

ポインター (メモリーとポインターの関係)

山本昌志*

2005 年 6 月 9 日

1 本日の学習内容

本日は、ポインターの基礎的なことを学習する。C 言語の学習でポインターはもっとも難しいと言われて
いるが、その内容は至って単純である。

メモリーのアドレスを格納する変数がポインターである。

本日は、このもっとも基本的なことを説明する。このことがしっかり理解できれば、ポインターなんか難
しくない。本日の学習内容は、以下の通り。

- コンピューターの基本構成とメモリーの概要について学習する。ここでは、アドレスとそこに格納され
るデータの関係が重要である。そして、データがどのように格納されるか、理解しなくてはならない。
- ポインターの基本的な使い方を示す。ここでは、ポインターに関係する演算子の使い方と意味が分か
れば良い。

2 コンピューター

2.1 コンピューターの基本構成

コンピューターは複雑な装置であるが、図 1 のような機能の集まりに分解できる¹。それぞれの機能 (装
置) は、制御信号線とデータ信号線で接続されている。そこを、0 と 1 のパルスの信号が信じられないくら
い高速でかつ調和を取って流れ、全体としてコンピューターが動作するのである。大量の信号が一つも間違
いなく伝送されるのは驚きである。この信号線は人間で言えば神経に当たり、そこに流れる内容 (情報) は、

制御信号 「… 下さい」とハードウェアに動作の指示をする信号

データ信号 「… を」とデータや命令の内容を示す信号

*独立行政法人 秋田工業高等専門学校 電気情報工学科

¹複雑なものを機能毎に分解するのは、プログラムのサブルーチン (関数) の考え方とおなじである。複雑なものは、このようにモ
ジュール単位に分割して考えるのは常套手段である

である。

コンピュータの基本構成は図 1 に示したとおりであるが、制御装置と演算装置、記憶装置、入力装置、出力装置を五大装置と呼ぶ。それぞれは、次のような働きがある。

制御装置 主記憶装置 (メインメモリー) に格納されているプログラム (命令) を受け取り、それを解読し、電気信号に変え、各装置に指令を出す。

演算装置 主記憶装置に格納されているデータを受け取り、制御装置の指令に従い、それを加工 (処理) する。

記憶装置 記憶装置は、主記憶装置と補助記憶装置がある。

主記憶装置 命令とデータからなるプログラムを格納する。

補助記憶装置 主記憶装置には容量に制限があるので、それを越えるものをここに格納する。また、主記憶装置は電源を切ると、データが失われるので、半永久的に残したい場合、補助記憶装置に格納する。ハードディスクや CD-ROM がこれに当たる。

入力装置 コンピューターの外部からデータを取り込む装置である。キーボードやマウス等がこれに当たる。

出力装置 主記憶装置に格納されているデータをコンピューター外部に出力する装置である。ディスプレイやプリンター等がこれに当たる。

モデムや LAN カードのように、入出力装置にもなっているものがあるので注意が必要である。

これらのなかで、制御装置と演算装置をまとめて中央制御装置 (CPU: Central Processing Unit) と言う。これを一つのチップにまとめたものを MPU (Micro Processing Unit) と言うことになっている。ただ、MPU と CPU はほとんど同義語として使われるので、本講義では全て CPU に統一する。その方が諸君もなじみ深いであろう。

2.2 メモリーと CPU

ここで、コンピュータ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 2 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、コンピュータの原理的なモデルを図 2 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ろうが、C 言語ではそこまで要求しない。

本日のメインテーマはポインターであるが、それを深く理解するために、メモリーの知識が必要である。しかし、そんなに難しいことではない。

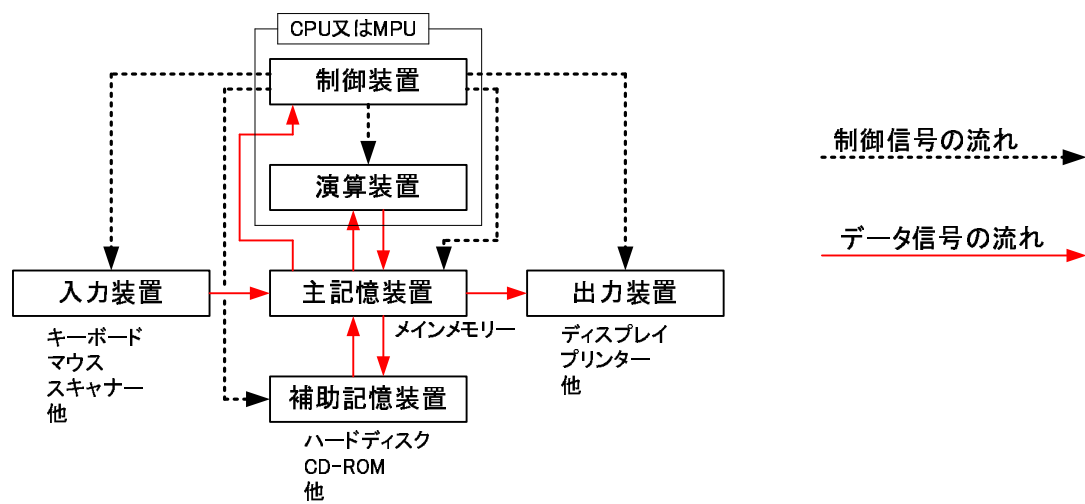


図 1: コンピューターの基本構成

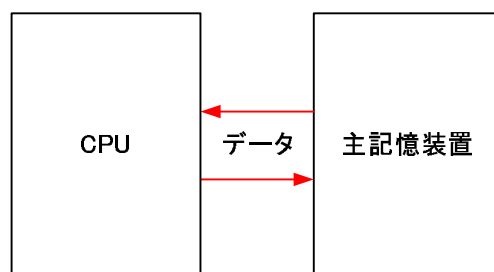


図 2: もっとも原始的なコンピューター

2.3 メインメモリのモデル

メインメモリ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、メインメモリのモデルを理解することが重要である。

メインメモリの役目は、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して初めて、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められている。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²。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3に示す。

図を見て分かるとおり、2進数の表現は桁数が多くて人間にとって大変である³。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桁で表すことができ、分かりやすくなる。その様子を図4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 3: メモリーのモデル (2 進数)

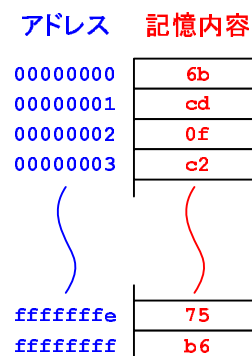


図 4: メモリーのモデル (16 進数)

²CPU 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。

³コンピューターにとっては全然大変でない

2.4 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。付録のリスト 3 にデータ型のバイト数を調べるプログラムを載せておく。

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、前回の講義の付録を見よ。
- $(55)_{10} = (37)_{16}$ 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。実際にこれを確かめるプログラムを、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 5 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 5 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。

アドレス	記憶内容	
bffff6ab	61	char a='a'
bffff6ac	37	
bffff6ad	00	int i=55
bffff6ae	00	
bffff6af	00	
bffff6b0	00	
bffff6b1	00	double x=-7.696151733398438e-4
bffff6b2	00	
bffff6b3	00	
bffff6b4	00	
bffff6b5	38	
bffff6b6	49	
bffff6b7	bf	

図 5: メモリー中に格納されたデータの例

2.5 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。この辺のことも、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 関数が書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 6 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ー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 6 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 を宣言しているがメモリー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、いずれもメモリーの中に格納される。
- 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、メモリー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。

3 ポインターとなにか

3.1 ポインターの例

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。2.3 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あてきそうである。具体的には、hoge とい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;
```

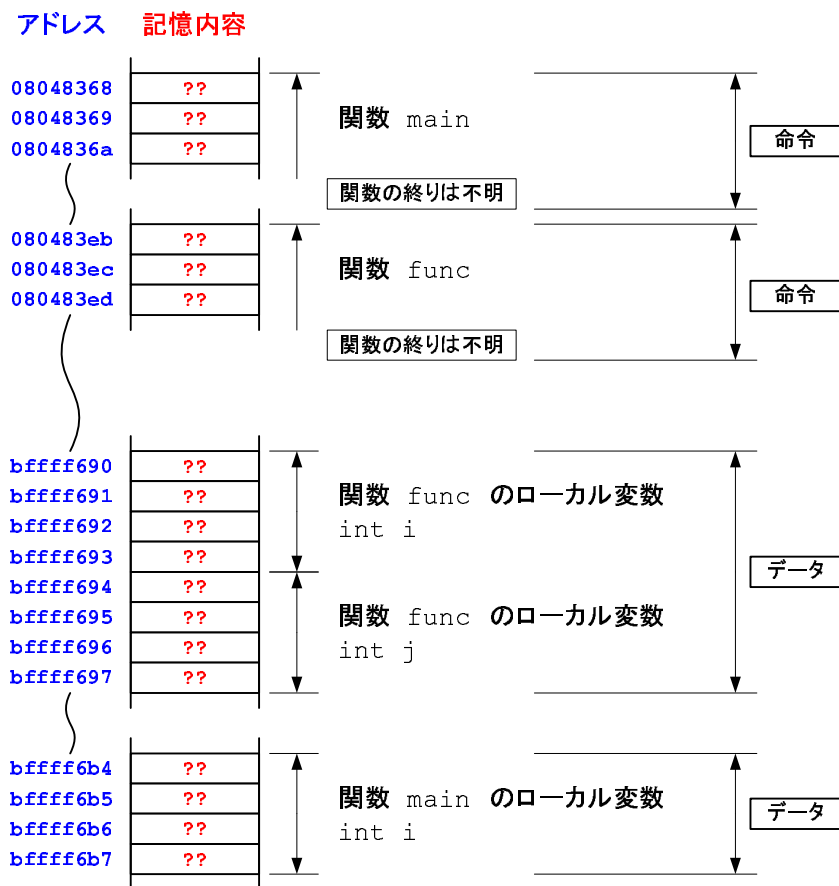



図 6: プログラムのメモリへの格納。記憶の内容は不明なので、??としている。

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⁴。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ー (pointer 指し示すもの) と言うのであろう。

⁴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。

- ポインターとは、アドレスを格納する変数のことである^a。
- ポインターの宣言には、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。
- ポインターが示しているデータの値を取り出すためには、ポインター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。

^a正確に言うとなんとなく違うが、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、ポインター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。

- ポインター変数 *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ー *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 整数型のポインター *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、ポインター *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 ポインター *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 ポインター *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、ポインターはアドレスなので、強制型変換 (キャスト) により、符号なし整数にしている⁵。

15 行 ポインター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。

⁵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7のように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。

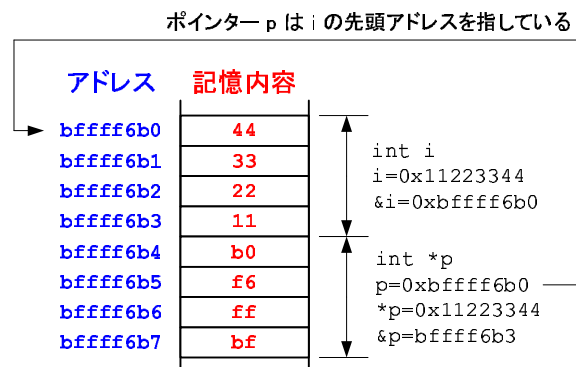


図 7: リスト 2 のプログラム実行後のメモリーの内容

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。

- 通常の変数には値が、ポインターにはアドレスを格納する。
- アドレス演算子&は、それに引き続く変数 (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、それに引き続くポインターの値を返す。

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

```