C 言語の学習
制御文・ポインター

山本昌志*

2004 年 5 月 19 日

1 本日の学習内容

C 言語の基本的な部分を学習する。内容は、教科書の 9〜10 章である。ただし、ポインター (10 章) は難しく、ここでの数値計算の講義ではほとんど陽には使われないので、余力のある者のみ実施せよ。

9 章 制御文

- 分岐の命令の使い方が分かる。
- 繰り返し文の使い方が分かる。

10 章 ポインター

- アドレスとデータ、ポインターの関係が分かる。

2 制御文 (9 章)

プログラミング言語の制御構造は、次の 3 通りから成り立っている。

順次 これはプログラムの文を上から下へと実行する構造で、特にこれを表す命令はない。

選択 値により、実行する文が異なる構造である。C 言語には `if` 文と `switch` 文がある。`switch` 文はプログラムの構造が分かり難くるので、`if` 文を使うことが望ましい。

繰り返し 同じ文を繰り返す構造である。C 言語には、`for` 文と `while` 文 `do-while` 文がある。

2.1 `if` 文 (p.138)

`if` 文には典型的な 3 つのパターンがある。`if` 文中の制御式は、0(ゼロ) の場合のみ偽であり、その他は全て真である。この真偽により、実行される文が 3 パターンあるのである。

*独立行政法人 秋田工業高等専門学校 電気工学科

2.1.1　if 文のみの場合

制御式 (条件) が真の場合には特定の文を実行するが、偽の場合には実行する文がない。実行する文が一つの場合は。if の制御式に引き続き書く。一方、複数の実行文がある場合は、中括弧 { 文 } でくくりブロック化する。そうするとブロック化された部分が上から下へと実行される。

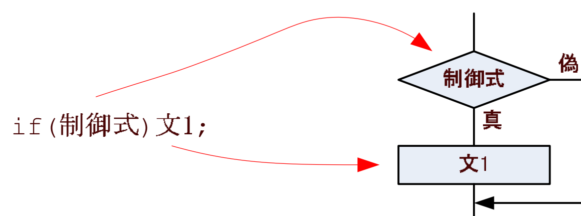

図 1: ブロックが無い場合

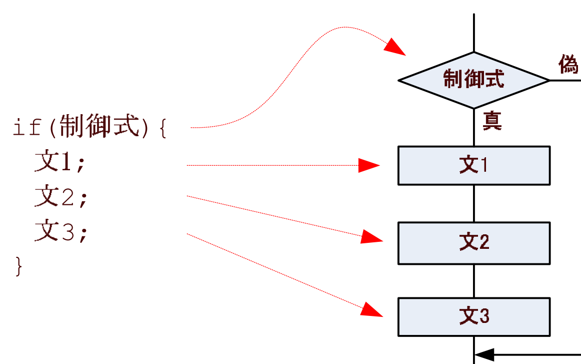

図 2: if 文のみの場合

2.1.2　if〜else の場合

制御式 (条件) が真偽に応じて、実行する文が異なる。

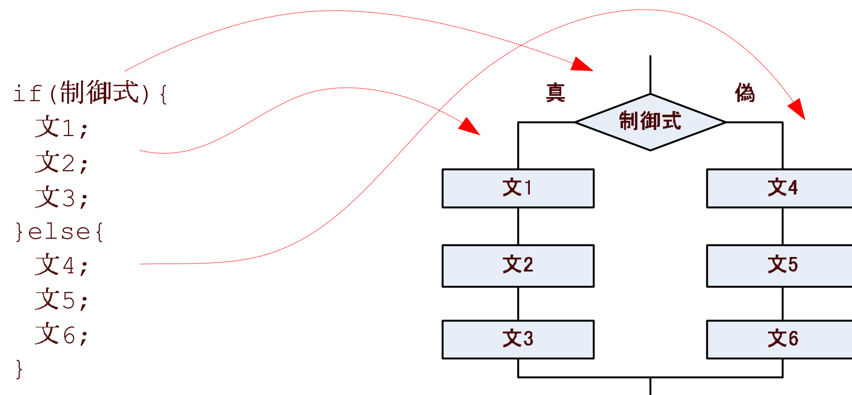

図 3: if〜else の場合

2.1.3 if〜else if〜else の場合

制御式 (条件) が多段階になっている。最初に真にマッチしたブロックを実行する。制御式が真にならない場合は、else のブロックを実行する。

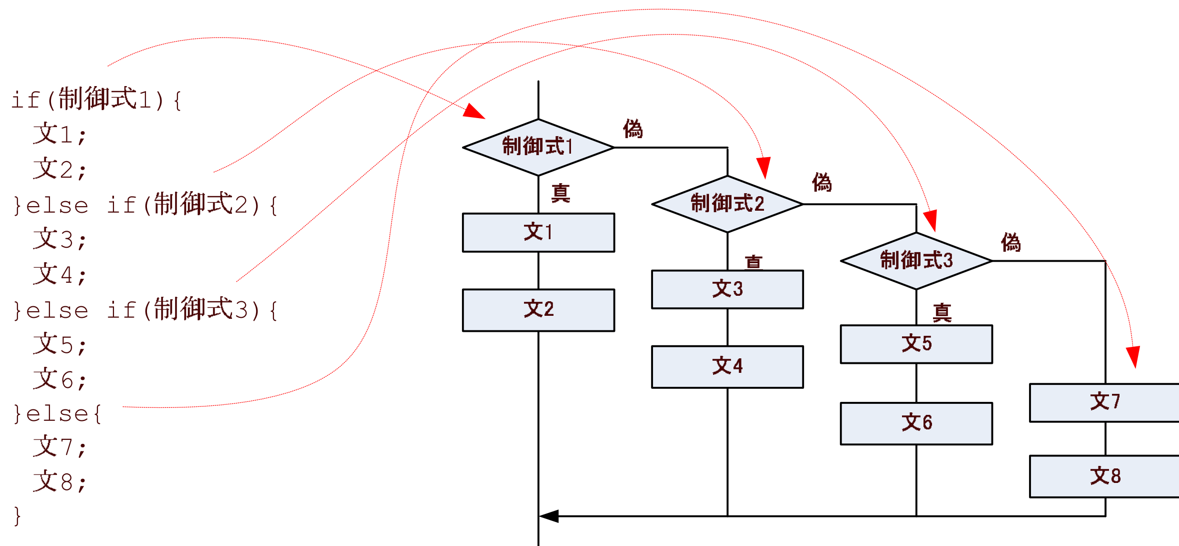

図 4: if〜else if〜else の場合

2.2 switch 文 (p.147)

式の値により、実行される文が決まる。

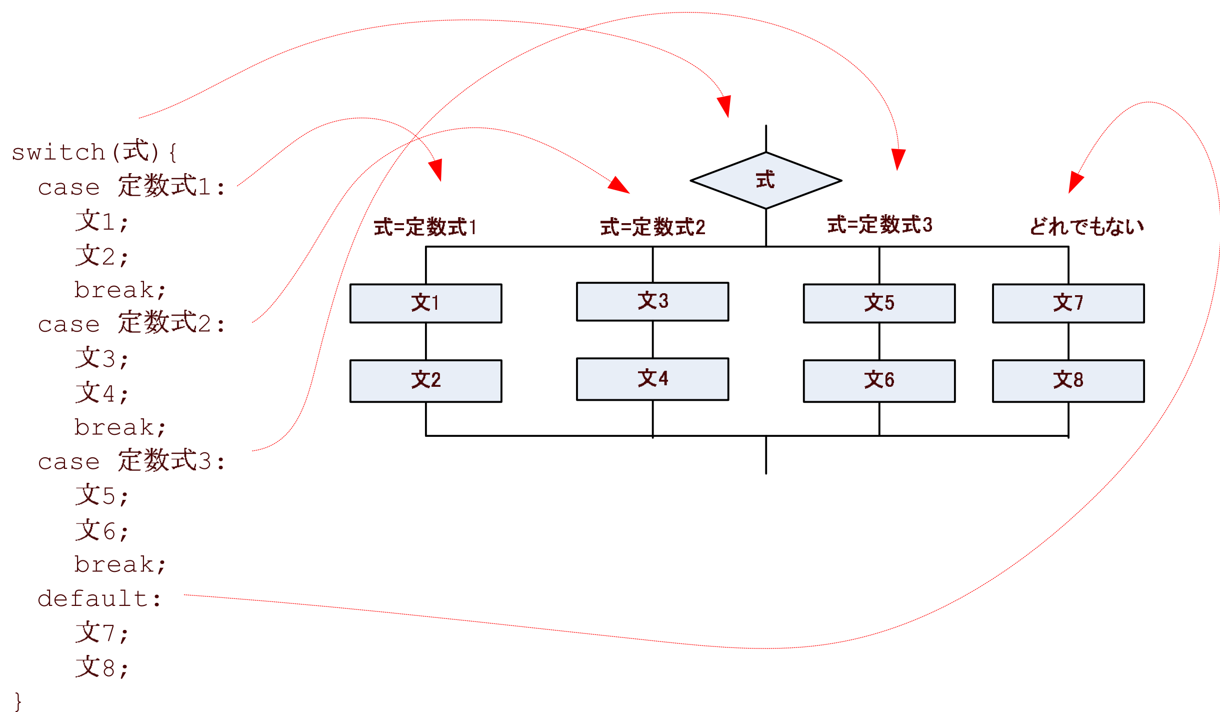

図 5: `switch` 文による分岐構造

2.3 指定回数繰り返し

条件に従い同じ文を繰り返し実行する構造をループ (繰り返し) と呼ぶ。繰り返し回数が分かっている場合は、`for` 文を使う。

2.3.1 `for` 文 (p.142)

ループの回数が予め分かっている場合に使う。実行順序は次の通りである。

1. 初期値設定式が評価 (実行) される。
2. 継続条件式を評価し、それが真ならばブロック化された文が実行される。偽ならば、`for` 文から抜ける。
3. 再設定式が実行される。
4. 継続条件式が偽になるまで、2→3→2←3→2→3→2→ · · · が繰り返される。

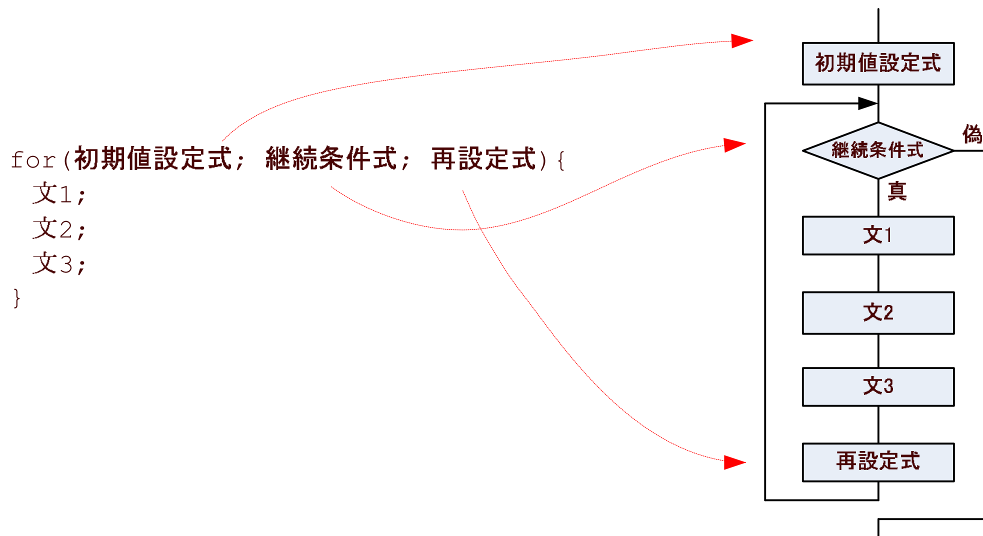

図 6: for 文による繰り返し (ループ) 構造

for 文を用いた 1〜100 までの和を用いたプログラムをリスト 1 に示す。

2 行 変数宣言。i は加算する数、sum は合計を格納する変数。

6 行 合計値の初期化。最初、合計はゼロである。

8-10 行 中括弧 {} のブロック内を i=1〜100 まで変えて 100 回計算する。

9 行 sum=sum+i と同じ

リスト 1: 1〜100 の和の計算

```
1  #include <stdio.h>
2
3  int main(){
4    int sum, i;
5
6    sum=0;      /* 初期化 */
7
8    for(i=1;i<=100;i++){
9      sum+=i;
10   }
11
12   printf("1〜100までの和は、%dです\n",sum);
13
14   return 0;
15 }
```

以下の練習問題を実施せよ。

[練習 1] 次の動作をするプログラムを作成せよ。最後の c の値はどのような値になるか?

1. 実数 a, b の初期値をそれぞれ、1.0 と 2.0 とする。
2. 実数 c を $c = (a+b)/2.0$ とする。
3. もし、c^2 が 2 より小さければ、c の値を a に代入する。反対に 2 よりも大きければ、c の値を b に代入する。
4. 操作 2〜3 を 100 回繰り返す。

[**練習 2**] 以下の級数展開式を用いてネピア数 (e) を計算せよ。少し難しいので、余裕のある者のみ実施せよ。

$$\begin{aligned} \mathrm{e} &= 1 + 1 + \frac{1}{2} + \frac{1}{3 \times 2 \times 1} + \frac{1}{4 \times 3 \times 2 \times 1} + \cdots \\ &= \sum_{i=0}^{\infty} \frac{1}{i!} \end{aligned} \tag{1}$$

2.4 不定回数繰り返し

繰り返し回数が予め分かっていない場合は、do while 文か while 文を用いる。それぞれの違いは、以下の通りである。

- `do-while` 文は、ループ出口で継続条件の判断を行う。
- `while` 文は、ループ入り口で継続条件の判断を行う。

2.4.1 `while` 文 (p.143)

入口で継続条件を判断するため、最初から偽の場合、ループは一度も回らない。ループを抜け出すためには、ループの文中に継続条件に使う値を変更するか、`break` 文を使う。

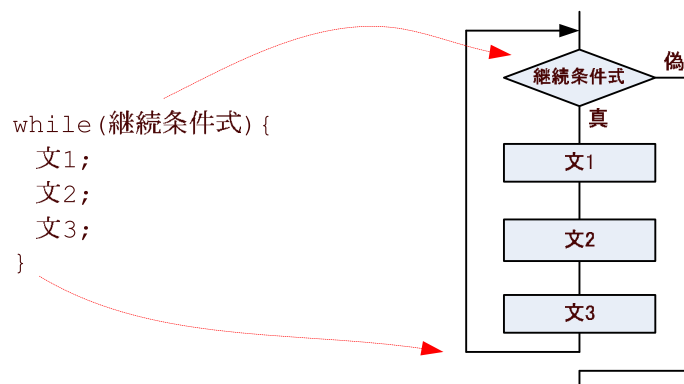

図 7: `while` 文による繰り返し (ループ) 構造

[**練習 1**] `while` 文を使って、1〜100 まで加算するプログラムを作成せよ。

2.4.2 do-while 文 (p.145)

出口で継続条件を判断するため、必ず 1 回はループが回る。ループを抜け出すためには、ループの文中に継続条件に使う値を変更するか、break 文を使う。

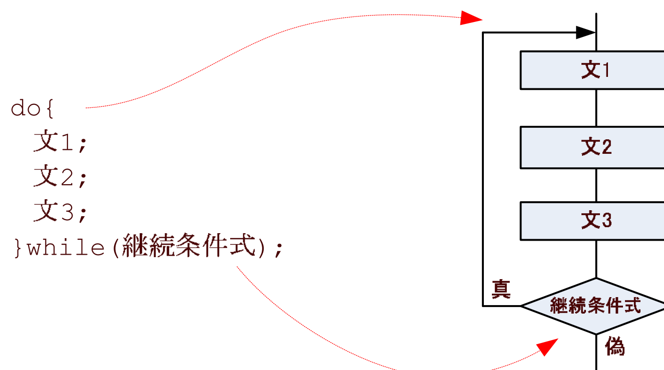

図 8: do-while 文による繰り返し (ループ) 構造

[練習 1] do-while 文を使って、1〜100 まで加算するプログラムを作成せよ。

2.5 break 文 (p.149)

ループから強制的に脱出するために、break 文がある。単独で使われることは希で、以下のように if 文と共に使われる。break 文を用いると、for 文や do while 文、while 文のループから抜け出せる。

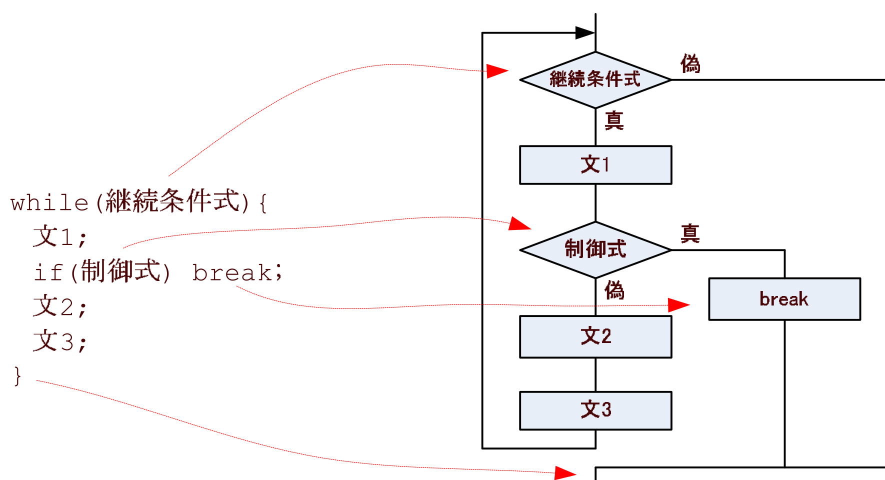

図 9: break 文による分岐構造

2.6 continue 文 (p.150)

ループ中の繰り返しを 1 回強制的にスキップするために使われる。これも単独で使われることは希で、以下のように if 文と共に使う。

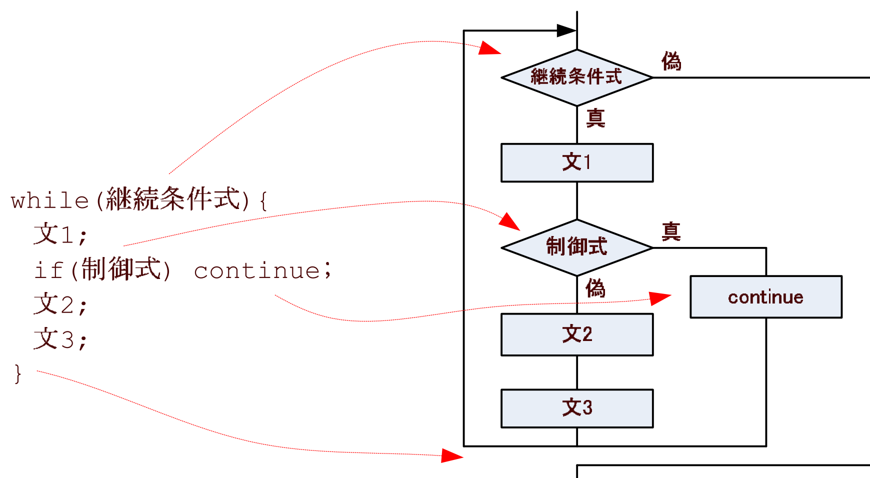

図 10: continue 文による繰り返し (ループ) 構造

2.7 goto 文とラベル (p.152)

強制的にプログラムの制御を移す。goto 文が示すラベルに実行が移る。if 文と共に用いられることも多いが、単独で用いられることもある。

ただし、goto 文はプログラムの流れがわかりにくくなりますので、使わないほうが良いとされている。行儀の良いプログラムを書くためには goto 文は使わないことになっている。ただし、初心者が書くような短いプログラムであれば使っても良いでろう。簡単だし、行儀が悪くてもプログラムを書くことになれる方が重要である。上達したら、goto 文を書かないようにすればよい。

ラベル名の後ろには、セミコロンではなく文の前に書いてコロンをつける。

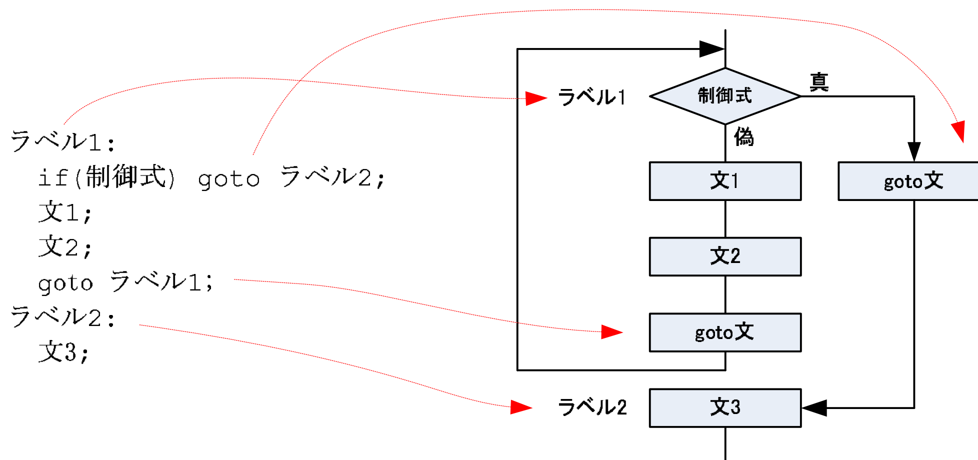

図 11: goto 文とラベルによる制御

3　ポインター (10 章)

今までのサンプルプログラムは、FORTRAN と置き換えが可能です。対応する FORTRAN の命令があり、理解は容易でしょう。しかし、ここで説明するポインター (pointer) は FORTRAN にない概念です。これこそ、C と FORTRAN の大きな違いで、C 言語のもっとも大きな特徴です。そのため、C 言語の勉強で、ここで挫折する人が多く居ます。みなさん、がんばって勉強してください。覚えることなど無く、その内容を理解すれば、ポインターなんか難しくありません。

リスト 2 の普通の変数 v1, v2, v3 は、整数を格納する変数です。コンパイラーが確保した領域に整数を格納します。コンピューターのメモリには、すべて、アドレス (番地) がついています。各アドレスには、1 バイト (=8 ビット) のデータが対応しています。このアドレスに従い、メモリー内のデータや命令を参照しています。プルグラムの実行時は、v1 等の変数が呼ばれると、それに対応したアドレスが呼ばれ、データの参照をします。

これら普通の変数のアドレスを参照したいときは、`&v1` のように、変数の前に`&`をつけます。これで、その変数の先頭アドレスを直接見ることが出来ます。例えば、`int` 型の変数であれば、4 バイト 1 のメモリー領域が必要なので、連続した 4 つのアドレスのメモリー領域を使用します。その先頭アドレスを`&v1` のようにして参照します。

定義	`int v1;`
データの参照	`v1`
アドレスの参照	`&v1`

一方、リスト 2 のポインター `p1`,`p2`,`p3` はアドレスを表す変数です。通常の変数とは異なり、コンパイラーにより確保された領域にアドレスを格納します。アドレス、即ち、値の場所を示すものだから、ポインターといいます。ポインターもアドレスというデータを格納するため、連続したメモリー領域が必要です。秋田高専のパソコンは 32 ビットアドレッシングのため、4 バイトのメモリー領域が必要です。

プログラム内で*p1 のように呼ばれると、`p1` が示しているアドレスの値を参照します。`p1` は先頭アドレ

スを示しているので、型に応じたバイト数を呼び出すことになります。そのため、ポインターにも、型が必要になります。

プログラム内で p1 と呼ぶと、そのポインターが示しているアドレスを参照します。また、&p1 と呼び出すと、ポインターの先頭アドレスを参照します。

定義	int *p1;
ポインターが示しているデータの参照	*p1
ポインターが示してるアドレスの参照	p1
ポインターのアドレスの参照	&p1

ポインターに関係する演算子を下表に示します。

演算子	機能
*	ポインターが指しているアドレスの内容を取り出す
&	変数が格納されているアドレスを取り出す

例えば、ポインターの勉強のために、リスト 2 を実行しましょう。

リスト 2: ポインターの学習プログラム

```
1  #include <stdio.h>
2  int main (){
3    int v1, v2, v3, *p1, *p2, *p3;
4    unsigned char *ip;
5    int i;
6
7    v1 = -1;
8    v2 = 2;
9    v3 = 3;
10
11   p1 = &v1;
12   p2 = &v2;
13   p3 = &v3;
14
15   printf ("\n");
16
17   printf ("\n ----- address hexadecimal -------\n");
18
19   printf ("&v1=%x\n", &v1);
20   printf ("&v2=%x\n", &v2);
21   printf ("&v3=%x\n", &v3);
22   printf ("&p1=%x\n", &p1);
23   printf ("&p2=%x\n", &p2);
24   printf ("&p3=%x\n", &p3);
25
26   printf ("\n ----- value decimal -------\n");
27
28   printf (" v1=%d\n", v1);
29   printf (" v2=%d\n", v2);
30   printf (" v3=%d\n", v3);
31   printf ("*p1=%d\n", *p1);
32   printf ("*p2=%d\n", *p2);
33   printf ("*p3=%d\n", *p3);
34
35   printf ("\n ----- value hexadecimal -------\n");
```

```
36
37    printf (" v1=%x\n", v1);
38    printf (" v2=%x\n", v2);
39    printf (" v3=%x\n", v3);
40    printf ("*p1=%x\n", *p1);
41    printf ("*p2=%x\n", *p2);
42    printf ("*p3=%x\n", *p3);
43
44
45    printf ("\n----- pointer hexadecimal -------\n");
46
47    printf ("p1=%x\n", p1);
48    printf ("p2=%x\n", p2);
49    printf ("p3=%x\n", p3);
50
51    printf ("\n---------------------------------\n");
52    printf ("\n    address     data\n\n");
53
54    for (i = 0; i<= (int)&v1-(int)&p3+3; i++){
55        ip = (char *) &p3 + i;
56        printf ("    %08x    %02x\n", ip, *ip);
57      }
58
59    printf ("\n");
60
61    return 0;
62
63  }
```

実行結果

```
 ----- address hexadecimal -------
&v1=bffff6b4
&v2=bffff6b0
&v3=bffff6ac
&p1=bffff6a8
&p2=bffff6a4
&p3=bffff6a0

 ----- value decimal -------
 v1=-1
 v2=2
 v3=3
*p1=-1
*p2=2
*p3=3

 ----- value hexadecimal -------
 v1=ffffffff
 v2=2
 v3=3
*p1=ffffffff
*p2=2
*p3=3
```

```
 ----- pointer hexadecimal -------
p1=bffff6b4
p2=bffff6b0
p3=bffff6ac

 --------------------------------

   address    data

   bffff6a0   ac
   bffff6a1   f6
   bffff6a2   ff
   bffff6a3   bf
   bffff6a4   b0
   bffff6a5   f6
   bffff6a6   ff
   bffff6a7   bf
   bffff6a8   b4
   bffff6a9   f6
   bffff6aa   ff
   bffff6ab   bf
   bffff6ac   03
   bffff6ad   00
   bffff6ae   00
   bffff6af   00
   bffff6b0   02
   bffff6b1   00
   bffff6b2   00
   bffff6b3   00
   bffff6b4   ff
   bffff6b5   ff
   bffff6b6   ff
   bffff6b7   ff
```

[**練習 1**] この結果を考察せよ。