#05
文字列の基本
2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学　先端理工学部

もくじ

今回 (#05) の内容

小テーマ: 文字列の基本

第 7 回：文字コードと制御文字
第 8 回：文字列データの表現

重要概念リスト

- getchar() **関数**
- EOF
- **文字の拡張表記**
- **文字列**, **文字列リテラル**
- **文字列リテラルは書き換え不可**
- **ナル文字** '\0'
- **ナル文字による文字列の終端**
- printf **での** %s **書式指定**
- **空文字列**
- scanf, **バッファオーバーラン**, **コンピュータウイルス**
- gets **を使っては駄目**. fgets **を使う**.

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(**作成したファイル類は** manaba **に提出**)

第 8-5 節　入出力と文字

□ getchar 関数と EOF p.244

int getchar(void)

標準入力より 1 文字を入力して返す

- **入力の終了または読み込みエラー時は EOF を返す**

EOF (オブジェクト形式のマクロ)

- **ファイル終端を表す** (EOF : End Of File)
- **負の値**

List 8-8 (**部分**) : **標準入力から標準出力へコピー**

```
int main(void)
{
  int ch;
  while ((ch = getchar()) != EOF)
    putchar(ch);
  return 0;
}
```

EOF の発生タイミング

★重要★

入力元がファイルの場合：
　ファイルの終わりで発生

入力元がキーボードの場合：
Ctrl-D **押下で** EOF **発生** (WSL/Linux/macOS/Unix **系 の場合**)

- **コントロールキーを押しながら** D **を押す**

注意：Ctrl-Z **ではない**

- Ctrl-Z **は「サスペンド」**
- **実行中のプログラムを一時中断してシェルに戻る**
- **そのプログラムはまだ実行中 (終了していない)**
- **元に戻るには** `fg` **コマンド**

生 Winodows **の場合**：Ctrl-Z **押下で** EOF **発生**

□ 入力から出力へのコピー p.245

List 8-8 (**部分**)

```
while ((ch = getchar()) != EOF)
  putchar(ch);
```

解説

1. `(ch = getchar())`
   getchar **関数を用いて標準入力より** 1 **文字を読む**
   読んだ文字を変数 ch **に代入**
   この式の値は代入した値
2. `while (... != EOF)`
   代入した値が EOF **文字 (ファイル終端) なら** while **文を終了**
   EOF **文字でなければ以下を実行**
3. **読んだ文字 (変数** ch **に保持) を標準出力へ**

□ 数字文字のカウント p.246

ファイル (標準入力) から次々と文字を読み各数字の出現回数をカウント
List 8-9 (**部分**; **読み込み＆勘定部**)

```
int main(void)
{
  int ch;
  int cnt[10] = {0}; /* 数字文字の出現回数 */
  while ((ch = getchar()) != EOF) {
    switch (ch) {
    case '0' : cnt[0]++; break;
    case '1' : cnt[1]++; break;
    case '2' : cnt[2]++; break;
    case '3' : cnt[3]++; break;
    case '4' : cnt[4]++; break;
    case '5' : cnt[5]++; break;
    case '6' : cnt[6]++; break;
    case '7' : cnt[7]++; break;
    case '8' : cnt[8]++; break;
    case '9' : cnt[9]++; break;
    }
  }
...
```

数字文字のカウント (つづき)

List 8-9 (**部分**; **つづき**; **表示部**)

```
...
  puts("数字文字の出現回数");
  for (int i = 0; i < 10; i++)
    printf("'%d':%d\n", i, cnt[i]);
  return 0;
}
```

数字文字のカウントの実行例 (キーボード入力)

実行例

```
3.14Hello1592world6535
8979You3238have462mail.
[Ctrl-D]
数字文字の出現回数
'0': 0
'1': 2
'2': 3
'3': 4
'4': 2
'5': 3
'6': 2
'7': 1
'8': 2
'9': 3
```

数字文字のカウントの実行例 (ファイル入力)

ファイル data0809-1.txt **の内容**

```
3.14Hello1592world6535
8979You3238have462mail.
```

実行例 (実行ファイルは list0809 **とする)**

```
$ ./list0809 < data0809-1.txt
数字文字の出現回数
'0': 0
'1': 2
'2': 3
'3': 4
'4': 2
'5': 3
'6': 2
'7': 1
'8': 2
'9': 3
```

リダイレクト機能を使用

- **実行プログラムの標準入力をファイルに切り替え**

バッファリングとリダイレクト (1/2) p.247

Q. List 8-8 で文字を 1 つ読むごとに表示されないのはなぜ?

- Enter キーを押してから表示が始まる (List 8-9 も同様)

(部分再掲) List 8-8

```
while ((ch = getchar()) != EOF)
  putchar(ch);
```

A. バッファリング (buffering) が行われているから

- まとまった量になるまで読み貯める
- 一杯になったら処理プログラムに入力データを渡す

バッファ：入出力データを一時的に貯めておくメモリ
バッファリング：入出力データを一時的に貯める入出力効率化法
バッファリング 3 種

- 完全バッファリング：バッファが一杯になるまで貯める
- 行バッファリング：行の終わりがくるまで貯める
- 無バッファリング：貯めずに即座に入出力処理にうつる

バッファリングとリダイレクト (2/2) p.247

リダイレクト (redirection)：**標準入出力の切り替え機能**

- C **言語の機能ではないです**
- Unix **のシェルや** Windows **コマンドラインの機能です**

例:

```
$ ./list0809 < in.txt > out.txt
```

`< in.txt`：**標準入力のリダイレクト**

- **キーボード (無指定時) から読む代わりに**
- **ファイル** in.txt **から読むよう切り替え**
  ※事前に入力データをファイルに作っておける

`> out.txt`：**標準出力のリダイレクト**

- **画面 (無指定時) へ書き込む代わりに**
- **ファイル** out.txt **へ書き込むように切り替え**
  ※実行結果をファイルに記録できる

プログラム出力を (ファイルを介して) 別プログラムの入力に出来る

□ 文字コードと数字文字 p.248

C 言語での文字

- **非負の整数値**
- **各文字に非負整数値の文字コードが対応**

実行環境により文字コード体系は異なる場合あり

- JIS X0201 (**いわゆる** JIS **コード**)
  7 ビット及び 8 **ビットの情報交換用符号化文字集合**
- ASCII
  American Standard Code for Information Interchange
- EBCDIC
  Extended Binary Coded Decimal Interchange Code

JIS X0201 **文字コード表**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				0	@	P	`	p				ー	タ	ミ		
1			!	1	A	Q	a	q			。	ア	チ	ム		
2			"	2	B	R	b	r			「	イ	ツ	メ		
3			#	3	C	S	c	s			」	ウ	テ	モ		
4			$	4	D	T	d	t			、	エ	ト	ヤ		
5			%	5	E	U	e	u			・	オ	ナ	ユ		
6			&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
7			'	7	G	W	g	w			ァ	キ	ヌ	ラ		
8			(	8	H	X	h	x			ィ	ク	ネ	リ		
9			)	9	I	Y	i	y			ゥ	ケ	ノ	ル		
A			*	:	J	Z	j	z			ェ	コ	ハ	レ		
B			+	;	K	[	k	{			ォ	サ	ヒ	ロ		
C			,	<	L	¥	l	|			ャ	シ	フ	ワ		
D			-	=	M	]	m	}			ュ	ス	ヘ	ン		
E			.	>	N	^	n	~			ョ	セ	ホ	゛		
F			/	?	O	_	o				ッ	ソ	マ	゜		

文字 0：**文字コード** 0x30
文字 1：**文字コード** 0x31
文字 2：**文字コード** 0x32
…
文字 9：**文字コード** 0x39

文字 A：**文字コード** 0x41
文字 B：**文字コード** 0x42
文字 C：**文字コード** 0x43
…

数字に対する switch/if 文の書き方：良い例/悪い例

書き方 A (やっちゃだめ)
可搬性なし (文字コード体系依存)

```
switch (ch) {
case 48 : cnt[0]++; break;
case 49 : cnt[1]++; break;
case 50 : cnt[2]++; break;
case 51 : cnt[3]++; break;
case 52 : cnt[4]++; break;
case 53 : cnt[5]++; break;
case 54 : cnt[6]++; break;
case 55 : cnt[7]++; break;
case 56 : cnt[8]++; break;
case 57 : cnt[9]++; break;
}
```

書き方 B (やっちゃだめ)
可搬性なし (文字コード体系依存)

```
if (ch>=48 && ch<=57)
  cnt[ch - 48]++;
```

書き方 C (こう書く)
☆可搬性あり☆

```
if (ch>='0' && ch<='9')
  cnt[ch - '0']++;
```

Q&A：数字に対する switch/if 文の書き方

文字コード表に書いてるんだから
`if (ch >= 48 && ch <= 57))  ...`
でいいのでは?

どの文字コード体系を使っているか
がそもそも事前に分からんのです.

めんどうです

48 **と** '0' **の違いだけなので労力は変わらないです.**
ちょっとのことで可搬性あがるよ.

めんどうです

プロフェッショナルはそうするものです

C 言語における数字に対応する整数

数字 '0', '1', '2', ..., '9' に対応する整数は 1 づつ増えてゆく

- **C 言語の規約**
- **文字コード体系に関わらず '5' − '0' は必ず 5 になる**

List 8-11 (**数字カウントの別実装**)

```
#include <stdio.h>
int main(void)
{
  int ch;
  int cnt[10] = {0}; /* 数字文字の出現回数 */
  while ((ch = getchar()) != EOF) {
    if (ch >= '0' && ch <= '9')
      cnt[ch - '0']++;
  }
  puts("数字文字の出現回数");
  for (int i = 0; i < 10; i++)
    printf("'%d':%d\n", i, cnt[i]);
  return 0;
}
```

□ 拡張表記 p.250

文字列や文字をソースコードで表す方法

- **文法上の制約で直接書けない文字も書けるように**
- **キーボードから直接入力できない文字も書けるように**

Q. **文字 ’ はどうやってソースコードに書く？** `ch=’’’;` **で** OK?
A. **それはエラーになるよ**. `ch=’\’’;` **としてね**

文字列リテラルでの表記

二重引用符を文字列の前後に書く

- **二重引用符**：**拡張表記** `\"` **で表す**
- **単一引用符**：**’ または拡張表記** `\’` **で表す**
- **他にもあり**

使用例

- `char *p = "ABC";`
- `printf("Say \"Hello!\" to %s\n", who);`

拡張表記 (つづき)

文字定数での表記

クオート ’ を文字の前後に書く

- 単一引用符 : ’ で表す
- 二重引用符 : " または 拡張表記 \" で表す
- 他にもあり

使用例

- `char ch = ’A’;`
- `char ch_quote = ’\’’;`

拡張表記の一般的な規則

単純拡張文字

- **バックスラッシュを前置して 1 文字を表す記法**
- **制御文字, 引用符など**

\\	**逆斜線文字 \ (バックスラッシュ)**
\?	**疑問符 ?**
\’	**単一引用符 ’**
\"	**二重引用符 "**

- **例**: `’\\’`
- **例**: `"Say \"Hello!\""`

バックスラッシュ: **文字 **
プログラミング環境により円記号 ¥ の場合あり

拡張表記の一般的な規則 (つづき)

単純拡張文字 (つづき; 制御文字)

`\a`	**警報** (alert)	**ベルまたは画面フラッシュ**
`\b`	**交代** (backspace)	**カーソルを 1 文字前に移動**
`\f`	**書式送り** (formfeed)	**改ページしてページ先頭へ**
`\n`	**改行** (newline, LF)	**改行して行頭へ**
`\r`	**復帰** (carrige return, CR)	**行頭へ**
`\t`	**水平タブ** (horizontal tab)	**次の水平タブ位置へ**
`\v`	**垂直タブ** (vertical tab)	**次の垂直タブ位置へ**

- **例** : `"Hello world\n"`

拡張表記 (つづき)

16 進拡張表記

\x*hh*	*hh* は任意の桁数の 16 進数	16 進数で *hh* の値を持つ文字

- 例：'\x31' (10 進数では 49)

8 進拡張表記

ooo	*ooo* は 1 から 3 桁の 8 進数	8 進数で *ooo* の値を持つ文字

- 例：'\061' (10 進数では 49)

第 9-1 節　文字列とは

文字列：理解のポイント

★★★ メモリ上にどうデータが配置されるのかを完全に理解する ★★★

文字列の理解にはこれが必須

ソースコードの字面だけであれこれ想像してもたいてい間違える

★重要★

□ 文字列リテラル　p.256

文字の並びを二重引用符""で囲んだもの

- `"ABC"`
- `"Say \"Hello\""`
- **定数のようなもの**

ナル文字 (null character) **が末尾に付加される**

例 1 : **文字列リテラル** `"123"`

1	2	3	\0

例 2 : **文字列リテラル** `"AB\tC"`

A	B	\t	C	\0

例 3 : **文字列リテラル** `"abc\0def"`

a	b	c	\0	d	e	f	\0

例 4 : **文字列リテラル** `""`　(**空文字列**)

\0

□ 文字列リテラルの大きさ p.256

文字列リテラルの大きさ $=$ **文字数** $+ 1$

- $+1$ **は末尾に付加されるナル文字** 1 **つぶん**
- sizeof **演算子で文字列リテラルの大きさが得られる**

例 1 : `sizeof("123")` $= 4$

1	2	3	\0

例 2 : `sizeof("AB\tC")` $= 5$ (6 **ではない**; `\t` **は** 1 **文字なので**)

A	B	\t	C	\0

例 3 : `sizeof("abc\0def")` $= 8$ (4 **ではない**; **途中の**`\0` **も続けて勘定**)

a	b	c	\0	d	e	f	\0

例 4 : `sizeof("")` $= 1$

\0

文字列リテラルの生存期間と記憶域

静的記憶域期間：**プログラム実行から終了までずっと存在**

```
void func(void) {
  puts("ABCD");
  puts("ABCD");
}
```

これはあまり気にしなくても良い

同一内容の文字列リテラルのメモリ上の配置

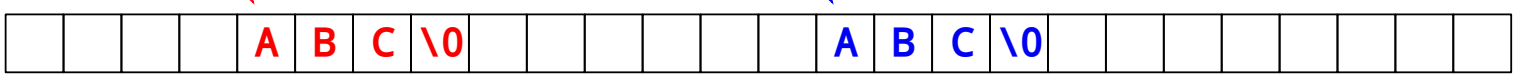
メモリへの配置には個別/おまとめの場合あり (処理系に依存)

配置例 1 (おまとめ配置された場合)

```
void func(void) {
  puts("ABC");
  puts("ABC");
}
```

配置例 2 (個別配置された場合)

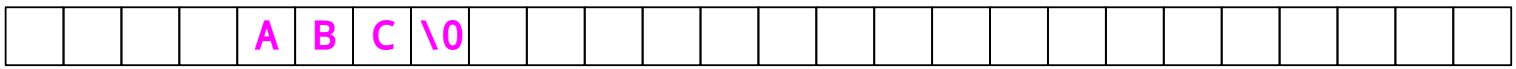
```
void func(void) {
  puts("ABC");
  puts("ABC");
}
```



文字列リテラルは書き換えないで!! (その 1)

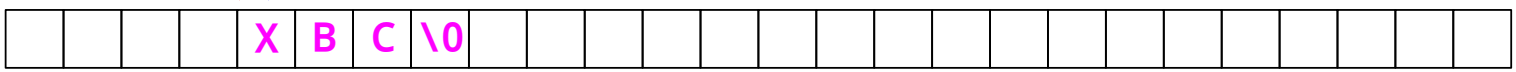
複数の同一の文字列リテラルがおまとめ配置されている場合あり

```
void foo(void) {
  char *p1 = "ABC";
  char *p2 = "ABC";
  ...
```



`p1[0] = 'X';` **を実行すると**
`p2[0]` **も意図せず** `X` **になってしまう (おまとめ配置時)**

```
void foo(void) {
  char *p1 = ...
  char *p2 = ...
  ...
```



プログラムの振る舞いが処理系依存でよろしくない

文字列リテラルは書き換えないで!! (その 2)

書き込み不可 (禁止) のメモリ領域 (ROM) へ配置される場合あり

`p1[0] = 'X';` と書き換えようとしても書き換わらない

場合によっては実行が強制終了になる

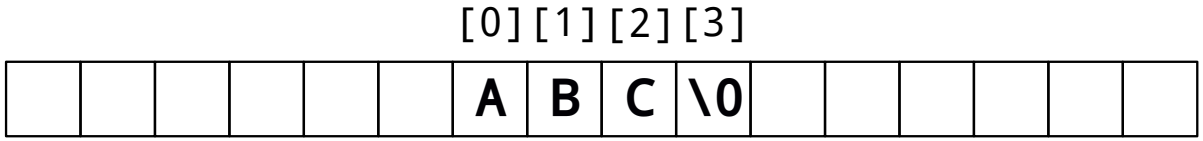
文字列リテラルは定数のようなもの :
書き換えされない前提でコンパイル&メモリ配置

□ 文字列 p.258

★重要★

char 型データの列がナル文字\0 で終端されているもの

文字列 **"ABC"**



文字列リテラルと文字列：必ずしも同じでない

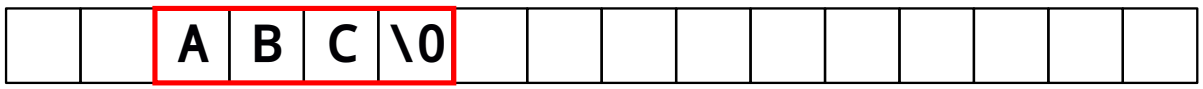
- **文字列リテラル：途中にナル文字が入っている場合あり**
- **文字列：途中にナル文字が入っていてはだめ**

例

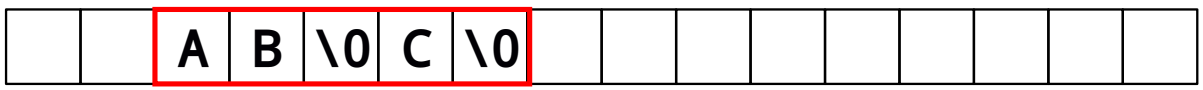
- `"ABC"` **文字列リテラルかつ文字列である**
- `"AB\0C"` **文字列リテラルだが文字列ではない**

メモリイメージ

`"ABC"` — **文字列である**文字列リテラル



`"AB\0C"` — **文字列ではない**文字列リテラル



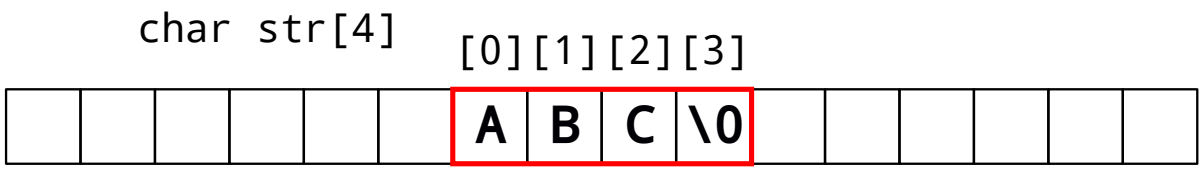
- 途中にナル文字が入っているため

文字列の使用例

List 9-2 : 文字配列への代入と printf での表示

```
#include <stdio.h>
int main(void)
{
  char str[4];  /* 文字列を格納する配列 */
  str[0] = 'A';  /* 代入 */
  str[1] = 'B';  /* 代入 */
  str[2] = 'C';  /* 代入 */
  str[3] = '\0'; /* 代入 */
  printf("文字列strは\"%s\"です。\n", str); /* 表示 */
  return 0;
}
```

- char 配列 (大きさ 4) の各要素に文字を代入
- 長さ 3 の文字列を構成 (ナル文字で終端)
- printf での文字列表示用の書式指定は %s



□ 文字配列の初期化 p.259

宣言時の初期化を使う方法

```
char str[4] = { 'A', 'B', 'C', '\0' };
```

- **要素数 4 の char 配列, 各要素を初期化**

初期化データを文字列リテラルで指定

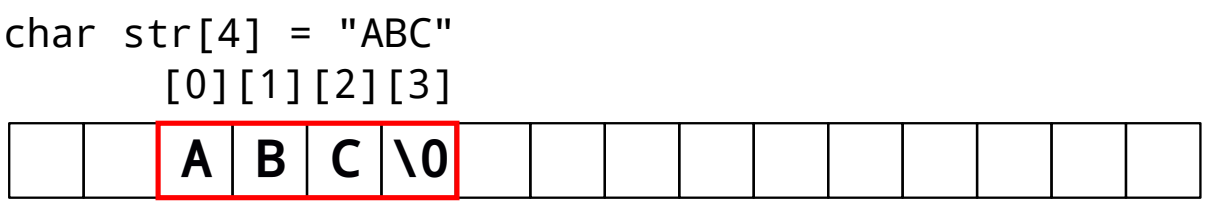
```
char str[4] = "ABC";
```

- **文字列リテラルを初期値とするのではない**
- **初期値の表現方法として文字列リテラルを使用している**

宣言する要素数を省略できる (☆普通はこの方法を使用☆)

```
char str[] = "ABC";
```

※いずれも同等 (どの書き方をしても同じ)



要素数指定を省略・初期化を使う例

List 9-3 : **文字配列の初期化と** printf **での表示**

```
#include <stdio.h>
int main(void)
{
  char str[] = "ABC";
  printf("文字列strは\"%s\"です。\n", str);
  return 0;
}
```

できない代入

配列 (文字列=文字の配列) には初期化子を代入できない

```
char s[4];
s = { 'A', 'B', 'C', '\0' };   /* エラー */
s = "ABC";                     /* エラー */
```

★重要★

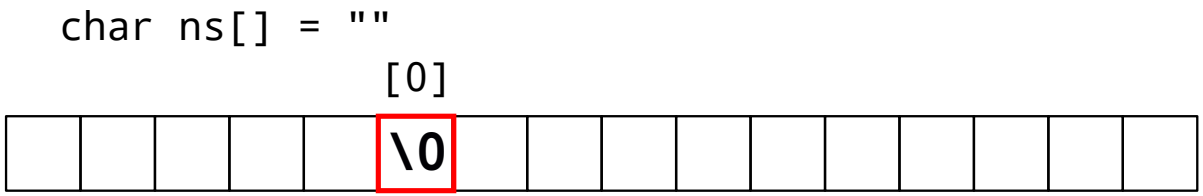
□ 空文字列 (null string) p.260

文字をひとつも含まない文字列 (終端のナル文字だけ)

- **長さ 0 の文字列**

```
char ns[] = "";
```

```
char ns[] = { '\0' };
```



★重要★

□ 文字列の読み込み p.260

キーボードから文字列を読み込む方法を紹介

List 9-4 : **名前を読み込み挨拶を表示**

```
#include <stdio.h>
int main(void)
{
   char name[48];
   printf("お名前は：");
   scanf("%s", name);
   printf("こんにちは、%sさん!!\n", name);
   return 0;
}
```

scanf

- **文字列読み込みの変換指定** "%s" **を使用**
- name **の前に** & **がないことに注意**

実行例

```
お名前は: Mike
こんにちは、Mikeさん!!
```

scanf の危険性

★重要★

List 9-4 (再掲) : 名前を読み込み挨拶を表示

```
#include <stdio.h>
int main(void)
{
   char name[48];
   printf("お名前は：");
   scanf("%s", name);
   printf("こんにちは、%sさん!!\n", name);
   return 0;
}
```

セキュリティホールの典型
危険!! こんなプログラムを書いてはダメ

Q. 何で危険なの?
A. このコードでは**バッファオーバラン**が起きるよ

- 入力に 48 文字以上を与えると配列の範囲を超えて書き込まれる!!!

バッファオーバーランとは

★重要★

想定の範囲を超えて読み込みが行われる現象

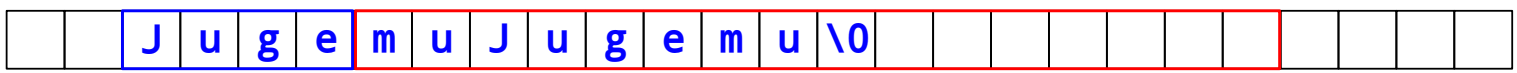
- 他の変数の値や実行制御用の値が意図しない値に書き換わる
- プログラムのミス (**重大なセキュリティホールにつながる**)

バッファ (読み込みメモリ) のサイズが 4 の場合

```
char name[4], addr[16];
printf("Your Name : ");
scanf("%s", name);
printf("Hello, %s!!\n", name);
```

変数のメモリへの割当の例

長い入力 (例 JugemuJugemu) を与えた場合 : 他の変数の値を破壊



【やってみた】バッファオーバーラン 【OK じゃん】

```
#include <stdio.h>
int main(void)
{
   char name[4];
   char addr[16] = "烏丸丸太町";
   printf("お名前は：");
   scanf("%s", name);
   printf("こんにちは、%sさん!!\n", name);
   printf("住所は%sでOK?\n", addr);
   return 0;
}
```

実行

```
お名前は：Bob
こんにちは、Bobさん!!
住所は烏丸丸太町でOK?
```

なんだちゃんと動いてるじゃん...
「バグないです. プログラム完成しますた」

【やってみた】バッファオーバーラン 【誤動作】

せんせい「長い入力でやってみて下さい」

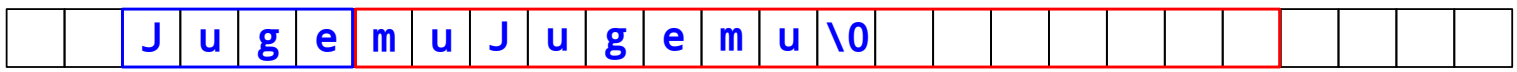
実行

```
お名前は：JugemuJugemu
こんにちは、JugemuJugemuさん!!
住所はmuJugemuでOK?
```

「あれ?」

変数のメモリへの割当

JugemuJugemu を与えた場合：他の変数の値を破壊

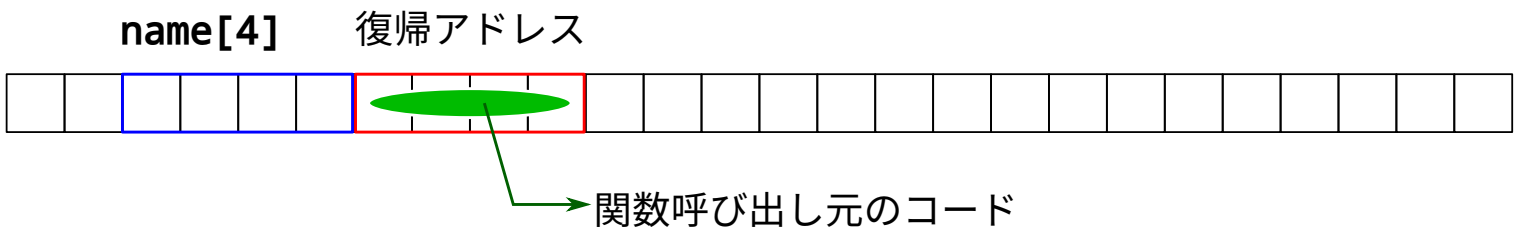


バッファオーバーランの悪用：外部から有害プログラムを送り込む (1/2)

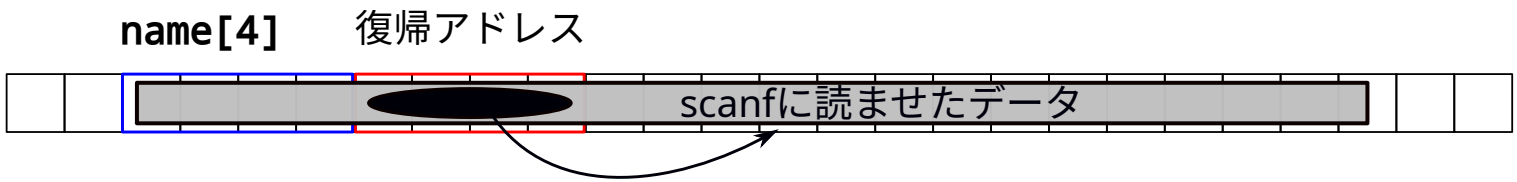
事案が発生する状況

- スタック上にバッファ (自動変数) が配置されている
- バッファすぐに関数呼出元への復帰アドレスが退避されている

1. **関数呼び出し時：自動変数と復帰アドレスがスタック上に取られる**
   - **復帰アドレス：**
     関数の呼び出し元への戻り先 (プログラムコードのアドレス)

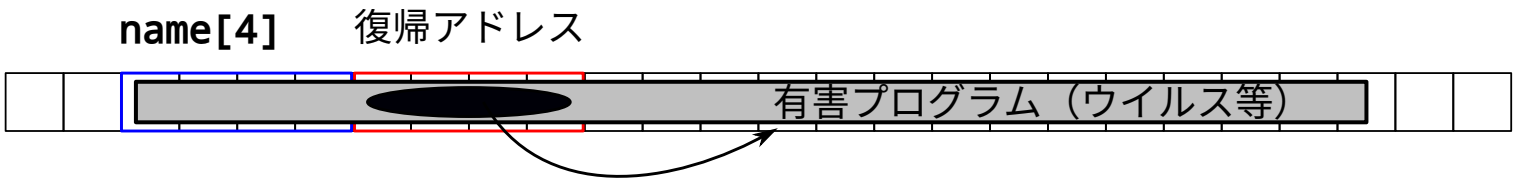


2. scanf **に変なものを読ませてやった**
   - **復帰アドレスが置かれている場所に上書き**
   - **ほかにも変なのを**



バッファオーバーランの悪用：外部から有害プログラムを送り込む (2/2)

3. scanf が読んだのは実は有害プログラムだった!!
   関数から復帰しようとしたら有害プログラムに実行が移ってしまう



外部から任意のプログラムコードを送り込んで実行されてしまう

- 遠隔操作プログラム
- カメラやマイクをこっそりオンにして私生活を覗き見するプログラム
- コンピュータ上のデータをどこかへこっそり送信するプログラム
- ファイルを暗号化して身代金を要求するプログラム
- SPAM メイルをあちこちに送るプログラム
- 他サイトへの不正アクセスを迂回・中継するプログラム

【やってみた】バッファオーバーラン 【クラッシュ】

「もっと長い入力を与えてみて下さい」

「やってみますね」

```
お名前は：JugemuJugemuGogounoSurikireKaijari
こんにちは、JugemuJugemuGogounoSurikireKaijariさん!!
住所はmuJugemuGogounoSurikireKaijariでOK?
*** stack smashing detected ***: <unknown> terminated
Abort (core dumped)
```

(((((゜Д゜;))))) ｶﾞｸｶﾞｸﾌﾞﾙﾌﾞﾙ

復帰アドレス (関数からのリターンアドレス) まで破壊

- **スタック内容が破壊**
- Linux **のセキュリティ機構が発動：変な実行がされずに済んだ**
- (Linux **以外だと有害コードに実行が移ってしまう場合があるよ**)

正しい scanf の使い方

★重要★

List 9-4 (改)：名前を読み込んで挨拶を表示する

```
#include <stdio.h>
int main(void)
{
   char name[48];
   printf("お名前は：");
   scanf("%47s", name); /* 47文字まで */
   printf("こんにちは、%sさん!!\n", name);
   return 0;
}
```

配列の要素数を超えて読み込まれないようにする

scanf で文字列を読み込むときは最大文字数を必ず指定すること

- 指定する最大文字数：ナル文字ぶんは含めない
- 文字配列の要素数が 48 なら scanf で読み込む最大文字数は 47
- ナル文字の記憶用に要素 1 つを残しておく

scanf による文字列の入力での注意

scanf 関数の仕様：スペース文字は文字列の区切りになる

先程のプログラムを実行するとこうなる

```
お名前は : Bill Brown
こんにちは、Billさん!!
```

Bill がひとつの文字列となって `scanf("%47s", name)` で読まれる

- Brown は読まれない

scanf 関数はよく調べて使おう

- 文字列の扱いはけっこう複雑
- バッファオーバーランに注意

ふだんから気をつけてプログラムを書く

scanf() **は危なげなので要注意**

- **読み込み幅を必ず明示的に指定する**

絶対に使ってはならない関数：gets()

- **代わりに** fgets() **を使う**

詳しくはたとえば以下を参照

「バッファオーバーラン ～その１・こうして起こる～」, **セキュア・プログラミング講座** C/C++ **言語編**, https://www.ipa.go.jp/security/awareness/vendor/programmingv1/b06_01.html, **情報処理推進機構** IPA, (2020/10/07 **閲覧**).

「バッファオーバーラン ～その２・「危険な関数たち」～」, **セキュア・プログラミング講座** C/C++ **言語編**, https://www.ipa.go.jp/security/awareness/vendor/programmingv1/b06_02.html, **情報処理推進機構** IPA, (2020/10/07 **閲覧**).

★重要★

□ 文字列を書式化して表示 p.261

printf **関数での文字列表示：いろいろな書式制御が可能**

List 9-5 (**主要部**)

```
char str[] = "12345";
printf("%s\n",    str);   /* そのまま  */
printf("%3s\n",   str);   /* 最低3桁   */
printf("%.3s\n",  str);   /* 3桁まで   */
rintf("%8s\n",   str);    /* 最低8桁で右よせ  */
printf("%-8s\n",  str);   /* 最低8桁で左よせ  */
```

出力 (**「↓」の場所で改行**)

```
12345↓
12345↓
123↓
   12345↓
12345   ↓
```

printf での文字列表示の書式指定

最小フィールド幅

- 少なくともこの桁数だけ表示が行われる
- 長い文字列では指定の桁数を超えて表示
- -を指定すると左寄せで表示 (無指定では右寄せで表示)
- 例：`%9.6s`　(最小フィールド幅は 9, 右寄せ)
- 例：`%-9.6s`　(左寄せ)

精度

- 表示する桁数の上限を指定
- 長い文字列では途中で表示を打ち切り
- 例：`%9.6s`　(精度は 6)

変換指定子

- `s` で文字列の表示を指定
- 例：`%9.6s`

おわり

番外編の課題：シーザー暗号 (例)

暗号文

```
FQNHJ BFX GJLNSSNSL YT LJY AJWD YNWJI TK XNYYNSL GD
MJW XNXYJW TS YMJ GFSP, FSI TK MFANSL STYMNSL YT IT:
TSHJ TW YBNHJ XMJ MFI UJJUJI NSYT YMJ GTTP MJW XNXYJW
BFX WJFINSL, GZY NY MFI ST UNHYZWJX TW HTSAJWXFYNTSX
NS NY, 'FSI BMFY NX YMJ ZXJ TK F GTTP,' YMTZLMY FQNHJ
'BNYMTZY UNHYZWJX TW HTSAJWXFYNTS?'
```

番外編の課題：シーザー暗号 (例)

解読結果 (平文)

```
ALICE WAS BEGINNING TO GET VERY TIRED OF SITTING BY
HER SISTER ON THE BANK, AND OF HAVING NOTHING TO DO:
ONCE OR TWICE SHE HAD PEEPED INTO THE BOOK HER SISTER
WAS READING, BUT IT HAD NO PICTURES OR CONVERSATIONS
IN IT, 'AND WHAT IS THE USE OF A BOOK,' THOUGHT ALICE
'WITHOUT PICTURES OR CONVERSATION?'
```

(Lewis Carroll, "ALICE'S ADVENTURES IN WONDERLAND")

番外編の課題：シーザー暗号 (方法説明と課題内容)

シーザー暗号：**以下の規則に基づく**

- **英文の文章を対象**; **アルファベットには大文字のみを使用**
- **暗号化の方法**: **アルファベットの各文字を** 5 **つ後ろへずらす**
  - A → F, B → G, C → H, ..., U → Z, V → A, ..., Z → E

平文	H	E	L	L	O	W	O	R	L	D
暗号文	M	J	Q	Q	T	B	T	W	Q	I

  - **アルファベット以外の文字 (数字や記号類) はそのまま**
- **解読方法**: **上記の逆**

用語：

- **平文** (plain text)：**元の文**
- **暗号文** (cipher text)：**暗号化された文**

課題内容：**以下の** 2 **つのプログラムを作成せよ**

1. **任意に与えられる平文を暗号化するプログラム**
2. **任意に与えられる暗号文を解読するプログラム**

番外編の課題：シーザー暗号 / 拡張 (1)

先述のシーザー暗号：ずらす文字は 5 に限定
今回：5 以外の値の場合でも暗号化したい

課題内容：以下の 2 つのプログラムを作成せよ

- ずらす値を実行時に任意に指定可能とすること
- 1 以上 26 以下

1. 任意に与えられる平文を暗号化するプログラム
2. 任意に与えられる暗号文を解読するプログラム

番外編の課題：シーザー暗号 / 拡張 (2)

拡張課題 (1): シーザー暗号での文字をずらす値を指定した

課題内容：ずらす文字の値を自動的に推測するプログラムの作成

- **与えられるのは暗号文のみ**
- **何文字文ずらされているか分からない**
- **ずらしている値を自動で判別して解読文を表示**