

#09 文字列とポインタ

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

- 1 第 11-1 節 文字列とポインタ
- 2 第 11-2 節 ポインタによる文字列の操作
- 3 第 11-3 節 文字列を扱うライブラリ関数
 - 使用しない方がよい関数
- 4 (独自) 文字列のプログラミング
 - 文字列の照合

今回 (#09) の内容：シラバスでの該当部分

小テーマ：文字列とポインタ

第 15 回：文字列を扱う標準ライブラリ関数

重要概念リスト

- 配列による文字列 / ポインタによる文字列
- メモリ上での文字列の配置
- 文字列リテラルは書き換え禁止
- 文字列関係のライブラリ関数群
- strcpy と strcat はバッファオーバーフローの危険性あり
- 文字列から数値へ変換する関数群

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

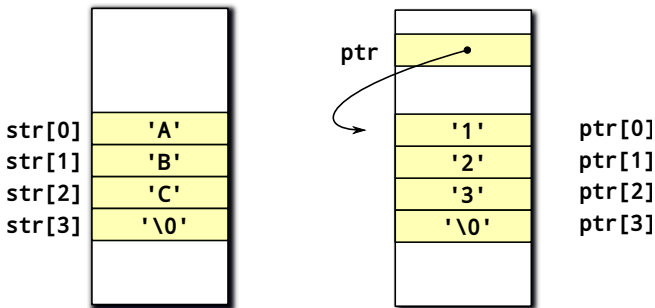
(作成したファイル類は manaba に提出)

第 11-1 節 文字列とポインタ

配列による文字列とポインタによる文字列 p.304

List 11-1 (部分) : 文字列 2 種 (配列/ポインタ)

```
#include <stdio.h>
int main(void)
{
    char str[] = "ABC";    /* 文字の配列 */
    char *ptr = "123";     /* 文字列へのポインタ */
    ... 略 ...
}
```

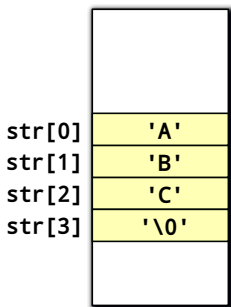


比較：メモリ上への配置

ソースコード上では似ていてもメモリ上では全くの別物

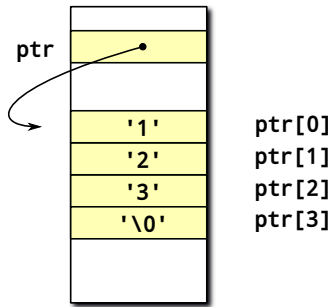
```
char str[] = "ABC";
```

文字の配列; 各要素を初期化



```
char *ptr = "123";
```

ポインタ; 文字列を指すよう初期化

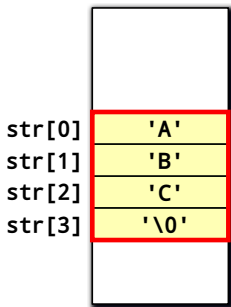


比較：メモリ上でのサイズ

メモリ上で占めるメモリサイズも違う

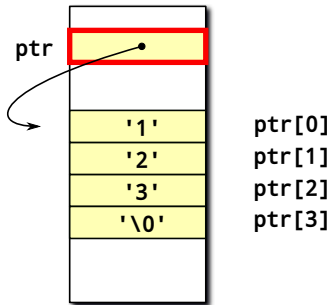
```
char str[] = "ABC";
```

`sizeof(str)`
= $4 \times \text{sizeof(char)}$
(配列全体のメモリサイズ)



```
char *ptr = "123";
```

`sizeof(ptr)`
= `sizeof(char*)`
(ポインタのメモリサイズ)

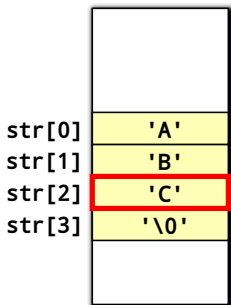


比較：文字列内の文字へのアクセス法

アクセスはどちらでも同様にできる

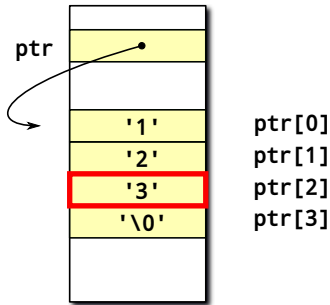
```
char str[] = "ABC";
```

str[i] で i 番目の要素をアクセス



```
char *ptr = "123";
```

ptr[i] で i 番目の要素をアクセス



変数への代入ができたりできなかったり

List 11-2 (部分); コンパイルエラー

```
char s[] = "ABC";  
s = "DEF"; /* エラー */
```

配列変数 (s) への代入：不可

- 配列へポインタを代入できない

List 11-3 (部分); コンパイル OK

```
char *p = "123";  
p = "456"; /* OK ! */
```

ポインタ変数の値の変更：可

- 変数値の変更にすぎない

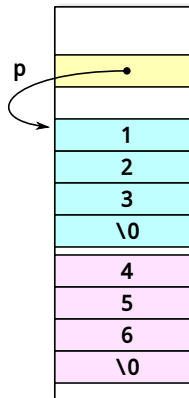
実行の様子 (1/2)

最初：p は 文字列 "123" へのポインタを値として持つ

→


```
int main(void){  
    char *p = "123" ;  
  
    p = "456" ;  
    ...  
}
```

ソースコード



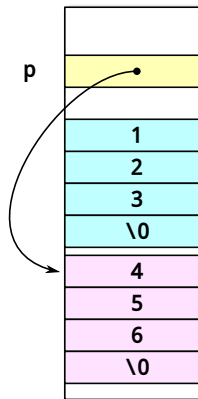
実行の様子 (2/2)

つぎ : p は 文字列 "456" へのポインタに変更



```
int main(void){  
    char *p = "123" ;  
  
    p = "456" ;  
    ...  
}
```

ソースコード



- 文字列 "123" はどの変数からも差されない
- このあとプログラムからは永久に参照不可
- ゴミ (garbage) となる

List 11-4 : 文字列の配列 2 種 / 文字の二次元配列とポインタの配列

```
#include <stdio.h>
int main(void)
{
    char a[][5] = {"LISP", "C", "Ada"};
    char *p[]    = {"PAUL", "X", "MAC"};
    for (int i = 0; i < 3; i++)
        printf("a[%d] = \"%s\"\n", i, a[i]);
    for (int i = 0; i < 3; i++)
        printf("p[%d] = \"%s\"\n", i, p[i]);
    return 0;
}
```

文字列の配列 (1) : 文字の 2 次元配列

```
char a[][5]  
    = {"LISP", "C", "Ada"};
```

char a[][5] は char a[3][5] の省略形

- 初期化データより要素数は 3 と分かるから

| | |
|---------|------|
| a[0][0] | 'L' |
| a[0][1] | 'I' |
| a[0][2] | 'S' |
| a[0][3] | 'P' |
| a[0][4] | '\0' |
| a[1][0] | 'C' |
| a[1][1] | '\0' |
| a[1][2] | '\0' |
| a[1][3] | '\0' |
| a[1][4] | '\0' |
| a[2][0] | 'A' |
| a[2][1] | 'd' |
| a[2][2] | 'a' |
| a[2][3] | '\0' |
| a[2][4] | '\0' |

文字列の配列 (2) : 文字列へのポインタ配列

```
char *p[]  
= {"PAUL", "X", "MAC"};
```

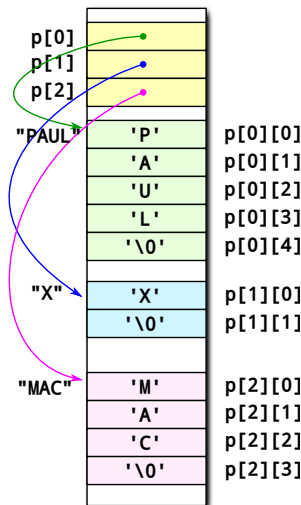
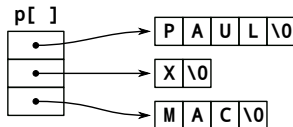
char *p[] は char *p[3] の省略形

- 初期化データより要素数は3と分かるから

注意：各文字列は連続配置されとは限らない

- PAUL, X, MAC の順とは限らない
- 間に空き領域があるかも知れない

図の別表現



第 11-2 節 ポインタによる文字列の 操作

文字列の長さを調べる p.310

実行例

文字列を入力してください：Hello
文字列 "Hello" の長さは5です。

List 11-5 (部分)：文字列の長さを調べる main 関数

```
int main(void)
{
    char str[128];
    printf("文字列を入力してください：");
    scanf("%s", str);
    printf("文字列\"%s\"の長さは%dです。 \n",
           str, str_length(str));
    return 0;
}
```

文字列の長さを調べる関数

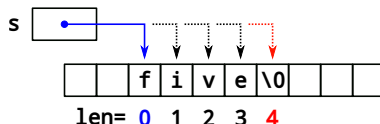
List 11-5 (部分) : 文字列の長さを調べる関数

```
int str_length(const char *s)
{
    int len = 0;
    while (*s++)
        len++;
    return len;
}
```

やっていること

- 文字列の先頭から 1 文字ずつ眺めてゆく
- '\0' に会うまでの文字数を勘定

str_length("five") の実行過程



str_length 精読 : while 文の部分に注目

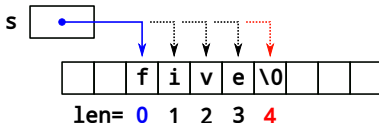
```
while (*s++)  
    len++;
```

*s++ : の動作と値

- ポインタ s が指す内容を返す
- ただしその後に s をインクリメント
(ポストインクリメント; post-increment)
- char z = *s; s++; z; のようなことを簡潔に記述
- s は次の要素を指す

while (*s++) : の動作と値

- ポインタ s が指す内容を調べる
- ただし s をポストインクリメント
- 値が 0 なら (ナル文字 '\0' なら) ループを終了



混乱したときは単純な書き方するのが吉

(再掲) どのタイミングで変数値が変化/参照されるのかわかりにくい
間違いやすい

```
while (*s++)  
    len++;
```

(絶対の自信がない限り) 単純な書き方するのが何かと安全

```
while (*s != '\0') {  
    s++;  
    len++;  
}
```

あるいは

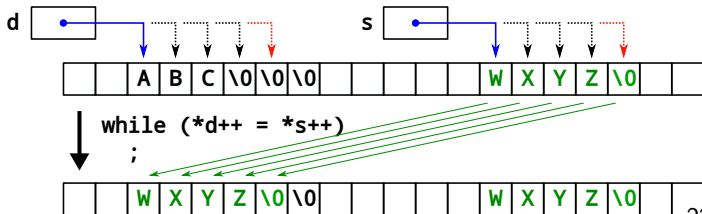
```
while (*s != '\0') {  
    s = s + 1;  
    len = len + 1;  
}
```

List 11-6 (部分) : 文字列をコピーする関数 (s の内容を d へ)

```
char *str_copy(char *d, const char *s)
{
    char *t = d;
    while (*d++ = *s++)
        ;
    return t;
}
```

やっていること

■ 文字列 s の先頭から 1 文字ずつ、文字列 d の先頭から書き写す
str_copy(d, s) の実行過程



str_copy 精読 : while 文の部分に注目

```
while (*d++ = *s++)  
    ;
```

*s++ の動作と値

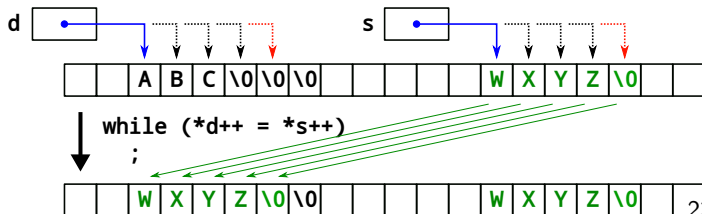
- s が指す内容を値とする
- s をインクリメント

*d++ = *s++ の動作

- s が指す内容を d が指す場所に書き込む (文字コピー)
- その後に d をインクリメント

while 文 : 0 (ナル文字 '\0') に会うまで繰り返す

- ナル文字 '\0' も書き込む (文字コピー)



while 文の部分の別の書き方

元: 不慣れなうちは戸惑う

```
while (*d++ = *s++)  
    ;
```

次も同様

```
while ((*d++ = *s++) != '\0')  
    ;
```

単純明快な別の書き方 (1)

```
int i = 0;  
while ((d[i] = s[i]) != '\0')  
    i++;
```

単純明快な別の書き方 (2): きっとこれが一番勘違いしにくい

```
for (int i = 0; s[i] != '\0'; i++)  
    d[i] = s[i];
```

今の時代のコンパイラを使えば多分どれも同程度の性能に最適化される

ポインタを返す関数 p.314

関数 `str_copy` は第一引数を返す

List 11-6 (部分; 再掲) : 文字列をコピーする関数

```
char *str_copy(char *d, const char *s)
{
    char *t = d;
    while (*d++ = *s++)
        ;
    return t;
}
```

関数の利用例 1 : 少々まどろっこしい

```
str_copy(str, tmp);
printf("str = \"%s\"\n", str);
```

関数の利用例 2 : 関数の返り値を利用すれば同じ動作を簡潔に書ける

```
printf("str = \"%s\"\n", str_copy(str, tmp));
```

(バグのあり) 文字列のコピー関数

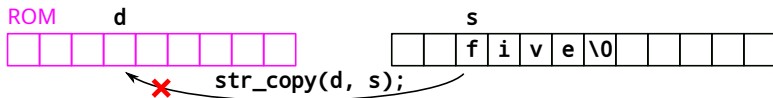
```
#include <stdio.h>
char *str_copy(char *d, const char *s)
{
    char *t = d;
    while (*d++ = *s++)
        ;
    return t;
}
int main(void)
{
    char *ptr = "1234";
    char tmp[128];
    ... 略 ...
    str_copy(ptr, tmp);
    ... 略 ...
}
```

バグ 1: 文字列リテラルへの書き込みを行っている

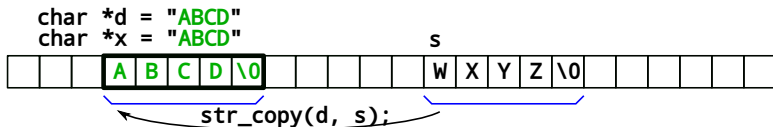
バグ 2: コピー先のサイズがコピー元よりも小さい

注意点 1 : 文字列リテラルへ書き込みしてはダメ

ROM 領域に文字列リテラルが配置されている可能性あり
(書き込んでも値は変わらない)



RAM 領域に配置の場合でも同一の文字列リテラルと共有の場合あり
(別の文字列リテラルの値が変わってしまう)

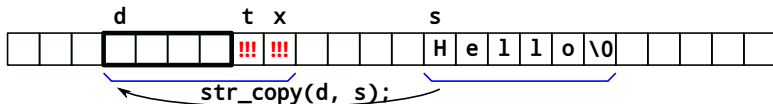


注意点 2 : コピー先の要素数を超えた書き込みはダメ

他の変数を上書きしてしまう

関数復帰のためのリターンアドレスが破壊される

(バッファオーバーフローの発生... セキュリティホールになる)



- 配列 d の要素数は 4
- 長さ 5 の文字列 (合計 6 要素) を書き込んでいる
- d の範囲を超えて書き込まれる : 変数 t と x を上書き

第 11-3 節 文字列を扱うライブラリ 関数

インクルードすべきヘッダ

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

strlen(s) — 長さ

strncpy(s,t,n) — s に t を上書きコピー

strncat(s,t,n) — s の後に t を連結

strcmp(s1,s2) — 大小比較

strncmp(s1,s2,n) — 大小比較 (文字数限定)

strchr(s,c) — 文字 c の検索 (先頭から)

strrchr(s,c) — 文字 c の検索 (末尾から)

strstr(s,t) — 文字中の文字列の検索

strdup(s) — s のコピーを生成

strlen : 文字列の長さ

size_t strlen(const char *s)

- 文字列 s の長さを返す (ナル文字は含まない)

例 : strlen(s) = 5

| | | | | | | | | | | | | |
|--|--|--|--|---|---|---|---|---|----|--|--|--|
| | | | | | | | | | | | | |
| | | | | s | | | | | | | | |
| | | | | H | e | l | l | o | \0 | | | |

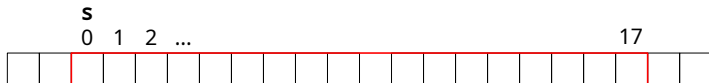
strncpy : 文字列のコピー (文字数上限つき)

`char *strncpy(char *dest, const char *src, size_t n)`

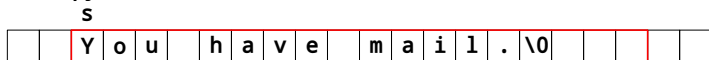
- 文字列 `dest` に文字列 `src` をコピー
- コピーするのは `src` の `n` 文字まで
- 要注意:

`n` 文字ちょうどのコピーの場合 `dest` は `\0` で終端されない

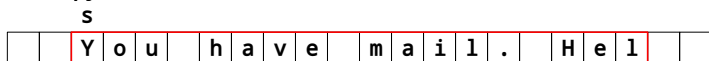
例



`strncpy(s, "You have mail.", 18)`



`strncpy(s, "You have mail. Hello world.", 18)`



strncat : 文字列の連結 (文字数上限つき)

`char *strncat(char *dest, const char *src, size_t n)`

- 文字列 `dest` の末尾に文字列 `src` の内容を連結
- `dest` の長さの制限は `n`

例

`s`
0 1 2 ... 17

| | | | | | | | | | | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|----|--|--|--|--|--|--|--|--|
| | | Y | o | u | | h | a | v | e | \0 | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|----|--|--|--|--|--|--|--|--|



`strncat(s, "mail.", 18)`

`s`

| | | | | | | | | | | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|----|--|--|
| | | Y | o | u | | h | a | v | e | | m | a | i | l | . | \0 | | |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|----|--|--|



`strncat(s, "mail. Welcome!", 18)`

`s`

| | | | | | | | | | | | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|--|---|---|----|
| | | Y | o | u | | h | a | v | e | | m | a | i | l | . | | W | e | \0 |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|--|---|---|----|

strcmp : 文字列の比較

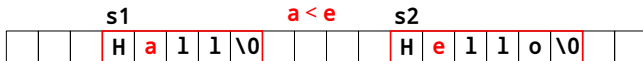
```
int strcmp(const char *s1, const char *s2)
```

- 文字列 s1 と s2 を比較
- cmp : compare (比較)

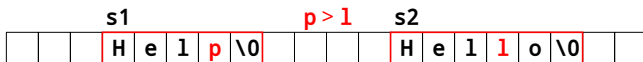
s1 = s2 の場合 : strcmp(s1,s2) = 0



s1 < s2 の場合 : strcmp(s1,s2) < 0



s1 > s2 の場合 : strcmp(s1,s2) > 0

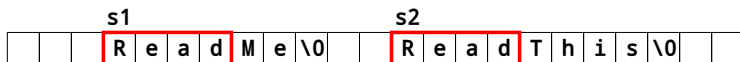


strncmp : 文字列の比較 (比較文字数の上限つき)

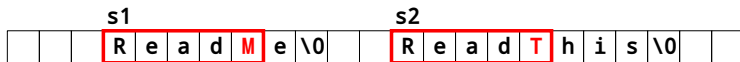
```
int strncmp(const char *s1, const char *s2, size_t n)
```

- 文字列 s1 と s2 を比較
- ただし先頭から n 文字までに限定
- s1 = s2 の場合 : 返り値 0
- s1 < s2 の場合 : 返り値 < 0
- s1 > s2 の場合 : 返り値 > 0

例 1: `strncmp(s1,s2,4) = 0`



例 2: `strncmp(s1,s2,5) < 0`

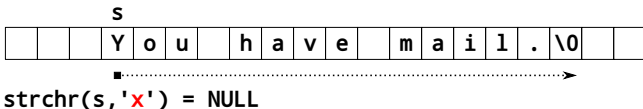
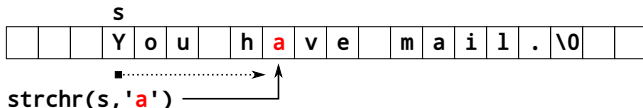


strchr : 文字の検索 (先頭から)

`char *strchr(const char *s, int c)`

- 文字列 `s` の中の文字 `c` を検索
- 検索は文字列の先頭から
- 見つければその文字へのポインタを返す
- 見つからなければ `NULL` ポインタを返す

例

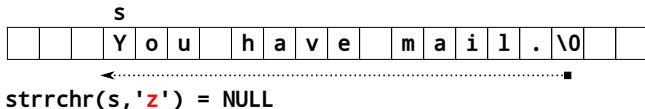
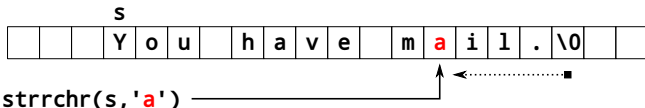


strrchr : 文字の検索 (末尾から)

`char *strrchr(const char *s, int c)`

- 文字列 `s` の中の文字 `c` を検索
- 検索は文字列の末尾から
- 見つければその文字へのポインタを返す
- 見つからなければ `NULL` ポインタを返す

例

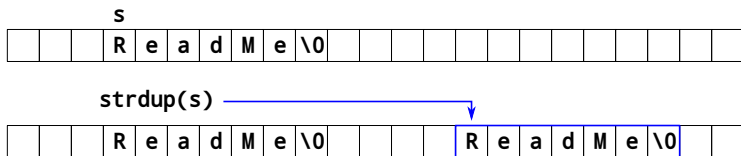


strdup : 文字列の複製の生成

`char *strdup(const char *s)`

- 新たな文字配列 (メモリ) を割り当て文字列 `s` をコピー
- 新たな文字配列へのポインタを返す (複製)
- (その文字列が不要になれば `free()` でメモリ解放すること)

例



dup : duplicate (複製)

使用しない方が良い関数

危険!! 使用を避けるべき関数 (その 1): strcpy

```
char *strcpy(char *s1, const char *s2)
```

- 文字列をコピーする関数
- コピー先の文字列配列 s1 の大きさ指定ができない
- s2 が長いとバッファオーバーフローが発生
- 代替: strncpy()

バッファオーバーフローが発生する例

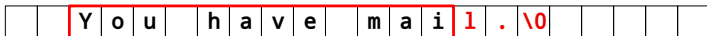
```
char s[12]
```

S

x y z i i



```
strcpy(s, "You have mail.")
```



危険!! 使用を避けるべき関数 (その 2) : strcat

```
char *strcat(char *s1, const char *s2)
```

- 文字列を連結する関数
- 連結結果を保持する文字列配列 s1 の大きさ指定ができない
- 連結結果が長いとバッファオーバーフローが発生
- 代替: `strncat()`, `snprintf()`

バッファオーバーフローが発生する例

```
char s[12]
```

S

x y z i i

| | | | | | | | | | | | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|----|--|--|--|--|--|--|--|--|--|
| | | Y | o | u | | h | a | v | e | \0 | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|----|--|--|--|--|--|--|--|--|--|

```
strcat(s, "mail.")
```

| | | | | | | | | | | | | | | | | | | | | | | |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|----|--|--|--|--|--|--|
| | | Y | o | u | | h | a | v | e | | m | a | i | l | . | \0 | | | | | | |
|--|--|---|---|---|--|---|---|---|---|--|---|---|---|---|---|----|--|--|--|--|--|--|

Q：そんな関数がなんで標準 C ライブラリに含まれているの？

A：バックワードコンパチビリティ（後方互換）のためだけです

- 昔のソースコードを今でも使用可能にするため仕方なく
- 新規の使用は非推奨

背景

- 昔のプログラマは皆 `strcat` や `strcpy` の正しい使い方を知っていた
- たとえバグがあっても被害は知れていた
- 計算機はネットワークに繋がっていなかった

現在

- 今は世界中からネットワーク経由で攻撃される
- `strcat` や `strcpy` の意図的な正しくない使い方をして攻撃
- 重大な被害が発生（個人データ流出/機密情報流出/etc.）

atoi・atol・atoll・atof：文字列を数値に変換

"123" や "51.7" のような文字列を数値に変換

インクルードすべきヘッダ

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

```
int atoi(const char *s)
```

— s が指す文字列を int 型の表現に変換

```
long atol(const char *s)
```

— s が指す文字列を long 型の表現に変換

```
long long atoll(const char *s)
```

— s が指す文字列を long long 型の表現に変換

```
double atof(const char *s)
```

— s が指す文字列を double 型の表現に変換

(まめ知識) 関数名の由来：ASCII to { integer, long, ... }

atoi 関数の使用例

List 11-12 : atoi 関数の使用例

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char str[128];
    printf("文字列を入力してください：");
    scanf("%s", str);
    printf("整数に変換すると%dです。 \n", atoi(str));
    return 0;
}
```

scanf は使わずに fgets を使いましょうね

(独自) 文字列のプログラミング

文字列の照合

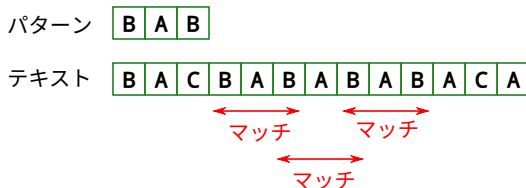
文字列の照合問題

入力

- テキスト (文書文字列; 通常は長い)
- パターン (検索文字列; 通常は短い)

出力

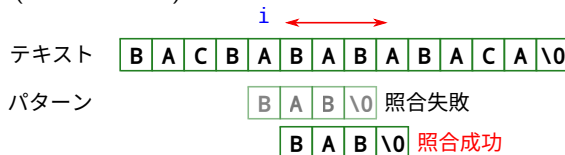
- テキスト中にパターンに一致する部分があるか否か (Yes/No)
- その位置 (何文字目か: 0 以上の値. なければ -1)



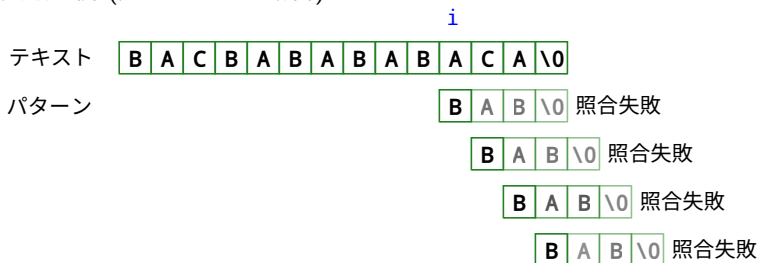
文字列の照合, 概要

テキスト s の i 文字目から
パターン pat を照合

■ 成功の例 (見つかる場合)



■ 失敗の例 (見つからない場合)



文字列の照合のアルゴリズム

テキスト *s* の中からパターン *pat* を検索

- ただし文字列 *s* の *i* 文字目から検索開始
- 最初に見つかった場所 (*s* の添字) を返す (見つからなければ -1)

```
int str_match(char s[], int i, char pat[])
{
    for (int j = i; s[j] != '\0'; j++) {
        for (int k = 0; ; k++) {
            if (pat[k] == '\0') { /* patの終端 */
                return j; /* 照合成功; マッチした位置 */
            }
            if (s[j+k] == '\0') { /* sの終端 */
                break; /* 照合失敗 */
            }
            if (s[j+k] != pat[k]) { /* 異なる文字 */
                break; /* 照合失敗 */
            }
        }
    }
    return -1; /* 照合失敗 */
}
```

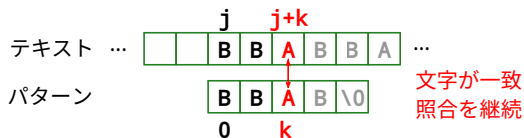
文字列の照合アルゴリズムの利用例

テキスト中に出現するパターンを全てを見つけて表示

```
int i = 0;
while (i >= 0) {
    i = str_match(s, i, pat); /*添字 i より照合開始 */
    if (i >= 0) { /* 添字 i でマッチ */
        printf("MATCH POSITION: %d\n", i)
        i++; /*次の検索開始位置*/
    } /* i<0ならマッチする場所はない */
};
```

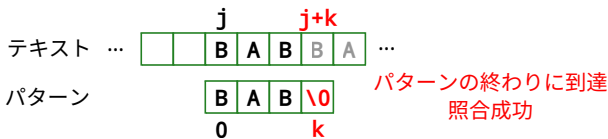
アルゴリズム解説 (0/3)

場合 0 : テキストとパターンの途中で文字が一致
次の文字を比較



アルゴリズム解説 (1/3)

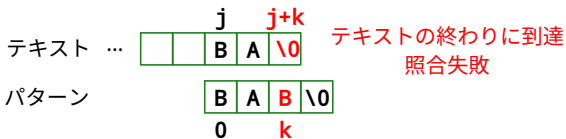
場合 1 : パターンの終わりに到達
照合成功



テキストとパターンは照合できている
(パターンの途中で失敗していない)

アルゴリズム解説 (2/3)

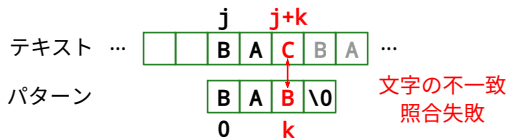
場合 2 : テキストの終わりに到達 (かつパターンの途中)
照合失敗



パターンの全文字が照合できていない
(パターンの途中で失敗した)

アルゴリズム解説 (3/3)

場合 3 : テキストとパターンの文字が不一致
照合失敗



パターンの全文字が照合できていない
(パターンの途中で失敗した)

おわり

番外編の課題 1

文字列前後の不要な空白を除去する関数の作成

■ `void trim_spc_head_tail(char *s);`

例

| | |
|--------------------------------|--------------------------------|
| <code>" abc "</code> | <code>"abc"</code> に書き換え |
| <code>" abc"</code> | <code>"abc"</code> に書き換え |
| <code>"abc "</code> | <code>"abc"</code> に書き換え |
| <code>" abc xyz "</code> | <code>"abc xyz"</code> に書き換え |
| <code>" "</code> | <code>" "</code> に書き換え |
| <code>""</code> | <code>""</code> のまま |
| <code>"abc"</code> | <code>"abc"</code> のまま |
| <code>"abc xyz"</code> | <code>"abc xyz"</code> のまま |

番外編の課題 2

2 つ以上続く空白を 1 つにする関数の作成

■ `void uniq_spc(char *s);`

例

| | |
|--------------------------------|----------------------------------|
| <code>"abc xyz hij"</code> | <code>"abc xyz hij"</code> に書き換え |
| <code>" abc "</code> | <code>" abc "</code> に書き換え |
| <code>"abc xyz "</code> | <code>"abc xyz"</code> に書き換え |
| <code>" abc xyz "</code> | <code>"abc xyz "</code> に書き換え |
| <code>""</code> | <code>""</code> のまま |
| <code>"abc"</code> | <code>"abc"</code> のまま |
| <code>"abc xyz"</code> | <code>"abc xyz"</code> のまま |