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、ポインタの基礎を理解することです。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、ポインタの基本を理解することです。まずは、ポインタと深い関係にある**変数**（*variable*）について、きちんと学習することにします。

Fig.1-1 を見てください。ここには、以下に示す 6 個の図が描かれています。

- **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*

点線の箱が**型**（*type*）で、実線の箱が変数です。

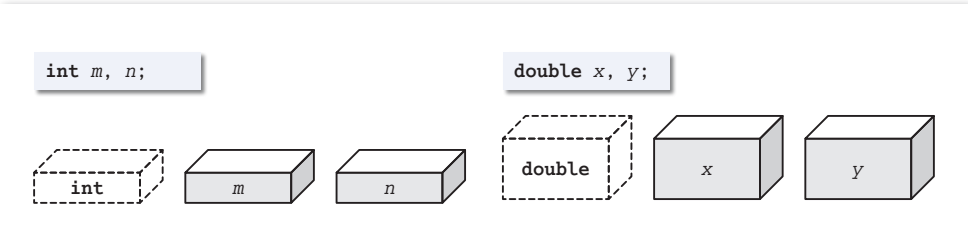

Fig.1-1 int型とdouble型の変数（オブジェクト）

型と変数の違いの大まかなイメージを表したのが、**Fig.1-2** です。

点線の箱である《型》は設計図です。たとえば、**int** 型は整数しか表せないように設計されており、**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。**Fig.1-1** の変数 *m* と *n* は **int** 型から作られた実体で、変数 *x* と *y* は **double** 型から作られた実体です。

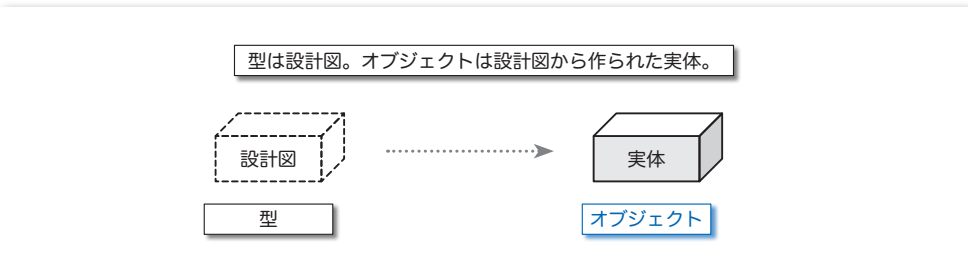

Fig.1-2 型とオブジェクト

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。

なお、標準Cでは、『オブジェクト』は以下のように定義されています。

> **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。

《タコ焼き》である個々のオブジェクトは、**Fig.1-3** a に示すように独立した箱として存在するものと感じられるかもしれませんが、そうではありません。図 b に示すように、**記憶域の一部分を占める箱なのです**。

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ことになります。私たちの住まいと同様、《場所》を表すのは“番地”です。その番地のことを**アドレス**（*address*）と呼びます。

図 b は、オブジェクト *n* が 212 番地に格納され、オブジェクト *x* が 216 番地に格納されている様子を表しています。

▶ 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。この図に示しているアドレスは、あくまでも一例です。

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。

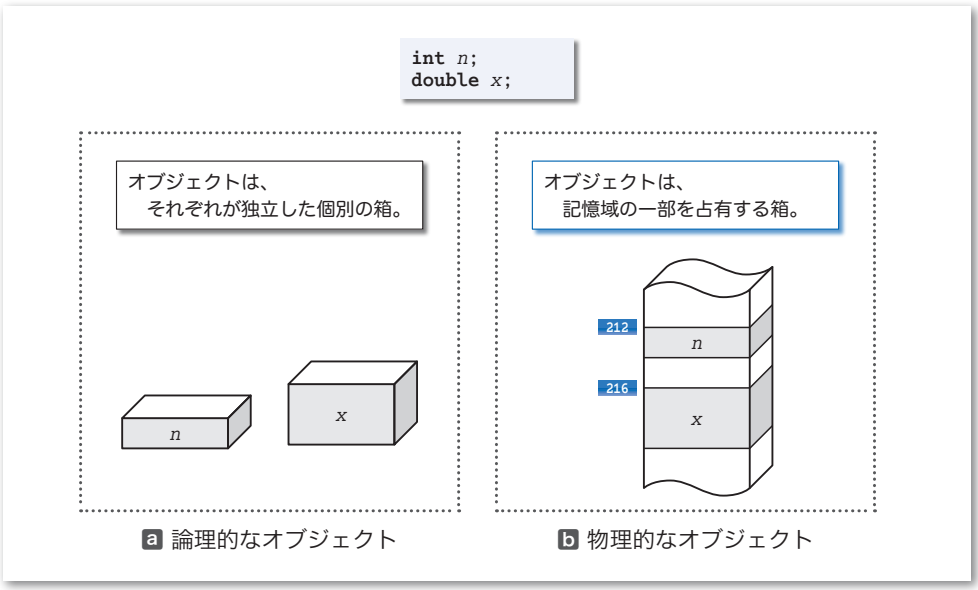

Fig.1-3 論理的なオブジェクトと物理的なオブジェクト

アドレス演算子 &

それでは、早速オブジェクトの《アドレス》を調べてみましょう。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を **List 1-1** に示します。

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;
}
```

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
```

オブジェクト *n1* と *n2* の宣言では、**初期化子**（**Column 1-1**）が与えられています。そのため、それぞれのオブジェクトは、15 と 73 で初期化されます。

オブジェクトのアドレスを表示するのが、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*）を利用しています。これは、**オペランド**（**Column 1-2**）のアドレスを取り出す演算子です。そのため、**&***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。

▶ 演算子 **&** は２種類あります。オペランドが一つの **&***x* という単項形式で利用される **&** 演算子が**アドレス演算子**で、*x* **&** *y* という２項形式で利用される **&** 演算子は**ビット論理 AND 演算子**です。

Column 1-1 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、青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** 型である *z* の初期値は、3.5 ではなく 3 となります。

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.1-4**）。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 **d** は decimal（10 進数）に由来して、**p** は pointer（ポインタ）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）。

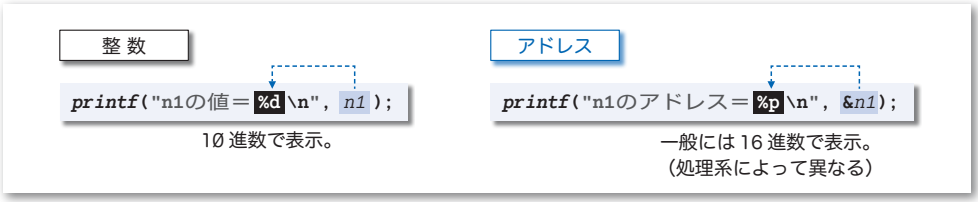

Fig.1-4 整数の表示とアドレスの表示

左ページの実行結果は、**Fig.1-5** 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。

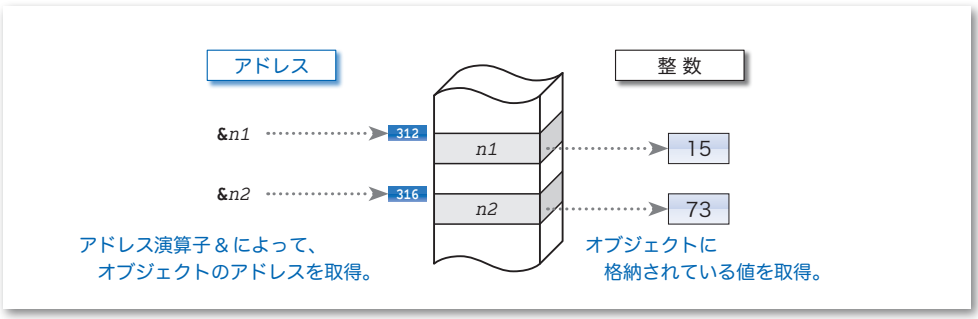

Fig.1-5 オブジェクトの値とアドレス

▶ 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という保証はありません。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。

Column 1-2 演算子とオペランド

演算子（*operator*）とは、演算を行うための記号 **+**，*****，**&** などのことです。演算の対象となる式は**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、オペランドの個数は演算子によって異なります。

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

▶ 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であって、変数すなわち《オブジェクト》です。

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

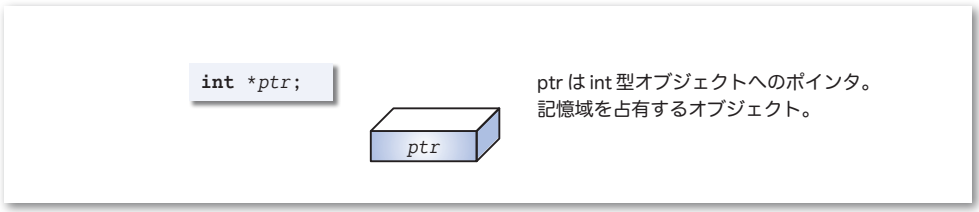

Fig.1-6 ポインタ

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 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
```

このプログラムは、**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います。値の代入の様子を示したのが、**Fig.1-7** です。

int 型の変数 *n* に 57 が代入され、***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。これは分かるでしょう。

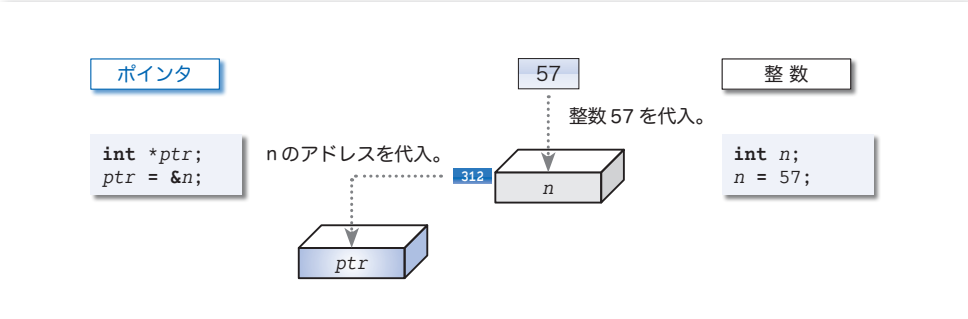

Fig.1-7 整数とポインタ

ポインタである *ptr* には **&***n* が代入されます。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* に代入される値は、*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** 型へのポインタ型です。**Fig.1-8** に示すように、**int** 型の値が『整数』であるのに対して、**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。

重要 Type 型のオブジェクト *x* にアドレス演算子 **&** 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。

▶ **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”という表現を用いることにします（Type 型という型が存在するわけではありません）。

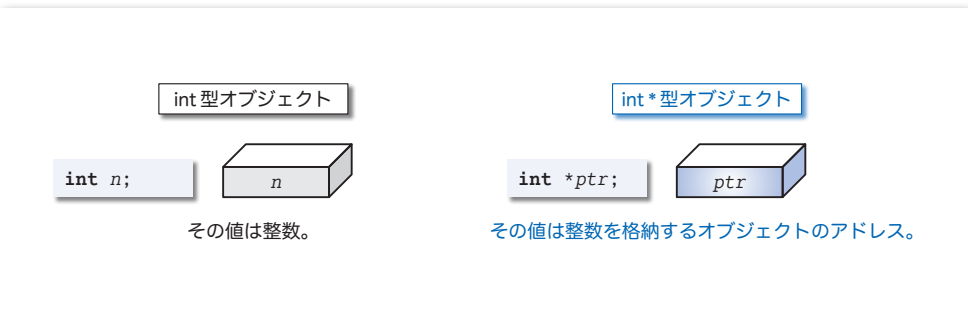

Fig.1-8 int型オブジェクトとint *型オブジェクト

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。

ここで、次のことを必ず覚えましょう。

重要 Type * 型のポインタ *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 **List 1-2** では、“*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.1-9** 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。

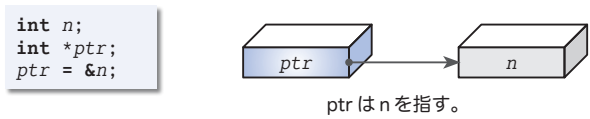

Fig.1-9 ポインタがオブジェクトを“指す”（その 1）

int 型の *n* と **int** * 型の *ptr* は、いずれ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。

Column 1-3 型について

型（*type*）は、オブジェクトや関数の返却値の意味を決定付けるものであり、**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もつことのできる値は整数に限られます。したがって、

```
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ようとしても、小数部は切り捨てられます。そのため、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。

このように、オブジェクトの振る舞いは、型に基づいて決定されるのです。

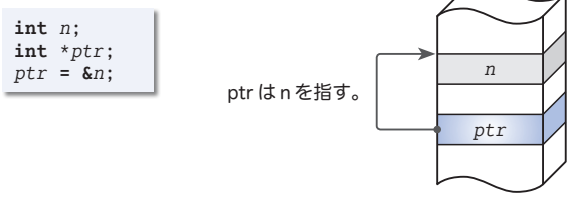

Fig.1-10 ポインタがオブジェクトを“指す”（その２）

いずれの図からも、

> 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。

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、次の形式で利用します。

```
(型) 式
```

この式全体は、**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。

変数 *x* と *y* が **int** 型であって、それらの平均値を求める例で、キャストについての理解を深めていきましょう。変数 *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、以下のように小数部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** 型が混在する算術演算では、**int** 型が**暗黙の型変換**によって **double** 型に変換されます。そのため、**double** 型どうしの除算が行われて、3.5 が得られます。

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

間接演算子 *

次に学習するのは、**List 1-3** のプログラムです。

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;
}
```

実行結果

```
n    の値＝57
*ptrの値＝57
```

前のプログラムと同様に、*ptr* に **&***n* が代入され、ポインタ *ptr* は *n* を指します。

網かけ部では、式 ***ptr** の値を表示しています。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。ポインタに * を適用した式は、**そのポインタが指すオブジェクトそのものを表す式となります。**

このプログラムでは、*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* そのものを表す式になります。

▶ 演算子 * が間接演算子とみなされるのは、***x** という単項形式で利用された場合です。*x * y* という２項形式で利用された場合は、乗算演算子となります。

一般に、*p* が *x* を指すときに“***p** が *x* そのものを表す”ことを、『***p** は *x* の**エイリアス**（*alias*）である。』と表現します。エイリアスとは、別名・あだ名のことです。式 ***p** のことを、変数 *x* に与えられた《あだ名》と考えるわけです。

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、*p* に間接演算子 * を適用した式 ***p** は *x* のエイリアス（別名）となる。

Fig.1-11 を見ながら、理解を深めていきましょう。図 a は、前ページの図と同じもので、ポインタ *ptr* がオブジェクト *n* を指している様子を表しています。

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* のエイリアスというわけです。

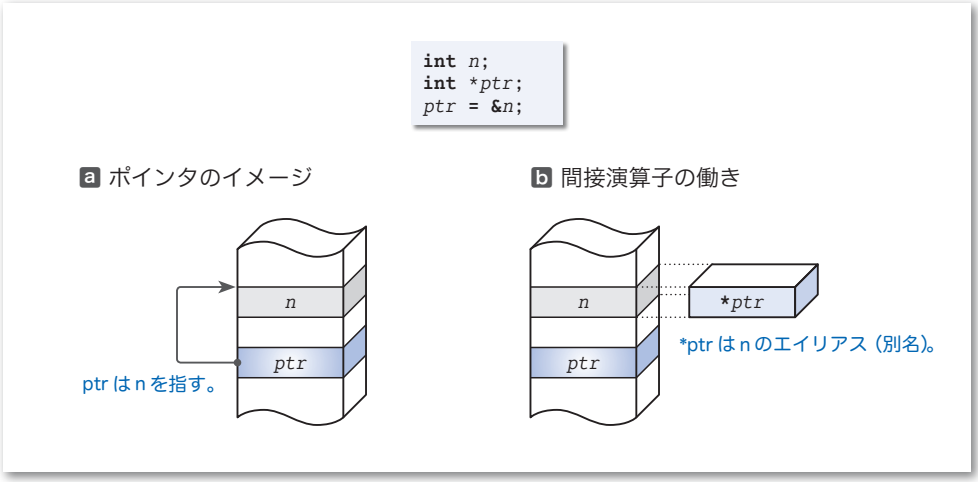

Fig.1-11 ポインタとエイリアス

すなわち、図aは“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。

網かけ部で *ptr の値を出力すると、n の値である 57 が表示されるのでしたね。その理由が、“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。

なお、『*ptr という変数が存在する』と勘違いしてはいけません。

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.22）によって -17 という値が得られるのであって、決して -n という変数が存在するわけではありません。

ポインタ ptr に間接演算子 * を適用した式 *ptr も同様です。決して、*ptr という変数が存在するわけではありません。

*

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 間接演算子 * を適用できるのは、ポインタのみである。

そのため、以下のプログラムは、コンパイルエラーとなります。

```
printf("*nの値＝%d\n", *n);      /* コンパイルエラー */
```

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 n1, n2;
    int *p;

    p  = &n1;   /* pにn1のアドレスを代入：pはn1を指す */
    *p = 123;   /* pが指すn1に123を代入 */

    p  = &n2;   /* pにn2のアドレスを代入：pはn2を指す */
    *p = 567;   /* pが指すn2に567を代入 */

    printf("n1の値＝%d\n", n1);
    printf("n2の値＝%d\n", n2);

    return 0;
}
```

実行結果

```
n1の値＝123
n2の値＝567
```

このプログラムでは、ポインタ *p* を通じて、変数 *n1* と *n2* に 123 と 567 を間接的に代入しています。その動作イメージを示したのが、**Fig.1-12** です。

1では、ポインタ *p* は *n1* を指しています。**p* への 123 の代入によって、*n1* に 123 が代入されます（図a）。2では、ポインタ *p* は *n2* を指しています。**p* への 567 の代入によって、*n2* に 567 が代入されます（図b）。

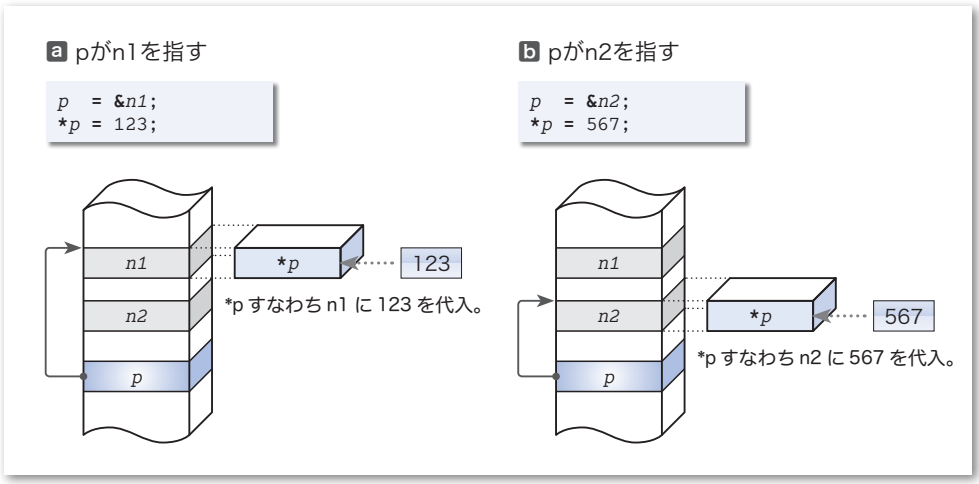

Fig.1-12 ポインタが指すオブジェクトへの値の代入

バイトとアドレス

アドレス付与の対象となる単位は、オブジェクトではなく**バイト**（*byte*）です。それでは、その1バイトとは何ビットなのでしょうか。

多くの環境では、1バイトは8ビットで構成されていますが、1バイトが9ビットや32ビットのコンピュータも実在します。そのため、標準Cでは、1バイトのビット数は処理系によって異なるものであって、その値は『少なくとも8である』と定義されています。

自分の使っている環境における1バイトのビット数を調べるのは簡単です。というのも、<limits.h>ヘッダ中でオブジェクト形式マクロ**CHAR_BIT**として定義されるからです。

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。

以下に示すのが、定義の一例です（**Fig.1-13**）。

CHAR_BIT

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

▶ 1バイトのビット数は、**char**型が占有するビット数と一致します（次ページで学習します）。

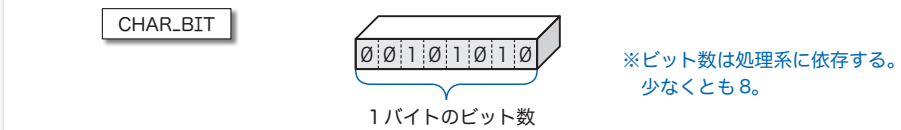

Fig.1-13 1バイトのビット数とCHAR_BIT

このマクロを利用して1バイトのビット数を表示するプログラムが、**List 1-5**です。

List 1-5 chap01/list0105.c

```
/* 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。
```

プログラムを実行してみましょう。お使いの処理系での1バイトのビット数が確認できます。

オブジェクトの大きさと sizeof 演算子

1 バイトのビット数が分かりました。それでは、各型のオブジェクトの大きさは、どうなっているのでしょうか。実は、この大きさも、処理系によって異なります。

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。そのプログラムが **List 1-6** 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4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*）を利用した、以下の式で取得できます。得られる値の単位は、ビットではなくバイトです。

```
sizeof ( 型名 )
```

sizeof(char) の値だけは、すべての処理系で 1 となり、それ以外の型の大きさは、処理系に依存することになっています。実行例に示す環境では、**int** 型が 2 バイトで **long** 型が 4 バイトです（**Fig.1-14**）。

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。

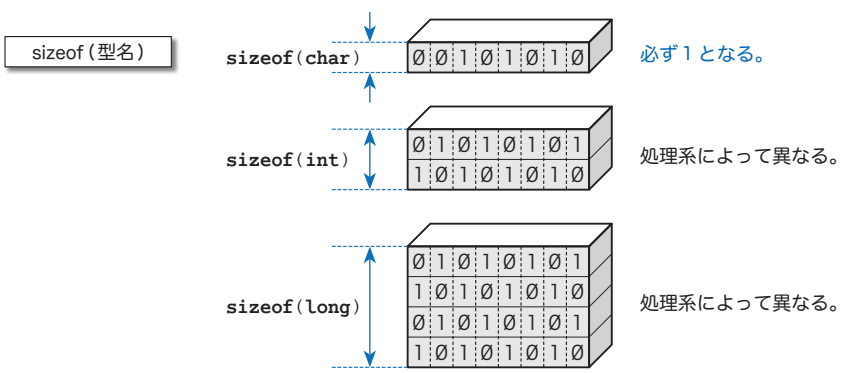

Fig.1-14 sizeof演算子が生成する値

この型は、**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。以下に示すのは、定義の一例です。

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 1-8**（p.25）で学習します。

sizeof 演算子が生成した値は、以下のように、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 1-5**）。

```
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、1の方法で **unsigned** 型にキャストして出力しています。

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こと。なお、大きな値が予想される（アドレス空間が広い）環境では、**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 1-5 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**（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、キャストが必要なのです。

型とビット数

Type 型オブジェクトが記憶域を占有するバイト数は、**sizeof**(Type) ですから、そのビット数は **sizeof**(Type) * **CHAR_BIT** となります。もしも 1 バイトが 8 ビットで、**sizeof**(**int**) の値が 4 であれば、**int** 型が占有するのは 32 ビットです。

もっとも、占有する全ビットが利用されるとは限りません。**Fig.1-15** に示すように、32 ビット中 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（28 ビットのみを用いて整数値を表す）処理系もあります。

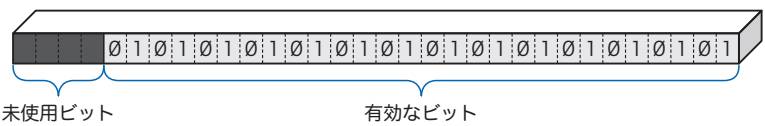

Fig.1-15 整数型のビット構成

すなわち、次のようにいえるのです。

重要 Type 型の有効なビット数が **sizeof**(Type) * **CHAR_BIT** であるとは限らない。

参考のために、**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。プログラムを実行して、お使いの処理系での **int** 型の有効なビット数を調べてみましょう。

List 1-7 chap01/list0107.c

```
/* int型の有効なビット数を表示 */

#include <stdio.h>

/*--- int型／unsigned int型の有効なビット数を求める ---*/
int int_bits(void)
{
    int count = 0;      /* 有効なビット数 */
    unsigned x = ~0U;   /* 全ビットが0であるunsigned int型値 */

    while (x) {
        if (x & 1U)     /* 最下位ビットが1であれば */
            count++;    /* countをインクリメント */
        x >>= 1;        /* 全ビットを右にシフトして最下位ビットを弾き出す */
    }
    return count;
}

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

    return 0;
}
```

実行結果一例

```
int型の有効なビットは16ビットです。
```

バイト順序

オブジェクトの内部に関して、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.1-16** 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。このとき、図**a**に示すように、下位バイトが先頭側（低アドレス）に配置される処理系と、図**b**に示すように、下位バイトが末尾側（高アドレス）に配置される処理系とがあります。

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。

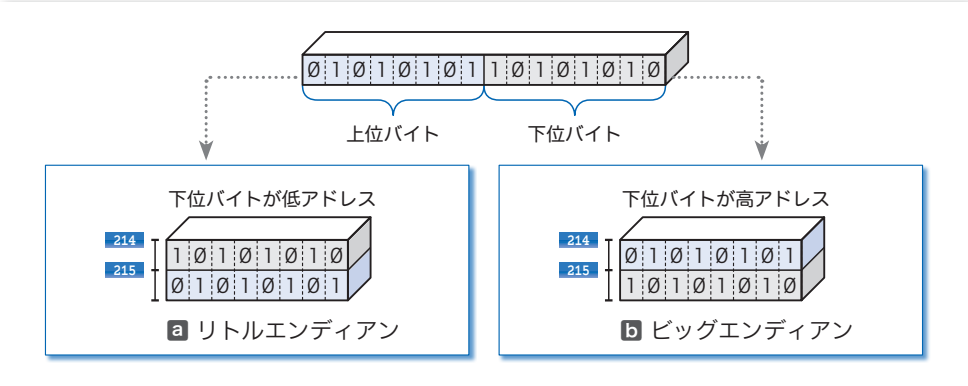

Fig.1-16 バイト順序とエンディアン

▶ 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なのでしょうか。ポインタの大きさも処理系によって異なるものの、やはり **sizeof** 演算子を使って調べられます。**List 1-8** のプログラムで確認しましょう。

List 1-8 chap01/list0108.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。
```

p.14 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.1-17**）。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。

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。

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

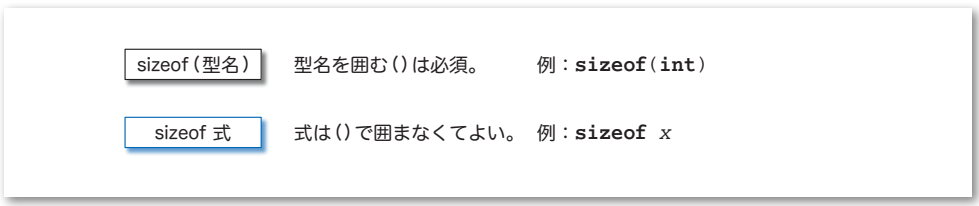

Fig.1-17 sizeof演算子の二つの形式

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。

※ 演習問題の解答は、p.327 以降に示しています。

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ね。

```
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)
```

Column 1-6 演算子の結合性と代入演算子

演算子について理解するためには、**結合性**（*associativity*）について必ず知っておかなければなりません。

ここで、同一の優先順位をもつ２項演算子を○と表して、以下の式を考えましょう。

```
a ○ b ○ c
```

この式の解釈は、演算子の結合性によって異なります。**左結合性**をもつ演算子は、

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

とみなされ、**右結合性**をもつ演算子は、

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

とみなされます。

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、代入時に小数部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。

ポインタの宣言と初期化

以下の宣言を考えましょう。

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

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。この宣言を分解すると、以下のようになります。

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

すなわち、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 複数のポインタを宣言するときは、それぞれの変数に * を忘れないように。

したがって、正しい宣言は、以下のようになります。

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

それでは、**List 1-9** のプログラムを考えましょう。

List 1-9 chap01/list0109.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
```

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 を指すようにするには、

```
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。

＊

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 chap01/list0110.cpp

```
/* ポインタの初期化（C++）*/

#include <iostream>

using namespace std;

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

    cout << " nの値＝" << n  << '\n';        // nの値
    cout << "*pの値＝" << *p << '\n';        // pが指すオブジェクトの値
}
```

実行結果

```
nの値＝123
*pの値＝123
```

網かけ部の宣言に着目しましょう。int と * がくっついています。C++ では、型名と * の間に空白を入れずに宣言するスタイルが主流です。

▶ C 言語でも C++ でも、型名と * との間の空白の有無によって、プログラムの意味が変わることはありません。

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 1-7 C++ の main 関数の返却値

標準 C++ では、return 文に出会うことなく、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することになっています。

```
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。

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` が結び付けられた式 `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.1-18** 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すことにします。左側の小さな文字が《型》で、右側の大きな文字が《値》です。

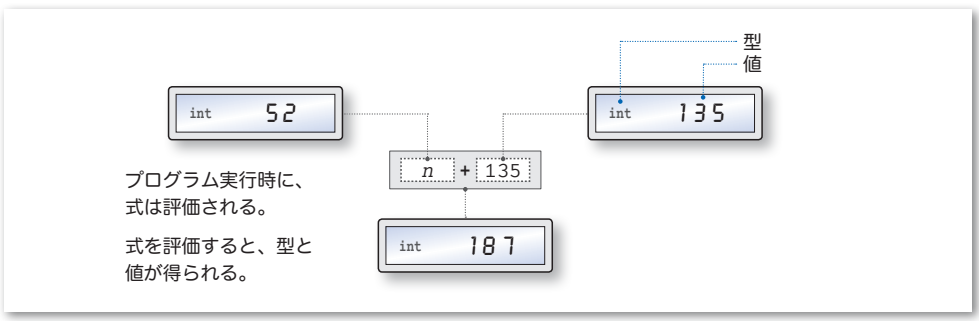

Fig.1-18 式と評価

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 ***** を適用した式について考えていきましょう。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.1-19** 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。

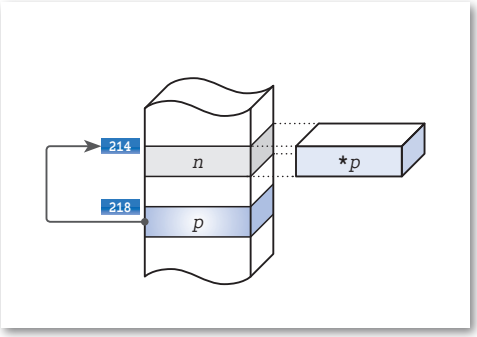

Fig.1-19 ポインタとオブジェクト

*

▪ 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.1-20** a です。

式 *n* を評価すると、**int** 型の 75 が得られます。これは分かりますね。

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。

そして、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、すなわち **int** 型の 75 です。

▶ 図には書いていませんが、**&***p* を評価して得られるのは、**int** ** 型の 218 です。ポインタへのポインタである **int** ** 型については、後の章で学習します。

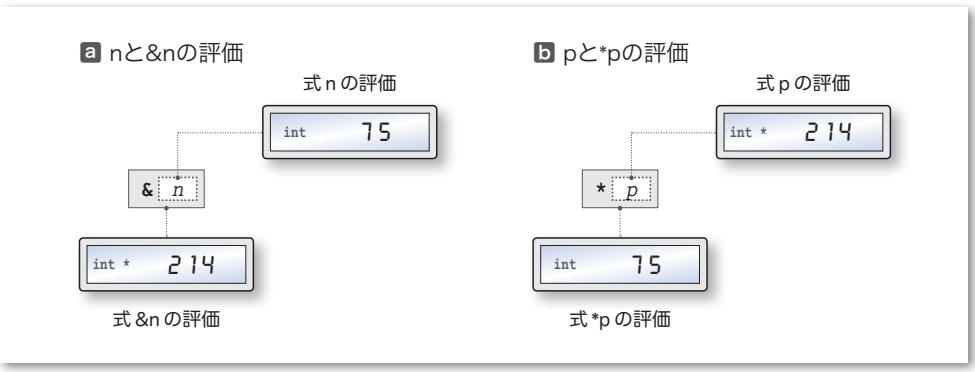

Fig.1-20 アドレス式と間接式の評価

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、一種の整数と考えられます（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 1-11** 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。

List 1-11 chap01/list0111.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、定義されません。

したがって、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という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なのです。

重要 ポインタの値を整数値に変換する場合は、なるべく **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 1-8 typedef 宣言

typedef 宣言は、型に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）。

ここで、*KINGAKU* という名前の型を、以下の宣言によって導入することにします。

```
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。そうすると、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、一目で分かり、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である処理系に移植する、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てください。**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ね。

register 記憶域クラス指定子とアドレス

次に考えるのは、**List 1-12** 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てみてください。

List 1-12 chap01/list0112.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*）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 **&** を適用できないことになっています。そのため、変数 n のアドレスを取得しようとする網かけ部がコンパイルエラーとなるのです。

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。

これは、**register** 宣言されたオブジェクトが、コンピュータの主記憶上ではなく、**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ことによります（**Fig.1-21**）。

▶ 標準Cでは、ハードウェア・環境・処理系などに依存するような内容のことがらは規定されていません。そのため、『**register** 宣言が、レジスタへの変数の格納を示唆するものである』と明文化されているわけではありません。

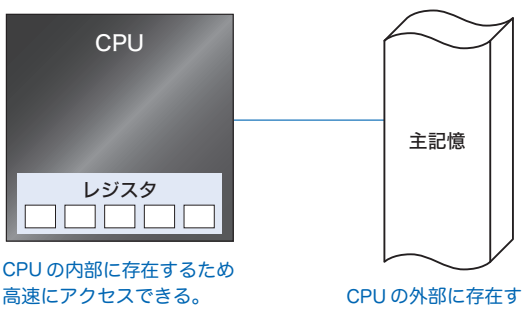

Fig.1-21 レジスタと主記憶

右に示すプログラムの変数 i や j のように、**for** 文や **while** 文などの繰返し文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 < 500; j++) {
        /*… 処理 …*/
    }
}
```

ただし、レジスタの数は限られていますので、レジスタに格納される変数は数個程度に限定されます。

どの変数をレジスタに優先的に割り当てればプログラム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。

しかし、コンパイルの技術が進歩した現在では、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割り当てればよいかを、コンパイラ自身が判断できるようになってきています。そのため、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。

▶ ただし、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や実行環境も存在します。

こ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。

List 1-13 のプログラムで確認しましょう。このプログラムは、ちゃんとコンパイルできますし、実行もできます。

List 1-13 chap01/list0113.cpp

```
// 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 n;

    cout << "&nの値は" << &n << "です。\n";      // ＯＫ！
}
```

実行結果一例

```
&nの値は124です。
```