Tiny Basic for Windows 初級編

tbasic.org [*1]

[2014 年 6 月版]

Tiny Basic for Windows 入門[*2]では最も基本的なコマンド，文について説明しました。ここでは，少し進んだプログラムを書く上で，知っていると便利なコマンド・文について説明します。

目次

[*1] http://www.tbasic.org

[*2] 以下では入門編と言うことにします。

1 配列変数 と Dim 文

1.1 配列変数の目的

コンピューターは一つ一つの処理は，単純なことしかできませんが，それが大量にあっても高速に処理するのは得意です。他方コンピューターに膨大な処理をさせるには，それを指示するプログラムが必要になります。プログラムは人間が書きますから，もし膨大な処理を指示するために，膨大な量のプログラムを書かなくてはならないとすれば，コンピューターの能力を十分に生かすことはできないでしょう。ですから，少ないプログラムで多くの処理が可能になるように，色々な仕組みが，コンピューターの開発，プログラミング言語の開発で工夫されてきました。

入門編で学んだ，For 文や While 文などの繰り返しの構文は，コンピューターの動作についてのそのような対応の一つでした。繰り返し構文は動作への対応でしたが，動作ではなく動作の対象への工夫もあります。コンピューター動作での対象はメモリーですが，そのメモリーの取り扱いについても工夫がなされています。

それらの内，最も基本的なものがここで説明する配列変数です。配列変数はどのようなプログラミング言語でも標準的に備えられて，どのような言語でプログラミングするにしても，配列変数の理解は本格的なプログラミングでの必須なことです。

配列変数の理解はプログラミングでの
基本的重要事項

1.2 10 個の数の入力

まず，例をとして，「10 個の数を入力し，それらを各々別の変数に代入する」プログラムを作ってみましょう。つまり，10 個の変数を用意し，それぞれに別々な値を入力するとします。例えば，10 人の身長を入力し，平均の計算に利用したい場合などが考えられます。

今まで学んだことを使って書いてみると，次のようになるでしょう。

```
'10 個の数の入力
Input "数を入力して下さい";A1
Input "数を入力して下さい";A2
Input "数を入力して下さい";A3
Input "数を入力して下さい";A4
Input "数を入力して下さい";A5
Input "数を入力して下さい";A6
Input "数を入力して下さい";A7
Input "数を入力して下さい";A8
Input "数を入力して下さい";A9
Input "数を入力して下さい";A10
```

確かにこれで 10 個の数が入力できます。しかし，もし，100 個の数，あるいはそれ以上の場合では，このような方法で書く気は起きないでしょう。また，この 10 個の場合でもこの書き方はあまりスマートとは言えません。何かもっと上手に書けそうな気がします。

例えば For to Next 文を使って書けないでしょうか。変数 A1 〜 A10 に注目して,

```
'10 個の数の入力
For  I = 1 to 10
   Input "数を入力して下さい";AI
Next I
```

としたらどうでしょうか。これなら随分と簡潔です。しかしこれは**誤り**です！このプログラムを実行すると,確かに 10 個の数は入力しますが，それらの数を 10 個の 変数に代入しません。

実は,

このプログラムには，1 つの変数 AI しかありません！！

実際，For I = 1 to 10 によって，変数 AI から 10 個の変数 A1 〜 A10 を作ることはできません。AI の I は変数名 AI を構成する記号で，数値変数 I とは関係がありません。

ではどうしたら良いのでしょうか。

実は，この様な場合のために，BASIC では今迄とは別の形の変数が用意されています。それが配列変数です。

1.3 配列変数

配列変数は
変数名 (I)
の形で表されるもので， I を 0,1,2,… と動かすことが出来，その各々が， 1 つの変数 になります。

例えば， A(1),A(2)，… , A(10) は配列変数で

```
A(I)  は  I = 1  のとき  A(1)
A(I)  は  I = 2  のとき  A(2)
…
A(I)  は  I = 10  のとき  A(10)
```

を表します。これに対して,

A1，A2，… , A10

は 10 個の普通の変数です。

配列変数を使って，上のプログラムを書き直すと，次が得られます。

```
'１０個の数の入力
For I = 1 TO 10
   Input "数を入力して下さい";A(I)
Next I
…
…
```

これでよいのですが，重要な補足があります。BASIC では 一般に 変数は プログラム に書けば，自動的に

確保されますが， これには例外があります。その例外が配列変数です。配列変数を使うときは，予め，どんな名前の配列変数をどのくらい使うかを，宣言して 置く必要があります。

宣言の仕方は簡単です。それは

Dim 変数名（最大値）

とすれば良いのです。 これを Dim 文といいます。Dim は Dimension (次元，大きさ) の意味で，配列変数として使う変数名と使う大きさを宣言します。BASIC は Dim 文 に出会うと，そこで宣言された，変数をメモリ上に確保します。例を見てみましょう。

```
Dim A(100)
```

は 配列変数

```
A(0),A(1),A(2), … ,A(100)
```

を使うと宣言します。また

```
Dim Moji$(100)
```

は配列文字列変数

```
Moji$(0),Moji$(1), … ,Moji$(100)
```

を使うと宣言します。前のプログラムに宣言文をつけ加えると，

```
'１０個の数の入力
Dim A(10)
For I = 1 TO 10
  INPUT "数を入力して下さい";A(I)
Next I
```

となります。

配列変数での宣言

配列変数 を使う 都度，Dim 文 で宣言をしなくてはならないのは面倒ですが，これには合理的な理由があります。

配列変数は特別の構造を持つため，その変数をメモリー上に確保する場合，使う最大数を一度に確保するのが効率的です[*3] 。しかしその最大数は使う目的によって様々です。しかも大きな最大数の配列は大きなメモリを必要とします。従って，必要の無いものはなるたけ，小さな最大数の配列にすることが，メモリーを効率的に使う方法です。

では，その配列変数がどれだけの最大数が必要か，どのようにして知るのでしょうか。 それはそのプログラムを書く人が知らせるしかありません。従って宣言が必要になるわけです ！

[*3] 勿論，使う都度順次確保していくことは不可能ではありません。しかし，順次確保する方法は，一般に実行速度が遅くなります。

今まで配列変数について 1 次元のものしか説明しませんでしたが，2 次元，3 次元の配列変数も使うことが出来ます。例えば，

```
Dim Mat(3,3)
```

は 2 次元の配列変数

```
Mat(0,0), Mat(0,1), Mat(0,2), Mat(0,3)
Mat(1,0), Mat(1,1), Mat(1,2), Mat(1,3)
Mat(2,0), Mat(2,1), Mat(2,2), Mat(2,3)
Mat(3,0), Mat(3,1), Mat(3,2), Mat(3,3)
```

を使うと宣言します。このとき， Mat(I,J) のような式が使えて，例えば

Mat(I,J) **は**　I = 1, J = 2 **のとき**,　Mat(1,2)

となります。纏めると，

配列変数は
　変数名（最大値）：１次元
　変数名（最大値 1, 最大値 2）：２次元
　…
の形のもの。
変数名 (I) は　I = 1 のとき，変数名 (1) を表す。
…
配列変数を使うときは Dim 文で宣言をする。
宣言は
　Dim 変数名（最大値）：１次元
　Dim 変数名（最大値 1, 最大値 2）：２次元
　…
の形

となります。

1.4　配列変数の例：フィボナッチ数の計算

配列変数を使った例を挙げましょう。配列変数は数列の計算が得意ですから，数列の例を挙げます。

ここでは，フィボナッチ数の計算をしてみましょう。
フィボナッチ数 f_n は

$$f_0 = 0,\ f_1 = 1,$$
$$n > 1 \text{ のとき}, f_n = f_{n-2} + f_{n-1}$$

と定義されます。

配列変数を使うと，このような漸化式で定義される数列が簡単に計算できます。

計算する n の最大値が余り大きくないとします。それを予め決めて，その配列を宣言します。そして後は定義に従って順次 $f_0, f_1, \ldots, f_n$ を計算します。

具体的なプログラムを挙げましょう。 プログラムではフィボナッチ数 f_i を Fib(i) と表すことにします。

例 1.1 (フィボナッチ数の計算 1).

```
Dim Fib(50)
Print "フィボナッチ の数"
Input "n (0<=n<=50) を入力して下さい";n
Fib(0)=0
Fib(1)=1
For i=2 to n
  Fib(i)=Fib(i-2)+Fib(i-1)
Next i
Print "Fib(";n;") = ";Fib(n)
End
```

説明をしましょう。説明のために行番号を付けますが，始めの 3 文字は，実際のプログラムには無いことに注意して下さい。

```
01 Dim Fib(50)
02 Print "フィボナッチ数"
03 Input "n (0<=n<=50) を入力して下さい";n
04 Fib(0)=0
05 Fib(1)=1
06 For i=2 to n
07   Fib(i)=Fib(i-2)+Fib(i-1)
08 Next i
09 Print "Fib(";n;") = ";Fib(n)
10 End
```

- 1 行で配列変数 Fib を添え字最大 50 と宣言します。ですから，ここでは Fib(50) が最大です。Fib(50) 以上の計算をするときはそれに合わせて，添え字の最大数を宣言すれば良いわけですが，フィボナッチ数は急速に大きくなる数で，BASIC の精度から，あまり大きな n のフィボナッチ数 f_n を計算することはそもそもできません。
- 4 行と 5 行で初期値 Fib(0) と Fib(1) を定義しています。
- 6 行〜8 行の For 文が実際の計算式です。
  7 行はフィボナッチ数の定義式そのままです。
  For 文は i=2 から始まりますから，最初に Fib(2) を

  ```
  Fib(2)=Fib(0)+Fib(1)
  ```

  を使って計算します。このとき，Fib(0) と Fib(1) は既に与えられています。次の繰り返しでは i=3 で Fib(3) を

  ```
  Fib(3)=Fib(1)+Fib(2)
  ```

  で計算しますが，この時点で， Fib(2)，Fib(1) は与えられていますから，計算することが出来ます。以下同様にして i=n まで計算することが出来ます。

この計算の場合，配列変数の宣言に少し注意が必要ですが，プログラムが定義式と殆ど同じ形に書けることは大変便利です。

一般に漸化式で表される数列の計算は配列変数を使うと簡単ですが，扱う数列によっては 非常に大きな配列が必要なときがあります。BASIC で扱える配列の大きさは 限度がありますから，時には 足りないことがあるかも知れません。そのような場合は，配列変数を使わないで，書く工夫が必要です。

フィボナッチの数の計算するプログラムを見ると，

```
Fib(i)=Fib(i-2)+Fib(i-1)
```

が計算の主要部で，ある項はそれ以前の２項によって計算されています。ですから，順次項を計算する過程は実際は，３つの変数で行われています。そのことに注意すると，配列変数を使わないで，３つの変数を使って書くことが，例えば次のようにできます。

例 1.2 (フィボナッチ数の計算 2).

```
Print "フィボナッチ数"
Input "n (0<=n<=50) を入力して下さい";n
a = 0
b = 1
For i=2 to n
    c = a + b
    a = b
    b = c
Next i
Fib = c
Print "Fib(";n;") = ";Fib
End
```

3 個の変数 a,b,c の役割を 6 行目実行後の時点での f_i との対応を表にしてみると次のようになっています。

	a	b	c
初期	f_0	f_1	
i=2	f_0	f_1	f_2
i=3	f_1	f_2	f_3
	$\cdots$		

7,8 行目で，a,b を 1 つずつずらしています。

このプログラムは配列変数を使ったものに比べると変数の入れ替え操作を行う分，見通しが悪くなっています。ですから，フィボナッチ数の場合のように，添え字の最大値が余り大きくなければ，配列変数を使った方が良いでしょう。tbasic では，かなり大きな宣言も可能ですので，普通は配列変数を使ったプログラムを作るのが良いでしょう。

数列の計算は
まず，配列変数を使って書いてみる。
宣言が大きくなり，できない場合に，それ以外の方法を考える。

2 Read Data 文

ここでは，多くのデータをプログラムに書き込む方法を説明します。

2.1 Read Data 文の目的

配列変数の項では 10 個の数を入力し，それらを各々別の変数に代入するプログラムを考えてみました。そこでは Input 文を使っていましたが，Input 文を使って，入力したデータはプログラムを終了すると消えてしまいます。しかし，時には同じデータを繰り返し利用したいこともあります。

この様な場合，このデータをプログラムの中に書き込んで置けば，プログラムを保存することで，データを保存し，改めてデータを入力しないで，繰り返し利用することが出来ます。

例えば，配列変数を使えば，次のようにできるでしょう。

例 2.1.
```
Dim A(10)
A(1) = 123
A(2) = 313
A(3) = 654
A(4) = 712
A(5) = 145
A(6) = 529
A(7) = 668
A(8) = 887
A(9) = 644
A(10) = 234
…
…
```

こうすれば，10 個の数値データをプログラムの中に書き込むことで，改めてデータを入力しないで，何度も利用することができます。

この方法は簡単で分かり易いものですが，いくつかの弱点を持っています。

- まず，データの数だけの配列変数が必要なことです。1000 個のデータがあれば，例えば Dim A(1000) としなくてはなりません。この場合，1000 個のデータを記述するプログラム領域と，実行時にその分のメモリー領域を両方使用します。データの個数が少なければ，これで良いですが，かなり多くなった場合，避けたい方法です。
- 次に，プログラムの途中にデータを書かなくてはなりません。プログラムの途中に大量のデータがあると，プログラム全体の見通しが，悪くなります。

この様な場合のために，BASIC では，Read Data 文が用意されています。

2.2 Read Data 文

使い方は次の通りです。

まず，使うデータを Data 文で書きます。 Data 文は

Data 定数, 定数,…, 定数

の形で使います。 ここで

- 定数は数値定数，又は文字列定数 いずれでも，又 混在してもかまいません。
- 沢山のデータを書くために，いくつ Data 文を使ってもかまいません。
- Data 文は普通プログラムの最後に置きます。

となります。

次に，プログラムの中で，それらデータが必要な部分でデータを読み出します。
読み出しは，Read 文を使います。 Read は Read (読む) の意味で，Read 文は

Read 変数名, 変数名,…, 変数名

の形で使います。ここで

- Read 文を実行すると，対応する データを指定した変数に代入します。
- 変数名は読み出す変数の型に一致していなくてはなりません。
- 一度 Read 文を実行すると，次に Read 文を実行したとき，対象となるデータは 1 つ先に進みます。

となります。

例を使って説明しましょう。例 2.1 を Read Data 文を使って書き直すと，次のようになります。

例 2.2.

```
Dim A(10)
For I = 1 to 10
   Read A(I)
Next I
…
…
Data 123, 313, 654, 712, 145
Data 529, 668, 887, 644, 234
```

最後の２行がデータです。この２行をまとめて，

```
Data 123, 313, 654, 712, 145, 529, 668, 887, 644, 234
```

としても構いません。

このプログラムでは例 2.1 と同じ内容のプログラムとするため，配列変数が使われていますが，Read Data 文で配列変数は，必須ではありません。例えば，データの内容を表示するだけなら，次のプログラムで可能です。

例 2.3.
```
For I = 1 to 10
   Read A
   Print A
Next I
Data 123, 313, 654, 712, 145, 529, 668, 887, 644, 234
End
```

例 2.2 のプログラム実行の様子を説明しましょう。例 2.2 を行番号を付けて再掲します。

例 2.2.（再掲）
```
01 Dim A(10)
02 For I = 1 to 10
03    Read A(I)
04 Next I
05 …
06 …
07 Data 123, 313, 654, 712, 145
08 Data 529, 668, 887, 644, 234
```

Data 文で書かれたデータはプログラム実行時にコンピュータによって，読み込まれ，最初から順に番号が付けられます。 ですから 7，8 行で

```
Data 123, 313, 654, 712, 145
Data 529, 668, 887, 644, 234
```

とすると，プログラム実行により，コンピュータ内部では

データ番号	1	2	3	4	5	6	7	8	9	10
データの内容	123	313	654	712	145	529	668	887	644	234

と記録されます。

次に Read 文について見てみましょう。Read は

データの番号を表すデータポインター (Pointer :指示子)

と言われるものを，参照しながら実行されます。 まず，

- プログラムを実行すると，ポインターの値は 1 にセットされます。

そして，Read の実行は

- ポインターの値のデータ番号のデータを読み込み，
- ポインターを 1 だけ増やします。

ですから，2，3，4 行の for 文

```
02 For I = 1 to 10
03    Read A(I)
04 Next I
```

を実行すると， Read A(I) では

- I = 1 のとき，ポインター = 1　(実行で 1 にセット) だから，
  A(1) にデータ番号 1 の データ 123 が代入される。

即ち　A(1) = 123 で，ポインターを 1 だけ増やす。即ち ポインター = 2

- I = 2 **のとき**，ポインター = 2　だから，
  A(2) にデータ番号 2 の データ 313 が代入される。
  即ち　A(2) = 313 で，ポインターを 1 だけ増やす。即ち ポインター = 3
- I = 3 のとき，ポインター = 3　だから，
  A(3) にデータ番号 3 の データ 654 が代入される。
  即ち　A(3) = 654 で，ポインターを 1 だけ増やす。即ち ポインター = 4
  …
- I = 10 のとき，ポインター = 10　だから，
  A(10) にデータ番号 10 の データ 234 が代入される。
  即ち A(10) = 234 ポインターを 1 だけ増やす。即ち ポインター = 11

従って，2，3，4 行を行なうと

```
A(1) = 123,  A(2) = 313,  A(3) = 654,  A(4) = 712,  A(5)  = 145,
A(6) = 529,  A(7) = 668,  A(8) = 887,  A(9) = 644,  A(10) = 234
```

が得られることになります。

2.3 Read Data 文の例

別の例を見てみましょう。これは，データが数値と文字列が混在する例です。

例 2.4.

```
' データの表示
Print  ,"名前","著書"
Print
For I = 1 TO 5
   Read N , X$, Y$
   Print N, X$, Y$
Next I
End
'------------------------
Data 1,"アルキメデス","球と円柱"
Data 2,"ガリレオ","新科学対話"
Data 3,"コペルニクス","天体の回転について"
Data 4,"ニュートン","プリンキピア"
Data 5,"ユークリッド","原論"
```

この例では，15 個のデータに対して，変数は 3 つしか使われていないことに，注意して下さい。

Data 文の部分を見ると，

```
Data 1,"アルキメデス","球と円柱"
Data 2,"ガリレオ","新科学対話"
Data 3,"コペルニクス","天体の回転について"
Data 4,"ニュートン","プリンキピア"
Data 5,"ユークリッド","原論"
```

となっています。ですから，コンピュータ内部では

データ番号	1	2	3	4	5
データ内容	1	"アルキメデス"	"球と円柱"	2	"ガリレオ"
データ番号	6	7	8	9	10
データ内容	"新科学対話"	3	"コペルニクス"	"天体の回転について"	4
データ番号	11	12	13	14	15
データ内容	"ニュートン"	"プリンキピア"	5	"ユークリッド"	"原論"

と記録されます。 Read 文の部分を見ると，

```
For I = 1 TO 5
   Read N , X$, Y$
   Print N, X$, Y$
Next I
```

となっています。 ここで，Read 文は 3つに分けて

```
Read N
Read X$
Read Y$
```

としても同じです。

- データが 数値，文字列，文字列 … となっているので，
- 読み込む変数も対応して 数値変数，文字列変数，文字列変数 になっています。

ですから，このプログラム

```
' データの表示
Print  ,"名前","著書"
Print
For I = 1 TO 5
   Read N , X$, Y$
   Print N, X$, Y$
Next I
End
'------------------------
Data 1,"アルキメデス","球と円柱"
Data 2,"ガリレオ","新科学対話"
Data 3,"コペルニクス","天体の回転について"
Data 4,"ニュートン","プリンキピア"
Data 5,"ユークリッド","原論"
```

を実行すると，

```
                名前                 著書

 1              アルキメデス         球と円柱
 2              ガリレオ              新科学対話
 3              コペルニクス         天体の回転について
 4              ニュートン           プリンキピア
 5              ユークリッド         原論
```

と実行画面に表示されます。

以上で，Read Data 文について一通りの説明が終りました。しかし，もう一つ説明しなくてはならないものがあります。

2.4 Restore 文

今までの例ではデータ文のデータを一度しか使いませんでした。でも問題によっては，何回も同じデータを使いたい場合があります。例えば，2 度，3 度と使うのはどうしたら良いのでしょうか。

例として，次の 100 個 のデータの標準偏差を求める問題を考えてみましょう。

```
1,3,4,2,9,4,6,5,8,2
7,7,5,8,5,2,1,9,7,3
6,3,8,4,2,1,0,7,3,4
5,9,6,7,3,2,3,1,4,2
3,4,3,4,8,4,3,7,6,6
6,1,3,5,7,8,2,0,1,7
8,0,4,4,9,3,6,5,0,3
6,8,7,2,5,4,7,2,6,7
4,5,0,5,4,7,9,6,3,5
0,2,1,1,3,8,6,9,2,0
```

標準偏差の定義は次の通りでした。

標準偏差

$X_1, X_2, X_3, \cdots, X_{100}$ を与えられたデータとすると，

$$\text{平均値 } M = \frac{(X_1 + X_2 + X_3 + \cdots + X_{100})}{100}$$

に対して，

$$\text{分散 } S = \frac{(X_1 - M)^2 + (X_2 - M)^2 + \cdots + (X_{100} - M)^2}{100}$$

としたとき，

$$\text{標準偏差} = \sqrt{S}$$

この定義に従って，標準偏差を計算するためには，平均値 M を求めるときと，分散 S を求めるときに，各々データ $X_1, X_2, \ldots X_{100}$ を使います[*4]。

配列変数にデータを読み込ませてあれば，それは何度も使えますから問題はありません。

[*4] 実は，工夫すると 1 度 $X_1, X_2, \ldots X_{100}$ を使うだけで計算することができますが。

例えば次のようなプログラムが書けるでしょう。

例 2.5.

```
'-----------------------------------
Dim A(100)
For I = 1 to 100
   Read A(i)
Next I
M = 0
For I = 1 to 100
   M=M+A(i)
Next I
M = M/100  :'平均
S = 0
For I = 1 to 100
   S = S + (A(i) - M)^2
Next I
S = Sqr(S/100) :'標準偏差
Print S
End

'データ
Data 1,3,4,2,9,4,6,5,8,2
Data 7,7,5,8,5,2,1,9,7,3
Data 6,3,8,4,2,1,0,7,3,4
Data 5,9,6,7,3,2,3,1,4,2
Data 3,4,3,4,8,4,3,7,6,6
Data 6,1,3,5,7,8,2,0,1,7
Data 8,0,4,4,9,3,6,5,0,3
Data 6,8,7,2,5,4,7,2,6,7
Data 4,5,0,5,4,7,9,6,3,5
Data 0,2,1,1,3,8,6,9,2,0
'-----------------------------------
```

このプログラムでは 100 個の配列変数の宣言をしています。100 個程度なら，余りメモリーを使うということではありませんので，配列変数に読み込ませるのが一番簡単です。ですから，今の場合，これで良いのですが。

しかしもっと多くのデータがある場合のことも考えて，余り変数を使わない方法はないでしょうか。

例えば次の様にしたらどうでしょう。

```
'-----------------------------------
M = 0
For I = 1 to 100
   Read A
   M = M + A
Next I
M = M/100  :'平均
S = 0
For I = 1 to 100
   Read A
   S = S + (A - M)^2
Next I
S = Sqr(S/100) :'標準偏差
Print S
End
```

```
' データ
Data 1,3,4,2,9,4,6,5,8,2
Data 7,7,5,8,5,2,1,9,7,3
Data 6,3,8,4,2,1,0,7,3,4
Data 5,9,6,7,3,2,3,1,4,2
Data 3,4,3,4,8,4,3,7,6,6
Data 6,1,3,5,7,8,2,0,1,7
Data 8,0,4,4,9,3,6,5,0,3
Data 6,8,7,2,5,4,7,2,6,7
Data 4,5,0,5,4,7,9,6,3,5
Data 0,2,1,1,3,8,6,9,2,0
'-----------------------------------
```

このプログラムでは，変数は 3 つ A, M, S しか使っていません。これでよいなら良いのですが,

実はこのプログラムは誤りを含んでいます。

それは，データポインターの問題です。

```
For I = 1 TO 100
   Read A
   M = M + A
Next I
```

を実行すると，データポインターは 101 になります。従って，次の

```
For I = 1 to 100
   Read A
   S = S + (A - M)*(A -M)
Next I
```

を実行したとき，I = 1 で Read A を行なうと，データ番号 101 のデータを読みに行きますが，データの個数は 100 個しかありませんから，エラーになります。

どうしたら良いのでしょうか。

実はこの様な場合のために，データポインターを 1 にする文があります。それは Restore です。

Restore は元に戻すの意味で,

```
Restore
```

だけで文になり，データポインターの値を 1 にセットします。

そこで，Restore を前のプログラムにつけ加えると，

例 2.6.

```
'-----------------------------------
M = 0
For I = 1 to 100
   Read A
   M = M + A
Next I
M = M/100  :'平均
S = 0
Restore
For I = 1 to 100
   Read A
   S = S + (A - M)^2
Next I
S = Sqr(S/100) :'標準偏差
Print S
End
'データ
Data 1,3,4,2,9,4,6,5,8,2
Data 7,7,5,8,5,2,1,9,7,3
Data 6,3,8,4,2,1,0,7,3,4
Data 5,9,6,7,3,2,3,1,4,2
Data 3,4,3,4,8,4,3,7,6,6
Data 6,1,3,5,7,8,2,0,1,7
Data 8,0,4,4,9,3,6,5,0,3
Data 6,8,7,2,5,4,7,2,6,7
Data 4,5,0,5,4,7,9,6,3,5
Data 0,2,1,1,3,8,6,9,2,0
'-----------------------------------
```

となり，これで正しく実行されます。

■ Read Data 文のまとめ

纏めると，

プログラムの中に比較的多くのデータを書き込む場合，

- Data 文を使って書く。
- 読むときは Read 文を使う。
- Read ： Read 変数名,…, 変数名 の形で用いて，対応するデータを指定した変数に代入する。
- Data ： Data 定数, 定数,…, 定数 の形で使う。
- Data 文は普通プログラムの最後に置く。
- Restore : データポインターを 1 にセットする。

となります。

3 ラベルと goto 文

N88BASIC を勉強した人は，goto 行番号にお馴染みだと思います。tbasic のプログラムでは行番号を使う必要はありませんが，行番号を使うことも出来ます。またラベルを使用して，goto ラベルという処理を行うことは出来ます。ここではこの説明をします。

ここで説明するものは 「ラベルと goto 文 」,「Restore 文 」です。

3.1 ラベルと goto 文

tbasic のプログラムでは行番号を使うことも出来ますが，これは主に旧 BASIC との互換性のためにものです。行を特定する方法としてはこの外にラベルが用意されていますので，むしろこちらを使うのが良いでしょう。

ラベルはどの言語でも標準的に備えられている機能です。

ラベルは名札と言う意味で，プログラムにおける行を特定します。
tbasic ではラベルは行の先頭で
*ラベル名
の形で使います。ラベル名は数値変数と同様に決めます。

このラベルの使い方は N88BASIC の方法と同様です。ラベルに対して，その場所に実行を無条件で移す goto 文を使うことが出来ます。例を挙げましょう。

例 3.1.
```
*Start
Input "正数を入力して下さい";N
If N< 0 then goto *Start
Print sqr(N)
End
```

- 1 行目の*Start がラベルです。
- 3 行目は，goto ラベル文で，N<0 の場合，*Start に処理を移します。

この例は非負数の入力に対してその平方根を出力するプログラムです。正または 0 を入力した場合，その平方根を表示し，負数を入力した場合，再入力をするものです。3 行目で N の正負を判定し，負の場合第 1 行目 *Start のところに実行を移します。

実は，上のプログラムはラベルを使わないで，次のように書くこともできます。

例 3.2.
```
While N < 0
   Input "正数を入力して下さい";N
Wend
Print sqr(N)
End
```

例 3.1 と例 3.2 はプログラムとしては同じ処理をします。では，どちらのプログラムが良いでしょうか。少し考えてみましょう。

- 例 3.1 と例 3.2 とも正数が入力されるまで最初の 3 行を繰り返します。
- 例 3.2 では While Wend 文が使われています。この While Wend 文は繰り返し文ですから，形を見ただけで最初の 3 行が繰り返されることが分かります。
- 例 3.1 では 1 行目と 3 行目の文の中を解釈して，その結果，その間を繰り返すと言うことが分かります。

この例の場合では，余り違いはありませんが，一般に，意味の解釈は，形の認識に比べて難しい作業です。このことから，形から内容が理解できる，例 3.2 の方が，分かりやすさという面で幾分優れていると言えます。

別の例を見ましょう。次の例 3.3 は 1〜100 を計算するプログラムです。

例 3.3.
```
I = 1
*Kurikaesi
   S=S+I
   I=I+1
If I <= 100 then goto *Kurikaesi
Print S
End
```

これもラベルを使わないで，次のように書くことが出来ます。

例 3.4.
```
For I=1 to 100
   S = S + 1
Next I
Print S
End
```

この場合は明らかに，例４の方が簡潔で見やすいですね。

もう一つ例を挙げましょう。次の例はどのような処理をするものでしょうか。

例 3.5.
```
I = 0
*L1
J = 0
*L2
If J=10 then goto *L3
S=S+I+J*10
J=J+1
goto *L2
*L3
I=I+1
If I<10 then goto *L1
Print S
End
```

ラベルを３つ使っていて複雑です。このプログラムがどのような動作をするか，瞬時に理解するのは殆ど不可能でしょう。また多少時間をかけても理解するのは簡単ではありません。

実はこれもラベルを使わないで書くことが出来ます。

例 3.6.
```
For I=0 to 9
   For J=0 to 9
      S = S + I + 10*J
   Next J
Next I
Print S
End
```

です。これは例 3.5 に比べて驚くべき簡単です。

このようにいくつかの例を見てくると，ラベルと goto 文を組み合わせてプログラムを作る理由が余り無いように見えます。実際,「構造化プログラミング」の考え方では goto 文の使用は極力避けることになっています。

元々初期のプログラミング言語では，If 文と goto 文以外の制御構造文が用意されていませんでした。そのため例 3.5 のようなプログラムが多用されました。このようなプログラムは理解が難しくなり，その結果，誤りやがあった場合の修正や，一部分の変更が困難になります。

このような反省から，goto 文を使わないで，すべてのプログラムが書けるための制御構文が導入され，現在のプログラミング言語となっています。ですから，現在のプログラミング言語では goto 文を使わないでプログラミングすることが可能です。つまり，原理的には

goto 文は必要ない

ということになります。

3.2 goto 文を使って良い場合

では，goto 文は本当に必要ないものなのでしょうか。現在のプログラミング言語で goto 文が残っているのは，古い言語との互換性の為だけでしょうか。

実は必ずしもそうではありません。元々プログラムを分かりやすくというのが，多くの制御構造文を導入する目的でした。しかし，特殊な場合ですが，goto 文を使った方が分かりやすいこともあります。そのような場合，分かりやすさという目的を重視すると，goto 文を使う方がやはり良いのです。

では，どのような場合 goto 文を使ったら良いのでしょうか。それは

例外処理と通常処理の分離には goto 文を使っても良い

ということです。例外処理とは本来目的とする以外の処理です。具体的にはエラー処理等が典型的です。

例外処理は goto 文を使わないでも If then Else 文を使うことで可能です。普通はこの方法で例外処理と通常処理の分離を行うのが良いでしょう。しかし例外状態が多い場合，goto 文で処理した方が良い場合があります。

実際のものではありませんが，例示のための例を挙げます。

ここでは入力用データを読み込んで処理を行うとします。このような場合，色々なエラーが想定されます。まず，データの型が不適である。データの値が不適である。データの個数が不適である。等々です。いずれのエラーの場合でも通常処理は行いません。しかしエラーに応じて別々の処理が必要なこともあります。

まず，goto 文を使って書いてみましょう。

例 3.7 (入力データの読み込み処理).

```
If Err=1 の状況 then
   エラー 1 用の処理
   goto *Quit
End if
If Err=2 の状況 then
   エラー 2 用の処理
   goto *Quit
End if
If Err=3 の状況 then
   エラー 3 用の処理
   goto *Quit
End if
If Err=4 の状況 then
   エラー 4 用の処理
   goto *Quit
End if

' 通常処理
...
*Quit
```

上の例 3.7 のプログラムでは各エラーに従って，独立に処理が記述されています。単純な構造ですが，その状況がよく分かります。

今度もこのプログラムを goto 文を使わないで書いてみます。

例 3.8 (入力データの読み込み処理).

```
If Err=1 の状況 then
   エラー 1 用の処理
Else
   If Err=2 の状況 then
      エラー 2 用の処理
   Else
      If Err=3 の状況 then
         エラー 3 用の処理
      Else
         If Err=4 の状況 then
            エラー 4 用の処理
         Else
            ' 通常処理
            ...
            ...
         End if
      End if
   End if
End if
```

例 3.7 も例 3.8 も機能的には全く同じです。例 3.8 の Else 文以下では，再び If 文が繰り返し使われています。このような場合，If 文で ElseIf を使うと上のプログラムは，より簡単に書けます。

ElseIf は，入門編で説明しませんでした。If 文の構文は，より詳しくは次のようになります。

```
If 論理式 1 then
   文 1
   ...
ElseIf 論理式 2 then
   文 2
   ...
Else
   文 3
   ...
End if
```

の形で使い，

- 論理式 1 が真のとき，文 1... を実行し，
- 論理式 1 が偽で，論理式 2 が真のとき，文 2... を実行し，
- そうでないとき，文 3... を実行します。

Elself は省略しても，いくつか続けても構いません。Else 以下は省略してもかまいません。

この ElseIf 構文を利用すると，上のプログラムは次にのようにも書けます。

例 3.9 (入力データの読み込み処理).

```
If Err=1 の状況 then
   エラー 1 用の処理
ElseIf Err=2 の状況 then
   エラー 2 用の処理
ElseIf Err=3 の状況 then
   エラー 3 用の処理
ElseIf Err=4 の状況 then
   エラー 4 用の処理
Else
    ' 通常処理
    ...
    ...
End if
```

以上 3 つのプログラム例を見てきました。実は

例 3.7，例 3.8，例 3.9 は機能的には全く同じです。
では，どれが一番良いでしょうか。

私は例 3.7 が一番良いと思います。

■例 3.8 について

まず例 3.8 を見てみましょう。元々 If then Else 文では then 以下と Else 以下は大体同じ程度の重要性を持つと言うニュアンスがあります。字下げもそれに対応しています。ですから，例 3.8 の記法を取ると，

通常処理とエラー処理の重要性の視覚的バランスが崩れます。

実際のプログラムではエラー処理 10 行，通常処理 100 行と言ったバランスの場合，深く字下げされた文が 90% にも続くのは，見やすいものではありません。

■例 3.9 について

次に例 3.9 を見てみましょう。例 3.8 に比べて，例 3.9 は大分見やすくなっています。それに行数も例 3.7 に比べて少なくなっています。恐らく，例 3.7 より例 3.9 の方が良いという立場もあるでしょう。ダイクストラの立場に忠実になれば，例 3.9 の方が良いプログラムになります。

しかし，例 3.9 の記法で，私が，気になるのは次の点です。

例外処理と通常処理が，普通の場合分けにより分けられている。

例外処理は例外ですから，普通の処理との独立性の明示があるのが良いと思います。

例 3.9 では，いくつかの例外処理が ElseIf というキーワードで論理的に連鎖しています。この記法よると，エラー 1 の状況と，エラー 2 の状況が論理的に関連している形式になっています。勿論，厳密に考えれば，エラー 1 の状況と，エラー 2 の状況は論理的関連がない筈はありませんから，それを意識的に明示するという意味の積極性がむしろあるとも言えます。

しかし，プログラムの現実の作成状況からすると，この論理的関連の連鎖はプログラムの保守性を低下させると思います。例外の追加や削除はプログラムの進化途中で良くあることです。その場合，例 3.9 に記法ではすべての，例外についてすべて考え直す必要が生じます。

これに対して，例 3.7 の方法だと，それぞれの例外処理がプログラム的にも独立していますから，追加・削除・移動が他の例外と独立に行えることになります。これは後々のプログラム変更の場合，大変負担が少なくなります。以上から，例 3.7 は例 3.9 よりも

冗長性はあるが，そのことが反って，
場合分けの論理的意味の独立性，後の変更での容易さに通じる。

と言えます。

まとめると，

・goto 文は出来るだけ使わない。
・例外処理が多い場合，例外処理と通常処理の分離には使って良い。

ということになります。

3.3 Retore ラベル

上では，goto ラベル 文は出来るだけ使わないと説明しました。ですからラベルも出来るだけ使わないようにと感じるかもしれません。しかしラベルの役割として，tbasic ではもう一つ別なものがあります。それは Restore 文と共に使うものです。

Restore はデータポインタを再設定する命令（文）でした。データポインタを再設定する場合 1 に設定するだけでなく，特定のデータポインタに設定出来ると便利です。tbasic ではこれを Restore ラベルで実現しています。

> Restore は
>
> Restore **ラベル**
>
> の形で使うと，データポインタをラベルの直後のデータに設定します。

例えば，次のような Data 文があったとします。データは全部で 60 個あります。

```
'-----------------------------------
'データ
*DataSet1
Data 1,3,4,2,9,4,6,5,8,2
Data 7,7,5,8,5,2,1,9,7,3
*DataSet2
Data 6,3,8,4,2,1,0,7,3,4
Data 5,9,6,7,3,2,3,1,4,2
*DataSet3
Data 3,4,3,4,8,4,3,7,6,6
Data 6,1,3,5,7,8,2,0,1,7
'-----------------------------------
```

このとき，

- ラベル*DataSet1 直後にあるデータのデータポインターは 1 で，データの値は 1 です。
- ラベル*DataSet2 直後にあるデータのデータポインターは 21 で，データの値は 6 です。
- ラベル*DataSet3 直後にあるデータのデータポインターは 41 で，データの値は 3 です。

ここで，例えば，

```
Restore *DataSet2
```

を実行すると，データポインターは，21 になります。同様に，

```
Restore *DataSet3
```

を実行すると，データポインターは，21 になります。これを利用すると，使うデータを状況に応じて使い分けることが出来ます。

例を挙げましょう。この例は Read Data 文の説明での例を少し変更したものです。このように，1 つのプログラムで 3 種類のデータセットを使い分けることが出来ます。

例 3.10 (データセットの選択).

```
N=20
Input "標準偏差を計算するデータセット番号（1, 2, 3）を入力して下さい",DS
'-----------------------------------
If DS=1 then Restore *DataSet1
If DS=2 then Restore *DataSet2
If DS=3 then Restore *DataSet3
M = 0
For I = 1 to N
   Read A
   M = M + A
Next I
M = M/N  :'平均
If DS=1 then Restore *DataSet1
If DS=2 then Restore *DataSet2
If DS=3 then Restore *DataSet3
S = 0
For I = 1 to N
   Read A
   S = S + (A - M)^2
Next I
S = Sqr(S/N) :'標準偏差
Print S
End
'-----------------------------------
'データ
*DataSet1
Data 1,3,4,2,9,4,6,5,8,2
Data 7,7,5,8,5,2,1,9,7,3
*DataSet2
Data 6,3,8,4,2,1,0,7,3,4
Data 5,9,6,7,3,2,3,1,4,2
*DataSet3
Data 3,4,3,4,8,4,3,7,6,6
Data 6,1,3,5,7,8,2,0,1,7
'-----------------------------------
```

このように Restore ラベルを使うと目的のデータを読む込むことが簡単に出来ます。

まとめると，

> Restore 文は
>
> Restore **ラベル**
>
> の形で使って，ラベルの直後のデータを読み込むことが出来る

となります。

4 BASIC 初級編プログラム例

4.1 ユークリッドの互除法

入門編の基本的なコマンド・文の項目で，次のプログラム例をあげましたが，内容の説明はしませんでした。

```
Print "a と b の最大公約数の計算"
Input "正整数 a = "; a
Input "正整数 b = "; b
While b> 0
   c = a mod b
   a = b
   b = c
Wend
Print "最大公約数 = ";a
End
```

これはユークリッドの互除法を使って，最大公約数を計算するプログラムですが，ここでは，上のプログラムも含めて，ユークリッドの互除法のプログラムについての説明をします。

ユークリッドの互除法は，2 つの数の最大公約数を計算する方法です。その方法は以下の通りです。

ユークリッドの互除法

自然数 a, b $(a > b)$ に対して，a を b で割った時の余りを r とする。以下同様な操作を行い，n 回の除法の定理を適用して，次のようになったとする。$a = r_0$，$b = r_1$，$r = r_2$ として表す。

$$
\begin{array}{lcllll}
r_0 & = & q_1 & \cdot\ r_1 & +\ r_2 & (0 < r_2 < r_1) \\
r_1 & = & q_2 & \cdot\ r_2 & +\ r_3 & (0 < r_3 < r_2) \\
r_2 & = & q_3 & \cdot\ r_3 & +\ r_4 & (0 < r_4 < r_3) \\
 & \cdots & & & & \\
r_{i-2} & = & q_{i-1} & \cdot\ r_{i-1} & +\ r_i & (0 < r_i < r_{i-1}) \\
 & \cdots & & & & \\
r_{n-2} & = & q_{n-1} & \cdot\ r_{n-1} & +\ r_n & (0 < r_n < r_{n-1}) \\
r_{n-1} & = & q_n & \cdot\ r_n & &
\end{array}
$$

このとき，

$$\gcd(a, b) = \gcd(b, r) = \gcd(r_0, r_1) = \cdots = \gcd(r_{n-1}, r_n) = r_n$$

となる。即ち，r_n が a と b の最大公約数になる。

このユークリッドの互除法の数学的な内容については，「プログラミングの背景」に説明がありますので，そちらを参照してください。

上の方法を見ると，r_0, r_1 より始めて，r_n を求めることで，最大公約数を求めています。この形は数列の計算と見ることもできます。数列の計算は配列変数で計算することが出来ますので，このユークリッドの互除法も配列変数を使うのが最も自然です。

プログラム作成方針

・配列変数 r(100) を宣言する。
・r(0)=a，r(1)=b として，
・上の計算方法で r(i) を順次計算する。
・r(i)=0 になったら，そのひとつ前の r(i-1) が a と b の最大公約数

ここで，r(i) は，r(i-2) を r(i-1) で割った余りですから

```
r(i)= r(i-2) mod r(i-1)
```

で求められます。これに注意すればプログラムは，以下の通りに簡単にでき上ります。

例 4.1 (ユークリッドの互除法 1).

```
Dim r(100)
Print "a と b の最大公約数の計算"
Input "正整数 a = "; a
Input "正整数 b = "; b
r(0)= a
r(1)= b
i=1
While r(i)> 0
   i = i + 1
   r(i)= r(i-2) mod r(i-1)
Wend
Print "最大公約数 = ";r(i-1)
End
```

ここで配列変数の宣言が r(100) で十分なことはラメの定理から保証されます。ラメの定理に依れば，計算の最大回数は b の桁数の 5 倍を超えないことが分かります。tbasic で整数として扱える最大数は高々 18 桁ですから，最大で 90 回程度です。従って，100 の宣言で十分です。

また，While 文で r(i)>0 を条件としていますから，繰り返しは，次の i，即ち，i=i+1 として，r(i) を求めます。

さて，上のプログラムを配列変数なしで書いてみましょう。

```
r(i)= r(i-2) mod r(i-1)
```

の関係で，次の剰余が求められますから，a,b,c の 3 つの変数でプログラムが書けることが分かります。この辺りの事情はフィボナッチ数の計算とまったく同様です。そこで，a,b,c に r(i-2),r(i-1),r(i) を対応させることで

```
c = a mod b
```

が得られます。While 文で b>0 を条件としていますから，c の計算後，a,b,c の間の値のずらしを実施します。

最大公約数は，b=0 になったときの一つ手前の a ですから，

```
Print "最大公約数 = ";a
```

となります。これをまとめると，次が得られます。これが入門編であげた例でした。

例 4.2 (ユークリッドの互除法 2).

```
Print "a と b の最大公約数の計算"
Input "正整数 a = "; a
Input "正整数 b = "; b
While b> 0
   c = a mod b
   a = b
   b = c
Wend
Print "最大公約数 = ";a
End
```

となります。

ユークリッドの互除法 2 は，配列変数を使わない分，
どちらかと言うと，ユークリッドの互除法 1 より，高度なプログラムです。

4.2 13 日の曜日を数える

プログラムの構成要素の項で，13 日の曜日を数えるプログラムを具体例としてあげました。そこでは，形式の説明でした。ここでは，そのプログラムとしての内容の説明をします。

まず，暦の性質を確認しましょう。現在使われている暦はグレゴリを暦といわれるものですが，1582 年に使われ始めたそうです。

グレゴリオ暦

・平年を 365 日，閏年を 366 日とする。
・閏年の規則：4 で割り切れる年を閏年とする。
但し 100 で割り切れて，400 では割り切れない年は平年とする。

この規則から，次が得られます。

命題 4.1. 400 年経つと暦が一周する。

証明. 実際，x 年が閏年であるために必要十分条件は，次の (1)，(2) を同時に満たすことになります。

(1) x は 4 で割り切れる。
(2) x が 100 で割り切れるときは，400 でも割り切れる。

従って，$y = x + 400$ とすると，x が上の (1)，(2) を満たすことと，y が同様な条件を満たすことは同じになります。ですから，x 年と，400 年後の y 年は平年，閏年が同じ，即ち同じ日数になります。このことから，どの年から初めて 400 年を調べても，その日数は同じになることが分かります。

そこで，例えば 1 年から 400 年のまでの間の日数を求めてみます。この間にある閏年は，4 の倍数の年で，100 年，200 年，300 年を除いた年ですから，全部で，$100 - 3 = 97$ 回あることになります。ですから，400 年の日数は，

$$365 \times 400 + 97 = 146097 \quad 日$$

になります。ここで，146097 は 7 で割り切れます。7 日で曜日は繰り返されますから，1 年 1 月 1 日の曜日と 401 年 1 月 1 日の曜日は同じになります。1 年と 401 年の平年，閏年，日数は同じですから，1 年と 401 年の各々月日の曜日は一致します。これらのことはすべての年で言えますから，400 年経つと暦が一周することになります。 □

そこで 400 年分 13 日の曜日を計算するすればよい訳です。

計算の仕組み

・400 年間の 13 日の曜日を数える。
・13 日は毎月ある。（閏年関係なし）
・合計 $400 \times 12 = 4800$ 日の曜日を数える。

ですから問題は，曜日の算出です。この計算は昔よくプログラミングの演習問題として出されたものですが，現在では曜日計算の関数が備わっている言語が多くありあます。tbasic でも DayOfWeek **関数**があり，年月を指定すればその曜日を返してくれます。今回はこれを使うことにします。これで全く簡単なプログラミングになります。

プログラム作成の方針

400 年間の毎月 13 日に対して，DayOfWeek 関数を使って曜日を決定し，それを数える。

DayOfWeek 関数は"2014/6/25" のような形式に対して，1 が日曜日,... 7 が土曜日のように返します。

そこで曜日の回数を数えるため，配列変数 DayNum(7) を用意します。
400 年はどこから初めても同じですが，見やすさと親しみから，2001 年から 2400 年としました。
問題は i 年 j 月 13 日を "i/j/13" の形に変換するところですが，Str$ 関数を使いましょう[*5]。

Str$ 関数は 数 *n* を文字列としての n に変換します。
例えば，Str$(123) と " 123"は同じになります。

```
DateStr$ = Str$(i)+"/"+Str$(j)+"/"+Str$(TDay)
DayNumber = DayOfWeek(DateStr$)
```

として曜日を求めます。TDay は対象日で，今は 13 日ですが，他の日（1〜27）でもできるよう変数としました。

これで，曜日が決定できますので，対応する DayNum(i) に 1 を加えることで曜日が数えられます。

```
DayNum(DayNumber)=DayNum(DayNumber)+1
```

これで，本質的なところは終わりです。

[*5] Str$関数で正の数を変換した場合，先頭に半角空白" "が付きます。これはを省くには，Trim$関数を使うと可能です。以下のプログラムでは，先頭の空白が含まれますが，DayOfWeek 関数での処理ではこの部分が無視されますので問題はありません。

表示結果を見やすくするために，

```
Print using "##.##";DayNum(i)/48;
```

としました。"##.##" は 小数点以下２位まで表示する指定です。DayNum(i) を 48 で割るのは，全体が 4800 であることが分かっていますから，その百分率を計算しています。Print using の最後がセミコロン ; になっていることに注意して下さい。

プログラム全体は以下の通りです。

例 4.3.

```
’ １３日の曜日を数える
TDay = 13   :’日にちの指定（1〜27）
Dim DayNum(7)
For i=2001 to 2400
   For j=1 to 12
      DateStr$ = Str$(i)+"/"+Str$(j)+"/"+Str$(TDay)
      DayNumber = DayOfWeek(DateStr$)
      DayNum(DayNumber)=DayNum(DayNumber)+1
   Next j
Next i
Print TDay;"日の曜日の出現割合（%）"
Print "  日     月     火     水     木     金     土"
For i=1 to 7
   Print using "##.##";DayNum(i)/48;
   Print "  ";
Next i
Print
End
```

4.3 抽選プログラム

ここでは，配列変数の応用として，ランダムに与えられたいくつかのものから，ある個数をランダムに選ぶことを考えましょう。これは例えば，

10 個のものの中から，ランダムに 3 個選ぶ

といった問題です。この場合，10 個のものに番号を付けて表せば，この問題は，

1 から 10 までの数のうち，ランダムに 3 つの数を選ぶ

と同じです。ここではこのようにして考えましょう。

まず，1 から 10 までの数のうち 1 つの数を選ぶのは乱数 Rnd を使うと簡単です。Rnd は次の性質を持つ関数です。

Rnd は $0 \leqq x < 1$ となる x をランダムに与えます。

ですから，Rnd*10 の整数部分 Int(Rnd*10) は 0 から 9 までの数をランダムに与えます。

従って，

```
x = Rnd
n = Int(x*10)+1
```

とすれば，n は 1 から 10 までの数をランダムに与えたものになります。

2 個以上の場合は，同様に，さらに数をランダムに取ることで実現できますが，同じ数が出てくることもあるので，少し工夫が必要です。

ひとつの考え方は，同じ数が出てきたら，もう一度取り直すと言う方法です。この方法で，10 個の数からランダムに 3 個の数を選ぶとしまよう。これは，次のように実現できます。

- 選ばれたことを示すために，配列変数を用意し，それに選ばれたことを記入する。
- A(n)=1 なら n は選ばれた数，A(n)=0 なら n は選ばれていないとする。
- 選んだ数 n に対して A(n) = 0 ならまだ選ばれていない数なので，新たに選んだとする。
- A(n) = 0 なら，既に選ばれたものであるので，もう一度行う。
- 最終的に 3 個選ばれたら終わりとする。

これは次のプログラムで実現できます。

```
Dim A(10)
C=0
While C<3
   x = Rnd
   n = Int(x*10)+1
   If A(n)=0 then
      A(n)=1
      C = C + 1
   End if
Wend
For i=1 to 10
   If A(i)=1 then Print i
Next i
```

ここで C は選ばれた個数を表しています。C が 3 になるまで繰り返します。

この方法は簡単に一般化できます。NumofData 個の数から，NumofSel 個の数を選ぶとすると，次のようになります。ここで，NumofData，NumofSel は仮に 300, 100 と設定してあります。

例 4.4.

```
Dim A(1000)
NumofData = 300
NumofSel = 100
C=0
While C < NumofSel
   x = Rnd
   n = Int(x*NumofData)+1
   If A(n)=0 then
      A(n)=1
      C = C + 1
   End if
Wend
For i=1 to NumofData
   If A(i)=1 then Print i
Next i
End
```

これはこれで，一応使えますが，何度も選び直すのが無駄と言えるでしょう。特に，選ぶ個数が全体の個数に近い場合，選び直すことが多くなり，実行に時間がかかる可能性があります。

そこで次に，「既に選んだ数を，取り除いて，残りの中から選んでいく」という方法を考えましょう。

この方法では「取り除く」ことが少し難しい処理です。そこで，配列変数の使い方の視点を変えて，次の方法で考えましょう。

> ・n 個の数を入れてある A(1),..., A(n) に対して，1 個取り除いた結果，
> ・取り除いて残った個数は n-1 で，
> ・A(1),..., A(n-1) には残った数を入れておく

■具体的方法

- n は，1 回取るたびに，1 ずつ減らす。
- A(1),..., A(n) に残った数を入れる方法
  ランダムに，A(1),...,A(n) から 1 つ取り，取り除くものが A(i) としたとき，そのデータを A(n) と入れ替える。

例えば，10 個のものから 3 個選ぶとして，5 番目，3 番目，7 番目の数が選ばれたとします。この場合，

- スタートして，A(1),...,A(10) から 1 つランダムに選ぶ。5 番目が選ばれたら，A(5) と A(10) の値を交換して，n から 1 を引いて $n = 9$ とする。
- 次に A(1),...,A(9) から 1 つランダムに選ぶ。3 番目が選ばれたとして，A(3) と A(9) の値を交換して，n から 1 を引いて $n = 8$ とする。
- 次に，A(1),...,A(8) から 1 つランダムに選ぶ。7 番目が選ばれたとして，A(7) と A(8) の値をを交換して，n から 1 を引いて $n = 7$ とする。
- n=7 となったので，選択終了。選ばれた数は A(8),A(9),A(10) に入っている。

以上のことを表にすると，以下のようになります。

	n	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
スタート時	10	1	2	3	4	5	6	7	8	9	10
A(5) 選択後	9	1	2	3	4	10	6	7	8	9	5
A(3) 選択後	8	1	2	9	4	10	6	7	8	3	5
A(7) 選択後	7	1	2	9	4	10	6	8	7	3	5

上での n の代わりに CNumofData と表し，Sel 番目を選ぶとすると，この処理は以下で実現されます。

```
Sel = Int(Rnd * CNumofData)+1
tmp = A(CNumofData)
A(CNumofData)=A(Sel)
A(Sel) = tmp
CNumofData = CNumofData - 1
```

全体の選択は選ぶ個数 NumofSel 回実行すれば良い訳ですから，以下で実現されます。

```
CNumofData = NumofData
For i=1 to NumofSel
   Sel = Int(Rnd * CNumofData)+1
   tmp = A(CNumofData)
   A(CNumofData)=A(Sel)
   A(Sel) = tmp
   CNumofData = CNumofData - 1
Next i
```

これを，プログラムに纏めると，次のようになります。

例 4.5.

```
' 抽選プログラム
Dim A(1000),B(1000)

' ----------  入力  ----------------------------
Input "データ数，抽選数を入力してください。",NumofData, NumofSel
If  ((NumofData > 1000) or (NumofData< NumofSel)) then
   Print "データ数，抽選数が不適です。"
   End
End if
'----------  初期化  -------------------------
For i=1 to NumofData
   A(i)=i
Next i
'----------  抽選  -------------------------
CNumofData = NumofData
For i=1 to NumofSel
   Sel = Int(Rnd * CNumofData)+1
   tmp = A(CNumofData)
   A(CNumofData)=A(Sel)
   A(Sel) = tmp
   CNumofData = CNumofData - 1
Next i
'----------  結果表示  -------------------------
Print NumofData, NumofSel
Print "選ばれたデータ番号"
For i= NumofData-NumofSel + 1 to NumofData
   B(A(i))=1
Next i
For i=1 to NumofData
   if B(i)=1 then print i
Next i
End
```

最後の結果表示は，

```
For i= NumofData-NumofSel + 1 to NumofData
   Print A(i)
Next i
```

でも構いませんが，この場合大小関係はばらばらです。そこで，上のプログラムでは表示が小さい順になるようにしています。配列変数 B(i) はそのためのものです。

また入力部分は少し複雑ですが，これは入力ボックスで 100,30 と言った形式で入力するための工夫です。