Bing
Here is a sample program in C++ that implements a circular queue of capacity (n-1) elements using an array of b elements, where the insertion and deletion operations are carried out using REAR and FRONT as array index variables respectively. Initially, REAR = FRONT = 0.

```
#include <iostream>
using namespace std;

const int n = 10; // capacity of the queue
int queue[n], REAR = 0, FRONT = 0;

void enqueue(int x) {
    if ((REAR + 1) % n == FRONT) {
        cout << "Queue is full." << endl;
    } else {
        queue[REAR] = x;
        REAR = (REAR + 1) % n;
    }
}

void dequeue() {
    if (REAR == FRONT) {
        cout << "Queue is empty." << endl;
    } else {
        cout << "Deleted element: " << queue[FRONT] << endl;
        FRONT = (FRONT + 1) % n;
    }
}

int main() {
    int choice, x;
    while (true) {
        cout << "Enter 1 to insert element into queue." << endl;
        cout << "Enter 2 to delete element from queue." << endl;
        cout << "Enter 3 to exit." << endl;
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter element to be inserted: ";
                cin >> x;
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                exit(0);
            default:
                cout << "Invalid choice. Try again." << endl;
        }
    }
    return 0;
```

```
}
```

The above program demonstrates the implementation of a circular queue of capacity (n-1) elements using an array of b elements. The enqueue() function is used to insert elements into the queue, while the dequeue() function is used to delete elements from the queue. The program also includes checks for queue full and queue empty conditions.
Please note that the program assumes that the user will input valid integers when prompted. If the user inputs invalid data, the program may behave unexpectedly.
I hope this helps! Let me know if you have any other questions.

Bing
Here is a shorter version of the C program for implementing a queue of integers that can check if the queue is empty, delete elements from the queue, and insert elements into the queue:

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

#define n 10 // capacity of the queue
int queue[n], REAR = 0, FRONT = 0;

void enqueue(int x) {
    if ((REAR + 1) % n == FRONT) {
        printf("Queue is full.\n");
    } else {
        queue[REAR++] = x;
        REAR %= n;
    }
}

void dequeue() {
    if (REAR == FRONT) {
        printf("Queue is empty.\n");
    } else {
        printf("Deleted element: %d\n", queue[FRONT++]);
        FRONT %= n;
    }
}

int main() {
    int choice, x;
    while (1) {
        printf("Enter 1 to insert element into queue.\n");
        printf("Enter 2 to delete element from queue.\n");
        printf("Enter 3 to exit.\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter element to be inserted: ");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                exit(0);
            default:
                printf("Invalid choice. Try again.\n");
        }
    }
    return 0;
}
```

This program uses an array to implement the queue data structure. The enqueue function inserts an element into the queue, while the dequeue function removes an element from the queue. The main function provides a menu to the user to choose between inserting an element, deleting an element, or exiting the program. If the user tries to insert an element into a full queue, or delete an element from an empty queue, the program will display an appropriate message.

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

struct customer {
    int customer_id;
    char purchase_date[11];
    float purchase_amount;
};

int compare_customers(const void *a, const void *b) {
    const char *date_a = ((struct customer *)a)->purchase_date;
    const char *date_b = ((struct customer *)b)->purchase_date;

    int year_comparison = strncmp(date_a, date_b, 4);
    if (year_comparison != 0) {
        return year_comparison;
    }

    int month_comparison = strncmp(date_a + 5, date_b + 5, 2);
    if (month_comparison != 0) {
        return month_comparison;
    }

    return strncmp(date_a + 8, date_b + 8, 2);
}

int main() {
    int n, i;
    printf("Enter the number of customers: ");
    scanf("%d", &n);

    struct customer *customers = malloc(n * sizeof(struct customer));
    if (customers == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("Enter details of customer %d:\n", i + 1);
        printf("Customer ID: ");
        scanf("%d", &customers[i].customer_id);
        printf("Purchase Date (YYYY-MM-DD): ");
        scanf("%s", customers[i].purchase_date);
        printf("Purchase Amount: ");
        scanf("%f", &customers[i].purchase_amount);
    }

    qsort(customers, n, sizeof(struct customer), compare_customers);

    printf("\nCustomer data sorted by purchase date in ascending
order:\n");
    printf("Customer ID\tPurchase Date\tPurchase Amount\n");
```

```
    for (i = 0; i < n; i++) {
        printf("%d\t\t%s\t%.2f\n", customers[i].customer_id,
customers[i].purchase_date, customers[i].purchase_amount);
    }

    free(customers);
    return 0;
}
```

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

struct customer {
    int customer_id;
    char purchase_date[11];
    float purchase_amount;
};

int compare_dates(const void *a, const void *b) {
    return strcmp(((struct customer *)a)->purchase_date, ((struct customer
*)b)->purchase_date);
}

int main() {
    int n, i;
    printf("Enter the number of customers: ");
    scanf("%d", &n);

    struct customer *customers = malloc(n * sizeof(struct customer));
    if (customers == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("Enter details of customer %d:\n", i + 1);
        printf("Customer ID: ");
        scanf("%d", &customers[i].customer_id);
        printf("Purchase Date (YYYY-MM-DD): ");
        scanf("%s", customers[i].purchase_date);
        printf("Purchase Amount: ");
        scanf("%f", &customers[i].purchase_amount);
    }

    qsort(customers, n, sizeof(struct customer), compare_dates);

    printf("\nCustomer data sorted by purchase date in ascending
order:\n");
    printf("Customer ID\tPurchase Date\tPurchase Amount\n");

    for (i = 0; i < n; i++) {
        printf("%d\t\t%s\t%.2f\n", customers[i].customer_id,
customers[i].purchase_date, customers[i].purchase_amount);
    }

    free(customers);
    return 0;
}
```

167. evaluating the arithmetic expression 2*3-(4+5), using stacks to evaluate postfix form
pseudocode
1. Initialize an empty stack
2. Split the postfix expression into elements
3. For each element in the postfix expression:
   a. If the element is an operand:
      i. Convert the operand to an integer and push it onto the stack
   b. If the element is an operator:
      i. Pop operand2 from the stack
      ii. Pop operand1 from the stack
      iii. Perform the operation with operand1 and operand2
      iv. Push the result back onto the stack
4. The final result is on the top of the stack
5. Pop the result from the stack
6. Output the result

Program

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

#define MAX_SIZE 100

// Stack structure
typedef struct {
    int items[MAX_SIZE];
    int top;
} Stack;

// Function prototypes
void push(Stack *s, int value);
int pop(Stack *s);
int evaluatePostfix(char *postfix);

int main() {
    // Postfix expression: 2 3 * 4 5 + -
    char postfix[] = "2 3 * 4 5 + -";

    int result = evaluatePostfix(postfix);

    printf("Result: %d\n", result);

    return 0;
}

void push(Stack *s, int value) {
    if (s->top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    } else {
        s->top++;
        s->items[s->top] = value;
```

```
    }
}

int pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    } else {
        return s->items[s->top--];
    }
}

int evaluatePostfix(char *postfix) {
    Stack s;
    s.top = -1;

    for (int i = 0; postfix[i] != '\0'; i++) {
        if (isdigit(postfix[i])) {
            // Operand, push onto the stack
            push(&s, postfix[i] - '0');
        } else if (postfix[i] == ' ') {
            // Skip whitespace
            continue;
        } else {
            // Operator, perform the operation
            int operand2 = pop(&s);
            int operand1 = pop(&s);

            switch (postfix[i]) {
                case '+':
                    push(&s, operand1 + operand2);
                    break;
                case '-':
                    push(&s, operand1 - operand2);
                    break;
                case '*':
                    push(&s, operand1 * operand2);
                    break;
                // Add more cases for other operators if needed
            }
        }
    }

    // The final result is on the top of the stack
    return pop(&s);
}
```

9.multiple stacks may be implemented using one array. where b1,b2,...bn denote bottom of respective stacks and arrow indicate direction of growth. Out of the above two alternatives which one would you prefer and why? Which alternative will need more time to declare stack full? It may be noted that stack is declared full when all the stacks are full.

PseudoCode
Data Structures:

- array: A single array to hold elements for multiple stacks.
- bottom_pointers: An array to keep track of the bottom of each stack.

Functions:

b. initializeMultipleStacks(total_size, stack_count):
    - Allocate memory for the array and bottom_pointers.
    - Set stack_size as total_size / stack_count.
    - Initialize bottom_pointers for each stack.

c. isFull(stack_number):
    - Check if the stack with stack_number is full.
    - Return true if the bottom pointer is at the upper limit for that stack.

d. push(stack_number, value):
    - Check if the stack is full using isFull function.
    - If not full, calculate the index in the array based on the bottom pointer.
    - Store the value at the calculated index.
    - Increment the bottom pointer.

e. pop(stack_number):
    - Check if the stack is not empty (bottom pointer > lower limit).
    - If not empty, decrement the bottom pointer.
    - Return the element at the updated bottom pointer.

f. freeMultipleStacks():
    - Free the allocated memory for the array and bottom_pointers.

Main Algorithm:

g. Initialize the multiple stacks using initializeMultipleStacks(total_size, stack_count).

h. Perform push and pop operations as needed on different stacks using push(stack_number, value) and pop(stack_number).

i. Free the allocated memory using freeMultipleStacks() when done.

program

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

#define MAX_STACKS 3 // Adjust the number of stacks as needed

typedef struct {
int total_size;
int stack_size;
int *array;
int *bottom_pointers;
} MultipleStacks;

MultipleStacks* initializeMultipleStacks(int total_size, int stack_count) {
MultipleStacks stacks = (MultipleStacks)malloc(sizeof(MultipleStacks));
stacks->total_size = total_size;
stacks->stack_size = total_size / stack_count;
stacks->array = (int*)malloc(sizeof(int) * total_size);
stacks->bottom_pointers = (int*)malloc(sizeof(int) * stack_count);

for (int i = 0; i < stack_count; i++) {
    stacks->bottom_pointers[i] = i * stacks->stack_size;
}

return stacks;

}

int isFull(MultipleStacks *stacks, int stack_number) {
return stacks->bottom_pointers[stack_number] == (stack_number + 1) * stacks->stack_size;
}

void push(MultipleStacks *stacks, int stack_number, int value) {
if (!isFull(stacks, stack_number)) {
int index = stacks->bottom_pointers[stack_number];
stacks->array[index] = value;
stacks->bottom_pointers[stack_number]++;
} else {
printf("Stack %d is full. Cannot push element %d.\n", stack_number, value);
}
}

int pop(MultipleStacks *stacks, int stack_number) {
if (stacks->bottom_pointers[stack_number] > stack_number * stacks->stack_size) {
stacks->bottom_pointers[stack_number]--;
return stacks->array[stacks->bottom_pointers[stack_number]];
} else {
printf("Stack %d is empty. Cannot pop.\n", stack_number);
return -1; // Assuming -1 represents an invalid value; adjust as needed
}
}
```

```
void freeMultipleStacks(MultipleStacks *stacks) {
free(stacks->array);
free(stacks->bottom_pointers);
free(stacks);
}

int main() {
MultipleStacks *stacks = initializeMultipleStacks(9, MAX_STACKS);

push(stacks, 0, 1);
push(stacks, 0, 2);
push(stacks, 1, 3);
push(stacks, 1, 4);
push(stacks, 2, 5);
push(stacks, 2, 6);

// Print the array after pushing elements
for (int i = 0; i < stacks->total_size; i++) {
    printf("%d ", stacks->array[i]);
}
printf("\n");

printf("Popped from stack 0: %d\n", pop(stacks, 0));
printf("Popped from stack 1: %d\n", pop(stacks, 1));
printf("Popped from stack 2: %d\n", pop(stacks, 2));

freeMultipleStacks(stacks);

return 0;

}
```

Bing
Here is a shorter version of the C++ program for implementing two stacks that can push and pop elements, and takes input from the user:

```
#include <iostream>
using namespace std;

class TwoStacks {
    int* arr;
    int size;
    int top1, top2;

public:
    TwoStacks(int n) {
        size = n;
        arr = new int[n];
        top1 = -1;
        top2 = size;
    }

    void push1(int x) {
        if (top1 < top2 - 1) {
            arr[++top1] = x;
        } else {
            cout << "Stack Overflow" << endl;
            exit(1);
        }
    }

    void push2(int x) {
        if (top1 < top2 - 1) {
            arr[--top2] = x;
        } else {
            cout << "Stack Overflow" << endl;
            exit(1);
        }
    }

    int pop1() {
        if (top1 >= 0) {
            return arr[top1--];
        } else {
            cout << "Stack Underflow" << endl;
            exit(1);
        }
    }

    int pop2() {
        if (top2 < size) {
            return arr[top2++];
        } else {
            cout << "Stack Underflow" << endl;
            exit(1);
        }
```

```
    }
};

int main() {
    int n, x;
    cout << "Enter the size of the array: ";
    cin >> n;
    TwoStacks ts(n);
    ts.push1(10);
    ts.push2(20);
    ts.push1(30);
    ts.push2(40);

    cout << "Popped element from stack1 is " << ts.pop1() << endl;
    cout << "Popped element from stack2 is " << ts.pop2() << endl;

    return 0;
}
```

This program uses a class to implement two stacks that can push and pop elements. The main function provides a prompt to the user to enter the size of the array, and then displays the popped elements from both the stacks..

Bing
Here is a shorter version of the C++ program for implementing infix to postfix conversion that takes input from the user:

```
#include <iostream>
#include <stack>
#include <string>

using namespace std;

int precedence(char op) {
    if (op == '$') {
        return 3;
    } else if (op == '*' || op == '/') {
        return 2;
    } else if (op == '+' || op == '-') {
        return 1;
    } else {
        return -1;
    }
}

string infixToPostfix(string exp) {
    stack<char> stack;
    string result;

    for (int i = 0; i < exp.length(); i++) {
        char c = exp[i];

        if (isalnum(c)) {
            result += c;
        } else if (c == '(') {
            stack.push(c);
        } else if (c == ')') {
            while (!stack.empty() && stack.top() != '(') {
                result += stack.top();
                stack.pop();
            }
            if (!stack.empty() && stack.top() != '(') {
                return "Invalid Expression";
            } else {
                stack.pop();
            }
        } else {
            while (!stack.empty() && precedence(c) <=
precedence(stack.top())) {
                result += stack.top();
                stack.pop();
            }
            stack.push(c);
        }
    }

    while (!stack.empty()) {
```

```
        result += stack.top();
        stack.pop();
    }

    return result;
}

int main() {
    string exp;
    cout << "Enter an infix expression: ";
    getline(cin, exp);
    cout << "Infix expression: " << exp << endl;
    cout << "Postfix expression: " << infixToPostfix(exp) << endl;
    return 0;
}
```

This program uses the getline function to take input from the user. The input is stored in the variable exp. The infixToPostfix function converts the infix expression to postfix expression using a stack data structure. The main function provides a prompt to the user to enter an infix expression, and then displays the infix expression and the corresponding postfix expression.