システムプログラミング 2
レポート

氏名: 今田　将也 (IMADA, Masaya)
学生番号: 09430509

出題日: 2019 年 12 月 02 日
提出日: 2020 年 01 月 27 日
締切日: 2020 年 01 月 27 日

1　概要

本演習では，SPIM という MIPS CPU シミュレータのハードウェア上に C 言語とアセンブリ言語を使用して文字の表示と入力のためのシステムコールライブラリを作成する． さらに，そのライブラリを使用して printf 関数相当を C 言語で作成する． 最後に，それらを利用した応用プログラムを動作させる．

なお、与えられた課題内容を以下に述べる．

1.1　課題内容

以下の課題についてレポートをする． プログラムは，MIPS アセンブリ言語及び C 言語で記述し，SPIM を用いて動作を確認している．

課題 2-1 SPIM が提供するシステムコールを C 言語から実行できるようにしたい． 教科書 A.6 節 「手続き呼出し規約」に従って，各種手続きをアセンブラで記述せよ． ファイル名は，syscalls.s とすること．また，記述した syscalls.s の関数を C 言語から呼び出すことで， ハノイの塔 (hanoi.c とする) を完成させよ．

`hanoi.c` のソースコード

```
 1: void hanoi(int n, int start, int finish, int extra)
 2: {
 3:   if (n != 0){
 4:     hanoi(n - 1, start, extra, finish);
 5:     print_string("Move disk ");
 6:     print_int(n);
 7:     print_string(" from peg ");
 8:     print_int(start);
 9:     print_string(" to peg ");
10:     print_int(finish);
11:     print_string(".\n");
```

```
12:     hanoi(n - 1, extra, finish, start);
13:   }
14: }
15: main()
16: {
17:   int n;
18:   print_string("Enter number of disks> ");
19:   n = read_int();
20:   hanoi(n, 1, 2, 3);
21: }
```

spim-gcc によって hanoi.s ができたら， hanoi.s, syscalls.s の順に SPIM 上でロードして実行.

実行例は以下の通り:

```
Enter number of disks> 3
Move disk 1 from peg 1 to peg 2.
Move disk 2 from peg 1 to peg 3.
Move disk 1 from peg 2 to peg 3.
Move disk 3 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 1.
Move disk 2 from peg 3 to peg 2.
Move disk 1 from peg 1 to peg 2.
```

課題 2-2 hanoi.s を例に spim-gcc の引数保存に関するスタックの利用方法について，説明せよ. そのことは，規約上許されるスタックフレームの最小値 24 とどう関係しているか． このスタックフレームの最小値規約を守らないとどのような問題が生じるかについて解説せよ.

hanoi.s のソースコード

```
 1          .file   1 "hanoi.c"
 2
 3   # -G value = 0, Arch = r2000, ISA = 1
 4   # GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2)
(mipsel-linux) compiled by GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2).
 5   # options passed:  -mno-abicalls -mrnames -mmips-as
 6   # -mno-check-zero-division -march=r2000 -O0 -fleading-underscore
 7   # -finhibit-size-directive -fverbose-asm
 8   # options enabled:  -fpeephole -ffunction-cse -fkeep-static-consts
 9   # -fpcc-struct-return -fsched-interblock -fsched-spec -fbranch-count-reg
10   # -fnew-exceptions -fcommon -finhibit-size-directive -fverbose-asm
11   # -fgnu-linker -fargument-alias -fleading-underscore -fident -fmath-errno
12   # -mrnames -mno-check-zero-division -march=r2000
13
14
15          .rdata
```

```
16          .align  2
17  $LC0:
18          .asciiz "Move disk "
19          .align  2
20  $LC1:
21          .asciiz " from peg "
22          .align  2
23  $LC2:
24          .asciiz " to peg "
25          .align  2
26  $LC3:
27          .asciiz ".\n"
28          .text
29          .align  2
30  _hanoi:
31          subu    $sp,$sp,24
32          sw          $ra,20($sp)
33          sw          $fp,16($sp)
34          move    $fp,$sp
35          sw          $a0,24($fp)
36          sw          $a1,28($fp)
37          sw          $a2,32($fp)
38          sw          $a3,36($fp)
39          lw          $v0,24($fp)
40          beq         $v0,$zero,$L3
41          lw          $v0,24($fp)
42          addu    $v0,$v0,-1
43          move    $a0,$v0
44          lw          $a1,28($fp)
45          lw          $a2,36($fp)
46          lw          $a3,32($fp)
47          jal         _hanoi
48          la          $a0,$LC0
49          jal         _print_string
50          lw          $a0,24($fp)
51          jal         _print_int
52          la          $a0,$LC1
53          jal         _print_string
54          lw          $a0,28($fp)
55          jal         _print_int
56          la          $a0,$LC2
57          jal         _print_string
58          lw          $a0,32($fp)
59          jal         _print_int
```

```
60          la          $a0,$LC3
61          jal         _print_string
62          lw          $v0,24($fp)
63          addu    $v0,$v0,-1
64          move    $a0,$v0
65          lw          $a1,36($fp)
66          lw          $a2,32($fp)
67          lw          $a3,28($fp)
68          jal         _hanoi
69  $L3:
70          move    $sp,$fp
71          lw          $ra,20($sp)
72          lw          $fp,16($sp)
73          addu    $sp,$sp,24
74          j           $ra
75          .rdata
76          .align  2
77  $LC4:
78          .asciiz "Enter number of disks> "
79          .text
80          .align  2
81  main:
82          subu    $sp,$sp,32
83          sw          $ra,28($sp)
84          sw          $fp,24($sp)
85          move    $fp,$sp
86          la          $a0,$LC4
87          jal         _print_string
88          jal         _read_int
89          sw          $v0,16($fp)
90          lw          $a0,16($fp)
91          li          $a1,1                       # 0x1
92          li          $a2,2                       # 0x2
93          li          $a3,3                       # 0x3
94          jal         _hanoi
95          move    $sp,$fp
96          lw          $ra,28($sp)
97          lw          $fp,24($sp)
98          addu    $sp,$sp,32
99          j           $ra
```

課題 2-3 以下のプログラム report2-1.c をコンパイルした結果をもとに， auto 変数と static 変数の違い，ポインタと配列の違いについてレポートせよ．

```
1: int primes_stat[10];
```

```
 2:
 3: char * string_ptr   = "ABCDEFG";
 4: char   string_ary[] = "ABCDEFG";
 5:
 6: void print_var(char *name, int val)
 7: {
 8:   print_string(name);
 9:   print_string(" = ");
10:   print_int(val);
11:   print_string("\n");
12: }
13:
14: main()
15: {
16:   int primes_auto[10];
17:
18:   primes_stat[0] = 2;
19:   primes_auto[0] = 3;
20:
21:   print_var("primes_stat[0]", primes_stat[0]);
22:   print_var("primes_auto[0]", primes_auto[0]);
23: }
```

課題 2-4 printf など，一部の関数は，任意の数の引数を取ることができる． これらの関数を可変引数関数と呼ぶ． MIPS の C コンパイラにおいて可変引数関数の実現方法について考察し，解説せよ．

課題 2-5 printf のサブセットを実装し， SPIM 上でその動作を確認する応用プログラム (自由なデモプログラム) を作成せよ． フルセットにどれだけ近いか，あるいは，よく使う重要な仕様だけをうまく切り出して， 実用的なサブセットを実装しているかについて評価する． ただし，浮動小数は対応しなくてもよい (SPIM 自体がうまく対応していない)． 加えて，この printf を利用した応用プログラムの出来も評価の対象とする．

1.2 xspim の実行方法

```
$ xspim -mapped_io&
```

でコンソール上で実行後，必要なアセンブリファイルを load し，run することで実行した．

1.3 c ソースコードからアセンブリファイルへの変換方法

```
$ spim-gcc file.c
```

でコンソール上で実行後，file.c に対応する file.s というアセンブリファイルが作られる．

2　課題 2-1

以下に作成したプログラムと，作成内容、また作成時の考察を記載する．

2.1　ハノイの塔について

ハノイの塔とは 3 本の杭と、中央に穴の開いた大きさの異なる複数の円盤から構成され，最初はすべての円盤が左端の杭に小さいものが上になるように順に積み重ねられている。円盤を一回に一枚ずつどれかの杭に移動させることができるが、小さな円盤の上に大きな円盤を乗せることはできないというルールに従いすべての円盤を右端の杭に移動させられれば完成。

解法に再帰的アルゴリズムが有効な問題として有名であり、プログラミングにおける再帰的呼出しの例題としてもよく用いられる。

2.2　プログラムの説明及び作成時の考察

作成は，手続き呼出し規約に基づいて，各ルーチンごとにスタックポインタをルーチンの開始時に確保し，終了時に破棄して呼び出された関数に戻る設計にしている．syscall でカーネルに所望することを$v0 レジスタへ格納し，syscall を呼び出している．

print_int に対応する関数は，4 行目から 13 行目に記載している．print_string に対応する関数は，15 行目から 24 行目に記載している．read_int に対応する関数は，26 行目から 35 行目に記載している．read_string に対応する関数は，37 行目から 46 行目に記載している．exit に対応する関数は，48 行目から 57 行目に記載している．print_char に対応する関数は，59 行目から 68 行目に記載している．read_char に対応する関数は，70 行目から 79 行目に記載している．

なお、今回の hanoi.c には用いないが，文字列をユーザから受け付ける read_string，数値をユーザから受け付ける read_int と文字を表示する print_char と文字をユーザから受け付ける read_char，そして，プログラムを終了する exit を作成した．

作成したプログラム中のラベルの先頭にアンダーバーをつけているがこれは，本演習で用いた gcc のルールでコンパイラに依存するものであるが，アセンブリ中で_function_name と記述しておくと，C 言語から function_name で呼び出すことができるからである．

2.3　作成したプログラム

syscalls.s

```
 1         .text
 2         .align  2
 3
 4 _print_int:
 5         subu  $sp, $sp, 24
 6         sw    $ra, 20($sp)
 7
 8         li    $v0, 1  # 1: print_int
 9         syscall
10
11         lw    $ra, 20($sp)
```

```
12          addu  $sp, $sp, 24
13          j     $ra
14
15  _print_string:
16          subu  $sp, $sp, 24
17          sw    $ra, 20($sp)
18
19          li    $v0, 4  # 4: print_string
20          syscall
21
22          lw    $ra, 20($sp)
23          addu  $sp, $sp, 24
24          j     $ra
25
26  _read_int:
27          subu  $sp, $sp, 24
28          sw    $ra, 20($sp)
29
30          li    $v0, 5  # 5: read_int
31          syscall
32
33          lw    $ra, 20($sp)
34          addu  $sp, $sp, 24
35          j     $ra
36
37  _read_string:
38          subu  $sp, $sp, 24
39          sw    $ra, 20($sp)
40
41          li    $v0, 8  # 8: read_string
42          syscall
43
44          lw    $ra, 20($sp)
45          addu  $sp, $sp, 24
46          j     $ra
47
48  _exit:
49          subu  $sp, $sp, 24
50          sw    $ra, 20($sp)
51
52          li    $v0, 10  # 10: exit
53          syscall
54
55          lw    $ra, 20($sp)
```

```
56          addu  $sp, $sp, 24
57          j     $ra
58
59 _print_char:
60          subu  $sp, $sp, 24
61          sw    $ra, 20($sp)
62
63          li    $v0, 11  # 11: print_char
64          syscall
65
66          lw    $ra, 20($sp)
67          addu  $sp, $sp, 24
68          j     $ra
69
70 _read_char:
71          subu  $sp, $sp, 24
72          sw    $ra, 20($sp)
73
74          li    $v0, 12  # 12: _read_char
75          syscall
76
77          lw    $ra, 20($sp)
78          addu  $sp, $sp, 24
79          j     $ra
```

2.4 実行結果

```
Enter number of disks> 3
Move disk 1 from peg 1 to peg 2.
Move disk 2 from peg 1 to peg 3.
Move disk 1 from peg 2 to peg 3.
Move disk 3 from peg 1 to peg 2.
Move disk 1 from peg 3 to peg 1.
Move disk 2 from peg 3 to peg 2.
Move disk 1 from peg 1 to peg 2.
```

3 課題 2-2

以下に課題内容に対する考察を記載する.

3.1 spim-gcc の引数保存に関するスタックの利用方法

説明のために，以下に `hanoi.s` の冒頭の数行を抜粋する.

表 1: スタックの様子

$sp	offset	内容	備考
新 sp	-24	-	未使用
	-20	-	未使用
	-16	-	未使用
	-12	-	未使用
	-8	$fp	フレームポインタ
	-4	$ra	戻りアドレス
旧$sp	0	$a0	第 1 引数
	+4	$a1	第 2 引数
	+8	$a2	第 3 引数
	+12	$a3	第 4 引数
	+16	??	呼出側で使用
	+20	??	呼出側で使用
	...	??	呼出側で使用

```
30 _hanoi:
31 subu $sp,$sp,24
32 sw     $ra,20($sp)
33 sw     $fp,16($sp)
34 move $fp,$sp
35 sw     $a0,24($fp)
36 sw     $a1,28($fp)
37 sw     $a2,32($fp)
38 sw     $a3,36($fp)
39 lw     $v0,24($fp)
```

31 行目で，スタックを 24 バイト分確保していることが分かる．しかし，35 行目から利用されているレジスタ$a0 $a3 の 4 つは，確保したスタックよりも後方の_hanoi を呼び出した側の関数が確保したスタックを使用している．ここで，新しく関数から呼び出された表 1 にスタックの様子を表に表してみる．MIPS のコンパイラは,1 つ目の引数は$a0 に,2 つ目の引数は$a1 にという具合に$a のレジスタを使って引数を渡すことになっている．しかし，$a0〜$a3 の 4 つしかないため，5 つ目の引数は，スタックに保存して渡す．また，手続き呼出し規約に基づくと，offset =0〜＋15 の領域が必要になる．

まとめると，関数を呼び出す側は$a0〜$a3 を保存する領域を余分に確保しておき，呼び出された側がその領域を使って引数を保存することになっている．

3.2 最小値規約について

最小値規約とは spim-gcc において，規約上許されるスタックフレームの最小値が 24 であるという規約である．全 24 バイトのうち 16 バイトは$a レジスタの 4 語分であり，残りの 8 バイトはフレームポインタに利用される$fp レジスタの 1 語分と戻りアドレスに利用される$ra レジスタである．

この決まりを守らない関数が，仮に呼出される側であった場合は，$a0〜$a3 の保存に自分で確保した領域しか使わないであろうから， 他の関数のスタック領域を破壊することがない．そのため，gcc から呼出しても問題がない．しかし，逆の場合, すなわち呼び出す側だった場合には，自分の関数のために確保したスタックを呼出し先が破壊することになるという問題がある．

この方法には利点がある．

利点 1 被呼出し関数が $a0〜$a3 の保存をするかしないかを決定できるので， 関数内で$a0〜$a3 を書換えなければ，この保存は省略できるため，メモリへの書込み処理が減り，高速化が望める．$a0〜$a3 を呼び出す側で保存することにしてしまうと，上記の 4 つの引数をメモリに格納する操作が必ず必要になる．これでは，引数をレジスタ渡しではなく，実体として渡していることになる．

利点 2 第 5 引数以降が第 4 引数までの確保領域と連続するため，被呼出し関数から見れば，第 1 引数からのすべての引数が規則正しくメモリ上に並ぶことになる．そのため，コンパイラの実装が容易になる．

C 言語との連携には，この規約を守る必要があるため， 最小のスタックフレームサイズは，24 バイトとなっている．(引数 1 つ目〜4 つ目 ($a0〜$a3),$ra,$fp の 6 レジスタ*4 バイト = 24 バイト)

4 課題 2-3

以下に課題内容に対する考察を記載している．先に，auto 変数と static 変数の違いについて述べた後にポインタと配列の違いについて C 言語とアセンブリの観点から述べる．

4.1 report2-1.c のコンパイル結果

```
 1 .file 1 "report2-1.c"
 2
 3  # -G value = 0, Arch = r2000, ISA = 1
 4  # GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2) (mipsel-linux)
compiled by GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2).
 5  # options passed:  -mno-abicalls -mrnames -mmips-as
 6  # -mno-check-zero-division -march=r2000 -O0 -fleading-underscore
 7  # -finhibit-size-directive -fverbose-asm
 8  # options enabled:  -fpeephole -ffunction-cse -fkeep-static-consts
 9  # -fpcc-struct-return -fsched-interblock -fsched-spec -fbranch-count-reg
10  # -fnew-exceptions -fcommon -finhibit-size-directive -fverbose-asm
11  # -fgnu-linker -fargument-alias -fleading-underscore -fident -fmath-errno
12  # -mrnames -mno-check-zero-division -march=r2000
13
14
15 .rdata
16 .align 2
17 $LC0:
18 .asciiz "ABCDEFG"
19 .data
```

```
20 .align 2
21 _string_ptr:
22 .word $LC0
23 .align 2
24 _string_ary:
25 .asciiz "ABCDEFG"
26 .rdata
27 .align 2
28 $LC1:
29 .asciiz " = "
30 .align 2
31 $LC2:
32 .asciiz "\n"
33 .text
34 .align 2
35 _print_var:
36 subu $sp,$sp,24
37 sw $ra,20($sp)
38 sw $fp,16($sp)
39 move $fp,$sp
40 sw $a0,24($fp)
41 sw $a1,28($fp)
42 lw $a0,24($fp)
43 jal _print_string
44 la $a0,$LC1
45 jal _print_string
46 lw $a0,28($fp)
47 jal _print_int
48 la $a0,$LC2
49 jal _print_string
50 move $sp,$fp
51 lw $ra,20($sp)
52 lw $fp,16($sp)
53 addu $sp,$sp,24
54 j $ra
55 .rdata
56 .align 2
57 $LC3:
58 .asciiz "primes_stat[0]"
59 .align 2
60 $LC4:
61 .asciiz "primes_auto[0]"
62 .text
63 .align 2
```

```
64 main:
65 subu $sp,$sp,64
66 sw $ra,60($sp)
67 sw $fp,56($sp)
68 move $fp,$sp
69 li $v0,2 # 0x2
70 sw $v0,_primes_stat
71 li $v0,3 # 0x3
72 sw $v0,16($fp)
73 la $a0,$LC3
74 lw $a1,_primes_stat
75 jal _print_var
76 la $a0,$LC4
77 lw $a1,16($fp)
78 jal _print_var
79 move $sp,$fp
80 lw $ra,60($sp)
81 lw $fp,56($sp)
82 addu $sp,$sp,64
83 j $ra
84
85 .comm _primes_stat,40
```

4.2 C 言語から見た static と auto の違い

説明のために，以下に課題の C のソースコードを一部抜粋する．

```
 1 int primes_stat[10];
 2 char * string_ptr   = "ABCDEFG";
 3 char   string_ary[] = "ABCDEFG";
 4 main()
 5 {
 6 int primes_auto[10];
 7 primes_stat[0] = 2;
 8 primes_auto[0] = 3;
 9 print_var("primes_stat[0]", primes_stat[0]);
10 print_var("primes_auto[0]", primes_auto[0]);
11 }
```

1 行目の関数外で宣言されている変数は，static(静的) 変数である．また，5 行目の関数内で宣言されている変数は，auto(自動) 変数であるという．以下にそれぞれの変数の特徴を示してみる．

auto 変数 関数の中で宣言され，その関数の実行開始時から 終了時までの間，その値を保持する．

static 変数 プログラムの開始から終了まで，値を保持しつづける．

両者の違いをアセンブラのソースコードを元に次節から調べてみる．

4.3 アセンブリにおける auto 変数

ソースコード内の auto 変数である `primes_auto` は 1.1 節の `report2-1.c` には明らかに区別されて存在している．しかし，アセンブリのソース 4.1 節からは該当の部分を簡単に発見はできなかった．60 行目にある文字列からラベル`$LC4` が使われているところを辿ってみると，78 行目において呼び出している `_print_var` の第 2 引数の内容が `primes_auto[0]` の値だと推測した．77 行目の`$a1` に入っている値すなわち，`16($fp)` のことである．つまり，新`$sp+16` バイト目であり，スタック上に存在していることになる．

そして，82 行目の操作によって，スタックを解放しているためこれ以降は値が使えなくなる．main 関数における自動変数宣言は main 関数の終了とプログラムの終了がほぼ同じような意味を持つため，意識をする必要はないように思う．以下にスタックの様子を示す．

表 2: スタックの様子

$sp	offsset	内容	備考
	-16	primes_auto	新$sp + 16 バイト目
..	..	..	..
新$sp →	-04	$ra	戻りアドレス
..	..	..	..
旧$sp →	+00	$a0	第 1 引数
	+04	$a1	primes_auto[0] の値

4.4 アセンブリにおける static 変数

`report2-1.c` をアセンブリに変えたコンパイル結果より，85 行目にて以下の記述を見つけた．

```
85 .comm _primes_stat,40
```

この宣言でデータセグメント内にデータを 40 バイト確保していた．これは，`_primes_stat` のみ仕様されるもので，プログラムの開始から終了まで，値を保持しつづけるという性質を持つことになる．`primes_stat` は常にその領域しか使用しないので，関数などが再帰的に呼び出された場合は，その領域を上書きすることがある．

そのため，プログラムの開始から終了まで，値を保持しつづける一方で， 固定された領域 (static な領域) のみを使用するので， 再帰やスレッドによる並行処理では，上書きの危険があるといえる．

4.5 C 言語における static というキーワード

C 言語において static という言葉は，2 つの意味を持っていた．1 つは，スタック上ではなく，プログラム中に静的に存在する領域にデータを確保するという意味である．もう 1 つは，static を付けると変数が外部から参照できる範囲が変化するということである．具体的には，関数外で static を付けて宣言した変数は，外部のファイルからは参照できない．簡単にいうと，複数の C 言語のファイルから構成されるプログラムにおいて，あるファイル内だけからしか参照できない変数を宣言できる．

static は，関数内でも有効に働くので，その場合は変数の有効範囲ではなく，記憶クラスを指定する．以下にその宣言と解釈した内容を表に示す．

表 3: static と int

宣言例	スコープ：見える範囲	記憶クラス（寿命）
static int a;（関数内）	関数内	静的（プログラム中）
static int a;（関数外）	ファイル全体	静的（プログラム中）
int a;（関数内）	関数内	自動（関数中）
int a;（関数外）	プログラム全体	静的（プログラム中）

4.6 ポインタと配列の C 言語での違い

C 言語でのポインタと配列の違いについて 4.6.1 節のソースコードを作り考察した．

配列 array とポインタ変数 pointer の値を表示する 9 行目と 10 行目の結果はいずれも同じであった．つまり配列は配列名だけだと，その配列の先頭アドレスを指すという事がわかる．すなわち，pointer と array で array 配列の値には同じようにアクセスすることができる．これは，12・13 行目，15・16 行目を表示した結果からわかる．

続いて，pointer と array のアドレスを見てみた．すると，両者は異なっていたが，&array と array は同じ値になっていた．一方，pointer は別のアドレスから配列 array の先頭アドレスを指していた．これは，ポインタにはアドレスを保存するメモリがあるが，配列にはアドレスを格納するメモリがないと言えるだろう．つまり，C 言語においてポインタはアドレスを格納する変数であるのに対し、配列は単なるアドレスであると考える．

4.6.1 作成したプログラム

```
 1 #include<stdio.h>
 2
 3 int main(void){
 4     char array[3] ='abc';//char 型 3 つ分と
 5     char *pointer;//char* 型 1 つ分のメモリが確保
 6
 7     pointer = array;//ポインタ変数が配列の先頭アドレスを指す
 8
 9     printf("array   = %p\n", array);
10     printf("pointer = %p\n", pointer);
11
12     printf("array[2]   = %c\n", array[2]);
13     printf("pointer[2] = %c\n", array[2]);
14
15     printf("*array   = %c\n", *array);
16     printf("*pointer = %c\n", *pointer);
17
18     printf("&array   = %p\n", &array);
19     printf("&pointer = %p\n", &pointer);
20
21     return 0;
22 }
```

4.6.2 出力結果

```
array   = 0061FF1D
pointer = 0061FF1D
array[2]   = c
pointer[2] = c
*array   = a
*pointer = a
&array   = 0061FF1D
&pointer = 0061FF18
```

4.7 ポインタと配列のアセンブラでの違い

アセンブラでの違いについて report2-1.s を見てみる．すると，17 行目から 27 行目にその違いが現れていた．ポインタで宣言した string_ptr は.word というアセンブリ指令にて，32 ビットの数値をメモリに順番に配置されている．その数値は，"ABCDEFG"ではなく，そのワードが示すラベルのアドレスが格納されている．一方，配列として宣言した string_ary には"ABCDEFG"というデータ自体が格納されている．

4.8 考察

以上のコードより，配列とは、多数の変数を順番つけでまとめて扱う方法で，ポインタとは、変数のショートカットを作る方法であると考える。ポインタと配列が似たような使い方が出来るのは配列の設計と関係あるのではないかと考えた。実際，C 言語では、配列を実現する手段として、ポインタを利用している。従って、ポインタ変数では、配列と同等のことが出来ると考える．

5 課題 2-4

5.1 概要

この節では，まずはじめに可変引数について説明し，その後 C 言語における可変引数関数の実現方法と，MIPS における可変引数の実現方法について考察し，解説を行う．

5.2 可変引数とは

可変引数とはプログラミング言語において，関数やメソッドやマクロの引数が固定ではなく任意の個数となっている引数のことである．可変長引数，可変個引数とも呼ばれる．そのような関数を可変長引数関数と言う．C 言語では，可変長の引数を扱うために，…を使った構文が用意されている．例えば以下のような記述である．

```
int myfunction(char *fmt, ...)
```

第 2 以降の引数の個数は不定で，0 個でも構わない．代表的な使用例としては，printf がある．

5.3 C 言語における可変引数関数の実現方法

可変引数を宣言した関数ではいくつか疑問がある．

疑問 1 呼び出された関数内で，引数をどう参照すればいいのか．第 1 引数は，変数名で参照できそうだが，第 2 引数以降を名前で参照することができない．

疑問 2 いくつの引数が呼ばれたかをどう判断するのか．また，それぞれの引数の型をどうやって知ればよいのか．

5.3.1 疑問 1 の考察

ここで再度，課題 2-1 の `hanoi()` 関数の冒頭を見てみる．

```
30 _hanoi:
31 subu $sp,$sp,24
32 sw     $ra,20($sp)
33 sw     $fp,16($sp)
34 move $fp,$sp
35 sw     $a0,24($fp)
36 sw     $a1,28($fp)
37 sw     $a2,32($fp)
38 sw     $a3,36($fp)
39 lw     $v0,24($fp)
```

上記の出力内容と表 1 より，第 2 引数は (旧`$sp` + 04) からの 4 バイトに順次格納されている．つまり，C 言語で可変引数関数を記述して第 2 引数以降の値を得ようとすると，(旧`$sp` + 04) の値を C 言語で取得する必要がある．しかし，C 言語からレジスタの値を直接得る方法がわからない．そこで，第 1 引数が名前で参照できることを利用する．

5.2 節の例の旧`$sp` ＝第 1 引数のアドレスすなわち`&fmt` となることから第 2 引数のアドレス＝`&fmt` から 4 バイト先として求められる．よって，第 n 引数のアドレス＝`&fmt` から 4 ×（ n − 1 ）バイト先として求められそうである．

5.3.2 疑問 2 の考察

前節で第 2 引数のアドレスを知る方法がわかった．しかし，C 言語で`&fmt` はポインタとして扱われるため，正確にはアドレスとは異なる．よって C 言語で記述するなら以下のようになる．

第 2 引数のアドレス `= ((char*)&fmt) + ((sizeof(fmt) + 3) / 4) * 4`

C 言語で (あるポインタ)+1 が実際のアドレスとしていくつ増えるかはポインタが指す型によって異なる．つまり，あるポインタ `p` のアドレスが 5000 のとき，p の型が `int*`の場合は，p+1 は 5004 である．具体的には，5000+`sizeof(int)` また，p の型が `char*`の場合は p+1 は 5001 である．この仕組みのおかげで， *(p+1) とした場合に p の型に基づいて，適切なアドレスから正しい値を取り出すことができる．

char*という型は，p+1 がそのままアドレス上で 1 増える．そのため，(char*)&fmt と型を指定することで，値をアドレスと同じように，+1 がそのままアドレスの+1 に相当する操作ができる．第 2 引数は，((sizeof(fmt)+3)/4)*4 バイト分先にあるので， 上記の式になる．

単なる sizeof(fmt) ではないのは，MIPS の gcc では引数の sizeof が 3 以下の場合は，4 の倍数に切り上げるようにメモリを使って引数を配置するので，それを考慮して，((x+3)/4*4) という操作をする必要がある．このようにして利用することで第 2 引数の値をレジスタ$a2 に得ることができる．

```
第 2 引数のアドレス = ((char*)&fmt) + ((sizeof(fmt) + 3) / 4) * 4;
レジスタ$a2 = *(int*) 第 2 引数のアドレス;
```

p2 は char* 型であるので，実際に中身を取り出す場合は，第 2 引数の型の ポインタにキャストしておく必要があります． つまり第 2 引数が int の場合は， a2 を int 型として上記のようになるわけです．同様に，第 3 引数以降も

```
第 3 引数のアドレス  = 第 2 引数のアドレス + ((sizeof(第 2 引数の型) + 3) / 4) * 4;
レジスタ$a3 = *(第 3 引数のポインタ型) 第 3 引数のアドレス;
```

となる

5.4 MIPS における可変引数の実現方法

GCC でどのようにコンパイルされているかマクロを用いて MIPS で調査を行った．

5.4.1 作成したプログラムの概要

関数 sum() は，可変長引数で int 型の値を読み込み，その総合計を算出するプログラムで，入力値が 0 ならば終了する．関数 call_sum() は，配列の要素を関数 sum() に渡すための関数である．ここでは 10 個の配列の値を渡す．

5.4.2 作成したプログラム

```
 1 #include <stdarg.h>
 2 int sum(int nfirst, ...)
 3 {
 4   int r = 0, n;
 5   va_list args;
 6
 7   va_start(args, nfirst);
 8   for (n = nfirst; n != 0; n = va_arg(args, int)) r += n;
 9   va_end(args);
10
11   return r;
12 }
```

```
13  int call_sum(int a[10])
14  {
15    return sum(a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]);
16  }
```

5.4.3 MIPS へのコンパイル結果

```
 1          .file   1 "mips_hikisu.c"
 2
 3   # -G value = 0, Arch = r2000, ISA = 1
 4   # GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2)
(mipsel-linux) compiled by GNU C version 2.96 20000731 (Red Hat Linux 7.3 2.96-113.2).
 5   # options passed:  -mno-abicalls -mrnames -mmips-as
 6   # -mno-check-zero-division -march=r2000 -O0 -fleading-underscore
 7   # -finhibit-size-directive -fverbose-asm
 8   # options enabled:  -fpeephole -ffunction-cse -fkeep-static-consts
 9   # -fpcc-struct-return -fsched-interblock -fsched-spec -fbranch-count-reg
10   # -fnew-exceptions -fcommon -finhibit-size-directive -fverbose-asm
11   # -fgnu-linker -fargument-alias -fleading-underscore -fident -fmath-errno
12   # -mrnames -mno-check-zero-division -march=r2000
13
14
15          .text
16          .align  2
17  _sum:
18          sw      $a0,0($sp)
19          sw      $a1,4($sp)
20          sw      $a2,8($sp)
21          sw      $a3,12($sp)
22          subu    $sp,$sp,24
23          sw      $fp,16($sp)
24          move    $fp,$sp
25          sw      $a0,24($fp)
26          sw      $zero,0($fp)
27          addu    $v0,$fp,28
28          sw      $v0,8($fp)
29          lw      $v0,24($fp)
30          sw      $v0,4($fp)
31  $L3:
32          lw      $v0,4($fp)
33          bne     $v0,$zero,$L6
34          j       $L4
35  $L6:
36          lw      $v1,0($fp)
```

```
37          lw      $v0,4($fp)
38          addu    $v0,$v1,$v0
39          sw      $v0,0($fp)
40          lw      $v0,8($fp)
41          addu    $v1,$v0,3
42          li      $v0,-4                  # 0xfffffffc
43          and     $v0,$v1,$v0
44          sw      $v0,8($fp)
45          lw      $v0,8($fp)
46          addu    $v0,$v0,0
47          move    $v1,$v0
48          lw      $v0,8($fp)
49          addu    $v0,$v0,4
50          sw      $v0,8($fp)
51          lw      $v0,0($v1)
52          sw      $v0,4($fp)
53          j       $L3
54  $L4:
55          lw      $v0,0($fp)
56          move    $sp,$fp
57          lw      $fp,16($sp)
58          addu    $sp,$sp,24
59          j       $ra
60          .align  2
61  _call_sum:
62          subu    $sp,$sp,48
63          sw      $ra,44($sp)
64          sw      $fp,40($sp)
65          move    $fp,$sp
66          sw      $a0,48($fp)
67          lw      $a0,48($fp)
68          lw      $v0,48($fp)
69          addu    $a1,$v0,4
70          lw      $v0,48($fp)
71          addu    $a2,$v0,8
72          lw      $v0,48($fp)
73          addu    $v1,$v0,12
74          lw      $v0,48($fp)
75          addu    $v0,$v0,16
76          lw      $v0,0($v0)
77          sw      $v0,16($sp)
78          lw      $v0,48($fp)
79          addu    $v0,$v0,20
80          lw      $v0,0($v0)
```

```
 81          sw      $v0,20($sp)
 82          lw      $v0,48($fp)
 83          addu    $v0,$v0,24
 84          lw      $v0,0($v0)
 85          sw      $v0,24($sp)
 86          lw      $v0,48($fp)
 87          addu    $v0,$v0,28
 88          lw      $v0,0($v0)
 89          sw      $v0,28($sp)
 90          lw      $v0,48($fp)
 91          addu    $v0,$v0,32
 92          lw      $v0,0($v0)
 93          sw      $v0,32($sp)
 94          lw      $v0,48($fp)
 95          addu    $v0,$v0,36
 96          lw      $v0,0($v0)
 97          sw      $v0,36($sp)
 98          lw      $a0,0($a0)
 99          lw      $a1,0($a1)
100          lw      $a2,0($a2)
101          lw      $a3,0($v1)
102          jal     _sum
103          move    $sp,$fp
104          lw      $ra,44($sp)
105          lw      $fp,40($sp)
106          addu    $sp,$sp,48
107          j       $ra
```

5.4.4 考察

sum() の内容から，まず,MIPS は引数用に利用できるレジスタが a0 から a3 までの 4 つなので,a1 から a3 をスタックに格納している様子が 1 7 行目以降でわかる．さらに、そこからスタックの先には 4 番目の引数以降が格納されていると考えることができる. そして 1 つずつデータをロードし，処理を行っていた．

６ 1 行目以降の call_sum() が呼び出されてからは，スタックを引数の個数分（４ ８）確保し，戻りアドレスを渡す a0 を除いた a1 から引数の値が順次読み出されていた．しかし，a3 については利用されておらず v1 が利用されている原因はわからなかった．第 4 引数以降は 77 行目，81 行目などより 4 ずつずらして引数を読み取っていることから，引数がアドレス的に続いていることが確認でき，MIPS では引数は最大限用意されているレジスタを利用し以降はスタックを利用することで可変引数の実装を実現していることを確認した．

6　課題 2-5

本節ではまず作成したプログラムを紹介した後に，作成する際のプログラムの作成方針と作成したプログラムに対する考察を述べる．また，その動作を確認した応用的なテストプログラムの結果とそれに対する考察も行う．

6.1　作成したプログラム

以下が作成したプログラムである．

```
 1  #define ROUNDUP_SIZEOF(x) (((sizeof(x)+3)/4)*4)
 2
 3  #define fill_zero         (1<<1)//000001 シフト演算
 4  #define alternative          (1<<2)//000010
 5  #define three_div       (1<<3)//000100
 6  #define capital       (1<<4)//001000
 7  #define with_sign       (1<<5)//010000
 8  #define left_start       (1<<6)//100000
 9
10  #define _isnumc(x) ( (x) >= '0' && (x) <= '9' )
11  #define _ctoi(x)   ( (x) -  '0' )//0という文字を基準として引き算すれば数字文字を示す数値になる
12
13  int my_strlen(char* str){
14      int length = 0; //文字列の長さを入れる箱
15
16     //文字列の長さを数える
17      while(*str++ != '\0'){
18          length++;
19      }
20      return length;
21  }
22
23  char * mystrchr(const char *s, int c)
24  {//*s に対応する文字を検索する　c は数値で持ってくる．そのほうができた．
25      char ch = (char) c;
26      while (*s) {
27          if (*s == ch)
28              return (char *) s;
29          s++;
30      }
31      return '\0';
32  }
33
34  void print_char(char c){
```

```
35      //ヌル文字が格納されいなければならない
36      //文字単体は asciiz でアセンブリで扱われないからヌル文字が入らない。その処理
37      //null 入れなくても動いた
38      char s[2];
39      s[0]=c;
40      s[1]='\0';
41      print_string(s);
42  }
43
44  void put_int(int n, int base, int length, char sign, int flags){
45
46      char *symbols_s = "0123456789abcdef";
47      char *symbols_c = "0123456789ABCDEF";
48      char buf[80];
49      int i = 0;
50      int pad = ' ';
51      char *symbols = symbols_s;
52
53      if(flags & capital){
54          symbols = symbols_c;
55      }
56
57      do {
58          buf[i++] = symbols[n % base];
59          if( (flags & three_div) && (i%4)==3) buf[i++] = ',';
60      } while (n /= base);
61
62      length = length - i;
63
64      if (!(flags & left_start)) {
65          if(flags & fill_zero){
66              pad = '0';
67          }
68          while (length > 0) {
69              length--;
70              buf[i++] = pad;
71          }
72      }
73
74      if (sign && base == 10){
75          buf[i++] = sign;
76      }
77
78      if (flags & alternative){
```

```
 79          if (base == 8){
 80              buf[i++] = '0';
 81          }
 82          else if (base == 16){
 83              buf[i++] = 'x';
 84              buf[i++] = '0';
 85          }
 86      }
 87
 88      while (i > 0){
 89          print_char(buf[--i]);
 90      }
 91
 92      while (length>0){
 93          length--;
 94          print_char(pad);
 95      }
 96
 97  }
 98
 99  void myprintf(char *fmt, ...){
100      //第２引数以降を格納するために利用する。
101      //fmt のアドレスに fmt のサイズ分追加する。char として扱うためにキャスト。なくてもうごいた
102      char *p = ((char*)&fmt)+ROUNDUP_SIZEOF(fmt);
103
104      while(*fmt){
105
106          int flags = 0;
107          int length = 0;
108          int precision = 0;
109          int tmp = 0;
110          char sign = '\0';
111          char *s = '\0';
112
113          if(*fmt == '%'){
114              fmt++;//次を見る
115
116              while (mystrchr("'-+#0", *fmt)) {
117                  switch (*fmt) {
118                  case '\'':
119                      flags |= three_div;
120                      break;
121                  case  '-':
```

```
122                     flags |= left_start;
123                     break;
124                 case  '+':
125                     flags |= with_sign;
126                     sign = '+';
127                     break;
128                 case  '#':
129                     flags |= alternative;
130                     break;
131                 case  '0':
132                     flags |= fill_zero;
133                     break;
134                 }
135                 fmt++;
136             }
137
138             while( _isnumc(*fmt)){
139                 length = (length*10)+_ctoi(*fmt++);
140             }
141
142             if (*fmt == '.'){
143                 fmt++;//次の数字を見る
144                 while (_isnumc(*fmt) ){
145                     precision = precision * 10 + _ctoi(*fmt++);
146                 }
147             }
148
149             switch(*fmt){
150             case 'd':
151             case 'i':
152                 //print_int(*(int*)p);//pの中身の値をintとしてキャストし表示
する
153                  if(*(int*)p < 0){
154                      *(int*)p *= -1;//そのままマイナスだと表示されない
155                   sign = '-';
156                 }
157                 put_int(*(int*)p,10,length,sign,flags);
158                 p = p + ROUNDUP_SIZEOF(int);
159                 break;
160             case 's':
161                 //print_string(*(char**)p);
162                 s = *(char**)p;
163                 if(s == '\0'){
164                     s = "(null)";
```

```
165                 }
166                 tmp = my_strlen(s);
167                 if (precision && precision < tmp){
168                     tmp = precision;//左 precision が 0 なら偽
169                 }
170                 length = length - tmp;
171                 if (!(flags & left_start)){
172                     while ( length > 0 ){
173                         length--;
174                         print_char(' ');
175                     }
176                 }
177                 while (tmp--){
178                     print_char(*s++);
179                 }
180                 while (length > 0){
181                     length--;
182                     print_char(' ');
183                 }
184                 p = p + ROUNDUP_SIZEOF(char*);
185 
186                 break;
187             case 'c':
188                 print_char(*(char*)p);
189                 p = p + ROUNDUP_SIZEOF(char);
190                 break;
191             case '%':
192                 print_char('%');
193                 break;
194             case 'X':
195                 flags |= capital;
196             case 'x':
197                 put_int(*(int*)p,16,length,sign,flags);
198                 p= p + ROUNDUP_SIZEOF(int);
199                 break;
200             case 'o':
201                 put_int(*(int*)p,8,length,sign,flags);
202                 p=p+ROUNDUP_SIZEOF(int);
203                 break;
204             }
205         }
206 
207         else{
208             print_char(*fmt);
```

```
209             }
210             fmt++;
211         }
212     }
213
214     int main()
215     {
216         myprintf("TEST\n");
217         myprintf("%%d   :%d\n%%5d  :%5d\n%%-5d :%-5d\n",100,100,100);
218         myprintf("%%5.2d:%5.2d\n",100,100,100);
219         myprintf("%%#x  :%#x\n%%X   :%X\n",15,15);
220         myprintf("%%#o  :%#o\n",15);
221         myprintf("%%s   :%s\n%%5s  :%5s\n%%5.2s:%5.2s\n","Say","Say","Say");
222         myprintf("%%c   :%c\n",'a');
223         myprintf("%%'d  :%'d\n",10000);
224         return 0;
225     }
```

6.2 作成方針

printf のサブセットを作成するにあたり，浮動小数は対応する必要がなかったため以下のサブセットを実装することにした. また，すべて可変引数関数に対応させなくてはならない.

1. %d および%i による正負の値の表示．また，それに付随し，最小表示桁数の表示%5d などに対応させたもの.

2. %x による 10 進数値を 16 進数で表示させるもの.

3. %o による 10 進数を 8 進数で表示させるもの.

4. %s での文字列の表示. また，それに付随し，最小表示文字数と、表示文字数の制限に対応させたもの. 例として，%5.2s といったものである.

5. %c での文字の表示.

6. %%での%のエスケープ.

7. 0 で空白を埋めたり，左詰めで表示させること.

6.3 考察

まず，%d,%s,%c の実装を行った. 表示桁数などは考慮せずに純粋に，可変引数から取得したものを表示し，その後次の引数を見るという動作を while() の中で行っている. 他のサブセットについても同様で，case 文による分岐で修飾指定子を判断するようにした.

次に，%x,%o の対応を行った.10 進数からの基底変換は，10 進数の余りと商から分かるため以下のように設定し，対応する文字を表示させることにした.

```
if (n >= base) radix_print(n / base, base);
 putchar("0123456789ABCDEF"[n % base]);
```

なお，複雑な修飾子の対応ができなかった，また，8 進数や 16 進数での表示も，10 進数と基底が違うだけなのでまとめることはできないかと考え，この方針ではない方法で改良を行ったため以降で説明をする．

6.4 関数の拡張１：フラグについて

一通り終えたところで，各種サブセットの拡張に対応させることを始めるためにいくつか準備をおこなった．

まず，様々な拡張に必要な及びオリジナルで追加した機能のためのフラグを定義した．

fill_zero 0 で埋めるためのフラグ．数値を表示する際に用いる．

alternative 8 進数と 16 進数を明示的に表示するためのもの．

three_div 三桁ごとにカンマを入れるかどうかのフラグ．あれば便利なため機能を追加した．

capital 16 進数で文字を大文字にするかどうかのフラグ．文字列を入れ替えるために使う．

with_sign 正の値ならば数値の前にプラスの符号を付与するためのフラグ．

left_start 左寄せで表示するためのフラグ．

6.5 関数の拡張２：別に定義した関数について

6.5.1 my_strlen

まず，関数 `my_strlen()` についてだが，これは受け取った文字列の文字数をカウントする．

6.5.2 mystrchr

次に，関数 `mystrchr()` について，これは受け取った文字列がもともと指定されている文字列の中に対応する文字があるかどうか調べる．あればその文字を返し，なければヌル文字を返す．文字に対する操作を行う．

6.5.3 print_char

関数 `print_char()` は，受け取った文字の後にヌル文字を入れ，文字列へと変えて 1 文字ずつ表示することを担っている．

6.5.4 put_int

次に，関数 put_int() だが，なぜこれを作ったか説明すると，10 進数も 8 進数も 16 進数も基底が異なるだけで数値を表示することには変わりない. また，10 進数を 10 で，10 進数を 8 で，10 進数を 16 で商や余りを出すことでそれぞれ基底変換ができると思ったため，すべての操作を基底をかえるだけで行えると良いと考えたからである.

put_int() の動作について説明をする. 変数 symbols_s，symbols_c は，基底に対する文字を取得する際に利用する. また，カンマや正負の符号は文字であるから数値なども含めて文字列として扱えるように buf[],i を利用している. 変数 pad には文字数を調整するための空白に対応する数値が入っている.

関数 put_int に入ると，まず大文字にするためのフラグが立っているか論理積演算を行う. そして，次に基底の変換を一桁ずつ行う. この際に，3 桁ごとにカンマを打つフラグ three_div があればカンマを挿入する. ここまでで，指定された最大文字表示数を使っているのでその分の変数 length を減らす. それが終わると，次は左詰め表示のフラグが立っていないときの処理を行う. この時に，0 で空白を埋めるフラグがあれば，pad を 0 に変更する. そして，length が 0 になるまで変数 pad の文字を格納する. 次に，10 進については符号があるため，符号があれば sign を付与する．そしてフラグの処理の最後に，alternative があれば，各基底に対する表示を行うために文字を格納する．そして，関数の最後で変数 i を一つづつ減らしていき格納した文字の新しいものから順に表示していく．表示文字数が足りない場合は pad の内容を表示する．

6.6 関数の拡張３：myprintf 関数

本命となる関数 myprintf について説明をする．この関数では，fmt の要素を一つづつ見ていき，%が現れたら修飾子に対応する myprintf の引数を表示させようとしている．fmt の要素がなくなるまでループする．

6.6.1 宣言した変数について

char *p 第 2 引数以降を格納するために利用する．fmt のアドレスに fmt のサイズ分追加する．char として扱うためにキャストしている．

int flags 各種のフラグと論理演算を行うために用いる．立っているフラグによって値が変わる．

int length 最大の表示文字幅を示す.

int precision 有効な文字数を表す．文字列の先頭からの数に対応する．

int tmp 文字列表示の際に最大の表示文字幅を計算するために用いる．

char sign 符号を表示するために格納する変数．初期値は空文字である．

char *s 引数として受け取った p の要素を格納し，扱いやすくする．

6.6.2 117 行目からの while 文

mystrchr() 関数によって，',-,+,#,0 が%以降に含まれているかどうか逐一検査する．以下に動作を記載する．

’が含まれていた場合　3桁ごとにカンマを入力するフラグ three_div を立てる．

-が含まれていた場合　左詰めのフラグ left_start を立てる．

+が含まれていた場合　正の値ならば数値の前にプラスの符号を付与するためのフラグ with_sign にフラグを立て，+を代入する．

#が含まれていた場合　型を明示して数値を出力するためのフラグ alternative を立てる．

0 が含まれていた場合　余白を 0 で埋めるためのフラグ fill_zero を立てる．

6.6.3　139 行目からの while 文

これは出力全体の桁数を計算する．例えば%12d と見つけたら，まずもともとの length(=0) に 10 をかけて，1 を足す．その後更新された length(=1) に 10 をかけて 2 を足す．すると length は 12 となり桁指定ができる．

6.6.4　146 行目からの if 文

ここでは，表示するデータの桁数を指定する．次の文字を見て length 同様の処理を行う．数字では無くなったら while ループを抜け出す．

6.6.5　150 行目からの switch 文

まず，case d，case i について，これまでの処理で負の場合でも+が表示される可能性がある．それを避けるために負の数かどうかの判定を行い，正しく-を表示するための前処理を行っている．その後関数 put_int() を１０進数で実行する．最後に引数を一つずらし break する．case x，case o についても同様で，基数を８と１６で分けているだけである．負の数については考慮しない．大文字の X のときは大文字にするフラグを立て，break させずにそのまま小文字の x と同様の処理を行う．最後に引数を一つずらし break する．

case s の場合は，まず取ってきた文字がヌルなら (null) と表示するために文字を代入する．その後変数 tmp に文字列の長さを格納．そして，表示文字数が限らている（0 以外．0 なら偽）かつ文字数よりも小さいならば，変数 presicion を tmp に格納．次に，length からその文字数だけ引きのこり何文字幅があるか計算する．左詰めのフラグ left_start がなければ残り幅が 0 になるまで空白を出力，そして表示文字数分だけ引数の文字を出力．左詰めの場合でまだ余白があれば空白を出力する．左詰めでなければスルーされる．最後に引数を一つずらし break する．

case c は純粋に引数の文字を出力し，引数を一つずらし break する．case %では%をエスケープさせるので出力するだけで break する．

そして switch を抜け，文字列を再度一つずつ見ていく．以上が関数の動作概要である．

6.7　テスト結果・評価結果

以下が main() での表示テストの結果である．

```
TEST
%d   :100
%5d  :  100
%-5d :100
%5.2d:  100
%#x  :0xf
%X   :F
%#o  :017
%s   :Say
%5s  :  Say
%5.2s:   Sa
%c   :a
%'d  :10,000
```

7 感想

本演習を通して，C 言語がどのようにコンパイルされてアセンブリへと変わるのかということを一つずつ確認しながら学ぶことができた．具体的には auto 変数と static 変数にはアセンブリ上で明らかな違いが出るということが興味深かった．

また，C 言語のサブセットについてはかなり苦労を強いられた．`%2d` や`%2.5d` などのフラグや最小フィールド幅が指定される形式の対応は自分ひとりではうまく実装できなかったため友人や TA の方に多くの助力をいただいた．今回は実装していない浮動小数点の対応についてどのような処理が必要なのかということも気になった．