付録 C　　処理系定義の動作について

C 言語には、言語規格で未規定（不定）や未定義とされる記述があります（コラム参照）。未規定とされるものの一部には処理系（コンパイラ）が動作を定義することになっているものがあり、これを「処理系定義の動作」といいます。処理系定義の動作となる項目は処理系ごとに動作が規定されています。すなわち同じ処理系であれば、いつも同じように動作します。

一方で、処理系が異なる場合には、ソースプログラム上の同じ記述が同じ動作とはならないことがあります。このため、プログラムの移植をしたり処理系を変更したりする場合に注意が必要です。また、特定の処理系だけに慣れている場合、そこでの処理系定義の動作を C 言語規格で規定された動作と思い込み、別な処理系を使う際に思わぬミスを引き起こすことがあります。したがって、開発を始める前に処理系定義の動作について確認しておくことが望ましいです。

処理系定義の動作（以下、処理系定義と略します）は、通常コンパイラのマニュアルに記述されています。ここでは、その代表的なものについて解説します。

処理系定義の代表例　１：動作環境

処理系定義の記述には、フリースタンディング環境（Freestanding environment）という言葉が登場することがあります。フリースタンディング環境とは平たく言えば OS のない環境です。このような環境では、プログラム起動時に呼び出す関数の名前と種類は処理系定義です。通常は `main` 関数が呼び出されますが、起動から `main` 関数に至るまでにどのような処理（見えない関数）が呼び出されているかは処理系によって異なります。

また、`main` 関数が終了したり `exit` などでプログラムを中断したりした場合、その後どのような処理になるかも処理系定義です。まず、そのようなことを起こさないプログラムとすることが第一ですが、もしそうなった場合にどのような動作になるかは知っておく必要があります。

処理系定義の代表例　２：文字コード

文字コードとは、文字や記号をコンピュータで扱うために文字や記号一つ一つに割り当てられた固有の数値のことです。文字集合と対応する文字コードの集合の対応関係を文字コード体系といいます。どのような文字コード体系を利用できるかは処理系定義です。表 1 は ASCII という 7 ビットの文字コード体系の表です。横軸が上位 3 ビット、縦軸が下位 4 ビットを表しています。たとえば英字 A は、上位 3 ビットが 4 で下位 4 ビットが 1 ですから、対応する文字コードは 0x41 と分かります。

ASCII とは異なる文字コード体系として 8 ビットの EBCDIC（エビシディック）があります。表 2 は EBCDIC 文字コード体系の表です。英字 A の文字コードは ASCII では 0x41 でしたが、EBCDIC では 0xC1 です。

下位ビット ＼ 上位ビット	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	´	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

表１ ASCII コード表

下位ビット ＼ 上位ビット	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	DS		SP	&	-						{	}	\	0
1	SOH	DC1	SOS				/		a	j	~		A	J		1
2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
3	ETX	TM							c	l	t		C	L	T	3
4	PF	RES	BYP	PN					d	m	u		D	M	U	4
5	HT	NL	LF	RS					e	n	v		E	N	V	5
6	LC	BS	ETB	UC					f	o	w		F	O	W	6
7	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
8		CAN							h	q	y		H	Q	Y	8
9		EM						`	i	r	z		I	R	Z	9
A	SMM	CC	SM		¢	!	|	:							
B	VT	CU1	CU2	CU3	·	$	,	#								
C	FF	IFS		DC4	<	*	%	@								
D	CR	IGS	ENQ	NAK	(	)	_	'								
E	SO	IRS	ACK		+	;	>	=								
F	SI	IUS	BEL	SUB	|	¬	?	"								

表２ EBCDIC コード表

このように、文字を表現するコードは使用する文字コード体系によって異なります。開発環境（ソースプログラムを扱う環境）と実行環境（実行ファイルが動作する環境）で使う文字コード体系が異なる場合、処理系がどのように対応しているか注意が必要です。

日本語（ひらがな、漢字など）には、1 文字に対して 2 バイト以上からなる文字コードが割り当てられています。この文字コードにも複数の体系があり、使用する体系によって値が異なりますから、処理系がどのように対応しているか注意が必要です。現在多くのパソコンなどでは、日本語を表すために 1 文字を 2 バイトで表現する Shift_JIS という文字コード体系を使っています。

また、日本語も含めた多言語の文字を統一的に扱うために Unicode（ユニコード）という文字コード体系が制定され、近年広く使われるようになっています。Unicode では、1 文字を 1 バイトから 6 バイトの多バイトで表現し、その表現の仕方（符号化方式）によって次の三つの主要なコード体系があります。

1. UTF-8 ：ASCII と同じ部分は 1 バイトで、それ以外を 2 から 6 バイトの可変長で表現する
2. UTF-16：16 ビットを単位として 1 文字を 1 単位（16 ビット）または 2 単位（32 ビット）で表現する
3. UTF-32：32 ビットの単位一つだけの固定長で 1 文字を表現する

C 言語では、多バイトの文字コード体系で表現される文字を、1 文字を一定のビット数の整数値として扱えるように「ワイド文字」という文法を導入して機能拡張してきています。たとえば C99 では、ワイド文字用の型 `wchar_t` を導入しており、それを扱うライブラリも追加されています。

ライブラリの例：

```
int vwprintf (const wchar_t * restrict format, va_list arg);
    // ワイド文字に対応した printf
```

また C11 では、`char16_t`（2 バイト長）、`char32_t`（4 バイト長）という型を追加しています。

ここで、wchar_t のサイズや、それぞれのワイド文字用の型がどのような文字コードに対応するかは処理系定義です。なお、C11 では __STDC_UTF_16__ と __STDC_UTF_32__ というマクロを導入しており、このマクロが定義されている場合、char16_t と char32_t の符号化がそれぞれ UTF-16 と UTF-32 であることを示しています。

使用する文字コード体系の違い以外でも、たとえば Shift_JIS で表現される漢字を使う場合の注意点があります。Shift_JIS では、2 バイトで 1 文字を表現しますが、その 2 バイト目の値が ASCII の「\」（バックスラッシュ（または「¥」））と同じになるものがあります。たとえば「表」という文字のコードは Shift_JIS では 0x955c です。この先頭から 2 バイト目の 0x5c は ASCII では「\」に相当します（表 1）。

Shift_JIS をサポートしていないコンパイラの場合、2 バイトで構成される 1 文字を単なる 1 バイトの文字の列とみなして「\」によるエスケープシーケンスの処理を行い、意図した表示にならないなどという不具合につながります。

例：

```
ソースコード：　　printf("表\n"); // バイト列：0x95 0x5c 0x5c 0x6e
                                  //          → 0x5c 0x5c(\\)は 0x5c(\)になる
出力　　：　　表 n                 // バイト列：0x95 0x5c 0x6e
                                  // 本来は「表」の後で改行することを意図
```

処理系定義の代表例　３：ポインタとアドレス

組込みソフトでは、アドレスの絶対値を扱うことが少なくありません。特定のアドレスにアクセスする場合にポインタを使いますが、この場合以下の例のようにポインタに整数を代入する（またはその逆の）演算が必要です。

```
unsigned char *addrp = (unsigned char *)0xffff0123L;
```

このような整数とポインタの変換が実際にどのような実行コードに変換されるかは、処理系定義です。また、アドレスの値をどのようなサイズで扱うかも処理系定義です。これらは処理系のみならず、実際にプログラムを実行するプロセッサのアーキテクチャにも大きく依存します。

処理系定義の代表例　４：配列

同じ配列の要素への二つのポインタを減算した結果のサイズは、必ずしもアドレスに対して適用されるビット幅のサイズとして保証されるわけではありません。処理系定義です。C99 の言語規格 X3010:2003 (ISO/IEC9899:1999) では、ポインタ同士の減算結果のサイズの型として、<stddef.h> で ptrdiff_t という型を定義しています。

処理系定義の代表例　５：整数

符号付きの整数型を符号と絶対値による表示、2 の補数表示、1 の補数表示、のどれで表現するかは処理系定義です。したがって、たとえば、

```
if (( intVal & 0x80000000 ) == 0x80000000 ) { // 最上位ビットが 1 なら・・・
```

といった処理は、符号付き整数の最上位ビットが符号ビット（負値の場合に ’1’）であるという表現を処理系が使っている場合にしか期待通りに動作しません。

また、規格外の値をトラップ表現、または通常値での表現のどちらにするかも処理系定義です。規格外の値とは、演算結果が変数のサイズに収まらなくなる場合の値を示します。符号なしの変数の場合は、演算結果が変数の表現範囲を超えると、その変数で表現できる最大値+1 による剰余となります。たとえば符号なし 8 ビットの変数で、演算結果が 257 になった場合、8 ビットの変数で表現できる最大値 255 に 1 を加えた 256 による剰余である 1 が演算結果となります。このような動作をラップアラウンドと言います。

一方、符号付きの変数では、演算結果が変数で表現できる範囲を超えた場合にオーバーフローとなります。このとき、演算結果を符号なし変数の場合と同様に（たまたま変数に残った値として）表現する場合と、トラップ表現という特別な値で表現する場合があります。トラップ表現とは、システムが内部処理用に特別に定義している値のことで、2 の補数表現を使っている場合は最上位ビットが 1 でそれ以外が 0 である値を使います。

処理系定義の代表例 ６：ビットフィールド

組込み用 C コンパイラでは、符号なし 8 ビットのサイズでビットフィールドを使用可能としたものがあり、マイコンの機能をビットに割り振っている内蔵レジスタのアクセスによく用います。

しかし、これは処理系定義であり、特定の処理系で正常に動作したものが別の処理系でも同じように動作する保障はありません。また、ビットの並びが上位ビットからか下位ビットからかも処理系定義です。

さらに、ビットフィールドを使ったからといって実際の実行コードが操作の対象（たとえば内蔵レジスタ）に対してビットアクセスをする命令になるかどうかも処理系定義です。1 ビットのビットフィールドを使っていても、実行コードでは、そのビットを含むバイトをアクセスしたリード・モディファイ・ライトになることもあり、思わぬ不具合の原因となりえます。

処理系定義の代表例 ７：volatile 修飾型のオブジェクトへのアクセス

volatile 修飾はコンパイラの最適化を抑制する場合に使用します。たとえば、割り込み待ちをする場合、割り込みハンドラで 1 になる変数をポーリングする方法があります。

```
while( InterruptFlag == 0 ) { ; }
```

この場合、変数 InterruptFlag を 1 にする処理は、このループの中にありませんから、コンパイラは最適化を行って単なる無限ループに置き換えてしまうかもしれません。volatile 修飾子は、このような最適化が実施されないよう抑制します。

volatile 修飾したオブジェクトは処理系に未知の方法で変更されることを暗示しています。volatile 修飾したオブジェクトへのアクセスを実行コードでどのように構成するかは、処理系定義です。

処理系定義の代表例 ８：前処理指令

前処理指令に関しては、以下のような処理系定義項目があります。

- <> または "" で指定したヘッダー名の連なりを、ヘッダーまたは外部ソースファイル名に対応させる方法
- 条件付きのインクルードを制御する定数式の文字定数の値が、実行文字セット中の同一の文字定数の値に一致するかどうか
- 条件付きのインクルードを制御する定数式の単一文字の文字定数が、負の値をとることがあるかどうか
- #include 指令内の前処理トークン（マクロ展開で生成されることもある）からヘッダー名を形成する方法
- #include 処理の入れ子制限
- 文字定数または文字列定数に # 演算子があるとき、汎用文字名で始まる \ 文字の前に \ 文字を挿入するかどうか
- 非 STDC の #pragma 動作
  C99（X3010:2003（ISO/IEC9899:1999））から、C 言語が標準で持っている #pragma 指令が追加されました。これを STDC（標準 C）の #pragma 指令といいます。
  これら以外の #pragma についての動作は処理系定義です。
- 翻訳の日付と時刻がわからないときの __DATE__ と __TIME__ の定義

処理系定義の代表例 ９：その他

インライン指示やレジスタ修飾子を指定しても、それらが実際に有効になるかどうかは処理系定義です。

以上、いくつかの処理系依存項目について説明しましたが、これら以外については実際に開発に使用する（バージョンの）コンパイラのマニュアルを参照してください。

コラム：未規定の動作と未定義の動作

C 言語には，注意すべき動作に次の四つがあります。

1. **未規定の動作**
2. **未定義の動作**
3. **処理系定義の動作**
4. **文化圏固有の動作**

（詳細は C99 言語規格「ISO/IEC 9899:1999 Programming Language C」Annex J を参照）

未規定の動作と未定義の動作は似ていますが、それぞれ以下のように異なった意味の言葉です。

■未規定の動作

未規定の動作とは、文法的には正しいものの、その実行結果が処理の仕方によって複数あり得るものを言います。たとえば、関数への実引数への評価順序がこれにあたります。

```
printf("%d %d ¥n", i, i++ );
```

というコードの場合、`i` と `i++` のどちらが先に評価されるかによって表示結果が異なります。未規定の動作にどのようなものがあるかについては X3010:2003(ISO/IEC9899:1999）の J.1 で列挙されています。未規定の動作となる記述はできるかぎり避けましょう。

■未定義の動作

未定義の動作とは、C 言語の規格上定義されていないもののことです。
たとえば、0 で除算した場合の動作がこれにあたります。未定義の動作にどのようなものがあるかについては X3010:2003(ISO/IEC9899:1999）の J.2 で列挙されています。
言語の規格として定義されていませんので、決して未定義の動作となる記述をしてはなりません。使用する静的解析ツールにおいて、未定義の動作に関してどれが検出可能でどれが検出不可能かを把握しておく必要があります。