第１章
ポインタの基本

Ｃ言語のプログラムは、

・ポインタを使用すれば、簡潔で効率よく実現できる。

・ポインタを使用しなければ、実現不可能あるいは困難な場合がある。

などの理由から、ポインタが多用されます。ポインタをマスターしなければ、Ｃ言語の本質を身に付けることはできないといっても過言ではありません。

もっとも、初心者にとっては、ポインタはなかなか理解しにくいもののようです。

本章では、ポインタの基本的な事項をやさしく学習していきましょう。

© 2001 本 PDF のプログラムを含むすべての内容は著作権法上の保護を受けています。
著者・発行者の許諾を得ず、無断で複写・複製をすることは禁じられております。

謝 辞 フォントを埋め込んだ PDF 文書の第三者への配布のライセンス料を特別に無償で許可してくださった株式会社ニイスの御厚意に感謝いたします。

1-1　ポインタとは

オブジェクトとアドレス

— C言語の学習を始めてから、半年ほど経ちましたが、いまだに“ポインタ”というものが、よく分かりません。そもそもポインタとは何ですか？

数値などを格納する普通の変数とは異なり、ポインタは変数を**指す**のです。

— 変数を指すって？

その話に入(はい)る前に、変数について理解しておきましょう。まずは**Fig.1-1**を見てください。ここには、**int** 型とその変数 *m*，*n*、**double** 型とその変数 *x*，*y* を示しています。

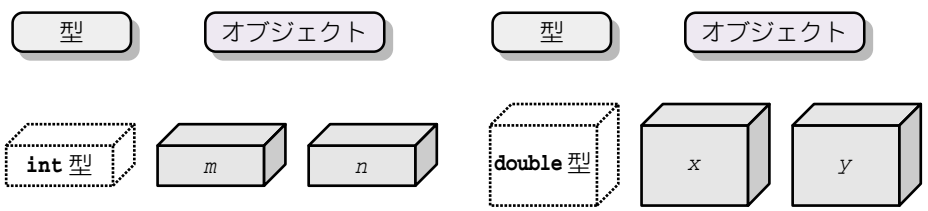


Fig.1-1　int 型と double 型のオブジェクト

— 点線の箱が《型》で、実線の箱が《変数》ですね。

はい。点線の箱である**型**は、その諸性質を内に秘めている**設計図**です。一方、実線の箱である**変数**は、設計図に基づいて作られた**実体**です。

実体はコンピュータの記憶域を占有しますので、*m*，*n* や *x*，*y* などの変数は、正式には**オブジェクト**（*object*）と呼ばれます。**Fig.1-2** のように考えましょう。

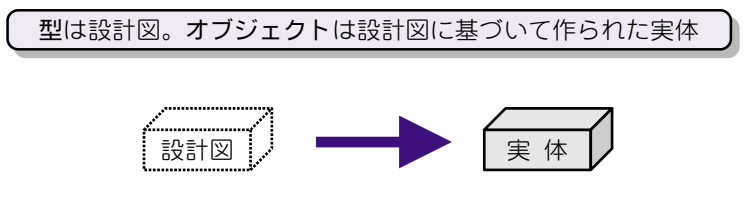


Fig.1-2　型とオブジェクト

型が“タコ焼き”を作るための《カタ》であるとすれば、オブジェクトは実際に食べることのできる“タコ焼き”です。

— なるほど。“型”から作られた実体が“オブジェクト”ですね。

そうです。さて、**Fig.1-3(a)**に示すように、個々のオブジェクトは、独立した箱であると考えてもよいものです。

しかし、実際には、図**(b)**に示すように、記憶域中の一部分を占める箱なのです。

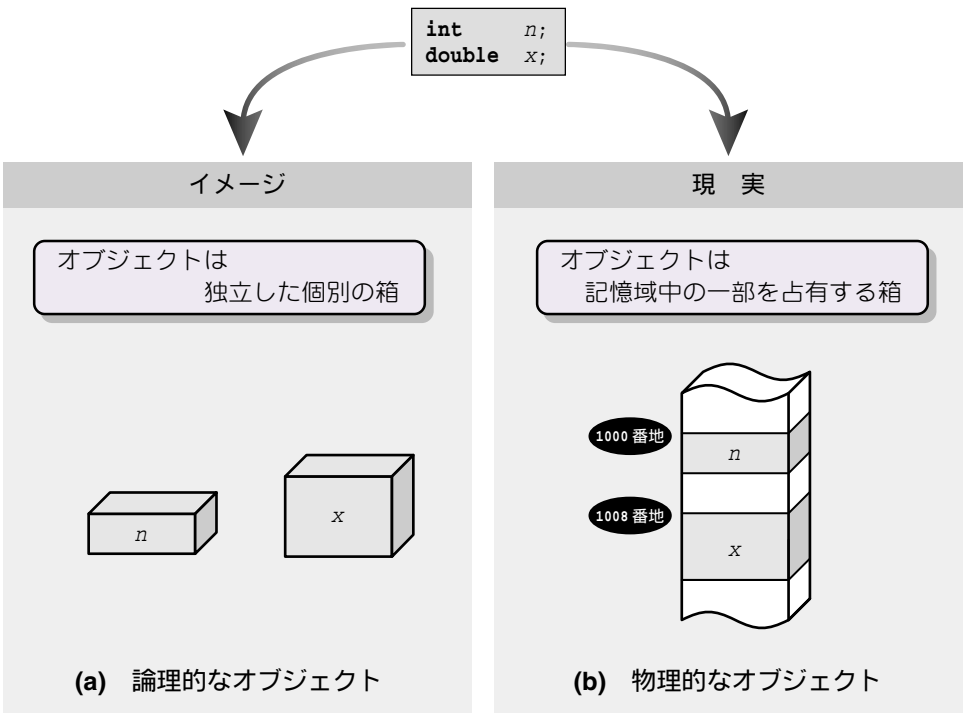


Fig.1-3　論理的なオブジェクトと物理的なオブジェクト

— なるほど。まっすぐな記憶域上にオブジェクトが雑居しているんだ。図**(b)**は、オブジェクト *n* が 1000 番地に格納され、オブジェクト *x* が 1008 番地に格納されている様子を表しているのですね。

そうです。記憶域上のどこに格納されているのかを示すのが**アドレス**（*address*）です。ちょうど、住所での“**○○番地**”に相当します。

▶ すべての実行環境で、記憶域が物理的にまっすぐ連続した領域として実現されるわけではありません。

アドレス演算子

オブジェクトのアドレスを調べてみましょう。プログラム例を **List 1-1** に示します。

▶ 日本の多くのパソコンで採用されている JIS コード体系では、逆斜線（バックスラッシュ）記号 \ の代わりに円記号 ¥ を使います。そのような環境でプログラムを打ち込む際には、円記号に置きかえてください。

List 1-1

```
/*
    オブジェクトの値とアドレスを表示
*/

#include  <stdio.h>

int main(void)
{
    int  nx = 15;        /* nxはint型 */
    int  ny = 73;        /* nyはint型 */

    printf("nxの値＝%d\n", nx);              /* nxの値を表示 */
    printf("nyの値＝%d\n", ny);              /* nyの値を表示 */

    printf("nxのアドレス＝%p\n", &nx);       /* nxのアドレスを表示 */
    printf("nyのアドレス＝%p\n", &ny);       /* nyのアドレスを表示 */

    return (0);
}
```

実行結果一例

```
nxの値＝15
nyの値＝73
nxのアドレス＝1000
nyのアドレス＝1008
```

▶ 実行例に示している ***1000*** および ***1008*** は一例であって、この値が表示されるわけではありません。なお、これ以降の実行例でも、処理系や実行環境などの条件によって、値が異なる部分は ***斜体の太字*** で示します。

— 変数 *nx* と *ny* は、それぞれ 15 と 73 で初期化されていますね。

はい。15 や 73 を**初期化子**（**Column 1-1**）と呼ぶのでしたね。

さて、オブジェクトが格納されているアドレスは、**アドレス演算子**（*address operator*）と呼ばれる**単項 & 演算子**（*unary* & *operator*）によって取り出します。したがって、**&***nx* はオブジェクト *nx* のアドレスとなり、**&***ny* はオブジェクト *ny* のアドレスとなります。

Column 1-1　初期化子と初期値

初期化子（*initializer*）は、オブジェクトの宣言において = 記号の右側におかれます。たとえば、以下の宣言では**紺色の部分**が初期化子です。

```
int  n = 4;
int  a[3] = {1, 2, 3};
```

なお、初期化子の値が、そのまま初期値になるとは限りません。たとえば、

```
int  z = 3.5;
```

と宣言した場合、*z* は **int** 型ですから、その初期値は 3.5 ではなく 3 となります。

重　要

オブジェクト *x* のアドレスは **&***x* によって取り出せる。

― アドレス演算子 **&** は**オペランド**（**Column 1-2**）のアドレスを取り出すのですね。

はい。アドレスの値を表示するために ***printf*** 関数に与える変換指定 **%p** の p は、ポインタすなわち *pointer* に由来します。

表示の形式は処理系に依存しますが、一般には数桁の 16 進数です（**Fig.1-4**）。

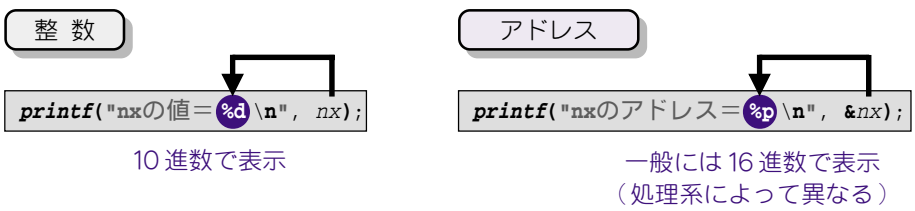


Fig.1-4　整数の表示とアドレスの表示

― この実行結果は、**Fig.1-5** のように、*nx* が 1000 番地に格納され、*ny* が 1008 番地に格納されていることを示しているのですね！

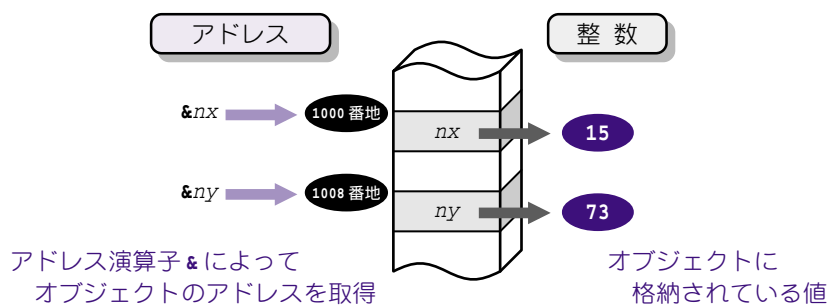


Fig.1-5　オブジェクトの値とアドレス

▶ アドレス演算子 **&** によって取得された値が、物理的なアドレスと一致するという保証はありません。したがって、処理系によっては、何らかの変換を行った値が得られます。

“プログラム上から見た”アドレスと考えましょう。

Column 1-2　演算子とオペランド

演算を行う記号 **+**，*****，**&** などが**演算子**（*operator*）であり、演算の対象となる式が**オペランド**（*operand*）です。たとえば、加算を行う式 *x* **+** *y* において、演算子は **+** であり、そのオペランドが *x* と *y* です。

ポインタとは

アドレスについて理解したところで、**ポインタ**（*pointer*）の話に進みましょう。まず、ポインタを宣言する方法は知っていますか。

1

― その程度でしたら分かっています。次のように * を付けて宣言します。

```
int  *p;          /* pは int *型のポインタ */
```

いいですね。*p*は“**int** 型”ではなく『**int** 型オブジェクトへのポインタ型』略して『**int** 型へのポインタ型』、あるいは単に『**int** * 型』と呼ばれる型となります。

もちろん、ポインタとして宣言された*p*は、『**int** 型オブジェクトへのポインタ型』の設計図から作られた実体であり、一種のオブジェクトです。

なお本書では、**Fig.1-6** に示すように、丸いグラデーションのかかった図（内側から外側にかけて濃くなっていく図）でポインタを表します。

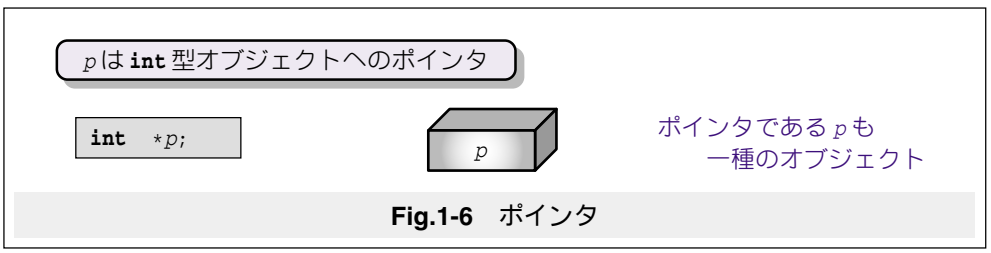


Fig.1-6　ポインタ

ポインタと普通のオブジェクトとの違いを **List 1-2** のプログラムで確認しましょう。

List 1-2

```
/*
    整数の値とポインタの値を表示
*/

#include  <stdio.h>

int main(void)
{
    int  nx;            /* nxはint型（整数） */
    int  *pt;           /* ptはint *型（ポインタ） */

    nx = 57;                            /* nxに57を代入 */
    pt = &nx;                           /* ptにnxのアドレスを代入 */

    printf(" nxの値＝%d\n",  nx);       /* nxの値を表示 */
    printf("&nxの値＝%p\n", &nx);       /* nxのアドレスを表示 */
    printf(" ptの値＝%p\n",  pt);       /* ptの値を表示 */

    return (0);
}
```

実行結果一例

```
 nxの値＝57
&nxの値＝1000
 ptの値＝1000
```

― **int** 型の *nx* に 57 を代入して、その値を表示していることは分かります。ポインタである *pt* に **&***nx* を代入するのは、どういう意味ですか。

アドレス演算子 **&** は、オペランドのアドレスを取り出しますので、*pt* には *nx* のアドレスが代入されることになります。その様子を示したのが **Fig.1-7** です。

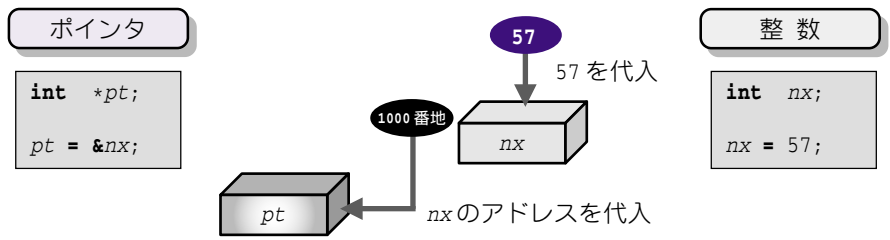


Fig.1-7　整数とポインタ

― なるほど。ポインタ *pt* には *nx* のアドレスである 1000 番地が代入されるんですね。だから、**&***nx* と *pt* は同じ 1000 という値になるんだ！

そうです。アドレス演算子 **&** について一般的にまとめると次のようになります。

重　要

Type 型のオブジェクト *x* に対して **&** 演算子を適用した **&***x* は、***Type*** ＊型のポインタであり、その値は *x* のアドレスである。

― なるほど。**&***nx* も *pt* も、その型は **int** 型へのポインタ型であって、その値がアドレスである 1000 番地なのですね。

はい。**Fig.1-8** に示すように、**int** 型が『整数』を値としてもつのに対して、**int** ＊型のポインタは『“整数を格納するオブジェクト”のアドレス』を値としてもちます。

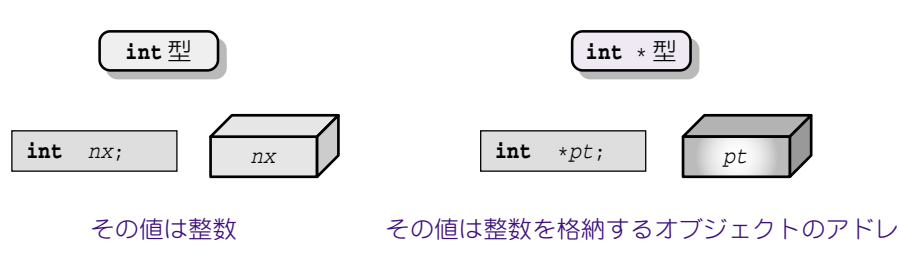


Fig.1-8　int 型と int へのポインタ型

ポインタはオブジェクトを指す

さて、point が“指す”という意味をもつ単語であることは知っていますね。

1

― はい。もちろんです。

それにerを付けたのがpointerです。したがって、ポインタは、“指すもの”、“指す人”、“指摘者”、“指針”となります。

そこで、ポインタに関して、次のことを必ず覚えてください。

重　要

ポインタ *p* の値がオブジェクト *x* のアドレスであるとき、

p は *x* を指す

という。

ポインタ *p* がオブジェクト *x* を指すことを図示したものが **Fig.1-9** です。

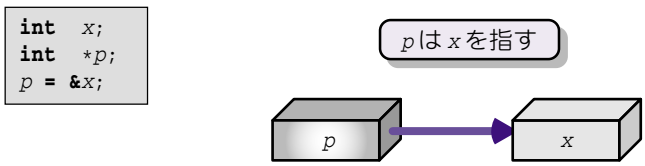


Fig.1-9　ポインタがオブジェクトを“指す”（その１）

― やっと、“指す”という話に戻りましたね！

Column 1-3　型について

オブジェクトや関数が返却する値の意味を決定付ける型（*type*）には、**オブジェクト型**（*object type*）、**関数型**（*function type*）、**不完全型**（*incomplete type*）の三種類があります。

さて、オブジェクト *n* が **int** 型で、*x* が **double** 型であるとします。**int** 型のオブジェクトは、その値として整数のみをもつことができます。したがって、

```
n = 3.5;        /* int型オブジェクトに浮動小数点数値を代入 */
```

と、浮動小数点数値を代入しようとしても、小数点以下は切り捨てられますから、*n* の値は 3 となります。一方、

```
x = 3.5;        /* double型オブジェクトに浮動小数点数値を代入 */
```

と代入すると、**double** 型である *x* の値は 3.5 となります。

このように、型に基づいてオブジェクトのふるまいが決定されるのです。

なお、*p* も *x* も記憶域の一部を占有するオブジェクトですから、この図は **Fig.1-10** のようにも表すことができます。

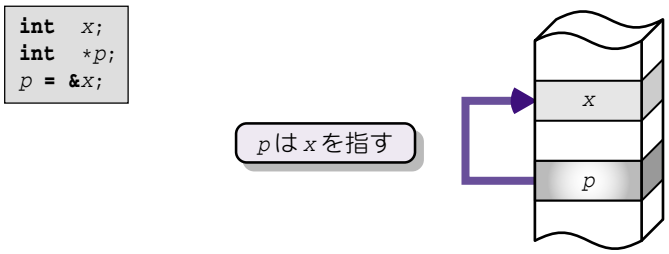


Fig.1-10　ポインタがオブジェクトを“指す”（その２）

― どちらの図も見たことがありますが、よく分からないんですよね。

なるほど。これらの図は、

ポインタ *p* がオブジェクト *x* を指す

というイメージを表すものですが、ポインタを初めて勉強するときの図としては、ちょっと難しいかもしれません。

▶ ポインタはオブジェクトだけでなく関数を指すこともできます。関数へのポインタについては、第９章で学習します。

Column 1-4　キャストによる型変換

キャスト演算子（*cast operator*）と呼ばれる **( )** 演算子は、

(*型*) *式*

という形式で利用し、*式*の値を*型*としての値に変換したものを生成します。また、このような型変換を行うことを**キャストする**（*cast*）といいます。

変数 *x* と *y* が **int** 型であり、それらの平均値を求める例でキャストについて考えましょう。*x* の値が 4 で *y* の値が 3 であれば、

```
(x + y) / 2
```

では、加算も除算も整数どうしの演算ですから、その結果も整数となります。したがって、以下のように小数点以下の部分が切り捨てられて結果は 3 となります。

```
7 / 2  →  3
```

それでは、次のように、加算の結果を **double** 型にキャストしてみましょう。

```
(double)(x + y) / 2
```

double 型と **int** 型との演算においては、**int** 型が暗黙の型変換によって **double** 型に変換されます。したがって、**double** 型どうしの除算が行われて、3.5 が得られることになります。

```
(double)7 / 2  →  7.0 / 2  →  7.0 / 2.0  →  3.5
```

間接演算子

次に **List 1-3** のプログラムを考えましょう。

1

List 1-3

```
/*
    ポインタが指すオブジェクトの値を表示
*/

#include  <stdio.h>

int main(void)
{
    int  nx;            /* nxはint型 */
    int  *pt;           /* ptはint *型 */

    nx = 57;                            /* nxに整数57を代入 */
    pt = &nx;                           /* ptにnxのアドレスを代入 */

    printf(" nxの値＝%d\n",  nx);       /* nxの値を表示 */
    printf("*ptの値＝%d\n", *pt);       /* ptが指すオブジェクトの値を表示 */

    return (0);
}
```

実行結果

```
nxの値＝57
*ptの値＝57
```

― 表示される *pt の値は、nx と同じ 57 となっていますね。

ポインタに**間接演算子**（*indirection operator*）と呼ばれる**単項 * 演算子**（*unary * operator*）を適用した式は、そのポインタが指すオブジェクトを表します。

― このプログラムでは、pt が nx を指しているわけですから、pt に * を適用した式 *pt は、nx そのものを意味するのですね。

そうです。一般に、p が x を指すとき、*p は x の**エイリアス**（*alias*）すなわち**別名**となります。変数 x に対して *p という『あだ名』が付いたと考えていいでしょう。

重　要

ポインタ p がオブジェクト x を指すとき、*p は x のエイリアス（別名）となる。

Fig.1-11 を見てください。前ページでも示した**(a)**の図は、ポインタ p がオブジェクト x を指している様子を表します。

このとき、*p は、オブジェクト x に与えられたエイリアスとなります。本書では、**(b)**に示すように、右側に飛び出た青い箱でエイリアスを表すことにします。

― なるほど。**(a)**はポインタがオブジェクトを指すことをイメージした図であるのに対して、**(b)**は間接演算子の働きをイメージした図なんですね。

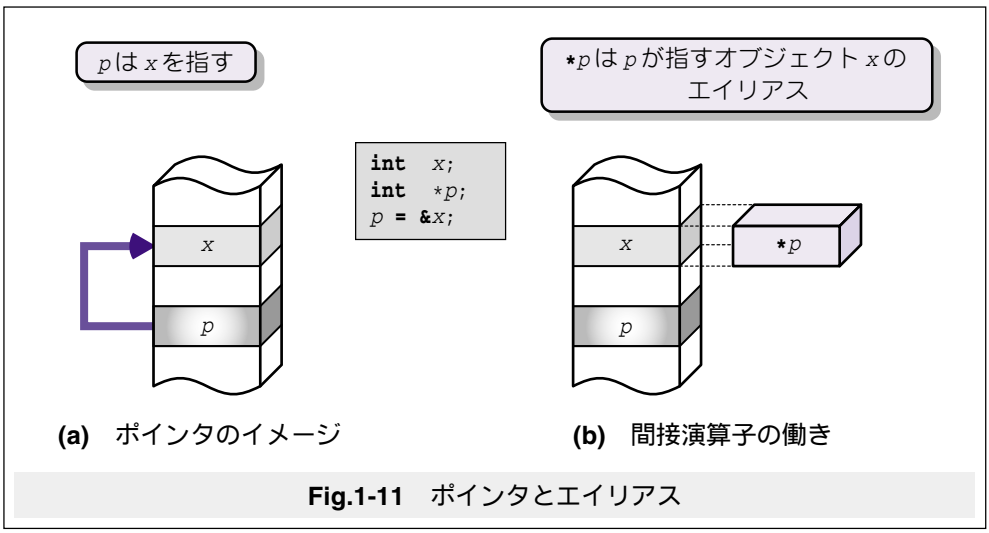


Fig.1-11 ポインタとエイリアス

そうです。さて、ここで *p という変数が存在すると勘違いしないでください。

— ん？ どういう意味ですか。

たとえば、**int** 型の変数 *n* の値が 50 であれば、-*n* の値は何でしょう。

— もちろん -50 です！

そうですね。変数 *n* に対して単項 - 演算子を適用した式 -*n* を評価することによって得られる値は -50 ですが、決して -*n* という変数は存在しないことは分かるでしょう。

ポインタ *p* に間接演算子 * を適用した式は *p ですが、そのような変数が存在するわけではありません。

— なるほど。単項*演算子は、ポインタを通じてオブジェクトを間接的に扱うための演算子だから、間接演算子と呼ばれることが分かりました。

なお、ポインタでない普通の**int**型オブジェクトに対して間接演算子を適用することはできません。

重 要

間接演算子はポインタに対してのみ適用できる。

— ということは、

```
printf("*nx の値＝ %d\n", *nx);        /* エラー */
```

はエラーとなりますね！

ポインタが指すオブジェクトへの代入

さあ、それでは **List 1-4** のプログラムは、もう理解できますね。

1

List 1-4

```
/*
    ポインタを通じて間接的にオブジェクトに値を代入
*/

#include  <stdio.h>

int main(void)
{
    int  nx, ny;
    int  *p;

    p  = &nx;    /* pにnxのアドレスを代入：pはnxを指す */
    *p = 100;    /* pが指すnxに100を代入 */

    p  = &ny;    /* pにnyのアドレスを代入：pはnyを指す */
    *p = 300;    /* pが指すnyに300を代入 */

    printf("nxの値＝%d\n", nx);
    printf("nyの値＝%d\n", ny);

    return (0);
}
```

実行結果

```
nxの値＝100
nyの値＝300
```

―　はい。**Fig.1-12** のような感じですね。アドレス演算子 **&** や間接演算子 ***** は、その意味を理解すれば、意外と簡単ですね！

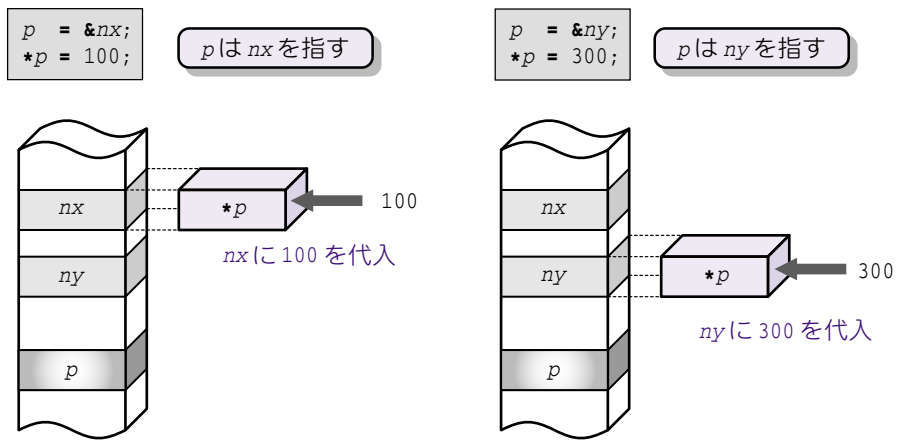


Fig.1-12　ポインタが指すオブジェクトへの代入

バイトとアドレス

― ところで、アドレスは、各オブジェクトに対して一つずつ与えられるのですか。

いいえ。アドレスは、各バイトに対して与えられます。

― ということは、８ビットごとにアドレスが付くんですね。

多くの環境ではそうです。しかし、１バイトが９ビットとか32ビットのコンピュータも実在します。したがって、C言語でも、1バイトのビット数は処理系によって異なるものであり、その値は『少なくとも８である』と定義されています。

― どうすれば、1バイトのビット数を調べることができるのですか。

その値は <limits.h> ヘッダ中で、**CHAR_BIT** として定義されています（**Fig.1-13**）。

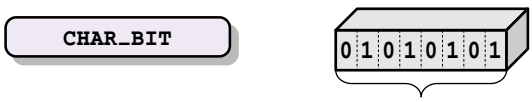


Fig.1-13　1バイトのビット数と**CHAR_BIT**

重　要

１バイトのビット数は処理系によって異なる。その値は <limits.h> ヘッダ中で **CHAR_BIT** として定義されている。

１バイトのビット数を表示するプログラム例を **List 1-5** に示します。

List 1-5

```
/*
    １バイトに含まれるビット数を表示
*/

#include  <stdio.h>
#include  <limits.h>

int main(void)
{
    printf("１バイトは%dビットです。\n", CHAR_BIT);

    return (0);
}
```

実行結果一例

```
１バイトは8ビットです。
```

オブジェクトの大きさと sizeof 演算子

— **char** 以外の型は、何ビットなのですか？

1

それも処理系によって異なります。**List 1-6** のプログラムで、**char** 型、**int** 型、**long** 型の大きさを調べてみましょう。

List 1-6

```
/*
    char型、int型、long型の大きさを表示
*/

#include  <stdio.h>

int main(void)
{
    printf("char型は%uバイトです。\n",  (unsigned)sizeof(char));
    printf("int  型は%uバイトです。\n",  (unsigned)sizeof(int));
    printf("long型は%uバイトです。\n",  (unsigned)sizeof(long));

    return (0);
}
```

実行結果一例

```
char型は1バイトです。
int  型は4バイトです。
long型は4バイトです。
```

ある型のオブジェクトのバイト数は、**sizeof 演算子**（*sizeof operator*）を利用した

　　sizeof（*型名*）

で得られます。

なお、**sizeof**(**char**)はあらゆる処理系で１となります。さて、君が実行した環境では、**int** 型が２バイト、**long** 型が４バイトのようですね（**Fig.1-14**）。

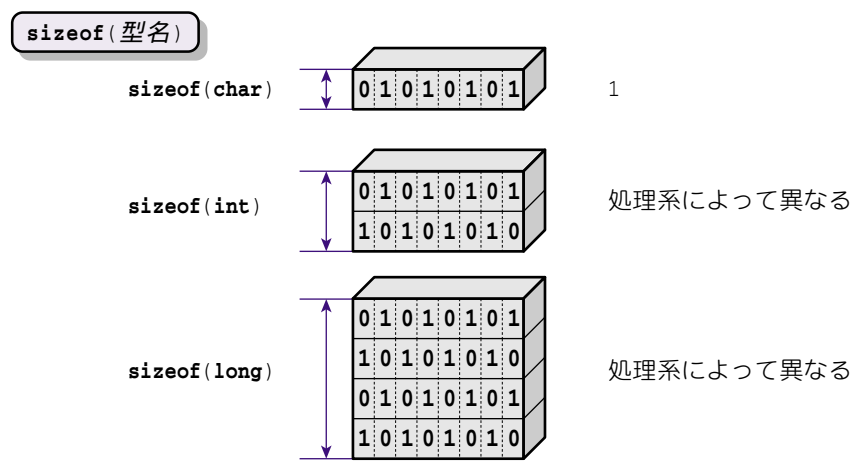


Fig.1-14　sizeof演算子の生成する値

— ところで **sizeof** 演算子が生成するのは **unsigned** 型なのですか？

いいえ。生成されるのは、<stddef.h> ヘッダで定義されている **size_t** 型です。

この **size_t** 型は符号無し整数型ですが、具体的に **unsigned short** 型、**unsigned int** 型、**unsigned long** 型のどれと等しいかは処理系によって異なります。

したがって、いずれかの符号無し整数型にキャスト（**Column 1-4**：*p.9*）した上で表示を行う必要があります（**Column 1-5**）。

重　要

sizeof演算子が生成した値は、**unsigned**型や**unsigned long**型などの符号無し整数型にキャストを行って表示せよ。なお、大きな値が予想される場合は、**unsigned long**型にキャストするのが無難である。

— ということは、

```
printf("int 型は %u バイトです。\n", (unsigned)sizeof(int));
```

あるいは

```
printf("int 型は %lu バイトです。\n", (unsigned long)sizeof(int));
```

などとすべきなのですね！

Column 1-5　sizeof演算子が生成する値の表示

sizeof 演算子が生成する値を ***printf*** 関数で表示する際に、符号無し整数型にキャストを行う必要がある理由を考えましょう。

たとえば、**int** 型が 2 バイトで、**long** 型が 4 バイトであり、さらに <stddef.h> ヘッダで

```
typedef  unsigned long  size_t;
```

と宣言されていたとします。このとき、**size_t** 型は **unsigned long** 型の同義語となりますから、その大きさは 4 バイトとなります。

▶ **typedef** 宣言については、*p.19* の **Column 1-6** を参照してください。

ここで、キャストを行わない

```
printf("long型は%uバイトです。\n", sizeof(long));
```

は、2 バイトの **unsigned int** 型の引数を期待する ***printf*** 関数に対して、4 バイトの **unsigned long** 型の値を渡すことになります。第 10 章で引数の受渡しについて学習しますが、これは不正であり、期待するような結果は得られません。

もちろん、**size_t**型が**unsigned int**型の同義語である処理系であれば、問題はありません。すなわち、キャストをしなければ、処理系によって正しく動作したりしなかったりする、可搬性の低いプログラムになるのです。

したがって、以下のようにキャストを行うことによって、処理系に依存することなく、渡す型と受け取る型を確実に一致させるようにしましょう。

```
printf("long型は%uバイトです。\n", (unsigned)sizeof(long));
printf("long型は%luバイトです。\n", (unsigned long)sizeof(long));
```

型とビット数

― それでは、**int** 型のビット数は **sizeof**(**int**) * **CHAR_BIT** で求めることができますね！

1

その通り！ … といいたいのですが、少し補足が必要です。もしも1バイトが8ビットで **sizeof**(**int**)が4であれば、**int** 型は確かに32ビットを占有します。

しかし、**Fig.1-15** に示すように、4ビットは未使用で28ビットのみを有効なビットとして使う処理系もあります。すなわち、一般的には、次のようになります。

重　要

Type 型の有効ビット数は **sizeof**(***Type***) * **CHAR_BIT** であるという保証はない。

Fig.1-15　整数型のビット構成

参考までに **int** 型の有効なビット数を表示するプログラムを **List 1-7** に示します。

List 1-7

```
/*
    int型の有効ビット数を表示
*/

#include  <stdio.h>

/*--- int型／unsigned int型のビット数を返す ---*/
int int_bits(void)
{
    int       count = 0;
    unsigned  x = ~0U;

    while (x) {
        if (x & 1U) count++;
        x >>= 1;
    }
    return (count);
}

int main(void)
{
    printf("int型の有効ビットは%dビットです。\n", int_bits());

    return (0);
}
```

実行結果一例

```
int型の有効ビットは16ビットです。
```

▶ 本プログラムの動作原理などは、ポインタでなくビット演算に関わることですので、詳しい解説は省略します。詳細は、『明解C言語 入門編』第7章をご覧ください。

バイト順序

Fig.1-16 のように、**int** 型が 2 バイト 16 ビットであるとします。このとき、図**(a)**に示すように、下位バイトが先頭側（低いアドレス）に配置される処理系もありますし、図**(b)**に示すように、下位バイトが末尾側（高いアドレス）に配置される処理系もあります。

ちなみに、下位バイトが低アドレスをもつものが**リトルエンディアン**、逆に高アドレスをもつものが**ビッグエンディアン**と呼ばれます。

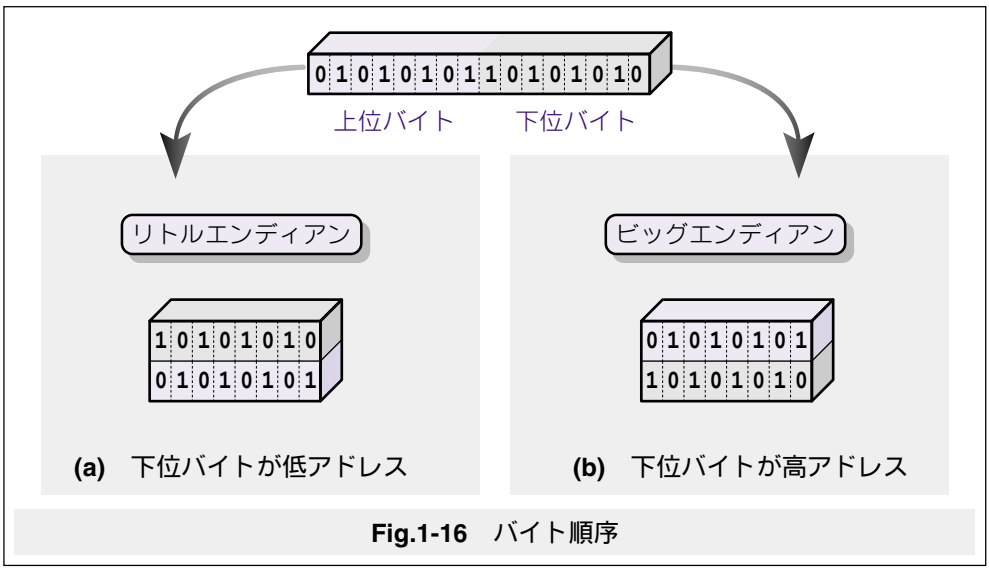


Fig.1-16　バイト順序

▶ Jonathan Swift の 1726 年の小説『ガリバー旅行記』で、小人国では“卵は太い方から割るべきだ。”とするビックエンディアンと“卵は細い方から割るべきだ。”とするリトルエンディアンとが対立する話に由来します。1981 年に、Danny Cohen の“On holy wars and a plea for peace”によって、この言葉がコンピュータの世界に持ち込まれました。

このように、バイト順序は処理系に依存してしまうのですが、ポインタに関して以下のことを覚えておきましょう。

重　要

複数バイトにまたがるオブジェクトへのポインタは、その先頭アドレスすなわち、最も低いアドレスを指すものであると考えよ。

▶ 大きさが *n* バイトのオブジェクトであれば、*x* 番地から *x* + *n* - 1 番地にわたって格納されます。したがって、厳密には、

《*x* 番地を先頭に *n* バイトにわたって格納されている》

などと表現すべきでしょうが、本書では、

《*x* 番地に格納されている》

と省略して表現することにします。

ポインタの大きさ

―　ところで、ポインタは何バイトですか？

1

ポインタの大きさも処理系によって異なりますが、**sizeof** 演算子を使って調べることができます。**List 1-8** のプログラムで確認しましょう。

List 1-8

```
/*
    int型とint *型の大きさを表示
*/

#include  <stdio.h>

int main(void)
{
    int  nx;        /* int型 */
    int  *pt;       /* int *型 */

    printf("int  型は%uバイトです。\n", (unsigned)sizeof(int));
    printf("int *型は%uバイトです。\n", (unsigned)sizeof(int *));

    printf(" nxは%uバイトです。\n", (unsigned)sizeof(nx));
    printf("*ptは%uバイトです。\n", (unsigned)sizeof(*pt));
    printf(" ptは%uバイトです。\n", (unsigned)sizeof(pt));
    printf("&nxは%uバイトです。\n", (unsigned)sizeof(&nx));

    return (0);
}
```

実行結果一例

```
int  型は2バイトです。
int *型は4バイトです。
 nxは2バイトです。
*ptは2バイトです。
 ptは4バイトです。
&nxは4バイトです。
```

―　**sizeof**(*nx*)という式がありますが、( )の中は型名でなくてもいいんですか？

先ほどは、**sizeof**(*型名*)を学習しましたが、**sizeof** 演算子には、もう一つの形式があります。

sizeof *式*

を評価すると、その*式*を表すのに何バイトが必要であるか、という値が得られます。

この形式では、文法上( )は不要ですから、

sizeof *nx*

とすればいいのですが、前後の式によっては紛らわしくなることがありますので、

sizeof(*nx*)

と、式を( )で囲むことにしましょう（**演習 1-2**）。

二つの **sizeof** については、**Fig.1-17** にまとめておきます。

―　なるほど。

実行結果から、僕の環境では、**int** 型は２バイトで、**int** * 型は４バイトであることが分かりました。

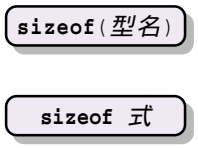


Fig.1-17　二つのsizeof

■ 演習 1-1

nx が **int** 型で、*pt* が **int** * 型であり、*pt* が *nx* を指しているとする。このとき、式 ***&***nx* と式 **&***pt* は何を意味するのだろうか。

その値や、大きさを表示するプログラムを作成して考察を行え。

■ 演習 1-2

以下に示す各式の値を表示するプログラムを作成して考察を行え。

```
sizeof*pt          sizeof(unsigned)-1        sizeofnx+2
sizeof&nx          sizeof(double)-1          sizeof(nx+2)
sizeof-1           sizeof((double)-1)        sizeof(nx+2.0)
```

ここで、*nx* は **int** 型で、*pt* は **int** * 型であるとする。

※（ ）がないと、プログラムが読みにくくなることが分かりますね。

Column 1-6　typedef宣言

初心者は、**typedef** 宣言を“新しい型を作る宣言”と勘違いすることが多いようですが、これは誤りです。正しくは、“既存の型に対して同義語を与える宣言”です。

たとえば、

```
typedef int  INTEGER;
```

は、***INTEGER*** を **int** の同義語と宣言します。したがって、プログラムの読みやすさなどの点を除けば、

```
INTEGER  a, x;
```

という宣言は、実質的には、

```
int  a, x;
```

と同じことになります。

なお、**typedef** 宣言は、プログラムの読みやすさを向上させるだけではありません。何らかの都合で ***INTEGER*** を **long int** 型の同義語に変更したい場合は、先ほどの宣言を

```
typedef long int  INTEGER;
```

に変更するだけでよく、

```
INTEGER  a, x;
```

には手を加える必要はありません。

ポインタの宣言と初期化

さて、*pt* と *pc* の二つを **int** へのポインタ型として一度にまとめて宣言してごらん。

1

— はい。次のように宣言します！

```
int  *pt, pc;
```

いえいえ、そのように宣言すると、

```
int  *pt;          /* pt は int * 型のポインタ */
int  pc;           /* pc は int   型の整数 */
```

と解釈されて、*pc* はポインタではなくて、ただの **int** 型のオブジェクトになります。

複数のポインタをまとめて宣言するときは、

```
int  *pt, *pc;
```

と、それぞれに * が必要です。

重　要

二つ以上のポインタを宣言するときは、それぞれに*を忘れないように。

それでは、**List 1-9** のプログラムを見てください。

List 1-9

```
/*
    ポインタの初期化
*/

#include  <stdio.h>

int main(void)
{
    int  nx = 100;      /* nxの値は100 */
    int  ny = 200;      /* nyの値は200 */
    int  *px = &nx;     /* pxはnxを指すポインタ */
    int  *py = &ny;     /* pyはnyを指すポインタ */

    printf(" nxの値＝%d\n",  nx);   /* nxの値 */
    printf(" nyの値＝%d\n",  ny);   /* nyの値 */
    printf("*pxの値＝%d\n", *px);   /* pxが指すオブジェクトの値 */
    printf("*pyの値＝%d\n", *py);   /* pyが指すオブジェクトの値 */

    return (0);
}
```

実行結果

```
 nxの値＝100
 nyの値＝200
*pxの値＝100
*pyの値＝200
```

— ポインタ *px* の宣言は、

```
int  *px = &nx;    /* px は nx を指すポインタ */
```

となっていますが、*px を &nx で初期化するのですか？

いいえ。これは、**int** 型の *px ではなく、**int** * 型の *px* の宣言です。したがって、ここで宣言されている *px* が **&***nx* で初期化されます。

重　要

ポインタ *p* が、***Type*** 型のオブジェクト *x* を指すように初期化するには、

```
Type *p = &x;
```

と宣言する。

参考までに、このプログラムを C++ で書きかえたものを **List 1-10** に示します。

List 1-10

```
/*
    ポインタの初期化（C++）
*/

#include  <iostream>

using namespace  std;

int main(void)
{
    int  nx = 100;      // nxの値は100
    int  ny = 200;      // nyの値は200
    int* px = &nx;      // pxはnxを指すポインタ
    int* py = &ny;      // pyはnyを指すポインタ

    cout << " nxの値＝" << nx << '\n';      // nxの値
    cout << " nyの値＝" << ny << '\n';      // nyの値
    cout << "*pxの値＝" << *px << '\n';     // pxが指すオブジェクトの値
    cout << "*pyの値＝" << *py << '\n';     // pyが指すオブジェクトの値

    return (0);
}
```

実行結果

```
nxの値＝100
nyの値＝200
*pxの値＝100
*pyの値＝200
```

このように、C++では、“**int***”といった具合に、型名と * の間に空白を入れずに、くっつけて書かれることが多いようです。ただし、意味は変わりません。

■ 演習 1-3

右のように宣言が行われており、*p1* は *x* を指し、*p2* は *y* を指している。

p1 が *y* を指し、*p2* が *x* を指すように変えるプログラム（部分）を示せ。

```
int  x, y;
int  *p1 = &x;
int  *p2 = &y;
```

■ 演習 1-4

右に示すプログラム部分の出力を示せ。

```
int  x = 50;
int  *p = &x;
printf("%d\n", 5**p);
```

register 記憶域クラス指定子とアドレス

それでは、**List 1-11** のプログラムを実行してみてください。

List 1-11

```
/*
    register記憶域クラス指定子付きで宣言されたオブジェクトのアドレス
*/

#include  <stdio.h>

int main(void)
{
    register int  nx;

    printf("&nxの値は%pです。\n", &nx);          /* エラー */

    return (0);
}
```

— あれっ？　エラーが発生して、コンパイルが中断されますよ。

はい。*nx* のように、**記憶域クラス指定子**（*storage class specifier*）である **register** を伴って定義されたオブジェクトにアドレス演算子 **&** を適用することはできません。

というのも、そのようなオブジェクトは、主記憶上ではなく、**レジスタ**（*register*）と呼ばれる CPU 内部の特殊な場所に格納される可能性があるからです（**Fig.1-18**）。

重　要

register記憶域クラス指定子を伴って定義されたオブジェクトにアドレス演算子を適用することはできない。

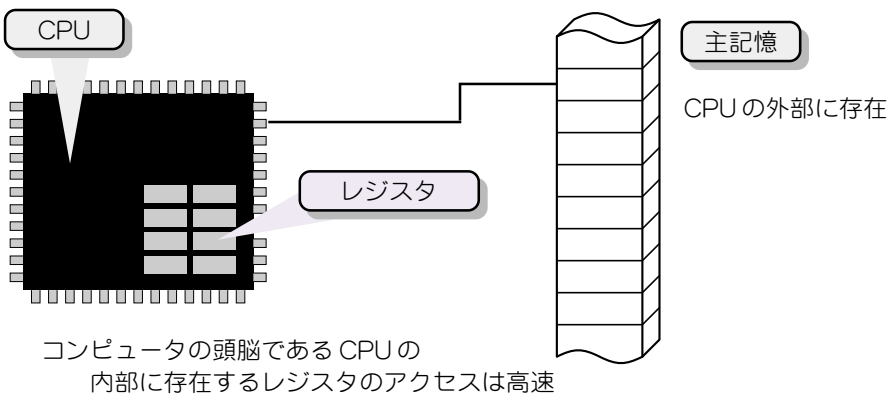


Fig.1-18　レジスタと主記憶

— ところで、**register** を指定することには、どのようなメリットがあるのですか。

右のように **for** 文や **while** 文で繰返しを制御するための変数や、頻繁に値を読み書きする変数がレジスタに格納されていれば、処理の高速化が期待できます。

```
register int  i, j;

for (i = 1; i < 1000; i++)
    for (j = 1; j < 3000; j++)
        /* 処理 */
```

ただし、レジスタは無限にはあるわけではないので、実際には数個程度の変数のみがレジスタに格納されることになります。

— なるほど。

どの変数をレジスタに格納すればプログラムが高速になるかの決定を、プログラム作成者に委ねようという目的で発明されたのが **register** です。

しかし、コンパイルの技術が進歩した現在では、プログラマによる**register**の指定が、決して高速化のためのヒントになり得ないこともあります。すなわち、プログラムを高速化するためには、どの変数をレジスタに格納したらいいかを、コンパイラ自身が判断できるようになってきています。

そのような背景もあって、C++ では、たとえ **register** 付きで定義されたオブジェクトであっても、そのアドレスを取得できるように言語仕様が変更されています。**List 1-12** のプログラムで確認しましょう。

List 1-12

```
/*
    register記憶域クラス指定子付きで宣言されたオブジェクトのアドレス（C++）
*/

#include  <iostream>

using namespace  std;

int main(void)
{
    register int  nx;

    cout << "&nxの値は" << &nx << "です。\n";        /* OK！ */

    return (0);
}
```

実行結果一例

```
&nxの値は1000です。
```

— 本当だ。ちゃんとコンパイルできますし、実行もできます。

▶ 標準Cでは、ハードウェアと結びつくようなことがらは規定されていませんので、**register** 宣言が、レジスタへの格納を示唆するものであるという定義はありません。

なお、コンパイル技術が進歩したとはいえ、プログラマによる **register** 宣言が非常に重要な意味をもつ処理系が存在することを忘れてはいけません。

ポインタを整数値に変換

List 1-13は、ポインタを整数値に変換して表示するプログラムです。

1

List 1-13

```
/*
    ポインタを整数値に変換して表示
*/

#include  <stdio.h>

int main(void)
{
    int  nx;            /* nxはint型 */
    int  *pt = &nx;     /* ptはnxを指すポインタ */

    /* nxへのポインタを符号無し整数値に変換して表示 */
    printf("&nx：%lu\n", (unsigned long)&nx);
    printf(" pt：%lu\n", (unsigned long)pt);

    return (0);
}
```

実行結果一例

```
&nx：1000
 pt：1000
```

― *nx*へのポインタを**unsigned long**型にキャストした上で表示しているのですね。

はい。変換指定**%p**を使わずに、ポインタを整数値として表示する必要があるときは、次のように心がけましょう。

重　要

ポインタの値を整数値に変換して表示する際は、**unsigned long**型にキャストするのが無難である。

― どうしてですか？

ポインタを整数に型変換する際は、要求される整数が**short**／**int**／**long**のいずれであるのかということや、その変換によって得られる値は、処理系定義となっています。さらに、変換後の領域の大きさが不十分な場合の動作も定義されません。

したがって、ある処理系でポインタを整数に変換した値が**unsigned int**型で表現できるとしても、他の処理系では**unsigned long**型でなければ不十分かもしれませんし、十分でない型への変換を行った場合の動作は定義されません。すなわち、どのような結果が得られるのかは分からないのです。

重　要

ポインタから整数値への型変換において必要な型は処理系に依存する。

したがって、**unsigned int** 型にキャストして表示する

```
printf(" pt：%u\n", (unsigned int)pt);
```

は、ポインタを変換した結果を **unsigned int** 型で表現できる処理系ではよいでしょうが、そうでない環境では、期待する結果は得られません。

— なるほど。それで、ポインタを整数に型変換する際は、最も大きな非負の整数値を表現できる **unsigned long** 型にキャストするのが無難なのですね！

そうです。

ちなみに、ポインタを整数へと型変換することによって得られる値が、実際の物理的なアドレスと等しいという保証はありません。したがって、変換後の値が物理的なアドレス値と等しくなることが分かっている環境でない限り、変換後の整数値をもとにして、何か特別なテクニックを使ってオブジェクトの領域にアクセスするのは危険です。

Column 1-7 演算子の結合性と代入演算子

以下の代入を考えましょう（ここで *a*，*b*，*x* は **int** 型であるとします）。

```
a = b = x;
```

この代入によって、*a* と *b* の値は、*x* と同じ値となりますが、このことを理解するためには、演算子の**結合性**（*associativity*）について知っておかなければなりません。

同一の優先順位をもつ２項の演算子を○と表した場合、式

```
a ○ b ○ c
```

が、

```
(a ○ b) ○ c                    /* 左結合性 */
```

とみなされる演算子は、左結合性をもち、

```
a ○ (b ○ c)                    /* 右結合性 */
```

とみなされる演算子は、右結合性をもちます。

たとえば、減算を行う２項 **-** 演算子は、左結合性をもちますので、式 `5 - 3 - 1` は `(5 - 3) - 1` とみなされます。

一方、単純代入演算子 **=** は、右結合性をもちますので、式 `a = b = x` は、`a = (b = x)` とみなされます。すなわち、最初に `b = x` の代入が行われ、その代入式を評価した値（代入式を評価すると、代入後の左オペランドの型と値が得られます）が *a* に代入されます。

したがって、C 言語のプログラムでは、

```
s = t = 0;
```

といった簡潔な代入が可能であり、好まれて使われるのです。

ただし、ちょっとした注意が必要です。*a* が **double** 型で *b* が **int** 型であり、

```
a = b = 1.5;
```

との代入を行うとします。代入式 `b = 1.5` を評価した値は、代入時に小数点以下の部分が切り捨てられるため、**int** 型の 1 です。その値が *a* に代入されますので、*a* の値は `1.5` ではなく `1.0` となります。決して、“*a* と *b* の両方に `1.5` を代入せよ。”という命令ではないことに注意しましょう。

1-2　関数呼出しとポインタ

値渡し

―　ここまでのプログラム例では、多少わざとらしくポインタが利用されてきました。実際にはどのようなときに使うのでしょうか。

そのあたりをきちんと理解するために、まずは、二つの**int**型オブジェクトの値を交換する関数を作ってみてください。

―　簡単、簡単。はい、できました。**List 1-14** です。

List 1-14

```
/*
    二つの整数値を交換（間違い）
*/

#include  <stdio.h>

/*--- xとyの値を交換（間違い） ---*/
void swap(int x, int y)
{
    int  temp = x;
    x = y;
    y = temp;
}

int main(void)
{
    int  a, b;

    puts("二つの整数を入力してください。");
    printf("整数Ａ：");    scanf("%d", &a);
    printf("整数Ｂ：");    scanf("%d", &b);

    swap(a, b);

    puts("整数ＡとＢの値を交換しました。");
    printf("Ａの値は%dです。\n", a);
    printf("Ｂの値は%dです。\n", b);

    return (0);
}
```

実行例

```
二つの整数を入力してください。
整数Ａ：54↵
整数Ｂ：87↵
整数ＡとＢの値を交換しました。
Ａの値は54です。
Ｂの値は87です。
```

実行すると …　あれぇ。*a* が 54、*b* が 87 のままで、値が入れかわっていません。

それは、引数の受渡しが次に示すメカニズムである**値渡し**（*pass by value*）によって行われるからです。

重　要

関数を呼び出す側は**実引数**（*argument*）として《値》を渡して、呼び出される側はその《コピー》を**仮引数**（*parameter*）に受け取る。

— コピーですか？

Fig.1-19 を見てください。

main 関数は、関数 *swap* に実引数 *a*，*b* の値である 54 と 87 を渡します。呼び出された関数 *swap* は、それらの値を仮引数 *x* と *y* に受け取ります。

関数 *swap* の中で *x* と *y* の値を交換しますが、受け取ったコピーを入れかえるのであって、*a*，*b* の値を入れかえるわけではありません。ですから **main** 関数の *a*，*b* の値は、関数 *swap* を呼び出した後も、それぞれ 54 と 87 のままなのです。

— 呼び出す側の実引数と、呼び出される側の仮引数は別ものなんですね。

そうです。ひらたくいえば、呼び出す側は上流から値を“たれ流す”だけです。

— それでは、引数の値を書きかえることはできないのですか？

はい。引数の受渡しはすべて《値渡し》ですから、そういうことになります。

値渡し　実引数の値が仮引数にコピーされる

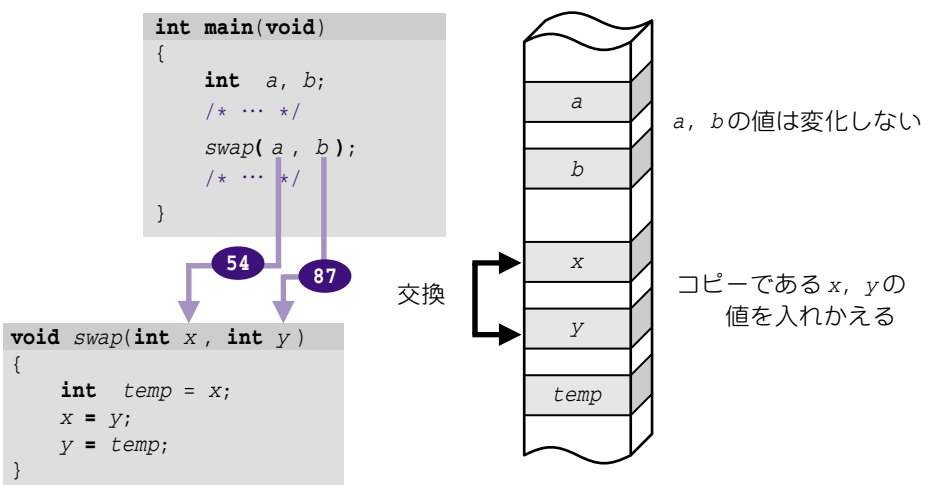


Fig.1-19　関数間の int 型引数の受渡し

ポインタを値渡し

ポインタを使用することによって、あたかも引数の値を書きかえたかのように見せかけることができます。それが **List 1-15** のプログラムです。

List 1-15

```
/*
    二つの整数値を交換
*/

#include  <stdio.h>

/*--- *xと*yの値を交換 ---*/
void swap(int *x, int *y)
{
    int  temp = *x;
    *x = *y;
    *y = temp;
}

int main(void)
{
    int  a, b;

    puts("二つの整数を入力してください。");
    printf("整数Ａ：");   scanf("%d", &a);
    printf("整数Ｂ：");   scanf("%d", &b);

    swap(&a, &b);              /* a, bへのポインタを渡す */

    puts("整数ＡとＢの値を交換しました。");
    printf("Ａの値は%dです。\n", a);
    printf("Ｂの値は%dです。\n", b);

    return (0);
}
```

実行例

```
二つの整数を入力してください。
整数Ａ：54↵
整数Ｂ：87↵
整数ＡとＢの値を交換しました。
Ａの値は87です。
Ｂの値は54です。
```

― 今度は *a* と *b* の値がきちんと入れかわってますね！

Fig.1-20 で説明しましょう。**main** 関数からの関数 *swap* の呼出しでは、*a*，*b* へのポインタである **&***a* と **&***b* を渡します。

― この図では、それらは 200 番地と 204 番地となっています。これらの値が関数 *swap* の仮引数 *x* と *y* に渡されるのですね。

はい。*x* と *y* は、**int** へのポインタ型です。これらに *a*，*b* のアドレスがコピーされますから、*x* は *a* を指し、*y* は *b* を指すことになります。

— ということは、*x は a のエイリアスとなり、*y は b のエイリアスとなりますね。

その通り。その *x と *y の値を交換するのですから、結局 a と b の値が入れかわります。

— なるほど。こうやって、引数の値を書きかえるのですね。

いえいえ、違います。呼び出す側は、ポインタの《値》であるアドレスを渡します。

呼び出された関数は、そのポインタに間接演算子を適用することによって、受け取ったアドレスに格納されているオブジェクトの値を、間接的にアクセス、すなわち値を読んだり書いたりできるのです。

実引数として渡されたのは、200 番地と 204 番地というアドレスであって、これらの引数自身の値を書きかえているのではありません。

重　要

オブジェクトへのポインタを仮引数として受け取れば、間接演算子を適用することによって、そのオブジェクトをアクセスできる。

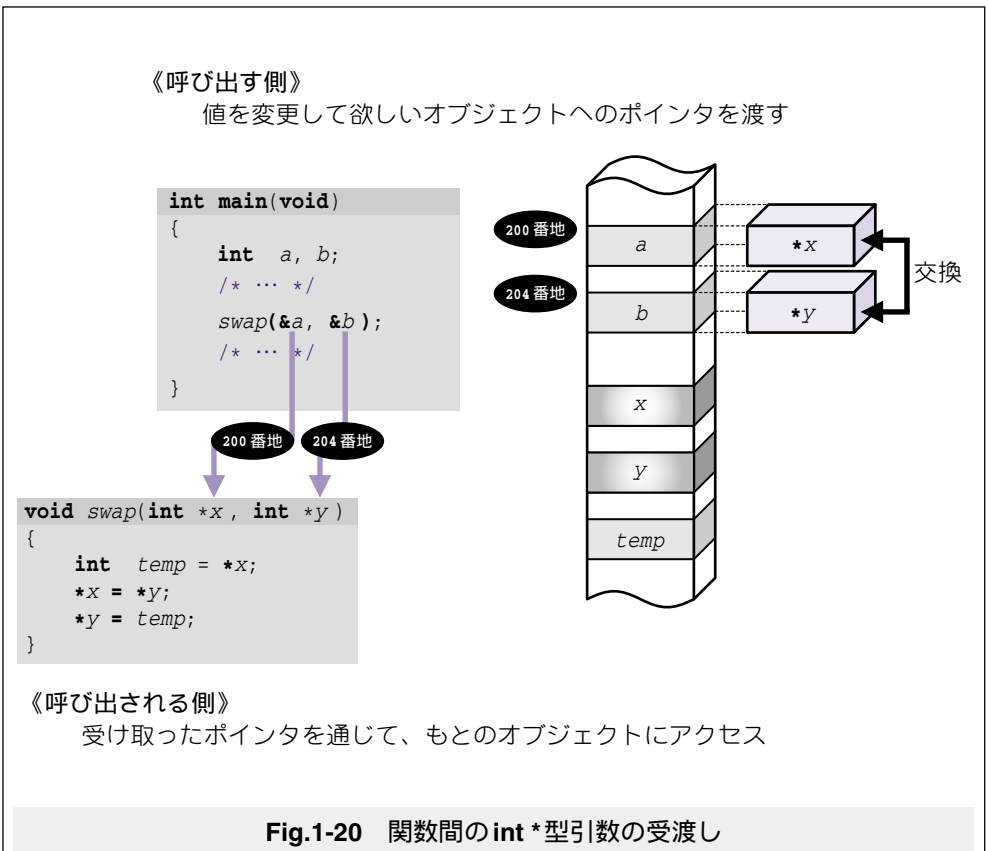


Fig.1-20　関数間の int *型引数の受渡し

参照渡し（C++）

C++ では、少し異なった実現も可能です。プログラムを **List 1-16** に示します。

List 1-16

```
/*
    二つの整数値を交換（C++）
*/

#include  <iostream>

using namespace  std;

//--- xとyの値を交換 ---//
void swap(int& x, int& y)
{
    int  temp = x;
    x = y;
    y = temp;
}

int main(void)
{
    int  a, b;

    cout << "二つの整数を入力してください。\n";
    cout << "整数Ａ：";   cin >> a;
    cout << "整数Ｂ：";   cin >> b;

    swap(a, b);

    cout << "整数ＡとＢの値を交換しました。\n";
    cout << "Ａの値は" << a << "です。\n";
    cout << "Ｂの値は" << b << "です。\n";

    return (0);
}
```

実行例

```
二つの整数を入力してください。
整数Ａ：54↵
整数Ｂ：87↵
整数ＡとＢの値を交換しました。
Ａの値は87です。
Ｂの値は54です。
```

― 仮引数の宣言には & が付いていますが、これはアドレス演算子ですか？

いいえ。これは、**参照**（*reference*）を宣言するための記号です。こうしておけば、**引数**の受渡しは**参照渡し**（*pass by reference*）によって行われます。

Fig.1-21 に示すように、仮引数 *x* と *y* はそのまま実引数 *a*，*b* のエイリアスとなります。

― なるほど。C言語よりも、こちらの方がすっきりしていて、見るからに気持ちがいい感じですね。

確かにそうです。逆に、C言語では《参照渡し》が不可能であるからこそ、オブジェクトの値の変更は、ポインタを使って間接的にやらざるを得ない、ともいえるでしょう。

重　要

C言語とは異なり、C++では参照渡しが可能である。

― 変数を渡すだけなのに、呼び出す側も呼び出される側もポインタを使って面倒な処理をしなければいけないというのも大げさな話ですね。

ただし、参照渡しには、関数呼出し式 *swap*(*a*, *b*)を見ただけでは、値を渡しているのか、参照を渡しているのかが分からないという欠点があります。

ここで、*a*, *b* があなたの預金通帳であって、それを友人である *swap* 君に渡すと考えてください。値渡しであれば、*swap* 君は通帳のコピーを受け取りますから、預金の状況などを知ることはできますが、勝手にお金をおろしたりすることはできません。

しかし、参照渡しであれば、*swap* 君は通帳そのものを受け取ることになります。

― ということは、通帳が返却されても、渡したときのままの状態であるという保証がないということですね。

はい。*swap* が友人でなく、家計をともにする配偶者であれば問題ないでしょうから、参照渡しは絶対に避けるべきものではありませんが、むやみに用いてもいけません。

参照渡し　実引数は仮引数の参照となり、同一オブジェクトを共有

```
int main(void)
{
    int  a, b;
    /* … */
    swap( a , b );
    /* … */
}
```

```
void swap(int& x , int& y )
{
    int  temp = x;
    x = y;
    y = temp;
}
```

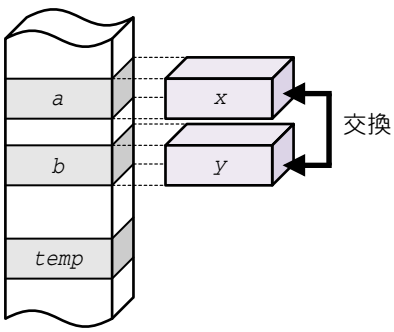


Fig.1-21　関数間の int & 型引数の受渡し（C ++）

値渡しのメリット

―　ところで《値渡し》には、どのようなメリットがあるのでしょうか。

大事なことですから、きちんと理解しておきましょう。**List 1-17** に示すプログラムを見てください。文字 `'*'` を *no* 個数だけ連続して表示する関数 *put_stars* と、それを呼び出してテストする **main** 関数とから構成されています。

List 1-17

```
/*
    文字'*'を連続して表示（第1版）
*/

#include  <stdio.h>

/*--- 文字'*'をno個連続して表示 ---*/
void put_stars(int no)
{
    int  i;

    for (i = 0; i < no; i++)
        putchar('*');
}

int main(void)
{
    int  count;

    printf("何個表示しますか：");
    scanf("%d", &count);

    put_stars(count);

    printf("\n%d個表示しました。\n", count);

    return (0);
}
```

実行例

```
何個表示しますか：15↵
***************
15個表示しました。
```

―　何のことはないプログラムですよね？

肝心なのはここからです。仮引数 *no* は、実引数 *count* の“コピー”ですから、その値を勝手に変更しても差し支えありません。したがって、**List 1-18** のようにも実現できます。

―　はぁ。短いプログラムですね。

関数 *put_stars* では、**while** 文によって繰返しを制御しています。*no* の値が 0 より大きいあいだ、その値をデクリメントしながら、文字 `'*'` の表示を繰り返します。

そのため、もはや変数 *i* は不要となっています。

List 1-18

```
/*
    文字'*'を連続して表示（第２版）
*/

#include  <stdio.h>

/*--- 文字'*'をno個連続して表示 ---*/
void put_stars(int no)
{
    while (no-- > 0)
        putchar('*');
}

int main(void)
{
    int  count;

    printf("何個表示しますか：");
    scanf("%d", &count);

    put_stars(count);

    printf("\n%d個表示しました。\n", count);

    return (0);
}
```

実行例

```
何個表示しますか：15↵
***************
15個表示しました。
```

仮引数 *no* の値は 0 になってしまう

実引数 *count* の値は変化しない

― 秒読みするときみたいに、5，4，3，2，1 とカウントダウンするんだ！

仮引数 *no* の値は、関数 *put_stars* を抜け出るときに 0 となりますが、呼び出す側の実引数である *count* とは無関係ですから構わないんですよね。

そうです。関数 *put_stars* としては、仮引数 *no* の値を、煮て食おうが焼いて食おうが、まったくの自由です。

逆に、呼び出す側である **main** 関数としては、渡した実引数の値が勝手に書きかえられる可能性がないため、安心して呼び出せるのです。

もし引数の受渡しが《参照渡し》であれば、このような恩恵は受けられません。

― 本当にその通りですね。たとえば *x* の値を表示するために、

```
printf("%d", x);
```

と関数を呼び出した後で、*x* の値が勝手に変わっていたら、とんでもないことになります。

重　要

値渡しのメリットを利用すれば、プログラムはコンパクトで効率よいものとなる可能性がある。

Column 1-8　値渡しと参照渡し

本文で説明したように、引数の受渡し方法としては、C言語では**値渡し**のみがサポートされており、C++では**値渡し**と**参照渡し**の両方がサポートされています。

これらの特徴をまとめて、比較を行いましょう。

値渡し

実引数として与えられた式を評価して得られた値が、仮引数に“代入されるかのように”コピーされます。したがって、

A　関数の独立性が損なわれにくい。

B　引数の授受の内部的なメカニズムが単純になる。

C　呼び出された側で仮引数の値を自由に変更しても構わない。

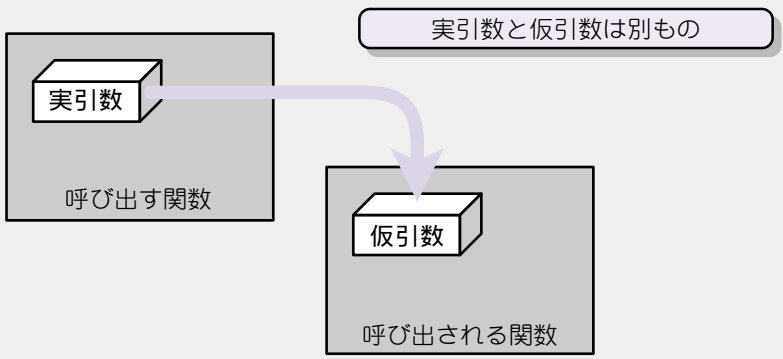
といったメリットがあります。

ただし、Cを裏返すと、

D　呼び出された側で仮引数の値を変更しても呼出し側に反映しない。

というデメリット（？）ともなります。

この方式のイメージを表したのが下の図です。仮引数は、実引数の単なるコピーに過ぎないことに注意しましょう。



参照渡し

関数呼出しの際に、実引数のアドレスが内部的に渡されて、仮引数が同じアドレスに配置されます。すなわち、実引数と仮引数は同一の記憶域を共有し、引数を通じて関数が結び付くことになります。したがって、

a　関数の独立性が損なわれやすい。

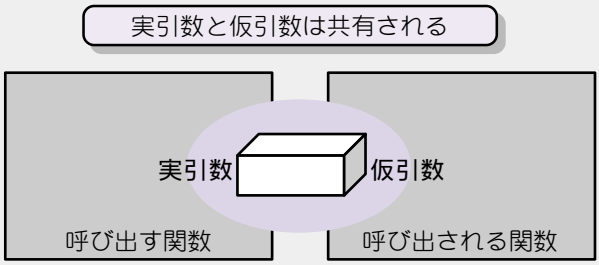
b　引数の授受の内部的なメカニズムが複雑になりやすい。

c　仮引数の値を変更すると実引数の値も変更される。

といったデメリットがあります。

この方式のイメージを表したのが次ページの図です。

二つの関数が引数を共有し、関数が強く結合するため、関数の部品としての独立性が損なわれる傾向があります。



さて、ここで、

```
void func(double& x) { /* … */ }
```

と参照渡しで**double**型の引数を受け取る関数を考えます(もちろん、C言語ではなくて、C++での話です)。このとき、

```
func(a * 5.0);
```

という呼出しでは、何が行われるのでしょうか。実引数である`a * 5.0`はオブジェクトではありませんから、その“アドレス”は存在しません。したがって、

> `a * 5.0`の値を格納するための**double**型のオブジェクトを一時的に作成して、そのアドレスを渡す。

といった取扱いが内部的に行われるのです。

さらに、《参照渡し》では、

d　再帰呼出しが不可能あるいは困難となる。

というデメリットがあります。たとえば、次の関数をよく考えれば分かると思います。

```
int factorial(int& n)
{
    if (n > 0)
        return (n * factorial(n - 1));
    else
        return (1);
}
```

このように、《値渡し》と《参照渡し》は、そのメカニズムがまったく異なるものです。

日本で出版されているC言語の書籍には、**List 1-15**に示した関数*swap*のように、実引数としてアドレスを渡し、それを仮引数がポインタとして受け取るものを《参照渡し》と解説しているものがあるようですが、それは誤りです。ポインタを値として渡しているだけであり、参照渡しとはメカニズムが違います。強いていえば、

『ポインタの値渡しによる《参照渡し》もどき』

です。

ちなみに、『プログラミング言語C』では次のように述べられています。

> Cでは、すべての関数の引数が“値で”受渡しされるからである。これは呼び出された関数には、呼出し元の変数ではなく一時変数によって引数の値が与えられたことを意味する。このため、呼び出されたルーチンが局所的なコピーではなく、元の引数にアクセスできるFortranのような“call by reference(参照による呼出し)”の言語やPascalの**var**パラメータとは、Cの性質は違ったものになっている。

ポインタと scanf 関数

ポインタの話に戻りましょう。関数から値を返す **return** 文の形式は、

```
return 式;
```

であって、返すことができるのは、*式*の値一つだけです。したがって、関数 *swap* のように二つ以上の値を戻したいときは、ポインタを使わざるを得ません。

― ポインタの実用例としては、関数の引数として使うのが第1番目に挙げられるんですね。

はい。このような使用法については必ず理解しておかねばなりません。おそらく、初心者が最初に出会うのは標準ライブラリである ***scanf*** 関数でしょう。

― そうです！ ***printf*** 関数による表示では **&** 演算子が不要なのに、***scanf*** 関数では必要ですから、とっつきにくかったことを覚えています。

printf 関数と ***scanf*** 関数の典型的な利用例を **List 1-19** に示します。

君の言うように、***printf*** 関数の呼出しでは、実引数に **&** 演算子を付ける必要はありませんが、***scanf*** 関数では必要です。

List 1-19

```
/*
    printf関数とscanf関数の利用例
*/

#include  <stdio.h>

int main(void)
{
    int   i;
    long  k;
    char  s[20];

    printf("整数を入力してください：");
    scanf("%d", &i);                    /* &が必要 */

    printf("整数を入力してください：");
    scanf("%ld", &k);                   /* &が必要 */

    printf("文字列を入力してください：");
    scanf("%s", s);                     /* 文字列の読込みでは&が不要 */

    printf("整数 i の値は%dです。\n",  i);          /* &は不要 */
    printf("整数 k の値は%ldです。\n", k);          /* &は不要 */
    printf("文字列sの値は%sです。\n",  s);          /* &は不要 */

    return (0);
}
```

実行例

```
整数を入力してください：15↵
整数を入力してください：99999↵
文字列を入力してください：Pointer↵
整数 i の値は15です。
整数 k の値は99999です。
文字列sの値はPointerです。
```

というのも、**Fig.1-22** に示すように、***scanf*** 関数に対して、

このアドレスに格納されているオブジェクトに読み込んだ値を入れてください。

と頼むからです。

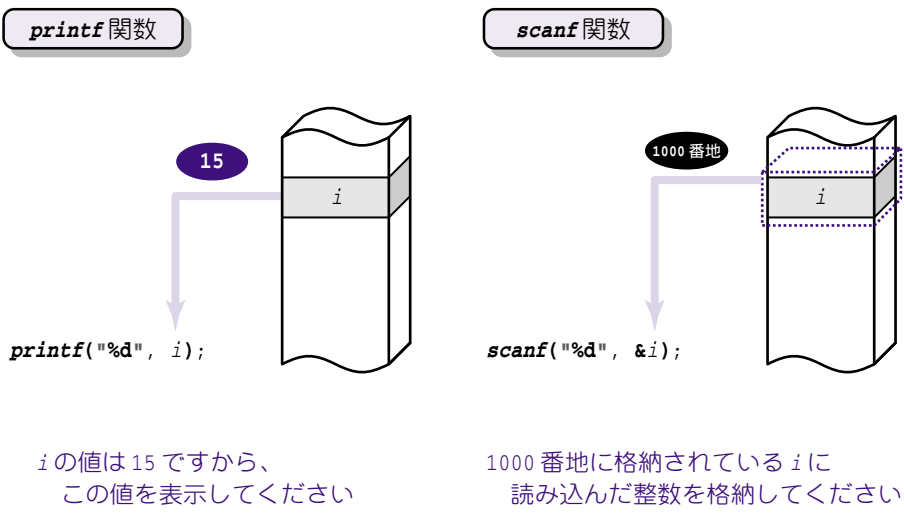


Fig.1-22　printf 関数と scanf 関数

―　C言語では、

引数の値の変更もどきを行おうと思ったら、ポインタという形で受渡しをしなければならない

から **&** 演算子を付けないといけないのですね。これで謎が解けました。

ところで、***scanf*** 関数で **%s** を指定して文字列を読み込むときには、**&** 演算子が必要でないのですが、どうしてですか。

まあまあ、あせらずに。すぐに分かるようになりますよ。

▶ 第４章で学習します（*p.116*）。

■ 演習 1-5

二つの整数 *x* と *y* の和を *wa* が指す変数に、差を *sa* が指す変数に代入する関数

```
void sum_diff(int x, int y, int *wa, int *sa) { /* … */ }
```

を作成せよ。

受け取ったポインタを別の関数に渡す

二つの整数値を交換する関数 *swap* を利用して、二つの整数値を昇順に並べましょう。そのプログラムを **List 1-20** に示します。

List 1-20

```
/*
    二つの整数値を昇順にソート
*/

#include  <stdio.h>

/*--- *xと*yの値を交換 ---*/
void swap(int *x, int *y)
{
    int  temp = *x;
    *x = *y;
    *y = temp;
}

/*--- *na≦*nbとなるようにソート ---*/
void sort2(int *na, int *nb)
{
    if (*na > *nb)
        swap(na, nb);        /* ポインタna, nbを渡す */
}

int main(void)
{
    int  a, b;

    puts("二つの整数を入力してください。");
    printf("整数Ａ：");   scanf("%d", &a);
    printf("整数Ｂ：");   scanf("%d", &b);

    sort2(&a, &b);           /* a, bへのポインタを渡す */

    puts("昇順にソートしました。");
    printf("Ａの値は%dです。\n", a);
    printf("Ｂの値は%dです。\n", b);

    return (0);
}
```

実行例

```
二つの整数を入力してください。
整数Ａ：55↵
整数Ｂ：23↵
昇順にソートしました。
Ａの値は23です。
Ｂの値は55です。
```

― 整数 *a*, *b* に値を読み込んで、*a* が *b* 以下となるように、並べかえるのですね。

はい。大小関係の基準に基づいて並べかえることを**ソートする**（*sort*）といいます。

関数 *sort2* は、*a*, *b* へのポインタを仮引数 *na* と *nb* に受け取ります。ここで、**na* の値が **nb* の値より大きければ、それらを交換するために、関数 *swap* を呼び出します。

— あれっ？　関数 *sort2* から、関数 *swap* を呼び出す式の実引数 *na*，*nb* に、アドレス演算子 **&** が付いてません。“弘法も筆の誤り”だ！

望洋に筆の誤りはありません（笑）。**int** へのポインタである関数 *sort2* の仮引数 *na*，*nb* には、*a* と *b* へのポインタがコピーされることは分かるでしょう。

— **Fig.1-23** では、それぞれ 200 番地と 204 番地ですね。

はい。*na* の値は *a* のアドレス、*nb* の値は *b* のアドレスですから、

200 番地の整数と 204 番地の整数を交換してください。

と、そのまま関数 *swap* に渡してあげればよいのです。

— なるほど！

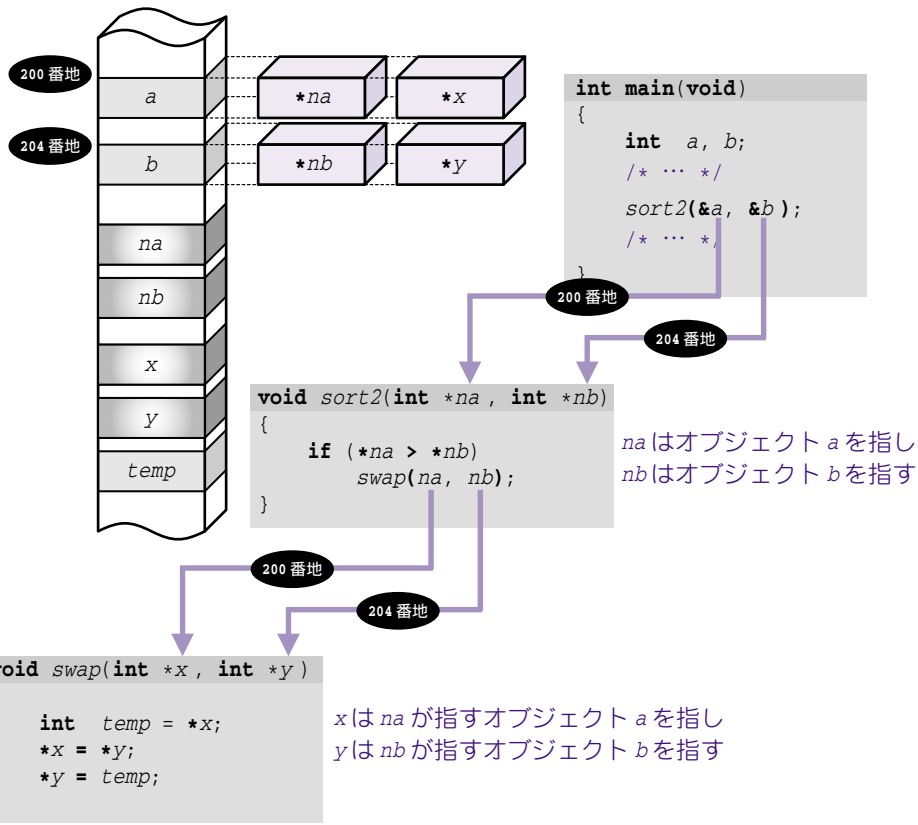


Fig.1-23　受け取ったポインタを他の関数に渡す

ポインタの指すオブジェクトに値を読み込む

少し似た例として、**List 1-21** に示すプログラムについても考えましょう。

このプログラムは、キーボードから読み込んだ実数値を、ポインタが指す変数に格納し、その値を表示します。

List 1-21

```
/*
    ポインタの指す変数に実数値を読み込んで表示
*/

#include  <stdio.h>

int main(void)
{
    double  nx;
    double  *pt = &nx;        /* ptはnxを指す */

    printf("実数値を入力してください：");
    scanf("%lf", pt);         /* 読み込んだ値をptが指す変数に格納 */

    printf("あなたは%.2fと入力しましたね。\n", *pt);

    return (0);
}
```

実行例

```
実数値を入力してください：72.5↵
あなたは72.50と入力しましたね。
```

— ***scanf*** 関数呼出しの実引数 *pt* には、アドレス演算子を付けなくていいのですか？

はい。***scanf*** 関数には、読み込んだ値を格納してもらうオブジェクトへのポインタを渡すのでしたね。このプログラムでは、ポインタ *pt* は、*nx* を指しており、そのアドレスを値としてもっています。したがって…

— ポインタ *pt* の値は **&***nx* ですから、*pt* をそのまま渡せばいいんですね。そうすると、読み込んだ値は、**Fig.1-24** の ***pt* の部分というか *nx* の領域に格納されます。

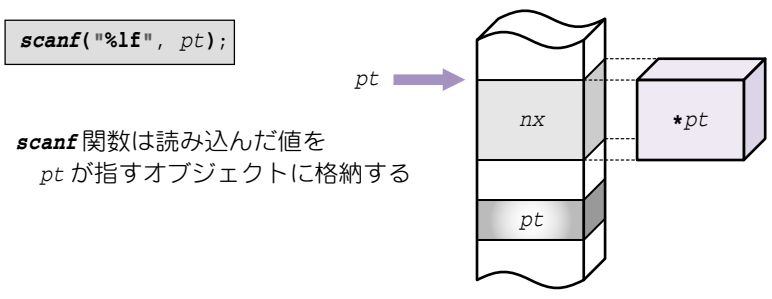


Fig.1-24　scanf関数とポインタ

そうです。*pt は nx のエイリアスですから、うまくいくのです。
それでは、もし、プログラムが

```
scanf("%lf", &pt);
```

となっていたら、どうなるでしょう。

— 読み込んだ値を、ポインタである pt に格納することになってしまいます。

そうです。読み込んだ値は、**Fig.1-25**の青い点線で囲んだ部分に格納されることになります。もし、ポインタの大きさが４バイトで、**double** 型の大きさが８バイトだったらどうなるでしょう。

— ポインタ pt が格納されている領域を超えた、黒い部分の４バイトにまで値が書き込まれてしまいます。

はい。したがって、このような誤りを犯さないように気を付けなければなりません。

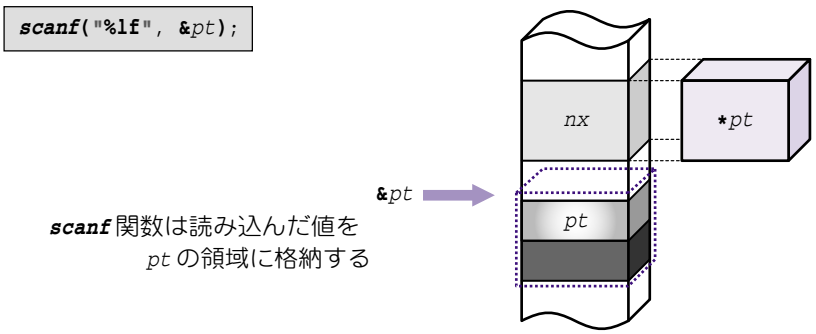


Fig.1-25　scanf 関数とポインタ（不正）

Column 1-9　main 関数の形式

次のように、**main** 関数の返却値型を **void** と宣言するプログラムを見受けます。

```
void main(void)
```

しかし、標準Ｃでは、OS 上で実行される**ホスト環境**での **main** 関数の返却値型は **int** と規定されています。したがって、

```
int main(void)
```

もしくは

```
int main(int argc, char *argv[])      /* この形式は第5章で学習します */
```

でなければなりません（ただし、引数の名前は argc や argv でなくても構いません）。本書に示すプログラムは、すべてこのスタイルで記述しています。

ポインタの型

― ポインタは、他のオブジェクトのアドレスを格納するのですよね。ということは、**int** へのポインタとか、**double** へのポインタといった区別は不要に思えるのですが…。

その考えは誤っています。**List 1-22** のプログラムで確認しましょう。

List 1-22

```
/*
    二つの浮動小数点数値を交換（間違い）
*/

#include  <stdio.h>

/*--- *xと*yの値を交換 ---*/
void swap(int *x, int *y)
{
    int  temp = *x;
    *x = *y;
    *y = temp;
}

int main(void)
{
    double  a, b;

    puts("二つの実数値を入力してください。");
    printf("実数Ａ：");    scanf("%lf", &a);
    printf("実数Ｂ：");    scanf("%lf", &b);

    swap(&a, &b);              /* a, bへのポインタを渡す */

    puts("実数ＡとＢの値を交換しました。");
    printf("Ａの値は%fです。\n", a);
    printf("Ｂの値は%fです。\n", b);

    return (0);
}
```

実行結果一例

```
二つの実数値を入力してください。
実数Ａ：55.7↵
実数Ｂ：8.29↵
実数ＡとＢの値を交換しました。
Ａの値は899.533371です。
Ｂの値は0.900654です。
```

▶ ＡやＢの値として表示される値は一例であり、ここに示す値であるとは限りません。なお、**sizeof**(**int**)と **sizeof**(**double**)が等しい処理系などでは、交換作業が正しく行われる可能性があります。

― あれっ。変な値が表示されます。

仮引数である *x* には *a* へのポインタを受け取り、*y* には *b* へのポインタを受け取ります。さて、**int** 型へのポインタに間接演算子 * を適用した型は **int** となります。したがって、このように呼び出された関数 *swap* は、**Fig.1-26** に示すように、**double** 型のオブジェクトである *a* と *b* の領域を、**int** 型として解釈しようとする、おかしなプログラムとなるのです。

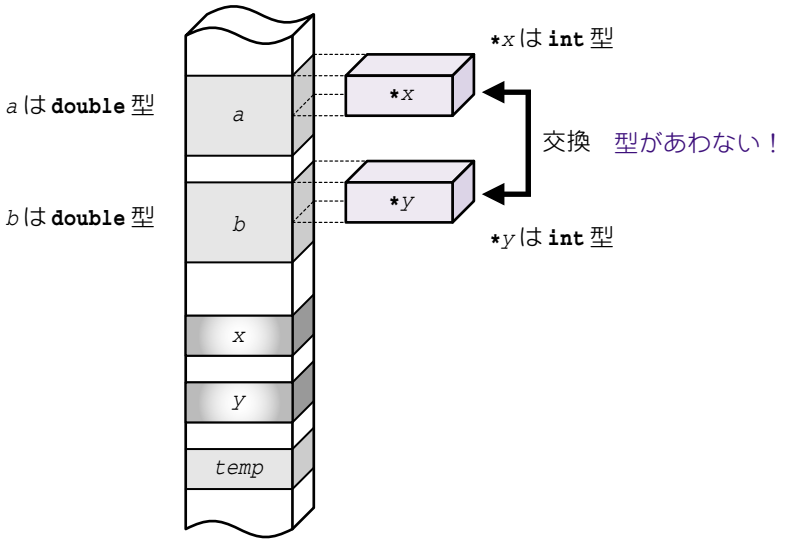


Fig.1-26　ポインタと型

— **double** 型と **int** 型の大きさは、違うのですね。

もちろん、それらの値がたまたま等しい処理系もあるでしょう。いずれにせよ、型が異なると、その記憶域上のビットの意味の解釈も異なりますので、**double**型の領域を、**int**型の値として解釈するようなことをしてはいけません。

重　要

Type型オブジェクトを指すポインタは、原則として***Type*** *型でなければならない。

▶ ポインタの型変換については、第７章で学習します。

■ 演習 1-6

pa, *pb*, *pc* が指す三つの **double** 型浮動小数点数値が ***pa* ≦ ***pb* ≦ ***pc* となるように、昇順にソートする関数

```
void sort3d(double *pa, double *pb, double *pc) { /* … */ }
```

を作成せよ。