構造化プログラミング

tbasic.org [*1]

[2018 年 4 月版]

> 我々が，プログラマーは正しいプログラムを書くだけでなく，その正しさを分かり易く示すべきであるという立場を取れば，上の考察は，プログラマーの活動に深い影響与えます。即ち，プログラマーが作り上げる対象は十分に構造化 (usefully structured) されていなくてはなりません。
>
> （E. W. Dijkstra, Notes on Structured Programming,1969)

Tiny Basic for Windows 入門編，初級編，ファイル操作編，グラフィック操作編では，tbasic でプログラムを書く基本的な方法を説明しました。ここでは少し進んで，良いプログラムを書く方法として広く知られている「構造化プログラミング」について考えてみましょう。

目次

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

1　構造化プログラミング

構造化プログラミングは 1960 年代後半から 1970 年頃にかけて，E.W. ダイクストラ達によって提唱されたプログラミングについての考え方です。この構造化プログラミングは具体的なプログラミング言語を対象として，明確に定義された方法ではありません。当時，アルゴリズム記述言語として使われていた，ALGOL 60 や 68，また実際のプログラミングに広く利用され，米国規格となった FORTRAN 66 などの言語を想定したプログラミングでの基本的姿勢の提唱でした。この主張は提唱当時から広く受け入れられ，多くの人々によって議論が深められ，検証され，その結果，プログラミングにおける基本的原則として認められました。そしてこの構造化プログラミングは，以後のプログラミング手法や，プログラミング言語の開発などに大きな影響を与えました。

他方，この考え方が提唱された当時に比べると，現在はプログラミング言語自身大きく進歩し，ダイクストラ達が提唱した状況とは大きく変化しています。しかしそれにも関わらずこの考え方は今でも有効です。それは現在のプログラミング技法や手法は色々ありますが，それらは全てこの考え方の延長線上にあるからです。またこの考え方は余りにも基本的なものとして受け入れられているために，これが意識されず当然のこととして扱われていることもあります。

ここではこの「構造化プログラミング」の考え方を，tbasic でプログラミングを行うことを例にして，簡単にまとめてみましょう[*2][*3]。

1.1　構造化プログラミングの目的

1940 年代後半にコンピュータが出現した当初は，プログラミングはすべて機械語で行われ，プログラムは難解な機械語を理解する，少数の専門家達によって書かれていました。また，機械語によるプログラムは，動作するコンピュータに依存するため，新しいコンピュータが開発される度に，新たにプログラムを書き直す必要がありました。そのことから，大きなプログラムや多くのプログラムを書き，利用するには限界がありました。

その後，幾種類ものコンピュータが開発され，特に 1950 年代に入ると商用のコンピュータが開発が始まりました。それに伴い，具体的コンピュータに依存しない，使いやすい高級言語への模索が始まりました[*4]。

[*2] 上述のように「構造化プログラミング」は特定の言語を想定して提唱された訳ではありません。主に当時よく使われていた手続き型言語を想定していました。ですから以下の説明は，適当に変更することで tbasic に限らず，現在使われている多くの言語に適用することができます。

[*3] ダイクストラ自身の提唱は原著の日本語訳

「構造化プログラミング」ダイクストラ他（サイエンス社，昭和 50 年）

によって読むことができます。

残念ながらこの書籍は現在絶版になっていますが，少し大きな図書館では利用できるでしょう。中古の書籍は入手可能のようですし，原著 Structured Programming (A.P.I.C. studies in data processing, no. 8) はまだ新品が手に入るようです。

また，この本にあるダイクストラの記述の基になったテキスト 'Notes on Structured Programming' は

E. W. Dijkstra Archive：http://www.cs.utexas.edu/users/EWD/

から無料で利用可能です。

[*4] 高級言語は数学で使う式に似た表現が使えるなど，プログラムをより人間に分かりやすい表現で記述可能なプログラム言語。処理を具体的コンピュータの装置に対してでなく，より抽象的な対象に対して行う形式のため，コンピュータに依存しない性質を持つ。そのため，新たなコンピュータを開発する度に，すべてプログラムを書き直さなくても，その代わりに対応する高級言語のコンパイラを開発するだけで，以前書かれた多くのプログラム資産を活用できることになります。

特に 1956 年の FORTRAN，1960 年の ALGOL など本格的高級言語の出現によりプログラミングが容易になっただけでなく，コンピュータに依存しないプログラムが書けるようになりました。

ことから，プログラマーの人数や，それらの人々が書くプログラムの量が飛躍的に増加しました。更に，コンピュータの性能の向上に伴い，それらのプログラムは次第に大規模になり[*5]，それを作るための費用も時間も大きくなっていきました。

そのような状況の中で，プログラムを使い捨てではなく，長期にわたって利用する一つの資産として活用できる「良いプログラム」に仕上げる方法が模索され始めました。

50 行くらいのプログラムであれば，「良いプログラムを書こう」といった標語や基本姿勢だけで，特別なプログラミング手法を意識しなくてもプログラムを書くことは出来ます。一度にプログラム全体を見渡して，色々検討すれば良いプログラムにすることも出来るかもしれません。しかし，その 100 倍である 5000 行のプログラムを書こうとすると，そのような方法ではうまく行きません。ですから，数万行から数十万行から構成されるプログラムを良いプログラムに仕上げるには，プログラムの作成手法について深い分析や検討が必要になります[*6] 。

プログラムを有用な資産として活用するためには次の問題に対応する必要があります。

- それらを如何に正しく動作するプログラムとして仕上げるか。
- 大規模なプログラムを如何に見通しよく管理するか。
- また後々，そのプログラムのある部分を修正する必要が生じたとき，別の人々によって，如何に間違いなく可能か。
- 更に，プログラムのある部分を後で如何に再利用可能か。

このような問題意識のもとで「良い」大規模なプログラムを書く為の手法の検討が，構造化プログラミング提唱の出発点で，一つの回答でした。

大規模なプログラムは，そのままで扱うのは困難です。そこで，構造化プログラミングでの，基本的な考え方は，

分割して統治せよ。(Divide and Rule)

です。即ち，「大規模なプログラムを可能な限り，分割し，各段階でのまとまりを検証可能な程度の行数のプログラムとし，それらの処理を，定められた理解しやすい構造の処理の連鎖として表現する。これにより，上記の「良いプログラム」とすることができる。」とする主張です。

[*5] ここで想定されている大規模なプログラムとは，複数の人々が作る数万行にも及ぶものです。

[*6] 私達が現在念頭にある tbasic のプログラムは高々数百行です。それでは高々数百行のプログラムを書く場合，そのような手法が必要なのでしょうか？答えは Yes，つまり必要ですし，有効です。即ち，「**構造化プログラミングは 100 行程度のプログラムを書く上でも有効** 」と言えます。構造化プログラミングは大規模プログラムのために考え出された手法ですが，それはプログラムを書く上での基本的作法，および言語の備えるべき仕様と言ったものを意味します。ですから，この態度は

小さいプログラムを書く場合でも有効ですし，考慮すべきもの

なのです。

このことをもう少し具体的に表すと，次のようになります。

- プログラムの目的を定め，それを出発点として，段階的にプログラムを精密化，構成する[*7]。
- 基本的制御構造のみを利用し，それらを組み合わせてプログラムを構成する。
- 処理を細分する際，処理のまとまりは抽象化し，他の処理と独立に動作するようにする。
- 細分された処理は階層的に見通し良く配置する。

これだけでは，単なる方針ですが，構造化プログラミング提唱の意義は，これらをより具体的に規定したことにあります。これらについては次項以降で詳しく説明します。

1.2 良いプログラムとは

構造化プログラミングは大規模なプログラムを，「良い」プログラムに仕上げるものと言いました。では良いプログラムとはどんなプログラムのことでしょう。似た問題を「プログラムとは」の項で既に一度取り上げました。そこでは

> 良いプログラマーは正しいプログラムを書くだけでなく，
> それが正しく動作することを，分かりやすく証明する必要がある

と書きました。この表現は実は「良いプログラムとは」に，同様に言い換えることが出来ます。即ち，良いプログラムとは

- 正しい動作をする。
- それが正しく動作することが，分かりやすく，良く理解できる 。

といえます。勿論この他に，「高速で動作する。」ということが必要ですが，これは少し別な問題なので，ここでは触れません。

そこで，「正しい動作をする」と「それが正しく動作することが，分かりやすく，良く理解できる」を考えてみましょう。「正しい動作をする」は「それが正しく動作することを理解」して始めて確認できます。ですから，(分かりやすく，良くということを除いて,) 上の二つは実質，殆ど同じことを意味しているとも言えます。

それでは，正しく動作することを理解するには，どうしたら良いでしょうか？「実行してみて，正しく動作すると確認する」ということも考えられます。しかしこの方法は，一見もっともですが，余り現実的ではありません。実際，多くのプログラムでは，起き得る全ての可能性について，試すことは不可能です。またもし，全ての可能性について容易に確認できる程度のものであれば，そもそもプログラムを書いて計算機に実行させる理由は余りありません。

とすると，何でもって正しく動作すると確認するのでしょうか。実行結果を見てでないとすると，それは，「プログラムの内容を見て論証で確認する 」しか無いのです。プログラムの実行は動的なもので，人間はもともと，動的なものを見て，正確に理解・確認することは不得意です。しかし，プログラム自体は文書として表現された静的なものです。その静的なものを正確に見て，それを論証により理解することで「正しく動作する」ことを確認できるのです。

[*7] ダイクストラはこのことを Step-wise Program Composition と読んでいます。また，Step-wise Refinement とも言われています。トップダウン型ソフトウエア開発と言っても良いでしょう。

そしてその論証がしやすいような構造を持つプログラムを作ろうと言うのが，構造化プログラミングの考えです。つまり主張していることは，

> 正しい動作をすることが，分かりやすく理解できるためには，
> プログラムが分かりやすく，理解しやすい構造を持つ必要がある。

ということで，そのために，

> 十分に構造化されたプログラムを書こう。

という主張です。

では構造化するとはどういうことでしょうか。

以下，構造化プログラミングでの構造化を理解するため，プログラムを構造的に書く方法を説明していきましょう。

1.3 プログラム理解ための道具

その方法を規定するために，まず，私達がプログラムを理解するときに，使う道具を把握する必要があります。つまり

> 私達はどのような方法で，プログラムを理解しているのか，或いは理解可能なのか

ということです。ダイクストラは上述の小論で，プログラムを理解するための道具として次の三つを挙げています。以下ではこれらについて説明します。

- 数え上げ
- 数学的帰納法
- 抽象化

1.3.1 数え上げ

数え上げは，具体的対象を順次列挙し，数え上げて，それぞれの場合にそのプログラムの動作を確認・証明し，正当性を理解する方法です。一通りではなく，幾つもの可能性のあるときも，ありうる可能性が有限個の場合は，それぞれの場合をすべて列挙し，検証することで，正当性を確認することができます。この方法は，プログラムに限らず，色々なものを確認する最も基本的で確実な方法でしょう。

簡単な tbasic プログラムの例を挙げます。以降特に断らない限り，プログラムはすべて tbasic 用のものです。ここで，プログラムに行番号が書かれていますが，これは説明のためのもので，実際のプログラムでは不要です。またこれ以降，例で書かれている行番号は特に断りのない限り，説明用で，実際のプログラムでは不要です。

例 1.3.1 数え上げ

```
10 Input "一辺の長さ N >0 を入力して下さい。", N
20 NS=N*N
30 Print "一辺が";N;"の正方形の面積は"; NS;"です。"
40 End
```

このプログラムは，一辺の長さを入力して，その面積を計算し表示するものです。このプログラムは全部で 4 行，すべて単一の処理をしていますので，このプログラムの正当性は，各行について，確認すれば証明されます。10 行，20 行，30 行と順に処理が実行され[*8]，それぞれの行が，言語の仕様[*9]から，目的通りの処理をすることが確認でき，プログラム全体として，目的通り動作することが検証されます。

場合分けの例は，次項数学的帰納法での例の中にあります。

1.3.2 数学的帰納法

数学的帰納法は，高校の数学の教科書でお馴染みの方法ですが，自然数における性質を証明する一般的方法です。ある事柄の全ての場合を尽くすのではなく，ある場合とその次の場合との関係を調べ，一般的な場合へと帰納する方法です。帰納法と言う名称ですが，論証の分類からは，「演繹法」に属します。

プログラムが扱う対象は必ずしも自然数だけではありませんが，プログラムを処理の集まりとみたとき，それは離散的な集まりで，数学的帰納法の適用できる対象です。実際のプログラムの中では，数え上げでは場合を尽くせないとき，適用します。

例えば，繰り返しにおいて，その部分が確かに目的通り動作するかどうかの確認のときなどに適用する方法です。一つの例を挙げましょう。

例 1.3.2 数学的帰納法

```
10 Input "大きな自然数>0 を入力して下さい。", M
20 n=1
30 i=0
40 While n < M
50    n=n*2
60    i=i+1
70 Wend
80 Print "2^";i;"乗 =";n;" >= ";M;" > 2^";i-1; "乗 =";n/2;"です。"
90 End
```

このプログラムは入力された自然数 $M > 0$ に対して，$2^i \geqq M$ となる最初の自然数 i，即ち，

$$2^{i-1} < M \leqq 2^i = n \qquad (*)$$

となる，n, i を求めるものです。プログラムは，$n = 1$ より始めて $n < M$ の条件を満たす間，順次 n を 2 倍し，初めて条件を満たさなくなったとき，$n = 2^i \geqq M$ となり，これが求めるものです。この説明で，正当性が良く確認できたとするのなら，これで終わりです。

しかし，ここでは更に厳密な説明・証明が必要とされるとしましょう。

まず，M は実行時に入力され，どのくらいの数であることは分かりませんから，具体的な場合分けでは証明できません。そこで，$M = 1, 2, 3, \ldots$ についての数学的帰納法を用います。

[*8] 40 行は，プログラムの終了を示す標識ですから，プログラムの実行を終了し，特別な処理はしません。

[*9] 今の場合は tbasic の仕様

証明

(1) $M = 1$ のとき n, i の初期値はそれぞれ $1, 0$ だから，この場合，$n = 1 = M$ となり 40 行 `While n<M` の条件は不成立である。従って，`While ... Wend` ルーチンは実行されず，プログラム終了時は $n = 1, i = 0$ となる。故にこのとき，$2^{0-1} = \frac{1}{2} < M = 1 \leqq 1 = 2^0 = n$ であり (*) は成立する。

(2) 次に，$M = M_0$ のとき，$n = n_0, i = i_0$ に対して，(*) が成立，即ち，

$$2^{i_0-1} < M_0 \leqq 2^{i_0} = n_0 \tag{*0}$$

と仮定する。このとき，プログラムで $M = M_0 + 1$ が入力されたとし，その場合，プログラムの実行により得られる $n = n_1, i = i_1$ に対して，

$$2^{i_1-1} < M_0 + 1 \leqq 2^{i_1} = n_1 \tag{*1}$$

が成立することを示す。

まず，40 行の条件式 `While n < M` は `While n < M0+1` であり，条件 $0 < n < M_0 + 1$ は

$$0 < n < M_0 \qquad \text{と} \qquad M_0 \leqq n < M_0 + 1$$

の 2 つの場合に分けられる。ここで，後者の条件式は，n が自然数であることに注意すれば，$n = M_0$ と書き直すことができる。また，n は単調増加だから，$M = M_0 + 1$ のときの，`While ... Wend` ルーチンは次のように分割できる。

`M=M0+1` のときの，`While ... Wend` ルーチンの分割

```
While n < M0+1
   n=n*2
   i=i+1
Wend
```

(P1)

$\Longrightarrow$

```
While n < M0
   n=n*2
   i=i+1
Wend
'---------
While n = M0
   n=n*2
   i=i+1
Wend
```

(P1')

ここで，(P1') での前半の `While ... Wend` ルーチンは `M=M0` での `While ... Wend` ルーチンに他ならないから，このルーチンの終了後は帰納法の仮定より，$n = n_0, i = i_0$ に対して (*0) を満たしている。

そこで，(*0) の後半の不等式を場合分けをして考える。

(i) $M_0 < 2^{i_0} = n_0$ の場合。このとき，n_0 は自然数だから $M_0 + 1 \leqq 2^{i_0} = n_0$ である。他方，この場合，$n_0 \neq M_0$ だから，(P1') での後半の `While ... Wend` ルーチンは実行されず，(P1') 終了後，$n = n_1 = n_0, i = i_1 = i_0$ となっている。このとき，$M_0 + 1 \leqq 2^{i_0} = n_0$ だったから，(*1) が成立する。

(ii) $M_0 = 2^{i_0} = n_0$ の場合。この場合，(P1') での後半の `While ... Wend` ルーチンは実行され，$n_1 = 2n_0, i_1 = i + 1$ となる。このとき，

$$2^{i_1-1} = 2^{i_0} < 2^{i_0} + 1 = M_0 + 1 < 2M_0 = 2 \cdot 2^{i_0} = 2^{i_1} = n_1$$

だから，この場合も，(*1) が成立する。

以上により，例 1.3.2 が目的通り正しく動作することが証明された。 □

1.3.3 抽象化

これは，私達が物事を認識する場合の基本的方法ですが，プログラムを理解する上でも重要な要素です。

> 事柄の本質的部分とそうでない部分とを分離し，
> その本質的部分を抜き出すことを抽象

といいます。しかし，実は何が本質的で，何がそうでないかを見極めるのは大変難しいことです。その意味で抽象化の方法を理解してから，プログラミングするというのは現実的ではありません。むしろ，実際のプログラミングを通して，抽象化を理解していくと考えた方が良いでしょう。このような意味から，プログラミングの勉強は単にプログラムを書くためだけでは無く，物事を深く理解するための一つの道であるとも言えます。

プログラムは対象とする言語の仕様に基づいて記述された文書ですから，これ自体抽象的な存在です。従って，厳格に考えれば，プログラムの構成要素すべてが抽象的存在です。プログラミングを行うことき，このことは常に意識している訳ではありませんが，言語マニュアルをよく読み，言語要素の性質を理解することは実は，抽象的思考を通して行っていることになります。また，後で説明する構造化されたプログラムを作る作業では，さらに進んで抽象的なものを作ることが求められます。

ここでは，プログラムの構成要素での例を一つ上げましょう。それは変数 (variable) です。変数は，プログラムにおける基本要素で，数やデータを保存等する入れ物ですが，現実のコンピュータの中に変数の実体があるわけではありません。元々，初期の機械語によるプログラミングでは，データはその型に応じて，使用するコンピュータのメモリの番地を指定して，保存等していました。この指定方法は個々のコンピュータに依存していたので，機械語によるプログラムは，異なる機種のコンピュータで動作させるためにはその都度書き換える必要がありました。これに対して，変数は具体的メモリの果たす役割を「数やデータを保存等する入れ物」として，抽象化し，実現していて，その指定方法はメモリの番地でなく変数名です。この指定方法は個々のコンピュータに依存せず，従って異なる機種のコンピュータであっても，その言語が対応していれば書き直す必要はありません[*10]。

更に，この「データの保存場所としての」変数は数学で使われる変数のように演算処理の対象としても扱われます。これは，保存されているデータ（変数の値）に対する演算処理として実現されています。保存されるデータは色々と変化しますから，変数はメモリ指定の抽象化を超えて，色々変化する数あるいはデータの抽象化としても実現されています。

上述の数学的帰納法の項での変数 `n`，`i`，`M` についての記述が，例えば，$n = 1$ であったり $n = n_0$ であったり，また $n = n_1$ であったりしましたが，これらを表すものとして変数 `n` があったわけです。`n` は一つの文字ですが，それは無限の可能性のある数の本質を抜き出して表しています。このような抽象化があって始めて，**無限の可能性のある入力値に対するプログラムを数行と言う有限の文字で記述できる**，上の例のようなプログラムを書くことができます。

プログラムに現れる抽象化の実際を，次のプログラム作成を例にして見てみましょう。以下は構造化プログラミングでのプログラムの作成の流れの例にもなっています。

[*10] 勿論，この機能は高級言語（コンパイラやインタプリタ）で実現されますが，実行コードは最終的には機械語で実現されます。従って，対応コンパイラやインタプリタは，異なる機種のコンピュータに対しては別々に作成しなければなりません。この別々に作成する部分が変数の個々のコンピュータの違いを吸収し，抽象的存在である変数を実現します。

例 1.3.3 抽象化　ある英文のテキストファイルを指定して，それに含まれるアルファベット A〜Z の個数を数え[*11]，それらの頻度を計算する。

(1) テキストファイルの取り扱い

テキストファイルは書式無し文書で，文字の並びですが[*12]，文書を行で分割し，行の集まりと見るか，文書全体を文字の一まとまりとして見るかによってプログラムでの扱いが異なります。行の集まりと見ると，文字列配列変数に格納されたデータとして扱うのが普通ですし，文書全体を文字の一まとまりとして見ると，単一の文字列変数に格納されたデータとして扱うのが普通です。ここでは，文書全体を文字の一まとまりとして見て，単一の文字列変数を使うことにします。

いずれにしても，テキストファイルは，何らかの文字列変数との対応で処理します。文字列変数や文字列配列変数の使い方はいろいろ考えられますが，この場合は，テキストファイルに対応する抽象的対象として解釈して利用します。

ここでは，読み込むテキストファイルとして `const.txt` を用意し，そのテキストファイルを格納する文字列変数として，`TText` を使うことにします。

(2) 度数表・頻度表の取り扱い

アルファベット A〜Z の出現数を数え，その頻度を調べることでしたから，例えば，目的は具体的に次のような結果の表[*13]を求めることかもしれません。

A	B	C	D	E	F	G	H	I	J	K	L	M
2675	612	1164	1230	5107	1021	444	2029	2433	96	53	1490	730
7.4%	1.7%	3.2%	3.4%	14.1%	2.8%	1.2%	5.6%	6.7%	0.3%	0.1%	4.1%	2.0%
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
2630	2729	767	46	2206	2676	3751	848	460	375	97	504	31
7.3%	7.5%	2.1%	0.1%	6.1%	7.4%	10.4%	2.3%	1.3%	1.0%	0.3%	1.4%	0.1%

このような表をプログラミングで実現するには，数値配列変数の利用が考えられます。配列変数のインデックス[*14]は非負整数のみですので，少し工夫が必要ですが，

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

といった対応関係を決めれば可能です。今の場合も，数値配列変数を度数表の抽象的対象として解釈して処理を行います。各度数を全体数で割ると頻度が得られます。

ここでは，配列変数 `FreqA2Z` を用意し `FreqA2Z(1)`:`A` の度数，⋯，`FreqA2Z(26)`:`Z` の度数と解釈して利用することにします。

(3) 度数表作成

[*11] ここで大文字小文字の違いは無視し，すべて大文字として扱うことにします。

[*12] 実はこれ自体も抽象的対象です。

[*13] これは `https://www.usconstitution.net/` にある米国憲法の Plain-text 版 const.txt を修正なしで調べた結果です。この版には Note が含まれているため，原文と少しだけ異なります。

[*14] 例えば，`A(i)` での `i` は `0, 1, 2, ...`のみ可。

テキストファイルと度数表の用意ができたので，テキストファイルからアルファベットの度数を実際に求める処理を考えます。この処理はテキストファイルの各文字に対して，それらがアルファベットのどの文字であるか調べて，それに対応する配列変数の度数を 1 だけ増やせば良いだけです。ですから，各文字に対して，それがアルファベットの場合，上の対応表の番号を返す関数があれば，度数表作成は可能です。そこで，各文字に対して，上の対応表の番号を返すユーザー定義関数 OrderA2Z を作りましょう。ここでは，OrderA2Z は次の仕様とします。

OrderA2Z の仕様

文字 Ch$に対して，Ch$="A"なら，OrderA2Z(Ch$) は 1 を返し，Ch$="B"なら，2 を返し，…，Ch$="Z"なら，26 を返し，それ以外は 0 を返す。

この OrderA2Z は，ASCII 表では，"A"〜"Z"が順に並んでいることを利用し，ASCII 関数から作ることができます。関数の作り方の詳細は 3.3 **副プログラムの作り方，使い方**で説明します。

ユーザー定義関数は，ユーザーが必要と考える機能を抽出して，仕様を定め，新たに作成する抽象的存在です。ユーザーが自由に作成するものですから，自由度が高い反面，如何に目的に適したものにするかにその効果が掛かっています。ユーザー定義関数は，後で説明する副プログラムの一つで，構造化プログラミングでの重要な抽象化の対象です。

(4) 結果の表示処理

度数表での総和を求めて，各度数を総和で割ると頻度が計算されます。A から Z まで，各々対応するアルファベット，度数，頻度を表示すると，結果が表示されます。

以上をまとめると次のような処理の擬似プログラムができます。

```
10  ’英文ファイルに含まれる A〜Z の出現度数，頻度の計算
20  ファイル用文字列変数，度数表を設定する
30  文字列変数に目的のファイルを格納する。
40  度数表を作成する。
50  度数表から頻度を計算し表示する。
60 End
```

この擬似プログラムの各項目を，実際のプログラミング言語の仕様に従ってコード化し，実際のプログラムを作成します。以下，tbasic を使って実際にコード化してみます。

```
10  ’英文ファイルに含まれる A〜Z の出現度数，頻度の計算
11  ChDir GetProgramDir
```

11 行はファイル読み込みをカレントディレクトリから行うための設定です。tbasic ではこの方法を推奨しています。ここで，読み込み対象のテキストファイルは，この処理プログラムと同じフォルダに保存されているとします。

```
20  ’ファイル用文字列変数，度数表を設定する
21  Dim TText as string
22  Dim FreqA2Z(26)
```

21 行は TText の変数宣言です。tbasic では普通変数は宣言しなくても使えますが，Dim を使って文字列変数として宣言することもできます。22 行で度数表を配列変数として宣言します。変数名は FreqA2Z としまし

た。FreqA2Z(i) (i=0, 1, 2, ..., 26) の初期値は 0 です。

```
30  ’文字列変数に目的のファイルを格納する。
31  ’ const.txt はこのプログラムと同じフォルダに保存しておくこと
32  TText = ReadAllText("const.txt")
33  LText = Len(TText)
```

31 行で，ReadAllText を使って目的のファイル，今の場合は const.txt を文字列変数 TText に読み込んでいます。32 行はその長さを数値変数 LText に設定しています。const.txt をこのプログラムと同じフォルダに保存することと 11 行の ChDir GetProgramDir でこのプログラムの保存フォルダをカレントディレクトリにすることで，ファイル名だけで const.txt を読み込むことができます。

```
40  ’度数表を作成する。
41  For i= 1 to LText
42     Ch$ = Mid$(TText,i,1)
43     OrdChar = OrderA2Z(Ch$)
44     FreqA2Z(OrdChar)=FreqA2Z(OrdChar)+1
45  Next i
```

41 行から 45 行が度数表の作成です。ユーザー定義関数 OrderA2Z を使っています。42 行で TText から 1 文字 Ch$を取り出し，43 行でその文字の番号を取得し，44 行でその番号の配列変数に度数を 1 だけ加えています。この処理を For 文で 1 から LText まで繰り返し，TText 全体を調べます。FreqA2Z(0) には，アルファベット以外の文字数の合計が設定されます。

```
50  ’度数表から頻度を計算し表示する。
51  Total = 0
52  For i=1 to 26
53     Total =Total+FreqA2Z(i)
54  Next i
55  For i=1 to 26
56     Print Chr(i+Asc("A")-1); ":"; FreqA2Z(i); " :"; Round((FreqA2Z(i)/Total)*100,1);"%"
57  Next i
60 End
```

51 行から 57 行が度数表から頻度計算，その表示部分です。51 行から 54 行でアルファベットの文字数の合計 Total を求めています。Total 計算は度数表の作成の部分でも可能ですが，その場合，効率が多少落ちます。55 行から 57 行が度数表，頻度の表示部分です。

```
70  Function OrderA2Z(Ch$) ’ A:1, ..., Z:26 を返す。それ以外の文字は 0 を返す。
71     Ch$=Ucase$(Ch$)
72     If ((Asc(Ch$)>= Asc("A")) and (Asc(Ch$)<= Asc("Z"))) then
73        OrderA2Z = Asc(Ch$)- Asc("A")+1
74     Else
75        OrderA2Z = 0
76     End If
77  End Function
```

関数 OrderA2Z は上のようになります。70 行から 77 行が OrderA2Z の定義部分です，71 行で引数 Ch$を大文字に変換し，それが A から Z の範囲にある場合，Ch$の A からの位置を返し，それ以外は 0 を返します。

以上を纏めると目的のプログラムとなります。

1.4 構造化言語

構造化プログラミングが可能な言語，構造化言語について簡単に説明します。元々，構造化プログラミングそのものの明確な定義がありませんから，構造化言語の明確な定義，備えておくべき仕様があるわけではありません。ここでは，一応の目安としての機能を挙げると，次になります。

- 基本的制御構造

  プログラムは適当な変換を行った後，概ねプログラムに書かれた順序でコンピュータのメモリに配置されます。実行は基本的に上から下への順序に従って，実行されます。このような構文を順構造文と言います。しかし，この方法だけだと実行する命令の数だけの命令をプログラムに書く必要があり，非効率的です。このことから，コンピュータが開発された極めて初期の段階から，命令の実行はプログラムに書かれた順序だけでなく，条件に応じて適当な位置にジャンプする仕組みが作られていました。この条件ジャンプを使えば複雑な実行順序を自由自在に制御できました。

  しかし，1950 年代後半から高級言語が開発されるようになり，次第に大きなプログラムが書かれるようになりると，条件ジャンプだけではなく，よく使う実行制御に便利な構造文として，分岐構造文，繰り返し構造文も導入されました。

  それらの中で順構造文，分岐構造文，繰り返し構造文は，それらで記述されたプログラムとそのプログラム実行との関係が，明確で，動作の正当性の検証が行いやすく，構造化プログラミングで重要な役割を果たします。これらの構文で構成されるプログラム構造を，基本的制御構造と言います。

- 副プログラム（ユーザー定義手続きやユーザー定義関数）

  プログラミングで画面表示など，いくつかの処理は定型的で，多くのプログラムで同じように利用します。そのような定型的な処理プログラムを，プログラムを作る度に新たに書くのは効率的ではありません。そこで，標準的で定型的な処理は，予め小プログラムを作っておいて，プログラムの中でそれらを随時利用する仕組みが考え出されました。そのような予め作成された定型的な小プログラムはライブラリと言われ，初期のアセンブラでも多く作成されました。

  1950 年代後半から利用され始めた高級言語では，それらのライブラリは初めから備えられているものとして，命令，内蔵関数という形で簡単に利用できるようになりました。また，それに加えてユーザーが新たなライブラリを作成できる仕組みとして，ユーザー定義関数，ユーザー定義手続きといったものも備えられました。

  ユーザー定義関数やユーザー定義手続きは，副プログラムとも言われます。副プログラムは，他のプログラムから呼ばれてから動作をします。これに対して，主プログラムはプログラム実行後直ちに実行されます。

  元々は，プログラムは主プログラムのみから構成されていましたが，プログラムが大きくなるにつれて，主プログラムのある部分の纏まりを，副プログラムとして別に記述するようになりました。これにより，プログラム全体の見通しが良くなり，同時にまた副プログラムは主プログラムとは独立に作成・管理・保守することができるようになりました。

  主プログラムはプログラミングの目的に応じて，その度に作成する必要がありますが，副プログラムは，主プログラムや別の副プログラムと独立にプログラミングされ，ユーザーのライブラリとして，別のプログラムでも利用することができます。

この副プログラムは，構造化プログラミングでの「分割して統治せよ」の考え方を実現する仕組みとして，重要な役割を果たします。

- 副プログラムでの変数の局所性

  主プログラムはしばしば，副プログラムとは別に書かれます。ですから，主プログラムで副プログラムを利用するとき，副プログラムの仕様[*15]だけに注目して，その中身については考えなくても良いようにできる[*16]ことは重要なことです。特に，主プログラムで使用される変数名と副プログラムで使用されている変数名とが同じものであったとき，同じ変数として扱われると混乱を招きます。それを避けるために主プログラムでは，副プログラム内部で使われている変数名とは別の変数名を使う必要があります。しかし，副プログラム内部で使用されている変数名を知ることは，副プログラムの中身を調べることになり，副プログラムの独立性の趣旨を損なうものです。そこで副プログラムの性質として次があると好都合です。

  > 副プログラム内部で使われた変数は，
  > 別の主プログラムや別の副プログラム内部で
  > 使われた同じ名前の変数があっても，実行上は別な変数として扱われる。

  このような性質を持つとき，副プログラム内部の変数は**局所的**或いは `private` であると言います。この局所性は，副プログラムの保守管理と主プログラムからの利用について大変好都合な性質です。構造化プログラミングでもこの性質は重要な役割を果たします。

- ユーザー定義変数型

  複雑なデータや，多くの対象を扱ったりする場合，多くの変数を一纏まりにして処理したほうがプログラムが見やすくなることがあります。いくつかの変数の組を一つの新たな変数の型としてユーザーが定義できる言語があります[*17]。この機能を使うと，多くの変数を必要とされるプログラムでも，保守しやすいプログラムを書くことができます。

以上挙げたこれらの機能があれば，構造化プログラミングが可能と言えますが，これらの機能がすべて備わってはいなければ不可能という訳ではありません。実際，構造化プログラミングは，態度・マナー・手法の提唱ですから，その考え方沿ったプログラミングに心掛ければ，アセンブラ以外の高級言語では程度の差こそあれ，殆ど可能と言えます。

次節以降これらについて，少し詳しく説明します。

[*15] その目的，使い方

[*16] ブラックボックス化

[*17] このようにして定義された型を構造体と言うことがあります。オブジェクト指向言語では，より進化したクラスと言われる型もあります。

2　基本的制御構造

この節では，基本的制御構造について，具体例を挙げながら詳しく説明します。

構造化プログラミングの目標として「分かりやすい構造をもつプログラムを書こう」がありました。では分かりやすい構造とはどんな構造でしょう。

プログラムはコンピュータ上で実行することを目的としたものですから，プログラムは現実のコンピュータの動作状況を簡単に把握できるような構造を持つことが必要です。そしてコンピュータの動作は時間と共に進みますから，その経過をプログラム上で容易に追うことが出来なければなりません。ですから， プログラムの構造は

> コンピュータの時間的状態変化との
> 明示的な対応関係をもつ必要がある。

と言えます。更に，それが正しい動作をすることを確認する必要がありますから，

> その処理の正当性を確認できる枠組みを持つ必要がある。

とも言えます。

コンピュータの動作から見れば，時間的経過による状態変化のみが動作状況ですが，プログラムの側からみると，動作状況はプログラムの中での対応する処理と，記述されている量の値です。記述されている量の値とその処理の意味から，次の処理が決まります。この決めることを制御と言い，その構造を制御構造と言います。ですから，分かりやすい構造を持つためには，コンピュータの動作とプログラムの対応関係を明示でき，その動作の正当性が確認できる制御構造を使用することが必要です。

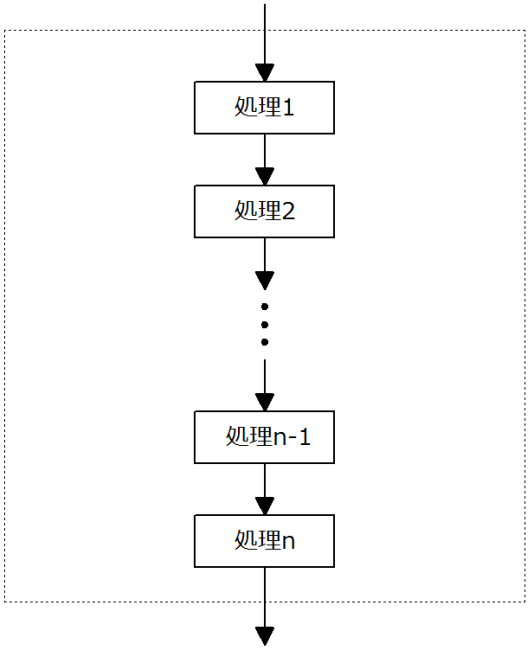
構造化プログラミングでは次の３種類の制御構造を使うと規定しています。

- 順構造
- 分岐構造
- 繰り返し構造

これらの構造で記述されたプログラムでは，コンピュータの動作の追跡と，正当性の検証が容易に可能となります。

2.1 順構造

一番単純な構造は，プログラムの記述とコンピュータの動作経過が一致ものです。それは順構造と言われるものです。



順構造は図のように上から下へと順にプログラムが記述され，コンピュータでの実行とプログラムでの**処理**が平行的に進みます[*18]。ですから，コンピュータの実行状況から，プログラムでどの部分の処理が行われているか良く分かります。また，その正当性は，上述の，数え上げの方法で処理１，処理２，...，処理 n-1，処理 n を順次確認していくことで得られます。

プログラムを，入力，計算処理，出力と３つに分けたとすると，これは３つの処理からなる順構造を持つと考えられます。

簡単な例を挙げましょう。次のプログラムは，入力された自然数 $N > 0$ 値に対して，1〜N 迄の和 S

$$S = 1 + 2 + \cdots + N$$

を出力するものです。

例 2.1.1 順構造

```
10   Input "自然数 N >0 を入力して下さい。", N
20   S=N*(N+1)/2
30   Print "1〜N 迄の和は";S;"です。"
40   End
```

[*18] ここで，処理とは，コンピュータに対する指示の一つのまとまりで，それ自身がいくつかの順構造に分割されることも，また以下に説明する，多分岐構造や繰り返し構造の処理の集まりに分割されることもあります。

10 行が入力，20 行が計算処理，30 行が出力になります。入力と出力の実行状態は外から見て分かります。またある時点でプログラムの実行を停止したとするとき，そこでの処理が 20 行が実施される前か後かは，変数 S の値を確認することで分かります。

更に，このプログラムが正しく動作することは，10 行と 30 行の処理については，対象とするプログラミング言語の仕様をマニュアルで確認することでできます。ここでは tbasic の仕様から確認できます。20 行の処理は

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

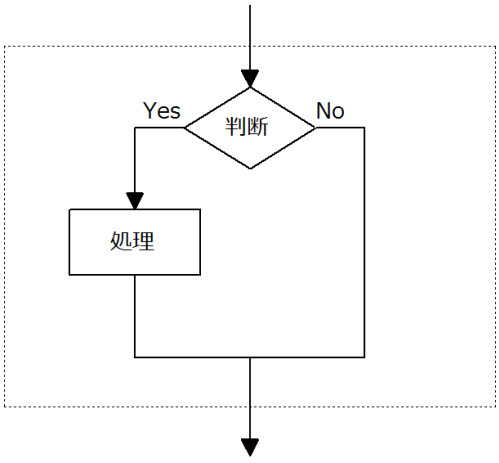
を公式として認めるか，或いは数学的帰納法で証明することによって確認できます。

2.2 分岐構造

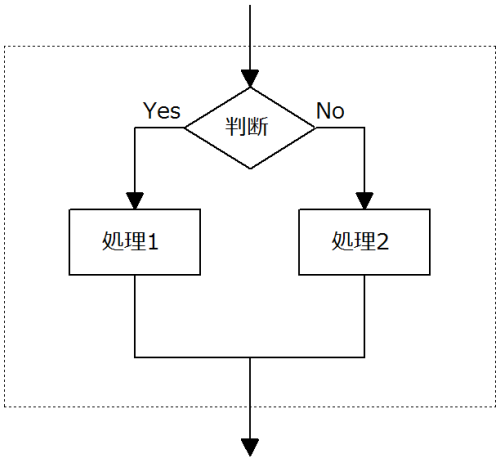
プログラムを大きな枠組みで考えると，順構造を持つと考えられますが，細かく見るとその処理が全て順構造を持っているとは限りません。現実的処理では順構造では実現できないものもあります。

実際の問題を解決する構造は色々と考えることが出来るでしょうが，構造化プログラミングでは，プログラムの中で使う制御構造を，上の順構造の他に，基本的にもう２つ，「分岐構造」と「繰り返し構造 」に限定します。

分岐構造は，まず，２分岐構造として，次の図で表されるものです。



２分岐構造１



２分岐構造２

2.2.1 ２分岐構造１

左の構造（２分岐構造１）は，ある条件を判断してそれが満たされれば，処理を行い，そうでない場合は何もしないで次に進みます。この構造は

～（判断）なら，～（処理）をする

と表現されます。ここで，判断の部分は値が真理値となる論理式です。この判断部分が非常に複雑な場合，一行の論理式として表されない場合もあるかもしれません。その場合は，別に論理変数を定義して論理式を計算して使うか，ユーザー定義関数（Function）を使って別に定義し，それを使って一行に収まる論理式にし

ます。

これに対して，ここでの処理部分は一行とは限らず，仕様上は何行でも構いません。勿論，余り多くの行数を書くのは，分かりやすいプログラムを書くということから言えば，好ましくありません。一画面に納まる程度にするのが適切です。それ以上の行数になる場合は，以下に説明する Sub などを使って，いくつかの処理に纏めます。

この構造では，判断の真理値が真の場合，処理が実行され，それ以外は何もしません。この構造の正当性は判断部分の論理式が言語の仕様に従って正しく表現されていることの確認と，処理の正当性の確認で示されます。

tbasic の構文で表すと， `If ～ then ～ End if` 文

```
If ～ （判断）then
   ～ （処理）
End if
```

がこれに当たります[*19]。

例を挙げましょう。次のプログラム例 2.2.1 は例 2.1.1 と目的は同じですが，N が自然数 $N > 0$ の場合だけ和を計算し，値を出力します。そうでない場合，何もしません。

これに対して，例 2.1.1 のプログラムでは，自然数 $N > 0$ ではない N を入力した場合，不適切な結果が表示されます。

例 2.2.1 ２分岐構造

```
10   Input "自然数 N >0 を入力して下さい。", N
20   If (N>0) and (Int(N)=N)  then
30      S=N*(N+1)/2
40      Print "1～N 迄の和は";S;"です。"
50   End if
60   End
```

ここで，10 行，20 行～50 行をそれぞれ一つの処理と見るの二つの処理からなる順構造をしています。従って，順構造の正当性の確認方法から 10 行の正当性と 20 行～50 行の正当性をそれぞれ確認することで全体の正当性が確認できます。

20 行～50 行の処理が２分岐構造 1 の構造をしています。

20 行の `(N>0) and (Int(N)=N)` が**判断**のための論理式です。この式は，tbasic の仕様から，N が正でかつ，N が整数であるときのみ真になりますから，N が自然数 $N > 0$ の場合だけを正しく表現しています。

30 行と 40 行が２分岐構造 1 での**処理**に当たる部分です。この部分が**処理**であることを明示するために文頭に空白が挿入されています。[*20]。この処理の正当性は例 2.1.1 で確認した通りです。

50 行は If 文の終わりを示す標識としての行ですから，実行には関係ありません。

[*19] tbasic の `If` 文にはこれ以外に 1 行で記述する `If （判断） then （判断）` の構文もあります。この構文は 1 行で記述し，`Enf if` がありません。この構文を単一行 `If` 文または，1 行 `If` 文と言うこともあります。これに対して，本項で挙げた `If ～ then ～ End if` 文は（処理）の部分が何行になっても良い性質があります。このことから，複数行 `If` 文または，ブロック `If` 文と言うこともあります。

[*20] このように文頭にいくつか空白を挿入することを**字下げ（indent）**と言います。字下げはいくつかの処理をまとめて明示するために使われ，その重要性は構造化プログラミングが提唱される頃には広く認識されていました。構造化プログラミングでも，字下げを適宜使用して，構造を見やすくすることが求められます。

2.2.2 ２分岐構造２

右の構造（２分岐構造２）は，ある条件を判断してそれが満たされれば，処理１を行い，そうでない場合は処理２を行い，次に進みます。

この構造は

～（判断）なら，～（処理１）をし，そうでなければ～（処理２）をする

と表現されます。２分岐構造１は，**（処理２）**を無くしたものですから，何もしない処理も可とすると[*21]，２分岐構造１は２分岐構造２の特殊な場合と考えられます。

この構造の正当性は判断部分の論理式が言語の仕様に従って正しく表現されていることの確認と，論理式の真，偽に応じて２つの場合分けを考え，それぞれの場合にそれぞれの処理の正当性の確認で示されます。

tbasic の構文で表すと， `If ～ then ～ Else ～ End if` 文

```
If ～ （判断） then
   ～ （処理１）
Else
   ～ （処理２）
End if
```

がこれに当たります[*22]。

例を挙げましょう。次のプログラム例 2.2.2 は例 2.1.1，2.2.1 のものと目的は同じですが，入力された N が自然数 $N>0$ であるかチェックしています。N が自然数 $N>0$ の場合だけ和を計算し，値を出力します。そうでない場合，「計算できません」と出力します。

例 2.2.2 ２分岐構造

```
10   Input "自然数 N >0 を入力して下さい。", N
20   If (N>0) and (Int(N)=N)  then
30      S=N*(N+1)/2
40      Print "1～N 迄の和は";S;"です。"
50   Else
60      Print "N が正整数でないと計算できません。"
70   End if
80   End
```

20 行～70 行の処理が２分岐構造２の構造をしています。

20 行の `(N>0) and (Int(N)=N)` が**判断**のための論理式です。

30 行と 40 行が２分岐構造２での**処理１**に当たり，60 行が**処理２**に当たる部分です。これらの処理の正当性の確認は例 2.2.1 と同様です。

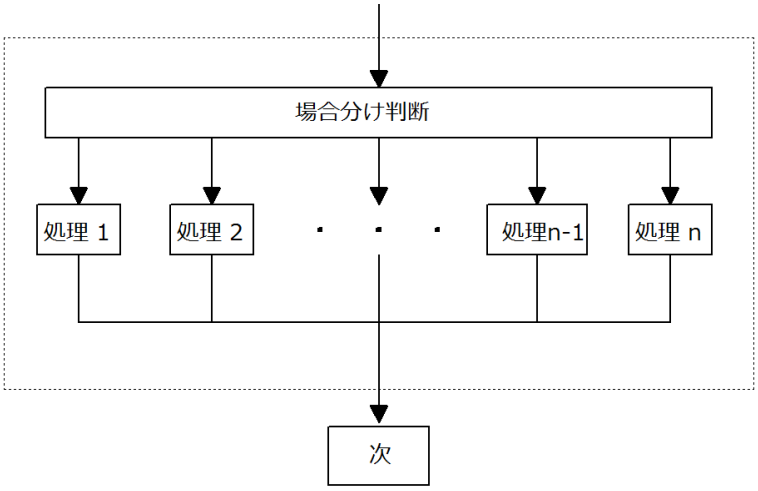
50 行と 70 行は If 文の区切りと終わりを示す標識としての行ですから，実行には関係ありません。

[*21] tbasic では空行も可能ですから，何もしない処理も処理と考えられます。

[*22] tbasic では，この形の `If` 文に対しても同様に単一行 `If` 文
`If ～ then ～ else ～`
の構文もあります。本項で挙げた `If` 文を同様に複数行またはブロック `If` 文と言うこともあります。

2.2.3 多分岐構造

分岐構造としは，これらを更に一般化した，多岐分岐構造があります。これは，次の図で表されます。



これは場合分け判断に応じて，それぞれの処理を行います。この構造は

～（場合分け判断）に応じて，それぞれ，
～（処理１），～（処理２），...，～（処理 n-1），～（処理 n）をする

と表現されます。

これらの構造の正当性は，各々起き得る場合それぞれについて，確認をしていく必要がありますが，これらの場合分けは有限通りですから，可能で，数え上げの方法で理解・確認されます。また処理の流れは，場合分け判断での値を確認することで，どの（処理 i ）が行われるか分かります。

この構造での，場合分け判断は３つ以上の場合分けになりますから，論理式の真偽ではできません。数値式，または文字列式の値によって判断されます。この判断の仕方は，プログラム言語によって，色々規定されていますので，使用する言語の仕様をよく確認する必要があります。

tbasic では，この構造を表すものとして，Select Case 文があります。tbasic でも様々な場合分けの方法がありますので，詳細はヘルプ（仕様書）で確認してください。ここでは最も標準的な構文を挙げます。場合分け判断では，それ以外という判断も考えられます。それを表すために，上の構文を少し拡張した，以下の Select Case 文が基本です。

```
Select Case 判断項目
   Case 場合 1
      処理 1
   Case 場合 2
      処理 2
     ...
   Case 場合 n-1
      処理 n-1
   Case 場合 n
      処理 n
   Case else
      処理それ以外
End Select
```

ここで，「Case else，処理それ以外」の部分は省略可能ですから，省略したものが，上の図のものです。

■じゃんけんプログラム１

例を挙げましょう。じゃんけんのプログラムです。コンピュータとユーザ（あなた）がじゃんけんの勝負をします。コンピュータは乱数を使って出し手を決め，ユーザは Input 文で出し手を入力します。

簡単のためまず，Case else を省略した例を挙げましょう。

例 2.2.3 じゃんけんプログラム１

```
010  Dim HShape$(3)
020  HShape$(1)="グー": HShape$(2)="チョキ": HShape$(3)="パー"
030  Randomize
040  CHS=Int(RND*3)+1
050  Input "1:グー，2:チョキ，3:パーを入力して下さい。",YHS
060  Print "コンピュータ：";HShape$(CHS);"，あなた：";HShape$(YHS)
070  Select case (CHS-YHS+3) mod 3
080      Case 0
090         Print "アイコ。"
100      Case 1
110         Print "あなたの勝ち。"
120      Case 2
130         Print "コンピュータの勝ち。"
140  End Select
150  End
```

10 行から，60 行までが前処理で，順構造をしています。70 行から 140 行が Select Case 文です。

前処理では，グー，チョキ，パーをそれぞれ 1,2,3 に割り当て，設定，入力，計算をします。出し手に対応して，20 行で，HShape$(i) に名前を対応させます。40 行でコンピュータの出し手 CHS を決めています。50 行であなたの出し手を入力します。

勝敗の判定は，場合分けでも可能ですが，計算で判定するには，差 (CHS-YHS+3) を mod 3 で取ったとき，0 ならアイコ，1 ならあなたの勝ち，2 ならコンピュータの勝ちとなることに注意します。

実際，グー，チョキ，パーでの勝敗は差 0,1,2,-1,-2 で決まり，mod 3 で非負としたとき，差 0,1,2 で決まるので，(CHS-YHS+3) mod 3 で上のような判定結果が得られるます。

このことは，厳密には，場合分けして，次のように表にしてみると，明確に分かります。

コンピュータの手	あなたの手	CHS	YHS	CHS-YHS	(CHS-YHS+3) mod 3	勝負の結果
グー	グー	1	1	0	0	アイコ
グー	チョキ	1	2	−1	2	コンピュータの勝ち
グー	パー	1	3	−2	1	あなたの勝ち
チョキ	グー	2	1	1	1	あなたの勝ち
チョキ	チョキ	2	2	0	0	アイコ
チョキ	パー	2	3	−1	2	コンピュータの勝ち
パー	グー	3	1	2	2	コンピュータの勝ち
パー	チョキ	3	2	1	1	あなたの勝ち
パー	パー	3	3	0	0	アイコ

その結果に基づいて，0, 1, 2 でそれぞれの場合について，80-90 行，100-110 行，120-130 行の Case 文で処理をしています。

140 行は　Select 文の終わりを示す標識ですので，何も実行はしません。

このプログラムは，入力された値が 1,2,3 であれば，正しい結果を表示しますから，これで良いのですが，もし，それ以外の値が入力された場合，どうなるでしょうか。試してみると分かりますが，入力する値に従って，エラーになったり，エラーにならず勝敗が表示されたりします。元々，1,2,3 を入力するように指示しているので，「仕方ない」とすることも考えられますが，もっと適切な対応があればよいとも言えます。

これは**入力処理**の問題で，実はプログラミングでは一つの重要な問題です。今の例位のプログラムでは，何の問題もありませんが，大きなプログラムの場合，想定された以外の値が入力された場合の処理を適切に行わないと，プログラムの異常処理が起きて，間違った結果になったり，エラーによる異常終了になったりします。ですから，少し大きなプログラムや，頻繁に使うプログラムでは，指定した値以外の値が入力された場合の処理も常に行うのが良いでしょう。つまり

ユーザーからの入力が伴うプログラムでは，
指定した値以外が入力された場合の処理も必要

となります。

■じゃんけんプログラム２（入力処理付）

上のプログラムに入力処理を追加してみましょう。ここでは文字列入力[*23]に変更し，G，g，T，t，P，p のいずれか 1 文字の入力をするものとします。大文字でも小文字でも良いことにします。それ以外の文字列が入力された場合は，その旨の表示（エラー等の表示）をするものとします。

これを実現する方法は色々考えられます。標準的には，入力文字列の長さが 1 であるかを調べ，その文字が，G，g，T，t，P，p のいずれかであるかを確認し，そうであれば，通常処理を行い，そうでなければ，エラー等の表示をするという方法が考えられます。この方法は，If 文または Select Case 文を使うことで，実現できますが，入れ子構造[*24]になります。入れ子構造は構造化プログラミングでも許される構造ですので，プログラムとしては全く問題はありませんが，多少大がかりな感じもします。

ここでは，多少技巧的ですが別な方法による例を次に挙げます。

アイデアは単純です。ユーザによる入力文字列が，条件に適した場合は，それぞれ出し手に対応してユーザ出し手数 YHS に 1, 2, 3 を設定し，それ以外は 0 に設定するような処理を施します。（この具体的方法は後述します。）そして，変数 EFlag [*25] として，出し手数 YHS が 1,2,3 の場合は 0 を返し，YHS が 0 の場合は，比較的大きなエラー数（今の場合は 6）を返すようにします。これは，EFlag=(1-YHS)*(2-YHS)*(3-YHS) で実現されます。これを実現したプログラムが次です。

[*23] 浮動小数点数の入力処理はいくつか難しい問題があります。tbasic の数値は浮動小数点数のみですので，確実な入力処理を行う場合，文字列を使うのが良いでしょう。
[*24] 今の場合，If または Select Case の処理の中に更に Select Case 文が入ります。
[*25] エラーフラグ

例 2.2.4 じゃんけんプログラム２（入力処理付）

```
010  Dim HShape$(3)
020  HShape$(0)="???":HShape$(1)="グー": HShape$(2)="チョキ": HShape$(3)="パー"
030  Randomize
040  CHS=Int(RND*3)+1
050  Input "g:グー，t:チョキ，p:パーを入力して下さい。",YH$
060  YH$=LCase$(YH$)
070  YHS = InStr(1,"GgTtPp",YH$)/2
080  EFlag = (1-YHS)*(2-YHS)*(3-YHS)
090  Print "コンピュータ：";HShape$(CHS);"，あなた：";HShape$(YHS)
100  Select case ((CHS-YHS+3) mod 3) + EFlag
110      Case 0
120         Print "アイコ。"
130      Case 1
140         Print "あなたの勝ち。"
150      Case 2
160         Print "コンピュータの勝ち。"
170      Case Else
180         Print "g,t,p のいずれかを入力して下さい。"
190  End Select
200  End
```

20 行で，YHS が 0 の場合，即ち，g, t, p 一文字以外を入力した場合，??? を表示するように，HShape$(0) を追加設定します。30 行，40 行は例 2.2.3 と同じです。50 行で，g, t, p の 1 文字を入力するように指示をします。60 行では，入力文字列を小文字に変換します[*26]。70 行が少し技巧的な処理をしています。小文字 YH$ に対して，InStr(1,"GgTtPp",YH$)/2 を求めると，YH$ が，g, t, p である場合，それぞれ 1, 2, 3 になり，それ以外は 0 になります。ここで，文字列"GgTtPp"は少し技巧的ですが，文字列"GgTtPp"に含まれる G, T, P は実は大文字であればどのような文字でも構いません[*27]。

このように YHS を計算して，((YHS-CHS+3) mod 3) + EFlag を計算すると，指定された文字入力であった場合，勝負の結果に応じて 0, 1, 2 になり，指定外の文字列入力であった場合には，6, 7, 8 のいずれかになります。この数値による分岐条件を Select Case 文で記述したものが，110 行〜180 行になります。

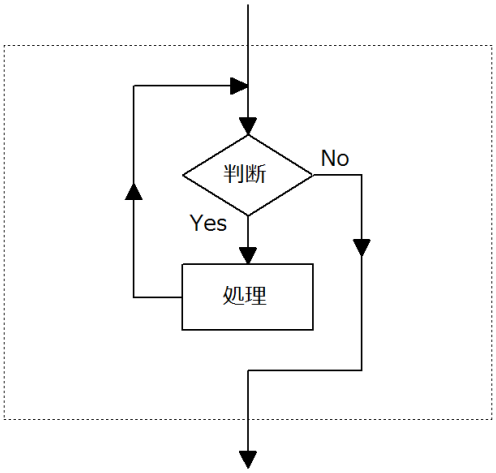
2.3 繰り返し構造

繰り返し構造は次の図で表されるもので，型としては２種類あります。ここでは，繰り返し構造１，繰り返し構造２と表すことにします。

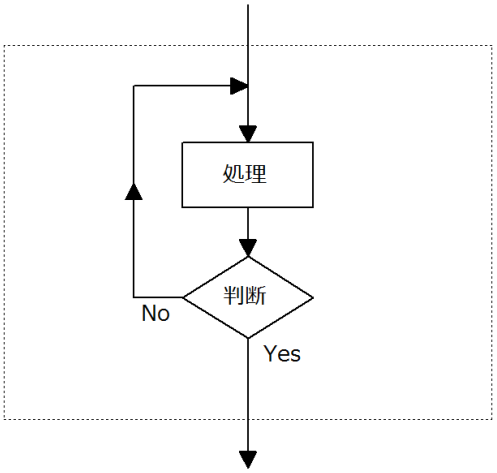
これらの構造での処理の回数は個々の判断・処理の内容で決まり，構造そのものからは決まりません。従ってそれらの処理を場合分けで全て尽くし，それらの正当性を確認することは常にできるとは限りません。この構造の正当性は，判断条件・処理によって証明方法は異なりますが，一般的には数学的帰納法を使えば確認することができます。

[*26] 大文字，小文字いずれでも良い場合，場合分けを減らすために，小文字（または大文字に）に変換して処理する方法はよく使われます。

[*27] また，YHS=InStr(1,"gtp",YH$) としない理由は，このようにすると例えば tp と 2 文字入力した場合も 2 を返してエラーにならないからです。



繰り返し構造１



繰り返し構造２

2.3.1 繰り返し構造１

繰り返し構造１では，まず＜**判断**＞をします，Yes なら処理を行い，再び＜**判断**＞を行います。この処理を No の判断がでるまで順次続けます。No の判断が出たら，この処理を終えて，次の処理に進みます。

この構造は

～（判断）である間，～（処理）をする

と表現されます。

tbasic の構文で表すと，`While Wend` 文

```
While ～ （判断）
   ～ （処理）
Wend
```

がこれに当たります[*28]。また，`For Next` 文もこの構造に準じたものです。実際，

```
For i=a to b
   ～ （処理）
Next i
```

は，これと同値な処理が

```
i=a
While i <= b （判断）
   ～ （処理）
  i = i + 1
Wend
```

と表され，順構造と繰り返し構造の合成になっています。

`While Wend` 文や `For Next` 文は Basic 入門のシリーズで既に多くの例がありますので，ここでは，簡単な `For Next` 文の例を一つあげます[*29]。

[*28] これ以外でも，`Do While` 文，`Do Until` 文があります。これについてはヘルプを参照して下さい。

[*29] `For Next` は繰り返す回数が定まっている場合に使います。`While Wend` は色々な場合に使えますが，分かりやすさの面から，`For`

次のプログラム例 2.3.1 は入力された文字列 In$ に対して，その逆順の文字列 Out$ を求めて表示するものです。

例 2.3.1 逆順文字列

```
10 Input "文字列を入力してください",In$
20 L=Len(In$)
30 Out$=""
40 For i=1 to L
50    Out$ = Mid$(In$,i,1) + Out$
60 Next i
70 Print L;"文字逆順文字列："; Out$
80 End
```

10 行で文字列 In$ を入力します。40 行〜60 行が繰り返し構文です。50 行で，In$ の左から i 番目の 1 文字を取り，Out$ の左に追加します。この操作を行うと，In$ の対象文字が，Out$ に，順番が入れ替わり追加されます。この繰り返し構文を L 回実行すると，In$ の左端の文字（左から 1 番目の文字）は Out$ で右端となり，In$ の右端の文字（左から L 番目の文字）は Out$ で左端となり，最終的に Out$ は In$ の文字列を逆順に並べた文字列になります。

この説明でこのプログラムの正当性は十分示されたと考えることもできます。しかしまだ多少の曖昧性があり，更に厳密な証明が必要かもしれません。ここでは，より厳密な証明が求められたとして，数学的帰納法を用いて，証明をしてみましょう[*30]。

■例 2.3.1 のプログラムの一般化

例 2.3.1 のプログラムで，帰納法の対象となる変数は入力文字列 In$ の長さ Len(In$)=L ですが，L についての帰納法で示すには，例 1.3.2 で While 文に対して行った条件の分割と同じような For 文の条件の分割だけでは無理で，For 文の中身，50 行の書き換えが必要となり，論理的に煩雑です。そこで，少し一般化した問題を考え，その問題を数学的帰納法で証明し，その特殊な場合として例 2.3.1 のプログラムの正当性を示すことにします。少し大げさですが，論理的には簡明な証明です。

例 2.3.1 では文字列 In$ 全体の逆順文字列を構成しましたが，ここでは，その部分列を構成します。即ち，任意に固定された正整数 L と 1 ≦ N ≦ L となる任意の整数 N に対して，Len(In$)=L となる文字列 In$ の左から N 文字を Out$ に逆順に出力する問題，例 2.3.1'P(L,N) を考えます。L=N のときが，例 2.3.1 のプログラムですから，この例 2.3.1'P(L,N) の動作の正当性が示され，例 2.3.1'P(L,N) が L=M のとき，例 2.3.1 と同値なら，例 2.3.1 のプログラムの正当性が示されることになります。

例 2.3.1'P(L,N) の作り方は簡単で，以下の通りです。

本質的な変更点は 40 行での For i=1 to L を For i=1 to N とするだけです。31，32 行は L を入力するためのものです。また，70 行は L を N に替えただけです。

Next で書ける場合は For Next を使うのがお奨めです。

*30 勿論，以下の説明は厳密すぎると思うかもしれません。ただ，冒頭に挙げたダイクストラの言葉にある，「プログラマーは正しいプログラムを書くだけでなく，その正しさを分かり易く示すべきである」との立場をとれば，少なくとも厳密な説明を求められた場合，それに対応できる準備はする必要はあるでしょう。

例 2.3.1' P(L,N)

```
10 Input "文字列を入力してください",In$
20 L=Len(In$)
30 Out$=""
31 Print L; "以下の自然数を入力してください。"
32 Input N
40 For i= 1 to N
50    Out$ = Mid$(In$,i,1) + Out$
60 Next i
70 Print N;"文字逆順文字列："; Out$
80 End
```

任意に正整数 L を取り，固定します。1 ≦ N ≦ L となる**すべて**の正整数 N に対して，例 2.3.1'P(L,N) のプログラムが，Len(In$)=L となる文字列 In$ の左から N 文字を Out$ に逆順に出力することを N についての帰納法で証明します[*31]。

証明

(1) N=1 のとき。即ち，例 2.3.1'P(L,1) が正しく動作することを証明する。この場合，For の条件式は i=1 to 1 となる。この場合，Len(In$) の左から 1 文字を Out$ に追加したことになるが，Out$ の初期値は空文字列なので，今の場合 Out$ は l 文字からなる。従って，例 2.3.1'P(L,1) は正しく動作する[*32]。

(2) 1 ≦ N < L となる N に対して，例 2.3.1'P(L,N) が正しく動作すると仮定する。このとき，例 2.3.1'P(L,N+1) も正しく動作することを証明する。例 2.3.1'P(L,N+1) での For ... Next ルーチンは For i=1 to N と For i=N+1 to N+1 と分割できる。

例 2.3.1'P(L,N+1) での For ... Next ルーチンの分割

```
40 For i= 1 to N+1
50    Out$=Mid$(In$,i,1)+Out$
60 Next i
```

⟹

```
41 For i= 1 to N
42    Out$=Mid$(In$,i,1)+Out$
43 Next i
50 '---------
61 For i= N+1 to N+1
62    Out$=Mid$(In$,i,1)+Out$
63 Next i
65 '--------
```

(P1) (P1')

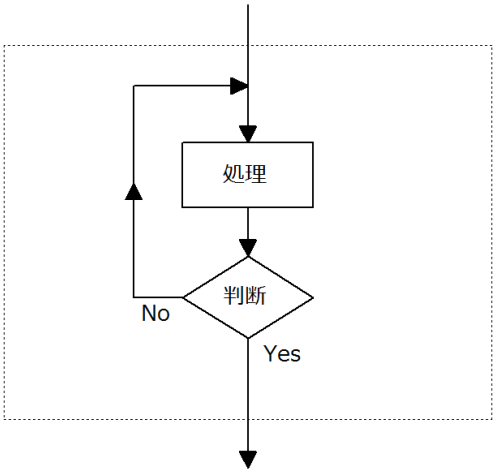
ここで，(P1')41 行〜43 行は例 2.3.1'P(L,N) のルーチンに他ならないから，帰納法の仮定より，このルーチン終了時，即ち，50 行実行時には， Out$ は入力時の In$ を左から N 文字逆順に追加した文字列になっている。続いて (P1')61 行〜63 行は i=N+1 のときだけ実行され，62 行は Out$=Mid$(In$,N+1,1)+Out$ となるから，Out$ は左側に In$ の左から N+1 番目の 1 文字を付け加えたものになる。従って， Out$ は In$ の左から N+1 文字を逆順にして付け加えたものになる。(P1) はこれと同じ動作をするから，従って，例 2.3.1'P(L,N+1) も正しく動作することが示された。

□

[*31] この数学的帰納法は N=1 から N=L までの帰納法ですから，普通の数学的帰納法と異なると感じるかもしれません。しかし，N>L の場合はそもそもの前提が偽で，常に成立ということになり，すべての N について示されたことになります。

[*32] 1 文字は逆順にしても同じものです。

2.3.2 繰り返し構造２



繰り返し構造２

繰り返し構造２では，まず処理を行い，その後＜**判断**＞を行います。それが No ならば，最初に戻り**処理**を行います。この処理を Yes の判断が出るまで順次続けます。この構造は

～（判断）になる迄，～（処理）をする

と表現されます。tbasic の構文で表すと，Do Loop Until 文

```
Do
   ～ （処理）
Loop Until （判断）
```

がこれに当たります[*33]。

例を一つ挙げます。じゃんけんの入力部分のプログラムです。次のプログラムは，例 2.2.4 じゃんけんプログラム２（入力処理付）での入力の状況で，YH$ に g，t，p （大文字可）のみを入力可とするものです。

例 2.3.2

```
10 Do
20    Input "g:グー，t:チョキ，p:パー を入力して下さい。",YH$
30    YH$=LCase$(YH$)
40 Loop Until ((YH$="g") or (YH$="t") or (YH$="p"))
50 Print YH$
```

10 行～40 行が Do Loop Until 文です。10 行は， Do Loop Until 文の始まりを表す標識です。20 行，30 行が**処理**，40 行の Loop Until 以下が＜**判断**＞部分です。

このプログラムの正当性は， Loop Until の条件式が３つの場合分けなので，証明は簡単です。

証明 20 行で， YH$ が入力され，30 行で YH$ が小文字に変換されるから，条件式

((YH$="g") or (YH$="t") or (YH$="p"))

を満たすのは， YH$が，g，t，p（大文字可）のみの場合である。この場合，処理が 50 行に進む。それ以外の入力の場合，処理が 20 行に戻り，再度入力画面になる。 □

[*33] これ以外でも， Do Loop While 文があります。これについてはヘルプを参照して下さい。

この例のように繰り返し構文であっても，条件式によっては，数学的帰納法を使わずに証明できる場合もあります。

例 2.3.2 のアイデアを使って，例 2.2.4 じゃんけんプログラム２（入力処理付）を何度もじゃんけんをするプログラムに変更しましょう。何度もじゃんけんをする部分は同様に Do Loop Until 文を使います。また，出し手以外に終了判定をする必要があるので，入力（g:グー, t:チョキ, p:パー, e:終り）として，g, t, p, e を入力文字列とするものとします。この場合，例 2.3.2 の 40 行を変更する必要があります。条件式として，

```
((YH$="g") or (YH$="t") or (YH$="p") or(YH$="e"))
```

とすることもできますが，ここでは汎用性を考えて，まず，入力文字列が一文字であるか判定して，次に，g，t，p，e であるか判定します。これは次の同値な条件式

```
((Len(YH$)=1) and (InStr(1,"gtpe",YH$)>0))
```

で表すことができます。こちらの条件式は判定する文字列が多くなった場合にも使えます。

これを利用したプログラムが次です。

例 2.3.3 じゃんけんプログラム 3（入力処理付）

```
010 Dim HShape$(4)
020 HShape$(1)="グー": HShape$(2)="チョキ": HShape$(3)="パー": HShape$(4)="終り"
030 Randomize
040 Do
050    CHS=Int(RND*3)+1
060    Do
070       Input "g:グー, t:チョキ, p:パー, e:終り を入力して下さい。",YH$
080       YH$=LCase$(YH$)
090    Loop Until ((Len(YH$)=1) and (InStr(1,"gtpe",YH$)>0))
100    YHS = InStr(1,"gtp",YH$)
110    EFlag = (1-YHS)*(2-YHS)*(3-YHS)
120    Print "コンピュータ：";HShape$(CHS);", あなた：";HShape$(YHS)
130    Select case ((CHS-YHS+3) mod 3) +EFlag
140        Case 0
150           Print "アイコ。"
160           Draw = Draw + 1
170        Case 1
180           Print "あなたの勝ち。"
190           YWin = YWin + 1
200        Case 2
210           Print "コンピュータの勝ち。"
220           CWin = CWin + 1
230    End Select
240    Print "あなたの結果 ：";YWin;"勝, ";CWin;"敗, ";Draw;"引き分け"
250 Loop Until YH$="e"
260 End
```

40 行〜250 行が Do Loop Until 文で入力文字列が YH$=e となるまで繰り返します。60 行〜90 行が入力処理の部分です。100 行の YHS の指定が YHS = InStr(1,"gtp",YH$) となり例 2.2.4 じゃんけんプログラム２（入力処理付）より，簡単になっています。今の場合，YH$ は g，t，p，e のいずれかですから，YH$="e" のとき， YHS=0 になり，例 2.2.4 の場合と同じ値になります。110 行〜230 行は例 2.2.4 の 80 行〜190 行と同様ですが，勝敗の集計のため， YWin，CWin，Draw の設定が追加されています。240 行で勝敗の集計結果が表示されます。

2.4 Goto 文の使用について

2.4.1 Goto 文有害論

これまで，構造化プログラミングで推奨する基本的制御構造（順構造，分岐構造，繰り返し構造）について説明をしてきました。またそれらを tbasic で記述するための構文として， If 文， Select case 文，For to Next 文， While Wend 文， Do 文等を挙げてきました。実は上述の基本構造を記述できる構文として Goto 文があります。Goto 文は，プログラムの実行をラベルで指定する場所にジャンプする自由度の高い構文です。

tbasic の構文で表すと，Goto 文は

Goto 文

```
Goto ラベル　または，If 条件式 then goto ラベル
```

の形で使います。この Goto 文と If 文を組み合わせれば，すべての基本構造を記述することができます。

しかし，基本構造を記述すための構文として Goto 文については，今迄敢えて説明しませんでした。それは，Goto 文は，使い方は簡単ですが，適切に使うのは難しく，使用は勧められないからです。Goto 文の使用については多くの人々が多くの議論をしていますが，中でもダイクストラによる議論が有名です。

ダイクストラは Goto 文の使用は有害であると主張する小論を 1968 年に発表しました[*34]。その中で，次のように述べています。「Goto 文が依拠しているところはあまりにも原始的です。そしてそれはプログラムを台無しにする誘惑を余りにも多く持っています。ですから，Goto 文は利用は慎むべきでしょう。」

他方，ダイクストラが構造化プログラミングを提唱した論文[*35]等では，Goto 文については明確には述べていません。ですから，ダイクストラの構造化プログラミングで，Goto 文禁止という規則は無いとも考えられます。しかし，ダイクストラの従来からの主張からすれば，Goto 文の使用を控えると解釈もできます。

ダイクストラの構造化プログラミングの提唱以来，多くの人々によって，構造化プログラミングでの Goto 文使用の可否について，議論が行われてきました。その議論の中で，

- Goto 文を使わなくても基本的制御構造文でプログラムは書ける。
- Goto 文はプログラムを分かりにくくすることもあるが，適切に使えば分かりやすくすることもある。
- 場合によっては Goto 文を使った方が効率が良いこともある。

などが認識されてきました。これらのことから「分かりやすい，効率の良い」Goto 文（「良い」Goto 文）の使用は可とする考えもあります。他方，この「良い」Goto 文を別の構文で置き換えるという考えも生まれました。このことから，以後の高級言語では，上に挙げた基本的制御構造文以外の構文が導入されました。特に，いくつかの脱出構文は「良い」Goto 文の代替として使うことができます[*36]。

脱出構文を使用することで，大部分の Goto 文を避けることができます。しかし，それでも Goto 文を使いたい状況があるかもしれません。その場合は，Goto 文の使用により，プログラムがむしろ分かりやすくなり，効率も落とさないことを確認すべきでしょう。

[*34] "Go To Statement Considered Harmful" Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

[*35] Notes on Structured Programming,1969

[*36] tbasic で使用できる脱出構文は以下の通りです。
Exit For：For 文の脱出，Exit Do：Do 文，While 文の脱出
Exit Sub，Exit Function：Sub，Function の終了。

2.4.2 Goto 文の書き直しの例

次の問題を考えます。

> 配列 A(1)，A(2)，A(3)，… ，A(10000) に相異なる整数が格納されていて，更に初期値が 0 の配列 B(1)〜B(10000) がある。
>
> 整数 x に対して，A(i)=x となる i があればその i に対し，B(i)=B(i)+1 とする。A(i)=x となる i が無ければ何もしない。

問題は単純です。配列の宣言・設定は既にできているとして，上の部分だけをプログラムにしてみます。Goto 文を使えば次のように書けるでしょう。

例 2.4.1

```
10 For i=1 to 10000
20   If A(i)=x then
30      B(i)=B(i)+1
40      Goto *Found
50   End if
60 Next i
70 *Found
```

40 行，70 行を削除しても，結果は同じになりますが，計算の効率は随分違いますので，この場合 For 文脱出のための Goto 文は有効です。しかし，この問題は Goto 文を使わずに次のようにも書けます[*37]。

例 2.4.2

```
10 i=1
20 While ((i=<10000) and (not (A(i)=x)))
30   i = i + 1
40 Wend
50 If i <= 10000 then B(i)=B(i)+1
```

こちらの方が行数も少なく，Goto 文を使っていません。内部的な処理としても，上の例 2.4.1 とほぼ同じで，効率も落ちません。ですから，こちらの方が良いプログラムと言えるかもしれません。ただ，この例 2.4.2 より上の例 2.4.1 の方が多少分かりやすい感じもします。

このようなとき，脱出構文を使うと，元の意図のままで Goto 文を使わないで，書くことができます。実際，Exit For を使って，Goto 文とほぼ同様なことが，以下のように簡単にできます。

例 2.4.3

```
10 For i=1 to 10000
20   If A(i)=x then
30      B(i)=B(i)+1
40      Exit For
50   End if
60 Next i
```

こちらの方が例 2.4.1，例 2.4.2 より分かりやすいプログラムです。ですから，このような場合には，While 文を使って書き直すよりも Exit For を使うのが良いでしょう。

[*37] 但しこの場合は，配列 A は A(10001) まで宣言する必要があります。

3 tbasic でのプログラムの構造

この節では，構造化プログラミングでのプログラムの構造，特にそれらを構成する，主プログラムと副プログラムについて説明します。主プログラムと副プログラムの一般的概略については既に，1.4 構造化言語の項で説明しました。ここでは，tbasic での使用法を中心に説明します。

3.1 主プログラムと副プログラム

主プログラムは，プログラム実行後，直ちに実行される部分です。これに対して副プログラムは，主プログラムから呼び出されて実行されます。

tbasic では，短いプログラムだけでなく大きなプログラムでも，副プログラムを使わないで，主プログラムのみで書くことも可能です。しかし，プログラムが大きくなるにつれて，主プログラムだけのプログラムでは，全体の見通しが悪くなり，修正や改良が難しくなります。このため，主プログラムのある部分の纏まりを副プログラムとして，独立に記述し個別に保守・管理することにより，プログラム全体の管理が容易になります。

副プログラムは，主プログラムとの値の引き渡し部分（引数部分）が副プログラム特有なものですが，副プログラム内での記述は，いくつかの注意点を考慮すれば，主プログラムでのものとほぼ同じです。

tbasic での副プログラムは関数型と手続き型の 2 種類があります。関数型は値を返し，手続き型は値を返しません。この関数型と手続き型は，使い方は多少違いますが，計算機内部の処理としては，あまり違いはありません。

例として，10 個の数を読み込み，その最小値と最大値を求めて表示する tbasic のプログラムを考えましょう。作り方は，いくつか考えられますが，ここでは，10 個の数の格納に配列変数を用い，その配列変数に対する処理として，実現することにします。

まず，例えば，普通にプログラムを書くと，次のようなプログラムができるでしょう。これは，主プログラムのみで構成されています。

例 3.1.1（主プログラムのみから構成されるプログラム）

```
010 Dim InD(10)
020 For i= 1 to 10
030    Input InD(i)
040 Next i
050 MaxD=InD(1)
060 For i=2 to 10
070    If InD(i)>MaxD then
080       MaxD = InD(i)
090    End if
100 Next i
110 MinD=InD(1)
120 For i=2 to 10
130    If InD(i)<MinD then
140       MinD = InD(i)
150    End if
160 Next i
170 Print "最大値=";MaxD
180 Print "最小値=";MinD
190 End
```

このプログラムは，20 行〜40 行がデータ入力，50 行〜100 行が最大値の計算，110 行〜160 行が最小値の計算，170 行〜180 行が結果表示です。これらをそれぞれ副プログラムにしてみましょう[*38]。

ここではデータ入力と結果表示を手続きとして，最大値，最小値を関数として，副プログラムにすることにします。tbasic では，手続きは Sub，関数は Function を使います。

tbasic での副プログラムの実際の作り方は後で説明しますが，値の受け渡し部分（引数部分），関数の場合関数値の設定以外殆ど，上のプログラムの対応部分と同じように作ることができます。名前をそれぞれ

- 手続き：データ入力 InputD,
- 関数：最大値 MaxVA,
- 関数：最小値 MinVA,
- 手続き：結果表示 PrintMaxMin

として，作ることにします。

手続き：データ入力 InputD は，配列変数 A() に対して，A(FNum)〜A(TNum) のデータを入力する手続きとすることにします。そのために引数として，A(),FNum,TNum を使い，次のように定義します。

例 3.1.2（副プログラム Sub InputD）

```
Sub InputD(A(),FNum,TNum)
   For i=FNum To TNum
      Input A(i)
   Next i
End Sub
```

引数部分以外，殆ど例 3.1.1 でのものと同じです。

関数：最大値 MaxVA，関数：最小値 MinVA は配列変数 A() に対して，A(FNum)〜A(TNum) の最大値，最小値を返す関数です。引数は A(),FNum,TNum とします。

例 3.1.3（副プログラム Function MaxVA, MinVA）

```
Function MaxVA(A(),FNum,TNum)
   MaxD=A(FNum)
   For i=FNum+1 to TNum
      If A(i)>MaxD then
         MaxD = A(i)
      End if
   Next i
   MaxVA = MaxD
End Function

Function MinVA(A(),FNum,TNum)
   MinD=A(FNum)
   For i=FNum+1 to TNum
      If A(i)<MinD then
         MinD = A(i)
      End if
   Next i
   MinVA = MinD
End Function
```

[*38] このくらいの長さのプログラムなら，特に副プログラムに分割する必要もありませんが，副プログラムの例として挙げます。

引数については Sub と同様ですが，戻り関数値の設定は，MaxVA = MaxD，MinVA = MinD のように名前に設定します。

手続き：結果表示 PrintMaxMin は，引数として，MaxD，MinD を使います。

例 3.1.4（副プログラム Sub PrintMaxMin）

```
Sub PrintMaxMin(MaxD, MinD)
   Print "最大値=";MaxD
   Print "最小値=";MinD
End Sub
```

これらの副プログラムを利用して，上の例 3.1.1 を書き直すと，次になります。Sub を利用するには Call を使います。また，Function は，内蔵関数と同様に使えます。

例 3.1.5（副プログラムを利用したときの主プログラム）

```
10 Dim InD(10)
20 Call InputD(InD(),1,10)
30 MaxD=MaxVA(InD(),1,10)
40 MinD=MinVA(InD(),1,10)
50 Call PrintMaxMin(MaxD, MinD)
60 End
```

或いは，

例 3.1.6（副プログラムを利用したときの主プログラム）

```
10 Dim InD(10)
20 Call InputD(InD(),1,10)
30 Call PrintMaxMin(MaxVA(InD(),1,10), MinVA(InD(),1,10))
40 End
```

これらの主プログラムはいずれも，元の主プログラムだけの例 3.1.1 より，大分短くなっています。また，主プログラムが短くなった代わりに，４つの副プログラムが新たに必要となりますが，副プログラムはいずれも元の主プログラムから少しの変更で作られたものです。これらの副プログラムでは主プログラムとの間のデータ引き渡しの部分が追加され，少し長くなりますが，副プログラムを一つの纏まりとして見ると比較的短いものです。副プログラムの機能は比較的限定されていて，プログラムも短いので，長いプログラムに比べて，見やすく内容も検証し易いものになります。また副プログラムは主プログラムや他の副プログラムと独立した纏まりとなっているので，同じ機能が他のプログラムで必要になった場合，コピーしてそのまま利用することもできます。

纏めると，主プログラムのみのプログラムを副プログラムに分けることにより，次の利点があります。

副プログラムへの分割の利点

- 主プログラム，各副プログラムは短く，機能が限定されているので，それに集中することで，
  (1) 内容が理解しやすくなる。
  (2) 内容の正当性の検証が容易になる。
- 副プログラムはそれだけで独立しているので，他のプログラムでの利用が可能となる。

1.1 構造化プログラミングの目的の最初の部分で，次の問題への対応が構造化プログラミングの目標であると説明しました。

- それらを如何に正しく動作するプログラムとして仕上げるか。
- 大規模なプログラムを如何に見通しよく管理するか。
- また後々，そのプログラムのある部分を修正する必要が生じたとき，別の人々によって，如何に間違いなく可能か。
- 更に，プログラムのある部分を後で如何に再利用可能か。

更に，そのことを具体化した表現として，

- 処理を細分する際，処理のまとまりは抽象化し，他の処理と独立に動作するようにする。

とも説明しました。ここで処理のまとまりとは，具体的には今見てきた副プログラムのことです。またここでの抽象化とは，副プログラムがどのような処理を行うかを正確に定めること，プログラミングの用語でいえば，仕様の確定のことと解釈できます。この言葉を使って上のことを表すと，

- 副プログラムの仕様を確定し，他の処理と独立に動作するようにする。

となります。

この例で見たように，

> プログラムを主プログラムと副プログラムに分けて，
> その各々を比較的小さい部分に分割して，プログラムを構成すれば，
> 上の目標を果たすことができる

ことが分かります。

以下の節では，副プログラムの作り方について更に詳しく説明します。

3.2 主プログラム，副プログラムの配置

ここでは，主プログラムと副プログラムの配置の仕方について説明します。

構造化プログラミングでは主プログラムと副プログラムの配置については，特に規定はありません。また実際，各プログラミング言語によって配置の仕様に違いがありますから，一般的に規定することはできません。ですから，使うプログラミング言語の作法に従って，配置を行います。

ここでは，tbasic における配置方法について説明します。tbasic では，主プログラムや副プログラムをどのような順番で並べるかは特に規則はありません。しかし，分かりやすさということから，次の方針で配置することを推奨します。

- 主プログラムを先頭に記述し，それ以下に下記の方針で副プログラムを配置する。
- 副プログラムの個数が少ない場合は，主プログラムで最初に使用された順序に従って配置する。
- 多くの副プログラムが使用されている場合，機能によって分類し副プログラムを纏めて配置する。
- コメント等によって副プログラムの説明，グループ分け等を適宜行う。

この方針で上のプログラムを配置すると次のようになります。

例 3.2.1（副プログラムを利用したときのプログラム）

```
' ---------主プログラム---------
Dim InD(10)
Call InputD(InD(),1,10)
MaxD=MaxVA(InD(),1,10)
MinD=MinVA(InD(),1,10)
Call PrintMaxMin(MaxD, MinD)
End

'---------副プログラム---------
Sub InputD(A(),FNum,TNum)
   For i=FNum To TNum
      Input A(i)
   Next i
End Sub

Function MaxVA(A(),FNum,TNum)
   MaxD=A(FNum)
   For i=FNum+1 to TNum
      If A(i)>MaxD then
         MaxD = A(i)
      End if
   Next i
   MaxVA = MaxD
End Function

Function MinVA(A(),FNum,TNum)
   MinD=A(FNum)
   For i=FNum+1 to TNum
      If A(i)<MinD then
         MinD = A(i)
      End if
   Next i
   MinVA = MinD
End Function

Sub PrintMaxMin(MaxD, MinD)
   Print "最大値=";MaxD
   Print "最小値=";MinD
End Sub
'--------------------------------
```

3.3 副プログラムの作り方，使い方

ここでは，tbasic における副プログラムの作り方について説明します。

まず，tbasic では副プログラムについては

副プログラムの中に副プログラムを書くことはできない。

しかし，

主プログラムや任意の副プログラムから

任意の副プログラムを呼んで利用することができる。

という規則があります。この規則は，tbasic に限らず比較的良くある規則です[*39]。

tbasic での副プログラムは Sub と Function があります。これらの内部での処理方法はよく似ていますが，値の返し方が異なるため，記述方法，使い方に多少の違いがあります。

3.3.1 Sub の作り方，使い方

まず， Sub [*40]について説明します。Sub は次の形をしています。

Sub の仕様

```
Sub サブプログラム名 (引数リスト)
    処理ブロック
End Function
```

ここで，サブプログラム名はその副プログラムを示す識別子（名前）で，数値変数名と同様に名前を付けることができます[*41]。引数リストは，呼んだプログラムから引き渡される変数，値を受け取る変数のリストです。使える変数は，数値変数，文字列変数，ブール変数とそれらの配列変数です。カンマで区切って並べます。処理ブロックは，Sub で行う処理を tbasic の文で普通に記述します。

上に挙げた例 3.1.4

```
Sub PrintMaxMin(MaxD, MinD)
   Print "最大値=";MaxD
   Print "最小値=";MinD
End Sub
```

で見ると，PrintMaxMin が Sub の名前です。それに続く MaxD，MinD が引数のリストです。２つの数値変数 MaxD，MinD によって構成されています。

Sub を主プログラム，または副プログラムから呼ぶ（利用する）には Call 文を使って次の形で使います。

Sub の呼び方

```
Call サブプログラム名 (引き渡しリスト)
```

ここで，引き渡しリストは，Sub の定義での引数リストにある各変数に引き渡す式，値，変数のリストです。このリストの値が対応する変数に代入され，Sub の処理ブロックが実行されます。

例えば，上の例 3.1.5 にある

```
Call PrintMaxMin(MaxD, MinD)
```

では，MaxD，MinD が引き渡しリストです。この場合は変数のリストになっています[*42]。または，上の例 3.1.6 にある

```
Call PrintMaxMin(MaxVA(InD(),1,10), MinVA(InD(),1,10))
```

[*39] 例えば，Visual Basic でも類似な規則があります。
[*40] Sub は subroutine の意味から名付けられたもので，手続き，procedure とも言われます。
[*41] 予約語や別の変数名と異なる，英字から始まる英数字 30 文字以下で構成された文字列です。
[*42] この場合，引き渡しリストの変数名が引数リストの変数名と同じになっていますが，違っていていも構いません。

では，MaxVA(InD(),1,10)，MinVA(InD(),1,10) が引き渡しリストです。この場合は，式のリストになっています。式の場合は，式を計算した結果の値が，引数リストに引き渡されます。このように引き渡しリストは，変数でも式でも結果の型が，Sub の定義での引数リストに対応していれば構いません。定数式，即ち，定数のリストでも構いません。例えば，

```
Call PrintMaxMin(5, 3)
```

のように呼ぶこともできます。この場合は，Sub PrintMaxMin では，MaxD=5，MinD=3 と代入されて，処理ブロックに実行が移ります。

3.3.2 Function の作り方，使い方

ここでは， Function について説明します。Function は次の形をしています。

関数の仕様

```
Function 関数名 (引数リスト)
    処理ブロック

    関数名=戻り値
    ・・・
End Function
```

ここで，関数名はその副プログラムを示す識別子（名前）で，返す型と同じ型の変数名と同様に名前を付けます[*43] 。引数リストは，呼んだプログラムから引き渡される変数，値を受け取る変数のリストです。使える変数は，数値変数，文字列変数，ブール変数とそれらの配列変数です。カンマで区切って並べます。処理ブロックは，Function で行う処理を tbasic の文で普通に記述します。戻り値は処理ブロックの適切な場所に，

関数名 = 戻り値

として設定します[*44]。

上に挙げた例 3.1.3

```
Function MaxVA(A(),FNum,TNum)
   MaxD=A(FNum)
   For i=FNum+1 to TNum
      If A(i)>MaxD then
         MaxD = A(i)
      End if
   Next i
   MaxVA = MaxD
End Function
```

で見ると，MaxVA が Function の名前です。それに続く A(),FNum,TNum が引数のリストです。配列変数 A() と２つの数値変数 FNum,TNum によって構成されています。戻り値は，

[*43] 文字列を返す場合は，文字列を示す接尾辞$ をつけるか，as string を使います。また，論理値を返す場合は as boolean を使います。

[*44] 場合分けで戻り値を決める場合など，複数回設定しても構いません。

```
  MaxVA = MaxD
```

と設定しています。

もう一つ例を挙げましょう。次の例は与えられた文字列に対してその文字列を逆順に並べた文字列を返す関数です。これは既に挙げた例 2.3.1 を関数に書き換えただけです。

例 3.3.1 逆順文字列

```
' 逆順文字列
Input "文字列を入力してください", In$
Print "逆順文字列は"; InvStr$(In$)
End

Function InvStr$(In$)
   L = Len(In$)
   Result$=""
   For i=1 to L
      Result$=Mid$(In$,i,1)+Result$
   Next i
   InvStr$=Result$
End Function
```

Function を主プログラム，または副プログラムから呼ぶ（利用する）には内蔵関数の場合と同様に使います。

Function の使い方

関数名 (引き渡しリスト) を戻り値として使う。

ここで，引き渡しリストは，Function の定義での引数リストにある各変数に引き渡す式，値，変数のリストです。このリストの値が対応する変数に代入され，Function の処理ブロックが実行され，戻り値として設定された値になります。

3.4 副プログラムと局所変数

前項では副プログラムの作り方と使い方を説明しました。ここでは，主プログラム，副プログラムにおける変数の適用範囲（可視範囲，スコープ）について説明します。

副プログラムのもつべき性質として，「構造化言語」の項の中で副プログラムでの変数の局所性について

> 副プログラムで使われた変数は，
> 別の主プログラムや別の副プログラムで
> 使われた同じ名前の変数があっても実行上は別な変数として扱われる。

と説明しました[*45]。ここではこれについて具体的な例で説明します。

[*45] これには注意べき例外として引数を経由した「参照引き渡し」があります。これについては以下の参照引き渡しの項で説明します。

次のプログラムは，1000 個のデータに対して，それらのうち最初から各 100 個づつの平均値を画面に表示するものです。130 行〜150 行で 1000 個のデータを乱数を使って作成し，160 行〜180 行でそれらのデータの 100 個づつの平均値を，平均値を求めるユーザー定義関数 MeanI を使って，順次計算しています。MeanI は引数 A(),FNum, TNum で配列 A() に対して A(FNum) から A(TNum) までの平均値を返す関数です。

例 3.4.1

```
110 Dim InD(1000)
120 RandoMize
130 For i=1 to 1000
140    InD(i)=RND*1000
150 Next i
160 For i=1 to 10
170   Print MeanI(InD(),(i-1)*100+1,i*100)
180 Next i
190 End

210 Function MeanI(A(),FNum, TNum)
220    Sum=0
230    For i=FNum to TNum
240       Sum = Sum + A(i)
250    Next i
260    MeanI = Sum/(TNum-FNum+1)
270 End function
```

160 行〜180 行の For 文の中で，関数 MeanI を使用しています。主プログラムで使われている変数 i と関数 MeanI の中で使われている変数 i とは変数名は同じですが，互いに独立で影響はありません[*46]。このことは，例えば，

```
175   Print "主プログラムでの i=";i

265  Print "MeanI 内での  i=";i
```

をそれぞれ対応する部分に追加したプログラムを実行することで確かめることができます。このように，副プログラムを呼ぶ側では，その副プログラムの中でどのような変数が使われているかを考える必要はありません。

3.5 副プログラムとのデータの引き渡し

前項では副プログラムでの変数の局所性の説明をしました。それによれば主プログラムと副プログラムでは仮に同じ名前の変数があっても，別なものとして動作します。しかし，副プログラムに目的の計算処理を行わせるためには，それを呼ぶプログラムと副プログラムの間で何らかのデータの受け渡しを行う必要があります。これを行うのが引数です。実際，副プログラムとその外部とのデータの受け渡しはこの引数を介してのみ行われます[*47]。

引き渡しの方法は 2 種類あります。

[*46] 実際，関数 MeanI の終了時，関数 MeanI の変数 i の値は，TNum+1 になります。例えば，主プログラムで変数 i の値が 1 のとき，170 行での関数 MeanI の引数は，A(),1,100 となり，この時の関数 MeanI の終了時，関数 MeanI の変数 i の値は 101 になります。

[*47] この例外として，グローバル変数あるいは Public 変数があります。

3.5.1 値引き渡し

値引き渡しは文字通り，引数を通して値を副プログラムへ引き渡す方法です。tbasic では，関数 Function の引き渡しは原則としてこの方法です[*48]。

例を挙げましょう。次の例の関数 Max3n は引数部分は a,b,c で a,b,c の最大数を返す関数です。

例 3.5.1

```
Input "3 数を入力してください。",a1,a2,a3
Print "最大数は";Max3n(a1,a2,a3)
End

Function Max3n(a,b,c) :' a,b,c の内の最大数を返す。
   Max3n = max(max(a,b),c)
End Function
```

この関数 Max3n の使い方は，内蔵関数と同様です。例にある，Max3n(a1,a2,a3) のように変数を使って呼ぶこともできますが， Max3n(a+x,b*y,3^2) のように式や，数値を使って呼ぶこともできます。いずれの場合も，呼んだ時点での値を対応する引数変数に渡して，副プログラムの計算処理に移ります[*49]。ですから，例えば，

```
x=2:y=3:z=1
Print Max3n(x^2,y,z)
```

は

```
Print Max3n(4,3,1)
```

と同じです。

3.5.2 参照引き渡し

参照引き渡し[*50]は値ではなく，変数を副プログラムへ引き渡す方法です。tbasic では， 呼ぶ側での Sub の引数に対応する項が変数であった場合，この引き渡しが行われます[*51]。参照引き渡しは，呼ぶときに使われた変数の情報を副プログラムに渡します。その結果，副プログラムの中で対応して受け取った変数は渡された変数のように振舞います。特に，副プログラム終了時，受け取った変数の値は呼ぶときに使われた変数の値と同じになります。

[*48] 例外は，後で説明する配列変数を引き渡す場合です。

[*49] 例えば，Call Max3n(a1,a2,a3) であれば，a1，a2，a3 の値を，それぞれ引数第１，２，３変数 a，b，c に設定して，副プログラム Max3n の処理に移ります。

[*50] **変数引き渡し**とも言います。

[*51] 呼ぶ側での Sub の引数に対応する項が変数ではなく，式や値の場合は，引数へは値引き渡しになります。

例を挙げましょう。次は 3 つの数の整列をする副プログラム Sort3N の例です。これは 3 つの数の入力に対し，それを小さい順に並べ替えた 3 つの数を返す副プログラムです。関数は一つの値しか返せませんので，このような処理を関数では実現できません[*52]。このように複数の値を返すことは，Sub の参照引き渡しを利用すると可能になります。

例 3.5.2

```
10 Input "3 つの数を入力";x,y,z
20 Call Sort3N(x,y,z)
30 Print x,y,z
40 End

110 Sub Sort3N(a,b,c)
120    If a > b then Call SwapN(a,b)
130    If b > c then Call SwapN(b,c)
140    If a > b then Call SwapN(a,b)
150 End Sub

210 Sub SwapN(a,b)
220    tmp = a
230    a=b
240    b=tmp
250 End Sub
```

20 行で，Call Sort3N(x,y,z) の形で，Sub Sort3N を呼んでいます。引数対応部分の呼ぶ形が (x,y,z) と変数になっていますから，これは参照引き渡しです。これにより Sub Sort3N でこれらの x,y,z はそれぞれ，a,b,c に変数として引き渡され，Sub Sort3N 内で，a,b,c に加えられた変更は，そのまま x,y,z に反映します。ですから，a,b,c への処理は x,y,z への処理と考えられます。

Sub Sort3N 内で更に Sub SwapN を呼んでいますが，これも参照引き渡しです。Sub SwapN の中では，220 行〜240 行で，変数 a,b の中身の交換を行っていますが，参照引き渡しなので，引き渡された変数の中身の交換が行われます。ですから，例えば，130 行での Call SwapN(b,c) が実行された場合，b と c の交換が行われます[*53]。

上の例は Sub の参照引き渡しが本質的な例でしたが，Sub の中で引数変数の値を変更しなければ，参照引き渡しであっても，引き渡された変数の値は変わりませんから，実際は値引き渡しと同じになります。

3.5.3 配列変数の引き渡し

多数のデータがあって，それらの平均値や標準偏差など定型的な計算をしたいときもあるでしょう。定型的な計算では，関数や Sub を使うのが構造化プログラミングの基本です。多数のデータは普通，配列に格納されていますから，このような場合，副プログラムに大きな配列を引き渡す必要があります。

このような例は今迄既にありましたが，特に説明はしませんでした。ここでは，配列変数を関数や Sub に引き渡す方法について説明します。

[*52] これは tbasic での話です。処理系によっては，色々なデータ型を返すことができるものもあり，そのような処理系では 3 つの数の組を定義し，その組をを返す関数を書けば，このようなことも関数で実現できます。

[*53] Sub Sort3N，Sub SwapN の中で変数 a,b 等が使われていますが，それぞれ局所変数ですから，別な変数です。しかし引数を経由した参照引き渡しにより 130 行にある Sub Sort3N での局所変数 b,c は，Sub SwapN での局所変数 a,b に引き渡され，それぞれ 220 行〜24 行での処理結果は Sub Sort3N での局所変数 b,c に反映されます。

まず関数の例を挙げます。次の例は，1000 個の配列に格納されている数の平均値を計算する関数（Function）のプログラムです。この例は既に挙げた例 3.4.1 の簡略形で，副プログラムは 210 行〜270 行の Function Mean で引数は A(),DNum です。A(1)〜A(DNum) の平均値を返します。

配列変数を副プログラムに渡す方法はすべて参照引き渡しです。渡し方は，下の例では，副プログラムでは，引数部分に 210 行でのように記述し，引き渡す側では，渡す項に 160 行で InD() と記述しています。参照引き渡しは，変数の持っている値[*54]を引き渡すのではなく，変数の情報を引き渡します。配列変数の情報は主にデータの型，データの個数，格納メモリの位置などです。これらの情報は，配列変数に格納されているデータの個数に関わらず一定で，配列変数に格納されているデータのメモリ量に比べて遥かに少量です。このことから，参照引き渡しによる引き渡しは，値引き渡しで全データをコピーして渡すより遥かに短い時間で可能になります[*55]。

例 3.5.3

```
110 Dim InD(1000)
120 RandoMize
130 For i=1 to 1000
140    InD(i)=RND*1000
150 Next i
160 Print Mean(InD(),1000)
170 End

210 Function Mean(A(), DNum)
220    Sum=0
230    For i=1 to DNum
240       Sum = Sum + A(i)
250    Next i
260    Mean = Sum/DNum
270 End Function
```

110 行で配列を宣言し，120 行〜150 行ではランダムなデータを作成しています。160 行で Function Mean を利用しています。

```
160 Print Mean(InD(),1000)
```

ユーザー定義関数は，このように内蔵関数と同様に利用できます。この利用で，処理が

```
210 Function Meant(A(),DNum) :
```

に移りますが，第 1 引数 A() は参照引き渡しで，Function Mean の中では，A() に対する処理は，実は主プログラムから渡された，InD() に対する処理になります。

Function Mean の中では A() は配列宣言をしていませんが，InD() は 110 行で配列宣言されており，この宣言が Function Mean の中で利用されます。第 2 引数 DNum は対応項が数値 1000 ですから，DNum=1000 として，Function Mean が始まります。

この例 Function Mean では，A() の値は変更されていませんから，ここでの参照引き渡しの役割は高速に配列変数全体を関数に引き渡すことにあります[*56]。

[*54] 配列変数場合は多量のデータの集まり

[*55] 例えば 10 万のデータを格納した配列変数 InD(1)〜InD(100000) を値引き渡しで副プログラムに渡すとするとその処理時間は数秒掛かるかもしれません。しかし他方，InD の配列変数としての情報はほぼ瞬時に渡すことができます。

[*56] 勿論配列変数引き渡しの場合，関数の中でも引き渡された配列変数の値を変更すると，引き渡された配列変数の対応する値が変更

次に Sub の例を挙げます。次の例は，1000 個の配列に格納されている数を整列するプログラムです。整列用の副プログラムは 1010 行〜1090 行の Sub SelSort で引数は D(),ND です。D(1)〜D(ND) の数を整列します。ここでの整列法は選択整列法です[*57]。Sub SelSort の中で，更に Sub Change(C(),i,j) を呼んでいます。このプログラムでは冒頭で宣言された大きな配列 A() を副プログラムへ渡していますが，参照引き渡しなので，高速に渡すことができます。

例 3.5.4

```
110 Dim A(1000)
120 Randomize
130 For i=1 to 1000
140    A(i)=RND*100
150 Next i
160 For i=1 to 1000
170    Print A(i)
180 Next i
190 Call SelSort(A(),1000)
200 For i=1 to 1000
210    Print A(i)
220 Next i
230 End

1010 Sub SelSort(D(),ND) : 'D(1)〜D(ND) の数を整列する。選択整列法
1020    For i=1 to ND-1
1030       mini = i
1040       For j=i+1 to ND
1050          If D(mini)>D(j) Then mini = j :' 比較，代入
1060       Next j
1070       If mini > i Then Call Change(D(),i,mini) :' 交換
1080    Next i
1090 End Sub

2010 Sub Change(C(),i,j)
2020    T=C(i)
2030    C(i)=C(j)
2040    C(j)=T
2050 End Sub
```

110 行で配列を宣言し，120 行〜150 行はランダムなデータを作成しています。160 行〜180 行はそのデータの確認のための表示です。

190 行で Sub SelSort を呼んでいます。

```
190 Call SelSort(A(),1000)
```

この呼び出しで，処理が

```
1010 Sub SelSort(D(),ND) :
```

に移りますが，第一引数 D() は参照引き渡しで，Sub SelSort の中では，D() に対する処理は実は主プログラムから渡された，A() に対する処理になります。Sub SelSort の中で D() は配列宣言をしていませんが，今の場合，引き渡された配列は A() なので，D(1000) まで，使えることになります。第２引数 ND は対応項が

されます。

*57 アルゴリズムについては，プログラミングの背景を参照してください。

数値 1000 ですから，値引き渡しになり，ND=1000 として，Sub SelSort が始まります。

また，Sub SelSort の中で，Sub Change(A(),i,j) を呼んでいますが[*58]，この引き渡しも参照引き渡しで，Sub Change(C(),i,j) での C() への操作は，実際は主プログラムから渡されてきた A() への操作になります。ですから，Sub Change(C(),i,j) で行われる，C(i) と C(j) との値の入れ替えは，A(i) と A(j) の入れ替えになり，最終的に，Sub SelSort が終わった時には，配列 A() の整列が終わります。その結果の確認のための表示が 200 行〜220 行です。

3.6 副プログラムの再帰的呼び出し

3.3 では，主プログラムや副プログラムでは任意の副プログラムを呼んで利用できることを説明しました。また前項では，副プログラムから別の副プログラムを呼ぶ例を挙げました。ここでは，自分自身を呼び出す副プログラムについて説明します。

自分自身を呼び出す副プログラムを再帰的プログラムと言います[*59]。再帰的プログラムについては，「プログラミングの背景」で説明しましたが[*60]，ここでは，構造化プログラミングの立場からの説明をします。

再帰的プログラムは，自分自身を呼び出しますが，呼び出す状況は，より簡単な状況として呼び出します。呼ばれた副プログラムは更に状況を簡単にして自分自身を呼び出します。これにより呼び出す毎に問題が簡単な状況になります。そしていずれは，問題が全く明らかな状況になります。そうなったところから順次元に戻り，最終的に求める結果を得るという方法です。纏めると，再帰プログラムは，

(1) 問題が全く明らかな状況
(2) 簡単な状況への帰着

の部分からなります。ここで，再帰プログラムでは，「(2) 簡単な状況への帰着」により，確実に（有限的にだけではなく**効率的に**），「(1) 問題が全く明らかな状況」になることが必要です。プログラムによっては，(1) への帰着が非効率的である場合もあり[*61]，この効率的になされるかの検証は重要です。

■逆順文字列

例を挙げましょう。次の例は例 3.3.1 で挙げた逆順文字列関数 InvStr$ の再帰版 InvStrR$です。

例 3.6.1 逆順文字列

```
’ 逆順文字列 再帰版
Input "文字列を入力してください", In$
Print "逆順文字列は"; InvStrR$(In$)
End
```

[*58] ここでは副プログラムの中でまた副プログラムを呼ぶ例として，Sub Change(A(),i,j) を挙げました。このようにすると副プログラムがすっきりして，見やすくなります。しかし今の場合，Sub Change(A(),i,j) を定義して呼ぶよりも，その中身を Sub SelSort で直接記述したほうが多少効率的です。副プログラムを呼ぶことは大きな負荷ではありませんが，直接記述するよりも多少のが掛かります。今の場合，ND が大きくて，Sub Change(A(),i,j) を呼ぶ回数が多いデータの場合，Sub Change(A(),i,j) を使った方が遅くなります。

[*59] より一般的に，いくつかの副プログラムを経由して，自分自身を呼び出すプログラムも再帰的プログラムと言います。自分自身を直接呼び出すものを，直接再帰プログラム，いくつかの副プログラムを経由するものを間接再帰プログラムと言います。

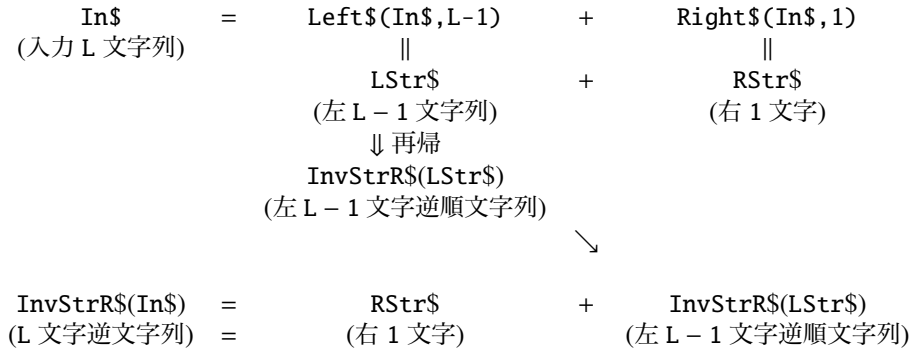
[*60] 必要なら「プログラミングの背景」再帰的アルゴリズムも参照してください。

[*61] フィボナッチ数の計算を再帰的プログラムで書いた時の非効率性は有名です。

```
010 Function InvStrR$(In$)
020    L = Len(In$)
030    If L <=1 Then
040       InvStrR$=In$
050    Else
060       LStr$ = Left$(In$,L-1)
070       RStr$ = Right$(In$,1)
080       InvStrR$=RStr$+InvStrR$(LStr$)
090    End If
100 End Function
```

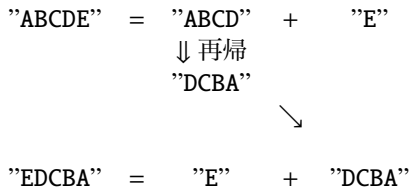
30 行，40 行が In$ が 1 文字の場合で「(1) 問題が全く明らかな状況」に当たります。1 文字は逆順としても同じですから，この場合， InvStrR$ は，1 文字 In$ をそのまま返します。

50 行〜90 行が「(2) 簡単な状況への帰着」に当たります[*62]。60 行，70 行で In$ を左側 L-1 文字 LStr$ = Left$(In$,L-1) と右側 1 文字 RStr$ = Right$(In$,1) に 2 つに分けます。 LStr$ は 1 文字減った文字列ですから，簡単な状況になりました[*63]。そこで，それを InvStrR$ で再帰的に処理し，その結果 InvStrR$(LStr$) を RStr$ と入れ替えたものが，InvStrR$になります。これを図示すると以下の通りです。



これを具体的に例示すると次になります。

例 5 文字の場合



非再帰版の例 3.3.1 では，繰り返し文 For Next 文が使われていました。そして For Next 文を含むプログラムの正当性の証明は，数学的帰納法を使いました。しかし，この再帰版では繰り返し文が使われていません。実はこの再帰プログラムは実質的に，繰り返し・数学的帰納法の過程を含んでいる言えます。そのため，プログラムの正当性は上の処理の各ステップを確認するだけで良く，より簡明になります。

[*62] ここの部分は 3 行で書かれていますが，
InvStrR$=Right$(In$,1)+InvStrR$(Left$(In$,L-1))
と 1 行で書くこともできます。

[*63] 文字数が 1 字減り，再帰呼び出しを行いますから，L-1 回の呼び出しで，「(1) 問題が全く明らかな状況」になります。必ずしも効率的とは言えませんが，非再帰版 For Next 文を使ったものとほぼ同じ効率です。

■拡張 Euclid の互除法

拡張 Euclid の互除法は，正整数 a, b に対して，

$$ax + by = \gcd(a, b)$$

となる整数 x, y を求める方法です[*64]。ユークリッドの互除法を逆にたどり，この組 (x, y) を求めることができるので，この名前があります[*65]。「拡張ユークリッドの互除法」の計算プログラムは，プログラミングの背景「拡張ユークリッドの互除法」で既に説明しました。ここでは，その再帰的プログラムについて説明します。

ユークリッドの互除法の原理は次の通りでした。

ユークリッドの互除法

正整数 a, b $(a > b > 0)$ 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。

$$\begin{array}{lllllll}
r_0 & = & q_1 & \cdot & r_1 & + & r_2 \qquad (0 < r_2 < r_1) \qquad (*1)\\
r_1 & = & q_2 & \cdot & r_2 & + & r_3 \qquad (0 < r_3 < r_2) \qquad (*2)\\
r_2 & = & q_3 & \cdot & r_3 & + & r_4 \qquad (0 < r_4 < r_3) \qquad (*3)\\
 & \cdots & & & & & \\
r_{i-2} & = & q_{i-1} & \cdot & r_{i-1} & + & r_i \qquad (0 < r_i < r_{i-1}) \qquad (*r_{i-1})\\
 & \cdots & & & & & \\
r_{n-2} & = & q_{n-1} & \cdot & r_{n-1} & + & r_n \qquad (0 < r_n < r_{n-1}) \qquad (*r_{n-1})\\
r_{n-1} & = & q_n & \cdot & r_n & & \qquad (*r_n)
\end{array}$$

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。

再帰的プログラムもこれを注意深く見ることで作ることができます。拡張 Euclid の互除法は自明な恒等式

$$r_{n-1} \cdot 0 + r_n \cdot 1 = r_n = \gcd(a, b) \qquad (*0)$$

から始めて，$(*r_{i-1})$ から得られる

$$r_i = r_{i-2} - r_{i-1}q_{i-1}$$

を $i = n,\ n-1, \ldots, 2$ と順次代入し，整理したものから得られます。実際，

$$r_{i-1} \cdot x_i + r_i \cdot y_i = \gcd(a, b)$$

とすると，この左辺に $r_i = r_{i-2} - r_{i-1}q_{i-1}$ を代入し，r_{i-2}, r_{i-1} について整理して，

$$r_{i-2} \cdot y_i + r_{i-1} \cdot (x_i - q_{i-1}y_i) = \gcd(a, b)$$

が得られます。従って，

$$x_{i-1} = y_i,\ y_{i-1} = x_i - q_{i-1}y_i \qquad (*)$$

[*64] この形の等式を Bézout(ベズー) の等式と言います。この解 x, y は一意ではありませんが，一組求めればすべての解を求めることができます。

[*65] 以下の説明の詳細はプログラミングの背景「ユークリッドの互除法」,「拡張ユークリッドの互除法」を参照してください。

と置くと，

$$r_{i-2} \cdot x_{i-1} + r_{i-1} \cdot y_{i-1} = \gcd(a, b) \qquad (**)$$

となります[*66]。

以上の性質を注意すれば，目的の再帰的プログラムは以下のように作成できます。

プログラムを，Sub を使って実現することとし，その名前を `Sub ExGCDR(x1,y1,r0,r1,c)` としましょう。これは，整数 $r_0 > r_1 > 0$ に対して，$r_0x_1 + r_1y_1 = \gcd(r_0, r_1)$ となる $x_1, y_1, c = \gcd(r_0, r_1)$ を求めるものです。

まず，$r_2 = r_0 \mod r_1$, $q_1 = r_0 ¥ r_1$ となることに注意します。

そこで，「(1) 問題が全く明らかな状況」は r_1 が r_0 を割り切る，即ち，$r_2 = 0$ で $c = r_1$ のときです。この場合は，自明な関係

$$r_0 \cdot 0 + r_1 \cdot 1 = r_1$$

が成立し，$x_1 = 0, y_1 = 1$ となります。

$r_2 \neq 0$ のときは，r_1, r_2 に対して「(2) 簡単な状況への帰着」の場合になります。ユークリッドの互除法を一つ進め，

$$r_1x_2 + r_2y_2 = \gcd(r_1, r_2) = c$$

を求めるために，再帰的に `Sub ExGCDR(x2,y2,r1,r2,c)` を呼び出します。これにより得られた，x_2, y_2, c から x_1, y_1, c は，上で得られた漸化式 $(*)$ により

$$x_1 = y_2,\ y_1 = x_2 - q_1y_2$$

で求められます。これをプログラムにすると次のようになります。

例 3.6.2 拡張 Euclid の互除法

```
Input "a>b>0", a,b
Call ExGCDR(x,y,a,b,c)
Print a;"*(";x;") + ";b;"*(";y;") = ";c
End

Sub ExGCDR(x1,y1,r0,r1,c)
   r2 = r0 mod r1: q1 = r0 ¥ r1
   If (r2=0) then
     x1=0: y1=1: c=r1:
   Else
      Call ExGCDR(x2,y2,r1,r2,c)
      x1=y2: y1=x2-q1*y2
   End If
End Sub
```

このプログラムでの，再帰呼び出し回数は，呼び出しを行う度に，ユークリッドの互除法が一つ先に進みますから，上の記号で，$n - 1$ 回です[*67]。この n は小さいことが知られていますから，この再帰プログラムの効率は，非再帰の場合と同様に良いものと言えます。

また，プログラムは，上に挙げた説明の結果をそのままコード化しただけですから，分かりやすく，その正当性も明白です。

[*66] x_i, y_i の定め方から，$x_n = 0$, $y_n = 1$ となります。従って，$x_{n-1} = 1$, $y_{n-1} = -q_{n-1}$ となり，この場合 $(**)$ は，

$$r_{n-2} \cdot x_{n-1} + r_{n-1} \cdot y_{n-1} = r_{n-2} - r_{n-1} \cdot q_{n-1} = r_n = \gcd(a, b)$$

即ち，$(*r_{n-1})$ を変形した式になります。ですから，$(*0)$ からではなく，$(*r_{n-1})$ から始めても同じ結果になります。

[*67] 最初の呼び出しを考慮すると n 回です。

3.7 構造化プログラミングでのプログラム作成例

これまで，構造化プログラミングの考え方と，構造化プログラミングでのプログラムの作成方法を説明してきました。ここでは，それらの纏めとして，上に説明した方法に基づいて，プログラムを作成する手順を例で説明します。

ここで作成するプログラムは，

> 2 から始めて小さい方から順に 1000 個の素数を表にして表示する。

というものです。

このプログラムは出力として，

```
 2    3    5    7   11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
・・・
(1000 個番目の素数まで表示する。)
```

といったものを想定しています。

プログラムを作成する前にその準備として，問題を十分に把握する必要があります。

3.7.1 準備：問題の把握

問題は素数についてですから，まずその性質を確認しましょう。ここでは，素数の無限性と素数判定の簡単な事実だけをあげます[*68]。

素数の無限性

素数は無限に存在する。

この素数の無限性はプログラムの作成そのものには影響しませんが，問題設定で，1000 個の素数が実際にあることの確認です。

基本的素数・合成数判定法

(1) 自然数 $n > 0$ に対して，n 未満のすべての自然数 (> 1) で割り切れなければ n は素数である。
(2) 自然数 $n > 0$ に対して，n 未満のすべての素数で割り切れなければ n は素数である。
(3) 自然数 $n > 0$ に対して，$\sqrt{n}$ 未満のすべての素数で割り切れなければ n は素数である。

(1) は素数の定義そのものです。また (2) と (3) は自然数の素因数分解可能性から得られる基本的事実です。
次に，これらの判定法について例で簡単に説明します。

[*68] 以下に挙げる事実はよく知られているものですが，「プログラミングの背景」の中にある「記号と準備（2017 年 06 月版）」にも証明，説明があります。

問題は，

$$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad \cdots$$

と続く自然数列から素数 $2, 3, 5, 7, \ldots$ を順次抜き出すことです。

これは，素数に対して，その次の素数を見つければよい訳です。

例えば，$2, 3, 5, 7$ と素数を抜き出した後，次の素数が 11 であることを見つけることを考えてみます。それは，$8, 9, 10$ が素数でないことと 11 が素数であることを判定すればよいわけです。$8, 9, 10$ が合成数であることは，2 または 3 で割り切れますから，すぐに判定できます。

11 が素数であることは (1) の方法ですと，11 が $2, 3, 4, 5, 6, 7, 8, 9, 10$ で割り切れないことから，11 が素数であることが結論できます。一方 (2) の方法ですと，11 が $2, 3, 5, 7$ で割り切れないことから，11 が素数であることが結論できます。更に，(3) の方法ですと，$\sqrt{11} = 3.31...$ ですから，11 が $2, 3$ で割り切れないことから，11 が素数であることが結論できます。(3) の方法は大分効率的です。

(3) の方法は，素数判定としては (1) に比較して大分効率的ですが，(3) の方法を使うためには，予め $\sqrt{n}$ 以下の素数を知っていなければなりません。つまり，$\sqrt{n}$ 以下の素数の表を計算途中で使えることが必要です。素数判定に素数表を使うのは少し違和感を感じるかもしれませんが，ここでの方法は，小さいほうから順に素数を求めるとき，既に求めた素数を覚えておいて，それを利用するというものです。ですから，むしろ合理的な方法です。そこで，ここでは素数表を内部的に作り，それを利用することにします[*69]。

このことから，素数表を使った，素数・合成数判定の方法として次が得られます。

素数・合成数判定法

$n > 1$ を自然数とする。

- $\sqrt{n}$ 以下の素数表が予め用意されているとする。
- $2, 3, 5, \cdots$ と小さいほうから順に $\sqrt{n}$ 以下の素数で割っていき，割り切れるものがあれば，合成数，そうでなければ素数である。

この素数・合成数判定法は，簡単な事実ですが，次の例で見られるように比較的強力です。

例 3.1. 113 は素数である。実際，$\sqrt{113} = 10.63\cdots$ であり，これ以下の素数は，$2, 3, 5, 7$ である。113 はこれらで割り切れないから素数である。

以上のことから，

$$2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \quad \cdots$$

より，素数表

$$2 \quad 3 \quad 5 \quad 7 \quad 11 \quad 13 \quad 17 \quad 19 \quad 23 \quad 29 \quad 31 \quad 37 \quad 41 \quad 43 \quad 47 \quad 53 \quad \cdots$$

を作成する方針として，次が得られます。

[*69] 素数表を使わずに，(1) の方法を多少効率化する方法も考えられますが，その分プログラムが複雑になります。

素数表の作成方針

(a) $2, 3, 5, \ldots$ と順次，次の素数を見つけ，素数表に追加する。
(b) 次の素数は次のようにして見つける
　・素数の候補を n とする。n の素数判定を行い，合成数なら次の素数候補について素数判定を行う。これを素数が見つかるまで繰り返す。
(c) 素数判定は，$\sqrt{n}$ 以下の素数で割ることにより行う。
(d) $\sqrt{n}$ 以下の素数は既に作成した素数表から利用する。

この方針のもとで，実際にプログラムを作成していきましょう[*70]。

3.7.2 主プログラムの作成

これから，擬似プログラム[*71]を段階的に精密化・構成し，目的とするプログラムを作成します。まず，全体の流れを大きくまとめた，主プログラムを作成します。

最初の擬似プログラムは次です。殆ど与えらえた問題を書いただけです。

Step1

```
10 '    2 から始めて小さい方から順に 1000 個の素数を表にして表示するプログラム
20   2 から始めて小さい方から順に 1000 個の素数を表にして表示する。
30 End
```

ここで表にして表示するの意味は，単に表形式で表示するという意味にも取れますが，上で説明したように，効率の面からここでは一度内部的に表を作成し，それを表示するということにします。このようにすると，20 行は次のように３つの処理に分割（精密化）することが出来ます。

Step2

```
10 '    2 から始めて小さい方から順に 1000 個の素数を表にして表示するプログラム
20   1000 個の数を格納する空の表を用意する。(表の定義)
30   空の表に 2, 3, 5, ... と 1000 個の素数を順に格納する。(素数表の作成)
40   2, 3, 5, ... と 1000 個の素数を表示する。(素数表の表示)
30 End
```

ここで，20 行の処理を tbasic のプログラムとして書いてみます。

表は配列変数を使うことにし，配列変数の名前を例えば PrimeT とします。これを Dim 文を使って

```
Dim PrimeT(1000)
```

と宣言します。ここで，表には，

PrimeT(1)=2, PrimeT(2)=3, ..., PrimeT(1000)=1000 個目の素数

と素数を順に格納するとします。

このようにすると，上の擬似プログラムは，次のように書き直すことができます。

[*70] 素数を求める良く知られた方法として，エラトステネスの篩があります。これは与えられた自然数以下の素数を効率的に求める方法です。今の問題と多少目的が異なりますが，エラトステネスの篩の応用として今の問題を解くこともできます。
[*71] 実際にはプログラムとして動作はしないが，処理の内容をプログラム的に表示したもの。

Step3

```
10 '   2 から始めて小さい方から順に 1000 個の素数を表にして表示するプログラム
20  Dim PrimeT(1000)
30  PrimeT(1)=2,  PrimeT(2)=3, ... , PrimeT(1000)=1000 個目の素数とする。
40  PrimeT(1), PrimeT(2), PrimeT(3), ..., PrimeT(1000) を表示する。
30 End
```

30 行での処理と 40 行の処理は互いに独立なので，それぞれ副プログラムとして書くことができます。汎用性を考えて, 引数をつけます。それらをそれぞれ例えば, tbasic の書き方で, MakePT(n, P()), PrintPT(P(), FN, TN) としましょう。配列変数 P() はいずれも参照引き渡しで，Sub での変更が P() に反映されます。ここで，MakePT(n, P()) は，配列 P() に n 個の素数を格納する，PrintPT(P(), FN, TN) は配列 P() の FN 番目から TN 番目まで表示するという意味にします。これらは，tbasic では Sub で実現しますから，主プログラムから利用する場合は，Call を使います。

このようにすると，上の擬似プログラムは，次に書き直すことができます。

Step4 主プログラム

```
10 '   2 から始めて小さい方から順に 1000 個の素数を表にして表示するプログラム
20  Dim PrimeT(1000)
30  Call MakePT(1000,PrimeT())
40  Call PrintPT(PrimeT(), 1, 1000)
30 End
```

となります。

これで，目的とするプログラムの主プログラムができ上りました。非常に短い順構造のプログラムです。このプログラムの正当性の検証は分割された処理それぞれについて行えば良いことになります。後は，MakePT(n, P()) と PrintPT(P(), FN, TN) を作成すれば良いわけですが，これらのプログラムは独立に作成できることに注意しましょう[*72]。

3.7.3 副プログラム MakePT(n, P()) の作成

このサブプログラムの仕様を確定する必要がありますが，次の通りとします。

MakePT(n, P()) の仕様

配列 P に P(1)=2 から始めて，順に n 個の素数を P(n)=n 番目の素数まで格納する。

MakePT(n, P()) を作成するために，関数 NextPrime(n,P()) を使います。プログラムで必要な関数は，素数 p に対してその次の素数 p' を与えるものですが，ここでは少し一般化して，自然数 n に対してそれより大きい次の素数 p を与えるものとします。即ち，NextPrime(n,P()) は，自然数 n より大きい最小の素数を

[*72] お互いのプログラムの内容を知ること無しにそれぞれのプログラムを作成できますから，処理の仕様を示して，全くの別のプログラマーにプログラム作成を依頼することもできます。

与えるとします。例えば，NextPrime(3,P()) は 5 を与えますし，NextPrime(6,P()) は 7 を与えます。ここで，配列 P() はその計算に n 以下の素数表 P() を使うことを示しています[*73]。

NextPrime(n,P()) の仕様

NextPrime(n,P()) は，自然数 n に対して，n より大きい最小の素数与える。特に，n が素数のときは，次の素数を与える。

この関数を仮定すれば，MakePT(n, P()) は次のように簡単に作れます。

Step4-1 副プログラム MakePT(n,P())

```
Sub MakePT(n,P()) 'n 番目まで素数表を作成する
   P(1)=2
   For i=2 to n
      P(i)=NextPrime(P(i-1),P())
   Next i
End Sub
```

このプログラム単純な繰り返し構造をしていますので，NextPrime(n,P()) が上のような仕様で正しく動作すれば，その正当性は示されます。

次に，NextPrime(n,P()) を作成します。これも，素数判定を行う関数 IsPrime(n,P()) を使うことにします。IsPrime(n,P()) を使えば，素数が見つかるまで，順次数を増やしていけば良いだけなのでプログラムは次のように簡単に書けます。

Step4-1-1 副プログラム NextPrime(n,P())

```
01 Function NextPrime(n,P()) :' 素数表 P() は sqr(n) 以下まで作成済みのこと
02    If n<=1  then
03       NextPrime =2
04    Else
05       If (n mod 2) = 0 then CP = n+1 Else CP = n+2
06       Do  while not IsPrime(CP,P())
07          CP = CP+2
08       Loop
09       NextPrime = CP
10    End If
11 End Function
```

このプログラムは，分岐構造と繰り返し構造が組み合わされた，比較的単純な構造をしています。

2, 3 行は n が 0,1 の場合に，次の素数として，2 を返すと設定しています。4 行以下が n が 2 以上の場合です。5 行は素数の最初の候補 (candidate for prime)(CP) の設定です。n が偶数の場合は 1 を加えて，次の奇数が素数候補になります。n が奇数の場合は 2 を加えて，次の奇数が素数候補になります。6〜8 行で CP を素数判定し，素数になるまで，2 を加えながら繰り返します。素数は無限にあるので，この繰り返しは必ず終わります。

[*73] 実際に使うのは，$\sqrt{n}$ 以下の素数表です。

次に，関数 IsPrime(n,P()) を作成します[*74]。

IsPrime(n,P()) の仕様

n が素数なら，True を返し，合成数なら False を返す。但し計算の際，素数表 P() を使う。

作り方は単純で，素数表にある $\sqrt{n}$ 以下の素数で順次割っていき，割り切れるものがあれば，合成数，そうでなければ素数と判定します。以下のプログラムはこの方針で作ったものです。

Step4-1-1-1 副プログラム IsPrime(n,P())

```
01 Function IsPrime(n,P()) as Boolean
02 ' 素数表 P() は P(1)=2, かつ sqr(n) 以下まで作成済みのこと
03    IsPrime = True
04    nsqrt = Int(Sqr(n))
05    PCounter = 1
06    PF=P(PCounter)
07    Do while PF<=nsqrt
08       If (n mod PF = 0) then
09          IsPrime = False
10          Exit Do
11       End if
12       PCounter = PCounter + 1
13       PF=P(PCounter)
14    Loop
15 End Function
```

3 行で初期設定 IsPrime = True を行います。7 行〜14 行が繰り返しルーチンです。P(1)=2 となっていますから，n=2,3 のときは，$\sqrt{2}$ = 1.4..., $\sqrt{3}$ = 1.7... より，繰り返しルーチンは実行されず，素数と判定されます。n=4 以上のときが，実際に繰り返しルーチンが実行されます。8 行で素数 PF が n を割り切るか判定をします。割り切れた場合，9 行で IsPrime = False と設定して，合成数と判定して，脱出構文 ExitDo を使って Do 文の脱出をします。このようなことが起きなければ，IsPrime = True のまま，サブプログラムが終わり，素数と判定されます。

これで，副プログラム MakePT(n,P() が作成されました。

[*74] tbasic には自然数 n に対して，その最小素因数を返す関数 PFactor があります。これを使うと，関数 IsPrime は次のように簡単に 1 行で書けます。

```
Function IsPrime(n) as boolean
    If PFactor(n)=n then IsPrime = True else IsPrime = False
End Function
```

しかし，PFactor(n) は tbasic の特有の関数で，普通の BASIC にはありませんので，ここではより一般的な状況ということで，PFactor(n) を使わず，準備の項で考えた素数表を使う方法のプログラムを作ります。

3.7.4 副プログラム PrintPT(P(), FN, TN) の作成

次に素数表印刷の副プログラム PrintPT の作成をします。数値の桁数に関わらず表として見やすくするために，7 桁右寄せで表示することにします。このような場合 Print Using を使うことも考えられますが，ここでは，右寄せ関数 FlushR$を定義して使うことにします。1 行に 10 個の素数を表示して，改行をするとします。

PrintPT(P(), FN, TN) の仕様

素数表 P() の FN 番目から TN 番目の素数を 1 行に 10 個づつ 7 桁右寄せで表示する。

ここで右寄せ関数 FlushR$の仕様は次の通りとします。

FlushR$(n, m) の仕様

整数 n を m 桁右寄せで表示する。

これを使うとプログラムは次のようになります。

Step4-2 サブプログラム PrintPT(P(),S,E)

```
10 Sub PrintPT(P(), S,E ) :' 表 P() の S 番目から E 番目まで，7 桁右寄せで表示
20    For i=S to E
30       Print FlushR$(p(i),7);" ";
40       If ((i - S + 1) mod 10) = 0 then Print
50    Next i
60 End Sub
```

単純な繰り返し文です。30 行では，7 桁右寄せで表示し，1 スペースを空けて，改行はしません。40 行で，10 個の数の表示の度に改行をしています。

右寄せ関数 FlushR$ も単純です。次のように 1 行で書くことができます。

Step4-2-1 サブプログラム FlushR$(n, m)

```
Function FlushR$(n, m):' 数 n を m 桁で右寄せ
   FlushR$=Right$(Space$(10)+Str$(n), m)
End function
```

ここでは，m=7 を想定しているので，Space$(10) としました。m が 10 を超える場合は，必要に応じて，その値を大きくすれば良いでしょう。

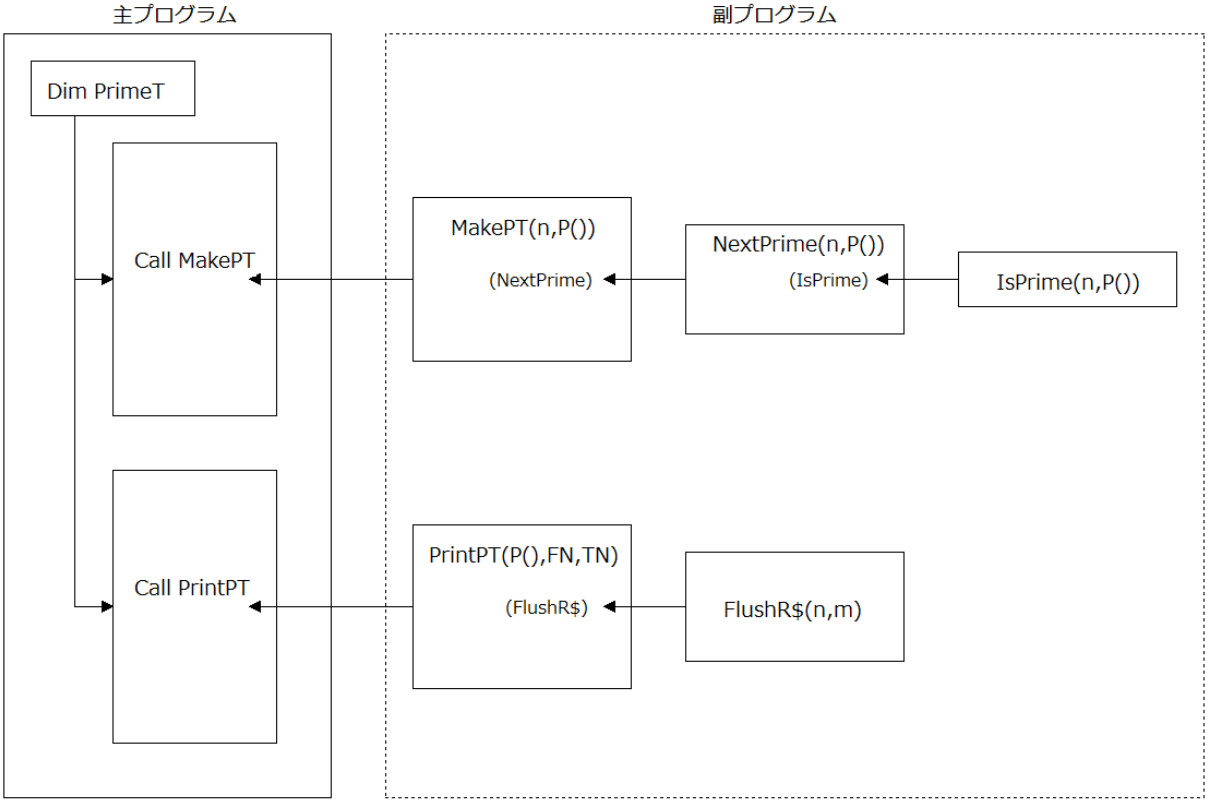
これですべてのプログラムができあがりました。まとめると以下のようになります。

例 3.7.1 素数を表示するプログラム

```
'   2 から始めて小さい方から順に 1000 個の素数を表にして表示するプログラム
 Dim PrimeT(1000)
 Call MakePT(1000,PrimeT())
 Call PrintPT(PrimeT(), 1, 1000)
End

Sub MakePT(n,P()) 'n 番目まで素数表を作成する
   P(1)=2
   For i=2 to n
      P(i)=NextPrime(P(i-1),P())
   Next i
End Sub

Function NextPrime(n,P()) :' 素数表 P() は sqr(n) 以下まで作成済みのこと
   If n<=1  then
      NextPrime =2
   Else
      If (n mod 2) = 0 then CP = n+1 Else CP = n+2
      Do  while not IsPrime(CP,P())
         CP = CP+2
      Loop
      NextPrime = CP
   End If
End Function

Function IsPrime(n,P()) as Boolean
' 素数表 P() は P(1)=2, かつ sqr(n) 以下まで作成済みのこと
   IsPrime = True
   nsqrt = Int(Sqr(n))
   PCounter = 1
   PF=P(PCounter)
   Do while PF<=nsqrt
      If (n mod PF = 0) then
         IsPrime = False
         Exit Do
      End if
      PCounter = PCounter + 1
      PF=P(PCounter)
   Loop
End Function

Sub PrintPT(P(), S,E ) :' 表 P() の S 番目から E 番目まで, 7 桁右寄せで表示
   For i=S to E
      Print FlushR$(p(i),7);" ";
      If ((i - S + 1) mod 10) = 0 then Print
   Next i
End Sub

Function FlushR$(n, m):' 数 n を m 桁で右寄せ
   FlushR$=Right$(Space$(10)+Str$(n), m)
End function
```

このプログラムの主プログラム，副プログラムの依存関係・配置を図示すると次のようになります。

主プログラム，副プログラムの配置



これらの関係・配置に注意すれば，主プログラム，副プログラムの修正や改良，検証が容易に可能であることが分かります。

このように構造化プログラミングは，元々大規模なプログラムを想定した手法ですが，この例に見るように，小規模なプログラムでも有効に働くことが分かります。

4　まとめ

大きなプログラムを如何に「**良いプログラム**」に仕上げるかが，構造化プログラミングの出発点でした。ここで「**良いプログラム**」とは，目的通りの動作をし，更に資産として蓄積価値のあるプログラムのことです。構造化プログラミングの提唱はプログラミングの一般的作法についての最初の体系的検討であり，多くの人々によって議論・検証されました。そしてその鍵となる姿勢は「分かりやすい」プログラムを書くことでした。言い換えれば，構造化プログラミングは，大きなプログラムを如何に分かりやすく書くかの方法論とも言えます。その方法は，副プログラムによるプログラムの細分化と基本制御構造によるプログラムの記述でした。

構造化プログラミングは大規模なプログラムに対する提唱ですが，この「分かりやすい」プログラムという考えは，小規模なプログラムに対しても有効です。構造化プログラミングの提唱以来，既に約半世紀が経過し，その間プログラミング言語及びその手法は，大きな発展を遂げています。しかし，構造化プログラミングの手法は現在でも有効で，その精神はプログラミングでの定石とも言えるものです。

構造化プログラミングの手法を基にして，「分かりやすい」プログラムを書きましょう。