C/C++ セキュアコーディングセミナー
2010年度版

整数

JPCERT コーディネーションセンター

はじめに　～整数オーバーフローの脅威～

「CとC++言語のプログラムにおいて、整数型の取り扱いに起因する脆弱性はこれまで軽視されており、現在その数を増やしつつある」

2005年、『C/C++セキュアコーディング』 Robert C. Seacord

2007年5月にMITREが公表したレポートによると

「整数オーバーフローは過去数年のあいだトップ10に入る脆弱性であったが、今日、OSベンダのアドバイザリにおいてバッファオーバーフローに続き2番目に多い脆弱性である」

"Vulnerability Type Distributions in CVE"

URL: http://cwe.mitre.org/documents/vuln-trends/index.html

なぜ整数に関する脆弱性がなくならないのか？

- 整数を入れる箱を、無限に続く数学的整数と混同してしまっている
- 信頼できないソースから取得した値を、チェックすることなくうっかりそのまま使ってしまう：
  - ポインタ演算における配列インデックス
  - オブジェクトの長さやサイズ
  - 配列やループカウンタの範囲
  - メモリ割当て関数の引数

「整数」セクションのゴール

- コンピュータにおける整数の取り扱いについて理解を深める
- Cの「整数変換のルール」をマスターする
- 整数演算の仕組みとエラー条件を理解する
- 整数に関する脆弱性の発生メカニズムを理解し、脅威の緩和方法を身につける

よくあるパターンの整数の脆弱性

```
u_int nresp;

nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp*sizeof(char*));
    for (i = 0; i < nresp; i++) {
        response[i] = packet_get_string(NULL);
    }
}

/* OpenSSH 3.3 における整数オーバーフローの脆弱性 */
```

nresp の値が1073741824のとき、
乗算の結果はオーバーフローする！

整数の脆弱性の例 1

GNU の Bourne Again Shell (**bash**) は Bourne Shell（**/bin/sh**）と互換性のあるプログラム

- 標準のシェルと同じ構文で、ジョブ管理、コマンド行編集、履歴などの追加機能を提供
- Linux 上で最も広く普及

バージョン 1.14.6 以前の **bash** に任意のコマンドを実行させてしまう脆弱性が存在

脆弱性のある bash のソースコード

JPCERT CC®

```
static int yy_string_get() {

    register char *string;
    register int c;

    string = bash_input.location.string;
    c = EOF;

     /* If the string doesn't exist, or is empty, EOF found. */
    if (string && *string) {
        c = *string++;
        bash_input.location.string = string;
    }
    return (c);
}
```

*string は単なる charとして宣言されている

コマンドラインから入力文字を1文字づつポインタで取り出し、int型変数cに格納

文字 ÿ (値はFF)がstringから入ると…

拡張前： 255(0xFF)

符号拡張後： -1(0xFFFFFFFF)

攻撃シナリオ

10 進文字コードの 255（8 進の 377）が -c オプションを通じて bash にコマンドとして渡されると、意図せずコマンド区切り文字と解釈される。

例：

- **bash -c 'ls¥377rm -rf /'**

このコマンドは、ls と rm -rf / の 2 つのコマンドを実行する。

コマンドインジェクション攻撃につながる

とられた対策

ゼロで符号拡張されるよう型宣言を変更:

```
register unsigned char *string;
```

C++ コンパイラでは代入文 `c = *string++;` でコンパイルエラーになるため、型宣言は変えず代入文で次のように対策する方法もある:

```
c = *(unsigned char*)string++;
```

脆弱性の修正は，変数宣言をひとつ直すだけ。
つまり，変数宣言がひとつ不適切なだけで，脆弱性がつくりこまれてしまうこともある.

もくじ

JPCERT CC®

整数表現と型
型変換のルール
演算とオーバーフロー
整数に関する脆弱性の事例研究
脅威の緩和方法
セキュリティコードレビューのポイント

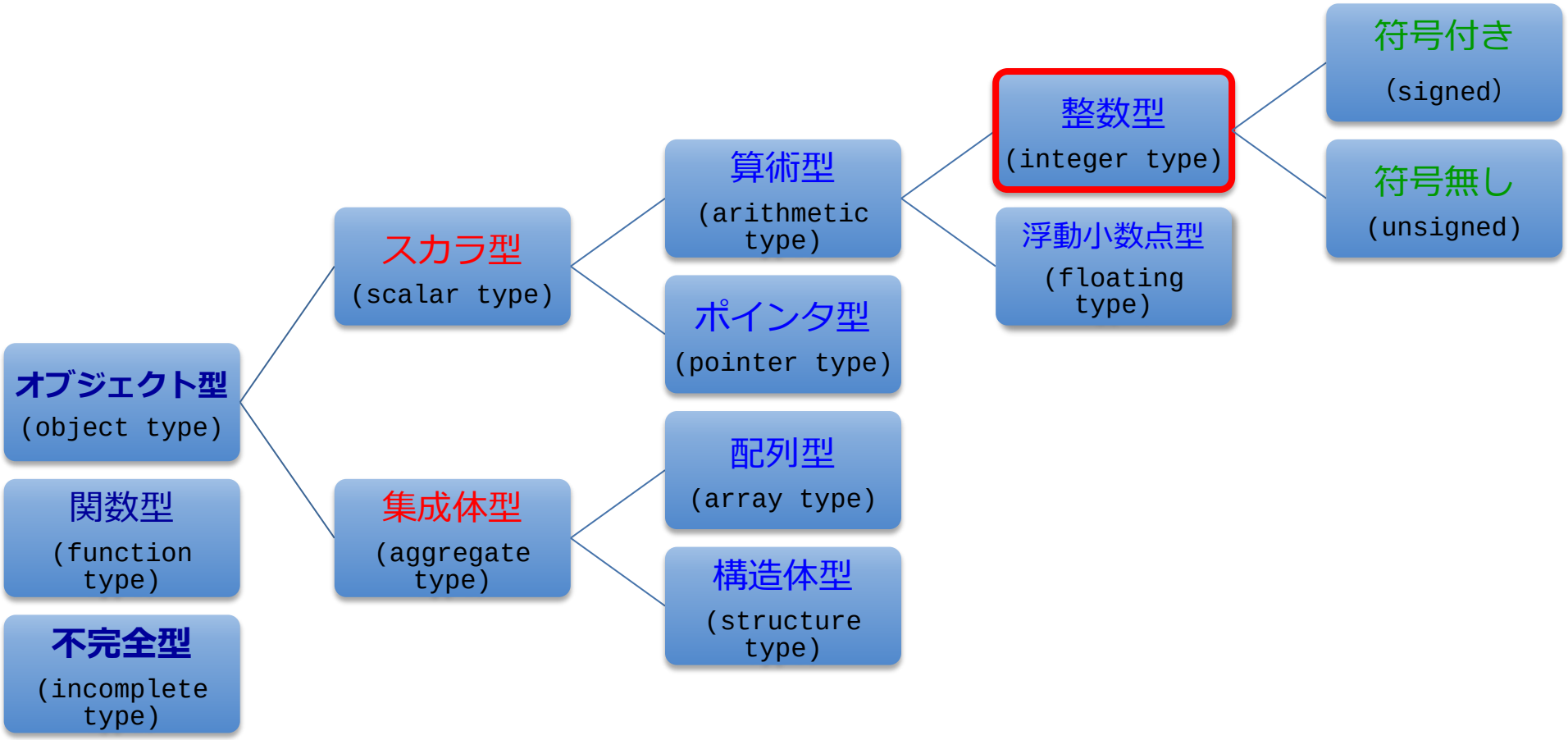
オブジェクト

オブジェクト（object）

- 実行環境中のデータを格納する記憶領域のこと
- オブジェクトの内容が値を表現する
- 「特定の型を持っていると解釈してもよい」

  「型」はそのオブジェクトをどう解釈するか、値の意味は何かといった情報を与える

Cのデータ型



本セミナーでは整数型を中心にとりあげます

整数型

標準符号付き整数型 (standard signed integer type)

- signed char (3つのchar型を総称して文字型と呼ぶ)
- short int
- int
- long int
- long long int

<stddef.h>に定義されたその他の型

- **ptrdiff_t**
  - 2 つのポインタの差を表す符号付き整数型
- **size_t**
  - **sizeof** 演算子の結果のための符号なし整数型
- **wchar_t**
  - 一意なコードを表現できる範囲の値を持つ整数型

「符号付き整数型」と「符号なし整数型」

JPCERT CC®

整数型は **signed** か **unsigned** かのいずれか

- **signed** 型にはそれに対応する **unsigned** 型がある

符号属性（signedness)という言葉で表されることもある

- MISRA-C:2004

各整数型は、異なるサイズの整数を表す

- 型とそのサイズのマッピングはアーキテクチャ依存

符号無し整数のバイナリ表現

符号無し整数の表現は単純

2を基数とする純粋なバイナリ表現として表現

2進から10進への変換

$0011\ 1011 = 2^5 + 2^4 + 2^3 + 2^1 + 2^0 = 59$

10進から2進への変換

$55 = 32 + 16 + 4 + 2 + 1$

$= (2^5) + (2^4) + (2^2) + (2^1) + (2^0)$

$= 0011\ 0111$

符号付き整数のバイナリ表現

符号付き整数は、Cの言語仕様で以下のいずれかの方法で表現される
と定められている

符号付き絶対値 (sign and magnitude)

1の補数 (ones complement)

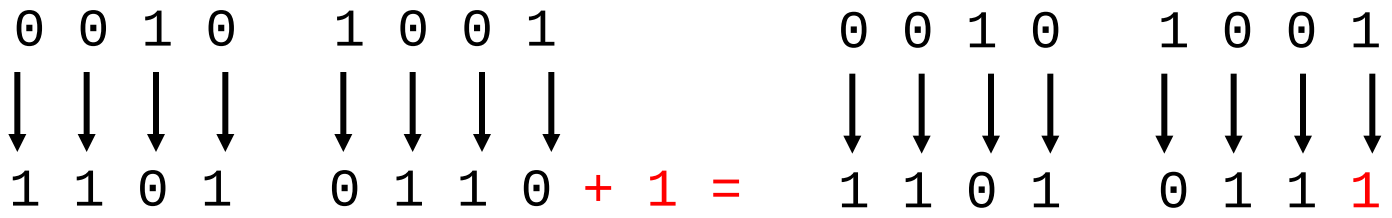
2の補数 (twos complement)

それぞれ負数を表現する仕組みが異なる

-ほとんどの処理系は2の補数表現を採用しています
-本セミナーでは2の補数表現を前提に解説しています

2の補数表現 (twos complement)

負の整数は、1の補数表現に1を加えて表す

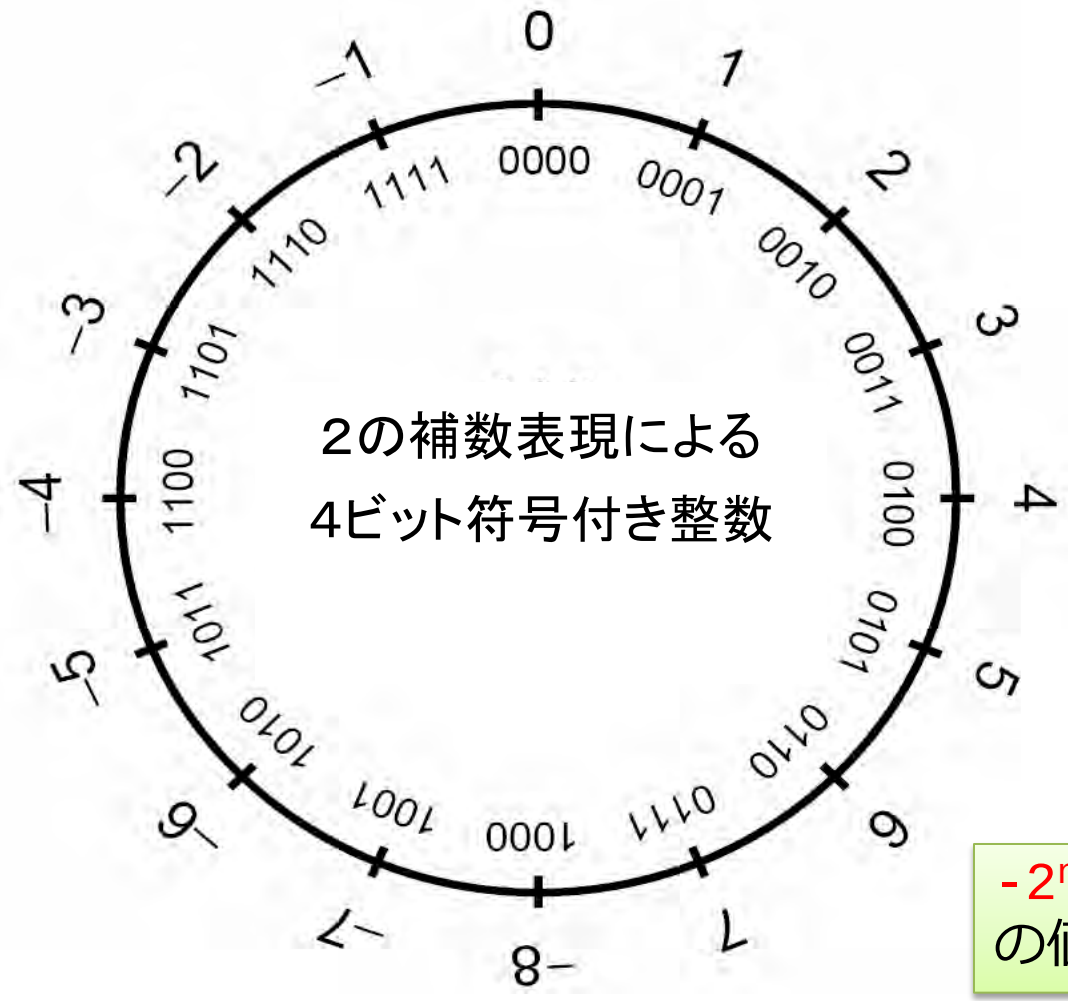


2の補数表現では、単一(正)の値で0を表現

符号は最高位ビットで表現

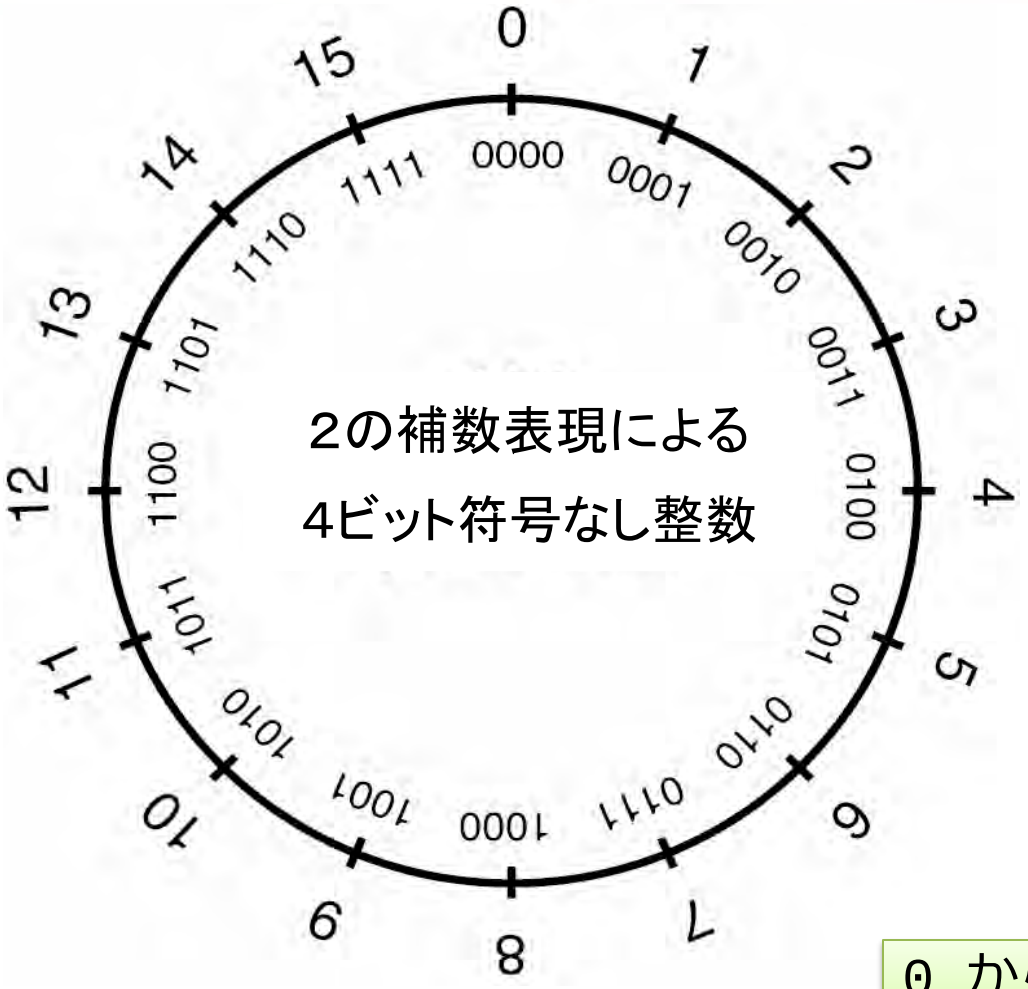
正の数の表記は符号付き絶対値表現と同じ

符号付き整数表現



-2^{n-1} から $2^{n-1}-1$ までの値を表現（2の補数）

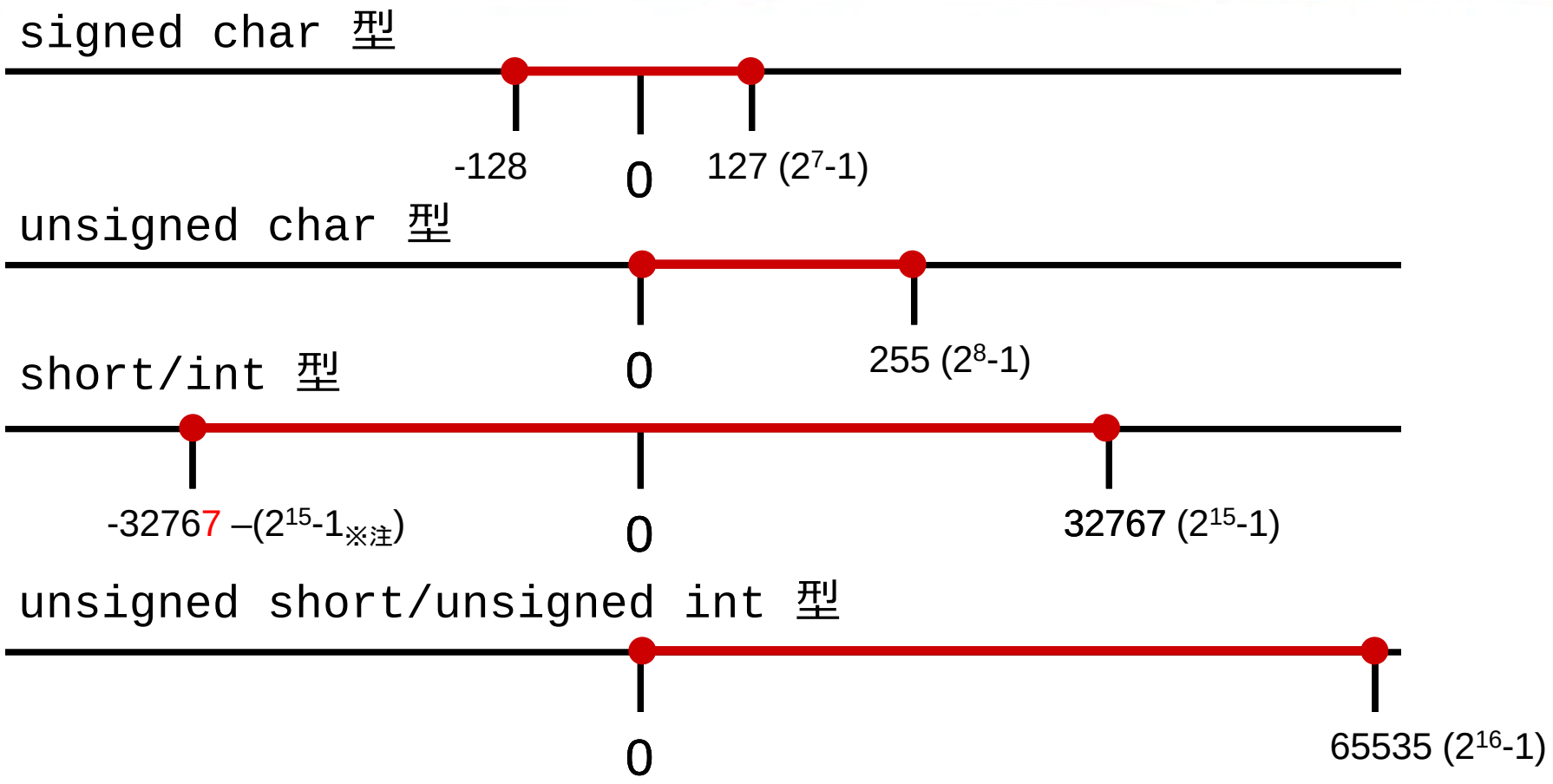
符号無し整数表現



0 から 2^n-1の値を表現

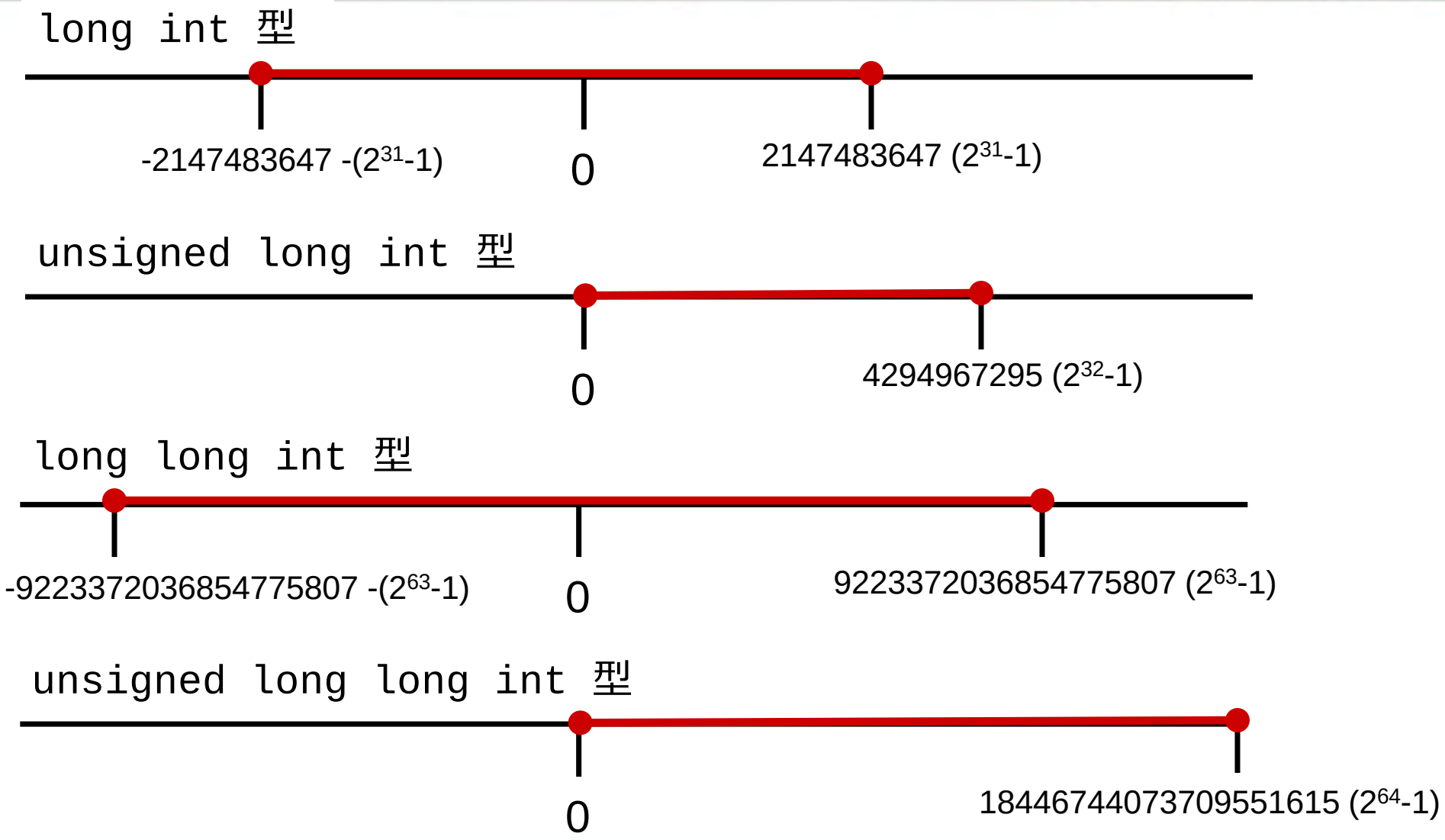
整数の範囲

言語仕様は、型の表現方法、符号の有無、割り当てるビット数に関して最小限の必要条件を定めるだけ。



※注　アーキテクチャの整数表現が符号付き絶対値表現を採用している可能性も考慮し、言語仕様では $-(2^{n-1})$ を必要条件に定めている。
最近のアーキテクチャはほとんど2の補数表現なので、$-(2^{15})$ であるが。

整数の範囲



データモデル

標準データ型に割り当てるビット幅を定義。

- ILP32—**int, long,** ポインタが32 bit

  今日の32bitコンピュータで最も一般的なデータモデル

- LP64—**longとポインタが**64 bit
- ILP64—**int, long,** ポインタが64 bit
- LLP64—**long long,** ポインタが64 bit

一般的データモデル

JPCERT CC®

データ型	iAPX68	IA-32	IA-64	SPARC-64	ARM-32	Alpha	64bit Linux, FreeBSD, OpenBSD
`char`	8	8	8	8	8	8	8
`short`	16	16	16	16	16	16	16
`int`	16	32	32	32	32	32	32
`long`	32	32	32	64	32	64	64
`long long`	N/A	64	64	64	64	64	64
ポインタ	16/32	32	64	64	32	64	64

INT00-C. 処理系のデータモデルについて理解する

標準データ型のビット幅を意識したコーディング

MISRAC-2004:ルール6.3(推奨):

基本型の代わりに，サイズ及び符号属性(signedness)を示す **typedef** をもちいなければならない.

- 処理系により型のサイズが異なる可能性のある基本型を直接使用しない

  基本型：char,int,short,long,long long,floatなど

- かわりに一目で型のサイズと符号属性がわかる型定義名を定義して使用する

```
typedef signed short int int16_t;
typedef unsigned long int uint32_t;
```

C99は基本型の最低限の大きさしか規定しない．型のサイズが異なる処理系に移植した際，プログラムは予期せぬ動作をする可能性がある.

整数オーバーフロー

```
unsigned int a;
a=0xe0000020;
a=a+0x20000020; //0x100000040はaに収まらない
                                    //整数オーバーフローが発生
```

値が最大値/最小値、つまり境界(boundary)を超えたときの動作は、符号付き整数型であるか符号無し整数型であるかにより異なる。

```
unsigned int a;
a=0;
a=a-1;   //-1はaの最小値であるゼロより小さい
         //整数アンダーフローが発生
```

符号無し整数のラップアラウンド（wrap around）

JPCERT CC®

符号無しオペランドを含む計算は，決してオーバフローしない。

C99，6.2.5 型

すなわち，結果を符号無し整数型で表現できないときは，その型で表現しうる最大値より1 だけ大きい数を法とする剰余を結果とする。

```
/* int型が16ビットの処理系の場合，左辺値には0が代入される */
uint16_t u16_var1 = 65535U;
uint32_t u32_var2;
u32_var2 = u16_var1 + 1;
```

(65535U + 1) mod 65536 の計算結果が結果の値となり、u32_var に代入される

符号付き整数のオーバーフローは「未定義の動作」

JPCERT CC®

- the value of the result of an integer arithmetic or conversion function cannot be represented
(7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.20.6.1, 7.20.6.2, 7.20.1

C99, J.2 未定義の動作

3.4.3 未定義の動作（undefined behavior）
可搬性がない若しくは正しくないプログラム構成要素を使用したときの動作，又は正しくないデータを使用したときの動作であり，この規格が何ら要求を課さないもの.

「未定義の動作」に依存するコード対し，コンパイラは無視したり，最適化して削除したり、コンパイルを中断することができる.
☞整数オーバーフローを引き起こすコードの動作は，予期せぬプログラムの動作を引き起こす可能性があるので注意！

まとめ

- 処理系が採用するデータモデルを意識してコーディング

  静的アサートを使って確認

- 符号無し整数はオーバーフローせず、ラップアラウンドする
- 符号付き整数はオーバーフローする

型変換のルール

整数の型変換

キャストにより明示的に行われることもあれば、型変換のルールに従って暗黙的に行われることもある

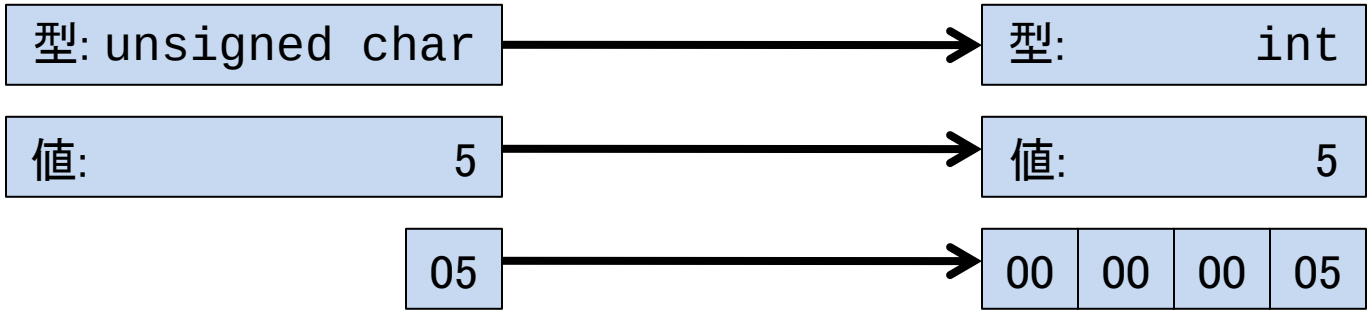
- 暗黙的な型変換が起こるのは、Cの仕様が混在する型に対する演算を許すから

型変換は、データの**欠損**や**誤った解釈**につながり、プログラムの想定しない動作につながる可能性がある

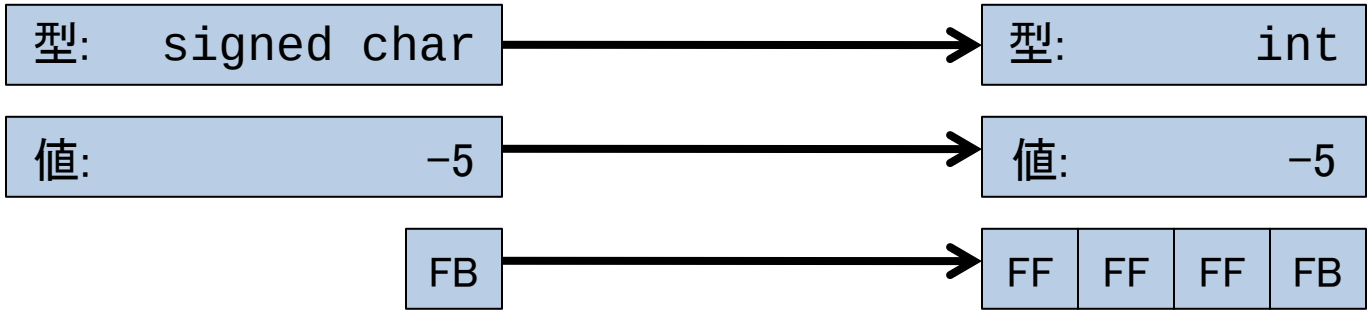
C99 が定める型変換のルール:

- 整数拡張 (integer promotion)
- 整数変換の順位 (integer conversion rank)
- 通常の算術型変換 (usual arithmetic conversion)

型変換：値が変わらない (value-preserving)

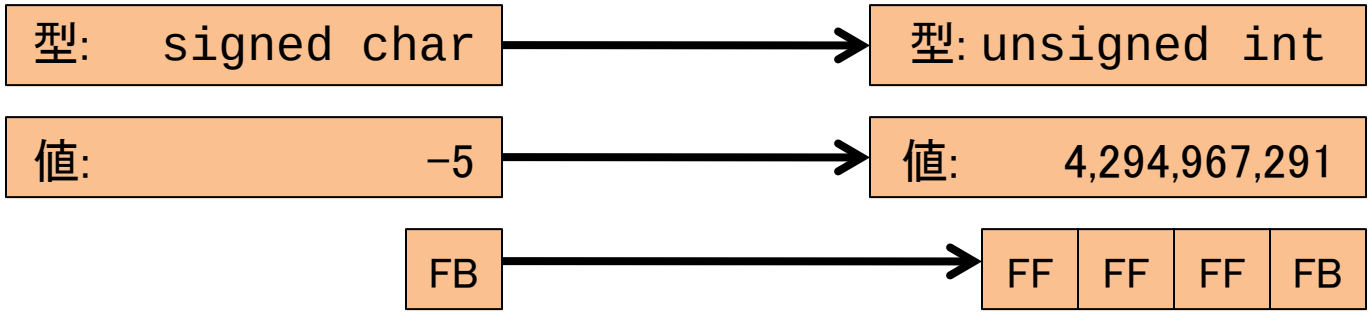


unsigned char から int への変換 (ゼロ拡張、big endian)

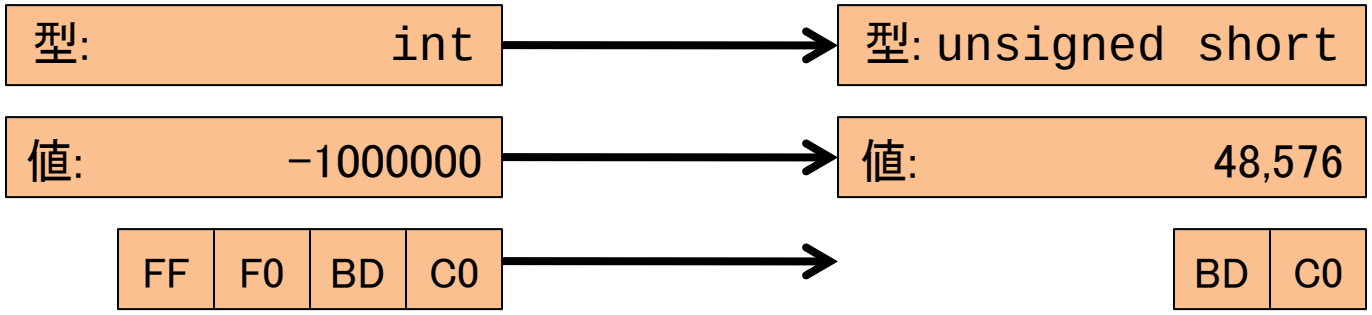


signed char から int への変換 (符号拡張、big endian)

型変換：値が変化する (value-changing)



signed char から unsigned int への変換 (符号拡張、big endian)



int から unsigned short への変換 (切り捨て、big endian)

整数拡張※1

Integer Promotions

※1「格上げ」と呼ばれることもある

整数拡張のルール

int より小さい整数型○○○○と○○○○は、まず int 型に型変換される

整数拡張の対象となる演算

- 通常の算術変換
- 引数の式
- 単項演算子(+、 -、 ~)のオペランド
- シフト演算子の両方のオペランド
- ビットフィールド

整数拡張の例

```
signed char cresult, c1, c2, c3;

c1 = 100;

c2 = 90;

c3 = -120;

cresult = c1 + c2 + c3;
```

c1 と c2 の合計は signed char 型の最大サイズを超える。

しかし、c1、c2、c3 はそれぞれintに整数拡張され、式全体は正しく評価される。

c1 の値が c2 の値に加算される。

合計は切り捨てられ、データを失うことなく cresult に格納される。

整数拡張の目的は、計算途中の値がオーバーフローして算術エラーが起こるのを防ぐことにある

整数拡張の意味

JPCERT CC®

> – 整数変換の順位がint 型及びunsigned int 型より低い整数型をもつオブジェクト又は式
> – Bool 型, int 型, signed int 型, 又はunsigned int 型のビットフィールド
> これらのものの元の型のすべての値をint 型で表現可能な場合, その値をint 型に変換する. そうでない場合, unsigned int 型に変換する. これらの処理を, 整数拡張（integer promotion）と呼ぶ.
>
> C99 6.3.1.1

符号拡張の特性

「ビットごとの補数演算子」(~) を (IA-32 の) **unsigned char** 型オペランドに適用すると、結果はかならず **signed int** 型の負の値になる。

値が 32 ビットに 0 拡張されるため

整数拡張の結果一覧

もとの型	整数拡張後の型	整数拡張の根拠
`unsigned char`	**int**	整数拡張する。もとの型の「整数変換の順位」が int 型より低いから
`char`	**int**	整数拡張する。もとの型の「整数変換の順位」が int 型より低いから
`short`	**int**	整数拡張する。もとの型の「整数変換の順位」が int 型より低いから
`unsigned short`	**int**	整数拡張する。もとの型の「整数変換の順位」が int 型より低いから
`unsigned int: 24`	**int**	整数拡張する。unsigned int のビットフィールドだから
`unsigned int: 32`	**unsigned int**	整数拡張する。unsigned int のビットフィールドだから
`int`	**int**	整数拡張しない。もとの型と int 型の順位が同じだから
`unsigned int`	**unsigned int**	整数拡張しない。もとの型と int 型の順位が同じだから
`long int`	**long int**	整数拡張しない。もとの型の順位が int 型より高いから
`float`	**float**	整数拡張しない。もとの型が整数型ではないから
`char *`	**char ***	整数拡張しない。もとの型が整数型ではないから

整数変換の順位

Integer Conversion Rank

整数変換の順位

すべての整数型は**整数変換の順位**をもつ

「通常の算術型変換」はこの順位にもとづいて行われる

「整数変換の順位」の定義

2つの符号付き整数型は、同じ表現を持つ場合であっても、同じ順位を持ってはならない

符号付き整数型の順位は、より精度の低い符号付き整数型の順位よりも高い

符号なし整数型の順位は、対応する符号付き整数型の順位と同じ

JIS X 3010 セクション6.3.1.1 論理型, 文字型及び整数型

整数変換の順位と型

順位	型
高い	`long long int, unsigned long long int`
	`long int, unsigned long int`
	`unsigned int, int`
	`unsigned short, short`
	`char, unsigned char, signed char`
低い	`_Bool`

通常の算術型変換

Usual Arithmetic Conversions

「通常の算術型変換」とは

算術演算時に異なる型のオペランドを共通の型に変換するためのルール

- 2項演算子の左右のオペランドを共通の型に合わせる
- 条件演算子 **(? :)** の第2、第3引数を共通の型に合わせる

2つのオペランドの型が異なると、共通の型に合わせる変換が生じる。

どちらか一方、もしくは両方のオペランドが変換される。

通常の算術型変換のルール

JPCERT CC®

1．両方のオペランドが同じ型を持つ場合、さらなる型変換は行わない。

2．両方のオペランドが同じ整数型（符号付き、または符号なし）を持つ場合、整数変換の順位の低い方の型を、高い方の型に変換する。

3．符号なし整数型を持つオペランドが、他方のオペランドの整数変換の順位より高いまたは等しい順位を持つ場合、符号付き整数型を持つオペランドを、符号なし整数型を持つオペランドの型に変換する。

4．符号付き整数型を持つオペランドの型が、符号なし整数型を持つオペランドの型のすべての値を表現できるならば、符号なし整数型を持つオペランドを、符号付き整数型を持つオペランドの型に変換する。

5．それ以外の場合、両方のオペランドを符号付き整数型を持つオペランドの型に対応する符号なし整数型に変換する。

両方のオペランドが同じ型を持つ場合、さらなる型変換は行わない。

`int + int` ⇒ 型変換の必要なし!

`unsigned long int * unsigned long int` ⇒ 同上

両方のオペランドが同じ整数型（符号属性が同じ）を持つ場合、整数変換の順位の低い方の型を、高い方の型に変換する。

順位が高い　　順位が低い

```
  unsigned long + unsigned int
⇒ unsigned long + unsigned long
```

ルール3

符号なし整数型を持つオペランドが、他方のオペランドの整数変換の順位より高いまたは等しい順位を持つ場合、符号付き整数型を持つオペランドを、符号なし整数型を持つオペランドの型に変換する。

順位が等しい

符号なし整数型のオペランドの型に変換

```
unsigned int + int ⇒ unsigned int + unsigned int
```

符号付き整数型を持つオペランドの型が、符号なし整数型を持つオペランドの型のすべての値を表現できるならば、符号なし整数型を持つオペランドを、符号付き整数型を持つオペランドの型に変換する。

符号つきオペランドが

符号なし整数の全ての値を表現できる

```
   long long int + unsigned int
⇒ long long int + long long int
```

それ以外の場合(符号付き整数の型が符号なし整数より順位が高いが、符号なし整数の値をすべて符号付き整数で表現できない場合)、両方のオペランドを符号付き整数型を持つオペランドの型に対応する符号なし整数型に変換する。

longもintも同じビット幅の場合、long int で unsigned int のすべての値を表現できない

```
   unsigned int + long int
⇒ unsigned long int + unsigned long int
```

クイズ：通常の算術型変換

JPCERT CC®

左オペランド	右オペランド	変換後の共通の型
int	float	
unsigned int	int	
unsigned char	unsigned short	
unsigned int	long int	
unsigned int	long long int	
unsigned int	unsigned long long int	

ILP32データモデルの場合

ルールのおさらい

- 両オペランドの型が同じなら，型変換はしない.
- 両オペランドが符号無しで大きさが異なる場合，大きい方のオペランドの型にあわせる.
- 両オペランドが符号付きで大きさが異なる場合，大きい方の型にあわせる.
- 一方のオペランドが符号無し，他方が符号付き，大きさは同じな場合，符号無しにあわせる.
- 一方のオペランドが符号無し，他方が符号付き，大きさが異なる場合，大きい方にあわせる.

符号無し整数の変換1

小さい符号なし ⇒ **大きい**符号なし

- 常に安全（値を0拡張）

```
unsigned char uc = 255;
unsigned int ui = 1024;

ui = (unsigned int)uc * ui;
```

uc の値は 0xFFから 0x000000FF に変換される

大きい符号なし ⇒ **小さい**符号なし

- 大きな値は切り捨てられる
- 下位ビットの値は保存される

```
unsigned char uc = 255;
unsigned long int ul = 1024;

ul = uc * (unsigned char)ul;
```

ul の値は 0x00000400から 0x00 に変換される

符号無し整数の変換2

符号なし ⇒ 対応する符号付き (ex. unsigned int ⇒ signed int)

ビットパターンは保持される、データの欠損はなし

最高位ビットは**符号**ビットになる

符号ビットが立っていると、符号と絶対値の両方が変化

```
unsigned int ui = 0x8D460000 /* decimal 2,370,174,976 */
printf("the value of (int)ui: %d¥n", (int)ui);



%a.out
the value of (int)ui: -1924792320
```

From unsigned	To	変換方法
char	char	ビット配列を保存し最上位ビットを符号ビットに
char	short	0 拡張する
char	long	0 拡張する
char	unsigned short	0 拡張する
char	unsigned long	0 拡張する
short	char	下位の１バイトを保存する
short	short	ビット配列を保存し最上位ビットを符号ビットに
short	long	0 拡張する
short	unsigned char	下位の１バイトを保存する
long	char	下位の１バイトを保存する
long	short	下位の１ワードを保存する
long	long	ビット配列を保存し最上位ビットを符号ビットに
long	unsigned char	下位の１バイトを保存する
long	unsigned short	下位の１ワードを保存する

データの欠損　データ解釈の誤り

符号付き整数の変換1

値が負でない符号付き整数 ⇒ 同じあるいはより大きな符号無し整数

- 値は変わらない（符号拡張）

符号付き整数 ⇒ より小さい符号付き整数

- 高位ビットは切り捨てられる

```
short sc = 0xf1f1;/* decimal 61,937*
printf(”converted to signed char: %hhd¥n", (signed char)sc);

%a.out
-15
```

符号付き整数の変換2

符号付き整数型 ⇒ 対応する符号なし整数型

- ビット配列は保存され、データは欠損しない
- 最高位ビットは、**符号ビット**としての機能を**失う**

符号付き整数の値が負でない ⇒ 値は変わらない

値が負の場合、大きい符号なし整数の値として評価

```
int i = INT_MIN;  /*-2,147,483,648 (0x80000000) */
printf(“%u¥n", (unsigned int)i);

%a.out
2147483648
```

From signed	To	変換方法
char	short	符号拡張する
char	long	符号拡張する
char	unsigned char	ビット配列を保存し、高位ビットは符号ビットの機能を失う
char	unsigned short	short に符号拡張する。short を unsigned short に変換
char	unsigned long	long に符号拡張する。long を unsigned long に変換
short	char	下位の１バイトを保存する
short	long	符号拡張する
short	unsigned char	下位の１バイトを保存する
short	unsigned short	ビット配列を保存し、高位ビットは符号ビットの機能を失う
short	unsigned long	long に符号拡張する。long を unsigned long に変換
long	char	下位の１バイトを保存する
long	short	下位のワードを保存する
long	unsigned char	下位の１バイトを保存する
long	unsigned short	下位の１ワードを保存する
long	unsigned long	ビット配列を保存し、高位ビットは符号ビットとしての機能を失う

データの欠損　データ解釈の誤り

型変換のまとめ

符号付き整数型と符号無し整数型の間で暗黙の型変換を行わない.

より大きな型からより小さな型へ暗黙の型変換を行わない.

関数の実引数や**return**式では暗黙の型変換を行わない.

全ての処理系，全ての値に関して唯一安全な整数の型変換は，符号の有無が同じで、かつより大きいサイズの型への変換のみ.

演算とオーバーフロー

整数演算とオーバーフロー

整数演算の結果、エラーや予期せぬ値が発生する可能性がある。

大部分の整数演算は、オーバーフローにつながる可能性がある。

演算子とオーバーフローの関係について考えてみよう

オーバーフローにつながる演算子

Op.	Overflow	Op.	Overflow	Op.	Overflow
+	✓	*=	✓	&	
-	✓	/=	✓	|	
*	✓	%=	✓	^	
/	✓	<<=	✓	~	
%	✓	>>=		!	
++	✓	&=		単項 +	
--	✓	|=		単項 -	✓
=		^=		<	
+=	✓	<<	✓	>	
-=	✓	>>		Etc.	

加算

2つの算術演算のオペランドの合計を求めたり、ポインタに整数を加えたりする。

両方のオペランドが算術型であれば、通常の算術型変換が行われる。

整数の加算がオーバーフローを引き起こすのは、和が割り当てられたビット数で表現できない場合。

乗算

乗算はオーバーフローを引き起こしやすい

- 小さなオペランドを乗算してもオーバーフローする可能性がある
- 乗算結果を格納する領域として、大きい方のオペランドの2倍のサイズを割当てれば防げる

乗算によるオーバーフローを防ぐには？

符号なし整数の最大値は 2^n-1

$(2^n-1) \times (2^n-1) = 2^{2n} - 2^{n+1} + 1 < 2^{2n}$

n=8で考えると、$2^8-1=256-1=255$

$(2^8-1)*(2^8-1)=255*255=65,025<2^{2*8}=2^{16}=65,536$

符号付き整数の最小値は -2^{n-1}

$-2^{n-1} \times -2^{n-1} = 2^{2n-2} < 2^{2n}$

$-2^{8-1}*-2^{8-1}=-2^7*-2^7=-128*-128=16,384<2^{16}=65,536$

アップキャスト

オペランドをそれぞれ少なくとも2倍のビット幅を持つ整数にキャストし、それから乗算する。

符号なし整数の場合

- 一段階大きい整数で上位ビットを確認
- 上位ビットが1つでも立っていればエラーを発行

符号付き整数の場合

- 全上位ビットと下位ビットの符号ビットがすべて0もしくは1であれば、オーバーフローは発生していない

誤ったアップキャストの例

```
void* AllocBlocks(size_t cBlocks) {
  // ブロックを割り当てていない場合はエラー
  if (cBlocks == 0) return NULL;

  // 結果を 64 ビットの整数にアップキャストし、
  // 32 ビットの UINT_MAX と照合して、
  // オーバーフローが発生していないことを確認する
  unsigned long long alloc = cBlocks * 16;
  return (alloc < UINT_MAX)
    ? malloc(cBlocks * 16)
    : NULL;
}
```

代入演算では、右辺式はまず右辺式の型の値として計算され、その後で左辺のオブジェクトに代入される

乗算の結果は32ビットの値となる。 結果は unsigned long long に代入されるが、乗算演算はすでにオーバーフローを起している可能性がある。

正しいアップキャストの例

```
void* AllocBlocks(size_t cBlocks) {

 // ブロックを割り当てていない場合はエラー
  if (cBlocks == 0) return NULL;

  // 結果を 64 ビットの整数にアップキャストし、
  // 32 ビットの UINT_MAX と照合して、
  // オーバーフローが発生していないことを確認する
  unsigned long long alloc;

  alloc = (unsigned long long)cBlocks * 16;
  return (alloc < UINT_MAX)

    ? malloc(cBlocks * 16)
    : NULL;
}
```

除算

32ビット(64ビットも)の整数の最小値を -1で割ると、整数オーバーフロー条件が発生する。

- 2,147,483,648 / -1 = 2,147,483,648

符号付き32bit整数として
表現できない、不正な値！

ビット単位のシフト

左シフト演算：

　　シフト式 << 加算式

右シフト演算：

　　シフト式 >> 加算式

整数拡張がオペランドに対して行われる。

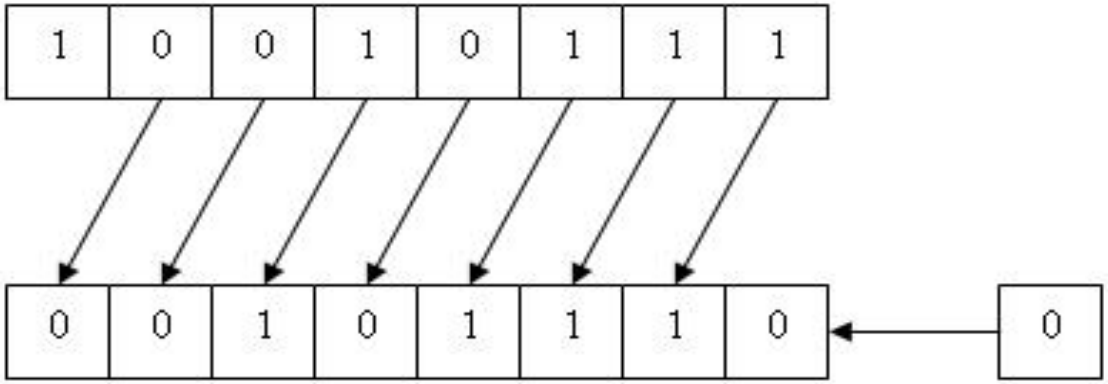
結果の型は、左オペランドを整数拡張した後の型。

右オペランドの値が以下の場合の動作は未定義

- 負の数
- 整数拡張した左オペランドの幅以上

左シフト

E1 << E2 の結果は、**E1** を **E2** ビット分左にシフトした値とする。空いたビットにはゼロを詰める。



E1 が **signed** 整数型 かつ 非負の値をもち、**E1 * 2^{E2}** が結果の型で 表現できる場合、それが結果の値となる。それ以外の場合、動作は未定義。

符号付き左シフトを安全に行うには

JPCERT CC®

signed int の左シフト

```
int si1, si2, sresult;
if ( (si1 < 0) || (si2 < 0) ||
     (si2 >= sizeof(int)*CHAR_BIT) ||
      si1 > (INT_MAX >> si2) ) {
  /* エラー */
}
else {
  sresult = si1 << si2;
}
```

左右のオペランドは非負の値であること

シフトは左オペランドのビット数より小さいこと

結果が表現可能かチェック

C99において、CHAR_BIT マクロは ビットフィールドでない最小のオブジェクトのビット数を定義する。

符号無し左シフト

E1 が式 **E1 << E2** において **unsigned** 型 を持つならば、結果の値は、**E1 * 2^{E2}** の、結果の型で表現可能な最大値より1 大きい値を法とする剰余とする。

- C99 では **unsigned** 型整数はオーバーフローせずラップアラウンドする
- ラップアラウンドの結果、予期せぬ値が生まれ、セキュリティ上の脆弱性につながることが多い

符号無し左シフトを安全に行うには

```
unsigned int ui1, ui2, uresult;
unsigned int mod1, mod2;
// wrap する場合
if ( (ui2 >= sizeof(unsigned int)*CHAR_BIT) ||
     (ui1 > (UINT_MAX >> ui2))) ) {
  /* エラー条件をハンドル */
}
else uresult = ui1 << ui2;
// modulo の場合
if (mod2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* エラー条件をハンドル */
}
else uresult = mod1 << mod2;
```

シフトが左オペランドのビット数より小さいこと

結果の値が表現可能かどうかチェック

シフトの結果がラップアラウンドしてもよい modulo の場合のチェック

右シフト

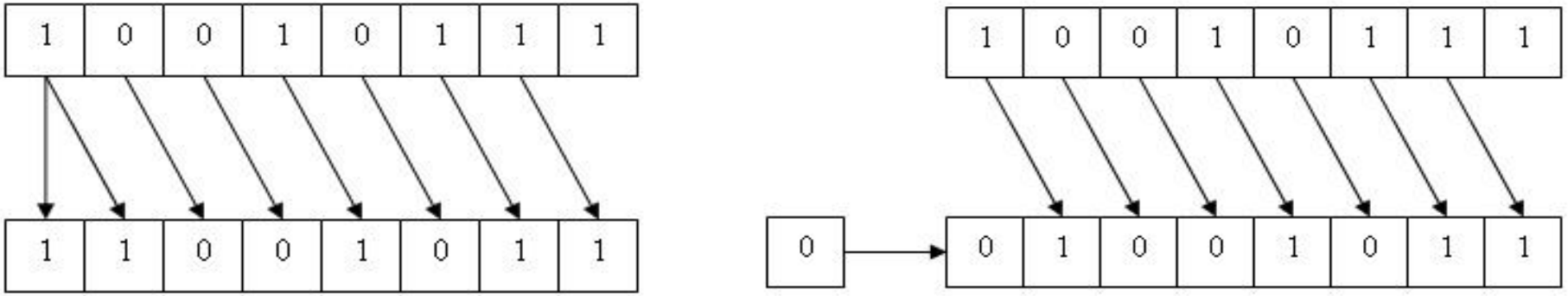
E1 >> E2 の結果は **E1** を **E2** ビット分右にシフトした値とする。

E1 が **unsigned** 型を持つ場合、又は **E1** が **signed** 型でかつゼロ以上の値をもつ場合、結果の値は、**$E1/2^{E2}$** の商の整数部分とする。

符号付き右シフト

E1 が **signed** 型でありかつ 負の値である場合、シフト後の値は 処理系定義であり、下図のいずれかとなる

- 算術 (signed) シフト
- 論理（**unsigned**）シフト



「C89 標準化委員会は、符号付き右シフト演算に符号の拡張を要求しないという点において、K&Rが認めた『処理系に自由度を持たせる』という考えを支持した。その理由は、符号の拡張を要求することで、高速なコードのスピードが低下する可能性があり、また符号を拡張するシフトを採用することの有用性が小さいから」

[C99 Rationale]

右シフトの脆弱性

右シフトが算術シフトである処理系では、符号ビットが伝播(propagate)する。

```
int stringify = 0x80000000;
char buf[sizeof("256")];
sprintf(buf, "%u", stringify >> 24);
```

- **stringify >> 24** の結果は **0xFFFFFF80** になる
- **sprintf()** 呼び出しで バッファオーバーフローが発生
- ビットを抽出したいのならば、

  ((stringify >> 24) & 0xff)

右シフトを安全に行うには

```
int si1, si2, sresult;
unsigned int ui1, ui2, result;
if ( (si2 < 0) ||
     (si2 >= sizeof(int)*CHAR_BIT) ) {
  /* エラー */
}
else sresult = si1 >> si2;
if (ui2 >= sizeof(unsigned int)*CHAR_BIT) {
  /* エラー */
}
else uresult = ui1 >> ui2;
```

右オペランドが非負であることをチェック

シフトは左オペランドのビット数よりも小さいこと

unsignedについても同様

シフトのまとめ

シフト演算子の右側のオペランドの値は，0以上，かつ左側のオペランドのビット幅未満でなければならない.

- 式をシフトする数が負である場合，またはシフトされる式の型のビット幅以上の場合は未定義の動作であるため，動作が保証されない.
- MISRA-C:2004 ルール12.8

整数に関する脆弱性の事例研究

整数の脆弱性の種類

JPCERT CC®

1. 整数オーバーフロー
   - 符号付き整数オーバーフロー *(signed integer overflow)*
   - 符号無し整数のラップアラウンド *(unsigned integer wrapping)*
2. 符号エラー *(signedness error)*
3. 切り捨て *(truncation)*
4. 非例外的(単純なロジック上のエラー)
5. 「未定義の動作」としてコンパイラに最適化されてしまう問題

過去に見つかった脆弱性の事例を見ていきましょう

整数オーバーフロー

(integer overflow)

整数オーバーフローとは

整数オーバーフロー：値がその型の最大値を超えて増加する、あるいは最小値を超えて減少する時に発生。

- 符号の有無に関係なく発生
- 実行時に発生(run-time error)
- 最小値を超えて減少することをアンダーフロー(underflow)と呼ぶこともある

符号付きのオーバーフローは、値を符号ビットに繰り越す時に発生する。

符号なしラップアラウンドは、内部表現がもはやその値を表現できなくなった時に発生する。

GOCR PNM イメージサイズの整数オーバーフロー脆弱性

```
struct{
  unsigned char *p;
  int x;  /* xsize */
  int y;  /* ysize */
  int bpp;
}
typedef struct pixmap pix;
…
void readpgm(char *name, pix *p) {
  /* read pgm */…
  png_readpaminit(fp, &inpam);
  p->x=inpam.width;
  p->y=inpam.height;
  if(!(p->p=(char *)malloc(p->x*p-
   >y)))
    F1("Error at malloc");
  for(i=0; i<inpam.height; i++){
    pnm_readpamrow(&inpam, tuplerow);
    for(j=0; j<pnam.width; j++)
      p->p[i*inpam.width+j]=sample;
  }
}
```

符号付き整数オーバーフローの実例

攻撃シナリオ

- 攻撃者は inpam.width と inpam.height に大きな整数値を与える
- malloc()の引数がオーバーフローし、誤って小さなメモリ領域が確保される
- p->pで境界外書き込みが発生し、任意のコード実行につながる

JPEG 処理ルーチンの脆弱性1

JPEGのフォーマット

コメントフィールドには、コメントフィールドの長さを示す 2 バイトの領域 **Length** がある。

Lengthの値には**Length**フィールド自身の2バイト分も含まれる。

したがって、**Data** の実際の長さは Length – 2byte した値。

JPEG 処理ルーチンの脆弱性2

問題の関数はメモリを確保するために、

- lengthの値を読み取り
  - コメント文字列のみの長さを求めるため-2をして
- その値にNULL終端文字1バイトを加えたメモリ領域を確保する

JPEG 処理ルーチンの脆弱なコード

```
void getComment(unsigned int len, char *src) {
  unsigned int size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}

int main(void) {
  getComment(1, "Comment ");
  return 0;
}
```

0 バイトの malloc() が成功する。

サイズが大きな正の値
(0xffffffff) として解釈される。

コメント長フィールドの値として 1 を持つ画像を作成することで、オーバーフローを引き起こす可能性がある。

符号無し整数のラップアラウンドの実例

なぜ整数オーバーフローが問題なのか？

乗算の結果が符号付き整数で表現できない値になると、メモリ割り当て関数は成功したように見えるが、実は必要とするより少ない領域しか確保できていない。

アプリケーションは割り当てられたバッファの終端を越えて書き込みを行い、ヒープオーバーフローが発生する可能性がある。

整数オーバーフローがバッファオーバーフローを引き起こす典型的でよくみられるパターンの脆弱性です

符号エラー

(signedness error)

次の場合に発生する

- 符号付き整数を符号無しとして解釈
- 符号無し整数を符号付きとして解釈

  - 2の補数表現において、最上位ビット(MSB)が符号の意味を持ったり、失ったりする
  - IA-32アーキテクチャでは、-1 と 2^{32}-1が誤解釈されうる

符号エラーの例

```
signed char cresult, c1, c2;
c1 = 100; /* 0x64 */
c2 = 90;  /* 0x5A */
cresult = c1 + c2; /* 0xBE */
```

cresult = -66
結果として得られるビットパターンは2の補数の8ビットとして表現しうるため、データのロスは発生せず、結果の和をリカバーすることができる。

int より小さな型の整数は、演算が行われる前に整数拡張され、 int 型もしくは unsigned int 型になる。

Linux kernel XDR における符号エラー

```
static inline u32 *
decode_fh(u32 *p, struct svc_fh *fhp) {
    int size;
    fh_init(fhp, NFS3_FHSIZE);
    size = ntohl(*p++);
    if (size > NFS3_FHSIZE)
        return NULL;
    memcpy(&fhp->fh_handle.fh_base, p, size);
    fhp->fh_handle.fh_size = size;
    return p + XDR_QUADLEN(size);
}
```

*p は攻撃者が制御できる XDR データに由来する

sizeの値が負の時、上限値チェックをスルーしてしまう

sizeがunsignedの非常に大きな値として評価される

外部から取得したデータの入力値検査の不備が、符号エラーにつながった

符号エラーの対策方針

負の長さは巨大な正の整数として解釈され、バッファオーバーフローにつながることが多い。

これは、整数 **`size`** が有効な値を持つように制限できれば回避できる：

- 範囲確認をより厳密にし、**`size`** が`0`より大きくかつ **`NFS3_FHSIZE`** より小さいことを保証する
- **`size`** を **`unsigned`** として宣言する
  - **`memcpy()`** 呼び出し時に、符号付き型から符号なし型への変換が起こらないようにする
  - そうすれば符号エラーも発生しない

切り捨て
(truncation)

切り捨てエラー (truncation error)

切り捨てエラーが発生するのは、

- ある整数をそれより小さい整数型に変換し
- 元の整数の値が小さい型の範囲に収まらない場合

元の値の下位ビットの値は保存されるが、上位ビットの値は消失する。

切り捨てエラーの例

```
unsigned char cresult, c1, c2;
c1 = 200; /* 0xC8 */
c2 = 90;  /* 0x5A */
cresult = c1 + c2;
```

c1+c2の計算は整数拡張の結果int幅で行われる。結果の値0x122は、1バイトで表現できない値

代入時に上位ビットが切り捨てられ、cresult = 34(0x22)
切捨てエラーが発生している

切り捨てが発生する脆弱なコード

```
bool func(char *name, long cbBuf) {
  unsigned short bufSize = cbBuf;
  char *buf = (char *)malloc(bufSize);
  if (buf) {
    memcpy(buf, name, cbBuf);
    if (buf) free(buf);
    return true;
  }
  return false;
}
```

cbBuf は bufSize を初期化するために使われている。また bufSize は、buf のためのメモリを確保するために使われる

long 型の cbBuf が memcpy()関数に長さとして渡される

```
void * memcpy(void *restrict s1, const void *restrict s2, size_t n);
memcpy() は s2 から s1 へ nバイトコピーする
```

コードの解説 1

long cbBuf は **unsigned short bufSize** に一時的に格納される。

- **unsigned short** の最大値は 65,535 (IA-32)
- **signed long** の最大値は 2,147,483,647

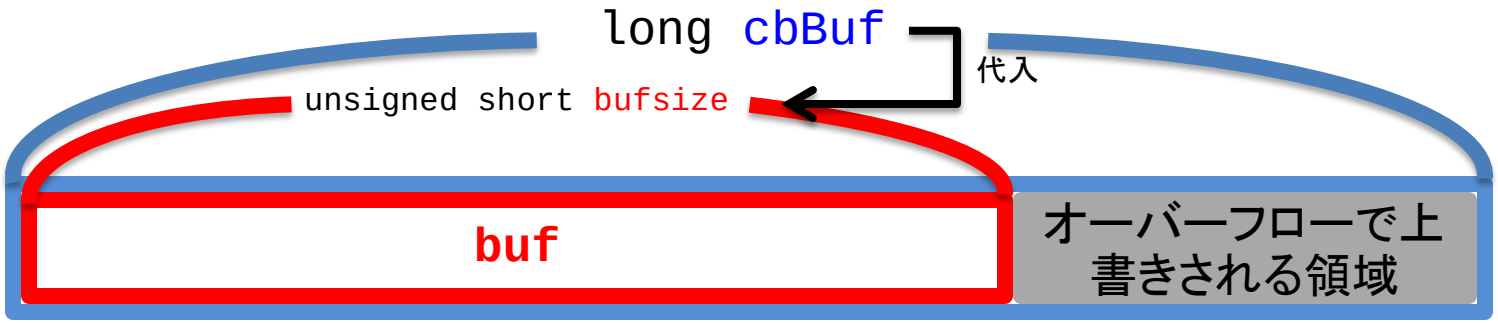
cbBuf の値が65,536～2,147,483,647の場合に切り捨てエラーが発生する。

コードの解説 2

bufSize が **malloc()** と **memcpy()** の引数に使われただけなら、単なるエラーで脆弱性ではない。

bufSize を使ってメモリを確保し、かつ **cbBuf** をサイズとして使って **memcpy()** を呼び出したのが問題。

- 1～2,147,418,112 (2,147,483,647-65,535) バイトの範囲で **buf** がオーバーフローしてしまう可能性がある。



memcpy(buf, name, cbBuf)でバッファオーバーフローが発生

単なるプログラムロジック上のエラーが問題になるケース

負の値のインデックス

JPCERT CC®

```
int *table = NULL;
int insert_in_table(int pos, int value){
  if (!table) {
    table = (int *)malloc(sizeof(int) * 100);
  }
  if (pos > 99) {
    return -1;
  }
  table[pos] = value;
  return 0;
 }
```

このコードの問題は何？

例外条件にあたらない整数エラー

pos の範囲検査が正しくない

- **変数 pos** は符号付き整数として宣言されており、関数に渡される値は正と負のどちらもありうる。
- 上限を超える正の値は **if(pos > 99)**で捕捉できるが、負の値は検査をすり抜けてしまう。

整数関連のエラーは、例外条件 (オーバーフローなど) が発生しなくても起こりうる

コンパイラによる最適化の問題

未定義の動作

定義(C99 セクション 3.4.3)

- 可搬性がないもしくは正しくないプログラム構成要素を使用したときの動作、又は正しくないデータを使用したときの動作であり、この規格が何ら要求を課さないもの。

未定義の動作の特定

- この規格の制約以外の箇所で現れる“…（し）なければならない”又は“…（し）てはならない”という要求をプログラムが守っていない場合、その動作は未定義とする。

なぜ「未定義の動作」などというものが存在するか

- 診断が困難な特定のプログラムエラーを捕捉しない許可を実装者に与えるため
- 規格合致しつつ言語拡張になり得る部分を特定するため。実装者は公式に未定義とされている動作の定義を定めることで、C 言語を拡張することができる

「MSC15-C. 未定義の動作に依存しない」の違反コード

```
#include <assert.h>
#include <limits.h>

int foo(int a) {
    assert(a + 100 > 1);
    printf("%d %d¥n", a +
   100, a);
    return a;
}
int main(void) {
    foo(100);
    foo(INT_MAX);
}
```

- 符号付き整数オーバーフローは「未定義の動作」
- コンパイラは未定義の動作に依存するコードをどう扱ってもよい
  - 完全に無視する
  - 予測不能な値を返す
  - あらかじめ規定された方法で処理する
  - 実行を中断する(診断メッセージを出力)
- コンパイラは未定義の動作に依存するコードを生成しなくてもよく、最適化することがある
- GCC4.21で-O2以上でコンパイルするとassert文は最適化により削除される

「MSC15-C. 未定義の動作に依存しない」の適合コード

```
#include <assert.h>
#include <limits.h>

int foo(int a) {
    assert(a < INT_MAX - 100);
    printf("%d %d¥n", a + 100,
a);
    return a;
}
int main(void) {
    foo(100);
    foo(INT_MAX);
}
```

- 未定義の動作に依存するコードはコードレビューが困難
  - コードがコンパイルされるのか、最適化により除去されるのかわからない
- いまコンパイラがコードを生成するからといって、次のバージョンが同じようにコードを生成するとは限らない
- コンパイラは未定義の動作に対して診断メッセージを出さなくてもよいので、コード内の未定義の動作を特定するのは困難

可搬性のあるコードを書くことが重要！

脅威の緩和方法

変数が取りうる値の範囲をチェックする

型の範囲検査の例

```
#define BUFF_SIZE 10
int main(int argc, char* argv[]){
  unsigned int len;
  char buf[BUFF_SIZE];
  len = atoi(argv[1]);
  if ((0<len) && (len<BUFF_SIZE) ){
    memcpy(buf, argv[2], len);
  }
  else
    printf("Too much data¥n");
}
```

len を unsigned として宣言するだけでは、値が0からUINT_MAX の間にあることしか保証しないので範囲制限としては不十分。

上限と下限の両方に対する明示的な検査を行うことで、memcpy()に範囲外の値が渡されないよう保証できる。

整数の入力値検査をおこなう

外部からの入力値について、上限と下限が定義可能かどうかを評価する。

- インタフェースでその制限を強制
- 内部エラーが発生してから不正な入力値を探し出すより、入力時に問題を発見・修正する方が簡単

コーディング規約を定めて以下を徹底する。

- 定数と変数を区別する
- 外部からの影響を受ける変数と、厳密に範囲制限された内部変数とを区別する

入力値検査はセキュアコーディングで最も重要な概念

強い型付け

問題：　オブジェクトサイズの表現

悪い ：

```
short total = strlen(argv[1])+ 1;
```

良い ：

```
size_t total = strlen(argv[1])+ 1;
```

これでも良い ：

```
rsize_t total = strlen(argv[1])+ 1;
```

size_t の注意点

負の数は、**size_t** のような符号なしの型に変換されると、非常に大きな正の数として表現される。

極端に大きなオブジェクトサイズは、オブジェクトのサイズが間違って計算されたことを示す場合が多い。

rsize_t は **RSIZE_MAX** より大きくならない。

アドレス空間の大きな計算機を対象とするアプリケーションでは、**RSIZE_MAX** を次のいずれか小さい方の値として定義する：

- サポートされている最大オブジェクトのサイズ
- **(SIZE_MAX >> 1)** （この上限が、正当な、しかし非常に大きなオブジェクトのサイズよりも小さい場合であっても）

rsize_t は **size_t** と同じ型、バイナリレベルで互換性がある

強い型付けの例

水の温度を(華氏)で格納する整数を次のように宣言：

- `unsigned char waterTemperature;`
- `waterTemperature` は 0 から 255 までの値を表現する符号無し8ビット値。

`unsigned char` は、

- 水（液体)の温度を表現するには合理的な型。水温は、華氏 32 度(氷点）から華氏 212 度 (沸点) までの範囲に収まる
- オーバーフローは防止できない
- ただし、無効な値 (0～31 と 213～ 255) を許可してしまう
- また、文字データ以外を表すのに使うべきでない

抽象データ型

解決法のひとつ：

- 抽象データ型を作り、**`waterTemperature`** をプライベート変数にして、ユーザが直接アクセスできないようにする
- この抽象データ型のユーザは、パブリックメソッドの呼び出しを通じてのみ、値のアクセスや更新、演算を行える
- これらのメソッドは、**`waterTemperature`** の値が有効な範囲を逸脱しないことを保証することで、型安全性を提供しないとダメ

適切に実装されれば、整数型の値域エラーは発生しない。

コンパイラによる検査を活用する

Visual C++ コンパイラによる検査

Visual C++ .NET 2003 のコンパイラは、整数の値をそれよりも小さい整数型に代入しようとすると警告 (C4244) を発する。

- /W1: __int64 型の値をunsigned int 型の値に代入しようとすると警告を発行。
- /W3, 4: 整数の型がそれよりも小さい整数の型に変換しようとすると「possible loss of data （データ欠損の可能性あり）」の警告を発行。

たとえば、/W4 では、次の代入式は警告の対象となる。

```
int main() {
    int b = 0, c = 0;
    short a = b + c;   // C4244
}
```

/W4 でコンパイルしよう

JPCERT CC®

VC++ のコンパイラオプション/RTCc

- より短い変数に値が代入されてデータの欠損が生じる時に報告するランタイムエラーチェック
- **runtime_checks** プラグマとあわせて使用する
  - `#pragma runtime_checks( "[runtime_checks]", {restore | off} )`
  - スタックポインタの破壊、ローカル配列のオーバーラン、スタックの破壊、未初期化なローカル変数の使用などのランタイムエラーもキャッチできる
- 開発ビルドで使用することで切り捨てエラーを見つける。リリースビルド時には無効になる

GCC の -ftrapv オプション

整数オーバーフローを実行時に自動検出し、プログラムをabortするメカニズム

- 符号付き整数同士の**加算**、**減算**、**乗算**をチェックしてくれる
- 事後条件に基づいたチェックを行う

使用上の注意：以下の式はチェックされない

- 符号無し整数同士の演算
- 符号付き整数と符号無し整数の混合した演算
- 符号付きから符号無しへの変換(およびその逆)
- 除算はチェックされない

事後条件とは？

符号無し整数の場合：加算の結果がどちらかのオペランドよりも小さければオーバーフローが発生している。

符号付きの整数の場合：`sum = lhs + rhs` を例にとると

- `lhs` が負の数でなく、かつ `sum < rhs` であるなら、オーバーフローが発生している
- `lhs` が負で、かつ `sum > rhs` であるなら、オーバーフローが発生している
- これ以外の場合は、オーバーフローしていない

符号付き整数の加算

符号付き 16 ビット整数の加算により発生するエラーを検出するために用いられる、gcc の実行時システム関数

```
Wtype __addvsi3 (Wtype a, Wtype b) {
  const Wtype w = a + b;
  if (b >= 0 ? w < a : w > a)
    abort ();
  return w;
}
```

加算演算を実行し、エラーが発生したかどうかを判断するために加算結果をそのオペランドと比較する

abort() は次の場合に呼び出される
- b が非負で、かつ w < a
- b が負で、かつ w > a

“GCC Internals” の arithmetic functions に詳細
http://gcc.gnu.org/onlinedocs/gccint/

安全な整数演算

安全な整数演算 1

最初にとるべき防御策は、範囲検査を徹底すること。

- 明示的 - 上限値と下限値の間に収まっているかチェック
- 暗黙的 - 強い型付け

別のアプローチは、個々の整数演算を保護すること。

- 非常に手間がかかり、実際に行うには大きなコストがかかる

信頼できないソースの影響を受ける可能性のある入力が 1つでもある
すべての整数演算について、安全な整数ライブラリを使用する。

SafeInt クラス

David LeBlanc 作成のC++テンプレートクラス

事前条件の手法を実装。処理を実行する前にオペランドの値を検査し、エラーが発生するかどうかを判断。

クラスはテンプレートとして宣言されるため、どの整数型に対しても適用できる。

関係する演算子は、添字演算子 **[]** を除き、すべてこの実装のもので置き換えられている。

MSDNの解説

- http://msdn.microsoft.com/ja-jp/library/dd570021.aspx

SafeInt の例

```
int main(int argc, char *const *argv) {
    try{
        SafeInt<unsigned long> s1(strlen(argv[1]));
        SafeInt<unsigned long> s2(strlen(argv[2]));
        char *buff = (char *) malloc(s1 + s2 + 1);
        strcpy(buff, argv[1]);
        strcat(buff, argv[2]);
    }
    catch(SafeIntException err) {
      abort();
    }
 }
```

変数 s1 と s2 が SafeInt 型として宣言されている

この + 演算子が呼び出される時には、SafeInt クラスの一部として実装されている安全なバージョンの演算子が用いられる

加算時のオーバーフロー条件

符号無し整数の加算演算の左側 (LHS: left-hand side) と右側 (RHS: right-hand side) の合計が以下に示す値を超えると、整数オーバーフローが発生する

- **UINT_MAX: unsigned int 型の加算時**
- **ULLONG_MAX: long long 型の加算時**

加算におけるオーバーフローのチェック方法

lhs と rhs の型が unsigned int かつ次の条件を満たす場合にオーバーフローが発生する。

```
lhs + rhs > UINT_MAX
```

加算による整数オーバーフローを回避するために、オペレーター+ は以下のチェックを行う：

```
lhs > UINT_MAX - rhs
```

あるいは、

```
~lhs < rhs
```

SafeInt の特徴

JPCERT CC®

SafeInt が優れている点

- **移植性が高い** - アセンブリ言語命令に依存していない
- **使い勝手がよい**
  - 演算子をインラインの式で使用できる
  - C++ の例外処理を利用する
  - exception handler クラスがテンプレートの引数
  - Visual Studio 2010 に取り込まれている

Microsoft Public License(MS-PL) でだれでも使える

- **http://www.codeplex.com/SafeInt**
- **gcc**でもコンパイルできる(もちろんVCも)
- 最新版 3.0.14 (2010年2月リリース)

安全な整数演算ライブラリを使うべきケース

以下のような整数値が、信頼できないソースによって操作される場合、安全な整数演算ライブラリを使用する。

- 構造体のサイズ
- 割り当てる構造体の数

```
void* CreateStructs(int StructSize, int HowMany) {
   SafeInt<unsigned long> s(StructSize);
   s *= HowMany;
   return malloc(s.Value());
}
```

構造体のサイズと数の積を求め、割り当てるメモリのサイズを決定する

乗算では整数オーバーフローが発生する可能性があり、バッファオーバーフローの脆弱性につながる

安全な整数演算ライブラリを使うべきでないケース

例外条件が発生しないのであれば、安全な整数演算ライブラリを使用する必要はない。

- タイトなループ
- 外部からの影響を受けない変数

```
…
char a[INT_MAX];
for (int i = 0; i < INT_MAX; i++)
a[i] = '¥0’;
…
```

まとめ

整数の脆弱性を回避するポイントは、計算機における

整数の振る舞い、取り扱いを理解する

⇒インデックス（またはその他のポインタ算術演算）、長さ、サイズ、ループカウンタとして使われている整数に注意

- `Safe Integer Library` を用いて例外条件を取り除く
- インデックスとして使われる整数値は範囲検査
- すべてのサイズと長さに **`size_t`** や**`rsize_t`** を使用

参考情報

参考情報

オンライン資料

Michael Howard. *Reviewing Code for Integer Manipulation Vulnerabilities.*
http://msdn.microsoft.com/en-us/library/ms972818.aspx

「Michael Howard's Web Log」のエントリ *Safe Integer Arithmetic in C.*
http://blogs.msdn.com/michael_howard/archive/2006/02/02/523392.aspx

Secure C Library. http://std.dkuug.dk/jtc1/sc22/wg14/www/docs/n1031.pdf

Safe Integer Operations.
https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding/312-BSI.html

blexim. *Basic Integer Overflows.* Phrack Magazine.
http://www.phrack.org/issues.html?issue=60&id=10#article

Oded Horovitz. *Big Loop Integer Protection.* Phrack Magazine.
http://www.phrack.org/issues.html?issue=60&id=9#article

Marco van de Voort. *Development Tutorial (a.k.a Build FAQ)*
http://www.stack.nl/~marcov/buildfaq.pdf

整数のセキュリティに関するCERT/カーネギーメロン大学のポータル
http://www.cert.org/secure-coding/integralsecurity.html

整数に関する CERT C のルール
INT30-C. 符号なし整数の演算結果がラップアラウンドしないようにする
INT31-C. 整数変換によってデータの消失や解釈間違いが発生しないことを保証する
INT32-C. 符号付き整数演算がオーバーフローを引き起こさないことを保証する
INT33-C. 除算および剰余演算がゼロ除算エラーを引き起こさないことを保証する
INT34-C. 負のビット数、あるいはオペランドのビット数以上シフトしない
INT35-C. 整数式をより大きなサイズの整数に対して比較や代入をする際には、事前に演算後のサイズで評価する

CERT C セキュアコーディングスタンダード

JPCERT CC®

整数に関する CERT C のレコメンデーション
INT00-C. 処理系のデータモデルについて理解する
INT01-C. オブジェクトのサイズを表現するすべての整数値に rsize_t もしくは size_t を使用する
INT02-C. 整数変換のルールを理解する
INT03-C. セキュアな整数ライブラリを使用する
INT04-C. 信頼できない入力源から取得した整数値は制限する
INT05-C. 可能性のあるすべての入力を処理できない入力関数を使って文字データを変換しない
INT06-C. 文字列トークンを整数に変換するには strtol() 系の関数を使う
INT07-C. 数値には符号の有無を明示した文字型のみを使用する
INT08-C. すべての整数値が範囲内にあることを確認する
INT09-C. 列挙定数が一意の値に対応することを保証する
INT10-C. % 演算子を使用する際、結果の剰余が正であると想定しない
INT11-C. ポインタ型から整数型への変換やその逆の変換は注意して行う
INT12-C. 式中で使用される単なるintのビットフィールドの型について想定しない
INT13-C. ビット単位の演算子は符号無しオペランドに対してのみ使用する
INT14-C. 同じデータに対してビット単位の演算と算術演算を行わない
INT15-C. プログラマ定義の整数型に対する書式指定入出力には、intmax_t もしくは uintmax_t を使用する
INT16-C. 符号付き整数の表現形式を想定しない
INT17-C. 処理系に依存しない方法で整数定数を定義する

MISRA-C:2004　整数関連のルール

JPCERT CC®

6.10 算術型変換

10.1 次の条件に該当する場合，整数型の式の値を異なる潜在型に暗黙的に変換してはならない.
a) 同じ符号属性 (signedness) をもつより大きな整数型への変換でない場合
b) 式が複合式である場合
c) 式が定数でなく，関数の実引数である場合
d) 式が定数でなく，**return** 式である場合

10.2 次の条件に該当する場合，浮動小数点型の式の値を異なる型に暗黙的に変換してはならない.
a) より大きな浮動小数点型への変換でない場合
b) 式が複合式である場合
c) 式が定数でなく，関数の実引数である場合
d) 式が定数でなく，**return** 式である場合

10.3 整数型の複合式の値は，式の潜在型と同じ符号属性 (signedness) をもつ，より小さな型へのキャストだけが許される.

10.4 浮動小数点型の複合式の値は，より小さな型へのキャストだけが許される.

10.5 ビット単位の演算子**~**及び**<<**が，潜在型の **unsigned char** 又は **unsigned short** であるオペランドに適用される場合，その結果は，そのオペランドの潜在型へ直ちにキャストさせる.

10.6 すべての符号なし型の定数には，接尾語 "**U**" を付けなければならない.

MISRA-C:2004　整数関連のルール

JPCERT CC®

6.12 式
12.1 式中における C 言語の演算子優先順位規則への依存は，極力制限すべきである.
12.2 式の値は，規格が認めるどのような順序で評価されようとも，同じでなければならない.
12.3 **sizeof** 演算子は，副作用がある式に用いてはならない.
12.4 論理演算子 **&&** 又は **||** の右側のオペランドには，副作用があってはならない.
12.5 論理演算子 **&&** 又は **||** のオペランドは，一次式でなければならない.
12.6 論理演算子 (**&&**, **||**, **!**) のオペランドは，実質的なブール型になるべきである．実質的なブール型である式は，(**&&**, **||**, **!**) 以外の演算子のオペランドとして用いるべきではない.
12.7 ビット単位の演算子は，潜在型が符号付きのオペランドに対して適用してはならない.
12.8 シフト演算子の右側のオペランドの値は，0以上，かつ，左側のオペランドの潜在型のビット幅未満でなければならない.
12.9 単行マイナス演算子を，潜在型が符号なしの式に用いてはならない.
12.10 カンマ演算子は，用いてはならない.
12.11 符号なし整数定数式の評価では，ラップアラウンドが発生しないようにすべきである.
12.12 浮動小数点数の潜在的なビット表現は，用いてはならない.
12.13 インクリメント (**++**) 演算子及びデクリメント (**--**) 演算子は，式中で他の演算子と混在させるべきではない.

MISRA-C:2004　整数関連のルール

6.6 型
6.1 単なる **char** 型は，文字データの格納及び使用に限って用いなければならない.
6.2 **signed char** 型及び **unsigned char** 型は，数値データの格納及び使用に限って用いなければならない.
6.3 基本型の代わりに，サイズ及び符号属性 (signedness) を示す **typedef** を用いなければならない.
6.4 ビットフィールドは，**unsigned int** 型又は **signed int** 型だけで定義しなければならない.
6.5 **signed int** 型のビットフィールドの幅は，2ビット長以上でなければならない.

MISRA-C研究会『組込み開発者におくる MISRA-C:2004』日本規格協会, 2006

～整数に関する脆弱性をつぶすために～

セキュリティコードレビューのポイント

ポイント1: 符号属性の変化する変換に注意する

JPCERT CC®

size_t や unsigned int 型の引数をとる関数に signed 型変数(ユーザ入力の影響を受ける)を渡していないかチェック

- read(),recvfrom(),memcpy(),memset(),bcopy()
- 負の入力値が可能だと、関数は非常に大きな正の値と解釈し、攻撃可能な状況が発生しうる

ネットワークやファイルから直接 length パラメータを取得する箇所をチェック

- length を signed 型変数として解釈するなら、負の値が入力された時の影響を考慮なくてはダメ

関数をレビューする際、引数の型と、呼び出し時に渡される変数の型をチェックし、何が起こるか正確に把握する(変換テーブルをチェックする)

ポイント2: 符号拡張に注意する

JPCERT CC®

`signed char`, `signed short`, それらの型へのポインタを操作するコードに着目

- `length` の値を使って文字列を操作するコードや、パケットを処理するコードなど
- `int` 以上の `signed` 型への変換、`unsigned int` への変換において符号拡張(`sign extension`)が発生する

バイナリのアセンブリに `movsx` 命令を探すのも効率的

符号拡張とゼロ拡張

JPCERT CC®

符号拡張の例

```
unsigned int l;
char c=5;
l=c;

mov    [ebp+var_5], 5
movsx eax, [ebp+var_5]

mov    [ebp_var_4], eax
```

ゼロ拡張の例

```
unsigned int l;
unsigned char c=5;
l=c;

mov    [ebp+var_5], 5
xor    eax, eax
mov    al, [ebp+var_5]
mov    [ebp+var_4], eax
```

ポイント3：切り捨ての発生がないかチェックする

整数値が char や short など小さな整数型に変換されるコードに切り捨てが発生しやすい

- 小さな整数型を使って length の値をトラックしたり演算結果を格納しているような箇所に注目
- ネットワークの処理を行うコードの、構造体定義のなかの変数などに多くみられる

ポイント4： 整数値の比較を行うコードは要注意

比較を行うコードに注目する

- メモリ割り当て、配列のインデックス、コピー操作を保護するために行われる比較演算に特に注意
- ことなる型同士の比較では、整数拡張(integer promotion)や通常の算術型変換(usual arithmetic conversion)がまずオペランドに対して適用されるので、これらの変換によって値が変化してしまう可能性を考慮したコーディングがされているかをチェック
- 注目すべき比較や演算の値を追いかけ、関数呼び出しに利用される過程で無効な値にならないかどうかをチェック
- sizeof()やstrlen()の結果との比較では、オペランドがunsignedに変換されうることに注意

ポイント5: sizeof演算子は意図した結果を返すか？

メモリ操作を行う関数の引数で、sizeofの引数にポインタを渡しているコードは疑う

- 式中のsizeof演算子は、オペランドがunsignedとして解釈される状況を生むので注意

整数オーバーフローを引き起こさないための注意ポイント

整数オーバーフローを防ぐ方法

「これをやれば絶対に整数オーバフローを防げる」という決定打はない！

1.unsigned 型をつかう

- 特にメモリを割り当て、添字を使って配列を走査するコードでは unsigned 型を使えば、上限だけレンジチェックすれば良い

2. 特定の処理系を想定したコードを書かない

- int, char, size_t など標準データ型は、処理系によって定義が異なることを想定する（bit幅、charがデフォルトでsignedかunsignedかなど）
- とくに、コードが32-bitと64-bitの両方でコンパイルされる場合は注意

3. 信頼できないソース(ユーザ)から得た数値は範囲内にあることをチェックして使う

「INT04-C. 信頼できない入力源から取得した整数値は制限する」から抜粋した脆弱なコード

```
int create_table(size_t length) {
  char **table;
  if (length > SIZE_MAX/sizeof(char *)){
    /* handle overflow */
    return -1;
  }
  size_t table_length = length*sizeof(char *);
  table = (char **)malloc(table_length);
  if (table == NULL) {
    /* Handle error condition */
    return -1;
  }
  /* ... */
  return 0;
}
```

ユーザが制御可能なlengthの値を直接つかっているため，想定外に大きなメモリ領域を割り当てさせられたり，malloc()を失敗させられる可能性がある．

lengthの範囲が[1, MAX_TABLE_LENGTH]に収まっているかチェックしてから値を使うべきだった．

セキュアな実装例は、CERT Cセキュアコーディングスタンダードの「INT04-C」を確認しよう！

4.メモリの割り当て・参照に使う値はチェックしてから使う

- 最終チェック後に値をいじらない(演算しない)
- 特に signed 型をつかったメモリ割り当てなど、値が暗黙的に unsigned 型に変換される場合に注意

5. コンパイラの警告メッセージはしっかり吟味

5.1.Microsoft Visual Studio CL

警告レベルは最大でコンパイル /W4

整数関連の警告にはとくに注意

- C4018 - signed 型と unsigned 型の数値を比較するには、コンパイラで signed 型の値を unsigned 型に変換する必要がある
- C4244 – 整数型がより小さな整数型にされ、データが失われた可能性がある
- C4389 - 演算で符号付きの変数と符号なしの変数が使用され、データが失われた可能性がある

開発ビルドでは /RTCc を使う。

- 値が小さなデータ型に代入されて切り捨てが発生する場合をランタイムエラーにできる。
- オーバーヘッドが大きく、リリースビルドや /O 最適化と一緒につかえない

5.2.GCC

整数関連のオプションをつけてコンパイルする

- `-Wconversion` – 問題のありそうな型変換について警告する。`signed` 型定数とunsigned型の暗黙的変換など。
- `-Wsign-compare` – signed値がunsigend値に変換されて比較演算が誤った結果を生み出す場合に警告

`-ftrapv` でランタイムエラーチェックをオンにする。

6. 整数変換のルールを理解し(覚え)た上で、情報の損失が発生するような型変換を行わない。

- 符号付き整数型と符号無し整数型の間で暗黙の型変換を行わない.
- より大きな型からより小さな型へ暗黙の型変換を行わない.
- 関数の実引数や**return**式で暗黙の型変換を行わない.
- 整数型と浮動小数点型の間で暗黙の型変換を行わない.

7．オーバーフローが発生しうる演算について、事前条件/事後条件をチェックする

- 演算子にはオーバーフローを引き起こすものとそうでないものがある。(ex. +, -, *, /, ++, --, +=, -=, *=, /=, <<=, >>=, <<, >>, unary-)
- 自分でチェックできない場合、SafeInt などのライブラリを使う