Sunaba 上級ガイド

平山 尚

2013 年 7 月 26 日 14 時 32 分 (GMT) バージョン

目次

第 1 章

まえがき

この文書は、書籍本体を読み終えられた人及び、書籍が対象とするよりもプログラミングに親しんでいる人向けに、Sunaba を解説するものである。

こちらの文書では、本体と比べると「普通」なプログラミング書籍に近づけてある。また、他のプログラミング言語経験がある人にとってわかりやすくするために、Sunaba の特徴である「日本語っぽい文法」を使わず、英語風文法のみで解説する。名前付きメモリ、部分プログラムなどの名前についても、日本語を用いない。

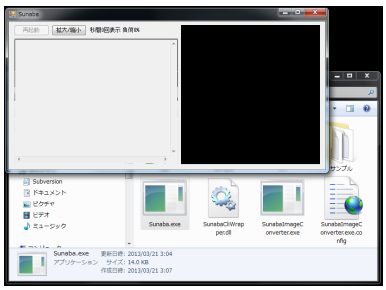
第 2 章

準備

まずは Sunaba での開発をするための準備をしよう。

Sunaba の配布サイトから最新版をダウンロードし、インストーラに従ってインストールする。

あとは、デスクトップに出来たショートカットからでもいいし、インストールされたフォルダにある Sunaba.exe を直接起動しても良い。ともかくも、Sunaba が起動する。



あとは、望みの場所に望みのソフトで Sunaba のソースコードを書き、このファイルを Sunaba のウィンドウにドラッグアンドドロップする。これで、Sunaba のプログラムが起動する。

試しに、Sunaba のインストールディレクトリ内にある「サンプル」フォルダ内にある「お手本.txt」を起動してみよう。

こんな感じに起動し、動かすことができる。

なお、ボタンにはなっていないが、pause キーを押すとプログラムを一時停止させることができる。もう一度押すと動き出す。正式にボタンを用意するかどうかは現在思案中である。

2.1　デバッガー

Sunaba.exe は、Sunaba プログラムを動かすことはできるが、それ以上の機能はない。

そこで、Sunaba プログラムがどう動いているかを調べたり、楽に間違いを直したりするための機能を追加したものを、SunabaDebugger.exe として用意してある。

といっても現状大した機能はなく、指定したメモリがいくつなのかを見たり、プログラムの外からメモリをいじったりする機能があるだけだ。いずれは、ここで名前付きメモリが見られたり、どの部分プログラムのどこにいるのかがわかったり、1 行づつプログラムを実行させたりすることができるようになるのかもしれない。

しかし、このプログラムはこの本の主要な読者にとっては無用なものである。機能に頼るよりも、推理を積み重ねて間違いを着実に直す技術を学んでほしいからだ。だから、本文中ではこのプログラムには触れていない。

第 3 章

ひとめぐり

では、Sunaba のプログラムを書いてみよう。

3.1 メモリの基本

まずは以下のものを打ちこんでみてほしい。

```
memory[65050] -> 999999
```

実行してみよう。

画面のほぼ真ん中に白い点が出るはずだ。つまり、これは白い点を描くプログラムである。

3.1.1 Sunaba の根本はメモリ

Sunaba は 2010 年代に作ったとは思えないほど原始的な言語であり、基本的にはメモリの番号を指定して直接値を書いたり読んだりする。

上のプログラムの意味は、「メモリの 65050 番が 999999 になる」である。

それぞれの番号のメモリはダイアルのようなものであり、「->」はそのダイアルを回すことを意味する記号である。左側の memory[65050] は、65050 番であることを示し、右側の 999999 はダイアルをいくつにセットするかを示す。左側の [] の中や、右側には、

```
memory[50000 + (memory[2] * 4)] -> (memory[1] * 12) + 6
```

のように、計算式を書くこともできる。

3.1.2 計算

Sunaba の計算は足し算、引き算、掛け算、割り算の 4 つに加えて、< や = などの比較もある。比較の類も足し算や掛け算と同様 2 つの数から 1 つの数を作ることに変わりは

ない。ただ、結果が「式が正しければ 1、そうでなければ 0」という「若干変わった計算」なだけだ。

計算は種類を問わず左から行うので、

```
x -> 1 + 2 * 3 / 4 = 5
```

と書くと、以下のように解釈される。

```
x -> (1 + 2) * 3 / 4 = 5
x -> (3 * 3) / 4 = 5
x -> (9 / 4) = 5
x -> (2 = 5)
x -> 0
```

「掛け算は優先」と学校で習ったかもしれないが忘れてほしい。

3.1.3　演算子

演算子は、「左右にある二つの数から一つの数を作るもの」である。その意味で、＋や-と、＜や＝の間に差はない。

演算子	＋	-	*	/
作用	加算	減算	乗算	除算
-7 と 5	-2	-12	35	-1

以下は条件が成立すれば 1、そうでなければ 0 を作る演算子である。

演算子	＜	＞	＜=	＞=	＝	!=
作用	左＜右	左＞右	左≦右	左≧右	左＝右	左≠右
-7 と 5	1	0	1	0	0	1

なお、「右にある一つの数から一つの数を作るもの」もある。マイナスだ。

```
a -> -b
```

とある時、b の左にある-は、引き算のマイナスではなく、符号反転演算子としてのマイナスである。しかしこれに関して混乱することはおそらくないだろう。

3.1.4　周辺機器の操作もメモリ経由

そして、マウスやキーボード、画面などの周辺機器は、メモリの特定の番号につながっている。この例では、65050 番が画面の中央あたりの画素につながっており、それを 999999 にすることで画素の色が白くなる。

どの番号が何につながっているのかは、例えば以下のようなものがある。

番号の範囲	説明
50000-50001	マウスカーソル座標 (順に X,Y)。
50002-50003	マウスボタン (順に左、右)
50004-50009	キーボード (順に上、下、左、右、スペース、エンター
60000-69999	画素の色を格納。

画面は縦横 100 画素の計 1 万画素から成り[*1]、以下のように並んでいる。

0	1	2	.....	97	98	99
100	101	102	.....	197	198	199
.	.	.		.	.	.
9900	9901	9902	.....	9997	9998	9999

これに 60000 を足したのがメモリの番号だ。だから、65050 番と言えば、左上から左に 50、下に 50 進んだところにある画素ということになり、ほぼ中央ということになる。

3.1.5 メモリの範囲

メモリは 0 番から 3 万番くらい[*2]は好きに使って良い。

```
memory[0] -> 4
memory[12345] -> memory[0]
```

というように、好きに番号を指定すれば良い。起動直後は全て 0 になっているが、「自分で値をセットするまでは読み出さない」という用心深い習慣を持つことは賢明だろう。

4 万番台は Sunaba が内部的に使っているため、使わないことをおすすめしておく。

そして、5 万番より上は各種周辺機器につながっており、さまざまな副作用がついてくるし、読み出し専用だったり、書き込み専用だったりもする。単なる計算には使えないと思って良い。

3.1.6 名前付きメモリ

```
x -> 4
```

のように memory 以外のものを左側に置くこともできる。「x」なるものが初めて出てきた場合、ここで 4 万番台のメモリで空いているものを探して、そこに「x」という名前をつける。これ以降は、この 4 万番台のどこかのメモリを「x」という名前で使えて便利

[*1] これは変えられる。

[*2] 正確な番号を知る方法もある

である。名前は何でも良いが、一部の記号やスペースは駄目である[*3]。

```
x -> 4
memory[0] -> x
```

というように書け、これは、

```
memory[1] -> 4
memory[0] -> memory[1]
```

のように書くよりも楽である。

3.2　実行する行を制御する

次は以下のようなプログラムを書いて動かしてみよう。

```
while 1
   x -> memory[50000]
   y -> memory[50001]
   if memory[50002] = 1
      memory[60000 + (y * 100) + x] -> 999999
```

マウスカーソルを黒い画面の上に持っていき、左クリックすると白い点が打たれる。そのままカーソルを動かせば、線が描かれる。つまり、これは簡易的なお絵描きソフトなわけである。

今度のプログラムにはマウスからの入力を見て動作を変えたり、ある範囲の行を繰り返したりする要素が入っている。

3.2.1　繰り返し

まず、最初の行に while がある。

```
while 1
```

while の右には式を書け、値が 0 でなければ繰り返し範囲を実行し、また while の行に戻ってくる。0 ならば実行せず、繰り返し範囲の下の行に移る。

繰り返し範囲は、while の次の行以降、字下げが while と同じになる前の行までである。

```
while 1
   memory[60000] -> 888888
   memory[60001] -> 888888
a -> 5
```

とあれば、「a -> 5」の a は while と同じインデントレベルなので、この行の前の 2 行が繰り返し範囲ということになる。

[*3] 0 から 9、A から Z、a から z、__、それにひらがま、漢字、カタカナ、というあたりは問題ないが、その他の文字についてはダメなものもある。

なお、「空白」となる文字には、半角スペース、全角スペース、タブの 3 種類があり、全角スペースは半角スペース 2 個に置き換えられる。混ぜないことをお勧めするが、固定幅フォントを使うエディタで書いていればおおよそ長さが合っていれば大丈夫ということになる。また、タブ幅は 8 を想定している。

さて、今の場合条件式が 1 で固定なので永遠に繰り返すが、もちろん式を書ける。

```
i -> 0
while i < 10
   memory[60000] -> 888888
   i -> i + 1
```

とすれば、i が 10 未満の間繰り返すことになる。これが Sunaba における基本的な繰り返しの使い方となる。

なお、繰り返し範囲で作った名前付きメモリは、繰り返し範囲の外からは使えない。

```
while 1
   a -> 5
b -> a
```

は、前もって a を作っていなければエラーになる。a は while の範囲を出たところで消えてなくなるからである。

3.2.2 条件実行

また、このプログラムには条件によって実行したりしなかったりする機能が使われている。

```
   if memory[50002] = 1
      memory[60000 + (y * 100) + x] -> 999999
```

という部分がそれだ。50002 番が 1 であれば、その下の行を実行して、画素を塗る。範囲は while と同様インデントで決まる。なお、if は while を用いれば実現できる機能であり、

```
   if 条件
      中身
```

は、

```
   flag -> 0
   while 条件 * (flag = 0)
      中身
      flag -> 1
```

と等価である。単に「初めて繰り返し範囲を実行した時に、条件が成り立たなくなるようにする」だけだ。これによって実行回数が 0 か 1 に限られる。if は実行回数が 0 か 1 に制限された while である。

3.2.3　外部入力取得

メモリの 50000 番台はマウスやキーボードなど外の機械につながっている。

```
if memory[50002] = 1
   memory[60000 + (y * 100) + x] -> 999999
```

というプログラムは、一見 50002 番にセットする部分がないために意味がないように見えるが、50002 番は左クリックされると 1 になり、されていなければ 0 になるメモリである。プログラムの知らない所で外の機械がダイアルを回しに来ると思えば良い。

このため、この if は、「左クリックされていれば」という意味になる。同様に、50000 番はマウスの x 座標、50001 番は y 座標を表す。

```
x -> memory[50000]
y -> memory[50001]
```

という具合だ。

```
   memory[60000 + (y * 100) + x] -> 999999
```

という行では、60000 に、y 座標の 100 倍を足し、さらに x 座標を足すことで、(x, y) に対応した画素のメモリ番号を作っており、マウスカーソルがある位置の画素を白く塗る処理になっている。

3.3　プログラムの分割

次は以下のプログラムを書いて実行してみよう。

```
def square(x, y)
   i -> 0
   while i < 4
      j -> 0
      while j < 4
         memory[60000 + ((y + i) * 100) + (x + j)] -> 999999
         j -> j + 1
      i -> i + 1

i -> 0
while i < 10
   square(i * 4, i * 4)
   i -> i + 1
```

4x4 画素の四角が斜めに描かれる。

3.3.1　部分プログラム

```
def square(x, y)
```

で始まる行は、プログラムの一部に名前をつけており、この機能を「部分プログラム」と呼ぶ。他の言語で「関数」「サブルーチン」「サブプログラム」などと呼ばれるものと同じだ。部分プログラムの範囲は while や if と同様に字下げで示される。

こうして名前をつけておくことで同じプログラムを何度も使い回せるし、わかりやすくもなる。

なお、部分プログラムを def で作るのは使う場所よりも後にあってもいい。

```
square(5, 4)

def sqare(x, y)
   中身
```

という具合に、後にあってもかまわない。

3.3.2 入力値

部分プログラムには入力値を渡すことができ、この例では、

```
   square(i * 4, i * 4)
```

というように i * 4 を二つ渡している。これが、部分プログラム側にある名前付きメモリである x と y にコピーされて、実行される。

なお、入力値がない部分プログラムは、

```
def noInput()
   中身
```

のように () の中身が空になる。これを参照する時も、

```
noInput()
```

のように中身を空にした括弧をつける。括弧を省略することはできない。これは、括弧がついていることで部分プログラムであることが一目でわかる、ということを私が重視したからである。

3.3.3 出力値

部分プログラムは出力もできる。

```
def modulo(a, b)
   out -> (a - (a / b))
```

out という名前付きメモリを作って、そこに部分プログラムから出したいものを覚えさせれば良い。この例では、計算した余りを、部分プログラムを参照した側に渡すことができる。例えば、

```
answer -> modulo(16, 5)
```

という具合だ。この場合、answer は 16 を 5 で割った余りである 1 になる。

```
modulo(16, 5)
```

という記述は、実行中にその出力値である 1 に置き換えられることになる。

出力したりしなかったりすることはできない

なお、部分プログラムは出力があるかないかのどちらかである。出力したりしなかったりすることはできない。より正確に言えば、繰り返しや条件実行の中で「だけ」out を設定することはできない。

```
def f()
   if 1
      out -> 5
```

は、条件実行によって out を設定したりしなかったりするので、エラーとなる。この例なら if は必ず通るので問題ないはずなのだが、Sunaba はそこまできちんとは判断しない。

このため、条件実行で出力する数を変えたい時には、

```
def f(a)
   out -> 0
   if a = 1
      out -> 1
```

という具合に、先に out に「条件実行を通らない時の数」を設定しておく必要がある。

```
def f(a)
   if a = 1
      out -> 1
   if a != 1
      out -> 0
```

はダメだということだ。Sunaba は条件まで見てはくれない。

部分プログラム外での出力

部分プログラムの外でも出力はできる。この場合、0 以外の数である時に限って、出力した数は Sunaba.exe のメッセージウィンドウに表示される。

この例では、4 を出力している。間違え探し用に使えるかもしれない。

3.3.4 名前付きメモリについて補足

名前付きメモリは部分プログラムごとに別々に存在する。中で作ったものは、外からは見えない。外で作ったものは、中では見えない。

```
def f()
   a -> 5

f()
b -> a
```

はエラーである。f() の中の a は、外からは見えないからだ。また、

```
def f()
   a -> 5

a -> 2
f()
b -> a
```

と書いた場合、b は 2 である。5 にはならない。f() の中にある a と、外にある a は関係ないからだ。

3.3.5 ファイル分割

複数のファイルに分けて書きたい場合、

```
include "他のファイル.txt"
```

という具合に include と書くと、指定したファイルの中身がこの位置に貼りつけられる。単に貼りつけられるだけだ。したがって、

```
i -> 0
while i < 10
include "他のファイル.txt"
   i -> i + 1
```

とあったとして、「他のファイル.txt」の中身が、

```
   memory[60000 + i] -> 999999
```

であったとすれば、

```
i -> 0
while i < 10
   memory[60000 + i] -> 999999
   i -> i + 1
```

となる。しかし、このような使い方はすすめない。部分プログラムを別のファイルに書いて整理するのが想定する使い方である。ファイルが 200 行を超えてきたら分割するのが良いだろう。

なお、同じファイルを二度 include すると、二度目は無視される。同じ定数を 2 回作ってしまってエラー、というようなことにはならないのでそこは便利である。

また、部分プログラムや、後述の定数は使う場所よりも下にあってもかまわないため、それらを使うための include であれば、ファイルの末尾でやってもかまわない。その方が邪魔にならないかもしれない。

3.4　その他の機能

他に便利な機能がいくつかあるので紹介しておく。

3.4.1　注釈

Sunaba は行の途中に#が出てくると、そこから行の最後までを無視する。なので、

```
#ここから本番
i -> 0 #回数
```

のように#の後に説明を添えておくとプログラムがわかりやすくなるだろう。

また、複数行に及ぶ注釈で毎行#を書くと面倒なので、/*から*/の間も注釈とみなすようにしてある。

```
/*
この部分プログラムの使い方

[以下のメモリをいじります]
5、6番
[入力]
縦と横
[出力]
成功すれば1、失敗すれば0
*/
def someSubProgram(x, y)
   ...
```

3.4.2　定数

```
const screen -> 60000
```

というように書くことで、定数を作れる。

定数を作れる場所は、「行頭に空白がない場所」、つまり、条件実行や繰り返し、部分プログラムの中でない場所に制限されるが、そのかわりにプログラム全域で使うことができる。

```
const screen -> 60000

def dot(x, y){
   memory[screen + (y * 100) + x] -> 999999
```

というようなことができるということだ。これによって、よく使う数に名前をつけておくことができる。

そして、部分プログラムと同様、定数を作った行より上でも使える。

```
def dot(x, y){
   memory[screen + (y * 100) + x] -> 999999

const screen -> 60000
```

でもかまわない。

3.4.3 メモリ操作の短縮記法

名前付きメモリや定数は、[] をつけることで memory[] と書くより短く書けることがある。

```
const screen -> 60000
```

と定数がある時、

```
memory[screen + 5] -> 999999
screen[5] -> 999999
```

の 2 行は同じ意味である。定数名または名前付きメモリ名の後に [] がある場合、[] の中の数と、定数または名前付きメモリの値を足したものをメモリ番号とする。これで、

```
const screen -> 60000

def dot(x, y){
   screen[(y * 100) + x] -> 999999
```

のように書けて楽になる。

第 4 章

仕様詳細

4.1　名前

名前付きメモリ、部分プログラムにつけられる名前には制限がある。

使える文字

使える文字は、以下に挙げる使えない文字を除いたものである。

!	#	^	*	(
)	-	+	=	{
}	[	]	:	;
"	’	<	>	,
.	/			

基本的に、Sunaba にて何らかの意味に使っている記号は名前には使えない。使っていない記号であっても、^,{,},:のように名前に使うとは思えない記号は使えなくしてある。

一方で、普通のプログラミング言語であれば使えない@や$、&、?などは許している。「どうよ？」、「ボス@2 面」、「出現確率 %」のような名前は使えると便利なこともあるだろう。ただし、これらの記号は全角と半角を区別しない。正確に言えば、全角を見つけると半角だと思って処理する。

使えない言葉

使える文字だけを使っていても、いくつかの名前は使えない。

なら	なかぎり	出力	定数	挿入
if	while	out	const	include

これらが使えると解釈が混乱してしまうし、仮にうまくやれたとしてもプログラムが読みにくくなるだろう。

```
なら + なかぎり なら
```

などというプログラムは解釈できなくもないが、読む気にはなるまい。

第 5 章

メモリ番号と外界とのやりとり

この章では、Sunaba のメモリの用法について詳しく説明する。

5.1 領域ごとの用途

領域ごとの用途は以下のようになっている。

番号の範囲	説明
0-39999	自由領域 + プログラム
40000-49999	Sunaba システムが使用 (名前付きメモリ等)
50000-54999	読み込み専用特別領域
55000-59999	書き込み専用特別領域
60000-69999	画素の色を格納。書き込み専用

5.1.1 自由領域とプログラム

プログラム本体もメモリを消費し、例えば 1000 個のメモリを使うプログラムであれば、39000 番から 39999 番に置かれる。このため、プログラムで好きに使えるのは 38999 番までとなる。何番まで使って良いかは、50010 番を見ると良い。

```
freeRegionEnd -> memory[50010]
```

とすると、使って良い番号がどこまでかがわかる。ただし、この値は「使って良い最後の番号 +1」なので、この値そのものは使えない。これは、

```
i -> 0
while i < freeRegionEnd
   ...
```

のように < を使って役立てることを想定しているからである。

プログラム本体が大きくなれば、好きに使えるメモリは減るが、お手本でも 2000 個程度であり、よほど大きなプログラムを書かない限りメモリを使い果たすようなことにはならないだろう。

5.1.2　Sunaba システム使用領域

40000 番から 50000 番までは、Sunaba が中で使うため、使ってはならない。

```
memory[40002] -> 10123123
```

などと書けば、一撃でプログラムが異常終了することになるだろう。

なお、部分プログラムから部分プログラムを参照する、というようなことをあまり大量にやると、このメモリが枯渇してエラーとなることがある。クイックソートなどの再帰形のアルゴリズムを作る際には注意すべきかもしれない。

5.1.3　読み込み専用特別領域

読みこみ専用特別領域は、外の機械の状態や、プログラムの実行情報などを取得するための場所である。

番号	説明
50000	マウスポインタ X
50001	マウスポインタ Y
50002-50003	マウス左、右ボタン
50004-50009	上下左右、スペース、エンターの順にキー入力
50010	自由領域がどこまでか
50011	横の画素数。画面の幅
50012	縦の画素数。画面の高さ
50013	画素領域の開始番号。つまり 60000
50014	フレーム番号
50015	現在実行中のプログラムの位置
50016	現在のスタックメモリ位置
50017	現在の部分プログラムの使用メモリ領域の開始番号
50018	平均フレーム間隔 (マイクロ秒)
50019	フレームごとの平均余裕時間 (マイクロ秒)
50020	総命令実行数上位 31 ビット
50021	総命令実行数下位 31 ビット

基本的には 50009 番までを使ってマウスやキー入力を取れば良い。ボタンやキーについては、押されていなければ 0、押されていれば 1 が入る。

50011 から 50013 までは、解像度を実行中に変えるためのものである。50014 から後ろの方は開発時に使う情報であり、あまり意味はない。

5.1.4 書き込み専用特別領域

書き込み専用特別領域は、外の機械に指令を出すための場所である。

番号	説明
55000	フレーム同期
55001	自動フレーム同期無効 (1 で無効、0 で有効)
55002	Unicode を書き込むことでメッセージウィンドウに文字表示
55003	プログラム一時停止
55004	横解像度設定。55000 番で反映。
55005	縦解像度設定。55000 番で反映。
55006	音声チャネル 0 の周波数
55007	音声チャネル 1 の周波数
55008	音声チャネル 2 の周波数
55009	音声チャネル 0 の減衰定数
55010	音声チャネル 1 の減衰定数
55011	音声チャネル 2 の減衰定数
55012	音声チャネル 0 の発音振幅。発音のスイッチでもある
55013	音声チャネル 1 の発音振幅。発音のスイッチでもある
55014	音声チャネル 2 の発音振幅。発音のスイッチでもある

55000 番のフレーム同期と、55001 番の自動フレーム同期無効は特に重要だ。

Sunaba は何もしなければ、60000 番以降への書き込みや、50000 番台の読み込みがプログラム内で発生した瞬間に、外部の機械との通信を行い、情報が到着したり、指令が実行されたりするのを待つ。

```
memory[60000] -> 999999
```

と書けば、画素の色情報をモニタに転送して、60000 番に対応する画素が白く塗られるまで待つ。したがって、下の行に進んだ時には画面に点が出ていることが保証される。

また、

```
if memory[50002] = 1
```

とあれば、この時にマウスのボタンが押されていないかを調べに行き、情報が得られるまで待つ。そのため、その時にマウスのボタンが押されていれば 1 になることが保証される。

しかし、外界とのやりとりには時間がかかり、とりわけ画面の反映には 1/60 秒もの時間がかかる。したがって、標準状態のままでは、1 秒に 60 個の画素しか塗ることはできない。

そこで、55001 番に 1 を入れておくことで、このように自動で外界とのやりとりをする機能を無効化できる。

```
memory[55001] -> 1
i -> 0
while i < 10000
   memory[60000 + i] -> 999999
   i -> i + 1
```

と書けば、最初の 1 行がない時には 10000/60 で 170 秒近くかかって画面を白く塗りつぶすが、最初の 1 行を書くだけで、これに要する時間が一瞬になる。

ただし、放っておいても画面に絵は出なくなるし、50000 番からの入力の更新も行われなくなる。例えば、

```
memory[55001] -> 1
while 1
   if memory[50002] = 1
      memory[60000] -> 999999
```

と書けば、起動の瞬間に左クリックしていない限り 50002 番が 1 になることは絶対にない。起動の瞬間の状態のまま更新されないからだ。逆に起動の瞬間にクリックされていたのであれば、永遠に 1 のままである。なので、この場合は、

```
memory[55001] -> 1
while 1
   if memory[50002] = 1
      memory[60000] -> 999999
   memory[55000] -> 1
```

のように最後に 55000 番に 1 を入れることで、外界とのやりとりを行う必要がある。ここに入れる数は何でも良い。すでに 1 であっても 1 を入れれば良いし、0 でも 100 でも良い。この番号に対して書きこみを行えば、それがスイッチとなって外界とのやりとりが行われる。

したがって、ゲームのような操作できるプログラムを作る場合、

```
while 1
   memory[55000] -> 1
   #入力を見て
   #何か計算をして
   #絵を描いて
```

というような構造になるはずである。

無限の繰り返しがあり、その繰り返しの度に 55000 番に 1 を入れる。この度におよそ 1/60 秒程度の時間が経つため、この繰り返しはおおよそ秒間 60 回実行されることになるわけだ。

文字出力

なお、50002 番に Unicode の文字番号を入れると、メッセージウィンドウに文字が出る。プログラムを作っている最中にうまく活用していただきたい。

例えば、

```
memory[55002] -> 65
```

と書けば、A が表示される。A は 65 番だからだ。

思い通りにプログラムが動かない場合などに活用するのがいいだろう。ただし、その目的であれば、画面の適当な位置に適当な色で点を描いても大差なく、「この条件実行が実行されていれば緑の点を出そう」というようなことはいくらでもできる。その意味で、さして重要な機能とは言えない。

中断

50003 番に 1 を入れると、プログラムが中断される。一度中断してしまうとプログラムの中からは再開できないので、これ単体では何の役にも立たないのだが、

Sunaba.exe の機能として、特定の番号のメモリに強制的に好きな数を入れられるので、50003 番を指定して 0 を入れることで好きなタイミングで再開させることができる。また、止まっている最中には当然メモリも変化しないため、Sunaba.exe でメモリの状態を調べるために止める、という用途に使える。例えば特定の場所で止めてみて、そこである番号のメモリが思った通りの数になっているかを調べてみる、などの用途がある。あるいは、コマ送りに使っても良いだろう。

解像度変更

50004、50005 番にそれぞれ横、縦の解像度をセットし、55000 番にて同期を取ることで解像度を変更できる。起動後すぐに設定すべきで、何度も変更すべきではない。

最大値はどちらも 500、最小値は 1 である。なお、500 までは許すものの、おそらく 200 を超えると動作速度的に耐えられないものになるだろう。点を一つ一つ塗る仕様であるため、単純に画面を真っ黒に塗りつぶすだけでも大変な時間がかかるからである。

音声再生

50006 番から 9 つは、3 チャネルある正弦波を作る機械を制御するためにある。

```
memory[55006] -> 440
```

```
memory[55009] -> 10
memory[55012] -> 1000
```

これで、440Hz の音が鳴る。減衰係数は 0 では全く減衰せず、1000 にすると一瞬で減衰してほとんど音が鳴らなくなる。

振幅は 0 から 10000 である。ただし、複数鳴らしている時に合計が 10000 を超えると音が割れることがあるので少し小さ目にしておくことをおすすめする。

第 6 章

サンプルライブラリ

SunabaGuideSample フォルダ内に library というフォルダがある。この中には、多少プログラミングに慣れた人で Sunaba を使って遊んでみたいという人向けに、よく使いそうな機能を部分プログラムとして用意してある。

あくまでも目的は「おもちゃ」であって、Sunaba が対象とする初心者に向けたものではない。このような機能を使うよりも、自分で苦しんで作り上げる方が学習効果は高いからだ。しかし、おもちゃで遊ぶのも悪いことではないし、プログラムを読んで参考にするのも悪くない。

さて、それぞれのファイルの内容は以下のようなものだ。

algorithm.txt	並び換えや検索など
error.txt	エラー番号リスト
graphics.txt	絵を描く
input.txt	マウスとキー入力を簡単に使う
ioMap.txt	特別なメモリの番号に名前をつける
memory.txt	メモリ確保
music.txt	音楽再生
system.txt	特殊メモリの操作などなど
utility.txt	雑多な機能

以下ではそれぞれのファイルの中にある部分プログラムを説明する。

6.1 algorithm.txt

メモリ上に並んで数を、大きさの順に並び換えたり、データを探し出したりする。

6.1.1　用法

まずは並び換えである。

```
#0番から9番までの10個が、だんだん大きくなるように並び換える
sortAscending(0, 10)
#0番から9番までの10個が、だんだん小さくなるように並び換える
sortDescending(0, 10)
```

また、並び換えが終わっていれば、その中から特定の数を素早く探してくることができる。

```
# 5を小さい順に並び換えた中から探し、何番目にあるかを得る
index -> binarySearchFromSortedAscending(10, 10, 5)
# 5を大きい順に並び換えた中から探し、何番目にあるかを得る
index -> binarySearchFromSortedDescending(10, 10, 5)
```

binarySearchFromSortedAscending() と、binarySearchFromSortedDescending() は見つからないと-1 を出力する。見つかれば、「もらった領域中で」何番かを出力する。上の例なら、例えば 5 が「もらった領域中で」0 から始めて 2 番にあったとしよう。もらった領域が 10 番からなので、メモリの番号は 12 番である。しかし出力されるのは 2 だ。

6.2　error.txt

いろんな部分プログラムが出力したり、Sunaba.exe のメッセージウィンドウに出力したりするエラー番号が何なのかを調べるためのファイル。

例えば、Sunaba.exe の方に「error:-5」と表示されている場合、error.txt を調べると-5 なのは「Error_OUT_OF_RANGE」なので、何か想定されている範囲を超えるようなことが起きたのだな、ということがわかる。間違えを直す時のヒントになるかもしれない。

-1	Error_MEMORY_DESTRUCTION	使ってはいけないメモリに書きこんだ
-2	Error_NO_MORE_MEMORY	メモリがもうない
-3	Error_NOT_INITIALIZED	XXX_initialize を参照してない
-4	Error_NOT_IMPLEMENTED	この部分プログラムはまだ作ってない
-5	Error_OUT_OF_RANGE	想定外の数を受け取った
-6	Error_ZERO_DIVISION	0 で割ろうとしている

6.3　graphics.txt

図形を描くための部分プログラムを用意してある。将来 Sunaba の仕様が変わって、画面の幅や高さ、画面につながったメモリの番号が変更になっても、これらの部分プログラ

ムを使っていればプログラムを書き直さずに対応できる。

6.3.1 drawCircle

円の輪郭を描く。

用法

```
drawCircle(centerX, centerY, radius, color)
```

centerX,centerY を中心として半径 radius の円を描く。線画である。色は color。

6.3.2 drawFilledCircle

円を描き、塗りつぶす。

用法

```
drawFilledCircle(centerX, centerY, radius, color)
```

$(centerX, centerY)$ を中心として半径 radius の円を描く。中は塗りつぶされる。色は color。

6.3.3 drawRectangle

傾いていない長方形の輪郭を描く。

用法

```
drawRectangle(left, top, width, height, color)
```

$(left, top)$ を左上として、幅 width、高さ height の長方形の輪郭を描く。色は color。

width,height の細かな扱いだが、これらが 0 の時には何も描かれず、1 ならば線になる。幅が w なら、右端の座標は w-1 となる。

6.3.4 drawFilledRectangle

傾いていない長方形を描く。

用法

```
drawFilledRectangle(left, top, width, height, color)
```

$(left, top)$ を左上として、幅 width、高さ height の長方形を描き、塗りつぶす。色は color。

width,height の細かな扱いだが、これらが 0 の時には何も描かれず、1 ならば線になる。幅が w なら、右端の座標は w-1 となる。

6.3.5 drawSquare

正方形の輪郭を描く。

用法

```
drawSquare(left, top, size, color)
```

$(left, top)$ を左上として、大きさ size の正方形の輪郭を描く。色は color。

size の細かな扱いだが、これらが 0 の時には何も描かれず、1 ならば点になる。大きさが s なら、右端の座標は s-1 となる。

6.3.6 drawFilledSquare

正方形を描く。

用法

```
drawFilledSquare(left, top, size, color)
```

$(left, top)$ を左上として、大きさ size の正方形を描き、塗りつぶす。色は color。

size の細かな扱いだが、これらが 0 の時には何も描かれず、1 ならば点になる。大きさが s なら、右端の座標は s-1 となる。

6.3.7 drawLine

線分を描く。

用法

```
drawLine(x0, y0, x1, y1, color)
```

$(x0, y0)$ から $(x1, y1)$ まで線を引く。両端を含む。色は color。

6.3.8 drawPoint

点を描く。

用法

```
drawPoint(x, y, color)
```

(x, y) に点を描く。色は color。

6.3.9 drawTriangle

三角形の輪郭を描く。

用法

```
drawTriangle(x0, y0, x1, y1, x2, y2, color)
```

$(x0, y0)$、$(x1, y1)$、$(x2, y2)$ を頂点とする三角形を描く。色は color。

6.3.10 drawFilledTriangle

三角形を描く。

用法

```
drawFilledTriangle(x0, y0, x1, y1, x2, y2, color)
```

$(x0, y0)$、$(x1, y1)$、$(x2, y2)$ を頂点とする三角形を描き、塗りつぶす。色は color。

6.3.11 clearScreen

画面全体を塗りつぶす。

用法

```
clearScreen(color)
```

画面全体を color で塗りつぶす。

6.3.12 drawPicture

画面に前もって作った絵のパターンを描く。

用法

```
drawPicture(left, top, width, height, pictureStart, pictureWidth)
```

$(left, top)$ を左上として、幅 width、高さ height の長方形を描画する。画素の色は前もって、pictureStart 番から始まるメモリに置いているものとする。また、用意した画の幅は pictureWidth で指定する。

したがって、0 番地から幅 50 の画像を用意して、その左上 20 × 20 部分を画面の中央に描画するならば、

```
drawPicture(40, 40, 20, 20, 0, 50)
```

となる。

6.3.13　multiplyColor

色を、赤、緑、青に分け、それぞれ掛け算をし、また合成したものを出力する。

掛け算は、おおよそパーセントで行われると思えば良い。50% と 50% を掛け算すれば 25% となる。正確な計算式は、

$$結果 = 色\ 0 \times 色\ 1/99$$

となる。

用法

```
result -> multiplyColor(color0, color1)
```

result は、color0 と color1 を掛け算したものになる。例えば、color0 が 777777 で、color1 が 505050 であれば、結果は 383838 となる。おおよそ半分になるわけだ。

元の色から青みを減らしたければ、例えば 999955 のように青のみが少ない色を掛ければいいし、赤だけを残したければ 990000 を掛ければ良い。

6.4　input.txt

6.4.1　概要

キー入力を楽に使うためのものである。普通にメモリを直接見れば「今押されていること」はわかるが、「今押されたばかりであること」や「今離されたばかりであること」はわからない。これを改善する。

用法

おおよそこんな感じとなる。

```
System_initialize() #これが必要
input -> Input_create() #inputなる名前付きメモリに出力を入れる
```

```
while 1 #一番外側の無限繰り返し
   Input_update(input) # これを毎回使う
   #押されているか？
   if Input_on(input, Input_MOUSE_LEFT)
      ...
   #押されたばかりか？
   if Input_triggered(input, Input_MOUSE_RIGHT)
      ...
   #離されたばかりか？
   if Input_released(input, Input_ENTER)
      ...
   #マウスカーソル
   x -> Input_pointerX(input)
   y -> Input_pointerY(input)

Input_destroy(input) #終わったら参照するが、しなくても良い。
```

まず System_initialize() を前もって参照しておく必要がある。中でメモリ確保を行うからだ。

その後は、Input_create() を参照して準備をする。出力値は全ての部分プログラムで使うので、どこかに覚えておく。

あとは、適当な頻度で Input_update() を参照する。ここでキー状態を更新する。

キー状態を取るには Input_on、Input_triggered、Input_release を使う。それぞれ、「押されているか」「押されたばかりか」「離されたばかりか」を 1 か 0 で出力する。いずれも入力に Input_create() の出力した数と、用意した定数 (Input_MOUSE_LEFT のような) を使う。この定数でどのキーが欲しいのかを指定する。

ポインタ位置は、Input_pointerX と Input_pointerY で取得する。

使い終わったら Input_destory に Input_create の出力を渡す。内部で使っていたメモリを解放する。

6.4.2 定数

Input_MOUSE_LEFT	左ボタン
Input_MOUSE_RIGHT	右ボタン
Input_UP	上キー
Input_DOWN	下キー
Input_LEFT	左キー
Input_RIGHT	右キー
Input_SPACE	スペースキー
Input_ENTER	エンターキー

6.5　ioMap.txt

メモリ番号を覚えておかずに済ますための定数を用意してある。いずれ Sunaba の仕様が変わって番号が変更されたとしても、この定数を使っていればプログラムを書き換えずに済むはずである。

ただし、基本的にはこの定数を使わずにプログラムを書けるように機能を用意してある。例えば、55000 番の MEM_SYNC を必要になる時には、System_sync() を使えば良い。

であるから、ここにはその一覧は載せない。

6.6　memory.txt

メモリ管理を行う。Sunaba はそのまま使う場合「何番から何番までは何に使う」と自分で決めねばならないが、それが面倒なこともある。

そこで、「いくつ欲しい」と言えば、それだけ切り出してきて先頭の番号を出力してくれる関数があれば楽である。使い終わったところはまた使い回せればなお良い。

なお、これを使う場合、番号を直接指定してメモリを使ってはならない。

6.6.1　用法

```
Memory_initialize() #まずこれを使うこと

start -> Memory_allocate(15) #startが15個つながった場所の先頭番号になる
Memory_deallocate(start) #もう使わないと宣言
```

まず Memory_initialize() を使っておく。これで準備が整う。

あとはメモリが欲しくなった時に、何個欲しいのかを Memory_allocate() に渡す。すると、先頭番号が出力される。例えば 100 個欲しいとして、6 番から空いているのであれば、6 が出力される。start[0] から start[99] までは好きに使って良い。start[100] を使えば、これは違反である。

使い終わったら、Memory_deallocate() にその番号を渡す。そうすると、その後 Memory_allocate() を使った時に今まで使っていたメモリ番号が出力されることもあるだろう。Memory_deallocate せずにひたすら Memory_allocate() ばかりしていれば、いつかメモリが足りなくなる。足りなくなると Memory_allocate() は-1 を出力する。

6.7　music.txt

音楽を再生するための機能である。おおよその使い方は以下のようになる。

```
Memory_initialize() #必須
loop? -> 1
channel -> 0
noteCount -> 4
notes -> Memory_allocate(4 * noteCount)
p -> notes
p -> set4(p, 0, BF4, 10, 5000) #時刻0、シb、減衰10、振幅5000
p -> set4(p, 100, A4, 10, 5000) #時刻100、ラ、減衰10、振幅5000
p -> set4(p, 200, C5, 10, 5000) #時刻200、ド、減衰10、振幅5000
p -> set4(p, 300, B4, 10, 5000) #時刻300、シ、減衰10、振幅5000
length -> 400
music -> Music_create(notes, noteCount, loop?, length, channel)
Memory_deallocate(notes) #もう解放していい
Music_setTempo(music, 50) #50%に遅く
Music_setVolume(music, 150) #ボリューム150%
Music_setPitch(music, -3) #全体を3半音下げる
while 1
   Music_update(music)
   System_sync()
```

create で作り、setTempo,setVolume,setPitch 等で調整しつつ、update を定期的に使えば音が鳴る。

create の第一入力は楽譜だ。ここではメモリを 16 個使って楽譜としている。1 項目 4 つで、順に、発音時刻、音の名前、減衰、振幅だ。

発音時刻は「何度目の update で発音するか」、名前は 0 から 127 で、1 増える度に半音上がる。60 がいわゆる中央ドだ。これは music.txt の中で C4、BF3 のような名前がつけられている。二文字目が F ならフラット、S ならシャープだ。数字はオクターブ番号である。

減衰は 0-1000 程度で、大きいほど音がすぐに消えるので鋭くなる。振幅は 0-10000 で、大きいほど音が大きい。ただし、複数鳴ったり、setVolume で大きくした結果「合計が」10000 を超えると音が割れるので注意すること。

create の第二入力は楽譜の音の数、次がループするかしないか。するなら 1 でしないなら 0。次が曲の長さ。最後がチャネル番号で 0、1、2 のどれかを指定する。

setTempo は再生速度をパーセントで指定する。setPitch は全体を何半音ずらすかを指定する。マイナスでも良い。ただし、ずらしすぎて結果 0 から 127 に収まらなくなるとエラーになる。

setVolume は全体のボリュームをパーセントで指定する。大きくなりすぎて 10000 を超えると音が割れる。

中では System_playSound を使っているだけなので、そちらの説明も参照すること。

6.8 system.txt

5 万番台以降の特殊メモリを使う雑多な機能をここに入れてある。

6.8.1 System_sync

```
System_sync()
```

外の機械との通信を行う。画面に絵が出て、キーボードやマウスの入力も更新できる。

6.8.2 System_disableAutoSync

```
System_disableAutoSync()
```

外の機械との通信が自動で行われないようにする。「手加減」のチェックボックスをオンにするのと同じ。

6.8.3 System_enableAutoSync

```
System_enableAutoSync()
```

外の機械との通信が自動で行われるようにする。「手加減」のチェックボックスをオフにするのと同じ。

6.8.4 System_break

```
System_break()
```

プログラムを停止させる。Sunaba.exe から該当する番号のメモリを 0 にすると動き始める。特定の場所で止めてメモリの状態を見たい場合などに使う。

6.8.5 System_outputChar

```
System_outputChar(32) #半角スペース
System_outputChar(65) #A
System_outputChar(31070) #神
```

Sunaba.exe のメッセージウィンドウに文字を表示する。入力は Unicode の文字番号。

6.8.6 System_outputString

```
System_outputString(0, 10) #10文字流す
```

Sunaba.exe のメッセージウィンドウに文字を表示する。表示する文字はメモリ中にあるとする。この例では 0 番から 10 個のメモリに、10 文字の Unicode 文字番号が入っていることを想定している。

6.8.7 System_outputNumber

```
System_outputNumber(123456789)
```

Sunaba.exe のメッセージウィンドウに数を表示する。System_outputChar なら 65 を渡せば A になるが、これを使えば 65 と表示できる。

間違いを直す時などに便利であろう。

6.8.8 System_outputNumber

```
System_outputNewLine()
```

Sunaba.exe のメッセージウィンドウ上で改行する。System_outputChar(10) と同じ。

6.8.9 System_die

```
if x > 10
   System_die(15)
```

入力値を Sunaba.exe のメッセージウィンドウに表示してから、メモリの-1 番をいじってエラーになり、プログラムが止まる。

「こんなことが起こったら間違い」ということが前もってわかっている時に、使うと間違い探しに役立つ。この例では、x は 10 以上にならないはずだ、と思っているわけである。その想定が外れた時に、この関数が発動してプログラムが止まるから、間違いがあることがわかる。

なお、表示は「error:入力値」の形を取る。

6.8.10 System_setResolution

```
System_setResolution(200, 150)
```

解像度を変更する。この例では横 200、縦 150 にしている。画素とメモリの対応関係が変化し、これまで描いた絵は真っ黒に消される。

プログラムの開始直後に使用することをおすすめしておく。途中で使ってうれしいことはあるまい。

6.8.11 System_screenWidth

```
width -> System_screenWidth()
```

画面の横解像度を得る。

6.8.12　System_screenHeight

```
height -> System_screenHeight()
```

画面の縦解像度を得る。

6.8.13　System_playSound

正弦波の音を鳴らす。

```
System_playSound(0, 440, 10, 2000)
```

最初の入力がチャネル番号。0、1、2 があり、つまり最大 3 つ重ねられる。

次が周波数。440 なら A4 と呼ばれる音で、いわゆる「ラ」である。

次が減衰。感じを見て設定してほしい。およそ 0 から 500 程度に設定することになる。

最後は「音の振幅」で、おおよそ大きさである。0 から 10000 を指定する。周波数によっても感じ方は様々なので、試行錯誤が必要になるだろう。なお、複数鳴らす時に合計が 1 万を超えると音が割れるので、そこも考慮に入れておくべきである。

なお、55000 番をいじって同期する前に同じチャネルについて再び呼ぶと、上書きされてなかったことになる。

6.9　utility.txt

雑多な部分プログラムを集めてある。

6.9.1　modulo

余りを計算する。

```
answer -> modulo(a, b) #answerはaをbで割った余りになる。1。
```

a(割られる数) がマイナスの場合、余りもマイナスになりうる。例えば、-16 を 5 で割った余りは、-1 になる。b(割る数) がマイナスの場合はプラスの場合と変わらない。16/-5 も 16/5 も、余りは 1 である。

6.9.2　random

ランダムっぽい数を出力する。コンピュータに本当のランダムはないから、ランダム「っぽい」としか言いようがない。

```
r -> random(r)
```

random に最初に渡す数はなんでもいい。それによって今後の展開が変わる。2 回目以降は、random が出力した数を渡す。

例えば、最初に 1 を渡した場合、2043、1028130、177871、584195 の順に出力する。出力される数の範囲は、1 以上 1048572 以下の数である。1048573 回使うと、元の値に戻ってくる。

この「ランダムっぽい数を生成する方法」はいくつか知られているが、ここでは線形合同法を採用している。興味があれば調べてみると良かろう。計算にかかる時間や、使うメモリの量、どれくらいランダムっぽいかなどがやり方によって異なり、なかなか奥深いものである。

サイコロが欲しい時

これをサイコロに使う場合は、

```
r -> random(r)
dice -> modulo(r, 6) + 1
```

のようにする。random は 0 以上の数を出力するので、6 で割った余りは 0 から 5 になる。だからこれで 6 通りの数が得られる。1 から 6 にしたければ 1 を加えれば良い。

6 を 20 にすれば 20 面サイコロになるし、100 にすれば 100 面サイコロになる。101 にすれば 0 から 100 までが得られるので、パーセントをランダムで変えたい時などにいいだろう。

6.9.3 abs

絶対値を返す。つまり、マイナスならプラスにする。

```
x -> abs(x)
```

x が-6 なら 6 になり、6 ならそのまま 6 になる。

```
if x < 0
   x -> -x
```

と同じであるが、このように部分プログラムにすることで 1 行で書け、場所も食わない。

6.9.4 sign

プラスなら 1 を、マイナスなら-1 を、ゼロなら 0 を出力する。

```
s -> sign(x)
```

プラスなら +1 づつしたいが、マイナスなら-1 づつしたい、というような時に便利である。

6.9.5　sqrt

平方根を計算する。

```
sqrt2 -> sqrt(4)
```

この例なら sqrt2 は 2 になる。

整数なので誤差があるが、四捨五入で近い方が選ばれる。例えば 8 の平方根は 2.82 くらいだが、これは 3 になる。

マイナスを渡すと、Error_OUT_OF_RANGE をメッセージに出力して止まる。

また、この計算にはかなりの時間 (足し算数百回分) がかかるため、あまり頻繁に使うと満足な速度では動かないだろう。

なお、この計算にはニュートン法なる方法を用いており、興味があれば中身を見てみると良い。コンピュータは本質的に整数しか扱えず、できる計算も加算、減算、乗算のみである。除算すら内部的にはそれらを組み合わせて実現している。だから当然平方根も同じように加算、減算、乗算でどうにかするしかない。それをどうしているのかを覗き見るのには丁度いい題材だろう。

なお、"sqrt"という名前は、英語の"SQuare RooT" から適当に子音を取り出してきたものである。

6.9.6　copyMemory

メモリの範囲をコピーする。

```
copyMemory(0, 100, 50)
```

この例なら、100 番から 50 個を、0 番から 50 個にコピーする。0 と 100 が、1 と 101 が、2 と 102 が、といった具合に同じ数になる。最後は 49 番と 149 番であることに注意。

6.9.7　setMemory

メモリの範囲をある数にする。

```
setMemory(0, 999999, 100)
```

この例なら、0 番から 100 個を全部 999999 にする。最後は 99 番であることに注意すること。

6.9.8 set1,set2,set3,set4,set5,set6,set7,set8,set9,set10

メモリの範囲に数をセットする。

```
set1(0, 0)
set2(0, 0, 1)
set3(0, 0, 1, 2)
set4(0, 0, 1, 2, 3)
set5(0, 0, 1, 2, 3, 4)
set6(0, 0, 1, 2, 3, 4, 5)
set7(0, 0, 1, 2, 3, 4, 5, 6)
set8(0, 0, 1, 2, 3, 4, 5, 6, 7)
set9(0, 0, 1, 2, 3, 4, 5, 6, 7, 8)
set10(0, 0, 1, 2, 3, 4, 5, 6, 7, 8 ,9)
```

単純にプログラムを短くするための機能で、最初の入力で指定した番号から順に、入力値をセットしていくだけである。set2 であれば、0 番が 0 に、1 番が 1 になる。set8 なら 7 番が 7 になるところまでだ。

このようなものがあるのは、Sunaba が機能不足だからであるが、Sunaba の目的からいってこれを解決するような文法は足していない。

なお、これらの関数には出力値がある。例えば set4 は、第一入力に 4 を足したものを、set6 は 6 を足したものを出力する。これを使うと、

```
p -> 0
p -> set4(p, 0, 1, 2, 3)
p -> set6(p, 0, 1, 2, 3, 4, 5)
p -> set7(p, 0, 1, 2, 3, 4, 5, 6)
#ここでpは最後にいじったメモリ+1の番号になっている
```

というように、適当な名前付きメモリを渡してはそこに受け取ることを繰り返すことで、番号を計算する必要なしに隙間なく詰められて便利である。これを使わなければ、

```
set4(p, 0, 1, 2, 3)
set6(p + 4, 0, 1, 2, 3, 4, 5)
set7(p + 10, 0, 1, 2, 3, 4, 5, 6)
```

という具合に 4,10 といった「今の場所」をいちいち計算せねばならないからだ。

ただし、この便利さはコンピュータに余計な計算をさせることで得られており、実行にかかる時間は当然伸びることになる。