

# C 言語の学習 ポインタ

山本昌志\*

2006 年 5 月 9 日

## 概要

ポインタについて、原理的なことを説明している。ポインタの使い方のテクニックは、まったく説明していない。使い方については、別の書籍で勉強する必要がある。

## 1 本日の学習内容

教科書 [1] の 10 章のポインタについて説明する。ただし、ポインタの内容は難しく、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。

### 10 章 ポインタ

- アドレスとデータ、ポインタの関係が分かる。

## 2 メモリー

今まで学習した C 言語の内容は、FORTRAN<sup>1</sup>と置き換えが可能である。対応する FORTRAN の命令がある。C 言語のプログラムは命令の書き換えのみで、FORTRAN のプログラムになる。しかし、ここで説明するポインタ (pointer) は FORTRAN にない機能である。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となっている。少しばかり難しく、ここで挫折する人も多くいる。しかし、その内容を理解すれば、ポインタなんか難しくないはずである。

メモリーが分からないと、ポインタは理解できない。そこで、ポインタの説明の前に、コンピューターのメモリーについて、説明する。ただし、諸君は、アセンブラ言語を既に学習しているので、この辺りのことはある程度理解していると思う。

---

\*独立行政法人 秋田工業高等専門学校 電気工学科

<sup>1</sup>諸君が 1 年生のときに学習した FORTRAN77 を指す。

## 2.1 メモリーと CPU の関係

ここで、コンピューターを構成する最小の部品を考える。そうすると CPU とメインメモリーがあれば良いことが分かる。これでも、メインメモリーにプログラムを格納して、CPU とデータの受け渡しを行い、データを処理することができる。図 1 のようなものである。事実、CPU と入出力装置の間に流れるデータは、メインメモリーを介している。従って、コンピューターの原理的なモデルを図 1 のように考えても良いだろう。

プログラマーはメモリーの内容について、ある程度自由に変更ができる。そのようなことから、メモリーを意識してプログラムを作成することが重要である。アセンブラー言語を使うとなると CPU についても意識が必要であるが、C 言語ではそこまで要求しない。

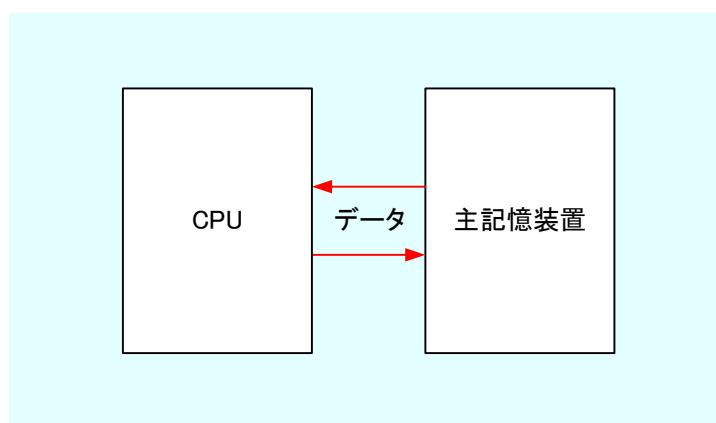


図 1: もっとも原始的なコンピューター

### 2.1.1 メインメモリーのモデル

メインメモリーのハードウェアの構造については、ここではどうでも良い。それよりか、メインメモリーのモデルを理解することが重要である。

メインメモリーの役目は、命令とデータからなるプログラムを記憶することである。そのプログラムは全て、0 と 1 の数字で表せ、2 進数で表現可能である。どのようなモデルでこの 2 進数が格納されているか学習する。

プログラムという情報は記憶するだけでは全く役に立たない。記憶した内容を取り出せて初めて、活用ができる。そこで、メモリーは記憶するための住所が決まっている。この住所のことをアドレス (address) と言い、0 から整数の番地がふってある。諸君が使っているパソコンのアドレスは 32 ビットで表現されている<sup>2</sup>。そして、一つの番地には、8 個の 0 と 1 が記憶できる。この様子を図 2 に示す。

図を見て分かる通り、2 進数の表現は桁数が多くて人間にとって大変である<sup>3</sup>。そこで、通常は、2 進数の 4 桁をまとめて、16 進数で表す。そうすると、アドレスは 16 進数 8 桁、記憶内容は 16 進数 2 桁で表す

<sup>2</sup>CPU によりアドレスの表現は異なり、32 ビットではないものもある。

<sup>3</sup>コンピューターにとっては全然大変でない

ことができ、分かりやすくなる。その様子を図3に示す。

ついでに述べておくが、1個の0あるいは1の情報量を1ビットと言う。8ビットで1バイトと言う。従って、メインメモリーの一つの番地(アドレス)には、1バイト(8ビット)の情報が記憶できる。

メモリーについて覚えておくことは、以下の通りである。

- アドレスは32ビットで表現している。これは16進数では8桁である。
- 一つのアドレスに8ビット(1バイト)記憶できる。
- 8ビットを1バイトと言う。

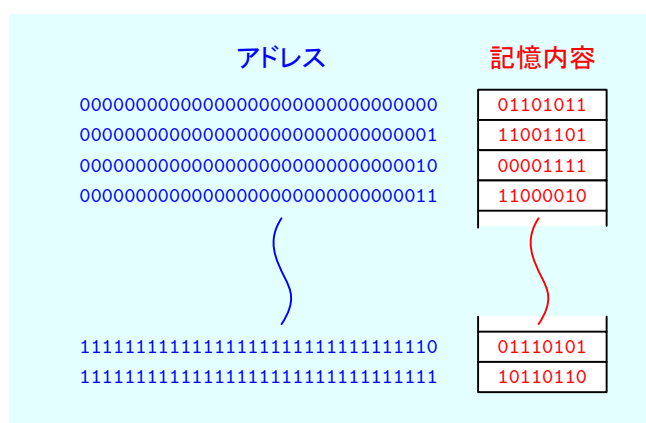


図2: メモリーのモデル(2進数)

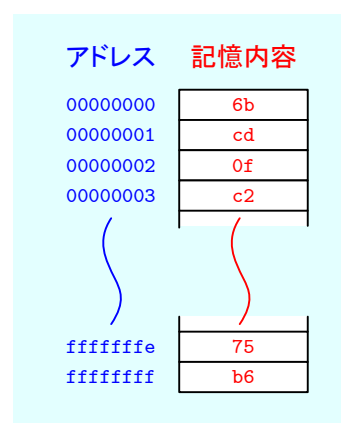


図3: メモリーのモデル(16進数)

## 2.2 データの型とバイト数

諸君が使う変数の型は、文字型と整数型、倍精度実数型がほとんどである。秋田高専内で使用されているパソコンのそれぞれのサイズは表1の通りである。文字型であれば、一つのアドレス内に格納することができるが、整数型では4つ、倍精度実数型では8個のアドレスが必要である。一つのアドレスに一つのデータが記憶されている訳ではない。sizeof(型) 演算子(教科書 p.126)を使うと、型が必要とするバイト数がわかる。

表1: 変数の型とバイト数

型名	データ型	バイト数	ビット数
文字型	char	1	8
整数型	int	4	32
倍精度実数	double	8	64

それでは、実際にメモリーにデータが格納する様子を見よう。次のようにプログラムに書いたとする。

```
double x=-7.696151733398438e-4;
int i=55;
char a='a';
```

それぞれのデータは、

- x の値のビットパターンは、私の講義ノートを見よ。
- $(55)_{10} = (37)_{16}$  から i のビットパターンは分かる。
- 文字の 'a' はアスキーコードの  $(61)_{16}$  である。

となっている。実際にこれを確認するプログラムを、付録のリスト 4 に示している。このプログラムを私のパソコンで実行させると、図 4 のようなメモリー配置になっていることが分かった。表 1 の通り、整数型と実数型は複数のアドレスにわたってデータが格納されていることが分かる。

鋭い学生は、データの並びが逆であることが分かるであろう。例えば、整数の i であるが、 $(55)_{10} = (00000037)_{16}$  なので、 $00 \rightarrow 00 \rightarrow 00 \rightarrow 37$  と並ぶと考えられるが、実際は図 4 の通り逆である。これは CPU がそのように作られているからである。このように逆に配置させる方法をリトルエンディアンと言う。Intel 社の CPU はリトルエンディアンである。一方、そのままのメモリーに配置する方法はビッグエンディアンと呼ばれる。このようにメモリーにデータを並べる方法は 2 通りあって、それをバイトオーダーと言う。ここで、理解しておくべきことは、以下の通りである。

- 必要なバイト数のメモリー領域を使いデータは格納される。

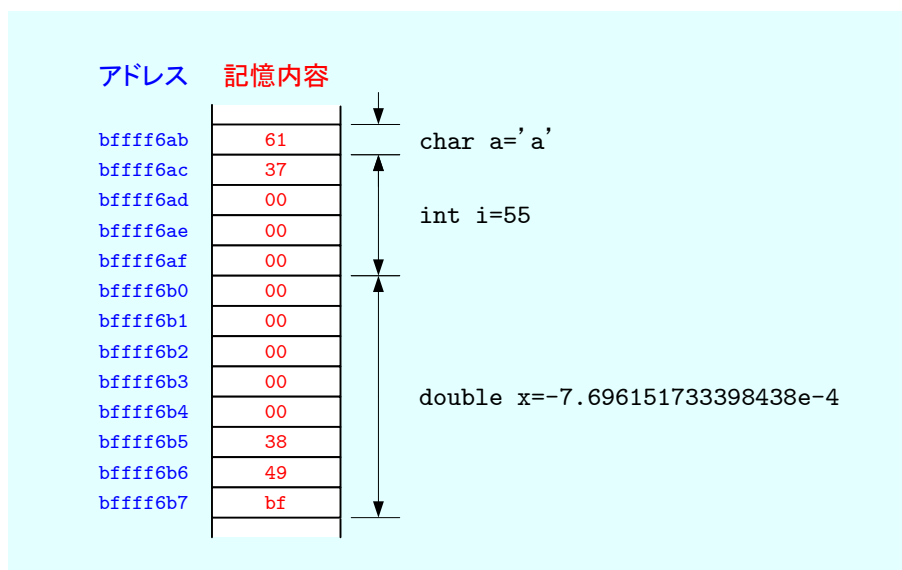


図 4: メモリー中に格納されたデータの例

## 2.3 プログラムが格納される様子

これまでは、メモリーのデータの格納方法を学習した。以前、プログラム(命令とデータ)は全てメモリーに格納されると述べた。ここでは、もう少し進んで、プログラムがメモリーの中にどのように格納されているか調べてみよう。この辺のことで、マシン語が分かると、ハッカー(クラッカーと言った方が適切かも)になれるかも…。

それでは、リスト 1 に示す簡単なプログラムで、データとメモリーの格納アドレスを調べてみよう。このプログラムの内容は、以下の通りである。まだ、詳細は分からなくても良いが、大体の流れをつかんで欲しい。

- 1 行 今のところおまじない
- 2 行 関数 func のプロトタイプ宣言
- 4-6 行 コメント文。プログラムの動作には無関係。プログラマーのために記述。
- 7 行 main 関数の始まり。int で整数を返すことを示し、void で引数が無いことを示している。
- 9 行 整数変数 i の宣言
- 11 行 結果を分かりやすくするために --- address ----- を表示。最後に \n で改行。
- 12 行 main 関数がかかれている先頭アドレスを表示。関数名はアドレスを表しており、変換指定子 %p でディスプレイに表示。 \t は、タブを表し、適当な空白が入る。
- 13 行 関数 func が書かれている先頭アドレスを表示。
- 14 行 変数 i の先頭アドレスを表示。変数名に & を付けると、その先頭アドレスを示すことになる。& はアドレス演算子である。
- 16 行 関数 func に処理が移り、その戻り値を変数 i に代入。
- 18 行 main 関数の終了を表し、呼び出し元(OS)に整数の 0 を返している。
- 19 行 main 関数のブロックの終わり。
- 21-23 行 コメント文
- 24 行 関数 func の始まり。int で整数を返すことを示している。仮引数は、整数型の i と j である。
- 26 行 変数 i の先頭アドレスを表示。
- 27 行 変数 j の先頭アドレスを表示。
- 29 行 関数 func の終了を表し、呼び出し元(ここでは main 関数)に整数の i と j の積を返している。
- 31 行 関数 func のブロックの終わり。

## リスト 1: メモリーのアドレス調査

```
1 #include <stdio.h>
2 int func(int i, int j);
3
4 /*=====*/
5 /*   メイン 関数   */
6 /*=====*/
7 int main(void){
8
9     int i;
10
11     printf("--- address -----\n");
12     printf("\tmain\t%p\n", main);
13     printf("\tfunc\t%p\n", func);
14     printf("\tmain-i\t%p\n",&i);
15
16     i=func(5,3);
17
18     return 0;
19 }
20
21 /*=====*/
22 /*   func 関数   */
23 /*=====*/
24 int func(int i, int j){
25
26     printf("\tfunc-i\t%p\n",&i);
27     printf("\tfunc-j\t%p\n",&j);
28
29     return i*j;
30
31 }
```

### 実行結果

```
--- address -----
      main    0x8048368
      func    0x80483eb
      main-i  0xbffff6b4
      func-i  0xbffff690
      func-j  0xbffff694
```

実行結果から、命令とデータは図 5 のようになっていることが分かるであろう。命令である関数は、大体近くのメモリ上に配置されている。しかし、データの内容を格納する変数は、ずっと離れてところにメモリが割り当てられている。

関数の中で宣言される変数は、ローカル変数と言い、その宣言した関数でのみアクセスが可能である。従って、同じ名前であるが、違う関数で宣言されたローカル変数は全く別物である。図 5 で分かるように、関数 main と関数 func で同じ名前のローカル変数 i を宣言してるがメモリ上の配置は全く異なる。このことから、名前は同じであるが、全く違うものであることが理解できる。

ここで、理解しておくべきことは、以下の通りである。

- プログラムは命令とデータから構成され、いずれもメモリの中に格納される。
- プログラムの関数 (これが命令) が格納されるアドレスは、関数名で参照できる。
- データが格納されるアドレスは、変数名の前に & を付けることで参照できる。& はアドレス演算子である。
- アドレスの表示には変換指定子 %p を使う。
- ローカル変数は名前が同じでも、メモリの配置場所は異なる。正確言うと、その関数が呼び出されたときのみ、ローカル変数はメモリーに割り当てられる。

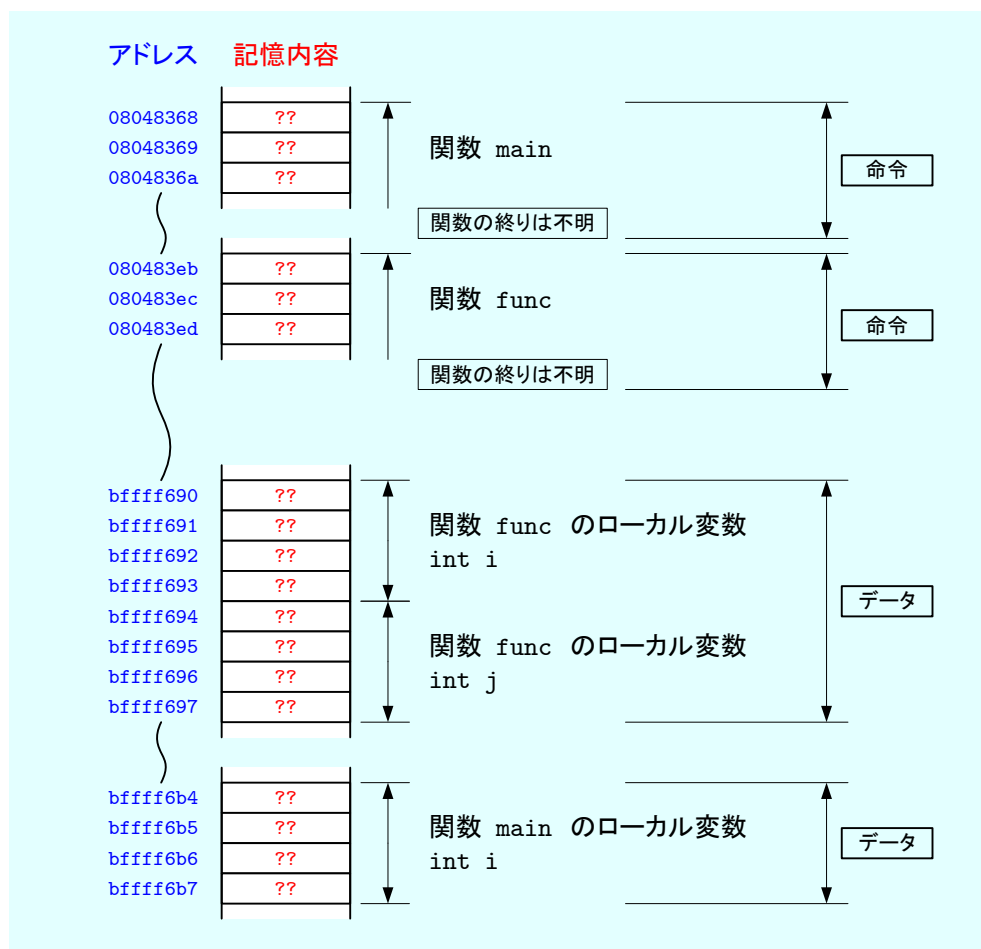


図 5: プログラムのメモリーへの格納。記憶の内容は不明なので、??としている。

## 3 ポインター (10章)

### 3.1 ポインターとはなにか

これまでの話で、メモリーというものが大体分かったと思う。そして、その内容とともに、アドレスが重要であることも分かったであろう。あるいは、アドレスを上手に操作すれば、いろいろなこと(悪いことも)ができそうだと分かったであろう。

アドレスを操作するとなると、アドレスを入れる変数が欲しくなる。2.1.1節で述べたように、アドレスは32ビットである。また、int型のデータも32ビットである。従って、int型の変数にアドレスを入れることができそうである。具体的には、hogeと言う変数のアドレスをint型の変数iに、次のような文で、

```
i=&hoge;
```

と代入する。しかし、これはコンパイラーにより警告が出され、推奨される方法でない<sup>4</sup>。たまたま、私が使っているコンパイラーでは警告で済んでいるが、エラーを出すものもあるであろう。そもそも、アドレスのビット数とint型のビット数が同じであるのは偶然にすぎない。

幸いなことに、C言語にはアドレスを格納する仕組みが用意されている。ポインターという変数を使い、アドレスが格納できるのである。そのアドレスを格納するポインター型変数は、

```
int *pi;  
double *px;
```

と宣言する。アスタリスク(\*)をつければ、ポインターの宣言になる。

整数型変数iと実数型変数xのアドレスは、&iと&xのようにすると取り出すことができる。アドレス演算子(&)を使うのである。取り出したアドレスは、ポインターに

```
pi=&i;  
px=&x;
```

のようにして代入できる。アドレス演算子(&)により変数の先頭アドレスを取り出して、代入演算子(=)を用いて、ポインター型変数に代入している。

ポインター機能は、アドレスの格納のみに止まらず、そのアドレスが示しているデータの内容も表すことができる。今までの例の通り、ポインターには変数の先頭アドレスが格納されている。そして、ポインターの宣言の型から、そのポインターが指しているデータの内容までたぐり寄せることができる。ポインターpiとpxが示しているデータの値を、整数型変数jと実数型変数yに代入する場合

```
j=*pi;  
y=*px;
```

とかく、ここで、アスタリスク(\*)は間接参照演算子で、ポインターが示しているアドレスのデータを取り出せるのである。このようにアドレスのみならず、そのアドレスのデータの型までポインターは持ってい

---

<sup>4</sup>キャスト(強制型変換)を使って警告を消すこともできるが邪道である。



るから、これが可能なのである。このことから、アドレスとは言わずにポインタ (pointer 指し示すもの) と言うのであろう。

- ポインタとは、アドレスを格納する変数のことである<sup>a</sup>。
- ポインタの宣言には、型名とアスタリスク (\*) を付ける。
- 変数のアドレスを取り出すには、変数名の前にアンパサンド (&) をつける。&はアドレス演算子である。
- ポインタが示しているデータの値を取り出すためには、ポインタ変数の前にアスタリスク (\*) をつける。\*は間接参照演算子である。

<sup>a</sup>正確に言うところとちょっと違うが、ほとんど正しい。また、アドレスはメモリの物理的なアドレスではなく、仮想アドレスである。この辺のところは余りにしないことにする。

## 3.2 プログラム例

実際のプログラムで見てみよう。リスト 2 のプログラムは、の動作は以下の通りである。これにより、ポインタの意味とそれに関わる演算子の動作の基礎的なことを理解する。

- ポインタ変数 `p` と整数変数 `i` を宣言する。
- 整数変数 `i` の先頭アドレスをポインタ `p` に代入する。
- 各種演算子を使って、`p` や `i` アドレス等を調べる。

このプログラムの各行の内容は、以下の通りである。1 行毎にきっちり理解することが重要である。

- 4 行 整数型のポインタ `p` を宣言している。`p` に整数型のデータの先頭アドレスを格納する。
- 5 行 整数型の変数 `i` を宣言し、 $(11223344)_{16}$  を代入している。
- 7 行 変数 `i` の先頭アドレスをアドレス演算子 `&` により取り出し、ポインタ `p` に代入している。
- 9 行 整数変数 `i` の先頭アドレスを変換指定子 `%p` により表示している。
- 10 行 ポインタ `p` の先頭アドレスを変換指定子 `%p` により表示している。
- 12 行 整数変数 `i` の値を 16 進数表示の変換指定子 `%0x` により表示している。
- 13 行 ポインタ `p` の値を 16 進数表示の変換指定子 `%0x` により表示している。ただし、ポインタはアドレスなので、強制型変換 (キャスト) により、符号なし整数にしている<sup>5</sup>。
- 15 行 ポインタが指し示すアドレスに格納されているデータを表示している。

<sup>5</sup>強制型変換しなくても実行は可能であるが、コンパイル時に型の不一致の警告がでる。

リスト 2: アドレスをポインターに代入して、変数のアドレスと内容を検査

```

1 #include <stdio.h>
2
3 int main(void){
4     int *p;
5     int i=0x11223344;
6
7     p=&i;
8
9     printf(" address i %p\n", &i);
10    printf(" address p %p\n", &p);
11
12    printf(" value i %0x\n", i);
13    printf(" value p %0x\n", (unsigned int)p);
14
15    printf(" value *p %0x\n", *p);
16
17    return 0;
18 }

```

実行結果

```

address i 0xbffff6b0
address p 0xbffff6b4
value i 11223344
value p bffff6b0
value *p 11223344

```

この実行結果から、メモリーは図 6 のようになっていることが分かる。ポインター p には、整数変数 i の先頭アドレスが格納されている。さらに、ポインター p に間接参照演算子\*を作用 (\*p) させることにより、ポインターが指し示すアドレスの内容を取り出している。また、どんな変数でも、アドレス演算子&で、メモリーのアドレスが取り出せている。これらのことをしっかり理解すると、ポインターは難しくない。

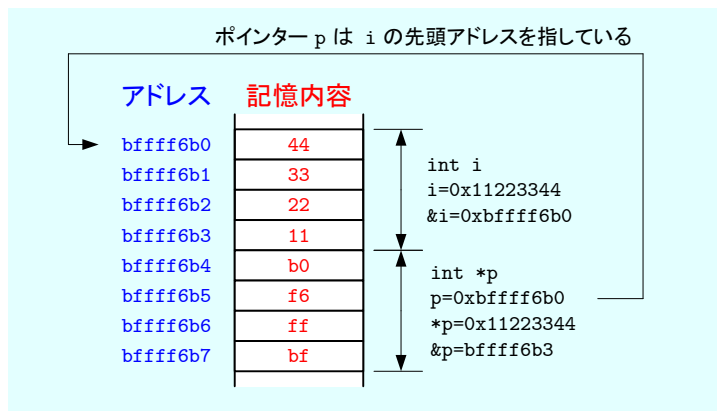


図 6: リスト 2 のプログラム実行後のメモリーの内容

### 3.3 ポインターに関する演算子

ポインターに関する演算子を表 2 にまとめておく。ただし、各変数は

```
char   c, *cp;
int    i, *ip;
double x, *xp;
```

と宣言したとする。

表 2: 演算子

演算子	通常の変数 (c, i, x)	ポインター (cp, ip, xp)	例
&	格納されている値	格納されているアドレス	c, i, x, cp, ip, xp
	変数のアドレス	ポインターのアドレス	&c, &i, &x, &cp, &ip, &xp
*	コンパイルエラーのため不可	ポインターが示す値	*cp, *ip, *xp

まとめると、重要なことは以下の通りである。

- 通常の変数には値を、ポインターにはアドレスを格納する。
- アドレス演算子&は、変数 (ポインター型変数も含む) のアドレス返す。
- 間接参照演算子\*は、ポインターが示すメモリーに格納されている値を返す。

## 4 付録

### 4.1 型のサイズを調べる

リスト 3: データ型によるバイト数調査プログラム

```
1 #include <stdio.h>
2
3 int main(void){
4
5     printf("----- size -----\n");
6     printf("\tchar\t%d\n", sizeof(char));
7     printf("\tint\t%d\n", sizeof(int));
8     printf("\tdouble\t%d\n", sizeof(double));
9
10    return 0;
11 }
```

#### 実行結果

```
----- size -----
      char      1
      int       4
      double    8
```

### 4.2 変数のアドレスと内容を調べる

リスト 4: データのアドレスと内容の調査プログラム

```
1 #include <stdio.h>
2
3 int main(void){
4
5     double x=-7.696151733398438e-4;
6     int i=55;
7     char a='a';
8     unsigned char *p;
9
10    printf("----- char a -----\n");
11    printf("%p\t%02x\n", &a, a);
12
13    p=(unsigned char *)&i;
14
15    printf("----- int i -----\n");
16    printf("%p\t%02x\n", p, p[0]);
17    printf("%p\t%02x\n", p+1, p[1]);
18    printf("%p\t%02x\n", p+2, p[2]);
19    printf("%p\t%02x\n", p+3, p[3]);
20
21    p=(unsigned char *)&x;
22
23    printf("----- double x -----\n");
24    printf("%p\t%02x\n", p, p[0]);
25    printf("%p\t%02x\n", p+1, p[1]);
```

```

26 | printf("%p\t%02x\n", p+2, p[2]);
27 | printf("%p\t%02x\n", p+3, p[3]);
28 | printf("%p\t%02x\n", p+4, p[4]);
29 | printf("%p\t%02x\n", p+5, p[5]);
30 | printf("%p\t%02x\n", p+6, p[6]);
31 | printf("%p\t%02x\n", p+7, p[7]);
32 |
33 | return 0;
34 | }

```

### 実行結果

```

--- char a -----
0xbffff6ab      61
--- int i -----
0xbffff6ac      37
0xbffff6ad      00
0xbffff6ae      00
0xbffff6af      00
--- double x -----
0xbffff6b0      00
0xbffff6b1      00
0xbffff6b2      00
0xbffff6b3      00
0xbffff6b4      00
0xbffff6b5      38
0xbffff6b6      49
0xbffff6b7      bf

```

### 参考文献

- [1] 林春比古. 新訂 C 言語入門 シニア編. ソフトバンク パブリッシング, 2004.