている。

exit

- インクルードファイル名 stdlib.h
- void exit(int status)

  - 戻り値なし、引数は 1 つで int 型。
  - stdlib.h をインクルードする必要あり

fputs

- インクルードファイル名 stdio.h
- int fputs(const char *string, FILE *stream)

  - 戻り値 int 型、引数は 2 つ。
  - stdio.h をインクルードする必要あり。
  - （const char *string は文字列か、文字配列を渡す）

….

ここまでくれば、自分で有用な関数を探して使用することができると思う。

関数を使って、円の面積を求めるプログラムを書き換えてみる。

```
#include <stdio.h>

double calc_area(double r)
{
 const double PI = 3.141592653589793;
 double area;
 area = PI*r*r;
 return area;
}

int main(int argc, char *argv[])
{
 double rad, area;
 while(1){
  printf("Enter a radius: ");
  scanf("%lf", &rad);
  if(rad < 0.0) break;
  area = calc_area(rad);
  printf("Area = %f¥n", area);
 }
 return 0;
}
```

練習問題 16

1. 練習問題 13 の切歯温度を可視温度に変換する部分を関数化せよ
2. 練習問題 14 の平均点を求める部分を関数化せよ

Supplement 1：return について

関数は、最初に return に達した時点で、たとえその下にまだ処理が残されていたとしても、終了する。

```
void star(int num){
 int i;
 return;
 for(i=0;i<num;i++){
   putchar('*');
 }
  putchar('¥n');
  num = 100;
   return;
}
```

上記関数は、整数型の変数を 1 つ宣言しただけで終了する。

Supplement 2：配列の引数について

普通の変数は引数の内容を変えることはできないが、配列の内容は変えることができる。

これは、配列を呼び出す関数に渡す時は、全ての要素を渡すのではなく、先頭アドレス、型などの情報を渡す。（すべての要素を渡すのは非効率である）

なので、配列に限っては、呼び出した側と同じ配列変数のアドレスを操作することになり、内容が変わってしまう。

配列でなくとも、変数のアドレスを渡すようにすれば、呼び出した関数から、変数の内容を変えることができる。（例：scanf）

配列の要素を渡したときは、当然内容は変えられない。（例：func(s[3][2]); ）詳しくは、C 言語や C++言語の入門書の、「ポインタ」の項を参照すること。

```
#include <stdio.h>

void change(int s[5])
{
  int i;  for(i=0;i<5;i++){
   s[i] = -s[i];
 }
}

int main(int argc, char *argv[])
{
 int a[5] ={1,2,3,4,5};
 change(a);  printf("%d %d %d %d %d¥n",
  a[0],a[1],a[2],a[3],a[4]);
  return 0;
}
```

実行結果

```
$ ./a.exe
-1 -2 -3 -4 -5
```

17. 構造体

データにまとまりをもたせたいことがある。例えば、練習問題 15 の学生のデータ（名前、英語の点、数学の点、科学の点）など。別々に変数を宣言するより、まとめた方がすっきりする。

前回の例だと、

```
const int N_STUDENTS = 100; /*100 人*/ ……..
char name[NUM][64];
int score_eng[NUM];
int score_math[NUM];
int score_sci[NUM];
```

問題はないが、、、、

このようなデータをまとめるために、構造体と呼ばれるものが C 言語には用意されている。前の例は、構造体を使うと

```
struct student{
        char name[64];
        int socre_eng;
        int score_math;
```

```
    int score_sci;
};
```

として、student という自分だけのデータ型をつくることができる。(上の部分は、構造体 student の定義) student 型を定義したら、

```
struct student data[NUM];
```

というようにして、変数を用意できる。(配列でなくとも、もちろんよい) 使い方は、”.”(ピリオド)で、中の変数にアクセスするということ以外は、普通の変数や、配列とおなじ。

```
data[0].score_eng = 75;
```

や

```
int a = data[2].score_mat;
printf(“%s¥n”,data[30].name);
```

など

練習問題 17

練習問題 15 のプログラムを構造体を使いもう一度作成せよ。

18. バイナリモードでのファイルの読み書き

ここまでは、fprintf で、テキスト形式で結果を目に見える形で保存してきた。

C 言語には、配列などの内容を内部表現のまま保存したり、読み込んだりする関数も用意されている。下記の関数を使用する

- size_t fwrite(const void *p, size_t size, size_t n, FILE *stream)
  - –p:書き込むデータの先頭アドレス
  - –size: 書き込むデータ一つ分の大きさ。sizeof 演算子を使うとよい
  - –n: 書き込むデータの数

- size_t fread(void *p, size_t size, size_t n, FILE *stream)
  - –p:書き込むデータを格納するメモリ領域の先頭アドレス
  - –size: 読み込むデータ一つ分の大きさ。sizeof 演算子を使うとよい
  - –n: 読み込むデータの数

fwrite, fread とも、stdio.h のインクルードが必要。また size_t という型は、今は 0 以上の整数の型と覚えておいても差し支えない(と思う) fopen でファイルをオープンするとき、

- 書き込むとき “w” → “wb”
- 読み込むとき “r” → “rb”

とする。b をつけないと、Windows の Visual Studio などで、問題が起こる場合がある。

書き込みの例

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

int main(int argc, char *argv[]){
 int i;
 int a[100];
 FILE *fp;
 for(i=0;i<100;i++){
  a[i] = rand();
 }
 fp = fopen("rand.dat", "wb");
  fwrite(a, sizeof(int), 100, fp);
  fclose(fp);

 return 0;
}
```

書き込み：

一様乱数を 100 個発生させ、それを rand.dat として保存する。

バイナリモード

配列 a の先頭から(1 次元配列の名前は、先頭要素のアドレスを意味)

int 型の大きさのデータを(4 とかの生の数字を入れないこと)

100 個

fp で指定されたファイルに書き込む

読み込みの例

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

int main(int argc, char *argv[])
{
  int i, r;
  int a[100];
  FILE *fp;
 fp = fopen("rand.dat", "rb");
  fread(a, sizeof(int), 100, fp);
 fclose(fp);
  for(i=0;i<100;i++){
     r = rand();
    printf("r=%d, a[%d]=%d: ", r, i, a[i]);
   if(r==a[i]) printf("same¥n");
    else  printf("not same¥n");
 }
 return 0;
}
```

読み込み：

rand.dat を読み込み、一様乱数を 100 個発生させ比較する（同じはず）

バイナリモード

配列 a の先頭から(1 次元配列の名前は、先頭要素のアドレスを意味)

int 型の大きさのデータを

100 個

fp で指定されたファイルに読み込む

19. これから

ここまでの知識で、だいぶ実用的なプログラムが作れるはずである。解こうとする問題を、まずはどのような流れで解くか考え、小さな処理に分割して、プログラムを組み立てる訓練をすること。他人が書いたプログラムを読んだり、実際に自分で何か作ったりするとよい。

さらに C 言語の学習を進めるのであれば、

- ポインタ
- 動的メモリ確保
- プログラムを複数のファイルに分ける

を勉強して、その後 C++に進むとのがおすすめ。C++コンパイラも cygwin で使用可能である(g++)。

Appendix 1：**scanf** について

scanf, fscanf 関数で複数の数値や文字列を読み取る場合、デフォルトでは、スペース、タブ、改行で区切る。この性質からスペースを含んだ文字列を読み取ることはできない。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 char s[80];
  scanf("%s",s);
  printf("you've entered [%s].¥n",s);
  return 0;
}
```

実行例

```
$ ./a.exe
SimulationStduies⏎
you've entered
[SimulationStduies].

$ ./a.exe
Simulation␣Studies⏎
you've entered [Simulation].
```

スペースを入れると、区切られていると認識される

次の例は、数値を入力するとき、1 個ずつ改行しなければいけないイメージがある

```
#include <stdio.h>
int main(int argc, char *argv[]){
 int i, j, k;
 scanf("%d",&i);
 scanf("%d",&j);
 scanf("%d",&k);
 printf("you've entered %d, %d and %d¥n.",i,j,k);
 return 0;
}
```

次の例は、3 個並べなければいけないイメージがある。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i, j, k;
 scanf("%d%d%d",&i,&j,&k);
 printf("you've entered %d, %d and %d.¥n",i,j,k);
  return 0;
}
```

しかし、

実行例 1	実行例 2
$ ./a.exe⏎ 1⏎ 2⏎ 3⏎ you've entered 1, 2 and 3	$ ./a.exe⏎ 1␣2␣3⏎ you've entered 1, 2 and 3
	実行例 3 $ ./a.exe⏎ 1␣2⏎ 3⏎ you've entered 1, 2 and 3.

実行結果はどちらでも同じである。改行もスペースも、単なる区切り文字に過ぎない。

Appendix 2: バッファリング

標準入出力やファイル入出力では、バッファリングが行われる。

キーボードから入力された文字列は、まずメモリ内に確保されているバッファ領域に入り、その後 scanf などで使用される。バッファ領域に入ったデータが実際に処理されるタイミング は

- バッファ領域が満タンになったとき
- 改行したとき

前のページの結果をバッファリングとあわせて考えてみよう 。

キーボードからの入力やディスプレイへの出力は、非常に遅いので、このような機能が付加されている。まず、出力のほうを考える。

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

int main(int argc, char *argv[])
{
 printf("Hello, ");
 sleep(1); /*n 秒(カッコ内の数値)休止する関数。stdlib.h をインクルードする必要がある
*/
 printf("World¥n");

 return 0;
}
```

実行すると、Hello,と表示され、1 秒後に World と表示されるような気がする。しかし、実行して 1 秒後に Hello,World と一気に表示される。最初の Hello,はバッファ領域に入りすぐには表示されず、World の後の改行で、ようやく処理に回され、 Hello,World すべてが表示される。

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

int main(int argc, char *argv[ ])
{
  printf("Hello, ");
  fflush(stdout);
  sleep(1);
  printf("World¥n");
  return 0;
}
```

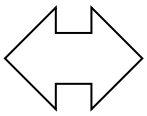
ただし、int fflush(FILE *): バッファ内の中身を強制的に出力させる関数。標準出力(stdout)だけでなく、普通のファイルに対しても使える。ファイルの場合、 fclose すれば、バッファの内容はフラッシュされるので、クローズしても出力されていないデータがあるのでは？という心配は無用。

(こんどこそ)実行すると、まず Hello,と表示され、1 秒後に World と表示される。Hello,を fflush で強制的に出力している。また、標準エラー出力はバッファリングされないようなので、fprintf(stderr,"Hello,");とすれば、fflush はいらない。

次に入力を考える。キーボードから入力した文字列で、scanf で使用されなかった文字列は、バッファ領域に残る。この性質があるため、 scanf や fscanf は、少し危険な関数だったりする(もちろん赤い少佐が言うように、使い方を誤らなければどうということはない)。

```
#include <stdio.h>

int main(int argc, char *argv[]){
 int i = 0;
 while(1){
  scanf("%d",&i);
  printf("you've entered %d. n",i);
  if(i<0) break;
 }
 return 0;
}
```



```
実行例
$ ./a.exe
1
you've entered 1.
2
you've entered 2.
-
1
you've entered
1.
$
```

しかし、間違ってAを入力すると、Aは整数ではないので、scanf(“%d”,&i)の実行に失敗する(おそらくAはバッファに残る)。失敗後 printf,if を通過し(i にはすでに 2 が入っている)、scanf (“%d”,&i)に戻るが、バッファに”A”が残っているためなのか、また scanf に失敗する。そして printf, if….というように、永久にループする。
これ以外にも、おかしくなる例は結構ある（改行がバッファに残っていることで、プログラムによっては妙な動作をする、など)。

無限ループに陥る実行例

$ ./a.exe
1　　⏎
you've entered 1.
2　　⏎
you've entered 2.
A⏎
you've entered 2.
you've entered 2.
you've entered 2.
…(無限ループ) /*ctrl+c で止めよう*/

安全対策の一つとしては、fgets と sscanf を使うことが考えられる。

- char *fgets(char *buf, int size, FILE *s)
  - ファイルから、1 行あるいは size−1 文字読み込み、末尾に¥0 をつけて buf に格納する
- sscanf(const char *buf, ….)
  - scanf, fscanf の仲間
  - 標準入力やファイルからでなく、buf で指定された文字配列から数値などを読み取る

いずれも、stdio.h のインクルードが必要。これらを使って書き直したプログラムは、

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
 char buf[256];
 int i = 0;
 while(1){
   fgets(buf, sizeof(buf), stdin);
   sscanf(buf, "%d",&i);
   printf("you've entered %d. n",i);
   if(i<0) break;
 }
 return 0;
}
```

```
実行例
$ ./a.exe
1
you've entered 1.
2
you've entered 2.
A
you've entered 2.
-1
you've entered -1.
```

fgets 関数で、1 行ずつ文字配列 buf に入力した文字列を格納しているので、バッファには何も残らない。ゆえに A を入力しても無限ループにはならない

Appendix 3:改行コード

改行コード（Newline character）	
Windows	CR+LF
昔の Mac	CR
Linux (Cygwin 含む)，Mac(MAC OS X)	LF

改行コードが違うと、テキストエディタで開いたとき、改行されない場合がある。

例：Linux で作ったテキストを Windows のメモ帳で開く対処例：

・改行コードを変換する

・いろいろな改行コードに対応したテキストエディタを使用する

（サクラエディタ、秀丸エディタ(有償)）

Appendix 4:main 関数の引数と戻り値

main 関数の引数について

- int main(int argc, char *argv[])
- プログラム実行時のコマンドライン引数の情報が入る
  - argc には 引数の数(int 型、プログラムファイルの名前含む)
  - argv には、実際に入力された文字列(char 型)

Exmaple：$ ./a.exe␣first␣second␣third⏎
argc = 4
argv[0][0～3]: ./a (argv[0]の中身は、処理系により違いがある)
argv[1][0～5]: first¥0
argv[2][0～6]: second¥0
argv[3][0～5]: third¥0

下記プログラムで、argc と argv の内容を確認することができる。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
 printf("argc = %d¥n", argc);
 for(i=0;i<argc;i++){
    printf("%d:%s¥n", i, argv[i]);
 }
 return 0;
}
```

コマンドライン上で
$ ./a.exe␣15
としても、”15”(argv[1][0”2])は、あくまで文字列である。整数として扱いたい場合は、atoi 関数で整数に変換する必要がある

```
 int a;
 a=atoi(argv[1]); /*to integer*/
```

実数の場合は(例：$ ./a.exe␣15.256⏎)、atof[6]関数を使う

```
 double b;
  b=atof(argv[1]);/*to floating point number*/
```

（atoi, atof 関数を使用するには stdlib.h をインクルードする必要がある）

では、戻り値はどうか？
…. int main(int argc, char *argv[]){ ….
return 0;

[6] sscanf を使う方法も考えられる

}

上の return 0 の 0 はどこへ行くのか？ これは、終了ステータスとして、OS に返される。慣例として、正常終了のきは 0、それ以外の時は非 0 の値を返す。 return 以外では、exit 関数でも、終了ステータスが返せる(exit 関数の引数が終了ステータスになる)。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a;
  a =atoi(argv[1]);
  printf("return %d¥n", a);
 return a;
}
```

このプログラムを実行すると

```
$ ./a.exe 3
 return 3

$ echo $?
3

$ ./a.exe 10
 return 10

$ echo $?
10
```

「$?」には直近の終了ステータスが入っている。シェルスクリプトなどから利用することもできる。Windows では、echo %errorlevel%。 ちなみに、ls コマンドの終了ステータスを見てみる。

```
$ ls
2nd

$ echo $?
0

$ ls aller
ls: aller にアクセスできません: No such file or directory

$ echo $?
2
```

存在しないファイル名を指定してエラーを起こしたところ、終了ステータスが 2 になった。

Appendix 6: 数学関数

C には、cosine, sine や平方根を計算するための関数 sqrt など、様々な数学関数が標準で用意されている。それらを使うには

•math.h をインクルードする必要がある

•コンパイルには、-lm (ハイフン エル エム)が必要

例 example.c

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

int main(int argc, char *argv[])
{
 double c;
  c = cos(3.14159/3.0);
  printf(“cos(3.14159/3.0) = %f¥n”,c);
return 0;
}
```

コンパイルは > gcc␣example.c␣**–lm**

Appendix 7: インデント

プログラムを見やすくするために、タブやスペースを使い、字さげしたり、改行をいれたりしよう。この資料では、あえてほとんどインデントは使っていない。
どんな字さげスタイルがあるか、インターネットで調べてみよう。indent コマンドなるものが使える場合もある。

また、プログラムの説明コメント(/*と*/で囲む)も積極的に入れて、他人が見ても理解しやすいコーディングに気を付けよう。

まとめ問題

1. bool 型、short 型、long 型について調べよ。また符号なし(unsigned)の変数についても調べよ。
2. int 型が、最大どのくらいの整数を扱えるか調べよ。
3. 度（degree）であらわされている角度を引数(double 型)でうけとり、ラジアンでの角度を戻り値(double)として返す関数を作成せよ。角度(ラジアン) = 角度(度)×$\pi/180$
4. 3 次元ベクトルの内積を計算する関数を作成せよ。double 型の配列 2 つ(3 次元ベクトル)を受け取り、それらの内積を計算、戻り値(double)とする。

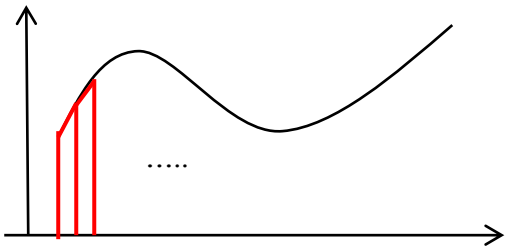
   $a=(a_x,a_y,a_z)$、$b=(b_x,b_y,b_z)$とすると、a と b の内積 $a \cdot b=a_xb_x+a_yb_y+a_zb_z$
5. n!を計算する関数を作成せよ。N を引数として受け取り結果を戻り値とする。

ただし、n は 0 以上の整数で、n! = n × (n”1) × (n”2) × … 2×1，ただし n=0 のとき 0!=1 である。

6. Napier 数 = 1+1/1!+1/2!+1/3!+1/4!+…である。7 で作成した関数を用いて、Napier 数を計算するプログラムを作成せよ。(もちろん適当なところで計算をやめるように作る)
7. 西暦 x 年がうるう年か否か調べるプログラムを作成せよ。うるう年は「x が 400 で割り切れる」あるいは「4 で割り切れるが 100 では割り切れない」年である。
8. rand()関数(0～RAND_MAX の整数の疑似乱数発生関数)を使って、2 次元ランダムウォークのプログラムを作成しよう。

   ルール:Rules

   1. 原点からスタート：initial position is the origin
   2. 毎ステップ x、-x、y または-y 方向に 1 ずつ進む
   3. 位置は、整数型とする。
9. 一様乱数を使って、π の近似値を求めよう。原点を中心とした半径 1 の円とそれをとり囲む一辺の長さ 2 の正方形を考える。-1.0～1.0 の間の乱数を振り(x,y)とする。(x,y)が円の中に入った否か確かめる。面積比より、円の中に入る確率は π/4 である。何度かこれを繰り返し、円の中に入った回数/総数が π/4 に等しいとして、π を推定せよ。ヒント：rand, srand 関数、RAND_MAX
10. 8 のランダムウォークを拡張することを考える。移動する方向を全方向、移動する距離を 0.0～5.0 にせよ。rand()関数を使って、0-5.0 の範囲と, 0～2 × π の範囲の乱数を発生させ(前者を r, 後者を theta とする)、 x=x+r*cos(theta), y=y+r*sin(theta)で移動するようにする。[理系向き]
11. じゃんけんプログラムを作成せよ。ユーザーとコンピュータの対戦。10 回戦で、何度勝ったかも表示させる。
12. 台形公式により(下の図参照)関数(y=f(x))を定積分するプログラムを作成し、自分で定義した関数を定積分し、きざみの幅をいろいろ変えて、手計算の場合とどれだけ違うか確認せよ。[理系の人向け]



FROM C to FORTRAN

C 言語との対比で、Fortran90 の初歩の初歩を解説する。大野の偏見と、無理やり強引に対比させることによって生じた不正確な記述もあることはご了承いただきたい。なお、Fortran90 のソースコードも cygwin でコンパイル可能である。コマンドは gfortran である。

プログラムの基本形約束事

FORTRAN 90	C 言語
基本的に 1 文は 1 行に収める。1 行にかけるのは 132 文字。1 行に収まらない時は、文の最後に&をつけて改行する。	途中で改行することも可能。文字列の途中とかは改行できない。
文の最後にセミコロンはいらない。ただし、1 行に複数の分を書きたい時は、セミコロンで区切る。	セミコロン必要
変数の名前などは、大文字小文字は区別されない。（文字列の中身とかはもちろん区別される）。変数などの名前付け規則はだいたい C と同じと考えてよい（かも）	大文字小文字は区別される。
コメントは、！の後全部。改行するまで	/* と */で囲む
ファイルの拡張子は f90	拡張子は c
Cygwin でのコンパイラのコマンド gfortran	Cygwin でのコンパイラのコマンド C コンパイラ：gcc，C++コンパイラ:g++

プログラムの基本形は、

```
program␣プログラムの名前変数の宣言

処理

end␣program␣プログラムの名前
```

である。上から下に向かって、処理が実行される。

Hello, World を作ってみると

```
program␣hello_world
 print␣*, "Hello, World"
end␣program␣hello_world
```

となる。大文字と小文字は区別されないので、

```
PROGRAM␣HELLO_WORLD
 PRINT␣*, "Hello, World"
End␣Program␣HELLO_WORLD
```

と書いてもよい。各自の趣味に応じて決めればよい。

変数とデータ型

	C 言語	FORTRAN 90
文字型	char	character
整数型	int	integer
単精度実数型	float	real
倍精度実数型	double	dp を整数型とする dp=selected_real_kind(14) real(kind=dp)
論理型	bool	logical
複素数型	“	complex
変数の変更をできないようにする	const	parameter
暗黙の型宣言	なし	あり

変数宣言方法を次のプログラムで見てみる。

```
program␣var
 implicit␣none
 integer␣::␣i
 integer, parameter␣::␣dp = selected_real_kind(14)
 real␣::␣sr
 real(kind = dp)␣::␣dr1, dr2
```

```
dr1 = 3.141592653589793
dr2 = 3.141592653589793_dp

print␣*, "dr1 = ", dr1
print␣*, "dr2 = ", dr2

end␣program␣var
```

実行結果

```
$ ./a.exe
 dr1 =    3.14159274101257 32
 dr2 =    3.1415926535897931
```

implicit none とは、暗黙の型宣言を無効にするためのものである。 FORTRAN では、暗黙の型宣言と言う規則があり、宣言なしで i,j,…n で始まる名前の変数は整数型、それ以外は実数型として使える。しかし、ミスの原因になるので、この規則は無効にして、使用する変数はすべて宣言する方が良い。

変数の宣言方法は、基本的には、型名をかいて、:: (コロン 2 個)につづき、変数名を書く[7]。

int a [C 言語]→ integer :: a [Fortran90]

複数書きたい時は、カンマで区切る。また、宣言と同時に初期化したいときは、C 言語と同様に、= 値と続ける。 ただし、宣言と同時に初期化すると、後述する save 属性(C でいう static)がつくことになる（みたい。経験的に)。

parameter は、型の後にカンマに続けて書く。変数 dp は、変更しないので、parameter を使っている。

const int a = 1 [C 言語]→ integer, parameter :: a = 1[Fortran90]

selected_real_kind の括弧の中の 14 は、少なくとも 14 桁の精度 と言うのを意味している（ということは倍精度。単精度の変数では、そこまでの精度はない)。 実数の定数であるが、何もしないと、単精度として扱われるので、倍精度実数の変数に代入しても、結果がよろしくない(実行結果参照)。倍精度の定数にするには、倍精度実数の変数の宣言で使った dp を数字の後に"_dp"のようにして付ける。[8]

---

[7] これ以外の宣言方法もある

[8] dp には、8 が入っていると思うが、全ての処理系でそうとは限らないらしいので、selected_real_kind で求めたものを入れるとよい。

logical 型は.true.と.false.がある(C 言語の bool 型では true と false).
次に、文字配列のサンプルプログラムを見てみよう。

```
program char
 implicit none  character :: a
 character(len = 11) :: b
a= "a"
b= 'john lennon'
 print *,a
 print *,b
end program char
```

実行結果

```
$ ./a.exe
 a
 john lennon
```

a は一文字、b は 11 文字記憶できる（len=は省略可能)。C 言語のように、終端文字の心配はしなくていいし、"で囲んでも、'で囲んでもよい[9]。また、C 言語では strcpy 関数を使わなければできないようなことも、Fortran 90 では手軽にできる(b への文字列の代入参照)。

計算

	C 言語	FORTRAN90
足し算	+	+
引き算	-	-
掛け算	*	*
割り算	/	/
余り	%	組み込み関数 mod(i,j) 整数 i を整数 j で割った余り
冪	pow 関数	**
括弧	( )	( )

Fortran90 では、簡単に冪の計算ができる。また、C 言語と同様、異なる型の混在する計算では、小さい型の数値が大きい型の方に変換されてから計算される。ただし、

[9] もちろん細かい規則はあるので、常にどちらでもいいとは言えない

Fortran では、実数より複素数のほうが大きい型である。しかも、倍精度実数より、単精度の複素数の型が大きいとみなされる。

次のプログラムの結果を見てみよう

```
program calc1
 implicit none
 integer :: i, j, k
 real :: a, b, c

 i = 5;  j = 2
 a = 2.0

 k = mod(i, j)
b= a**2
c= a**0.5

 print *, k
 print *, b
 print *, c

end program calc1
```

実行結果

```
$ ./calc1.exe
           1  ← 5÷2 のあまり
   4.0000000  ← 2.0 の二乗
   1.4142135  ←2.0 の  0.5 乗(平方根)
```

Fortran90 には、数多くの数学系の関数があり、それらも使用可能である。実数が引数の場合、単精度か倍精度かで、内部で使用する関数が違ってくる（倍精度の数値を渡せば倍精度用の関数、単精度の実数を渡せば単精度用の関数を自動で選んでくれるので、精度はあまり気にせず使用できる）。

ここで、0 で割ったらどのような数値が出るか、次のプログラムで試してみある。

```
program inf
 real :: v,m,n
 v = 1.0; m = 0.0;
 n = v/m
 print *, n
```

```
end program inf
```

```
$ ./a.exe
      +Infinity
```

-1.0 の平方根は？

```
program nan
 implicit none
 real :: a,b

 b = -1.0;
 a = sqrt(b)

 print *, a
end program nan
```

```
$ ./a.exe
          NaN
```

Infinity(無限大)や NaN(not a number)が出たら要注意。0.0/0.0 でも NaN がでる。

キーボードからの入力

C 言語では、scanf を使って、標準入力から数値などを入力するようにしていたが、Fortran90 でこれを実現するのは、read である。read を使った数値の入力は、以下のサンプルプログラムを見てもらいたい。

```
program indata
 implicit none
 integer, parameter :: dp = selected_real_kind(14)
 integer :: i
 real :: r
 real(kind=dp) :: d

 read *, i, r, d

 print *, i, r, d

end program indata
```

実行例

$ ./a.exe

1 2.0 3.0⏎

　　1　2.0000000　　3.0000000000000000

$ ./a.exe

1⏎

2⏎

3⏎

　　1　2.0000000　　3.0000000000000000

$ ./a.exe

1,2,3.5⏎ ← カンマでも区切ってよい

　　1　2.0000000　　3.5000000000000000

実行例を見ると、ほぼ scanf と同様な動きをすると考えてよい。さらに、scanf のように細かい書式(“%f%d”など)は必須ではなく、並んでいる変数で判断して代入してくれるので結構楽である。scanf と異なる挙動を示すプログラムの例

```
program indata2
 implicit none
 integer, parameter :: dp = selected_real_kind(14)
 integer :: i, j
 real :: r
 real(kind=dp) :: d

 read *, i, r, d
 read *, j
 print *, i, r, d
 print *, j

end program indata2
```

実行例

$ ./a.exe

1 2.0 3.0 4 5 6

7

　　1　2.0000000　　3.0000000000000000

　　7

4 5 6 が無視された。まったく scanf と同じだと思ってはいけない(特に数値を並べて書くとき)。

半径を受け取り、円の面積と円周を求めるプログラムは、

```
program circarea
 implicit none
 integer, parameter :: dp=selected_real_kind(14)
 real(kind=dp),parameter :: pi = 3.14159265358979_dp
 real(kind=dp) :: radius
 real(kind=dp) :: area, circum
 print *,"Enter a radius "
 read *, radius

 circum = 2.0_dp * pi * radius  area
= pi*(radius**2)

 print *,"Enshu = ", circum, "Menseki = ", area

end program circarea
```

条件分岐 1(IF)

数値を比較のための演算子の比較

	C 言語	FORTRAN 90
A は B より大きい	A > B	A > B または A .gt. B
A は B 以上（≧）	A >= B	A >= B または A .ge. B
A は B より小さい	A < B	A < B または A .lt. B
A は B 以下 (≦)	A <= B	A <= B または A .le. B
A と B は等しい	A == B	A == B または A .eq. B
A と B は異なる(≠)	A != B	A /= B または A .ne. B

最後の 1 つだけ新たけ憶えれば、あとは C 言語と同じ記号が使える。

論理演算子の比較

	C 言語	FORTRAN 90
OR	A || B	A .or B

AND	A && B	A .and. B
NOT	!A	.not. A

C 言語と記号は違うが、and や or の前後にピリオドをつけるだけなので、さほど覚えるのに苦労はない。

Fortran 90 の if の基本形は、

```
if(条件 1) then
 処理 1
else␣if(条件 2) then
 処理 2
else␣if(条件 3) then
 処理 3
else
 処理 n
end␣if
```

である。中括弧{} で囲まない代わりに、then と end if が来るのが違うだけで、あとは C 言語と同じと考えてよい。

もちろん、else if や else は必ずしも使わなくてもよい。例えば

```
if(条件) then
 処理
end if
```

や

```
if(条件 1) then
 処理 1
else
 処理 2
end if
```

のような形でも使うことができる。例として、comp_int.c を Fortran 90 で書き換えてみると、下記のようになる。

```
program compint
 implicit none
 integer :: a, b

 print *, "input a number(a)"
 read *, a
```

```
 print *, "input a number(b)"
 read *, b

 if (a > b) then
   print *, "a is bigger than b"
 else if(a==b) then
   print *, "a is equal to b"
 else
   print *, "a is smaller than b"
 end if
end program compint
```

なお 1 行で終わる程度のものなら
if(条件) 処理
と言う書き方もできる

条件分岐 2 (CASE)

Fortran90 には、C 言語の switch と同様な働きをする case 構文がある。しかも、範囲指定ができるなど、switch より多機能である。 case 構文の基本形は、

```
select␣case(式)
   case(値 1)
    処理 1
   case(値 2)
    処理 2
   .....
  case␣default
    処理 n
 end␣select
```

式は、整数、文字、論理が使用できる（実数系は C 言語でも駄目だった)。また、 C 言語の break にあたるものは入れなくてよい。値のところは、範囲指定も可能である。例えば、値 A:値 B とすると、A～B、値 A: とすると、A～、
：値 B とすると～B を表す。範囲指定する場合は、重複する区間があってはならない。

例として、switch123.c を Fortran90 に直すと、

```
program switch123
 implicit none
 integer :: choice

 print *,"input a number"
 read *, choice

 select case(choice)
   case (1)
     print *,"one"
   case (2)
     print *,"two"
   case (3)
     print *,"three"
   case default
     print *,"1,2 or 3 is required"
 end select

end program switch123
```

となる。範囲指定を使い少し書き換えると

```
program switch123_kai
 implicit none
 integer :: choice

 print *,"input a number"
 read *, choice

 select case(choice)
   case (:1)
     print *,"LE1"
   case (2:3)
     print *,"two or three"
   case (4:)
     print *,"GE4"
```

```
 end select

end program switch123_kai
```

繰り返し実行

C 言語の for にあたるものとして、do～end do がある。基本形は、 do␣[制御変数=初期値, 限界値[, 増分]]

　処理

end␣do

制御変数とは、C 言語の for でいうと、for(i=0;…の i のことである。増分は省略可能で、省略したときは制御変数が+1 される。さらに書くと、do の後の制御変数=初期値、なども省略可能である。

```
program hello10
 implicit none
 integer :: i

 do i=1, 10
   print *,"i=",i , "Hello, World"
 end do

end program hello10
```

C 言語では、ループを抜け出たりするための、break や continue があった。

Fortran 90 でもそれに対応したものはある。

C 言語	FORTRAN90
break	exit
continue	cycle

```
program hello10_kai
 implicit none
 integer :: i = 1

 do
```

```
  if (mod(i, 2) == 1) then
    i = i + 1
    cycle
  end if

  print *, "i=",i ,"Hello, World"

  if (i == 10) then
    exit
  end if

  i = i + 1
 end do

end program hello10_kai
```

実行結果

```
$ ./a.exe
  i=           2 Hello, World
  i=           4 Hello, World
  i=           6 Hello, World
  i=           8 Hello, World
  i=          10 Hello, World
```

i が奇数のとき、cycle により Hello, World を表示せず、 i が 10 になったら、exit によりループを終了している。

すでに気づいていると思うが、print を実行すると、改行される。また、文字列と一緒に数値を表示すると、どうも表示が気に食わない。print にも C 言語の printf のように、書式を指定できる。

書式指定子の対比

	C 言語	FORTRAN90
文字、文字列	%c (1 文字), %s(文字列)	A
整数	%d	I (Iw)
実数	%f	F (Fw.d)
実数(指数表示)	%e, %E	E (Ew.d)
空白	“	X

w:フィールドの幅、d:小数点以下の桁。Fw.d は、「w 桁の幅をとって、そこに実数を表示する。小数点以下は 5 桁表示する。」という意味。Iw は「w 桁の幅をとって、そこに整数を表示する。」という意味。

C では、%w.df や%wd となっていたところが、Fw.d や Iw となったと思えばよい。また nFw.d(たとえば 2F10.5)と書くことで、n 個の Fw.d という意味になり、同じような数値を表示したいとき便利である。空白の場合は、空白が一つでも 1X と書かなければならない。また C 言語と違い、w や d は必ず書かなければいけない。

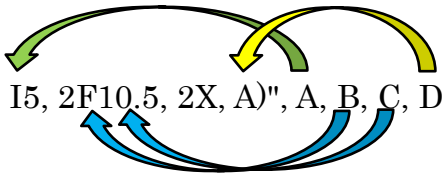
サンプルを見てみる。*があった部分に、書式を入れている。(今回は大文字で)

```
PROGRAM Form1
 INTEGER :: A
 REAL :: B, C
 CHARACTER(11) :: D

 A = 100
 B = 1.41421
 C = 1.73205
 D = "John"

 PRINT "(2X, I5, 2F10.5, 2X, A)", A, B, C, D

END PROGRAM Form1
```



次のような指定もできる。

```
PROGRAM Form1_KAI
 INTEGER :: A
 REAL :: B, C
 CHARACTER(11) :: D

A = 100
B = 1.41421
C = 1.73205
D = "John Lennon"

 PRINT 100, A, B, C, D
 100 FORMAT(2X, I5, 2F10.5, 2X, a)
```

```
END PROGRAM Form1_KAI
```

PRINT 100,a,b,c,d とは、文番号 100 に書式がありますよ、という意味になる。文番号は 5 桁以下の正の整数で自由につけられる。FORMAT のところは、特に何かが実行されるわけではない。

また、書式では、(2X, I5, 2(F10.5, 2X), a)のような指定も可能、(F10.5,2X)が 2 個という意味になる。

繰り返し実行 2(do while)

C 言語の while に対応しているのが、Fortran90 の do while である。制御変数の操作は、プログラマに任せられている。

基本形は

do while(条件)

　処理

end do

これを使うと、hello10 は

```
program hello10_2
 implicit none
 integer :: i = 1

 do while(i<=10)
   print *,"i=",i, "hello, world"
   i = i + 1
 end do

end program hello10_2
```

ファイル入出力

ファイルへの書き込み、ファイルからの読み込みに関しては、次の対応があると思う。

C 言語	FORTRAN90
printf	print, write
scanf	read
fprintf	write
fscanf	read

print はすでに出てきているので（read も出ているけど)、write と read について、解説する。

write および read は

write (制御指定子) 変数（複数ある時は、カンマで区切っていく）
read (制御指定子) 変数（複数ある時は、カンマで区切っていく）

である。制御指定子は、

- unit = 装置識別子
- fmt = 書式識別子
- advance：改行するか否か

などがある。

装置識別子は、整数の番号。ただし、

- 0: 標準エラー出力 (stderr)
- 5: 標準入力 (stdin)
- 6: 標準出力 (stdout)

なので、これ以外の番号を使用する。これは C 言語の fp と思えばよく、ファイルなどをこれで区別する。書式識別子は、print のところですでに紹介した。句は、”yes”または”no”とする。これは改行するか否かである。デフォルトでは yes となっていて、改行される。例えば

write(unit=6, fmt=“(2x, 2f10.5)”) x, y

unit や fmt は * を置くことで省略できる（画面に表示したり、キーボードから入力させたいとき)。
例えば

print *,"i=",i,"hello, world" → write(*,*),"i=",i,"hello, world"

ファイルのオープン、クローズは、(C 言語では fopen, fclose)
open(オープン指定子)
close(クローズ指定子)
で行う。
オープン指定子は、以下のようなものがある

- unit=装置番号
- file=フィル名(文字列)

- status：”old”(すでにファイルが存在する。読み込むとき), “new” （新たにファイルを作るとき。書き込むとき）, “replace”（既に存在するファイルを消して新たにファイルを作る）のどれか。
- action=入出力動作：”read”, “write”, “readwrite”のどれか
- iostat：ファイルオープンに成功すると句が 0、失敗すると正の数値

クローズ指定子は、以下のようなものがある

- unit=装置番号もちろんこれ以外にもあるが、最初はこれだけでよい。

書き込みのサンプルとして、hello10_2 をファイルに書き込むように修正してみる。

```
program hello10_2f
 implicit none
 integer :: i = 1, iostatus

 open(unit=10, file="hello.txt", status="new", &
   action="write", iostat = iostatus)

 if(iostatus > 0) stop "can't open file"

 do while(i<=10)
   write(unit=10, fmt="(a, i2, 2x, a)") "i=", i, "hello, world"
   i = i + 1
 end do

 close(unit=10)
end program hello10_2f
```

if は、ファイルがオープンに成功したがどうか調べて、失敗したら終了するようにしている(stop でプログラムは終了する)。

C 言語で fp=fopen… ; if(fp==NULL){ … exit(1);}をしていると思えばよい。

次に、読み込みのサンプルとして、身長、体重のデータを読み込み、BMI を計算して画面に書き出すプログラムを作成してみる。

```
program calcbmi
 implicit none
 integer :: iostatus
 real :: height, weight, bmi
```

```
open(unit=10, file="data.txt", status="old", action="read", &
 iostat = iostatus)

if(iostatus > 0) stop "can't open file"

do
read (unit=10, fmt=*, iostat = iostatus) height, weight

if(iostatus < 0 .or. height < 0.0 .or. weight < 0.0) then
 ! read all data or less than zero
  exit
end if

bmi = weight/height**2
print "(a, f5.1)", " bmi is ", bmi

end do

close(unit=10)
end program calcbmi
```

※ ファイルの最後まで行くと、iostat は負の値になることを利用している.

ここで、print の書式を例えば(a, f2.1)に変えて実行してみると、数値がアスタリスクに代わっていることがわかる。 Fortran90 では、書式に指定した幅に収まりきれない時、アスタリスクに変えて出力する[10]。

配列

Fortran90 での 1 次元配列の宣言方法は、

型, dimension(要素数) :: 変数[11]

整数型の場合

Integer, dimension(5) :: a

である。こうすることで、

a(1)～a(5)まで整数型として使える。C 言語では、[ ] であったが、Fortran では、普通の( )を使う。また、インデックスが 0 からではなく、1 からのことに注意[12]。

---

[10] C 言語では幅が足りなくても無理やり表示する

[11] これ以外の宣言方法もある

[12] インデックスが 1 からにしない方法もある

値の代入は C 言語と同様に

a(1) = -1; a(2) = 3

などとする。一気に代入する方法もあり、 a= (/1,2, 3, 4, 5/) とすると、
a(1) = 1; a(2)=2; a(3)=3; a(4)=4; a(5)=5 としたことと同じ。

配列の宣言は、生の数字を使ってハードコーディングせず、

```
 integer, parameter :: n = 5
integer, dimension(n) :: a
```

とすることもできる。ここまで紹介したことをすべて使って意味のないサンプルプログラムを作ると、

```
program array1
 implicit none
 integer, parameter :: n = 5
 integer, dimension(n) :: a
 integer :: i

 a = (/1,2,3,4,5/)

 do i=1, n
   print *, a(i)
 end do

end program array1
```

実行結果

```
$ ./a.exe
           1
           2
           3
           4
           5
```

多次元配列の宣言は、

integer, parameter :: nx = 5, ny=4, nz=3
integer, dimension(nx, ny, nz) :: a

などとする。使い方は a(1,1,2) = 2 などなど[13]。メモリの中では、一番左の添え字から順に並んでいる(C 言語は一番右)。多次元配列への一括の値代入は、もう少し知識が必要なので割愛する。

配列のまとめとして、pass.c の Fortran90 版を作ってみる

```
program passf90
 implicit none
 integer, parameter :: numstudents = 10
 integer, parameter :: unitnum = 10
 integer, parameter :: dp = selected_real_kind(14)
 real(kind=dp) :: ave = 0.0_dp
 integer, dimension(numstudents) :: score
 integer :: iostatus, i

 open(unitnum, file = "score.txt", status = "old",&
  action="read", iostat=iostatus)

 if(iostatus > 0) stop "can't open file"

 do i=1, numstudents
   read (unitnum, *) score(i)
   ave = ave + score(i)
 end do

 close(unitnum)

 ave = ave/numstudents

 open(unitnum, file = "pass.txt", status = "new",&
  action="write", iostat=iostatus)

  if(iostatus > 0) stop "can't open file"

 write(unitnum, "(1x, a, f5.1)") "average = ", ave
 do i=1, numstudents
   if(score(i) >= ave) then
     write(unitnum, "(1x, i3)") i
```

[13] C 言語では各次元のインデックスを[ ] で囲んだ

```
  end if
 end do

 close(unitnum)
end program passf90
```

なお、Fortran には、配列を操作する組み込み関数などが豊富に用意されているので、調べること。

サブルーチンと関数

C 言語の関数にあたるものに、サブルーチンや関数がある。

サブルーチン基本形は、

subroutine 名前(引数並び)

　変数などの宣言

　処理

end subroutine 名前

呼び出す時は、 call サブルーチンの名前
(引数) とする。

以下に BMI を計算するサンプルプログラムを示す。

```
program caclBMI_s
 implicit none
real :: height, weight, bmi

 write(6,"(2x, a)", advance="no") "input height (m) and weight (kg): "
 read *, height, weight

 call calcbmi(height, weight, bmi)

 write(6, "(a, f5.1)") "bmi is ", bmi

end program celtofah_s

subroutine calcbmi(h, w, bmi)
```

```
 implicit none
 real, intent(in) :: h, w
 real, intent(out) :: bmi

 bmi = w/h**2

end subroutine calcBMI_s
```

C 言語と違い、引数の内容を変えることができる。CalcBMI 中で、BMI の値を書き変えて、呼び出し元に戻している。また、

```
 real, intent(in) :: h, w
 real, intent(out) :: bmi
```

で、引数の型や属性を記述している。intent(in)な仮引数は、サブルーチン内で書き変えられない。受け取るだけ。intent(out)は、書き変えて、呼び出し元に値を戻すことができる。この他に、intent(inout)という、受け取りかつ書き変え可能な属性もある。指定しないと、intent(inout)扱いになる。out や inout 属性の引数は、C++言語でいう参照渡しであろう。

関数

サブルーチンは、引数を通じて値を返していたが、名前で結果を返すことができる関数も Fortran90 に存在する。関数を使って上のプログラムを書き変えると、

```
program calcBMI_f
 implicit none
real :: height, weight, bmi
 real :: calcbmif

 write(6,"(2x, a)", advance="no") "input height(m) and weight(kg): "
 read *, height, weight

bmi = calcbmif(height, weight)

 write(6, "(a, f5.1)") "bmi is ", bmi

end program calcbmi_f

function calcbmif(h, w)
 implicit none
```

```
 real, intent(in) :: h, w
 real :: calcbmif

 calcbmif = w/h**2

end function calcBMI_f
```

関数は、C 言語ライクな呼び出しができる。呼び出し元でも calcBMIF を宣言しなければならない。また、関数内では、関数と同じ名前の変数の値が、呼び出し元に返される[14](intent(out)属性の引数で値を返すこともできちゃうみたいだけど)。

ここまでは、メインとなるプログラムの外に、サブルーチンや関数を書いてきたが、この書き方だと、コンパイラは引数の型のチェック等をすることができない。使い方を誤らなければ問題ないが、C 言語のプロトタイプ宣言のようなことをすると、コンパイラがチェックすることができる。以下はその例。

```
program calcBMI_I
 implicit none
real :: height, weight, bmi

 interface
   function calcbmif(h, w)
     real, intent(in) :: h, w
     real :: calcbmif
   end function calcbmif
 end interface

 write(6,"(2x, a)", advance="no") "input height(m) and weight (kg): "
 read *, height, weight

 bmi = calcbmif(h, w)

 write(6, "(a, f5.1)") "bmi is ", bmi

end program calcbmi_i

function calcbmif(h, w)
```

[14] これがいやなら、result という句を使って、別な変数の値が戻るようにできる。

```
 implicit none
 real, intent(in) :: h, w
 real :: calcbmif

 calcbmif = w/h**2

end function calcBMI_I
```

interface～end interface のあいだに、関数の引数の型や戻り値の型などの情報の部分を書く(C 言語とほぼ同じだ)。

また、サブルーチンや関数は、プログラム本体に収めることもできる。メインのプログラムのうしろに、contains と書いて、それに続けて(かつ end program より前)に関数やサブルーチンを書く。

```
program calcBMI_IN
 implicit none
real :: height, weight, bmi

 write(6,"(2x, a)", advance="no") "input height(m) and weight (kg): "
 read *, height, weight

 bmi = calcbmif(h, w)

 write(6, "(a, f5.1)") "bmi is ", bmi

 contains

function calcbmif(h, w)
 implicit none
 real, intent(in) :: h, w
 real :: calcbmif

 calcbmif = w/h**2

end function calcbmif
end program calcBMI_IN
```

関数やサブルーチンをメインのプログラムの内部に書くと、interface で引用の仕様を書いたり、戻り値の型宣言もしなくて良い。また、こうすると、内部に置いた関数やサブルーチンからメインのプログラムの変数に、C 言語のグローバル変数のようにアクセスできるようになる。C 言語と同様、関数やサブルーチン内に同じ名前の関数があれば、そちらが優先される。また、関数やサブルーチン内の変数は、親元からはアクセスできない。(ミスの元なので、関数やサブルーチンから大域要素にアクセスするプログラムを書くのは、やめたほうが良い)。

では C 言語の静的変数(static をつけた変数)は、Fortran90 でどうすればよいか？ 次のようにすればよい。

型, save :: 変数 = 初期値

宣言と同時の初期化が必要。たとえば、

integer, save :: i = 0

save をつけると、関数やサブルーチン内の変数の値が保存される。

構造体

Fortan90 でも構造体を使用することができる。構造体の定義は

type␣型名

　変数…

end␣type␣型名

とかく。この型の変数宣言は、

type(型名) :: 変数名

でよい。内部の要素にアクセスるには、%を使う。まとめると

C 言語	FORTRAN90
struct	type
アクセスするときは、 .	%

例

```
struct student{
        char name[64];
        int socre_eng;
        int score_math;
        int score_sci;
```

};
を Fortran では、
type(student)
  character(64) :: name
  integer :: score_eng
  integer :: socre_math
  integer :: score_sci
end type student
（student が型名にあたる）
type(student) :: person
としたならば、要素へのアクセスは、%を使って

  person%score_eng = 87
のようにする。

バイナリモードでのファイルの読み書き

open のところに、form="unformatted"といれ、read, write のところに、書式関係の引数を書かない。整数型の要素　100 個の配列を書き込むプログラムとそれを読み込むプログラムを下記に示す。

書き込みは下記プログラム

```
program write_bin
 implicit none
 integer, parameter :: n = 100
 integer, parameter :: unitnum = 10
 integer, dimension(n) :: a
 integer :: iostatus, i

 do i=1, 100
   a(i) = i
 end do

 open(unitnum, file = "bin.dat", status = "new",&
  action="write", iostat=iostatus, form="unformatted")

 if(iostatus > 0) stop "can't open file"
```

```
 write (unitnum) a

 close(unitnum)

end program write_bin
```

読み込みは下記プログラム

```
program read_bin
 implicit none
 integer, parameter :: n = 100
 integer, parameter :: unitnum = 10
 integer, dimension(n) :: a
 integer :: iostatus, i

 open(unitnum, file = "bin.dat", status = "old",&
  action="read", iostat=iostatus, form="unformatted")

 if(iostatus > 0) stop "can't open file"

 read (unitnum) a

 close(unitnum)

 write(*, "(100i4)") a

end program read_bin
```

実行結果

$ ./write_bin.exe

$./read_bin.exe

```
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25
 26  27  28  29  30  31  32  33  34
 35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
 52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
 69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
 86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

確かに読み書きできた。しかし、ls –l を実行してみると bin.dat のファイルサイズが、少し大きい

-rw-r—r-- 1 ohno None 408 Jul 24 12:55 bin.dat

整数型 100 個なので、ファイルサイズは 400 バイトのはずが、408 バイトになっている。これは、データにそれぞれ 4 バイトのヘッダとフッタが付加されたからである[15]。これは、変数ごとにつく。例えば

write(unitnum) A
write(unitnum) B

とすると、全体のはじめと最後に 4 バイト付く(ヘッダ+A+B+フッタ)のではなく、A の前後と B の前後に 4 バイトずつ付き(ヘッダ+A+フッタ+ヘッダ+B+フッタ)、合計 16 バイト大きくなる。

Fortran で読み込むのは問題ないが、C 言語で作ったプログラムで読み込むときは、最初と最後の 4 バイトを読み飛ばさなければいけない。

form = “unformatted”ではなく、form=“binary”とすると、このヘッダとフッタの付加をさせないこともできるが、全てのコンパイラでこれが使用可能と言うわけではない。(gfortran ではコンパイル時にエラーが出る。SGI サーバーの Intel コンパイラでは通った)

これから

ここまでで、初歩の範囲はカバーしているのでないかと思う。さらに Fortran の学習を進めるのであれば、モジュール、割り付け配列(C 言語の動的メモリ確保に相当(すると思う))などを勉強するとよい。

この資料には、間違いが含まれている可能性は大いにあります（人間のすることなので)。特に Fortran の部分は短時間で作成して、プログラムは Word に張り付けてから、整形したり変数の名前を変えたりしているので、動作しないものが含まれているかもしれません。間違い探しをするつもりで、勉強するといいかも。

練習問題

C 言語のところにある練習問題、まとめ問題を全部 Fortran で解いてみる

[15] ファイルサイズが記録されている。8 バイトずつつく処理系もあるとか。

参考文献

1. 大角盛広,「C 言語入門」, 西東社（1995）（名著だと思います。これで C 言語を憶えました）
2. ラリー・ニーホフ，サンフォード・リーストマー，渡辺了介（翻訳），ドキュメントシステム（翻訳），「入門 Fortran90」, ピアソンエデュケーション（2001）

・2012 年 7 月 25 日 初版
・2012 年 8 月 1 日 修正
・2021 年 3 月 26 日 修正
・2021 年 5 月 30 日 修正