第１章

ポインタの基本

C言語のプログラムは、

- ポインタを使うと、簡潔で効率がよくなる。
- ポインタを使わなければ、実現不可能あるいは困難なことがある。

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言語の本質は身につかないといっても過言ではありません。

本章では、ポインタの基本的な事項を学習していきます。

1–1 ポインタとは

本節の目的は、“そもそもポインタとは何であるのか”から始めて、ポインタの基礎を身につけることです。しっかりと学習していきましょう。

オブジェクトとアドレス

本節の目的は、ポインタの基礎を学習することです。まずは、ポインタと深い関係にある**変数**（*variable*）について、きちんと理解することにします。

Fig.1-1 を見てください。以下に示す6個の“箱”が描かれています。

- **int** 型
- **int** 型の変数 *m*
- **int** 型の変数 *n*
- **double** 型
- **double** 型の変数 *x*
- **double** 型の変数 *y*

もちろん、点線の箱が**型**（*type*）で、実線の箱が変数です。

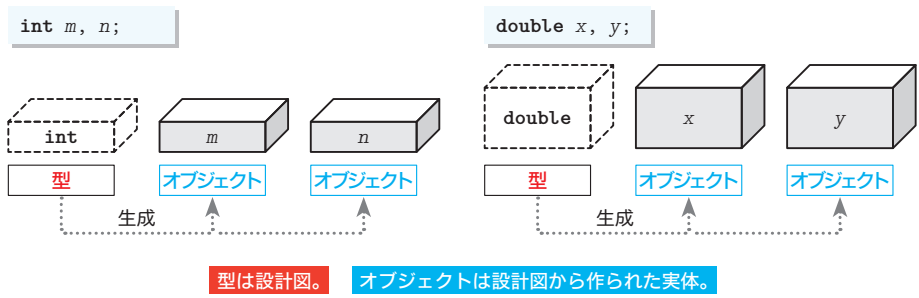


Fig.1-1 int型とdouble型の変数（オブジェクト）

点線の箱である《**型**》は**設計図**です。たとえば、**int** 型は整数だけを表せるように設計されていて、**double** 型は小数部をもつ実数も表せるように設計されています。

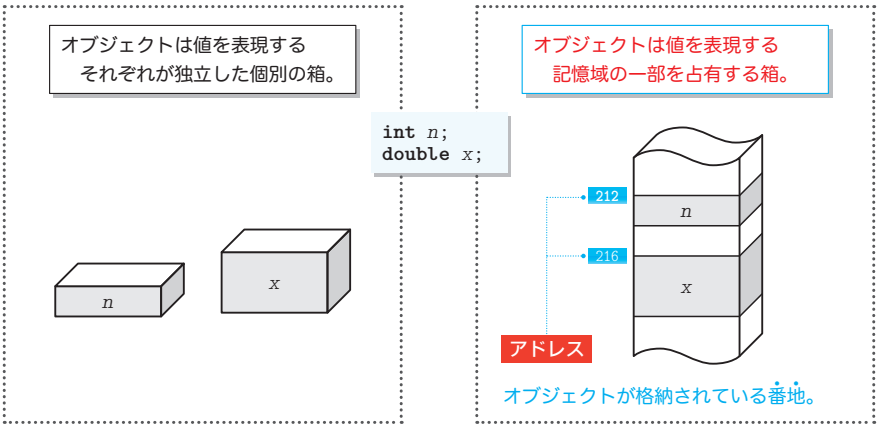
一方、実線の箱で表された《**変数**》は、設計図に基づいて作られた**実体**です。

実体である変数には、値を入れたり取り出したりできます。変数が値を保持できるのは、コンピュータの**記憶域**（メモリ）を占有しているからです。そのため、*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。

以下に示すのが、標準Cでの『オブジェクト』の定義です。

その内容によって、値を表現することができる実行環境中の記憶域の領域。

ここで、型のことを、《タコ焼き》を作るための《カタ》と考えてみましょう。そうすると、型から作られた実体であるオブジェクトは、《カタ》から作られて実際に食べられる《タコ焼き》ということになります。



a 論理的なオブジェクト　　b 物理的なオブジェクト

Fig.1-2　論理的なオブジェクトと物理的なオブジェクトのイメージ

個々のオブジェクトは、Fig.1-2 aに示すように独立した箱として存在するように感じられるかもしれませんが、実際は、図bに示すように、**記憶域の一部を占める箱です**。

多くのオブジェクトが、広大な空間の記憶域上に雑居しているため、個々のオブジェクトの“場所”を何らかの方法で表す必要があります。私たちの住まいと同様、“場所”を表すのが《**番地**》です。その番地のことを**アドレス**（*address*）と呼びます。

図bが表すのは、**int** 型オブジェクト *n* が 212 番地に格納され、**double** 型オブジェクト *x* が 216 番地に格納されている様子です。

▶　オブジェクトが格納されるアドレスは、環境によって異なります。また、たとえ同一環境であっても、プログラムを実行するたびに変わるのが一般的です。

なお、本書の図では、**低アドレス（値の小さいアドレス）が上側または左側となり、高アドレス（値の大きいアドレス）が下側または右側となるように表記します。**

Column 1-1　型について

型（*type*）は、オブジェクトの意味や、関数の返却値の意味を決定づけるものであり、**オブジェクト型**（*object type*）、**関数型**（*function type*）、**不完全型**（*incomplete type*）の３種類があります。

ここで、オブジェクト *n* が **int** 型で、*x* が **double** 型であるとします。**int** 型のオブジェクトが表せる値は整数に限られます。そのため、たとえ、

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

と浮動小数点値を代入しても、小数部は切り捨てられます。すなわち、代入後の変数 *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。オブジェクトの振る舞いが、型に基づいて決定される例です。

アドレス演算子 &

それでは、早速(さっそく)オブジェクトの《アドレス》を調べてみましょう。**List 1-1** に示すのが、オブジェクトのアドレスを取得して表示するプログラムです。

▶ 逆斜線記号 \ の代わりに円記号 ¥ を使わなければならない環境も少なくありません（日本特有の特殊な事情です）。そのような環境では、すべての \ は ¥ に読みかえるようにしましょう。

List 1-1 chap01/list0101.c

```
/* オブジェクトの値とアドレスを表示 */

#include <stdio.h>

int main(void)
{
    int n1 = 15;     /* n1はint型のオブジェクト */
    int n2 = 73;     /* n2はint型のオブジェクト */

    printf("n1の値＝%d\n", n1);          /* n1の値を表示 */
    printf("n2の値＝%d\n", n2);          /* n2の値を表示 */

    printf("n1のアドレス＝%p\n", &n1);   /* n1のアドレスを表示 */
    printf("n2のアドレス＝%p\n", &n2);   /* n2のアドレスを表示 */

    return 0;
}
```

アドレス演算子 & によってアドレスを取得 !!

実行結果一例

```
n1の値＝15
n2の値＝73
n1のアドレス＝312
n2のアドレス＝316
```

15 と 73 で初期化されたオブジェクト *n1* と *n2* のアドレスを取得して表示するのが網かけ部です。オブジェクトのアドレスを調べるために、**アドレス演算子**（*address operator*）と呼ばれる**単項 & 演算子**（*unary & operator*）を使っています。アドレス取出しの対象は**オペランド**（**Column 1-2**）ですから、**&***n1* と **&***n2* は、それぞれオブジェクト *n1* と *n2* のアドレスとなります。

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。

▶ 2種類の **&** 演算子が存在しますので、混同しないようにしましょう。
- 単項演算子である**アドレス演算子** … 例 **&***x*
- 2項演算子である**ビット論理 AND 演算子** … 例 *x* **&** *y*

アドレス値の表示のために ***printf*** 関数に与える変換指定は **%p** です（**Fig.1-3**）。表示の書式は処理系に依存しますが、多くの処理系では 4 桁〜 8 桁程度の 16 進数です。

▶ 変換指定の **p** は pointer（ポインタ）に由来して、**d** は decimal（10 進数）に由来します。
なお、本書に示す実行結果は、**%p** によるポインタの出力結果を、16 進数ではなく 10 進数としています（計算しやすくするためです）。

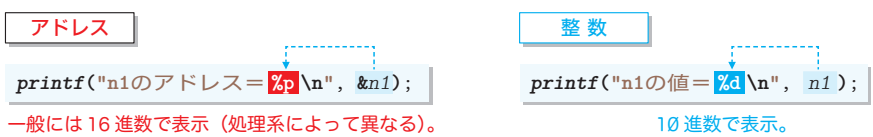


Fig.1-3 アドレスの表示と整数の表示

左ページの実行結果は、**Fig.1-4** のように、*n1* が 312 番地に格納され、*n2* が 316 番地に格納されていることを示しています。

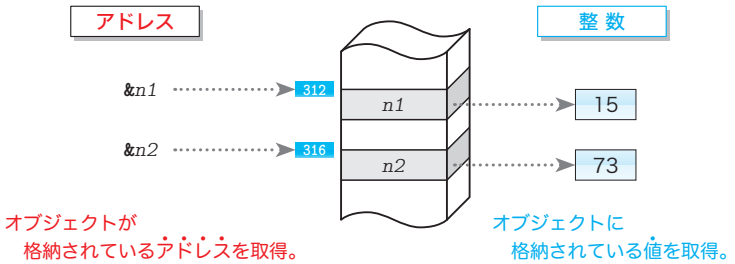


Fig.1-4　オブジェクトの値とアドレス

▶ 実行例のアドレス *312* と *316* は、一例です（この値が表示されるわけではありません）。処理系や実行環境などの条件によって変化する数値の実行結果は、*太字の斜体*で示します。

なお、アドレス演算子 **&** によって取得された値が、そのコンピュータ上での物理的なアドレスと一致する保証はありません。環境や処理系によっては、何らかの変換を行った後の値が得られることもあります。あくまでも、“プログラム上から見た”アドレスと考えましょう。

バイトとアドレス

アドレス付与の対象は、オブジェクトではなく**バイト**（*byte*）です。

1 バイトのビット数は処理系によって異なるため、<limits.h> ヘッダ中でオブジェクト形式マクロ **CHAR_BIT** として定義されます。以下に示すのが、定義の一例です。

CHAR_BIT

```
#define CHAR_BIT   8      /* 定義の一例：値は処理系によって異なる */
```

値は処理系依存ですが、**少なくとも 8 である**ことが保証されます。

▶ たとえば、**int** 型が 16 ビットであれば、オブジェクト *n1* は、312 番地から 313 番地の 2 バイトにわたって格納されます（p.12）。

Column 1-2　演算子とオペランド

演算子（*operator*）とは、演算を行うための **+**，*****，**&** などの記号のことです。演算の対象となる**式**（p.18）は**オペランド**（*operand*）と呼ばれます。たとえば、加算を行う式 *x* **+** *y* において、演算子は **+** であり、オペランドは *x* と *y* です。

以下のように、演算子によってオペランドの個数が異なります。

- **単項演算子**（*unary operator*）　オペランドが 1 個の演算子。　例 *x*++
- **2 項演算子**（*binary operator*）　オペランドが 2 個の演算子。　例 *x* + *y*
- **3 項演算子**（*ternary operator*）　オペランドが 3 個の演算子。　例 *x* ? *y* : *z*

ポインタとは

オブジェクトやアドレスについて理解できたところで、**ポインタ**（*pointer*）の話に進みましょう。まずは、宣言です。ポインタの宣言の一例を以下に示します。

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

ここで宣言されている *ptr* の型は、『**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**』と呼ばれます。

▶ いうまでもなく、単なる『**int** 型』とはまったく異なる型です。

ポインタとして宣言された *ptr* 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、すなわち《オブジェクト》です。

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 chap01/list0102.c

```
/* 整数の値とポインタの値を表示 */

#include <stdio.h>

int main(void)
{
    int n;          /* nはint型（整数）*/
    int *ptr;       /* ptrはint *型（ポインタ）*/

    n = 57;         /* nに57を代入 */
    ptr = &n;       /* ptrにnのアドレスを代入 */

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

    return 0;
}
```

実行結果一例

```
 n の値＝57
&n の値＝312
ptrの値＝312
```

本プログラムでは、**int** 型の変数 *n* と、**int** ＊型の変数 *ptr* に対して、値の代入と表示を行っています。値の代入の様子を示したのが、**Fig.1-5** です。

int 型の変数 *n* には 57 が代入されています。***printf*** 関数による出力では、その値が取り出されて 57 と表示されます。

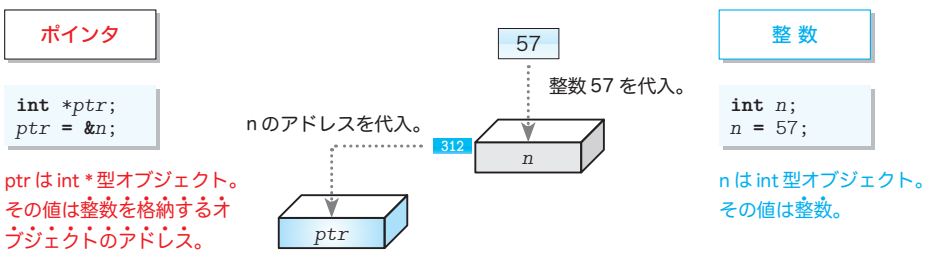


Fig.1-5 整数とポインタ

一方、ポインタである *ptr* には **&***n* が代入されています。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、*ptr* に代入されるのは、*n* のアドレスです。

ここに示す図では、ポインタ *ptr* に代入されるのは“312 番地”です。そのため、**&***n* の値と *ptr* の値は、いずれも 312 となります。

&*n* と *ptr* の型は、いずれも『**int** 型オブジェクトへのポインタ型』すなわち『**int** ＊型』です。図に示すように、**int** 型の値が単なる『**整数**』であるのとは異なり、**int** ＊型の値は『**“整数を格納するオブジェクト”のアドレス**』です。

もし変数 *n* が **double** 型で変数 *ptr* が **double** ＊型であっても、同じことが成立します。一般的にまとめると、次のようになります。

重 要 Type 型オブジェクト *x* にアドレス演算子 **&** を適用した **&***x* は、Type ＊型ポインタであり、その値は *x* のアドレスである。

▶ **int** 型オブジェクトにアドレス演算子 **&** を適用すると **int** ＊型となり、**double** 型オブジェクトにアドレス演算子 **&** を適用すると **double** ＊型となります。もちろん、**float** 型や **long** 型なども同様です。型全般に通じる規則や法則などを示す際に、“Type 型”という表現を使います（Type 型という型が存在するわけではありません）。

ポインタはオブジェクトを指す

英語の point は、『**指す**』という意味の動詞です。その point の末尾に er を付けて名詞にした単語が pointer です。《**ポインタ**》という語句は、『指すもの』、『指す人』、『指摘者』、『指針』という意味です。次のことを必ず覚えましょう。

重 要 Type ＊型ポインタ *p* の値が、Type 型オブジェクト *x* のアドレスであるとき、
『*p* は *x* を**指す**。』
と表現する。

ここで考えているプログラムでは、**int** ＊型ポインタ *ptr* の値が、**int** 型オブジェクト *n* のアドレスですから、“*ptr* は *n* を指す”ことになります。

そのイメージを表した図が **Fig.1-6** です。矢印の始点の箱がポインタであり、終点の箱が指されているオブジェクトです。

ポインタ *ptr* がオブジェクト *n* を指す。

というイメージがつかめるでしょう。

▶ 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。関数を指すポインタについては、第 8 章で学習します。

```
int n;
int *ptr;
ptr = &n;
```

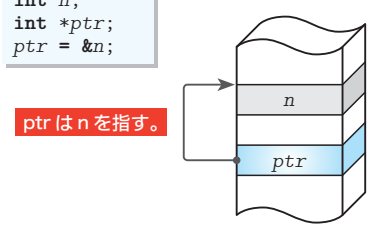


Fig.1-6 ポインタがオブジェクトを“指す”

間接演算子 *

ポインタが“指す”オブジェクトは、どのように利用するのでしょうか。**List 1-3** のプログラムで考えていきましょう。

List 1-3 chap01/list0103.c

```
/* 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 n;      /* nはint型（整数） */
    int *ptr;   /* ptrはint *型（ポインタ） */

    n = 57;     /* nに57を代入 */
    ptr = &n;   /* ptrにnのアドレスを代入 */

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

    return 0;
}
```

ptr が指すオブジェクトの値を間接演算子 * によってを取得 !!

実行結果

```
n    の値＝57
*ptrの値＝57
```

ptr に **&***n* が代入されて“ポインタ *ptr* が *n* を指す”のは、前のプログラムと同じです。

*

網かけ部では、式 ***ptr*** の値を **%d** によって 10 進の整数値として表示しています。実行すると 57 と表示されますので、この式の値が *n* と一致することが分かります。

ポインタに対して適用される演算子 * は、**間接演算子**（*indirection operator*）と呼ばれる **単項 * 演算子**（*unary * operator*）です。間接節演算子 * をポインタに適用した式は、**そのポインタが指すオブジェクトを表す式となります。**

本プログラムでは、*ptr* が *n* を指していますので、*ptr* に間接演算子 * を適用した式である ***ptr*** は、*n* そのものを表す、ということです。

一般に、*p* が *x* を指すときに“***p* が *x* を表す**”ことを、『***p* は *x* の**エイリアス（*alias*）**である。**』と表現します。エイリアスは、『別名』、『あだ名』という意味です。式 ***ptr*** は、変数 *x* に与えられた《あだ名》と考えればよいわけです。

重 要 Type 型のポインタ *p* が Type 型オブジェクト *x* を指すとき、間接演算子 * を *p* に適用した式 ***p* は *x* のエイリアス**（別名）＝あだ名となる。

なお、間接演算子 * のオペランドが指す先は、オブジェクトではなく関数でも構いません（関数を指すポインタについては、第 8 章で学習します）。

▶ アステリスク記号 * は、文脈によって異なる働きをします。混乱しないようにしましょう。

- ２項演算子である**乗算演算子** … 式 *x* * *y*
- 単項演算子である**間接演算子** … 式 ***x***
- ポインタを宣言するための**区切り子** … 宣言 **int** ***x***;

本書では、演算子は**太字**で表記して、演算子ではない区切り子は細字で表記しています。

Fig.1-7 を見ながら、理解を深めていきましょう。図**a**は、p.7 の **Fig.1-6** と同じです。この図が表すのは、『ポインタ *ptr* がオブジェクト *n* を指す』様子です。

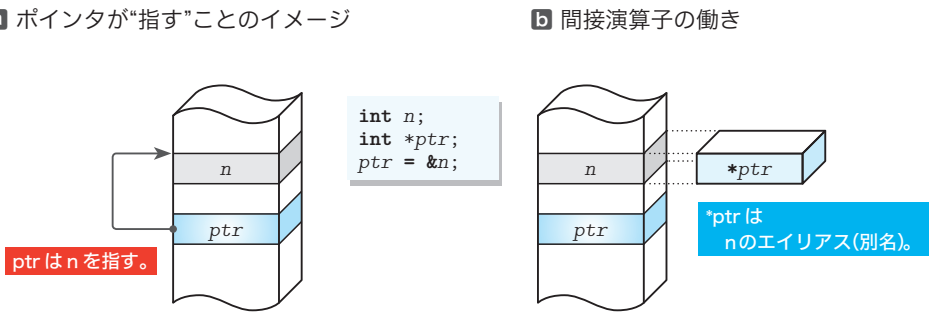


Fig.1-7　ポインタとエイリアス

一方の図**b**は、『*****ptr* がオブジェクト *n* のエイリアスである』ことを表しています。点線で結ばれた *****ptr* の箱が、*n* のエイリアスです。

▶ すなわち、図**a**が“**ポインタがオブジェクトを指す**”ことをイメージ化した図であるのに対して、図**b**は“**間接演算子の働き**”をイメージ化した図です。

プログラム**網かけ部**での *****ptr* の値の出力によって *n* の値と同じ 57 が表示される理由が、“式 *****ptr* が *n* のエイリアスだからである”、ということが分かりました。

*

単項 ***** 演算子が**間接演算子**と呼ばれるのは、ポインタを介してオブジェクトを**間接的**に扱うための演算子だからです。ポインタに対して間接演算子 ***** を適用して、ポインタが指すオブジェクト**そのもの**を間接的にアクセスする（読み書きする）ことを、“**参照外し**”と呼びます。

決して『*****ptr* という変数が存在する』わけではありませんので注意しましょう。

▶ たとえば、**int** 型の変数 *n* の値が 17 であれば、**-***n* の値は **-**17 です。変数 *n* に対して単項 **-** 演算子を適用した式 **-***n* の**評価**（p.18）によって **-**17 という値が得られるのであって、**-***n* という変数が存在するわけではありません。それと同じです。

なお、ポインタでないオブジェクト（たとえば **int** 型のオブジェクト）に対して間接演算子を適用することはできません。

重 要　ポインタでない変数には間接演算子 ***** を適用できない。

そのため、以下のコードはコンパイルエラーとなります。

```
printf("*nの値＝%d\n", *n);      /* エラー：非ポインタには*を適用できない */
```

ポインタが指すオブジェクトへの代入

前のプログラムは、ポインタが指す変数の値を取り出すものでした。今度は、ポインタが指す変数に値を書き込んでみましょう。それが、**List 1-4** に示すプログラムです。

List 1-4 chap01/list0104.c

```
/* 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 sw;
    int n1 = 15;
    int n2 = 73;
    int *p;

    printf("n1は%dでn2は%dです。\n", n1, n2);
    printf("どっちを変更(n1…1／n2…2)：");
    scanf("%d", &sw);

    if (sw == 1)
        p = &n1;      /* pにn1のアドレスを代入：pはn1を指す */
    else
        p = &n2;      /* pにn2のアドレスを代入：pはn2を指す */
    *p = 99;          /* pが指すオブジェクトに99を代入 */

    printf("nは%dでn2は%dです。\n", n1, n2);

    return 0;
}
```

代入先を実行時に動的に決定 !!

実行例 1

```
n1は15でn2は73です。
どっちを変更(n1…1／n2…2)：1
n1は99でn2は73です。
```

実行例 2

```
n1は15でn2は73です。
どっちを変更(n1…1／n2…2)：2
n1は15でn2は99です。
```

1 では、キーボードから読み込んだ値に応じて、*&n1* と *&n2* のいずれかをポインタ *p* に代入します。そして、2 では、ポインタ *p* が指すオブジェクトに 99 を代入します。

二つの実行例と **Fig.1-8** とを見比べながら、理解していきましょう。

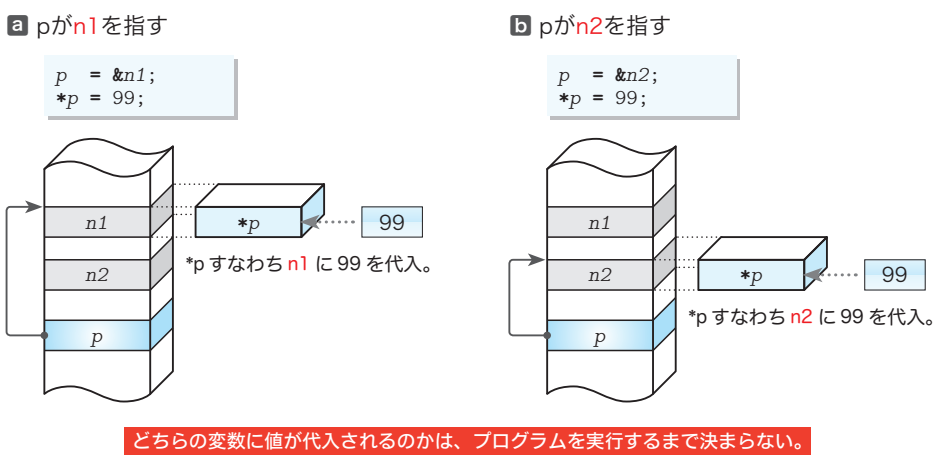


Fig.1-8 ポインタが指すオブジェクトへの値の代入

a ポインタ *p* に *&n1* が代入されており、*p* は *n1* を指しています。その状態で **p* に 99 を代入します。**p* は *n1* のエイリアスですから、99 の代入先は ***n1*** です。

b ポインタ *p* に *&n2* が代入されており、*p* は *n2* を指しています。その状態で **p* に 99 を代入します。**p* は *n2* のエイリアスですから、99 の代入先は ***n2*** です。

プログラム上で直接的には値が代入されていない変数 *n1* あるいは *n2* の値が変更されるのは、ちょっと不思議な感じです。

アクセス先（読み書き先）の決定は、プログラムのコンパイル時に静的（スタティック）に行われるのではなく、プログラムの実行時に動的（ダイナミック）に行われます。

そのため、“**p* = 99”というコードだけからは、代入先の特定は不可能です。

重要 ポインタをうまく活用すれば、アクセス先を実行時に動的に決定するプログラムが作れるようになる。

▶ “**静的な**（*static*）”は、時間が経過しても変化しないことを、“**動的な**（*dynamic*）”は、時間の経過とともに変化することを意味する語句です。

ちなみに、1の箇所は、条件演算子 **? :** を用いると、以下のように簡潔に実現できます。

```
p = (sw == 1) ? &n1 : &n2;
```

register 記憶域クラス指定子とアドレス

次に考えるのは、**List 1-5** のプログラムです。まずは、このプログラムをコンパイルしてみて、コンパイルエラーとなることを確認しましょう。

List 1-5 chap01/list0105.c

```
/* 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 n;

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

    return 0;
}
```

実行結果

コンパイルエラーとなるため実行できません。

記憶域クラス指定子（*storage class specifier*）**の一種である register を伴って定義されたオブジェクトに対してはアドレス演算子 & を適用できないのです。**

▶ どの変数を**レジスタ**（CPU 内部の高速かつ小容量の領域）に割り当てればプログラムの実行が高速になるかの決定を、プログラマ自身で行えるようにするために導入されたのが **register** です。とはいえ、コンパイル技術が進歩した現在では、どの変数をレジスタに割り当てるとプログラムが高速化するのかを、コンパイラ自身が判断できます。

そのため C++ では、**register** 付きで定義されたオブジェクトに対してもアドレス演算子 **&** を適用できるように、言語仕様が変更されています。

オブジェクトの大きさと sizeof 演算子

各型のオブジェクトは記憶域をどれだけ占有するのでしょうか。ここでは、**char** 型、**int** 型、**long** 型のオブジェクトの大きさを調べてみることにします。**List 1-6** に示すのが、そのプログラムです。

List 1-6 chap01/list0106.c

```
/* char型とint型と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2バイトです。
long型は4バイトです。
```

各型のオブジェクトの大きさは、**sizeof 演算子**（*sizeof operator*）を使用した、以下の式で取得できます。この式が生成する**値**の単位は、ビットではなく**バイト**です。

```
sizeof（型名）
```

sizeof(**char**) は、すべての処理系で 1 ですが、**それ以外の型の大きさは、処理系に依存します。**

実行例に示す環境では、**Fig.1-9** に示すように、**int** 型が 2 バイトで **long** 型が 4 バイトです。

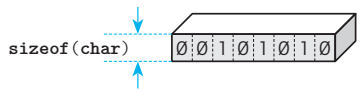
＊

sizeof 演算子が生成する値の**型**は <stddef.h> ヘッダで定義されている **size_t 型**です。

これは、**unsigned short** 型、**unsigned int** 型、**unsigned long** 型のいずれかの**符号無し整数型**と等価な型です。どの型の同義語となるのかは、処理系によって異なります。

以下に示すのが、定義の一例です。

char の大きさだけは処理系に依存しない。



char 以外の大きさは処理系に依存する。

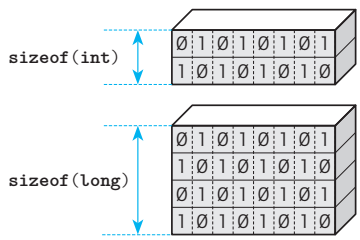


Fig.1-9 sizeof 演算子が生成する値

size_t 型

```
typedef unsigned int  size_t;   /* 定義の一例：処理系によって異なる */
```

▶ このように宣言された場合、**size_t** は **unsigned int** 型の同義語となります。同義語を定義する **typedef** 宣言については、**Column 1-5**（p.21）で学習します。

sizeof 演算子が生成した値は、以下（右ページ）のように、符号無し整数型にキャストした上で表示を行う必要があります（**Column 1-3**）。

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

本プログラムでは、unsigned 型にキャストして出力しています（1の方法です）。

重要 sizeof 演算子が生成した値の表示は、unsigned 型または unsigned long 型にキャストした上で行う。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。

▶ short や long を伴わない単独の unsigned は、unsinged int 型のことです。

Column 1-3 キャスト演算子／sizeof 演算子が生成する値の表示

▪ キャスト演算子

キャスト演算子（*cast operator*）と呼ばれる ( ) 演算子は、次の形式の式で利用します。

```
( 型 ) 式
```

この式は、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。

たとえば、(int)9.6 は、double 型の浮動小数点定数 9.6 の値から、小数部を切り捨てた int 型の整数値 9 を生成します。また、(double)5 は、int 型の整数定数 5 の値から、double 型の浮動小数点値である 5.0 を生成します。

ちなみに、英語の cast は非常に多くの意味をもつ語句です。他動詞の cast には、『役を割り当てる』『投げかける』『ひっくりかえす』『計算する』『曲げる』『ねじる』などの意味があります。

▪ sizeof 演算子が生成する値の表示におけるキャスト

sizeof 演算子が生成する値を *printf* 関数で表示する際に、符号無し整数型へのキャストが必要となる理由を考えましょう。

ここでは、int 型と long 型の大きさが２バイトと４バイトであり、さらに <stddef.h> ヘッダで

```
typedef unsigned long  size_t;  /* size_tはunsigned long型の同義語 */
```

と宣言されている環境を例にとります。このとき、unsigned long 型の同義語となる size_t 型の大きさは４バイトです。

キャストを行うことなく *printf* 関数による表示を行ったらどうなるでしょう。

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

書式指定 %u を第１引数に受け取った *printf* 関数は、２バイトの unsigned int 型の値を第２引数に受け取ることを期待します。ところが、実際に渡されるのは、４バイトの unsigned long 型の値です。これでは、正しい表示は行えません。

もっとも、size_t 型が unsigned int 型の同義語である処理系であれば、上に示した関数呼出しであっても、うまくいきます。

キャストをしないプログラムは、処理系や環境によって正しく動作したり、動作しなかったりする、可搬性（かはんせい）（他の環境への移植のしやすさ）の低いものとなることが分かりました。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ためには、キャストが必要です。

ポインタの大きさ

オブジェクトの大きさが分かりました。それでは、ポインタは何バイトなのでしょうか。ポインタの大きさは処理系によって異なるものの、**sizeof** 演算子を使って調べられます。

List 1-7 のプログラムで確認しましょう。

List 1-7 chap01/list0107.c

```
/* int型とint *型の大きさを表示 */

#include <stdio.h>

int main(void)
{
    int n;      /* int型 */
    int *p;     /* 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(" nは%uバイトです。\n", (unsigned)sizeof(n));
    printf("*pは%uバイトです。\n", (unsigned)sizeof(*p));
    printf(" pは%uバイトです。\n", (unsigned)sizeof(p));
    printf("&nは%uバイトです。\n", (unsigned)sizeof(&n));

    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。
 nは2バイトです。
*pは2バイトです。
 pは4バイトです。
&nは4バイトです。
```

int * 型の大きさを表示。

p.12 では、**sizeof**(型名) を学習しました。**sizeof** 演算子には、以下に示す、もう一つの形式があります。

sizeof 式

この式を評価して得られるのは、オペランドの式を表すのに何バイトが必要であるか、という値です。

この形式では、オペランドの式を囲む ( ) が不要であるため、“**sizeof**(*n*)” ではなく “**sizeof** *n*” とできます（**Fig.1-10**）。もっとも、文脈によっては読みにくく紛らわしくなることがありますので、本書では、式を ( ) で囲むことにします。

▶ どちらの形式であっても、**sizeof** と (とのあいだには空白を入れることができます。

sizeof(Type) と **sizeof**(Type *) が一致するとは限りません。実行結果に示す環境では、**int** 型が 2 バイトで、**int** * 型は 4 バイトです。

sizeof (型名)　型名を囲む () は必須。　例：**sizeof**(**int**)

sizeof 式　式は () で囲まなくてよい。　例：**sizeof** *n*

Fig.1-10　sizeof 演算子の二つの形式

バイト順序

オブジェクトの内部表現に関して、一つやっかいなことがあります。それは、**バイトを並べる順序が処理系に依存する**ことです。

その具体例を示したのが、**Fig.1-11** です。この図は、**int** 型が 2 バイト 16 ビットであるとしています。図**a**のように、下位バイトが先頭側（低アドレス）に配置される処理系もあれば、図**b**のように、下位バイトが末尾側（高アドレス）に配置される処理系もあります。

下位バイトが低アドレスをもつ方式は**リトルエンディアン**と呼ばれ、逆に高アドレスをもつ方式は**ビッグエンディアン**と呼ばれます。

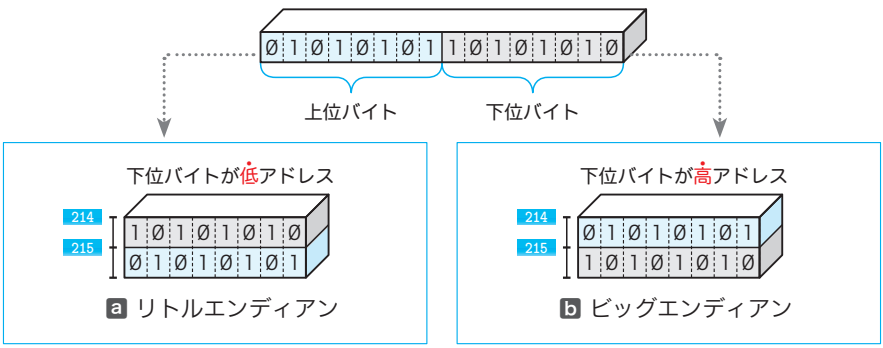


Fig.1-11　バイト順序とエンディアン

▶　エンディアンという用語は、Jonathan Swift の 1726 年の小説『ガリバー旅行記』で、小人国では“卵は太いほうから割るべきだ。”とするビックエンディアンと“卵は細いほうから割るべきだ。”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導入されました。

ポインタと関連して、少なくとも以下のことを覚えておかなければなりません。

重要　複数バイトにまたがるオブジェクトへのポインタの値は、オブジェクトの先頭アドレスである。

▶　大きさが *n* バイトのオブジェクトは、その先頭が *x* 番地であれば、*x* 番地から *x* + *n* − 1 番地にわたって格納されます（図の場合、オブジェクトは、214 番地から 215 番地の 2 バイトにわたって格納されています）。このことは、

『***x* 番地を先頭に *n* バイトにわたって格納されている。**』

と表現すべきです。しかし、これでは長くなりすぎますので、本書（の大部分の箇所）では、

『***x* 番地に格納されている。**』

と省略して表現します。

ポインタの宣言と初期化

以下に示すポインタの宣言を考えましょう。

```
int * pt, pc;        /* ptとpcをint *型のポインタとして宣言したつもり */
```

変数 pt と pc の両方を、int へのポインタ型として宣言しているように見えますが、そうではありません。この宣言は、次のように解釈されるのです。

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

そのため、pc はポインタではなくて、ただの int 型オブジェクトとなります。複数のポインタをまとめて宣言するときは、各変数の前に * が必要です。

重 要 複数のポインタの宣言時は、それぞれの変数に * を忘れないようにしよう。

以下に示すのが、正しい宣言です。

```
int *pt, *pc;        /* ptとpcはint *型のポインタ */
```

それでは、**List 1-8** のプログラムを考えましょう。

List 1-8 chap01/list0108.c

```
/* ポインタの初期化 */

#include <stdio.h>

int main(void)
{
    int n = 123;     /* nの値は123 */
    int *p = &n;     /* pはnを指すポインタ */

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

    return 0;
}
```

実行結果

```
 nの値＝123
*pの値＝123
```

これまでのプログラムでは、ポインタに対しては**代入**のみを行っていました。本プログラムでは、ポインタを**初期化**するように宣言しています。その宣言である網かけ部は、変数 p の宣言であり、その初期化子が &n です。『p が &n で初期化される』ため、ポインタ p は n を指すことになります。

重 要 Type 型へのポインタ p が、Type 型オブジェクト x を指すよう初期化するには、

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

と宣言する。

なお、網かけ部の宣言によって『*p が &n で初期化される』と勘違いしてはいけません。というのも、ここで宣言されているのは、int 型の *p ではなく、int * 型の p だからです。

一般に、ポインタ *p* を初期化していない（何らかのオブジェクトを正しく指しているとは限らない）状態で、******p* による参照外しを行うと、思いもよらぬ結果を生じることになります。**ポインタは、可能であれば宣言時に初期化すべきです。**

演習 1–1

変数 *n* が **int** 型で、変数 *p* が **int *** 型であって、*p* が *n* を指しているとする。このとき、式 ***&***n* と、式 **&***p* は何を意味するのかを説明せよ。

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うこと。

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するとともに、各式の値を説明せよ（前問と同様に、変数 *n* は **int** 型で、変数 *p* は **int *** 型であるとする）。

※ オペランドの式を囲む ( ) 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。

```
sizeof*p          sizeof(unsigned)-1        sizeof n+2
sizeof&n          sizeof(double)-1          sizeof(n+2)
sizeof-1          sizeof((double)-1)        sizeof(n+2.0)
```

演習 1–3

以下のように宣言が行われており、ポインタ *p1* は *x* を指し、ポインタ *p2* は *y* を指している。二つのポインタの値を入れかえて、*p1* が *y* を指し、*p2* が *x* を指すようにするプログラムを作成せよ。

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

演習 1–4

以下に示すプログラム部分の実行結果を示せ。

```
int x = 55;
int *p = &x;
printf("%d\n", 5**p);
```

Column 1-4　初期化子と初期値

初期化子（*initializer*）とは、オブジェクトに初期値を与えるために、オブジェクトの宣言において等号記号 = の右側に置かれる式です。たとえば、以下の宣言では、赤文字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、整数のみを表す **int** 型変数 *z* の初期値は、3.5 ではなく 3 となります。

式の評価

アドレス演算子 & と間接演算子 * に対する理解を深めるために、“式”と“評価”について学習します。

式

まずは、**式**（*expression*）を理解しましょう。厳密な定義ではないのですが、式とは、以下のものの総称です。

- **変数**
- **定数**
- **変数や定数を演算子で結合したもの**

一例として、以下の式を考えます。

```
x = n + 135
```

ここでは、`x`，`n`，135，`n + 135`，`x = n + 135` のいずれもが式です。

一般に、○○演算子とオペランドとが結合された式は、○○式と呼ばれます。たとえば、代入演算子によって `x` と `n + 135` が結びつけられた式 `x = n + 135` は、**代入式**（*assignment expression*）です。

式の評価

原則として、**すべての式には型と値があります**（特別な型である `void` 型の式だけは、例外的に値がありません）。その値は、プログラム実行時に調べられます。式の値を調べることを**評価**（*evaluation*）といいます。

評価のイメージの具体例を示したのが **Fig.1-12** です（この図は、`int` 型変数 `n` の値が 52 であると仮定しています）。

変数 `n` の値が 52 ですから、`n`，135，`n + 135` の各式を評価した値は 52，135，187 となります。もちろん、三つの値の型はいずれも `int` 型です。

本図のように、本書では、ディジタル温度計のような図で評価値を示します。左側の小さな文字が《**型**》で、右側の大きな文字が《**値**》です。

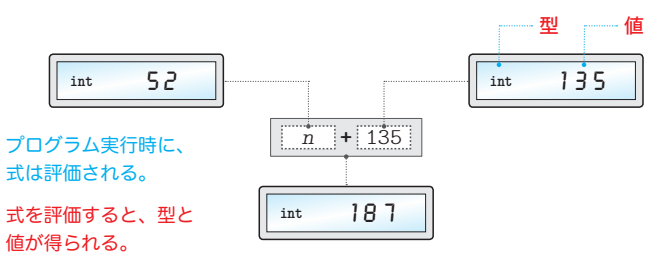


Fig.1-12　式と評価

アドレス演算子と間接演算子を適用した式の評価

アドレス演算子 **&** と間接演算子 ***** を適用した式の評価について考えていきましょう。ここでは、以下のように宣言された変数 *n* と *p* を例にとります。

```
int n = 75;          /* nはint型の整数 */
int *p = &n;         /* pはintへのポインタ型 */
```

なお、**Fig.1-13** に示すように、変数 *n* は 214 番地に格納されて、変数 *p* は 218 番地に格納されているものとします。

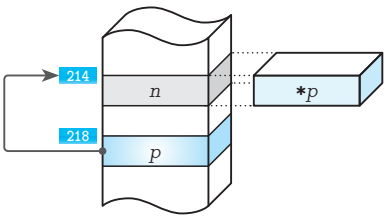


Fig.1-13　ポインタとオブジェクト

*

▪ 式 n と式 &n の評価

式 *n* と式 **&***n* の評価の様子を示したのが、**Fig.1-14** a です。

- 式 *n* を評価すると、**int** 型の 75 が得られます。
- 式 **&***n* を評価すると、**int *** 型の値が得られます。その値は、*n* が格納されているアドレスである 214 です。

▪ 式 p と式 *p の評価

式 *p* と式 *****p の評価の様子を示したのが、図 b です。

- 式 *p* の評価によって得られる型は、**int *** 型であり、その値は、指す先のオブジェクトである *n* が格納されているアドレス 214 です。
- 式 *****p を評価して得られるのは、*p* が指すオブジェクトの型と値です。すなわち **int** 型の 75 です。

▶　この図には示していませんが、式 **&***p* を評価すると、**int **** 型の 218 が得られます。**int **** 型は、“ポインタへのポインタ”です（後の章で学習します）。

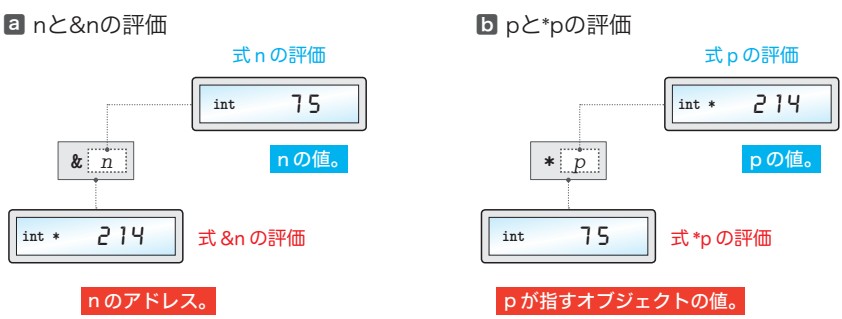


Fig.1-14　アドレス式と間接式の評価

ポインタから整数値への変換

アドレスには小数点以下の部分がありませんから、整数の一種と考えられます（少なくとも実数ではありません）。アドレス値を整数値に変換してみましょう。**List 1-9** に示すのは、ポインタを整数値に変換して表示するプログラムです。

▶ **整数型と浮動小数点型とポインタ型をあわせてスカラ型**（*scalar type*）**と呼びます。**scalar とは、『数』、あるいは、『数と同等な性質をもつ量』のことです。スカラに大きさはありますが、方向はありません（方向をもつのは vector です）。

List 1-9 chap01/list0109.c

```
/* ポインタを整数値に変換して表示 */

#include <stdio.h>

int main(void)
{
    int n;          /* nはint型 */
    int *p = &n;    /* pはint *型 */

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

    return 0;
}
```

実行結果一例

```
&n : 214
 p : 214
```

n は **int** 型の変数で、それをポインタ *p* が指しています。式 **&***n* と式 *p* は、いずれも *n* へのポインタです。本プログラムでは、これらの値を **unsigned long** 型にキャストした上で表示しています。

ポインタを整数に型変換する場合、得られる整数が **short** ／ **int** ／ **long** のいずれであるのか、あるいは、どのような値が得られるのか、といったことは、処理系に依存することになっています。さらに、変換後の領域の大きさが不十分である場合の動作は、標準 C では定義されません。

たとえば、ポインタを整数に変換した値が、ある処理系で **unsigned int** 型で表現できたとしても、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。また、十分でない大きさの型への変換を行った場合は、プログラムの動作は保証されません（環境によっては、プログラムの実行が中断するかもしれません）。

重 要 ポインタから整数値への型変換において必要な変換先の型は処理系に依存する。また、不十分な大きさの整数型への変換の結果は定義されていない。

もし、以下のようにポインタ値を **unsigned int** 型にキャストして表示したら、どうなるのかを考えてみましょう。

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

ポインタを整数に変換した結果を **unsigned int** 型で表現できる処理系では問題ないのですが、そうでない環境では、アドレス値が正しく表示される保証がありませんし、プログラムの実行が中断するかもしれません。

ポインタを整数値に型変換する際は、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です。

重要 ポインタの値を整数値に変換する場合は、原則として **unsigned long** 型にキャストすべきである。

なお、**ポインタを整数値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に一致するという保証はありません。**

▶ 変換後の値が物理的なアドレス値に一致することが保証される環境でない限り、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コンピュータの特定アドレスの領域にアクセスするのは危険です。

Column 1-5 typedef 宣言

typedef 宣言は、型に対して《同義語》を与える宣言であり、既存の型に新しい名前を与えます（新しい型を作り出す宣言ではありません）。この宣言の形式は、以下のようになっています。

```
typedef  a  b;        /* 既存の型aに同義語としてbを与える */
```

この宣言によって、既存の型 a に対して、同義語 b が与えられます。

＊

型に同義語を与えることのメリットについて、具体例で考えていきましょう。ここでは、家計を管理するプログラムでの《金額》を表す型を考えることにします。**int** 型では値の表現範囲が限られているため、金額を表すには不適切です（最も表現範囲が小さい処理系では -32767 ～ 32767 しか表せません）。

そこで、*KINGAKU* という名前の型を、以下の宣言によって導入します。

```
1 typedef long  KINGAKU;      /* 既存の型longに同義語としてKINGAKUを与える */
```

この **typedef** 宣言によって、*KINGAKU* は **long** 型の同義語となります。そのため、家計の金額を表す変数は以下のように宣言できます。

```
2 KINGAKU Aginkou, Bginkou, Okozukai;     /* KINGAKU型変数の宣言 */
```

ただの **long** 型の変数でなく、家計の金額の宣言であることが一目で分かるため、プログラムの可読性（読みやすさ）が向上します。

さて、金額として大きな数値が必要ない、あるいは、**int** 型の表現範囲が金額を表すのに十分な処理系に移植する、などの理由によって、*KINGAKU* を **int** 型の同義語に仕様変更することになったとします。そのときは、宣言1を、以下のように変更します。

```
3 typedef int  KINGAKU;       /* 既存の型intに同義語としてKINGAKUを与える */
```

ここで、宣言2の変更が不要であることに注意しましょう。**typedef** 宣言の導入は、プログラムの**拡張性**（仕様変更の行いやすさ）や**可搬性**（他の環境への移植のしやすさ）を向上させることが分かります。

1–2 関数の引数としてのポインタ

C言語の学習過程でポインタに初めて出あうのが、関数の引数として利用される文脈です。本節では、関数の引数としてのポインタを学習します。

値渡し

現実のC言語のプログラミングでポインタの利用を避けられない局面の一つが、関数の引数としてのポインタです。そのあたりをきちんと理解していきましょう。

まず最初に学習するのが、**List 1-10** のプログラムです。

List 1-10 chap01/list0110.c

```
/* 二つの整数値を交換（誤り）*/

#include <stdio.h>

/*--- xとyの値を交換（誤り）---*/
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とbの値を交換（？）*/

    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。
```

二つの **int** 型オブジェクトの値を交換するという意図で作られた関数 *swap* と、それをテストする **main** 関数とで構成されるプログラムです。**main** 関数は、**実引数**（*argument*）として a と b を渡し、関数 *swap* は**仮引数**（*parameter*）として x と y を受け取ります。

▶ 以下に示すのが、標準Cにおける実引数と仮引数の定義です。

- **実引数**　関数呼出し式において、括弧で囲み、コンマで区切った並びの中の式、または関数形式のマクロ呼出しにおいて、括弧で囲み、コンマで区切った並びの中の前処理字句の列。“*actual argument*”または“*actual parameter*”として知られている。
- **仮引数**　関数宣言もしくは関数定義の一部として宣言され、関数に入る時点で値を得るオブジェクト、または関数形式マクロ定義におけるマクロ名の直後の括弧で囲まれコンマで区切られた並びに現れる識別子。“*formal argument*”または“*formal parameter*”として知られている。

プログラムを実行してみましょう。関数 *swap* では、受け取った二つの値 *x* と *y* を交換します。ところが、関数 *swap* 呼出し後の変数 a と b の値は、キーボードから読み込んだ値のままであって入れかわっていません。

このような結果となるのは、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 関数間の引数の受渡しは《値渡し》で行われる。
関数を呼び出す側は実引数の式の《値》を渡し、その値が呼び出された関数の仮引数に代入される。そのため、仮引数は実引数の《コピー》にすぎない。

本プログラムの動作のイメージを示した **Fig.1-15** を見ながら理解しましょう。

main 関数は、関数 *swap* に対して、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。その際、*x* には 54 が代入されて、*y* には 87 が代入されます。

関数 *swap* の中では *x* と *y* の値を交換しますが、コピーの値を入れかえるだけであって、オリジナルである a と b には手をつけていません。その結果、**main** 関数の a と 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》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。

重 要 呼び出された関数側で、呼出し側の実引数の値を書きかえることはできない。

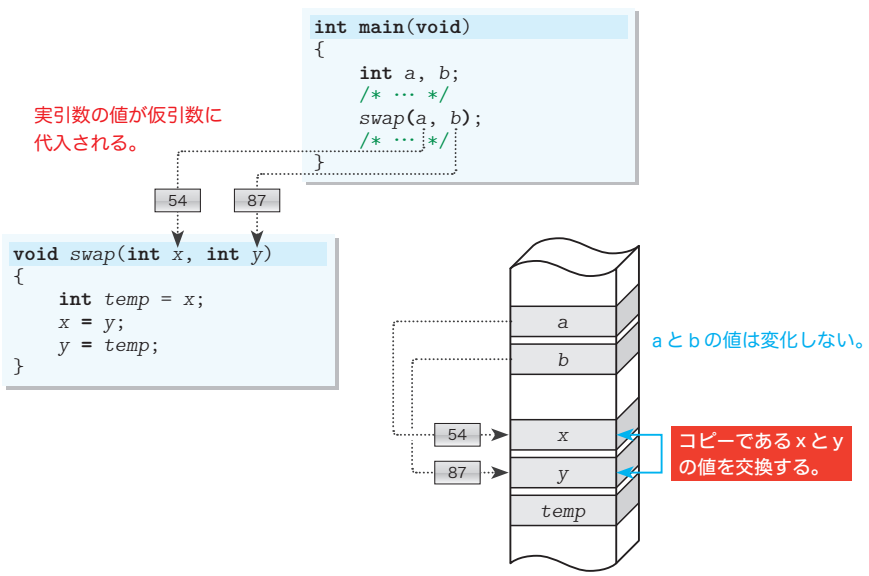


Fig.1-15　関数間の int 型引数の受渡し

ポインタの値渡し

呼び出された関数では、呼出し側の実引数の値を書きかえられないことが分かりました。

ところが、ポインタを使えば、引数の値を書きかえたかのように見せかけることが可能です。そのテクニックを用いて書き直したのが、**List 1-11** のプログラムです。

List 1-11 chap01/list0111.c

```
/* 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とbの値を交換（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。
```

まずは、プログラムを実行してみましょう。変数 a と b の値の交換は、期待どおりに行われます。

それでは、関数 *swap* を呼び出す網かけ部を、**Fig.1-16** を見ながら理解していくことにします。関数呼出しで渡している実引数 **&**a と **&**b は、変数 a へのポインタと、変数 b へのポインタです。

▶ この図では、それぞれ 212 番地と 216 番地と仮定しています。これらのアドレスが関数 *swap* の仮引数 *x* と *y* に渡されます。

関数 *swap* の仮引数 *x* と *y* の型は、“**int** 型”ではなく“**int** へのポインタ型”です。仮引数 *x* には **&**a すなわち a のアドレスが代入され、仮引数 *y* には **&**b すなわち b のアドレスが代入されます。これらの代入の結果、**ポインタ *x* は a を指して、ポインタ *y* は b を指すようになります。**

関数 *swap* の本体では二値の交換を行います。**交換の対象は、*x と *y の値です。**前者は a のエイリアスであり、後者は b のエイリアスですから、交換の結果、a と b の値が入れかわります。

仮引数に受け取ったポインタに間接演算子 * を適用することによって、実引数として渡されたアドレスに格納されているオブジェクトの値を、間接的にアクセスできることが分かるでしょう。

重要 オブジェクトへのポインタを仮引数に受け取れば、ポインタへの間接演算子 * の適用によって、そのオブジェクトそのものにアクセスでき、呼出し元が用意したオブジェクトの値を呼び出された側で変更できる。

なお、実引数として渡されたのは、a のアドレス（212 番地）と b のアドレス（216 番地）です。これらの引数自体の値を、関数 swap の中で書きかえているわけではありません。勘違いしないようにしましょう。

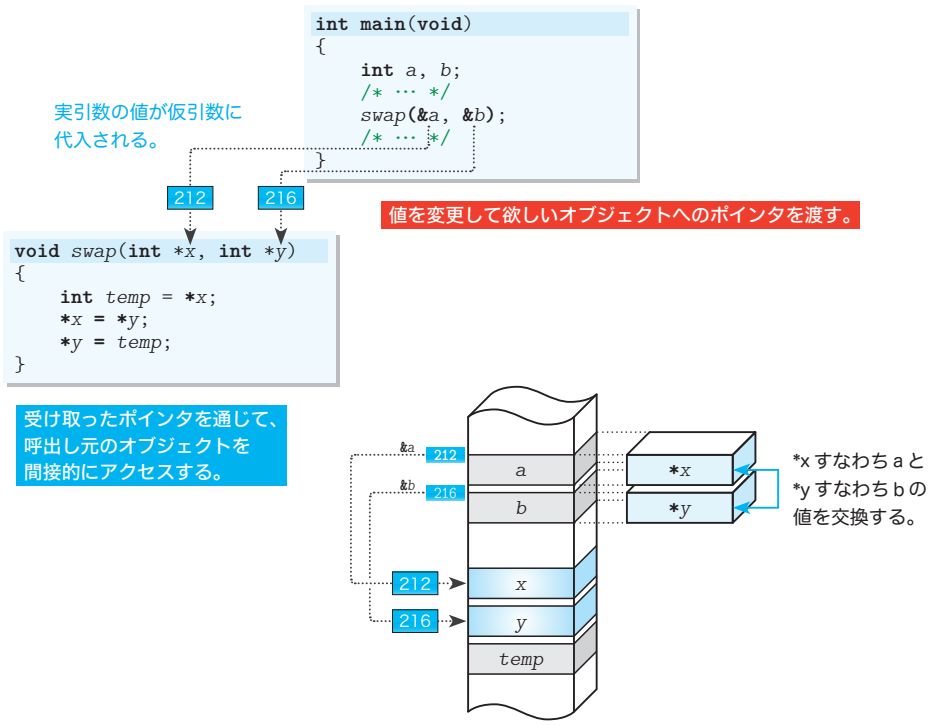


Fig.1-16 関数間の int * 型引数の受渡し

▶ **ポインタの指す先が実行時に動的に決定する**（p.11）ことを思い出しましょう。本プログラムでは、変数 a とそれを指す x、変数 b とそれを指す y がたまたま一対一に対応しているため、そのあたりのことが分かりにくくなっています。

もしも別の変数（たとえば int 型の c と d）を swap(&c, &d) として呼び出せば、ポインタ x は c を指して、y は d を指すことになります。

参照渡し（C++）

C++ では、少し異なった実現が可能です。**List 1-12** に示すのが、そのプログラムです。

List 1-12 chap01/list0112.cpp

```
// 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);      /* aとbの値を交換（a，bへの参照を渡す）*/

    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";
}
```

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。
```

関数 *swap* の仮引数 *x* と *y* の宣言に & が付いています。この & はアドレス演算子ではなく、**参照**（*reference*）を宣言するための記号です。仮引数を“参照”として宣言すると、引数の受渡しは、値渡しではなく、**参照渡し**（*pass by reference*）となります。

参照渡しでは、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。そのため、**Fig.1-17** 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。

関数 *swap* の中で、*x* と *y* の値を交換することは、**main** 関数の a と b を交換することに相当します。

本プログラムは、“ポインタの値渡し”を行う **List 1-11**（p.24）よりも、すっきりしています。逆に考えると、C 言語では《参照渡し》が不可能であるからこそ、ポインタを使った間接的な処理に頼らざるを得ない、ともいえます。

重 要 C 言語では値渡しのみがサポートされるのに対し、C++ では値渡しと参照渡しの両方がサポートされる。

ただし、参照渡しには、欠点もあります。関数呼出し式 *swap*(a，b) の見た目だけでは、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ことです。

欠点は、それだけではありません。ここで、a，bが預金通帳であって、それを友人である*swap*君に渡すと考えてみましょう。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。

ところが、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。そのため、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はありません。

*swap*が友人でなく、家計をともにする家族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。ただし、利用にあたっては、細心の注意が必要です。

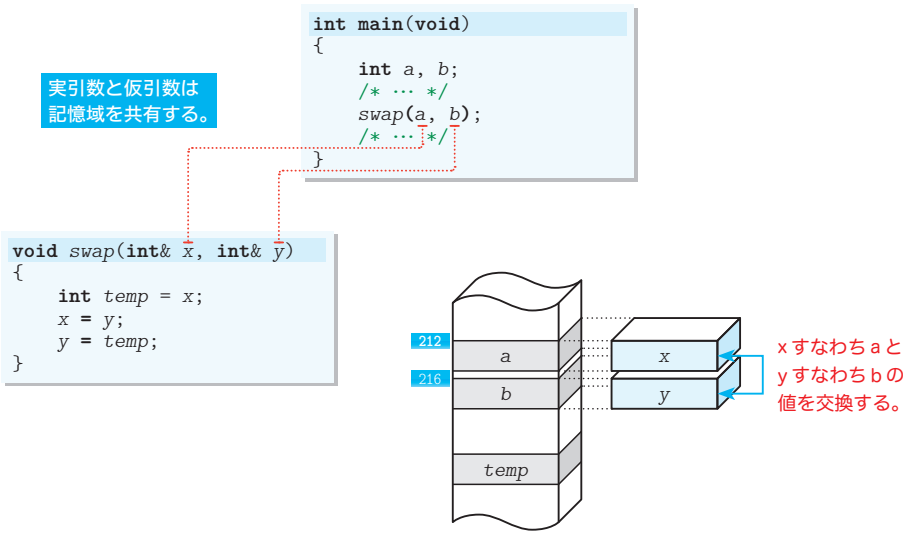


Fig.1-17　関数間のint＆型引数の受渡し（C++の参照渡し）

▶　C言語の書籍の中には、**List 1-11** に示した関数*swap*のように、引数としてポインタをやりとりする方式を《参照渡し》と解説しているものがありますが、それは完全な誤りです。ポインタの《値渡し》は、参照渡しとはメカニズムが根本的に異なります。あえて表現すると、次のようになります。

『ポインタの値渡しによる《参照渡し》もどき』

ちなみに、『プログラミング言語C』（参考文献10）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

関数から値を返すときに利用する **return** 文は、次の形式です。

```
return 式 ;
```

ここでの式は省略可能であり、**関数から返却できる値は、0 個あるいは 1 個に限られます**。そのため、関数 *swap* のように 2 個以上の値を戻したいときは、**return** 文による返却の手段は使えません。そこで、ポインタ型の仮引数を利用して、間接的なやりとりを行うことになります。

さて、ほとんどの初学者がポインタの実用例として初めて出あうのは、関数の引数としての用法、とりわけ標準ライブラリである ***scanf*** 関数の呼出しです。たとえば、**List 1-13** に示すようなプログラムです（***printf*** 関数と ***scanf*** 関数の典型的な利用例です）。

List 1-13 chap01/list0113.c

```
/* printf関数とscanf関数の利用例 */

#include <stdio.h>

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

    printf("整数nを入力せよ：");
 1  scanf("%d", &n);                  /* int型10進数の読込み：&が必要 */

    printf("整数kを入力せよ：");
 2  scanf("%ld", &k);                 /* long型10進数の読込み：&が必要 */

    printf("文字列sを入力せよ：");
 3  scanf("%s", s);                   /* 文字列の読込み：&が不要 */

    printf("整数 n の値は%dです。\n",  n);       /* &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nを入力せよ：15
整数kを入力せよ：123456
文字列sを入力せよ：ABC
整数 n の値は15です。
整数 k の値は123456です。
文字列sの値はABCです。
```

1 と 2 は整数値の読込みです。***scanf*** 関数の呼出しでは、実引数にアドレス演算子 **&** を適用しています。というのも、**Fig.1-18** b 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!!

と依頼する必要があるからです。

実引数の値を直接変更してもらうことができないからこそ、ポインタという形で受渡しを行うこと（行わざるを得ないこと）が分かりました。

▶ ***printf*** 関数による表示において、実引数の変数への **&** 演算子の適用が不要なのは、図 a に示すように、値だけを渡せばよいからです。

a printf関数の呼出しにおける引数の受渡し　b scanf関数の呼出しにおける引数の受渡し

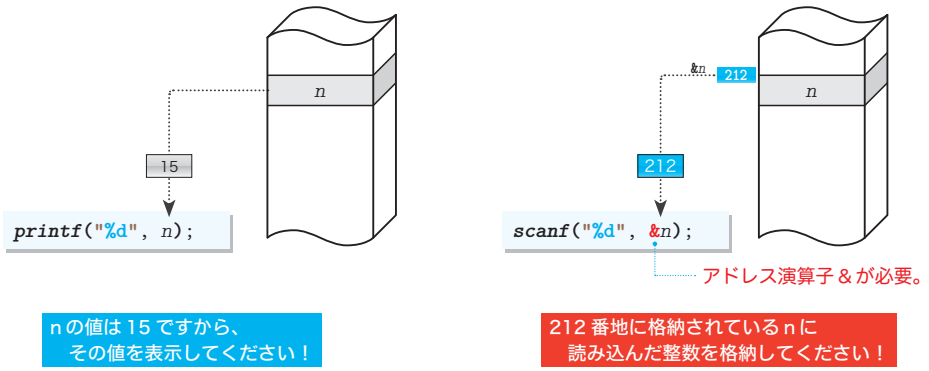


Fig.1-18　printf 関数と scanf 関数における引数の受渡し

次に、文字列を読み込む3に着目しましょう。1や2とは異なり、ここでの ***scanf*** 関数の呼出しの第２引数は、*&s* ではなく、単なる *s* となっています。

このように、文字列の読込みでは、アドレス演算子 **&** の適用は不要です。

▶　その理由は、第４章で詳しく学習します。

演習 1–5

二つの整数 *x* と *y* の和と差を求めて、その値を、*wa* が指す変数と *sa* が指す変数に代入する関数を作成せよ。

```
void sum_diff(int x, int y, int *wa, int *sa);
```

Column 1-6　C++ の main 関数の返却値

標準 C++ では、**return** 文に出あうことなく、プログラムの流れが **main** 関数の末尾に到達した場合は、以下の文が実行されたのと同じように動作します。

```
return 0;
```

そのため、C++ では、**main** 関数の末尾に **return 0;** を記述しないプログラミングスタイルが主流となっています（p.26 の **List 1-12** もそのスタイルに従っています）。

受け取ったポインタを別の関数に渡す

二値を交換する関数 *swap* をうまく応用して、二つの整数値を昇順にソート（*sort*）するプログラムを作りましょう。

▶『昇順ソート』は、小さいほうから順に並ぶように並べかえることです。なお、『降順ソート』であれば、大きいほうから順に並べかえます。

List 1-14 に示すのが、そのプログラムです。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ます。

List 1-14 chap01/list0114.c

```
/* 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;
}

/*--- *n1≦*n2となるようにソート ---*/
void sort2(int *n1, int *n2)
{
    if (*n1 > *n2)
      1 swap(n1, n2);    /* *n1の値と*n2の値を交換 */
}

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);

  2 sort2(&a, &b);        /* a≦bとなるようにソート */

    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;
}
```

アドレス演算子 & は不要 !!

アドレス演算子 & は必要 !!

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。
```

Fig.1-19 を見ながら理解していきましょう

main 関数から関数 *sort2* を呼び出す 2 では、実引数が &a と &b となっています。二つの変数 a と b の値ではなくアドレスを渡すのは、変数をソートしてもらうために、値の変更を関数 *sort2* に依頼するからです。

呼び出された関数 *sort2* は、a，b へのポインタを仮引数 *n1* と *n2* に受け取って、*n1* が指す変数の値 **n1* と、*n2* が指す変数の値 **n2* を昇順にソートします。

もし **n1* が **n2* の値以下であれば、“ソートずみ” であって、何もする必要がありません。

そうでないとき、すなわち、*n1 の値が *n2 の値より大きいときは、二つの値の交換が必要です。そこで、関数 swap を呼び出して二値の交換を行います。

そのために行う呼出しが 1 です。**関数 swap を呼び出す式の実引数 n1，n2 には、（値の変更を依頼するにもかかわらず）アドレス演算子 & が適用されていません。**

関数 sort2 の仮引数 n1，n2 には、a と b へのポインタがコピーされています。いずれも int へのポインタ型であって、n1 の値は a のアドレス、n2 の値は b のアドレスですから、関数 swap を呼び出す 1 の呼出しでは、

212 番地に入っている整数と、216 番地に入っている整数の値を交換してください!!

と、関数 swap に依頼します。受け取ったアドレスをそのまま渡せば、a のアドレスと b のアドレスが関数 swap に渡されて、うまくいきます。

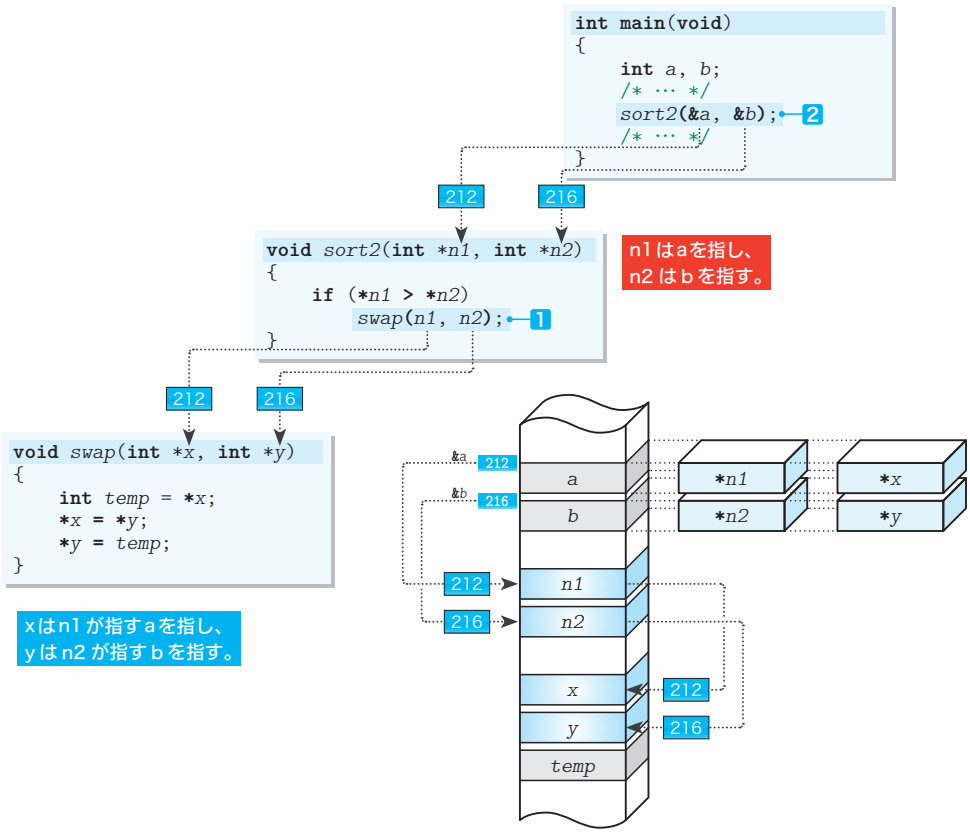


Fig.1-19　仮引数に受け取ったポインタを他の関数にそのまま渡す

▶　変数 n1 と n2 にアドレス演算子 & を適用して swap(&n1, &n2) とすると、関数 swap に渡されるのが、変数 a と b のアドレスではなく、n1 と n2 のアドレスとなってしまいます。また、その型は int * 型ではなく、int ** 型となります（後の章で学習する “ポインタへのポインタ型” です）。

ポインタの指すオブジェクトに値を読み込む

前のプログラムと少し似た例として、**List 1-15** に示すプログラムについて考えていくことにします。

List 1-15 chap01/list0115.c

```
/* 実数値のペアを昇順に読み込む */

#include <stdio.h>

/*--- *n1≦*n2となるように実数値を読み込む ---*/
void scan2double(double *x1, double *x2)
{
    printf("１番目：");
    scanf("%lf", x1);                    ①

    do {
        printf("２番目：");
        scanf("%lf", x2);                ②
    } while (*x2 < *x1);
}

int main(void)
{
    double a, b, c, d;

    puts("ＡとＢを昇順に入力せよ。");
    scan2double(&a, &b);          /* a≦bとなるように読み込む */

    puts("ＣとＤを昇順に入力せよ。");
    scan2double(&c, &d);          /* c≦dとなるように読み込む */

    printf("ＡとＢの差は%fです。\n", b - a);
    printf("ＣとＤの差は%fです。\n", d - c);

    return 0;
}
```

実行例

```
ＡとＢを昇順に入力せよ。
１番目：13.5
２番目：6.7
２番目：32.55
ＣとＤを昇順に入力せよ。
１番目：740.0
２番目：755.5
ＡとＢの差は19.050000です。
ＣとＤの差は15.500000です。
```

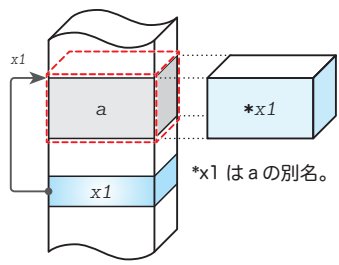
関数 *scan2double* は、*****x2 の値が *****x1 以上となるようにキーボードから読み込みます（２番目に読み込んだ値 *****x2 が *****x1 より小さければ、**do** 文の働きによって再入力を促します）。

＊

キーボードからの読込みを行う ① と ② において、***scanf*** 関数に渡す実引数 *x1* と *x2* にアドレス演算子 **&** が適用されていないことに注意しましょう。

ここで、ポインタ *x1* に **&***a* を受け取っていて、*x1* が *a* を指しているとします。***scanf*** 関数にポインタ *x1* をそのまま渡すと、読み込んだ値は、**Fig.1-20** の**赤い点線**で囲まれた部分、すなわち変数 *a* の領域に格納されます（***scanf*** 関数に渡すのは、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だからです）。

```
scanf("%lf", x1);
```



読み込んだ値は、x1 が指すオブジェクトに格納される。

Fig.1-20 正しい読込み

もし、1が以下のようになっていたらどうなるかを検討しましょう。

```
scanf("%lf", &x1);      /* 誤り */
```

読み込んだ値は、ポインタ *x1* が指す領域、すなわち Fig.1-21 の**赤い点線**で囲まれた部分に格納されることになります。

ここで、ポインタの大きさが４バイトで、**double** 型の大きさが８バイトであれば、キーボードから読み込まれた実数値は、ポインタ *x1* が格納されている領域を超えた黒い部分にまで書き込まれます。もし、ここに他の変数が格納されていれば、その変数の値が書きかえられます。いずれにせよ、**何らかのデータを破壊するのは確実です。**

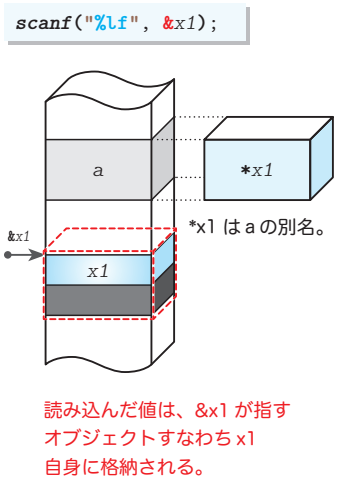


Fig.1-21　誤った読込み

それだけではありません。ポインタ *x1* の値が書きかえられるため、**ポインタとしての機能（ポインタ *x1* が変数 a を指している、という情報）も失われてしまいます。**

ポインタの型

関数 *scan2double* を、以下のように呼び出したらどうなるでしょう。

```
int n1, n2;
scan2double(&n1, &n2);      /* double *に対してint *を渡す */
```

もし **sizeof**(**int**) が２で **sizeof**(**double**) が８であるとしたら、読み込んだ実数値は、**&***n1* や **&***n2* を先頭とする８バイトの領域に書き込まれます。そうすると、先ほどと同様に、何らかのデータが破壊されます。

型が異なると、占有する記憶域の大きさや、その記憶域上のビットの意味の解釈が異なります。**ある型の領域を別の型の値として解釈するようなことは避けなければなりません。**

重 要　Type 型オブジェクトを指すポインタは、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

x1，*x2*，*x3* が指す三つの **double** 型浮動小数点値が ***x1*** ≦ ***x2*** ≦ ***x3*** となるように、昇順にソートする関数を作成せよ。

```
void sort3d(double *x1, double *x2, double *x3);
```