記憶領域とポインタ仮引数について

田中　寛※

要旨：C言語プログラミングにおいて、関数の呼び出しの実引数として配列が指定され、さらに、その関数の記述で様々の表現の仮引数がとられた時に、記憶領域で何が起きてどのように処理されるかを調べた。その結果、仮引数の表現が何であっても、記憶領域に実際に取られるのは、ポインタであった。

キーワード：プログラミング、C言語、記憶領域、配列、ポインタ

§1. はじめに

プログラム言語としてのC言語にとって、それで書かれたプログラムが動作するコンピュータとは、メモリといういろいろな俳優が登場する舞台において、一連の命令であるプログラムという演出家の手によって登場人物が動かされることによって演出家の考えた意図を表現するかのように、様々な情報処理が実現される道具である。コンピュータのイメージは、例えば、電子回路を設計する技術者にとっては演算回路の複雑な集合であったり、パソコンの設計者にとっては現在の技術水準にとって最適なアーキテクチャを実現しようとする考察対象であったりする。つまり、扱う立場によって考察対象となるコンピュータというものが様々な抽象度の度合いで見ることができる道具であるのである。現代技術の中でも、コンピュータはこの抽象度が格段に高いことを認識することが重要である。このことは、コンピュータの用途がいわゆる汎用であると称されることにも由来する。

現代社会において、コンピュータが情報社会の中心の道具として遇される理由は、取り扱われる情報が何らかの形でコンピュータのソフトウェアによって処理されるからである。ビジネス情報の流通からインターネットによる個人的情報交換までのあらゆる情報は、何らかの情報を得たり与える人に対する直接最終的にソフトウェアを媒介物としている。たとえコンピュータというハードウェアしか使っていないと考える場合があったとしても、そのハードウェア上で動いているソフトウェアを利用していることに必ずなっている。この意味で、経済社会にとっての情報社会になっていくことの意義は、広い意味でのコンピュータという少数の種類のハードウェアの上で、人間の思考という無限の可能性を持つものに依拠するソフトウェアというものを作成することによって、多様な可能性を持つビジネスを実現できるようになることである。すなわち、経済社会にとっての当面の焦点は、コンピュータ・ソフトウェアのアイディアである。

コンピュータ・ソフトウェアのアイディアがあったとすると、それがビジネスとして実現可能であるかなどの検討を経て、最終的なそのアイディアの実現は何らかのコンピュータ言語で作成されたプログラムによることになる。つまり、ソフトウェアビジネスを推進

※青森公立大学

する側としては、最初に述べた舞台と演出家の抽象度のレベルでコンピュータを捉えなければならないことになる。しかしながら、今の日本では、このことはビジネスサイドではほとんど認識さえされていない。また、その人材を養成するべきである大学でも同様である。日本の経済社会が、情報社会というグローバルに世界中で進展している現実に対応しなくなっていることを認識しなくてはならない。このままでは、日本は世界から取り残されることは必死である。

ビジネスサイドの問題とともに、技術サイドにも問題が無いわけではない。C言語というのは、もともとはOSのUNIXを記述する為に開発された。[1] しかし、今や、C言語から発展させられたC++言語やjava言語がソフトウェア開発の主たる言語となって、その基本となるC言語はますます教育の為の言語として欠かせないものとなった。また、C++言語やjava言語を学ぶ為には、C言語を知っていなければ理解できないことが多々あることも事実である。そのC言語を学ぶ上で、最も学習の困難な項目としてポインタがある。プログラマとして現場で働いていても完全に理解できていないものがかなりいることも否定できないのが現実である。そのうちのほとんどが、配列とポインタとの関係が理解できていないことを自覚しているものと推測される。ところが、配列とポインタとの関係について明確に述べられた文献[2]を残念ながら目にすることは少ない。その理解されていない問題の要点は、後に詳述することにしている記憶領域といものの番地データだけを記憶するポインタと異なり、これも記憶領域にある配列にはその番地データを記憶する物理的仕組みが無いということだけである。

演劇における演出家に擬されるコンピュータ・プログラムで利用できるC言語における命令は、体系的に、演算子、文、関数に分けられる。演算子は、ほぼ機械語命令に正確に対応しており、[3] 文は、構造化の考え方に従っている。[4] これら二つは、関数に比べてかなり単純である。関数は、あるまとまった機能を記述したものを何回でも利用できるようにする仕組みである。その関数の文法では、定義と呼び出し、記述をそれぞれ適当な別々の場所で記されなければならない。このうち、定義と記述は一回だけ行われなければならないが、呼び出しはもともとの考えから複数回可能である。[5] また、関数を利用するには、プログラム全体として何らかの形でそれぞれの回数回だけ記述されておればよく、一つのソースプログラムの中で全てが記されている必要が無い。このことによって、複数の人によるプログラム作成作業が可能になり、産業としてのソフトウェア業が可能となる反面で、プログラマ間のコミュニケーションのとてつも無いぐらい複雑な問題が発生する。

関数を呼び出す際に、ある機能を処理する為に必要となるデータや処理結果を受け取るデータを実引数として指定する。[6] また、関数の処理結果を受け取ることは、引数ではなく戻り値としても受け取ることができる[7]ことはともかくとて、実引数をいろいろとかえて一つの関数を呼び出して何回でも利用できるのである。その呼び出しの実引数に対応するデータを、関数の記述において仮引数として受け取ったことにして、関数の処理内容の記述がなされる。それらの値の受け渡しが、単純な変数の場合は値渡しという方法で、また、配列の場合は番地渡しという方法でなされると記述されていることもある。[8] 言葉と

しては、確かに違いのあることが分かるが、実際にコンピュータで何が行われているのかが明確でない。先に述べたポインタと配列の関係がプロと思われる人の間でもよく分かっていない一因にもなっている。

この論文の目的は、関数の呼び出しの実引数として配列が指定され，さらに，その関数の記述において、様々の表現の仮引数がとられた時に、コンピュータの構成要素として最重要であるものの一つである記憶領域で何が起きて、どのように処理されるかに焦点を当て、ポインタと配列の関係を明らかにする。§2においては、記憶領域とはどういうもので、C言語のプログラムとどのような関係があるかを明らかにする。§3においては、関数の記述において仮引数として配列ないしポインタを指定すると記憶領域に何が起きるかを示す。これらのセクションでは、必要なプログラムを提示し、コンピュータで動かした結果によって説明を裏付けることにする。最後の§4において、得られた結果をまとめる。

§2. 記憶領域

電子デバイスとしてのメモリは、パソコンを買う際にどれだけの量を買うかが問題となる。昨今の一般に販売されているメーカ製のパソコンでは、128MBから256MBという量が多いようである。単体としてのメモリは、数枚から十数枚の超LSIを組み合わせて構成するが、今は、超LSIの128Mb世代から256Mb世代への移行期にあるようである。実際に販売される商品としてのメモリ量は、超LSI作成の技術進歩にかなりの程度依存していて、しかも、OSのバージョンアップに伴い最低必要メモリ量が急激に増え続けている。そして、32ビットCPUで扱うことができる限界である4GBに近付いてきている。この面で64ビットCPUの必要性が高まっている。[9]

電子デバイスとしてのメモリは、1ビットごとに扱うことができてよさそうなものであるが、メモリ量の数え方で分かるように、1ビットの８倍の1バイトが、メモリ量を換算する単位である。全てのメモリをバイト単位で順に並べて、ゼロからの番号を付ける。その番号のことを番地という。1Kバイト目は、1024－1番地であり、1Mバイト目は、1024×1024－1番地という具合である。物理的には複数の超LSIで構成されているのであるから、メモリは必ずしも連続していないわけであるが、連続していると考えられるのはあくまでも抽象化して考えるという意味での記憶領域についてである。

C言語において、この記憶領域にとることができる基本型とそのバイト数を求めるプログラムをリスト1に示す。そのプログラムをApple社製のPower Book G4 12'のgccコンパイラ(GCC version 1151, based on gcc version 3.1)でコンパイルして得られた実行プログラムの実行結果もリスト1に示す。このプログラムを同じハードウェア上の別のコンパイラでコンパイルした時には、その実行結果がリスト1とは異なるかもしれない。[10] 整数型や浮動少数点型の複数あるデータの型は、C言語の仕様では大きさの順番が決まっているだけで、その占めるバイト数は決められてはいないのである。

```
[uranus49:~/Documents/paper] tanaka% ./a.out
size of char = 1
size of short = 2
size of int = 4
size of long = 4
size of long long = 8
size of float = 4
size of double = 8
size of long double = 8
[uranus49:~/Documents/paper] tanaka% cat pr1_1.c
#include <stdio.h>

main()
{
   printf("size of char = %d\n",sizeof(char));
   printf("size of short = %d\n",sizeof(short));
   printf("size of int = %d\n",sizeof(int));
   printf("size of long = %d\n",sizeof(long));
   printf("size of long long = %d\n",sizeof(long long));
   printf("size of float = %d\n",sizeof(float));
   printf("size of double = %d\n",sizeof(double));
   printf("size of long double = %d\n",sizeof(long double));
}
[uranus49:~/Documents/paper] tanaka%
```

リスト1．データの基本型を画面に表示するプログラムを実行した実行結果およびソースプログラムである。

コンピュータとしての活動の中核部分としての記憶領域は、番地の順番に一列に並んでいるバイトの箱であると考えて間違いは無い。しかし、そのことを視覚的に表現することは、現実的にはかなり困難である。なぜなら、紙の上でもディスプレイの画面上でも、横にも縦にも一列に並べることを行うと、その列はすぐに紙やディスプレイからはみ出てしまうからである。C言語にとって表現される必要のあることは、記憶領域の中に変数なり配列などが存在しており、必要がある場合にそれに名前が付いていることを知ることである。一列のバイトの箱によっても、このことは表現できるが、ある枠で囲った平面で記憶領域を表しておいて、その図中でC言語にとって必要なものを表現するようにした方が、より柔軟性がある。そして、必要がある場合に、名前や番地データをその図中にある一定の規則を決めて書き込めるようにすれば良い。

リスト1で求められたデータの基本型の変数が、記憶領域で占める様子をビット単位の箱で示したのが、図1である。一つの箱には、ビットの1か0が入る。

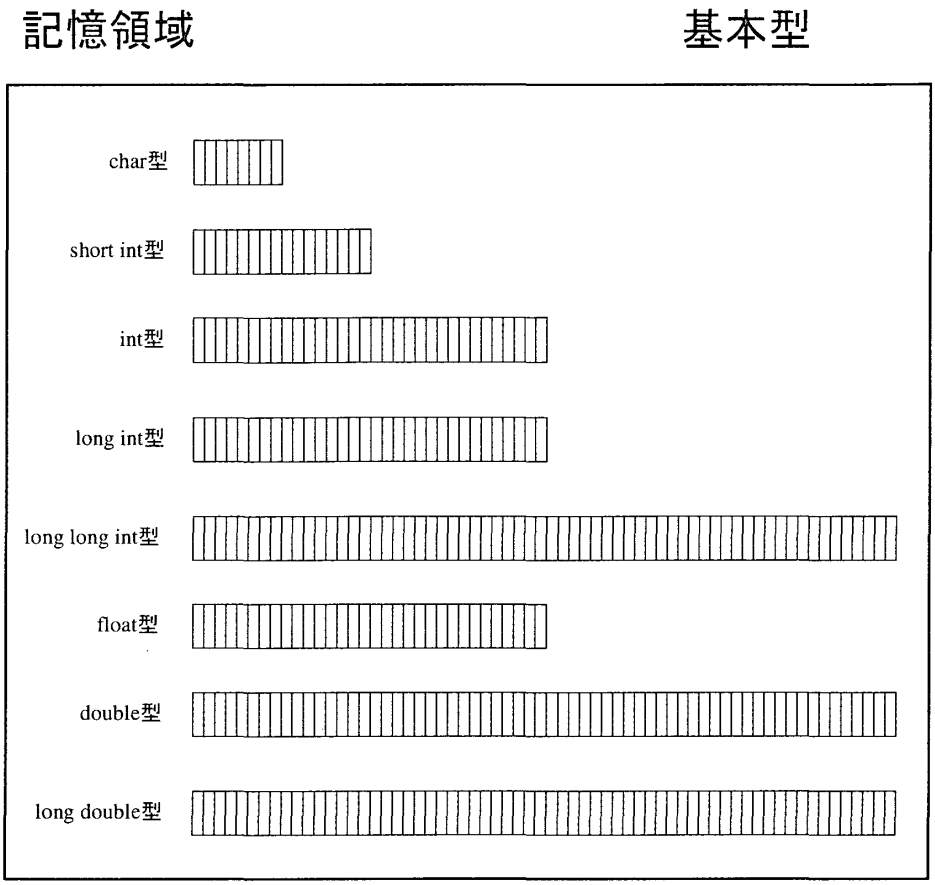

図1．取ることができるデータの基本型(リスト1に対応)を記憶領域(大きく太い四角で囲む)の中に示す。細かい四角一つが1ビットをあらわす。

§3. 配列を仮引数にしたとき

関数という機能を利用する為には、§1で述べたように、関数名の定義、関数の呼び出し、関数の記述のすべてがワンセットで、何らかの形でプログラムに含まれなければならない。関数名の定義は、プログラム中には明示されないで、プリプロセッサで読み込まれるインクルードファイルに存在するかもしれないし、int型の戻り値を持つ関数名は定義が明示されなくても、その関数が呼び出されるだけで暗黙で定義されたものとみなされる。また、関数の記述がプログラム中に無くても、デフォルト・ライブラリないし指定するライブラリからリンカによって実行プログラムに取り込まれるという仕組みもある。これらの結果、関数を呼び出すことがプログラム中に一回あれば、関数の機能が利用できるようになる場合もあることになる。

関数の呼び出しと記述は、コンピュータが実行する命令の執行とその執行内容にそれぞれ対応する。従って、記憶領域をその実行に当たっては利用するが、関数の呼び出しと記述に関するなにかが記憶領域にあることを、プログラムを作成することにとって直接知る必要は無い。このようにいうと、実際のコンピュータで実行される全ての命令が、物理的

媒体としてのメモリにあらかじめ記憶させておいて、それをハードウェアの演算実行装置に呼び出すことによって、コンピュータとして動作するようにしてあるというプログラム内臓方式のものであることを知っている人は不思議に思うかも知れない。しかし、プログラム内臓方式であることを直接利用しなければならないのは、コンピュータ・アーキテクチャを設計するもっと物理的に近い抽象レベルであり、C言語でプログラムを作成する抽象レベルではそのことを詳しく知っていなければならない訳ではない。すなわち、命令と記憶領域という大枠の中で命令の一種として考えるということで括ることができる。

関数の定義は、命令と記憶領域という大枠で考えると、記憶領域に関数の戻り値の箱を確保する命令である。従って、変数や配列の定義と同じ系列の命令である。同じ名前で変数や配列を定義できないのと同様に、関数名の定義も一つしかできないのは当然である。しかし、変数や配列はスコープが異なれば同じ名前のものを定義できるのであるが、関数に関しては、リンカの段階に初めてその名前の関数の呼び出しと記述とが繋げられて解決が図られるのであるから、一つのプログラム中で同じ名前の関数を二つ以上定義できるようにはなれない。この面において、関数の定義は、変数や配列の定義とは異なる性質がある。

関数の呼び出しと関数の記述の間でのデータのやり取りをする仕組みの一つが、引数である。そして、もう一つが戻り値である。関数の呼び出しで指定するのが実引数であり、関数の記述で指定するのが仮引数である。実引数とは、ソースプログラムが確定した段階で、値とか記憶領域での名前が確定していなければならないという意味で、「実」なのである。仮引数は、関数の記述を作成する段階では、それを呼び出して利用する際の実引数が複数あることを前提としているので、値を仮に何か入れるとか記憶領域での仮の名前なので、「仮」なのである。しかし、それらを名前を付けて新たに記憶領域にとるのであって、実体がないという意味で「仮」なのではない。

実引数がある値として確定している場合、仮引数には値渡しが行われる。すなわち、仮引数として名前が付けられ記憶領域での場所を特定されてとられた箱に、実引数の値がコピーされて入れられる。実引数の確定した値が番地データでもこの例外ではない。つまり，仮引数として番地データを受け取れるようにポインタの箱がとられて、実引数の番地データがコピーされて入れられる。これらは、至って単純である。

実引数が配列の名前の場合、仮引数には番地渡しのやり方でデータの受け渡しがなされるとされている。[8] そして、実引数の配列名と同じ記憶領域に、仮引数の配列名が付いているかのように、解説される。以下では、二次元配列である実引数に対応する仮引数の記述の仕方によって、どのように変わるのかどうかを、具体的なプログラムを実行した結果によって見ることにする。着目する点は、仮引数として記憶領域に何がとられるかということである。

まず，最初は，関数main()において、5×4の二次元のchar型配列aを実引数として関数b()を呼び出す。関数b()の記述において、仮引数として5×4の二次元のchar型配列cをとる。関数main()における配列aと、関数b()における仮引数配列cの大きさを、sizeof演算子を用い

て画面にそれぞれを表示させるプログラムである。実行結果とソースプログラムをリスト2に示す。この実行結果は驚くべきものである。関数b()の記述での仮引数としてとったchar

```
[uranus49:~/Documents/paper] tanaka% ./a.out
size of a = 20
address of a= bffffd30
size of c = 4
data of c = bffffd30
[uranus49:~/Documents/paper] tanaka% cat pr1_2.c
#include <stdio.h>

main()
{
   char a[5][4];

   printf("size of a = %d\naddress of a= %x\n",sizeof(a),&a);
   b(a);
}

b(char c[5][4])
{
   c[4][3]='z';
   *(*(c+3)+2)='p';
   printf("size of c = %d\ndata of c = %x\n",sizeof(c),c);
}
[uranus49:~/Documents/paper] tanaka%
```

リスト2．仮引数として配列の形式で取ったときのプログラムを実行した実行結果およびソースプログラムである。

型の配列cは、その大きさがわずか4バイトであるということである。そして、その4バイトの箱に入っているデータは、関数main()でとった配列aの先頭の番地である。関数b()が終了直前での記憶領域の状態を図2に示す。

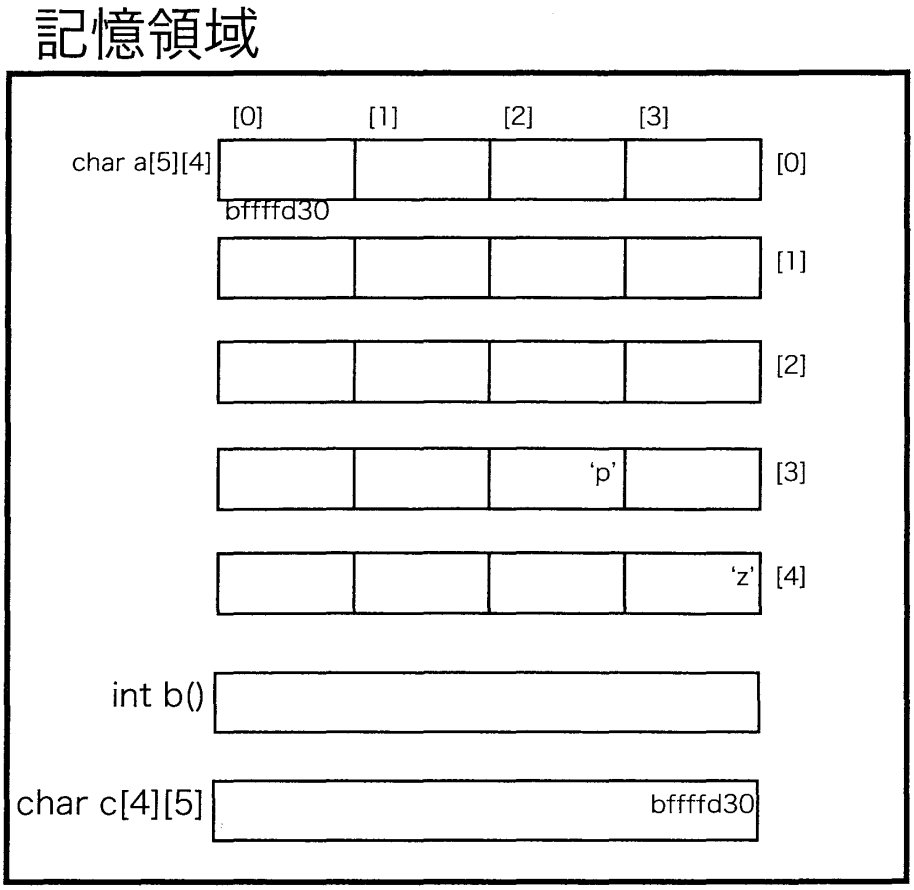

図2. 仮引数として配列の形式で取ったときの記憶領域図(リスト2に対応)である。小さい四角が1バイト、長い四角が4バイトをあらわす。以下の図でも同じである。

次に、関数b()の記述において、仮引数として、二次元ではあるが最初の添字の大きさを省略したchar型配列cをとった。実行結果とソースプログラムをリスト3に示す。関数b()の

```
[uranus49:~/Documents/paper] tanaka% ./a.out
size of a = 20
address of a= bffffd30
size of c = 4
data of c = bffffd30
[uranus49:~/Documents/paper] tanaka% cat pr1_3.c
#include <stdio.h>

main()
{
   char a[5][4];

   printf("size of a = %d\naddress of a= %x\n",sizeof(a),&a);
   b(a);
}
```

```
b(char c[][4])
{
   c[4][3]='z';
   *(*(c+3)+2)='p';
   printf("size of c = %d\ndata of c = %x\n",sizeof(c),c);
}
[uranus49:~/Documents/paper] tanaka%
```

リスト3．仮引数として配列をとるが、行の個数を指定しないときのプログラムを実行した実行結果およびソースプログラムである。

終了直前の記憶領域の様子を図3に示す。リスト2及び図2とこれらをそれぞれ比較すると、実行結果及び記憶領域の様子はほとんど何も変わらない。

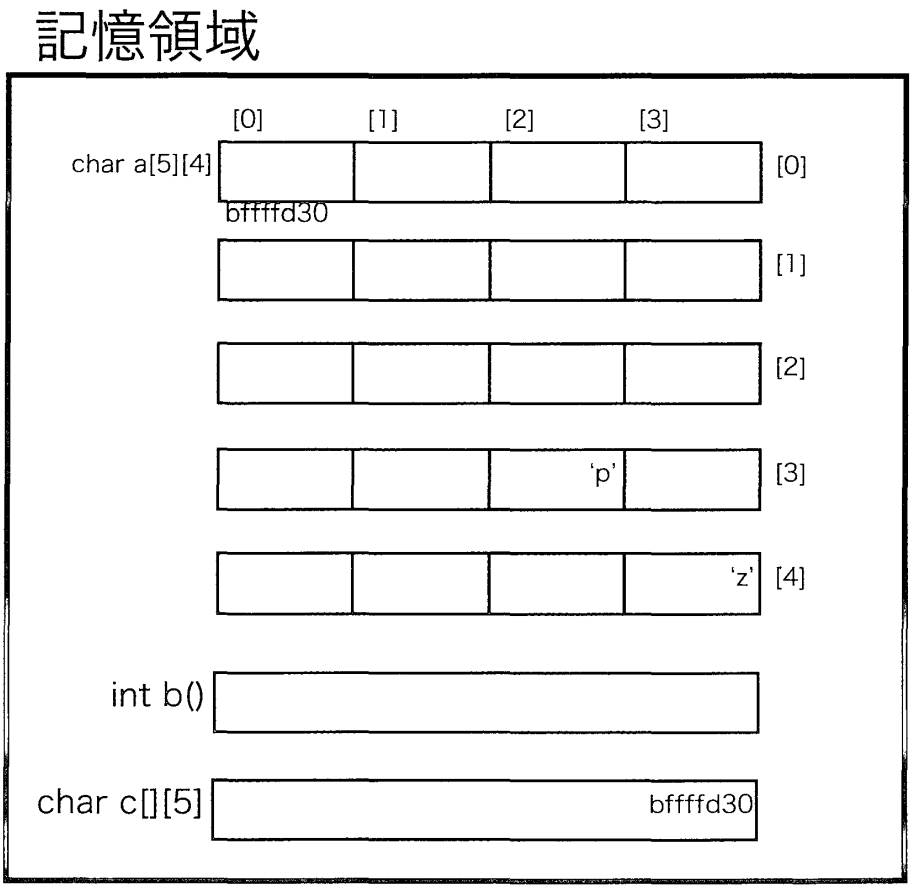

図3．仮引数として配列をとるが、行の個数を指定しないときの記憶領域図(リスト3に対応)である。

さらに、関数b()の記述において、仮引数として、二次元でその添字の大きさを全部省略したchar型配列cとした。実行結果とその実行結果になるソースプログラムをリスト4に示す。また、関数b()の終了直前の記憶領域の様子を図4に示す。この場合、配列cを二次元として扱うことに失敗し、コンパイルエラーになったので、それら行をコメントアウトにした。つまり、列の大きさまでを省略すると、次の行がどこから始まるかの情報が失われて

```
[uranus49:~/Documents/paper] tanaka% ./a.out
size of a = 20
address of a= bffffd30
size of c = 4
data of c = bffffd30
[uranus49:~/Documents/paper] tanaka% cat pr1_4.c
#include <stdio.h>

main()
{
   char a[5][4];

   printf("size of a = %d\naddress of a= %x\n",sizeof(a),&a);
   b(a);
}

b(char c[][])
{
//   c[4][3]='z';
//   *(i*(c+3)+2)='p';
   printf("size of c = %d\ndata of c = %x\n",sizeof(c),c);
}
[uranus49:~/Documents/paper] tanaka%
```

リスト4. 仮引数として配列をとるが、行も列も個数を指定しないときのプログラムを実行した実行結果およびソースプログラムである。

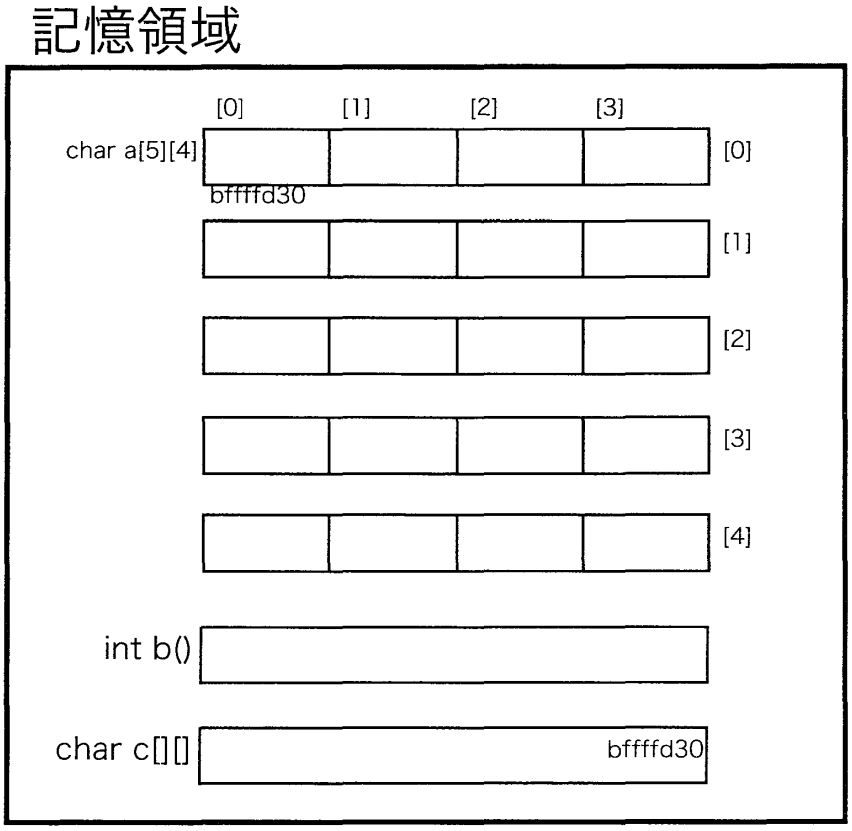

図4. 仮引数として配列をとるが、行も列も個数を指定しないときの記憶領域図(リスト4に対応)である。

しまうようである。その為に、リスト4で示したソースファイルでコメントアウトしてある行のように、二次元配列の要素を直接指定することができなくなったのである。この列の大きさが省略できないことは、どの教科書的書籍にも書かれているが、その多くの著者はその理由を考えたことも無いのではないだろうか。単に、そうしないとうまくいかないとしているのであろう。いずれにしろ、実用的には、全ての次元においてその大きさを省略することは、あまり意味が無いのである。

しかしながら、ただ一つの例外が想定される。それは、実引数としての配列の各行が文字列として完結しており、しかも、各行で文字列の長さが異なる場合は、仮引数を配列として定義する表現としては、ここで述べた通りに列の大きさを省略せざるを得ない。その場合には、行数は確定しており、仮引数としてその情報が関数の記述に伝えられる必要がある。その代表的な事例が、main関数の仮引数の定義である。しかし、C言語仕様より外側に位置するOSなどの他の複雑な事情があるので、ここでは詳しく触れることはしない。その場合においても、仮引数として記憶領域で占める大きさが、4バイトのポインタであることが変わることは無いことに注意するべきである。

最後に、関数b()の記述において、仮引数として、char型ポインタcにした。実行結果とそのソースプログラムをリスト5に示し、また、関数b()の終了直前の記憶領域の様子を図5に示す。この場合は、関数呼び出しの以前には記憶領域において二次元配列であったもの

```
[uranus49:~/Documents/paper] tanaka% ./a.out
size of a = 20
address of a= bffffd30
size of c = 4
data of c = bffffd30
[uranus49:~/Documents/paper] tanaka% cat pr1_5.c
#include <stdio.h>

main()
{
   char a[5][4];

   printf("size of a = %d\naddress of a= %x\n",sizeof(a),&a);
   b(a);
}

b(char *c)
{
   c[19]='z';
   *(c+3*4+2)='p';
   printf("size of c = %d\ndata of c = %x\n",sizeof(c),c);
}
[uranus49:~/Documents/paper] tanaka%
```

リスト5．仮引数としてポインタの形式で取ったときのプログラムを実行した実行結果およびソースプログラムである。

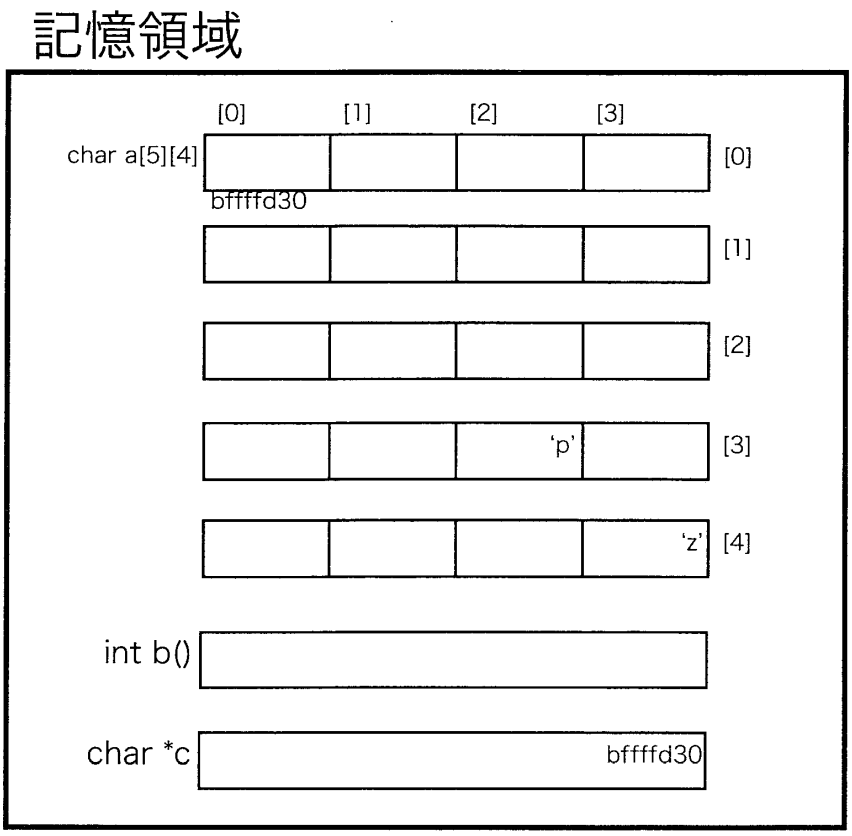

図5. 仮引数としてポインタの形式で取ったときの記憶領域図(リスト5に対応)である。

を、仮引数としてポインタで定義をすると、そのポインタを記憶領域での一次元配列と同等に扱わなければならないことが分かる。

§4. おわりに

本論文においては、関数呼び出しの際に指定する実引数において二次元配列であるものに対応するその関数記述において定義する仮引数を、文法的に許される形式でいろいろの記述を試した。その結果、仮引数名が付けられた記憶領域での箱の大きさが全て4バイトであり、仮引数の文法的表面的な記述形式がいかなるものであっても、仮引数が4バイトの大きさを持っているポインタであることが分かった。そして、そのポインタの箱の中には、実引数の配列の記憶領域の先頭の番地データがコピーされて入っていた。ただし、仮引数の形式によっては、配列としての情報が失われてしまって、実用的に利用し難いことになる場合も生じる事も分かった。また、関数main()の仮引数を解明する為の足がかりになる事柄も得られた。

本研究の発端は、配列として定義された仮引数の記憶領域内での大きさがいかなるものであるかという興味である。そして、それをポインタとして定義しても、さらに、多次元配列の場合に配列としての最上位の添字の大きさを省略しても、文法的に全く問題とならないことが分かった。これらの事項を何回も認識しているうちに、記憶領域というものが、

C言語プログラミングという数多くある抽象レベルの一つであり、コンピュータの構成要素として命令とともに重要であり、図として一貫したやり方でそのイメージを表現すべきであることが分かるようになった。逆に、記憶領域をそういう風に表現をすることが、アルゴリズムをプログラミングに反映する時に要求される精緻な考察にとって有用である事を知るに至った。さらに、教育的にも、学習者に対して具体的イメージを持たせる事ができるという意味で非常に役に立つ。

関数呼び出しの際の実引数から、関数記述の仮引数に対するデータの引き渡し方に、「値渡し」と「番地渡し」があるとされる。しかし、本論文で見たように、二つの引き渡し方があるわけではなく、「値渡し」とされるものは、その型に応じた仮引数に実引数の値コピーされるのであり、「番地渡し」とされるものは、引き渡されるデータが番地データであるので、ポインタの仮引数にそれがコピーされるのである。仮引数の表現がポインタでないものでも文法的に許されるので、認識の混乱が生じているに過ぎない。さらに、実引数において配列名を指定したときの、配列名が指定された意味を番地データが単に与えられたものとして解釈すれば、上で述べた「値渡し」の場合と同じになる。

(2003年12月19日受付、2003年12月24日受理)

参考文献

1) 柴田望洋：「明快C言語　入門編」，ソフトバンク（1990）p290.
2) P.J.Plauger and Jim Brodie：「Standard C プログラマーズ　リファレンス(虎岩登訳)」，アスキー（1994）p120.
3) 橋本和明，山本洋介山，leye：「いまどきのアセンブラプログラミング」，毎日コミュニケーションズ（2003）p267-p479.
4) 三田典玄：「実習C言語」，アスキー（1998）p13.
5) 内田保雄：「基礎からのC言語」，工学社（2003）p228.
6) 田中廣：「C言語ワンポイントマスター　3関数」，日刊工業新聞社（1993）p9.
7) 同上p21.
8) 藤本壱：「Linux/FreeBsd で学ぶC言語」，新紀元社（1999）p213ー p219.
9) 渡邊利和：COMPUTER WORLD（IDGジャパン），1no1（2004）p52.
10) 美多幸夫：「C言語ちょー入門」，広文社（2000）p103.

Abstract

We research what happens or how is process in the memory area while a real argument of function call is an array and virtual arguments of function description are some types in programming with means of C language. The result is that the pointer in real memory area is only taken even in the forms of any types of virtual argument.