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む蟻地獄があります。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》などについて、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 "max2X1.h" は、受け取った二つの引数 a，b のうち、大きいほうの値を求める関数形式マクロ *max2* を定義するヘッダです。

List 1-1 chap01/max2X1.h

```
/*
    関数形式マクロmax2を定義するヘッダ "max2X1.h"（見えないエラーが潜む）
*/

#define max2(a, b)  ((a) > (b) ? (a) : (b))
```

このヘッダをインクルードして *max2* を利用するプログラム例を **List 1-2** に示します。

List 1-2 chap01/max2X1test.c

```
/*
    関数形式マクロmax2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 "max2X1.h"

int main(void)
{
    int x, y;

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

    printf("max2(x, y) = %d\n", max2(x, y));

    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。

コンパイルすると、このプログラムからインクルードする "max2X1.h" に対して、

```
エラー  予期しない EOF です。
```

との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。』ということですから、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、あくまでも一例です。

このプログラムに潜在するのは《**見えないエラー**》であるため、印刷されたプログラムリストを眺めるだけでは、誤りの正体は分かりません。

ヘッダ "max2X1.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 max2(a, b)  ((a) > (b) ? (a) : (b))EOF
```

改行文字が欠如。

b 正しいヘッダ

```
#define max2(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** 指令の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。

ところが、**#define** 指令や **#include** 指令などの**前処理指令**（*preprocessing directive*）は、改行文字で終わる必要があります。すなわち、図bに示すのが、正しい実現例です。

重 要 前処理指令行の末尾には、必ず改行文字を付けよう。

さて、よく考えると、この《**重要**》は少し変です。途中の行には、改行文字は必ず入っています。また、終端を表す改行文字が必要なのは、前処理指令に限られません。

早速《**重要**》をいいかえることにします。

重 要 ヘッダを含めたソースファイルの最後の行には、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ありません。

*

『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、一部の国のみに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、どちらを選ぶでしょう？

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ので、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しよう。

見えにくいエラー

ヘッダ "max2X1.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、決して少なくないようです。**Fig.1-2** に示すように、変数 *x*, *y* の宣言の行における“**int**”の左側の空白が、日本語（いわゆる全角文字）の空白文字になっているのです。

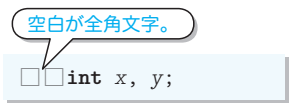

Fig.1-2 不正な空白

もちろん、これは許されません。**半角の空白文字か、水平タブ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わないようにしよう。

▶ 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容易に防げます。

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。

▶ ただし、前処理指令では、# から改行文字までのあいだに利用できる空白類文字が、**空白文字**と**水平タブ文字**のみに限定されます。

さて、字下げ（インデント）のために使われる“水平タブ”は、幅（タブ位置の間隔）が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に、文字がずれて読みにくくなってしまうことがあります。

水平タブ文字を適当な個数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。なお、タブの幅は起動時に指定できるようになっています。

▶ 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つけておかなければなりません。

List 1-3 chap01/detab.c

```
/*
    detab … 水平タブ文字を展開する
*/

#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;      /* 既定の幅は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;
}
```

実行方法

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のタブ幅は8です。

＊

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
```

タブ幅は、ファイルごとの指定も可能です。

```
> detab -t4 test.c -t8 xyz.c
```

見落としやすいエラー

次に取り上げる **List 1-4** は、最初の **List 1-1** とは別のプログラマが作成したヘッダです。ここにも誤りが潜んでいますが、今回は“見た目で分かる”ものです。

List 1-4 chap01/max2X2.h

```
/*
*/  関数形式マクロmax2を定義するヘッダ "max2X2.h"（見落としやすいエラーが潜む）
#define max2 (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〔"chap01/max2X2test.c"〕をコンパイルすると、マクロ *max2* 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
```

▶ ただし、文字列リテラルや文字定数の中の TRUE は置換の対象外です。

▪ 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。

▶ ただし、引数を受け取らない、すなわち、() 内が空の関数形式マクロもあります。

マクロ名の直後に**空白類文字が続けばオブジェクト形式マクロとなり、(が続けば関数形式マクロとなります**。したがって、識別は容易です。

List 1-4 をよく見ましょう。*max2* と (とのあいだに空白があります。

そのため、**Fig.1-3** a に示すように、*max2* がオブジェクト形式マクロとみなされて、その *max2* が、

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

に置換されるのです。

図 b が、正しい宣言とその展開結果です。

a 誤 … max2はオブジェクト形式マクロ

```
#define max2 (a, b)  ((a) > (b) ? (a) : (b))
```

```
max2(x, y)   余分な空白文字。
  ⬇ 置換
(a, b) ((a) > (b) ? (a) : (b))(x, y)
```

b 正 … max2は関数形式マクロ

```
#define max2(a, b)  ((a) > (b) ? (a) : (b))
```

```
max2(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（のあいだに空白を入れないように。

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

```
z = max2 (x, y);    /* 呼出しでは、max2と(のあいだ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。

＊

なお、識別子を（）で囲むと、**マクロの展開が抑制されます**。そのため、

```
z = (max2)(x, y);   /* マクロではなく関数を呼び出す */
```

では、*max2* というマクロとして展開されるのではなく、*max2* という関数が呼び出されます。

重 要 識別子を（）で囲むと、マクロの展開が抑制される。

実際に確認してみましょう。それが、**List 1-5** のプログラムです。

List 1-5 chap01/max2.c

```
/*
    同名の関数とマクロとを使い分けるプログラム例
*/

#include <stdio.h>

/*--- マクロ版 ---*/
#define max2(a, b)  ((a) > (b) ? (a) : (b))

/*--- 関数版 ---*/
int (max2)(int a, int b)
{
    puts("関数版max2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("max2(x, y) = %d\n\n", max2(x, y));      /* マクロ版 */

    printf("(max2)(x, y) = %d\n", (max2)(x, y));    /* 関数版 */

    return 0;
}
```

実行例

```
xの値は：15
yの値は：7
max2(x, y) = 15

関数版max2が呼び出されました。
(max2)(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とを使い分ける際に利用できます。必ず覚えておきましょう。

前処理指令内の空白

前処理指令内の空白といえば、私がC言語を初めて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作ったCプログラムの1行目である

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

を書く際に、『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。#の前はもちろん、#と include のあいだにも、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、プログラムが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

#if 指令の例を示しました。この指令の《定石》ともいえる利用法を学習しましょう。

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 */
*/
```

この */ は注釈とみなされない。

● **Fig.1-5** 不正なコメントアウト

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色文字の部分です。

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、**プログラムの読み手に伝えたい情報**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、よい方法です。

条件判定に用いられる式の値が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 指令を使おう。

さて、処理系によっては、#if と Ø とのあいだにスペースがない

```
#ifØ
```

を受け付けるようです。しかし、if と、それに続く式（この場合の Ø）のあいだには空白が必要です。

重 要 #if と続く式とのあいだには必ず空白を入れよう。

以上をまとめたのが、**Fig.1-7** です。# の前や、# と if のあいだには空白を入れても入れなくてもよいのですが、if と式のあいだには空白が必要です。

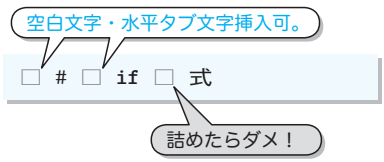

Fig.1-7 #if 指令と空白

＊

さて、デバッグ時などに、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ことがあります。そのような場合は、**List 1-6** のように実現しましょう。

List 1-6 chapØ1/debug.c

```
/*
    #if指令によるプログラムのコメントアウト
*/

#include <stdio.h>

#define DEBUG  Ø ·····1 に書きかえると…·····

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 Ø;
}
```

実行結果1
aの値は5です。

実行結果2
aの値は1です。

プログラムの冒頭で *DEBUG* が Ø と定義されていますので、プログラムの網か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 指令をうまく使おう。

▶ **Column 1-1**（p.12）では、標準ライブラリ NDEBUG マクロを使うデバッグ法を学習します。

インクルードガード

マクロの定義は、（同じである限り）何度も行えます。そのため、1に示すような複数回の定義が可能です。

```
1 #define para  10
  #define para  10
```

それでは、2の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。

```
2 #include "max2.h"
  #include "max2.h"
```

『そんなことはしないよ。』と思われるかもしれません。しかし、**気付かない内に**よく行われていることです。

たとえば、ヘッダ "abcd.h" 中で（*max2* の定義が必要なために）"max2.h" をインクルードしているとします。そうすると、

```
#include "max2.h"   /* "max2.h"を直接インクルード */
#include "abcd.h"   /* "max2.h"を"abcd.h"を通じて間接的にインクルード */
```

では、"max2.h" が２回インクルードされます。

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《**重複定義**》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"
```

＊

List 1-7 に示すのが、何度インクルードしても問題が起こらないヘッダです。

List 1-7 chap01/max2.h

```
/*
    インクルードガードされた"max2.h"
*/

#ifndef __MAX2
#define __MAX2

#define max2(a, b)  ((a) > (b) ? (a) : (b))

#endif
```

２回目以降のインクルード時には読み飛ばされる。

▪ 初めてインクルードしたとき

__MAX2 は定義されていませんので、網かけ部で *__MAX2* と *max2* とが定義されます。

▪ ２回目以降にインクルードしたとき

__MAX2 は定義ずみであって **#ifndef** *__MAX2* の判定が成立しないため、網かけ部は読み飛ばされます。

重要 ヘッダは、２回目以降のインクルード時に、定義などを含む本体部分が読み飛ばされるように**インクルードガード**（*inlcude guard*）しよう。

▶ 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。

関数形式マクロと実行効率

関数形式マクロ *max2* 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 = max2(max2(a, b), max2(c, d));
```

これを展開したものを **Fig.1-8** に示します。

展開後の式を見ても、何を行う式なのか、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 = max2(max2(a, b), max2(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max2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 = max2(max2(max2(a, b), c), d);
```

これだと、**Fig.1-9** のように展開されます。

```
x = max2(max2(max2(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max2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** 文を羅列して実現すると、演算の効率がよくなります。

プログラムは 4 行になって見かけ上は長くなりますが、何ごとも**素直が一番**です。

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

重 要 プログラムの見かけが短いほうが、実行時の効率がよいとは限らない。

関数形式マクロの副作用

以下の代入を考えましょう。

```
z = max2(x++, y);
```

xとyの大きいほうの値がzに代入された後に、xがインクリメントされるように見えます。しかし、そうではありません。マクロを展開してみましょう。

```
z = ((x++) > (y) ? (x++) : y);
```

xのインクリメントが最大で2回行われることが分かります（**Column 2-2**：p.36）。このように、実引数が複数回評価されることなどの原因で、期待するものとは異なる結果が生じてしまうことは、マクロの**副作用**（*side effect*）と呼ばれます。

重 要 関数形式マクロを使うときは、副作用の有無を事前に調べよう。

この例から、以下の教訓が得られます。

重 要 引数を複数回評価する関数形式マクロは、原則として定義すべきでない。

基本的に、関数形式マクロは、抑制的に利用すべきものです。定義する際は、コメントやドキュメントなどで副作用の有無を含めた詳細な仕様を利用者に提供すべきです。

Column 1-1 NDEBUGマクロとassert

List 1-6（p.9）では、プログラム中にマクロを定義してデバッグする手法を解説しました。C言語では、デバッグ支援のために、**NDEBUG**マクロと***assert***マクロとを組み合わせた手法が利用できるようになっています。

まずは、**NDEBUG**マクロです。このマクロ自体は、標準ライブラリとして定義が提供されるものではありません。ソースプログラムの中で

```
#define NDEBUG
```

と定義することもできますが、**通常はコンパイル時のオプションなどの指定によって定義します。**

どの処理系も、**NDEBUG**マクロの定義方法がマニュアルなどに書かれていますので、お使いの処理系のマニュアルを調べてみましょう。

たとえば、Visual Studioの場合、プロジェクトがリリースモードであれば、このマクロは定義されず、デバッグモードであれば自動的に定義されるようになっています。

NDEBUGは、“デバッグするときに定義されて、デバッグしないときに定義されないマクロ”と理解するとよいでしょう。

もう一つの***assert***マクロは、**NDEBUG**マクロが定義されているかどうかによって挙動が変わるマクロです。

ソースファイル中で<assert.h>をインクルードする時点で、**NDEBUG**がマクロ名として定義されている場合は、***assert***マクロは、以下のように定義されます。

```
#define assert(ignore) ((void)0)
```

すなわち、***assert***マクロは、実質的に何も行わないマクロとなります。

一方、NDEBUG マクロが定義されている場合、*assert* マクロは、以下の仕様でプログラムに《診断機能》を付け加えます。

```
void assert(int expression);
```

このマクロは、*expression* が 0 のときに限り、偽の値をもたらしたことに関する情報を標準エラーストリームに書き込むとともに、*abort* 関数を呼び出して、プログラムを強制終了します。

なお、出力する呼出しに関する情報は、処理系に依存することになっていますが、少なくとも以下の情報が含まれます。

- 実引数のテキスト
- ソースファイル名を表すマクロ __FILE__
- ソース行番号を表すマクロ __LINE__ の値

以下に示すのが、出力の情報の形式の一例です。

```
Assertion failed : 実引数のテキスト , file ファイル名 , line 行番号
```

assert マクロを利用したプログラム例を **List 1C-1** に示します。

List 1C-1 chap01/assert_div.c

```
/*
    assertマクロの利用例
*/

#include <stdio.h>
#include <assert.h>

/*--- aをbで割ったときの商と剰余を表示 ---*/
void div(int a, int b)
{
    assert(b != 0);

    printf("%dを%dで割った商は%dで剰余は%dです。\n", a, b, a / b, a % b);
}

int main(void)
{
    int a, b;

    printf("a = ");
    scanf("%d", &a);

    printf("b = ");
    scanf("%d", &b);

    div(a, b);

    return 0;
}
```

実行例

```
a = 7↵
b = 2↵
7を2で割った商は3で剰余は1です。
```

実行結果一例

```
a = 7↵
b = 0↵
Assertion failed: b != 0, file \C\chap01\assert_div.c, line 11
```

関数 *div* は、a を b で割った商と剰余を表示する関数です。もし b が 0 であれば、正しい除算は行えません（除算を行うと実行時エラーとなります）。そこで、b != 0 が成立しない場合（すなわち b が 0 であるとき）に、エラーメッセージを表示してプログラムを強制終了させます。

関数の冒頭に書かれた *assert*(b != 0) は、

『この関数は b が 0 でないことを期待しています。そうでない場合は、強制終了しますよ。』

と読めます。

ちなみに、英語の assert という語句は、『断言する』『主張する』という意味です。

C++ での max2 の実現

C++ では、C 言語よりもスマートに *max2* を実現できます。ざっと学習しましょう。

インライン関数

List 1-8 に示すように、定義に **inline** 指定子が付加された関数は、**インライン関数**（*inline function*）となります。インライン関数は、関数形式マクロと同様に、展開されて埋め込まれます。関数呼出しに伴うオーバヘッドがないため、**マクロ版と同等な高速性**が期待できます（本ヘッダのテストプログラムは〔"chap01/max2_inline_test.cpp"〕です）。

▶ 繰返し文を含むような、複雑あるいは大規模な関数は、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なコードが生成されます。

List 1-8 chap01/max2_inline.h

```
/*
    インライン関数max2を定義するヘッダ（C++）
*/

//--- インライン関数 ---//
inline int max2(int a, int b)
{
    return a > b ? a : b;
}
```

マクロとは異なり、インライン関数では、実引数が複数回評価されることによる**副作用の問題は生じない**ことが保証されます。

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、ここに示した *max2* は、処理対象が **int** 型に限定されることが欠点です。

多重定義

引数の型や個数が異なるのであれば、同一名の関数を複数個定義できる**関数多重定義**（*function overloading*）が行えます。それが **List 1-9** に示す "max2_overload.h" です。

List 1-9 chap01/max2_overload.h

```
/*
    関数多重定義によるmax2を定義するヘッダ（C／C++）
*/

#ifdef __cplusplus                                          /* C++ */
    inline int    max2(int    a, int    b) { return a > b ? a : b; }
    inline long   max2(long   a, long   b) { return a > b ? a : b; }
    inline double max2(double a, double b) { return a > b ? a : b; }
#else                                                       /* C */
    #define max2(a, b)  ((a) > (b) ? (a) : (b))
#endif
```

C++ で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ように工夫しています（本ヘッダのテストプログラムは〔"chap01/max2_overload_test.cpp"〕です）。

ここに示すインライン関数版は、処理対象が **int** 型、**long** 型、**double** 型に限定されます。

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、適用可能な型が限定されるという制約から解放されます。

関数テンプレートとして *max2* を定義するのが、**List 1-10** の "max2_template.h" です。

▶ 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。

List 1-10 chap01/max2_template.h

```
/*
    関数テンプレートによるmax2を定義するヘッダ（C／C++）
*/

#ifdef __cplusplus                                          /* C++ */
    template <typename Type> Type max2(const Type& a, const Type& b)
    {
        return a > b ? a : b;
    }
#else                                                       /* C */
    #define max2(a, b)  ((a) > (b) ? (a) : (b))
#endif
```

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 chap01/max2_template_test.cpp

```
/*
    関数テンプレートmax2を利用するプログラム（C++）
*/

#include <iostream>

#include "max2_template.h"

using namespace std;

int main(void)
{
    int a, b;
    double x, y;

    cout << "整数aの値：";   cin >> a;
    cout << "整数bの値：";   cin >> b;
    cout << "実数xの値：";   cin >> x;
    cout << "実数yの値：";   cin >> y;

    cout << "max2(a, b) = " << max2(a, b) << '\n';
    cout << "max2(x, y) = " << max2(x, y) << '\n';

    return 0;
}
```

実行例

```
整数aの値：15
整数bの値：7
実数xの値：1.25
実数yの値：3.14
max2(a, b) = 15
max2(x, y) = 3.14
```

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の生成が、コンパイラによって**自動的に行われます**。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。