第１章
見えないエラー

とある砂漠に、人間をも喰いつくす妖怪がひそんでいる蟻地獄があります。その蟻地獄は、人間の目では見えません。あなたは勇気を奮ってその砂漠を横断できますか？

本章では、たった一行のヘッダを題材として、目に見えないエラーや見えにくいエラーなどの《落とし穴》を紹介します。

もっとも、プログラミングの落とし穴というものは、本質的に見えにくいものですが…。

1-1　見えないエラー

本節では、私たちが気付かないうちにプログラム中に忍び込む〔**見えないエラー**〕や〔**見えにくいエラー**〕について、実例をもとに学習していきます。

見えないエラー

List 1-1 に示す "max.h" は、受け取った二つの引数 a, b のうち、大きい方の値を返す関数形式マクロ max を定義するヘッダです。

List 1-1

```
/*
    関数形式マクロmaxを定義するヘッダ "max.h"（見えないエラーが潜む）
*/

#define max(a, b)       ((a) > (b) ? (a) : (b))
```

このヘッダをインクルードして、max を利用するプログラム例が **List 1-2** です。

List 1-2

```
/*
    関数形式マクロmaxを利用するプログラム （見えにくいエラーが潜む）
*/

#include  <stdio.h>

#include  "max.h"

int main(void)
{
    int  x, y;

    printf("xの値は：");   scanf("%d", &x);
    printf("yの値は：");   scanf("%d", &y);

    printf("max(x, y) = %d\n", max(x, y));

    return (0);
}
```

コンパイルすると、このプログラムからインクルードされる "max.h" に対して、

エラー　予期しない EOF です。

というエラーメッセージが表示されます。『予期していないファイルの終端です。』とのことですから、プログラムに必要な〔**何か**〕が欠落していると推測されます。

▶ 具体的なメッセージは、処理系によって異なります。本書に示すエラーメッセージや警告メッセージは、一例です。

このプログラムに潜在するのは〔**見えないエラー**〕ですから、印刷されたプログラムリ

ストを眺めるだけでは、どこが誤りかは分かりません。

ヘッダ "max.h" の内部を覗(のぞ)いてみましょう。それが、**Fig.1-1 a**です。

a 不正なヘッダ

```
#define max(a, b)       ((a) > (b) ? (a) : (b)) EOF
```

改行文字がない。

b 正しいヘッダ

```
#define max(a, b)       ((a) > (b) ? (a) : (b)) ↵
EOF
```

注： EOF はファイルの終端を、↵ は改行文字を表す。

Fig.1-1　ヘッダの実現

マクロを定義する **#define** 指令行の末尾に改行文字がなく、いきなりファイルの終端となっています。

しかし、**#define** や **#include** などの**前処理指令**（*preprocessing directive*）は、改行文字で終わらなければなりません。正しいのは、**b**に示す実現です。

重要　前処理指令行の末尾には、必ず改行文字を付けよう。

よく考えると、この〔**重要**〕は少し変です。最後の行を除く途中の行には、改行文字は必ず入っています。また、前処理指令でなくても、終端を表す改行文字は必要です。

早速〔**重要**〕をいいかえることにします。

重要　ヘッダを含めソースファイルの最後の行にも必ず改行文字を付けよう。

不正な**a**は、一部の処理系では許容されますが、そうでない処理系への可搬性は損なわれます。

『私は××の処理系しか使わないから。』とか『私はワークステーションしか使っていないので、パソコン用の処理系のことは関係ないから。』といった感じで、可搬性の重要性を受け入れない人が、あまりにも多いようです。これまでに、そういった趣旨のお手紙を何通もいただきました。

しかし、本当にそうでしょうか。もし、一部の国にのみ通用するパスポートと、全世界に通用するパスポートを、**同程度の労力や出費**で入手できるとしたら、あなたはどちらを選びますか？

プログラム開発時に、それほど余計なコストがかからない範囲で構いませんから、次のように心がけましょう。

重要　プログラムはできるだけ可搬性が高くなるように実現せよ。

見えにくいエラー

ヘッダ "max.h" に改行文字を挿入した上で、再び **List 1-2** のプログラムをコンパイルしてみます。そうすると、今度は、次のメッセージが出力されます。

エラー　不正な文字 '0x81' です。

エラー　不正な文字 '0x40' です。

▶ メッセージ中の文字コードは、日本語文字コードとして、〔シフト JIS コード〕を利用している処理系での値です。

このエラーを経験したことがある人は、決して少なくないでしょう。**Fig.1-2** に示すように、変数 *x*, *y* の宣言において、**int** の左側の空白が、日本語（いわゆる全角）の空白文字になっているのです。

Fig.1-2　不正な空白

もちろん、これは許されません。**タブ文字か半角の空白文字を使うべきです。**

重要　プログラム中の空白として、日本語文字の空白文字を使ってはいけない。

▶ 空白文字を□などの記号で代替表示するエディタや、日本語文字上のカーソル幅がそれに即した大きさで表示されるエディタを使えば、このようなエラーは防げます。

プログラム中の空白として利用できるのは、**空白文字、改行、水平タブ、垂直タブ、書式送り**です。これらは**空白類文字**（*white-space character*）と呼ばれます。

▶ 前処理指令では、# から改行文字の間に利用できる空白類文字は、**空白文字**と**水平タブ文字**のみに限定されます。

字下げのために使われるタブ文字の幅は環境によって異なるため、別の環境で作られたプログラムの表示や印刷の際は、文字がずれて読みにくくなります。

タブ文字を適当な幅の空白文字へ変換して表示するプログラムを **List 1-3** に示します。タブ幅が実行時に指定できるため、使い勝手のいいものです。

▶ 本プログラムで利用している ***fopen*** 関数や ***fclose*** 関数などは、第 8 章で学習します。

＊

ここで紹介した二つのエラーは、印刷されたプログラムリストを眺めるだけでは、なかなか発見できないものです。処理系が出力するエラーメッセージの意図を的確に把握する能力を身に付けておかなければなりません。

List 1-3

```
/*
    detab … 水平タブ文字を展開する "detab.c"
*/

#include  <stdio.h>
#include  <stdlib.h>

/*--- srcからの入力をタブを展開してdstへ出力 ---*/
void detab(FILE *src, FILE *dst, int width)
{
    int  ch;
    int  pos = 1;

    while ((ch = fgetc(src)) != EOF) {
        int  num;
        switch (ch) {
         case '\t':
            num = width - (pos - 1) % width;
            for ( ; num > 0; num--) {
                fputc(' ', dst);
                pos++;
            }
            break;
         case '\n':
            fputc(ch, dst); pos=1; break;
         default:
            fputc(ch, dst); pos++; break;
        }
    }
}

int main(int argc, char *argv[])
{
    int   width = 8;     /* 既定幅は8 */
    FILE  *fp;

    if (argc < 2)
        detab(stdin, stdout, width);          /* 標準入力 → 標準出力 */
    else {
        while (--argc > 0) {
            if (**(++argv) == '-') {
                if (*++(*argv) == 't')
                    width = atoi(++*argv);
                else {
                    fputs("パラメータが不正です。\n", stderr);
                    return (1);
                }
            } else if ((fp = fopen(*argv, "r")) == NULL) {
                fprintf(stderr, "ファイル%sがオープンできません。\n", *argv);
                return (1);
            } else {
                detab(fp, stdout, width);     /* ストリームfp → 標準出力 */
                fclose(fp);
            }
        }
    }
    return (0);
}
```

本プログラム detab は、オペレーティングシステムのコマンドライン上から実行します。"test.c" という名前のファイルをタブ幅 4 で表示するには、次のように実行します。

```
> detab -t4 test.c↵
```

指定を省略した場合は、タブ幅は 8 となります。

複数のファイルを指定すると、連続して出力されます。

```
> detab test.c xyz.c↵
```

見落としやすいエラー

次に取り上げる **List 1-4** は、最初に示した **List 1-1** とは別のプログラマが作成したものです。ここに潜む誤りは分かりますか？

List 1-4

```
/*
    関数形式マクロmaxを定義するヘッダ "max.h"（見落としやすいエラーが潜む）
*/

#define max (a, b)      ((a) > (b) ? (a) : (b))
```

このヘッダをインクルードするプログラムをコンパイルすると、マクロ *max* を呼び出す箇所に対して、次のエラーメッセージが表示されます。

エラー　不正な構文です。

このエラーを正確に理解するために、2種類のマクロを簡単に復習しましょう。

オブジェクト形式マクロ（object-like macro）

たとえば、次のようなマクロです。コンパイル時に、*TRUE* が 1 に置換されます。

```
#define TRUE   1
```

▶ 文字列リテラルや文字定数中の *TRUE* は置換の対象外となります。

関数形式マクロ（function-like macro）

単純に置換されるのではなく、引数を含めた展開が行われます。

▶ 引数がなく、（ ）内が空となっている関数形式マクロもあります。

両者を区別する基準は単純です。

マクロ名の後に、**空白類文字が続けばオブジェクト形式マクロ**、**(が続けば関数形式マクロ**です。

List 1-4 をよく見ましょう。*max* と (の間に空白があります。

そのため、**Fig.1-3 a** に示すように、*max* はオブジェクト形式マクロとみなされ、それが、

```
(a, b) ((a) > (b) ? (a) : (b))
```

に置換されることになります。

図**b**が、正しい宣言とその展開結果です。

a 誤 … max はオブジェクト形式マクロ

```
#define max (a, b)  ((a) > (b) ? (a) : (b))

max(x, y)
  ⬇ 置換
(a, b) ((a) > (b) ? (a) : (b))(x, y)
```

余分な空白文字。

b 正 … max は関数形式マクロ

```
#define max(a, b)  ((a) > (b) ? (a) : (b))

max(x, y)
  ⬇ 展開
((x) > (y) ? (x) : (y))
```

Fig.1-3　不正なマクロと正しいマクロ

プログラムを読みやすくするために、空白やタブを入れるのは大事なことです。しかし、どこにでも入れていいわけではありません。

重要　関数形式マクロの定義では、マクロ名と（ の間に空白を入れてはいけない。

ただし、以下に示す例のように、関数形式マクロの呼出し側では、**マクロ名と（ の間に空白を入れても構いません**。

```
z = max (x, y);          /* 呼出しでは、maxと(の間に空白があってもよい */
```

▶ 関数形式マクロという用語は、《関数と同じように呼び出せる》ことに由来します。

＊

識別子を（ ）で囲んで呼び出すと、**マクロの展開が抑制されて**、*max* という関数が呼び出されます。

```
z = (max)(x, y);         /* マクロではなく関数を呼び出す */
```

重要　識別子を（ ）で囲むとマクロの展開が抑制される。

List 1-5 のプログラムで確認しましょう。

List 1-5

```
/*
    同名の関数とマクロを呼び分けるプログラム例
*/

#include  <stdio.h>

/*--- マクロ版 ---*/
#define max(a, b)   ((a) > (b) ? (a) : (b))

/*--- 関数版 ---*/
int (max)(int a, int b)
{
    puts("関数版maxが呼び出されました。");
    return (a > b ? a : b);
}

int main(void)
{
    int  x, y;

    printf("xの値は：");   scanf("%d", &x);
    printf("yの値は：");   scanf("%d", &y);

    printf("max(x, y) = %d\n\n", max(x, y));        /* マクロ版を呼び出す */

    printf("(max)(x, y) = %d\n", (max)(x, y));      /* 関数版を呼び出す */

    return (0);
}
```

実行例

```
xの値は：15↵
yの値は：7↵
max(x, y) = 15

関数版maxが呼び出されました。
(max)(x, y) = 15
```

このテクニックは、同一の名前をもつ関数と、関数形式マクロを、使い分けて呼び出すときに利用できます。

前処理指令内の空白

前処理指令内の空白といえば、私が初めてC言語を使ったときのことを思い出します。私の人生において初めて書いたCプログラムの1行目である

```
#include  <stdio.h>
```

は、『C言語は自由形式だから。』と考えて、# の左側に空白文字を入れました。ところが、そのとき使用していた処理系は、意味不明なエラーメッセージを出力して、私のプログラムを受け付けてくれませんでした。当時は、前処理指令の # は、行の先頭に位置しなければならないという規則を定めた処理系が多かったようです。

もちろん標準Cでは、そのような制限はありません。# の前だけでなく、# と **include** の間にも、空白文字や水平タブ文字があってよいのです。

Fig.1-4 に示すように、前処理指令に適当なインデントを設けると、プログラムは読みやすくなります。

```
#if defined(__DOHC__)
    #include  <double.h>
#else
    #include  <single.h>
#endif
```

Fig.1-4　前処理指令のインデント

#if 指令と注釈

Fig.1-4 で利用している **#if** 指令の有効な利用例をご存知ですか。

Fig.1-5 のプログラムは、何らかの理由で、

```
a = x;
```

という文をそっくりコメントアウトしようという意図で作られたものです。

```
/*
    a = x;        /* aにxを代入 */
*/
```

これは注釈とはみなされない。

Fig.1-5　間違ったコメントアウト

しかし、**注釈は〔入れ子〕にはできません。** 注釈とみなされるのは**青文字**の部分です。

入れ子になった注釈を許す処理系も存在しますが、プログラムの可搬性を考えると、そのことに依存すべきではありません。

そもそも注釈は、**プログラムを読む人間に伝えたい情報**を与えるものであって、プログラムをコメントアウトするものではありません。

Fig.1-6 のように、**#if** 指令を用いて記述するのが最もよいやり方です。

条件判定に用いられる式の値は、0 すなわち偽ですから、網かけ部は、コンパイル時に読み飛ばされます。

読み飛ばされる。

```
#if 0
    a = x;        /* aにxを代入 */
#endif
```

Fig.1-6　正しいコメントアウト

重要　注釈は、プログラムをコメントアウトするためのものではない。プログラムのコメントアウトには、**#if** 指令を利用せよ。

処理系によっては、#if と 0 の間にスペースがない

```
#if0
```

を受け付けるようです。しかし、if と、それに続く式（この場合は 0）の間は、空いていなければなりません。

重要 #if と続く式の間には必ず空白を入れよ。

以上をまとめたのが、**Fig.1-7** です。# の前や # と if の間には空白を入れても入れなくともよいのですが、if と式の間には必ず空白を入れるようにしましょう。

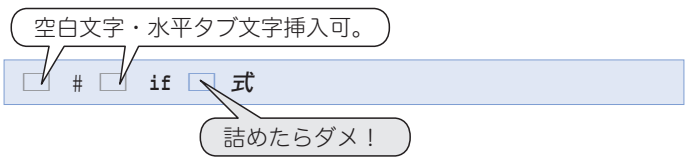

Fig.1-7　#if 指令と空白

デバッグ時など、プログラムの一部をコメントアウトしたり／しなかったり … と頻繁に切りかえる場合は、**List 1-6** のように実現するとよいでしょう。

List 1-6

```
/*
    #if指令によるプログラムのコメントアウト
*/

#include  <stdio.h>

#define DEBUG    0

int main(void)
{
    int  a = 5;
    int  x = 1;

#if DEBUG == 1
    a = x;              /* aにxを代入 */
#endif

    printf("aの値は%dです。\n", a);

    return (0);
}
```

実行結果 1
aの値は5です。

1に書きかえると

実行結果 2
aの値は1です。

プログラムの冒頭で DEBUG が 0 と定義されていますので、プログラムの網掛け部は読み飛ばされて無視されます。

この部分を読み飛ばしたくなければ DEBUG の定義を 1 に変えます（実行結果②）。

重要 プログラムのデバッグに伴う、プログラム部分の有効化／コメント化の実現には #if 指令をうまく利用せよ。

ヘッダの実現

マクロの定義は、（同じである限り）何度も行えます。Aに示すように、2回繰り返されても構いません。

```
A #define para  10
  #define para  10
```

それでは、Bに示すように同一ヘッダを複数回インクルードするとどうなるでしょう。

```
B #include  "max.h"
  #include  "max.h"
```

『そんなことはしないよ。』と思われるかもしれませんが、**気付かないだけで**、よく行っていることです。

たとえば、ヘッダ "abc.h" 中で（*max* の定義が必要なために）"max.h" をインクルードしているとします。そうすると、

```
#include  "max.h"    /* "max.h"を直接インクルード */
#include  "abc.h"    /* "max.h"を"abc.h"を通じて間接的にインクルード */
```

では、"max.h" を2回インクルードすることになります。

右に示す例のように、変数や関数の定義を含むヘッダを複数回インクルードすると、それらを〔**再定義**〕することによるエラーとなります。

```
/* "def.h" */
int  a;
```

```
#include  "def.h"
#include  "def.h"
```

＊

何度インクルードされても問題ないようにしたヘッダを **List 1-7** に示します。

List 1-7

```
/*
    多重のインクルードに対するガードを施した "max.h"
*/

#if !defined(__MAX)
#define __MAX

#define max(a, b)   ((a) > (b) ? (a) : (b))

#endif
```

2回目以降は読み飛ばされる。

最初にインクルードされたとき

__MAX は定義されていませんから、網掛け部によって *__MAX* と *max* が定義されます。

2回目以降にインクルードされたとき

__MAX は定義済みであって、!defined(*__MAX*) の判定が成立しませんから、網掛け部が読み飛ばされます。

重要　ヘッダは、2回目以降のインクルード時に、その本体部分が読み飛ばされるように実現せよ。

▶　第7章では、構造体の宣言を含んだ、より複雑なヘッダの実現法を学習します。

マクロと実行効率

マクロ *max* を利用して、四つの数値 *a*, *b*, *c*, *d* の最大値を求めてみましょう。

```
x = max(max(a, b), max(c, d));
```

これを展開したものを **Fig.1-8** に示します。

何をしたいのか、さっぱり理解できないでしょう。読みにくいだけではありません。このマクロ呼出しは、演算効率も非常に悪いものです。

```
x = max(max(a, b), max(c, d));
```

理解困難

⬇ 展開

```
x = ((((a) > (b) ? (a) : (b))) > (((c) > (d) ? (c) : (d))) ?
     (((a) > (b) ? (a) : (b))) : (((c) > (d) ? (c) : (d))));
```

Fig.1-8　四値の最大値を求める max の呼出しと展開結果（1）

なお、四つの数値の最大値は、次のようにしても求められます。

```
x = max(max(max(a, b), c), d);
```

これだと、**Fig.1-9** のように展開されます。

```
x = max(max(max(a, b), c), d);
```

理解困難

⬇ 展開

```
x = ((((((a) > (b) ? (a) : (b))) > (c) ? (((a) > (b) ? (a) : (b))) : (c))) > (d) ?
     ((((((a) > (b) ? (a) : (b))) > (c) ? (((a) > (b) ? (a) : (b))) : (c))) > (d));
```

Fig.1-9　四値の最大値を求める max の呼出しと展開結果（2）

なんだか、ますます悪くなってしまいました。

これらの手続きにおいて、> 演算子による比較は、いったい何回行われるでしょうか。回数を数えるまでもなく、演算効率が悪いことは明白です。

まさに〔**見た目からは気付きにくい問題点**〕といえます。

右に示すように、**if** 文を羅列して実現した方が、演算の効率はよくなります。プログラムは 4 行になって見かけ上は長くなります。

```
x = a;
if (b > x) x = b;
if (c > x) x = c;
if (d > x) x = d;
```

しかし、何ごとも**素直が一番**です。

重要　プログラムの見かけが短い方が、実行時の効率がよいとは限らない。

C++ での max の実現

C++ では、C 言語よりもスマートに *max* を実現できます。簡単に紹介します。

インライン関数

List 1-8 に示すように、関数定義に **inline** 指定子を加えると、その関数は**インライン関数**（*inline function*）となります。

List 1-8

```
//
//  インライン関数maxを定義するヘッダ（C++）  "max.h"
//

//--- インライン関数 ---//
inline int max(int a, int b)
{
    return (a > b ? a : b);
}
```

関数形式マクロと同様に、プログラム中に展開されて埋め込まれますので、関数呼出しに伴うオーバヘッドがありません。**マクロ版と同等な高速性**が期待できます。

▶ 繰返し文を含むような複雑・大規模な関数は、インラインに展開されない可能性があります。その場合は、通常の関数と同じように扱われます。

マクロでは、呼出し *max*(x++, y) が、((x++) > (y) ? (x++) : (y)) に展開されてインクリメントが 2 回行われるという**副作用**が発生します。しかし、インライン関数では、このような**副作用の問題は生じない**ことが保証されます。

ここに示す実現は、**int** 型しか扱えないことが欠点です。

多重定義

引数の型や個数が異なるとき、同一名の関数を複数個定義できる**関数多重定義**（*function overloading*）を利用した "max.h" を **List 1-9** に示します。

C++ 特有に定義されるマクロ名 **__cplusplus** によって、C++ ではインライン関数版を提供し、C 言語では関数形式マクロ版を提供するヘッダです。

List 1-9

```
/*
    関数多重定義によるmaxを定義するヘッダ（C／C++）  "max.h"
*/

#if defined __cplusplus                                          /* C++ */
    inline int    max(int    a, int    b) { return (a > b ? a : b); }
    inline long   max(long   a, long   b) { return (a > b ? a : b); }
    inline double max(double a, double b) { return (a > b ? a : b); }
#else                                                            /* C */
    #define max(a, b)   ((a) > (b) ? (a) : (b))
#endif
```

関数形式マクロ版 *max* が、> 演算子で比較可能なすべての型の引数に対して有効であるのに対して、ここに示すインライン関数版は、適用される型が **int**, **long**, **double** の三つに限られることが欠点です。

関数テンプレート

引数として〔型〕を受け取る**関数テンプレート**（*function template*）を利用すると、この制約から解放されます。

max を関数テンプレートとして定義する **"max.h"** を **List 1-10** に示します。

List 1-10

```
/*
    関数テンプレートによるmaxを定義するヘッダ（C／C++）　"max.h"
*/

#if defined __cplusplus                                          /* C++ */
    template <class Type> Type max(Type a, Type b)
    {
        return (a > b ? a : b);
    }
#else                                                            /* C */
    #define max(a, b)   ((a) > (b) ? (a) : (b))
#endif
```

▶ C++ では関数テンプレートを提供し、C 言語では関数形式マクロ版を提供します。

関数テンプレートを呼び出すと、呼び出し側の型に即した関数の実体が、コンパイラによって**自動的に生成されます**。したがって、**int** 用、**double** 用といった具合で、個別に関数を作る手間から解放されます。

このヘッダをインクルードして利用するプログラム例を **List 1-11** に示します。

List 1-11

```
//
//  関数テンプレートmaxを利用するプログラム（C++）
//

#include  <iostream>

#include  "max.h"

using namespace  std;

int main(void)
{
    int  x, y;

    cout << "xの値は：";   cin >> x;
    cout << "yの値は：";   cin >> y;

    cout << "max(x, y) = " << max(x, y) << endl;

    return (0);
}
```

実行例

```
xの値は：15
yの値は：7
max(x, y) = 15
```

1-2　初期化

匿名希望の読者の方から、次のような質問をいただきました。

> 友人の作ったプログラムを参考にしながらC言語の学習を進めています。友人のプログラムには、初期化していないにもかかわらず値が0になっている変数と、そうでない変数があるようです。これらの違いを教えてください。

変数の初期化を怠(おこた)るミスは、私たちプログラマが起こしやすいものです。軽微な作業ミスが、プログラム実行上の重大なエラーにつながることもあります。

本節では、初期化について学習します。

初期化と代入

変数の〔初期化〕と聞くと、次のような宣言が思い浮かぶのではないでしょうか。

```
int  x = 5;
```

初期値を与える5は**初期化子**（*initializer*）と呼ばれ、*x*は5で初期化されます。

なお、**List 1-12**のように初期化子を{ }で囲んで宣言することもできます。

List 1-12

```
/*
    単一のオブジェクトを{ }で囲んだ初期化子によって初期化
*/

#include  <stdio.h>

int main(void)
{
    int  n = {4.5};

    printf("n = %d\n", n);

    return (0);
}
```

実行結果

```
n = 4
```

▶　このプログラムがコンパイル不能であったり、おかしな実行結果が得られるのであれば、その処理系は標準Cに準拠していないことになります。

それでは、次の宣言を考えましょう。

```
int  y = 97.2;
```

int型では取り扱えない小数点以下の部分は切り捨てられますから、*y*の初期値は97.2ではなく97となります。

したがって、次のようにいえるでしょう。

重要　初期化子の値がそのまま初期値になるとは限らない。

初期化と代入は、見かけは似ていますが、値を入れるタイミングが違います。

変数を生成するときに値を入れるのが〔**初期化**〕であって、いったん生成された変数に値を入れるのが〔**代入**〕です。

```
int  m = 3;      /* 初期化 */
int  x;
/* … */
x = 0;           /* 代入 */
```


重要 初期化は、変数の生成時に値を格納する操作であり、既に生成されている変数に値を格納する代入とは異なる。

オブジェクト

次に示す二つの宣言を考えましょう。

```
const int   a;          /* 定数変数（？）*/
int         b;          /* 変数 */
```

ご存知のとおり、*a* の値は変更できません。**変数**（*variable*）という語句は、値が可変であることを意味しますから、*a* を変数と呼ぶのは適当ではありません。

だからといって、*a* を〔定数〕と呼ぶこともできません。整数定数 `1053`、浮動小数点定数 `32.5`、文字定数 `'x'` など、値を直接表現したものが〔定数〕です。

正式な用語は、**オブジェクト**（*object*）です。*a* も *b* もオブジェクトです。

標準Cでのオブジェクトの定義を **Fig.1-10** に示します。『値をもった、適切な大きさの記憶域』と理解しましょう。

> 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。ビットフィールド以外のオブジェクトは、連続する一つ以上のバイトからなる。そのバイトの個数、順序および符号化規則は、明示的に規定するかまたは処理系定義とする。
>
> オブジェクトを参照する場合、オブジェクトは、特定の型をもっていると解釈してもよい。

Fig.1-10　オブジェクトの定義

〔変数〕と〔オブジェクト〕の関係は、**名前**（*name*）と**識別子**（*identifier*）の関係と似ています。いずれも、後者が正式な用語ですが、厳密な区別を必要としない文脈においては、前者を使ってもまったく構いません。C言語のバイブルである、

B. W. Kernighan and Dennis M. Ritchie “The C Programming Language”

でも、*variable*（変数）と *object*（オブジェクト）が混同して使われています。用語の不統一であると揚げ足を取る人もいるようですが … 。

*

さて、オブジェクトの初期化と、その寿命を決定する記憶域期間には、深い関連があります。復習を兼ねながら、そのあたりを学習しましょう。

自動記憶域期間をもつオブジェクトの初期化

1

List 1-13 のプログラムを見てください。

List 1-13

```
/*
    自動記憶域期間をもつオブジェクトの初期化
*/

#include  <math.h>
#include  <stdio.h>

void func(int no)
{
    register int  i;
    auto int      x = 100;

    printf("x = %d\n", x);

    for (i = 0; i < no; i++) {
        double  x = sin((double)i / no);
        printf("x = %f\n", x);
    }
    printf("x = %d\n", x);
}

int main(void)
{
    func(10);

    return (0);
}
```

実行結果

```
x = 100
x = 0.000000
x = 0.099833
x = 0.198669
x = 0.295520
x = 0.389418
x = 0.479426
x = 0.564642
x = 0.644218
x = 0.717356
x = 0.783327
x = 100
```

関数 *func* の仮引数 *no* と、関数内で定義されている変数 *i* と *x* は、この関数の実行中にのみ存在するものです。関数 *func* の実行が終了したら消えてしまいます。

このように、宣言されているブロック { … } を抜け出るまで生きるオブジェクトの寿命が**自動記憶域期間**（*automatic storage duration*）です。

自動記憶域期間が与えられるオブジェクトを **Fig.1-11** に示します。

■ 関数が受け取る仮引数

■ 関数の中で、以下のように定義されたオブジェクト
- 記憶域クラス指定子なしで定義されたオブジェクト
- 記憶域クラス指定子 **auto** を伴って定義されたオブジェクト
- 記憶域クラス指定子 **register** を伴って定義されたオブジェクト

Fig.1-11　自動記憶域期間をもつオブジェクト

▶ 記憶域クラス指定子 **auto** は、省略しても同じです。**register** は、『高速にアクセスできるレジスタに割り当てた方がいいかもしれませんよ。』と処理系に示唆するものです。ただし、**register** 宣言されたオブジェクトが、レジスタに格納されるとは限りません。

関数 *func* には、二つの *x* があります。最初に宣言された *x* は、関数末尾の } までの寿命であり、**for** 文の中で宣言された *x* は、**for** 文末尾の } までの寿命です。

▶ 同一名の識別子が複数ある場合、より内側のブロックで宣言されたものが〔見える〕ようになり、外側のものが一時的に〔見えなく〕なります。

このことは、実行時のオブジェクトの寿命である〔記憶域〕の問題ではなく、ソースプログラム上での識別子が通用する範囲である〔スコープ〕の問題です。

for 文内で宣言する *x* の初期化子は、関数呼出し式です。このように、自動記憶域期間をもつオブジェクトの初期化子は、**定数でなくても構いません**。

この *x* は、**for** 文による繰返しのたびに毎回生成されて、***sin***`((double)`*i* `/` *no*`)` が返却する値で初期化されます。

オブジェクトの生成や初期化のタイミングは、以下のように理解しましょう。

重要 自動記憶域期間をもつオブジェクトは、プログラムの流れがその宣言を通過するときに生成されて初期化される。

＊

明示的に初期化子が与えられていない自動記憶域期間をもつオブジェクトは、不定値で初期化されます。

重要 初期化子が与えられていない自動記憶域期間をもつオブジェクトは、不定値で初期化される。

このことを、**List 1-14** のプログラムで確認しましょう。

List 1-14

```
/*
    初期化子の与えられていない自動記憶域期間をもつオブジェクトが
    不定値すなわちゴミの値で初期化されることを確認
*/

#include  <stdio.h>

int main(void)
{
    int  x;                         /* 不定値で初期化される */

    printf("x = %d\n", x);

    return (0);
}
```

実行結果一例

```
x = 957
```

▶ 実行によって表示される値は不定です。

変数 *x* の初期値は 957 かもしれませんし、-38 かもしれません。もちろん、たまたま 0 ということもあります。

静的記憶域期間をもつオブジェクトの初期化

自動記憶域期間とは対照的な寿命が**静的記憶域期間**（*static storage duration*）です。これらの違いを **List 1-15** のプログラムで学習しましょう。

List 1-15

```
/*
    オブジェクトの記憶域期間（静的／自動）と初期化
*/
#include  <stdio.h>

int  ft = 0;                    /* 静的記憶域期間 */

void func(void)
{
    int          at = 0;        /* 自動記憶域期間 */
    static int   st = 0;        /* 静的記憶域期間 */

    ft++;
    at++;
    st++;
    printf("ft = %d  at = %d  st = %d\n", ft, at, st);
}

int main(void)
{
    int  i;

    for (i = 0; i < 8; i++)
        func();

    return (0);
}
```

実行結果

```
ft = 1  at = 1  st = 1
ft = 2  at = 1  st = 2
ft = 3  at = 1  st = 3
ft = 4  at = 1  st = 4
ft = 5  at = 1  st = 5
ft = 6  at = 1  st = 6
ft = 7  at = 1  st = 7
ft = 8  at = 1  st = 8
```

関数の外で定義された *ft* は、各関数の実行とは無関係に、プログラムの起動時から終了時まで存在しなければなりません。これが静的記憶域期間です。

また、*st* のように関数内で **static** 記憶域クラス指定子を加えて宣言されたオブジェクトにも静的記憶域期間が与えられます。

静的記憶域期間が与えられるオブジェクトをまとめたものを **Fig.1-12** に示します。

- ■ 関数の外で定義されたオブジェクト
- ■ 関数の中で **static** を伴って定義されたオブジェクト

Fig.1-12　静的記憶域期間をもつオブジェクト

これらのオブジェクトはプログラムの実行を通じて生き続けます。初期化は、一度だけ行われるのであり、プログラムの流れが宣言を通過するたびに行われることはありません。

重要　静的記憶域期間をもつオブジェクトは、プログラム開始の準備段階の生成時に、一度だけ初期化され、プログラムの実行を通じて生き続ける。

□静的記憶域期間

関数 *func* の呼出しとは関係なく、変数 *ft* と *st* は、プログラムの実行を通じて値が保持されます。最初は 0 に初期化され、関数が呼び出されるたびにインクリメントされますから、その値は、**関数 *func* を呼び出した回数**となります。

□自動記憶域期間

変数 *at* は、関数 *func* が呼び出されてプログラムの流れが宣言を通過するたびに、生成されて 0 で初期化されます。

静的記憶域期間をもつ変数 *ft* や *st* の宣言から初期化子を取り除いて、プログラムを右のように変更してみてください。『*ft* や *st* は不定値で初期化されることになるのではないか。』という心配をよそに、**変更前と同じ実行結果が得られます**。

これは、次の理由によります。

```
int  ft;

void func(void)
{
    int          at = 0;
    static int   st;
    /*  (中略)  */
}
```

重要　初期化子が指定されていない静的記憶域期間をもつオブジェクトは、0 で初期化される。

なお、静的記憶域期間をもつオブジェクトの初期化子は、**定数式**でなければなりません。

Bの宣言はエラーとなります。

```
void func(void)
{
    int        x  = sin(0.9);  /* A OK */
    static int st = sin(0.9);  /* B エラー */
}
```

＊

二つの記憶域期間に関して、ポイントをまとめたものが **Table 1-1** です。

■ **Table 1-1　記憶域期間とオブジェクトの初期化**

	自動記憶域期間	静的記憶域期間
生　成	プログラムの流れが、その宣言を通過する際に生成される。	**main** 関数実行開始前のプログラム準備段階において生成される。
初期化	明示的に初期化しなければ不定値で初期化される。	明示的に初期化しなければ 0 で初期化される。
初期化子	定数式でなくともよい（ただし配列・構造体は除く）。	定数式でなければならない。
破　棄	その宣言を含むブロックを抜け出る際に破棄される。	プログラム実行終了後の後始末の段階で破棄される。

▶　もう一つの記憶域期間である**割付け記憶域期間**は、第 3 章で学習します。

識別子の有効範囲と初期化

宣言された識別子は、その宣言子の直後、すなわち**初期化子より前**の時点から、その名前が使えるようになります。

このことを **List 1-16** のプログラムで確認しましょう。

List 1-16

```
/*
    不定値である自分自身の値で初期化
*/

#include  <stdio.h>

int  x = 1;

int main(void)
{
    int  x = x;                    /* 自分自身の値で初期化 */

    printf("x = %d\n", x);

    return (0);
}
```

実行結果一例

```
x = 9572
```

▶ 実行によって表示される値は不定です。

main 関数での変数 *x* の宣言における初期化子 *x* は、ここで宣言している *x* 自身のことであって、関数の外で定義された *x* ではありません。

自動記憶域期間をもつ変数 *x* の初期値は不定値ですから、*x* はその不定値で初期化されます。したがって、この初期化子は蛇足（だそく）であって、何の意味もありません。

List 1-17 のプログラムの各変数は、どのような値で初期化されるのでしょうか。みなさんの課題としますので、考えてみてください。

List 1-17

```
/*
    変数の有効範囲と初期化
*/

#include  <stdio.h>

int  z = 1;

int main(void)
{
    int  x = z;
    int  z = 0;
    int  y = z;

    return (0);
}
```

実 行 結 果

*printf*関数の呼出しを追加して、*x*, *y*, *z*の値を確かめてみましょう。

C++ の初期化子

C++ では、以下の二つの宣言は同じです。

```
int  i = 5;          /* 形式A：CとC++に共通 */
int  i(5);           // 形式B：C++特有でC言語では不可
```

これは、**List 1-18** のプログラムで確認できます。実行してみてください。

List 1-18

```
//
//  ( )形式の初期化子によって初期化（C++）
//

#include  <iostream>

using namespace  std;

int main(void)
{
    int  x = 5;              /* xを5で初期化 */
    int  y(5);               /* yを5で初期化 */

    cout << "x = " << x << endl;
    cout << "y = " << y << endl;

    return (0);
}
```

実行結果

```
x = 5
y = 5
```

形式Bは、クラスオブジェクトの初期化と共通性をもたせるよう導入されたものです。以下に示すクラス *Complex* を例にして、要点を説明します。

```
class Complex {                                  // 複素数クラス
    double  re, im;
public:
    Complex(double r, double i = 0.0)            // コンストラクタ
    {
        re = r;
        im = i;
    }
    // ...
};
```

クラス *Complex* 型のオブジェクトの宣言例を示します。

```
Complex  a(5.0, 7.5);        // 宣言X：引数は二つ
Complex  b(5.0);             // 宣言Y：引数は一つ
```

( ) の中に、コンマで区切って引数を並べて、コンストラクタに渡します。ちなみに、引数が一つの場合は、次のように ( ) を使わず宣言できます。

```
Complex  b = 5.0;            // 宣言Z：引数は一つ
```

〔宣言Y〕は形式Bであり、〔宣言Z〕は形式Aです。**int** 型や **double** 型などの基本型オブジェクトが、クラス型と同じように初期化・宣言できるようになっているのです。

配列の初期化

配列の初期化を行う典型的な宣言例を示します。

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

青文字の {1, 2, 3} が配列 a に対する初期化子です。その中の 1, 2, 3 が各要素に対する初期化子であり、配列の要素 a[0], a[1], a[2] は、順に 1, 2, 3 で初期化されます。

List 1-19 のプログラムをコンパイル・実行してみてください。

List 1-19

```
/*
    自動記憶域期間をもつ配列を初期化
*/

#include  <stdio.h>

int  vx[3] = {1, 2, 3};      /* 静的 */

int main(void)
{
    int  i;
    int         ma[3] = {1, 2, 3};      /* 自動：K＆Rでは不可 */
    static int  ms[3] = {1, 2, 3};      /* 静的 */

    for (i = 0; i < 3; i++)
        printf("vx[%d] = %d  ma[%d] = %d  ms[%d] = %d\n",
                                        i, vx[i], i, ma[i], i, ms[i]);
    return (0);
}
```

実行結果

```
vx[0] = 1  ma[0] = 1  ms[0] = 1
vx[1] = 2  ma[1] = 2  ms[1] = 2
vx[2] = 3  ma[2] = 3  ms[2] = 3
```

K&R では、自動記憶域期間をもつ配列に初期化子を与えることはできませんでした。関数中で **static** を与えない配列を初期化することは不可能だったのです。

もちろん、標準Cでは、そのような制限はありません。

配列 ma の宣言に対して、次のエラーを出力する処理系は、標準Cに準拠していないことになります。

エラー　自動記憶域期間をもつ配列の初期化はできません。

List 1-20 のプログラムに進みましょう。要素数が 3 である配列 b に対して、初期化子が一つだけ与えられています。このような場合、次の規則が適用されます。

重要　配列に対する初期化子の個数が、配列の要素数に満たないとき、初期化子が与えられていない要素は 0 で初期化される。

つまり、**Fig.1-13** のように解釈されます。初期化子が与えられていない b[1] と b[2] を 0 で初期化しない処理系は、標準Cに準拠していません。

全要素を 0 で初期化する宣言を **Fig.1-14** に示します。非常に短く宣言できます。

もちろん、初期化子が配列の要素数より多くなることは許されません。たとえば、要素

List 1-20

```
/*
    初期化子が与えられていない要素が0で初期化されることを確認
*/

#include  <stdio.h>

int main(void)
{
    int  i;
    int  b[3] = {1};          /* 先頭から順に1,0,0で初期化される */

    if (b[1] != 0  ||  b[2] != 0)
        puts("正しく初期化されていません。");
    else
        for (i = 0; i < 3; i++)
            printf("b[%d] = %d\n", i, b[i]);

    return (0);
}
```

実行結果

```
b[0] = 1
b[1] = 0
b[2] = 0
```

```
int  b[3] = {1};
  ⬇ このように解釈される
int  b[3] = {1, 0, 0};
```

Fig.1-13　省略された初期化子

```
int  x[1000] = {0};
```

全要素を0で初期化するには、
一つだけ0を与えればよい。

Fig.1-14　すべての要素を 0 で初期化

数が 2 である配列に対して、三つの初期化子を与える宣言はエラーです。

```
int  c[2] = {1, 2, 3};                    /* エラー */
```

配列に対して全く初期化子を与えずに、次のように宣言したらどうなるでしょう。

```
int  d[3];
```

関数の外で定義されるか、関数内で **static** 付きで定義されれば〔静的記憶域期間〕が与えられますから、**全要素が 0 で初期化されます**。

関数内で **static** を伴うことなく定義されると、〔自動記憶域期間〕が与えられますので、**全要素の初期値は不定値**となります。これは、単独のオブジェクトの場合と同じです。

なお、自動記憶域期間であっても、配列に対する初期化子は、必ず定数式でなければなりません（単独のオブジェクトでは、そうではありませんでしたね：p.17）。

これまでに、以下の質問を何度もいただきました。

右のプログラムのように、配列の複数要素に一括して値を代入しようとすると、エラーになるのはどうしてですか。

```
int  x[3];
/* … */
x = {0, 1, 2}; /* エラー */
```

{ … } は、初期化子のための特別な構文であり、それ以外では使えません。C 言語では、配列に一括して値を代入することはできません。

多次元配列の初期化

多次元配列の初期化に関しては、1次元配列に対する初期化の規則が再帰的に適用されます。**List 1-21** を例にして考えていきます。

List 1-21

```
/*
    2次元配列の初期化を確認
*/

#include  <stdio.h>

int main(void)
{
    int  i, j;
    int  x[3][2] = {{0, 1},
                    {2, 3},
                    {4, 5},
                   };
    for (i = 0; i < 3; i++)
        for (j = 0; j < 2; j++)
            printf("x[%d][%d] = %d\n", i, j, x[i][j]);

    return (0);
}
```

実行結果

```
x[0][0] = 0
x[0][1] = 1
x[1][0] = 2
x[1][1] = 3
x[2][0] = 4
x[2][1] = 5
```

配列 x 中の要素は、**Fig.1-15** に示すように、x[0][0], x[0][1], x[1][0], x[1][1], x[2][0], x[2][1] の順で、記憶域上に並んでいます。

すべての要素に対して初期化子が与えられており、その順に初期化されます。

```
int  x[3][2] = {{0, 1},
                {2, 3},
                {4, 5},
               };
```

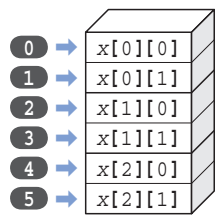

Fig.1-15　2次元配列の初期化（その1）

宣言の初期化子を横に並べてみましょう。

```
int  x[3][2] = { {0, 1}, {2, 3}, {4, 5}, };
```

最後のコンマ文字 , が余計であるように感じられます。このコンマを取って、

```
int  x[3][2] = { {0, 1}, {2, 3}, {4, 5} };
```

としても、まったく同じ初期化が行われます。

最後のコンマは、初期化子を縦に並べた際の見かけ上のバランスをとるためのもので、あってもなくても構いません。

初期化子の構文は、**Fig.1-16** に示すように複雑です。

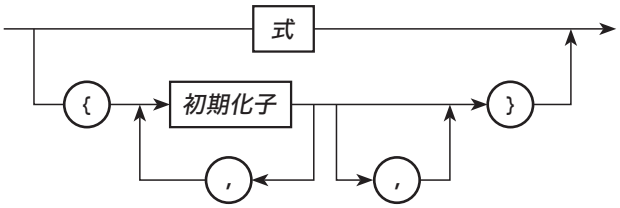

Fig.1-16　初期化子の構文図

この図が示すように、1 次元配列の初期化においても、次のように宣言できます。

```
int  d[3] = {1, 2, 3,};      /* 最後の要素の後に,があってもよい */
```

＊

配列に対する初期化子が不足していれば、その要素は 0 で初期化されるという規則は、多次元配列においても成立します。**Fig.1-17** で確認しましょう。

```
int x[3][2] = { {0},
                {2, 3},
                {4},
              };
```

⬇ このように解釈される

```
int x[3][2] = { {0, 0},
                {2, 3},
                {4, 0},
              };
```

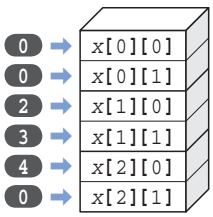

Fig.1-17　２次元配列の初期化（その２）

なお、多次元配列に対する初期化子は、必ずしも { } を入れ子にする必要はありません。**Fig.1-18** に示すように、先頭から順に初期化が行われます。

```
int x[3][2] = {0, 1, 2};
```

⬇ このように解釈される

```
int x[3][2] = { {0, 1},
                {2, 0},
                {0, 0},
              };
```

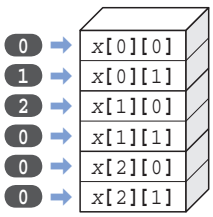

Fig.1-18　２次元配列の初期化（その３）

▶　文字列の初期化については第 4 章で、構造体や共用体の初期化については第 7 章で学習します。

typedef 名が与えられた配列の初期化

typedef 名が与えられた配列の初期化について考えていきます。まずは、**typedef** 宣言とは何かを思い出しましょう。

重要 **typedef** 宣言は、新しい型を作るのではなく、既存の型に新しい名前を与える宣言である。

たとえば、

```
typedef int  INTEGER;        /* INTEGERはintの同義語 */
```

と宣言すると、それ以降、*INTEGER* は **int** と同じ意味となります。すなわち、

```
INTEGER  a;                  /* aは実質的にint型 */
```

は、以下の宣言と実質的には同じです。

```
int  a;                      /* aはint型 */
```

ちなみに、**typedef** 宣言を〔新しい型を作る宣言〕と、多くの人が勘違いしているようです。正しく理解しましょう。

＊

それでは、**List 1-22** に示すプログラムを考えます。要素型が **int** で要素数が 5 である配列型に対して、*Int5ary* という **typedef** 名を与えています。

List 1-22

```
/*
    typedef名による配列の初期化を確認
*/

#include  <stdio.h>

int main(void)
{
    int  i;
    typedef int  Int5ary[5];     /* 要素型がintで要素数が5の配列型 */
    Int5ary  x = {1, 2, 3};

    for (i = 0; i < 5; i++)
        printf("x[%d] = %d\n", i, x[i]);

    return (0);
}
```

実行結果

```
x[0] = 1
x[1] = 2
x[2] = 3
x[3] = 0
x[4] = 0
```

typedef 名が与えられた配列の初期化は、基本的には、通常の配列と同様です。

要素数が 5 である配列 *x* に対して、初期化子が三つしか与えられていませんので、不足部分の要素は 0 で初期化されます。このことは、実行結果からも分かります。

すなわち、プログラム網掛け部の宣言は、実質的には、

```
int  x[5] = {1, 2, 3};
```

と同一です。

次に、**List 1-23** に示すプログラムを考えましょう。

List 1-23

```
/*
    typedef名による不完全な配列の初期化を確認
*/

#include  <stdio.h>

int main(void)
{
    int   i;
    typedef int IntAry[];        /* 要素型がintの配列型 */
    IntAry  a = {1, 2, 3};              /* 要素数は3 */
    IntAry  b = {1, 2, 3, 4, 5};        /* 要素数は5 */

    for (i = 0; i < 3; i++)
        printf("a[%d] = %d\n", i, a[i]);

    for (i = 0; i < 5; i++)
        printf("b[%d] = %d\n", i, b[i]);

    return (0);
}
```

実行結果

```
a[0] = 1
a[1] = 2
a[2] = 3
b[0] = 1
b[1] = 2
b[2] = 3
b[3] = 4
b[4] = 5
```

このプログラムでは、要素型が **int** の配列型に対して、*IntAry* という **typedef** 名を与えています。ただし、**要素数は与えられていません**。

配列 *a* と *b* は、いずれも *IntAry* 型として宣言されています。

与えられた初期化子の個数から配列の要素数が決定されますので、配列 *a* の要素数は 3、配列 *b* の要素数は 5 となります。すなわち、*a*, *b* の宣言は、実質的に、

```
int  a[3] = {1, 2, 3};
int  b[5] = {1, 2, 3, 4, 5};
```

と同じことになります。

▶ *IntAry* は、要素数が不定の不完全型です。不完全型のオブジェクトを作り出すことはできません。初期化子の個数から要素数が決定して完全型となります。