

# システムプログラミング序論

## 第7回

### 関数ポインタ， 不正なメモリアクセス

大山恵弘

積み残し

# switch 文

```
switch (X) {  
  case A:  
    P;  
    break;  
  case B:  
    Q;  
    break;  
  case C:  
    R;  
    break;  
  
  ...  
  
  default:  
    Y;  
}
```

- X の値が A ならば P, B ならば Q, C ならば R を実行する
- どれでもなければ Y を実行する
- X の値は尽くされていなくてもよいし, default: 以降の部分はなくともよい
- **break** の書き忘れに注意

# void \* 型

- あらゆるポインタを保持できる変数の型
  - `int *` 型, `char *` 型, . . .
  - 汎用ポインタと呼ばれることがある

```
int main(void)
{
    int x = 3;
    void *p;
    char *q;
    ...
    p = (void *)&x; /* OK */
    q = (char *)p; /* OK */
    ...
    return 0;
}
```

```
int main(void)
{
    void *r;
    int *s;
    ...
    r = malloc(1000); /* OK */
    s = (int *)r; /* OK */
    ...
    return 0;
}
```

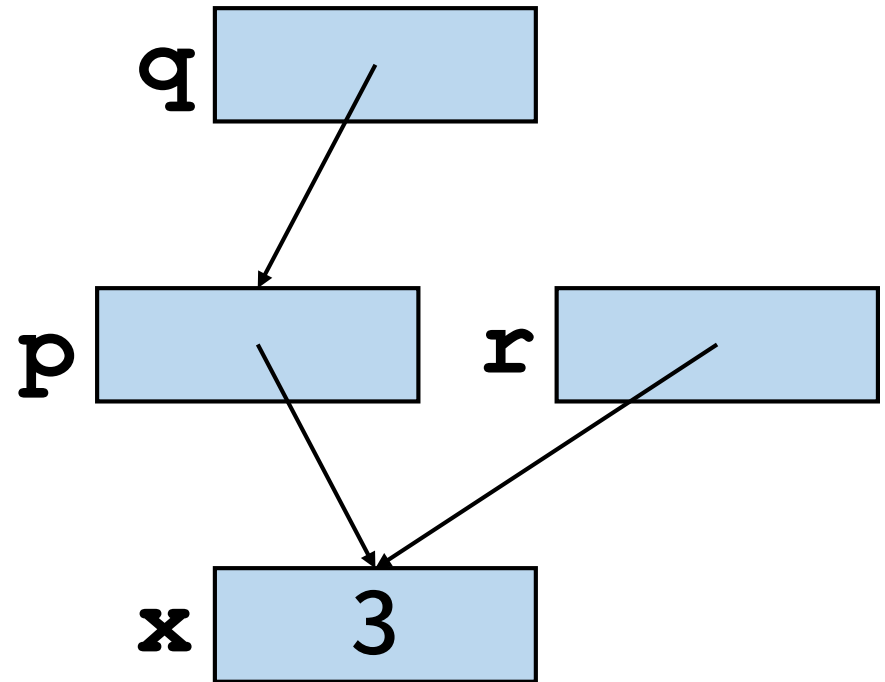
# ポインタのポインタ

- ポインタ型のデータを指すポインタ

```
int main(void)
{
    int x = 3;
    int *p, *r;
    int **q;

    p = &x; /* OK */
    q = &p; /* OK */
    r = *q; /* OK */
    /* 3 is displayed */
    printf("%d\n", *r);

    return 0;
}
```



# 自己参照構造体

- 自分自身の型のデータを指すポインタをメンバに含む構造体
  - リストや木などの再帰的なデータ構造を扱うプログラムで大抵出てくる

```
struct list_t {  
    int x;  
    struct list_t *next;  
}; /* OK */
```

```
struct tree_t {  
    int x;  
    struct tree_t *left;  
    struct tree_t *right;  
}; /* OK */
```

```
struct inf_t {  
    int x;  
    struct inf_t next;  
}; /* not OK */
```

# 「関数名() 」という表現

- `a` が関数であることを示唆するために、`a()` という表現が用いられることがある
  - `a` が関数であることが一目で分かる
    - `int` 型の変数でもなく、文字列でもなく、型の名前でもなく
  - `a` が引数を受け取る関数であったとしても、`a()` と書くことが多い
  - ライブラリ関数のマニュアルにも、この表現が用いられている
    - `man fopen` や `man putchar` を実行してみよう

# 初期化付き配列宣言における 配列サイズの省略

```
int main(void)
{
    int a[3] = { 2, 4, 6 };
    int i;
    for (i = 0; i < 3; i++) {
        printf("%d\\n", a[i]);
    }
    return 0;
}
```

```
int main(void)
{
    int a[] = { 2, 4, 6 };
    int i;
    for (i = 0; i < sizeof(a)
/ sizeof(int); i++) {
        printf("%d\\n", a[i]);
    }
    return 0;
}
```

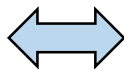



# 関数ポインタ

# 関数ポインタとは

- 関数の実体である機械語プログラムが格納されたメモリ領域の先頭を指すポインタ
  - 変数、配列への代入が可能
  - 関数の引数や返り値として利用可能
  - そのポインタが指す関数を呼び出すことが可能
  - 加減乗除は不可

```
int fact(int n)
{
    if (n == 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```



fact:  ココ

```
0x100000ed0: push    %rbp
0x100000ed1: mov     %rsp, %rbp
0x100000ed4: sub     $0x10, %rsp
0x100000ed8: mov     %edi, -0x8(%rbp)
0x100000edb: cmpl    $0x1, -0x8(%rbp)
0x100000ee2: jne     0x100000ef4
0x100000ee8: movl    $0x1, -0x4(%rbp)
0x100000eef: jmpq    0x100000f13
0x100000ef4: mov     -0x8(%rbp), %eax
0x100000ef7: mov     -0x8(%rbp), %ecx
0x100000efa: sub     $0x1, %ecx
0x100000f00: mov     %ecx, %edi
0x100000f02: mov     %eax, -0xc(%rbp)
0x100000f05: callq   0x100000ed0
0x100000f0a: mov     -0xc(%rbp), %ecx
0x100000f0d: imul    %eax, %ecx
0x100000f10: mov     %ecx, -0x4(%rbp)
0x100000f13: mov     -0x4(%rbp), %eax
0x100000f16: add     $0x10, %rsp
0x100000f1a: pop     %rbp
0x100000f1b: retq
```

# 関数ポインタの変数宣言

```
int (*fp)(int x, int y);
```

- 変数 `fp` には、「`int` 型の変数を2つ受け取り、`int` 型の返り値を返す関数へのポインタ」を入れることができる
  - `*fp` の周りの括弧は必ず付ける．付けないと意味が変わる

```
int *f(int x, int y);
```

cf. 「`int` 型の変数を2つ受け取り、`int *` 型の返り値を返す関数」の宣言

- すごく雑に言えば、関数ポインタの変数を宣言するには、
  - 対象の関数宣言の関数名の部分を、宣言したい変数名に変え、
  - その変数を (`*` と ) でくくる

# 関数ポインタが指す関数の 呼び出し

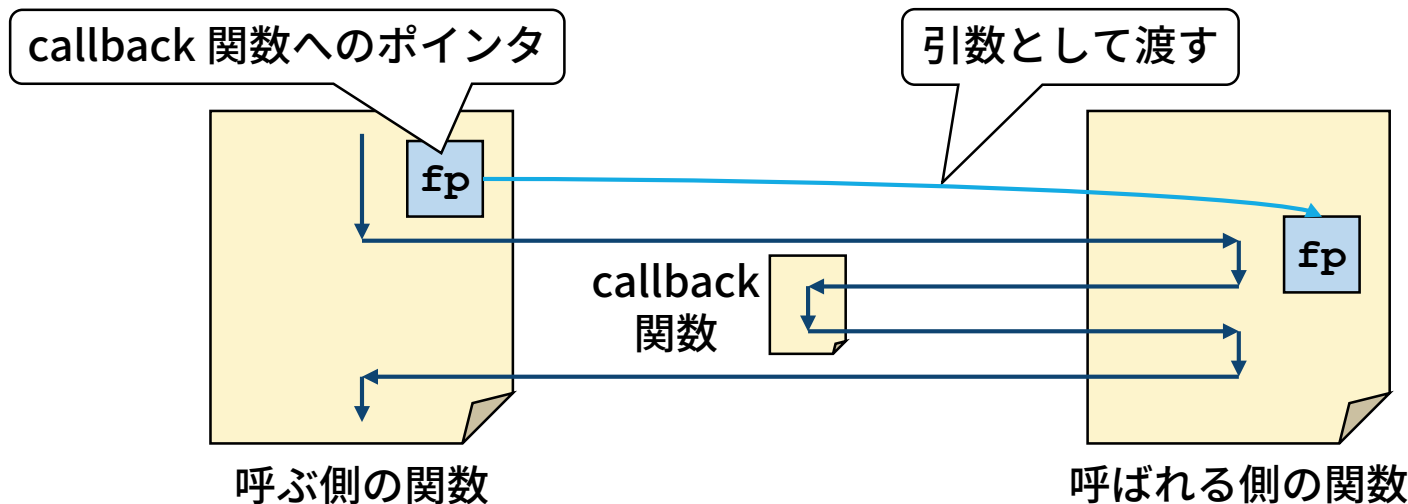
```
int round(double x)
{
    ...
}

int main(void)
{
    int (*fp)(double x);
    ...
    fp = round;
    ...
    a = (*fp)(3.14);
    ...
    return 0;
}
```

- どの文も同じ効果をもたらす
  - `a = round(3.14)`
    - 普通の関数呼び出し
  - `a = (*fp)(3.14)`
    - 関数ポインタ経由の関数呼び出し  
(関数ポインタであることを明示)
  - `a = fp(3.14)`
    - 関数ポインタ経由の関数呼び出し  
(関数ポインタであることを非明示)

# いつ関数ポインタを使う？

- 典型的な使用場面：コールバック関数
  - 事前に指定した別の関数をあるタイミングで呼ばせたいことがある
    - 特定の関数の実行時やイベント発生時など
  - 「呼ばせたい関数」へのポインタを変数や関数引数を通じてプログラム部分に与えることにより，その部分に望みの動作をさせられる
    - そのプログラム部分は「このイベントが発生したら，事前に受け取っていた関数ポインタの関数を呼び出す」などの形で書かれる



# 例1：関数の実行時間の計測

```
int time_func(void (*fp)(void))
{
    time_t tmb, tme;
    tmb = time(NULL);
    (*fp)();
    tme = time(NULL);
    return tme - tmb;
}
```

関数ポインタ fp が指す関数の  
実行の前後で時刻を取得

```
int main (int argc, char *argv[])
{
    int i;
    void (*fps[])(void) = {func_0, func_1, func_2};
    for (i = 0; i < 3; i++) {
        printf("func_%d: %d s¥n", i, time_func(fps[i]));
    }
    return 0;
}
```

関数ポインタで  
配列を初期化

# 例2： atexit 関数

- main 関数の終了後に実行すべき関数を指定するためのライブラリ関数
  - プログラムの「後始末」の実行などに用いられる

```
#include <stdio.h>
#include <stdlib.h>

void final_message(void)
{
    puts("in final_message");
}

int main(int argc, char *argv[])
{
    atexit(final_message);
    puts("in main");
    return 0;
}
```



in main in final_message
-----------------------------

# 例3：qsort 関数

```
void  
qsort(void *base, size_t nel, size_t width,  
      int (*compar)(const void *, const void *));
```

配列の先頭

要素の個数

要素のサイズ

2つの要素を比較する関数へのポインタ

base

width  
(== sizeof(int))

compar

“The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.”



nel (== 10)



# 大小を比較する関数の例

```
int compare_int(void *ptr_a, void *ptr_b)
{
    int a, b;
    a = *((int *)ptr_a);
    b = *((int *)ptr_b);
    return a - b;
}
```

キャストを使い, void \* 型のポインタを  
int \* 型のポインタとして解釈させている

```
int compare_str_reverse(void *ptr_a, void *ptr_b)
{
    return -strcmp((char *)ptr_a, (char *)ptr_b);
}
```

辞書順の逆順に並べたい場合

# 現実のコードにおける 関数ポインタの使用例

- xv6 OS におけるシステムコール処理

```
static int (*syscalls[])(void) ={
    [SYS_fork]    sys_fork,
    [SYS_exit]    sys_exit,
    [SYS_wait]    sys_wait,
    ...
};

void syscall(void) {
    ...
    num = proc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num])
    {
        proc->tf->eax = syscalls[num] ();
    } else {
        ...
    }
}
```

配列に関数ポインタを入れている  
(designated initializer という  
新しめの記法を使用)

syscalls 配列の num 番目に入っている  
ポインタの先の関数を呼び出している

# 不正なメモリアクセス

# 確保したメモリ領域の外へのアクセス

- 多くの言語では，そのようなアクセスの記述自体が難しかったり，  
そういうアクセスをしても検知され，安全に処理される
  - Java, Python, Ruby, Go, OCaml, Scheme, Haskell, ...
- C言語では，容易に記述できる

```
void bug1(void)
{
    int buf[100];
    buf[200] = 5;
    ...
}
```

```
void bug2(void)
{
    int *nums;
    nums = (int*)malloc(3);
    nums[0] = 0;
    nums[1] = 1;
    nums[2] = 3;
    ...
}
```

```
void bug3(void)
{
    int *p = (int *)0xdeadbeef;
    int n = *p;
    ...
}
```

```
void bug4(void)
{
    char *p = NULL;
    char c;
    c = *p;
    ...
}
```

```
void bug5(void)
{
    char buf[5];
    char *s = "Hello";
    strcpy(buf, s);
    ...
}
```

```
void bug6(void)
{
    char str[] = {
        'y', 'e', 's'
    };
    puts(str);
}
```

```
void bug7(void)
{
    struct list *x;
    x = malloc(...);
    ...
    free(x);
    printf("%d¥n",
            x->num);
}
```

# 確保したメモリ領域外をアクセスしたら何が起こるのか？ (仕様)

- The behavior is **undefined** in the following circumstances:

...

- The value of a pointer to an object whose lifetime has ended is used
- An array subscript is out of range, even if an object is apparently accessible with the given subscript ...

(ISO/IEC 9899:1999, Annex J.2)

- 何が起こるかわからない
  - プログラムが突然終了するのも仕様通り
  - プログラムが動き続けるのも仕様通り
  - 「必ず0が読み込まれる．書き込みは無視される」のも仕様通り
  - 「無限ループで固まる」のも仕様通り

# 確保したメモリ領域外をアクセスするプログラム

```
#include <stdio.h>

int main(void)
{
    int arr[10]; /* valid range: arr[0]~arr[9] */

    printf("arr[10] = %d\\n", arr[10]); /* bug */
    arr[10] = 3; /* bug */
    printf("arr[10] = %d\\n", arr[10]); /* bug */

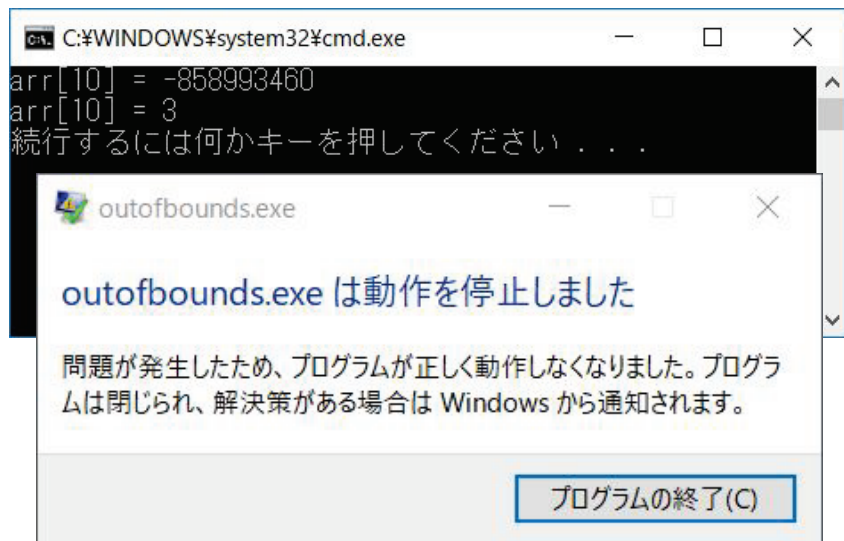
    printf("arr[10000] = %d\\n", arr[10000]); /* bug */
    arr[10000] = 4; /* bug */
    printf("arr[10000] = %d\\n", arr[10000]); /* bug */

    return 0;
}
```

# 確保したメモリ領域外をアクセスしたら何が起こるのか？ (現実)

```
crocusxx$ ./a.out  
arr[10] = 1728735360  
arr[10] = 3  
Segmentation fault: 11
```

```
pentas-compc$ ./a.out  
arr[10] = 0  
arr[10] = 3  
セグメンテーション違反です (コアダンプ)
```



```
ubuntu.u.tsukuba.ac.jp$ ./a.out  
arr[10] = 9888256  
arr[10] = 3  
Segmentation fault (コアダンプ)
```

# 多くのCプログラマの頭の中にある経験則

- 確保したメモリ領域の外のアドレスにアクセスした場合：
  - そのアドレスが，有効なメモリ範囲に近い場合：
    - 普通に読めて，読める値は0であることも，変な値であることもある
    - 普通に書いて，書いた値は以降の読み出しで読み出せる
  - そのアドレスが，有効なメモリ範囲に遠い場合：
    - 読んでも，書いても，セグメンテーション違反でプログラムが強制終了
  - そのアドレスが NULL（ゼロ）である場合：
    - 読んでも，書いても，セグメンテーション違反でプログラムが強制終了
- 現状では，かなり高い確率で正しい
  - 詳しくはオペレーティングシステムの講義で説明されるはず



# C言語の仕様および処理系に 存在する巨大な問題

- 不正なメモリアドレスを読み書きしても、プログラムが普通に動き続ける（ことがある）
  - Cプログラムの脆弱性や不具合の温床
    - 例：2014年4月のIEの脆弱性，OpenSSLのHeartbleed脆弱性
  - 読んではいけない場所が読めてしまう
  - 書いてはいけない場所に書けてしまう
- 根本的解決は期待できない
  - 世界中の賢い人々が20年以上研究してきたが，根本的解決はされていない
  - 問題を「緩和」する方法なら，山のように考案された
  - C言語を使う限り，この問題は避けられないと考えてよい

# Q. このプログラムの 問題点は？

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buf[16];
    if (argc != 2) {
        exit(1);
    }
    puts(argv[1]);
    strcpy(buf, argv[1]);
    puts(buf);
    return 0;
}
```

# A. argv[1] が想定以上に長い場合に，誤動作する

```
$ gcc insecure.c
```

```
$ ./a.out hogehogehoge
```

```
hogehogehoge
```

```
hogehogehoge
```

12字

```
$ ./a.out hogehogehogehogehogehoge
```

```
hogehogehogehogehogehoge
```

```
hogehogehogehogehogehoge
```

24字

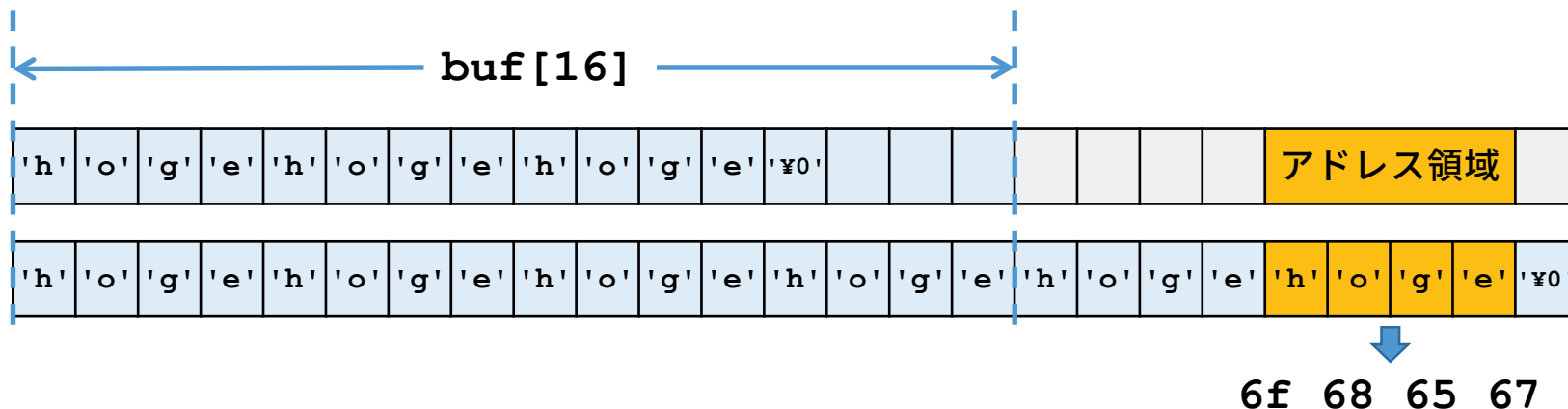
```
Abort trap: 6
```

!?

```
$
```

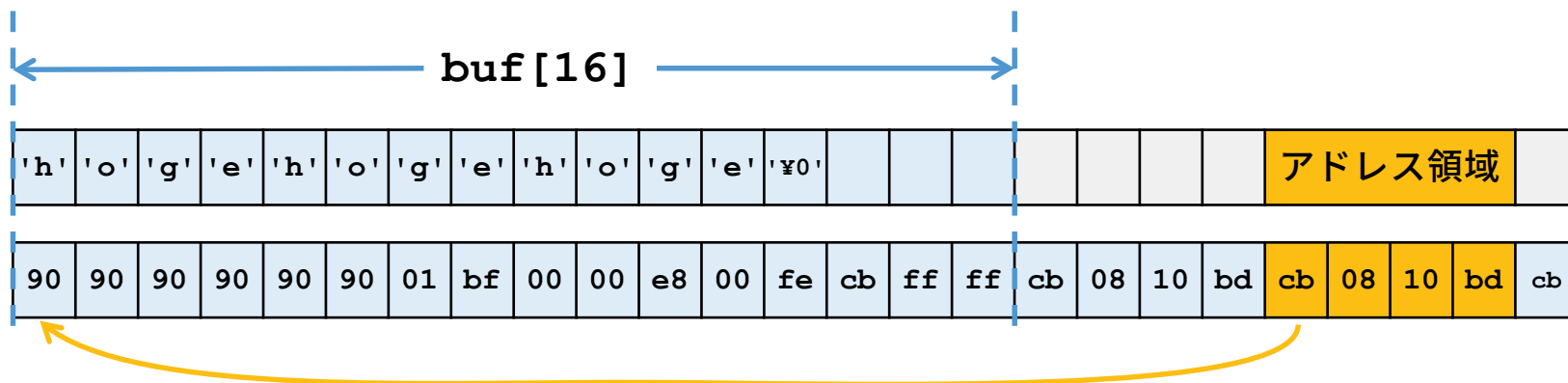
# 何が起きた？

- プログラムの実行に必要なメモリが破壊された
  - 16文字分しかないバッファに16文字以上書き込もうとした
  - バッファに隣接する別の領域まで上書きしてしまった
  - 関数から戻る際に必要となる情報を記録した領域（コールスタック）を上書きしてしまった
  - main 関数の終了後、文字列をアドレスとして解釈し、それが指す場所を参照しようとした
  - 使用できないアドレスだったので trap が発生して異常終了した



もしも...

- 文字の代わりにアドレス，しかも `buf` のあたりのアドレスを書き込んでいたら…
- かつ，`buf` の中に機械語命令列のデータを書き込んでいたら…
  - その機械語命令が実行される！
  - コマンドライン引数に与える文字列を工夫すれば，任意のプログラムを実行させることができる！



どれくらい危険なことか、想像をはたらかせてみよう

# バッファオーバーフロー攻撃

- 確保したメモリ領域の境界を越える部分に書き込ませることにより、プログラムを異常終了させたり、プログラムに望みの処理を実行させる攻撃
  - 望みの処理の例：危険なライブラリ関数呼び出し、攻撃者が注入したコードの実行、...
  - 書き込ませるデータとしてコマンドライン引数が使われるのは稀で、ネットワークやファイルのデータが使われることが多い
  - この攻撃を受けやすいのはC言語のプログラム
    - メモリアクセスが配列の境界を越えているかどうかを、基本的にはチェックしないため
    - 他の多くのプログラミング言語のプログラムでは、この攻撃は成功しない
  - 一般に、攻撃を許すようなプログラムの誤りは脆弱性と呼ばれる

# Double Free

- すでに解放したメモリ領域を再度解放しようとする処理
  - やってはいけない
  - やってしまった場合、動作は不定
    - エラーが出て強制終了ならラッキーと思ってよい

```
p = malloc(256);  
...  
free(p);  
...  
free(p);  
...
```

```
$ ./a.out  
a.out(92439,0x7fffb44a7380) malloc: *** error  
for object 0x7fe44d001000: pointer being  
freed was not allocated  
  
*** set a breakpoint in malloc_error_break to  
debug  
Abort trap: 6  
$
```

# 解放済みのメモリ領域へのアクセス

- やはり，やってはいけないし，動作は不定
  - エラーが出て強制終了ならラッキーだが，出ないことが多い
    - 静かにゴミデータを読んだり，データを壊したりする

```
int *p;  
p = malloc(4096);  
...  
free(p);  
...  
*(p + 10) = 5;
```

```
$ ./a.out  
(エラーが出ない)  
$
```

```
int *f(void)  
{  
    int x = 3;  
    int *y;  
    y = &x;  
    return y;  
}  
  
int main(void)  
{  
    int *a;  
    a = f();  
    printf("%d\n", *a);  
    return 0;  
}
```

```
$ ./a.out  
3  
(いつも3とは限らない)  
$
```



# malloc が失敗して確保できていないメモリ領域へのアクセス

- 同様

```
int *p;  
p = malloc(1000000000000000);  
*(p + 100000) = 5;
```

```
$ ./a.out  
(エラーが出ない)  
$
```

# 脆弱性を入れないために

- 確保されていないメモリ領域や，確保したメモリ領域を越えた部分にアクセスしないよう，入念にチェックする
- `malloc` と `free` の対応を入念にチェックする
- 文字列の長さを検査しない関数はできるだけ使わない
  - `strcat` → `strncat`, `snprintf` など
  - `strcpy` → `strncpy`, `snprintf` など
- 攻撃を防ぐためのツールや仕組みを利用する
  - `stack-protector`, `DEP`, `ASLR`, ...
  - どれも完全ではないので過信は禁物
- 脆弱性に関する情報をこまめにチェックする

# セキュアコーディング

- 安全なプログラムを書くための方法論については非常によく研究され，リソースも蓄積されている



PDF版が無料

# 信頼性の高いプログラムを書くために

このあたりの資料を読んでもみると良いでしょう

- CERT C コーディングスタンダード
  - <https://www.jpcert.or.jp/sc-rules/>
- IPA セキュア・プログラミング講座
  - <https://www.ipa.go.jp/security/awareness/vendor/programming/>

# まとめ

- 関数ポインタ

- 変数へのポインタと同様に，関数へのポインタを変数へ格納したり，関数間で受け渡したりすることができる
- 既存のプログラムの動作を変更したり拡張したりする際に有用

- エラーとエラー処理

- メモリアクセスのエラー
- バッファオーバーフロー攻撃
  - 確保したメモリ領域の範囲外にあるデータを上書きさせることにより，プログラムの動作を変える

# おわりに

- 講義で扱った内容はシステムプログラミングの初歩の初歩
  - 開発や研究の現場で活躍するには、何十倍もの勉強と訓練が必要
- システムプログラミングの技術者としてレベルアップするには？
  - 演習課題のような課題を100問、できれば1000問、解いて下さい
    - ネット上には良い課題が大量に転がっています
  - ls や cat などのコマンドを自分でいくつか自作して下さい
  - 簡単なエディタやシェルや Web サーバやファイルシステムを自作して下さい
  - Linux カーネルにちょっとした新機能を入れて下さい
  - ネットワークカードやストレージやグラフィクスカードのデバイスドライバのソースコードを読んで、小さな改造を入れて下さい
  - 自分で設計した小さいプログラミング言語のコンパイラを作して下さい
- 学部生（学類生）時代から自分で突っ走って、システムプログラミング分野で有名になる人々が、昔も今もいる
  - 「未踏 OS」などのキーワードでググってみよう

# 中間試験

- 日時： 2019年12月2日（月） 3限 12:15～
- 試験時間： 60分
- 場所： 3A202（この部屋）
- 試験方式： 筆記試験
- 持ち込み不可
- 答案を提出しての途中退出可能
- 試験直後に解答を発表，解説