基本情報技術者

試験対策テキストⅣ【アルゴリズム編】

無料体験入学者用

TAC

はじめに

基本情報技術者試験は，2023年春からの通年化に伴い，ITを活用したサービス，製品，システム及びソフトウェアを作る人材に必要な基本的知識・技能をもち，実践的な活用能力を身に付けた者を対象とした試験となっています。そのため，現代のIT社会に必要な幅広い知識が問われています。

本書は，基本情報技術者試験の科目B試験の中核となる「アルゴリズムとプログラミング」の内容に対応するテキストです。そして，IT分野について初心者の方でも無理なく学習が行えるよう，基礎的な用語や考え方を分かりやすく解説するように心がけました。

本書により，読者のみなさんが基本情報技術者試験に合格されることを願ってやみません。

TAC情報処理講座

アルゴリズム　目次

Part1

アルゴリズムの基礎

このPartでは，コンピュータに処理を行わせるための手順である“アルゴリズム”について学習していきます。

1-1 アルゴリズムとは何か

アルゴリズムを直訳すると“算法”となります。一般に「計算の手順」，あるいは「必要な処理結果を得るための手順」を意味します。なお，ここで説明されているすべてを覚えることはありません。概要をつかんでおきましょう。

基本知識の整理

アルゴリズムの意味

アルゴリズムというと，難しい計算をイメージするかもしれませんが，直接的な算術計算だけを意味するものではありません。たとえば，「東京から大阪へ行くための経路を求める」といった問題を解くための手順もアルゴリズムとよんでよいのです。

参考：JISによる定義

JIS(日本産業規格)では，「明確に定義された有限個の規則の集まりであって，有限回適用することにより問題を解くもの」と定義しています。ここで“問題を解く”とは，コンピュータで処理させ，結果を得ることをいいます。より具体的にいえば，「コンピュータに“データ”を入力し，これをプログラムで“処理”することで，利用者に役立つ(より付加価値の高い)“情報”を導くこと」といえます。

複数の計算手順

一般に，ある問題の答えを得るための手順，すなわちアルゴリズムは複数存在します。たとえば，「1から10までの和」を得るには，

1＋2＋3＋…＋9＋10

というように1から順に一つずつ大きい数を足していってもよいですし，逆に

10＋9＋8＋…＋2＋1

というように10から順に一つずつ小さい数を足していってもよいわけです。どちらも答えは55ですね。また，少し工夫すると，

1＋10＝11
2＋ 9＝11
3＋ 8＝11

4＋7＝11

5＋6＝11

というように，11の組が五つできますから，

11×5

としても結果は55となります。

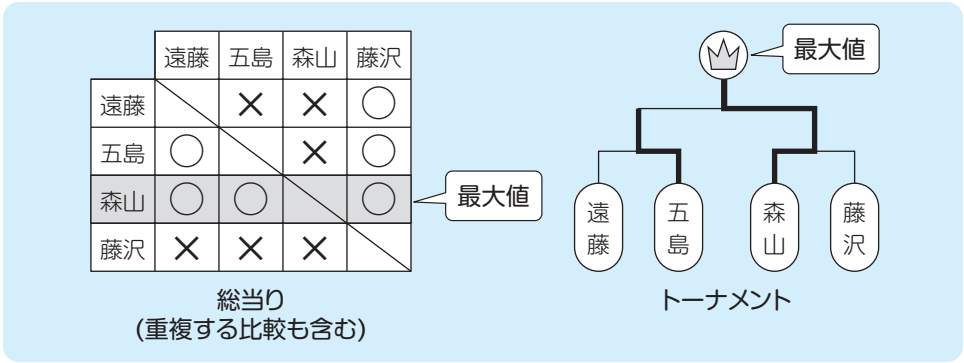
理解を深めよう

アルゴリズムの効率

ある問題を解くためのアルゴリズムは一つとは限りません。例として「受験者の試験結果から最高点を求める」ためのアルゴリズムを考えてみましょう。

最高点は受験者の得点を“総当り”に比較しても求めることができます。総当りの結果，ほかのすべての得点よりも高い得点が最高点となるでしょう。

また，受験者の得点を“トーナメント”で比較することでも最高点を求めることはできます。



しかし，単に最高点を求めるのであれば，総当りで求めるようなことは，まずしません。なぜならば，効率が悪いからです。たとえば4人の受験者がいる場合，総当りに必要な比較回数は，自分自身との比較を除けば，

4×3＝12［回］

となります。つまり，総当りでは4人の受験者の中から最高点を選ぶのに「12回の比較」を行なわなければならないわけです。

これに対して，4人の受験者の得点を“トーナメント”で比較する場合を考えてみましょう。その際の比較回数（試合回数）は，

2＋1＝3［回］

で済むことになります。

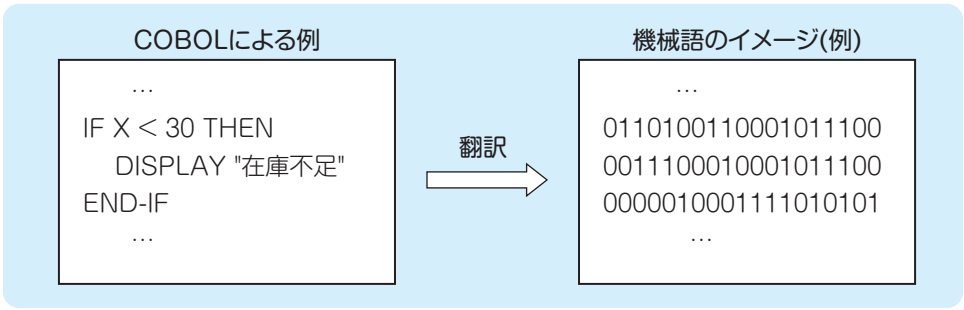
コンピュータのCPUは，こうした比較処理を一つひとつ処理していきますから，同じ結果が得られるのであれば，より少ない処理となるようなアルゴリズムのほうがよいことがわかるでしょう。

発展知識

プログラム言語とアルゴリズム

プログラムは，コンピュータに一連の動作を行わせるための命令の集まりです。コンピュータのCPUは，“0”と“1”の数字の並び（数字列）で表された機械語（マシン語ともいいます）による命令しか理解することができません。“0”と“1”の数字列では，人間が直接理解するには困難を伴います。そこでCやCOBOL，Javaといった，より人間が用いる言語（自然言語という）に近い**プログラム言語**が開発されてきました。このようなプログラム言語のことを**高水準言語**とよびます。

高水準言語は，機械語に比べればはるかに人間が理解しやすいものになっていますが，CPUは高水準言語を直接理解し，その命令を実行することはできません。そこで，**コンパイラ**や**インタプリタ**などの**言語プロセッサ**によって，高水準言語の命令を機械語命令に置き換えてから，実行することになります。



コンピュータに目的の動作を行わせるためには，意味のある命令の並びでなければなりません。目的の動作を行わせて，要求する処理結果を得るためには，命令の並びは処理手順を表すようになっていなければならないのです。この処理手順がアルゴリズムです。

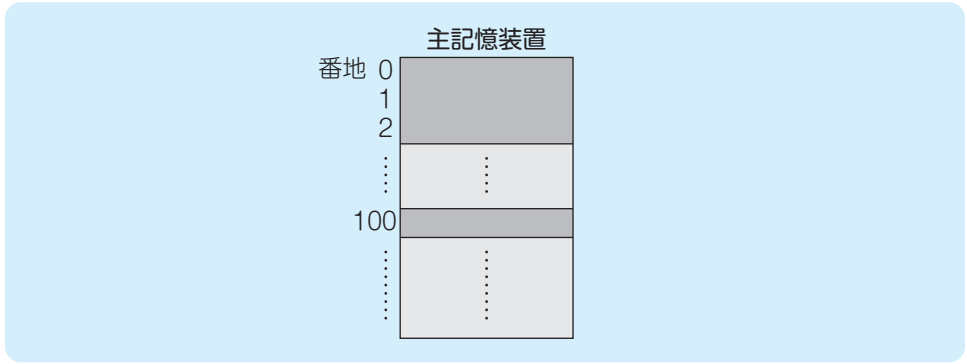
たとえば，上記の図のCOBOLによる記述は，「Xが30 より小さかったら“在庫不足”と表示する」ことを示しています。在庫数が30未満であれば，在庫不足として注意を促すような目的をもつアルゴリズムをCOBOLで表現したものといえます。

データの格納場所と記憶装置

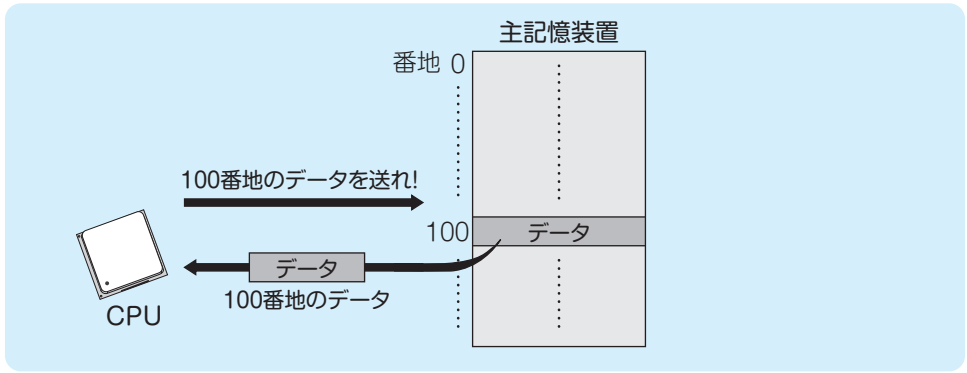
コンピュータに何か作業をさせる場合には，必要なデータを与えておかなければなりません。そのためにはデータを格納する場所がなければなりません。また，処理した後のデータを格納する場所も必要となります。この役割を担うのが記憶装置です。

記憶装置は，半導体メモリで構成される主記憶装置と，ハードディスク装置などの補助記憶装置に分かれます。ここでは，それぞれの装置の詳細には触れませんが，これらの装置にはデータの格納場所を示す“アドレス”（番地ともいう）とよばれるものがつけられているということを理解しておきましょう。

たとえば，主記憶装置には次の図のようにアドレスがつけられています。



コンピュータのCPUは，このアドレスを指定して，主記憶装置からデータを取り出したり，反対にデータを格納したりします。



参考：プログラム言語とデータの格納場所

プログラム言語を用いてプログラミングを行う場合，データの格納場所を考えておかなければなりません。機械語でプログラムを記述する場合には，データの格納場所も“0”と“1”の数字列で示されたアドレスを用いて指定する必要があります。

しかし，CやJavaなどの高水準言語では，直接格納場所のアドレスを指定しなくてもよいようになっています。これらの高水準言語には，“変数”などとよばれる，データを格納する入れものが用意されています。この変数をプログラム内で定義すると，翻訳された機械語プログラムを実行する際に，それぞれの変数用の格納場所として，主記憶装置上に特定のアドレスが割り当てられます。こうした変数をプログラム内であらかじめ定義することを変数の“宣言”とよびます。

1-2 変数と定数

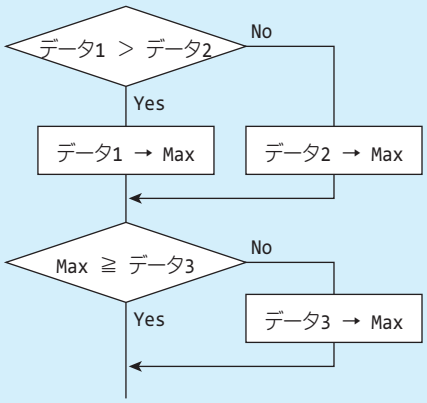
ここでは，アルゴリズムの基礎知識として，変数や定数について学習します。

基本知識の整理

アルゴリズムの表現方法

アルゴリズムは，さまざまなプログラム言語で表現できます。しかしながら，あるプログラム言語で記述すると，そのプログラム言語を理解できない人にはアルゴリズムがわかりません。また，いきなりプログラム言語で記述すると，全体像が把握しにくくなります。そこで，プログラム言語で記述する前に，アルゴリズムだけに注目して，それを記述する必要がでてきます。このアルゴリズムを記述することを目的とした表現技法の代表例に，**流れ図**（**フローチャート**）と**擬似言語**があります。

フローチャート



擬似言語

```
if (データ1 が データ2 より大きい)
  Max ← データ1
else
  Max ← データ2
endif
if (データ3 が Max より大きい)
  Max ← データ3
endif
```

流れ図には視覚的なわかり易さがあります。その反面，記述の自由度が高く，注意しないと煩雑で分かりにくいものになりがちです。擬似言語は視覚的なわかりやすさこそ流れ図に譲りますが，プログラム言語に近くプログラミングとの相性は抜群です。

変数とは

変数は，データを格納する“箱”のようなものと考えてください。プログラムはこの変数を対象に処理を行うので，データをプログラムで処理する場合には，処理に先立ちデータを変数に格納しなければなりません。アルゴリズムの学習をするために，変数とはどういうものかをしっかりと理解しましょう。

定数とは

定数とは，読んで字のごとく値の定まったものをいいます。数値の5は常に5ですから，すなわち定数です（正しくは**数値定数**とよびます）。また，定数には数値だけでなく文字もあり，これを**文字定数**とよぶことがあります。

変数の性質

変数には次にあげるような特徴があります。重要ですから，しっかり覚えましょう。

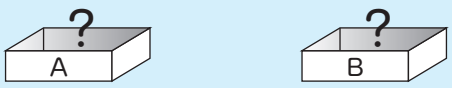
①変数には名前（変数名）がついている

変数には，名前をつけられます。これを**変数名**とよびます。この変数名によって，どの変数であるか判別されるわけです。変数名は自由につけてかまいませんが，わかりやすくする必要があります。流れ図を記述する場合は特に制約はありませんが，プログラム言語を用いてプログラミングする場合は，そのプログラム言語ごとの制約（文字種類や文字数など）に従わなければなりません。

②変数の中には最初は何が入っているかわからない

プログラムの内部で変数Aを宣言しておく（Aという名前の変数を使用することをコンピュータシステムに知らせる）と，コンピュータシステムはデータを格納する領域をメモリ上に確保します。このとき，確保した領域にどのようなデータが格納されていたかはわかりません。この場合，変数に値が格納されていない状態として，**未定義**（不定）とよびます。つまり，変数Aを用意しただけでは，その変数Aは最初は“未定義”となります。

この性質は重要です。



③変数には，値を入れることができる（代入）

変数に値を入れることを**代入**とよびます。たとえば，擬似言語において，変数Aに10（数値定数）を代入するときには，

A ← 10　（あるいは　10 → A）

のように記述します。変数には数値・文字などの値を代入することができますが，数値を格

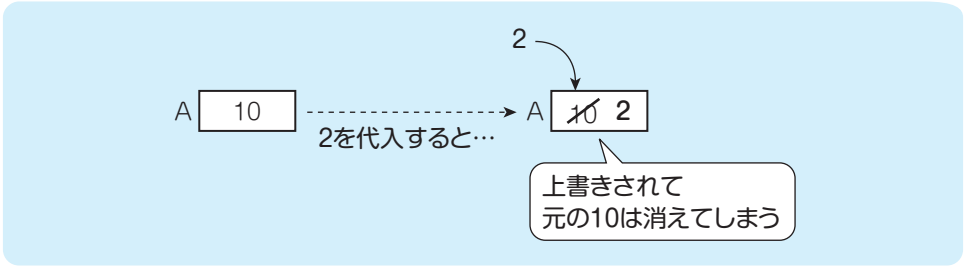
納するための変数に文字を代入したり，文字を格納するための変数に数値を代入することはできません。なお，変数に**初期値**（最初の値）を代入することを，**初期化**といいます。

また，変数に**未定義の値**を代入することもでき，その場合，その変数は“未定義”になります。

A ← 未定義の値

④変数に格納できる値は一つだけである

変数には，同時に一つの値しか格納できません。たとえば，あらかじめ変数Aに10が格納されている場合に，その変数Aに2を代入する場合を考えてみましょう。この代入により，変数Aの内容は10から2に更新され，もともと格納されていた値10は消えてしまいます。

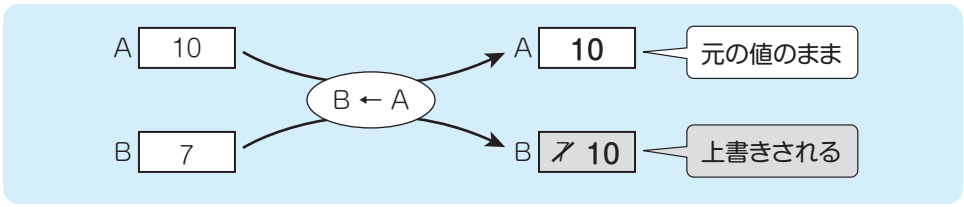


⑤変数には，ほかの変数の値を代入することができる

変数には，ほかの変数の値を代入することができます。たとえば，変数Aの内容を変数Bに代入するときには，

B ← A

のように記述します。このとき，変数Aの内容が変数Bにコピーされると考えましょう。移動するわけではないので，変数Aの内容は変わりません（元の値のまま）。



⑥変数には，演算結果を代入することができる

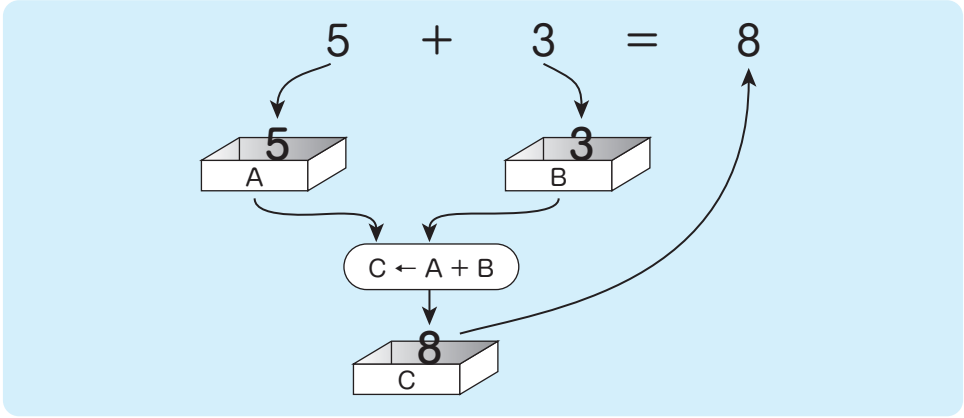
変数には，演算の結果を代入することができます。たとえば，

B ← A ＋ 5

は，「変数Aの内容に5を加えた結果を，変数Bに代入する」ということです。もちろん，変数どうしの演算でもかまいません。この場合は，

C ← A ＋ B

のように記述され，「変数Aの内容と変数Bの内容の合計を，変数Cに代入する」という意味になります。たとえば，変数Aに5，変数Bに3が格納されているとすると，「C ← A ＋ B」によって変数Cに“5＋3”の結果である「8」が格納されることになるわけです。



演算子には，通常の四則演算子（＋－×÷）のほかに，剰余を求める**mod**を使います。以下は，式の一例です。

式	意　味
B ← A × 2	Bには「Aを2倍した値」を代入する
C ← A ＋ B	Cには「AとBの和」を代入する
B ← A mod 2	Bには「Aを2で割った余り」を代入する
A ← B ＋ C ÷ D	Aには「B」＋「CをDで割った値」を代入する
A ← (B ＋ C) ÷ D	Aには「BとCの和をDで割った値」を代入する

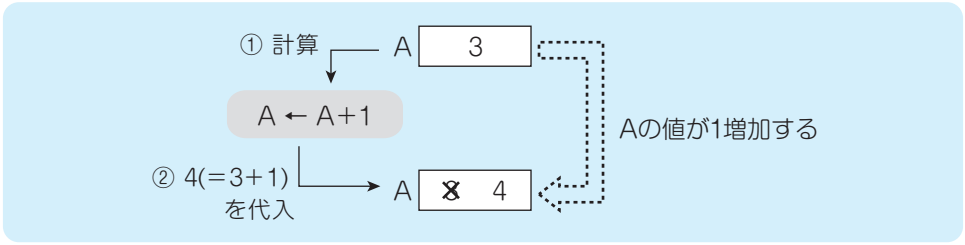
変数の内容は，代入によって変わるものであり，式によって変わるものではないことに注意してください。たとえば，「B ← A × 2」を実行したとき，Bの値はA×2で更新されますが，Aの内容は変化しません。

⑦自分自身に計算結果を代入することができる

計算元のデータが格納されている変数に計算結果を代入することもできます。これは，ある変数の値を増加／減少させる演算です。以下に一例を示します。

式	意　味
A ← A ＋ 1	変数Aを1増加させる
A ← A － B	変数AからBの値を減じる
A ← A × 2	変数Aを2倍する
A ← A ÷ B	変数Aの値を1/B倍する

たとえば，変数Aの内容が3であるときに「A ← A ＋ 1」が実行されたときには，次の順序で計算・代入が行われ，結果としてAが1増加したことになるのです。



変数の値を増減する処理は，**インクリメント**（増加）／**デクリメント**（減少）ともよばれ，プログラムでは頻出します。

インクリメントやデクリメントを行う変数は，適切に初期化されていなければなりません。なぜなら，変数の値が未定義であった場合，それを増加・減少しても意味はないからです。

文字定数と変数名の区別

流れ図を書く場合，ある数値や文字が定数を意味するのか変数名であるのかを区別できなければなりません。たとえば，単に

A

とあった場合，Aという変数名の変数を意味しているのか，文字のデータである文字定数のAなのかわかりません。そこで，文字定数を記述する場合は，

"A"，'A'

というように，ダブルクォーテーション（" "）やシングルクォーテーション（' '）でくくります。ですから，単に

A

とある場合は，変数A（変数名Aの変数）を意味し，

"A"，'A'

とある場合には，文字データのAであることになります。

さらに，"10" と記述する場合があります。これは，文字データとしての "10"（文字定数）を意味します。単に

10

とある場合は，数値データとしての10（数値定数）を意味します。数値データは数値の演算に使えますが，文字データは数値の演算には使えません。つまり，５＋３を計算して結果の８を変数Aに格納したい場合は，

A ← 5 + 3

と記述します。"5"，'3' のようにダブルクォーテーションやシングルクォーテーションでくくると，数値計算の対象データとはみなされないわけです。

なお，"10" や10はそれぞれ文字定数，数値定数を意味しますから，数字だけを使って変数名とすることはできないことを理解しておいてください。ただし，

A10

というような変数名は使うことができます。

理解を深めよう

型と宣言

変数は，取り扱うデータの種類によって，いくつかのグループに分けられます。このグループを**型**とよびます。代表的な型には次のものがあります。

整数型	整数を取り扱う変数
実数型	実数を取り扱う変数
文字型	文字を取り扱う変数
文字列型	文字のまとまり(文字列)を取り扱う変数
論理型	true(真)またはfalse(偽)の2値を取り扱う変数

整数型の変数は，実数を保持できません。ある式の値を整数型変数に格納するような場合には，値の小数点以下が切り捨てられて代入されます。

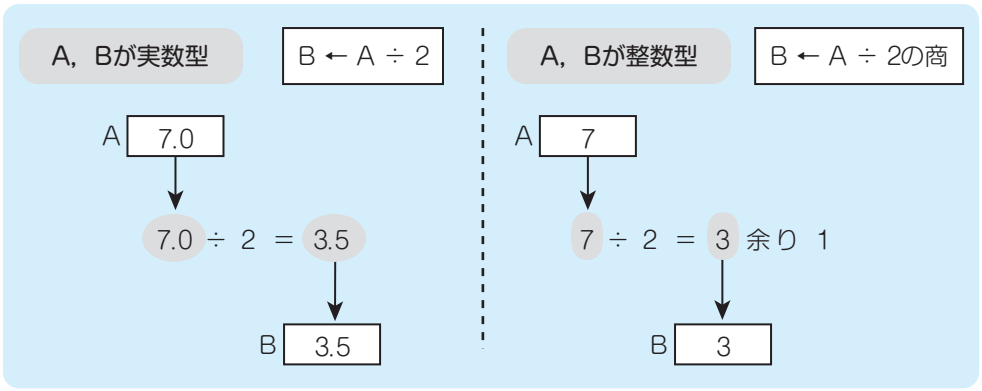
また，擬似言語では，「整数同士の除算で，整数の商を結果として得る」場合，

　　7 ÷ 2 の商

のように記述することで，

　　7 ÷ 2 ＝ 3 余り 1　　（7＝3×2＋1の3が商，1が剰余）

という除算結果の商である 3 が求められます。



擬似言語では，変数を用いる場合に前もって変数を宣言します。変数宣言は，型と変数名との組合せで行います。

```
整数型: N　　← 整数型変数 N を用いる
実数型: R　　← 実数型変数 R を用いる
文字型: C　　← 文字型変数 C を用いる
文字列型: S　← 文字列型変数 S を用いる
論理型: F　　← 論理型変数 F を用いる
```

変数の型と定数の代入

整数や実数の定数は，普段用いている書き方で指定できます。整数値であっても，実数として取り扱っていることを明示するために「.0」をつけることもあります。

```
整数型: N
実数型: R
N ← 3
R ← 3.0
```

文字定数は，変数と区別するためにダブルクォーテーションやシングルクォーテーションで囲みます。

```
文字型: C
文字列型: Word
C ← "A"              ← 文字型変数Cに，文字Aを代入
Word ← "ABC"         ← 文字列型変数Wordに，文字列ABCを代入
```

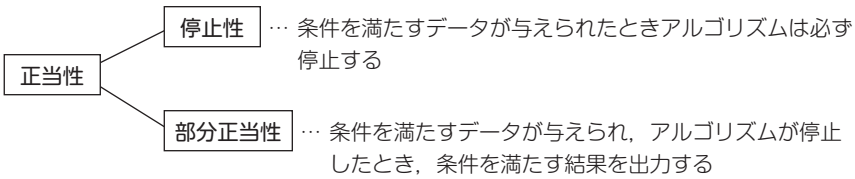
論理型変数には，trueまたはfalseを設定します。

```
論理型: Flag
Flag ← true
```

参考：アルゴリズムの正当性

アルゴリズムは正しくなければなりません。アルゴリズムが正しいことを評価する基準に正当性があります。

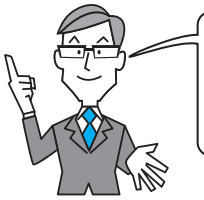
正当性は停止性と部分正当性という側面をもちます。



停止性と部分正当性をあわせた基準を，全正当性とよびます。全正当性を満たすアルゴリズムは，条件を満たすデータが与えられたとき「必ず停止し，かつ出力条件を満たす結果を出力」します。

1-3　基本制御構造　その1

～ 順次と分岐 ～



アルゴリズムの表現方法である擬似言語と流れ図(フローチャート)を用いて，基本制御構造とよばれる三つの処理構造を学習します。ここでは，順次と分岐を取り上げます。

基本知識の整理

基本制御構造

プログラム構造の基本となるのは，

順次

選択（分岐）

繰返し（ループ）

の三つの制御構造であり，これらを**基本制御構造**といいます。

また，この基本制御構造だけを用いて読み易いプログラムを書く，という考え方を**構造化プログラミング**といいます。

擬似言語プログラムの書き方

擬似言語プログラムは，**宣言部**と**処理部**から構成されます。

プログラム	宣言部	プログラム(手続，関数)や変数を宣言する
	処理部	プログラムで行う処理を記述する

変数の宣言では，プログラムで使用する変数について，型と変数のリストを記述します。同じ型の変数であれば，1行にまとめても構いません。また，宣言と同時に，初期値を代入することもできます。

```
整数型: A, B, C          /* 三つの整数型変数A，B，Cを用いる */
実数型: Rate ← 2.5       /* 初期値を2.5とした実数型変数Rateを用いる */
```

処理部には，プログラムで実行する命令を順番に記述します。

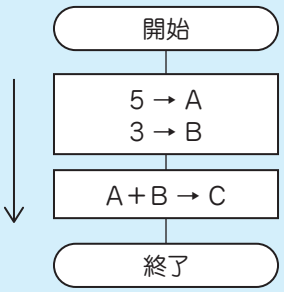
```
A ← 0        /* 最初の処理：Aに0を代入する */
B ← A        /* 次の処理 */
C ← B        /* 最後の処理：これでA～Cがすべて0になった */
```

記述された各処理は，原則として上から下へ順に処理されていきます。このような制御構造を，“**順次**”といいます。

流れ図の書き方

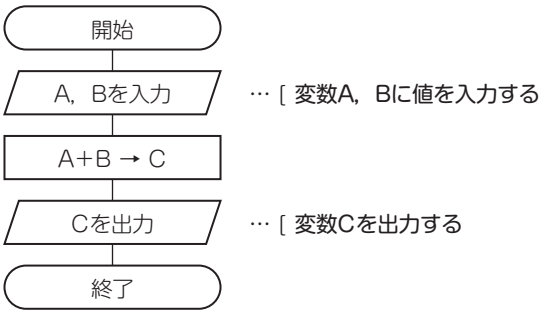
流れ図は，一番上の開始を表す端子から処理が始まり，一番下の終了を表す端子で処理が終わります。実行する処理は，長方形の記号（基本処理記号）の中に記述します。

流れ図に記述された処理は，擬似言語と同様に上から下へ順に処理されていく（順次）と考えてください。一つの長方形の中に，複数の処理を記述することもできますが，この場合も上に記述された処理が先に行われます。



参考：データの入出力(流れ図)

流れ図では，データの入出力を記述するための記号があります。平行四辺形の記号中に入出力内容を記述します。たとえば，変数Aと変数Bにデータを入力して(キーボードなどから)，AとBの内容の和を変数Cに格納して，そのCの内容を出力する(ディスプレイに表示するなど)流れ図は，次の図のように記述することができます。



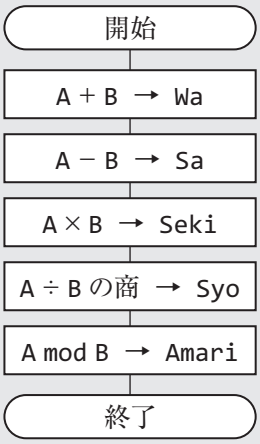
Example

変数A，B（共に整数型，値は格納済み）の四則演算を行うアルゴリズムを，擬似言語及び流れ図で表したものです。A，Bの和（足し算の答え）をWa，差（引き算の答え）をSa，積（掛け算の答え）をSeki，商（割り算の答え）をSyo，割り算の余りをAmariという変数にそれぞれ格納します。

〔擬似言語〕

```
○Calculate()
  整数型: A, B, Wa, Sa, Seki, Syo, Amari
  Wa ← A + B
  Sa ← A − B
  Seki ← A × B
  Syo ← A ÷ B の商
  Amari ← A mod B
```

〔流れ図〕

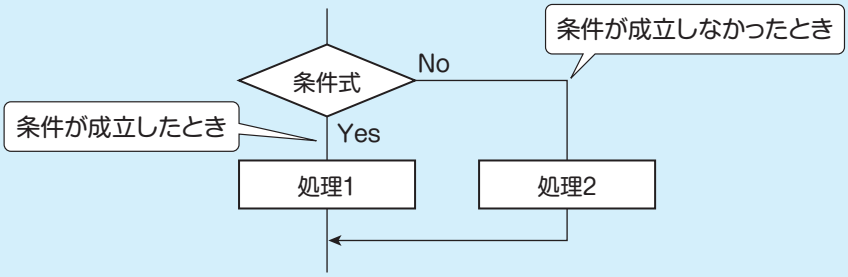


分岐(選択)

条件によって，実行する処理を分けることを，“**分岐**(**選択**)”といいます。
擬似言語プログラムでは，**if**文(**if ～ else**文)を用いて，次のように分岐を記述します。

```
if (条件式)
  処理1        … 条件が成立したときの処理
else
  処理2        … 条件が成立しなかったときの処理
endif
```

一方，流れ図では，ひし形の記号に条件式を記述し，その条件の判定結果によって，次のようにYes，Noに分岐させます。

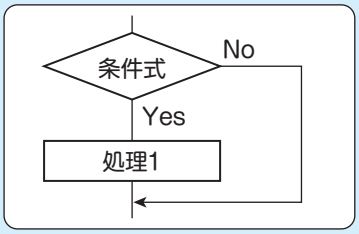


これらの例では，条件が成立した場合(条件式が真の場合)には，処理1が実行されます。このときには処理2は実行されないわけです。逆に，条件が成立しなかった場合(条件式が偽の場合)には，処理2が実行され，処理1は実行されません。
また，条件が成立しなかった場合に何も処理を行わないのであれば，次のように記述します。

擬似言語

```
if (条件式)
  処理1
endif
```

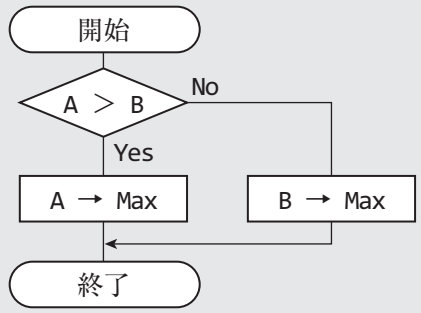
流れ図



Example

変数A，B(共に整数型，値は格納済み)のうち，大きい方の値を変数Maxに格納します。

```
if (A が B より大きい)
  Max ← A
else
  Max ← B
endif
```

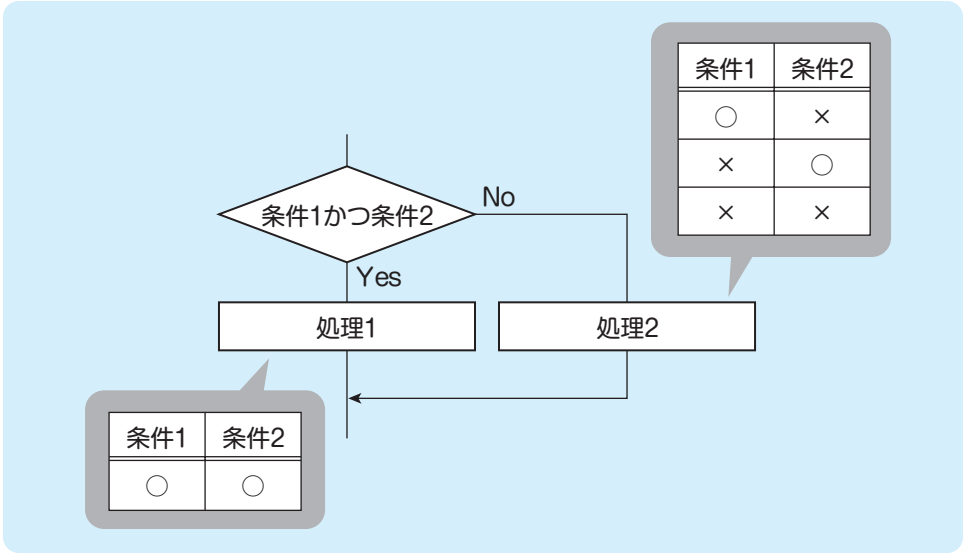


理解を深めよう

複合条件

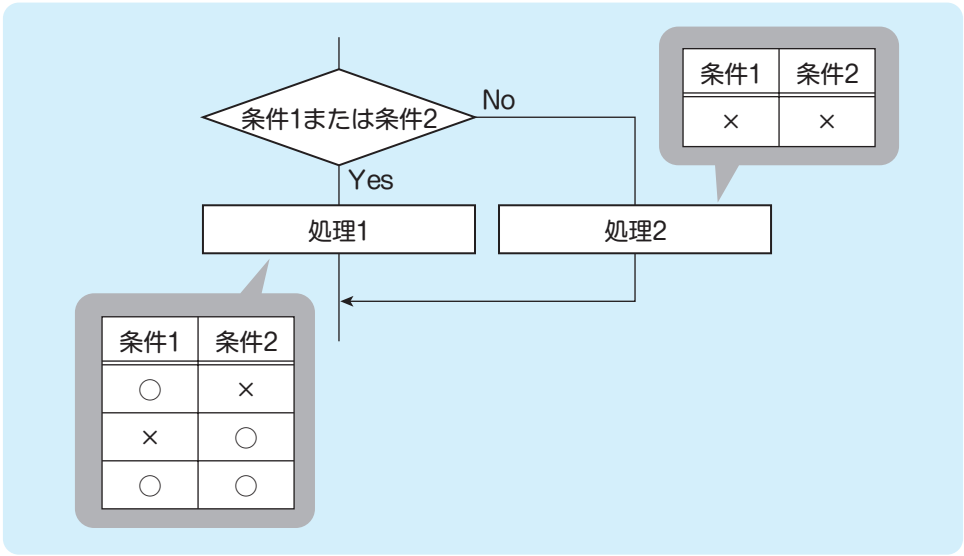
複数の条件を“かつ(and)”や“または(or)”で結んだものを**複合条件**とよびます。この“かつ(and)”は次の図のように条件1と条件2がともに満たされたときYes側に進み，どちらか一方の条件が満たされない場合や，ともに満たされない場合はNo側へ進むことになります。

図中の“○”は条件が満たされたことを，“×”は条件が満たされないことを示していると考えてください。たとえば，判断記号の中が“A ≧ 10 かつ A ≦ 20”となっていた場合，変数Aの内容が10以上20以下であれば，両方の条件を満たしますからYes側へ進み，処理1が実行されることになります。



“かつ”で結ばれた条件

次に“または(or)”を考えてみましょう。この“または”は次の図のように条件1と条件2のどちらか一方，あるいは両方が満たされたときYes側に進み，両方の条件が満たされない場合はNo側へ進むことになります。たとえば，判断記号の中が“A < 10 または A > 20”となっていた場合，変数Aの内容が10より小さいか，あるいは20より大きければ，どちらか一方の条件を満たしますからYes側へ進み，処理1が実行されることになります。



“または”で結ばれた条件

また，条件を入れ子構造にすることもできます。

Example

(1) 年齢(Age)が20代であれば顧客区分(Div)を“A”に，そうでなければ“B”にする。

〔プログラム1〕

```
if ((Age が 20 以上) and (Age が 29 以下))
  Div ← "A"
else
  Div ← "B"
endif
```

〔プログラム2〕

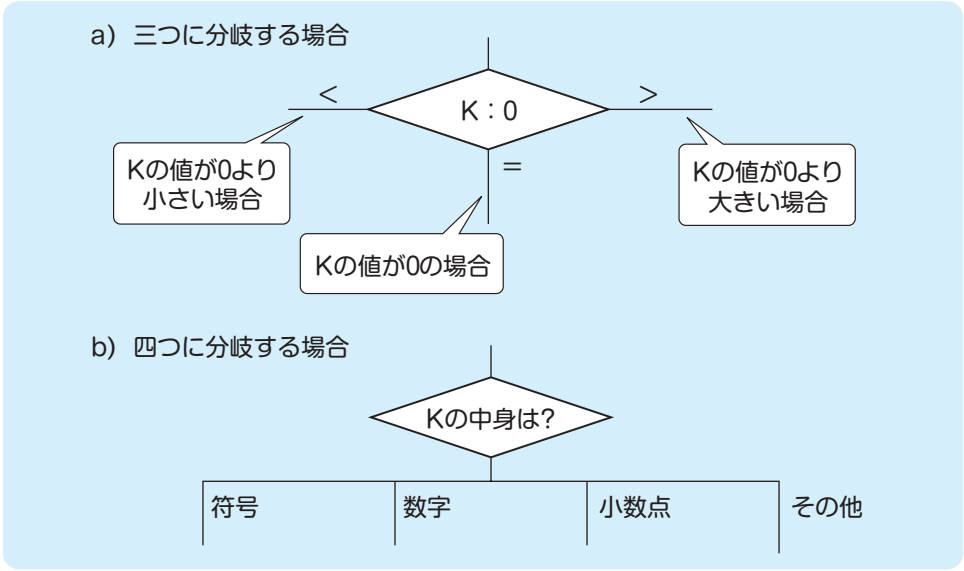
```
Div ← "B"
if ((Age が 20 以上) and (Age が 29 以下))
  Div ← "A"
endif
```

(2)　年齢(Age)が20代の男性(S＝1)の顧客区分(Div)を“A”に，20代女性(S＝2)の顧客区分を“B”に，それ以外を“C”にする。

```
Div ← "C"
if ((Age が 20 以上) and (Age が 29 以下))
  if (S が 1 と等しい)
    Div ← "A"
  else
    Div ← "B"
  endif
endif
```

多分岐

ここまで説明してきた分岐は，処理が二つに分かれる構造ですから“**二分岐**”とよびます。これに対して三つ以上に分岐先が分かれる分岐の構造を“**多分岐**”とよびます。流れ図では，多分岐を次のように記述できます。



多分岐の構造

擬似言語プログラムでは，if文に「**elseif**」を組み込んで，次のように記述します。

```
if (K が 0 と等しい)
  処理1     … K＝0のときの処理
elseif (K が 0 より小さい)
  処理2     … K＜0のときの処理
else
  処理3     … K＞0のときの処理
endif
```

このif文では，条件式を上から順に評価し，最初に真になった条件式のところに記述された処理を実行します。条件式のいずれも真にならなかった場合は，elseに記述された処理を実行します。

なお，elseifは複数組み込むことができます。また，条件式のいずれも真にならなかった場合のelseを省略することもできます。

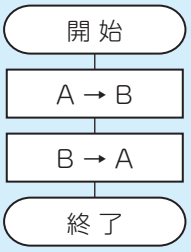
1-4　変数どうしの内容の交換

変数に格納された内容(データ)を交換する処理は頻繁に登場します。基礎中の基礎ですから正しく理解しましょう。

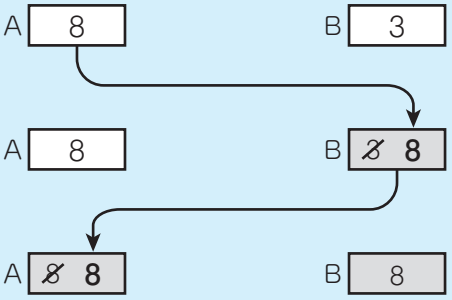
基本知識の整理

変数の性質と誤った交換の方法

変数に格納できる値は一つだけです。ですから，ある値を格納している変数に新たに値を格納すると，元の値は上書きされてしまいます。それを念頭に，右の流れ図を見てください。



この流れ図では，変数どうしの中身を正しく交換することができません。なぜなら，最初に行われる“A → B”で変数Aの内容が変数Bにコピーされ，Bの元の内容が失われてしまうからです。次の図はAの元の内容が8，Bの元の内容が3であった場合に上の流れ図を実行した場合の様子を示しています。結果として，A，Bの内容はともに元のAの内容である8になってしまうのです。もちろん，流れ図の処理の順序を入れ替えてもうまくいきません。その場合は，ともに元のBの内容である3になってしまいます。



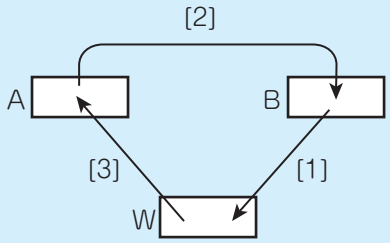
理解を深めよう

正しい交換の方法

先の手順ではどうしてうまくいかないのでしょう。それは，「変数に格納できる値は一つだけ」だからです。そこで，もう一つ別の変数を用意してあげることにしましょう。ここではそれを変数Wとします。

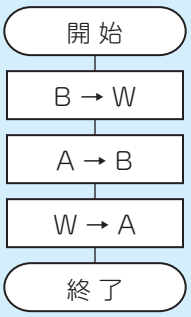
変数Aの内容をいきなり変数Bにコピーするのではなく，元のBの内容がなくならないように「あらかじめBの内容をWにコピーしておき」，その後で「Aの内容をBにコピーし，次にWからAにコピーする」という方法をとればよいのです。このときの変数Wは，変数Bの内容を“**退避**”しておくために用いられているので，“**退避用変数**”あるいは“**作業用変数**”とよばれます（ワーク変数，または単にワークとよぶこともあります）。

[1] W ← B　(Wは退避用変数)
[2] B ← A
[3] A ← W



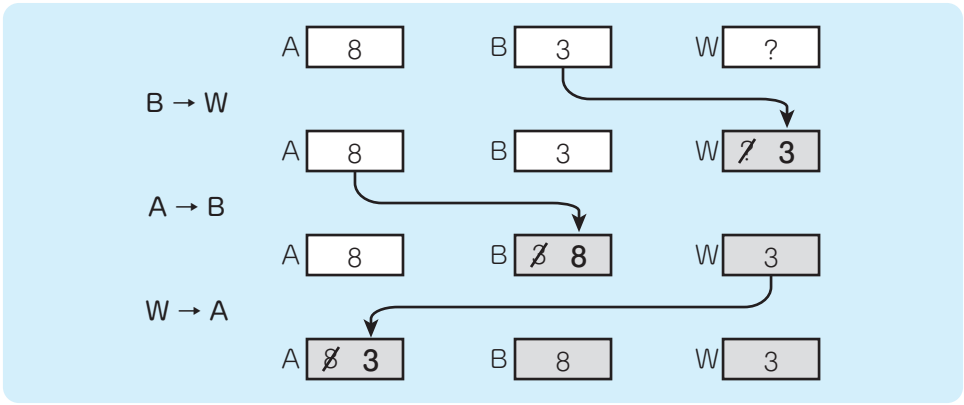
退避用変数Wを用いた交換のイメージ

上記の手順を流れ図で示すと次のようになります。



正しい交換の流れ図

流れ図に従って処理を進めていく場合の様子を次に示します。



正しい交換の様子

今度はうまくＡとＢの内容を交換できました。なお，このとき，退避用変数Ｗの最初の中身は処理結果に影響しませんから，特に値を定めておく必要はありません（図中の？は変数の内容が不定であることを示しています）。

さて，上記の例では，はじめに変数Ｂの内容を退避用変数Ｗに退避しました。もちろん，変数Ａの内容を退避用変数Ｗに退避しても正しく交換を行うことができます。

その場合の処理順序は，

① Ｗ ← Ａ
② Ａ ← Ｂ
③ Ｂ ← Ｗ

となります。本当にうまくいくか，図を書いて確かめてみるとよいでしょう。

Example

変数Ａ，Ｂの内容を入れ替えるプログラムを作成します。Ａ，Ｂは共に整数型であり，宣言と同時に値を設定しています。

```
○SwapData()          // プログラム名はSwapData
  整数型: A ← 8      // 整数型の変数A，Bを用いる
  整数型: B ← 3
  整数型: Work       /* 整数型の退避用変数Workを用いる */
  Work ← A           /* AとBを交換する手順 */
  A ← B
  B ← Work
```

なお，ここで使用している

/* … */　や　// …

は，いずれも“**注釈**”を記述するためのものです。プログラムを実行する際には，これらの“注釈”は無視されるため，実行に影響はありません。

1-5 基本制御構造 その2

～ 繰返し ～

擬似言語と流れ図(フローチャート)を用いて，基本制御構造の一つである繰返し(ループ)構造を学習します。

基本知識の整理

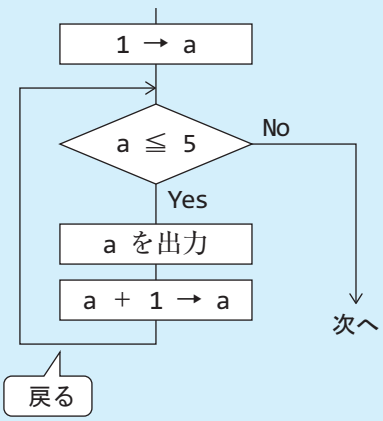
繰返し処理(ループ処理)

コンピュータの処理では，同じ処理を繰り返す場面が数多く登場します。そのような場合に使用するのが，“**繰返し**(**ループ**)処理です。

擬似言語プログラムでは，**while**文を用いて次のように記述します。

```
while (継続条件)
  継続条件が成立している間繰り返す処理
endwhile
```

〔流れ図〕



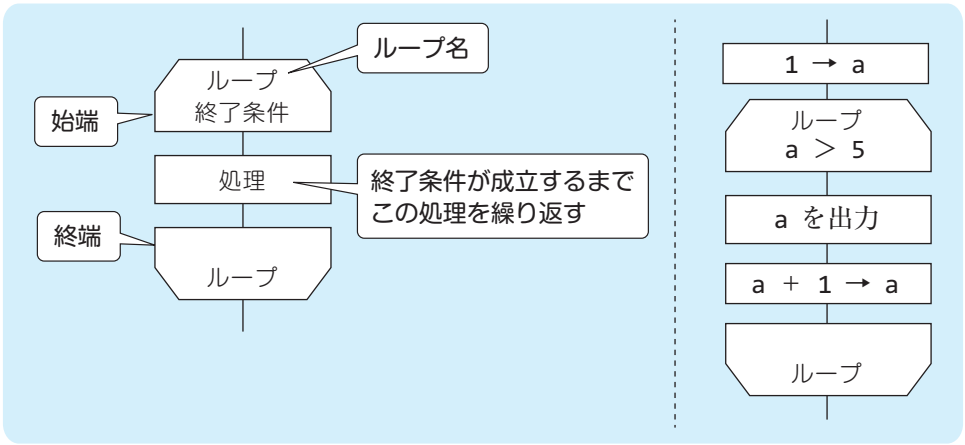
〔擬似言語〕

```
a ← 1          継続のための条件
while (a が 5 以下)
  a を出力
  a ← a + 1    繰り返す
endwhile
```

whileとendwhileに挟まれた範囲が，一つの繰返し処理を表しています。while文では，**継続条件**（ループを続ける条件）が記述され，この条件が成立している（真である）間，処理を繰り返します。なお，継続条件には“かつ（and）”や“または（or）”で結んだ，複合条件を用いることもできます。

流れ図における繰返し処理

流れ図では，繰返し処理を次のように記述します。始端と終端には，同じループであることを表す“ループ名”が記述されます。



流れ図では，擬似言語と異なり，**終了条件**（ループを止める条件）を記述します。そのため，流れ図の繰返し処理では，終了条件が真となるまで，処理を繰り返すことになります。

> 擬似言語と流れ図の繰返し処理で使用する条件を，確実に覚えておきましょう。
>
> **擬似言語の繰返し処理：継続条件**
> **流れ図の繰返し処理　：終了条件**

前判定繰返し処理

ここまで扱ってきた「while文」及び「流れ図で始端に終了条件が記述されている繰返し処理」のことを，**前判定繰返し処理**といいます。前判定繰返し処理では，

条件の評価 → 処理 → 条件の評価 → 処理 → … → 条件の評価 → ループ終了

という流れとなります。つまり，継続条件や終了条件のの評価を行った後に処理を行うような繰返しとなるわけです。

後判定繰返し処理

前判定繰返し処理とは異なり，処理を実行してから継続条件や終了条件の評価を行うような繰返しを，**後判定繰返し処理**といいます。後判定繰返し処理では，

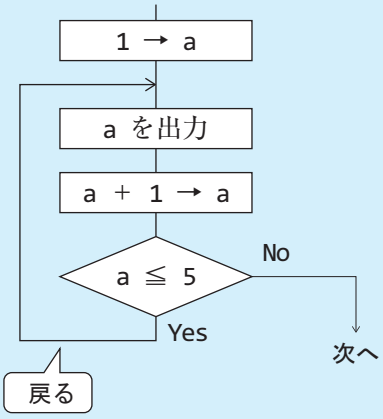
処理 → 条件の評価 → 処理 → 条件の評価 → … → 条件の評価 → ループ終了

という流れとなります。

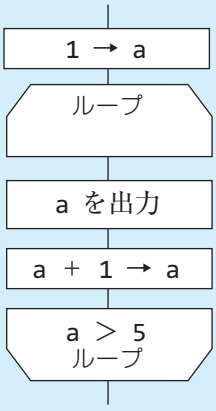
擬似言語プログラムでは，**do ～ while**文を用いて次のように記述します。do ～ while文でも，while文と同様に継続条件が成立している(真である)間，処理を繰り返します。

```
do
  継続条件が成立している間繰り返す処理
while (継続条件)
```

〔流れ図1〕



〔流れ図2〕



〔擬似言語〕

```
a ← 1
do
  a を出力
  a ← a + 1
while (a が 5 以下)
```

繰り返す

後判定

理解を深めよう

前判定と後判定

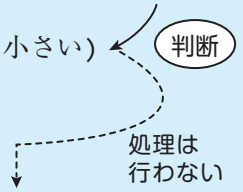
前判定繰返し処理と後判定繰返し処理の違いは，ループに入る前にすでに継続条件が偽となっているような場合に現れます。前判定型の繰返しでは，次の図の右のプログラムのように，ループに入る前に継続条件が偽となっている場合，ループ内部の処理

Cnt ← Cnt + 1

は**1回も実行されません**。そのため，ループ終了時のCntは10のままです。

```
○Check-Before()
  整数型: Cnt
  Cnt ← 0
  while (Cnt が 10 より小さい)
    Cnt ← Cnt + 1
  endwhile
```

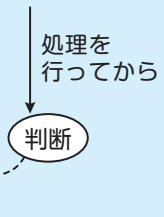
```
○Check-Before()
  整数型: Cnt
  Cnt ← 10
  while (Cnt が 10 より小さい)
    Cnt ← Cnt + 1
  endwhile
```



ところが，後判定型の繰返しでは，仮にループに入る前に継続条件が偽となっている場合でも，次の図の右のプログラムのように，**少なくとも1回はループ内の処理を行う**ことになります。そのため，ループ終了時のCntは11に更新されています。

```
○Check-After()
  整数型: Cnt
  Cnt ← 0
  do
    Cnt ← Cnt + 1
  while (Cnt が 10 より小さい)
```

```
○Check-After()
  整数型: Cnt
  Cnt ← 10
  do
    Cnt ← Cnt + 1
  while (Cnt が 10 より小さい)
```



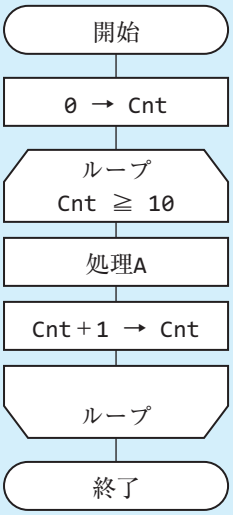
両者の違いをまとめておきましょう。

> 前判定繰返し処理：ループ内の処理を「1回も行わないことがある」
> 後判定繰返し処理：ループ内の処理を「少なくとも1回は行う」

継続条件とループカウンタ

同じ処理を一定回数繰り返す場合は，繰返しの回数を数える必要があります。この“何回繰り返したか”を数えるための変数を“**ループカウンタ**”とよびます。次の図は処理Aを10回繰り返す処理を，擬似言語と流れ図でそれぞれ表したものです。

```
○Loop10Times()
  整数型: Cnt
  Cnt ← 0
  while (Cnt が 10 より小さい)
    処理A
    Cnt ← Cnt + 1
  endwhile
```



これらは，Cntという変数をループカウンタとして用いています。ループカウンタはループ内で処理Aを1回実行した直後に1加算されています。ところで，ループに入る前にループカウンタCntに0を代入しています。これは，変数の内容が最初は未定義（不定）であるからです。このように，変数（ここではループカウンタ）にアルゴリズムの目的に合致するようにあらかじめ値を設定することを“**初期化**”とよびます。

さて，0に初期化されたCntは，ループ（繰返し）処理の1回目で，1となります。2回目では2となり，3回目では3となります。このように考えていくと，Cntの内容が次のように変化していくことがわかります。

ループカウンタCntの内容の変化

	Cntの値	処理Aの実行回数
1回目の条件判定 （ループ開始直前）	0	0回
2回目の条件判定 （ループ1回目終了直後）	1	1回
3回目の条件判定	2	2回
4回目の条件判定	3	3回
5回目の条件判定	4	4回
6回目の条件判定	5	5回
7回目の条件判定	6	6回
8回目の条件判定	7	7回
9回目の条件判定	8	8回
10回目の条件判定	9	9回
11回目の条件判定 （ループ10回目終了直後）	10	10回

ここで，10回目の条件判定を考えてみましょう。その直前，ループの9回目で，ループカウンタCntの内容は8から9になります。そして，処理Aは9回実行し終えたことになります。あと1回実行する必要があります。ですから，継続条件が“Cnt ＜ 10”（終了条件は“Cnt ≧ 10”）となっているのです。Cntの内容が9では，まだ継続条件は真ですから，もう一度，ループ内の処理Aを行います(ループ10回目)。そして，ループ10回目を終えて始端に戻ってきたとき，Cntは10になっていますから，継続条件が偽（終了条件は真）となり，endwhile（終端）の次へ処理が移るわけです。

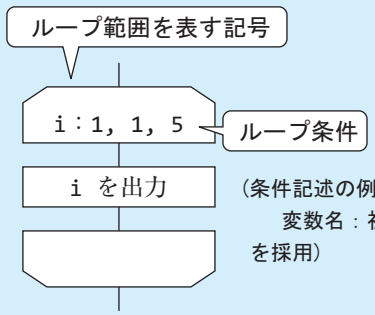
for 文による繰返し処理

擬似言語プログラムでは，C言語やJavaで用いる**for**文に似たループも記述できます。

```
for（制御記述）
  繰り返す処理
endfor
```

例えば次のような制御記述がよく用いられ，流れ図でも似たような記述ができます。

〔流れ図〕



（条件記述の例として
変数名：初期値，増分，終値
を採用）

〔擬似言語〕

```
for（i を 1 から 5 まで 1 ずつ増やす）
  i を出力
endfor
```

forでは多様な
条件指定が可能

左の流れ図では，ループの開始時点で1回だけループカウント用変数iに初期値として1が格納され，繰返しごとに，iには増分1が加えられます。このループは，i＝5(終値)の処理を実行するまで繰り返されます。右の擬似言語プログラムのfor文では，これを

i を 1 から 5 まで 1 ずつ増やす

というように，言葉で表しています。したがって，この例にあるループでは，

① iを1で初期化
② i＝1 を出力
③ iを1増やして，2とする
② i＝2 を出力
③ iを1増やして，3とする
② i＝3 を出力
③ iを1増やして，4とする
② i＝4 を出力
③ iを1増やして，5とする
② i＝5 を出力
④ ループを抜ける

というように②と③部分が繰り返され，「1，2，3，4，5」の順に値が出力されます。

1-6　繰返しを用いた簡単な処理

繰返し処理は，同じ処理を繰り返す場合に便利です。ここで基礎をマスターしましょう。

基本知識の整理

1から100までの整数の合計を求める

1から100までの整数の合計を求める場合，どのように行えばよいでしょうか。アルゴリズムをはじめて学習する方が思いつくのは，次のようなプログラムでしょうか。このプログラムでは変数Totalに合計を求めています。

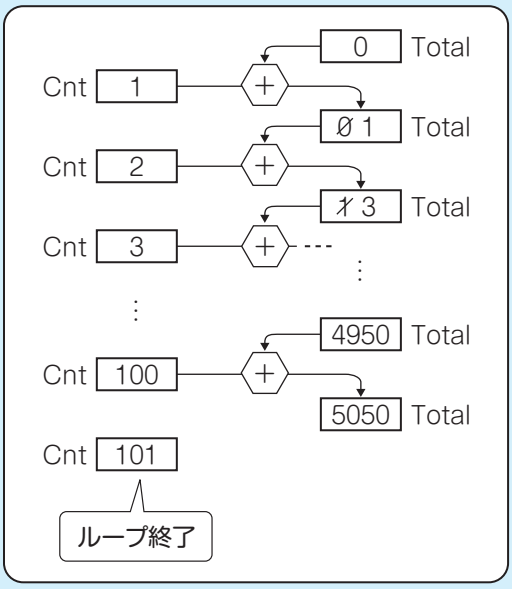
```
○Gokei()
  整数型: Total
  Total ← 0                    /* 初期化処理 */
  Total ← 1 + Total
  Total ← 2 + Total
        ⋮
  Total ← 99 + Total
  Total ← 100 + Total
```

1から100までの整数の合計を求めるプログラム(その1)

しかし，このプログラムでは同じような処理を100個も並べて記述しなければなりません。また，後になって「1から100までではなく，1から1000までの合計を求めたい」という要求が出たときに，その変更は容易ではありません。

繰返しを用いれば，次のように「スマートに」アルゴリズムを記述することができます。

```
○Gokei()
  整数型: Cnt, Total
  Cnt ← 1
  Total ← 0
  while (Cnt が 100 以下)
    Total ← Total + Cnt
    Cnt ← Cnt + 1
  endwhile
```



1から100までの整数の合計を求めるプログラム(その2)

このようにすることで，プログラムを簡潔に示すことができます。

また，ループの繰返しの継続条件を変更するだけで，1からnまでの整数の合計を求めるプログラムに変えることができるのです。たとえば，1から1000までの整数の合計を求めるように変更するには，ループの継続条件を

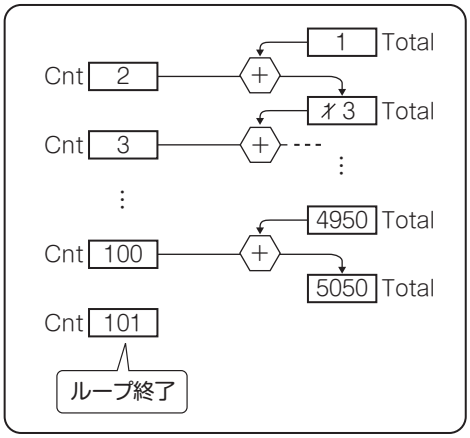
Cnt が 1000 以下

とすればよいわけです。

参考：初期値と繰返し回数

前出のプログラムでは，変数Totalの初期値を0とし，ループを100回繰り返して，1から100までの整数の合計を求めています。しかし，次のようにTotalの初期値を1，ループカウンタCntの初期値を2とすることで，ループの繰返し回数を1回減らすことができます。

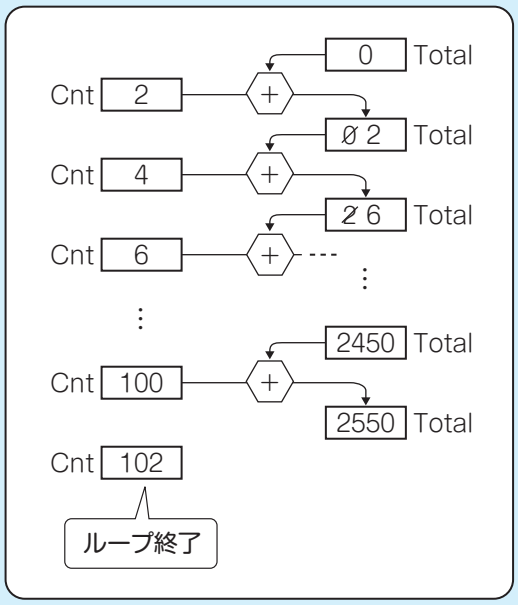
```
○Gokei()
  整数型: Cnt, Total
  Cnt ← 2
  Total ← 1
  while (Cnt が 100 以下)
    Total ← Total + Cnt
    Cnt ← Cnt + 1
  endwhile
```



1から100のうちの偶数の合計を求める

少し変更して,「1から100のうちの偶数の合計を求めるプログラム」を考えてみましょう。ループカウンタCntは，1から一つずつ増やしていましたが，これを「2から2ずつ増やす」ように変更すればよいわけです。プログラムは次のようになります。

```
○EvenGokei()
  整数型: Cnt, Total
  Cnt ← 2
  Total ← 0
  while (Cnt が 100 以下)
    Total ← Total + Cnt
    Cnt ← Cnt + 2
  endwhile
```



参考：初期化が必要な場合

「変数の内容は，最初は**未定義(不定)**なので，**初期化**が必要」と記述してきましたが，常に初期化が必要というわけではありません。たとえば，3×2の値を変数Aに格納する処理であれば，

A ← 3×2

とすればよく，この処理の前にAを初期化しておく必要はありません。では，どのような場合に初期化が必要になるのでしょうか。それは，ループ内で**足し込み処理**を行う場合などです。足し込み処理とは，

Seki ← Seki + A

や

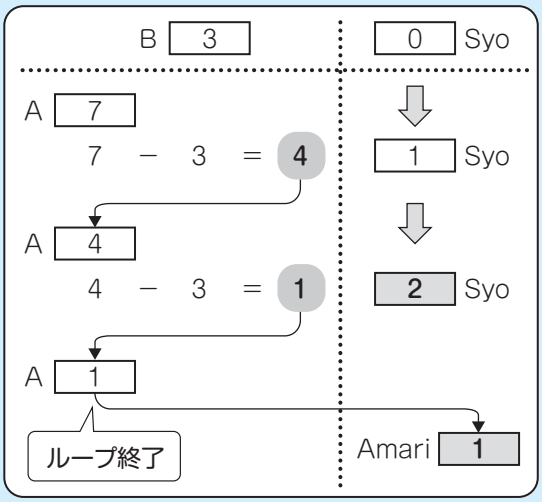
Cnt ← Cnt + 1

のように，ある変数に値を加えたり引いたりしたのち，その変数自身に結果を格納しなおすような処理をいいます。このような処理では，最初のSekiやCntの値が未定義であると，最後の結果が正しく求められないからです。ですから，このような場合は，ループに入る前に初期値を設定する初期化処理が必要になるのです。

割り算の答えを引き算の繰返しで求める

割り算の答えを求めるには，もちろん割り算を行えばよいのですが，割り算を使わなくても引き算の繰返しで答えを得ることができます。たとえば，7÷3の答えは2あまり1ですが，これは7から3を引いていき，引ききれなくなったときの「引くことができた回数」と「残りの数」にほかなりません。ここで，7が変数Aに，3が変数Bに格納されているものとして，次の図を見てください。

```
○Divide()
  整数型：A, B, Syo, Amari
  Syo ← 0
  while (A が B 以上)
    A ← A − B
    Syo ← Syo + 1
  endwhile
  Amari ← A
```



ループの1回目では，AからBを引いてそれを再びAに格納しています。そのとき，Syoの値を1増やしています。これは，“1回引けた”ということを意味します。そして，ループの先頭に戻り，継続条件をチェックします。継続条件の

A が B 以上

は，これを満たすときは，まだAからBを引くことができる(ここでは負の数は考えません)ことを意味します。ループの2回目ではAは4ですから，繰返し処理を続行します。今度は，Aの値は1になり，Syoは2となります。ループの3回目を行おうとしたとき，継続条件を満たしませんから，ループ内の処理は行わず，

Amari ← A

として，Aに残っていた1を変数Amariに格納して処理を終了します。このようにして割り算を使わずに，A÷Bの結果である“2あまり1”を変数Syoと変数Amariに求めることができたわけです。ほかの値でも正しく処理されることを確かめてみてください。

1-7　引数と返却値

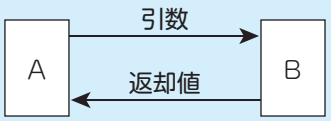
ここでは「プログラム同士のやり取り」に焦点をあてます。

基本知識の整理

引数と返却値

プログラムは，別のプログラムを呼び出すことができます。つまり，処理の一部を他のプログラムに肩代わりさせるのです。このとき，呼出し側のプログラムは，呼び出すプログラムに**引数**を与えることができます。また，呼び出されたプログラムは，処理結果を**返却値**（**戻り値**）で呼出し側に返すことができます。

（AがBを呼び出す場合）



引数：処理に必要なデータ
返却値：処理結果

次は，二つの整数の比較を何回か行うプログラムです。比較自体は副プログラムCmpが行い，大きな方を返却します。プログラムMainはCmpを呼び出して，返却された値を表示します。

```
○Main()
  整数型: Ret
  Ret ← Cmp(5, 3)
  Ret を表示
  Ret ← Cmp(2, 6)
  Ret を表示
```

```
○整数型: Cmp(整数型: P, 整数型: Q)
  if (P が Q より大きい)
    return P
  else
    return Q
  endif
```

プログラムMainは最初に5，次に6を表示します。詳しく見てみましょう。

Mainは，5と3を引数としてCmpを呼び出します。Cmpはそれを P，Qという変数（仮引数）で受け取ります。CmpはP(＝5)，Q(＝3)を比較し，**return**文で大きな方の値を返却します。

return分を実行するとCmpの処理は終了し，制御は呼出し元のMainに戻ります。

Mainは返却値(＝5)を，変数Retで受け取り，その値を表示します。

次に2と6を引数としてCmpを呼び出して，同様の処理を行います。

プログラムCmpの宣言に注目してください。カッコ内では，引数の型と変数名を宣言します。プログラムCmpは，二つの整数型の引数を変数P，Qで受け取ります。また，Cmp自体も整数型で宣言されています。これは，Cmpが整数値を返却することを表しているのです。

Example

変数A～Cに格納された値の最大値を表示します。A～Cは引数で受け取ることとします。

```
○DispMax(整数型: A, 整数型: B, 整数型: C)
  整数型: Max
  if (A が B より大きい)
    Max ← A
  else
    Max ← B
  endif
  if (C が Max より大きい)
    Max ← C
  endif
  Maxを表示
```

AとBを比較して
大きいほうをMaxに代入

MaxとCを比較して
Cのほうが大きかったらMaxに代入

Example

引数で受け取った整数値の「桁数」を求めて返却します。たとえば，5983を受け取った場合には，その桁数である4を返します。

```
○整数型: GetDigit(整数型: Val)
  整数型: Digit, Limit
  if (Val が 0 より小さい)
    Val ← Val × (−1)
  endif
  Digit ← 1
  Limit ← 10
  while (Limit が Val 以下)
    Digit ← Digit + 1
    Limit ← Limit × 10
  endwhile
  return Digit
```

絶対値をとる

条件評価	Limit	Val	Digit
1回目	10	5983	1
2回目	100	5983	2
3回目	1000	5983	3
4回目	10000	5983	4

➡ 処理終了

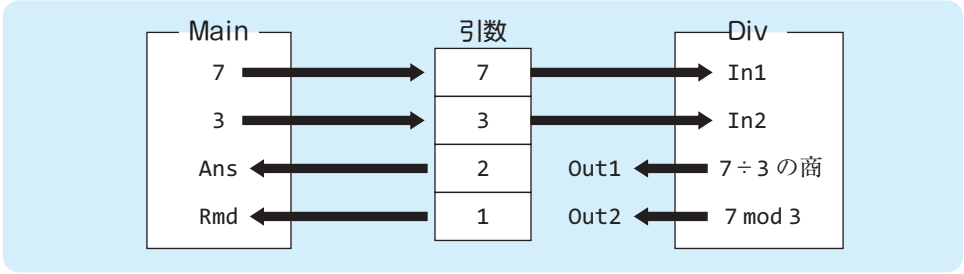
理解を深めよう

引数を用いた入出力

返却値は一つの値しか戻せません。二つ以上の値を戻す場合には，出力用の引数を用います。次のプログラムは，二つの整数の除算を行い，結果と剰余を表示します。除算そのものは副プログラムDivが行います。プログラムMainは，Divを呼び出すさいに，除算を行う2数と，結果の格納に用いる変数を引数で指定します。各引数が入力用か出力用かは，プログラムの仕様として別途定義しておきます。

```
○Main()
  整数型： Ans，Rmd
  Div(7，3，Ans，Rmd)
  Ans，Rmdを表示
```

```
○Div(整数型： In1，整数型： In2，
        整数型： Out1，整数型： Out2)
  Out1 ← In1 ÷ In2 の商
  Out2 ← In1 mod In2
```



大域変数

複数のプログラムが参照できる変数を，**大域変数**と呼びます。大域変数は引数で渡すことなく参照，更新することができます。大域変数は，非常に使い勝手のよい変数ですが，複数のプログラムが更新できるため，思わぬエラーの引き金となりかねず，原因の究明も簡単ではありません。

大域変数の宣言は，プログラムの宣言より前に行います。

次のプログラムは，大域変数Cntを更新します。Prog1だけを見た限りは，Cntは100に更新されるように思えますが，表示結果は0です。なぜならば，Prog2でもCntを更新しているからです。

```
大域：整数型： Cnt
○Prog1()
  Cnt ← 0
  Prog2(100)
  Cnt ← Cnt + 100
  Cntを表示
```

```
○Prog2(整数型： D)
  Cnt ← Cnt − D
```

1-8 配列と繰返し処理

配列と繰返し処理は，アルゴリズムを勉強していく上で非常に重要なテーマです。なぜ配列が必要なのか，配列のメリットは何かについて考えていきましょう。

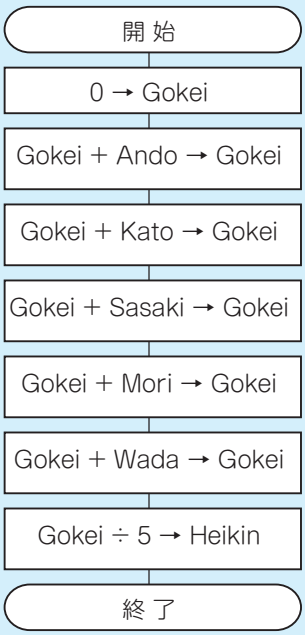
基本知識の整理

変数の弱点と配列

変数には一つの値しか格納できません。ですから，複数のデータを格納する場合にはデータの数だけ変数を用意する必要があります。たとえば，学生のテストの成績の平均点を求める場合を考えてみましょう。安藤さん，加藤さん，佐々木さん，森さん，和田さんという５人の学生がいたならば，区別がつくように，

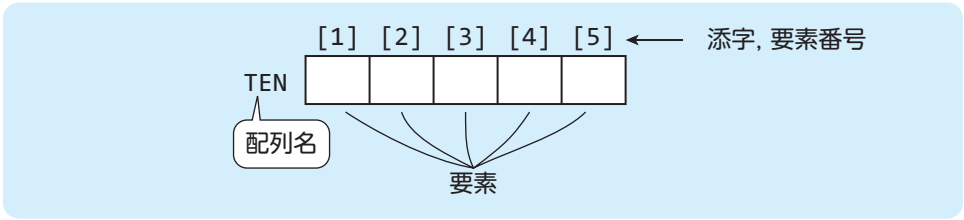
Ando，Kato，Sasaki，Mori，Wada

というような変数に得点を格納しておきます。さらに，合計点を格納するGokeiと平均点を格納するHeikinという変数を用意して，次のような流れ図を書くことになります。



この場合は5人ですからまだよいのですが，人数が増えるとそれだけ変数の数は多くなり，またGokeiへの足し込みを繰り返しているのにループを使うこともできず，流れ図も長くなってしまいます。そこで，用意されるものが**配列**です。配列は「同じ変数名をもつ同じ型の変数の集まり」です。上の得点のように同じ種類のデータを格納するような場合に用いることができます。

「同じ変数名」と聞いて「どうやって区別するの？」と困惑された方もおられると思います。配列ではそれぞれに「**添字**（そえじ）」，または「**要素番号**」とよばれる番号をつけて区別しています。たとえば，5人の得点を格納するためには，下の図のようなイメージで配列を作成します。



配列のイメージ

図の配列は，配列名がTEN，要素数が5となります。**要素**とは，配列を構成する個々の変数のことです。それぞれの要素は添字によって区別され，

TEN[1]，TEN[2]，TEN[3]，TEN[4]，TEN[5]

のように表現されます。個々の要素の性質は，変数の性質とまったく同じと考えてかまいません。

また，添字は上のように数値定数で表してもよいですし，変数を用いることもできます。たとえば，添字として変数iを用いることにすれば，

TEN[i]

と表現することもできます。この場合，添字である変数iの値が3であれば，

TEN[3]

を，変数iの値が1であれば，

TEN[1]

を意味することになります。

さらに，配列の添字に計算式を指定することができます。たとえば，変数iを用いて

TEN[i×2＋1]

と表現することもでき，この場合の変数iの値が3であれば，

TEN[7]

を指すことになります。

なお，流れ図では，配列の各要素を次のように表すこともあります。

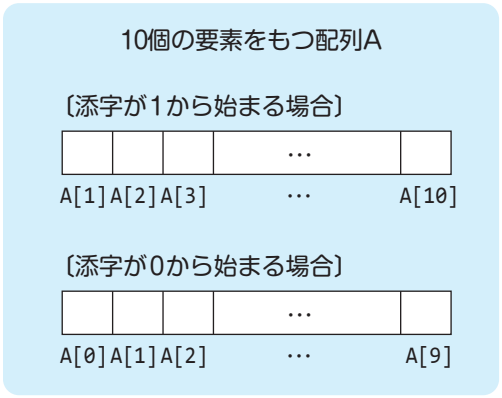
TEN(1)，TEN(2)，TEN(3)，TEN(4)，TEN(5)

ただし，これは括弧の種類が異なるだけで，扱い方は同じです。

配列の添字

配列の添字は0から始まる場合と，1から始まる場合があります。これは，プログラムが前提とする言語や，プログラムの性質によって変わります。

たとえば，次の図のように10個の要素をもつ配列の場合，添字が1から始まる場合はA[1]～A[10]の要素となり，添字が0から始まる場合はA[0]～A[9]の要素となります。したがって，配列の要素数がN個の場合には，次のような要素をもちます。



N個の要素をもつ配列Aでは
添字が1から始まる場合：A[1]～A[N]
添字が0から始まる場合：A[0]～A[N－1]

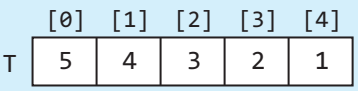
以降の説明では，特に断りを入れない限り，**添字は0から始まるもの**とします。

配列の表し方

配列の内容の始まりを表す“{”と，配列の内容の終わりを表す“}”を用いて，配列を表すことができます。例えば，配列Tを，

```
{5, 4, 3, 2, 1}
```

と表せば，次のような要素をもった配列になります。



配列の宣言

配列は，“型：配列名”や“型：配列名［要素数］”などの方法で宣言します。また，宣言と同時に，配列の内容を代入することもできます。

```
整数型の配列：Table      /* 整数を扱う配列Aを宣言 */

整数型の配列：T[10]      /* 整数を扱う10要素の配列Aを宣言 */
                         /* 使用できる要素はA[0]～A[9] */

整数型の配列：Table ← {2, 3, 5, 7, 11}    // 配列の内容を代入
```

理解を深めよう

配列と繰返し

配列の操作は，繰返し構造を用いるとシンプルに記述できます。たとえば，A［0］～A［9］を0で初期化してみましょう。まず，左側の（繰返しを用いない）プログラムに注目してください。各要素に0を代入する処理は，配列Aの添字以外はすべて同じ記述になっています。しかも，その添字の値は1ずつ増加しています。ですから，これを繰返し処理とすれば，右側の（繰返しを用いる）プログラムのように，簡潔にプログラムを記述できるのです。その際，添字をループカウンタとして扱えばよいのです。

```
(繰返しを用いない)
○Init()
  整数型の配列：A[10]
  A[0] ← 0
  A[1] ← 0
  A[2] ← 0
       ⋮
  A[9] ← 0
```

```
(繰返しを用いる)
○Init()
  整数型の配列：A[10]
  整数型：Idx
  for (Idx を 0 から 9 まで 1 ずつ増やす)
    A[Idx] ← 0
  endfor
```

A[Idx] ← 0 をidxが0～9で繰り返す
A[0] ← 0，A[1] ← 0，…，A[9] ← 0

Example

TEN[1]～TEN[N]の各要素に，N人の学生のテストの得点が格納されており，その平均点を変数Heikinに求めます。なお，整数に対する演算子“÷”は，実数として計算します。

```
○Average(整数型の配列: TEN, 整数型: N)
  整数型: Idx, Gokei
  実数型: Heikin
  Gokei ← 0
  Idx ← 1
  while (Idx が N 以下)
    Gokei ← Gokei + TEN[Idx]
    Idx ← Idx + 1
  endwhile
  Heikin ← Gokei ÷ N
```

Idx	処 理
1	Gokei ← Gokei＋TEN[1]
2	Gokei ← Gokei＋TEN[2]
⋮	⋮
N	Gokei ← Gokei＋TEN[N]
N＋1	終了

Example

A[0]～A[99]の各要素に，整数1～100を順番に代入します。すなわち，

A[0]←1，A[1]←2，…，A[99]←100

を順次行います。

```
○Init()
  整数型の配列: A[100]
  整数型: Idx
  for (Idx を 0 から 99 まで 1 ずつ増やす)
    A[Idx] ← Idx + 1
  endfor
```

Idx	処 理
0	A[0] ← 0＋1
1	A[1] ← 1＋1
⋮	⋮
99	A[99] ← 99＋1
100	終了

Example

配列From[0] ～ From[N－1]に格納された値を，配列Toにコピーします。すなわち，

To[0]←From[0]，To[1]←From[1]，…，To[N－1]←From[N－1]

を行います。配列Fromは引数で受け取ります。

```
○Copy(整数型の配列: From, 整数型: N)
  整数型の配列: To
  整数型: Idx ← 0
  while (Idx が N より小さい)
    To[Idx] ← From[Idx]
    idx ← Idx + 1
  endwhile
```

Idx	処　理
0	To[0] ← From[0]
1	To[1] ← From[1]
⋮	⋮
N−1	To[N−1] ← From[N−1]
N	終了