TECHI Vol.58

信頼性＆再利用性を高める C言語プログラミング

あいまいな仕様，環境依存 ……つまずきやすい場所を徹底解説

整数型
浮動小数点型
配列・ポインタ活用
演算子/最適化
スタック

鹿取 祐二●著

高信頼性
ソフトウェア

CQ出版社

第1章
整数型

第1章では，整数型の文法に対する問題点や内容を解説します．移植性を損なう原因のほとんどは，ここで紹介する内容にあります．この内容を理解し，対策を実践できれば，移植性の高いプログラムにかなり近づけると考えて良いでしょう．

1．値の範囲

整数型の値の範囲は曖昧である

● 文法

▶型における値の範囲とサイズ

表1に示すのは，C99で規定されている整数型の値の範囲とサイズです．表内だけでなく，表の欄外に記載した次の2点も重要です．

- 同じ型の符号付きと符号なしでサイズが異なってはならない
- 表現可能な値の範囲はshort ≦ int ≦ long ≦ long longでなければならない

表中の［ ］は，省略可能を意味します．short型は正確に記述するならsigned short intですが，signedとintは省略可能なので単にshortと記述しても同じ翻訳結果になります．整数型には，真理値である0と1の2つの値をとる_Bool型もありますが，文法に対する大きな問題点がないので，ここでは省略しました．

▶曖昧なポイント

値の範囲やサイズが確定している整数型は，ほとんどありません．サイズが確定しているのはchar型関係だけで，それ以外はどれも「少なくとも＊＊＊」や「＊＊＊ビット以上」と曖昧です．

組み込み用途に使われるマイコンや処理系（コンパイラ）に的を絞れば，業界標準の列が示すように，サイズはほぼ確定します．それでもint型とunsigned int型だけはサイズが決まりません．8ビットや16ビット・マイコンなら，ほとんどの場合でint型は16ビットとなります．32ビット以上のマイコンなら，int型は32ビットとなります．

このような曖昧な文法で，どうすれば移植性を保て

表1　整数型の値の範囲とサイズ
C99で規定されている内容．「少なくとも＊＊＊」や「＊＊＊ ビット以上」などの曖昧な表現が多い

型	値の範囲	サイズ	
		C言語の規定	業界標準
char	signed char または unsigned char	8ビット	8ビット
signed char	−127 ～ +127	8ビット	8ビット
unsigned char	0 ～ +255	8ビット	8ビット
[signed] short [int]	**少なくとも** −32,767 ～ +32,767	16ビット**以上**	16ビット
unsigned short [int]	**少なくとも** 0 ～ +65,535	16ビット**以上**	16ビット
[signed] int	**少なくとも** −32,767 ～ +32,767	16ビット**以上**	**16 または 32ビット**
unsigned int	**少なくとも** 0 ～ +65,535	16ビット**以上**	**16 または 32ビット**
[signed] long [int]	**少なくとも** −2,147,483,647 ～ +2,147,483,647	32ビット**以上**	32ビット
unsigned long [int]	**少なくとも** 0 ～ +4,294,967,295	32ビット**以上**	32ビット
[signed] long long [int]	**少なくとも** −9,223,372,036,854,775,807 ～ +9,223,372,036,854,775,807	64ビット**以上**	64ビット
unsigned long long [int]	**少なくとも** 0 ～ +18,446,744,073,709,551,615	64ビット**以上**	64ビット

同じ型の符号付きと符号なしでサイズが異なってはならない
short ≦ int ≦ long ≦ long long でなければならない
［ ］は省略可能を意味する
long long型はC99で新たに追加された整数型である

リスト1　C言語の曖昧な文法の例（char型を使った数値計算）
このプログラムは，実行環境によって結果が変わってしまう．原因は，char型の符号の有無が文法で決まっていないためだ

```
char a;
                    RL78とRXのコンパイラでは真になる
    a = 200;
    if( a == 200 )
        printf("char is unsigned char."); // RL78,RX
    else
        printf("char is signed char.");   // H8,SH
                    H8とSHのコンパイラでは偽になる
```

るのでしょうか．対策はいろいろありますが，まずは次の2つに注目します．

● 方法1：char型は数値計算に使わない

▶符号の取り扱いが決まっていないchar型

1つ目は，単なるchar型の符号です．

char型以外の整数型は，符号なしを意味するunsignedの指定がなければ，全て符号付きのsignedになります．**表1**でも，char型関係以外は全てsignedに省略可能の［ ］が付いていますが，単なるchar型にはその記載がありません．signed char型とunsigned char型は，別々の行に記載されています．

値の範囲も他の型の記載とは異なり，具体的な数値ではなく「signed charまたはunsigned char」と記載されています．

これは単なるchar型は符号が決まっていないことを意味します．char型は，signed char型の符号付きとunsigned char型の符号なしのどちらで扱っても良く，その選択は処理系に任されています．この理由は，char型が数値計算用ではなく，文字コード格納用の型であるためです．

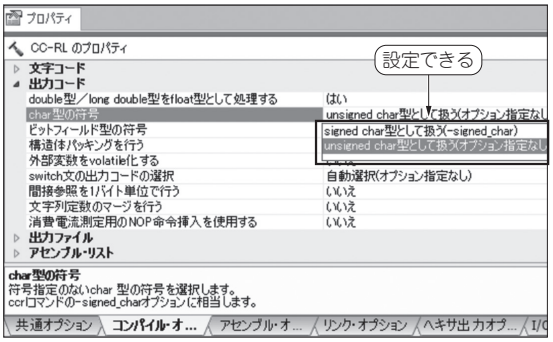


図1　処理系によってはオプションでchar型の振る舞いを変更できる場合がある
RL78用のコンパイラであるCC-RLの例．ただし，オプションによって結果が変わるような記述は好ましくない

▶数値計算すると処理系によって実行結果が変わる

char型を使って数値計算すると，移植性が失われる可能性があります．signed char型とunsigned char型は，どちらも0～127の範囲であれば表現できるので，その範囲を超えた値を扱うと移植性が失われます．

リスト1では，char型の変数に200を代入しています．char型をunsigned char型として扱う処理系なら200を代入できますが，signed char型として扱う処理系は最大値の127を超えているため代入できません．代入できない値は捨てられるので，処理系依存となります．結果，後続のif文で変数の値を200と比較した場合，処理系によって評価結果が真になったり偽になったりします．例えば，RL78とRXのコンパイラでは真，SHとH8のコンパイラでは偽になります．

以上のことから，char型を数値計算に使ってはいけません．数値計算用に8ビットの変数を使いたいときは，signed char型またはunsigned char型を使い，符号を明確に指定しましょう．

なお，MISRA-Cにも同様のルールが存在し，考え方は同じです．

▶文字コードを扱うときはchar型を使う

逆説的な言い方をすれば，符号を明確にしたsigned char型やunsigned char型で文字コード（文字定数や文字列）を扱うのも移植性を損なう結果になります．

例えば，文字列の長さを求めるstrlen関数ですが，プロトタイプは次の通りです．

```
size_t  strlen(const char *str);
```

プロトタイプでは引数のポインタが単なるchar型になっているため，処理系によって符号の有無が変わります．もし，プログラムで次のように記述すると，処理系がchar型をunsigned char型で扱う場合は，型がプロトタイプと異なるので，エラーとなります．

```
signed char str[] = { 'A', 'B', 'C',
                                '¥0' };
strlen( str );
```

▶オプションで振る舞いを変更できる処理系もある

全てではありませんが，一部の処理系にはchar型の取り扱いを変更できるオプションがあります．RL78用のコンパイラであるCC-RLには，**図1**に示すようなオプションがあります．ただし，このようなオプションによって結果が変わるような記述は好ましくありません．移植性を保つためには，オプションの設定に関わらず，同一の結果となるように記述しましょう．

● 方法2：int型の範囲は±32767を超さない

2つ目はint型の値の範囲です．int型では16ビットを超える値を扱ってはいけません．

第2章
浮動小数点型

表1　浮動小数点型の値の範囲とサイズ
float ≦ double ≦ long doubleでなければならない．3つの型は同一精度でも構わない

型	値の範囲	サイズ
float	絶対値で少なくとも10^{-37}～10^{+37}	–
double	絶対値で少なくとも10^{-37}～10^{+37}	–
long double	絶対値で少なくとも10^{-37}～10^{+37}	–

8．内部表現

浮動小数点型には内部表現の規定がない

● 文法

▶型…3種類あるが内部表現の規定はない

表1に示すのは，C99における浮動小数点型の規定（値の範囲とサイズ）です．

浮動小数点型にはfloat，double，long doubleの3つの型がありますが，表現可能な値の最低値（絶対値）が決まっているだけで，サイズや内部表現に関しては規定がありません．3つの型は同じ精度でも構いません．精度に差を付ける場合は，float≦double≦long doubleの条件さえ守られていれば良いことになっています．

▶表現形式…主に10進法表現の2種類を使う

浮動小数点定数の表現形式は，初期の文法では小数点形式と指数形式（いずれも10進法表現）の2種類が存在していました．

- 小数点形式（10進法表現）
  `3.141592`
- 指数形式（10進法表現）
  `3141592E-6`

小数点形式は，見ての通り普段使う表記に近い形です．

指数形式のE（小文字のeでもよい）以降の指数部は，手前の定数値に乗算する10のべき乗の値を示します．前述の例は，3141592×10^{-6}を表しているので，小数点形式の浮動小数点定数と同じ値を表現しています．

小数点形式と指数形式に加え，C99からは16進法表現の浮動小数点定数が記述できるようになりました．10進法表現では，どうしても発生する恐れのある誤差を抑止するために用意された形式ですが，筆者にはその必要性が理解できませんでした．実用的ではないと思うので，本書では16進法表現の浮動小数点定数の解説は省略します．

● 整数型のような自動的な型変化はない

整数定数に型が存在するように，浮動小数点定数にも型が存在します．整数定数とは異なり，浮動小数点定数は表現した値によって型が変化することはありません．理由は**表1**に示したように，浮動小数点型は全て同じ精度でも構わないからです．同一精度ならば型を変更する意味がありません．浮動小数点定数では，double型を基本として扱います．

整数定数と同じように接尾子を付けることで型の割り当てを変えられます．これは浮動小数点型の精度が異なっていたときの対策です．

接尾子にはFまたはf，Lまたはlがあります．Fやfを付けるとfloat型，Lやlを付けるとlong double型になります．

● 代入規則はシンプル

浮動小数点型の代入規則は，次の通りです．内部表現すら決まっていないので，簡単な規則になっています．

- 代入先の型で表現できる値：変化せずに代入
- 代入先の型で表現できない値：処理系依存になる

これは浮動小数点型同士や浮動小数点型と整数型の間の代入にも言えます．

浮動小数点型を整数型に代入したときだけは特別な規定があり，「小数点以下は切り捨て」で代入されます．四捨五入は行いません．四捨五入を実行したければ，自ら補正後に代入するしかありません．例えば，小数点第1位で四捨五入して整数型に代入するのであれば，0.5を加算してから代入するなどの処理が必要です．

①片方がlong double型なら，他方もlong double型に変換して演算を行い，結果はlong double型となる
②片方がdouble型なら，他方もdouble型に変換して演算を行い，結果はdouble型となる
③片方がfloat型なら，他方もfloat型に変換して演算を行い，結果はfloat型となる
④これ以降は整数型の算術変換規則が続く

図1　浮動小数点型を含む算術変換規則
④以降は，第1章で紹介した整数型の算術変換規則が続く

● 算術変換規則…整数型＜浮動小数点型の精度

算術変換規則は，異なる型の間で演算を行うときの規則です．文法では整数型よりも浮動小数点型の方が大きな値が表現できると考えています．

その考えに基づいた算術変換規則を**図1**に示します．**図1**の④以降は，第1章で紹介した整数型の算術変換規則が続きます．

9．誤差

誤差により値を正確に表現できない場合がある

● 処理系の内部表現はIEEE 754に準拠している

浮動小数点型の文法の欠点は，なんと言っても内部表現を決めなかったことです．内部表現が決まっていなければ「手の出しようがない」というのが正直な感想です．それは処理系の開発者も同じ考えのようです．内部表現を決めなければ，コンパイラを作りようがないのは当然です．

その結果，どの処理系も世界中で通用する規格を採用しました．それがIEEE 754（Institute of Electrical and Electronics Engineers，Standard for Floating-Point Arithmetic）で，数種類の浮動小数点型の表現形式が規格化されています．

図2に示すのは，IEEE 754の中でも代表的な（より多く採用されている）表現形式の単精度形式，倍精度形式，拡張倍精度形式です．

どの表現形式も浮動小数点値を3つの部分に分離して管理しています．左側から符号部，指数部，仮数部です．表現形式ごとに実際の浮動小数点値の計算方法は多少変化しますが，簡単に説明すると**図3**のようになっています．

● 内部表現形式から実数値を計算してみる

IEEE 754で規格化された表現形式の詳細を紹介します．特に覚える必要はありませんが，ここでは参考として紹介します．

どの表現形式も同じ考えですが，32ビットの単精度形式を例に，格納されているビット・パターンから実際の実数値を計算してみます．単精度形式に次のビット・パターンが格納されていたとします．分かりやすくするため，符号部，指数部，仮数部の境界にスペースを入れています．

1 10000000 11000000000000000000000

▶符号部

符号部は，整数型と同じ考え方です．0ならば正，1ならば負を意味します．数式で表すと$(-1)^{符号部}$です．

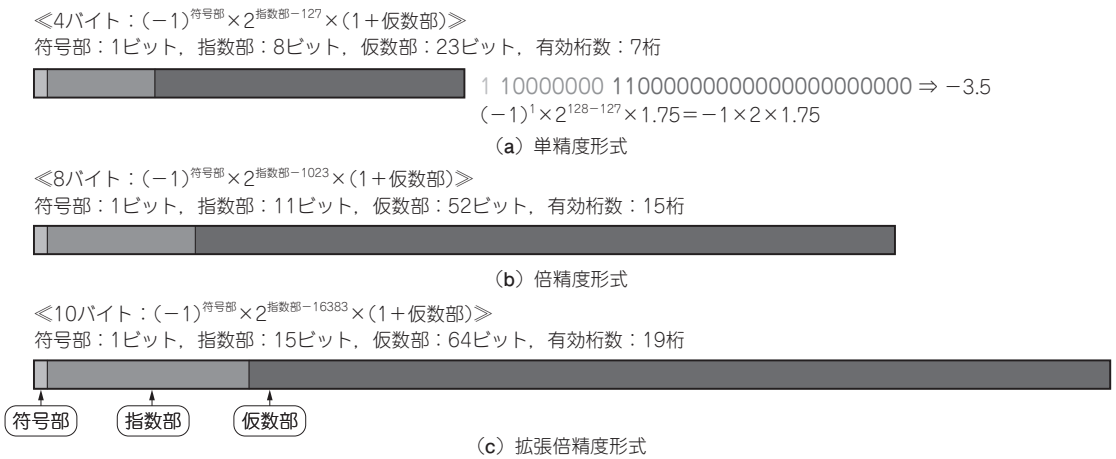


図2　IEEE 754で規格化されている浮動小数点型の内部表現形式
代表的な（より多く採用されている）表現形式を抜粋

第3章
派生型

12. typedef

難しい配列やポインタは typedefで分かりやすくなる

C言語で一番難しいのはポインタ型です．それは間違いないと思いますが，難しいポインタでC言語の理解度を判断するのは誤りです．筆者はポインタよりも整数型や浮動小数点型の算術変換の方が重要だと思っています．極端な話，ポインタを使わなくてもプログラムは作成できますが，算術変換を知らないと正しい数値計算が行えません．筆者の個人的な意見ですが，組み込みシステムの開発では構造体を指すポインタが理解できていれば十分です．次に示すような，ポインタの配列，ポインタを指すポインタ，2次元配列を指すポインタなどは，組み込みシステム開発では必要ないと思います．

```
int *a[5];     // ポインタの配列
int **pa;      // ポインタを指すポインタ
int (*pb)[5]; // 2次元配列を指すポインタ
```

本書では難しいポインタの宣言や使い方は解説しません．これらのポインタの解説は，既に多く出版されている書籍に譲ります．その代わりに，本稿ではtypedefを使って難しい配列やポインタを分かりやすくする方法を紹介します．

● 識別子の置き換えを行うキーワードtypedef

▶使い方

typedefは，識別子の置き換えを行うキーワードで，#defineと少しだけ似ています．例えば，次のように宣言した場合で考えてみます．

```
typedef int seisu;
```

太字部分が置き換え対象，下線部が置き換え後の識別子です．このように宣言すると，以降の処理ではseisuと記述すればint型を意味することになります．

```
seisu a, b;
```

このように宣言すれば，aとbは共にint型の変数になります．特に難しいこともなく，ここまでであれば#defineとあまり変わらないという印象だと思います．

▶機能1：ポインタ型の名前の置き換え

しかし，ここからがtypedefならではの機能です．typedefは，正確には型の名前を置き換えるキーワードです．型であれば，何でも識別子と置き換えられます．例えば，次のように宣言した場合で考えてみます．

```
typedef int *point;
```

このように宣言すると，int型を指すポインタ型に対して，pointという型名を付けたことになります．間違えないようにしましょう．スペースがあると，誰しもそこで区切りたくなるのですが，typedefは型の名前の置き換えであり，int *までが型なので，pointはint型を指すポインタ型になります．

```
point a, b;
```

このように宣言すれば，aとbは共にint型を指すポインタ型の変数となります．*がないのにaもbもポインタ変数になります．

▶機能2：配列型の名前の置き換え

それではもう1つ見ておきましょう．

```
typedef int array[10];
```

このように宣言すると，int型10個の配列型に対して，arrayという型名を付けたことになります．int [10]が型なので，arrayはint型10個の配列型です．そのため，次のように宣言すれば，aとbは共にint型10個の配列になります．

```
array a, b;
```

このように，[ ]がないのにa，bはint型10個の配列ということになります．

▶よく使われる用途…構造体の型名

ここまでがtypedefの機能ですが，これらの用途は一般的ではなく，構造体に使うことが多いようです．構造体は，プログラマが新たに作り出すことのできる型です．C言語が初めから準備している型ではないので，typedefで新たな型名を付けるというのは極めて自然な考え方です．

typedefで置き換えた型名には，多くの場合，英大文字が使われます．これは#defineを使ったマクロ名と同じ考え方で，後でリストを見返したときに置

き換えていることが一目で分かるようにするための配慮です．結果として，次のような使い方がtypedefでは一番多い用途となります（太字部分が型，下線が識別子）．

```
typedef struct ctag {
      struct ctag *next;
      int abc;
      int xyz;
} NEWTYPE;
```

● その1…ややこしい2重のポインタを見てみる

▶2重のポインタが指す先は2次元配列ではない

C言語で配列とポインタを勉強しているときに，しばしば勘違いする内容があります．それはポインタ型を指すポインタ型，いわゆる2重のポインタは2次元配列を指すためのポインタである，という考えです．そのため，次のように記述すると文法違反になります．

```
int a[5][10], **pa;
  pa = a; // 文法違反 ……(1)
```

勘違いしやすい理由は，普通のポインタと1次元配列ならば，次の内容が成立するからです．

```
int a[10], *pa;
  pa = a; // 問題なし
```

筆者は初心者のころ，この理由が分かりませんでした．今にして思えば当然の内容ですが，入門者にはかなり難しい内容だと思います．ここでは，なぜ2次元配列だと文法違反になるのかの理由を，前述のtypedefで説明します．

詳しい説明に入る前に，1つ覚えておいてほしいことがあります．前述の普通のポインタと1次元配列では，「int型の配列の先頭アドレスは，int型を指すポインタに代入できる」ではなく，「ある型の配列の先頭アドレスは，ある型を指すポインタに代入できる」と考えることです．

```
ある型 a[10], *pa;
  pa = a;
```

これは，ある型が配列として宣言できない関数型（void型）以外であれば通用します．それをtypedefで確認します．

▶試しに2次元配列を指すポインタを宣言してみる

2次元配列を指すポインタの宣言とは，どのような宣言なのでしょうか．答えは次の宣言であり，aとpaの関係は**図1**のようになります．

```
int a[5][10], (*pa)[10];
  pa = a; ……(2)
```

そう言われても入門者には難しい宣言ですよね．では，typedefを使って次のような型を宣言してみましょう．

```
typedef int XYZ[10];
```

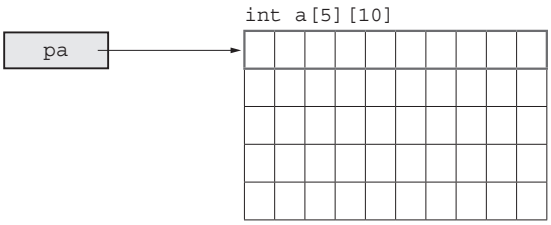


図1　2次元配列を指すポインタのイメージ
ポインタ型変数paは，int a[5][10]で宣言された2次元配列を指している

太字部分が置き換え対象，下線部が置き換え後の識別子です．これでint型10個の配列型がXYZ型になりました．そうすると前項の内容により，ある型をXYZ型にすれば，「XYZ型の配列の先頭アドレスは，XYZ型を指すポインタに代入できる」が成立することになります．

```
XYZ a[5], *pa;
  pa = a;
```

この宣言ならば，難しくないと思います．そして，この宣言をXYZ型を使わずに記述すると，式(2)の宣言になるのです．XYZ型はint型10個の配列型であり，それが次の太字部分のようにaにもpaにも存在しています．

```
int a[5][10], (*pa)[10];
  pa = a;
```

このように2次元配列を指すポインタは，小括弧を使った少し難しい宣言となります．また，全ての2次元配列が指せる訳ではなく，元々の型と1次元目の要素数（右側の要素数）が一致していないと指せないことを理解しておいてください．

▶2重のポインタが指す先はポインタ配列だ!

2重のポインタが指すものは，2次元配列ではなく，**図2**に示すポインタの配列です．

```
int *a[5], **pa;
  pa = a;……(3)
```

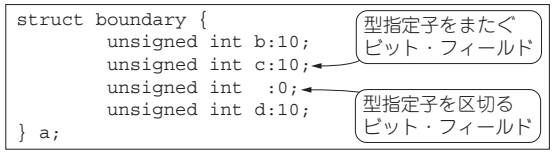
これも入門者には難しい宣言ですよね．そこでtypedefを使って，次のように宣言してみましょう．

```
typedef int *VW;
```

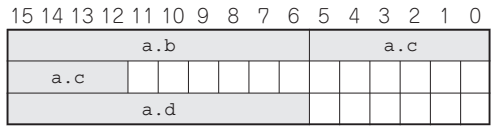
太字部分が置き換え対象，下線部が置き換え後の識別子です．これでint型を指すポインタ型がVW型になりました．そうすると前述の内容により，ある型をVW型にすれば，「VW型の配列の先頭アドレスは，VW型を指すポインタに代入できる」が成立することになります．

```
VW a[5], *pa;
  pa = a;
```

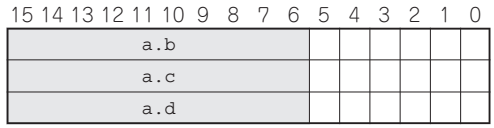
この宣言ならば難しくないと思います．この宣言をVW型を使わないで記述すると，式(3)の宣言になるのです．VW型は，int型を指すポインタ型であり，



（a）ソースコード



（b）型指定子をまたいでも連続して割り付けを行う例



（c）型指定子をまたぐ場合は次のフィールドに移動する例

図14　移植性が無い理由③…型指定子をまたぐビット幅の記述
int型は16ビット，ビッグ・エンディアン，MSBから割り付けて記載している

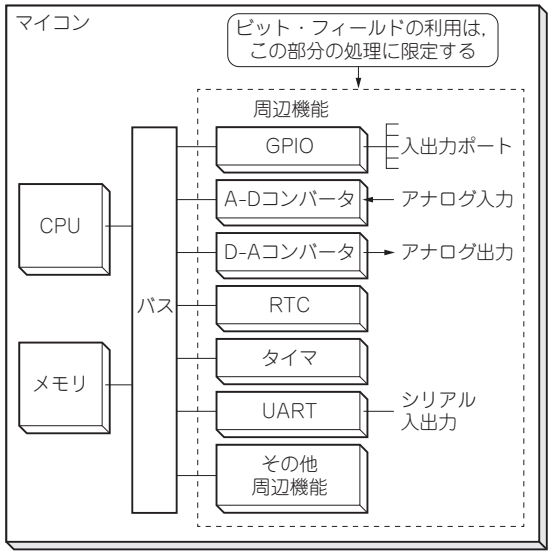


図15　ビット・フィールドの使いどころ…周辺機能の制御に限定するのがオススメ

うに強制的にすき間を空けて，次の型指定子の先頭に割り付けても良いことになっています．当然ながら，ビットの割り付け位置が変わってしまうので，避けた方が良い記述です．

▶ビット幅0指定で型指定子区切りになるハズだが…

宣言時のビット幅には，0指定もできるようになっています．ビット幅0は，そこで型指定子を区切ることを意味します．もちろん，残りのビット幅を計算して，無名のビット・フィールドを使っても同じ結果になりますが，計算するよりはビット幅0の方が簡単と言えます．ただし，ビット幅0の指定は，文法のリファレンスがK&Rの時代には存在していませんでした．しかも，あまり使われることのない文法であることから，ビット幅0をサポートしていない処理系も多々ありました．従って，ビット幅0の利用は避けた方が良いでしょう．

アドレス：FFF90H　　リセット時：0FFFH R/W

略号	15	14	13	12	11～0
ITMC	RINTE	0	0	0	ITCMP11-ITCMP0

図16　16ビット・メモリ操作命令の制約があるRL78/G14のITMCレジスタ
このレジスタは16ビット・メモリ操作命令で設定する制約があるが，普通にビット・フィールドで記述すると1ビット操作命令に展開されてしまう．volatile型修飾子を使えば，問題を解決できる可能性がある

■ ビット・フィールドの使いどころと注意点

● 周辺機能制御のみに適用するのがオススメ

これまで紹介したように，ビット・フィールドは曖昧な部分の多い文法なので，移植性を求めるのは不可能だと言わざるを得ません．しかし，ビット単位での操作を頻繁に行う周辺機能制御には欠かせない文法であることも事実です．

そこで中途半端な解決策ではありますが，**図15**に示す通りビット・フィールドの適用部分は周辺機能制御のみとするのが賢明だと思います．本来，周辺機能はマイコン固有の機能です．そうであれば，めったに移植することはないと思うので，ビット・フィールドを使っても問題にはならないでしょう．

同一シリーズのマイコンであれば，周辺機能も同じことが多く，その場合は処理系も同じでしょうから，こちらも問題は発生しないと思います．

ここからはビット・フィールドを周辺機能制御に適用するときにチェックすべきポイントを紹介します．

● ポイント1…アクセス・サイズとアクセス命令を確認する

1つ目は，アクセス・サイズとアクセス命令です．周辺機能のレジスタの中には，アクセス・サイズとアクセス命令に制約を持つものがあります．

▶16ビット・メモリ操作命令の制約がある場合

例えば，RL78には，インターバル・タイマのコンペア値を設定するITMCレジスタがあります（**図16**）．このITMCレジスタの特徴は，16ビット・メモリ操作命令（movw命令）を使うことが義務付けられていることです．そこでビット・フィールドの型指定子には，16ビット・サイズであるunsigned short型

リスト3　switch文に無駄な記述がある場合は処理が統合される

```
switch( a )  {
case 1:
    b = 100;
    break;
case 2:
    b = 100;
    break;
}
```

（a）caseラベルの1でも2でも同じ処理を行っている

```
switch( a )  {
case 1:
case 2:
    b = 100;
    break;
}
```

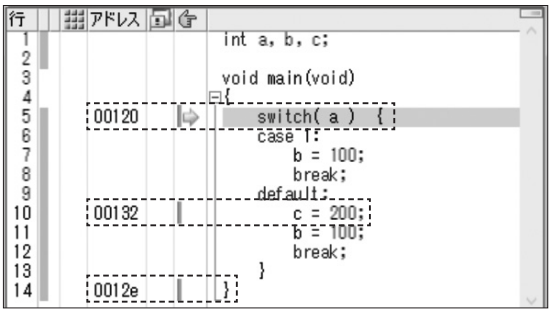
（b）caseラベルを並べて記述

```
switch( a )  {
case 1:
    b = 100;
    break;
default:
    c = 200;
    b = 100;
    break;
}
```

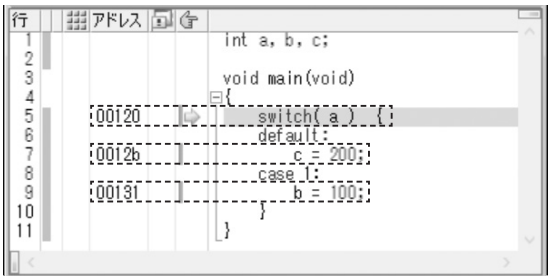
（c）caseラベルとdefaultラベルに分けて記述

```
switch( a )  {
default:
    c = 200;
case 1:
    b = 100;
}
```

（d）（c）の例を最小限で記述したもの



（a）リスト3（c）のプログラムの最適化結果



（b）リスト3（d）のプログラムの最適化結果

図3　最適化によるswitch文の処理の統合

switch文では処理の統合が行われることもあります．処理の統合とは，異なるcaseラベルに記述された文を1カ所にまとめることです．

かなり極端な例ですが，**リスト3**（a）のswitch文は無駄です．caseラベルの1でも2でも同じ処理を行っており，このような記述をする人はいないと思います．本来であれば**リスト3**（b）のようにcaseラベルを並べて記述するべきです．

しかし，**リスト3**（c）のような記述ならどうでしょうか．一見すると問題ないような気もしますが，このプログラムをダウンロードした後の表示は**図3**（a）の通りです．やはり，番地の表示のない行があります．これも最適化によって処理が統合されたからです．

▶switch文によくある解釈違い

そもそもswitch文の文法は非常に難しく，解釈を間違えている場合も少なからずあります．その内容は次の通りです．

- caseラベルは小さい順に並べる
  誤りです．caseラベルは順不同で，同じ値のものでない限り，自由な順番で記述できます
- break文は必ず使用する
  誤りです．break文はなくても構いません．break文がない場合，caseラベルをまたいで文が実行されます
- defaultラベルは必ず最後に記述する
  誤りです．defaultラベルはなくても良く，記述場所も自由で，switch文の先頭にあっても構いません

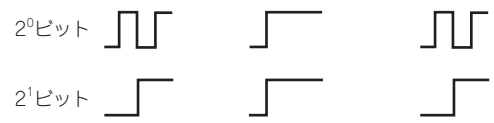
この内容から，**リスト3**（c）の例を最小限で記述すると，**リスト3**（d）のようになります．defaultラベルをswitch文の先頭に記述し，break文をあえて記述しませんでした．これならば**図3**（b）に示す通り，最適化されることもなく，全ての文に番地が表示されます．

筆者は，**リスト3**（d）のような記述がswitch文本来の機能であると考えています．全てのcaseラベルにbreak文を記述した構造は，if文の入れ子と同じ流れであり，必要性を感じません．

▶最適化を信用してあえて記述するのも手

MISRA-Cでは**リスト3**（d）のような使い方に制約を設けています．もし，break文を外す場合は，コメントなどでそのことを記載したり，不要であってもdefaultラベルは必ず記載することを推奨しています．転ばぬ先の杖のごとく，バグ対策が制約の目的です．

もし，バグ対策が目的なのであれば，これも最適化に頼ってはいかがでしょうか．MISRA-Cの通り，プログラムは分かりやすい形式で記述し，コード効率の向上は最適化に任せるということです．ただし，番地表示のない行に対しては，ブレークポイントが設定で



(a)端子から出力したいパルス　(b)volatile型修飾子がない場合(**リスト9**)　(c)volatile型修飾子がある場合(**リスト10**)

図3　I/Oポートから出力されるパルス

リスト11　volatile型修飾子による最適化抑止②…周辺機能のレジスタに対する書き込み処理

変数bにvolatile型修飾子を付ければ書き込み処理の削除を抑止できる

```
volatile unsigned char b;

void main(void)
{
    b = 1;
    b = 2;
    b = 3;
}
```

(a) ソースコード

```
《オフセット》《コード》      《命令》
00000000 E50000       oneb    !LOWW(_b)
00000003 CF000002     mov     !LOWW(_b), #0x02
00000007 CF000003     mov     !LOWW(_b), #0x03
```

(b) コンパイル結果

リスト10だと，最適化によって最後の書き込み命令以外を省略するので，**図3**(b)のように各々の端子は“H”が出力し続ける状態になります．このように周辺機能のレジスタに対する書き込み処理が削除されると，さまざまな問題を引き起こす可能性があります．

▶対策…volatile型修飾子を使う

このような場合は，volatile型修飾子を使えば最適化を抑止できます．**リスト10**(a)もvolatile型修飾子を使えば**リスト11**(a)となり，**リスト11**(b)のような結果になります．変数bの書き込み命令が3つとも残っています．これなら**図3**(c)に示すように期待通りのパルスが出力できます．

● 抑止が必要なケース②…周辺機能レジスタへの読み込み処理

無駄な読み込み処理の削除もvolatile型修飾子で抑止できます．最適化を抑止する理由も書き込み処理と同じで，操作の対象が周辺機能のレジスタであった場合，時間の経過とともに自動的に変化する可能性があります．

そのため，**リスト12**(a)のようにvolatile型修飾子を使います．コンパイル結果[**リスト12**(b)]の最後の3行がwhile文の処理に対応しています．volatile型修飾子の効果により，変数cの読み込み処理が削除されずに残されています．これならば変数cが周辺機能のレジスタであっても問題は発生しません．

リスト12　volatile型修飾子による最適化抑止③…周辺機能のレジスタからの読み込み処理

変数cにvolatile型修飾子を付ければ読み込み処理の削除を抑止できる

```
volatile int c;

void main(void)
{
    c = 1;
    while( c != 0 )
        ;
}
```

(a) ソースコード

```
《オフセット》《コード》《ラベル》《命令》
00000000 E6           onew    ax
00000001 BF0000       movw    !LOWW(_c), ax
00000004 AF0000 @1_1  movw    ax, !LOWW(_c)
00000007 6168         or      a, x
00000009 DF00         bnz     $@1_1
```

(b) コンパイル結果

*

このように書き込み/読み込み処理の削除は，操作の対象が変数ではなく，周辺機能のレジスタである場合に問題となる可能性が非常に高いです．volatile型修飾子を利用すれば最適化による削除を抑止できます．

35. 変数に対する最適化の抑止①…局所変数

局所変数は積極的に最適化を実施する

● 局所変数にはvolatileは不要

結論から言えば，局所変数に対してvolatile型修飾子を指定する必要はありません．理由は簡単であり，局所変数はそれが宣言された関数内部でしか使用できないからです．実引数でアドレスを渡さない限り，他の関数からは操作できません．言い換えれば，変数が知らぬ間に変化したり，変数の変化によって影響を受けたりする他の関数は存在しないのです．あくまでも変数に対する操作は全て目的の関数内に記述されているため，極限まで最適化を施しても問題となることはありません．

● 局所変数にvolatileを指定すると性能が劣化する

逆に局所変数にvolatile型修飾子を指定してしまうと著しく性能が劣化する恐れがあります．局所変数はregister記憶クラスが指定可能であり，型や個数は処理系により左右されますが，高速にアクセス可能なCPU内部レジスタに割り付けることができま

第6章
静的変数領域の削減

表1　C言語の変数の分類…静的変数と動的変数
局所変数と大域変数とは異なり，記憶クラスと密接に関わっている

宣言場所 / 記憶クラス	局所変数（関数，複文の先頭）	大域変数（関数外部）
なし	動的変数①	静的変数②
auto	動的変数⑤	文法違反
static	静的変数③	静的変数④
register	動的変数⑥	文法違反

第6章は，静的変数領域の削減について紹介します．「静的変数」の一言で，どの変数であるかが理解できれば良いのですが，宣言場所や記憶クラスも絡んでおり，多少複雑です．そこで静的変数の文法的な説明も含めて，その削減方法を紹介します．

38．static記憶クラス
staticには隠蔽と静的の2つの意味がある

● 変数は静的変数と動的変数に分かれる

C言語の変数は，その性質により静的変数と動的変数の2つに分かれます．これは局所変数や大域変数とは異なる意味の言葉であり，両者を混同してはいけません．これらの言葉は記憶クラスとも密接に関わっています．内訳は**表1**の通りです．

▶局所変数と大域変数のおさらい

表1の横軸は，変数宣言が行われている場所の内訳です．関数や複文の先頭[注1]で宣言されたものが局所変数です．関数外部で宣言されたものが大域変数です．

局所変数の宣言場所には，{　}の複文の先頭も含まれます．局所変数は，**リスト1**の変数aのように関数の先頭で宣言するのが一般的ですが，変数bのように複文の先頭でも宣言できます．

ただし，複文の先頭で宣言された変数のスコープは，その複文内に限られます．**リスト1**に示す通り，変数bは複文から外れると使えません．変数bのように，同じ関数内であっても使える場所と使えない場所が混在するので，一般的にはあまり使われません．

リスト1　局所変数の宣言場所とスコープ
複文の先頭で宣言した変数は，その複文内でしか使えない

```
void func(void)
{
int a;
        // 変数aのみ使用可能
    {
    int b;
        // 変数aとbが使用可能
    }
        // 変数aのみ使用可能
}
```

▶覚えるべきは4通りの組み合わせ

表1の縦軸は，記憶クラスの指定です．文法上，記憶クラス指定子には**表1**に記載したもの以外にもexternとtypedefがありますが，どちらも記憶領域を確保するものではないので除外しています．また，このうち記憶クラスがautoとregisterのものは覚えなくても良いと思います．理由はコラム1に示します．

結果，覚えるべき内容は，局所変数，大域変数ともに記憶クラスの指定なしとstaticの4カ所です．変数の特性としては，**表1**に示す①記憶クラスの指定なしの局所変数のみが動的変数，②～④が静的変数となります．⑤と⑥は①と同じですが，使う機会がないので覚える必要はありません．

▶static記憶クラスの意味を解説

筆者は，static記憶クラスをライブラリのソースコード以外は利用する必要がないと考えています．しかし，MISRA-Cの考え方は真逆で，static記憶クラスの積極的な利用を推奨しています．

従って，MISRA-Cを採用する場合は，static記憶クラスの意味を覚える必要があります．ところが，static記憶クラスには2つの意味があるので厄介です．ここではstatic記憶クラスの意味を大域変数，局所変数の順番で紹介します．

注1：C99から局所変数は複文の先頭以外にも，文と文の間やforの前処理でも宣言可能となりましたが，ここでは複文の先頭でのみ宣言可能であったC89の文法で紹介します．

コラム1 記憶クラスautoとregisterを使わなくても良い理由

▶auto記憶クラス…局所変数への指定は省略できる

auto記憶クラスは本来，動的変数への割り当てを指定するものです．従って，絶対に動的変数となることがない大域変数に対しては文法違反であり，元々動的変数となる局所変数に対しては省略可能となっています．「auto記憶クラスを指定したら最適化せずに必ずスタック領域を使用する」という縛りがあれば使い道もありますが，大域変数には指定できない，局所変数には指定してもしなくても一緒，ということで使う理由が存在しません．

▶register記憶クラス…処理系にお任せでOK

register記憶クラスは，動的変数の記憶場所をスタック領域ではなくCPU内部レジスタに変更し，処理性能を飛躍的に向上させるものです．動的変数とはならない大域変数には指定できません．局所変数に指定してもCPUの制約上，割り当て可能な個数に限界があります．そもそもCPU内部レジスタには割り当て不可能な型なども存在します．

20年くらい前の処理系であれば比較的プログラマの意思を尊重し，register記憶クラスの指定あり/なしでコンパイル結果が変化しました．しかし，近年の処理系の中にはまったくプログラマの意思を無視し，register記憶クラスを指定してもコンパイル結果が変化しないものも存在します．

従って，最適化技術の発達した近年ではregister記憶クラスは処理系にお任せで良いと思います．なお，register記憶クラスを指定しても，変数の特性は変化しません．

リスト2　大域変数は通常だと他のソース・ファイルからでも参照できる

extern記憶クラスを使って宣言すれば利用可能

```
int abc;
```

（a）ファイルX

```
extern int abc;
// 宣言以降の関数では変数abcが利用可能
```

（b）ファイルY

リスト3　static記憶クラスが指定されていると他のソース・ファイルから参照できなくなる

extern記憶クラスを使って宣言しても利用不可

```
int abc;
static int def;
```

（a）ファイルX

```
extern int abc;
extern int def;
// 宣言以降の関数では変数abcが利用可能
// 一方，変数defは利用不可能
```

（b）ファイルY

39．大域変数に指定するstatic記憶クラス

大域変数に対して使うstatic記憶クラスは変数の破損を防ぐ隠蔽の意味を持つ

● 大域変数に指定するstaticの意味は隠蔽

大域変数に対してstatic記憶クラスを指定する意味は隠蔽です．static記憶クラスが指定された大域変数の識別子（名前）は隠蔽され，他のソース・ファイルからはextern記憶クラスを使っても参照できなくなります．

▶隠蔽の具体例

具体例で見てみましょう．例えば，**リスト2**に示す通り，単純な大域変数は他のソース・ファイルからでもextern記憶クラスを使って宣言すれば利用可能です．しかし，**リスト3**のようにstatic記憶クラスが指定されると，extern記憶クラスを使って宣言しても利用不可能となります．

▶隠蔽した変数を参照しようとするとこうなる

利用不可能の意味を間違えないようにしましょう．このコードはコンパイルのときではなく，リンクのときにエラーになります．

コンパイラはファイルごとに翻訳するので，extern記憶クラスで宣言されていれば，他のソース・ファイルに実体があるものと仮定して命令を生成します．しかし，その実体はstatic記憶クラスで隠蔽されています．リンカはシステム全体が翻訳対象なので，リンクのときに「未定義エラー（Undefined Symbol）」となります．

このエラーに対しては，タグ・ジャンプ機能が働きません．エラー・メッセージをダブルクリックしても，エラーのある行にはジャンプできないと思います．もし発生したら，エラーとなった識別子を検索して，どこがエラーの原因なのかを調べましょう．

● 隠蔽の目的…変数の破損を防ぐ

隠蔽の目的やメリットは，予期せぬアクセスから発生する変数の破損を防ぐことにあります．

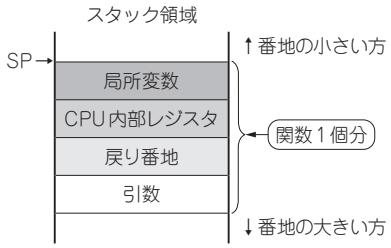


図4　スタック領域に保存されるデータの内容
関数が呼び出されたときの保存内容

リスト2　このプログラムの関数を実行したときのスタックの使われ方を見てみる

```
void main(void)
{
     関数( 引数 );     // 引数の格納
}                       // 関数呼び出し

void 関数(引数の宣言)
{
局所変数の宣言          // 局所変数の確保
     文                 // レジスタの退避
     文
}
```

グは関数呼び出しが行われたときです．関数が呼び出されると，これらのデータが番地の大きい方から小さい方へと保存（または確保）されていきます．関数が終了するとき，番地の小さい方から順番に必要なデータは復旧され，必要のないデータは破棄されながら，関数呼び出し前の状態に戻ります．

また，マイコンのエンディアンがビッグ/リトルのどちらであっても，不思議なことにスタック領域は番地の大きい方から小さい方に向かって使用されます．またスタック領域は，ほとんどのマイコンでSPと名の付くスタック・ポインタと呼ばれるCPU内部レジスタで管理します．

ここまで説明した内容を確認します．大ざっぱですが，**リスト2**の例でスタック領域の使われ方を解説します．関数呼び出し前のスタック領域は**図5**（**a**）の通りとします．

● ステップ①…引数を格納

関数呼び出しのとき，最初にスタック領域に格納されるのは**図5**（**b**）が示すように引数です．

C言語での引数の渡し方はコピー渡しなので，引数はスタック領域にコピーしながら格納します．**図5**（**b**）では，引数が1つしか記載されていませんが，複数個が指定された場合は複数個格納されます．

▶全ての引数が格納される訳ではない

全ての引数がスタック領域に格納されるとは限りません．近年の処理系は，コピー先としてスタック領域ではなくCPU内部の汎用レジスタを使うことが多いです．スタック領域は単なるメモリなので，汎用レジスタを利用した方が性能的に優れているという訳です．

▶格納規則は複雑で処理系により異なる

ただし，汎用レジスタには数に限りがあります．引数の型（サイズ）にも影響を受けます．結果，引数の汎用レジスタへの格納規則は非常に複雑です．同じマイコンであっても処理系が異なると規則が違うことが

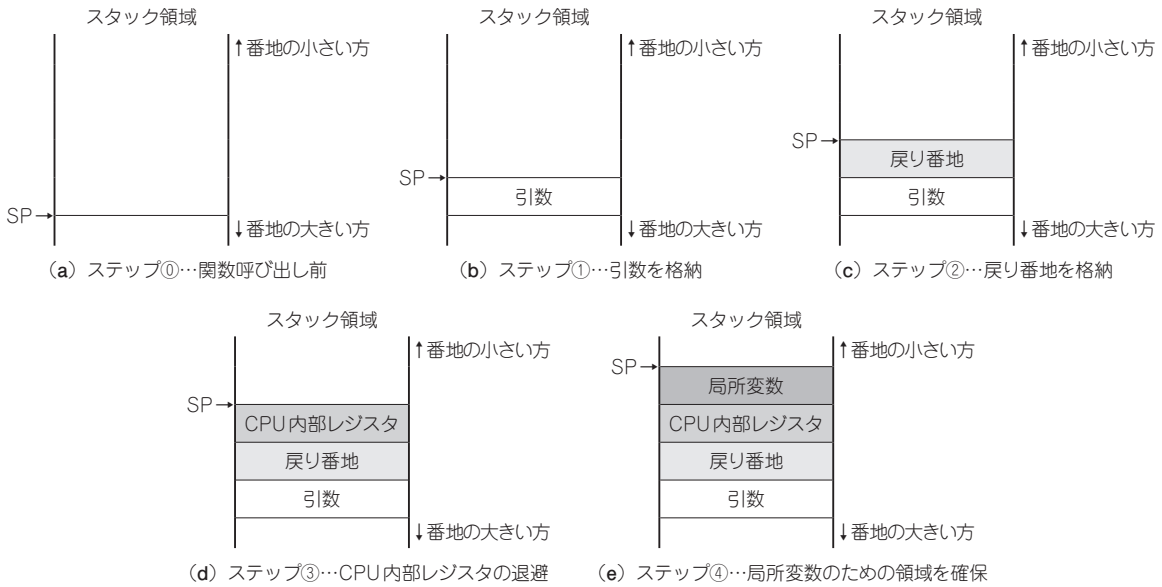


図5　関数呼び出し時のスタック領域の使われ方

コラム　スタック領域が破壊されてプログラムが暴走したらどうなる？

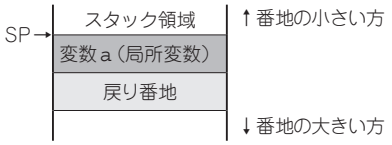
スタック領域が破壊されたら，プログラムはどんな動作になるのでしょうか．必ず暴走すると思われるかも知れませんが，実際はそうでもありません．本文の**図4**を利用して説明すると，引数，CPU内部レジスタ，局所変数の領域に関しては，「引数の値が突然変化した」とか，「局所変数の値がいつの間にか不定な値となった」といった具合に暴走とまではならず，変数が破壊される結果となります．

これに対して，戻り番地の領域が破壊されると，プログラムは間違いなく暴走します．そこでRL78を使い，**リストA**に示す戻り番地を破壊するプログラムを実行してみましょう．

main関数から引数なしでsub関数を呼び出すと，スタック領域には戻り番地が格納され，sub関数に制御が移ります．sub関数はvolatile型修飾子を指定して強制的に変数a（局所変数）をスタック領域に確保します．そうするとスタック領域は**図A**のようになります．この状態で(&a)[1]の記述で変数aの1つ先，すなわち戻り番地の領域を強制的にゼロにして，関数を終了します．

このプログラムをシングルステップで実行すると，sub関数終了時は**図B**に示すような表示となります．プログラムがゼロ番地に遷移し，いわゆるプログラムが暴走した状態です．

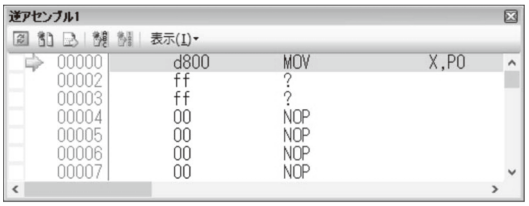
以上のようにスタック領域の中でも戻り番地の領域が破壊されるとプログラムは必ず暴走します．デバッグ中にプログラムを停止したとき，C言語ではなく，アセンブリ言語の命令列が表示されたら，スタック領域の破壊を疑ってください．



図A スタック領域の状態

リストA
戻り番地を破壊するプログラム

```
void main(void)
{
    sub( );
    while( 1 )  ;
}

void sub(void)
{
volatile long a;

    (&a)[1] = 0;
}
```



図B プログラム実行結果

48. スタック・サイズの計算方法
まずは手動で計算する

● 必須アイテム①…関数呼び出し経路図
▶どの関数がどの関数を呼び出すのかを表現

スタック・サイズを計算するには，自身が作成したシステムの関数呼び出し経路図が必要です．

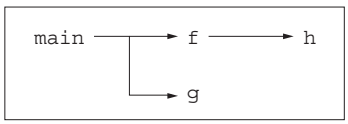


図6　スタック・サイズ計算の必須アイテム…関数呼び出し経路図
この例では，main関数はf関数とg関数を呼び出し，f関数はh関数を呼び出すことを意味している

図6に示すのは，どの関数がどの関数を呼び出すのかを表現した関数呼び出し経路です．**図6**の場合，main関数はf関数とg関数を呼び出し，f関数はh関数を呼び出すことを意味しています．この経路図における関数の呼び出し関係だけを示すと，**リスト3**のようになります．

リスト3　図6の関数呼び出し関係のみをC言語のソースコードで示すとこうなる

```
void main(void)
{
    f( );
    g( );
}

void f(void)
{
    h( );
}

void g(void)
{

}

void h(void)
{

}
```

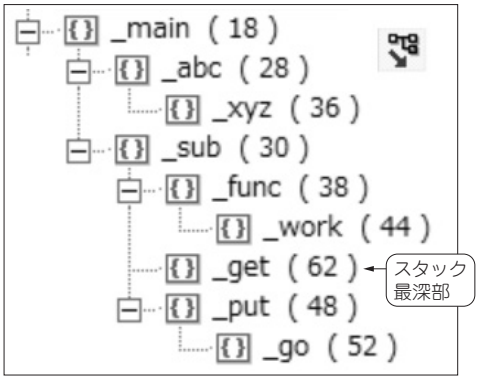


図15　呼び出し情報ビューの表示方式②…Show Used Stack（使用済みスタック表示形式）
経路を実行するのに必要なスタック・サイズが分かりづらい反面，どの関数が経路の最深部なのかは簡単に分かる

値が一番大きい関数を探します．その値は「Max」が示す値でもあるので，それと同じ値が表示されている関数がスタック最深部の関数になります．今回の例であればget関数がスタック最深部の関数です．

▶手順2：割り込み関数をレベルごとにスタック最深部へ移動

割り込みレベルの一番低い関数を選択して，ドラッグ＆ドロップでスタック最深部の関数へ移動させます．今回の例であれば，割り込みレベル3のintp0関数を選択し，ドラッグ＆ドロップでget関数へ移動させます．すると，**図16**（**a**）のようにintp0関数がスタック最深部の関数に変わります．

次にレベル1のintit関数，intad関数，intkr関数を選択し，ドラッグ＆ドロップでintp0関数へ移動させます．すると，**図16**（**b**）のようにintkr関数がスタック最深部の関数に変わります．

次にレベル0のintsr0関数と，intsre0関数を選択し，ドラッグ＆ドロップでintkr関数へ移動させます．

▶手順3：スタック使用量を確認

全ての割り込み関数の移動が完了したら，表示を「Show Required Stack」に戻します．すると**図16**（**c**）のように表示されます．main関数の経路の実行に必要なスタック・サイズが102バイトになっています．割り込み関数を移動したことにより，スタック使用量に割り込みが考慮された数値になりました．

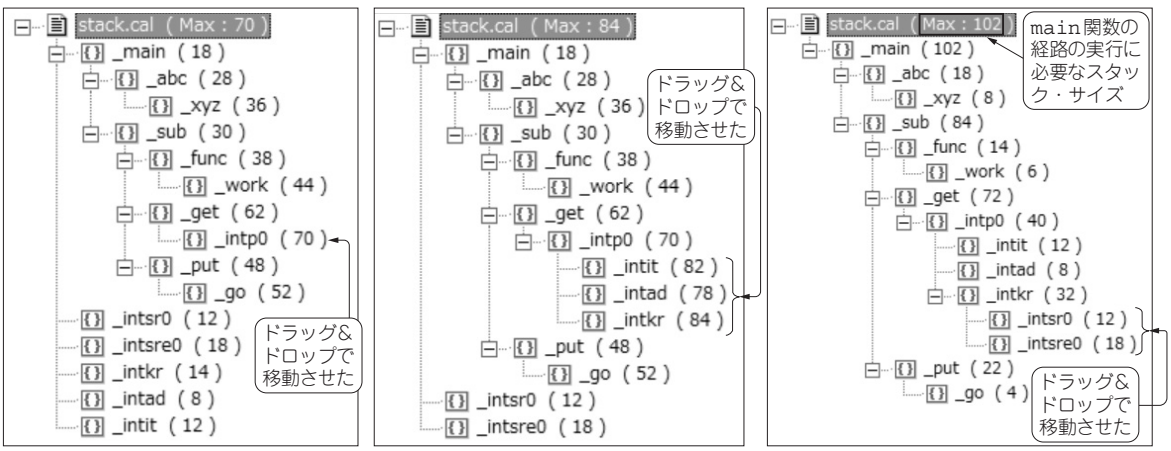
ここまでの操作を行って，ようやく「Max」の示す数値が意味を持ちます．すなわちシステム全体のスタック・サイズです．

リスト5　標準ライブラリ関数のsin関数をポインタ経由で呼び出すプログラム
このプログラムをCall Walkerで解析するとどうなるか見てみる

```
#include <math.h>

double a, b;

void main(void)
{
double (*point)(double) = sin;

    a = point( b );
}
```

● 処理系に依存するスタック解析にも対応

Call Walkerは，関数ポインタや標準ライブラリ関数，ランタイム・ライブラリにも対応しています．そこで，**リスト5**に示すプログラムのスタック解析結果



（**a**）割り込みレベル3の割り込み関数移動後の表示　（**b**）割り込みレベル1の割り込み関数移動後の表示　（**c**）割り込みレベル0の割り込み関数移動後の表示

図16　割り込み関数をレベルごとにスタック最深部へ移動していく様子

第8章
動的スタック領域の削減

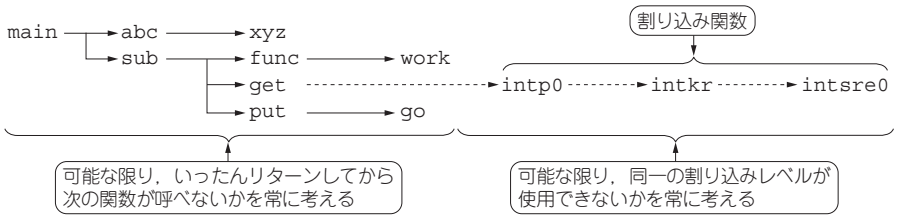


図1　関数呼び出しのネスト・レベルが深くなるほどスタック使用量も増大する

第8章は，動的領域であるスタック領域の削減について紹介します．スタックの使用量は，プログラミングの経験値がはっきりと出る項目です．素人だとスタック領域は肥大化し，玄人だとスタック領域はそれほど大きくなりません．

素人と言われないためにも，動的領域であるスタック領域の削減方法を理解しましょう．

50．スタック領域の減らし方
ネスト・レベルを減らすのが最優先

● 方法①…関数呼び出しのネスト・レベルを減らす

図1に示す通り，スタック使用量は関数の呼び出しネスト・レベルが深くなると増大します．削減するためには，ネスト・レベルを減らすのが1番です．

例えば，必要以上に関数を細分化して，スタック使用量のことを考えず，ネスト・レベルの深いプログラムを設計する人が居たとします．それは関数の再利用性の観点では良いかもしれませんが，スタック使用量の観点だと誤りです．動的領域の削減を優先するのであれば，関数は常にいったん戻してから次の関数が呼べないかを考えるべきです．関数呼び出しのネスト・レベルが深くなればなるほどスタック使用量が増加すると考えましょう．

● 方法②…割り込みのネスト・レベルを減らす

最近は，多重割り込みのレベル数が15程度あるマイコンが多くなってきました．本書で取り上げているルネサス エレクトロニクスのマイコンでは，SHとRXが15段階の多重割り込みが可能です．割り込みレベル数が豊富ですが，これも関数の呼び出しネスト・レベルと同様に使用する割り込みレベル数を減らすことでスタック使用量を削減できます．

▶多重割り込みを使うとスタック使用量が増大する

15段階あるからといって，15段階全てを使うと，**図1**に示す通り，使用した割り込みレベルは，その中で最もスタック領域を消費する割り込み関数を全て加算しなければなりません．**図1**の場合はわずか3レベルですが，これが15レベルとなったときを想像してみてください．**図1**の4倍程度のスタック使用量となることは容易に想像できます．可能な限り同一の割り込みレベルで処理できないかを常に考えるべきです．

▶同一割り込みレベルでは遅延を考慮しよう

もちろん，同一の割り込みレベルを使った場合，考慮しなければならないのは遅延です．同一である他の割り込みは受け付けられません．今実行中の割り込みが終了しなければ受け付けられないので，その間は処理の遅延が発生することになります．その遅延時間がシステムに悪影響を与えないか検討する必要があります．

その検討を面倒と思ってはいけません．このような検討ができるかどうかで組み込みプログラマとしての評価が決まってしまいます．スタック領域を削減するためにも，可能な限り同一の割り込みレベルが使用できないかと考えるようにしましょう．

リスト1　再帰関数の例①…自然数の階乗を求めるfact関数

```
int fact(int n)
{
    return n == 1 ? 1 : n * fact( n-1 );
}
```

リスト2　再帰関数の例②…2つの自然数の最大公約数を求めるgcm関数

```
int gcm(int a, int b)
{
    return b == 0 ? a : gcm( b, a%b );
}
```

図2　スタック・サイズ自動計算ツールCall Walkerにおける再帰関数の表示

● 組み込み系ではご法度な「再帰呼び出し」

▶数学的な計算を行う関数でよく登場する

再帰呼び出しとは，自分で自分を呼び出す関数のことです．同じ処理の繰り返しであれば，再帰関数として記述できる可能性が高いです．

再帰呼び出しの代表例は，数学的な計算を行う関数です．例えば，自然数の階乗を求める関数は，**リスト1**のように記述できます．

数学では，自然数nの階乗$n!$は，$n \times (n-1)!$と等しいことになっていて，階乗を求めるfact関数は，この公式をそのままコード化しています．ここで階乗の計算をfact関数に置き換えてみます．そうすると，fact(n)の結果は，return文の最後に記述されているn*fact(n-1)で良いことになります．

ただし，このままでは終わりがないので，条件式を使い，nが1のときは再帰呼び出しせず，1の階乗結果である1を返しています．

階乗以外にも，**リスト2**のように2つの自然数の最大公約数を求める関数も再帰関数で作成できます．この再帰関数は，ユークリッドの互除法を利用しています．詳細は省きますが，**リスト2**のgcm関数も再帰呼び出しの代表例です．

▶引数によってスタックの使用量が変わってしまう

数学上の公式をそのままプログラムに応用できる再帰呼び出しですが，スタックの使用量が引数の値によって変化することから，組み込み系ではご法度の関数と言えます．**リスト1**を見れば分かりますが，fact関数は引数の値と同じ回数だけ呼び出されます．

結果，再帰呼び出しの関数は，1回当たりのスタック使用量×呼び出しネスト回数が，実際に必要なスタック使用量です．

前回紹介したスタック・サイズ自動計算ツールCall Walkerでは，再帰呼び出し関数を使うと，**図2**のように矢印が回っているような表示になります．

▶組み込み系では再帰関数は使用しない

再帰関数は，システム動作中でもスタック領域を動的に増やすことが可能な仮想記憶機構を持つ環境下でのみ使用が許されます．一般的には仮想記憶機構を持たない組み込み系において，再帰関数は使用してはならないと考えましょう．

リスト3　再帰関数を使わずにfact関数とgcm関数を記述した例
組み込み系ではこの記述の方が好まれる

```
int fact(int n)
{
int ret;
    for( ret=1 ; n!=1 ; n-- )
        ret *= n;
    return ret;
}

int gcm(int a, int b)
{
int r;
    while( b )  {
        r = a % b;    a = b;    b = r;
    }
    return a;
}
```

再帰関数よりは陳腐に見えますが，**リスト3**の方が組み込み系では好まれる記述です．

51．関数の引数の個数

汎用レジスタに格納できる個数に抑えよう

● スタック領域に格納される情報を考える

ネスト・レベル以外でスタック使用量を削減する方法は，スタック領域に格納されるデータを個別に考えるしかありません．以前に紹介した通り，スタック領域には**図3**に示すデータが格納されます．この中で削減対象から外れるものは戻り番地です．関数からの戻り番地だけは固定のサイズであり，現状市販されているマイコンだと2バイトか4バイトとなります．どのように関数を記述しても変化しないサイズですから，削減対象からは除外します．

結果，考えるべきは引数，CPU内部レジスタ，局所変数として使用される3つの領域となります．本章

ISBN978-4-7898-4555-7
C3055 ￥2600E
CQ出版社

定価2,860円（本体2,600円）⑩