信息学竞赛

GESP C++ 六级认证（四）

数据结构与面向对象专题

类与对象 · 栈 · 队列 · 循环队列

大纲

1. 面向对象编程基础
2. 栈的类实现与应用
3. 队列的类实现与应用
4. 循环队列的类实现与应用
5. 典型应用场景分析

面向对象编程基础

类与对象的概念

类 是对象的蓝图或模板，定义了对象的属性和行为。

对象 是类的实例，具有具体的属性值。

类的三大特性

- **封装**：将数据和方法包装在一起
- **继承**：从已有类创建新类
- **多态**：同一接口不同实现

类的基本结构

```
class ClassName {
  private:
    // 私有成员，只能在类内部访问
    data_type private_var;

  public:
    // 公有成员，可以在类外部访问
    data_type public_var;

    // 构造函数：创建对象时自动调用
    ClassName(parameters);

    // 析构函数：对象销毁时自动调用
    ~ClassName();

    // 成员函数
    return_type function_name(parameters);
};
```

栈的基本概念

栈的定义与特性

栈 是一种后进先出（LIFO）的线性数据结构：

- **基本操作**：
  - `push(x)`：将元素 x 压入栈顶
  - `pop()`：弹出栈顶元素
  - `top()`：获取栈顶元素
  - `empty()`：判断栈是否为空
- **时间复杂度**：
  - 所有操作：$O(1)$

栈的数组实现

数据结构定义

```
const int MAX_SIZE = 1000;

// 栈数据结构
int stack[MAX_SIZE];  // 存储栈元素的数组
int top = -1;         // 栈顶指针，初始为-1表示空栈
```

栈操作函数

```
// 判断栈是否为空
bool stack_empty() { return top == -1; }

// 判断栈是否已满
bool stack_full() { return top == MAX_SIZE - 1; }

// 入栈操作
bool stack_push(int value) {
    if (stack_full()) {
        cout << "Stack overflow!" << endl;
        return false;
    }
    stack[++top] = value;
    return true;
}

// 出栈操作
bool stack_pop() {
    if (stack_empty()) {
        cout << "Stack underflow!" << endl;
        return false;
    }
    top--;
    return true;
}

// 获取栈顶元素
int stack_top() {
    if (stack_empty()) {
        cout << "Stack is empty!" << endl;
        return -1;
    }
    return stack[top];
}

// 获取栈大小
int stack_size() { return top + 1; }
```

栈的应用：括号匹配

问题描述

检查一个字符串中的括号是否匹配，包括 `()`，`[]`，`{}`。

算法思路

- 遍历字符串，遇到左括号入栈
- 遇到右括号时，检查栈顶是否匹配
- 最后栈应为空

括号匹配实现

```
bool is_balanced(string expr) {
    // 使用字符栈
    char char_stack[MAX_SIZE];
    int char_top = -1;

    for (char ch : expr) {
        if (ch == '(' || ch == '[' || ch == '{')
            // 左括号入栈
            char_stack[++char_top] = ch;
        else if (ch == ')' || ch == ']' || ch == '}') {
            // 右括号检查匹配
            if (char_top == -1)
                return false;

            char top_char = char_stack[char_top--];

            if ((ch == ')' && top_char != '(') ||
                (ch == ']' && top_char != '[') ||
                (ch == '}' && top_char != '{'))
                return false;
        }
    }

    return char_top == -1; // 栈应为空
}
```

队列的基本概念

队列的定义与特性

队列 是一种先进先出（FIFO）的线性数据结构：

- **基本操作**：
  - `enqueue(x)`：将元素 x 加入队尾
  - `dequeue()`：从队首移除元素
  - `front()`：获取队首元素
  - `empty()`：判断队列是否为空
- **时间复杂度**：
  - 所有操作：$O(1)$

队列的数组实现

数据结构定义

```
const int MAX_SIZE = 1000;

// 队列数据结构
int queue[MAX_SIZE];   // 存储队列元素的数组
int front = 0;         // 队首指针
int rear = -1;         // 队尾指针
int count = 0;         // 队列中元素个数
```

队列操作函数

```
// 判断队列是否为空
bool queue_empty() { return count == 0; }

// 判断队列是否已满
bool queue_full() { return count == MAX_SIZE; }

// 入队操作
bool queue_enqueue(int value) {
    if (queue_full()) {
        cout << "Queue is full!" << endl;
        return false;
    }
    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = value;
    count++;
    return true;
}
```

```
// 出队操作
bool queue_dequeue() {
    if (queue_empty()) {
        cout << "Queue is empty!" << endl;
        return false;
    }
    front = (front + 1) % MAX_SIZE;
    count--;
    return true;
}

// 获取队首元素
int queue_front() {
    if (queue_empty()) {
        cout << "Queue is empty!" << endl;
        return -1;
    }
    return queue[front];
}

// 获取队列大小
int queue_size() { return count; }
```

队列应用：广度优先搜索

BFS 算法框架

```
void bfs(int start, int graph[][100], int n) {
    bool visited[100] = {false};

    // 初始化队列
    int bfs_queue[MAX_SIZE];
    int q_front = 0, q_rear = -1, q_count = 0;

    // 起点入队
    bfs_queue[++q_rear] = start;
    q_count++;
    visited[start] = true;

    while (q_count > 0) {
        int current = bfs_queue[q_front];
        q_front = (q_front + 1) % MAX_SIZE;
        q_count--;

        cout << "Visiting: " << current << endl;

        // 遍历相邻节点
        for (int i = 0; i < n; i++) {
            if (graph[current][i] == 1 && !visited[i]) {
                bfs_queue[++q_rear] = i;
                q_count++;
                visited[i] = true;
            }
```

循环队列的实现

循环队列的概念

循环队列 通过循环使用数组空间来解决普通队列的空间浪费问题：

- **特点**：
  - 队尾可以循环回到数组开头
  - 更有效地利用存储空间
  - 避免数据搬移操作

循环队列的数组实现

数据结构定义

```
const int MAX_SIZE = 1000;

// 循环队列数据结构
int circular_queue[MAX_SIZE];
int cq_front = 0;      // 队首指针
int cq_rear = 0;       // 队尾指针
bool cq_full = false; // 队列满标志
```

循环队列操作函数

```
// 判断循环队列是否为空
bool cq_empty() { return (cq_front == cq_rear) && !cq_full; }

// 判断循环队列是否已满
bool cq_full() { return cq_full; }

// 入队操作
bool cq_enqueue(int value) {
    if (cq_full) {
        cout << "Circular queue is full!" << endl;
        return false;
    }
    circular_queue[cq_rear] = value;
    cq_rear = (cq_rear + 1) % MAX_SIZE;
    // 检查队列是否已满
    if (cq_rear == cq_front)
        cq_full = true;

    return true;
}

// 出队操作
bool cq_dequeue() {
    if (cq_empty()) {
        cout << "Circular queue is empty!" << endl;
        return false;
    }
    cq_front = (cq_front + 1) % MAX_SIZE;
    cq_full = false; // 出队后队列不可能满
    return true;
}
```

```
// 获取队首元素
int cq_front() {
    if (cq_empty()) {
        cout << "Circular queue is empty!" << endl;
        return -1;
    }
    return circular_queue[cq_front];
}

// 获取队列大小
int cq_size() {
    if (cq_full)
        return MAX_SIZE;
    if (cq_rear >= cq_front)
        return cq_rear - cq_front;
    return MAX_SIZE - cq_front + cq_rear;
}
```

综合应用：表达式求值

问题描述

实现一个简单的表达式求值器，支持加减乘除运算。

算法思路

- 使用两个栈：操作数栈和运算符栈
- 处理运算符优先级
- 从左到右扫描表达式

表达式求值实现

```
const int MAX_SIZE = 1000;

// 操作数栈
int num_stack[MAX_SIZE];
int num_top = -1;

// 运算符栈
char op_stack[MAX_SIZE];
int op_top = -1;

// 获取运算符优先级
int get_priority(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// 执行运算
int calculate(int a, int b, char op) {
    switch (op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return a / b;
    default: return 0;
    }
}
```

```
// 表达式求值
int evaluate_expression(string expr) {
    for (int i = 0; i < expr.length(); i++) {
        char ch = expr[i];

        if (isdigit(ch)) {
            // 处理数字
            int num = 0;
            while (i < expr.length() && isdigit(expr[i])) {
                num = num * 10 + (expr[i] - '0');
                i++;
            }
            i--;
            num_stack[++num_top] = num;
        } else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            // 处理运算符
            while (op_top >= 0 &&
                   get_priority(op_stack[op_top]) >= get_priority(ch)) {
                int b = num_stack[num_top--];
                int a = num_stack[num_top--];
                char op = op_stack[op_top--];
                num_stack[++num_top] = calculate(a, b, op);
            }
            op_stack[++op_top] = ch;
        }
    }

    // 处理剩余的运算符
    while (op_top >= 0) {
        int b = num_stack[num_top--], a = num_stack[num_top--];
        char op = op_stack[op_top--];
        num_stack[++num_top] = calculate(a, b, op);
    }

    return num_stack[num_top];
}
```

性能分析与比较

时间复杂度比较

数据结构	插入操作	删除操作	查找操作
栈	$O(1)$	$O(1)$	$O(n)$
队列	$O(1)$	$O(1)$	$O(n)$
循环队列	$O(1)$	$O(1)$	$O(n)$

空间复杂度

- 所有实现：$O(n)$，其中 n 为数组大小

内存管理注意事项

1. 边界检查

```
// 在每次操作前检查边界条件
bool stack_push(int value) {
    if (top == MAX_SIZE - 1) {  // 检查栈满
        cout << "Stack overflow!" << endl;
        return false;
    }
    stack[++top] = value;
    return true;
}
```

2. 空栈检查

```
int stack_top() {
    if (top == -1) {  // 检查栈空
        cout << "Stack is empty!" << endl;
        return -1;
    }
    return stack[top];
}
```

面向对象设计的优势

1. 封装性

```
class Stack {
private:
    int* data;          // 内部实现细节被隐藏
    int topIndex;

public:
    void push(int value);  // 对外提供简洁接口
    int pop();
};
```

2. 可维护性

- 修改内部实现不影响外部使用
- 代码结构清晰，易于理解和调试

3. 可复用性

```
// 可以在其他项目中直接使用Stack类
Stack s1(100);  // 创建容量为100的栈
Stack s2;       // 使用默认容量
```

错误处理与异常安全

1. 边界检查

```
bool push(int value) {
    if(full()) {
        // 返回false而不是让程序崩溃
        return false;
    }
    data[++topIndex] = value;
    return true;
}
```

2. 异常处理

```
int calculate(int a, int b, char op) {
    switch(op) {
        case '/':
            if(b == 0) {
                throw runtime_error("除零错误");
            }
            return a / b;
        // ... 其他操作
    }
}
```

性能分析与比较

时间复杂度比较

操作	栈	队列	循环队列
插入	$O(1)$	$O(1)$	$O(1)$
删除	$O(1)$	$O(1)$	$O(1)$
查找	$O(n)$	$O(n)$	$O(n)$

空间复杂度

- 所有实现：$O(n)$，其中 n 为容量

实战建议

1. 类设计原则

- **单一职责**：每个类只负责一个功能
- **开放封闭**：对扩展开放，对修改封闭
- **依赖倒置**：依赖抽象而不是具体实现

2. 数据结构选择

- **栈**：函数调用、表达式求值、撤销操作
- **队列**：任务调度、消息传递、BFS算法
- **循环队列**：数据流处理、环形缓冲区

3. 代码规范

```
// 好的命名规范
class Stack {
private:
    int* data_;          // 成员变量加后缀_
    int topIndex_;

public:
    void push(int value);  // 动词命名函数
    bool empty() const;    // 布尔函数用is/has开头
};
```

总结

面向对象实现的优势

1. **封装性**：隐藏实现细节，提供清晰接口
2. **可维护性**：代码结构清晰，易于修改和扩展
3. **可复用性**：类可以在多个项目中重复使用
4. **安全性**：通过访问控制保护数据完整性

适用场景

- 需要长期维护的大型项目
- 需要代码复用的场景
- 团队协作开发
- 对代码质量要求高的场景

奇思妙学信息学竞赛

GESP C++ 六级认证（四）

数据结构与面向对象专题

类与对象 · 栈 · 队列 · 循环队列