
プログラミング応用演習 I

第3週

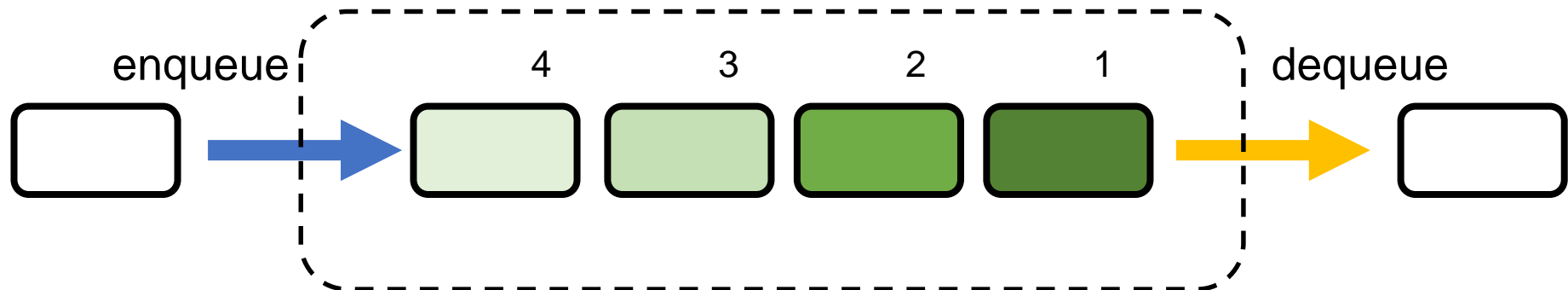
キュー、スタック

第3回のアルゴリズム

- キュー/Queue
 - データを保存するためのデータ構造
 - Data structures for storing data
 - FIFO(First In First Out)
- スタック/Stack
 - データを保存するためのデータ構造
 - Data structures for storing data
 - FILO(First In Last Out)

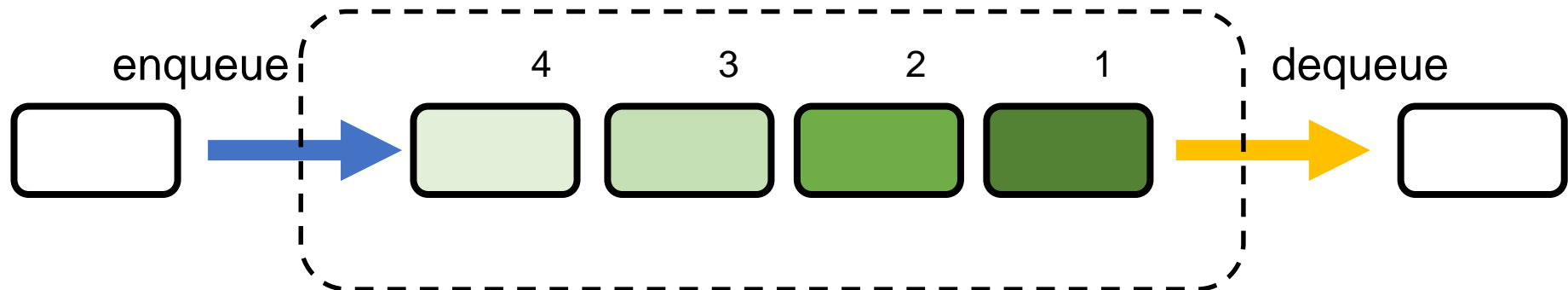
データ構造 / キュー (Queue)

- データを保存するためのデータ構造
 - 逐次入出力が繰り返されるデータを一時的に蓄える
- 順にデータを入力したとき、最初に追加されたデータが、最初に出てくる
 - FIFO(First in first out)と呼ばれる
 - 待ち行列と呼ばれることがある
 - データを入れることをエンキュー (enqueue)と言う
 - データを取り出すことをデキュー (dequeue)と言う
- 入ってきた順番にデータを処理したいときに使われる



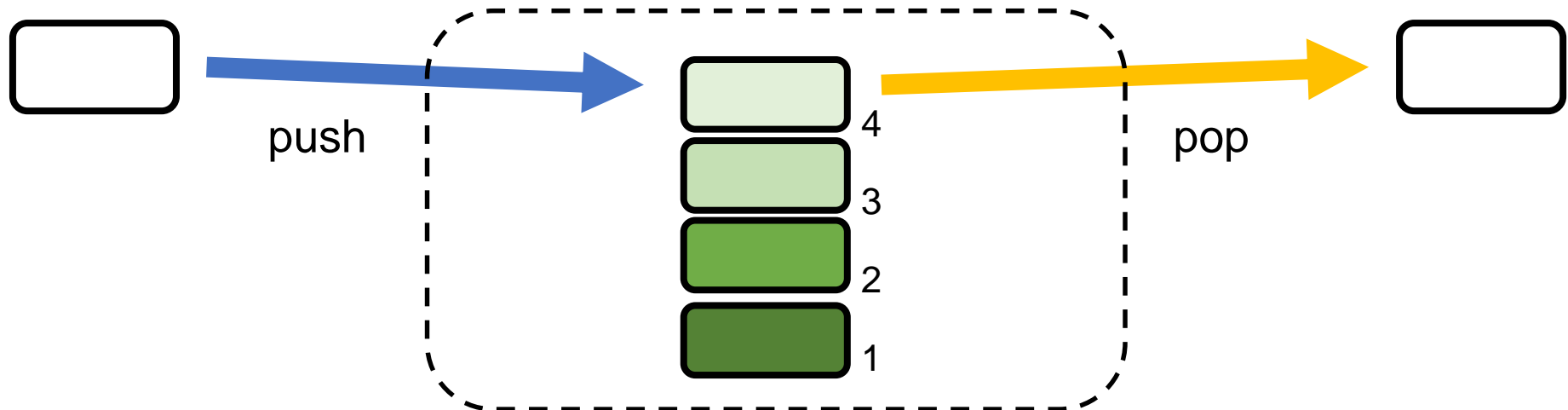
データ構造 / キュー (Queue)

- Data structures for storing data
 - Temporary storage of data with repeated sequential input/output
- When data is sequentially added, the first data added will get the first
 - Called FIFO(First in first out)
 - 待ち行列 in Japanese
 - The process of inserting data is called "enqueue".
 - The process of retrieving data is called "dequeue".
- Used when users need to process data in the order in which they come in.



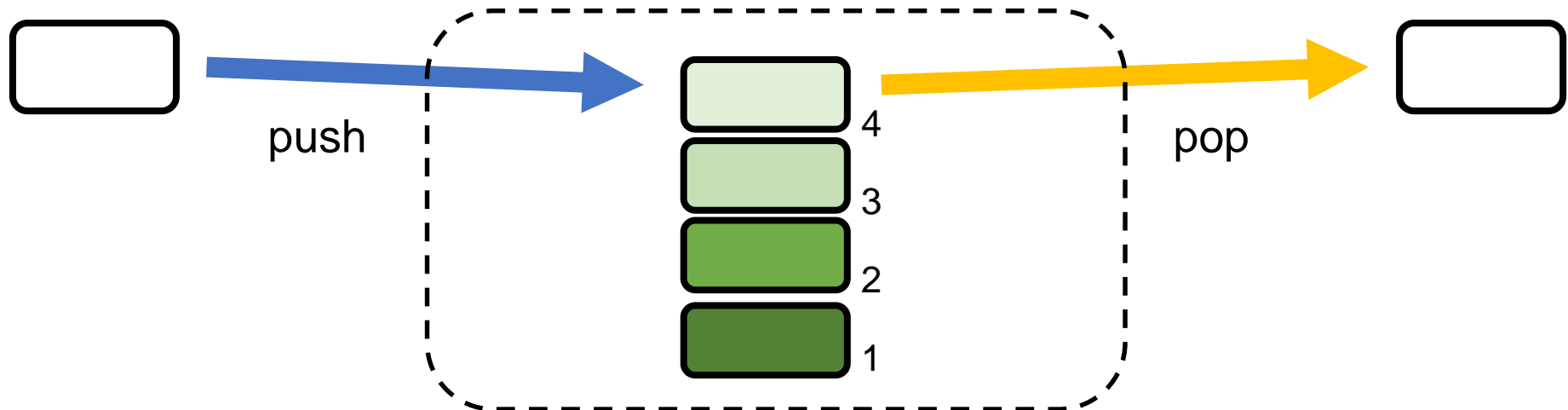
データ構造 / スタック (Stack)

- データを保存するためのデータ構造（ここはキューと同じ）
 - 逐次入出力が繰り返されるデータを一時的に蓄える
- 最後に入れたデータを最初に取り出す（最初に入れたデータが最後に出てくる）
 - LIFO (Last in first out) / FILO (First in last out) と呼ばれる
 - データを入れることをプッシュ (push) と言う
 - データを取り出すことをポップ (pop) と言う
- 計算機の実装に関わりが深い



データ構造 / スタック (Stack)

- Data structures for storing data (Same as Queue)
 - Temporary storage of data with repeated sequential input/output
- The last data you put in comes out first (the first data you put in comes out last)
 - Called LIFO (Last in first out) / FILO (First in last out)
 - The process of inserting data is called "push".
 - The process of retrieving data is called "pop".
 - Highly involved in the implementation of computers.



キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う / Using Structure
 - 関数へはポインターを渡す / Passing a pointer to a function

Header / Define / Structure

```
/* header files */
#include <stdio.h>
#include <stdlib.h>

/* define */
#define ARRAY_MAX 5
#define STR_MAX 256

/* structure */
typedef struct queue {
    int array[ARRAY_MAX];
    int front;
    int end;
} Queue;
```

Data storage(memory) array
Indicating start of data front
Indicating end of data end

Function Declarations

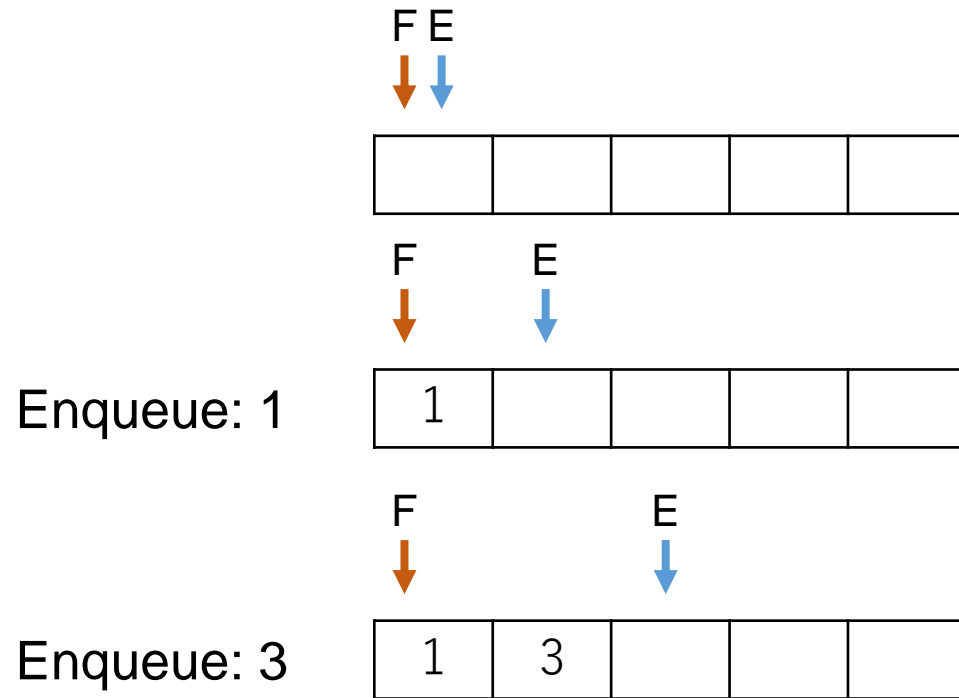
```
/* declearation functions */
void enqueue(Queue *x);
void dequeue(Queue *x);
void print_queue(Queue *x);
/* For Assignment 3.1 */
/* int find_next(int current); */

/* for debug */
void print_detailed_queue(Queue *x);
```

キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う
 - 関数へはポインターを渡す

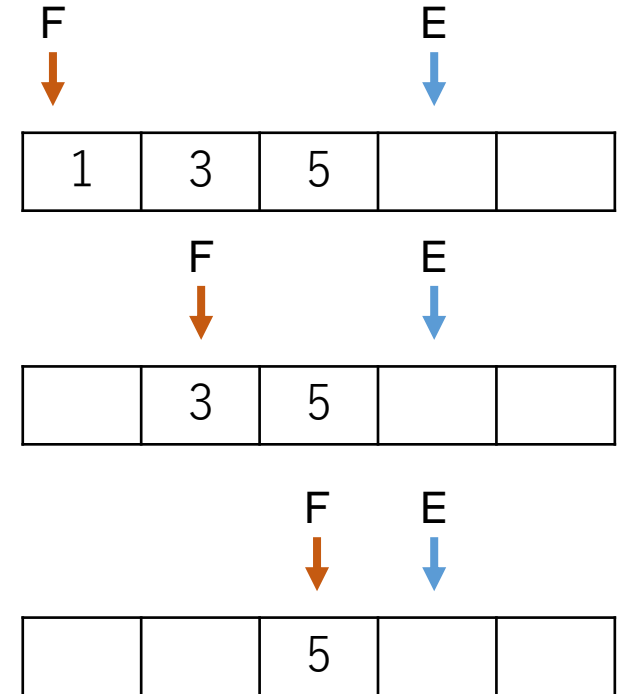
F / Queue.front
E / Queue.end



Enqueue: 5

Dequeue: (1)

Dequeue: (3)



キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う

Main

```
/* main */
int main(void) {
    /* Variables */
    Queue x;
    int num;
    char str[STR_MAX];

    /* Initial settings */
    x.front = 0;
    x.end = 0;

    /* Main loop */
    while(1) { /* Infinite loop */
        puts("Enter the operation number.");
        printf("1. Enqueue, 2. Dequeue, 3. Print, 4. Exit: ");
        fgets(str, sizeof(str), stdin);
        num = atoi(str);
```

Switch (selecting operation by a user input)

```
// selected operation for Queue x
switch(num) {
    case 1: enqueue(&x);
        break;
    case 2: dequeue(&x);
        break;
    case 3: print_queue(&x);
        break;
    case 4: exit(0);
    case 9: print_detailed_queue(&x); /* not explained */
        break;
    default: printf("-- You input wrong key [%d] --", num);
}
```

Get user input
Selecting operation

キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う

```
/* enqueue */
void enqueue(Queue *x) {
    char str[STR_MAX];
    if (x->end == ARRAY_MAX) {
        puts("-- No resource remained. --");
    } else {
        printf("Input a number to enqueue: ");
        fgets(str, sizeof(str), stdin);
        x->array[x->end++] = atoi(str);
    }
}
```

In assignment 3.1
IF find_next(x->end) == x->front then
"No resource remained"

In assignment 3.1
x->array[x->end] = atoi(str);
x->end = find_next(x->end);

```
/* dequeue */
void dequeue(Queue *x) {
    if (x->front == x->end) {
        puts("-- There is no data. --");
    } else {
        printf("Dequeue: %d\n", x->array[x->front++]);
    }
}
```

In assignment 3.1
deleted_num = x->array[x->front];
x->front = find_next(x->front);

キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う

```
/* print all data */
void print_queue(Queue *x) {
    int i;
    if (x->front == x->end) {
        puts("-- There is no data. --");
    } else {
        for (i = x->front; i < x->end; i++) {
            printf("%d ", x->array[i]);
        }
        putchar('\n');
    }
}
```

キューの実装 (Program of Queue)

- 配列での実装例 / ソースコードは Classroom の lecture_3_1.c
 - 構造体を使う

```
/* print all data (detailed) */
void print_detailed_queue(Queue *x) {
    int i;
    for (i = 0; i < ARRAY_MAX; i++) {
        if ((x->front < x->end) && (x->front <= i) && (i < x->end)) {
            /* 終端に到達していない時 */
            printf("%3.3d : %3.3d : %d¥n", i, i - x->front, x->array[i]);
        } else if ((x->front > x->end) && (x->front >= i) && (x->end < i)) {
            /* 終端を超えた時 */
            printf("%3.3d : %3.3d : %d¥n", i, i - x->end, x->array[i]);
        } else {
            /* No data */
            printf("%3.3d : ***¥n", i);
        }
    }
}
```

スタックを使った計算機の実装(逆ポーランド記法)

- 逆ポーランド記法

- $\langle \text{値1} \rangle \langle \text{値2} \rangle \langle \text{演算子} \rangle$ の並びのとき $\langle \text{値1} \rangle \langle \text{演算子} \rangle \langle \text{値2} \rangle$ の形で計算する
- 括弧の処理が要らない
- 逆ポーランド記法がスタックというデータ構造と極めて相性がよい

Example of reverse polish

$2\ 3\ +\ 4\ 5\ +\ *$

$\Rightarrow (2 + 3) * (4 + 5) = 45$

Reverse polish algorithm using stack

```
While there are tokens left,  
  Read a token,  
    If the token is a value,  
      Push it onto the stack.  
    Else if the token is an operator,  
      Pop the top two values from the stack.  
      Evaluate the operator for those values.  
      Push the result back onto the stack.  
If there is only one value on the stack,  
  Return the value as a result of calculation.
```

Program using Stack (Reverse polish)

- Reverse polish

- Representing $\langle \text{value1} \rangle \langle \text{value2} \rangle \langle \text{operator} \rangle$ means to
calculate $\langle \text{value1} \rangle \langle \text{operator} \rangle \langle \text{value2} \rangle$
- Processing of brackets is not required.
- Reverse Polish notation is highly compatible with the data structure of a stack.

Example of reverse polish

2 3 + 4 5 + *

$\Rightarrow (2 + 3) * (4 + 5) = 45$

Reverse polish algorithm using stack

While there are tokens left,

Read a token,

If the token is a value,

Push it onto the stack.

Else if the token is an operator,

Pop the top two values from the stack.

Evaluate the operator for those values.

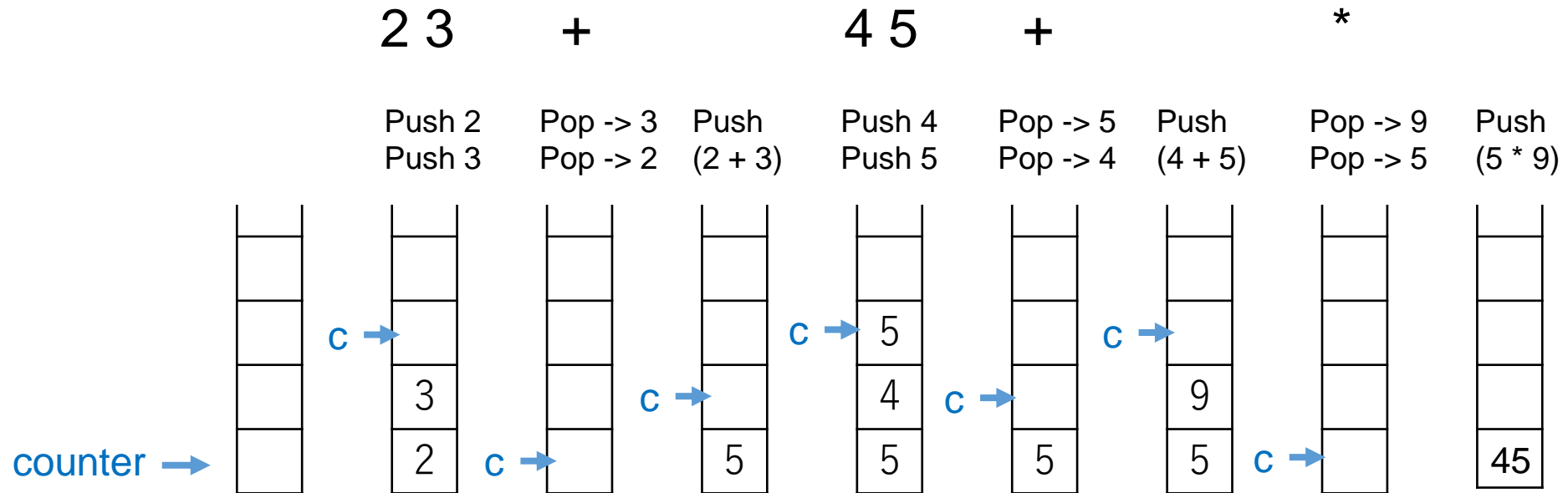
Push the result back onto the stack.

If there is only one value on the stack,

Return the value as a result of calculation.

• 逆ポーランド記法

- $\langle \text{値} \rangle \langle \text{値} \rangle \langle \text{演算子} \rangle$ の並びのとき $\langle \text{値} \rangle \langle \text{演算子} \rangle \langle \text{値} \rangle$ の形で計算する
- 括弧の処理が要らない
- 逆ポーランド記法がスタックというデータ構造と極めて相性がよい



スタックの実装1

- ソースコードは Classroom の lecture_3_2.c

```
/* header files */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
/* define */
#define MAX 10
/* structure */
typedef struct stack {
    double array[MAX];
    int counter;
} Stack;
/* functions */
void read_rpn(void);
void push(Stack *x, double n);
double pop(Stack *x);

/* main */
int main(void) {
    read_rpn();
    return 0;
}
```

• ヒント

- push
 - $x \rightarrow \text{counter} \geq \text{MAX}$
 - オーバーフローが起こる
- pop
 - $x \rightarrow \text{counter} \leq 0$
 - アンダーフローが起こる

• スタック構造体

- array データ格納用配列
- counter データ個数を表す

• スタック操作関数

- この関数(push, pop)の実装が課題3.2です
- いずれも、操作するStack構造体を参照渡しで引数にとります
- Push / 引数 n をスタックに積む(返回值なし)
- Pop / スタックから値を取り出して返回值とする

Program of Stack 1

- Source code is lecture_3_2.c in Classroom

```
/* header files */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
/* define */
#define MAX 10
/* structure */
typedef struct stack {
    double array[MAX];
    int counter;
} Stack;
/* functions */
void read_rpn(void);
void push(Stack *x, double n);
double pop(Stack *x);

/* main */
int main(void) {
    read_rpn();
    return 0;
}
```

- Hint

- push
 - $x \rightarrow \text{counter} \geq \text{MAX}$
 - An overflow occurs
- pop
 - $x \rightarrow \text{counter} \leq 0$
 - An underflow occurs

- Struct of representing stack

- array Array for storing data
- counter Representing number of data

- Functions for operating stack

- Functions(push, pop) are assignment 3.2
- Both take a Stack structure as a pass-by-reference argument
- Push / Stacking argument n (no return value)
- Pop / Retrieve a value from the stack and return it

スタックの実装2

- ソースコードは Classroom の lecture_3_2.c

```
/* four basic arithmetic operations on positive
real numbers */
void read_rpn(void) {
    Stack x;
    int ch;
    double num1, num2;
    x.counter = 0;
    printf("Enter an equation in Reverse Polish
Notation: ");
    while((ch = getchar()) != EOF) {
        if (isdigit(ch)) { // the token is a digit
            // convert a character code to a number
            ungetc(ch, stdin);
            scanf("%lf", &num1);
            push(&x, num1);
        } else { // the token is not a digit
```

- メインの処理を行う関数
 - mainからread_rpn関数のみを読んでいる
- 文字の読み込み
 - ch に入力文字を入れる
- 数字の読み込み
 - ch が 数字だったら / 一つ戻して
 - 数字を読み込む
- 数字だったら
 - pushする

Program of Stack 2

- Source code is lecture_3_2.c in Classroom

```
/* four basic arithmetic operations on positive
real numbers */
void read_rpn(void) {
    Stack x;
    int ch;
    double num1, num2;
    x.counter = 0;
    printf("Enter an equation in Reverse Polish
Notation: ");
    while((ch = getchar()) != EOF) {
        if (isdigit(ch)) { // the token is a digit
            // convert a character code to a number
            ungetc(ch, stdin);
            scanf("%lf", &num1);
            push(&x, num1);
        } else { // the token is not a digit
```

- Main processing function
 - Only read_rpn function is called in main
- Read character
 - Read character from console, substitute for ch
- Read number
 - If ch is number / return one character
 - Read number
- If it is a number
 - **Push** it to stack

スタックの実装3

```
} else { // the token is not a digit
    switch(ch) {
        case '+':
            num2 = pop(&x); num1 = pop(&x);
            push(&x, num1 + num2);
            break;
        case '-':
            num2 = pop(&x); num1 = pop(&x);
            push(&x, num1 - num2);
            break;
        case ' ': break;
        case '\n':
            if (x.counter <= 1) {
                printf("Answer: %.15f\n", pop(&x));
                exit(0);
            } else {
                puts("Stack is not empty.");
                exit(1);
            }
        default:
            puts("Wrong characters entered.");
            exit(1);
    }
}
```

ex_3_2.c

- 数字でない場合の処理
 - switch文で分岐する
- 入力が '+' の場合
 - **pop**を2回する
 - popした2つの数字を **足す**

他の演算子を省略している

- 入力が **改行** の場合
 - スタックに値が1つだったら
 - **Pop**して回答として表示

Program of Stack 3

```
} else { // the token is not a digit
    switch(ch) {
        case '+':
            num2 = pop(&x); num1 = pop(&x);
            push(&x, num1 + num2);
            break;
        case '-':
            num2 = pop(&x); num1 = pop(&x);
            push(&x, num1 - num2);
            break;
        case ' ': break;
        case '\n':
            if (x.counter <= 1) {
                printf("Answer: %.15f\n", pop(&x));
                exit(0);
            } else {
                puts("Stack is not empty.");
                exit(1);
            }
        default:
            puts("Wrong characters entered.");
            exit(1);
    }
}
```

e_3_2.c

- Processing, if ch is not a number
 - Branching using switch statement
- If ch is '+'
 - Pop twice
 - Add the two numbers popped

Other operators are omitted.

- If ch is Enter
 - If there is only one value on the stack
 - Pop and it shows as the answer

第3回、第4回に出てくるC言語の使い方

- 構造体

- ほかの言語ではオブジェクトやクラスに相当
- 複数の型のデータをまとめて一つの型として扱えるようにしたもの

- ポインター

- 基本的にC言語で記述している変数はメモリ領域とセットです
- メモリ領域の場所(アドレス)を表すのがポインターです
- 関数にポインターを渡すと副作用が得られます
 - 渡した変数の内容を書き換えることができる

- 動的なメモリ領域の確保

- 実行時に大きさを決めてメモリを確保することが出来ます
- 通常の変数宣言は静的な領域確保

Using C language

- Struct
 - Equivalent to objects or classes in other languages
 - Data of multiple types combined together to be handled as a single type
- Pointer
 - Basically, the variables described in C are associated with a memory area.
 - The pointer represents the location (address) of the memory area.
 - Passing a pointer to a function provides side effects
 - Can modify the contents of a passed variable.
- Dynamic memory allocation
 - The size of the memory can be determined at runtime and allocated
 - Normal variable declaration is static memory allocation

構造体の例

- 複数の型のデータをまとめて一つの型として扱えるようにしたもの
 - 構造体内のそれぞれのデータ変数をメンバと呼ぶ
 - 関数内での変数宣言と同じように書く
- 配列は同じ型のデータを複数まとめたもの

構造体型定義

```
/* structure 構造体型定義 */
struct queue {
    int array[ARRAY_MAX]; //配列メンバ
    int front; //メンバ
    int end; //メンバ
};

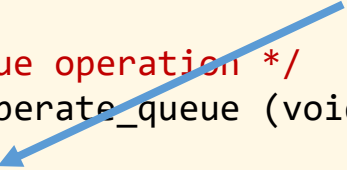
/* queue operation */
void operate_queue (void) {
    struct queue x; /* 構造体の変数宣言*/
    x.front = 10; // メンバへの代入
}
```

typedefの使い方

```
/* structure 構造体型定義 */
struct queue {
    int array[ARRAY_MAX];
    int front;
    int end;
};

/* typedef 宣言 */
typedef struct queue Queue;

/* queue operation */
void operate_queue (void) {
    Queue x; // Queue型が使える
}
```



typedefの使い方2

```
/* structure 構造体型定義 */
typedef struct queue {
    int array[ARRAY_MAX];
    int front;
    int end;
} Queue;

/* typedef と struct定義を
同時に書いている */

/* queue operation */
void operate_queue (void) {
    Queue x; // Queue型が使える
}
```


Example of Struct

- Data of multiple types combined together to be handled as a single type
 - Each variable in the struct is called "member variable".
 - Write the same as variable declarations in a function.

Struct type declaration

```
/* struct type declaration */
struct queue {
    int array[ARRAY_MAX]; //member array
    int front; //member variable
    int end; //member variable
};
```

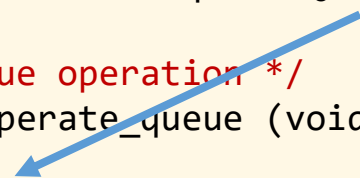
```
/* queue operation */
void operate_queue (void) {
    struct queue x;
    x.front = 10; //
}
```

Using typedef

```
/* struct type declaration */
struct queue {
    int array[ARRAY_MAX];
    int front;
    int end;
};
```

```
/* typedef declaration */
typedef struct queue Queue;
```

```
/* queue operation */
void operate_queue (void) {
    Queue x; // Using Queue type
}
```



Using typedef2

```
/* struct type declaration */
typedef struct queue {
    int array[ARRAY_MAX];
    int front;
    int end;
} Queue;
/* Using typedef and struct
at the same time */
```

```
/* queue operation */
void operate_queue (void) {
    Queue x; // Using Queue type
}
```

構造体

- 通常の変数の型を組み込み型と言います
 - 整数(符号付き、符号なし) char, short, int, long, long long
 - 浮動小数点 float, double
- 構造体は組み込み型、構造体 と その配列をメンバーにできます
 - 通常の変数の宣言と同じようにする
- 宣言した構造体は型として認識され、変数や引数の型として使えます
- メンバへのアクセスは、. (ドット)で行います

```
/* queue operation */  
void operate_queue (void) {  
    Queue x; // Queue型が使える  
    x.front = 1; // メンバアクセス  
}
```

Struct

- Normal variable types are called built-in types
 - Integer(signed, unsigned) / char, short, int, long, long long
 - Floating point number / float, double
- Struct can be built-in types, struct and their arrays can be members
 - Declare as normal variable declarations.
- The declared struct is recognized as a type and can be used as a variable or argument type
- Members are accessed by . (dot)

```
/* queue operation */  
void operate_queue (void) {  
    Queue x; // Queue type  
    x.front = 1; // accessing member  
}
```

ポインター

- 基本的にC言語で記述している変数はメモリ領域とセットです
 - メモリ領域の場所(アドレス)を表すのがポインターです
 - 関数にポインターを渡すと副作用が得られます
 - 渡した変数の内容を書き換えることができる
- 変数は(コンパイル後の実行時)にメモリ上に配置されます(スタックと呼ぶ)
- 動的(実行時)にメモリ領域を確保することもできます(ヒープと呼ぶ)
- ポインターはそのアドレスモデルを直接扱うもので、他の言語にはあまり無く最初は理解が難しいと思います
- 使えるメモリ領域は決まっていて、不正なアドレスを使うとエラーとなります
 - segmentation fault等の実行時エラーになります
 - 確保された領域のアドレスをポインターとして使用すべきです

Pointer

- Basically, the variables described in C are associated with a memory area.
 - The pointer represents the location (address) of the memory area.
 - Passing a pointer to a function provides side effects
 - Can modify the contents of a passed variable.
- Variables are placed in memory (while compilation) (called the stack)
- Dynamic (runtime) memory allocation is also possible. (called heap)
- Pointers directly handle that address model, which is not common in other languages and may be difficult to understand at first.
- The memory area that can be used is limited, and an error will occur if an invalid address is used.
 - Run-time errors occur, such as segmentation fault, etc.
 - The address of the allocated area should be used as a pointer

ポインターに関する文法

- 変数のポインターを得る <変数名>

```
int x = 10;  
int *ptr = &x;
```

- ポインター変数の宣言 型名 *<変数名>;

- ポインター変数は必ず正しいアドレスを入れる必要があります
- 例: 変数のポインターを代入、確保した領域を代入

```
int x = 10;  
int *ptr = &x;
```

ptrが変数名のポインター変数の宣言
int型のポインター

- ポインター変数の内容を得る(読み込み、書き込みが可能) *<変数名> = 値;

```
int y = *ptr;  
*ptr = 20;
```

Grammar about pointers


- Get a pointer of the variable `&<variable name>`

```
int x = 10;  
int *ptr = &x;
```

- Declaration of Pointer Variable `type *<variable name>;`
 - Pointer variables must always be set to the correct address
 - Example: Assign a pointer to a variable, Assign an allocated region

```
int x = 10;  
int *ptr = &x;
```

Declaration of pointer variable (ptr is the variable name)
Pointer of int type



- Get the contents of a pointer variable (can be read or written)

```
int y = *ptr;  
*ptr = 20;
```

```
*<variable name> = value;
```

ポインターに関する文法

- 構造体のポインターのとき
 - メンバへのアクセスが `->` になる
 - (通常は `.` でアクセス)

```
/* 構造体メンバアクセス*/  
Queue q;  
q.front = 10;
```

```
/* 構造体メンバアクセス */  
/* ポインター */  
Queue *q_ptr = &q;  
q_ptr->front = 20;
```


Grammar about pointers

- For a pointer of a struct
 - Accessing members by “->”
 - (Accessing members by “.”, normal variable)

```
/* Accessing members */  
Queue q;  
q.front = 10;
```

```
/* Accessing members */  
/* pointer */  
Queue *q_ptr = &q;  
q_ptr->front = 20;
```

ポインター(メモリ領域とポインター)

- 変数はメモリに配置され、メモリにはアドレスがついています
 - ポインターはメモリのアドレスを表します
 - アドレスは16進数で表しています(0x始まり)
 - アドレス一つには1バイトの値が保持されています

アドレス	内容
0xFFA4A900	--
0xFFA4A904	20
0xFFA4A908	--
0xFFA4A90C	--

← int x = 20; // 変数の宣言 (メモリに配置)

int *ptr = &x;
// このときの ptr は (アドレスの値)

*ptr = 11;
// こうすると?

ポインター(メモリ領域とポインター)

- 変数はメモリに配置され、メモリにはアドレスがついています
 - ポインターはメモリのアドレスを表します
 - アドレスは16進数で表しています(0x始まり)
 - アドレス一つには1バイトの値が保持されています

アドレス	内容
0xFFA4A900	--
0xFFA4A904	20 -> 11
0xFFA4A908	--
0xFFA4A90C	--

← int x = 20; // 変数の宣言 (メモリに配置)

int *ptr = &x;
// このときの ptr は (アドレスの値)

*ptr = 11;
// こうすると? (アドレスへ内容を記入)

x の値は??? (11になっている)

ポインター(メモリ領域とポインター)

- 変数はメモリに配置され、メモリにはアドレスがついています
 - ポインターはメモリのアドレスを表します
 - アドレスは16進数で表しています(0x始まり)
 - アドレス一つには1バイトの値が保持されています

アドレス	内容
0xFFA4B900	10
0xFFA4B904	20
0xFFA4B908	30
0xFFA4B90C	40

```
int ary[4] = {10, 20, 30, 40};  
// 変数配列の宣言 (メモリに配置)
```

int ary[0]

int ary[1]

int ary[2]

int ary[3]

&ary

// aryのポインターは

Pointer(memory address and pointer)

- Variables are allocated in memory, and memory has an address
 - Pointer represents an address
 - Addresses are expressed in hexadecimal (start with 0x)
 - One address holds one byte of value

Address	Contents
0xFFA4A900	--
0xFFA4A904	20
0xFFA4A908	--
0xFFA4A90C	--

← int x = 20; // Declaration of Variables
int *ptr = &x;
// In this case, what is ptr (address value)

*ptr = 11;
// What if you do this?

Pointer(memory address and pointer)

- Variables are allocated in memory, and memory has an address
 - Pointer represents an address
 - Addresses are expressed in hexadecimal (start with 0x)
 - One address holds one byte of value

Address	Contents
0xFFA4A900	--
0xFFA4A904	20 -> 11
0xFFA4A908	--
0xFFA4A90C	--

int x = 20; // Declaration of Variables

int *ptr = &x;

// In this case, what is ptr (address value)

*ptr = 11;

// What if you do this?

Contents of x ??? (x is 11)

Pointer(memory address and pointer)

- Variables are allocated in memory, and memory has an address
 - Pointer represents an address
 - Addresses are expressed in hexadecimal (start with 0x)
 - One address holds one byte of value

Address	Contents
0xFFA4A900	10
0xFFA4A904	20
0xFFA4A908	30
0xFFA4A90C	40

```
int ary[4] = {10, 20, 30, 40};  
// Declaration of Variable Arrays
```

int ary[0]

int ary[1]

int ary[2]

int ary[3]

&ary
// pointer of ary

ポインター(ポインター渡しによる副作用)

- 関数の引数を、型にすると値渡し、ポインターにすると参照渡しと呼ぶ
- 値渡しの場合は、値がコピーされる
- 参照渡しはポインターの値が渡るので、ポインターの内容が書き換わる
- 関数の中で変数が書き換わって欲しいときは参照渡しをする

```
/* pass-by-value */
void func_variable(int i)
{
    i = -1;
}
/* pass-by-reference */
void func_pointer(int *ptr)
{
    *ptr = -1;
}
```

```
int x = 10;
func_variable(x);
// x == 10

func_pointer(&x);
// x == -1
```


Pointer (side effect of passing pointer to a function)

- A function argument is a type, called pass-by-value
 - it is a pointer, called pass-by-reference
- The value is copied, when pass-by-value
- The pointer is passed, when pass-by-reference
 - Then, contents of variable can be modified
- When you want the value of a variable to be changed in a function, use pass-by-reference

```
/* pass-by-value */  
void func_variable(int i)  
{  
    i = -1;  
}  
/* pass-by-reference */  
void func_pointer(int *ptr)  
{  
    *ptr = -1;  
}
```

```
int x = 10;  
func_variable(x);  
// x == 10  
  
func_pointer(&x);  
// x == -1
```

ポインターを理解するサンプル1 / Example for pointer1

- Classroom に pointer_sample.c があるので、それをコンパイルして実行
- Using pointer_sample.c in classroom

```
/* struct definition */
typedef struct queue {
    double array[5];
    int front;
    int end;
} Queue;

/* pass-by-value */
void func_variable(Queue q)
{
    q.front = -1;
    q.end = -1;
}

/* pass-by-reference */
void func_pointer(Queue *q)
{
    q->front = -1;
    q->end = -1;
}
```

ポインターを理解するサンプル2 / Example for pointer2

```
Queue q; /* struct as a variable */
q.front = 3;
q.end   = 7;
q.array[4] = 3.33;

Queue q_array[10]; /* array of struct */
q_array[0].front = 9;
q_array[0].end   = 11;
q_array[2].front = 13;
q_array[2].end   = 17;

Queue *q_ptr; /* pointer */
/* do not use q_ptr before initializing */
/* memory allocation */
q_ptr = (Queue *)malloc(sizeof(Queue) * 10);
printf("sizeof(Queue) = %ld\n", sizeof(Queue));
q_ptr->front = 2;
q_ptr->end   = 5;
q_ptr->array[0] = 4.5;

/* variable => pointer */
Queue *q_ptr2 = &q;
printf("q_ptr2->front = %d\n", q_ptr2->front);
printf("q_ptr2->end   = %d\n", q_ptr2->end);
printf("q_ptr2->array[4] = %lf\n", q_ptr2->array[4]);

/* pointer => variable */
printf("(q_ptr).front = %d\n", (q_ptr).front);
printf("(q_ptr).end   = %d\n", (q_ptr).end);
```

で、それをコンパイルして実行

<実行結果の出力/output>

sizeof(Queue) = 48

q_ptr2->front = 3

q_ptr2->end = 7

q_ptr2->array[4] = 3.330000

(q_ptr).front = 2

(q_ptr).end = 5

ポインターを理解するサンプル3 / Example for pointer3

- Classroom に pointer_sample.c があるので、それをコンパイルして実行
- Using pointer_sample.c in classroom

```
/* pass-by-value, pass-by-reference */
func_variable(q);
puts("pass-by-value");
printf("q.front = %d¥n", q.front);
printf("q.end   = %d¥n", q.end);
puts("");

func_pointer(&q);
puts("pass-by-reference");
printf("q.front = %d¥n", q.front);
printf("q.end   = %d¥n", q.end);
puts("");
```

<実行結果の出力/output>

pass-by-value

q.front = 3

q.end = 7

pass-by-reference

q.front = -1

q.end = -1

ポインターを理解するサンプル4 / Example for pointer4

```
/* addressing pointer */
Queue *q_ptr3; /* pointer */
/* do not use q_ptr before initializing */
/* memory allocation */
q_ptr3 = (Queue *)malloc(sizeof(Queue) * 10);
/* access as array */
q_ptr3[2].front = 12;
q_ptr3[2].end   = 13;
puts("add pointer value");
printf("(q_ptr3 + 2)->front = %d\n", (q_ptr3 + 2)->front);
printf("(q_ptr3 + 2)->end   = %d\n", (q_ptr3 + 2)->end);
puts("");

Queue *tmp_ptr = q_ptr3;
tmp_ptr++;
tmp_ptr++;
puts("increment pointer(address)");
printf("tmp_ptr->front = %d\n", tmp_ptr->front);
printf("tmp_ptr->end   = %d\n", tmp_ptr->end);
puts("");

/* address */
printf("%lx : q_ptr3\n", (unsigned long)q_ptr3);
printf("%lx : tmp_ptr\n", (unsigned long)tmp_ptr);
printf("tmp_ptr - q_ptr3 = %ld\n", (unsigned long)tmp_ptr - (unsigned
long)q_ptr3);
puts("");

printf("%lx : q_ptr3 + 3\n", (unsigned long)(q_ptr3 + 2));
printf("%lx : &(q_ptr3[3])\n", (unsigned long)&(q_ptr3[2]));
```

れをコンパイルして実行

<実行結果の出力/output>

add pointer value

(q_ptr3 + 2)->front = 12

(q_ptr3 + 2)->end = 13

increment pointer(address)

tmp_ptr->front = 12

tmp_ptr->end = 13

564c753b88a0 : q_ptr3

564c753b8900 : tmp_ptr

tmp_ptr - q_ptr3 = 96

564c753b8900 : q_ptr3 + 3

564c753b8900 : &(q_ptr3[3])

動的メモリ領域の確保

- 動的なメモリ領域の確保

- 通常の変数宣言は静的な領域確保
- 違いは領域の大きさをプログラムを書くときに決める必要があるかどうか

```
/* main */
int main(void) {
    // local memory allocation 静的確保
    Queue q_var;

    // dynamic memory allocation 動的確保
    Queue *q_ptr;
    q_ptr = (Queue *)malloc(sizeof(Queue));

    // how to access the members
    q_var.front = 2;
    q_ptr->front = 2;

    return 0;
}
```

ポインターへのアクセスを使う

```
/* main */
int main(void) {
    // local memory allocation 静的確保
    Queue q_array[10];

    // dynamic memory allocation 動的確保
    Queue *q_array_ptr;
    q_array_ptr = (Queue *)malloc(sizeof(Queue) * 10);

    // how to access the members
    q_array[2].front = 2;
    q_array_ptr[2].front = 2;

    return 0;
}
```

配列へのアクセス

q_array と q_array_ptr は同じように使える
(コンパイル時は同じ型として処理される)

Dynamic memory allocation

- Dynamic memory allocation
 - Normal variable declaration is static memory allocation
 - The difference is whether the size of the area needs to be determined when writing the program

```
/* main */
int main(void) {
    // local memory allocation 静的確保
    Queue q_var;

    // dynamic memory allocation 動的確保
    Queue *q_ptr;
    q_ptr = (Queue *)malloc(sizeof(Queue));

    // how to access the members
    q_var.front = 2;
    q_ptr->front = 2;

    return 0;
}
```

Accessing to pointer

```
/* main */
int main(void) {
    // local memory allocation 静的確保
    Queue q_array[10];

    // dynamic memory allocation 動的確保
    Queue *q_array_ptr;
    q_array_ptr = (Queue *)malloc(sizeof(Queue) * 10);

    // how to access the members
    q_array[2].front = 2;
    q_array_ptr[2].front = 2;

    return 0;
}
```

Accessing to array

q_array and q_array_ptr can be used as the same


その他のC言語の構文

- Switch文

- 値が一致するケース(case)に飛ぶ
- if/else if/else を使えば同じことが書ける

```
// Syntax of switch
switch( num ) {
case 1:
    /* process of case 1 */
    break;
case 2:
    /* process of case 2 */
    break;
/* add other cases */
default:
    /* process of default */
    /* there is no case for num */
}
```

num == 1 ならここが実行される
break; が無いと次の行も実行される



Other C language syntax

- Switch statement
 - Jump to the case with matching value
 - You can write the same thing using if/else if/else

```
// Syntax of switch
switch( num ) {
case 1:
    /* process of case 1 */
    break;
case 2:
    /* process of case 2 */
    break;
/* add other cases */
default:
    /* process of default */
    /* there is no case for num */
}
```

If num == 1, processing here,
Without break; the next line is also processed

