Cから体感するRL78マイコン(CC-RL版)

(C)2010-2012,2017 てきーらサンドム　R1.05 2017/8/15

2章　C言語をかじる（見て触って覚える）
2.1　行儀を正して
2.1.1　標準ヘッダをインクルードする
2.1.2　フォーマルな関数の定義方法
2.1.3　関数の終わりにリターン文
2.2　固定した文字の表示
2.2.1　文字列リテラルの連結
2.2.2　改行やタブ（エスケープ・シーケンス）
2.3　数値（変数）の表示
2.3.1　変数を使うための宣言
2.3.2　式（変数）の値の表示
2.3.3　式と定数
2.3.4　型の変換（暗黙の変換とキャストによる明示的変換）
2.4　配列と繰返し処理
2.4.1　配列の宣言
2.4.2　配列の初期化方法
2.4.3　配列の使い方とポインタ
2.4.4　繰返し処理
2.5　キー入力に応じた動作の選択
2.5.1　キーから1行取得（gets関数）
2.5.2　書式付き入力（sscanf関数）
2.5.3　if文とbreak文
2.5.4　switch文
2.5.5　論理演算式
2.5.6　代入と複合代入
2.6　関数電卓
2.6.1　数学関数
2.6.2　#define文

2.7　文字列の操作と構造体
2.7.1　文字列の操作関数
2.7.2　構造体（struct）と typedef
2.7.3　構造体の使い方
2.7.4　カンマ演算子（コンマ演算子）の使いどころ
2.8　関数の使い方と変数の通用範囲
2.8.1　関数プロトタイプ宣言
2.8.2　関数の呼び出し方法
2.8.3　変数の通用範囲
2.9　補足
2.9.1　ポインタの加減算
2.9.2　用語の補足

2章　C言語をかじる

この章では，基本的な C 言語プログラムについて解説します。パソコンでも動作するようなレベルですので，C言語をある程度知っている人は読み飛ばしてもよいでしょう。

ただし一部に，今回使うマイコン特有の説明も含まれています（水色マーカ部分)。

2.1　行儀を正して

１章で解説したリスト1－1のプログラムは動くには動くのですが，コンパイル時に次の2つのワーニング（警告）が出ます。

main¥main_1-1.c(2):W0523076:関数宣言はプロトタイプが必要です。

main¥main_1-1.c(3):W0523077:呼び出される関数はプロトタイプが必要です。

もう少し詳しい解説がコンパイラのヘルプで見ることができます。

これらのメッセージをクリックした上で、右クリックして

「メッセージに関するヘルプ」を選択すると表示されます。

ワーニングが出ても動く可能性はありますが，動いたとしてもすぐに発覚しない恐ろしいバグが含まれている可能性が十分あります。ワーニングは必ず消すように修正します。

次のリスト2－1は2つのワーニングを消したものです。

```
/* リスト2－1  行儀のよいプログラム */
#include <stdio.h>                  /* 解説 2.1.1 */
int main (void) {                   /* 解説 2.1.2 */
    printf("今日も元気だ！");
    return 0;                       /* 解説 2.1.3 */
}
```

2.1.1　標準ヘッダをインクルードする

関数を使う場合は，どんな関数なのかを示した”関数プロトタイプ宣言（2.8.1 参照）”が必要です。

しかし，printf のような標準関数については，すでに宣言が書かれたファイル（ヘッダと言う）があるので，それを読み込むだけで良いのです。

ヘッダ読み込みは次のように書きます。

#include <stdio.h>

└─ 標準ヘッダ名：使用する関数の種類によって決まっています。
printf の場合は，stdio.h です。

一般に main 関数は必須の関数なので、プロトタイプ宣言は不要なコンパイラが多いようですが、CC-RL では必要なので、

```
int main(void);
```

という行も必要です。ドライバ・キットでは stdio.h の中に記述しているので、上記の include 記述を書くだけでよいです。

2.1.2　フォーマルな関数の定義方法

リスト 1－1 の main 関数の定義方法は，手抜きした書き方です。

一般的な関数定義はリスト 2－1 のように書きます。

int main(void) {具体的動作}
①　②　　③

①戻り値の型

関数の計算結果を戻り値（返り値，返却値）と言います。

C 言語で扱う数値には，数値範囲や用途により何種類かの型（後述）があります。

main 関数の戻り値は，標準的には int 型(そのマイコンで標準的に扱う整数値)と決められています。

戻り値が無い関数もあり，その場合は，void を付け加えます。

CC-RL では，戻り値が無い main 関数も許されているので，void main とも書けます。

②関数名

C 言語で標準的に用意されている関数名以外であれば自由に名前を決められます。

③引数の型

y=f(x)の x のように関数に与える数値を引数と言います。

引数がある場合はその型を書きますが，無い場合は void と書きます。

main 関数は引数を伴う形式（ main(int argc, char* argv[]) ）も許されていますが，CC-RL の標準スタートアップ・ルーチン(cstart.asm:main を呼び出すプログラム)や ドライバ・キット同梱のスタートアップ・ルーチン（RESET.asm）では実引数を格納しないので void とします。

2.1.3　関数の終わりにリターン文

関数の計算（具体的動作）が終了して，戻り値を返すには，リターン文を使います。

書き方：

```
return 0;
       └── 戻り値。戻り値が無い関数の場合は，この部分は書かない。
```

戻り値が無い関数ではリターン文を省略可能ですが，書くのが行儀がよいです。

main 関数の戻り値は，パソコンでは OS がプログラムの終了状態（異常の有無）を判定するために使用しています。前述の標準スタートアップ・ルーチンや RESET.asm では戻り値を使用してないので何を返しても良く，とりあえず 0 を返しています。規格上は main 終了後に後始末関数 exit(引数：main 戻り値)と等価な処理をすることになっていますが，CC-RL 標準スタートアップ･ルーチンでは、戻り値をクリアして無限ループに入ります。RESET.asm では何もせず無限ループに入ります。

2.2　固定した文字の表示

ここでは，printf 関数を使って，複数行の表示をしてみます。

```
/* リスト 2－2　文字列リテラルの連結と
                エスケープ・シーケンスの使用例 */
#include <stdio.h>
int main (void) {
    printf("今日も元気だ！¥n"
           "全てが順調だ！¥n"
           "A¥b¥"¥t ってね。");
    return 0;
}
```

実行結果 2－2

```
今日も元気だ！
全てが順調だ！
"    ってね。
```

2.2.1　文字列リテラルの連結

リスト 2－2 では，printf 関数の引数として""で囲んだ文字の並びを与えています。

""で囲まれた一連の文字を文字列リテラルと言います。

printf 関数は，基本的には第 1 引数の文字列の通りに表示します。ただし¥ や % で始まる部分は特殊な扱いになります。¥については次項で説明します。%については 2.3 節で説明します。

リスト 2－2 のように複数の文字列リテラルをカンマで区切らずに書くと，連結されて 1 つの文字列リテラルと見なされます。

"文字列 A"　"文字列 B"　→　"文字列 A 文字列 B"
　　　　　↑
　　　　この部分の空白や改行，タブ，コメントは無視されます。

従って，リスト 2－2 の printf 関数は，次の 1 行と同じです。

```
printf("今日も元気だ！¥n 全てが順調だ！¥nA¥b¥"¥t ってね。");
```

補足：CS+for CC では、文字列内の文字コード種別をプロパティの中で選択できます。

画面左のプロジェクト・ツリーの"CC-RL(ビルド・ツール)"を右クリックし、

"プロパティ"→"コンパイル・オプション" →"文字コード"

の中で「SJIS」を選択します(LCD 表示フォントを SJIS コードで検索のため)。

他のコードを選択すると「不正な多バイトコード」というワーニングが出ます。

2.2.2　改行やタブ（エスケープ・シーケンス）

¥記号で始まる文字の組合せをエスケープ・シーケンスと言います。

改行やタブといった特殊な動作や，”のような特別な文字の表示に使います。

実際の表示例（実行結果 2－2）を見ると，

・¥n の部分で改行されています。

・３行目先頭の文字 A は¥b で消されています。

・３行目に”が表示されています（文字列リテラルの終わりとは見なされてない）。

・タブによりスペースが空いています。

表 2－2　主なエスケープ・シーケンス

表記	対応コード。() 内は 16 進数表現	一般的用法
¥0	0（0x0）	ヌル文字
¥b	8（0x7）	バックスペース
¥t	9（0x7）	タブ
¥n	10（0xA）	改行
¥r	13（0xD）	復帰
¥”	34（0x22）	“
¥’	39（0x27）	‘
¥?	63（0x3F）	?
¥¥	92（0x5C）	¥
¥x16 進数字	0～255（0～0xFF）	任意のコード

C 言語（特にマイコン用）では，16 進数を良く使います。

16 進数を表現するには，0x または 0X につづけて 16 進数字を書きます。

16 進数字は，10 進数の 0～15 に対応して，0～9，A～F または a～f を使います。

2.3　数値（変数）の表示

ここでは，数式で使う変数とその表示方法について説明します。

リスト 2-8 に加減乗除の計算例を示します。

```
/* リスト 2−3 変数の定義，計算式と表示 */
#include <stdio.h>
int main (void) {
    int x = 100;                 /* 解説 2.3.1 */

    printf("x+3=%d¥n"            /* 解説 2.3.2 */
           "x-105=%4d¥n"
           "x*3=%+5d¥n"
           "x/3=%d, %4.1f¥n",
           x + 3,                /* 解説 2.3.3 */
           x - 105,
           x * 3,
           x / 3, x / 3.0);
    return 0;
}
```

実行結果 2−3

```
x+3=103
x-105=   -5
x*3= +300
x/3=33, 33.3
```

CC-RL での補足

上のリストでは浮動小数点(小数点付き数値)が使われています。浮動小数点を使う必要が無ければ、ライブラリ・サイズの小さい tiny 版を使うのが良いでしょう。

tiny 版を使うには、stdio.h のインクルードより前にマクロ__PRINTF_TINY__ を定義すれは良いです。

tiny 版の制約事項の詳細は、CC-RL コンパイラ・ユーザーズ・マニュアルのライブラリ関数仕様の printf の項を参照してください。

2.3.1　変数を使うための宣言

変数を使うには，最初に変数に名前を付けるとともに，その変数がどこまでの数値を扱えるかといったような型を指定する必要があります。次のように書きます。

```
int x, y, z;
```

変数名の並び：名前には英数字および_が使用できます。名前の先頭は英字または_で始めます。
大文字／小文字は区別されるので，a と A は別の名前になります。

型指定子：表 2－3 に型指定子の種類とその数値範囲を示します。

変数には初期値を与えることが出来ます。次のように変数名の後ろに = 記号と初期値を書きます。

```
int x = 100, y = 200;
```

表 2－3　CC-RL での数値変数の型と数値範囲

型指定子	数値範囲	データ・サイズ
_Bool	0,1	8 ビット（1 バイト）
char	-128～+127	8 ビット（1 バイト）
unsigned char	0～255	
short, int	-32768～32767	16 ビット（2 バイト）
unsigned short, unsigned int	0～65535	
long	-2147483648～+2147483647	32 ビット（4 バイト）
unsigned long	0～4294967295	
float	±約 1.18×10^{-38}～3.4×10^{38}	
long long	±約 9.2^{-18} の範囲の整数	64 ビット(8 バイト)
double	±約 2.2×10^{-308}～1.8×10^{308}	

注 1：int は，一般的にはそのマイコン（計算機）に適した計算サイズになります（通常 16 ビット以上)。そのため 32 ビット・マイコンであれば long と同じサイズになります。もしマイコンを変えたときにサイズが変わると困る場合は short，long を使用します。short，long もサイズが保証されている分けではないですが，多くのマイコンにおいてそれぞれ 2 バイト，4 バイトとなっています。

注 2：_Bool は CS+forCA(CA78K0R コンパイラ)ではサイズが 1bit でしたが、CC-RL では、ビット・フィールド表現に変更しない限り 1bit サイズにはなりません。

注 3：char は、デフォルト・オプションでは「unsigned char 型扱い」となっています。ドライバ・キットでは、「signed char 型扱い」に変更してあります。

2.3.2　式（変数）の値の表示

printf 関数で式の値を表示するには，第 1 引数の文字列の中で，表示したい位置に % 記号から始まる特別な指定を行い，その部分を置き換える式を printf 関数の 2 番目以降の引数として並べます。%指定は次のように書きます。

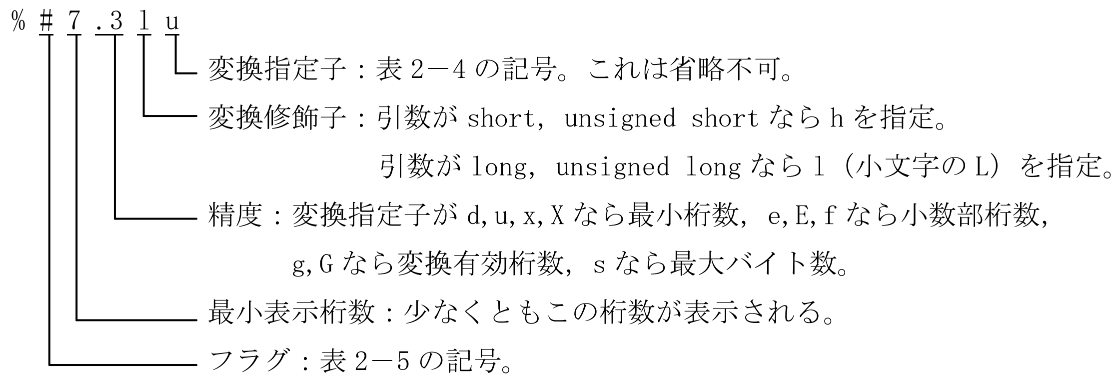


表 2－4　おもな変換指定子

記号	意味
d	符号付き整数の表示。負の場合のみ－を表示する。
u	符号無し整数の表示（引数が unsigned 整数型）。
x,X	16 進数字で表示。x は a～f，X は A～F を 16 進数文字として用いる。
f	整数と小数で表示。
e,E	指数表現。指数部の前に e または E を表示。
g,G	f,e,E の内最短で済む形式。
c	文字コードと見なして文字に置き換える。
s	文字列表示（具体例は 2.4 節を参照）。
%	%を表示（変数や式の置き換えをしない）。

表 2－5　フラグ

記号	意味
－	左詰めにする。
＋	常に＋，－の符号をつける。
空白	上記＋の代わりに空白をつける。
0	最小表示桁数に満たないときは 0 詰めする。
＃	変換指定子に応じて意味が変わる。 x：先頭に 0x を付ける。X：先頭に 0X を付ける。 e,E,f：必ず小数点をつける。g,G：小数部の 0 を略さない。

2.3.3　式と定数

(1)式

リスト 2−3 で示すとおり，加減乗除には，+ − ＊ / の記号を使います。

これらの記号を演算子と言います。演算子の一覧を表 2−6 に示します。

演算子はグループ単位で優先順位が決められています。

例：x + y ＊ z　→　y ＊ z が先に計算される。

またグループ内では結合方向が定められています。

例：x ＊ y % z　→　x ＊ y が先に計算される。

2 項演算で評価順序が不定の演算子もあります。

例：++b + c / b → ++b の計算が先か c/b の計算が先かで結果が変わる。

表 2−6　演算子一覧　　　（優：優先順位，結：結合方向，評：評価順序(不:不定)）

<table>
<tr><th>演算子記号【意味，解説または使用例】</th><th>優</th><th>結</th><th>評</th></tr>
<tr><td>[]【配列添字付け，2.4.1～2.4.3】</td><td rowspan="3">1</td><td rowspan="3">→</td><td>不</td></tr>
<tr><td>()【関数呼出し，2.8.2】.【直接メンバ，2.7.3】->【間接メンバ，2.7.3】</td><td>左</td></tr>
<tr><td>++【後置インクリメント，2.4.4(1)】--【後置デクリメント，2.4.4(1)】</td><td>左</td></tr>
<tr><td>++【前置インクリメント，2.4.4(1)】--【前置デクリメント，2.4.4(1)】
&【アドレス演算，2.4.3(2)】＊【間接参照】
+【正】-【負】~【反転，2.5.5】!【論理否定，2.5.5】
sizeof【変数の大きさ，2.9.1】</td><td>2</td><td>←</td><td>右</td></tr>
<tr><td>（型名）【キャスト，2.3.4】</td><td>3</td><td>←</td><td>右</td></tr>
<tr><td>＊【乗算，2.3.3】/【除算，2.3.3】%【剰余】</td><td>4</td><td>→</td><td>不</td></tr>
<tr><td>+【加算，2.3.3】-【減算，2.3.3】</td><td>5</td><td>→</td><td>不</td></tr>
<tr><td><<【左シフト】>>【右シフト】</td><td>6</td><td>→</td><td>不</td></tr>
<tr><td><【右側より小さい，2.4.4(1)】>【右側より大きい，2.4.4(1)】
<=【右側以下，2.4.4(1)】>=【右側以上，2.4.4(1)】</td><td>7</td><td>→</td><td>不</td></tr>
<tr><td>==【等しい，2.4.4(1)】!=【等しくない，2.4.4(1)】</td><td>8</td><td>→</td><td>不</td></tr>
<tr><td>&【ビット単位の AND，2.5.5】</td><td>9</td><td>→</td><td>不</td></tr>
<tr><td>^【ビット単位の排他 OR，2.5.5】</td><td>10</td><td>→</td><td>不</td></tr>
<tr><td>|【ビット単位の OR，2.5.5】</td><td>11</td><td>→</td><td>不</td></tr>
<tr><td>&&【論理 AND，2.5.5】</td><td>12</td><td>→</td><td>左</td></tr>
<tr><td>||【論理 OR，2.5.5】</td><td>13</td><td>→</td><td>左</td></tr>
<tr><td>？：【条件演算，2.4.4(2)】</td><td>14</td><td>←</td><td>左</td></tr>
<tr><td>=【代入，2.5.6】*= /= %= += -= <<= >>= &= ^= |=【複合代入，2.5.6】</td><td>15</td><td>←</td><td>不</td></tr>
<tr><td>,【カンマ演算，2.7.4】</td><td>16</td><td>→</td><td>左</td></tr>
</table>

(2) 定数と型

式では，変数だけでなく定数もよく使います。定数は書き方により型が決まります。

・小数点または指数表現（e か E が付く）ならば浮動小数点型（double）になります。
　ただし末尾に F が付くと float になります。

　例：　123.456，123e-4，123.456E7　　→　double 型
　　　　123.456F，123e-4F　　　　　　→　float 型

・それ以外は整数型で，数字の大きさにより int→unsigned int→long→unsigned long
　の順に自動的に決まります。ただし小さい数字でも末尾に L が付くと long です。

　例：　-128，　255，　32767，　-32768　→　int 型
　　　　65535　　　　　　　　　　　　　→　unsigned int 型
　　　　100000，-65000，2L　　　　　　→　long 型
　　　　2147483648　　　　　　　　　　→　unsigned long 型

・16 進数を表現するには，0x または 0X につづけて 16 進数字を書きます。16 進数字は，
　10 進数の 0～15 に対応して，0～9，A～F または a～f を使います。

　例：　0x12af　　　→　int 型
　　　　0x1234ABcd　→　long 型

・シングルクォーテーションで文字を囲むと，対応する文字コードが数値定数（int 型）
　になります。エスケープ・シーケンス（表 2－2）も使えます。

　例：　‘2’ → 50（0x32），　‘A’ → 65（0x41），　‘¥n’ → 10（0x0a）

CC-RL での補足事項

RL78 は 8 ビット・マイコンをベースにした 16 ビット・マイコンのため、たとえば 16 ビットの論理演算(and, or, exor)命令が無いなど、多少演算能力が劣る面があります。そのため従来(CA78K0R コンパイラ)は、コード効率や処理速度を少しでも上げるべく、“char 型演算を符号拡張しない”ことがデフォルトになっていたオプションがありました。

しかし、CC-RL では見当たりません(もしかしたら見落としているのかも)。タイミング・クリティカルな部分で char 演算を多用している場合は、注意が必要かもしれません。

2.3.4　型の変換（暗黙の変換とキャストによる明示的変換）

算術式においては，次のような型の変換が自動的に行われます。

・int 未満の型は int に拡張する。

・同じサイズで符号の有無が異なる型との演算は，そのまま符号無し扱いで計算する（数値の範囲によっては正しく計算されない)。

・異なる型との演算は，大きい方の型に変換して演算する。

次の場合は自動的な型変換は起きません。

・計算結果が扱う数値範囲を越える可能性があっても変換されない。

もし暗黙の（自動的な）型変換では問題がある場合は，あらかじめ大きな型に入れ直すか，キャストという明示的な型変換を行います。キャストは次のように書きます。

```
(long) x
  |    └── 対象となる変数や式
  └─────── 変換後の型
```

```
/* リスト2−4 型の変換，キャストの例 */
#include <stdio.h>
int main (void) {
    int x = 1000, y = -30, u;
    unsigned int z = 10;
    char i = 100;
    long j, k, m;

    u = i * i;           /* int 拡張される(規格準拠時) */
    y = y / z;           /* 意図しない結果になる */
    j = x * i;           /* 結果が int 範囲を越える */
    k = (long)x * i;     /* キャストによる型変換 */

    printf(" int 未満:%d¥n", u);
    printf("  異符号:%d¥n", y);
    printf(" int 超過:%ld¥n", j);
    printf("キャスト:%ld¥n", k);
    return 0;
}
```

実行結果 2−4

```
 int 未満:10000
  異符号:6550
 int 超過:-31072
キャスト:100000
```

2.4　配列と繰返し処理

配列とは，同じ型のいくつかの変数を順番に並べたものです。

配列は，表計算のように複数データに対して同じ処理を繰り返す場合に役立ちます。

また，C 言語では文字列を char 型配列に格納して操作することになっています。

ここでは配列の宣言方法と繰返し処理について説明します。

リスト 2－5 は，家族の名前と年令，年齢差を表示する例です。

```
/* リスト 2－5 配列と繰返し処理（恐ろしいバグ） */
#include <stdio.h>
int main(void) {
    int def[3];                          /* 解説 2.4.1 */
    int age[] = {39, 24, 6};             /* 解説 2.4.2 */
    static char* const name[] = {
        "父", "母", "宗次郎"
    };
    int i;

    printf("名前    年令 差¥n");
    for (i = 0; i < 3; i++) {            /* 解説 2.4.4 */
        def[i] = age[i] - age[i + 1];    /* 解説 2.4.3 */
        printf("%-8s%3d %3d¥n",
               name[i], age[i], def[i]);
    }
    return 0;
}
```

実行結果 2－5

名前	年令	差
父	39	15
母	24	18
宗次郎	6	-9

注:ドライバ・キット同梱の main_2-5.c では、int age[]={39,24,6,15};と記述しているため実行結果 2-5 のように-9 が得られていますが、上記のままだと別の値が表示されます(大きな数値の場合は 1 行目が書き換えられてしまいます)。

2.4.1　配列の宣言

1 次元の配列（変数が 1 列に並んだもの）は，次のように宣言します。

```
int def[3];
 |   |  |
 |   |  └──── 要素数。
 |   └─────── 配列名
 └─────────── 型指定子（配列の個々の要素の型）
```

2 次元の配列（変数が縦横の表のように並んだもの）は，次のように宣言します。

```
int c[10][20];
 |  |  |   |
 |  |  |   └──── 要素数（最終的な 1 次元配列内の要素数）
 |  |  └──────── 要素数（1 次元配列の個数）
 |  └─────────── 配列名
 └────────────── 型指定子（配列の個々の要素の型）
```

同様に[]を足して行けば，多次元の配列を宣言することができます。

2.4.2　配列の初期化方法

配列に初期値を入れるには，初期値のならびを{}でくくるか文字列リテラルを使います。

例：

```
int age[3] = {39, 24, 6};
int x[3] = {5, 6, 7, 8, 9};    ・・・多すぎるとエラーになる
int x[5] = {5, 6, 7};          ・・・足りない部分は 0 となる
int x[] = {5, 6, 7};           ・・・x[3]と見なされる
char c[] = "ABC";              ・・・c[4] = {'A','B','C', 0}と同じ
        （注：文字列リテラルは最後 0 が自動的に付け足される）
char d[3] = "ABC";             ・・・d[3] = {'A','B','C'}と同じ
        （注：文字列リテラルの 0 があふれるがエラーにはならない）
```

C 言語では，文字列専用の変数がないので，文字列は char 型配列に格納して操作します。
文字列を格納する場合，最後の文字の後に 0 を付け加える，のが原則です。

また，日本語（シフト JIS コード）を格納する場合は，1 文字あたり 2 バイト必要です。

多次元の配列の初期化は，次元数に応じて{}でくくります。

```
例：　int x[2][3] = { {10,20,30}, {50,60,70} };
　　　char name[3][7] = {″父″, ″母″, ″宗次郎″};　　　・・・文字列の配列
```

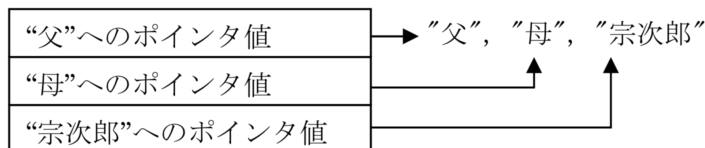
ここで，上記の配列 name には，次のように文字が格納されます（漢字は 2 バイト占有)。

父	¥0				
母	¥0				
宗	次		郎		¥0

初期値として格納されるので，配列の内容は後で書き換えることが出来ます。
しかし，もし書き換えないとしたら，使わないメモリが 8 バイト無駄になります。

無駄を押さえたい場合は，次のように文字列リテラルへのポインタ配列とします。

```
例：　char *name[3] = {″父″, ″母″, ″宗次郎″};　　　・・・ポインタ配列
```



ポインタとは変数や配列，文字列リテラルが存在する場所（メモリ上の番地=アドレス）を示すものです（詳細後述)。配列名の前にある*印でポインタ型だということを示しています。

また，初期値のままで変更されることが無い場合は， const を付けます。

```
例：　const char name[3][7] = {″父″, ″母″, ″宗次郎″};
　　　const char* name[3] = {″父″, ″母″, ″宗次郎″};
```

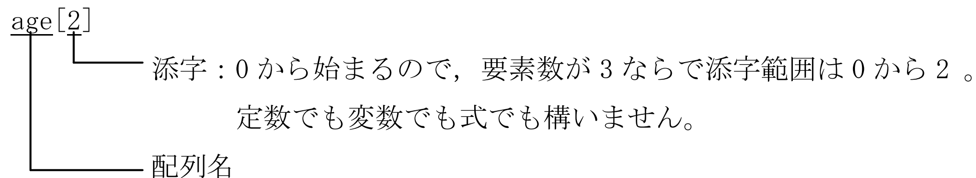
ここで，name[3][7]の方は隙間が空いた状態の配列が変更されないことを意味します。
*name[3]の方はポインタの値が変更されないことを意味します。

CC-RL では，静的変数（詳細 2.8）に const を付けると変数を ROM（値を変更できないメモリ）に配置します。一般に 1 チップ・マイコン（マイクロコントローラ）では RAM（値を変更可能なメモリ）が少ないので，なるべく ROM への配置を多くします。
ただしポインタの場合，const 修飾が char にかかると ROM へのポインタ値を持つ RAM 上の変数となってしまうため，リスト 2－5 の static char* const name[] のように const を変数名の前に持っていきます。

2.4.3　配列の使い方とポインタ

(1)個々の要素の指定

配列の要素を使うには，配列名の後ろに[]を付けて，[]内に添字（個々の変数を示す 0 から始まる識別番号）をつけます。1 次元配列ならば次のようになります。



C 言語の恐ろしいところは（柔軟性が高いとも言うが），age[3]のように宣言外を指定しても動いてしまうことです。参照ではなく，代入する方に使ったとしたら，プログラムが暴走する危険すら出てきます。

実際リスト 2-5 も age[i+1]の部分が最終的には age[3]となってしまい，不正な場所を参照しています。

CC-RL では age[3]のように宣言外を明示した場合はワーニングが出ますが、リスト 2-5 のように変数の範囲が超えてもワーニングが出ないので注意が必要です。

(2)配列をポインタで指定

一般に配列（文字列）を扱う関数では，配列（文字列）データそのものを渡すのではなく，代わりとして配列（文字列）の先頭要素へのポインタを引き渡す方式が多く取られています。

ポインタとは変数や配列の要素が存在する場所（マイコンのメモリ上の番地＝アドレス）を指し示すものです。関数へ渡すポインタ値は次のように指定できます。

①配列名は，0 番目の要素へのポインタとなる。

例：	a	(a が 1 次元配列の場合，a[0]へのポインタとなる)
	b	(b が 2 次元配列の場合，b[0][0] へのポインタとなる)
	b[n]	(b が 2 次元配列の場合，b[n][0] へのポインタとなる)

②変数または配列の要素にアドレス演算子&を付ける。

例：　&a[0]　　　(&記号は論理演算子にも使うので混同しないこと)

　　　&b[n][0]

③文字列リテラルを書く（参照する場合にしか使えない)。

例：　“ABC”　　　(文字列“ABC”へのポインタ（具体的には A を指し示すポインタ))

④上記３つに整数（定数または変数）を加算した式

例：　a+6　　　(&a[6]と等しい)

　　　“ABC”+1　(文字列”BC”へのポインタと等しい)

リスト 2-5 の printf 関数では，名前（文字列）を表示するために，ポインタ値 name[i]を引数として渡しています。

少しややこしいかもしれませんが，name は上記③の値（文字列リテラルへのポインタ）を要素とする配列になっています。

仮に name を 2 次元配列

```
char name[3][7] = {“父”,”母”,”宗次郎”};
```

と定義して，文字列そのものを配列に格納した場合も，printf 関数の引数としては name[i]を渡すことになります。

ただし，この場合の name[i]は，name[i][0]へのポインタを示します（内部的には name + i * 7 の計算結果が引き渡れます)。

逆にポインタ値に[添字]を付けると，配列と定義してなくても配列のように使えます。つまり，1 次元配列 *name[]の場合は，name[0][0]のように 2 次元配列のように使うことが出来ます。

＊＊＊　注意　＊＊＊

ポインタを使い間違えると，配列の添字を間違えるのと同様にあっさりと暴走したり気味の悪い挙動不審な現象が出るので厳重に注意します。

2.4.4　繰返し処理

同じような処理を何度も繰り返す場合は，繰返し文を使います。

繰返し文には，for 文，while 文，do 文の３つがあります。

(1)for 文

for 文は，繰返し回数をカウントできるため，一定回数の繰返し処理に最も良く使われます。次のように書きます

```
for (i = 0; i < 3; i++) def[i] = 0;
     ①      ②      ③      ④
```

①繰返し処理に入る前に１回だけ実行する式です。

②繰返し処理の先頭で毎回計算し，式の結果が非０であれば④を実行します。
　式の結果が０であれば繰返しを終了します。式の内容によっては④を一度も実行せずに終了することもあり得ます。
　この式を省略した場合は常に１と見なされて，永久に繰返しを行います。

③１回繰返し処理を終えるごとに実行する式です。

④繰返し条件が成立した時に実行する文です。色々な動作を記述したい場合は，{}で１まとめにくくります。

ここで，②では繰返し回数の大小比較の演算子（表 2－6 の<,>,<=,>=,==,!=）が良く使用されます。これらの演算結果は，条件が成立すると 1，不成立だと 0 になります。例えば i < 3 の場合，i が 0,1,2 であれば結果が 1，i が 3 だと結果が 0 になります。

また，③では繰返し回数をカウントするためにインクリメント／デクリメント演算子（表 2－6 の++，--）を良く使います。++は変数を+1 します。--は変数を-1 します。

前置（++変数，--変数）の場合は，+1，-1 されてから値が参照されます。

後置（変数++，変数--）の場合は，値が参照されてから+1，-1 されます。

③では値の参照をしてないので，どちらを使っても問題ありませんが慣例として後置を使います。

次のように値を参照する場合は，結果が異なるので注意します。

```
x = y++;    ・・・x には+1 される前の y の値が代入されます。
x = ++y;    ・・・x には+1 された後の y の値が代入されます。
```

(2)while 文

while 文は，条件式が満たされている間は繰返し処理を行います。次のように書きます。

while (Collatz > 1) x++;
①　②

①繰返し処理の先頭で毎回計算し，式の結果が非 0 であれば②を実行します。
式の結果が 0 であれば繰返しを終了します。式の内容によっては②を一度も実行せずに終了することもあり得ます。
無限に繰り返したい場合は，while(1)と記述します。

②繰返し条件が成立した時に実行する文です。色々な動作を記述したい場合は，{}で1まとめにくくります。

while 文を使った例をリスト 2−6 に示します。変数 Collatz が 1 より大きいときは計算を繰り返します。

この計算の中で使用している条件演算子（表 2−6 の?:）の動作は次のとおりです。

Collatz & 1 ? Collatz * 3 + 1 : Collatz / 2

- Collatz & 1：条件式
- Collatz * 3 + 1：条件式の結果が非 0 の時に実行する式
- Collatz / 2：条件式の結果が 0 の時に実行する式

（この例では，Collatz が奇数なら結果が 1 になる）

```
/* リスト 2−6 コラーツ予想（いつかは 1 になる） */
#include <stdio.h>
int main (void) {
    int Collatz = 9;
    int count = 0;

    while (Collatz > 1) {
        Collatz = Collatz & 1 ?
                  Collatz * 3 + 1 :
                  Collatz / 2;
        printf("%d,", Collatz);
        count++;
    }
    printf("n=%d", count);
    return 0;
}
```

実行結果 2−6

```
28,14,7,22,11,34
,17,52,26,13,40,
20,10,5,16,8,4,2
,1,n=19
```

(3)do 文

do 文も条件式が満たされている間は繰返し処理を行います。繰返しの判断が最後に行われるため，最低 1 回は繰返し処理を実行します。次のように書きます。

```
do x++; while (z > 1);
    ①          ②
```

①繰返し処理です。色々な動作を記述したい場合は，{}で 1 まとめにくくります。

②繰返し処理の最後で毎回計算し，式の結果が非 0 であれば①を実行します。

式の結果が 0 であれば繰返しを終了します。

do 文の例をリスト 2－7 に示します。

```
/* リスト 2－7 そろばん塾 */
#include <stdio.h>
int main(void) {
    char i = 0;
    const int j[] = {100,300,500,0};

    printf("願いましては，");
    do {
        j[i + 1] ?
            printf("%d 円なり，", j[i]) :
            printf("%d 円では¥?", j[i]);
    } while (j[++i]);

    return 0;
}
```

実行結果 2－7

```
願いましては，10
0 円なり，300 円な
り，500 円では?
```

2.5　キー入力に応じた動作の選択

ここから先の説明にはキー・マトリクス（第２段階のハードウエア）を使用します。実際にハードウエアを動かす場合は，1.4.2 項のキー操作を参照してください。

キー入力の例として，四則電卓プログラムをリスト 2−8 に示します。

```
/* リスト 2−8　四則電卓 */
#include <stdio.h>
int main (void) {
  char c, line[33];
  int i;
  float x, y;

  while (1) {
    while (gets(line) == NULL);          /* 解説 2.5.1 */
    i = sscanf(line, "%f%c%f%c",         /* 解説 2.5.2 */
               &x, &c, &y, &c);
    if (i > 1 && c == '#') break;  /* 解説 2.5.3, 2.5.5 */
    else if (i != 3) c = 0;

    switch (c) {                         /* 解説 2.5.4 */
      case '+':
          x += y;                        /* 解説 2.5.6 */
          break;
      case '-': x -= y;
                break;
      case '*': x *= y; break;
      case '/': x /= y; break;
      default :  c = 0; break;
    }
    if (c) printf("=%g¥n", x);           /* 解説 2.5.3 */
    else   printf("error¥n");
  }
  return 0;
}
```

実行結果 2−8
（赤字は入力例）

```
123+567◿
=690
0.001/3e8◿
=3.33333e-12
```

2.5.1　キーから1行取得（gets 関数）

gets 関数はキーから1行の文字列を取得する関数です。

1行とは改行コード（¥n，本ボードでは Enter キー）までの文字列のことです。

改行コード自体は捨てられてしまいますが，文字列の終わりを示す0は付きます。

gets 関数の引数と戻り値は次のとおりです。

引数　：入力した文字列を格納する char 型配列へのポインタを指定します。
配列サイズは，文字列の終わりの0を含めて十分格納できるように決めます。

戻り値：成功すると引数と同じポインタ値を返します。
失敗すると（キー入力のない状態だと）NULL を返します。

ここで，NULL とは0（ゼロ）のことです。

ポインタ型では，0は「メモリ上のどこも指してない」という特別な意味を持ちます。

逆に言うと，もし仮にメモリ上の0番地を使う場合であっても0を指定できません。

元々ポインタに対して番地を直接数値で指定することは原則禁止で，もし 0 番地のポインタ値を取得したいなら，0番地に配置された変数名にアドレス演算子を付けます。

（注：CC-RL では0番地に変数が配置されることはありません）

2.5.2　書式付き入力（sscanf 関数）

入力された文字列を分解して，①数値，②演算記号，③数値，④その他の文字（誤入力判定用）を得るために，リスト2－8では sscanf 関数を使っています。

sscanf 関数の引数と戻り値は次の通りです。

引数1：入力文字列へのポインタ。

引数2：入力書式文字列（詳細後述）。

引数3以降：変換結果を格納する変数へのポインタ。ポインタなので変数名に&をつけます。&を忘れてもコンパイル・エラーにならないので間違えやすいです。

戻り値：int 型。入力文字列が空の場合は EOF（-1）を返す。
それ以外の場合は，変換結果を格納できた数を返す。

注1：sscanf ではなく scanf 関数を使うとキーから入力した文字列を直接解析してくれますが，1行単位の解析ではないため，間違った入力をすると次の解析が行の途中から始まってしまい，何度も error 表示が出ます。

入力書式文字列は，入力文字列を先頭から順番に解析する場合に，種類や桁数を対応付けするもので，大きくは次の 3 種類があります。

・空白文字：入力文字列中の空白，タブ，改行を読み飛ばす指示。
解析位置において，それらが 1 つも検出できなければ変換終了。

・%で始まらないその他の文字：
解析位置において，その通りの文字が検出できない場合は変換終了。

・%で始まる文字の組合せ：
数値や文字への変換方法の指定。次のように指定。

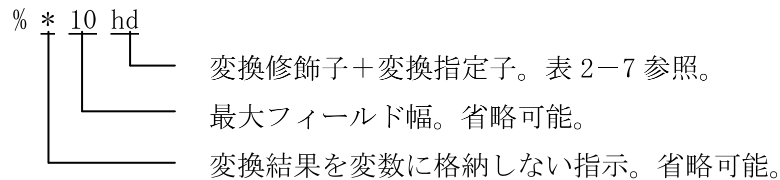


表 2－7　代表的な変換修飾子＋変換指定子

記号	内容
d，i	int 型へ変換。d は 10 進表記のみ。i は 0x または 0X をつけた 16 進表記も可能。
ld，li	long 型へ変換。表記は同上。
hd，hi	short 型へ変換。表記は同上。
u，x	unsigned int 型へ変換。u は 10 進表記のみ。x は 16 進表記のみで，0x または 0X はあっても無くても良い。
lu，lx	unsigned long 型へ変換。表記は同上。
hu，hx	unsigned short 型へ変換。表記は同上。
f	float 型へ変換。
lf	double 型へ変換。
c	1 文字を char 型変数へ格納。最大フィールド幅に 2 以上を指定した場合は char 型配列へ格納するが，文字列の終わりを示す 0 は付加されない。
s	文字列を char 型配列に格納。空白または改行(¥n)の直前までが格納される。文字列の終わりを示す 0 も付加される。

2.5.3 if 文と break 文

(1)if 文

if 文は，条件が成立するかしないかで，実行する動作を選択する場合に使用します。

書き方：

```
if（式 1）文 1
else if（式 2）文 2
else 文 3
```

式 1 の値が非 0（ゼロ）なら文 1 を実行します。

式 1 の値が 0 なら，式 2 が 0 かどうか判定され，非 0 なら文 2 を実行します。

式 2 が 0 の場合は文 3 を実行します。

else if の行はいくつあっても良いですし，なくても良いです。

else の行はなくても良いですが，書く場合は対応する if 文に対して 1 回だけです。

文 1～3 は，単独の文の他に{と}で囲んだ複合文を書くことも出来ます。

式の中で良く使われる論理演算子（&&や||）については 2.5.5 を参照してください。

(2)break 文

break 文は，前述の繰返し文（for，while，do）や後述の switch 文の中から強制的に抜け出すための文です。

2.5.4 switch 文

リスト 2－8 では加減乗除の記号の判定を switch 文で行っています。switch 文は，式の値を色々な定数と比較して，一致する定数ごとに異なる処理を実行することができます。

書き方：

```
switch（式）{
  case 定数 1: 文
        :
  case 定数 n: 文
  default: 文
}
```

ここで，default とは case で指定した定数 1～定数 n 以外の場合を示します。

各 case の後ろには複数の文を書くことができ，通常は break 文で終えます。

break 文が無い場合は，その下の case も実行されてしまいます。

逆に複数の定数に対して同じ処理が必要であれば，break 文をつけないでおきます。

2.5.5　論理演算式

if 文や繰返し文では，論理演算（かつ，または）を良く使用します。論理演算子には，ビットごとの論理演算子という似たものがありますが，表 2−8 に示すとおり，動作が異なります。

```
/* リスト 2−9 論理演算子の動作の違い */
#include <stdio.h>
int main (void) {
    int i = 25, j = -26, k;

    k = i || (j = 0);             /* j = 0 は実行されない */
    k = !(j + 26 && i++);         /* i++は実行されない */
    printf(“&と&&=%2d:%2d¥n”
           “|と||=%2d:%2d¥n”
           “~と! =%2d:%2d¥n” “i=%d, j=%d¥n”,
           i & j, i && j, i | j, i || j,
           ~k, !k, i, j);
    return 0;
}
```

実行結果 2−9

```
&と&&= 0: 1
|と||=-1: 1
~と! =-2: 0
i=25, j=-26
```

表 2−8　論理演算子，ビットごとの論理演算子の動作詳細

記号	演算結果
&&	AND 演算子。&&の左側の値が 0 なら結果を 0 とし，右側は評価しない。 左側が非 0 なら右側を評価し，それが 0 なら結果を 0，非 0 なら結果を 1 とする。
||	OR 演算子。||の左側の値が非 0 なら結果を 1 とし，右側は評価しない。 左側が 0 なら右側を評価し，それが 0 なら結果を 0，非 0 なら結果を 1 とする。
!	否定演算子。!の右側の値が 0 なら結果を 1 とし，非 0 なら結果を 0 とする。
&	ビットごとの AND 演算子。同じビット位置ごとに値を比較し，両者が共に 1 なら結果の当該位置を 1 とする。そうでなければ当該位置を 0 とする。
|	ビットごとの OR 演算子。同じビット位置ごとに値を比較し，両者のどちらかが 1 なら結果の当該位置を 1 とする。そうでなければ当該位置を 0 とする。
^	ビットごとの排他 OR 演算子。同じビット位置ごとに値を比較し，両者が共に 1 か共に 0 なら結果の当該位置を 0 とする。そうでなければ当該位置を 1 とする。
~	ビットごとに 0 を 1 に，1 を 0 に変える。

論理演算におけるその他の注意

・int 未満の型は整数拡張されるので，特に~は注意します。

例：unsigned char i = 255;

→ 変数 i のビットパターンは，11111111 だから~i は 0 になりそうですが，

int に拡張されて，0000000011111111 となるから~i は 0 になりません。

char j = -1;

→ 変数 j は int 拡張されるとビットパターンは，1111111111111111 になります。

従って，~j は 0 になります。

2.5.6　代入と複合代入

リスト 2－8 では，四則演算の部分に複合代入演算子（+=など）が使われています。

複合代入とは，右辺で使った変数に対して結果を格納する場合の短縮表記です。

例：　x = x + y; → x += y;

書く手間がちょっと省ける程度に見えますが，生成されるオブジェクト・コードが良くなる場合がしばしばあります。

x = x + y を厳密に解釈すると，x + y を先に計算して目に見えないテンポラリに格納し，その値を x に代入するということになります。

一方，x += y は y を直接 x に加算するということになります。分りにくいかもしれませんが，マイコンには複合代入用の機械語命令が備わっているから，それを活用した記述だと理解してください。

必ず良くなるわけでもありませんが，悪くなることは無いでしょう。

また代入（=記号）は，どこで書かれていても常に代入を意味します。例えば，

```
if (k = 1)
```

などと書くと，この if 文の式は常に 1 となって後ろの文が必ず実行されます。

k の値を 1 以外でテストすればすぐ気がつきますが，万一 k がもともと 1 の場合しかテストしてなかったら，恐ろしいことに「正常に動いている」ように見えてしまいます。

2.6　関数電卓

次のリスト 2-10 は，リスト 2－8 の四則電卓に数学関数を付け加えたものです。ただし，例として三角関数の一部のみを付け加えています。また四則演算は記述を省略しています。

三角関数は角度で入力，表示しています（内部でラジアンに変換)。

```
/* リスト 2－10 関数電卓 */
#include <stdio.h>
#include <math.h>                           /* 解説 2.6.1 */

#define INPUT_LIMIT      32                 /* 解説 2.6.2 */
#define SIN              's'
#define COS              'c'
#define ARCTAN           'a'
#define PI               3.14159
#define rad(a)           ((a) * PI / 180)
#define deg(a)           ((a) * 180 / PI)

int main (void) {
    char c, line[INPUT_LIMIT + 1];
    float x, y;

    while (1) {
        while (gets(line) == NULL);
        if (sscanf(line, "%f%c%f%c",
                &x, &c, &y, &c) > 3) c = 0;
        switch (c) {
          case SIN: x = sin(rad(x)); break;
          case COS: x = cos(rad(x)); break;
          case ARCTAN: x = deg(atan2(x, y)); break;
          default : c = 0;  break;
        }
        if (c) printf("=%g¥n", x);
        else printf("error¥n");
    }
    return 0;
}
```

実行結果 2－10

(赤字は入力例)

```
30s◿
=0.5
10a10◿
=45
```

2.6.1　数学関数

主な数学関数を表 2－9 に示します。次のヘッダをインクルードして使います。

math.h

数学関数はもともと科学技術計算用のため，基本的な関数名は精度の高い double 型になっています。関数名の後ろに f がつくと float 型になります。

補足：CA78K0R では，double 型も float 型と同じ精度になっており、実行結果 2-10 は、それぞれ 0.4999996 45.00003 となっていました。CC-RL では sinf,atan2f に変更しても 0.5，45 と表示されます。オブジェクトサイズが変わらないので、内部的に double が呼ばれている可能性があります(実行時間の差は未確認)

表 2－9　主な数学関数（基本型は double 型，後に f が付く関数名は float 型用）

関数名	内容
sin，sinf	sin 関数。引数はラジアン単位。
cos，cosf	cos 関数。引数はラジアン単位。
tan，tanf	tan 関数。引数はラジアン単位。
asin，asinf	arc-sin 関数。戻り値は-π/2～π/2。
acon，acosf	arc-cos 関数。戻り値は 0～π。
atan，atanf	arc-tan 関数。戻り値は-π/2～π/2。
atan2，atan2f	引数 y，x に対して arc-tan(y/x)を計算。戻り値は-π～π。
sinh，sinhf	sinh 関数。
cosh，coshf	cosh 関数。
tanh，tanhf	tanh 関数。
exp，expf	指数関数。
log，logf	自然対数。
log10，log10f	10 を底とする対数。
pow，powf	引数 x，y に対して x の y 乗を求める。
sqrt，sqrtf	平方根。
fabs，fabsf	絶対値。
fmod，fmodf	引数 x，y に対して x/y の剰余を求める。
floor，floorf	引数以下の最大の整数を求める。
ceil，ceilf	引数以上の最小の整数を求める。

2.6.2 #define 文

リスト 2-10 では，定数に意味の分かる名前をつけたり，良く使う定数や式の記述を簡素化するために#define 文を使っています。

(1)単純置き換え

プログラム中に現れる単語や変数名などを単純に置き換える使い方です。リスト 2－10 では，文字 s が sin 関数の意味だと言うことを明確化するために SIN と名付けています。

書き方：

```
#define  PXY  (x + y)
```

- (x + y)：置き換わる値。式などは優先順序を守る()を付ける。
- PXY：マクロ名（置き換え対象）

(2)関数形式

関数のように引数を付けることができる使い方です。リスト 2－10 では，角度←→ラジアン変換の部分をマクロにしています。

書き方：

```
#define deg(a) ((a) * 180 / 3.14)
```

- ((a) * 180 / 3.14)：置き換わる値。仮引数名は実際の引数に置換される。引数の優先順序を守る()と，置き換えた式の優先順序を守る()を付ける。
- deg(a)：マクロ名(仮引数名の並び)

使用例：

```
i = j % deg(x + y); → i = j % ((x + y) * 180 / 3.14);
```

もしかっこがないと → i = j % x + y * 180 / 3.14; となって計算順序が狂う。

(3)複数行での記述方法

#define 文は，定義途中での単純な改行は許されていません。

改行する場合は，実質 1 行と見なしてくれる¥改行を行います。

¥改行は，¥の直後で改行し，次の行の先頭から書き始めます。

次の行でインデント（字下げ）したい場合は，空白が許されている場所で¥改行します。

例：

```
    #define ultra_super_long_name_function_macro( first_coefficient, second_c¥
oefficient)    ((unsigned short)( first_coefficient) << 4 | ¥
               (unsigned char)( second_coefficient) + 0x0f )
```

2.7 文字列の操作と構造体

ここでは，文字列の操作関数と構造体という変数のかたまりについて説明します。

リスト 2-11 は，弁当の売り上げ集計のプログラム名です。

弁当の名前の略号（かま，かに，ま）と個数を入力すると，最後に売り上げの集計結果を表示します。

```
/* リスト 2－11 弁当の売上集計 */
#include <stdio.h>
#include <string.h>                    /* 解説 2.7.1 */
int main(void) {
    char line[33], moji[5];
    int i, j;
    long total;
    typedef struct {                   /* 解説 2.7.2 */
        char    index[5];
        char    renketu[7];
        int     tanka;
        int     urisuu;
    } BENTO;

    BENTO table[3] = {
        {"かま", "めし", 600},
        {"かに", "ずし", 800},
        {"ま", "くの内", 700}
    };

    while (1) {
        while (gets(line) == NULL);
        if (sscanf(line, "%s %d", moji, &j) < 2) break;
        for(i = 0; i < 3; i++)
            if (!strcmp(moji,              /* 解説 2.7.1 */
                table[i].index)) break;    /* 解説 2.7.3 */
        if (i < 3) table[i].urisuu += j;
    }

    for (i = 0, total = 0; i < 3; i++) {  /* 解説 2.7.4 */
        strcpy(line, table[i].index);
        strcat(line, table[i].renketu);
        printf("%8s ¥¥%3dx%2d",
                line, table[i].tanka, table[i].urisuu);
        total += (long)table[i].tanka * table[i].urisuu;
    }
    printf("合計 %ld 円¥n", total);
    return 0;
}
```

実行結果 2－11

(赤字は入力例)

```
かに 3◿
かま 6◿
ま 2◿
かに 4◿
ま 1◿
かに 5◿
かま 2◿
◿
かまめし ¥600x 8
かにずし ¥800x12
まくの内 ¥700x 3
合計 16500 円
```

2.7.1　文字列の操作関数

文字列の複写・連結・比較，などの代表的な文字列操作関数を表 2－10 に示します。次のヘッダをインクルードして使います。

string.h

いずれも文字列へのポインタ（2.4.3(2)参照）を使用します。

リスト 2－11 では，入力された弁当の略号と table 構造体に格納された略号を strcmp 関数で比較し，一致したところの要素番号を使用しています。また弁当の略号と続く文字から弁当名を組み立てるために strcpy 関数，strcat 関数を使用しています。

その他の文字列操作関数については，ヘルプのライブラリ関数の説明を見てください。

表 2－10　代表的な文字列操作関数

関数名	内容
strcpy	文字列のコピー 引数 1：格納される文字列へのポインタ。十分格納できるサイズであること。 引数 2：コピーする文字列へのポインタ。文字列リテラルでもよい。 戻り値：引数 1 がそのまま戻される。
strcat	文字列の追加 引数 1：追加される文字列へのポインタ。十分格納できるサイズであること。 引数 2：追加する文字列へのポインタ。文字列リテラルでもよい。 戻り値：引数 1 がそのまま戻される。
strcmp	文字列の比較 引数 1,2：比較する文字列へのポインタ。文字列リテラルでもよい。 戻り値：等しければ 0。一致しない場合は，最初の不一致文字のコード差（引数 1 側の文字－引数 2 側の文字）。
strlen	文字列の長さを調べる 引数 1：文字列へのポインタ。文字列リテラルでも良い。 戻り値：文字列の長さ（¥0 を含まない）。
strchr	文字列中から指定文字の位置を得る 引数 1：走査する文字列へのポインタ。文字列リテラルでも良い。 引数 2：指定文字のコード。文字定数でも良い。 戻り値：指定文字を見つけた位置へのポインタ。指定文字がなければヌル。
strstr	文字列中から指定文字列の位置を得る 引数 1：走査する文字列へのポインタ。文字列リテラルでも良い。 引数 2：指定文字列。文字列リテラルでも良い。 戻り値：指定文字列を見つけた位置へのポインタ。見つからなければヌル。

2.7.2　構造体（struct）と typedef

リスト 2－11 では，弁当略号や単価など異なる型を集めた構造体と呼ばれる変数を使用しています。

(1)構造体（struct）

構造体は，色々な型の変数を一まとめにできます。基本的には次のように宣言します。

```
struct tagname{ int a; char c; } x, y;
```

- x, y：左記の構造を持つ変数名の並び。ここを省略した場合はタグ名のみの宣言となる。
- int a; char c;：メンバ（構成要素）の並び。
- tagname：タグ名。メンバの並びに対する名前。
- struct～}までが構造体指定子。

・タグ名はメンバの並びを他で引用するのに必要です。引用しなければ省略可能です。

　省略例：　struct { int a; char c; } x, y;

・他で宣言するタグ名を引用する場合はメンバの並びは省略します。

　引用例：　struct tagname x, y;

(2)typedef

typedef は，型指定子や修飾子に別の名前を付けることが出来ます。特に構造体指定子のように長い指定子に短い名前をつけることが出来ます。

宣言例：

```
typedef struct { int a; char c; } KOZO;
typedef unsigned short US, UI16;
```

- US, UI16：新たな型指定子名（複数の名前付け可能）
- unsigned short：型指定子や修飾子

使用例（変数定義）：　US x;

使用例（キャスト）：　x = (US)a * b;

(3)構造体の初期化

構造体の初期化も配列同様に{}のなかに初期値を並べます。

初期値がメンバ数に対して不足する場合は，0 で補われます。多すぎるとエラーです。

構造体や配列が多重化されている場合は，その分{}でくくります。

2.7.3　構造体の使い方

(1)メンバの直接指定

構造体変数名の後ろにドット（.）を付けてメンバ名を続けると，メンバを変数として操作できます。

リスト2－11のように構造体の配列の場合は，添字の後に.メンバ名を続けます。

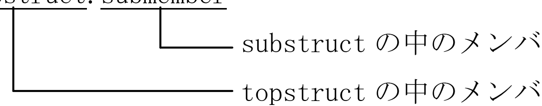
例：　table[i].urisuu

さらにメンバが配列の場合で，その要素を指定したければメンバに添字をつけます。

例：　table[i].index[0]

もしメンバが構造体で，そのメンバを指定したければ，.メンバを追加します。

例：　topstruct.substruct.submember



(2)全体の指定

構造体変数名を単独で使うと，メンバ全てのデータをひとまとめにした実体のある変数として扱うことを意味します。

巨大な構造体を定義すると扱いが大変になるので（特に関数の引数や戻り値），その場合はポインタで操作するのが楽になります。配列名はポインタ値になりましたが，構造体変数名はポインタ値にはなりませんので，アドレス演算子&を付けて

&構造体変数名

のようにポインタ値を指定します。

なおリスト2－11の変数名tableは，構造体の配列名なので，tableをポインタ値として扱えます。

(3)ポインタからのメンバ指定

構造体変数へのポインタからメンバを指定するには，

ポインタ->メンバ名

のように書きます。直接指定（.）と間接指定（->）の関係は次のようになっています。

構造体名.メンバ名　　←等価→　　(&構造体名)->メンバ名

(*構造体へのポインタ). メンバ名　　←等価→　　構造体へのポインタ->メンバ名

ただしここの*記号は間接参照（ポインタで指し示される位置の中身）を意味します。

2.7.4　カンマ演算子（コンマ演算子）の使いどころ

リスト 2－11 の for 文で，最初に i と total という２つの変数を初期化しています。

このように，通常１つの式しか書けないところに実質 2 つ以上の式を書くのにカンマ演算子が使えます。

書き方：　　式 1，式 2，・・・，式 n

動作は，式 1 から順番に計算し，最後の式 n の結果を全体の計算結果とします。ただし演算子の優先順位が最低であるので，もし全体の結果を変数に格納するには，

x = (式 1，式 2，・・・，式 n)；

のようにカッコでくくります。また，関数の引数の部分で使う場合も引数の区切りと区別するためにカッコでくくります。

y = f(引数 1，(式 1，式 2，・・・，式 n)，引数 3)；

一般的には，カンマ演算子を使わなくても，対象となる文の前後に分けて書いたり，{} でくくれる場合がほとんどです。例外として，関数形式のマクロ定義（#define 文，2.6.2(2) 参照）の場合に，順序立てた計算の最後の結果を返すためには重宝します。

2.8　関数の使い方と変数の通用範囲

次のリスト 2－12 は，弁当の売上集計を関数を使って書いた例です。

```
/* リスト 2－12 弁当の売上集計(関数版) */
#include <stdio.h>
#include <string.h>

int  index_search(char *c);                                    /* 解説 2.8.1 */
char *sum(int i);

#define SHURUI  3
static char* const index[SHURUI] = {"かま","かに","ま"};        /* 解説 2.8.3 */
static int         urisuu[SHURUI];
char    line[33];

int main(void) {
    char moji[5];
    int i, j;

    while (1) {
        while (gets(line) == NULL);
        if (sscanf(line, "%s %d", moji, &j) < 2) break;
        if ((i = index_search(moji)) != EOF) urisuu[i] += j;    /* 解説 2.8.2 */
    }

    for (i = 0; i < SHURUI; i++) printf("%s", sum(i));
    printf("%s", sum(EOF));
    return 0;
}

int index_search(char * c) {
    int i;
    for (i = 0; i < SHURUI; i++) if (!strcmp(index[i], c)) break;
    if (i < SHURUI)     return i;
    else                return EOF;
}

char *sum(int i) {
    static long total;
    char   *moji[SHURUI] = {"めし","ずし","くの内"};
    int    tanka[SHURUI] = {600, 800, 700};

    if (i == EOF) sprintf(line, "合計 ¥¥%ld¥n", total);
    else {
        sprintf(line, "%s%s ¥¥%3dx%2d", index[i], moji[i], tanka[i], urisuu[i]);
        total += (long)tanka[i] * urisuu[i];
    }
    return line;
}
```

2.8.1　関数プロトタイプ宣言

2.1 で説明しましたが，関数を使う前にプロトタイプ宣言が必要です。標準ライブラリ関数のプロトタイプ宣言はヘッダ・ファイルに書かれていますが，自分で作る関数は自分で書くしかありません。

書き方：int index_search(char *c);

char *c → 引数の型の並び。引数の名前は必須ではないが，普通は関数定義の書き始めと一致させる。引数がなければ void と書く。

int → 戻り値の型。戻り値がなければ void と書く。

2.8.2　関数の呼び出し方法

関数の呼び出しは，式の書けるところであれば，どこにでも書けます。

sum(x)

x → 式の値が関数へ引き渡されます。変数そのものが渡されるわけでありません。

関数名のあとのカッコは必須です（かっこがあると関数呼び出しと見なされます)。

特に引数の無い関数でカッコを忘れると，動作しないにもかかわらずエラーにはなりません。

例：

fun();　・・・fun が引数無し関数なら，正常な関数呼び出しと見なされます。

fun;　・・・fun が関数であっても実行されません。
CC-RL では、「式が作用しない」というワーニングが出ます。
エラーにはならないので、見落とさないように注意が必要です。

x = fun;　・・・fun が関数であっても実行されません。
x の型が関数へのポインタ型でなければワーニングがでます。
エラーにはならないので、見落とさないように注意が必要です。

なお，引数無しの関数呼び出しで，カッコ内に void と書くことは出来ません。

```
func(/* void */);
```

のようにコメントは書けます。

ただし#define による関数形式のマクロでは/* void */も書けません。

2.8.3 変数の通用範囲

変数は，宣言する位置（関数の内か外か）と記憶クラス指定子（static または extern）の有無で，名前の通用する範囲や初期化の条件が異なります（表 2－11 参照)。

リスト 2－12 の例では次のようになっています。

(1)関数外で宣言（このファイル内の全ての関数で使用可能)。

```
static char* const index[]={};    ・・・static 指定のため他ファイルでは使えません。
static int urisuu[SHURUI];        ・・・static 指定のため他ファイルでは使えません。
                                        初期値の指定がないので 0 で初期化されます。
char     line[33];                ・・・他のファイルで，extern char line[];と宣言
                                        すると，この変数を使用できます。
```

(2)main 関数内で宣言（関数内でのみ有効)

```
    char moji[5];  int  i, j;     ・・・初期値の指定がないので初期値は不定です。
```

(3)index_search 関数内で宣言（関数内でのみ有効)

```
    int i;                        ・・・main 関数の i とは別扱いです。index_search
                                        関数が呼び出されたときだけ存在します。
```

(4) sum 関数内で宣言（関数内でのみ有効)

```
    static long total;            ・・・初期値 0 で初期化されます。
                                        他関数からは見えませんが，常時存在します。

    char    *moji[] = {};  ┐
    int     tanka[] = {};  ┘      ・・・この２つは，sum 関数が呼び出されたときだけ
                                        存在し，呼び出されるごとに初期値の代入が行
                                        なわれます。
```

最後の二つは，関数が呼び出されるごとに初期化が発生するので非効率な例です。一般的には，関数外で定義するか static 修飾して初期化を 1 回で済ませます。

また，一般に関数内で扱える自動変数（記憶クラス指定無しの変数）の合計サイズには制限があるので，大きな配列を宣言する場合は関数外で宣言します。

CC-RL では，静的変数を const 修飾して ROM に配置できます。次のように書きます。

```
        static char* const moji[] = {};    (const char* は NG。2.4.2 項参照)
        static int const tanka[] = {};     (こちらは const int でも同じ)
```

表2−11

<table>
<tr><th>宣言位置</th><th>通用範囲</th><th>記憶クラス指定子</th><th>存在期間</th><th>初期化</th></tr>
<tr><td rowspan="3">関数内</td><td rowspan="3">・宣言した関数内だけで使用できる。
・関数外で同名の変数が宣言されていても，関数内の宣言が優先される。</td><td>無し（または auto）</td><td>関数が呼び出されてから終了するまでの間。</td><td>・初期値の指定があれば，関数が呼び出されるつど初期化される。
・初期値の指定が無い場合は不定となる。</td></tr>
<tr><td>static</td><td>常時</td><td rowspan="5">・プログラム起動時に 1 回だけ初期化される。
・初期値の指定が無い場合は 0 で初期化される。</td></tr>
<tr><td>extern</td><td>常時（実体は別の場所で定義）</td></tr>
<tr><td rowspan="3">関数外</td><td rowspan="3">・宣言した位置以降のファイル内で使用できる。
・ただし同名の変数が宣言されている関数内では，関数内の宣言が優先される。
・記憶クラス指定子が無い場合は，他のファイルでも使用できる（使用する側には extern 指定した宣言が必要）。
・static 指定した場合は，他のファイルでは使用できない（別の変数として扱われる）。</td><td>無し</td><td rowspan="2">常時</td></tr>
<tr><td>static</td></tr>
<tr><td>extern</td><td>常時（実体は別の場所で定義）</td></tr>
</table>

注：常時存在する変数を静的変数，関数が呼ばれたときだけ存在する変数を自動変数と呼びます。
関数が終了すると自動変数に割り当てられていたメモリ領域は解放され，別の関数で使用できるようになります。

2.9　補足

2.9.1　ポインタの加減算

ポインタを変数などが存在する場所（メモリ上の番地=アドレス）と説明しましたが，

ポインタの加減算の操作は，配列の添え字の加減算と同じであり，物理メモリ・アドレスの加減算とは異なります。

例:

```
long x[10], *p;
p = &x[0];
```

であるとき

p + 1　は　&x[1] に等しい。

しかしｐの物理アドレスに対してｐ + 1の物理アドレスは，+4 されています（long のサイズは4バイトのため)。

もしなんらかの理由でポインタを物理アドレスに変換して管理する場合は，この点を注意します。

特に構造体の場合は，メンバの合計サイズと構造体のサイズがしばしば異なります。

例：

```
typedef struct {long x; char y; long z;} A;
```

とした場合，A のサイズは多くの場合 9 バイトになりません。

long 型変数のメモリ上の配置が制限されるためで，例えば long を 4 バイト単位のアドレスにしか配置できない場合は，char y の後ろに 3 バイトのダミーが配置されることになります。

また int 型のようにマイコンによりサイズが変わるものもあります。

型のサイズを調べるには，sizeof 演算子を使用して，

sizeof(int)

sizeof(struct {long x; char y; long z;})

sizeof(A)　　　・・・A が typedef で定義されている場合

のように調べます。

2.9.2　用語の補足

用語“戻り値”は，JIS 規格では“返却値”とされています。しかし，JIS 規格が出てきたのは，世の中に C 言語が普及した後だったので，JIS 規格外の用語も一般に通用しています。

表 2－1　用語の対比

本書	「プログラミング言語 C」K&R 著，石田晴久訳	CC78K0R ヘルプ	JIS X 3010	ISO/IEC 9899
戻り値，値を返す	戻り値，値を返す	返り値	返却値	return value
変数	(大部分は)変数	オブジェクト	オブジェクト	object
通用範囲	通用範囲	有効範囲	有効範囲	scope
存在期間	存在時間	ライフタイム	生存期間	lifetime
ヌル	null	ヌル	ナル	null
等値演算子	等値演算子	等値演算子	等価演算子	(equality operator)

注:本来オブジェクトは，本書で言う変数よりも広い概念。

参考：JIS X 3010 の閲覧方法（2010 年 5 月時点）

日本工業標準調査会　http://www.jisc.go.jp/index.html

→「JIS 検索」→規格番号欄に“X3010”と入力