プログラミング及び実習Ⅰ・第8回

2024年8月1日

今回の内容

8.1 いろいろな演算子

プログラム中で、1つあるいは複数の値を組み合せて、別の値を計算する作用をもつものを**演算子**と呼びます[1]。+、−、*、/、% といった四則演算子がその代表ですし、式の先頭につけることのできる + や − といった符号も演算子の仲間です。C には他にもいろいろとな演算子が用意されています。今回は、知っておくと便利な演算子をいくつか紹介します。

メモ

代入演算子　変数へ値を代入(格納)するには、

```
変数 = 式;
```

という形の文を実行すればよいということを学びました。たとえば、

```
x = y * y;
```

を実行することで、変数 y に格納された値の2乗を計算して、その計算結果を変数 x に格納することができます。この代入という作業を行う文は、特別な形をしているように見えますが、実は、単に

```
式;
```

という形の文の1例でしかありません。つまり「変数 = 式」という形をしたものも「式」の一種なのです。上の例では、y や y * y がそれぞれ1つの式であって、それらを計算した結果の値というものがあるのと同じように、x = y * y 全体も1つの式になり、やはりそれを計算した結果の値を持ちます。* が、その両側の式の値から、その積を計算して返す演算子として働くのと同じように、= は左辺の変数名と右辺の式の値を使って、代入という作業を行うと同時に右辺の値をそのまま返す演算子として働きます。ですから、たとえば

```
x = (y = 1+2+3);
```

[1]「演算子」という言葉は、すでに「情報基礎」の最終回で説明しました

のような文を書くことができます。この例の場合、まず 1+2+3 が計算されて、結果の 6 が変数 y に格納されるとともに、y = 1+2+3 という式の計算結果が 6 となり、その 6 が変数 x にも格納されることになります。実際には、上の ( ) は省略することができて、

```
x = y = 1+2+3;
```

と書けば十分です。こう書いても (x = y) = 1+2+3; の意味にはなりません。これは、= という演算子は、より右にあるものが優先されて (強く) 結び付くからです。逆に、+、−、*、/、% といった四則演算の演算子は、より左にあるものが優先されて (強く) 結び付くことに注意してください。また、= という演算子は、四則演算の演算子より結び付きの優先度が低い (より弱く結び付く) ので

```
x = y = 1+2+3;
```

と書いても

```
(x = (y = 1))+2+3;
```

のような意味にはなりません。

メモ

代入演算子の仲間　C には変数に格納されている値を変更するような演算子が他にもいろいろと用意されています。知っておくと便利な代入演算子の仲間を表にまとめてみました。

演算子名	表記	使用例	計算による効果	式の値 (計算結果)
加算代入	+=	x += a	x = x + a と同じ	変数 x の新しい値
減算代入	-=	x -= a	x = x - a と同じ	変数 x の新しい値
乗算代入	*=	x *= a	x = x * a と同じ	変数 x の新しい値
除算代入	/=	x /= a	x = x / a と同じ	変数 x の新しい値
剰余代入	%=	x %= a	x = x % a と同じ	変数 x の新しい値
前置増分	++	++ x	x = x + 1 と同じ	変数 x の新しい値
前置減分	--	-- x	x = x - 1 と同じ	変数 x の新しい値
後置増分	++	x ++	x = x + 1 と同じ	変数 x の古い値
後置減分	--	x --	x = x - 1 と同じ	変数 x の古い値

増分演算子 ++ や減分演算子 -- は変数の前に置くか後に置くかによって、その意味が違っています。たとえば、

```
x = 1;
tPrintf("%d\n", x++);
tPrintf("%d\n", x);
```

を実行すると、1 と 2 が表示されますが、

```
x = 1;
tPrintf("%d\n", ++x);
tPrintf("%d\n", x);
```

の場合、2 と 2 が表示されます。x++ も ++x も、変数 x に格納されている値を 1 増やすことには変りはありませんが、x++ という式の値は増やす前の x の値となるのに対して、++x は増やした後の x の値となります。

式中にこれらの演算子が現れた場合の結合の優先度は

(強い)　後置増減分 > 前置増減分や符号としての +- > 四則演算 > (四則演算付き) 代入　(弱い)

のようになっています。また、四則演算付き代入演算子も (普通の代入演算子と同じように) より右にあるものが優先されて (強く) 結び付きます。たとえば、

```
x += y *= - 3 + ++ z
```

は

```
x += (y *= ((- 3) + (++ z)))
```

と書いたのと同じ意味になります。演算子が結び付く優先順序や方向に自信がない場合は、適当に ( ) を書いておくと安心です。C の演算子に関する詳細については、付録の「C の演算子一覧」や参考書等を参照してください。

メモ

条件演算子　条件演算子は、式の中で場合分けの計算を可能にする 3 項演算子です。変数 x に、変数 y と z の最大値を代入するには、if 文による場合分けを行って、たとえば

```
if (y >= z)
    x = y;
else
    x = z;
```

のように書けばよいわけですが、これと同じことを条件演算子を使って、

```
x = (y >= z) ? y : z;
```

のような文で実現することもできます。一般に「条件式 ? 式$_1$: 式$_2$」のように書くと、「条件式」が成り立っているときには 式$_1$ が計算されてその値がこの式全体の値となり、成り立っていないときには 式$_2$ が計算されてその値が式全体の値となります。上の例では、プログラムの意味が理解しやすいように条件式を ( ) で囲んでいますが、条件演算子 ? : は、代入演算子よりも結合の優先度が高く、>= のような比較演算子よりも優先度が低いため、

```
x = y >= z ? y : z;
```

のように、( ) を省略して書いても同じ意味になります。

メモ

8.2 いろいろな制御文

if 文による場合分け (選択) や、while 文による繰り返し (反復) など、実行の流れを制御しているプログラムの構造を**制御構造**と言います。また、これら 2 つの文を含めて、break 文など、プログラムの実行の流れを制御するための文を一般に**制御文**と呼びます。C には、if 文や while 文、break 文の他にも、いろいろな制御文が用意されています。

メモ

for 文 C には、決った回数の繰り返しを行うために便利な制御構造として **for 文**と呼ばれる構文が用意されています。for 文は

for(初期設定式 ; 継続条件式 ; 再設定式) 文

という形の構文です。もちろん、複数の行に跨って書いても構いません。この for 文全体も 1 つの文となります。for 文は、

```
初期設定式 ;
while( 継続条件式 ) {
    文
    再設定式 ;
}
```

という while 文を使った繰り返しと同じように実行されます。たとえば、

```
sum = 0;
i = 1;
while (i <= n) {
    sum += i;
    i ++;
}
```

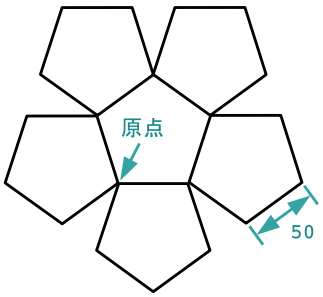
という while 文を使ったプログラムで、1 から n までの整数の和を変数 sum に格納することができますが、これと同じことをするプログラムを、for を使って

```
sum = 0;
for (i = 1; i <= n; i++)
    sum += i;
```

のように簡潔に書くことができます。また、

```
k = 0;
while (k < n) {
    i = 0 ;
    while (i < n) {
        tForward(50);
        tTurn(-360.0/n);
        i = i + 1;
    }
    tForward(50);
    tTurn(360.0/n);
    k = k + 1;
}
```

n = 5 の場合



というプログラムは、正 n 角形を (正 n 角形の回りに)n 個描くもの (k と k は整数値の変数) ですが、これと等価なプログラムを、for 文を使って

```
for (k = 0; k < n; k++) {
    for (i = 0; i < n; i++) {
        tForward(50);
        tTurn(-360.0/n);
    }
    tForward(50);
    tTurn(360.0/n);
}
```

のように書くこともできます。

for 文の初期設定式、継続条件式、再設定式は必要がなければ省略しても構いません (ただし、それでもこれら 3 つを区切っている ; は必要です)。継続条件式を省略すると、そこに常に成り立つような条件が書かれているのと等価になります。つまり、

```
for (;;) 文
```

は

```
while (1) 文
```

と等価になります。

前回紹介した break 文は、for 文や、次節で紹介する do 文による繰り返しを途中終了するためにも使うことができます。

メモ

do 文　do 文 は、

```
do 文 while( 継続条件式 );
```

という形の構文で、やはり繰り返しの処理を行うために用いられます。while というキーワード (予約語) が含まれていますが、これも do 文の構文の一部であって、決して while 文の始りではありません。

do 文 は、

```
文
while( 継続条件式 ) 文
```

という 2 つの文からなるプログラムと同じように実行されます。do 文 では、繰り返される「文」の部分が、無条件で必ず 1 回は実行されることに注意してください。

メモ

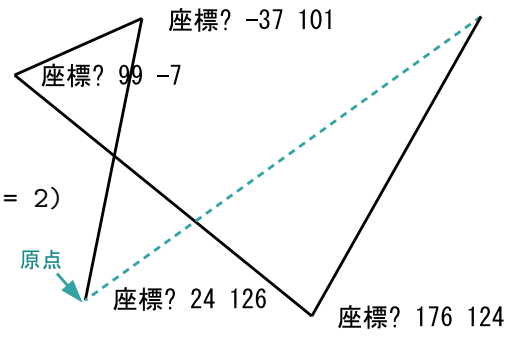
次の broute.c は、tScanf を使ってキーボードから x 座標と y 座標の値を読み込み、カメを指定された点に次々と移動させていくプログラムですが、原点からの距離が 200 を超える点に移動すると、この繰り返しを終了し、(ペンを上げて) 原点に戻るようにしています。

broute.c

```
#include <turtle.h>

main ()
{
    double x, y;

    do {
        tPrintf("座標? ");
        if (tScanf("%lf%lf", &x, &y) != 2)
            break;
        tMoveTo(x, y);
    } while (x*x + y*y <= 40000);
    tPenUp();
    tMoveTo(0, 0);
}
```



この do 文の部分は

```
x = y = 0;
while (x*x + y*y <= 40000) {
    tPrintf("座標? ");
    if (tScanf("%lf%lf", &x, &y) != 2)
        break;
    tMoveTo(x, y);
}
```

と書き換えても同じように動作しますが、`x = y = 0;` という文による変数の初期化や、`while` 文の条件式の最初のチェックは (本来は必要の無い) 無駄な作業になってしまいますので、`do` 文を使った方がより自然なプログラムになります。

C 言語には、この他にも `switch` 文や `continue` 文、`goto` 文といった構文があり、プログラムの実行の流れを制御することができます。詳細は参考書等を参照してください。

メモ

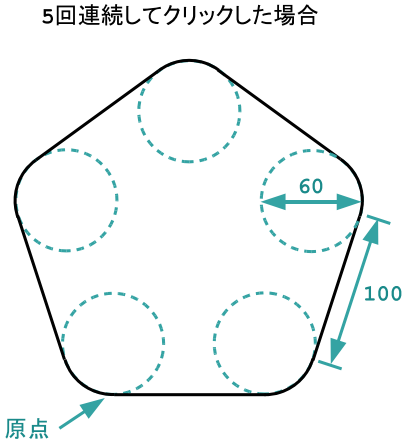
8.3 演習問題

1. 第 5 回の例題プログラム `nested.c` (第 5 回配布資料 5 ページ) と同じことをするプログラム `cnested.c` を、`while` 文の代りに `for` 文を使って書きなさい。また、`i = i+1` の代りに `i++` を使うなど、今回解説した演算子が (効果的に) 使えるところがあれば、できるだけ使うようにしなさい。

2. 前回の演習問題 `primes.c` (この資料の 13 ページ) と同じことをするプログラム `cprimes.c` を、`while` 文の代りに `for` 文を使って書き、コンパイル、実行し、正しく動作することを確認しなさい。ただし、`k = k+1` の代りに `k++` を使うなど、今回解説した演算子が (効果的に) 使えるところがあれば、できるだけ使うようにしなさい。

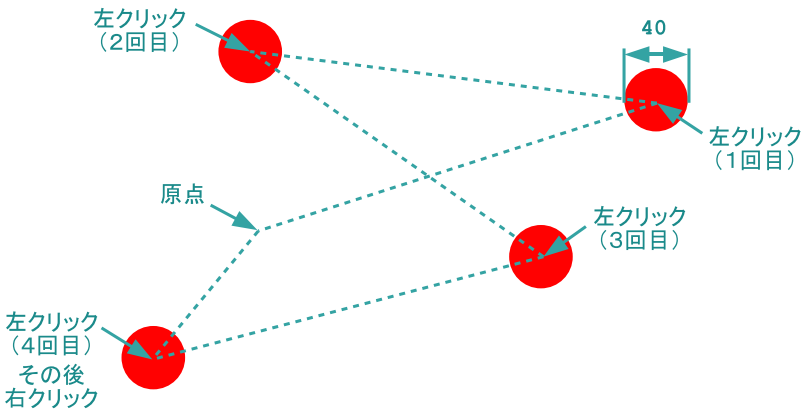
3. ウィンドウ内を連続して n 回クリックすると、下図のような角の丸い n 角形を描くプログラム `rpolygon.c` を作成し、コンパイル、実行して、正しく動作することを確認しなさい。ただし、`while` 文ではなく、`for` 文を使ったプログラムにしなさい。また、`i = i+1` の代りに `i++` を使うなど、今回解説した演算子が (効果的に) 使えるところがあれば、できるだけ使うようにしなさい。

この図形は、長さ 100 の線分と、半径 30 の円周の一部 (円弧) をつなぎ合わせたものとなっています。原点の位置は、このつなぎ目の 1 つになっています。



円周の一部 (円弧) は tArc という関数 (この科目のタートルグラフィックスの機能の 1 つ) を呼び出すことで描くことができます。この関数の使い方については、付録の「タートルグラフィックスの機能一覧」を参照してください。

4. ウィンドウ内を左クリックする度に、クリックされた位置を中心とする半径 20 の赤い円盤を描いていくプログラム dots.c を作成し、コンパイル、実行して、正しく動作することを確認しなさい。ただし、左ボタン以外でクリックされると、カメは仕事を止めて (ペンを上げて) 原点に戻るようにしなさい。



ヒント: 円盤 (内部が塗り潰された円) は tMark、tCircle、tFill の 3 つの関数の呼び出しを組み合わせることで描くことができます。また、塗り潰しの色を赤色に変更するためには、tSetColor を呼び出すようにしてください。たとえば

```
#include <turtle.h>

main ()
{
    tSetColor(1.0, 0.0, 0.0);
    tPenUp();
    tMoveTo(100, 100);
    tForward(50);
    tTurn(90);
    tMark();
    tCircle(50);
    tFill();
}
```

というプログラムは、(100, 100) を中心とする半径 50 の赤い円盤を描きます。ペンの上下は、関数 tFill による塗り潰しには影響しません。tMark を呼び出しているのは、続く tCircle の呼び出しで 1 周する円周の内部だけを塗り潰したいからです。tMark を呼び出しておかないと、原点からのカメの移動の軌跡すべての内部が塗り潰されてしまいます。これらの関数の詳細については、付録の「タートルグラフィックスの機能一覧」を参照してください。

8.4 今回の実習内容

1. プリントをもう一度読み返しましょう。6 ページの例題 broute.c については、ソースプログラムを作成し、コンパイル、実行して、正しく動作することを確認してください。プログ

ラムが完成したら「課題の提出と確認」の Web ページから提出してください。

2. 演習問題の 1 と 2 に取り組みましょう。それぞれのプログラムが完成したら、「課題の提出と確認」の Web ページからの提出してください。演習問題の 3 と 4 は自習用で、成績には影響ありません。
3. クイズに答えてください。前回までと同様に「課題の提出と確認」の Web ページで「第 8 回クイズ」を選択し、「送信」のボタンをクリックしてクイズに答えてください。

プログラミング及び実習 I ・第 8 回・終わり

8.5 付録: C の演算子一覧

優先順位	演算子名	表記	書式	計算の効果 (副作用)	式の値 (評価結果)
1(左)	関数呼び出し	`()`	f`(...)`	まず f を評価した後、引数を評価 (順序は不定) して、f を呼び出す	f の戻り値
	配列添字	`[]`	a`[`b`]`	a と b を評価 (順序は不定)	配列 a の添字 b の要素
	直接メンバ	`.`	a`.`m	a を評価	構造体/共用体 a のメンバ m
	間接メンバ	`->`	a`->`m	a を評価 — (`*`a)`.`m と同じ	アドレス a に格納されている構造体/共用体のメンバ m
	後置増分	`++`	x `++`	x `=` x `+ 1` と同じ	x の古い値
	後置減分	`--`	x `--`	x `=` x `- 1` と同じ	
2(右)	前置増分	`++`	`++` x	x `=` x `+ 1` と同じ	x の新しい値
	前置減分	`--`	`--` x	x `=` x `- 1` と同じ	
	記憶量	`sizeof`	`sizeof` x	x は評価されない	x の記憶に必要なバイト数
	アドレス	`&`	`&` x	x を左辺値として評価	x の記憶領域の先頭アドレス
	間接参照	`*`	`*` a	a を評価	アドレス a に格納されている値
	単項プラス	`+`	`+` a		a
	単項マイナス	`-`	`-` a		$-a$
	ビット反転	`~`	`~` a		a をビット毎に反転した値
	否定	`!`	`!` a		a が 0 なら 1、そうでなければ 0
3(右)	キャスト	(型指定)	(t) a		a の値を型 t に変換したもの
4(左)	乗算	`*`	a `*` b	a と b を評価 (順序は不定)	$a \times b$
	除算	`/`	a `/` b		a / b
	剰余	`%`	a `%` b		a を b で割った余り
5(左)	加算	`+`	a `+` b		$a + b$
	減算	`-`	a `-` b		$a - b$
6(左)	左シフト	`<<`	a `<<` b		a を左に b ビットシフトした値
	右シフト	`>>`	a `>>` b		a を右に b ビットシフトした値
7(左)	小なり	`<`	a `<` b		$a < b$ なら 1、そうでなければ 0
	以下	`<=`	a `<=` b		$a \leqq b$ なら 1、そうでなければ 0
	以上	`>=`	a `>=` b		$a \geqq b$ なら 1、そうでなければ 0
	大なり	`>`	a `>` b		$a > b$ なら 1、そうでなければ 0
8(左)	等値	`==`	a `==` b		$a = b$ なら 1、そうでなければ 0
	不等	`!=`	a `!=` b		$a \neq b$ なら 1、そうでなければ 0
9(左)	ビット積	`&`	a `&` b		a と b のビット毎の論理積
10(左)	ビット差	`^`	a `^` b		a と b のビット毎の排他的論理和
11(左)	ビット和	`|`	a `|` b		a と b のビット毎の論理和
12(左)	論理積	`&&`	a `&&` b	まず a を評価して、0 でないなら b を評価	$a = 0$ なら 0、そうでなければ b
13(左)	論理和	`||`	a `||` b	a を評価して、0 なら b を評価	$a \neq 0$ なら a、そうでなければ b
14(右)	条件	`? :`	c `?` a `:` b	条件 c を評価した後、a または b を評価	c が 0 なら b、そうでなければ a
15(右)	単純代入	`=`	x `=` a	左辺値として評価した x に a を評価した結果を代入 (x と a の評価順序は不定)	x の新しい値
	加算代入	`+=`	x `+=` a	x `=` x `+` a と同じ	
	減算代入	`-=`	x `-=` a	x `=` x `-` a と同じ	
	乗算代入	`*=`	x `*=` a	x `=` x `*` a と同じ	
	除算代入	`/=`	x `/=` a	x `=` x `/` a と同じ	
	剰余代入	`%=`	x `%=` a	x `=` x `%` a と同じ	
	左シフト代入	`<<=`	x `<<=` a	x `=` x `<<` a と同じ	
	右シフト代入	`>>=`	x `>>=` a	x `=` x `>>` a と同じ	
	ビット積代入	`&=`	x `&=` a	x `=` x `&` a と同じ	
	ビット差代入	`^=`	x `^=` a	x `=` x `^` a と同じ	
	ビット和代入	`|=`	x `|=` a	x `=` x `|` a と同じ	
16(左)	順次	`,`	a`,` b	a を評価した後、b を評価	b の値

8.6 付録: タートルグラフィックスの機能一覧

関数の呼び出し方	働き	戻り値
tForward(d)	実数値 d だけ前へ進む	進んだ距離 d (実数値)
tBackward(d)	実数値 d だけ後ろへ戻る	戻った距離 d (実数値)
tTurn(t)	反時計回りに t° だけ向きを変える (t は実数値)	x 軸の正の向きを 0° としたときのカメの新しい向き (実数値)
tMoveTo(x, y)	実数値の座標 (x, y) へ移動する	移動した距離 (実数値)
tPenUp()	ペンを上げる	変更前にペンを上げていれば 0、下げていれば 1 (整数値)
tPenDown()	ペンを下げる	
tGetClick()	マウスボタンがクリックされるのを待つ	クリックされたボタンの番号 (整数値 — 左=1、中=2、右=3)
tCheckClick(t)	マウスボタンがクリックされるのを最大 t 秒待つ (t は実数値)	クリックされたボタンの番号 (整数値 — クリックされなかった=0、左=1、中=2、右=3)
tClickButton()	特に何もしない	
tClickCount()	特に何もしない	連続してマウスクリックされた回数 (整数値)
tClickX()	特に何もしない	マウスクリックされた位置の x 座標 (整数値)
tClickY()	特に何もしない	マウスクリックされた位置の y 座標 (整数値)
tPrintf(f, ···)	出力書式文字列 f と残りの引数で指定された文字列を、カメの右側に表示する	なし
tScanf(f, ···)	キーボードから入力された数値等を入力書式文字列 f で解釈し、残りの引数で指定された変数等に格納する (変数の前に & が必要)	読み込めた数値等の個数 (整数値)
tSetColor(r, g, b)	光の 3 原色 (赤、緑、青) の各成分の強さを 0.0 ～ 1.0 の範囲の実数値 r、g、b でそれぞれ指定して、ペンや塗り潰し、文字列の色を変更する	変更前の色を表す整数値 (赤、緑、青の強さを、それぞれ 0 ～ 255 の範囲の整数 R、G、B で表したときの $65536R + 256G + B$)
tSetBackground (r, g, b)	tSetColor と同様の方法で、ウィンドウの背景色を変更する	
tFill()	tMark、または tFill や tFillAll を最後に呼び出した時点から現在までのカメの移動 (ペンを上げていてもよい) の軌跡の内部を塗り潰す[2]	なし
tFillAll()		
tMark()	tFill や tFillAll で使われる軌跡の始まりをカメの現在位置に設定する	なし
tCircle(r)	反時計回りに半径 r の円周上を一周する — 半径 r が負の場合は時計回りとなる (r は実数値)	移動した距離 (実数値)

[2]軌跡の内部が複数の領域に分割される場合、tFill は、外側のものから順に、塗り潰す領域と塗り潰さない領域を交互に設定するが、tFillAll はすべての領域を塗り潰す。

関数の呼び出し方	働き	戻り値
tArc(r, t)	反時計回りに半径 r の円周上を t° だけ移動する— 半径 r が負の場合は時計回りとなる (r、t は実数値)	移動した距離 (実数値)
tHeadTo(x, y)	実数値の座標 (x, y) の方向へ向きを変える	x 軸の正の向きを 0° としたときのカメの新しい向き (実数値)
tTurnTo(t)	x 軸の正の向き基準として、反時計回りに t° の方向へ向きを変える (t は実数値)	
tSetSpeed(s)	カメの移動速度を、毎秒 s ピクセルに設定する (初期値は毎秒 200 ピクセル) — s が 0.0 の場合は最高速度に設定する (s は実数値)	変更前の移動速度 (実数値)
tSetWidth(w)	カメが描く軌跡の線幅を w ピクセルに設定する (w は整数値)	変更前の線幅 (実数値)
tSetFontSize(s)	文字列の描画に使われる文字の大きさ (高さ) を s (整数値) ピクセルに設定する (s の初期値は 16)	変更前の文字の大きさ (整数値)
tHide()	カメを表示しないようにする (表示されていなくてもカメは仕事をすることができる)	変更する前に表示されていなければ 0、されていれば 1 (整数値)
tShow()	カメを表示する	
tSetSize(s)	カメの大きさを s ピクセルに設定する (初期値は 20 ピクセル)	変更前の大きさ (実数値)
tSetShellColor (r, g, b)	光の 3 原色 (赤、緑、青) の各成分の強さを 0.0 〜 1.0 の範囲の実数値 r、g、b でそれぞれ指定して、カメの甲羅の色を変更する	変更前の色を表す整数値 (赤、緑、青の強さを、それぞれ 0 〜 255 の範囲の整数 R、G、B で表したときの $65536R + 256G + B$)
tX()	特に何もしない	カメの現在位置の x 座標 (実数値)
tY()	特に何もしない	カメの現在位置の y 座標 (実数値)
tOrientation()	特に何もしない	x 軸の正の向きを 0° としたときのカメの現在の向き (実数値)
tCancel(n)	直前の n 個の描画処理 (tMark を含む) を取り消す	なし
tClear()	これまでの描画処理 (tMark を含む) をすべて取り消す	なし
tErase(n, m)	n 個前から m 個前までの描画処理で描かれた部分を消去する (tMark やカメの移動の軌跡は取り消されない)	なし
tPause(t)	プログラムの実行を t 秒間停止する (t は実数値)	なし
tMaximizeWindow()	ウィンドウを最大化する	なし
tNormalizeWindow()	ウィンドウを通常の大きさに戻す	なし
tResizeWindow(w, h)	ウィンドウの描画領域の幅を w ピクセルに、高さを h ピクセルにする (w、h は整数値)	なし