Tiny Basic for Windows ファイル操作編

tbasic.org [*1]

[2023 年 04 月版]

Tiny Basic for Windows 入門編，初級編では，プログラムを書く上で，基本的な処理について説明しました。ここでは大掛かりな処理を行う際に必要となる，ファイル処理について説明します。第 I 部では基礎知識，第II部では具体的な操作法について説明をします。

目次

第 I 部　基礎知識編 3

1 ファイル 3

2 テキストファイルとバイナリファイル 4
- 2.1 テキストファイル . . . 4
- 2.2 バイナリファイル . . . 5

3 符号化文字集合 7
- 3.1 アスキー (ASCII) コード . . . 7
- 3.2 拡張アスキーコード . . . 8
- 3.3 日本語符号化文字集合 . . . 9
- 3.4 国際符号化文字集合：ユニコード . . . 11

4 文字符号化方式：エンコーディング 12
- 4.1 アスキーエンコーディング . . . 12
- 4.2 日本語文字集合用符号化方式 . . . 12
- 4.3 ユニコード用符号化方式 . . . 13
- 4.4 tbasic で利用できるエンコーディング . . . 13

5 ファイル名と拡張子 15
- 5.1 ファイル名 . . . 15
- 5.2 拡張子 . . . 15
- 5.3 アイコンと拡張子 . . . 16

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

第 II 部　ファイル操作 18

6 tbasic で扱えるファイル 18
6.1 テキストファイル 18
6.2 バイナリファイル 19

7 ファイル処理の概要 20
7.1 ファイル読み書きの原理 20
7.2 ファイルの場所の特定 21

8 テキストファイル処理 25
8.1 WriteAllLines と ReadAllLines 25
8.2 WriteAllText と ReadAllText 28
8.3 まとめ 29

9 テキストファイル処理プログラム例 30
9.1 行番号の削除・追加 30
9.2 成績処理プログラム 33
9.3 演習問題 44

10 バイナリファイルの取り扱い 45
10.1 バイナリファイルの作成・書き込み 45
10.2 バイナリファイルの読み込み 50

第 III 部　付録 54

11 従来型ファイル処理 54
11.1 ファイルの Open 54
11.2 ファイルへの書き込み：Output 55
11.3 ファイルの読み込み：Input 57
11.4 ファイルへの追加書き込み：Append 61
11.5 ファイルの Close 62

第 I 部
基礎知識編

BASIC で多くのデータを扱う方法として，初級編で説明した DATA 文による方法がありました。DATA 文はプログラムの中にデータを独立して記述することで，機能を分離し，見やすさと，保守の容易さを実現しています。このことから，プログラム内容に関係の深い，比較的少ないデータを扱う場合は DATA 文は効率的な良い方法です。しかし，大きなデータを扱う観点からすると，DATA 文には，いくつかの弱点があります。

■ Data 文の弱点

- データとプログラムが一体化しているため，1 つのプログラムで 1 つのデータセットしか扱えない。
- 大量のデータを扱いにくい。
- BASIC プログラム内部に含まれ，他のプログラミング言語で作成したデータとの互換性が弱い。

これらの弱点は，外部のデータファイル扱えれば克服することができます。このファイル操作編では，外部データファイルを tbasic で扱う際に必要な，基礎知識とその方法について説明します。

1　ファイル

ここではコンピューターを操作する上で，重要な概念の一つであるファイルについて説明します。

コンピューターを使って色々な処理を行う場合，その処理している，或いは処理した結果のデータはコンピューターの主メモリ上にあります。コンピューターの主メモリは高速な動作で読み書きできますが，普通，電源を切るとその中身はすべて消えてしまいます。

しかし，そのデータを保存して，再利用したい場合も多くあります。そのような場合，電源を切ってもデータが消えない，ハードディスク等の記憶媒体に保存します。保存する場合，現在のコンピューターのメモリ状態をそのまま保存すると，後からそのデータを利用しようとしたとき，必ずしも使いやすいものではありません。

そこで，後からの利用のしやすさと，保存する媒体の効率的利用のため，必要とするデータをひとまとまりにして，名前を付けて保存します。このようなデータのまとまりがファイルです。つまり，

ファイル（File）

読み・書き処理の対象となるデータのひとまとまり。普通，ディスク等に保存されたもの。

となります。ファイルの物理的な具体的保存形式は，保存する媒体，保存に利用する OS により定まるファイルシステムによって決まりますが，OS を通してファイルにアクセスすることで，ユーザーはファイルが物理的にどのように保存されるか知る必要はありません[*2]。

[*2] 実際にデータを保存する物理的媒体は，USB メモリ，ハードディスク，ネットワークドライブなど種々なものがありますが，ユーザーは OS を媒介にすることでその物理的保存形式を意識しないで，どの媒体に対しても同じ操作でファイルを処理することができます。

コンピューターが処理するデータの最小単位はビット (0, または 1) ですから，OS を介して利用するユーザーから見ると，ファイルはファイル名によって扱うことのできる 0,1 データのひとまとまりと言えます。0,1 データのことをバイナリデータと言いますが，その意味で，コンピューターで扱うファイルは，すべてバイナリファイルと言うことができます。

2　テキストファイルとバイナリファイル

2.1　テキストファイル

コンピューターで扱うファイルは，すべてバイナリファイルであると上で述べました。その中で特別に重要なファイル形式: テキストファイルがあります。テキストファイルは人間が読む文書として作られたファイルの意味ですが，文字列と改行から構成されるファイルです。この文字列は適当な解釈で文字として認識できるものの並びです。この解釈法のことを文字エンコーディング（Character Encoding）と言います。この解釈法によって漢字や種々の国の文字列をテキストファイルとして表すことができます。このことから，テキストファイルのより具体的な定義として，「適当なエンコーディングによって，文書を表すファイルと解釈できるものをテキストファイル」ということもできます。

テキストファイルとそれ以外のファイルを区別するために，テキストファイル以外のファイルをバイナリファイルと言います。このようにバイナリファイルの名称は２種類の意味がありますが，普通は，後者，即ち，テキストファイル以外のファイルを示します。

まとめると，

・テキストファイル：
　人間が読む文書として作られたファイル。
　文字列と改行から構成される。
・漢字や種々の国の文字を含むものもテキストファイルとして扱える。
・テキストファイル以外のファイルがバイナリファイル。
・テキストファイルには対応するエンコーディングがある。

となります。

テキストファイルはメモ帳のようなエディターで読むことのできるファイルです。またエディターで扱うファイルは基本的にテキストファイルです。ですから次のように言うこともできます。

エディターで扱うファイルがテキストファイル。

これに対して，ワープロ固有のファイルはテキストファイルではないことに注意して下さい。

テキストファイルは文字列と改行コードで構成されます。テキストファイルでは，改行コードは特別な意味を持ちます。改行コードが次のように OS に依存しているからです。

改行コード
MS-DOS，Windows：CR+LF（アスキーコード 13 と 10）
Mac：CR（アスキーコード 13）
Unix：LF（アスキーコード 10）

例えば，Windows では，文字列の途中に，CR+LF コードを加えると，その部分で改行されます。
次は，tbasic での具体的な例です。

例 2.1.

```
A$="abc"+chr$(13)+Chr$(10)+"def"
```

に対して，Print A$ とすると，abc の後が改行され，実行画面上に，

```
abc
def
```

と表示されます。

このようにテキストファイルの形式は OS に依存したものですが，依存部分は改行コードだけです。従って，この部分の変換を行えば，別の OS でも読むことができます。つまり

テキストファイルは OS に依存しない共通形式をもつ
（改行は対応したものに変換する）

ということができます[*3]。しかし，現在のエディターは改行コードを適宜，変換解釈する機能を備えているものが多いようです。ですから，上の例のように，改行コードを直接操作すること以外，改行コードの違いにを，余り意識する必要は無いでしょう。

2.2 バイナリファイル

テキストファイル以外のファイルをバイナリファイルといいます。バイナリファイルはコンピューターに解釈させるためのファイルです。テキストファイルは 1 つの形式を持っていますが，これに対して，バイナリファイルは特に制限はありません。コンピューターが理解できる適当な形式が定まっていて，その形式に従ってファイルの中身をコンピューターが解釈できればよい訳です。ですから，その種類は色々あります。つまり，

バイナリファイルの形式は色々ある

ということです。例をあげると

[*3] FTP ソフトでアスキー転送という用語がありますが，これはこの改行変換を行う転送であることを意味します。

バイナリファイルの例

・ワープロ文書：
人間が読む文書として作られたファイルでも，
ワープロ文書等のファイルはテキストファイルでありません。
それらはワープロシステムを通して読むための文書だからです。

・画像・音声ファイル：
静止画や動画，音声用のファイルは全てバイナリファイルです。

・実行形式プログラム：
独立に実行できる形式をもったファイルはバイナリファイルです。

などがあります。

バイナリファイルでも OS に依存するファイルと依存しないファイルがあります。

画像や音声等のファイルは OS に依存しないものが多数あります。例えば jpeg や gif ファイルは OS に依存しません。このようなファイルは windows や mac，Unix などで共通に使うことができます。

他方，実行ファイルは当然ですが，OS に依存します。例えば tbasic の本体である TBasic.exe は Windows 用のバイナリファイルです。ですから，TBasic.exe を Mac や Linux 上にファイルとしてコピーすることはできますが，tbasic を起動することはできません。

3　符号化文字集合

前項ではファイルについて，特にテキストファイルについて説明しました。そこでは，テキストファイルは漢字や種々の国の文字を含むものも扱えると述べました。テキストファイルは基本的にバイト文字列として構成され，適当な解釈の下で文字として扱うことができます。

その解釈法「**文字エンコーディング**」を定めるためには，まず，使用可能な文字と，その対応方法を決める必要があります。このように文字とその対応を決めたものを**符号化文字集合**（Coded Character Set）と言います。

ここでの説明が不足の場合は，別文書「Tiny Basic for Windows でのユニコードの取扱い」，「ユニコードへ」でも同じ話題を説明していますので，そちらも参照してください。

3.1　アスキー (ASCII) コード

最も基本的な符号化文字集合はアスキーコード表です。

コンピューターが開発され，コンピューター上で最初に共通的に利用された文字はアルファベットでした。この規格が ASCII（American Standard Code for Information Interchange）です。ASCII は英字の大文字，小文字といくつかの記号そして，制御コードと言われる，合計 128 個のコードが規定されています。全体は以下の通りです。

ASCII コード表

	0		1		2		3		4		5		6		7	
0	0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
a	10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
b	11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
c	12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
d	13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
e	14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
f	15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

ここで，32 SP は空白を表し，33 から 126 までが文字を表し，残りが制御コードを表します。

この表で，使用可能な文字（英数記号）とそれを示す番号[*4]が規定されています。最も基本的な符号化文字集合が定められていると言えます。

[*4] 例えば A は 65，X は 88 と定められています。

3.2 拡張アスキーコード

コンピューターの広範な利用のためには，勿論アスキーコードだけでは不十分です。ASCII は 128 個（7 ビット）のコードから構成されていますが，1 バイトは 8 ビットで，1 バイトでは 256 個の表現が可能です。そこで，ASCII で未使用の 128 から 255 までの部分を利用することが考えられました。そのコードが拡張アスキーコードです。

日本でもその空き領域の利用が考えられ，JIS X 0201（1969 年）で，コード番号 161 から 223 までに以下のようなカタカナなどが割り当てられました。初期のコンピューターではこれを利用して日本語を表現しました。

拡張 ASCII コード表（日本）拡張部分

	8		9		10		11		12		13		14		15	
0	128		144		160		176	ー	192	タ	208	ミ	224		240	
1	129		145		161	。	177	ア	193	チ	209	ム	225		241	
2	130		146		162	「	178	イ	194	ツ	210	メ	226		242	
3	131		147		163	」	179	ウ	195	テ	211	モ	227		243	
4	132		148		164	、	180	エ	196	ト	212	ヤ	228		244	
5	133		149		165	・	181	オ	197	ナ	213	ユ	229		245	
6	134		150		166	ヲ	182	カ	198	ニ	214	ヨ	230		246	
7	135		151		167	ァ	183	キ	199	ヌ	215	ラ	231		247	
8	136		152		168	ィ	184	ク	200	ネ	216	リ	232		248	
9	137		153		169	ゥ	185	ケ	201	ノ	217	ル	233		249	
a	138		154		170	ェ	186	コ	202	ハ	218	レ	234		250	
b	139		155		171	ォ	187	サ	203	ヒ	219	ロ	235		251	
c	140		156		172	ャ	188	シ	204	フ	220	ワ	236		252	
d	141		157		173	ュ	189	ス	205	ヘ	221	ン	237		253	
e	142		158		174	ョ	190	セ	206	ホ	222	゛	238		254	
f	143		159		175	ッ	191	ソ	207	マ	223	゜	239		255	

これに対して，北米、西ヨーロッパ、オーストラリア、アフリカ等で用いられる規格では 128 以降では異なる文字が割り当てられています。例えば，元々の IBM PC では，コードページ 437 と言う独自の拡張 ASCII を持っていました。そこでは 128 から 255 まで，次のように文字が割り当てられています。

拡張 ASCII コード表（IBM PC）拡張部分

	8		9		10		11		12		13		14		15	
0	128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
1	129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	β	241	±
2	130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
3	131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
4	132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
5	133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
6	134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	μ	246	÷
7	135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
8	136	ê	152	ÿ	168	?`	184	╕	200	╚	216	╪	232	Φ	248	°
9	137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
a	138	é	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
b	139	ï	155	¢	171	½	187	╗	203	╦	219	▉	235	δ	251	√
c	140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
d	141	í	157	¥	173	!`	189	╜	205	═	221	▌	237	φ	253	^{2}
e	142	Ä	158	P_{ts}	174	≪	190	╛	206	╬	222	▐	238	ε	254	■
f	143	Å	159	f	175	≫	191	┐	207	╧	223	▀	239	∩	255	

このように，同じアスキーコードでも 128 以降では，全く異なった文字を表します。この違いは OS によって決まるものですが，例えば，日本語 MSDOS では，ASCII コード 171 は「ｵ」を表しますが，元々の PCDOS では，ASCII コード 171 は「½」を表します。

3.3 日本語符号化文字集合

コンピューターで色々な処理を行うとき，ASCII で定義された文字では，拡張 ASCII を用いたとしても，明らかに不足です。日本語の場合では，漢字やひらがなを利用する必要があります。コンピューターでの処理の基本単位は 1 バイト（8 ビット）ですが，その場合 256 種類のものしか表現できません。そこで 1 バイトではなく 2 バイトなど複数バイトを用いて文字を表現する方法が工夫されました。2 バイトを使うと $256 \times 256 = 65536$ 種類の文字が表現できます。日本語で日常的に扱う文字・記号・漢字の種類はこのくらいあれば十分です。

■ JIS X 0213

日本での符号化文字集合の最初の規格は，1969 年に制定された拡張 ASCII に対応した JIS X 0201（初版）です。しかし，これはカタカナのみの対応で，漢字は含まれていませんでした。

漢字を含む文字集合の最初の規格は JIS X 0208 で，1978 年に制定されました。この規格はその後 1997 年まで何回か改正されました。

現在使われている JIS X 0213 規格は JIS X 0208 や以前の規格を引用規格とし，包括的な拡張版として 2000 年に初版が制定されました。その後何度か改正されて，現時点[*5]では 2012 年の改正が最新版です。現在

[*5] 2021 年 10 月最終確認

のコンピューターの OS では，文字集合としてこの JIS X 0213 がサポートされています[*6]。

JIS X 0213 は 2 面，各面 94 区 94 点で構成されて，全 17672 点の内 11233 字を規定しています[*7]。

例えば，1 面 1 区，1 面 50 区，2 面 94 区はそれぞれ次のように規定されています。

JIS X 0213 1 面 1 区

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0		<SP>	、	。	,	.	・	:	;	?	!	゛	゜	´	｀	¨
16	^	￣	_	ヽ	ヾ	ゝ	ゞ	〃	仝	々	〆	〇	ー	―	‐	／
32	＼	～	∥	｜	…	‥	‘	’	“	”	（	）	〔	〕	［	］
48	｛	｝	〈	〉	《	》	「	」	『	』	【	】	＋	−	±	×
54	÷	＝	≠	＜	＞	≦	≧	∞	∴	♂	♀	°	′	″	℃	￥
80	＄	¢	£	％	＃	＆	＊	＠	§	☆	★	○	●	◎	◇	

JIS X 0213 1 面 50 区

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0		瓣	劬	劭	劼	劵	勁	勍	勗	勞	勣	勦	飭	勠	勳	勵
16	勸	勹	匆	匈	甸	匍	匐	匏	匕	匸	匣	匯	匱	匳	匸	區
32	卆	卅	丗	卉	卍	凖	卞	卩	卮	夘	卻	卷	厂	厖	厠	厦
48	厥	厮	厰	ム	參	簒	雙	叟	曼	燮	叮	叨	叭	叺	吁	吽
54	呀	听	吭	吼	吮	吶	吩	吝	呎	咏	呵	咎	咳	呱	呷	咢
80	咒	呻	咀	呪	咄	咐	咆	哇	唪	咸	咥	咬	哄	哈	咨	

JIS X 0213 2 面 94 区

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
0		亹	傌	傭	傷	僂	僉	僣	僦	僬	僭	僵	僾	儁	儂	儃
16	儇	儋	儐	儔	儕	儘	儜	儡	儢	儣	儦	儩	儬	儭	儯	儱
32	儳	儴	儵	儺	儻	儼	儽	儾	兂	兇	兊	兏	兓	兕	兗	兙
48	兛	兝	兞	兟	兡	兦	兩	兪	兯	兲	兺	兾	冀	冁	冂	冉
54	冏	冑	冓	冕	冖	冗	冘	冢	冣	冤	冥	冦	冩	冪	冫	冱
80	冲	冴	冶	冷	凇	凊	凍	凐	凒							

このように日本語文書を扱う場合，JIS X 0213 の文字集合は完全とは言えませんが，かなり十分なもののなっています。

このような文字集合の考え方は世界中の多くの国々で採用され，それぞれの国々に合わせた文字集合が定められています。

[*6] JIS X 0213 は文字としては，JIS X 0208 を含んでいますが，字形が変更されている文字もあります。

[*7] ここで現れる 94 と言う数はアスキーコード表で，！から˜までの印刷可能文字数 94 個に由来します。初期のエンコーディングでは，この範囲に漢字等のコードを配置していました。

しかし，現在では，一つの文書の中に，いくつもの国々の言語が混在する場合もあります。JIS X 0213 の文字集合にも外国文字を表すものも多くありますが，それでも十分ではありません。この状況はそれぞれの国での文字集合を定めた場合，同じことが生じます。そこで，このことに対応するために，世界中で使われている文字を一つの体系で扱えるようにすることが考えられました。

3.4 国際符号化文字集合：ユニコード

1980 年代の後半から，世界中の文字を一つの体系で表す規格を作成するプロジェクトが始まり，1990 年ユニコードコンソーシアム（The Unicode Consortium）として組織化されました。1991 年 Unicode Standard 第 1 版が規定され，そこでは，7,161 文字がからなる文字集合が規定されました。その後現在まで改定・追加が続けられ，2023 年 1 月現在の最新版は第 15 版（Version 15.0）では 149,186 文字が規定されています。このユニコード文字集合には，欧米で使われているラテン文字はもちろん，日本語[*8]の他，ギリシア語，ロシア語，アラビア語，中国語，ハングルなどが含まれています。その他，数学などの多くの記号，絵文字やゲームの記号も含まれています[*9][*10]。

ユニコードの符号化文字集合は，16 進数で番号付けされ，4 桁の 16 進数領域をまとめて面と呼んでいます。全体は基本多言語面（Basic Multilingual Plane：BMP）とその他 16 個，合計 17 個の面で構成されています。

基本領域	基本多言語面	U+0000	～	U+FFFF
拡張領域	第 1 面	U+10000	～	U+1FFFF
	第 2 面	U+20000	～	U+2FFFF
	第 3 面	U+30000	～	U+3FFFF
	・・・			
	第 14 面	U+E0000	～	U+EFFFF
	第 15 面	U+F0000	～	U+FFFFF
	第 16 面	U+100000	～	U+10FFFF

各面とも文字はその番号に U+ を付け加えて表すことになっています。例えば，672C 番目の文字は，U+672C と表します。ユニコードに対応した文書であれば，これらの文字が使用可能となり，多くの国々の文字が混在した内容を記述することができます。

まとめると，

・符号化文字集合は，コンピューターで用いる文字のあつまりの規格
・日本語の現在の符号化文字集合は JIS X 0213
・外国語を含む文書を作成する際の符号化文字集合は，ユニコード文字集合

となります。

[*8] JIS X 0213 を含む
[*9] 例えばトランプの図柄なども含まれています。
[*10] これら膨大な文字集合の全貌は http://www.unicode.org/charts/ で見ることができます。

4　文字符号化方式：エンコーディング

符号化文字集合はコンピューターで使用する文字を規定するものですが，これらの文字をどのような方法でコンピューター内部で利用するかの方法は規定していません。

コンピューター内部で利用する仕組は文字符号化方式（文字エンコーディング：Character Encoding）と言われ，別に規定されています。[*11]

4.1　アスキーエンコーディング

現在のコンピューターで扱うファイルの基本単位はバイト（8 ビット）で，10 進数では，0 から 255 を表します。アスキーコード表は，0 から 127 までの数で構成されていますから，この数はそれ自身，文字を解釈する方法です。

ですからアスキーコードは，自然なエンコーディングを備えていると言えます。アスキーコードが規定されて以降，標準的なコンピューターは，OS レベルでこの解釈法をサポートしていました。ですから，アスキーコードで書かれた文書は世界中のコンピューターで普通に読むことができます。

これに対して，拡張アスキー表の拡張部分（127 から 255）状況は異なります。国によって解釈が異なることがあるからです。

日本では，JIS X 0201 が拡張アスキー表を規定していますが，日本語対応 OS では，この対応がサポートされていました。また，以下に説明する `Shift_JIS` や `EUC` は，アスキーコードと互換性を持つよう拡張されています。

4.2　日本語文字集合用符号化方式

現在日本で良く用いられている符号化方式は，日本語文字集合用（JIS X 0208，JIS X 0213）の符号化方式と，ユニコード文字集合用の符号化方式です。

(1) JIS
JIS X 0208 に規定される規格が ISO-2022-JP と言われるものです。現在の最新規格は JIS X 0213 に規定され，ISO-2022-JP-2004 と言われています。４バイト等のモード切替コードと，１バイト，または２バイトを使って文字を表します。使用する各々のバイトは，１バイトのうち７ビットを使用する７ビット符号です。メールが７ビットアスキーから構成されることとの整合性を保つ規格です。

(2) Shift_JIS
JIS X 0208 対応の Shift_JIS は DOS パソコンでの日本語規格として広く利用されてきました。また現在でも Windows で広く利用されています[*12]。
JIS X 0213 対応では Shift_JIS-2004 として規定されています。１バイト，または２バイトを使って文字を表します。漢字の位置の指定が移動することからシフト JIS と名付けたと言われています。

[*11] コンピューターでの具体的実装は，バイトの並び方（BOM:Byte Order Mark）などにより，ここでの符号化方式の中で更に細かく分かれる場合もあります。

[*12] 現在，Windows 上で標準的に使用されている Shift_JIS は，JIS X 0208 対応です。そのため JIS X 0213 で新たに追加された文字は使えません。しかし，フォント自体は JIS X 0213 対応となっています。

(3) EUC
EUC は Extended Unix Code の名の通り，UNIX や Linux で広く用いられていた規格です[*13]。日本語用では，JIS X 0208 対応が EUC-JP と言われ，JIS X 0213 対応が EUC-JIS-2004 と言われています。３，４バイトのモード切替コードと，１バイト，または２バイトを使って文字を表します。

4.3 ユニコード用符号化方式

(1) UTF-8
UTF-8 は，ユニコード文字を，１バイト～４バイトを使って表現する方法です。１バイトで表される場合，アスキー文字と対応しているという優位な性質があります。近年では Linux 系の OS では，標準符号化方式として採用するものが増えています。

(2) UTF-16
UTF-16 はユニコード文字を２バイト（16 ビット）または４バイトを使って表す方法です。基本領域文字は２バイトで表すことが可能なことから，実際に使う文書では効率的な方法とされています。このことから広く利用されており，Windows OS では，内部的には UTF-16 を用いています。

(3) UTF-32
UTF-32 はユニコード文字をすべて４バイト（32 ビット）を使って表す方法です。すべて同じバイト数を使って表現することから，メモリーの使用量について無視できる環境であれば分かりやすいく，効率的な方法です。

これら各々の方法によって，文字をコンピューターで利用し，ファイルとして保存します。同じ内容を表すテキストファイルであっても，その符号化方式が異なれば，そのファイルの具体的な内容＝ビットの並びが異なり，また容量も異なることがあります。

4.4 tbasic で利用できるエンコーディング

現在 Windows 上で，実際に使われているエンコーディングは，`Shift-JIS` と `UTF-8` です[*14]。ですからこれら２種を利用すれば，コンピューター上でほとんどの処理が可能です。

しかし，場合によって他のエンコーディングを利用する必要があるかも知れません。`JIS` は `iso-2022-jp` という名称でメールファイルとして，現在でもよく使われています。また，`EUC` もあるかも知れません。これらを含めて，tbasic では，以下のエンコーディングをサポートしています。

エンコーディングは，上で説明した以上に少し，細分化されています。名称とエンコーディングの判別は多少難しい面もありますので，tbasic で使用するエンコーディング名は独自なものもあります。エンコーディングについてさらに詳しい説明が別文書「ユニコードへ」にありますので，必要な場合は，そちらを参照してください。

[*13] 現在では，以下に説明するユニコードが使われています。
[*14] Windows での内部エンコーディングは `UTF-16` です。

tbasic での利用できる具体的エンコーディング（Encoding）は次の通りです。

表 1

tbasic での Encoding 名	CodePage	名前	説明
SJIS	932	shift_jis	日本語 (Shift-JIS)
EUC	20932	EUC-JP	日本語 EUC
JIS	50220	iso-2022-jp	日本語 JIS
UTF-8	65001	utf-8	Unicode(BOM 無し，UTF-8)
UTF-8BOM	65001	utf-8	Unicode(BOM 付，UTF-8)
UTF-16LE	1200	utf-16	Unicode(BOM 無し，UTF-16 Little Endian)
UTF-16BE	1201	unicodeFFFE	Unicode(BOM 無し，UTF-16 Big Endian)
UTF-16LEBOM	1200	utf-16	Unicode(BOM 付，UTF-16 Little Endian)
UTF-16BEBOM	1201	unicodeFFFE	Unicode(BOM 付，UTF-16 Big Endian)
UTF-32LE	12000	utf-8-32	Unicode(BOM 無し，UTF-16 Little Endian)
UTF-32BE	12001	utf-32	Unicode(BOM 無し，UTF-16 Big Endian)
UTF-32LEBOM	12000	utf-8-32	Unicode(BOM 付，UTF-16 Little Endian)
UTF-32BEBOM	12001	utf-32	Unicode(BOM 付，UTF-16 Big Endian)

5　ファイル名と拡張子

前項では，コンピューターでの文字の解釈法について説明しました。この解釈法についての知識はテキストファイルを扱うとき必要になります。

ここでは，実際にファイルを処理する際に必要となるファイル名の付け方などについて説明します。

5.1　ファイル名

ファイルはデータのひとまとまりをディスクに保存したものでした。それを利用するためには，それを特定する名前をつけることが必要です。ファイル名の命名規則は比較的自由で，使えないいくつかの記号を避けて，分かりやすいように適切に命名すれば良いだけです。

> Windows でのファイル名の付け方
> ・ファイル名は，以下の使えない記号を避けて，分かりやすい名前を付ければよい。
> ・使えない記号：¥ / : * ? " < > は使えない。
> ・空白や漢字も使える。

MS-DOS でのファイル名は半角 8 文字[*15]でしたが，Windows ではかなり長い名前も使えます（最大半角 200 字程度）。しかし，余り長いものは避けましょう。またこの文字数制限はフォルダ名等も含んだパス名の文字数も含んだものですので，あまり深いフォルダを使うとトラブルのもとになります。

5.2　拡張子

ファイル名にはピリオド（或いはドット）「.」も使うことができます。このピリオドはいくつも使うことができますが，最後のピリオドの後の部分を拡張子と言います。この拡張子は普通ファイルの性質を表す名前として使われます。

> 拡張子
> ・ファイル名の最後のピリオドの後の部分を拡張子と言う。
> ・拡張子はファイルの性質を表す名前として使う。

例をあげましょう。

例 5.1.

- Test.txt
- Test.doc
- Test.dat

[*15] 半角という用語は，現在では避けるべきでかも知れません。正確には ASCII 文字です。

最初の Test.txt では txt が拡張子です。また，Test.doc では doc が拡張子になります。Test.dat では dat が拡張子です。

Windows の エクスプローラー[*16] の初期設定では，拡張子は表示されません。もし現在使用のコンピューターで拡張子が表示されていなかったら，エクスプローラー の「表示」で「ファイル名拡張子」の部分のチェックして[*17]見える設定にすることを強く勧めます。

■拡張子の意味の例

拡張子はファイルの性質を示す特別な意味を持っています。以下に Windows で用いられる，いくつかの代表的な例をあげます。

拡張子	種類	意味
txt	テキストファイル	テキストファイル
exe	バイナリファイル	実行形式プログラム
doc	バイナリファイル	Word 2003 文書
docx	バイナリファイル	Word 2007 以降文書
xls	バイナリファイル	Excel 2003 ファイル
xlsx	バイナリファイル	Excel 2007 以降ファイル
jpg	バイナリファイル	jpeg 画像ファイル
csv	テキストファイル	カンマ区切りデータファイル
tbt	テキストファイル	tbasic テキストファイル
html	テキストファイル	HTML 文書
htm	テキストファイル	HTML 文書
xml	テキストファイル	拡張可能マークアップ言語
pdf	バイナリファイル	Adobe 社の開発した文書形式

拡張子はこの他にも沢山あります。これらの例を見ると，拡張子は例外はありますが[*18]，大体半角 3 文字です。これは MSDOS では拡張子は 3 文字以下という規則を踏襲しているからです。現在の Windows では，かなり長い拡張子も原理的には可能ですが，半角 3 文字を使うのが良いでしょう。

5.3 アイコンと拡張子

Windows では普通，アイコンを使ってファイルを表します。使うアイコンはファイルの種類によって分類されています。アイコンの画像は通常，画像ファイルとして，どこかに保存されていて[*19]，ファイルとアイコンは独立です。ファイルとアイコンを結びつけるのは拡張子とその関係を記述したデータベースです。Windows システムはファイルの拡張子から，対応付けのデータベースによって表示アイコンの種類を選んでいます。拡張子をアイコンとの対応関係のデータベースに登録することを関連付けと言います。この関連付け

[*16] Windows 標準のファイル管理ツール
[*17] Windows 8.1 の場合，他のバージョンでも同様な設定が可能です。
[*18] これの例外として，html がありますが。
[*19] ソフトウエアに内蔵されていることもあります。

データベースでは，拡張子とアイコンの関係だけでなくアイコンをダブルクリック等したときの動作，例えば起動するソフトウエアなどの情報も保存しています。この関連付けは，普通は利用するアプリケーションソフトがインストール時に行いますので，利用者はそれを普通は意識する必要はありません。しかし，

拡張子はそれを表すアイコンとそれを利用するソフトウエアと密接に関係している

ことの理解は重要です。不用意に拡張子を変更すると，ダブルクリックしてもファイルの中身を見ることができなくなったり，対応するソフトウエアが起動できなくなるなどのトラブルを引き起こすこともあります。

■ tbasic のプログラムの拡張子

tbasic のプログラムは標準では，拡張子は tbt です。この意味は

t basic text

から取ったものです。tbasic のエディターでプログラムを作り，それを保存すると，特に指定しない限り，拡張子は tbt になります。

tbasic の環境設定で関連付けを登録すると，tbt と（TBasic.exe に埋め込まれている）アイコンファイルと関連付けが行われ，アイコンが tbt 文書になります。そして更にこのアイコンをダブルクリックすることで tbasic が起動するようになります。

このようにファイルの拡張子は重要な意味があります。ですから，ファイル名は大体自由に付けることができますが，拡張子は慣例に従って付けなくてはいけません[*20]。まとめると

拡張子
・拡張子はファイルの性質を示す。
・拡張子は利用するソフトウエアを示す。
・拡張子は慣用に従って付ける。
・拡張子の変更は慎重に行う。

となります。

[*20] tbasic での拡張子 tbt は，関連付けの目的でのみ使用されています。プログラムとしては単にテキストファイルですので，拡張子が txt であっても，プログラムとして動作させることはできます。

第 II 部
ファイル操作法

6　tbasic で扱えるファイル

tbasic で扱うファイルは大きく分けると，tbasic のプログラムファイルとデータファイルです。tbasic のプログラムは内蔵のエディターで作成するのが普通ですが，でき上がるプログラムファイルは普通のテキストファイルです。ですから，別の（メモ帳のような）エディターで作成したり編集することもできます。

また，tbasic で作成できるデータファイルは普通はテキストファイルです。またテキストファイルであれば，tbasic のデータファイルとして扱うことができます。

さらに，Ver. 1.6 より，テキストファイルを含めた一般のバイナリファイルを，小さなものであれば扱うことができるようになりました。ですから，

・tbasic で扱えるファイルは基本はテキストファイル。
・小さなバイナリファイルも扱うことはできる。

と言えます。

6.1　テキストファイル

■順編成テキストファイル

プログラムで扱うファイルは計算結果や計算のためのデータを保存することに使います。このようなファイルをデータファイルと言います。データファイルを構造で分類すると，

- 順編成ファイル
- ランダムアクセスファイル

に分けることができます。

順編成ファイルは，

テープ の上に 1 列に書かれた文字列の集まり

と考えられます。

テキストファイルはこの形式のファイルです。この形式のファイルは，常に最初から順に読んでいくと言う制約はありますが，形式についての制限が少なく，一般的に良く使われるファイル形式です。

これに対して，ランダムアクセスファイルは一定の形式を予め決めてから使うもので，ランダムアクセスや，データの一部分書き換えができるなど便利な面もあります。しかし，大掛かりになりますので，本格的なデータベースを作るときなどに用います。一般にテキストファイルに形式を導入して，ランダムアクセスファイルとして，処理することも可能ですが，この処理に対応する命令・関数は現在の tbasic ではサポートしていません。

tbasic で扱えるテキストファイルは基本的に順編成テキストファイルです。(順編成テキストファイルと言いましたが，実は，これは普通のテキストファイルと同じものです。)

順編成テキストファイルしか扱えないのは機能が少ないと思うかもしれません。確かに，大きなデータベース等の処理をしたい場合，ランダムアクセスファイルは魅力的です。しかし，順編成テキストファイルはどのような OS であっても，またどのようなやプログラミング言語であっても標準的に取り扱いができる極めて一般的汎用的なファイルです。このことから，この形式のファイルは広く使われています。例えば，異なるデータベースの変換ファイルとしてよく使われる csv ファイルもこの順編成テキストファイルです。またこの他，web 用の html 文書や種々のプログラミング言語のソースプログラムファイルもテキストファイルです。勿論，メールの本文や書式なしの文書はテキストファイルです。

テキストファイルは扱いが簡単で，制約も少ないので小規模なデータを扱うのに適しています。ですから，

> 極めて大規模なデータベースを作るのでなければ，
> 順編成テキストファイルで十分

と言えます。

6.2 バイナリファイル

テキストファイルではないファイルをバイナリファイルと言いましたが，ここでのバイナリファイルは，この意味ではなく，テキストファイルを含む一般のファイルです。コンピューターで扱うファイルは 0,1 で構成されるので，この意味で全てのファイルはバイナリファイルと言い方もしました。ここでのバイナリファイルはこの意味です。

ファイルは，0,1 で構成されています。しかし，ソフトウエアでファイルを読み込んで利用する場合は，ユーザーが目にするものは，0,1 データではなく，テキストファイルであれば，文章ですし，画像ファイルであれば，画面に表現された，画像です。これは，ファイルを目的に合った適当な形に変換して表示されているからです。

普通はこれで十分ですが，時にはファイルの具体的な内容を調べたいことがあるかも知れません。そのような目的でファイルを調べるとき，バイナリファイルとして取り扱うと言います。

tbasic でも Ver. 1.6 より，このような目的で比較的小さなファイルの中身を，直接調べることができるようになりました。

具体的な使用法は「10 バイナリファイルの取り扱い」を参照してください。

7　ファイル処理の概要

前節では，ファイルについての基礎的知識を説明しました。ここでは，ファイル処理の概要を説明します。

7.1　ファイル読み書きの原理

ファイルはディスク等の外部メモリーに作成します。この外部メモリーはハードディスクや USB メモリなどの色々な物理的媒体です。これらの媒体に記録されるファイルは，OS によるファイルシステムで管理され，それらへのアクセスは OS に付随するツールを通して行います。ですから，その扱いは個々のプログラミングシステム固有のものではなく，OS の処理方法に依ります。さらに言えば，それらの方法の大枠は OS にも依りません。

■ファイル処理の３段階

ファイルの読み書き処理は基本的には次の３つの部分からなります。

(1) ファイル用バッファの設定
　ファイルに対応した一定量のバッファ[*21]（ディスク等との読み書きに対する暫定メモリ）を用意する。

(2) ファイルの読み書き処理（読み書きはバッファを介して行われる。）
- ファイルから読み込みの場合，ファイルから一定のデータをバッファに読み込み，そこから必要なデータを随時読み込む。
- ファイルへ書き込みの場合，書き込むデータを随時バッファに書き込み，一定量に達した場合，それをディスクに書き込む。

(3) ファイル用バッファの終了処理
- 読み込みの場合，バッファを開放する。
- 書き込みの場合，バッファに残っているデータをディスクに書き込み，バッファを開放する。

これらの処理はプログラミングでは，普通次のような処理として表されます。

■読み込みの場合

- Open
- Read
- Close

■書き込みの場合

- Open
- Write（Print）
- Close

通常ファイル処理はこのような一連の処理として実行されますが，ここで，`Open`（前処理）や `Close`（後処理）は定型的なので，まとめて

[*21] ここで，バッファはキュー（Queue）のことです。Queue は先入れ先出しのデータ型で，一列に並んだデータを入力した順に出力します。

- ファイル読み込み
- ファイル書き込み

と処理する方法もあります。

7.2 ファイルの場所の特定

ファイルを扱う場合，対象とするファイルの場所（パス）を正確にプログラムで記述する必要があります。

■ファイルの場所（パス）

コンピューターでのファイルは木構造（tree）として格納されています。ファイルをまとめて入れておく場所として，フォルダー（ディレクトリ）があります[*22]。フォルダーの中には，ファイルや更にフォルダーを入れることができます。このようにして多くのファイルを整理して保存し，利用が可能となります。

最上位の場所をルート（root）と言います。例えば，C ドライブのルートは C:¥で表されます。

例えば，tbasic の set を，C ドライブのルートにある Local と言うフォルダーの中に展開，保存したとき，TBasic というフォルダーでき，その中に配布のファイルが保存されます。この状況を木構造として表すと，次のようになります[*23]。

```
C:¥Local¥TBasic
|   BTutor.chm
|   TBasic.exe
|   TBasic.ini
|   TBWHelp.chm
|
|---doc
|       201810Introtbasic150.pdf
|       201810tbasicSpec150.pdf
|
|---samples
      |   Calendar.tbt
      |   Direct.tbt
・・・
      |
      |---Game
      |       15game.tbt
      |       Biorhythm.tbt
      |
      |---Graphic
      |       ColorNo.tbt
      |       ColorSample.tbt
・・・
      |---Unicode
              ANK.tbt
              CJK.tbt
              FrenchGermanInput.tbt
              Helloworldtbt.tbt
```

ここで例えば，上の ANK.tbt は，

[*22] ディレクトリとフォルダーは同じ意味ですが，ディレクトリは MS-DOS や Linux の用語です。Windows では，フォルダーというのが標準的ですが，MS-DOS での伝統に従って，ファイルを文字的に操作する場合，ディレクトリという用語を使うこともあります。

[*23] ・・・の部分は省略したことを表します。

C ドライブの Local フォルダの中にある TBasic フォルダの中の sample フォルダの中の Unicode フォルダの中にある ANK.tbt

と示されます。これを

```
C:¥Local¥TBasic¥sample¥Unicode¥ANK.tbt
```

と表します[*24]。このこのような表示をパス（path）と言いいます。

■ファイルの場所（パス）の表し方

ファイルの場所（パス）を記述する方法は上の方法が正式なもので，フルパスと言います。しかし，この表現方法がしばしば煩雑であることは確かです。

例えば，配布の TBasic フォルダを winuser というユーザー名のドキュメントフォルダに保存したとします。この場合，ドキュメントフォルダのパスは標準的には

```
C:¥Users¥winuser¥Documents
```

になります。ですからこの場合。上の ANK.tbt のパスは，

```
C:¥Users¥winuser¥Documents¥TBasic¥sample¥Unicode¥ANK.tbt
```

となります。かなり複雑です。

ファイル操作を行う場合，対象となるファイルを正確に指示する必要があります。しかし，ファイルを処理する度に，上のような表示を使わなければならないのは煩雑です。またそれにより誤りを起こし易くなり。避けたいところです。

このようなことを避けるためにカレントディレクトリ（フォルダ）と言う概念があります。

■カレントディレクトリの利用

カレントディレクトリ

カレント（current）とは「現在の」の意味で，カレントディレクトリとは現在，処理の対象となっているディレクトリのこと。

Windows での処理では，ファイルを特定する場合，その対象となるファイルのアイコンをクリック等を行います。この場合，対象となるファイルはアクティブなフォルダの中にあります。このアクティブなフォルダがカレントディレクトリです。

ソフトウェアの実行の視点からすると，ソフトウェアが実行される場合すべて，その実行フォルダまたは作業フォルダがそのソフトウエアにとってのカレントディレクトリです。

ファイルを特定する場合，カレントディレクトリを基準にして指示すると便利な場合がしばしばあります。そこで現在のコンピューターファイルシステムでは，この方法「カレントディレクトリを基準にしてファイルを指示する方法」がサポートされています。

[*24] ここで大文字，小文字が使われています。表示名としては，大文字，小文字の区別はありますが，パスの表示方法としては大文字，小文字の区別はありません。ですから
C:¥LOCAL¥TBASIC¥SAMPLE¥UNICODE¥ANK.TBT
としても同じです。

■カレントディレクトリを基準にしてファイルを特定する方法

- カレントディレクトリでの直接指定
  カレントディレクトリにあるファイルはファイル名のみで指示できる。
  例えば，ユーザー winuser が，配布の TBasic フォルダをドキュメントフォルダに保存したとします。
  そして，現在のカレントディレクトリが上の例にある Unicode であったとします[*25]。
  このとき，ANK.tbt は，単に
  ```
  ANK.tbt
  ```
  とするだけで指示できます。
- カレントディレクトリ経由での相対パス指定
  カレントディレクトリの一つ上にあるファイルは
  ```
  ..¥
  ```
  を先頭につけて指示できます。
  例えば，今の例で，一つ上のフォルダーにある Calendar.tbt は，
  ```
  ..¥Calendar.tbt
  ```
  で指示できます。同様に，一つ上のフォルダーの中にあるフォルダー Game の中にある 15game.tbt は
  ```
  ..¥Game¥15game.tbt
  ```
  で，二つ上のフォルダーにある TBasic.exe は
  ```
  ..¥..¥TBasic.exe
  ```
  で指示できます。

カレントディレクトリを使ってファイルを指定した方が多くの場合，簡単になり，間違いが少なくなります。実際，上の例をフルパス（絶対パスとも言います。）で表すと，それぞれ

```
C:¥Users¥winuser¥Documents¥TBasic¥sample¥Unicode¥ANK.tbt
C:¥Users¥winuser¥Documents¥TBasic¥sample¥Calendar.tbt
C:¥Users¥winuser¥Documents¥TBasic¥sample¥Game¥15game.tbt
C:¥Users¥winuser¥Documents¥TBasic¥TBasic.exe
```

となります。まとめると，

ファイルの指定方法
・カレントディレクトリのファイルの直接指定
・カレントディレクトリからの相対指定
・フルパスを使った絶対指定

となります。

[*25] この例では Unicode ディレクトリは
C:¥Users¥winuser¥Documents¥TBasic¥sample¥Unicode
になります。

■ tbasic でのファイルの指定法

tbasic でのファイルの指定法について説明します。tbasic でのファイルの指定は，上に説明した普通に行われている方法で行います。

フルパスを使った絶対指定は，tbasic でもいつでも可能ですが，必ずしも使いやすいものではありません。これに対して，カレントディレクトリまたはそれからの相対指定は使いやすく便利なものです。そのことから，tbasic ではカレントディレクトリからの指定の使用を推奨しています。

ただそのためには，現在のカレントディレクトリが何処なのか知らなければなりませんし，必要に応じてその設定もできなければなりません。そのために，tbasic ではこれらに関するいくつかの命令・関数が以下のように備わっています。

- GetCurrentDir：カレントディレクトリをフルパスの文字列として返す関数。
- GetProgramDir：現在実行中のプログラムが保存ディレクトリをフルパスの文字列として返す関数。
- ChDir：カレントディレクトリを指定したディレクトリに変更する命令。

これらを組み合わせるとファイルの指定がカレントディレクトリを基準とした指定を簡単に使うことができます。次のシナリオでプログラムを作成するとします。

シナリオ１

・あるプログラムとそこで使うデータ用のファイルはそのプログラムと同じディレクトリに保存する。

このシナリオの場合，プログラム構成を次のようにすることで，ファイル処理が可能となります。

例 7.1 (シナリオ 1 の雛形).

```
CDirBackup$=GetCurrentDir
ChDir GetProgramDir

' 以後カレントディレクトリはプログラムが保存されたディレクトリ
'
'・・・　色々な処理：使用ファイルの指定はファイル名のみで可能
'

ChDir CDirBackup :'End の直前にカレントディレクトリを元に戻す。
End
```

8 テキストファイル処理

ファイル処理は Open，Read(Write)，Close の３段階で行われると説明しました。ここで，Read(Write) を一気に行うとすると，Open と Close はまとめて一つの命令として簡易的に処理することができます。

.NET 系の言語では，この簡易的ファイル読み書きの方法が提供されています。tbasic でもこれと類似な方法をサポートしています。ここではそれについて説明します。

8.1 WriteAllLines と ReadAllLines

tbasic で扱うファイルはテキストファイルと説明しました。テキストファイルは行の集まりと言えますが，これは文字列の配列変数によって構成されると考えられます。そこで，テキストファイルと文字列配列を対応させて，一気に読み書きする方法が考えられます。

まず，ファイルの作成・書き込みについて説明します。

■テキストファイルの作成・書き込み

テキストファイルの作成・書き込みには WriteAllLines を使います。

WriteAllLines

WriteAllLines は 指定した文字列配列の内容のテキストファイルを作成します。

使い方

```
WriteAllLines(FileName, Lines())
WriteAllLines(FileName, Lines(), Encoding)
```

WriteAllLines は指定したファイル FileName に文字列配列 Lines() の内容を書き込みます。tbasic では，Lines(0) に記述された行数を書き込みます[*26]。Val(Lines(0)) が正数でない場合は，Lines(1) から Lines の宣言された行数までを書き込みます。

Encoding が指定されている場合，指定された Encoding で書き込みます。指定していない場合，デフォルトの Encoding で書き込みます[*27]。ファイルが存在する場合は，ファイルを削除し，新たにファイルを作成して内容を書き込みます。ファイルが書き込めない場合はエラーになります。ここで，Encoding は，エンコーディング名または，コードページ番号で指定します。

[*26] この仕様は Visual Basic とは少し異なります。

[*27] デフォルト Encoding は tbasic 起動時は UTF-8 です。SetEncoding コマンドで設定が可能です。

前節で説明したシナリオ 1 の状況で例をあげましょう。説明のために行番号を付けますが，実際のプログラムでは行番号はありません[*28]。

例 8.1 (ファイルの書き込み).

```
010 CDirBackup$=GetCurrentDir
020 ChDir GetProgramDir
030
040 Dim TLines$(1000)
050 FN$="Test.txt"
060 TLines$(1)= "1 行目の内容"
070 TLines$(2)= "2 行目の内容"
080 TLines$(0)= Str$(2)
090 WriteAllLines(FN$,TLines$())
100
110 ChDir CDirBackup$
120 End
```

説明.

- まずこのプログラムを適当な名前，例えば WTest.tbt として適当なフォルダーに保存します。GetProgramDir は保存されたプログラムでの実行が必須です。
- 10,20 行がシナリオ 1 の状況設定です。これにより 20 行の実行で，このプログラムを保存したフォルダーがカレントディレクトリになります。
- 40 行は作成するファイル内容を保存する文字列配列変数の宣言です。必要な行数だけの宣言で良いわけですが，ここでは，多少多めに 1000 行宣言しています。変数名は別なものでも構いません。
- 50 行で作成するファイル名を指定しています。
- 60 行で作成するファイルの 1 行目の内容を指定しています。
- 70 行で作成するファイルの 2 行目の内容を指定しています。この例は 2 行しかありませんが，最大配列変数で宣言された最大数までの行を指定できます。今の場合なら，最大 1000 行可能です。
- 80 行で書き込む行数を指定しています。この場合は 2 行ですが，文字列として指定する必要がありますから，Str$関数を使って，Str$(2) としています。
- 90 行で書き込み処理を行っています。これにより，このプログラムが保存されているフォルダーに，Test.txt という 2 行の内容をもつテキストファイルが作成されます。
- 110 行で，カレントディレクトリを元のものに戻しています。

□

このプログラムの実行により，

1 行目の内容 2 行目の内容

という 2 行の内容からなるファイル Test.txt が作成されます。実際に，作成された Test.txt のアイコンをダブルクリックして内容を確認してみましょう。

次に，ファイルの読み込みについて説明します。

[*28] 以後 tbasic で作成するファイルの拡張子は便宜上 txt とします。規則上は txt でなくても良いのですが，txt は普通のエディターで開けますので，処理対象ファイルの内容が簡単に確認できるという便利さがあります。

■テキストファイルの読み込み

テキストファイルの読み込みには ReadAllLines を使います。

ReadAllLines

ReadAllLines は指定したテキストファイルの各行を指定した文字列の配列変数に格納します。

使用法

```
RLines()=ReadAllLines(FileName)
RLines()=ReadAllLines(FileName, Encoding)
```

ReadAllLines は指定したファイル FileName の内容の各行を，指定した文字列配列変数 RLines に格納します。格納する配列変数 RLines は予め宣言されていなければなりません。読み込んだ行数は返す配列変数の 0 番目に文字列表現で格納されます。ファイルが宣言された以上の行数を持つ場合は，0 番目には ”-1” が格納されます[*29]。

Encoding が指定されている場合，指定された Encoding で読み取ります。指定していない場合，自動判定して読み込みます[*30]。ここで，Encoding は，エンコーディング名または，コードページ番号で指定します。

例で説明しましょう。上で説明したシナリオ 1 の状況で作成した Test.txt を読み込む例をあげましょう。説明のために行番号を付けますが，実際のプログラムでは行番号はありません。

例 8.2.

```
010 CDirBackup$=GetCurrentDir
020 ChDir GetProgramDir
030
040 Dim RL$(10000)
050 FN$="Test.txt"
060 RL$()=ReadAllLines(FN$)
070 For i=1 To Val(RL$(0))
080    Print RL$(i)
090 Next i
100
110 ChDir CDirBackup$
120 End
```

説明.

- まずこのプログラムを適当な名前，例えば RTest.tbt として先ほどの Test.txt が保存されているフォルダーに保存します。GetProgramDir は保存されたプログラムでの実行が必須です。
- 10,20 行がシナリオ 1 の状況設定です。これにより 20 行の実行で，このプログラムを保存したフォルダーがカレントディレクトリになります。
- 40 行は読み込むファイル内容を格納する文字列配列変数 RL$の宣言です。入力ファイルの行数だけ宣言で良いわけですが，入力ファイルの行数は不明な場合が多いので，多めに宣言します。ここでは，10000 行宣言しています。変数名は別なものでも構いません。
- 50 行で作成するファイル名を指定しています。
- 60 行で読み込み処理を行い，Text.txt の内容が文字列配列変数 RL$に格納されます。格納された行数は RL$(0) に文字列として格納されます。

[*29] この仕様は Visual Basic とは少し異なります。

[*30] 殆どの場合，自動判定で正しく読み取れますが，正しく読み取れない場合は，エンコーディング名を指定して読み込んでください。

- 70 行から 90 行が読み込んだ内容を For 文で実行画面に表示しています。読み込んだ行数は Val(RL$(0)) で与えられます。For i=1 to Val(RL$(0)) の i に対して，RL$(i) を表示します。
- 110 行で，カレントディレクトリを元のものに戻しています。

□

このプログラムを実行すると実行画面に

1 行目の内容
2 行目の内容
OK

と表示されます。

8.2 WriteAllText と ReadAllText

前項では，行単位での読み書きについての方法を説明しました。似た方法ですが，ファイル内容すべてを 1 つの文字列として読み書きする方法もあります。

文字列は人間が認識できる文字の並びでした。テキストファイルは文字列と改行の並びでしたが，実は改行は ASCII 文字から構成されるもので，コンピューター上で標準的に認識できる（1 文字または 2 文字の）文字列の一種です。ですから，

テキストファイルは改行を含んだ一つの文字列

です。

■テキストファイルの作成・書き込み

WriteAllText

WriteAllText は 指定した文字列の内容のテキストファイルを作成します。

使い方

```
WriteAllText(FileName, Text)
WriteAllText(FileName, Text, Encoding)
```

WriteAllText は指定したファイル FileName に文字列 Text の内容を書き込みます。Encoding が指定されている場合，指定された Encoding で書き込みます。指定していない場合，デフォルトの Encoding で書き込みます。ファイルが存在する場合は，ファイルを削除し新たにファイルを作成して内容を書き込みます。

■テキストファイルの読み込み

ReadAllText

ReadAllText は 指定したファイルの内容の指定した文字列に格納します。

使い方

```
RT$=ReadAllText(FileName)
RT$=ReadAllText(FileName, Text, Encoding)
```

ReadAllText は指定したファイル FileName の内容を文字列 RT$に格納格納します。Encoding が指定されている場合，指定された Encoding で読み込み格納します。指定していない場合，自動判定して読み込みます。

WriteAllText と ReadAllText での文字列には改行が含まれれます。ですから，この場合，文字列の処理には含まれる改行コードについての考慮が必要で，幾分処理が複雑になります。また，文字列がかなり大きな場合，処理に依っては，時間のかかることも考えられます。ですから普通のファイル処理には余り適していないかも知れません。この命令が有用な場合は，ファイルに含まれる文字列等の一括変換やファイルの Encoding の変換などでしょう。

このようなものの例を 1 つあげます。次のプログラムはファイル FNS$に含まれる文字列"コンピュータ" を文字列"コンピューター"に置換し，それをファイル FNT$に保存します[*31]。

例 8.3 (置換処理).
```
CDirBackup$=GetCurrentDir
ChDir GetProgramDir

FNS$="STest.txt"
FNT$="TTest.txt"
A$="コンピュータ"
B$="コンピューター"

RL$=ReadAllText(FNS$)
WT$=Replace$(RL$,A$,B$)
WriteAllText(FNT$,WT$)

ChDir CDirBackup$
End
```

8.3 まとめ

この節では，ファイルの簡易的処理法について説明しました。この方法はファイルの読み書きを一気に行い，Open や Close と言った命令を（明示的に）使わないことに特徴があります。

またこれにより，ファイル処理が基本的にコンピューター内部での文字列処理の問題に帰着されることも意味します。特に，WriteAllLines，ReadAllLinse による処理はファイル処理を普通の文字列処理の問題に帰着させます。

文字列処理はプログラミング処理の中での最も基本的なもので，大きな能力を持っています。これを活用することで，ファイル処理がファイルをあまり意識せずに，強力な処理が可能となります。

ですから，

> ファイル処理は，可能な限り
> WriteAllLines，ReadAllLines
> を使うのが良い。

と言えます。

[*31] このプログラムは STest.txt に文字列"コンピューター"が含まれると，その文字列は"コンピューター−"に置換されます。このようなことを避けるためにはいくつかの処理が必要です。

9　テキストファイル処理プログラム例

9.1　行番号の削除・追加

この文書の説明の中で，しばしば

> 説明のために行番号を付けますが，実際のプログラムでは行番号はありません。

といった表現があります。

ここでは，ファイル操作の一つの例として，このようにプログラムに行番号を追加したり，削除したりすることを考えます。

行番号付のプログラムでは，行番号の数値を Goto 文など，プログラムの中で使用している場合が多くあります。このようなプログラムでは，行番号を削除すると，プログラムとして，動作しなくなります。しかし，逆に元々行番号がないプログラムは，行番号をプログラムの中で使っていることはありません。このようなプログラムに，形式的に行番号を付けた場合，そのプログラムは，付けないプログラムと同様な動作をします。

ここではこのようなプログラムを対象に行番号の追加，削除を考えます。

9.1.1　行番号の追加

まず，行番号無しのプログラムに行番号を追加する処理を考えましょう。

どのような行番号を付けるかによって多少の違いはありますが，いずれにしてもプログラムは簡単です。ここでは

test1.tbt

```
'test1.tbt
For i=1 To 10
   Print i*i
Next i
End
```

$\Longrightarrow$

test2.tbt

```
01 'test1.tbt
02 For i=1 To 10
03    Print i*i
04 Next i
05 End
```

のように番号を付けましょう。行番号を指定桁数で表示し，指定桁未満の数は，先頭を 0 で詰めることにします。但し，プログラム全体の行数は指定桁以下であるとします。

例えば，数 i を 4 桁以下の数として，i を必要なら先頭を 0 で詰めで，4 桁で表わすには，例えば

```
Right$(Str$(10000+i),4)
```

とすれば良いでしょう。もう少し直接的に，

```
Right$("000"+Trim$(Str$(i)),4)
```

でも可能です[*32]。

このことに注意して，以下のプログラムができます。ここでは，説明のために行番号を付けますが，実際のプログラムでは行番号はありません。

[*32] Str$(i) の結果は，i が正の場合，符号分として先頭に 1 文字空白が付きます。例えば，Str$(3)=" 3"となります。この先頭の空白を除くため，Trim$(Str$(i)) を使っています。

例 9.1 (行番号追加).

```
01 ' 行番号追加プログラム
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim InF$(1000)
06 Dim OutF$(1000)
07 InFName$ ="test1.tbt"
08 OutFName$="test2.tbt"
09 Keta=2
10 InF$()=ReadAllLines(InFName$)
11 OutF$(0)=InF$(0)
12 Gyousu=Val(InF$(0))
13 DCounter = 10^Keta
14
15 For i=1 To Gyousu
16    OutF$(i)= Right$(Str$(DCounter+i),Keta)+" "+InF$(i)
17 Next i
18 WriteAllLines(OutFName$,OutF$())
19
20 ChDir CDirBackup$
21 End
```

説明.

- 2，3，20 行はシナリオ 1 の設定。
- 10 行で InFName$を文字列配列変数 InF$に読み込み。
- 13 行で Keta 分の 0 詰め用設定。
- 16 行で行番号の追加，出力用文字列変数 OutF$(i) に格納。
- 19 行で OutF$を出力。

□

このプログラムを実行すると，上の左側のプログラムから右側のプログラムが得られます。

また，番号付けが 10 ずつのもの，例えば，

test1.tbt

```
'test1.tbt
For i=1 To 10
   Print i*i
Next i
End
```

$\Longrightarrow$

test2.tbt

```
010 'test1.tbt
020 For i=1 To 10
030    Print i*i
040 Next i
050 End
```

にするには，以下のように

```
・・・
09 Keta=3
・・・
15    OutF$(i)= Right$(Str$(DCounter+i*10),Keta)+" "+InF$(i)
・・・
```

と 2 行を変更すれば可能です。

9.1.2　行番号の削除

前項で，行番号の追加を行いました。ここではその逆に，追加したプログラムから，その行番号を削除するプログラムを作成しましょう。これは各行先頭の何文字かを削除するだけですから，簡単です。

プログラムは以下のようになります。ここでも，行番号は説明用で，実際のプログラムではありません。

例 9.2 (行番号の削除).

```
01 ' 行番号削除プログラム
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim InF$(1000)
06 Dim OutF$(1000)
07 InFName$="test2.tbt"
08 OutFName$="test1.tbt"
09 Keta=2
10 InF$()=ReadAllLines(InFName$)
11 OutF$(0)=InF$(0)
12 Gyousu=Val(InF$(0))
14
15 For i=1 To Gyousu
16    LenofLine = Len(InF$(i))
17    OutF$(i)= Right$(InF$(i),LenofLine-Keta-1)
18 Next i
19 WriteAllLines(OutFName$,OutF$())
20
21 ChDir CDirBackup$
22 End
```

説明.

- 16 行で入力行の長さを求める。
- 17 行で，入力行から，Keta+1 文字少ない文字列を右側から取り
  （＝先頭から Keta+1 文字削除した文字列），それを出力行とする。
- 他は行番号の追加プログラムと同様。

□

■最後に

タイトルを見ると何か大げさなプログラムがありそうですが，実際は簡単なものでした。

テキスト変換は読み込む内容，出力する内容が確定して定型的なものなら，意外と簡単に実現できます。特に大量のデータの単純な置き換えなどは，このような処理プログラムで実現するのに適しています。

BASIC は計算だけという訳ではありません。
テキスト処理についてもかなりの能力を持っています。

現在のバージョンでは，正規表現といったことは無理ですが，それでも多くのことが可能です。そして色々な処理が意外と簡単に，そして高速に実現します。色々とチャレンジしてみて下さい。

9.2 成績処理プログラム

ここでは，ファイル処理の基本を成績処理プログラムを例にして説明します。ここでの説明は，プログラムとしては，tbasic を使いますが，基本的にはどの BASIC でも殆んど同様に可能ですし，考え方，手法は別の言語でも適用可能です。

コンピューターというと，計算がすぐに頭に浮かびますが，コンピューターは計算だけでなく文書処理も得意です。ワープロやエディターを見れば，そのことは納得できますが，そのワープロやエディターもプログラム言語によって書かれ，動かされています。つまり，

プログラム言語は文書処理も得意

なのです。このことは，多くの汎用言語に通用する事実です。勿論，BASIC，そして tbasic でも通用します。

文書処理は状況により，色々なことが考えられますが，基本的には，与えられた文書を色々に加工することです。ここでは成績処理プログラムを例にして説明しますが，他の状況でも応用可能でしょう。

9.2.1 成績処理プログラムの概要

次のような処理を行うことを考えましょう。

(1) 英・数・国，３科目の複数人の成績データを入力する。
(2) 合計点と順位を計算する。
(3) 成績データを個票で出力する。

このような処理を行う場合，現在では，Excel のような優れた表計算ソフトがありますから，これを使って行うのが一般的でしょう。そして，このような処理を行う BASIC のプログラムを作成して，それを実際に使用するのは，効率からみると現実的ではないかも知れません。

それでは，プログラムを書く意味は無いのでしょうか？

Excel のような表計算ソフトは，

手で簡単に処理できること

を基本に置いています。これは，比較的少数のデータを少数回処理するのに適しています。手で行う処理は，後でその処理手続きを確認できません。例えば，10000 件のデータの，あるフィールドを手でドラッグしてコピーする処理を考えてみましょう。このような処理は，少数回のものであれば，慎重に行うことで，正しい処理が可能です。しかし，そのような処理を 100 回行って最終的に処理が終了するようなものは，なかなか大変です。そしてその 100 回の処理の途中で，間違いが無かったことを保証するのは難しいものです。

勿論，このような大規模な処理を Excel で行うことは可能です。それはマクロを使うことです。マクロによって信頼の置ける大規模処理が可能になります。実はこのマクロは将にプログラミングそのものです。そして，Excel におけるマクロ言語は VBA という BASIC 言語です。

ですから，VBA(Visual Basic for Applications) を使うことで，かなり大掛かりな文書処理を含めた，計算処理を行うことができます。しかし，一方 VBA は Excel の中から実行するものですから，それなりに重い処理になります。ちょっとした処理ならば，Excel を起動するまでも無く，別の方法で行うほうが簡単な事もあります。

コンピューターを上手に利用するための原則として，私は次があると考えています。

> コンピューター活用の原則
> 一つの方法・手段に拘るのではなく，
> 使用可能な方法の中から最も効率的なものを選択して，
> それを用いる

この原則に従えば，全体の処理を，すべて 1 つのプログラム・処理系で行うのではなく，いくつかの機能に分割し，それぞれ分割処理を適切な方法で実行し，それを統合して求める結果を得るのが適切な方法でしょう。勿論，機能分割することのデメリットもありますから，それも考慮し全体の効率を評価する必要はあります。

ですから，以下の説明・プログラム例は次のような位置づけであると考えて下さい。

- いくつかの部分を BASIC で処理する方法を説明します。
- しかし，現実的には別の手段で行ったほうが良い場合もあります。
  この場合は，BASIC での処理の例として考えて下さい。
- ここで説明する方法は，例えば Excel のマクロでも実現できますから，
  それへのヒントと考えて下さい。
- それでも，これらの方法は場合によっては，かなりの効率的な方法でもあることあります。

9.2.2 データの入力

色々なデータに対して，計算処理を行う場合，その元になるデータは既に与えられているか，手入力等で実際に入力することになります。例えば，直前に実施された記述式試験の成績データは，いずれかの段階で手入力する必要があります。

しかし，試験を受けた人の名前は，既に既にファイルとしてあるかも知れません。ここでは，次の状況を考えましょう。

- 氏名の一覧のファイルはすでにある。
- 採点済み試験答案用紙がある。

この状況で，最終的に，英・数・国，３科目の複数人の成績データを入力する方法を考えましょう。最終的には

氏名，英語，数学，国語

の順で，カンマ区切りでのテキストデータファイルを作るとします。これは例えば，

```
日本太郎,80,90,80
```

のような形式が 1 行となっているテキストファイルです。

この形式は csv (Comma Separated Value) 形式といわれ，汎用のデータ形式です。Excel でも拡張子が csv

の，この形式のファイルは表として読みこむことができます。1000 名分であれば，1000 行のテキストファイルです。

氏名については既にリスト simei.txt があるものとしましょう。例えば，

```
日本太郎
東京花子
大阪一郎
京都春子
・・・
```

という一覧の内容をもつファイルです。この各人に対して，英・数・国の点数を入力するわけですが，このくらい（例えば，４名）のものであれば，Excel で氏名の一覧を読み込み，各行のセルに対して入力していくのが現実的です。

また，最終的に実際に作るファイル seiseki.csv は

```
日本太郎,80,90,80
東京花子,70,50,60
大阪一郎,65,70,50
京都春子,50,45,90
・・・
```

というものですから，上の simei.txt を適当なエディターを使って，上の様な形式で実際に入力することも可能でしょう。そこでここでは少し大掛かりなデータの入力を想定してみます。例えば，3 項目で無く 10 項目程度

```
日本太郎,80,90,80,70,65,80,90,90,80,70
```

のようなものです。これくらいになると，手入力する場合，対応項目を間違えて入力することも考えられます。そこで，実際に科目別入力ファイルを作ってそれを合成することにしましょう。

それは，例えば，英語入力用のファイルとして，EInput.txt として

```
日本太郎: 英語:
東京花子: 英語:
大阪一郎: 英語:
京都春子: 英語:
・・・
```

を作り，これにエディターを使って，採点済み試験答案用紙から点数を入力します。点数を入力した結果，EInput.txt は

```
日本太郎: 英語:80
東京花子: 英語:70
大阪一郎: 英語:65
京都春子: 英語:50
・・・
```

のようになります。これらを各科目（数学：MInput.txt，国語：JInput.txt）について作成し，これから，まとめて，最終的に上にあげた seiseki.csv を作成します。例えば，

EInput.txt

```
日本太郎: 英語:80
東京花子: 英語:70
大阪一郎: 英語:65
京都春子: 英語:50
・・・
```

MInput.txt

```
日本太郎: 数学:90
東京花子: 数学:50
大阪一郎: 数学:70
京都春子: 数学:45
・・・
```

JInput.txt

```
日本太郎: 国語:80
東京花子: 国語:60
大阪一郎: 国語:50
京都春子: 国語:90
・・・
```

から，以下を作成します。

seiseki.csv

```
日本太郎,80,90,80
東京花子,70,50,60
大阪一郎,65,70,50
京都春子,50,45,90
・・・
```

さて，以上の処理を実現するプログラムを書いて見ましょう。

■入力ファイルの作成

以下は 氏名のリスト simei.txt から，成績入力用のファイル EInput.txt，MInput.txt，JInput.txt を作成するものです。説明のために行番号を付けていますが実際のプログラムにはありません。

例 9.3 (入力用ファイルの作成).

```
01 ' simei.txt から EInput.txt, MInput.txt, JInput.txt を作る
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim S$(1000)
06 Dim EI$(1000)
07 Dim MI$(1000)
08 Dim JI$(1000)
09 S$()=ReadAllLines("simei.txt")
10 For i=1 To Val(S$(0))
11    EI$(i)=S$(i)+": 英語:"
12    MI$(i)=S$(i)+": 数学:"
13    JI$(i)=S$(i)+": 国語:"
14 Next i
15 EI$(0)=S$(0)
16 MI$(0)=S$(0)
17 JI$(0)=S$(0)
18 WriteAllLines("EInput.txt",EI$())
19 WriteAllLines("MInput.txt",MI$())
20 WriteAllLines("JInput.txt",JI$())
21
22 ChDir CDirBackup$
23 End
```

説明.

- 2，3，22 行はシナリオ 1 の設定。
- 5，6，7，8 行で simei.txt を読み込み用文字列配列変数 S$の設定，

EInput.txt，MInput.txt，JInput.txt 用出力文字列配列変数 EI$，MI$，JI$の設定。

- 9 行で simei.txt を S$に読み込み。
- Val(S$(0)) は読み込んだ行数＝氏名の数。
- 11〜13 行で各ファイル出力用の文字列を設定。15〜17 行で各ファイル出力の行数を設定。
- 18〜20 行で各ファイルを出力。

□

英・数・国用に殆ど同じ処理を３つ記述しているのは，多少スマートさを欠きますが，３つ位なら，この方が分かりやすいとも言えます。またこの個数が多い場合，このような書き方を避けることも可能です。

上のプログラムは単純なものですが，ファイル処理を使うと，工夫によりリストファイルから，色々なリストを使ったファイルを作ることができる例と考えて下さい。

■データ入力

上のプログラムで作成された，EInput.txt，MInput.txt，JInput.txt に適当なエディターを使ってデータを入力します。

拡張子を csv にすることで，Excel を使って入力することもできます。

■ファイルの合成

複数のファイルから必要な部分を抽出し，合成をする場合，文字列から，必要な部分を抽出することが必要になります。ここでの抽出処理は簡単のため，tbasic 独自の Split$関数を使うことにします[*33][*34]。そこでまず，Split$の使い方を説明します。

Split$

Split$ は区切り文字に従って文字列を分解します。

使い方

Split$(A$,B$)

但し，A$ は配列変数ではないとします。

Split$(A$,B$) は文字列 A$を区切り文字 B$を使って分解し，最初の区切りまでを区切りを除いて返します。この関数を実行すると，文字列変数 A$ から最初の区切り文字までが削除されます。

例えば，A$="日本太郎: 英語:80" の場合，区切り文字: で，日本太郎，英語，80 をそれぞれ分離することができます。次のプログラム

```
A$="日本太郎: 英語:80"
A1$=Split$(A$,":")     :' A$="英語:80"になる。
A2$=Split$(A$,":")     :' A$="80"になる。
Print A1$
Print A2$
Print A$
```

を実行すると，実行画面に

```
日本太郎
英語
80
```

[*33] Visual BASIC でも Split 関数があります。目的は同じですが，tbasic では配列の戻り値がありませんので，使い方は違います。

[*34] Split$を使わずに，標準的な BASIC の関数だけで，このためのプログラムを書くことも可能です。

と表示されます。

次は成績が入力された EInput.txt，MInput.txt，JInput.txt から，求める seiseki.csv を作るプログラムです。ここでも，説明のために行番号を付けていますが実際のプログラムにはありません。

例 9.4 (ファイルの合成).

```
01 ' EInput.txt, MInput.txt, JInput.txt から，seiseki.csv  を作る
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim SA$(1000)
06 Dim EI$(1000)
07 Dim MI$(1000)
08 Dim JI$(1000)
09 EI$()=ReadAllLines("EInput.txt")
10 MI$()=ReadAllLines("MInput.txt")
11 JI$()=ReadAllLines("JInput.txt")
12 Ninnzu = Val(EI$(0))
13 For i=1 To Ninnzu
14    Tmp$=EI$(i)
15    SimeiT$=Split$(Tmp$,":")
16    Dummy$=Split$(Tmp$,":")
17    SeisekiE$=Tmp$
18    Tmp$=MI$(i)
19    Dummy$=Split$(Tmp$,":")
20    Dummy$=Split$(Tmp$,":")
21    SeisekiM$=Tmp$
22    Tmp$=JI$(i)
23    Dummy$=Split$(Tmp$,":")
24    Dummy$=Split$(Tmp$,":")
25    SeisekiJ$=Tmp$
26    SA$(i)=SimeiT$+","+SeisekiE$+","+SeisekiM$+","+SeisekiJ$
27 Next i
28 SA$(0)=Str$(Ninnzu)
29 WriteAllLines("Seiseki.csv",SA$())
30
31 ChDir CDirBackup$
32 End
```

説明.

- 2，3，31 行はシナリオ 1 の設定。
- 5，6，7，8 行で seiseki.csv 書き込み用文字列配列変数 SA$，
  EInput.txt，MInput.txt，JInput.txt 用文字列配列変数 EI$，MI$，JI$の設定。
- 9，10，11 行で EInput.txt，MInput.txt，JInput.txt を EI$，MI$，JI$に読み込み。
- Val(EI$(0)) は読み込んだ行数＝氏名の数。Val(MI$(0))，Val(JI$(0)) と同じ。
- 14〜17 行で EInput.txt より，各行氏名 SimeiT$と点数 SeisekiE$を抽出。
- 18〜21 行で MInput.txt より，各行点数 SeisekiM$を抽出。
- 22〜25 行で JInput.txt より，各行点数 SeisekiJ$を抽出。
- 26 行で seiseki.csv の各出力行を設定。csv ファイルの出力のため，区切り文字をカンマ”,” とする。
- 28 行で seiseki.csv の出力行数を設定。
- 29 行で seiseki.csv を出力。

□

このプログラムは，３つのファイルの指定された部分を抽出し，合成するものでしたが，同様な方針でいくつのファイルであっても，必要な部分を抽出し合成することは可能です。

9.2.3 データの処理

次に

```
日本太郎,80,90,80
東京花子,70,50,60
大阪一郎,65,70,50
京都春子,50,45,90
...
```

といった形式の seiseki.csv ができたとします。これから，これを集計，計算した結果を kekka.csv というファイルに書き出すことを考えます。ここで，kekka.csv は

```
日本太郎,80,90,80,250,1
東京花子,70,50,60,180,4
大阪一郎,65,70,50,185,2
京都春子,50,45,90,185,2
...
```

の形をしているものとします。これらは

氏名，英語，数学，国語，合計，順位

を表しています。

ここで，合計点の計算は簡単ですが，順位の計算は少し考察が必要です。

次の事実に注目しましょう。

- ある点の人の順位は，その点より高い点数の人の合計人数に 1 を加えることで得られる。

そこで，配列変数 goukeiNinzu(i) を用意し，そこに，各合計 i 点の人の人数を格納します。更に順位を求めるために，配列変数 NinzuGE(i) を用意し，i 点以上の人数を格納します。合計点は 0〜300 点ですから，配列宣言は

```
Dim goukeiNinzu(300)
Dim NinzuGE(300)
```

とします。各人の合計点を計算する度に，その合計点の人数を 1 加えることで，各合計点の人数を求めることができます。合計点の人数は，すべての人の点数が分かってからでないと求められません。ですから，次のように順位計算は 3 段階になります。

- 各合計点の計算と各点の合計人数の計算
- 各合計点のその点以上の人数の計算
- 各合計点の人の順位の計算

以上のことに注意して，プログラムを作成すると次のようになります。ここでも，行番号は説明のためで，実際のプログラムにはありません。

例 9.5 (集計・計算処理).

```
01 ' seiseki.csv から kekka.csv を作る
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim SA$(1000)
06 Dim goukeiten(1000)
07 Dim goukeiNinzu(300)
08 Dim NinzuGE(300)
09
10 SA$()=ReadAllLines("seiseki.csv")
11 Ninzu= Val(SA$(0))
12 For i=1 To Ninzu
13    L$ = SA$(i)
14    Simei$ = Split$(L$,",")
15    Eng  = Val(Split$(L$,","))
16    Math = Val(Split$(L$,","))
17    Jpn  = Val(Split$(L$,","))
18    Total= Eng + Math + Jpn
19    goukeiten(i) = Total
20    goukeiNinzu(Total)=goukeiNinzu(Total)+1
21 Next i
22 ' goukeiNinzu(Total) には Total 点を取った人数が入る。
23
24 ' 順位の計算 上位の点数の人数を加える。
25 NinzuGE(300)=goukeiNinzu(300)
26 For i=299 To 0 Step -1
27    NinzuGE(i)=goukeiNinzu(i)+NinzuGE(i+1)
28 Next i
29 ' NinzuGE(i) には i 点以上を取った人数が入る。
30
31 For i=1 To Ninzu
32    Total  = goukeiten(i)
33    If Total=300 Then
34       jyun = 1
35    Else
36       jyun = NinzuGE(Total+1)+1
37    End If
38    SA$(i)=SA$(i)+","+Str$(Total)+","+Str$(jyun)
39 Next i
40 WriteAllLines("kekka.csv",SA$())
41
42 ChDir CDirBackup$
43 End
```

説明.

- 2，3，42 行はシナリオ 1 の設定。
- 10 行で seiseki.csv を文字列配列変数 SA$に読み込み。
- 14〜17 行で英・数・国の点数を抽出。氏名は使用しないが読み飛ばし用。
- 18 行で合計点 Total の計算。19 行で i 番目の人の合計点として，配列変数 goukeiten(i) に格納。
- 20 行で，Total を取った人の人数を 1 増やす。
- 25〜28 行で合計点 i 点以上を取った人の人数の計算，NinzuGE(i) に格納。
- 33〜37 行で合計 Total 点の人の順位を計算。
- 38 行で元データに，合計点と順位を追加，40 行で kekka.csv を出力。

□

この計算処理の部分は，Excel を使って seiseki.csv を操作することで，プログラムを書かずに同じ結果を求めることはできます。しかし，プログラムによる自動化も場合によっては，効果的です。

9.2.4 個票の出力

ここでは，今作った kekka.csv を使って個票（個人ごとの票）を出力するプログラムを考えます。

BASIC で出力する結果は テキストですから，ワープロの様な綺麗な出力はできませんが，一応可能です。綺麗な出力を望む場合は，Excel による出力や Word の差し込み印刷を使った出力などを考えるべきでしょう。

個票を出力する方法としては，大きく分けて，2 通りが考えられます。

- プログラムの中で，個票すべてを作り，出力する。
- テンプレートを作り，そこにデータを差し込み，個票を作る。

ここでは 2 番目の方法で作りましょう。テンプレートとして次の様な出力を得られるようにします。

```
----成績表------------
氏名:
英語:
数学:
国語:
----------------------
合計:
順位:
----------------------
```

具体的出力結果は，例えば

```
----成績表------------
氏名: 日本太郎
英語: 80
数学: 90
国語: 80
----------------------
合計:  250
順位:  1
----------------------
```

といった形のものです。作成の原理は単純です。テンプレートとして，temp.txt を用意します。例えば，

temp.txt

```
----成績表------------
氏名: @SIMEI
英語: @EIGO
数学: @SUUGAKU
国語: @KOKUGO
----------------------
合計: @GOUKEI
順位: @JYUN
----------------------
```

とします。このときこのテキストを読み込んで，@SIMEI 等を対応するものに置換して，出力するだけです。

次がそのプログラムです。

例 9.6 (個票の作成).

```
01 ' kekka.csv, temp.txt から kohyo.csv を作る
02 CDirBackup$=GetCurrentDir
03 ChDir GetProgramDir
04
05 Dim SA$(1000)
06 Dim KA$(10000)
07 Dim TP$(100)
08
09 SA$()=ReadAllLines("kekka.csv")
10 TP$()=ReadAllLines("temp.txt")
11 Ninzu=Val(SA$(0))
12 Kousu=Val(TP$(0))
13 Counter=1
14
15 For i=1 To Ninzu
16    L$=SA$(i)
17    Simei$ = Split$(L$,",")
18    Eng$   = Split$(L$,",")
19    Math$  = Split$(L$,",")
20    Jpn$   = Split$(L$,",")
21    Total$ = Split$(L$,",")
22    jyun$  = Split$(L$,",")
23    For j=1 To Kousu
24       L$=TP$(i)
25       L$ = Replace$(L$,"@SIMEI",Simei$)
26       L$ = Replace$(L$,"@EIGO",Eng$)
27       L$ = Replace$(L$,"@SUUGAKU",Math$)
28       L$ = Replace$(L$,"@KOKUGO",Jpn$)
29       L$ = Replace$(L$,"@GOUKEI",Total$)
30       L$ = Replace$(L$,"@JYUN",jyun$)
31       KA$(Counter)=L$
32       Counter = Counter + 1
33    Next j
34    KA$(Counter)=""
35    Counter = Counter + 1
36    KA$(Counter)=""
37    Counter = Counter + 1
38 Next i
39
40 KA$(0)=Str$(Counter -1)
41 WriteAllLines("kohyo.txt",KA$())
42
43 ChDir CDirBackup$
44 End
```

説明.

- 2，3，43 行はシナリオ 1 の設定。
- 9 行で seiseki.csv を文字列配列変数 SA$に読み込み。
- 10 行で temp.txt を文字列配列変数 TP$に読み込み。
- 17〜22 行で氏名，英，数，国，合計の点数，順位を抽出。
- 25〜30 行で，"SIMEI" 等の置換。各行置換対象文字列があるときだけ，置換が実行される。
- 31 行で出力用文字列変数 KA$(Counter) に格納。
- 41 行で kohyo.txt を出力。

□

このプログラムを実行すると，

```
----成績表------------
氏名：日本太郎
英語：80
数学：90
国語：80
----------------------
合計：  250
順位：  1
----------------------


----成績表------------
氏名：東京花子
英語：70
数学：50
国語：60
----------------------
合計：  180
順位：  4
----------------------
...
```

といった内容の kohyo.txt が作られます。これを印刷して，適当に切れば個票のでき上がりです。

ここでは，単純なテキストをテンプレートとしましたが，多少複雑なテンプレートでも可能です。例えば，TEX のソース文書や html 文書でも作ることもできます。

■まとめ

以上で，求める成績処理プログラムができ上がりました。これらは 4 本のプログラムからできていて，その各々は数十行という短いものです。しかし，処理能力はかなりあります。原理的には，何人分でもこのプログラムで高速に処理できるでしょう。テキスト処理は予想以上に高速に実行されます。例えば，試しに 1 万人分のデータを作って実験してみて下さい[*35]。

以上のプログラムは機能としてはささやかなものですが，必要であれば多くの機能を加えることができます。状況に応じて色々な機能を付け加えて，実行してみて下さい。

[*35] 大きなデータで実行する場合は，配列変数の宣言数の調整は必要です。

9.3 演習問題

ここでは tbasic でのプログラミングでのファイルの取り扱いについて説明してきました。その纏めとして，演習問題を挙げます。基本的には，ここまでの説明でできるものです。数の処理と言うより，文字列処理の問題です。以下のプログラムを作りましょう。

(1) 1001 から 2000 までの素因数分解表を以下の形のテキストファイル Factors.txt として出力するプログラム。

```
 1001: 7: 11: 13
 1002: 2: 3: 167
...
 1999: 1999
 2000: 2: 2: 2: 2: 5: 5: 5
```

但し，プログラムでは WriteAllLines と PFactor を使ってよい。

(2) 上で作成した Factors.txt を読み込んで，そのうち 3 つの素数の積として表される自然数を出力するプログラム。

```
 1001:7:11:13
 1002:2:3:167
...
 1990:2:5:199
 1996:2:2:499
```

(3) 上で作成した Factors.txt を読み込んで，下の形のテキストファイル Factorization.txt として出力するプログラム。

```
 1001=7*11*13
 1002=2*3*167
...
 1999=1999
 2000=2^4*5^3
```

10　バイナリファイルの取り扱い

ここではバイナリファイルの取り扱いについて説明をします。この節では多少特殊な内容を扱います。更に後半の例では，エンコーディングについての知識が必要になります[*36]。

また，Sub や Function と言った副プログラムも使います。これらについては，別文書「構造化プログラミング」，特にそのうち「3 tbasic でのプログラムの構造」を参照してください。

10.1　バイナリファイルの作成・書き込み

tbasic でサポートされているバイナリファイルについての書き込み命令 `WriteAllBytes` です。

`WriteAllBytes`

`WriteAllBytes` は 指定したバイト列の内容のバイナリファイルを作成します。

使い方

`WriteAllText(FileName, BArray())`

`WriteAllBytes` は指定したファイル FileName にバイトファイル列 `BArray()` の内容を書き込みます。ファイルが存在する場合は，ファイルを削除し新たにファイルを作成して内容を書き込みます。tbasic では，`BArray(0)` に記述された行数を書き込みます[*37]。`BArray(0)` が正数でない場合は，`BArray(1)` から `BArray` の宣言された行数までを書き込みます。

バイナリファイルはエンコーディングは関係ないことに注意してください。

■簡単な例

簡単な使用例をあげます。次の例は，２文字のアルファベット「Ab」を Ab.txt に書き込むものです[*38]。

例 10.1 (アスキーの書き込み).

```
10 ChDir GetProgramDir
20 Dim BArray(10)
30
40 BArray(1)=65  ' 65="A"のアスキーコード
50 BArray(2)=98  ' 98="b"のアスキーコード
60 BArray(0)=2
70
80 WriteAllBytes("Ab.txt",BArray())
90 End
```

説明.

- まずこのプログラムを適当な名前，例えば `WriteAllBytes1.tbt` として適当なフォルダー例えば，`WABTest` に保存します。`GetProgramDir` は保存されたプログラムでの実行が必須です。
- 10 行で `WABTest` がカレントディレクトリになります。

[*36] それらの知識について慣れていない人は，別文書「ユニコードへ」または，種々のウキペディア等を参考にしてください。

[*37] この仕様は Visual Basic とは少し異なります。

[*38] 以下の例では，行番号が付いていますが，いつもの通り，これは説明用のもので，実際のプログラムにはありません。

- 20 行は書き込むバイト列を格納する数値配列変数 BArray の宣言です。ここでは，例えば 10 と宣言しました。大きなファイルを書き込むときには大きな数として宣言します。変数名は別なものでも構いません。
- 40 行で書き込む最初のバイトを指定しています。65 は”A” のアスキーコードです。
- 50 行で書き込む２番目のバイトを指定しています。98 は”b” のアスキーコードです。
- 60 行で書き込むバイト数を 2 と指定してます。準備はここまでです。
- 80 行で WriteAllBytes を使って，BArray の内容を Ab.txt という名前で出力します。
- このプログラムを実行すると，WABtest フォルダーに Ab.txt というファイルが作成されます。適当なエディターで Ab.txt を開くと，Ab と言う内容のファイルであることが確認できます。

□

■ JIS X 0213 の一覧の作成

もう一つ例として，JIS X 0213 の文字一覧を作りましょう。Unicode は JIS X 0213 を含みますから，それに含まれる文字を表すことができます。ですから，Unicode の環境で JIS X 0213 の文字一覧を作成することは可能ですが，それには，その対応表が必要になります。

しかし，Shift_JIS2004 エンコーディングを利用すれば，状況は比較的簡単です。実際，１，２面の並びをそのまま Shift_JIS2004 エンコーディングでファイルに書き込めば出来上がりです。

まず，書き込み規則を確認します。Shift_JIS2004 では，２バイトコードは JIS X 0213 の $2 \cdot 94 \times 94$ の表を２バイトを使ってコード化します。対応は以下の通りと規定されています。

JIS X 0213 の JIS_2004 での対応

面番号を m，区番号を k，点番号を t とし，符号化表現の第 1 バイトを S_1，第２バイト目を S_2 とすると S_1，S_2 は次で計算されます。

(1) S_1 の計算

m	k(10 進)	k(16 進)	S_1(16 進)
1	1〜62	1〜3E	$(k + 101)$ div 2
1	63〜94	3F〜5E	$(k + 181)$ div 2
2	1,3〜5,8,12〜15	1,3〜5,8,C〜F	$(k + 1DF)$ div 2 $-$ $(k$ div 8$) \times 3$
2	78〜94	4E〜5E	$(k + 19B)$ div 2

ここで，div 2 は 2 で割ったときの商を表します。

(2) S_2 の計算

(a) t が奇数の場合

t(10 進)	t(16 進)	S_2(16 進)
1〜63	1〜3F	$t + 3F$
64〜94	40〜5E	$t + 40$

(b) t が偶数の場合

t(10 進)	t(16 進)	S_2(16 進)
1〜94	1〜5E	$t + 9E$

この計算式を利用して，次のようにプログラムを作成しました。

少し長いので，分割して説明します。このプログラム全体は，同梱の ¥Samples¥Advanced フォルダーに MakeJIS213ShiftJIS2004.tbt としてあります。

例 10.2 (JISX0213 一覧の作成 1：主プログラム).

```
010 '  Shift_JIS2004 Code を書き込む
020 ' tbasic 1,6 以上
030 ' JISX0213All.txt を読むには，Shift_JIS2004 対応エディターが必要
040
050 ChDir GetProgramDir
060 Public A(100000)
070 Public Dt$
080 Dt$ = ""
090 Call MakePlane1
100 Call MakePlane2
110
120 n = Len(Dt$)/2
130 Call ToArray(Dt$,n,A())
140
150 A(0) = n
160 WriteAllBytes("JISX0213All.txt",A())
170 End
```

説明.

この部分は，主プログラムです。使われる Sub や Function は，以下の部分で記述されます。

- 10 行〜30 行はコメント文です。
- 40 行〜80 行は変数宣言とその初期化です。変数は以下の Sub で使うので，Public 宣言をしています。Dt$は，書き込むバイト列の 16 進表現文字列です。2 文字で 1 バイトを表します。最終的に書き込むバイト列は数値配列ですが，文字列の方が扱いやすいので，この方法を採りました。
- 90 行，100 行が主処理です。Call MakePlane1 は第 1 面の作成，Call MakePlane2 は第 2 面の作成です。 1 面と 2 面では構成が異なりますので，別になっています。
- 120 行と 130 行が文字列 Dt$をバイト配列 A の値に変換します。2 文字で 1 バイトですから，$\mathrm{Len}(\mathrm{Dt\$})/2$ 個の配列に格納されます。ToArray(Dt$,n,A()) はバイト文字列をバイト数値配列への変換の Sub です。
- 150 行と 160 行が WriteAllBytes で書き込みを行っています。

□

例 10.3 (JISX0213 一覧の作成 2：一覧作成 Sub).

```
180
190 Sub MakePlane1
200 CR$="0D0A"
210 For j=1 to 94
220    For i=1 to 94
230       Tmp1$=Right$("0"+hex$(S1(1,j,i)),2)
240       Tmp2$=Right$("0"+hex$(S2(j,i)),2)
250       Dt$=Dt$+Tmp1$+Tmp2$
260    Next i
270    Dt$=Dt$+CR$
280 Next j
290 End Sub
```

```
300
310 Sub MakePlane2
320    Dim ROWS(26)
330    CR$="0D0A"
340    ROWData$="0103040508121314157879808182838485868788899091929394"
350    For i=1 to 26
360       Rows(i)=Val(Mid$(ROWData$,2*i-1,2))
370    Next i
380    For i=1 to 26
390       For j=1 to 94
400          Tmp1$=Right$("0"+hex$(S1(2,Rows(i),j)),2)
410          Tmp2$=Right$("0"+hex$(S2(Rows(i),j)),2)
420          Dt$=Dt$+Tmp1$+Tmp2$
430       Next j
440       Dt$=Dt$+CR$
450    Next i
460 End Sub
470
480
```

説明.

この部分は，MakePlane1 と MakePlane2 の部分です。

- 190 行〜290 行は MakePlane1：第 1 面の作成です。第 1 面は，一部の例外を除いて 94×94 の表なので，位置を表す関数 S1 と S2 を使って，２重の For 文で記述しています。Tmp1$が第 1 バイト，Tmp2$が第２バイトを示します。Hex$を使って文字列として格納しています。CR$は改行を示す文字列です。
- 310 行〜460 行は MakePlane2：第２面の作成です。第２面は，実際には２６区しかありませんので，その処理を行っています。340 行の ROWData$は実際に使われる区を表す文字列です。最初から２文字で使われる区の番号（1,3,4,5,8,12〜15,78〜94）が列挙されています。
- 380 行〜450 行では，ROWData を使って，第 1 面と同様に，第２面を作成しています。

□

例 10.4 (JISX0213 一覧の作成 3：補助関数，Sub).

```
490
500 Function S1(m,k,t)
510    Result = 0
520    Select Case m
530       Case 1
540          Select Case k
550             Case 1  To 62 : result = (k + &h101) \ 2
560             Case 63 To 94 : result = (k + &h181) \ 2
570             Case Else : Print "Error!"
580          End Select
590       Case 2
600          Select Case k
610             Case 1, 3,4,5,8,12,13,14,15
620                result = ((k + &h1DF) \ 2) - ((k \ 8)*3)
630             Case 78 To 94
640                result = (k + &h19B) \ 2
650             Case Else : Print "Error!";k,t
660          End Select
670       Case Else
680          Print "Error ! m>2"
690    End Select
700    S1 = Result
```

```
710 End Function
720
730 Function S2(k,t)
740    Result = 0
750    If (k mod 2)=1 then
760       Select Case t
770          Case 1  To 63 : Result = t + &h3F
780          Case 64 To 94 : Result = t + &h40
790       End Select
800    ElseIf (k mod 2)=0 then
810       Result = t + &h9E
820    Else
830       Print "Error";k,t
840    End If
850    S2 = Result
860 End Function
870
880
890 Sub ToArray(Dt$,n,A())
900    For i=1 to n
910       A(i) = Val("&h"+Mid$(Dt$,(2*i-1),2))
920    Next i
930 End Sub
```

説明.

この部分は，Sub と主プログラムで使用される関数の定義部分です。16 進数を表示するのに”&h” 接頭語を使います。

- 500 行〜860 行は S1 の定義です。式は JIS X 0213 の JIS_2004 での対応の内容をそのまま関数式に書いたものです。S1 は第 1 面と第 2 面で定義が異なりますから，少し複雑です。
- 730 行〜710 行は S2 の定義です。これも JIS X 0213 の JIS_2004 での対応の内容をそのまま関数式に書いたものです。
- 810 行〜930 行は，バイト文字列をバイト数値配列変数に変換する Sub です。

□

10.2 バイナリファイルの読み込み

バイナリファイルの読み込みは，関数 ReadAllBytes を使います。

ReadAllBytes

ReadAllBytes は 指定したファイルの内容の指定したバイト配列 BArray() に格納します。

使い方

```
BArray()$=ReadAllBytes(FileName)
```

ReadAllBytes は指定したファイル FileName の各バイトをバイト配列 BArray() に格納します。格納するバイト配列変数 BArray は予め宣言されていなければなりません。読み込んだ行数は返す配列変数の 0 番目に文字列表現で格納されます。ファイルが宣言された以上の行数を持つ場合は，0 番目には ”-1” が格納されます。この場合，宣言の上限数のところまで読み込みます。

■簡単な例

簡単な使用例をあげます。次の例は，上で作成した Ab.txt を読み込むものですが，少しの変更でどのようなファイルでも読み込めます。

例 10.5 (ファイルの読み込み).

```
10 ChDir GetProgramDir
20 Dim BArray(1000)
30
40 BArray() = ReadAllBytes("Ab.txt")
50
60 For i=1 to BArray(0)
70    Print BArray(i)
80 Next i
90 End
```

説明.

- まずこのプログラムを適当な名前，例えば WriteAllBytes1.tbt として書き込みの例でのフォルダー，WABTest に保存します。GetProgramDir は保存されたプログラムでの実行が必須です。
- 10 行で WABTest がカレントディレクトリになります。
- 20 行は読み込むバイト列を格納する数値配列変数 BArray の宣言です。ここでは，例えば 1000 と宣言しましが，大きなファイルを読み込むときには大きな数として宣言します。変数名は別なものでも構いません。
- 40 行で BArray に，上の例で作成した"Ab.txt"の内容を読み込みます。何バイト読み込んだかは，BArray(0) に格納されます。
- 60 行〜80 行で読み込んだ内容を表示しています。60 行の For 文は，BArray(0) まで表示することを指定しています。

□

■ BOM 判定

もう一つ例をあげましょう。次の例は少しだけ興味があるかもしれません[*39]。

ユニコードのテキストファイルには，BOM と言う記号が先頭に付加されているものがあります。この BOM によって，どの型のユニコードであるか指定するものです[*40]。BOM はファイルの先頭高々４バイトの値で示されます。具体的には次で定められています。

BOM

- UTF-8: 0xEF 0xBB 0xBF
- UTF-16:BE 0xFE 0xFF
- UTF-16:LE 0xFE 0xFF
- UTF-32:BE 0x00 0x00 0xFE 0xFF
- UTF-32:LE 0xFF 0xFE 0x00 0x0

これらの値を使うと BOM 付のユニコードファイルを判定することができます。

次がそのプログラムです。少し長いので，分割して説明します。このプログラム全体は，同梱の ¥Samples¥Unicode フォルダーに CheckBOM.tbt としてあります。

例 10.6 (BOM 判定：主プログラム).

```
010 ' tbasic 1.6 以上
020 ' Unicode テキストが BOM をもつかどうか判定
030
040 Public A(10)
050 ChDir GetProgramDir
060
070 FName$ = SelectOpenFile
080 Print getFileEncodingName(FName$):' 確認用
090 A()=ReadAllBytes(FName$)
100
110 If IsUTF8 then
120    Print "BOM 付 UTF8"
130 ElseIF IsUTF16BE then
140    Print "BOM 付 UTF16 Big Endian"
150 ElseIF IsUTF16LE then
160    Print "BOM 付 UTF16 Little Endian"
170 ElseIF IsUTF32BE then
180    Print "BOM 付 UTF32 Big Endian"
190 ElseIF IsUTF32LE then
200    Print "BOM 付 UTF32 Little Endian"
210 Else
220    Print "BOM なしファイル"
230 End If
240 End
```

[*39] 実は，tbasic の内蔵関数 getFileEncodingName を使うと更に詳しい情報が得られますので，この以下のプログラムの実用的意味は余りありません。むしろ，getFileEncodingName がどのようにして作られているかの例示の意味があります。実際，getFileEncodingName はこのようなことを体系的に大がかりに行って実現しています。

[*40] BOM についての詳しい説明は，別文書「ユニコードへ」または，ウキペディア等を参照してください。

説明.

この部分は，主プログラムです。

- 40 行は読み込み用のバイト数値配列の宣言です。Function で使われるので，Public 宣言をしています。BOM は高々４バイトなので，４個の宣言でも良いのですが，10 を宣言しています。
- 70 行は判定するファイルの読み込みです。80 行は，tbasic の内蔵関数 getFileEncodingName を呼んで，ファイルの形式の確認をしています。以下のプログラムが正しく動作することの確認です。90 行は対象ファイルの読み込みです。
- 110 行〜230 行がこのプログラムの主要部です。それぞれ，以下に定義されるユーザー定義関数 IsUTF8，IsUTF16BE，IsUTF16LE，IsUTF32BE，IsUTF32LE を使って，ファイルの BOM を判定しています。

□

例 10.7 (BOM 判定：関数定義).

```
250
260 Function IsUTF8 as Boolean
270    If ((A(1)=&hEF) and (A(2)= &hBB) and (A(3)= &hBF)) then
280       IsUTF8 = True
290    Else
300       IsUTF8 = False
310    End If
320 End Function
330
340 Function IsUTF16BE as Boolean
350    If ((A(1)=&hFE) and (A(2)= &hFF)) then
360       IsUTF16BE = True
370    Else
380       IsUTF16BE = False
390    End If
400 End Function
410
420 Function IsUTF16LE as Boolean
430    If ((A(1)=&hFF) and (A(2)= &hFE)) then
440       IsUTF16LE = True
450    Else
460       IsUTF16LE = False
470    End If
480 End Function
490
500 Function IsUTF32BE as Boolean
510    If ((A(1)=&h00) and (A(2)= &h00) and (A(3)= &hFE) and (A(3)= &hFF)) then
520       IsUTF32BE  = True
530    Else
540       IsUTF32BE  = False
550    End If
560 End Function
570
580 Function IsUTF32LE as Boolean
590    If ((A(1)=&hFF) and (A(2)= &hFE) and (A(3)= &h00) and (A(3)= &h00)) then
600       IsUTF32LE  = True
610    Else
620       IsUTF32LE  = False
630    End If
640 End Function
```

説明.

この部分は，主プログラムで使用する Function の定義部分です。

- 260 行～320 行は IsUTF8 の定義です。真偽を返す関数なので，as Boolean 宣言をしています。270 行が BOM 付 UTF-8 かどうかの判定をしています。最初の３バイトが 0xEF 0xBB 0xBF であるか判定します[*41]。先頭３バイトが 0xEF 0xBB 0xBF なら，True を返し，そうでなければ，False を返します。
- 340 行以下，同様に，IsUTF16BE，IsUTF16LE，IsUTF32BE，IsUTF32LE について定義しています。

□

[*41] 16 進数表示なので，接頭語 &h を使っています。

第 III 部
付録

11　従来型ファイル処理

第Ⅱ部ではファイルの読み書きの簡易処理について説明しました。ここでは，伝統的な幾分旧式なファイル処理法について説明します。

これから新たにプログラムを作成する場合は，第Ⅱ部での方法を推奨します。しかし，ここで説明する方法は以下の状況で必要になることがあるかも知れません。

(1) 以前作ったプログラムを修正し，利用したい場合。
(2) 膨大なファイルを取り扱い，前節の方法ではメモリー不足になり，実行できない。

実際に必要になるのは (1) の場合が殆どでしょう。現在のコンピューターのメモリはギガの単位ですが，我々が使うテキストファイルの大きさには膨大と思われるものでもメガ単位です。余程大きなファイルでなければ十分にメモリーに収まるでしょう[*42]。

11.1　ファイルの Open

まずファイルの Open の説明をします。

ファイルを tbasic で処理する場合，**ファイル番号**というものを使います。即ち，

ファイル処理はファイル番号を通して行う。

となります。ファイル番号は内部のバッファに対応しています。外部のファイルとコンピューター内部との仲立ちをする通路のようなものです。tbasic の場合ファイル番号は 1〜8 で，同時に 8 個のファイル処理が可能です。少ないと思うかもしれませんが，ファイル番号は色々なファイルに割り当てることができますから，これでほぼ十分です。

この仲立ちとなるファイル番号（バッファ）と実際のファイルとの割り当てをすることで，ファイルを操作できます。

この割り当てをするのが open 文です。open 文は次の 3 種類の使い方があります。

Open

指定したファイルをファイル番号に割り当てます。

使い方

(1) `Open ファイル名 For Output As #ファイル番号`
(2) `Open ファイル名 For Input As #ファイル番号`
(3) `Open ファイル名 For Append As #ファイル番号`

[*42] 例えば，夏目漱石の「坊ちゃん」の青空文庫版のテキストファイル（ルビ付き）の大きさは，2900 余行，200K 余バイトで十分メモリーに収まります。

ここでファイル名は，実際にディスクに保存されているもしくは保存しようとしているファイルの名前です。パスで指定します。シナリオ 1 での状況なら，単にファイルの名前になります。またファイル番号は 1 から 8 までの数値です。

(1) は書き込みのファイルとして使う場合，
(2) は読み込み用のファイルとして使う場合，
(3) は追加書き込みのファイルとして使う場合です。

例をあげましょう。いずれもシナリオ 1 の状況を想定しています。

例 11.1 ((1) 書き込み用 Open).

```
Open "Test.txt" For Output As #1
```

これは「Test.txt というファイルを書き込み用として用意し，ファイル番号 1 を割り当てる。」を意味します。この文を実行すると書き込み用として新しい Test.txt 言う空のファイルが作成されます。ではもし Test.txt というファイルが既にあった場合はどうなるでしょうか。この場合は

既にあったファイルは消されて，新しい空のファイルが作られます。

ですから，Output 用のファイル名は決して間違わないようにして下さい。
この文を実行するとファイル番号 1 を使って Test.txt にデータを書き込むことが可能になります。

例 11.2 ((2) 読み込み用 Open).

```
Open "Test.txt" For Input As #2
```

これは「Test.txt というファイルを読み込み用として，ファイル番号 2 を割り当てる。」を意味します。この文を実行することで，以後ファイル番号 2 を使って Test.txt の中身を読み出すことができます。

例 11.3 ((3) 追加書き込み用 Open).

```
Open "Test.txt" For Append As #3
```

これは「Test.txt というファイルを追加書き込み用として，ファイル番号 3 を割り当てる。」ということを意味します。この宣言を行うことで，以後ファイル番号 3 を使って Test.txt にデータを追加書き込みができます。

次にこれらの処理を具体例で説明をします。まず，ファイルへの書き込み方法です。

11.2 ファイルへの書き込み：Output

書き込み用に Open したファイルはファイル番号を使って書き込むことができます。
書き込み用ファイルは最初は常に空のファイルです。

ファイルに書き込むのは Print # 文を使います。

使い方は Print 文と同様です。Print 文を使って実行画面に表示するように，Print #文を使ってファイルに書き込むことができます。

例をあげましょう。次のプログラムは 数表ファイルを作成しています。

例 11.4.

```
ChDir GetProgramDir
Open "NTable.txt" For Output As #1
For i=1 To 10
   Print #1,i, i*i, Sqr(i)
Next i
Close
```

このプログラムを実行すると，

```
1                    1                    1
2                    4                    1.41421356237309505
3                    9                    1.73205080756887729
4                    16                   2
5                    25                   2.2360679774997897
6                    36                   2.4494897427831781
7                    49                   2.64575131106459059
8                    64                   2.8284271247461901
9                    81                   3
10                   100                  3.16227766016837933
```

の内容を持つ NTable.txt が作成されます[*43]。そして，プログラムを保存したディレクトリにできた，NTable.txt をダブルクリックして内容を確認しましょう[*44]。

同様に

例 11.5.

```
ChDir GetProgramDir
Open "NTable2.txt" For Output As #1
For i=1 To 10
   Print #1,i;",";Tab(10); i*i;",";Tab(20); Sqr(i)
Next i
Close
```

を実行すると，

```
1,         1,         1
2,         4,         1.41421356237309505
3,         9,         1.73205080756887729
4,         16,        2
5,         25,        2.2360679774997897
6,         36,        2.4494897427831781
7,         49,        2.64575131106459059
8,         64,        2.8284271247461901
9,         81,        3
10,        100,       3.16227766016837933
```

の内容を持つ NTable2.txt が作成されます。

上の２つの例は主に数値を出力するものでした[*45]。Print #文は文字列も Print と同様に出力することが

[*43] 以前のように，このプログラムは一度保存してから実行してください。

[*44] NTable.txt をダブルクリックして開いたとき，開いたエディターのフォントが等幅フォント（例えば MS ゴシック）でないと表示が乱れる場合があります。

[*45] 実は，文字カンマ' を出力しています

できます。

次の例は，少し技巧的ですが

```
1 と 1^1 の"和の，計算"，" 1+ 1= 2"
2 と 2^2 の"和の，計算"，" 2+ 4= 6"
3 と 3^3 の"和の，計算"，" 3+ 27= 30"
4 と 4^4 の"和の，計算"，" 4+ 256= 260"
5 と 5^5 の"和の，計算"，" 5+ 3125= 3130"
```

を出力することを考えてみます。この場合，数値と文字列が混在しています。特に，表示文字の中にダブルコーテーション"があります。ダブルコーテーション"は BASIC では特別な意味があるので，この文字の出力には，"の代わりに Chr$(32) を使う必要があります。また，指数数字の空白がないこと，つまり 2^ 2 ではなく，2^2 のようになっていることに注意しましょう。この出力は複雑ですが，次のようにして実現できます。

例 11.6.

```
ChDir GetProgramDir
Open "SNTable.txt" For Output As #1
For i=1 To 5
   Print #1,i;" と";i;"^";Trim$(Str$(i));" の"+Chr$(34)+"和の，計算";Chr$(34);",";
   Print #1,Tab(25);Chr$(34);i;"+";i^i;"=";i+i^i;Chr$(34)
Next i
Close
```

ここでは Print #を 2 行で書いていますが，これは Print #1 以下が長くなるためです。1 行目の Print #1 の行末がセミコロン; で終わっていることに注意してください。Print 文と同様に，セミコロンで終わった行は改行が無く，次のデータがその後にすぐ続きます。

次にファイルの読み込みについて説明します。

11.3 ファイルの読み込み：Input

読み込み用に Open したファイルはファイル番号を使って読み込むことができます。

tbasic で扱うファイルは順編成テキストファイルでした。読み込みは常にファイルの先頭から最後に向かって順次読み込みます。そして，次のように進みます。

- Open した直後の読み込み位置はファイルの先頭です。
- 読み込み位置は一つデータを読み込むたびに，読み込みデータの位置は進みます。
- 次の読み込みデータの位置はその直前に読み込んだデータの次になります。
- 読み込みを繰り返すことによって，全てのデータを読み込むことができます。

しかし，全て読み込んで，読み込むデータが無くなってしまった時，更に読み込もうとするとエラーになり，プログラムが停止してしまいます。

そのため

> 読み込む前に必ず
> データが残っているか確認をする必要があります。

この確認をするための関数が Eof 関数です。

Eof 関数

Eof は読み込み位置がファイルの最後かどうかを返す関数です。

Eof(ファイル番号) は読み込み位置がファイルの最後のとき，True，そうでないとき False を返します。

従って Eof が真でないとき，読み込むことができます。これを使ってデータ読み込みを書いてみると，

```
Open "Test.txt" For Input As #1
If not Eof(1) Then
   データの読み込み
End If
```

となります。これは一回だけの読み込みです。連続的に読み込む場合は，次のようにします。

```
Open "Test.txt" For Input As #1
While not Eof(1)
    データの読み込み
Wend
```

これはデータがある限り読み込むという文になっています。

■ファイル読み込み用コマンド

読み込み用のコマンドは Input #ファイル番号と Line Input #ファイル番号です。

ファイル入力命令

Input # はデータの区切りを，空白（数値変数の場合），カンマ，改行として，読み込みます。

Line Input # は改行をデータの区切りとして読み込みます。

■ Input # での数値の読み込み

Input #

Input # 文で数値変数を指定すると，ファイルから数値を読み取ることができます。

例をあげましょう。いずれもシナリオ 1 の状況を想定しています。

例 11.7.

```
ChDir GetProgramDir
Open "NTable2.txt" For Input As #1
While not Eof(1)
   Input #1, N
   Print N
Wend
End
```

このプログラムは NTable2.txt の内容を数値とみて，順次読み込みそれを表示します。NTable2.txt の内容が前の例で作ったもの

```
1,        1,        1
2,        4,        1.41421356237309505
3,        9,        1.73205080756887729
4,        16,       2
5,        25,       2.2360679774997897
6,        36,       2.4494897427831781
7,        49,       2.64575131106459059
8,        64,       2.8284271247461901
9,        81,       3
10,       100,      3.16227766016837933
```

であったとすると，上のプログラムを実行すると

```
 1
 1
 1
 2
 4
 1.41421356237309505
 3
 9
 1.73205080756887729

 （この部分省略）

 9
 81
 3
 10
 100
 3.16227766016837933
```

と実行画面に表示されます。

■ Input # での文字列の読み込み

Input #

Input # 文で文字列変数を指定すると，ファイルから文字列を読み取ることができます。

例をあげましょう。

例 11.8.

```
ChDir GetProgramDir
Open "SNTable.txt" For Input As #1
While not Eof(1)
   Input #1, S$
   Print S$
Wend
End
```

このプログラムは先ほど作成した SNTable.txt の内容を文字列とみて，順次読み込みそれを表示しています。SNTable.txt の内容は

```
1 と 1^1 の"和の，計算"，" 1+ 1= 2"
2 と 2^2 の"和の，計算"，" 2+ 4= 6"
3 と 3^3 の"和の，計算"，" 3+ 27= 30"
4 と 4^4 の"和の，計算"，" 4+ 256= 260"
5 と 5^5 の"和の，計算"，" 5+ 3125= 3130"
```

でした。このとき，上のプログラムを実行するとどうのような出力が得られるでしょうか。実際に実行してみると，

```
1 と 1^1 の"和の
計算"
1+ 1= 2
2 と 2^2 の"和の
計算"
2+ 4= 6
3 と 3^3 の"和の
計算"
3+ 27= 30
4 と 4^4 の"和の
計算"
4+ 256= 260
5 と 5^5 の"和の
計算"
5+ 3125= 3130
```

と画面に表示されます。文字列での Input # はカンマ，改行を区切りとします。先頭と末尾の空白は無視されます。読み込み時に空白は区切り文字ではありません。

また，” がデータ区切りの先頭にある場合，” で囲まれた文字列は一つのデータとしてみます。この場合，” はデータの区切り文字で，読み込まれません。” が先頭にない場合は，” はそのまま読み込まれます。

このように文字列での Input #は多少複雑な動作をします。

■ LineInput # での文字列の読み込み

LineInput #

LineInput # 文で文字列変数を指定すると，ファイルから文字列を 1 行ずつ一括して読み込みます。

例 11.9.

```
ChDir GetProgramDir
Open "SNTable.txt" For Input As #1
While not Eof(1)
   Line Input #1, L$
   Print L$
Wend
End
```

このプログラムは，先程の SNTable.txt の中身を 1 行づつ，読み込みそれを表示しています。実行すると

```
1 と 1^1 の"和の，計算"，" 1+ 1= 2"
2 と 2^2 の"和の，計算"，" 2+ 4= 6"
```

```
 3 と 3^3 の"和の，計算"， " 3+ 27= 30"
 4 と 4^4 の"和の，計算"， " 4+ 256= 260"
 5 と 5^5 の"和の，計算"， " 5+ 3125= 3130"
```

と画面に表示されます。つまり，テキストの内容をそのまま読み込みます。

11.4 ファイルへの追加書き込み：Append

次に，ファイルへの追加書き込みの説明をします。

追加書き込み用に Open したファイルはファイル番号を使って追加書き込みができます。

> 追加書き込み用に Open したファイルは，
> そのファイルの最後に追加書き込みされる以外，
> 普通の書き込む場合と全く同じです。

次のプログラムは上述のファイル NTable.txt に追加書き込みをしています。プログラムを NTable.txt のあるディレクトリに保存します。そして

例 11.10.

```
ChDir GetProgramDir
Open "NTable2.txt" For Append As #1
For i=11 to 20
   Print #1,i;",";Tab(10); i*i;",";Tab(20); sqr(i)
Next i
Close
End
```

を実行すると，NTable2.txt の内容は

```
1,          1,          1
2,          4,          1.41421356237309505
3,          9,          1.73205080756887729
4,          16,         2
5,          25,         2.2360679774997897
6,          36,         2.4494897427831781
7,          49,         2.64575131106459059
8,          64,         2.8284271247461901
9,          81,         3
10,         100,        3.16227766016837933
11,         121,        3.31662479035539985
12,         144,        3.46410161513775459
13,         169,        3.60555127546398929
14,         196,        3.74165738677394139
15,         225,        3.87298334620741689
16,         256,        4
17,         289,        4.12310562561766055
18,         324,        4.24264068711928515
19,         361,        4.35889894354067355
20,         400,        4.47213595499957939
```

となります。

11.5　ファイルの Close

Open したファイルは，そのファイルの使用が終わったら Close する必要があります。ファイルをクローズするとは外部ファイルとコンピューター内部の仲立ち（通路）を外すことです。これを行う理由は次のようなことからです。

- 書き込み用ファイルを Close すると，書き込みバッファに残っているデータを確実にファイルに出力します。
- 読み込み用ファイルを Close すると，再びオープンすることが可能になり，ファイルの先頭から再度読み込みが可能になります。
- ファイルを Close することで，使用しているファイル番号を解放し，別のファイルに割り当てることを可能にします。
- ファイルを Close することで，外部ファイルとのアクセスを絶ち，外部ファイルの破損を防ぎます。

Close の仕方は簡単です。

```
Close #ファイル番号
```

です。例えば，ファイル番号 1 のファイルをクローズするには

```
Close #1
```

とすればよいだけです。

```
Close
```

とすると全てのファイルが Close されます。

「Tiny Basic for Windows ファイル操作編」更新記録

- （2023 年 04 月版）Ver. 1.6 用に文書内容を一部追加・修正。文書の構成を変更。バイナリファイルの取り扱いの項を追加。
- （2020 年 06 月版）pdf 文書の作成方法を変更。プログラムの書式を統一・微調整。従来型ファイル処理の項を一部修正・追加。演習問題の追加。その他の文書内容は 2014 年 08 月版とほぼ同じ。
- （2014 年 08 月版）初版公開