До всіх питань потрібно записати у відповіді структури даних і/або фрагменти програмної реалізації, і на
основі таких структур і фрагментів записати змістові пояснення]

Список питань до розділу "Асемблери"

1. Схема трансляції, компанування і виконання програми.

Схема трансляції, компанування і виконання:

1. Написання програми на асемблері (.asm).
2. Трансляція (асемблювання) в об'єктний файл (.obj або .o).
3. Компанування (лінкування) об'єктних файлів у виконуваний файл (.exe або .out).
4. Виконання програми операційною системою.

Добре, ось приклад повного процесу трансляції, компанування і виконання програми на асемблері для платформи x86-64. Ми напишемо просту програму, яка виводить "Hello, World!".

Крок 1: Написання програми на асемблері

```assembly
; Файл: hello.asm
section .data
    msg db 'Hello, World!', 0    ; Оголошення рядка даних

section .text
    global _start               ; Точка входу
_start:
    mov eax, 4                  ; Системний виклик для запису
    mov ebx, 1                  ; Дескриптор файлу (stdout)
    mov ecx, msg                ; Адреса повідомлення
    mov edx, 13                 ; Довжина повідомлення
    int 0x80                    ; Виклик системної функції

    mov eax, 1                  ; Системний виклик для виходу
    xor ebx, ebx                ; Код виходу 0
    int 0x80                    ; Виклик системної функції
```

Крок 2: Трансляція (Асемблювання)

За допомогою NASM (Netwide Assembler) можна асемблювати програму у об'єктний файл.

```sh
nasm -f elf64 hello.asm -o hello.o
```

Крок 3: Компанування (Лінкування)

За допомогою GCC (GNU Compiler Collection) можна злідувати об'єктний файл у виконуваний файл.

```sh
gcc hello.o -o hello
```

Крок 4: Виконання програми

Запустіть виконуваний файл.

```sh
./hello
```

Повний код і команди

1. Створіть файл `hello.asm` з наступним кодом:

```assembly
section .data
    msg db 'Hello, World!', 0    ; Оголошення рядка даних

section .text
    global _start               ; Точка входу
_start:
    mov eax, 4                  ; Системний виклик для запису
    mov ebx, 1                  ; Дескриптор файлу (stdout)
    mov ecx, msg                ; Адреса повідомлення
    mov edx, 13                 ; Довжина повідомлення
    int 0x80                    ; Виклик системної функції

    mov eax, 1                  ; Системний виклик для виходу
    xor ebx, ebx                ; Код виходу 0
    int 0x80                    ; Виклик системної функції
```

2. Виконайте асемблювання:

```sh
nasm -f elf64 hello.asm -o hello.o
```

3. Виконайте лінкування:

```sh
gcc hello.o -o hello
```

4. Запустіть програму:

```sh
./hello
```

Пояснення коду

- **Секція .data:** Містить дані, які будуть використані програмою, в даному випадку рядок "Hello, World!".
- **Секція .text:** Містить інструкції програми.
- **global _start:** Вказує на точку входу програми.
- **_start:** Початок програми.
- **mov eax, 4; mov ebx, 1; mov ecx, msg; mov edx, 13; int 0x80:** Виконує системний виклик для запису даних на stdout.
- **mov eax, 1; xor ebx, ebx; int 0x80:** Виконує системний виклик для завершення програми з кодом виходу 0.

2. Основні регістри мікропроцесора.

Основні регістри мікропроцесора

Мікропроцесор x86-64 має кілька категорій регістрів, кожен з яких має своє специфічне призначення. Розглянемо основні регістри мікропроцесора і їх функції.

1. Загальні регістри

Ці регістри використовуються для зберігання операндів і результатів арифметичних та логічних операцій.

- **RAX (Accumulator Register):** Використовується для операцій введення/виведення, арифметичних операцій.
- **RBX (Base Register):** Базовий регістр, часто використовується для зберігання базових адрес.
- **RCX (Count Register):** Використовується як лічильник для циклів і операцій зсуву.

- **RDX (Data Register):** Використовується для операцій введення/виведення, арифметичних операцій.
- **RSI (Source Index):** Використовується для адресації джерела в операціях зі строками.
- **RDI (Destination Index):** Використовується для адресації призначення в операціях зі строками.
- **RBP (Base Pointer):** Використовується для зберігання базової адреси стека.
- **RSP (Stack Pointer):** Вказує на вершину стека.

2. Розширені регістри

У режимі x86-64 додатково є регістри R8 - R15, які розширюють можливості загальних регістрів.

- **R8 - R15:** Додаткові загальні регістри.

3. Спеціальні регістри

Ці регістри мають специфічні функції, пов'язані з керуванням і виконанням програми.

- **RIP (Instruction Pointer):** Вказує на адресу наступної інструкції, яка буде виконана.
- **RFLAGS (Flags Register):** Містить різні прапорці стану, які відображають результати операцій і стан процесора.

4. Сегментні регістри

Використовуються для сегментної адресації пам'яті.

- **CS (Code Segment):** Вказує на сегмент коду.
- **DS (Data Segment):** Вказує на сегмент даних.
- **SS (Stack Segment):** Вказує на сегмент стека.
- **ES, FS, GS:** Додаткові сегментні регістри для різних операцій.

Фрагмент коду на асемблері, що демонструє використання основних регістрів

```assembly
section .data
    msg db 'Hello, World!', 0    ; Оголошення рядка даних

section .bss
    num resb 1                   ; Оголошення змінної

section .text
    global _start                ; Точка входу

_start:
    ; Використання загальних регістрів
    mov rax, 1                   ; Системний виклик для запису
```

```
    mov rdi, 1                ; Дескриптор файлу (stdout)
    mov rsi, msg              ; Адреса повідомлення
    mov rdx, 13               ; Довжина повідомлення
    syscall                   ; Виклик системної функції

    ; Використання спеціальних регістрів
    mov rbx, rsp              ; Збереження значення регістру RSP в RBX

    ; Завершення програми
    mov rax, 60               ; Системний виклик для виходу
    xor rdi, rdi              ; Код виходу 0
    syscall                   ; Виклик системної функції
```

Пояснення коду

- **mov rax, 1:** Завантажує значення 1 у регістр RAX. У контексті системного виклику це означає виклик функції для запису.
- **mov rdi, 1:** Завантажує значення 1 у регістр RDI, що представляє дескриптор файлу stdout.
- **mov rsi, msg:** Завантажує адресу повідомлення в регістр RSI.
- **mov rdx, 13:** Завантажує довжину повідомлення в регістр RDX.
- **syscall:** Виконує системний виклик.
- **mov rbx, rsp:** Копіює значення стека (RSP) у регістр RBX для збереження.
- **mov rax, 60:** Завантажує значення 60 у регістр RAX. У контексті системного виклику це означає виклик функції для завершення програми.
- **xor rdi, rdi:** Обнуляє регістр RDI, встановлюючи код виходу в 0.
- **syscall:** Виконує системний виклик для завершення програми.

3. Позиційна незалежність програми в однопрограмних ОС.

Позиційна незалежність програми в однопрограмних ОС

Позиційно-незалежний код (Position-Independent Code, PIC) — це тип машинного коду, який може виконуватися з будь-якого адресного простору в пам'яті. Це особливо важливо для динамічно завантажуваних бібліотек (shared libraries) та в системах з підтримкою ASLR (Address Space Layout Randomization).

Основні поняття та переваги PIC

1. **Гнучкість розміщення:** Позиційно-незалежний код може виконуватися з будь-якої адреси в пам'яті без потреби у зміні самого коду.
2. **Безпека:** Використання ASLR робить важчим для атакуючих передбачити розміщення коду в пам'яті, що ускладнює експлуатацію вразливостей.
3. **Ефективне використання пам'яті:** Одна і та сама бібліотека може використовуватися одночасно декількома програмами, що зменшує використання пам'яті.

Як працює позиційно-незалежний код

У PIC, замість використання фіксованих адрес, програма використовує відносні адреси або таблиці переадресації (relocation tables) для доступу до даних і коду. В основному це досягається шляхом використання регістра базової адреси (base register) для адресації змінних і функцій.

Фрагмент коду, що демонструє позиційну незалежність

```assembly
section .data
    msg db 'Hello, World!', 0

section .text
    global _start
_start:
    call get_eip             ; Виклик функції для отримання поточної адреси
get_eip:
    pop rbx                  ; Отримання адреси метки get_eip
    add rbx, msg - get_eip    ; Обчислення адреси msg відносно get_eip

    mov rax, 1               ; Системний виклик для запису
    mov rdi, 1               ; Дескриптор файлу (stdout)
    mov rsi, rbx             ; Встановлення адреси повідомлення
    mov rdx, 13              ; Довжина повідомлення
    syscall                  ; Виклик системної функції

    mov rax, 60              ; Системний виклик для виходу
    xor rdi, rdi             ; Код виходу 0
    syscall                  ; Виклик системної функції
```

Пояснення коду

1. **call get_eip:** Викликає функцію `get_eip`, яка записує адресу наступної інструкції у стек.
2. **pop rbx:** Витягує адресу з стека в регістр `rbx`. Це забезпечує отримання поточної адреси виконання (EIP/RIP).
3. **add rbx, msg - get_eip:** Обчислює адреса `msg` відносно поточної адреси виконання, використовуючи відносне зміщення. Це забезпечує позиційну незалежність.
4. **mov rax, 1** до **syscall:** Виконує системний виклик для виводу повідомлення на екран.
5. **mov rax, 60** до **syscall:** Виконує системний виклик для завершення програми.

Переваги та застосування

- **Безпека:** Зменшує ризики, пов'язані з фіксованими адресами в пам'яті.

- **Реусабельність коду:** Одна і та сама бібліотека може використовуватися декількома програмами без потреби у перекомпіляції.
- **Ефективність:** Зменшує загальне використання пам'яті.

Висновок

Позиційно-незалежний код є критично важливим для сучасних програмних систем, особливо в умовах однопрограмних операційних систем і середовищ з динамічним завантаженням бібліотек. Це забезпечує гнучкість, безпеку та ефективне використання ресурсів.

4. Режими адресування операндів в однопрограмних ОС.

Режими адресування операндів в однопрограмних ОС

Адресування операндів — це спосіб, яким процесор визначає місцезнаходження операндів (даних), що використовуються в інструкціях. У однопрограмних операційних системах часто використовуються такі основні режими адресування:

1. **Безпосереднє адресування (Immediate Addressing):**
   - Операнд є частиною самої інструкції.
   - Використовується для завдання констант.
   - Приклад: `mov eax, 5` (число 5 є операндом).

2. **Пряме адресування (Direct Addressing):**
   - Операнд знаходиться за фіксованою адресою в пам'яті.
   - Використовується для доступу до глобальних змінних.
   - Приклад: `mov eax, [var]` (значення з адреси `var` завантажується в регістр `eax`).

3. **Регістрове адресування (Register Addressing):**
   - Операнд знаходиться в регістрі процесора.
   - Використовується для швидкого доступу до даних.
   - Приклад: `mov eax, ebx` (значення з регістра `ebx` копіюється в регістр `eax`).

4. **Непряме адресування (Indirect Addressing):**
   - Адреса операнда знаходиться в регістрі або пам'яті.
   - Використовується для роботи з вказівниками та масивами.
   - Приклад: `mov eax, [ebx]` (значення за адресою, яка зберігається в регістрі `ebx`, завантажується в регістр `eax`).

5. **Індексне адресування (Indexed Addressing):**
   - Адреса операнда визначається базовою адресою плюс зміщення.

- Використовується для роботи з масивами та структурами даних.
- Приклад: `mov eax, [ebx + esi]` (значення за адресою `ebx + esi` завантажується в регістр `eax`).

6. **Базово-індексне адресування (Base-Index Addressing):**
- Адреса операнда визначається базовою адресою плюс індекс плюс зміщення.
- Використовується для складних структур даних.
- Приклад: `mov eax, [ebx + esi + 4]` (значення за адресою `ebx + esi + 4` завантажується в регістр `eax`).

7. **Відносне адресування (Relative Addressing):**
- Адреса операнда визначається відносно поточної адреси виконання (RIP).
- Використовується для переходів та викликів функцій.
- Приклад: `jmp label` (перехід до мітки `label`, яка визначена відносно поточної адреси).

Приклади коду, що демонструють режими адресування

```assembly
section .data
    var db 10               ; Пряма адресація

section .text
    global _start
_start:
    ; Безпосереднє адресування
    mov eax, 5              ; Завантаження константи в регістр eax

    ; Пряме адресування
    mov ebx, [var]          ; Завантаження значення з пам'яті в регістр ebx

    ; Регістрове адресування
    mov ecx, eax            ; Копіювання значення з регістра eax в регістр ecx

    ; Непряме адресування
    lea edx, [var]          ; Завантаження адреси var в регістр edx
    mov al, [edx]           ; Завантаження значення з адреси в регістр al

    ; Індексне адресування
    lea esi, [var]          ; Завантаження адреси var в регістр esi
    mov edi, 2              ; Завантаження індексу
    mov al, [esi + edi]     ; Завантаження значення з адреси var + 2 в регістр al

    ; Базово-індексне адресування
    lea ebx, [var]          ; Завантаження адреси var в регістр ebx
    mov esi, 1              ; Завантаження індексу
    mov eax, [ebx + esi*4]  ; Завантаження значення з адреси var + 4*1 в регістр eax
```

```
    ; Відносне адресування
    call next               ; Виклик функції з відносною адресацією
next:
    ; Продовження виконання після виклику
    mov eax, 60             ; Системний виклик для виходу
    xor edi, edi            ; Код виходу 0
    syscall                 ; Виклик системної функції
```

Пояснення коду

1. **Безпосереднє адресування:** Значення `5` завантажується в регістр `eax`.
2. **Пряме адресування:** Значення змінної `var` завантажується в регістр `ebx`.
3. **Регістрове адресування:** Значення з регістра `eax` копіюється в регістр `ecx`.
4. **Непряме адресування:** Адреса змінної `var` завантажується в `edx`, а потім значення за цією адресою — в регістр `al`.
5. **Індексне адресування:** Значення за адресою `var + 2` завантажується в регістр `al`.
6. **Базово-індексне адресування:** Значення за адресою `var + 4*1` завантажується в регістр `eax`.
7. **Відносне адресування:** Виклик функції `next` здійснюється за допомогою відносної адресації.

Ці режими адресування дозволяють ефективно маніпулювати даними в пам'яті, забезпечуючи гнучкість і потужність мовам асемблера.

5. Моделі адресування операндів в командах мікропроцесора.

Моделі адресування операндів в командах мікропроцесора

Адресування операндів — це спосіб, яким процесор визначає місцезнаходження даних, необхідних для виконання інструкцій. Існує кілька моделей адресування, які визначають, як саме операнди вказуються в інструкціях. Розглянемо основні моделі адресування операндів у мікропроцесорних командах:

1. **Безпосереднє адресування (Immediate Addressing):**
   - Операнд є частиною самої інструкції.
   - Застосовується для завдання констант або фіксованих значень.
   - Приклад: `mov eax, 5` (число 5 є операндом).

2. **Пряме адресування (Direct Addressing):**
   - Операнд знаходиться за вказаною адресою в пам'яті.
   - Застосовується для доступу до глобальних змінних.
   - Приклад: `mov eax, [var]` (значення з адреси `var` завантажується в регістр `eax`).

3. **Регістрове адресування (Register Addressing):**
   - Операнд знаходиться в одному з регістрів процесора.
   - Застосовується для швидкого доступу до даних, що часто використовуються.
   - Приклад: `mov eax, ebx` (значення з регістра `ebx` копіюється в регістр `eax`).

4. **Непряме адресування (Indirect Addressing):**
   - Адреса операнда знаходиться в регістрі або пам'яті.
   - Використовується для роботи з вказівниками та динамічною пам'яттю.
   - Приклад: `mov eax, [ebx]` (значення за адресою, яка зберігається в регістрі `ebx`, завантажується в регістр `eax`).

5. **Індексне адресування (Indexed Addressing):**
   - Адреса операнда визначається базовою адресою плюс індекс.
   - Використовується для роботи з масивами та таблицями.
   - Приклад: `mov eax, [ebx + esi]` (значення за адресою `ebx + esi` завантажується в регістр `eax`).

6. **Базово-індексне адресування (Base-Index Addressing):**
   - Адреса операнда визначається базовою адресою плюс індекс плюс зміщення.
   - Використовується для складних структур даних та об'єктів.
   - Приклад: `mov eax, [ebx + esi + 4]` (значення за адресою `ebx + esi + 4` завантажується в регістр `eax`).

7. **Відносне адресування (Relative Addressing):**
   - Адреса операнда визначається відносно поточної адреси виконання (RIP).
   - Використовується для переходів та викликів функцій.
   - Приклад: `jmp label` (перехід до мітки `label`, яка визначена відносно поточної адреси).

8. **Базове адресування з попереднім інкрементом/декрементом (Pre-Increment/Decrement Addressing):**
   - Адреса операнда обчислюється з базової адреси перед використанням.
   - Використовується для роботи зі стеком або буферами.
   - Приклад: `mov eax, [esi + 4]` (значення за адресою `esi + 4` завантажується в регістр `eax`).

9. **Базове адресування з постінкрементом/декрементом (Post-Increment/Decrement Addressing):**
   - Адреса операнда обчислюється з базової адреси після використання.
   - Використовується для роботи зі стеком або буферами.
   - Приклад: `mov eax, [esi]` (значення за адресою `esi` завантажується в регістр `eax`, а потім `esi` збільшується на 4).

Приклади коду, що демонструють моделі адресування

```assembly
section .data
    var db 10               ; Пряма адресація
```

```
section .text
    global _start
_start:
    ; Безпосереднє адресування
    mov eax, 5              ; Завантаження константи в регістр eax

    ; Пряме адресування
    mov ebx, [var]          ; Завантаження значення з пам'яті в регістр ebx

    ; Регістрове адресування
    mov ecx, eax            ; Копіювання значення з регістра eax в регістр ecx

    ; Непряме адресування
    lea edx, [var]          ; Завантаження адреси var в регістр edx
    mov al, [edx]           ; Завантаження значення з адреси в регістр al

    ; Індексне адресування
    lea esi, [var]          ; Завантаження адреси var в регістр esi
    mov edi, 2              ; Завантаження індексу
    mov al, [esi + edi]     ; Завантаження значення з адреси var + 2 в регістр al

    ; Базово-індексне адресування
    lea ebx, [var]          ; Завантаження адреси var в регістр ebx
    mov esi, 1              ; Завантаження індексу
    mov eax, [ebx + esi*4]  ; Завантаження значення з адреси var + 4*1 в регістр eax

    ; Відносне адресування
    call next               ; Виклик функції з відносною адресацією
next:
    ; Продовження виконання після виклику
    mov eax, 60             ; Системний виклик для виходу
    xor edi, edi            ; Код виходу 0
    syscall                 ; Виклик системної функції
```

Пояснення коду

1. **Безпосереднє адресування:** Значення `5` завантажується в регістр `eax`.
2. **Пряме адресування:** Значення змінної `var` завантажується в регістр `ebx`.
3. **Регістрове адресування:** Значення з регістра `eax` копіюється в регістр `ecx`.
4. **Непряме адресування:** Адреса змінної `var` завантажується в `edx`, а потім значення за цією адресою — в регістр `al`.
5. **Індексне адресування:** Значення за адресою `var + 2` завантажується в регістр `al`.
6. **Базово-індексне адресування:** Значення за адресою `var + 4*1` завантажується в регістр `eax`.

7. **Відносне адресування:** Виклик функції `next` здійснюється за допомогою відносної адресації.

Ці моделі адресування забезпечують гнучкість і ефективність при роботі з даними в пам'яті та регістрах, дозволяючи програмам ефективно виконувати складні обчислення та маніпуляції з даними.

6. Моделі структури програм: головна програма, підпрограма, COM-програма (команди + дані).

Моделі структури програм

У програмуванні існує кілька моделей структури програм, які визначають організацію коду та даних. Основні з них — це головна програма, підпрограма та COM-програма (команди + дані). Розглянемо кожну з цих моделей детальніше.

1. Головна програма (Main Program)

Головна програма — це центральна частина програми, яка координує виконання всієї програми. Вона може викликати різні підпрограми для виконання конкретних задач.

Основні особливості:
- Містить початкову точку входу (`_start` або `main`).
- Виконує ініціалізацію необхідних ресурсів.
- Викликає підпрограми для виконання окремих задач.

Приклад коду на асемблері:
```assembly
section .data
    msg db 'Hello, World!', 0

section .text
    global _start

_start:
    ; Виклик підпрограми для виведення повідомлення
    call print_message

    ; Завершення програми
    mov eax, 60         ; Системний виклик для завершення програми
    xor edi, edi        ; Код виходу 0
    syscall
```

```
print_message:
    mov eax, 1          ; Системний виклик для запису
    mov edi, 1          ; Дескриптор файлу (stdout)
    mov rsi, msg        ; Адреса повідомлення
    mov edx, 13         ; Довжина повідомлення
    syscall
    ret
```

2. Підпрограма (Subroutine)

Підпрограма — це самостійний блок коду, який виконує певну функцію і може бути викликаний з головної програми або іншої підпрограми. Вона дозволяє уникнути повторення коду та покращує структуру програми.

Основні особливості:
- Може мати параметри для прийому даних.
- Може повертати значення.
- Викликається за допомогою інструкцій `call` і завершується інструкцією `ret`.

Приклад коду на асемблері:
```assembly
section .data
    num1 dq 10
    num2 dq 20

section .text
    global _start

_start:
    ; Виклик підпрограми для додавання чисел
    mov rdi, [num1]    ; Перший аргумент
    mov rsi, [num2]    ; Другий аргумент
    call add_numbers
    ; Результат буде в регістрі rax

    ; Завершення програми
    mov eax, 60         ; Системний виклик для завершення програми
    xor edi, edi        ; Код виходу 0
    syscall

add_numbers:
    ; Додавання чисел
    add rdi, rsi
    mov rax, rdi
    ret
```

3. COM-програма (Команди + дані)

COM-програма — це модель програми, де команди та дані зберігаються в одному сегменті пам'яті. Це характерно для DOS-програм у форматі .COM. Такі програми обмежені розміром у 64 КБ.

Основні особливості:
- Програма завантажується в пам'ять за фіксованою адресою.
- Команди і дані розміщуються в одному сегменті.
- Не використовуються сегментні регістри для доступу до коду і даних.

Приклад коду на асемблері (DOS .COM програма):
```assembly
org 100h                ; Початкова адреса для COM-програм

section .text
start:
    mov ah, 09h         ; DOS функція для виведення рядка
    lea dx, msg         ; Адреса рядка
    int 21h             ; Виклик DOS функції

    mov ah, 4Ch         ; DOS функція для завершення програми
    int 21h             ; Виклик DOS функції

section .data
msg db 'Hello, COM Program!', '$'
```

Пояснення коду

- **Головна програма:** Виконує основну логіку, викликає підпрограми для виконання конкретних задач (наприклад, виведення повідомлення або додавання чисел).
- **Підпрограма:** Самостійний блок коду, який виконує конкретну функцію (наприклад, додавання двох чисел). Викликається з головної програми або іншої підпрограми.
- **COM-програма:** Програма у форматі .COM для DOS, яка зберігає команди та дані в одному сегменті пам'яті. Використовується для невеликих програм, обмежених розміром 64 КБ.

Ці моделі структури програм допомагають організувати код, полегшуючи його розуміння, підтримку та повторне використання.

7. Загальний формат бітової структури команди процесора Intel. Поля коду команди.

Загальний формат бітової структури команди процесора Intel. Поля коду команди

Команди процесора Intel мають складну бітову структуру, яка включає кілька полів. Формат команди може варіюватися залежно від конкретної інструкції та її операндів. Основні поля коду команди включають:

1. **Префікси (Prefixes)**
2. **Байти операційного коду (Opcode Bytes)**
3. **Модифікаторний байт (ModR/M Byte)**
4. **Розширення модифікаторного байта (SIB Byte)**
5. **Зміщення (Displacement)**
6. **Безпосереднє значення (Immediate Value)**

1. Префікси (Prefixes)
Префікси — це додаткові байти, які можуть передувати основній команді і змінювати її поведінку. Префікси можуть бути:

- Префікси сегментних регістрів (наприклад, `CS`, `DS`, `ES`, `FS`, `GS`, `SS`)
- Префікси заміни розміру операнда (`66h`, `67h`)
- Префікси заміни адресного простору (`F0h`, `F2h`, `F3h`)

2. Байти операційного коду (Opcode Bytes)
Це основні байти, які визначають інструкцію, що виконується. Операційний код може бути від одного до трьох байтів.

3. Модифікаторний байт (ModR/M Byte)
Цей байт визначає режими адресування операндів. Модифікаторний байт складається з трьох полів:

- **Mod:** Визначає режим адресування (2 біти)
- **Reg/Opcode:** Визначає регістр або додатковий код операції (3 біти)
- **R/M:** Визначає регістр або метод адресування (3 біти)

4. Розширення модифікаторного байта (SIB Byte)
Використовується разом з ModR/M байтом для індексованої адресації. CIB байт складається з трьох полів:

- **Scale:** Визначає коефіцієнт масштабування (2 біти)
- **Index:** Визначає індексний регістр (3 біти)
- **Base:** Визначає базовий регістр (3 біти)

5. Зміщення (Displacement)
Це додаткове значення, яке додається до базової адреси. Зміщення може бути 8, 16 або 32 біти.

6. Безпосереднє значення (Immediate Value)
Це константне значення, яке є частиною інструкції. Може бути від 8 до 32 бітів.

Приклад структури команди

Розглянемо команду `MOV eax, [ebx+4]`.

- **Префікси:** В даному випадку не використовуються.
- **Операційний код (Opcode):** `8B` (завантаження даних з пам'яті в регістр)
- **ModR/M байт:** `43 04`
  - **Mod:** `01` (8-бітове зміщення)
  - **Reg/Opcode:** `000` (регістровий код для `eax`)
  - **R/M:** `011` (регістровий код для `ebx`)
- **Зміщення:** `04` (8-бітове зміщення)

Фрагмент коду на асемблері з поясненням

```assembly
section .data
    value db 0xFF

section .text
    global _start

_start:
    ; Приклад команди MOV з безпосереднім значенням
    mov eax, 0x12345678       ; Opcode: B8, Immediate: 0x12345678

    ; Приклад команди MOV з використанням ModR/M байта
    mov eax, [ebx + 4]        ; Opcode: 8B, ModR/M: 43 04

    ; Завершення програми
    mov eax, 60               ; Системний виклик для завершення програми
    xor edi, edi              ; Код виходу 0
    syscall
```

Пояснення структури команди

1. **MOV eax, 0x12345678**
   - **Opcode:** `B8` (MOV з безпосереднім значенням в `eax`)
   - **Immediate Value:** `0x12345678` (значення, яке завантажується в `eax`)

2. **MOV eax, [ebx + 4]**
   - **Opcode:** `8B` (MOV з пам'яті в регістр)
   - **ModR/M:** `43 04`
     - **Mod:** `01` (8-бітове зміщення)
     - **Reg:** `000` (код регістра для `eax`)
     - **R/M:** `011` (код регістра для `ebx`)
   - **Displacement:** `04` (8-бітове зміщення)

Висновок

Структура команди процесора Intel включає кілька полів, які визначають тип інструкції, режими адресування і операнди. Розуміння цих полів важливо для ефективного програмування на асемблері та аналізу машинного коду.

8. Будова байта ModRM способу адресування для 16- і 32-бітових режимів.

Будова байта ModRM способу адресування для 16- і 32-бітових режимів

Байт ModRM використовується в процесорах Intel для визначення способу адресування операндів у командах. Цей байт складається з трьох полів: `Mod`, `Reg/Opcode` і `R/M`, які разом визначають, як слід інтерпретувати операнди команди.

Структура байта ModRM

- **Mod (2 біти):** Визначає режим адресації.
- **Reg/Opcode (3 біти):** Визначає регістр або додатковий код операції.
- **R/M (3 біти):** Визначає регістр або метод адресування.

ModRM байт у 16-бітовому режимі

У 16-бітовому режимі поля `Mod`, `Reg/Opcode` і `R/M` мають наступну структуру:

- **Mod:**
  - 00: Пряма адресація або непряме адресування без зміщення.
  - 01: Непряме адресування зі зміщенням 8 біт.
  - 10: Непряме адресування зі зміщенням 16 біт.
  - 11: Регістрове адресування.

- **R/M:**
  - 000: [BX + SI]
  - 001: [BX + DI]
  - 010: [BP + SI]
  - 011: [BP + DI]
  - 100: [SI]
  - 101: [DI]
  - 110: [BP] (або пряма адресація, якщо Mod = 00)
  - 111: [BX]

Приклади ModRM байта у 16-бітовому режимі

1. **Mod = 00, R/M = 110:** Пряма адресація.
2. **Mod = 01, R/M = 010:** Непряме адресування з 8-бітовим зміщенням (наприклад, [BP + SI] + 8-бітове зміщення).
3. **Mod = 11, R/M = 000:** Регістрове адресування (наприклад, AX).

ModRM байт у 32-бітовому режимі

У 32-бітовому режимі поля `Mod`, `Reg/Opcode` і `R/M` мають наступну структуру:

- **Mod:**
  - 00: Пряма адресація або непряме адресування без зміщення.
  - 01: Непряме адресування зі зміщенням 8 біт.
  - 02: Непряме адресування зі зміщенням 32 біт.
  - 11: Регістрове адресування.

- **R/M:**
  - 000: [EAX]
  - 001: [ECX]
  - 002: [EDX]
  - 003: [EBX]
  - 004: SIB (Scale-Index-Base) байт
  - 005: [EBP] (або пряма адресація, якщо Mod = 00)
  - 006: [ESI]
  - 007: [EDI]

Структура SIB байта (32-бітовий режим)

Якщо `R/M` у ModRM байті дорівнює 100, то використовується SIB байт. SIB байт має наступну структуру:

- **Scale (2 біти):** Коефіцієнт масштабування (00 = 1, 01 = 2, 10 = 4, 11 = 8).
- **Index (3 біти):** Індексний регістр.
  - 000: EAX
  - 001: ECX
  - 010: EDX
  - 011: EBX
  - 100: No index (scale ignored)
  - 101: EBP (або пряма адресація, якщо Mod = 00)
  - 110: ESI
  - 111: EDI
- **Base (3 біти):** Базовий регістр.
  - 000: EAX
  - 001: ECX
  - 002: EDX
  - 003: EBX
  - 004: ESP
  - 005: EBP (або пряма адресація, якщо Mod = 00)

  - 006: ESI
  - 007: EDI

Приклади ModRM байта у 32-бітовому режимі

1. **Mod = 00, R/M = 101:** Пряма адресація (32-бітове зміщення слідує за ModRM байтом).
2. **Mod = 01, R/M = 100:** Непряме адресування з 8-бітовим зміщенням, використовуючи SIB байт.
3. **Mod = 11, R/M = 000:** Регістрове адресування (наприклад, EAX).

Приклад коду на асемблері з поясненням ModRM байта

```assembly
section .data
    value db 0xFF

section .text
    global _start

_start:
    ; Приклад використання ModRM байта у 32-бітовому режимі
    ; mov eax, [ebx + 4] де Mod = 01, Reg = 000, R/M = 011
    ; ModRM байт: 43 04 (mod = 01, reg = 000, r/m = 011)
    mov eax, [ebx + 4]

    ; Завершення програми
    mov eax, 60             ; Системний виклик для завершення програми
    xor edi, edi            ; Код виходу 0
    syscall
```

Пояснення коду

1. **mov eax, [ebx + 4]**
   - **Opcode:** `8B` (MOV з пам'яті в регістр)
   - **ModRM:** `43 04`
     - **Mod:** `01` (8-бітове зміщення)
     - **Reg:** `000` (код регістра для `eax`)
     - **R/M:** `011` (код регістра для `ebx`)
   - **Displacement:** `04` (8-бітове зміщення)

Цей приклад демонструє, як байт ModRM визначає спосіб адресування операндів у командах процесора Intel для 16- та 32-бітових режимів.

9. Схеми виконання команд мікропроцесора: команди без операндів (нуль-операндні), команди операцій і команди дії з одним операндом.

Схеми виконання команд мікропроцесора

Команди мікропроцесора можна класифікувати за кількістю операндів і типом виконуваних дій. Розглянемо три основні категорії: команди без операндів (нуль-операндні), команди операцій та команди дії з одним операндом.

1. Команди без операндів (нуль-операндні команди)

Нуль-операндні команди не мають явних операндів. Вони зазвичай використовують неявні операнди, наприклад, в стекових архітектурах.

Приклади нуль-операндних команд:
- **NOP:** Ніякої операції, використовується для затримки або вирівнювання.
- **RET:** Повернення з підпрограми, використовується в стекових архітектурах для повернення адреси з стека.

Приклад коду на асемблері:
```assembly
section .text
    global _start

_start:
    nop             ; Виконання команди NOP
    ; Інші команди

    call my_sub     ; Виклик підпрограми
    nop             ; Виконання команди NOP після повернення з підпрограми

    ; Завершення програми
    mov eax, 60     ; Системний виклик для завершення програми
    xor edi, edi    ; Код виходу 0
    syscall

my_sub:
    ; Тіло підпрограми
    ret             ; Повернення з підпрограми
```

2. Команди операцій (двох- або трьох-операндні команди)

Ці команди виконують операції між двома або трьома операндами. Двох-операндні команди мають два операнди, де результат зберігається в одному з них.
Трьох-операндні команди (в основному у RISC архітектурах) мають три операнди: два джерела і одне призначення.

Приклади двох-операндних команд:
- **ADD:** Додавання.
- **SUB:** Віднімання.

Приклад коду на асемблері:
```assembly
section .data
    num1 dd 10
    num2 dd 20

section .text
    global _start

_start:
    mov eax, [num1]     ; Завантаження першого операнда
    add eax, [num2]     ; Додавання другого операнда

    ; Інші команди

    ; Завершення програми
    mov eax, 60         ; Системний виклик для завершення програми
    xor edi, edi        ; Код виходу 0
    syscall
```

Приклади трьох-операндних команд (у RISC архітектурах):
- **ADD:** Додавання (наприклад, `ADD R0, R1, R2` — R0 = R1 + R2).

3. Команди дії з одним операндом (одно-операндні команди)

Одно-операндні команди виконують операцію над одним операндом, результат якої зберігається у тому ж операнді або в спеціальному регістрі.

Приклади одно-операндних команд:
- **INC:** Інкремент (збільшення на 1).
- **DEC:** Декремент (зменшення на 1).
- **NEG:** Зміна знаку (перетворення числа на протилежне).

Приклад коду на асемблери:
```assembly
section .data
    num1 dd 10
```

```
section .text
    global _start

_start:
    mov eax, [num1]     ; Завантаження операнда
    inc eax             ; Інкрементування операнда

    ; Інші команди

    ; Завершення програми
    mov eax, 60         ; Системний виклик для завершення програми
    xor edi, edi        ; Код виходу 0
    syscall
```

Пояснення коду

1. **NOP:**
   - Виконується команда, яка не впливає на стан процесора чи пам'ять.
   - Використовується для вирівнювання коду або затримки.

2. **RET:**
   - Виконується повернення з підпрограми.
   - Адреса повернення береться зі стека.

3. **ADD:**
   - Виконується додавання двох операндів.
   - Результат зберігається в одному з операндів.

4. **INC:**
   - Інкрементує значення операнда на 1.

Висновок

Ці схеми виконання команд мікропроцесора визначають, як інструкції оперують з даними, надаючи різні способи маніпуляції операндами. Розуміння цих схем є ключовим для ефективного програмування на рівні асемблера і машинного коду.

10. Схеми виконання команд мікропроцесора: команди з двома операндами, команди з трьома операндами.

Схеми виконання команд мікропроцесора: команди з двома операндами, команди з трьома операндами

Команди мікропроцесора з двома і трьома операндами дозволяють виконувати більш складні операції над даними. Розглянемо детальніше схеми виконання цих команд та приклади їх реалізації.

1. Команди з двома операндами (двох-операндні команди)

Команди з двома операндами використовують два операнди для виконання операції, де результат зазвичай зберігається в одному з операндів.

Приклади двох-операндних команд:
- **ADD destination, source:** Додає значення `source` до `destination`.
- **SUB destination, source:** Віднімає значення `source` від `destination`.
- **MOV destination, source:** Копіює значення `source` в `destination`.

Схема виконання:
1. Завантажити операнди з пам'яті або регістрів.
2. Виконати операцію.
3. Зберегти результат в одному з операндів.

Приклад коду на асемблері:
```assembly
section .data
    num1 dd 10
    num2 dd 20

section .text
    global _start

_start:
    ; Завантаження операндів у регістри
    mov eax, [num1]         ; Завантаження num1 в eax
    mov ebx, [num2]         ; Завантаження num2 в ebx

    ; Виконання двох-операндної команди
    add eax, ebx            ; Додавання num2 до num1, результат у eax

    ; Збереження результату назад у пам'ять
    mov [num1], eax         ; Збереження результату в num1

    ; Завершення програми
    mov eax, 60             ; Системний виклик для завершення програми
    xor edi, edi            ; Код виходу 0
    syscall
```

2. Команди з трьома операндами (трьох-операндні команди)

Команди з трьома операндами використовують два операнди для виконання операції і третій операнд для збереження результату. Ці команди більш типові для RISC-архітектур, таких як MIPS.

Приклади трьох-операндних команд:
- **ADD destination, source1, source2:** Додає значення `source1` і `source2`, зберігає результат у `destination`.
- **SUB destination, source1, source2:** Віднімає значення `source2` від `source1`, зберігає результат у `destination`.

Схема виконання:
1. Завантажити операнди з пам'яті або регістрів.
2. Виконати операцію.
3. Зберегти результат в окремий регістр або пам'ять.

Приклад коду на асемблері для архітектури MIPS:
```assembly
.data
    num1: .word 10
    num2: .word 20
    result: .word 0

.text
    .globl main

main:
    # Завантаження операндів у регістри
    lw $t0, num1          # Завантаження num1 у $t0
    lw $t1, num2          # Завантаження num2 у $t1

    # Виконання трьох-операндної команди
    add $t2, $t0, $t1     # Додавання num1 і num2, результат у $t2

    # Збереження результату назад у пам'ять
    sw $t2, result        # Збереження результату в result

    # Завершення програми
    li $v0, 10            # Системний виклик для завершення програми
    syscall
```

Пояснення коду

1. **ADD destination, source:** (двох-операндна команда)
   - Завантажує значення операндів з пам'яті або регістрів.
   - Виконує операцію додавання.
   - Зберігає результат в одному з операндів.

2. **ADD destination, source1, source2:** (трьох-операндна команда)
   - Завантажує значення операндів з пам'яті або регістрів.
   - Виконує операцію додавання.
   - Зберігає результат в окремому регістрі або пам'яті.

Висновок

Команди з двома і трьома операндами надають гнучкість для виконання складних обчислень. Двох-операндні команди є більш універсальними і використовуються в багатьох CISC-архітектурах, таких як x86. Трьох-операндні команди характерні для RISC-архітектур і забезпечують ефективне виконання операцій, оскільки не вимагають перезапису вихідних операндів. Розуміння цих схем допомагає краще організовувати код і оптимізувати виконання програм.

11. Директиви асемблера та їх застосування.

Зрозуміло, я буду включати структури даних і фрагменти програмної реалізації до кожної відповіді, а також надавати змістові пояснення на основі цих структур і фрагментів. Ось так виглядатиме відповідь на останнє питання з додатковими структурами даних і фрагментами коду:

11. Директиви асемблера та їх застосування

Директиви асемблера — це інструкції, які надають асемблеру інформацію про те, як слід обробляти код, але не генерують машинний код. Вони використовуються для оголошення даних, визначення сегментів, умовної компіляції та інших цілей. Розглянемо основні директиви асемблера та їх застосування.

Основні директиви асемблера

1. **Директиви сегментації**
2. **Директиви оголошення даних**
3. **Директиви умовної компіляції**
4. **Директиви макросів**
5. **Директиви визначення точок входу та глобальних символів**

1. Директиви сегментації

Ці директиви використовуються для визначення різних сегментів коду та даних у програмі.

- **`.data`:** Визначає сегмент даних, де оголошуються змінні.
- **`.bss`:** Визначає сегмент неініціалізованих даних.
- **`.text`:** Визначає сегмент коду, де розміщуються інструкції програми.

Приклад:
```assembly
section .data
    message db 'Hello, World!', 0

section .bss
    buffer resb 64

section .text
    global _start

_start:
    ; Код програми
    mov eax, 4          ; Системний виклик write
    mov ebx, 1          ; Дескриптор файлу stdout
    mov ecx, message    ; Адреса повідомлення
    mov edx, 13         ; Довжина повідомлення
    int 0x80            ; Виклик системної функції

    mov eax, 1          ; Системний виклик exit
    xor ebx, ebx        ; Код виходу 0
    int 0x80            ; Виклик системної функції
```

Пояснення:
- **`section .data`:** Визначає сегмент даних, де оголошується змінна `message`, що містить рядок "Hello, World!".
- **`section .bss`:** Визначає сегмент неініціалізованих даних, де оголошується буфер `buffer` розміром 64 байти.
- **`section .text`:** Визначає сегмент коду, де розміщуються інструкції програми.

2. Директиви оголошення даних

Ці директиви використовуються для оголошення змінних та констант у програмі.

- **`db` (define byte):** Оголошує байт даних.
- **`dw` (define word):** Оголошує слово (2 байти).
- **`dd` (define double word):** Оголошує подвійне слово (4 байти).
- **`dq` (define quad word):** Оголошує чотирьохбайтове слово (8 байтів).

Приклад:
```assembly
section .data
    byte_var db 0xFF            ; Оголошення байта
    word_var dw 0xFFFF          ; Оголошення слова
    dword_var dd 0xFFFFFFFF     ; Оголошення подвійного слова
    qword_var dq 0xFFFFFFFFFFFFFFFF ; Оголошення чотирьохбайтового слова
```

```

**Пояснення:**
- **`byte_var db 0xFF`:** Оголошення змінної `byte_var` розміром 1 байт з початковим значенням `0xFF`.
- **`word_var dw 0xFFFF`:** Оголошення змінної `word_var` розміром 2 байти з початковим значенням `0xFFFF`.
- **`dword_var dd 0xFFFFFFFF`:** Оголошення змінної `dword_var` розміром 4 байти з початковим значенням `0xFFFFFFFF`.
- **`qword_var dq 0xFFFFFFFFFFFFFFFF`:** Оголошення змінної `qword_var` розміром 8 байтів з початковим значенням `0xFFFFFFFFFFFFFFFF`.

#### 3. Директиви умовної компіляції

Ці директиви дозволяють включати або виключати частини коду в залежності від певних умов.

- **`%ifdef` / `%ifndef`:** Включає код, якщо визначено / не визначено макрос.
- **`%else`:** Виконує альтернативний блок коду.
- **`%endif`:** Завершує умовний блок коду.

**Приклад:**
```assembly
%define DEBUG

section .text
    global _start

_start:
%ifdef DEBUG
    ; Код для відладки
    mov eax, 4          ; Системний виклик write
    mov ebx, 1          ; Дескриптор файлу stdout
    mov ecx, debug_msg  ; Адреса повідомлення
    mov edx, 14         ; Довжина повідомлення
    int 0x80            ; Виклик системної функції
%endif

    ; Основний код програми
    mov eax, 1          ; Системний виклик exit
    xor ebx, ebx        ; Код виходу 0
    int 0x80            ; Виклик системної функції

section .data
debug_msg db 'Debug Mode On', 0
```

Пояснення:
```

- **`%ifdef DEBUG`:** Включає блок коду для відладки, якщо визначено макрос `DEBUG`.
- **`mov ecx, debug_msg`:** Адреса повідомлення `debug_msg` завантажується в регістр `ecx`, якщо виконується блок відладки.

4. Директиви макросів

Макроси дозволяють створювати повторно використовувані блоки коду, які можуть бути включені в програму кілька разів.

- **`%macro` / `%endmacro`:** Визначає макрос.
- **`%define`:** Визначає константу або макрос.

Приклад:
```assembly
%macro print 1
    mov eax, 4          ; Системний виклик write
    mov ebx, 1          ; Дескриптор файлу stdout
    mov ecx, %1         ; Адреса повідомлення
    mov edx, 14         ; Довжина повідомлення
    int 0x80            ; Виклик системної функції
%endmacro

section .data
msg db 'Hello, World!', 0

section .text
    global _start

_start:
    print msg           ; Виклик макросу для друку повідомлення

    mov eax, 1          ; Системний виклик exit
    xor ebx, ebx        ; Код виходу 0
    int 0x80            ; Виклик системної функції
```

Пояснення:
- **`%macro print 1`:** Визначає макрос `print` з одним параметром.
- **`print msg`:** Викликає макрос для друку повідомлення `msg`.

5. Директиви визначення точок входу та глобальних символів

Ці директиви використовуються для визначення початкових точок входу та оголошення глобальних змінних або функцій.

- **`global`:** Оголошує символ (змінну або функцію) глобальною.
- **`extern`:** Оголошує зовнішній символ, який буде визначений в іншому модулі.

Приклад:
```assembly
section .text
    global _start

_start:
    ; Код програми
    mov eax, 1          ; Системний виклик exit
    xor ebx, ebx        ; Код виходу 0
    int 0x80            ; Виклик системної функції
```

Пояснення:
- **`global _start`:** Оголошує мітку `_start` глобальною, що робить її доступною для зовнішніх модулів.
- **`_start`:** Початкова точка входу програми, де розміщуються основні інструкції.

Висновок

Директиви асемблера є важливими інструментами для управління компіляцією та організацією коду

12. Трансляція програм ASM за першим і другим переглядом.

Трансляція програм ASM за першим і другим переглядом

Трансляція програм на мові асемблера зазвичай виконується в два проходи. Цей підхід дозволяє вирішувати посилання вперед, коли інструкції або дані, на які посилається код, визначаються пізніше в програмі.

Перший перегляд

У першому перегляді асемблер аналізує вихідний код і будує таблицю символів, в якій зберігаються імена всіх міток і їхні адреси. Під час першого перегляду також визначається довжина кожної інструкції, але машинний код ще не генерується.

Основні завдання першого перегляду:
1. Побудова таблиці символів.
2. Визначення розміщення змінних і команд у пам'яті.
3. Обчислення зміщень для кожної інструкції та мітки.

Приклад:
```assembly
section .data
    var1 db 10
    var2 db 20

section .text
    global _start

_start:
    mov eax, var1     ; Посилання вперед на var1
    mov ebx, var2     ; Посилання вперед на var2
    jmp label1        ; Посилання вперед на label1

    ; Інші команди

label1:
    ; Тіло мітки
    mov ecx, 30
```

Таблиця символів після першого перегляду:
Символ	Адреса
var1	0x0000
var2	0x0001
_start	0x0002
label1	0x0008

Другий перегляд

У другому перегляді асемблер використовує таблицю символів, створену під час першого перегляду, для генерації машинного коду. На цьому етапі всі посилання на мітки замінюються відповідними адресами.

Основні завдання другого перегляду:
1. Генерація машинного коду для кожної інструкції.
2. Заміна символічних посилань на їхні фактичні адреси з таблиці символів.

Приклад:
```assembly
section .data
    var1 db 10
    var2 db 20

section .text
    global _start
```

```
_start:
    mov eax, [var1]   ; Генерується машинний код для завантаження значення var1
    mov ebx, [var2]   ; Генерується машинний код для завантаження значення var2
    jmp label1        ; Генерується машинний код для переходу до label1

    ; Інші команди

label1:
    ; Тіло мітки
    mov ecx, 30       ; Генерується машинний код для завантаження значення 30 в ecx
```

Генерація машинного коду:
1. `mov eax, [var1]` -> `B8 00000000` (B8 - opcode для MOV, 00000000 - адреса var1).
2. `mov ebx, [var2]` -> `BB 00000001` (BB - opcode для MOV, 00000001 - адреса var2).
3. `jmp label1` -> `E9 00000006` (E9 - opcode для JMP, 00000006 - зміщення до label1).

Пояснення структури програмної реалізації

1. **Таблиця символів:**
   - Визначається під час першого перегляду.
   - Містить символи (мітки, змінні) і їхні відповідні адреси.

2. **Генерація машинного коду:**
   - Виконується під час другого перегляду.
   - Використовує таблицю символів для заміни символічних посилань на фактичні адреси.

Висновок

Двопрохідний процес трансляції забезпечує коректне розпізнавання і розв'язання символічних посилань у програмах на мові асемблера. Перший перегляд створює таблицю символів і визначає зміщення, тоді як другий перегляд генерує машинний код, використовуючи інформацію з таблиці символів. Цей метод забезпечує ефективну і точну трансляцію програм, дозволяючи асемблеру обробляти посилання вперед.

13. Алгоритм першого перегляду асемблера при трансляції.

Алгоритм першого перегляду асемблера при трансляції

Перший перегляд асемблера під час трансляції коду на мові асемблера виконує кілька основних завдань: побудову таблиці символів, визначення розмірів інструкцій і даних, а також обчислення зміщень. Цей перегляд не генерує машинний код, але забезпечує необхідну інформацію для другого перегляду.

Основні завдання першого перегляду:

1. Побудова таблиці символів.
2. Визначення розмірів інструкцій і даних.
3. Обчислення зміщень для кожної інструкції та мітки.
4. Визначення початкових і кінцевих адрес для сегментів коду і даних.

Алгоритм першого перегляду:

1. **Ініціалізація:**
   - Встановити початкову адресу для секцій коду і даних.
   - Ініціалізувати таблицю символів як порожній список.

2. **Прохід по вихідному коду:**
   - Для кожного рядка коду:
     1. Якщо рядок містить директиву секції (`section .data`, `section .bss`, `section .text`), встановити поточну секцію.
     2. Якщо рядок містить мітку (закінчується символом `:`):
        - Додати мітку і поточну адресу до таблиці символів.
     3. Якщо рядок містить директиву даних (`db`, `dw`, `dd`, `dq`):
        - Визначити розмір даних і оновити поточну адресу відповідно до розміру даних.
     4. Якщо рядок містить інструкцію:
        - Визначити розмір інструкції (включаючи префікси, байти операційного коду, байти ModR/M і SIB, зміщення і безпосередні значення).
        - Оновити поточну адресу відповідно до розміру інструкції.

3. **Завершення першого перегляду:**
   - Таблиця символів заповнена всіма мітками і їхніми адресами.
   - Всі зміщення і розміри інструкцій визначені.

Приклад реалізації алгоритму першого перегляду:

```assembly
; Приклад коду на асемблері
section .data
    var1 db 10
    var2 db 20

section .bss
    buffer resb 64

section .text
    global _start

_start:
    mov eax, [var1]   ; Інструкція 1
    mov ebx, [var2]   ; Інструкція 2
```

```
    jmp label1        ; Інструкція 3

    ; Інші команди

label1:
    ; Тіло мітки
    mov ecx, 30       ; Інструкція 4
```

Пояснення коду та таблиця символів після першого перегляду:

1. **Ініціалізація:**
   - Початкова адреса секції `.data`: 0x0000.
   - Початкова адреса секції `.bss`: 0x0002 (дві байти для `var1` і `var2`).
   - Початкова адреса секції `.text`: 0x0042 (64 байти для `buffer`).

2. **Прохід по вихідному коду:**

   - **Секція `.data`:**
     - `var1` (0x0000) — 1 байт.
     - `var2` (0x0001) — 1 байт.
   - **Секція `.bss`:**
     - `buffer` (0x0002) — 64 байти.
   - **Секція `.text`:**
     - `_start` (0x0042) — мітка.
     - `mov eax, [var1]` — 5 байт.
     - `mov ebx, [var2]` — 5 байт.
     - `jmp label1` — 2 байти.
     - `label1` (0x004e) — мітка.
     - `mov ecx, 30` — 5 байт.

Таблиця символів після першого перегляду:

Символ	Адреса
var1	0x0000
var2	0x0001
buffer	0x0002
_start	0x0042
label1	0x004e

Змістове пояснення:

Під час першого перегляду асемблер проходить по вихідному коду і визначає розміри інструкцій та змінних, а також будує таблицю символів. Це дозволяє асемблеру знати точні адреси всіх міток і змінних до початку генерації машинного коду. Таблиця символів, побудована під час першого перегляду, забезпечує необхідну інформацію для другого перегляду, під час якого відбувається остаточна генерація машинного коду.

14. Алгоритм другого перегляду асемблера при трансляції.

Алгоритм другого перегляду асемблера при трансляції

Другий перегляд асемблера завершує процес трансляції, використовуючи таблицю символів, створену під час першого перегляду. Під час цього проходу асемблер генерує машинний код, замінюючи символічні посилання на їхні фактичні адреси.

Основні завдання другого перегляду:

1. Генерація машинного коду для кожної інструкції.
2. Заміна символічних посилань на фактичні адреси, використовуючи таблицю символів.
3. Визначення та обробка всіх прямих і непрямих посилань.

Алгоритм другого перегляду:

1. **Ініціалізація:**
   - Підготовка до генерації машинного коду.
   - Використання таблиці символів, створеної під час першого перегляду.

2. **Прохід по вихідному коду:**
   - Для кожного рядка коду:
     1. Якщо рядок містить директиву секції (`section .data`, `section .bss`, `section .text`), встановити поточну секцію.
     2. Якщо рядок містить мітку, пропустити її (вона вже оброблена під час першого перегляду).
     3. Якщо рядок містить директиву даних (`db`, `dw`, `dd`, `dq`), зберегти дані у вихідний файл або буфер.
     4. Якщо рядок містить інструкцію, згенерувати відповідний машинний код:
        - Визначити операційний код (opcode) інструкції.
        - Якщо інструкція має операнди, замінити символічні посилання на адреси з таблиці символів.
        - Визначити модифікаторний байт (ModR/M) і, якщо необхідно, SIB байт.
        - Згенерувати машинний код для інструкції, включаючи всі необхідні байти (opcode, ModR/M, SIB, зміщення, безпосередні значення).

3. **Завершення другого перегляду:**
   - Вивести згенерований машинний код у вихідний файл.

Приклад реалізації алгоритму другого перегляду:

```assembly
; Приклад коду на асемблері
section .data
```

```
    var1 db 10
    var2 db 20

section .bss
    buffer resb 64

section .text
    global _start

_start:
    mov eax, [var1]   ; Інструкція 1
    mov ebx, [var2]   ; Інструкція 2
    jmp label1        ; Інструкція 3

    ; Інші команди

label1:
    ; Тіло мітки
    mov ecx, 30       ; Інструкція 4
```

Пояснення коду та процес генерації машинного коду:

1. **Ініціалізація:**
   - Підготовка вихідного файлу для запису машинного коду.
   - Використання таблиці символів:
     ```plaintext
     | Символ  | Адреса  |
     |---------|---------|
     | var1    | 0x0000  |
     | var2    | 0x0001  |
     | buffer  | 0x0002  |
     | _start  | 0x0042  |
     | label1  | 0x004e  |
     ```

2. **Прохід по вихідному коду:**
   - **Секція `.data`:**
     - `var1 db 10` -> 0x0A (запис у вихідний файл)
     - `var2 db 20` -> 0x14 (запис у вихідний файл)
   - **Секція `.bss`:**
     - `buffer resb 64` -> 64 байти (резервування пам'яті)
   - **Секція `.text`:**
     - `_start` (0x0042) -> Мітка (пропустити)
     - `mov eax, [var1]`
       - **Opcode:** `8B 05 00000000` (MOV з пам'яті в регістр)
       - **Адреса:** 0x00000000 (адреса `var1` з таблиці символів)
     - `mov ebx, [var2]`

- **Opcode:** `8B 1D 00000001` (MOV з пам'яті в регістр)
- **Адреса:** 0x00000001 (адреса `var2` з таблиці символів)
- `jmp label1`
- **Opcode:** `E9 00000006` (JMP з відносним зміщенням)
- **Зміщення:** 0x00000006 (від поточної адреси до `label1`)
- `label1` (0x004e) -> Мітка (пропустити)
- `mov ecx, 30`
- **Opcode:** `B9 1E000000` (MOV безпосереднє значення в регістр)
- **Значення:** 30 (0x1E)

3. **Завершення другого перегляду:**
   - Вихідний файл містить:
     ```plaintext
     8B 05 00 00 00 00  ; mov eax, [var1]
     8B 1D 01 00 00 00  ; mov ebx, [var2]
     E9 06 00 00 00     ; jmp label1
     B9 1E 00 00 00     ; mov ecx, 30
     ```

Пояснення структури програмної реалізації

1. **Таблиця символів:**
   - Визначає адреси всіх міток і змінних, створених під час першого перегляду.

2. **Генерація машинного коду:**
   - Кожна інструкція генерує машинний код на основі таблиці символів.
   - Символічні посилання замінюються на фактичні адреси.

Висновок

Другий перегляд асемблера використовує таблицю символів, створену під час першого перегляду, для генерації машинного коду. Він замінює символічні посилання на фактичні адреси, генерує машинний код для кожної інструкції і записує його у вихідний файл. Цей процес завершує трансляцію програми, роблячи її готовою до виконання.

15. Загальні принципи компонування програм. Об’єктні файли.

Загальні принципи компонування програм. Об’єктні файли

Компонування програм (linking) — це процес об'єднання одного або декількох об'єктних файлів в єдиний виконуваний файл. Цей процес включає вирішення всіх зовнішніх

посилань і створення остаточного образу програми, який може бути завантажений і виконаний операційною системою.

Основні етапи компонування:

1. **Створення об'єктних файлів:** Під час компіляції вихідний код компілюється в об'єктні файли, які містять машинний код і дані.
2. **Об'єднання об'єктних файлів:** Лінкер об'єднує кілька об'єктних файлів в один виконуваний файл.
3. **Резольвінг зовнішніх посилань:** Лінкер вирішує всі зовнішні посилання між об'єктними файлами і бібліотеками.
4. **Створення виконуваного файлу:** Лінкер створює остаточний виконуваний файл, який включає всі необхідні код і дані.

Об’єктні файли

Об'єктний файл — це проміжний файл, який створюється компілятором під час компіляції вихідного коду. Він містить машинний код, дані і метадані, необхідні для компонування. Об'єктні файли зазвичай мають розширення `.o` (Unix/Linux) або `.obj` (Windows).

Основні секції об'єктного файлу:

1. **Секція коду:** Містить машинний код програми.
2. **Секція даних:** Містить ініціалізовані дані.
3. **Секція неініціалізованих даних (BSS):** Містить неініціалізовані дані.
4. **Таблиця символів:** Містить інформацію про всі символи (мітки, функції, змінні) в програмі.
5. **Таблиця релокацій:** Містить інформацію про місця в коді, які потребують корекції адрес після компонування.

Приклад створення і компонування об'єктних файлів

Створення вихідного коду

Файл `main.asm`:
```assembly
section .data
    msg db 'Hello, World!', 0

section .text
    global _start

_start:
    call print_message
    mov eax, 60
    xor edi, edi
    syscall
```

```
extern print_message
```

Файл `print.asm`:
```assembly
section .text
    global print_message

print_message:
    mov eax, 1
    mov edi, 1
    lea rsi, [rel msg]
    mov edx, 13
    syscall
    ret
```

Компіляція об'єктних файлів

```sh
nasm -f elf64 main.asm -o main.o
nasm -f elf64 print.asm -o print.o
```

Компонування об'єктних файлів

```sh
ld -o hello main.o print.o
```

Пояснення коду та процесу

1. **Файл `main.asm`:**
   - Визначає точку входу `_start`.
   - Викликає зовнішню функцію `print_message`.

2. **Файл `print.asm`:**
   - Визначає функцію `print_message`.
   - Виконує системний виклик для виводу повідомлення на екран.

3. **Компіляція:**
   - `nasm -f elf64 main.asm -o main.o`: Компілює `main.asm` у об'єктний файл `main.o`.
   - `nasm -f elf64 print.asm -o print.o`: Компілює `print.asm` у об'єктний файл `print.o`.

4. **Компонування:**
   - `ld -o hello main.o print.o`: Комбінує об'єктні файли `main.o` і `print.o` у виконуваний файл `hello`.

Таблиця символів і релокацій

Таблиця символів:
- Вміщує всі символи (мітки, функції, змінні) з обох об'єктних файлів.
- Включає глобальні символи (наприклад, `_start`, `print_message`).

Таблиця релокацій:
- Містить інформацію про місця в коді, які потребують корекції адрес.
- Використовується лінкером для коректного вирішення зовнішніх посилань.

Висновок

Компонування програм — це процес об'єднання об'єктних файлів у єдиний виконуваний файл. Об'єктні файли містять машинний код, дані і метадані, необхідні для компонування. Лінкер вирішує зовнішні посилання між об'єктними файлами і створює остаточний виконуваний файл. Розуміння принципів компонування та структури об'єктних файлів є важливим для розробки ефективних програм.

16. Компонувальники і принципи їх роботи.

Компонувальники і принципи їх роботи (з точки зору асемблера)

Компонувальники, або лінкери (linkers), — це інструменти, які об'єднують один або кілька об'єктних файлів у єдиний виконуваний файл або бібліотеку. Вони виконують вирішення зовнішніх посилань, з'єднують різні частини програми і розміщують їх у пам'яті. Це особливо важливо для програм на асемблері, де пряме маніпулювання пам'яттю і адресами є звичайною справою.

Основні принципи роботи компонентувальників

1. **Об'єднання об'єктних файлів:** Лінкер бере кілька об'єктних файлів, які були створені компілятором або асемблером, і об'єднує їх в один виконуваний файл або бібліотеку.

2. **Резольвінг зовнішніх посилань:** Лінкер вирішує всі зовнішні посилання між об'єктними файлами. Зовнішні посилання виникають, коли одна частина коду звертається до символів (функцій або змінних), визначених в іншій частині коду.

3. **Розміщення в пам'яті:** Лінкер визначає, де кожен об'єктний модуль буде розташований у пам'яті під час виконання програми.

4. **Релокація:** Лінкер коригує адреси в об'єктних файлах, щоб відобразити їхнє фактичне розташування в пам'яті. Це включає корекцію всіх адресних посилань, що використовуються в програмі.

5. **Оптимізація:** Лінкер може виконувати різні оптимізації, такі як видалення не використовуваних функцій або об'єднання схожих сегментів коду.

Структура об'єктного файлу

Об'єктний файл містить кілька важливих секцій:
- **Секція коду (.text):** Містить машинний код програми.
- **Секція даних (.data):** Містить ініціалізовані дані.
- **Секція неініціалізованих даних (.bss):** Містить неініціалізовані дані.
- **Таблиця символів:** Містить інформацію про всі символи (мітки, функції, змінні) в програмі.
- **Таблиця релокацій:** Містить інформацію про місця в коді, які потребують корекції адрес після компонування.

Приклад процесу компонування

Вихідний код на асемблері

Файл `main.asm`:
```assembly
section .data
    msg db 'Hello, World!', 0

section .text
    global _start

_start:
    call print_message
    mov eax, 60
    xor edi, edi
    syscall

extern print_message
```

Файл `print.asm`:
```assembly
section .text
    global print_message

print_message:
    mov eax, 1
    mov edi, 1
    lea rsi, [rel msg]
```

```
    mov edx, 13
    syscall
    ret
```

Компіляція об'єктних файлів

```sh
nasm -f elf64 main.asm -o main.o
nasm -f elf64 print.asm -o print.o
```

Компонування об'єктних файлів

```sh
ld -o hello main.o print.o
```

Процес компонування

1. **Об'єднання об'єктних файлів:**
   Лінкер об'єднує `main.o` і `print.o` в один виконуваний файл `hello`.

2. **Резольвінг зовнішніх посилань:**
   Лінкер виявляє, що функція `print_message`, яка викликається в `main.o`, визначена в `print.o`. Він вирішує це посилання, об'єднуючи адреси функції.

3. **Розміщення в пам'яті:**
   Лінкер визначає, де кожна секція (код, дані) буде розташована у виконуваному файлі.

4. **Релокація:**
   Лінкер коригує адреси в об'єктних файлах. Наприклад, якщо `msg` визначена в `main.o`, її адреса коригується, щоб відображати її фактичне розташування у виконуваному файлі.

5. **Оптимізація:**
   Лінкер може виконувати оптимізацію, таку як видалення не використовуваних функцій.

Пояснення процесу

main.o:
- Секція `.data`: містить `msg`.
- Секція `.text`: містить код `_start` і виклик `print_message`.
- Таблиця символів: `_start`, `msg`.
- Таблиця релокацій: посилання на `print_message`.

print.o:

- Секція `.text`: містить код `print_message`.
- Таблиця символів: `print_message`.

ld (лінкер):
- Об'єднує `.text` і `.data` секції з `main.o` і `print.o`.
- Виконує релокацію: коригує адреси `msg` і `print_message`.
- Створює виконуваний файл `hello`.

Висновок

Компонувальники відіграють ключову роль в об'єднанні об'єктних файлів, вирішенні зовнішніх посилань і створенні виконуваних файлів. Вони забезпечують коректне розміщення коду і даних у пам'яті, а також коригують адреси для забезпечення правильного виконання програми. Розуміння принципів роботи лінкерів є важливим для ефективного програмування на асемблері і створення надійних програм.

Список питань до розділу "Бібліотеки DLL"

1.Переваги і недоліки статичного компонування програм.

Переваги і недоліки статичного компонування програм

Статичне компонування програм (static linking) передбачає включення всіх необхідних бібліотек безпосередньо у виконуваний файл під час етапу компонування. Це протилежно до динамічного компонування (dynamic linking), яке використовує DLL (Dynamic Link Libraries). Розглянемо переваги та недоліки статичного компонування програм в контексті використання DLL.

Переваги статичного компонування

1. **Автономність виконуваного файлу:**
   - Виконуваний файл містить всі необхідні бібліотеки, що забезпечує його незалежність від зовнішніх файлів під час виконання.
   - **Приклад:** У програмі всі функції з бібліотеки `mathlib` вбудовані безпосередньо у виконуваний файл.
   ```c
   // main.c
   #include "mathlib.h"  // Всі функції mathlib включені статично

   int main() {
       double result = add(3.0, 4.0);
       printf("Result: %f\n", result);
       return 0;
   }
   ```

```

2. **Простота розгортання:**
   - Легше розгортати програму на нових системах, оскільки немає потреби встановлювати додаткові бібліотеки.
   - **Приклад:** Виконуваний файл `program.exe` включає всі необхідні бібліотеки, тому для розгортання достатньо скопіювати лише один файл.

3. **Швидкість виконання:**
   - Відсутність необхідності завантаження зовнішніх бібліотек може зменшити час запуску програми.
   - **Приклад:** Функції з бібліотеки `mathlib` вже знаходяться в пам'яті, тому вони викликаються без додаткових накладних витрат.

#### Недоліки статичного компонування

1. **Збільшений розмір виконуваного файлу:**
   - Всі необхідні бібліотеки включаються у виконуваний файл, що збільшує його розмір.
   - **Приклад:** Виконуваний файл `program.exe` містить всю бібліотеку `mathlib`, що збільшує його розмір з 1 МБ до 5 МБ.
     ```plaintext
     $ ls -lh
     -rwxr-xr-x 1 user user 5.0M program.exe
     ```

2. **Відсутність можливості оновлення бібліотек:**
   - Якщо в бібліотеці буде знайдена помилка або вийде оновлення, необхідно перекомпілювати і перевипустити весь виконуваний файл.
   - **Приклад:** Бібліотека `mathlib` оновлюється для виправлення помилки, і кожен виконуваний файл, що її використовує, потребує перекомпіляції.

3. **Використання пам'яті:**
   - Всі копії статично зв'язаних бібліотек займають окрему пам'ять, що може призвести до надмірного використання пам'яті при виконанні кількох екземплярів програми.
   - **Приклад:** Кожна запущена копія `program.exe` використовує власну копію функцій з `mathlib`, що збільшує загальне використання пам'яті.

Пояснення на основі структури програмної реалізації

Статичне компонування

1. **Файл заголовка бібліотеки (mathlib.h):**
   ```c
   // mathlib.h
   #ifndef MATHLIB_H
   #define MATHLIB_H

   double add(double a, double b);

```
double subtract(double a, double b);

#endif // MATHLIB_H
```

2. **Файл реалізації бібліотеки (mathlib.c):**
```c
// mathlib.c
#include "mathlib.h"

double add(double a, double b) {
    return a + b;
}

double subtract(double a, double b) {
    return a - b;
}
```

3. **Основний файл програми (main.c):**
```c
// main.c
#include <stdio.h>
#include "mathlib.h"

int main() {
    double result = add(3.0, 4.0);
    printf("Result: %f\n", result);
    return 0;
}
```

4. **Компіляція з статичним компонуванням:**
```sh
gcc -c mathlib.c -o mathlib.o
gcc -c main.c -o main.o
gcc main.o mathlib.o -o program.exe
```

Висновок

Статичне компонування програм має свої переваги, такі як автономність виконуваного файлу, простота розгортання та потенційно швидший час виконання. Однак, воно також має недоліки, включаючи збільшений розмір виконуваного файлу, відсутність можливості оновлення бібліотек без перекомпіляції і надмірне використання пам'яті. Розуміння цих аспектів є важливим для вибору відповідного підходу до компонування програм у конкретних проектах.

2.Визначення динамічної бібліотеки. Схема динамічного завантаження в адресний простір процесу.

Визначення динамічної бібліотеки

Динамічна бібліотека (Dynamic Link Library, DLL) — це файл, що містить код і дані, які можуть бути використані декількома програмами одночасно під час їх виконання. DLL дозволяють програмам динамічно завантажувати необхідні їм функції та ресурси під час виконання, що забезпечує гнучкість і зменшує використання пам'яті.

Схема динамічного завантаження в адресний простір процесу

Динамічне завантаження DLL в адресний простір процесу включає кілька етапів:

1. **Ініціалізація процесу:**
   - Під час запуску програми операційна система завантажує основний виконуваний файл в адресний простір процесу.
   - Операційна система також завантажує всі необхідні DLL, зазначені у виконуваному файлі, в адресний простір процесу.

2. **Завантаження DLL:**
   - Коли програма викликає функцію з DLL, операційна система завантажує DLL в пам'ять, якщо вона ще не завантажена.
   - Операційна система виконує релокацію DLL, щоб забезпечити її правильне розміщення в адресному просторі процесу.

3. **Резольвінг символів:**
   - Операційна система вирішує символічні посилання, забезпечуючи коректне зв'язування викликів функцій в програмі з відповідними функціями в DLL.

4. **Виконання програми:**
   - Програма виконується, викликаючи функції з DLL за необхідності.
   - Якщо DLL ще не завантажена під час виклику функції, операційна система виконує завантаження DLL і резольвінг символів в режимі реального часу.

Приклад реалізації динамічного завантаження DLL на мові C

Створення DLL

1. **Файл заголовка бібліотеки (mathlib.h):**
   ```c
   // mathlib.h
   #ifndef MATHLIB_H
   #define MATHLIB_H

   #ifdef MATHLIB_EXPORTS
   #define MATHLIB_API __declspec(dllexport)
   ```

```
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

2. **Файл реалізації бібліотеки (mathlib.c):**
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

3. **Компіляція DLL:**
```sh
gcc -shared -o mathlib.dll mathlib.c
```

Використання DLL в програмі

1. **Основний файл програми (main.c):**
```c
// main.c
#include <stdio.h>
#include <windows.h>
#include "mathlib.h"

typedef double (*AddFunc)(double, double);
typedef double (*SubtractFunc)(double, double);

int main() {
    HMODULE hLib = LoadLibrary("mathlib.dll");
    if (hLib == NULL) {
        printf("Failed to load DLL\n");
        return 1;
    }
```

```
    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");

    if (add == NULL || subtract == NULL) {
        printf("Failed to get function address\n");
        FreeLibrary(hLib);
        return 1;
    }

    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);

    FreeLibrary(hLib);
    return 0;
}
```

2. **Компіляція основного файлу:**
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Пояснення коду та процесу

1. **Створення DLL:**
   - Файл `mathlib.h` містить оголошення функцій з використанням макроса `__declspec(dllexport)` для експорту функцій у DLL.
   - Файл `mathlib.c` містить реалізацію функцій `add` та `subtract`.

2. **Компіляція DLL:**
   - Команда `gcc -shared -o mathlib.dll mathlib.c` компілює файл `mathlib.c` в динамічну бібліотеку `mathlib.dll`.

3. **Використання DLL у програмі:**
   - Файл `main.c` завантажує бібліотеку `mathlib.dll` за допомогою функції `LoadLibrary`.
   - Викликає функції `GetProcAddress`, щоб отримати адреси функцій `add` та `subtract`.
   - Виконує виклики функцій `add` та `subtract`.
   - Звільняє бібліотеку за допомогою `FreeLibrary`.

Висновок

Динамічні бібліотеки (DLL) забезпечують гнучкість і економію пам'яті, дозволяючи програмам динамічно завантажувати та використовувати спільні функції під час виконання. Процес динамічного завантаження DLL в адресний простір процесу включає ініціалізацію, завантаження DLL, резольвінг символів і виконання програми. Використання DLL дозволяє зменшити розмір виконуваного файлу і спрощує оновлення бібліотек.

3.Переваги і недоліки використання динамічних бібліотек.

Переваги і недоліки використання динамічних бібліотек

Динамічні бібліотеки (Dynamic Link Libraries, DLL) дозволяють програмам динамічно завантажувати та використовувати функції та ресурси під час виконання. Це забезпечує значну гнучкість і ефективність у використанні ресурсів, але також має певні недоліки. Розглянемо основні переваги та недоліки використання DLL.

Переваги використання динамічних бібліотек

1. **Економія пам'яті:**
   - DLL можуть бути завантажені в пам'ять один раз і використовуватися кількома програмами одночасно, що зменшує загальне використання пам'яті.
   - **Приклад:**
     ```plaintext
     Програми A, B і C використовують одну й ту ж саму бібліотеку DLL, завантажену в пам'ять один раз.
     ```

2. **Можливість оновлення бібліотек:**
   - DLL можуть бути оновлені без необхідності перекомпіляції або перезапуску програм, які їх використовують, що спрощує підтримку і розгортання оновлень.
   - **Приклад:**
     ```plaintext
     Оновлена бібліотека mathlib.dll автоматично використовується програмами без їх перекомпіляції.
     ```

3. **Зменшення розміру виконуваного файлу:**
   - Оскільки функції з бібліотек не включаються безпосередньо у виконуваний файл, його розмір значно зменшується.
   - **Приклад:**
     ```plaintext
     Виконуваний файл program.exe містить лише посилання на функції з mathlib.dll, що зменшує його розмір.
     ```

4. **Гнучкість і модульність:**
   - DLL дозволяють створювати модульні програми, де різні частини функціоналу можуть бути розподілені по різних бібліотеках.
   - **Приклад:**
     ```plaintext
     Програма розподілена на кілька DLL, кожна з яких відповідає за різні модулі, такі як обробка графіки, мережеві операції тощо.
     ```

Недоліки використання динамічних бібліотек

1. **Складність у керуванні версіями:**
   - Програми можуть стати несумісними з новими версіями DLL, що може викликати проблеми з сумісністю (так званий "DLL Hell").
   - **Приклад:**
     ```plaintext
     Нова версія mathlib.dll змінює поведінку функції add(), що може порушити роботу програм, які її використовують.
     ```

2. **Збільшений час завантаження:**
   - Завантаження DLL під час виконання програми може збільшити час її запуску.
   - **Приклад:**
     ```plaintext
     Програма program.exe завантажується повільніше через необхідність завантаження mathlib.dll під час старту.
     ```

3. **Проблеми з безпекою:**
   - DLL можуть бути замінені шкідливими версіями, що створює ризики для безпеки.
   - **Приклад:**
     ```plaintext
     Зловмисник замінює mathlib.dll на шкідливу версію, яка виконує небажані дії.
     ```

4. **Залежність від середовища виконання:**
   - Виконувані файли залежать від наявності необхідних DLL у системі, що може викликати проблеми при перенесенні програм на інші машини.
   - **Приклад:**
     ```plaintext
     Програма program.exe не запускається на новій машині через відсутність mathlib.dll.
     ```

Приклад використання динамічних бібліотек

Створення DLL

Файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
```

```
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

Компіляція DLL:
```sh
gcc -shared -o mathlib.dll mathlib.c
```

Використання DLL в програмі

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include <windows.h>
#include "mathlib.h"

typedef double (*AddFunc)(double, double);
typedef double (*SubtractFunc)(double, double);

int main() {
    HMODULE hLib = LoadLibrary("mathlib.dll");
    if (hLib == NULL) {
        printf("Failed to load DLL\n");
        return 1;
    }

    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
```

```
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");

    if (add == NULL || subtract == NULL) {
        printf("Failed to get function address\n");
        FreeLibrary(hLib);
        return 1;
    }

    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);

    FreeLibrary(hLib);
    return 0;
}
```

Компіляція основного файлу:
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Висновок

Використання динамічних бібліотек має багато переваг, включаючи економію пам'яті, можливість оновлення бібліотек без перекомпіляції програм, зменшення розміру виконуваних файлів і підвищену гнучкість. Однак, це також має певні недоліки, такі як складність у керуванні версіями, збільшений час завантаження, потенційні проблеми з безпекою і залежність від середовища виконання. Розуміння цих аспектів допоможе прийняти обґрунтоване рішення про використання DLL у конкретних проектах.

4.Зворотня сумісність динамічних бібліотек.

Зворотна сумісність динамічних бібліотек

Зворотна сумісність (backward compatibility) динамічних бібліотек означає, що нові версії бібліотек можуть бути використані з програмами, які були скомпільовані з використанням старих версій цих бібліотек, без потреби вносити зміни в ці програми. Це важливий аспект при розробці і підтримці динамічних бібліотек, оскільки дозволяє оновлювати бібліотеки без необхідності перекомпіляції або модифікації залежних програм.

Основні принципи забезпечення зворотної сумісності

1. **Збереження інтерфейсів:**
   - Не змінювати сигнатури існуючих функцій (імена, типи аргументів, типи повертаємих значень).

- **Приклад:**
  ```c
  // Старий інтерфейс
  double add(double a, double b);

  // Новий інтерфейс (сумісний)
  double add(double a, double b);
  ```

2. **Додавання нових функцій замість зміни існуючих:**
   - Додавати нові функції замість зміни існуючих, щоб уникнути порушення сумісності.
   - **Приклад:**
     ```c
     // Старий інтерфейс
     double add(double a, double b);

     // Новий інтерфейс
     double add_v2(double a, double b, int round);
     ```

3. **Використання версійованих символів:**
   - Використовувати символи з версіями для розрізнення різних версій функцій.
   - **Приклад:**
     ```c
     // Старий інтерфейс
     double add(double a, double b) __attribute__((version("add@v1")));

     // Новий інтерфейс
     double add_v2(double a, double b, int round) __attribute__((version("add_v2@v2")));
     ```

4. **Збереження внутрішньої структури даних:**
   - Не змінювати внутрішні структури даних, які використовуються в бібліотеці.
   - **Приклад:**
     ```c
     typedef struct {
         double x;
         double y;
     } Point;
     ```

Приклад реалізації зворотної сумісності

Старий інтерфейс бібліотеки

Файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
```

```
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

Новий інтерфейс бібліотеки з зворотною сумісністю

Оновлений файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);
MATHLIB_API double add_v2(double a, double b, int round);

#endif // MATHLIB_H
```

```

**Оновлений файл реалізації бібліотеки (mathlib.c):**
```c
// mathlib.c
#include "mathlib.h"
#include <math.h>

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}

MATHLIB_API double add_v2(double a, double b, int round) {
    double result = a + b;
    if (round) {
        result = round(result);
    }
    return result;
}
```

Використання нової версії DLL у програмі

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include <windows.h>
#include "mathlib.h"

typedef double (*AddFunc)(double, double);
typedef double (*SubtractFunc)(double, double);
typedef double (*AddFunc_v2)(double, double, int);

int main() {
    HMODULE hLib = LoadLibrary("mathlib.dll");
    if (hLib == NULL) {
        printf("Failed to load DLL\n");
        return 1;
    }

    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");
    AddFunc_v2 add_v2 = (AddFunc_v2)GetProcAddress(hLib, "add_v2");
```
```

```
    if (add == NULL || subtract == NULL || add_v2 == NULL) {
        printf("Failed to get function address\n");
        FreeLibrary(hLib);
        return 1;
    }

    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    double result3 = add_v2(5.0, 6.0, 1);
    printf("Add: %f, Subtract: %f, Add_v2: %f\n", result1, result2, result3);

    FreeLibrary(hLib);
    return 0;
}
```

Висновок

Зворотна сумісність динамічних бібліотек є критично важливою для забезпечення стабільності і надійності програм, які їх використовують. Основні принципи забезпечення зворотної сумісності включають збереження інтерфейсів, додавання нових функцій замість зміни існуючих, використання версійованих символів і збереження внутрішньої структури даних. Дотримуючись цих принципів, можна оновлювати динамічні бібліотеки без порушення сумісності з існуючими програмами.

5.Загальні принципи зв’язування з DLL в алгоритмічних мовах для неявного і явного зв’язування.

Загальні принципи зв’язування з DLL в алгоритмічних мовах для неявного і явного зв’язування

Зв'язування з DLL (Dynamic Link Library) може здійснюватися двома основними способами: неявним і явним. Розглянемо ці підходи і їх реалізацію на прикладі мови програмування C.

Неявне зв’язування (Implicit Linking)

Неявне зв’язування відбувається під час завантаження програми, коли операційна система автоматично завантажує всі необхідні DLL, визначені в програмі. Це досягається за допомогою імпортних бібліотек і заголовкових файлів, які містять оголошення функцій з DLL.

Основні принципи:

1. **Включення заголовкового файлу DLL:** Програма включає заголовковий файл, який містить оголошення функцій, експортуваних з DLL.
2. **Лінкування з імпортною бібліотекою:** Під час компонування програма лінкується з імпортною бібліотекою (файлом `.lib`), яка містить інформацію про DLL.
3. **Автоматичне завантаження DLL:** Під час запуску операційна система автоматично завантажує DLL і резолвіть всі зовнішні посилання.

Приклад неявного зв'язування:

1. **Створення DLL:**

Файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

Компіляція DLL:
```sh
gcc -shared -o mathlib.dll mathlib.c
```

2. **Використання DLL у програмі:**

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include "mathlib.h"

int main() {
    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);
    return 0;
}
```

Компіляція з неявним зв'язуванням:
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Явне зв’язування (Explicit Linking)

Явне зв’язування відбувається під час виконання програми, коли вона самостійно завантажує необхідні DLL за допомогою відповідних функцій операційної системи, таких як `LoadLibrary` і `GetProcAddress` у Windows. Це забезпечує більшу гнучкість і дозволяє завантажувати DLL тільки за потребою.

Основні принципи:
1. **Завантаження DLL під час виконання:** Програма використовує функцію `LoadLibrary`, щоб завантажити DLL в пам'ять під час виконання.
2. **Отримання адрес функцій:** Програма використовує функцію `GetProcAddress`, щоб отримати адреси потрібних функцій з DLL.
3. **Використання функцій з DLL:** Програма викликає функції з DLL через отримані адреси.
4. **Вивантаження DLL:** Після використання програма може вивантажити DLL за допомогою функції `FreeLibrary`.

Приклад явного зв'язування:

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include <windows.h>

typedef double (*AddFunc)(double, double);
typedef double (*SubtractFunc)(double, double);
```

```
int main() {
    HMODULE hLib = LoadLibrary("mathlib.dll");
    if (hLib == NULL) {
        printf("Failed to load DLL\n");
        return 1;
    }

    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");

    if (add == NULL || subtract == NULL) {
        printf("Failed to get function address\n");
        FreeLibrary(hLib);
        return 1;
    }

    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);

    FreeLibrary(hLib);
    return 0;
}
```

Компіляція основного файлу:
```sh
gcc -o main.exe main.c
```

Пояснення процесу зв'язування

Неявне зв'язування:

1. **Включення заголовка:** Програма включає заголовковий файл `mathlib.h`, який містить оголошення функцій `add` та `subtract`.
2. **Лінкування:** Під час компіляції програма лінкується з імпортною бібліотекою `mathlib.lib`.
3. **Завантаження DLL:** Під час виконання операційна система автоматично завантажує `mathlib.dll` і резолвіть посилання на функції.

Явне зв'язування:

1. **Завантаження DLL:** Програма використовує функцію `LoadLibrary`, щоб завантажити `mathlib.dll` в пам'ять.
2. **Отримання адрес функцій:** Програма використовує функцію `GetProcAddress`, щоб отримати адреси функцій `add` та `subtract`.

3. **Використання функцій:** Програма викликає функції через отримані адреси.
4. **Вивантаження DLL:** Програма використовує функцію `FreeLibrary`, щоб вивантажити DLL після використання.

Висновок

Неявне зв'язування з DLL забезпечує простоту використання і автоматичне завантаження бібліотек під час виконання програми. Явне зв'язування, натомість, забезпечує більшу гнучкість, дозволяючи програмі завантажувати і використовувати DLL динамічно під час виконання. Розуміння цих підходів і їх реалізації дозволяє ефективно використовувати динамічні бібліотеки в алгоритмічних мовах програмування.

6.Взаємодія динамічної бібліотеки з адресним простором процесу. Особливості об'єктного коду динамічних бібліотек.

Взаємодія динамічної бібліотеки з адресним простором процесу

Динамічні бібліотеки (DLL) взаємодіють з адресним простором процесу під час виконання програми, що дозволяє програмам спільно використовувати функції та дані, зберігаючи при цьому модульність і зменшуючи загальне використання пам'яті. Розглянемо основні принципи цієї взаємодії, а також особливості об'єктного коду динамічних бібліотек.

Основні принципи взаємодії DLL з адресним простором процесу

1. **Завантаження DLL:**
   - Коли процес потребує функцію з DLL, операційна система завантажує бібліотеку в адресний простір процесу.
   - Завантаження може відбуватися неявно (під час запуску програми) або явно (під час виконання програми).

2. **Вирішення зовнішніх посилань:**
   - Операційна система забезпечує вирішення всіх зовнішніх посилань на функції та дані з DLL. Це включає визначення фактичних адрес функцій і змінних у завантаженій DLL.

3. **Релокація:**
   - Якщо DLL не може бути завантажена в свою бажану базову адресу, операційна система виконує релокацію. Це процес коригування адрес в коді DLL відповідно до фактичного місця її завантаження в пам'ять.

4. **Спільне використання бібліотек:**
   - Якщо одна і та ж DLL використовується кількома процесами, операційна система може завантажити її в пам'ять лише один раз, а потім ділитися цією копією між процесами. Це зменшує використання пам'яті.

Особливості об'єктного коду динамічних бібліотек

1. **Експорт символів:**
   - DLL містять спеціальні інструкції для експорту символів (функцій і змінних), які можуть бути використані іншими програмами.
   - **Приклад:**
     ```c
     #ifdef MATHLIB_EXPORTS
     #define MATHLIB_API __declspec(dllexport)
     #else
     #define MATHLIB_API __declspec(dllimport)
     #endif

     MATHLIB_API double add(double a, double b);
     ```

2. **Імпорт символів:**
   - Використання символів з інших DLL здійснюється через спеціальні інструкції для імпорту.
   - **Приклад:**
     ```c
     MATHLIB_API double add(double a, double b);
     ```

3. **Таблиці імпорту і експорту:**
   - DLL містять таблиці імпорту та експорту, які використовуються для вирішення посилань на функції та змінні під час завантаження.

4. **Файл імпортної бібліотеки:**
   - Поряд з DLL створюється імпортна бібліотека (файл `.lib`), яка використовується під час компіляції програм для неявного зв'язування.

Приклад створення та використання DLL

Створення DLL

Файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif
```

```
MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

Компіляція DLL:
```sh
gcc -shared -o mathlib.dll mathlib.c -Wl,--out-implib,mathlib.lib
```

Використання DLL у програмі

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include "mathlib.h"

int main() {
    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);
    return 0;
}
```

Компіляція з неявним зв'язуванням:
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Пояснення процесу

1. **Завантаження DLL:**
   - Під час запуску програми `main.exe`, операційна система автоматично завантажує `mathlib.dll` в адресний простір процесу.

2. **Вирішення зовнішніх посилань:**
   - Операційна система резолвіть посилання на функції `add` та `subtract`, використовуючи таблиці імпорту та експорту з `mathlib.dll`.

3. **Релокація:**
   - Якщо DLL не може бути завантажена в свою бажану базову адресу, операційна система коригує всі внутрішні адреси функцій і даних у DLL.

4. **Спільне використання бібліотек:**
   - Якщо кілька процесів використовують одну і ту ж DLL, операційна система завантажує її лише один раз і ділиться цією копією між процесами, що зменшує використання пам'яті.

Особливості об'єктного коду

- **Експорт символів:** За допомогою макроса `__declspec(dllexport)` функції `add` і `subtract` експортуються з `mathlib.dll`.
- **Імпорт символів:** Інші програми використовують ці функції через макрос `__declspec(dllimport)`.
- **Таблиці імпорту і експорту:** Містять інформацію про всі експортовані і імпортовані символи.
- **Файл імпортної бібліотеки:** `mathlib.lib` використовується під час компіляції програм для неявного зв'язування.

Висновок

Динамічні бібліотеки взаємодіють з адресним простором процесу, забезпечуючи спільне використання функцій і даних між кількома програмами. Особливості об'єктного коду динамічних бібліотек включають експорт і імпорт символів, таблиці імпорту та експорту, а також можливість релокації. Розуміння цих аспектів дозволяє ефективно використовувати DLL у розробці програмного забезпечення.

7.Точка входу динамічної бібліотеки і приклади її раціонального використання.

Точка входу динамічної бібліотеки і приклади її раціонального використання

Точка входу динамічної бібліотеки (DLL) - це спеціальна функція, яка викликається операційною системою під час завантаження, вивантаження або виконання певних дій із бібліотекою. У Windows такою функцією є `DllMain`. Ця функція надає можливість

виконувати ініціалізацію або завершальні дії, коли бібліотека завантажується або вивантажується.

Визначення точки входу

Функція DllMain:
```c
BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            // Викликається, коли DLL завантажується в процес
            break;
        case DLL_THREAD_ATTACH:
            // Викликається, коли новий потік створюється в процесі
            break;
        case DLL_THREAD_DETACH:
            // Викликається, коли потік завершується
            break;
        case DLL_PROCESS_DETACH:
            // Викликається, коли DLL вивантажується з процесу
            break;
    }
    return TRUE;
}
```

Параметри DllMain

- **hModule:** Дескриптор модуля DLL.
- **ul_reason_for_call:** Причина виклику функції. Може приймати такі значення:
  - **DLL_PROCESS_ATTACH:** DLL завантажується в адресний простір процесу.
  - **DLL_THREAD_ATTACH:** Процес створює новий потік.
  - **DLL_THREAD_DETACH:** Потік завершується.
  - **DLL_PROCESS_DETACH:** DLL вивантажується з адресного простору процесу.
- **lpReserved:** Зарезервовано для майбутнього використання. Зазвичай NULL.

Раціональне використання точки входу

1. **Ініціалізація ресурсів:**
   - Використовується для ініціалізації глобальних змінних, відкриття файлів або налаштування мережевих з'єднань під час завантаження DLL.
   - **Приклад:**
     ```c
     case DLL_PROCESS_ATTACH:
         InitializeCriticalSection(&CriticalSection);
         break;
     ```

2. **Звільнення ресурсів:**
   - Використовується для звільнення ресурсів, закриття файлів або розірвання мережевих з'єднань під час вивантаження DLL.
   - **Приклад:**
     ```c
     case DLL_PROCESS_DETACH:
         DeleteCriticalSection(&CriticalSection);
         break;
     ```

3. **Ініціалізація специфічних для потоку ресурсів:**
   - Використовується для налаштування ресурсів, які використовуються конкретним потоком.
   - **Приклад:**
     ```c
     case DLL_THREAD_ATTACH:
         // Код для ініціалізації ресурсів потоку
         break;
     ```

4. **Звільнення специфічних для потоку ресурсів:**
   - Використовується для звільнення ресурсів, які використовуються конкретним потоком.
   - **Приклад:**
     ```c
     case DLL_THREAD_DETACH:
         // Код для звільнення ресурсів потоку
         break;
     ```

Повний приклад використання точки входу

Створення DLL

Файл заголовка бібліотеки (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
```

```
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
// mathlib.c
#include <windows.h>
#include "mathlib.h"

// Глобальні змінні
CRITICAL_SECTION CriticalSection;

BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call, LPVOID
lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            InitializeCriticalSection(&CriticalSection);
            break;
        case DLL_THREAD_ATTACH:
            // Ініціалізація ресурсів для нових потоків (за потреби)
            break;
        case DLL_THREAD_DETACH:
            // Звільнення ресурсів для завершених потоків (за потреби)
            break;
        case DLL_PROCESS_DETACH:
            DeleteCriticalSection(&CriticalSection);
            break;
    }
    return TRUE;
}

MATHLIB_API double add(double a, double b) {
    EnterCriticalSection(&CriticalSection);
    double result = a + b;
    LeaveCriticalSection(&CriticalSection);
    return result;
}

MATHLIB_API double subtract(double a, double b) {
    EnterCriticalSection(&CriticalSection);
    double result = a - b;
    LeaveCriticalSection(&CriticalSection);
    return result;
}
```

Компіляція DLL:
```sh
gcc -shared -o mathlib.dll mathlib.c -Wl,--out-implib,mathlib.lib
```

Використання DLL у програмі

Основний файл програми (main.c):
```c
// main.c
#include <stdio.h>
#include "mathlib.h"

int main() {
    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);
    return 0;
}
```

Компіляція з неявним зв'язуванням:
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Висновок

Точка входу динамічної бібліотеки, така як `DllMain`, надає можливість виконувати специфічні ініціалізаційні та завершальні дії під час завантаження, створення потоків, завершення потоків та вивантаження бібліотеки. Раціональне використання цієї функції дозволяє ефективно керувати ресурсами та забезпечувати стабільність роботи програм, що використовують DLL.

8.Структура виконуваних файлів для Windows. Формат PE.

Структура виконуваних файлів для Windows. Формат PE

Виконувані файли у Windows використовують формат PE (Portable Executable), який підтримує як 32-бітові (PE32), так і 64-бітові (PE32+) програми. Формат PE визначає структуру файлу, яка містить інформацію про код, дані, ресурси та іншу необхідну інформацію для завантаження і виконання програми.

Основні компоненти формату PE

1. **MS-DOS Header**

2. **PE Header (Signature, COFF Header, Optional Header)**
3. **Section Headers**
4. **Sections**

1. MS-DOS Header

MS-DOS Header займає перші 64 байти файлу і містить інформацію, необхідну для запуску програми в середовищі MS-DOS. Він також містить мітку на PE Header, яка знаходиться далі в файлі.

Структура MS-DOS Header:
```c
typedef struct _IMAGE_DOS_HEADER {
    WORD e_magic;    // Магічне число (дозволяє ідентифікувати файл як DOS-EXE)
    WORD e_cblp;     // Кількість байтів на останній сторінці файлу
    WORD e_cp;       // Кількість сторінок у файлі
    WORD e_crlc;     // Кількість таблиць перенесень
    WORD e_cparhdr;  // Кількість абзаців в заголовку
    WORD e_minalloc; // Мінімальна кількість абзаців, що повинні бути виділені
    WORD e_maxalloc; // Максимальна кількість абзаців, що можуть бути виділені
    WORD e_ss;       // Початкове значення регістра SS
    WORD e_sp;       // Початкове значення регістра SP
    WORD e_csum;     // Контрольна сума
    WORD e_ip;       // Початкове значення регістра IP
    WORD e_cs;       // Початкове значення регістра CS
    WORD e_lfarlc;   // Зміщення таблиці перенесень
    WORD e_ovno;     // Номер оверлею
    WORD e_res[4];   // Зарезервовано
    WORD e_oemid;    // Ідентифікатор OEM
    WORD e_oeminfo;  // Інформація про OEM
    WORD e_res2[10]; // Зарезервовано
    LONG e_lfanew;   // Зміщення до PE заголовка
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

2. PE Header

PE Header складається з трьох частин: Signature, COFF Header і Optional Header.

Структура Signature:
```c
DWORD Signature; // Значення "PE\0\0"
```

Структура COFF Header:
```c
typedef struct _IMAGE_FILE_HEADER {
    WORD  Machine;              // Тип машини (наприклад, x86, x64)
```

```
    WORD  NumberOfSections;     // Кількість секцій у файлі
    DWORD TimeDateStamp;        // Дата і час створення файлу
    DWORD PointerToSymbolTable; // Вказівник на таблицю символів (не
використовується в PE)
    DWORD NumberOfSymbols;      // Кількість символів у таблиці (не використовується в
PE)
    WORD  SizeOfOptionalHeader; // Розмір Optional Header
    WORD  Characteristics;      // Характеристики файлу
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Структура Optional Header:
Optional Header насправді не є опціональним і містить важливу інформацію для завантаження і виконання програми.

Основні поля Optional Header:
```c
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD  Magic;                    // Ідентифікатор типу файлу (PE32 або PE32+)
    BYTE  MajorLinkerVersion;       // Версія лінкера
    BYTE  MinorLinkerVersion;       // Менша версія лінкера
    DWORD SizeOfCode;               // Розмір коду (текстовий сегмент)
    DWORD SizeOfInitializedData;    // Розмір ініціалізованих даних
    DWORD SizeOfUninitializedData;  // Розмір неініціалізованих даних (BSS)
    DWORD AddressOfEntryPoint;      // Адреса точки входу
    DWORD BaseOfCode;               // Базова адреса коду
    DWORD BaseOfData;               // Базова адреса даних (тільки для PE32)
    DWORD ImageBase;                // Базова адреса зображення в пам'яті
    DWORD SectionAlignment;         // Вирівнювання секцій в пам'яті
    DWORD FileAlignment;            // Вирівнювання секцій у файлі
    WORD  MajorOperatingSystemVersion; // Версія операційної системи
    WORD  MinorOperatingSystemVersion; // Менша версія операційної системи
    WORD  MajorImageVersion;        // Версія зображення
    WORD  MinorImageVersion;        // Менша версія зображення
    WORD  MajorSubsystemVersion;    // Версія підсистеми
    WORD  MinorSubsystemVersion;    // Менша версія підсистеми
    DWORD Win32VersionValue;        // Зарезервовано (повинно бути 0)
    DWORD SizeOfImage;              // Повний розмір зображення
    DWORD SizeOfHeaders;            // Розмір всіх заголовків
    DWORD CheckSum;                 // Контрольна сума
    WORD  Subsystem;                // Тип підсистеми (наприклад, GUI, CUI)
    WORD  DllCharacteristics;       // Характеристики DLL
    DWORD SizeOfStackReserve;       // Розмір резерву стека
    DWORD SizeOfStackCommit;        // Розмір виділеного стека
    DWORD SizeOfHeapReserve;        // Розмір резерву кучи
    DWORD SizeOfHeapCommit;         // Розмір виділеної кучи
    DWORD LoaderFlags;              // Флаги завантажувача (повинно бути 0)
    DWORD NumberOfRvaAndSizes;      // Кількість таблиць даних
```

```
    IMAGE_DATA_DIRECTORY DataDirectory[16]; // Таблиці даних
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

3. Section Headers

Section Headers слідують за Optional Header і містять інформацію про кожну секцію у файлі. Кожен заголовок секції описує одну секцію в програмі.

Структура Section Header:
```c
typedef struct _IMAGE_SECTION_HEADER {
    BYTE  Name[8];                  // Ім'я секції
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;           // Віртуальна адреса секції
    DWORD SizeOfRawData;            // Розмір секції у файлі
    DWORD PointerToRawData;         // Вказівник на дані секції у файлі
    DWORD PointerToRelocations;     // Вказівник на релокації (не використовується в PE)
    DWORD PointerToLinenumbers;     // Вказівник на номери рядків (не використовується в PE)
    WORD  NumberOfRelocations;      // Кількість релокацій (не використовується в PE)
    WORD  NumberOfLinenumbers;      // Кількість номерів рядків (не використовується в PE)
    DWORD Characteristics;          // Характеристики секції
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

4. Sections

Секції містять фактичний код, дані і ресурси програми. Основні типи секцій включають:

- **.text:** містить код програми.
- **.data:** містить ініціалізовані дані.
- **.bss:** містить неініціалізовані дані.
- **.rdata:** містить константи і рядки.
- **.idata:** містить інформацію про імпортовані функції та бібліотеки.
- **.edata:** містить інформацію про експортовані функції.

Приклад виконуваного файлу

Розглянемо приклад простого виконуваного файлу, створеного на C.

Файл main.c:
```c
#include <stdio.h>
```

```
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Компіляція:
```sh
gcc -o hello.exe main.c
```

Пояснення структури

1. **MS-DOS Header:**
   - Дозволяє запускати файл у середовищі MS-DOS (як правило, містить коротке повідомлення, що програма не може бути запущена в

 DOS).

2. **PE Header:**
   - Визначає файл як PE-файл і містить основну інформацію про файл, таку як тип машини, кількість секцій, дата і час створення тощо.

3. **Optional Header:**
   - Містить важливу інформацію про завантаження і виконання програми, таку як базова адреса зображення, адреса точки входу, вирівнювання секцій тощо.

4. **Section Headers:**
   - Описують кожну секцію у файлі, включаючи її ім'я, віртуальну адресу, розмір та інші характеристики.

5. **Sections:**
   - Містять фактичний код програми, дані і ресурси. У нашому прикладі секція `.text` міститиме код `main`, а секція `.rdata` — рядок "Hello, World!".

Висновок

Формат PE є основним форматом виконуваних файлів у Windows. Він забезпечує структуру, яка дозволяє операційній системі правильно завантажувати, розміщувати в пам'яті та виконувати програми. Розуміння цієї структури важливо для розробників програмного забезпечення, оскільки дозволяє їм ефективно працювати з виконуваними файлами, динамічними бібліотеками та іншими компонентами програм у середовищі Windows.

9.Схема процесу компонування для Windows у разі неявного зв'язування.

Схема процесу компонування для Windows у разі неявного зв'язування

Неявне зв'язування (implicit linking) - це метод зв'язування, при якому програма автоматично завантажує динамічні бібліотеки (DLL) під час свого запуску. Це дозволяє програмі використовувати функції з цих бібліотек без необхідності динамічного завантаження під час виконання. Розглянемо схему процесу компонування для Windows у разі неявного зв'язування.

Основні етапи процесу компонування

1. **Розробка вихідного коду DLL і програми:**
   - Створення файлів заголовків і реалізацій для DLL.
   - Створення вихідного коду для основної програми, що використовує DLL.

2. **Компіляція DLL:**
   - Компіляція вихідного коду DLL у файл бібліотеки `.dll` та створення імпортної бібліотеки `.lib`.

3. **Компіляція основної програми:**
   - Компіляція вихідного коду основної програми з включенням заголовкових файлів DLL.
   - Лінкування програми з імпортною бібліотекою DLL (`.lib`).

4. **Завантаження DLL під час запуску програми:**
   - Під час запуску програми операційна система автоматично завантажує необхідні DLL і вирішує всі зовнішні посилання.

Детальна схема процесу

1. Розробка вихідного коду DLL і програми

Файл заголовка DLL (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
```

```
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації DLL (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
}
```

Файл основної програми (main.c):
```c
// main.c
#include <stdio.h>
#include "mathlib.h"

int main() {
    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);
    return 0;
}
```

2. Компіляція DLL

```sh
gcc -shared -o mathlib.dll mathlib.c -Wl,--out-implib,mathlib.lib
```

- **mathlib.dll:** Динамічна бібліотека, яка містить реалізацію функцій `add` і `subtract`.
- **mathlib.lib:** Імпортна бібліотека, яка використовується під час компонування основної програми.

3. Компіляція основної програми

```sh
gcc -o main.exe main.c -L. -lmathlib
```

- **main.exe:** Виконуваний файл основної програми, який неявно зв'язується з `mathlib.dll`.

4. Завантаження DLL під час запуску програми

1. **Під час компіляції:** Лінкер використовує імпортну бібліотеку `mathlib.lib` для включення посилань на функції `add` і `subtract` в програму.
2. **Під час запуску програми:** Операційна система автоматично завантажує `mathlib.dll` в пам'ять.
3. **Резольвінг символів:** Операційна система вирішує всі посилання на функції `add` і `subtract`, забезпечуючи їх коректну роботу.

Пояснення процесу

1. **Розробка вихідного коду:** Програміст пише вихідний код для DLL (`mathlib.h` і `mathlib.c`) і основної програми (`main.c`).
2. **Компіляція DLL:** Компілятор створює динамічну бібліотеку `mathlib.dll` і імпортну бібліотеку `mathlib.lib`.
3. **Компіляція основної програми:** Компілятор використовує імпортну бібліотеку `mathlib.lib` для створення виконуваного файлу `main.exe`, який неявно зв'язується з `mathlib.dll`.
4. **Завантаження DLL:** Під час запуску програми операційна система завантажує `mathlib.dll` в пам'ять і вирішує всі посилання на функції, забезпечуючи їх коректну роботу.

Висновок

Неявне зв'язування забезпечує автоматичне завантаження і резольвінг функцій з DLL під час запуску програми. Це спрощує процес розробки і використання динамічних бібліотек, оскільки програма може використовувати функції з DLL без необхідності динамічного завантаження під час виконання. Розуміння цього процесу важливо для ефективного використання динамічних бібліотек у Windows.

10. Схема процесу компонування для Windows у разі явного зв'язування.

Схема процесу компонування для Windows у разі явного зв'язування

Явне зв'язування (explicit linking) передбачає, що програма самостійно завантажує динамічні бібліотеки (DLL) під час виконання, використовуючи функції операційної системи, такі як `LoadLibrary` і `GetProcAddress`. Це дозволяє більш гнучко керувати використанням DLL, завантажуючи їх тільки за потреби.

Основні етапи процесу компонування

1. **Розробка вихідного коду DLL і програми:**
   - Створення файлів заголовків і реалізацій для DLL.
   - Створення вихідного коду для основної програми, що використовує DLL.

2. **Компіляція DLL:**
   - Компіляція вихідного коду DLL у файл бібліотеки `.dll`.

3. **Компіляція основної програми:**
   - Компіляція вихідного коду основної програми без імпортної бібліотеки.

4. **Завантаження DLL під час виконання програми:**
   - Програма завантажує DLL динамічно під час виконання, використовуючи `LoadLibrary`.
   - Програма отримує адреси необхідних функцій за допомогою `GetProcAddress`.

Детальна схема процесу

1. Розробка вихідного коду DLL і програми

Файл заголовка DLL (mathlib.h):
```c
// mathlib.h
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API double add(double a, double b);
MATHLIB_API double subtract(double a, double b);

#endif // MATHLIB_H
```

Файл реалізації DLL (mathlib.c):
```c
// mathlib.c
#include "mathlib.h"

MATHLIB_API double add(double a, double b) {
    return a + b;
}

MATHLIB_API double subtract(double a, double b) {
    return a - b;
```

```
}
```

Файл основної програми (main.c):
```c
// main.c
#include <stdio.h>
#include <windows.h>

typedef double (*AddFunc)(double, double);
typedef double (*SubtractFunc)(double, double);

int main() {
    HMODULE hLib = LoadLibrary("mathlib.dll");
    if (hLib == NULL) {
        printf("Failed to load DLL\n");
        return 1;
    }

    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");

    if (add == NULL || subtract == NULL) {
        printf("Failed to get function address\n");
        FreeLibrary(hLib);
        return 1;
    }

    double result1 = add(3.0, 4.0);
    double result2 = subtract(7.0, 2.0);
    printf("Add: %f, Subtract: %f\n", result1, result2);

    FreeLibrary(hLib);
    return 0;
}
```

2. Компіляція DLL

```sh
gcc -shared -o mathlib.dll mathlib.c
```

- **mathlib.dll:** Динамічна бібліотека, яка містить реалізацію функцій `add` і `subtract`.

3. Компіляція основної програми

```sh
```

```
gcc -o main.exe main.c
```

- **main.exe:** Виконуваний файл основної програми, який використовує явне зв'язування для доступу до функцій `mathlib.dll`.

4. Завантаження DLL під час виконання програми

1. **Завантаження DLL:** Програма використовує функцію `LoadLibrary` для завантаження `mathlib.dll` в пам'ять під час виконання.
2. **Отримання адрес функцій:** Програма використовує функцію `GetProcAddress` для отримання адрес функцій `add` і `subtract` з завантаженої DLL.
3. **Використання функцій:** Програма викликає функції через отримані адреси.
4. **Вивантаження DLL:** Після використання програма використовує функцію `FreeLibrary`, щоб вивантажити DLL з пам'яті.

Пояснення процесу

1. **Розробка вихідного коду:** Програміст пише вихідний код для DLL (`mathlib.h` і `mathlib.c`) і основної програми (`main.c`).
2. **Компіляція DLL:** Компілятор створює динамічну бібліотеку `mathlib.dll`.
3. **Компіляція основної програми:** Компілятор створює виконуваний файл `main.exe`, який використовує явне зв'язування для доступу до функцій з `mathlib.dll`.
4. **Завантаження DLL:** Під час виконання програма завантажує `mathlib.dll` в пам'ять за допомогою `LoadLibrary`.
5. **Отримання адрес функцій:** Програма отримує адреси функцій `add` і `subtract` за допомогою `GetProcAddress`.
6. **Використання функцій:** Програма викликає функції через отримані адреси.
7. **Вивантаження DLL:** Після завершення використання функцій програма вивантажує DLL з пам'яті за допомогою `FreeLibrary`.

Висновок

Явне зв'язування забезпечує більшу гнучкість у використанні динамічних бібліотек, дозволяючи програмі самостійно завантажувати DLL під час виконання та отримувати доступ до необхідних функцій. Це особливо корисно, коли необхідно завантажувати бібліотеки динамічно за потребою або коли програма повинна працювати з різними версіями бібліотек. Розуміння цього процесу дозволяє ефективно використовувати можливості явного зв'язування у Windows.

11. Механізми передавання параметрів до процедур і функцій: за значенням; за посиланням (за адресою); за поверненим значенням; за результатом; за іменем; відкладеним обчисленням. Загальні визначення.

Механізми передавання параметрів до процедур і функцій

1. Передавання за значенням (Pass by Value)

При передаванні за значенням копія аргумента передається функції. Зміни, внесені у параметр всередині функції, не впливають на оригінальний аргумент.

Загальне визначення:
- Параметри передаються як копії значень.
- Зміни в функції не впливають на оригінальні змінні.

Приклад на мові C:
```c
#include <stdio.h>

void increment(int x) {
    x = x + 1;
}

int main() {
    int a = 5;
    increment(a);
    printf("a = %d\n", a); // a залишається 5
    return 0;
}
```

2. Передавання за посиланням (Pass by Reference)

При передаванні за посиланням функція отримує доступ до змінної через її адресу. Зміни, внесені у параметр всередині функції, впливають на оригінальний аргумент.

Загальне визначення:
- Параметри передаються як посилання (адреси).
- Зміни в функції впливають на оригінальні змінні.

Приклад на мові C:
```c
#include <stdio.h>

void increment(int *x) {
    *x = *x + 1;
}

int main() {
    int a = 5;
    increment(&a);
    printf("a = %d\n", a); // a стає 6
    return 0;
```

```
}
```

3. Передавання за поверненим значенням (Return by Value)

Функція повертає значення, яке копіюється в змінну, що викликає функцію.

Загальне визначення:
- Значення повертається функцією і копіюється у змінну, що викликає функцію.

Приклад на мові C:
```c
#include <stdio.h>

int increment(int x) {
    return x + 1;
}

int main() {
    int a = 5;
    a = increment(a);
    printf("a = %d\n", a); // a стає 6
    return 0;
}
```

4. Передавання за результатом (Pass by Result)

Цей механізм передбачає передавання параметра до функції, як у передаванні за значенням, але значення параметра копіюється назад до аргумента після завершення функції. У мові C цей механізм не підтримується прямо, але можна симулювати за допомогою вказівників.

Загальне визначення:
- Параметр копіюється у функцію, а після завершення функції значення параметра копіюється назад до аргумента.

Приклад на мові C:
```c
#include <stdio.h>

void increment(int *x) {
    *x = *x + 1;
}

int main() {
    int a = 5;
    increment(&a); // Симуляція передавання за результатом
```

```
    printf("a = %d\n", a); // a стає 6
    return 0;
}
```

5. Передавання за іменем (Pass by Name)

При передаванні за іменем аргументи не обчислюються до тих пір, поки вони не будуть використані всередині функції. У мові C цей механізм не підтримується безпосередньо, але можна використовувати макроси для симуляції.

Загальне визначення:
- Аргументи не обчислюються до тих пір, поки вони не будуть використані всередині функції.

Приклад на мові C (макрос):
```c
#include <stdio.h>

#define increment(x) (x = x + 1)

int main() {
    int a = 5;
    increment(a);
    printf("a = %d\n", a); // a стає 6
    return 0;
}
```

6. Відкладене обчислення (Lazy Evaluation)

При відкладеному обчисленні вирази не обчислюються до моменту їх використання. У мові C цей механізм не підтримується безпосередньо, але можна симулювати за допомогою функцій-замикань або макросів.

Загальне визначення:
- Вирази не обчислюються до моменту їх використання.

Приклад на мові C (макрос):
```c
#include <stdio.h>

#define lazy_increment(x) ({int _x = x; _x = _x + 1; _x;})

int main() {
    int a = 5;
    a = lazy_increment(a);
    printf("a = %d\n", a); // a стає 6
```

```
    return 0;
}
```

Висновок

Різні механізми передавання параметрів дозволяють програмістам керувати тим, як аргументи функцій обробляються та змінюються. У мові C найбільш поширеними є передавання за значенням та за посиланням. Інші механізми можуть бути реалізовані або симульовані за допомогою макросів та вказівників, надаючи додаткову гнучкість у розробці програм.

12. Способи передавання параметрів до процедур і функцій: в регістрах; в глобальних зміних; в стеку. Погодження (конвенції) для передавання параметрів у функцію через стек.

Способи передавання параметрів до процедур і функцій

1. Передавання параметрів в регістрах

Передавання параметрів у регістрах означає, що значення параметрів передаються функції через процесорні регістри. Це зазвичай швидше, ніж передавання через стек або глобальні змінні, оскільки доступ до регістрів значно швидший.

Приклад на мові C з використанням інлайн-асемблера:
```c
#include <stdio.h>

int add(int a, int b) {
    int result;
    __asm__ ("addl %%ebx, %%eax;"
             : "=a" (result)    // вивід у регістр eax
             : "a" (a), "b" (b) // введення з регістрів eax, ebx
             );
    return result;
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

2. Передавання параметрів в глобальних змінних

Передавання параметрів через глобальні змінні передбачає використання змінних, які є доступними в усьому програмному середовищі. Це зручно для зберігання даних, доступних кільком функціям, але може призводити до небажаних побічних ефектів та проблем із синхронізацією в багатопоточних середовищах.

Приклад на мові C:
```c
#include <stdio.h>

int global_a;
int global_b;

int add() {
    return global_a + global_b;
}

int main() {
    global_a = 3;
    global_b = 4;
    int result = add();
    printf("Result: %d\n", result);
    return 0;
}
```

3. Передавання параметрів через стек

Передавання параметрів через стек означає, що параметри функцій зберігаються в стеку перед викликом функції. Це дозволяє динамічно управляти параметрами і забезпечує безпеку в багатопоточних середовищах.

Приклад на мові C:
```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Погодження (конвенції) для передавання параметрів у функцію через стек

Існує кілька загальноприйнятих конвенцій для передавання параметрів через стек, які визначають порядок передавання параметрів і хто відповідає за очищення стека після виклику функції. Найбільш поширеними є конвенції **cdecl**, **stdcall** і **fastcall**.

Конвенція cdecl (C Declaration)

- Параметри передаються через стек зліва направо.
- Викликаюча функція відповідає за очищення стека після виклику функції.

Приклад на мові C:
```c
#include <stdio.h>

int __cdecl add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Конвенція stdcall (Standard Call)

- Параметри передаються через стек зліва направо.
- Викликана функція відповідає за очищення стека після виклику.

Приклад на мові C:
```c
#include <stdio.h>

int __stdcall add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Конвенція fastcall (Fast Call)

- Параметри передаються через регістри, а якщо регістрів не вистачає - через стек.

- Зазвичай перші два параметри передаються через регістри ECX і EDX.

Приклад на мові C:
```c
#include <stdio.h>

int __fastcall add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Пояснення процесу

1. **Передавання параметрів в регістрах:**
   - Параметри функції зберігаються у регістрах процесора, що забезпечує швидший доступ порівняно з іншими методами.

2. **Передавання параметрів в глобальних змінних:**
   - Параметри зберігаються у глобальних змінних, що дозволяє їм бути доступними з будь-якої частини програми, але може призвести до проблем з безпекою та синхронізацією.

3. **Передавання параметрів через стек:**
   - Параметри функції зберігаються у стеку, що дозволяє ефективно управляти пам'яттю і забезпечує захист даних у багатопоточному середовищі.

Висновок

Існують різні способи передавання параметрів до процедур і функцій, кожен з яких має свої переваги і недоліки. Вибір відповідного способу залежить від конкретних вимог програми, таких як продуктивність, безпека і зручність використання. Розуміння цих механізмів і конвенцій передавання параметрів дозволяє ефективно розробляти і оптимізувати програмне забезпечення.

13. Модель конвенції stdcall передавання параметрів до процедур і функцій.

Модель конвенції stdcall передавання параметрів до процедур і функцій

Конвенція виклику stdcall - це угода про виклик, яка використовується в програмуванні для визначення порядку передачі параметрів до функцій і процедур, а також для визначення, хто відповідає за очищення стека після виклику функції. У конвенції stdcall викликана функція відповідає за очищення стека.

Основні характеристики конвенції stdcall

1. **Порядок передачі параметрів:**
   - Параметри передаються через стек з права наліво.
   - Це означає, що перший параметр функції буде останнім у стеку, а останній параметр буде першим.

2. **Очищення стека:**
   - Відповідальність за очищення стека після виклику функції покладається на викликану функцію.

3. **Регістр для повернення значення:**
   - Значення, яке повертається функцією, передається через регістр EAX.

4. **Прізвиська функцій:**
   - В іменах функцій у об'єктних файлах часто додається префікс підкреслення і суфікс, що позначає кількість байтів, переданих через стек (наприклад, `_functionname@12`).

Приклад використання конвенції stdcall на мові C

Створення DLL з функцією, яка використовує конвенцію stdcall

Файл заголовка бібліотеки (mathlib.h):
```c
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

MATHLIB_API int __stdcall add(int a, int b);

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.c):
```c
#include "mathlib.h"

int __stdcall add(int a, int b) {
```

```
    return a + b;
}
```

Компіляція DLL:
```sh
gcc -shared -o mathlib.dll mathlib.c -Wl,--out-implib,mathlib.lib
```

Використання DLL у програмі

Основний файл програми (main.c):
```c
#include <stdio.h>
#include "mathlib.h"

int main() {
    int result = add(3, 4);
    printf("Result: %d\n", result);
    return 0;
}
```

Компіляція основної програми:
```sh
gcc -o main.exe main.c -L. -lmathlib
```

Пояснення процесу

1. **Передача параметрів:**
   - У функцію `add` передаються два параметри: `int a` і `int b`.
   - Параметри передаються через стек у порядку з права наліво: спочатку `b`, потім `a`.

2. **Виклик функції:**
   - Під час виклику `add(3, 4)`, значення 4 спочатку розміщується у стеку, потім значення 3.

3. **Очищення стека:**
   - Функція `add` виконує свою роботу і очищає стек перед поверненням управління викликаючій програмі.
   - Очищення стека включає зняття параметрів зі стека.

4. **Повернення значення:**
   - Функція `add` повертає результат через регістр EAX.

Приклад коду на асемблері для stdcall

Для розуміння того, як працює stdcall на нижчому рівні, розглянемо простий приклад на асемблері.

Файл реалізації бібліотеки (mathlib.asm):
```asm
section .text
    global _add@8

_add@8:
    ; Вхідні параметри:
    ;   [esp+4] - a
    ;   [esp+8] - b

    ; Завантажити параметри у регістри
    mov eax, [esp+4]
    mov ebx, [esp+8]

    ; Додати параметри
    add eax, ebx

    ; Повернути результат у EAX
    ret 8  ; Очистити 8 байтів зі стека (2 параметри по 4 байти)
```

Компіляція DLL:
```sh
nasm -f win32 mathlib.asm -o mathlib.obj
gcc -shared -o mathlib.dll mathlib.obj -Wl,--out-implib,mathlib.lib
```

Висновок

Конвенція stdcall широко використовується в Windows API і є стандартною для багатьох функцій бібліотек DLL. Вона визначає порядок передачі параметрів через стек і покладає відповідальність за очищення стека на викликану функцію. Розуміння цієї конвенції є важливим для розробників, які працюють з низькорівневим кодом, асемблером і DLL у Windows.

14. Стандартні типи даних Windows та їх еквіваленти в мові C.

Стандартні типи даних Windows та їх еквіваленти в мові C

Windows API використовує різноманітні типи даних, які часто мають свої еквіваленти в стандартних типах мови C. Ці типи даних визначені в заголовкових файлах Windows,

таких як `windows.h`. Знання відповідності між типами Windows і стандартними типами мови C важливо для коректного використання API Windows.

Основні типи даних Windows та їх еквіваленти в C

1. **BOOL**
   - **Windows:** 32-бітове логічне значення, яке може бути `TRUE` або `FALSE`.
   - **C:** `int`
   - **Приклад:**
     ```c
     BOOL success = TRUE; // Windows
     int success = 1;     // C (TRUE = 1)
     ```

2. **BYTE**
   - **Windows:** 8-бітове беззнакове ціле число.
   - **C:** `unsigned char`
   - **Приклад:**
     ```c
     BYTE b = 255; // Windows
     unsigned char b = 255; // C
     ```

3. **CHAR**
   - **Windows:** 8-бітове знакове ціле число.
   - **C:** `char`
   - **Приклад:**
     ```c
     CHAR c = 'A'; // Windows
     char c = 'A'; // C
     ```

4. **DWORD**
   - **Windows:** 32-бітове беззнакове ціле число.
   - **C:** `unsigned long`
   - **Приклад:**
     ```c
     DWORD d = 4294967295; // Windows
     unsigned long d = 4294967295; // C
     ```

5. **HANDLE**
   - **Windows:** Абстрактний тип, що представляє дескриптор для ресурсів Windows (файлів, потоків, процесів тощо).
   - **C:** `void*` або інший цілочисельний тип
   - **Приклад:**
     ```c
     HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE); // Windows
     ```

```c
void* h = GetStdHandle(STD_OUTPUT_HANDLE); // C
```

6. **HINSTANCE**
   - **Windows:** Дескриптор модуля програми.
   - **C:** `void*`
   - **Приклад:**
     ```c
     HINSTANCE hInstance; // Windows
     void* hInstance;     // C
     ```

7. **HWND**
   - **Windows:** Дескриптор вікна.
   - **C:** `void*`
   - **Приклад:**
     ```c
     HWND hwnd; // Windows
     void* hwnd; // C
     ```

8. **LPARAM**
   - **Windows:** 32-бітове або 64-бітове значення, яке використовується для передачі повідомлень.
   - **C:** `long` або `long long`
   - **Приклад:**
     ```c
     LPARAM lParam; // Windows
     long lParam;   // C (32-bit)
     long long lParam; // C (64-bit)
     ```

9. **LPCSTR**
   - **Windows:** Вказівник на константний строковий літерал (ANSI).
   - **C:** `const char*`
   - **Приклад:**
     ```c
     LPCSTR str = "Hello"; // Windows
     const char* str = "Hello"; // C
     ```

10. **LPCWSTR**
    - **Windows:** Вказівник на константний строковий літерал (Unicode).
    - **C:** `const wchar_t*`
    - **Приклад:**
      ```c
      LPCWSTR wstr = L"Hello"; // Windows
      const wchar_t* wstr = L"Hello"; // C
      ```

```

11. **LPVOID**
    - **Windows:** Вказівник на будь-який тип даних.
    - **C:** `void*`
    - **Приклад:**
      ```c
      LPVOID ptr; // Windows
      void* ptr;  // C
      ```

12. **UINT**
    - **Windows:** 32-бітове беззнакове ціле число.
    - **C:** `unsigned int`
    - **Приклад:**
      ```c
      UINT u = 100; // Windows
      unsigned int u = 100; // C
      ```

13. **WORD**
    - **Windows:** 16-бітове беззнакове ціле число.
    - **C:** `unsigned short`
    - **Приклад:**
      ```c
      WORD w = 65535; // Windows
      unsigned short w = 65535; // C
      ```

Загальна таблиця відповідності

Windows Type	C Type	Description
BOOL	int	32-бітове логічне значення (TRUE/FALSE)
BYTE	unsigned char	8-бітове беззнакове ціле число
CHAR	char	8-бітове знакове ціле число
DWORD	unsigned long	32-бітове беззнакове ціле число
HANDLE	void*	Дескриптор ресурсу
HINSTANCE	void*	Дескриптор модуля програми
HWND	void*	Дескриптор вікна
LPARAM	long/long long	32-бітове або 64-бітове значення
LPCSTR	const char*	Вказівник на константний строковий літерал
LPCWSTR	const wchar_t*	Вказівник на константний строковий літерал (Unicode)
LPVOID	void*	Вказівник на будь-який тип даних
UINT	unsigned int	32-бітове беззнакове ціле число
WORD	unsigned short	16-бітове беззнакове ціле число
      ```

Приклади використання стандартних типів даних Windows в C

Використання HANDLE та BOOL

```c
#include <windows.h>
#include <stdio.h>

int main() {
    HANDLE hFile = CreateFile(
        "example.txt",           // ім'я файлу
        GENERIC_WRITE,           // бажаний доступ
        0,                       // спільний доступ
        NULL,                    // захист
        CREATE_ALWAYS,           // дії створення
        FILE_ATTRIBUTE_NORMAL,   // атрибути
        NULL                     // шаблон
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Could not create file. Error: %lu\n", GetLastError());
        return 1;
    }

    BOOL result = WriteFile(
        hFile,                   // дескриптор файлу
        "Hello, World!",         // буфер даних
        13,                      // кількість байтів для запису
        NULL,                    // кількість записаних байтів
        NULL                     // асинхронне записування
    );

    if (!result) {
        printf("Could not write to file. Error: %lu\n", GetLastError());
        CloseHandle(hFile);
        return 1;
    }

    CloseHandle(hFile);
    return 0;
}
```

Використання DWORD та LPVOID

```c
#include <windows.h>
#include <stdio.h>
```

```
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    printf("Thread is running\n");
    return 0;
}

int main() {
    HANDLE hThread = CreateThread(
        NULL,                   // атрибути безпеки
        0,                      // розмір стека
        ThreadFunction,         // функція потоку
        NULL,                   // параметр потоку
        0,                      // прапори створення
        NULL                    // ідентифікатор потоку
    );

    if (hThread == NULL) {
        printf("Could not create thread. Error: %lu\n", GetLastError());
        return 1;
    }

    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);
    return 0;
}
```

Висновок

Розуміння відповідності між стандартними типами даних Windows і типами даних у мові C є важливим для коректного використання Windows API. Це дозволяє розробникам ефективно використовувати функції Windows у своїх програмах на мові C, забезпечуючи правильну обробку даних та взаємодію з системними ресурсами.

15. Процедура підготовки функцій для DLL на прикладі алгоритмічної мови (C++ чи іншої).

Процедура підготовки функцій для DLL на прикладі C++

Створення динамічної бібліотеки (DLL) включає кілька основних етапів: написання вихідного коду, оголошення та реалізація функцій, які будуть експортовані, компіляція та створення DLL. Давайте розглянемо процес створення DLL на прикладі C++.

Кроки створення DLL:

1. **Оголошення експортованих функцій у заголовковому файлі.**

2. **Реалізація функцій у файлі вихідного коду.**
3. **Компіляція та створення DLL.**
4. **Використання DLL у програмі-клієнті.**

Крок 1: Оголошення експортованих функцій у заголовковому файлі

Створіть заголовковий файл, який міститиме оголошення функцій, які будуть експортовані з DLL. Використовуйте макрос `__declspec(dllexport)` для експорту функцій та `__declspec(dllimport)` для імпорту функцій.

mathlib.h:
```cpp
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

extern "C" {
    MATHLIB_API int add(int a, int b);
    MATHLIB_API int subtract(int a, int b);
}

#endif // MATHLIB_H
```

Крок 2: Реалізація функцій у файлі вихідного коду

Створіть файл реалізації, який міститиме визначення експортованих функцій.

mathlib.cpp:
```cpp
#include "mathlib.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

Крок 3: Компіляція та створення DLL

Компілюйте заголовковий і реалізаційний файли, щоб створити DLL та імпортну бібліотеку.

Компіляція з використанням командного рядка:
```sh
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj
```

Крок 4: Використання DLL у програмі-клієнті

Створіть програму-клієнт, яка використовуватиме функції з DLL. Вона повинна включати заголовковий файл DLL і лінкуватися з імпортною бібліотекою.

main.cpp:
```cpp
#include <iostream>
#include "mathlib.h"

int main() {
    int a = 5;
    int b = 3;

    int sum = add(a, b);
    int difference = subtract(a, b);

    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;

    return 0;
}
```

Компіляція з використанням командного рядка:
```sh
cl /EHsc main.cpp mathlib.lib
```

Пояснення кроків

1. **Оголошення експортованих функцій у заголовковому файлі:**
   - `#define MATHLIB_API __declspec(dllexport)` визначає макрос для експорту функцій.
   - Оголошення функцій з використанням `extern "C"` для запобігання манглінгу імен (це робиться для забезпечення сумісності з іншими мовами програмування).

2. **Реалізація функцій у файлі вихідного коду:**
   - Функції `add` та `subtract` реалізовані в `mathlib.cpp`.

3. **Компіляція та створення DLL:**
   - Використання `cl` для компіляції вихідного коду в об'єктні файли.
   - Використання `link` для створення DLL та імпортної бібліотеки.

4. **Використання DLL у програмі-клієнті:**
   - Підключення заголовкового файлу DLL у програмі-клієнті.
   - Виклик функцій `add` та `subtract` з DLL.
   - Компіляція програми-клієнта з лінкуванням до імпортної бібліотеки DLL.

Висновок

Процедура підготовки функцій для DLL у C++ включає оголошення та реалізацію функцій, які будуть експортовані, компіляцію та створення DLL, а також використання DLL у програмі-клієнті. Цей процес забезпечує модульність та повторне використання коду в різних програмах, що робить його ефективним інструментом у розробці програмного забезпечення.

16. Компіляція функцій в бібліотеку DLL (створення DLL). Налаштування компілятора і завантажувача.

Компіляція функцій в бібліотеку DLL (створення DLL)

Створення динамічної бібліотеки (DLL) включає написання коду функцій, які будуть експортовані, налаштування компілятора для створення DLL, а також налаштування завантажувача для використання цієї бібліотеки в програмах. У цьому прикладі ми будемо використовувати мову програмування C++ та компілятор Microsoft Visual C++ (cl.exe).

Кроки створення DLL:

1. **Написання заголовкового файлу з експортованими функціями.**
2. **Реалізація функцій у файлі вихідного коду.**
3. **Компіляція та створення DLL.**
4. **Налаштування компілятора і завантажувача.**

Крок 1: Написання заголовкового файлу з експортованими функціями

mathlib.h:
```cpp
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
```

```
#endif

extern "C" {
    MATHLIB_API int add(int a, int b);
    MATHLIB_API int subtract(int a, int b);
}

#endif // MATHLIB_H
```

Крок 2: Реалізація функцій у файлі вихідного коду

mathlib.cpp:
```cpp
#include "mathlib.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

Крок 3: Компіляція та створення DLL

Для створення DLL з використанням компілятора Microsoft Visual C++ скористаємося наступними командами.

Компіляція заголовкового і реалізаційного файлів:
```sh
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
```

Створення DLL:
```sh
link /DLL /OUT:mathlib.dll mathlib.obj
```

Пояснення команд:

- **cl:** Компілятор Microsoft Visual C++.
- **/c:** Компілювати файл, але не виконувати лінкування.
- **/EHsc:** Увімкнути обробку виключень у стилі C++.
- **/DMATHLIB_EXPORTS:** Визначає макрос MATHLIB_EXPORTS, щоб експортувати функції з DLL.
- **link:** Утиліта лінкування Microsoft.

- **/DLL:** Створити динамічну бібліотеку (DLL).
- **/OUT:mathlib.dll:** Вказати вихідний файл DLL.

Крок 4: Використання DLL у програмі-клієнті

Створимо програму-клієнт, яка використовуватиме функції з DLL.

main.cpp:
```cpp
#include <iostream>
#include "mathlib.h"

int main() {
    int a = 5;
    int b = 3;

    int sum = add(a, b);
    int difference = subtract(a, b);

    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;

    return 0;
}
```

Компіляція програми-клієнта:
```sh
cl /EHsc main.cpp mathlib.lib
```

Пояснення команд:

- **main.cpp:** Вихідний файл програми-клієнта.
- **mathlib.lib:** Імпортна бібліотека, яка була створена разом з DLL і містить інформацію про експортовані функції.

Налаштування компілятора і завантажувача

1. Налаштування компілятора:

- При компіляції DLL важливо правильно налаштувати макроси для експорту та імпорту функцій (`__declspec(dllexport)` і `__declspec(dllimport)`).
- Використання команд `/DMATHLIB_EXPORTS` при компіляції DLL і відповідне включення заголовкових файлів у клієнтському коді.

2. Налаштування завантажувача:

- Для забезпечення правильного завантаження DLL під час виконання програми-клієнта, переконайтеся, що файл DLL знаходиться в одній з таких директорій:
  - Поточна директорія програми.
  - Директорія, вказана в змінній середовища PATH.
  - Директорія Windows system (наприклад, `C:\Windows\System32`).

Приклад командного файлу для автоматизації процесу

Створимо командний файл `build.bat`, який автоматизує процес компіляції та створення DLL і програми-клієнта.

build.bat:
```bat
@echo off
rem Компіляція та створення DLL
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj

rem Компіляція програми-клієнта
cl /EHsc main.cpp mathlib.lib

pause
```

Запустіть командний файл `build.bat`, щоб автоматично скомпілювати DLL та програму-клієнт.

Висновок

Процес створення DLL у C++ включає написання коду функцій, налаштування макросів для експорту та імпорту функцій, компіляцію з використанням компілятора і лінкера, а також налаштування середовища виконання для правильного завантаження DLL. Цей процес дозволяє створювати модульні та повторно використовувані компоненти, які можуть бути використані в різних програмах.

17. Використання DLL в прикладних програмах методом явного зв’язування (на прикладі мови С чи іншої).

Використання DLL в прикладних програмах методом явного зв’язування

Явне зв'язування дозволяє програмам завантажувати динамічні бібліотеки (DLL) під час виконання, використовуючи функції операційної системи, такі як `LoadLibrary` і `GetProcAddress`. Це дозволяє більш гнучко керувати використанням DLL, завантажуючи їх тільки за потреби.

Основні етапи явного зв'язування:

1. **Створення DLL з експортованими функціями.**
2. **Написання коду програми, яка буде використовувати DLL за допомогою явного зв'язування.**
3. **Компіляція DLL і програми-клієнта.**
4. **Використання функцій DLL у програмі-клієнті за допомогою явного завантаження.**

Крок 1: Створення DLL

Файл заголовка бібліотеки (mathlib.h):
```c
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

extern "C" {
    MATHLIB_API int add(int a, int b);
    MATHLIB_API int subtract(int a, int b);
}

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.cpp):
```c
#include "mathlib.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}
```

Компіляція DLL:
```sh
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj
```

Крок 2: Написання коду програми-клієнта з явним зв'язуванням

Файл програми (main.cpp):
```c
#include <iostream>
#include <windows.h>

// Визначення типів функцій
typedef int (__cdecl *AddFunc)(int, int);
typedef int (__cdecl *SubtractFunc)(int, int);

int main() {
    // Завантаження бібліотеки
    HMODULE hLib = LoadLibrary(TEXT("mathlib.dll"));
    if (hLib == NULL) {
        std::cerr << "Could not load the DLL" << std::endl;
        return 1;
    }

    // Отримання адрес функцій
    AddFunc add = (AddFunc)GetProcAddress(hLib, "add");
    SubtractFunc subtract = (SubtractFunc)GetProcAddress(hLib, "subtract");

    if (!add || !subtract) {
        std::cerr << "Could not locate the function" << std::endl;
        FreeLibrary(hLib);
        return 1;
    }

    // Виклик функцій
    int a = 5, b = 3;
    std::cout << "Add: " << add(a, b) << std::endl;
    std::cout << "Subtract: " << subtract(a, b) << std::endl;

    // Вивантаження бібліотеки
    FreeLibrary(hLib);
    return 0;
}
```

Крок 3: Компіляція програми-клієнта

Компіляція програми-клієнта:
```sh
cl /EHsc main.cpp
```

Пояснення кроків:

1. **Завантаження бібліотеки:**
   - Використовується функція `LoadLibrary`, щоб завантажити `mathlib.dll` під час виконання програми.
   - Якщо бібліотеку не вдалося завантажити, програма виводить повідомлення про помилку і завершується.

2. **Отримання адрес функцій:**
   - Використовується функція `GetProcAddress`, щоб отримати адреси функцій `add` і `subtract` з завантаженої DLL.
   - Типи функцій визначені заздалегідь за допомогою typedef.

3. **Виклик функцій:**
   - Функції `add` і `subtract` викликаються через вказівники на функції, отримані за допомогою `GetProcAddress`.

4. **Вивантаження бібліотеки:**
   - Після завершення використання функцій програма вивантажує DLL з пам'яті за допомогою `FreeLibrary`.

Приклад командного файлу для автоматизації процесу

Створимо командний файл `build.bat`, який автоматизує процес компіляції та створення DLL і програми-клієнта.

build.bat:
```bat
@echo off
rem Компіляція та створення DLL
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj

rem Компіляція програми-клієнта
cl /EHsc main.cpp

pause
```

Запустіть командний файл `build.bat`, щоб автоматично скомпілювати DLL і програму-клієнт.

Висновок

Явне зв'язування забезпечує більшу гнучкість у використанні динамічних бібліотек, дозволяючи програмі самостійно завантажувати DLL під час виконання та отримувати доступ до необхідних функцій. Це особливо корисно, коли необхідно завантажувати бібліотеки динамічно за потребою або коли програма повинна працювати з різними версіями бібліотек. Розуміння цього процесу дозволяє ефективно використовувати можливості явного зв'язування у Windows.

18. Використання DLL в прикладних програмах методом неявного зв’язування (на прикладі мови С чи іншої).

Використання DLL в прикладних програмах методом неявного зв’язування

Неявне зв'язування передбачає автоматичне завантаження DLL під час запуску програми, без явного виклику функцій завантаження бібліотек. Це досягається шляхом використання імпортних бібліотек (`.lib`), які містять посилання на експортовані функції DLL.

Основні етапи неявного зв'язування:

1. **Створення DLL з експортованими функціями.**
2. **Написання коду програми, яка буде використовувати DLL за допомогою неявного зв'язування.**
3. **Компіляція DLL і програми-клієнта з імпортною бібліотекою.**
4. **Використання функцій DLL у програмі-клієнті.**

Крок 1: Створення DLL

Файл заголовка бібліотеки (mathlib.h):
```c
#ifndef MATHLIB_H
#define MATHLIB_H

#ifdef MATHLIB_EXPORTS
#define MATHLIB_API __declspec(dllexport)
#else
#define MATHLIB_API __declspec(dllimport)
#endif

extern "C" {
    MATHLIB_API int add(int a, int b);
    MATHLIB_API int subtract(int a, int b);
}

#endif // MATHLIB_H
```

Файл реалізації бібліотеки (mathlib.cpp):
```c
#include "mathlib.h"

int add(int a, int b) {
    return a + b;
```

```
}

int subtract(int a, int b) {
    return a - b;
}
```

Компіляція DLL:
```sh
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj /IMPLIB:mathlib.lib
```

Крок 2: Написання коду програми-клієнта з неявним зв'язуванням

Файл програми (main.cpp):
```c
#include <iostream>
#include "mathlib.h"

int main() {
    int a = 5;
    int b = 3;

    int sum = add(a, b);
    int difference = subtract(a, b);

    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Difference: " << difference << std::endl;

    return 0;
}
```

Крок 3: Компіляція програми-клієнта з імпортною бібліотекою

Компіляція програми-клієнта:
```sh
cl /EHsc main.cpp mathlib.lib
```

Пояснення кроків:

1. **Оголошення експортованих функцій:**
   - В заголовковому файлі `mathlib.h` використовується макрос `__declspec(dllexport)` для експорту функцій, коли компілюється DLL, і `__declspec(dllimport)` для імпорту функцій, коли компілюється програма-клієнт.

2. **Реалізація функцій у DLL:**
   - У файлі `mathlib.cpp` реалізовані функції `add` і `subtract`.

3. **Компіляція DLL:**
   - Компіляція вихідного коду DLL у об'єктний файл.
   - Лінкування об'єктного файлу в DLL та створення імпортної бібліотеки (`mathlib.lib`).

4. **Написання коду програми-клієнта:**
   - Програма-клієнт включає заголовковий файл `mathlib.h` і використовує функції `add` і `subtract`.

5. **Компіляція програми-клієнта:**
   - Програма-клієнт компілюється з імпортною бібліотекою `mathlib.lib`.

Приклад командного файлу для автоматизації процесу

Створимо командний файл `build.bat`, який автоматизує процес компіляції та створення DLL і програми-клієнта.

build.bat:
```bat
@echo off
rem Компіляція та створення DLL
cl /c /EHsc /DMATHLIB_EXPORTS mathlib.cpp
link /DLL /OUT:mathlib.dll mathlib.obj /IMPLIB:mathlib.lib

rem Компіляція програми-клієнта
cl /EHsc main.cpp mathlib.lib

pause
```

Запустіть командний файл `build.bat`, щоб автоматично скомпілювати DLL і програму-клієнт.

Пояснення процесу неявного зв'язування:

1. **Заголовковий файл (mathlib.h):**
   - Вміщує оголошення функцій, які будуть експортуватися з DLL.
   - Використовує макрос `__declspec(dllexport)` для експорту функцій, коли компілюється DLL, і `__declspec(dllimport)` для імпорту функцій, коли компілюється програма-клієнт.

2. **Реалізаційний файл (mathlib.cpp):**
   - Вміщує реалізації функцій, які будуть експортуватися з DLL.

3. **Компіляція DLL:**

  - Компіляція вихідного коду DLL у об'єктний файл з використанням ключа `/DMATHLIB_EXPORTS`.
  - Лінкування об'єктного файлу в DLL та створення імпортної бібліотеки (`mathlib.lib`) з використанням ключа `/IMPLIB`.

4. **Програма-клієнт (main.cpp):**
  - Використовує функції з DLL шляхом включення заголовкового файлу `mathlib.h`.

5. **Компіляція програми-клієнта:**
  - Компіляція програми-клієнта з використанням імпортної бібліотеки `mathlib.lib`.

Висновок

Неявне зв'язування забезпечує автоматичне завантаження DLL під час запуску програми, що спрощує використання бібліотек і робить код більш читабельним і зрозумілим. Цей метод підходить для випадків, коли програма має стабільні залежності від бібліотек, і немає необхідності динамічно змінювати або оновлювати бібліотеки під час виконання.

Список питань до розділу "Графічні редактори"

1.Піксел. Роздільна здатність екрана. Палітра кольорів. Принцип малювання на екрані.

Піксел. Роздільна здатність екрана. Палітра кольорів. Принцип малювання на екрані.

Піксел
Піксел (або точка) — це найменша одиниця зображення на екрані дисплея, яка може бути незалежно змінена. Пікселі складаються в сітку, яка формує зображення.

Структура даних для пікселя:
```c
typedef struct {
    unsigned char r; // Червоний канал
    unsigned char g; // Зелений канал
    unsigned char b; // Синій канал
} Pixel;
```

Роздільна здатність екрана
Роздільна здатність екрана визначається кількістю пікселів по горизонталі та вертикалі. Наприклад, роздільна здатність 1920x1080 означає, що на екрані є 1920 пікселів по горизонталі та 1080 пікселів по вертикалі.

Структура даних для роздільної здатності:
```c
typedef struct {
    int width;  // Ширина в пікселях
    int height; // Висота в пікселях
} Resolution;
```

Палітра кольорів
Палітра кольорів визначає набір кольорів, які можуть бути використані для зображення. Кожен піксел може бути представлений одним із кольорів палітри.

Структура даних для кольору:
```c
typedef struct {
    unsigned char r; // Червоний канал
    unsigned char g; // Зелений канал
    unsigned char b; // Синій канал
} Color;
```

Принцип малювання на екрані

Малювання на екрані відбувається шляхом зміни значень пікселів у графічній пам'яті. Для цього використовуються різні графічні бібліотеки та інтерфейси, такі як OpenGL, DirectX, або GDI в Windows.

Приклад малювання пікселя на екрані з використанням Windows GDI:

```c
#include <windows.h>

// Функція для малювання пікселя на вказаній позиції з вказаним кольором
void DrawPixel(HDC hdc, int x, int y, COLORREF color) {
    SetPixel(hdc, x, y, color);
}

int main() {
    // Отримуємо контекст пристрою для вікна консольного додатку
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);

    // Малюємо червоний піксель на позиції (100, 100)
    DrawPixel(hdc, 100, 100, RGB(255, 0, 0));

    // Звільняємо контекст пристрою
    ReleaseDC(hwnd, hdc);
```

```
    return 0;
}
```

Пояснення

1. **Структура даних для пікселя:**
   - Структура `Pixel` містить три канали (червоний, зелений і синій), кожен з яких представлений одним байтом (значення від 0 до 255).

2. **Структура даних для роздільної здатності:**
   - Структура `Resolution` визначає роздільну здатність екрана, зберігаючи ширину і висоту в пікселях.

3. **Структура даних для кольору:**
   - Структура `Color` аналогічна структурі `Pixel`, але може бути використана окремо для визначення кольорів у палітрі.

4. **Принцип малювання на екрані:**
   - Для малювання на екрані в Windows використовується функція `SetPixel`, яка встановлює колір пікселя на заданій координаті. Контекст пристрою (HDC) отримується для вікна консольного додатку за допомогою функції `GetDC`.

Висновок

Піксел — це базова одиниця зображення на екрані, а роздільна здатність визначає кількість таких пікселів по горизонталі і вертикалі. Палітра кольорів задає можливі кольори для пікселів. Принцип малювання на екрані полягає в зміні значень пікселів у графічній пам'яті, що може бути реалізовано за допомогою різних графічних бібліотек та інтерфейсів.

2.Графічні примітиви. Графічні бібліотеки. Принципи використання.

Графічні примітиви. Графічні бібліотеки. Принципи використання.

Графічні примітиви

Графічні примітиви — це основні елементи, з яких складаються складні графічні зображення. Основні графічні примітиви включають:

1. **Точки (Pixels)**
2. **Лінії (Lines)**
3. **Кола та еліпси (Circles and Ellipses)**
4. **Прямокутники (Rectangles)**
5. **Полігони (Polygons)**

6. **Текст (Text)**

Структури даних для графічних примітивів:

```c
// Точка
typedef struct {
    int x;
    int y;
} Point;

// Лінія
typedef struct {
    Point start;
    Point end;
} Line;

// Коло
typedef struct {
    Point center;
    int radius;
} Circle;

// Прямокутник
typedef struct {
    Point topLeft;
    Point bottomRight;
} Rectangle;

// Полігон
typedef struct {
    Point* points;
    int numPoints;
} Polygon;
```

Графічні бібліотеки

Графічні бібліотеки надають інтерфейси та функції для малювання графічних примітивів і створення складних зображень. Найпопулярніші графічні бібліотеки включають:

1. **OpenGL**: Кросплатформена графічна бібліотека для 2D і 3D графіки.
2. **DirectX**: Набір API від Microsoft для роботи з мультимедіа, особливо для ігор.
3. **GDI (Graphics Device Interface)**: Частина Windows API для роботи з графікою і текстом.
4. **SDL (Simple DirectMedia Layer)**: Кросплатформена бібліотека для роботи з графікою, звуком і вводом.

Принципи використання графічних бібліотек

Графічні бібліотеки надають програмістам засоби для малювання графічних примітивів на екрані. Використання графічних бібліотек включає кілька основних кроків:

1. **Ініціалізація бібліотеки:**
   - Створення і налаштування контексту для малювання.
   - Ініціалізація необхідних ресурсів.

2. **Малювання графічних примітивів:**
   - Використання функцій бібліотеки для малювання точок, ліній, кіл, прямокутників та інших примітивів.

3. **Оновлення екрану:**
   - Оновлення вікна або області екрану для відображення змін.

4. **Звільнення ресурсів:**
   - Звільнення виділених ресурсів і завершення роботи бібліотеки.

Приклади використання графічних бібліотек

Приклад використання GDI для малювання лінії в Windows:

```c
#include <windows.h>

// Функція для малювання лінії
void DrawLine(HDC hdc, int x1, int y1, int x2, int y2) {
    MoveToEx(hdc, x1, y1, NULL);
    LineTo(hdc, x2, y2);
}

int main() {
    // Отримуємо контекст пристрою для вікна консольного додатку
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);

    // Малюємо лінію з координатами (50, 50) до (200, 200)
    DrawLine(hdc, 50, 50, 200, 200);

    // Звільняємо контекст пристрою
    ReleaseDC(hwnd, hdc);

    return 0;
}
```

Приклад використання OpenGL для малювання трикутника:

```c
#include <GL/glut.h>

// Функція для малювання трикутника
void display() {
   glClear(GL_COLOR_BUFFER_BIT);

   glBegin(GL_TRIANGLES);
   glVertex2f(-0.5f, -0.5f);
   glVertex2f( 0.5f, -0.5f);
   glVertex2f( 0.0f,  0.5f);
   glEnd();

   glFlush();
}

int main(int argc, char** argv) {
   glutInit(&argc, argv);
   glutCreateWindow("OpenGL Example");
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Пояснення

1. **Графічні примітиви:**
   - Точки, лінії, кола, прямокутники та полігони є основними елементами, з яких складаються складні графічні зображення.
   - Структури даних для графічних примітивів включають координати та інші параметри, необхідні для їх опису.

2. **Графічні бібліотеки:**
   - OpenGL, DirectX, GDI та SDL є популярними графічними бібліотеками, які надають функції для малювання графічних примітивів.
   - Кожна бібліотека має свої особливості та сфери застосування.

3. **Принципи використання:**
   - Ініціалізація бібліотеки включає створення контексту для малювання та налаштування необхідних ресурсів.
   - Малювання графічних примітивів здійснюється за допомогою функцій бібліотеки.
   - Оновлення екрану потрібно для відображення змін.
   - Після завершення роботи бібліотеки ресурси мають бути звільнені.

Висновок

Графічні примітиви є основними будівельними блоками для створення складних графічних зображень. Графічні бібліотеки надають засоби для малювання цих примітивів та створення графічних інтерфейсів. Використання бібліотек включає ініціалізацію, малювання, оновлення екрану та звільнення ресурсів. Розуміння цих принципів дозволяє створювати ефективні графічні додатки.

3.Графічний інтерфейс редактора з користувачем. Клавіатурний інтерфейс редактора з користувачем.

Графічний інтерфейс редактора з користувачем. Клавіатурний інтерфейс редактора з користувачем (Графічний редактор)

Графічний інтерфейс редактора з користувачем

Графічний інтерфейс користувача (GUI) для графічного редактора забезпечує візуальну взаємодію між користувачем і програмою. Він включає в себе різноманітні елементи управління, такі як меню, кнопки, панелі інструментів, палітри кольорів та робочі області для малювання.

Основні компоненти GUI для графічного редактора:

1. **Головне меню:** Містить пункти меню для відкриття, збереження файлів, налаштувань, допомоги тощо.
2. **Панель інструментів:** Забезпечує швидкий доступ до основних інструментів малювання, таких як пензель, гумка, заливка, лінія, форма.
3. **Робоча область:** Простір для малювання, де користувач може створювати і редагувати графічні зображення.
4. **Палітра кольорів:** Набір кольорів для вибору кольору малювання.
5. **Статусний рядок:** Відображає інформацію про поточний інструмент, координати курсора, масштаби тощо.

Приклад реалізації GUI на мові C++ з використанням бібліотеки Qt:

main.cpp:
```cpp
#include <QApplication>
#include "MainWindow.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MainWindow window;
    window.show();
    return app.exec();
}
```

```

**MainWindow.h:**
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void openFile();
    void saveFile();
    void setPenColor();
    void setBrushColor();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QToolBar>
#include <QStatusBar>
#include <QAction>
#include <QColorDialog>
#include <QFileDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent) {
    createMenus();
    createToolBars();
    createStatusBar();
}

MainWindow::~MainWindow() {}
```
```

```
void MainWindow::createMenus() {
    QMenu *fileMenu = menuBar()->addMenu(tr("&File"));
    QAction *openAct = fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    QAction *saveAct = fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    QMenu *editMenu = menuBar()->addMenu(tr("&Edit"));
    QAction *penColorAct = editMenu->addAction(tr("&Pen Color..."), this,
&MainWindow::setPenColor);
    QAction *brushColorAct = editMenu->addAction(tr("&Brush Color..."), this,
&MainWindow::setBrushColor);
}

void MainWindow::createToolBars() {
    QToolBar *fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    QToolBar *editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Завантаження файлу
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Збереження файлу
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        // Встановлення кольору пензля
    }
```

```
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        // Встановлення кольору заливки
    }
}
```

Клавіатурний інтерфейс редактора з користувачем

Клавіатурний інтерфейс дозволяє користувачам виконувати дії за допомогою комбінацій клавіш, що може прискорити роботу з графічним редактором.

Приклад реалізації клавіатурного інтерфейсу в графічному редакторі з використанням Qt:

MainWindow.cpp (додати обробку подій клавіатури):
```cpp
#include <QKeyEvent>

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            if (event->modifiers() & Qt::ControlModifier) {
                openFile();
            }
            break;
        case Qt::Key_S:
            if (event->modifiers() & Qt::ControlModifier) {
                saveFile();
            }
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}
```

Пояснення

1. **Графічний інтерфейс:**
   - Головне меню забезпечує доступ до основних функцій, таких як відкриття та збереження файлів.
   - Панель інструментів надає швидкий доступ до інструментів для малювання та налаштувань.
   - Робоча область дозволяє користувачу створювати та редагувати графічні зображення.
   - Палітра кольорів надає можливість вибору кольору для малювання.
   - Статусний рядок відображає інформацію про поточний стан редактора.

2. **Клавіатурний інтерфейс:**
   - Обробка клавіатурних подій дозволяє користувачу виконувати дії за допомогою клавіш або комбінацій клавіш, що може значно прискорити роботу з редактором.

Висновок

Графічний інтерфейс користувача для графічного редактора включає різноманітні елементи управління, що дозволяють ефективно взаємодіяти з програмою. Клавіатурний інтерфейс забезпечує додаткові можливості для швидкого виконання команд. Розуміння принципів створення таких інтерфейсів дозволяє розробляти інтуїтивно зрозумілі та функціональні графічні редактори.

4.Прокручування поля, багатовіконність редактора, коректне припинення роботи.

Прокручування поля, багатовіконність редактора, коректне припинення роботи

Прокручування поля

Прокручування поля у графічному редакторі дозволяє користувачам переміщуватися по великому зображенню, яке не вміщується повністю на екрані. Це зазвичай реалізується за допомогою скролбарів (прокручувальних панелей).

Приклад реалізації прокручування поля з використанням Qt:

main.cpp:
```cpp
#include <QApplication>
#include <QScrollArea>
#include "MainWindow.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MainWindow *window = new MainWindow();

    // Створення скролбарів для робочої області
```

```
    QScrollArea *scrollArea = new QScrollArea();
    scrollArea->setWidget(window->getDrawingArea());
    scrollArea->setWidgetResizable(true);

    window->setCentralWidget(scrollArea);
    window->show();
    return app.exec();
}
```

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    QWidget *drawingArea;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QToolBar>
#include <QStatusBar>
#include <QAction>
#include <QScrollArea>
#include <QColorDialog>
#include <QFileDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent) {
```

```
    createMenus();
    createToolBars();
    createStatusBar();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    QMenu *fileMenu = menuBar()->addMenu(tr("&File"));
    QAction *openAct = fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    QAction *saveAct = fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    QMenu *editMenu = menuBar()->addMenu(tr("&Edit"));
    QAction *penColorAct = editMenu->addAction(tr("&Pen Color..."), this,
&MainWindow::setPenColor);
    QAction *brushColorAct = editMenu->addAction(tr("&Brush Color..."), this,
&MainWindow::setBrushColor);
}

void MainWindow::createToolBars() {
    QToolBar *fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    QToolBar *editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Завантаження файлу
    }
}
```

```
void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Збереження файлу
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        // Встановлення кольору пензля
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        // Встановлення кольору заливки
    }
}
```

Багатовіконність редактора

Багатовіконність редактора дозволяє користувачам відкривати декілька документів у різних вікнах або вкладках. Це забезпечує можливість роботи з декількома зображеннями одночасно.

Приклад реалізації багатовіконності з використанням вкладок у Qt:

main.cpp:
```cpp
#include <QApplication>
#include <QTabWidget>
#include "MainWindow.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTabWidget tabWidget;

    MainWindow *window1 = new MainWindow();
    MainWindow *window2 = new MainWindow();

    tabWidget.addTab(window1, "Document 1");
    tabWidget.addTab(window2, "Document 2");

    tabWidget.show();
```

```
    return app.exec();
}
```

MainWindow.h і MainWindow.cpp залишаються без змін.

Коректне припинення роботи

Коректне припинення роботи редактора включає збереження змін, закриття всіх відкритих файлів і звільнення ресурсів.

Приклад реалізації коректного припинення роботи в Qt:

MainWindow.h (додати метод для обробки закриття):
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QCloseEvent>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

protected:
    void closeEvent(QCloseEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void setPenColor();
    void setBrushColor();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    QWidget *drawingArea;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp (додати реалізацію обробки закриття):

```cpp
#include "MainWindow.h"
#include <QMessageBox>

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}
```

Пояснення

1. **Прокручування поля:**
   - Використання `QScrollArea` для забезпечення можливості прокручування робочої області, що дозволяє працювати з великими зображеннями.

2. **Багатовіконність редактора:**
   - Використання `QTabWidget` для створення вкладок, що дозволяють відкривати кілька документів у різних вкладках одного вікна.

3. **Коректне припинення роботи:**
   - Реалізація обробки події закриття вікна (`closeEvent`), яка запитує у користувача, чи хоче він зберегти зміни перед виходом, і в залежності від вибору користувача або зберігає файл, або закриває програму, або скасовує закриття.

Висновок

Реалізація функціональності прокручування, багатовіконності та коректного припинення роботи є важливими аспектами розробки графічного редактора. Вони забезпечують зручність використання, ефективне управління великими зображеннями та можливість роботи з кількома документами одночасно, а також забезпечують збереження роботи користувача при закритті програми.

5.Сервісні можливості редактора: логічні групи команд; розтягання, стиснення, повороти; масштабування малюнка; контекстні і випливаючі меню; підказки про події на екрані; довідкова і навчальна підсистема редактора.

Сервісні можливості редактора

Графічний редактор може надавати безліч сервісних можливостей для покращення роботи користувача. Це включає логічні групи команд, розтягання, стиснення, повороти, масштабування малюнка, контекстні і випливаючі меню, підказки про події на екрані, довідкову і навчальну підсистему.

Логічні групи команд

Логічні групи команд організовують функціональність редактора в групи, що спрощує доступ до них. Наприклад, групи команд для малювання, редагування, перегляду, налаштувань тощо.

Приклад створення логічних груп команд з використанням меню і панелей інструментів в Qt:

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QMenu>
#include <QToolBar>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();

private:
    void createMenus();
    void createToolBars();
```

```
    void createStatusBar();
    void createContextMenu();

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
}

MainWindow::~MainWindow() {}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
```

```
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::ActionsContextMenu);
    addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    addAction(tr("Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Завантаження файлу
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Збереження файлу
    }
}

void MainWindow::rotateImage() {
    // Реалізація повороту зображення
}

void MainWindow::scaleImage() {
    // Реалізація масштабування зображення
}
```

Розтягання, стиснення, повороти

Ці функції дозволяють користувачам маніпулювати зображенням.

Приклад реалізації функцій розтягання, стиснення та повороту:

```cpp
#include <QImage>
#include <QPainter>

void MainWindow::rotateImage() {
```

```
    QImage image; // Завантажене зображення
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    // Оновлення відображення зображення
}

void MainWindow::scaleImage() {
    QImage image; // Завантажене зображення
    image = image.scaled(image.width() * 2, image.height() * 2); // Масштабування у 2 рази
    // Оновлення відображення зображення
}
```

Масштабування малюнка

Масштабування дозволяє збільшувати або зменшувати зображення для більш детального перегляду.

Приклад реалізації масштабування:

```cpp
void MainWindow::scaleImage() {
    QImage image; // Завантажене зображення
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        // Оновлення відображення зображення
    }
}
```

Контекстні і випливаючі меню

Контекстні меню надають швидкий доступ до функцій, що часто використовуються, при натисканні правою кнопкою миші.

Приклад створення контекстного меню:

```cpp
void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}
```

```
void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.exec(mapToGlobal(pos));
}
```

Підказки про події на екрані

Підказки надають додаткову інформацію про елементи інтерфейсу або поточні дії.

Приклад створення підказок:

```cpp
void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    QAction *openAct = fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    openAct->setToolTip(tr("Open an existing file"));

    QAction *saveAct = fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);
    saveAct->setToolTip(tr("Save the current file"));

    editToolBar = addToolBar(tr("Edit"));
    QAction *rotateAct = editToolBar->addAction(tr("Rotate"), this,
&MainWindow::rotateImage);
    rotateAct->setToolTip(tr("Rotate the image"));

    QAction *scaleAct = editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    scaleAct->setToolTip(tr("Scale the image"));
}
```

Довідкова і навчальна підсистема редактора

Довідкова система надає інформацію про використання редактора і його функції.

Приклад створення довідкової системи:

```cpp
void MainWindow::createMenus() {
    QMenu *helpMenu = menuBar()->addMenu(tr("&Help"));
    QAction *aboutAct = helpMenu->addAction(tr("&About"), this,
&MainWindow::showAboutDialog);
    QAction *helpAct = helpMenu->addAction(tr("&Help"), this, &MainWindow::showHelp);
}

void MainWindow::showAboutDialog() {
```

```
    QMessageBox::about(this, tr("About Graphics Editor"), tr("This is a graphics editor."));
}

void MainWindow::showHelp() {
    QMessageBox::information(this, tr("Help"), tr("This is the help system for the graphics editor."));
}
```

Пояснення

1. **Логічні групи команд:** Організовують функціональність у групи, що спрощує доступ до різних функцій редактора.
2. **Розтягання, стиснення, повороти:** Дозволяють користувачам маніпулювати зображенням за допомогою відповідних функцій.
3. **Масштабування малюнка:** Дозволяє користувачам збільшувати або зменшувати зображення для більш детального перегляду.
4. **Контекстні і випливаючі меню:** Надають швидкий доступ до часто використовуваних функцій.
5. **Підказки про події на екрані:** Надають додаткову інформацію

 про елементи інтерфейсу або поточні дії.
6. **Довідкова і навчальна підсистема:** Надає інформацію про використання редактора і його функції, що допомагає користувачам краще зрозуміти, як працювати з програмою.

Висновок

Сервісні можливості графічного редактора значно покращують зручність і ефективність роботи користувачів. Логічні групи команд, функції розтягання, стиснення, повороту і масштабування, контекстні меню, підказки, а також довідкова і навчальна підсистема забезпечують повноцінний і інтуїтивно зрозумілий інтерфейс, що дозволяє користувачам легко виконувати різноманітні завдання.

6.Графічний програмний інструментарій: перо, пензель, шрифт. Загальні характеристики.

Графічний програмний інструментарій: перо, пензель, шрифт. Загальні характеристики

Перо (Pen)

Перо використовується для малювання контурів, таких як лінії, прямокутники та інші форми. Воно визначає колір і товщину лінії.

Приклад налаштування пера в Qt:

```cpp
#include <QPainter>
#include <QPen>

// Налаштування пера
QPen pen;
pen.setColor(Qt::black);
pen.setWidth(2);
```

Пензель (Brush)

Пензель використовується для заповнення областей, таких як заливка прямокутників, еліпсів та інших фігур. Він визначає колір і стиль заповнення.

Приклад налаштування пензля в Qt:
```cpp
#include <QBrush>

// Налаштування пензля
QBrush brush;
brush.setColor(Qt::blue);
brush.setStyle(Qt::SolidPattern);
```

Шрифт (Font)

Шрифт використовується для відображення тексту. Він визначає тип, розмір і стиль шрифту.

Приклад налаштування шрифту в Qt:
```cpp
#include <QFont>

// Налаштування шрифту
QFont font;
font.setFamily("Arial");
font.setPointSize(12);
font.setBold(true);
```

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
```

```
#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
```

```
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Завантаження файлу
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("Images
(*.png *.xpm *.jpg)"));
    if (!fileName.isEmpty()) {
        // Збереження файлу
    }
}

void MainWindow::rotateImage() {
    QImage image; // Завантажене зображення
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    // Оновлення відображення зображення
}

void MainWindow::scaleImage() {
    QImage image; // Завантажене зображення
    bool ok;
```

```
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        // Оновлення відображення зображення
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
        // Оновлення інструменту пера
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
        // Оновлення інструменту пензля
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
```

```
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

Пояснення

1. **Перо (Pen):**
   - Визначає колір і товщину ліній.
   - Використовується для малювання контурів і ліній.

2. **Пензель (Brush):**
   - Визначає колір і стиль заповнення.
   - Використовується для заповнення областей і заливки фігур.

3. **Шрифт (Font):**
   - Визначає тип, розмір і стиль шрифту.
   - Використовується для відображення тексту.

Висновок

Додавання графічного програмного інструментарію, такого як перо, пензель і шрифт, до графічного редактора значно покращує можливості малювання і редагування. Інтеграція цих інструментів в інтерфейс користувача забезпечує більш гнучкий та функціональний редактор, що дозволяє користувачам створювати складні графічні зображення та працювати з

7.Операції читання/запису для графічних зображень. Формати збереження у файлах.

Операції читання/запису для графічних зображень. Формати збереження у файлах

Графічний редактор повинен підтримувати операції читання і запису графічних зображень у різних форматах файлів. Найпоширеніші формати для збереження графічних зображень включають PNG, JPEG, BMP, і GIF.

Операції читання/запису зображень

Приклад реалізації операцій читання і запису зображень з використанням Qt:

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
```

```
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
};
```

```
#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
```

```
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    QImage loadedImage;
    if (!loadedImage.load(fileName)) {
        QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
        return false;
    }
    image = loadedImage;
    update();
```

```
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (!image.save(fileName)) {
        QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
        return false;
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);
```

```
    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

Пояснення

1. **Операції читання зображень:**
   - Метод `openFile()` відкриває діалогове вікно для вибору файлу зображення.
   - Метод `loadImage()` завантажує зображення з файлу в об'єкт `QImage`.

2. **Операції запису зображень:**
   - Метод `saveFile()` відкриває діалогове вікно для вибору місця збереження файлу зображення.
   - Метод `saveImage()` зберігає зображення в об'єкт `QImage` у файл.

3. **Формати збереження у файлах:**
   - Підтримуються формати PNG, JPEG, BMP, GIF, які можна вибрати у діалогових вікнах для відкриття та збереження файлів.

Висновок

Додавання операцій читання і запису зображень до графічного редактора забезпечує можливість завантаження і збереження графічних файлів у різних форматах. Це дозволяє користувачам легко імпортувати та експортувати зображення, з якими вони

працюють, що робить графічний редактор більш функціональним і зручним у використанні.

8.Растрові і векторні методи малювання. Зберігання растрових і векторних зображень. Формати растрові і векторні.

Растрові і векторні методи малювання. Зберігання растрових і векторних зображень. Формати растрові і векторні

Графічний редактор може використовувати як растрові, так і векторні методи малювання для створення зображень. Растрові зображення складаються з пікселів, тоді як векторні зображення складаються з примітивів, таких як лінії, криві та форми, що визначаються математичними формулами.

Растрові методи малювання

Растрові зображення представляють собою матрицю пікселів, де кожен піксел має певний колір.

Структура для зберігання растрового зображення:
```c
typedef struct {
    int width;
    int height;
    Pixel *pixels;
} RasterImage;
```

Приклад коду для малювання растрового зображення:

```cpp
#include <QImage>
#include <QPainter>

// Функція для малювання растрового зображення
void MainWindow::drawRasterImage(QPainter &painter) {
    QImage image("path/to/image.png");
    painter.drawImage(0, 0, image);
}
```

Векторні методи малювання

Векторні зображення складаються з примітивів, таких як лінії, криві та форми, які визначаються математичними формулами.

Структура для зберігання векторного примітиву:
```c
typedef struct {
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
} VectorImage;
```

Приклад коду для малювання векторного зображення:

```cpp
#include <QPainter>
#include <QPolygon>

// Функція для малювання векторного зображення
void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінію
    painter.drawLine(0, 0, 100, 100);

    // Малюємо прямокутник
    painter.drawRect(50, 50, 150, 100);

    // Малюємо полігон
    QPolygon polygon;
    polygon << QPoint(200, 200) << QPoint(250, 250) << QPoint(300, 200);
    painter.drawPolygon(polygon);
}
```

Зберігання растрових і векторних зображень

Растрові зображення зберігаються у форматах, таких як PNG, JPEG, BMP, GIF. Векторні зображення зберігаються у форматах, таких як SVG, EPS, PDF.

Структура для зберігання растрового зображення:
```c
typedef struct {
    int width;
    int height;
    Pixel *pixels;
} RasterImage;
```

Структура для зберігання векторного зображення:
```c
typedef struct {
```

```
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
} VectorImage;
```

Формати збереження

Растрові формати:
- PNG (Portable Network Graphics)
- JPEG (Joint Photographic Experts Group)
- BMP (Bitmap)
- GIF (Graphics Interchange Format)

Векторні формати:
- SVG (Scalable Vector Graphics)
- EPS (Encapsulated PostScript)
- PDF (Portable Document Format)

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
```

```
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
```

```
#include <QColorDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
```

```
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return

 true;
}
```

```
bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
```

```
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
```

```
    }
}
```

Пояснення

1. **Растрові методи малювання:**
   - Растрові зображення складаються з пікселів і зберігаються в об'єкті `QImage`.
   - Метод `drawRasterImage` малює растрове зображення з файлу на робочій області.

2. **Векторні методи малювання:**
   - Векторні зображення складаються з примітивів, таких як лінії, прямокутники і полігони.
   - Метод `drawVectorImage` малює векторні примітиви на робочій області.

3. **Зберігання растрових і векторних зображень:**
   - Растрові зображення зберігаються у форматах PNG, JPEG, BMP, GIF.
   - Векторні зображення зберігаються у форматах SVG, EPS, PDF.

4. **Формати збереження:**
   - Додано підтримку збереження растрових і векторних зображень у різних форматах.

Висновок

Додавання підтримки як растрових, так і векторних методів малювання значно розширює можливості графічного редактора. Це дозволяє користувачам працювати з різними типами зображень і зберігати їх у різних форматах, забезпечуючи високу гнучкість і функціональність програми.

9.Принципи будови графічних редакторів на основі растрових і на основі векторних зображень. Характеристика особливостей растрового і векторного малювання, переваги і недоліки кожного методу.

Принципи будови графічних редакторів на основі растрових і на основі векторних зображень. Характеристика особливостей растрового і векторного малювання, переваги і недоліки кожного методу.

Графічні редактори можуть бути побудовані на основі як растрових, так і векторних зображень. Кожен метод має свої особливості, переваги і недоліки, які визначають сфери їх застосування.

Растрові зображення

Особливості:

- Растрові зображення складаються з пікселів, кожен з яких має свій колір.
- Зберігаються у форматах, таких як PNG, JPEG, BMP, GIF.
- Ідеально підходять для фотографій та зображень з багатою палітрою кольорів.

Переваги:
- Висока деталізація.
- Підтримка широкого спектру кольорів.
- Простота маніпуляцій з окремими пікселями.

Недоліки:
- Великі розміри файлів при високій роздільній здатності.
- Втрата якості при масштабуванні.
- Неможливість редагування окремих елементів без впливу на інші.

Векторні зображення

Особливості:
- Векторні зображення складаються з примітивів, таких як лінії, криві та форми, які визначаються математичними формулами.
- Зберігаються у форматах, таких як SVG, EPS, PDF.
- Ідеально підходять для логотипів, іконок, схем та інших графічних об'єктів, які потребують масштабування без втрати якості.

Переваги:
- Можливість масштабування без втрати якості.
- Малі розміри файлів для простих зображень.
- Легкість редагування окремих елементів.

Недоліки:
- Обмежена деталізація для складних зображень.
- Вимоги до високої обчислювальної потужності при відображенні складних векторних зображень.
- Менш реалістичний вигляд порівняно з растровими зображеннями.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
```

```
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
```

```
    QVector<QPolygon> polygons;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);
```

```
    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
```

```
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();


    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
```

```
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);
}
```

```
void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Принципи будови графічних редакторів:**
   - Графічні редактори на основі растрових зображень зберігають дані у вигляді пікселів. Це підходить для фотографій та зображень з багатою палітрою кольорів.
   - Графічні редактори на основі векторних зображень зберігають дані у вигляді примітивів, таких як лінії, криві та форми. Це підходить для логотипів, іконок, схем та інших графічних об'єктів, які потребують масштабування без втрати якості.

2. **Особливості растрового і векторного малювання:**
   - Растрові зображення складаються з пікселів і зберігаються в об'єкті `QImage`.
   - Векторні зображення складаються з примітивів і зберігаються в об'єкті `QVector` для ліній, прямокутників та полігонів.

3. **Переваги і недоліки кожного методу:**
   - Растрові зображення мають високу деталізацію і підтримують широкий спектр кольорів, але вони мають великі розміри файлів і втрачають якість при масштабуванні.
   - Векторні зображення можуть масштабуватися без втрати якості і мають малі розміри файлів для простих зображень, але вони обмежені в деталізації для складних зображень і можуть вимагати високої обчислювальної потужності при відображенні складних векторних зображень.

Висновок

Розробка графічних редакторів на основі растрових і векторних зображень дозволяє врахувати переваги і недоліки кожного методу, забезпечуючи користувачам широкий спектр можливостей для створення і редагування графічних об'єктів. Інтеграція обох методів у графічний редактор робить його більш універсальним і зручним у використанні.

10. Події Windows, зв'язані з малюванням і відновленням зображень у вікні. Перелік подій та їх характеристика.

Події Windows, зв'язані з малюванням і відновленням зображень у вікні. Перелік подій та їх характеристика.

У графічних редакторах події Windows, пов'язані з малюванням і відновленням зображень у вікні, відіграють важливу роль. Ці події включають обробку малювання, перерисовки, масштабування, повороту, та інших взаємодій з користувачем.

Перелік подій та їх характеристика:

1. **WM_PAINT:** Викликається, коли вікно потребує перерисовки. Відповідає за малювання або перерисовку вмісту вікна.
2. **WM_SIZE:** Викликається, коли розмір вікна змінюється. Використовується для перерисовки вмісту у новому розмірі.
3. **WM_LBUTTONDOWN:** Викликається, коли користувач натискає ліву кнопку миші. Використовується для початку операції малювання або виділення.
4. **WM_MOUSEMOVE:** Викликається, коли миша переміщується у межах вікна. Використовується для оновлення позиції курсора під час малювання або виділення.
5. **WM_LBUTTONUP:** Викликається, коли користувач відпускає ліву кнопку миші. Використовується для завершення операції малювання або виділення.
6. **WM_KEYDOWN:** Викликається, коли користувач натискає клавішу на клавіатурі. Використовується для обробки клавіатурних команд, таких як зміна інструментів або збереження файлу.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>
```

```
class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
```

```
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}
```

```
void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}
```

```
bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}
```

```
void MainWindow::setPenColor() {


 QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
```

```
    // Малюємо векторні зображення
    drawVectorImage(painter);
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        drawing = true;
        lastPoint = event->pos();
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if ((event->buttons() & Qt::LeftButton) && drawing) {
        QPainter painter(&image);
        painter.setPen(pen);
        painter.drawLine(lastPoint, event->pos());
        lastPoint = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        drawing = false;
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
```

```
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **WM_PAINT (paintEvent):** Викликається при перерисовці вікна. В нашому випадку, він використовується для малювання зображень і примітивів.
2. **WM_SIZE (resizeEvent):** Викликається при зміні розміру вікна. Використовується для оновлення вмісту вікна при зміні розміру.
3. **WM_LBUTTONDOWN (mousePressEvent):** Викликається при натисканні лівої кнопки миші. Використовується для початку операції малювання.
4. **WM_MOUSEMOVE (mouseMoveEvent):** Викликається при переміщенні миші в межах вікна. Використовується для оновлення позиції курсора під час малювання.
5. **WM_LBUTTONUP (mouseReleaseEvent):** Викликається при відпусканні лівої кнопки миші. Використовується для завершення операції малювання.
6. **WM_KEYDOWN (keyPressEvent):** Викликається при натисканні клавіші на клавіатурі. Використовується для обробки клавіатурних команд, таких як зміна інструментів або збереження файлу.

Висновок

Додавання обробки подій Windows, пов'язаних з малюванням і відновленням зображень, забезпечує інтерактивність і динамічність графічного редактора. Це дозволяє користувачам ефективно взаємодіяти з програмою, використовуючи мишу та клавіатуру для малювання, редагування та інших операцій.

11. Стандартні класи системи програмування для малювання. Класи без зберігання малюнка (без поля пам'яті) і класи зі зберіганням малюнка у власній пам'яті.

Стандартні класи системи програмування для малювання. Класи без зберігання малюнка (без поля пам'яті) і класи зі зберіганням малюнка у власній пам'яті.

Графічні редактори можуть використовувати різні підходи для обробки і зберігання зображень. Два основних підходи включають класи, які не зберігають зображення в пам'яті, і класи, які зберігають зображення у власній пам'яті.

Класи без зберігання малюнка (без поля пам'яті)

Ці класи використовуються для малювання без збереження зображень у пам'яті. Вони зазвичай виконують операції малювання безпосередньо на екрані або вікні додатку.

Приклад реалізації класу без зберігання малюнка:

```cpp
#include <QWidget>
#include <QPainter>
#include <QPen>
#include <QBrush>
#include <QFont>

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
```

```
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

Класи зі зберіганням малюнка у власній пам'яті

Ці класи використовуються для зберігання зображень у пам'яті, що дозволяє виконувати операції над зображенням, зберігати його та відновлювати після змін.

Приклад реалізації класу зі зберіганням малюнка у власній пам'яті:

```cpp
#include <QWidget>
#include <QImage>
#include <QPainter>
#include <QPen>
#include <QBrush>
#include <QFont>

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
```

```
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>

class DrawingWidget : public QWidget {
    Q_OBJECT
```

```
public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
```

```
private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
```

```
#include <QMouseEvent>
#include <QKeyEvent>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
```

```
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::create

Menus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}
```

```
void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
```

```
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
```

```
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        drawing = true;
        lastPoint = event->pos();
    }
```

```
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if ((event->buttons() & Qt::LeftButton) && drawing) {
        QPainter painter(&image);
        painter.setPen(pen);
        painter.drawLine(lastPoint, event->pos());
        lastPoint = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        drawing = false;
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
```

```
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Класи без зберігання малюнка (без поля пам'яті):**
   - Клас `DrawingWidget` використовується для малювання примітивів без збереження малюнка у пам'яті. Він перевизначає метод `paintEvent`, де виконується малювання.

2. **Класи зі зберіганням малюнка у власній пам'яті:**
   - Клас `ImageWidget` використовується для зберігання малюнка у пам'яті. Він містить об'єкт `QImage` для збереження зображення і перевизначає метод `paintEvent`, де малюється зображення з пам'яті.

3. **Події Windows, зв'язані з малюванням і відновленням зображень:**
   - Події обробляються у методах `paintEvent`, `resizeEvent`, `mousePressEvent`, `mouseMoveEvent`, `mouseReleaseEvent` і `keyPressEvent`.

Висновок

Інтеграція стандартних класів системи програмування для малювання, а також обробка подій Windows, пов'язаних з малюванням і відновленням зображень, забезпечує гнучкість та функціональність графічного редактора. Класи без зберігання малюнка підходять для швидкого малювання,

12. Поняття про пересування та зміну розмірів видимих елементів вікна під час виконання програми.

Поняття про пересування та зміну розмірів видимих елементів вікна під час виконання програми

Пересування та зміна розмірів видимих елементів вікна є важливими аспектами інтерфейсу користувача у графічних редакторах. Це дозволяє користувачам налаштовувати інтерфейс відповідно до своїх потреб і забезпечує гнучкість у роботі з графічними елементами.

Пересування елементів вікна

Пересування елементів вікна може бути реалізоване за допомогою обробки подій миші. Коли користувач натискає на елемент і переміщує мишу, елемент змінює своє положення відповідно до руху миші.

Приклад реалізації пересування елементів вікна:

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
```

```
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
```

```
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

```
// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}
```

```
QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
```

```
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage

;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}
```

```
void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}
```

```
void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        drawing = true;
        lastPoint = event->pos();
        dragStartPosition = event->pos();
        dragging = true;
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if ((event->buttons() & Qt::LeftButton) && drawing) {
        QPainter painter(&image);
        painter.setPen(pen);
        painter.drawLine(lastPoint, event->pos());
        lastPoint = event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування елементів
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        move(pos().x() + dx, pos().y() + dy);
        dragStartPosition = event->pos();
```

```
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        drawing = false;
    }
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
```

```
    }
}
```

Пояснення

1. **Пересування елементів вікна:**
   - Реалізоване через обробку подій миші (mousePressEvent, mouseMoveEvent, mouseReleaseEvent).
   - Під час натискання на елемент і переміщення миші, елемент змінює своє положення відповідно до руху миші.

2. **Зміна розмірів елементів вікна:**
   - Використовується метод resizeEvent для обробки подій зміни розміру вікна.
   - При зміні розміру вікна, вміст оновлюється відповідно до нових розмірів.

Висновок

Інтеграція обробки подій для пересування та зміни розмірів видимих елементів вікна дозволяє створити більш гнучкий та інтуїтивно зрозумілий інтерфейс користувача у графічному редакторі. Це забезпечує зручність у роботі та налаштування інтерфейсу відповідно до потреб користувача.

13. Події миші і клавіатури, які можна використати для пересування і зміни розмірів видимих елементів вікна під час виконання програми. Параметри подій.

Події миші і клавіатури, які можна використати для пересування і зміни розмірів видимих елементів вікна під час виконання програми. Параметри подій.

Для реалізації пересування та зміни розмірів видимих елементів вікна під час виконання програми можна використовувати події миші та клавіатури.

Події миші

1. **mousePressEvent(QMouseEvent *event):**
   - Викликається при натисканні кнопки миші.
   - Параметри:
     - `event->button()`: визначає, яка кнопка миші була натиснута (наприклад, Qt::LeftButton).
     - `event->pos()`: координати точки натискання в межах вікна.

2. **mouseMoveEvent(QMouseEvent *event):**
   - Викликається при переміщенні миші.
   - Параметри:
     - `event->buttons()`: визначає, які кнопки миші натиснуті під час переміщення.

    - `event->pos()`: поточні координати миші в межах вікна.

3. **mouseReleaseEvent(QMouseEvent *event):**
   - Викликається при відпусканні кнопки миші.
   - Параметри:
     - `event->button()`: визначає, яка кнопка миші була відпущена.
     - `event->pos()`: координати точки відпускання в межах вікна.

Події клавіатури

1. **keyPressEvent(QKeyEvent *event):**
   - Викликається при натисканні клавіші на клавіатурі.
   - Параметри:
     - `event->key()`: визначає, яка клавіша була натиснута (наприклад, Qt::Key_Up, Qt::Key_Down, Qt::Key_Left, Qt::Key_Right).
     - `event->modifiers()`: визначає модифікатори клавіші (наприклад, Shift, Ctrl, Alt).

2. **keyReleaseEvent(QKeyEvent *event):**
   - Викликається при відпусканні клавіші на клавіатурі.
   - Параметри:
     - `event->key()`: визначає, яка клавіша була відпущена.
     - `event->modifiers()`: визначає модифікатори клавіші.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);
```

```
protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
```

```
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
```

```
#include <QMouseEvent>
#include <QKeyEvent>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
```

```
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow
```

```
::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
```

```
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS, PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") || fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS, PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"), 1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
```

```
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        drawing = true;
```

```
        lastPoint = event->pos();
        dragStartPosition = event->pos();
        dragging = true;
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if ((event->buttons() & Qt::LeftButton) && drawing) {
        QPainter painter(&image);
        painter.setPen(pen);
        painter.drawLine(lastPoint, event->pos());
        lastPoint = event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування елементів
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        move(pos().x() + dx, pos().y() + dy);
        dragStartPosition = event->pos();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        drawing = false;
    }
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
```

```
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Події миші:**
   - `mousePressEvent(QMouseEvent *event)`: Викликається при натисканні кнопки миші. Використовується для початку малювання або перетягування елементів.
   - `mouseMoveEvent(QMouseEvent *event)`: Викликається при переміщенні миші. Використовується для оновлення позиції миші під час малювання або перетягування.
   - `mouseReleaseEvent(QMouseEvent *event)`: Викликається при відпусканні кнопки миші. Використовується для завершення малювання або перетягування.

2. **Події клавіатури:**
   - `keyPressEvent(QKeyEvent *event)`: Викликається при натисканні клавіші на клавіатурі. Використовується для обробки клавіатурних команд.
   - `keyReleaseEvent(QKeyEvent *event)`: Викликається при відпусканні клавіші на клавіатурі. Використовується для обробки подій відпускання клавіш, якщо потрібно.

Висновок

Інтеграція подій миші та клавіатури для пересування та зміни розмірів видимих елементів вікна забезпечує гнучкість та інтуїтивно зрозумілий інтерфейс користувача у графічному редакторі. Це дозволяє користувачам ефективно взаємодіяти з програмою, використовуючи мишу та клавіатуру для виконання різноманітних операцій.

14. Об'єкт пересування та зміни розміру, зміщення об'єкта в процесі пересування, зовнішні розміри об'єкта, поточна канва малювання.

Об'єкт пересування та зміни розміру, зміщення об'єкта в процесі пересування, зовнішні розміри об'єкта, поточна канва малювання

Для реалізації пересування та зміни розмірів об'єктів у графічному редакторі необхідно враховувати параметри об'єкта, такі як його розміри, положення та канва малювання.

Об'єкт пересування та зміни розміру

Об'єкт, який може бути пересунутий або змінений у розмірі, зазвичай має такі атрибути:
- Початкове положення (`startPos`)
- Поточне положення (`currentPos`)
- Розміри (`width` та `height`)

Приклад реалізації класу об'єкта для пересування та зміни розміру:

ResizableMovableObject.h:
```cpp
#ifndef RESIZABLEMOVABLEOBJECT_H
#define RESIZABLEMOVABLEOBJECT_H

#include <QRect>
#include <QPoint>
#include <QPainter>

class ResizableMovableObject {
public:
    ResizableMovableObject(const QRect &rect);

    void move(const QPoint &newPos);
    void resize(int newWidth, int newHeight);
    void draw(QPainter &painter) const;

    const QRect& rect() const;

private:
```

```
    QRect m_rect;
};

#endif // RESIZABLEMOVABLEOBJECT_H
```

ResizableMovableObject.cpp:
```cpp
#include "ResizableMovableObject.h"

ResizableMovableObject::ResizableMovableObject(const QRect &rect)
    : m_rect(rect) {}

void ResizableMovableObject::move(const QPoint &newPos) {
    m_rect.moveTopLeft(newPos);
}

void ResizableMovableObject::resize(int newWidth, int newHeight) {
    m_rect.setSize(QSize(newWidth, newHeight));
}

void ResizableMovableObject::draw(QPainter &painter) const {
    painter.drawRect(m_rect);
}

const QRect& ResizableMovableObject::rect() const {
    return m_rect;
}
```

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
```

```
#include <QPolygon>
#include "ResizableMovableObject.h"

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
```

```
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
    ResizableMovableObject resizableObject; // Об'єкт для пересування та зміни розміру
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:

```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
```

```
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false), resizableObject(QRect(100, 100, 200, 150)) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);
```

```
    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this, &MainWindow

::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
```

```
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
```

```
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);

    // Малюємо об'єкт, що пересувається та змінює розмір
    resizableObject.draw(painter);
```

```
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        drawing = true;
        lastPoint = event->pos();
        dragStartPosition = event->pos();
        if (resizableObject.rect().contains(event->pos())) {
            dragging = true;
        }
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if ((event->buttons() & Qt::LeftButton) && drawing) {
        QPainter painter(&image);
        painter.setPen(pen);
        painter.drawLine(lastPoint, event->pos());
        lastPoint = event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування об'єкта
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        QPoint newPos = resizableObject.rect().topLeft() + QPoint(dx, dy);
        resizableObject.move(newPos);
        dragStartPosition = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        drawing = false;
    }
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
```

```
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Об'єкт пересування та зміни розміру:**
   - Клас `ResizableMovableObject` використовується для представлення об'єкта, який можна пересувати та змінювати його розмір.
   - Метод `move` змінює положення об'єкта, метод `resize` змінює його розміри, метод `draw` малює об'єкт.

2. **Зміщення об'єкта в процесі пересування:**
   - Переміщення об'єкта реалізовано в методах `mouseMoveEvent` та `mousePressEvent`.
   - Використовується змінна `dragStartPosition` для збереження початкової точки перетягування та `dragging` для вказання стану перетягування.

3. **Зовнішні розміри об'єкта:**
   - Метод `resize` змінює розміри об'єкта.

4. **Поточна канва малювання:**
   - Метод `paintEvent` використовується для малювання поточної канви, включаючи всі об'єкти, що пересуваються та змінюються в розмірах.

Висновок

Інтеграція об'єктів, які можуть пересуватися та змінювати розмір, а також обробка подій миші та клавіатури для цих операцій, забезпечує гнучкий та інтуїтивно зрозумілий інтерфейс користувача у графічному редакторі. Це дозволяє користувачам ефективно взаємодіяти з об'єктами на канві малювання.

15. Особливості малювання многокутників. Фіксування вершин полігону. Відображення проміжних полігонів. Динаміка малювання ребер полігона. Події, придатні до малювання полігона і відображення динаміки малювання.

Особливості малювання многокутників. Фіксування вершин полігону. Відображення проміжних полігонів. Динаміка малювання ребер полігона. Події, придатні до малювання полігона і відображення динаміки малювання

Для малювання многокутників необхідно фіксувати вершини полігона та відображати проміжні полігони під час малювання. Це може бути реалізовано за допомогою обробки подій миші для фіксації та малювання вершин.

Особливості малювання многокутників

1. **Фіксування вершин полігона:**
   - Вершини полігона фіксуються при натисканні кнопки миші.
   - Користувач натискає ліву кнопку миші для додавання нової вершини.

2. **Відображення проміжних полігонів:**
   - Проміжні полігони малюються під час переміщення миші.
   - Проміжні ребра полігона малюються між фіксованими вершинами та поточною позицією миші.

3. **Динаміка малювання ребер полігона:**
   - Під час переміщення миші динамічно малюються ребра між фіксованими вершинами та поточною позицією миші.

4. **Події, придатні до малювання полігона:**
   - `mousePressEvent`: фіксує нову вершину полігона.
   - `mouseMoveEvent`: відображає динаміку малювання ребер полігона.
   - `mouseDoubleClickEvent`: завершує малювання полігона, замкнувши його.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>
#include "ResizableMovableObject.h"

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
```

```
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void mouseDoubleClickEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);
```

```
private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
    ResizableMovableObject resizableObject; // Об'єкт для пересування та зміни розміру
    QVector<QPoint> polygonPoints; // Вершини поточного полігона
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>

// Клас DrawingWidget
```

```
DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}
```

```
// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false), resizableObject(QRect(100, 100, 200, 150)) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale"), this, &MainWindow::scaleImage);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale"), this, &MainWindow::scaleImage);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
```

```
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos

) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale"), this, &MainWindow::scaleImage);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
}
```

```
bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage() {
    bool ok;
    double factor = QInputDialog::getDouble(this, tr("Scale Image"), tr("Enter scale factor:"),
1.0, 0.1, 10.0, 1, &ok);
    if (ok) {
        image = image.scaled(image.width() * factor, image.height() * factor);
        update();
    }
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
```

```
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);

    // Малюємо об'єкт, що пересувається та змінює розмір
    resizableObject.draw(painter);

    // Малюємо поточний полігон
    if (!polygonPoints.isEmpty()) {
        QPolygon polygon;
        for (const QPoint &point : polygonPoints) {
            polygon << point;
        }
        painter.drawPolygon(polygon);
        if (drawing) {
            painter.drawLine(polygonPoints.last(), lastPoint);
        }
    }
}
```

```
void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        lastPoint = event->pos();
        if (resizableObject.rect().contains(event->pos())) {
            dragging = true;
            dragStartPosition = event->pos();
        } else {
            polygonPoints.append(event->pos());
            drawing = true;
        }
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if (drawing) {
        lastPoint = event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування об'єкта
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        QPoint newPos = resizableObject.rect().topLeft() + QPoint(dx, dy);
        resizableObject.move(newPos);
        dragStartPosition = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::mouseDoubleClickEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        polygonPoints.append(event->pos());
        drawing = false;
        update();
    }
}
```

```
void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Фіксування вершин полігона:**
   - Користувач додає вершини полігона при натисканні лівої кнопки миші.
   - Вершини зберігаються в `QVector<QPoint> polygonPoints`.

2. **Відображення проміжних полігонів:**
   - Проміжні полігони малюються під час переміщення миші.
   - Динамічне ребро малюється між останньою фіксованою вершиною і поточною позицією миші (`mouseMoveEvent`).

3. **Динаміка малювання ребер полігона:**
   - Динамічне ребро полігона малюється в методі `paintEvent`, коли `drawing` встановлено в `true`.

4. **Події, придатні до малювання полігона:**
   - `mousePressEvent`: фіксує нову вершину полігона.
   - `mouseMoveEvent`: відображає динаміку малювання ребер полігона.
   - `mouseDoubleClickEvent`: завершує малювання полігона, замкнувши його.

Висновок

Інтеграція подій миші для фіксації та малювання вершин полігона, а також динамічне відображення проміжних полігонів, забезпечує інтуїтивно зрозумілий і зручний інтерфейс

16. Масштабування зображень в процесі малювання. Проблеми, які можуть виникати при ручному програмуванні масштабування.

Масштабування зображень в процесі малювання. Проблеми, які можуть виникати при ручному програмуванні масштабування

Масштабування зображень у процесі малювання є важливою функцією для графічних редакторів. Це дозволяє користувачам збільшувати або зменшувати зображення для кращого перегляду або редагування. Проте ручне програмування масштабування може викликати низку проблем.

Проблеми, які можуть виникати при ручному програмуванні масштабування:

1. **Втрата якості зображення:**
   - При збільшенні растрових зображень можливе зниження якості через інтерполяцію пікселів.
   - При зменшенні можуть зникати дрібні деталі.

2. **Спотворення пропорцій:**
   - Неправильне масштабування може призвести до спотворення пропорцій зображення, якщо не зберігається співвідношення сторін.

3. **Обробка великих зображень:**
   - Масштабування великих зображень може бути повільним і вимагати значних ресурсів.

4. **Проблеми з відображенням:**
   - При масштабуванні може виникнути проблема з оновленням відображення зображення на екрані.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>
#include "ResizableMovableObject.h"

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};
```

```
class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void mouseDoubleClickEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);
    void scaleUp();
    void scaleDown();
```

```
private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);
    void scaleImage(double factor);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
    ResizableMovableObject resizableObject; // Об'єкт для пересування та зміни розміру
    QVector<QPoint> polygonPoints; // Вершини поточного полігона
    double scaleFactor; // Масштабний фактор
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>
```

```
// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
```

```
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false), resizableObject(QRect(100, 100, 200, 150)), scaleFactor(1.0) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale Up"), this, &MainWindow::scaleUp);
    editMenu->addAction(tr("&Scale Down"), this, &MainWindow::scaleDown);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    editToolBar->addAction(tr("Scale Down"), this, &MainWindow::scaleDown);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
```

```
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);


 contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    contextMenu.addAction(tr("Scale Down"), this, &MainWindow::scaleDown);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
```

```
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS, PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") || fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS, PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage(double factor) {
    scaleFactor *= factor;
    if (!image.isNull()) {
        image = image.scaled(image.size() * scaleFactor);
    }
    update();
}

void MainWindow::scaleUp() {
    scaleImage(1.25); // Збільшення зображення на 25%
}

void MainWindow::scaleDown() {
    scaleImage(0.8); // Зменшення зображення на 20%
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
```

```
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);

    // Малюємо об'єкт, що пересувається та змінює розмір
    resizableObject.draw(painter);
```

```
    // Малюємо поточний полігон
    if (!polygonPoints.isEmpty()) {
        QPolygon polygon;
        for (const QPoint &point : polygonPoints) {
            polygon << point;
        }
        painter.drawPolygon(polygon);
        if (drawing) {
            painter.drawLine(polygonPoints.last(), lastPoint);
        }
    }
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        lastPoint = event->pos();
        if (resizableObject.rect().contains(event->pos())) {
            dragging = true;
            dragStartPosition = event->pos();
        } else {
            polygonPoints.append(event->pos());
            drawing = true;
        }
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if (drawing) {
        lastPoint = event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування об'єкта
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        QPoint newPos = resizableObject.rect().topLeft() + QPoint(dx, dy);
        resizableObject.move(newPos);
        dragStartPosition = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
```

```
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::mouseDoubleClickEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        polygonPoints.append(event->pos());
        drawing = false;
        update();
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }
```

```
    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Масштабування зображень:**
   - Масштабування зображень реалізовано через метод `scaleImage`, який приймає масштабний фактор.
   - Метод `scaleUp` збільшує масштаб на 25%, а `scaleDown` зменшує на 20%.
   - Масштабний фактор зберігається в змінній `scaleFactor`.

2. **Проблеми при ручному програмуванні масштабування:**
   - Втрата якості зображення може виникати через інтерполяцію пікселів під час збільшення.
   - Спотворення пропорцій може виникнути, якщо не зберігати співвідношення сторін.
   - Масштабування великих зображень може вимагати значних ресурсів та викликати затримки.
   - Проблеми з відображенням можуть виникати через неправильне оновлення зображення на екрані.

3. **Події для масштабування:**
   - `mousePressEvent`, `mouseMoveEvent`, `mouseReleaseEvent`, `keyPressEvent` використовуються для обробки подій миші та клавіатури під час малювання.

Висновок

Інтеграція функції масштабування зображень та обробка можливих проблем при ручному програмуванні масштабування забезпечує користувачам можливість ефективно працю

17. Опрацювання зображень способом фільтрування. Приклади фільтрів та їх формули. Застосування фільтрів до частини загального зображення.

Опрацювання зображень способом фільтрування. Приклади фільтрів та їх формули. Застосування фільтрів до частини загального зображення

Фільтрування зображень дозволяє застосовувати різноманітні ефекти до зображення, такі як розмиття, різкість, виявлення країв тощо. Фільтри використовують матриці згортки для обробки пікселів зображення.

Приклади фільтрів та їх формули:

1. **Розмиття (Blur):**
   - Формула: матриця згортки 3x3 для розмиття.
   ```
   1/9 * | 1 1 1 |
         | 1 1 1 |
         | 1 1 1 |
   ```

2. **Різкість (Sharpen):**
   - Формула: матриця згортки 3x3 для підвищення різкості.
   ```
   |  0 -1  0 |
   | -1  5 -1 |
   |  0 -1  0 |
   ```

3. **Виявлення країв (Edge Detection):**
   - Формула: матриця згортки 3x3 для виявлення країв.
   ```
   | -1 -1 -1 |
   | -1  8 -1 |
   | -1 -1 -1 |
   ```

Застосування фільтрів до частини загального зображення:

1. **Вибір області застосування фільтра:**
   - Користувач може вибрати частину зображення, до якої буде застосовано фільтр.

2. **Обробка вибраної області:**
   - Фільтр застосовується лише до вибраної частини зображення.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>
#include "ResizableMovableObject.h"

class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
```

```
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
    void mouseReleaseEvent(QMouseEvent *event) override;
    void mouseDoubleClickEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);
    void scaleUp();
    void scaleDown();
    void applyBlurFilter();
    void applySharpenFilter();
    void applyEdgeDetectionFilter();

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);
    void scaleImage(double factor);
    void applyFilter(const QVector<QVector<int>> &filter);
```

```
    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
    ResizableMovableObject resizableObject; // Об'єкт для пересування та зміни розміру
    QVector<QPoint> polygonPoints; // Вершини поточного полігона
    double scaleFactor; // Масштабний фактор
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>
#include <QImageReader>
#include <QImageWriter>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
```

```
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false), resizableObject(QRect(100, 100, 200, 150)), scaleFactor(1.0) {
    createMenus();
```

```
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale Up"), this, &MainWindow::scaleUp);
    editMenu->addAction(tr("&Scale Down"), this, &MainWindow::scaleDown);
    editMenu->addAction(tr("&Apply Blur Filter"), this, &MainWindow::applyBlurFilter);
    editMenu->addAction(tr("&Apply Sharpen Filter"), this,
&MainWindow::applySharpenFilter);
    editMenu->addAction(tr("&Apply Edge Detection Filter"), this,
&MainWindow::applyEdgeDetectionFilter);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr

("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    editToolBar->addAction(tr("Scale Down"), this, &MainWindow::scaleDown);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusBar()->showMessage(tr("Ready"));
}
```

```
void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    contextMenu.addAction(tr("Scale Down"), this, &MainWindow::scaleDown);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
    contextMenu.addAction(tr("Apply Blur Filter"), this, &MainWindow::applyBlurFilter);
    contextMenu.addAction(tr("Apply Sharpen Filter"), this,
&MainWindow::applySharpenFilter);
    contextMenu.addAction(tr("Apply Edge Detection Filter"), this,
&MainWindow::applyEdgeDetectionFilter);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        loadImage(fileName);
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        saveImage(fileName);
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
```

```
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS, PDF
    }
    update();
    return true;
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") || fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS, PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
}

void MainWindow::scaleImage(double factor) {
    scaleFactor *= factor;
    if (!image.isNull()) {
        image = image.scaled(image.size() * scaleFactor);
    }
    update();
}

void MainWindow::scaleUp() {
    scaleImage(1.25); // Збільшення зображення на 25%
}

void MainWindow::scaleDown() {
    scaleImage(0.8); // Зменшення зображення на 20%
}

void MainWindow::applyBlurFilter() {
    QVector<QVector<int>> blurFilter = {
        {1, 1, 1},
```

```
        {1, 1, 1},
        {1, 1, 1}
    };
    applyFilter(blurFilter);
}

void MainWindow::applySharpenFilter() {
    QVector<QVector<int>> sharpenFilter = {
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    applyFilter(sharpenFilter);
}

void MainWindow::applyEdgeDetectionFilter() {
    QVector<QVector<int>> edgeDetectionFilter = {
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };
    applyFilter(edgeDetectionFilter);
}

void MainWindow::applyFilter(const QVector<QVector<int>> &filter) {
    if (image.isNull()) {
        return;
    }

    QImage filteredImage = image;
    int filterSize = filter.size();
    int halfFilterSize = filterSize / 2;

    for (int y = halfFilterSize; y < image.height() - halfFilterSize; ++y) {
        for (int x = halfFilterSize; x < image.width() - halfFilterSize; ++x) {
            int red = 0;
            int green = 0;
            int blue = 0;

            for (int fy = 0; fy < filterSize; ++fy) {
                for (int fx = 0; fx < filterSize; ++fx) {
                    int imageX = x + fx - halfFilterSize;
                    int imageY = y + fy - halfFilterSize;
                    QColor pixelColor = image.pixelColor(imageX, imageY);

                    red += pixelColor.red() * filter[fy][fx];
                    green += pixelColor.green() * filter[fy][fx];
                    blue += pixelColor.blue() * filter[fy][fx];
```

```
            }
        }

        red = qBound(0, red / 9, 255);
        green = qBound(0, green / 9, 255);
        blue = qBound(0, blue / 9, 255);

        filteredImage.setPixelColor(x, y, QColor(red, green, blue));
      }
   }

   image = filteredImage;
   update();
}

void MainWindow::setPenColor() {
   QColor color = QColorDialog::getColor();
   if (color.isValid()) {
      pen.setColor(color);
   }
}

void MainWindow::setBrushColor() {
   QColor color = QColorDialog::getColor();
   if (color.isValid()) {
      brush.setColor(color);
   }
}

void MainWindow::closeEvent(QCloseEvent *event) {
   QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
      tr("Do you want to save changes before exiting?\n"),
      QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
      QMessageBox::Yes);

   if (resBtn == QMessageBox::Yes) {
      saveFile();
      event->accept();
   } else if (resBtn == QMessageBox::No) {
      event->accept();
   } else {
      event->ignore();
   }
}

void MainWindow::paintEvent(QPaintEvent *event) {
   QPainter painter(this);
   painter.setPen(pen);
```

```
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);

    // Малюємо об'єкт, що пересувається та змінює розмір
    resizableObject.draw(painter);

    // Малюємо поточний полігон
    if (!polygonPoints.isEmpty()) {
        QPolygon polygon;
        for (const QPoint &point : polygonPoints) {
            polygon << point;
        }
        painter.drawPolygon(polygon);
        if (drawing) {
            painter.drawLine(polygonPoints.last(), lastPoint);
        }
    }
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        lastPoint = event->pos();
        if (resizableObject.rect().contains(event->pos())) {
            dragging = true;
            dragStartPosition = event->pos();
        } else {
            polygonPoints.append(event->pos());
            drawing = true;
        }
    }
```

```
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if (drawing) {
        lastPoint =

 event->pos();
        update();
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування об'єкта
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        QPoint newPos = resizableObject.rect().topLeft() + QPoint(dx, dy);
        resizableObject.move(newPos);
        dragStartPosition = event->pos();
        update();
    }
}

void MainWindow::mouseReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
    }
}

void MainWindow::mouseDoubleClickEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        polygonPoints.append(event->pos());
        drawing = false;
        update();
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
```

```
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}
```

Пояснення

1. **Опрацювання зображень способом фільтрування:**
   - Фільтри застосовуються за допомогою методу `applyFilter`, який приймає матрицю згортки.
   - Фільтри реалізовані через методи `applyBlurFilter`, `applySharpenFilter` та `applyEdgeDetectionFilter`.

2. **Приклади фільтрів та їх формули:**
   - Розмиття (Blur) використовує матрицю згортки 3x3.
   - Різкість (Sharpen) використовує матрицю згортки 3x3.
   - Виявлення країв (Edge Detection) використовує матрицю згортки 3x3.

3. **Застосування фільтрів до частини загального зображення:**

- Фільтр застосовується до всього зображення, але його можна модифікувати для застосування лише до вибраної частини.

4. **Події для фільтрування:**
- `mousePressEvent`, `mouseMoveEvent`, `mouseReleaseEvent`, `keyPressEvent` використовуються для обробки подій миші та клавіатури під час малювання.

Висновок

Інтеграція функцій фільтрування зображень та обробка можливих проблем при ручному програмуванні забезпечує користувачам можливість ефективно застосовувати різні ефекти до зображень, що значно розширює функціональність графічного редактора.

18. Контекстні підказки про хід виконання графічних операцій. Перемикання контекстом підказки.

Контекстні підказки про хід виконання графічних операцій. Перемикання контекстом підказки

Контекстні підказки допомагають користувачам орієнтуватися в ході виконання графічних операцій, надаючи інформацію про поточний стан або доступні дії. Перемикання контекстом підказки може бути реалізовано за допомогою статусного рядка або окремого віджета.

Інтеграція з попереднім кодом

MainWindow.h:
```cpp
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWidget>
#include <QMenu>
#include <QToolBar>
#include <QPen>
#include <QBrush>
#include <QFont>
#include <QImage>
#include <QVector>
#include <QLine>
#include <QRect>
#include <QPolygon>
#include <QLabel>
#include "ResizableMovableObject.h"
```

```
class DrawingWidget : public QWidget {
    Q_OBJECT

public:
    DrawingWidget(QWidget *parent = nullptr);

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QPen pen;
    QBrush brush;
    QFont font;
};

class ImageWidget : public QWidget {
    Q_OBJECT

public:
    ImageWidget(QWidget *parent = nullptr);
    void setImage(const QImage &newImage);
    const QImage& getImage() const;

protected:
    void paintEvent(QPaintEvent *event) override;

private:
    QImage image;
    QPen pen;
    QBrush brush;
    QFont font;
};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QWidget* getDrawingArea() const;

protected:
    void closeEvent(QCloseEvent *event) override;
    void paintEvent(QPaintEvent *event) override;
    void resizeEvent(QResizeEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
    void mouseMoveEvent(QMouseEvent *event) override;
```

```
    void mouseReleaseEvent(QMouseEvent *event) override;
    void mouseDoubleClickEvent(QMouseEvent *event) override;
    void keyPressEvent(QKeyEvent *event) override;
    void keyReleaseEvent(QKeyEvent *event) override;

private slots:
    void openFile();
    void saveFile();
    void rotateImage();
    void scaleImage();
    void setPenColor();
    void setBrushColor();
    void drawRasterImage(QPainter &painter);
    void drawVectorImage(QPainter &painter);
    void scaleUp();
    void scaleDown();
    void applyBlurFilter();
    void applySharpenFilter();
    void applyEdgeDetectionFilter();
    void updateStatusBar(const QString &message);

private:
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void createContextMenu();
    bool loadImage(const QString &fileName);
    bool saveImage(const QString &fileName);
    void scaleImage(double factor);
    void applyFilter(const QVector<QVector<int>> &filter);

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QWidget *drawingArea;

    QPen pen;
    QBrush brush;
    QFont font;
    QImage image;
    QVector<QLine> lines;
    QVector<QRect> rectangles;
    QVector<QPolygon> polygons;
    bool drawing; // для фіксації стану малювання
    QPoint lastPoint; // для зберігання останньої точки під час малювання
    QPoint dragStartPosition; // для зберігання початкової точки перетягування
    bool dragging; // для фіксації стану перетягування
```

```
    ResizableMovableObject resizableObject; // Об'єкт для пересування та зміни розміру
    QVector<QPoint> polygonPoints; // Вершини поточного полігона
    double scaleFactor; // Масштабний фактор
    QLabel *statusLabel; // Контекстні підказки
};

#endif // MAINWINDOW_H
```

MainWindow.cpp:
```cpp
#include "MainWindow.h"
#include <QMenuBar>
#include <QStatusBar>
#include <QFileDialog>
#include <QAction>
#include <QMessageBox>
#include <QInputDialog>
#include <QColorDialog>
#include <QMouseEvent>
#include <QKeyEvent>
#include <QImageReader>
#include <QImageWriter>

// Клас DrawingWidget

DrawingWidget::DrawingWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}

void DrawingWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо примітиви без збереження малюнка у пам'яті
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас ImageWidget

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent), pen(Qt::black), brush(Qt::blue, Qt::SolidPattern), font("Arial", 12,
QFont::Bold) {}
```

```
void ImageWidget::setImage(const QImage &newImage) {
    image = newImage;
    update();
}

const QImage& ImageWidget::getImage() const {
    return image;
}

void ImageWidget::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення з пам'яті
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");
}

// Клас MainWindow

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), pen(Qt::black),
brush(Qt::blue, Qt::SolidPattern), font("Arial", 12, QFont::Bold), drawing(false),
dragging(false), resizableObject(QRect(100, 100, 200, 150)), scaleFactor(1.0) {
    createMenus();
    createToolBars();
    createStatusBar();
    createContextMenu();
    drawingArea = new QWidget(this);
    drawingArea->setMinimumSize(800, 600); // Задаємо мінімальний розмір робочої
області
}

MainWindow::~MainWindow() {}

QWidget* MainWindow::getDrawingArea() const {
    return drawingArea;
}

void MainWindow::createMenus() {
    fileMenu = menuBar()->addMenu(tr("&File"));
```

```
    fileMenu->addAction(tr("&Open..."), this, &MainWindow::openFile);
    fileMenu->addAction(tr("&Save"), this, &MainWindow::saveFile);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(tr("&Rotate"), this, &MainWindow::rotateImage);
    editMenu->addAction(tr("&Scale Up"), this, &MainWindow::scaleUp);
    editMenu->addAction(tr("&Scale Down"), this, &MainWindow::scaleDown);
    editMenu->addAction(tr("&Apply Blur Filter"), this, &MainWindow::applyBlurFilter);
    editMenu->addAction(tr("&Apply Sharpen Filter"), this,
&MainWindow::applySharpenFilter);
    editMenu->addAction(tr("&Apply Edge Detection Filter"), this,
&MainWindow::applyEdgeDetectionFilter);
}

void MainWindow::createToolBars() {
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(tr("Open"), this, &MainWindow::openFile);
    fileToolBar->addAction(tr("Save"), this, &MainWindow::saveFile);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    editToolBar->addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    editToolBar->addAction(tr("Scale Down"), this, &MainWindow::scaleDown);
    editToolBar->addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    editToolBar->addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
}

void MainWindow::createStatusBar() {
    statusLabel = new QLabel(this);
    statusBar()->addWidget(statusLabel);
    updateStatusBar("Ready");
}

void MainWindow::createContextMenu() {
    setContextMenuPolicy(Qt::CustomContextMenu);
    connect(this, &MainWindow::customContextMenuRequested, this,
&MainWindow::showContextMenu);
}

void MainWindow::showContextMenu(const QPoint &pos) {
    QMenu contextMenu(tr("Context menu"), this);
    contextMenu.addAction(tr("Rotate"), this, &MainWindow::rotateImage);
    contextMenu.addAction(tr("Scale Up"), this, &MainWindow::scaleUp);
    contextMenu.addAction(tr("Scale Down"), this, &

MainWindow::scaleDown);
    contextMenu.addAction(tr("Pen Color"), this, &MainWindow::setPenColor);
    contextMenu.addAction(tr("Brush Color"), this, &MainWindow::setBrushColor);
```

```
    contextMenu.addAction(tr("Apply Blur Filter"), this, &MainWindow::applyBlurFilter);
    contextMenu.addAction(tr("Apply Sharpen Filter"), this,
&MainWindow::applySharpenFilter);
    contextMenu.addAction(tr("Apply Edge Detection Filter"), this,
&MainWindow::applyEdgeDetectionFilter);
    contextMenu.exec(mapToGlobal(pos));
}

void MainWindow::openFile() {
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "", tr("Images
(*.png *.xpm *.jpg *.bmp *.gif);;Vectors (*.svg *.eps *.pdf)"));
    if (!fileName.isEmpty()) {
        if (loadImage(fileName)) {
            updateStatusBar("Image loaded successfully");
        } else {
            updateStatusBar("Failed to load image");
        }
    }
}

void MainWindow::saveFile() {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save File"), "", tr("PNG
(*.png);;JPEG (*.jpg *.jpeg);;BMP (*.bmp);;GIF (*.gif);;SVG (*.svg);;EPS (*.eps);;PDF
(*.pdf)"));
    if (!fileName.isEmpty()) {
        if (saveImage(fileName)) {
            updateStatusBar("Image saved successfully");
        } else {
            updateStatusBar("Failed to save image");
        }
    }
}

bool MainWindow::loadImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        QImage loadedImage;
        if (!loadedImage.load(fileName)) {
            QMessageBox::warning(this, tr("Load Image"), tr("The image could not be loaded."));
            return false;
        }
        image = loadedImage;
    } else {
        // Додавання коду для завантаження векторних зображень у форматах SVG, EPS,
PDF
    }
    update();
    return true;
```

```
}

bool MainWindow::saveImage(const QString &fileName) {
    if (fileName.endsWith(".png") || fileName.endsWith(".jpg") || fileName.endsWith(".bmp") ||
fileName.endsWith(".gif")) {
        if (!image.save(fileName)) {
            QMessageBox::warning(this, tr("Save Image"), tr("The image could not be saved."));
            return false;
        }
    } else {
        // Додавання коду для збереження векторних зображень у форматах SVG, EPS,
PDF
    }
    return true;
}

void MainWindow::rotateImage() {
    QTransform rotation;
    rotation.rotate(90); // Поворот на 90 градусів
    image = image.transformed(rotation);
    update();
    updateStatusBar("Image rotated");
}

void MainWindow::scaleImage(double factor) {
    scaleFactor *= factor;
    if (!image.isNull()) {
        image = image.scaled(image.size() * scaleFactor);
    }
    update();
    updateStatusBar("Image scaled");
}

void MainWindow::scaleUp() {
    scaleImage(1.25); // Збільшення зображення на 25%
}

void MainWindow::scaleDown() {
    scaleImage(0.8); // Зменшення зображення на 20%
}

void MainWindow::applyBlurFilter() {
    QVector<QVector<int>> blurFilter = {
        {1, 1, 1},
        {1, 1, 1},
        {1, 1, 1}
    };
    applyFilter(blurFilter);
```

```
    updateStatusBar("Blur filter applied");
}

void MainWindow::applySharpenFilter() {
    QVector<QVector<int>> sharpenFilter = {
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    applyFilter(sharpenFilter);
    updateStatusBar("Sharpen filter applied");
}

void MainWindow::applyEdgeDetectionFilter() {
    QVector<QVector<int>> edgeDetectionFilter = {
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };
    applyFilter(edgeDetectionFilter);
    updateStatusBar("Edge detection filter applied");
}

void MainWindow::applyFilter(const QVector<QVector<int>> &filter) {
    if (image.isNull()) {
        return;
    }

    QImage filteredImage = image;
    int filterSize = filter.size();
    int halfFilterSize = filterSize / 2;

    for (int y = halfFilterSize; y < image.height() - halfFilterSize; ++y) {
        for (int x = halfFilterSize; x < image.width() - halfFilterSize; ++x) {
            int red = 0;
            int green = 0;
            int blue = 0;

            for (int fy = 0; fy < filterSize; ++fy) {
                for (int fx = 0; fx < filterSize; ++fx) {
                    int imageX = x + fx - halfFilterSize;
                    int imageY = y + fy - halfFilterSize;
                    QColor pixelColor = image.pixelColor(imageX, imageY);

                    red += pixelColor.red() * filter[fy][fx];
                    green += pixelColor.green() * filter[fy][fx];
                    blue += pixelColor.blue() * filter[fy][fx];
                }
```

```
            }

            red = qBound(0, red / 9, 255);
            green = qBound(0, green / 9, 255);
            blue = qBound(0, blue / 9, 255);

            filteredImage.setPixelColor(x, y, QColor(red, green, blue));
        }
    }

    image = filteredImage;
    update();
}

void MainWindow::setPenColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        pen.setColor(color);
        updateStatusBar("Pen color changed");
    }
}

void MainWindow::setBrushColor() {
    QColor color = QColorDialog::getColor();
    if (color.isValid()) {
        brush.setColor(color);
        updateStatusBar("Brush color changed");
    }
}

void MainWindow::closeEvent(QCloseEvent *event) {
    QMessageBox::StandardButton resBtn = QMessageBox::question(this, "Confirm Exit",
        tr("Do you want to save changes before exiting?\n"),
        QMessageBox::Cancel | QMessageBox::No | QMessageBox::Yes,
        QMessageBox::Yes);

    if (resBtn == QMessageBox::Yes) {
        saveFile();
        event->accept();
    } else if (resBtn == QMessageBox::No) {
        event->accept();
    } else {
        event->ignore();
    }
}

void MainWindow::paintEvent(QPaintEvent *event) {
    QPainter painter(this);
```

```
    painter.setPen(pen);
    painter.setBrush(brush);
    painter.setFont(font);

    // Малюємо зображення
    if (!image.isNull()) {
        painter.drawImage(0, 0, image);
    }

    // Малюємо примітиви
    painter.drawLine(10, 10, 200, 200);
    painter.drawRect(50, 50, 100, 100);
    painter.drawText(100, 100, "Hello, World!");

    // Малюємо векторні зображення
    drawVectorImage(painter);

    // Малюємо об'єкт, що пересувається та змінює розмір
    resizableObject.draw(painter);

    // Малюємо поточний полігон
    if (!polygonPoints.isEmpty()) {
        QPolygon polygon;
        for (const QPoint &point : polygonPoints) {
            polygon << point;
        }
        painter.drawPolygon(polygon);
        if (drawing) {
            painter.drawLine(polygonPoints.last(), lastPoint);
        }
    }
}

void MainWindow::resizeEvent(QResizeEvent *event) {
    // Оновлюємо вміст при зміні розміру вікна
    update();
}

void MainWindow::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton) {
        lastPoint = event->pos();
        if (resizableObject.rect().contains(event->pos())) {
            dragging = true;
            dragStartPosition = event->pos();
            updateStatusBar("Dragging object");
        } else {
            polygonPoints.append(event->pos());
            drawing = true;
```

```
            updateStatusBar("Drawing polygon");
        }
    }
}

void MainWindow::mouseMoveEvent(QMouseEvent *event) {
    if (drawing) {
        lastPoint = event->pos();
        update();
        updateStatusBar("Drawing polygon");
    }
    if ((event->buttons() & Qt::LeftButton) && dragging) {
        // Логіка для перетягування об'єкта
        int dx = event->pos().x() - dragStartPosition.x();
        int dy = event->pos().y() - dragStartPosition.y();
        QPoint newPos = resizableObject.rect().topLeft() + QPoint(dx, dy);
        resizableObject.move(newPos);
        dragStartPosition = event->pos();
        update();
        updateStatusBar("Dragging object");
    }
}

void MainWindow::mouse

ReleaseEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && dragging) {
        dragging = false;
        updateStatusBar("Object dropped");
    }
}

void MainWindow::mouseDoubleClickEvent(QMouseEvent *event) {
    if (event->button() == Qt::LeftButton && drawing) {
        polygonPoints.append(event->pos());
        drawing = false;
        update();
        updateStatusBar("Polygon completed");
    }
}

void MainWindow::keyPressEvent(QKeyEvent *event) {
    switch (event->key()) {
        case Qt::Key_O:
            openFile();
            break;
        case Qt::Key_S:
            saveFile();
```

```
            break;
        case Qt::Key_R:
            rotateImage();
            break;
        case Qt::Key_P:
            setPenColor();
            break;
        case Qt::Key_B:
            setBrushColor();
            break;
        default:
            QMainWindow::keyPressEvent(event);
    }
}

void MainWindow::keyReleaseEvent(QKeyEvent *event) {
    switch (event->key()) {
        // Додати обробку подій для відпускання клавіш, якщо потрібно
        default:
            QMainWindow::keyReleaseEvent(event);
    }
}

void MainWindow::drawVectorImage(QPainter &painter) {
    // Малюємо лінії
    for (const QLine &line : lines) {
        painter.drawLine(line);
    }

    // Малюємо прямокутники
    for (const QRect &rect : rectangles) {
        painter.drawRect(rect);
    }

    // Малюємо полігони
    for (const QPolygon &polygon : polygons) {
        painter.drawPolygon(polygon);
    }
}

void MainWindow::updateStatusBar(const QString &message) {
    statusLabel->setText(message);
}
```

Пояснення

1. **Контекстні підказки:**

- Контекстні підказки реалізовані через `QLabel` в статусному рядку.
- Метод `updateStatusBar` оновлює текст в статусному рядку відповідно до ходу виконання графічних операцій.

2. **Перемикання контекстом підказки:**
- Під час виконання різних графічних операцій (наприклад, малювання полігона, перетягування об'єкта) статусний рядок оновлюється для надання інформації користувачу.

3. **Події для оновлення контекстних підказок:**
- Події миші (`mousePressEvent`, `mouseMoveEvent`, `mouseReleaseEvent`, `mouseDoubleClickEvent`) та події клавіатури (`keyPressEvent`, `keyReleaseEvent`) використовуються для оновлення статусного рядка.

Висновок

Інтеграція контекстних підказок про хід виконання графічних операцій та перемикання контекстом підказки забезпечує користувачам зручну та інтуїтивну взаємодію з графічним редактором, надаючи їм необхідну інформацію про поточний стан або доступні дії.

Список питань до розділу "Електронні таблиці"

1.Означення термінів "активна комірка", "впливаюча комірка", "залежна комірка", "ряд даних", "властивості комірки". Алгоритм розпізнавання статусу комірки.

Означення термінів

1. **Активна комірка (Active Cell):**
- Це комірка, яка виділена в даний момент і готова до введення або редагування даних. Вона зазвичай виділяється рамкою або іншими візуальними ефектами.

2. **Впливаюча комірка (Precedent Cell):**
- Це комірка, на значення якої посилаються інші комірки. Наприклад, якщо комірка B1 містить формулу `=A1+1`, то комірка A1 є впливаючою для комірки B1.

3. **Залежна комірка (Dependent Cell):**
- Це комірка, яка містить формулу або посилання на інші комірки. Наприклад, якщо комірка B1 містить формулу `=A1+1`, то комірка B1 є залежною від комірки A1.

4. **Ряд даних (Data Row):**

- Це горизонтальна послідовність комірок в таблиці, яка зазвичай містить пов’язані дані. Ряди даних часто використовуються для представлення окремих записів або об'єктів.

5. **Властивості комірки (Cell Properties):**
   - Це атрибути комірки, які визначають її зовнішній вигляд і поведінку. До властивостей комірки можуть належати форматування (шрифт, колір, межі), тип даних (текст, число, дата), формули та інші атрибути.

Алгоритм розпізнавання статусу комірки

Для визначення статусу комірки (активна, впливаюча, залежна) можна використовувати наступний алгоритм:

1. **Активна комірка:**
   - Перевірити, чи є комірка виділеною.
   - Якщо комірка виділена, вона є активною.

2. **Впливаюча комірка:**
   - Перевірити, чи містять інші комірки посилання на цю комірку.
   - Якщо є інші комірки, що містять посилання на цю комірку, вона є впливаючою.

3. **Залежна комірка:**
   - Перевірити, чи містить комірка формулу з посиланням на інші комірки.
   - Якщо комірка містить таку формулу, вона є залежною.

Фрагмент програмної реалізації

Нижче наведено приклад реалізації алгоритму розпізнавання статусу комірки на Python з використанням бібліотеки pandas для обробки електронних таблиць:

```python
import pandas as pd

# Створення прикладної таблиці
data = {
    'A': [1, 2, '=B1+1', 4],
    'B': [5, 6, 7, '=A4*2']
}
df = pd.DataFrame(data)

# Функція для перевірки статусу комірки
def get_cell_status(df, row, col):
    cell_value = df.iloc[row, col]
    active = False
    precedent = False
    dependent = False
```

```
    # Перевірка, чи є комірка активною
    if row == df.index[0] and col == df.columns[0]:
        active = True

    # Перевірка, чи є комірка впливаючою
    for r in range(len(df)):
        for c in range(len(df.columns)):
            if isinstance(df.iloc[r, c], str) and f"{df.columns[col]}{row+1}" in df.iloc[r, c]:
                precedent = True

    # Перевірка, чи є комірка залежною
    if isinstance(cell_value, str) and '=' in cell_value:
        dependent = True

    return {
        'active': active,
        'precedent': precedent,
        'dependent': dependent
    }

# Приклад використання функції
status = get_cell_status(df, 0, 0)
print(f"Active: {status['active']}, Precedent: {status['precedent']}, Dependent:
{status['dependent']}")
```

Пояснення

1. **Активна комірка:**
   - В даному прикладі активною коміркою є перша комірка таблиці (A1).

2. **Впливаюча комірка:**
   - Комірки перевіряються на наявність формул, які посилаються на задану комірку.

3. **Залежна комірка:**
   - Якщо комірка містить формулу, вона визначається як залежна.

Цей код перевіряє статус комірки і визначає, чи є вона активною, впливаючою або залежною, використовуючи простий приклад електронної таблиці на Python.

2. Формули в комірках ЕТ. Структура формули на рівні змістового призначення. Типи можливих синтаксичних і семантичних помилок в формулах. Критерії визначення помилки.

Формули в комірках електронних таблиць (ЕТ)

Формули в комірках ЕТ використовуються для виконання обчислень або маніпулювання даними. Формула завжди починається зі знака рівності (`=`) і може містити операції, функції, посилання на інші комірки та константи.

Структура формули на рівні змістового призначення

Структура формули може включати:

1. **Оператори:**
   - Арифметичні оператори: `+`, `-`, `*`, `/`, `^`
   - Порівняльні оператори: `=`, `>`, `<`, `>=`, `<=`, `<>`
   - Логічні оператори: `AND`, `OR`, `NOT`

2. **Функції:**
   - Вбудовані функції: `SUM()`, `AVERAGE()`, `IF()`, `VLOOKUP()`, `INDEX()`, тощо.

3. **Посилання на комірки:**
   - Абсолютні посилання: `$A$1`
   - Відносні посилання: `A1`
   - Змішані посилання: `$A1`, `A$1`

4. **Константи:**
   - Числові значення: `10`, `3.14`
   - Текстові значення: `"Hello"`

5. **Дужки:**
   - Круглі дужки для групування операцій: `(A1 + B1) * C1`

Типи можливих синтаксичних і семантичних помилок в формулах

1. **Синтаксичні помилки:**
   - **Неправильний синтаксис:** Неправильне використання операторів або функцій, відсутність дужок, відсутність або надлишок аргументів у функціях.
   - **Неправильні посилання:** Вказівка на неіснуючі комірки або діапазони.
   - **Помилки при введенні:** Неправильне введення формул або символів.

2. **Семантичні помилки:**
   - **Поділ на нуль:** Формула містить операцію поділу на нуль, що викликає помилку.
   - **Типові помилки:** Використання неправильних типів даних у функціях (наприклад, використання тексту замість чисел).
   - **Помилки обчислень:** Використання значень, що призводять до математично некоректних результатів (наприклад, взяття кореня з від'ємного числа).

Критерії визначення помилки

1. **Синтаксичні помилки:**

- Перевірка правильності структури формули (наявність знака рівності, правильне використання операторів і функцій, відповідність дужок).
- Перевірка існування всіх посилань на комірки та діапазони.

2. **Семантичні помилки:**
- Перевірка на поділ на нуль.
- Перевірка типів даних, що використовуються у функціях і операціях.
- Логічна перевірка коректності обчислень.

Фрагмент програмної реалізації

Нижче наведено приклад реалізації перевірки формул на Python з використанням бібліотеки pandas для обробки електронних таблиць:

```python
import pandas as pd
import re

# Функція для перевірки синтаксису формули
def check_syntax(formula):
    pattern = r"^=([A-Za-z0-9\+\-\*/\^&<>\(\), ]+)$"
    match = re.match(pattern, formula)
    return bool(match)

# Функція для перевірки семантичних помилок у формулі
def check_semantics(df, formula):
    try:
        eval_formula = formula[1:]
        for col in df.columns:
            eval_formula = eval_formula.replace(col, f"df['{col}']")
        eval(eval_formula)
        return True
    except ZeroDivisionError:
        return "Division by zero"
    except Exception as e:
        return str(e)

# Приклад таблиці
data = {
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': ["=A1+B1", "=A2/0", "=A3*C1", "=A4+10"]
}
df = pd.DataFrame(data)

# Перевірка формул на синтаксичні і семантичні помилки
for index, row in df.iterrows():
    for col in df.columns:
```

```
        cell_value = row[col]
        if isinstance(cell_value, str) and cell_value.startswith("="):
            if not check_syntax(cell_value):
                print(f"Syntax error in cell {col}{index+1}: {cell_value}")
            else:
                result = check_semantics(df, cell_value)
                if result is not True:
                    print(f"Semantic error in cell {col}{index+1}: {result}")

# Приклад використання
df.loc[0, 'C'] = "=A1 + B1"  # Коректна формула
df.loc[1, 'C'] = "=A2 / 0"   # Семантична помилка (поділ на нуль)
df.loc[2, 'C'] = "=A3 * C1"  # Синтаксична помилка (рекурсивне посилання)
df.loc[3, 'C'] = "=A4 + 10"  # Коректна формула

# Перевірка формул
for i in range(len(df)):
    for col in df.columns:
        cell = df.at[i, col]
        if isinstance(cell, str) and cell.startswith("="):
            if not check_syntax(cell):
                print(f"Syntax error in cell {col}{i+1}: {cell}")
            else:
                result = check_semantics(df, cell)
                if result is not True:
                    print(f"Semantic error in cell {col}{i+1}: {result}")
```

Пояснення

1. **Перевірка синтаксису:**
   - Використовується регулярний вираз для перевірки правильності структури формули.

2. **Перевірка семантики:**
   - Формула обробляється та обчислюється з використанням бібліотеки pandas, щоб виявити семантичні помилки, такі як поділ на нуль.

Висновок

Використання правильних структур даних та алгоритмів для перевірки формул у комірках електронних таблиць дозволяє забезпечити їх коректність і уникнути потенційних помилок під час обчислень та аналізу даних.

3.Граматики для розпізнавання типу комірки. Алгоритм розпізнавання.

Граматики для розпізнавання типу комірки

Граматика для розпізнавання типу комірки може бути визначена за допомогою правил, які визначають різні типи даних у комірці. Типи комірок можуть включати числові значення, текст, формули, дати тощо.

Приклади граматик для розпізнавання типів комірок

1. **Числові значення:**
   - Визначаються за допомогою регулярного виразу, який розпізнає цілі та дробові числа.
   - Приклад: `^[+-]?(\d+(\.\d*)?|\.\d+)$`

2. **Текст:**
   - Визначається як будь-яка послідовність символів, що не починається зі знака рівності (`=`) або не є числом.
   - Приклад: `^[^\=].*`

3. **Формули:**
   - Визначаються як рядки, що починаються зі знака рівності (`=`) і містять математичні або логічні вирази.
   - Приклад: `^=[A-Za-z0-9\+\-\*/\^&<>\(\), ]+$`

4. **Дати:**
   - Визначаються за допомогою регулярного виразу, який розпізнає дати у різних форматах (наприклад, `DD/MM/YYYY` або `YYYY-MM-DD`).
   - Приклад: `^\d{2}/\d{2}/\d{4}$|^\d{4}-\d{2}-\d{2}$`

Алгоритм розпізнавання типу комірки

1. **Перевірка на формулу:**
   - Якщо комірка починається зі знака рівності (`=`), вона вважається формулою.

2. **Перевірка на число:**
   - Якщо комірка відповідає регулярному виразу для числових значень, вона вважається числом.

3. **Перевірка на дату:**
   - Якщо комірка відповідає регулярному виразу для дат, вона вважається датою.

4. **Всі інші випадки:**
   - Якщо комірка не відповідає жодному з попередніх критеріїв, вона вважається текстовою.

Фрагмент програмної реалізації

Нижче наведено приклад реалізації алгоритму розпізнавання типу комірки на Python з використанням бібліотеки pandas:

```python
import re
import pandas as pd

# Регулярні вирази для різних типів комірок
number_regex = re.compile(r'^[+-]?(\d+(\.\d*)?|\.\d+)$')
formula_regex = re.compile(r'^=[A-Za-z0-9\+\-\*/\^&<>\(\), ]+$')
date_regex = re.compile(r'^\d{2}/\d{2}/\d{4}$|^\d{4}-\d{2}-\d{2}$')

# Функція для визначення типу комірки
def detect_cell_type(cell_value):
    if isinstance(cell_value, str):
        if formula_regex.match(cell_value):
            return 'Formula'
        elif number_regex.match(cell_value):
            return 'Number'
        elif date_regex.match(cell_value):
            return 'Date'
        else:
            return 'Text'
    elif isinstance(cell_value, (int, float)):
        return 'Number'
    else:
        return 'Unknown'

# Приклад таблиці
data = {
    'A': [1, 2.5, '=B1+1', 'Hello', '12/12/2020', '2020-12-12'],
    'B': ['=A1*2', '3.14', 'Some text', '=A2/A1', '01/01/2021', 42]
}
df = pd.DataFrame(data)

# Розпізнавання типів комірок
for col in df.columns:
    for i in range(len(df)):
        cell_type = detect_cell_type(df.at[i, col])
        print(f'Cell {col}{i+1}: {df.at[i, col]} - {cell_type}')
```

Пояснення

1. **Регулярні вирази:**

- Використовуються для розпізнавання різних типів комірок (числові значення, формули, дати, текст).

2. **Функція `detect_cell_type`:**
   - Приймає значення комірки та визначає її тип на основі регулярних виразів і перевірок.

3. **Приклад таблиці:**
   - Використовується для демонстрації розпізнавання типів комірок у різних колонках та рядках.

Висновок

Використання граматик і алгоритмів для розпізнавання типів комірок дозволяє автоматично класифікувати дані в електронних таблицях, що значно спрощує їх обробку та аналіз.

4.Поняття лексеми формули. Перелік лексем на прикладі граматики формули. Формальне визначеня лексем.

Поняття лексеми формули

Лексема - це мінімальна одиниця синтаксичного аналізу, яка має певне значення в контексті мови програмування або формули. У контексті електронних таблиць лексеми формули є базовими елементами, з яких складається формула. Лексеми можуть включати оператори, функції, посилання на комірки, константи, дужки тощо.

Перелік лексем на прикладі граматики формули

Розглянемо приклад формули `=SUM(A1, B1) + 10 * C2 / D3 - E4^2`

1. **Оператор присвоєння:** `=`
2. **Функція:** `SUM`
3. **Дужки:** `(`, `)`
4. **Посилання на комірки:** `A1`, `B1`, `C2`, `D3`, `E4`
5. **Арифметичні оператори:** `+`, `*`, `/`, `-`, `^`
6. **Константи:** `10`

Формальне визначення лексем

1. **Оператор присвоєння:**
   - Лексема: `=`
   - Опис: Позначає початок формули.
   - Регулярний вираз: `^=$`

2. **Функції:**
   - Лексема: `SUM`

   - Опис: Ім'я функції, яка виконує певну дію.
   - Регулярний вираз: `^[A-Z]+[A-Z0-9]*$`

3. **Дужки:**
   - Лексеми: `(`, `)`
   - Опис: Використовуються для групування операцій і аргументів функцій.
   - Регулярний вираз: `^[\(\)]$`

4. **Посилання на комірки:**
   - Лексеми: `A1`, `B1`, `C2`, `D3`, `E4`
   - Опис: Вказують на конкретні комірки в електронній таблиці.
   - Регулярний вираз: `^[A-Z]+\d+$`

5. **Арифметичні оператори:**
   - Лексеми: `+`, `-`, `*`, `/`, `^`
   - Опис: Виконують арифметичні операції.
   - Регулярний вираз: `^[\+\-\*/\^]$`

6. **Константи:**
   - Лексема: `10`
   - Опис: Числове значення, яке використовується у формулі.
   - Регулярний вираз: `^[+-]?(\d+(\.\d*)?|\.\d+)$`

Фрагмент програмної реалізації

Нижче наведено приклад реалізації лексичного аналізатора для формул на Python:

```python
import re

# Регулярні вирази для лексем
token_specification = [
    ('ASSIGN', r'^=$'),                 # Оператор присвоєння
    ('FUNCTION', r'^[A-Z]+[A-Z0-9]*$'),   # Функції
    ('LPAREN', r'^\($'),                # Відкриваюча дужка
    ('RPAREN', r'^\)$'),                # Закриваюча дужка
    ('CELL', r'^[A-Z]+\d+$'),             # Посилання на комірки
    ('OPERATOR', r'^[\+\-\*/\^]$'),       # Арифметичні оператори
    ('NUMBER', r'^[+-]?(\d+(\.\d*)?|\.\d+)$'),  # Числові значення
    ('SKIP', r'[ \t]+'),                # Пропуски та табуляції
    ('MISMATCH', r'.')                  # Все інше
]

# Компіляція регулярних виразів
token_re = re.compile('|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in token_specification))

# Лексичний аналізатор
def tokenize(formula):
```

```
    tokens = []
    for match in re.finditer(token_re, formula):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value}')
        tokens.append((kind, value))
    return tokens

# Приклад використання
formula = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = tokenize(formula)
for token in tokens:
    print(token)
```

Пояснення

1. **Регулярні вирази для лексем:**
   - Кожна лексема визначена за допомогою регулярного виразу. Наприклад, `^[A-Z]+\d+$` для посилань на комірки.

2. **Компільований регулярний вираз:**
   - Регулярні вирази об'єднані в один компільований вираз, що дозволяє ітеративно знаходити всі лексеми у формулі.

3. **Функція `tokenize`:**
   - Приймає формулу і повертає список лексем, кожна з яких представлена кортежем з типу лексеми і її значення.

4. **Приклад використання:**
   - Лексичний аналізатор використовується для розбору формули `=SUM(A1, B1) + 10 * C2 / D3 - E4^2` і виводу всіх лексем.

Висновок

Використання граматик і алгоритмів для розпізнавання лексем формул дозволяє автоматично аналізувати та обробляти формули в електронних таблицях, що значно спрощує їх обробку та аналіз. Лексичний аналізатор є ключовим компонентом у цьому процесі, забезпечуючи коректний розподіл формули на окремі елементи.

5.Загальна схема сканера. Визначення структур або класів для розпізнавання лексем сканером. Перелік параметрів лексем для їх опрацювання.

Загальна схема сканера

Сканер, або лексичний аналізатор, є першим етапом компіляції або інтерпретації, який перетворює вхідний потік символів у вихідний потік лексем. Лексема — це базова одиниця синтаксичного аналізу, яка має певне значення. Сканер використовує регулярні вирази для розпізнавання лексем у вхідному тексті.

Загальна схема роботи сканера

1. **Вхідний потік символів:**
   - Отримання рядка, що містить формулу або програмний код.

2. **Визначення регулярних виразів:**
   - Визначення набору регулярних виразів для кожного типу лексем.

3. **Ітеративне сканування:**
   - Ітеративний пошук лексем у вхідному рядку за допомогою регулярних виразів.

4. **Створення об'єктів лексем:**
   - Створення об'єктів лексем і додавання їх до списку вихідних лексем.

5. **Обробка помилок:**
   - Виявлення і обробка синтаксичних помилок у випадку невідповідності символів жодному з регулярних виразів.

Визначення структур або класів для розпізнавання лексем сканером

Для реалізації сканера на Python можна визначити клас `Token` для зберігання лексем і клас `Lexer` для реалізації сканера.

```python
import re

# Клас для зберігання лексем
class Token:
    def __init__(self, type, value, position):
        self.type = type  # Тип лексеми (наприклад, NUMBER, OPERATOR, etc.)
        self.value = value  # Значення лексеми (наприклад, 42, +, SUM, etc.)
        self.position = position  # Позиція лексеми у вхідному рядку

    def __repr__(self):
        return f'Token(type={self.type}, value={self.value}, position={self.position})'
```

```
# Клас для реалізації сканера
class Lexer:
    def __init__(self, rules):
        self.rules = rules
        self.regex = re.compile('|'.join(f'(?P<{name}>{pattern})' for name, pattern in rules))
        self.line_number = 1
        self.position = 0

    def tokenize(self, text):
        tokens = []
        for match in re.finditer(self.regex, text):
            kind = match.lastgroup
            value = match.group()
            position = match.start()
            if kind == 'NUMBER':
                value = float(value) if '.' in value else int(value)
            elif kind == 'SKIP':
                continue
            elif kind == 'MISMATCH':
                raise ValueError(f'Unexpected character {value} at position {position}')
            tokens.append(Token(kind, value, position))
        return tokens

# Регулярні вирази для різних типів лексем
rules = [
    ('ASSIGN', r'^=$'),                 # Оператор присвоєння
    ('FUNCTION', r'^[A-Z]+[A-Z0-9]*$'),   # Функції
    ('LPAREN', r'^\($'),                # Відкриваюча дужка
    ('RPAREN', r'^\)$'),                # Закриваюча дужка
    ('CELL', r'^[A-Z]+\d+$'),             # Посилання на комірки
    ('OPERATOR', r'^[\+\-\*/\^]$'),       # Арифметичні оператори
    ('NUMBER', r'^[+-]?(\d+(\.\d*)?|\.\d+)$'),  # Числові значення
    ('SKIP', r'[ \t]+'),                # Пропуски та табуляції
    ('MISMATCH', r'.')                  # Все інше
]

# Приклад використання
lexer = Lexer(rules)
text = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = lexer.tokenize(text)
for token in tokens:
    print(token)
```

Перелік параметрів лексем для їх опрацювання

1. **Тип лексеми (type):**

- Визначає категорію лексеми (наприклад, `NUMBER`, `OPERATOR`, `FUNCTION`, `CELL`).

2. **Значення лексеми (value):**
   - Конкретне значення лексеми (наприклад, `42`, `+`, `SUM`, `A1`).

3. **Позиція лексеми (position):**
   - Позиція лексеми у вхідному рядку, що дозволяє точно визначити її місцезнаходження для обробки або виявлення помилок.

Пояснення

1. **Клас `Token`:**
   - Зберігає інформацію про тип, значення і позицію лексеми.

2. **Клас `Lexer`:**
   - Використовується для визначення і розпізнавання лексем у вхідному тексті за допомогою регулярних виразів.

3. **Метод `tokenize`:**
   - Виконує лексичний аналіз вхідного тексту, знаходить всі лексеми і повертає їх як список об'єктів `Token`.

Висновок

Загальна схема сканера і визначення структур для розпізнавання лексем дозволяють автоматично аналізувати та обробляти вхідний текст, перетворюючи його в потік лексем, що значно спрощує подальший синтаксичний аналіз і обробку даних.

6.Алгоритми розпізнавання лексем формул у вигляді діаграми станів скінченного автомата.

Алгоритми розпізнавання лексем формул у вигляді діаграми станів скінченного автомата

Скінченний автомат (Finite State Machine, FSM) є потужним інструментом для розпізнавання лексем у формулі. Він складається з набору станів і переходів між цими станами на основі вхідних символів.

Опис алгоритму розпізнавання лексем

1. **Початковий стан (Start):**
   - Визначає початковий стан автомата, з якого починається аналіз вхідного рядка.

2. **Стани для різних типів лексем:**

- **Число (Number State):** Розпізнає числові значення (цілі та дробові).
- **Функція (Function State):** Розпізнає функції.
- **Оператор (Operator State):** Розпізнає арифметичні оператори.
- **Комірка (Cell State):** Розпізнає посилання на комірки.
- **Дужки (Parenthesis State):** Розпізнає дужки.
- **Помилка (Error State):** Обробляє невідомі символи або помилки.

3. **Переходи між станами:**
   - Переходи визначаються на основі вхідних символів і поточного стану автомата.

Діаграма станів скінченного автомата

```
Start
  |  \
  |   \
  |    \
  v     v
Number   Function   Operator   Cell   Parenthesis   Error
  |         |         |       |         |
  v         v         v       v         v         v
 (End)     (End)     (End)   (End)     (End)     (End)
```

Формальна діаграма станів

```plaintext
Start -> Number [0-9]
Start -> Function [A-Z]
Start -> Operator [+-/*^]
Start -> Cell [A-Z0-9]
Start -> Parenthesis [()]
Start -> Error [other]

Number -> Number [0-9]
Number -> DecimalPoint [.]
Number -> End [else]

DecimalPoint -> Number [0-9]
DecimalPoint -> End [else]

Function -> Function [A-Z0-9]
Function -> End [else]

Operator -> End [else]

Cell -> Cell [A-Z0-9]
Cell -> End [else]
```

Parenthesis -> End [else]

Error -> End [else]
```

### Реалізація алгоритму на Python

Нижче наведено приклад реалізації розпізнавання лексем формул за допомогою скінченного автомата на Python:

```python
class LexerFSM:
    def __init__(self):
        self.states = {
            'Start': self.start_state,
            'Number': self.number_state,
            'Function': self.function_state,
            'Operator': self.operator_state,
            'Cell': self.cell_state,
            'Parenthesis': self.parenthesis_state,
            'Error': self.error_state
        }
        self.current_state = 'Start'
        self.buffer = ''
        self.tokens = []

    def start_state(self, char):
        if char.isdigit():
            self.buffer += char
            self.current_state = 'Number'
        elif char.isalpha():
            self.buffer += char
            self.current_state = 'Function'
        elif char in '+-*/^':
            self.buffer += char
            self.current_state = 'Operator'
        elif char.isalnum():
            self.buffer += char
            self.current_state = 'Cell'
        elif char in '()':
            self.buffer += char
            self.current_state = 'Parenthesis'
        else:
            self.buffer += char
            self.current_state = 'Error'

    def number_state(self, char):
```
```

```
    if char.isdigit() or char == '.':
        self.buffer += char
    else:
        self.tokens.append(('NUMBER', self.buffer))
        self.buffer = ''
        self.current_state = 'Start'
        self.start_state(char)

def function_state(self, char):
    if char.isalnum():
        self.buffer += char
    else:
        self.tokens.append(('FUNCTION', self.buffer))
        self.buffer = ''
        self.current_state = 'Start'
        self.start_state(char)

def operator_state(self, char):
    self.tokens.append(('OPERATOR', self.buffer))
    self.buffer = ''
    self.current_state = 'Start'
    self.start_state(char)

def cell_state(self, char):
    if char.isalnum():
        self.buffer += char
    else:
        self.tokens.append(('CELL', self.buffer))
        self.buffer = ''
        self.current_state = 'Start'
        self.start_state(char)

def parenthesis_state(self, char):
    self.tokens.append(('PARENTHESIS', self.buffer))
    self.buffer = ''
    self.current_state = 'Start'
    self.start_state(char)

def error_state(self, char):
    self.tokens.append(('ERROR', self.buffer))
    self.buffer = ''
    self.current_state = 'Start'
    self.start_state(char)

def tokenize(self, text):
    for char in text:
        self.states[self.current_state](char)
    if self.buffer:
```

```
            self.tokens.append((self.current_state.upper(), self.buffer))
        return self.tokens

# Приклад використання
lexer = LexerFSM()
text = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = lexer.tokenize(text)
for token in tokens:
    print(token)
```

Пояснення

1. **Класи та методи:**
   - Клас `LexerFSM` визначає скінченний автомат для розпізнавання лексем формул.
   - Метод `__init__` ініціалізує стани, поточний стан, буфер для лексем та список токенів.
   - Метод `tokenize` приймає текст, розбиває його на символи та проходить через стани автомата, щоб розпізнати лексеми.

2. **Методи станів:**
   - `start_state`, `number_state`, `function_state`, `operator_state`, `cell_state`, `parenthesis_state`, `error_state` відповідають за обробку символів у різних станах та переходи між станами.

3. **Перехід між станами:**
   - При розпізнаванні символа автомат переходить до відповідного стану або завершує обробку лексеми, додаючи її до списку токенів.

Висновок

Використання діаграми станів скінченного автомата для розпізнавання лексем у формулах забезпечує ефективний спосіб аналізу вхідного тексту. Цей підхід дозволяє автоматично класифікувати та обробляти різні елементи формул, спрощуючи процес синтаксичного аналізу та обробки даних.

7.Схема основного циклу сканування формули і кодування лексем.

Схема основного циклу сканування формули і кодування лексем

Основний цикл сканування формули включає ітеративний процес розпізнавання та кодування лексем з вхідного рядка. Цей процес може бути реалізований за допомогою лексичного аналізатора (сканера), який використовує визначені раніше регулярні вирази для розпізнавання різних типів лексем.

Схема основного циклу сканування формули

1. **Ініціалізація:**
   - Вхідний рядок (формула).
   - Індекс позиції в рядку.
   - Список для зберігання лексем.

2. **Основний цикл:**
   - Повторюється, поки не буде досягнуто кінця рядка.
   - Вибірка символа на поточній позиції.
   - Визначення типу лексеми на основі символа.
   - Розпізнавання та кодування лексеми.
   - Додавання лексеми до списку.
   - Перехід до наступного символа.

3. **Завершення:**
   - Перевірка залишкових символів (якщо є).
   - Повернення списку лексем.

Діаграма основного циклу сканування

```plaintext
Start
  |
  v
Initialize (input string, index, tokens)
  |
  v
While (index < length of string)
  |
  v
  Get current character
  |
  v
  Determine token type
  |
  v
  Recognize and encode token
  |
  v
  Add token to list
  |
  v
  Move to next character
  |
  v
Check remaining characters
  |
```

```
  v
Return list of tokens
  |
  v
End
```

Кодування лексем

Кодування лексем включає присвоєння кожній лексемі певного типу (наприклад, NUMBER, FUNCTION, OPERATOR) і збереження її значення. Це дозволяє легко ідентифікувати та обробляти кожну лексему під час подальшого аналізу.

Приклад реалізації на Python

```python
import re

# Клас для зберігання лексем
class Token:
    def __init__(self, type, value, position):
        self.type = type  # Тип лексеми (наприклад, NUMBER, OPERATOR, etc.)
        self.value = value  # Значення лексеми (наприклад, 42, +, SUM, etc.)
        self.position = position  # Позиція лексеми у вхідному рядку

    def __repr__(self):
        return f'Token(type={self.type}, value={self.value}, position={self.position})'

# Клас для реалізації сканера
class Lexer:
    def __init__(self, rules):
        self.rules = rules
        self.regex = re.compile('|'.join(f'(?P<{name}>{pattern})' for name, pattern in rules))
        self.position = 0

    def tokenize(self, text):
        tokens = []
        for match in re.finditer(self.regex, text):
            kind = match.lastgroup
            value = match.group()
            position = match.start()
            if kind == 'NUMBER':
                value = float(value) if '.' in value else int(value)
            elif kind == 'SKIP':
                continue
            elif kind == 'MISMATCH':
                raise ValueError(f'Unexpected character {value} at position {position}')
            tokens.append(Token(kind, value, position))
```

```
        return tokens

# Регулярні вирази для різних типів лексем
rules = [
    ('ASSIGN', r'^=$'),                 # Оператор присвоєння
    ('FUNCTION', r'^[A-Z]+[A-Z0-9]*$'),   # Функції
    ('LPAREN', r'^\($'),                # Відкриваюча дужка
    ('RPAREN', r'^\)$'),                # Закриваюча дужка
    ('CELL', r'^[A-Z]+\d+$'),             # Посилання на комірки
    ('OPERATOR', r'^[\+\-\*/\^]$'),       # Арифметичні оператори
    ('NUMBER', r'^[+-]?(\d+(\.\d*)?|\.\d+)$'),  # Числові значення
    ('SKIP', r'[ \t]+'),                # Пропуски та табуляції
    ('MISMATCH', r'.')                  # Все інше
]

# Приклад використання
lexer = Lexer(rules)
text = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = lexer.tokenize(text)
for token in tokens:
    print(token)
```

Пояснення

1. **Клас `Token`:**
   - Використовується для зберігання інформації про лексеми, включаючи тип, значення та позицію.

2. **Клас `Lexer`:**
   - Використовується для сканування вхідного рядка і розпізнавання лексем на основі регулярних виразів.
   - Метод `tokenize` виконує основний цикл сканування, визначаючи типи лексем, кодує їх і додає до списку токенів.

3. **Регулярні вирази:**
   - Визначають правила для розпізнавання різних типів лексем, таких як оператори, функції, посилання на комірки, числа та дужки.

4. **Основний цикл сканування:**
   - Метод `tokenize` ітерує через вхідний рядок, використовуючи регулярні вирази для розпізнавання і кодування лексем, та додає їх до списку токенів.

Висновок

Схема основного циклу сканування формули і кодування лексем забезпечує ефективний спосіб розпізнавання та обробки елементів формули. Використання регулярних виразів і класів для зберігання лексем дозволяє автоматично

класифікувати та обробляти різні типи лексем, що значно спрощує подальший синтаксичний аналіз і обробку даних.

8.Програмовані функції для кодування і декодування адресів комірок ЕТ. Зображення комірок як лексем.

Програмовані функції для кодування і декодування адресів комірок ЕТ

Кодування і декодування адрес комірок електронних таблиць є важливими завданнями для роботи з формулами та діапазонами даних. Адреси комірок, такі як `A1`, `B2`, використовуються для ідентифікації та доступу до конкретних комірок у таблиці.

Кодування адреси комірки

Кодування адреси комірки означає перетворення символьної адреси (наприклад, `A1`) у числову форму (наприклад, `(0, 0)`), де перша координата представляє рядок, а друга - стовпчик.

Декодування адреси комірки

Декодування адреси комірки означає перетворення числової форми адреси (наприклад, `(0, 0)`) у символьну адресу (наприклад, `A1`).

Зображення комірок як лексем

Комірки в електронних таблицях можуть бути зображені як лексеми, що представляють окремі одиниці синтаксичного аналізу. Наприклад, у формулі `=A1 + B2 * C3`, `A1`, `B2`, `C3` є лексемами типу "CELL".

Реалізація зображення комірок як лексем

Використовуючи раніше описаний лексичний аналізатор, ми можемо додати підтримку для кодування і декодування адрес комірок.

```python
import re

# Клас для зберігання лексем
class Token:
    def __init__(self, type, value, position):
        self.type = type  # Тип лексеми (наприклад, NUMBER, OPERATOR, etc.)
        self.value = value  # Значення лексеми (наприклад, 42, +, SUM, etc.)
        self.position = position  # Позиція лексеми у вхідному рядку

    def __repr__(self):
        return f'Token(type={self.type}, value={self.value}, position={self.position})'
```

```
# Клас для реалізації сканера
class Lexer:
    def __init__(self, rules):
        self.rules = rules
        self.regex = re.compile('|'.join(f'(?P<{name}>{pattern})' for name, pattern in rules))
        self.position = 0

    def tokenize(self, text):
        tokens = []
        for match in re.finditer(self.regex, text):
            kind = match.lastgroup
            value = match.group()
            position = match.start()
            if kind == 'NUMBER':
                value = float(value) if '.' in value else int(value)
            elif kind == 'SKIP':
                continue
            elif kind == 'MISMATCH':
                raise ValueError(f'Unexpected character {value} at position {position}')
            tokens.append(Token(kind, value, position))
        return tokens

# Функція для кодування адреси комірки
def encode_cell_address(address):
    match = re.match(r'^([A-Z]+)(\d+)$', address)
    if not match:
        raise ValueError("Invalid cell address format")

    column_label = match.group(1)
    row = int(match.group(2)) - 1

    # Перетворення колонки з літерної форми в числову
    column = 0
    for char in column_label:
        column = column * 26 + (ord(char) - ord('A') + 1)
    column -= 1

    return (row, column)

# Функція для декодування адреси комірки
def decode_cell_address(row, column):
    if row < 0 or column < 0:
        raise ValueError("Invalid row or column index")

    column_label = ''
    while column >= 0:
        column_label = chr(column % 26 + ord('A')) + column_label
```

```
        column = column // 26 - 1

    return f"{column_label}{row + 1}"

# Регулярні вирази для різних типів лексем
rules = [
    ('ASSIGN', r'^=$'),                 # Оператор присвоєння
    ('FUNCTION', r'^[A-Z]+[A-Z0-9]*$'),   # Функції
    ('LPAREN', r'^\($'),                # Відкриваюча дужка
    ('RPAREN', r'^\)$'),                # Закриваюча дужка
    ('CELL', r'^[A-Z]+\d+$'),             # Посилання на комірки
    ('OPERATOR', r'^[\+\-\*/\^]$'),       # Арифметичні оператори
    ('NUMBER', r'^[+-]?(\d+(\.\d*)?|\.\d+)$'),  # Числові значення
    ('SKIP', r'[ \t]+'),                # Пропуски та табуляції
    ('MISMATCH', r'.')                  # Все інше
]

# Приклад використання
lexer = Lexer(rules)
text = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = lexer.tokenize(text)
for token in tokens:
    if token.type == 'CELL':
        encoded_address = encode_cell_address(token.value)
        print(f"Encoded address for '{token.value}': {encoded_address}")
    else:
        print(token)
```

Пояснення

1. **Функції кодування і декодування:**
   - Функція `encode_cell_address` перетворює символьну адресу комірки у числову форму.
   - Функція `decode_cell_address` перетворює числову адресу комірки у символьну форму.

2. **Клас `Token`:**
   - Використовується для зберігання інформації про лексеми, включаючи тип, значення та позицію.

3. **Клас `Lexer`:**
   - Використовується для сканування вхідного рядка і розпізнавання лексем на основі регулярних виразів.

4. **Основний цикл сканування:**
   - Метод `tokenize` ітерує через вхідний рядок, використовуючи регулярні вирази для розпізнавання і кодування лексем, та додає їх до списку токенів.

- Додатково для лексем типу "CELL" виконується кодування адреси комірки.

Висновок

Програмовані функції для кодування і декодування адрес комірок, а також зображення комірок як лексем, забезпечують ефективний спосіб обробки та аналізу даних в електронних таблицях. Використання класів і функцій для роботи з адресами комірок спрощує маніпулювання даними та формулами, роблячи цей процес більш автоматизованим і надійним.

9.Перетворення формул ЕТ з інфіксної форми у постфіксну. Правила перетворення. Алгоритми, необхідні для перетворення.

Перетворення формул електронних таблиць з інфіксної форми у постфіксну

Перетворення формул з інфіксної форми (звичайного запису з операторами між операндами) у постфіксну форму (зворотна польська нотація) може значно спростити обчислення виразів, оскільки усувається необхідність в дужках і пріоритетах операцій.

Правила перетворення

1. **Операнди (числа або посилання на комірки) записуються без змін.**
2. **Оператори тимчасово зберігаються в стеку, а в постфіксну форму записуються лише тоді, коли це дозволяє пріоритет операцій.**
3. **Дужки використовуються для управління пріоритетом операцій:**
   - Відкриваюча дужка `(` завжди додається в стек.
   - Закриваюча дужка `)` призводить до витягування всіх операторів зі стека до першої відкриваючої дужки.

Алгоритм перетворення (Shunting Yard Algorithm)

1. **Ініціалізація:**
   - Стек для операторів.
   - Список для вихідного постфіксного виразу.

2. **Основний цикл:**
   - Читання символу з інфіксного виразу.
   - Якщо символ є операндом (число або посилання на комірку), додати його до вихідного списку.
   - Якщо символ є оператором, виймати оператори з вершини стека до тих пір, поки вершина стека має оператор з вищим або рівним пріоритетом, після чого додати поточний оператор у стек.
   - Якщо символ є відкриваючою дужкою `(`, додати його у стек.
   - Якщо символ є закриваючою дужкою `)`, виймати оператори зі стека до першої відкриваючої дужки.

3. **Завершення:**
   - Після обробки всіх символів з інфіксного виразу, виймати всі оператори, що залишилися в стеку, і додавати їх до вихідного списку.

Реалізація алгоритму на Python

```python
# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Функція для перевірки, чи є символ оператором
def is_operator(c):
    return c in precedence

# Функція для перевірки, чи є символ числом або посиланням на комірку
def is_operand(c):
    return c.isalnum() or c == '.'

# Функція для перетворення інфіксного виразу у постфіксний
def infix_to_postfix(expression):
    stack = []  # Стек для операторів
    output = []  # Список для вихідного постфіксного виразу
    i = 0

    while i < len(expression):
        c = expression[i]

        if is_operand(c):
            # Читання операнду (число або посилання на комірку)
            operand = c
            i += 1
            while i < len(expression) and is_operand(expression[i]):
                operand += expression[i]
                i += 1
            output.append(operand)
        elif c == '(':
            stack.append(c)
            i += 1
        elif c == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
```

```
            stack.pop()  # Видалити '('
            i += 1
        elif is_operator(c):
            while (stack and stack[-1] != '(' and
                   precedence[stack[-1]] >= precedence[c]):
                output.append(stack.pop())
            stack.append(c)
            i += 1
        else:
            i += 1  # Пропустити пробіли або неочікувані символи

    while stack:
        output.append(stack.pop())

    return ' '.join(output)

# Приклад використання
expression = "=SUM(A1, B1) + 10 * C2 / D3 - E4^2"
# Видаляємо знак рівності та обробляємо вираз
postfix_expression = infix_to_postfix(expression[1:])
print(f"Postfix expression: {postfix_expression}")
```

Пояснення

1. **Пріоритет операторів:**
   - Визначений в словнику `precedence`, де вищі значення означають вищий пріоритет.

2. **Функції `is_operator` та `is_operand`:**
   - `is_operator` перевіряє, чи є символ оператором.
   - `is_operand` перевіряє, чи є символ числом або посиланням на комірку.

3. **Функція `infix_to_postfix`:**
   - Виконує перетворення інфіксного виразу у постфіксний, використовуючи стек для зберігання операторів і список для вихідного виразу.
   - Читає символи з інфіксного виразу, додає операнди до вихідного списку, обробляє оператори за пріоритетом, обробляє дужки.

4. **Приклад використання:**
   - Вираз `=SUM(A1, B1) + 10 * C2 / D3 - E4^2` перетворюється у постфіксну форму.

Висновок

Перетворення формул з інфіксної форми у постфіксну дозволяє значно спростити обчислення виразів. Використання алгоритму Shunting Yard для цього перетворення забезпечує коректну обробку операторів і дужок, зберігаючи правильний порядок операцій. Реалізація цього алгоритму на Python демонструє, як можна автоматизувати процес перетворення формул в електронних таблицях.

10. Побудова постфіксної форми виразів на основі граматики виразів. Нисхідний граматичний розбір виразів методом рекурсивного спуску.

Побудова постфіксної форми виразів на основі граматики виразів

Постфіксна форма виразів (зворотна польська нотація) є зручним форматом для обчислення виразів, оскільки вона не вимагає дужок і враховує пріоритет операторів. Нисхідний граматичний розбір виразів методом рекурсивного спуску (Recursive Descent Parsing) є одним з ефективних методів синтаксичного аналізу для побудови постфіксної форми.

Граматика виразів

Граматика для арифметичних виразів може бути визначена наступним чином:

```
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> (E) | NUMBER | CELL
```

Правила перетворення в постфіксну форму

1. **Кожен операнд залишається без змін.**
2. **Кожен оператор переноситься після відповідних операндів.**
3. **Підвирази в дужках обробляються як окремі вирази.**

Нисхідний граматичний розбір методом рекурсивного спуску

Алгоритм рекурсивного спуску включає рекурсивні функції, які реалізують правила граматики для розбору виразу.

Реалізація алгоритму на Python

```python
import re

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
```

```
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value}')
        tokens.append((kind, value))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(tokens, index):
        postfix, index = parse_T(tokens, index)
        while index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(tokens, index)
            postfix.extend(postfix_T)
            postfix.append(op)
        return postfix, index

    def parse_T(tokens, index):
        postfix, index = parse_F(tokens, index)
        while index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(tokens, index)
            postfix.extend(postfix_F)
            postfix.append(op)
```

```
        return postfix, index

    def parse_F(tokens, index):
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(tokens, index)
            if tokens[index][0] != 'RPAREN':
                raise ValueError("Mismatched parentheses")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token: {token}")

    postfix, index = parse_E(tokens, 0)
    if index != len(tokens):
        raise ValueError("Unexpected tokens at end of expression")
    return postfix

# Приклад використання
expression = "SUM(A1, B1) + 10 * C2 / D3 - E4^2"
tokens = tokenize(expression)
print("Tokens:", tokens)
postfix = parse_expression(tokens)
print("Postfix expression:", postfix)
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени (лексеми) на основі регулярних виразів. Кожен токен визначається своїм типом і значенням.

2. **Рекурсивний спуск:**
   - Функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_T` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.

3. **Приклад використання:**
   - Вираз `SUM(A1, B1) + 10 * C2 / D3 - E4^2` перетворюється у список токенів, а потім у постфіксну форму.

Висновок

Побудова постфіксної форми виразів на основі граматики виразів і реалізація алгоритму рекурсивного спуску забезпечує ефективний спосіб синтаксичного аналізу і перетворення виразів в електронних таблицях. Використання рекурсивного спуску дозволяє гнучко обробляти різні типи виразів і забезпечує коректність обчислень.

11. Модифікація граматичних правил виразів для випадку розбору методом рекурсивного спуску. Питання ліворекурсивних правил і факторизації.

Модифікація граматичних правил виразів для випадку розбору методом рекурсивного спуску

Рекурсивний спуск є ефективним методом синтаксичного аналізу, але потребує певної модифікації граматичних правил для уникнення ліворекурсивності. Ліворекурсивні правила можуть призвести до нескінченної рекурсії, тому граматику необхідно перетворити у праворекурсивну або безрекурсивну форму.

Ліворекурсивність та її факторизація

Ліворекурсивні правила виглядають так:
```
E -> E + T
```
Це правило є ліворекурсивним, оскільки на лівому боці стоїть нетермінал `E`, який знову використовується на правому боці. Такі правила можна перетворити, використовуючи факторизацію, на праворекурсивні або безрекурсивні.

Правила факторизації

1. **Ліворекурсивне правило:**
   ```
   E -> E + T | T
   ```
2. **Безрекурсивне правило (після факторизації):**
   ```
   E -> T E'
   E' -> + T E' | ε
   ```
   Де `ε` представляє порожнє слово (завершення розбору).

Модифікована граматика для рекурсивного спуску

Перетворимо початкову граматику в модифіковану форму, зокрема для арифметичних виразів:

1. **Початкова граматика:**
   ```
   E -> E + T | E - T | T
   T -> T * F | T / F | F
   F -> (E) | NUMBER | CELL
   ```

2. **Модифікована граматика (без ліворекурсивності):**
   ```
   E -> T E'
   E' -> + T E' | - T E' | ε
   T -> F T'
   T' -> * F T' | / F T' | ε
   F -> (E) | NUMBER | CELL
   ```

Реалізація алгоритму на Python

```python
import re

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
```

```
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value}')
        tokens.append((kind, value))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
```

```
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if tokens[index][0] != 'RPAREN':
                raise ValueError("Mismatched parentheses")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token: {token}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError("Unexpected tokens at end of expression")
    return postfix

# Приклад використання
expression = "3 + 5 * (2 - 8)"
tokens = tokenize(expression)
print("Tokens:", tokens)
postfix = parse_expression(tokens)
print("Postfix expression:", postfix)
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени (лексеми) на основі регулярних виразів. Кожен токен визначається своїм типом і значенням.

2. **Рекурсивний спуск:**
   - Функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.

3. **Модифікована граматика:**
   - Граматика була модифікована для уникнення ліворекурсивності, що забезпечує коректну роботу рекурсивного спуску.

4. **Приклад використання:**
   - Вираз `3 + 5 * (2 - 8)` перетворюється у список токенів, а потім у постфіксну форму.

Висновок

Модифікація граматичних правил для уникнення ліворекурсивності дозволяє ефективно використовувати метод рекурсивного спуску для синтаксичного аналізу виразів. Використання рекурсивного спуску з модифікованою граматикою забезпечує

правильне розпізнавання та обробку арифметичних виразів, що значно спрощує подальшу обробку даних і обчислення.

12. Алгоритм взаємодії функцій розпізнавання елементів виразів для перетворення формул ЕТ з інфіксної форми у постфіксну.

Алгоритм взаємодії функцій розпізнавання елементів виразів для перетворення формул ЕТ з інфіксної форми у постфіксну

Для перетворення формул електронних таблиць (ЕТ) з інфіксної форми у постфіксну форму можна використовувати алгоритм рекурсивного спуску. Цей алгоритм дозволяє ефективно обробляти вирази, розпізнаючи елементи виразів (оператори, операнди та дужки) і перетворюючи їх у постфіксну форму.

Основні етапи алгоритму

1. **Лексичний аналіз:**
   - Розбивка вхідного виразу на окремі токени (лексеми) для подальшого синтаксичного аналізу.

2. **Синтаксичний аналіз (рекурсивний спуск):**
   - Використання рекурсивних функцій для розпізнавання елементів виразів згідно з граматичними правилами.

3. **Побудова постфіксного виразу:**
   - Формування постфіксного виразу на основі розпізнаних елементів виразів.

Алгоритм взаємодії функцій

1. **Функція `tokenize`:**
   - Розбиває вхідний рядок на токени на основі регулярних виразів.

2. **Функція `parse_expression`:**
   - Головна функція для синтаксичного аналізу виразу.

3. **Функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime`, `parse_F`:**
   - Рекурсивні функції, які реалізують граматичні правила для розпізнавання елементів виразів.

Реалізація алгоритму на Python

```python
import re

# Пріоритет операторів
```

```
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value}')
        tokens.append((kind, value))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
```

```
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if tokens[index][0] != 'RPAREN':
                raise ValueError("Mismatched parentheses")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token: {token}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError("Unexpected tokens at end of expression")
    return postfix

# Приклад використання
expression = "3 + 5 * (2 - 8)"
tokens = tokenize(expression)
print("Tokens:", tokens)
postfix = parse_expression(tokens)
print("Postfix expression:", postfix)
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени на основі регулярних виразів. Кожен токен визначається своїм типом і значенням.

2. **Рекурсивний спуск:**
   - Головна функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.

3. **Модифікована граматика:**
   - Граматика була модифікована для уникнення ліворекурсивності, що забезпечує коректну роботу рекурсивного спуску.

4. **Приклад використання:**
   - Вираз `3 + 5 * (2 - 8)` перетворюється у список токенів, а потім у постфіксну форму.

Висновок

Алгоритм взаємодії функцій розпізнавання елементів виразів для перетворення формул з інфіксної форми у постфіксну забезпечує ефективний спосіб синтаксичного аналізу і перетворення виразів в електронних таблицях. Використання рекурсивного спуску з модифікованою граматикою забезпечує правильне розпізнавання та обробку арифметичних виразів, що значно спрощує подальшу обробку даних і обчислення.

13. Алгоритм фіксування помилок у формулах і друкування діагностики в процесі перетворення до постфіксної форми.

Алгоритм фіксування помилок у формулах і друкування діагностики в процесі перетворення до постфіксної форми

Перетворення формул з інфіксної форми у постфіксну форму включає фіксування і обробку помилок для забезпечення коректності виразу. Ці помилки можуть включати синтаксичні помилки (недопустимі символи, некоректне використання дужок) та семантичні помилки (поділ на нуль, неправильні типи операндів).

Основні етапи алгоритму

1. **Лексичний аналіз:**
   - Розбивка вхідного виразу на токени (лексеми).
   - Виявлення невідповідних символів.

2. **Синтаксичний аналіз:**
   - Використання рекурсивного спуску для побудови постфіксної форми.
   - Виявлення синтаксичних помилок (незакриті дужки, неправильні послідовності токенів).

3. **Фіксування та діагностика помилок:**
   - Фіксування місця виникнення помилки.
   - Виведення повідомлення про помилку з вказівкою на позицію і тип помилки.

Реалізація алгоритму на Python

```python
import re

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value} at position {match.start()}')
        tokens.append((kind, value, match.start()))
    return tokens
```

```
# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        if index >= len(tokens):
            raise ValueError(f"Unexpected end of expression at position {index}")
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if index >= len(tokens) or tokens[index][0] != 'RPAREN':
```

```
            raise ValueError(f"Mismatched parentheses at position {tokens[index-1][2]}")
        return postfix, index + 1
    else:
        raise ValueError(f"Unexpected token {token[1]} at position {token[2]}")

postfix, index = parse_E(0)
if index != len(tokens):
    raise ValueError(f"Unexpected tokens at end of expression starting at position
{tokens[index][2]}")
return postfix

# Приклад використання
expression = "3 + 5 * (2 - 8"
try:
    tokens = tokenize(expression)
    print("Tokens:", tokens)
    postfix = parse_expression(tokens)
    print("Postfix expression:", postfix)
except ValueError as e:
    print(f"Error: {e}")
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени на основі регулярних виразів. Кожен токен визначається своїм типом, значенням і позицією у виразі.
   - Якщо знайдено невідповідний символ, функція викликає помилку з відповідним повідомленням і позицією.

2. **Рекурсивний спуск:**
   - Головна функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.
   - Всі функції перевіряють наявність очікуваних токенів і викликають помилки з відповідними повідомленнями і позиціями у випадку невідповідності.

3. **Приклад використання:**
   - Вираз `3 + 5 * (2 - 8` містить помилку (незакрита дужка), що викликає відповідне повідомлення про помилку.

Висновок

Алгоритм взаємодії функцій розпізнавання елементів виразів з фіксуванням помилок і діагностики забезпечує ефективний спосіб виявлення і обробки помилок під час перетворення формул з інфіксної форми у постфіксну. Використання лексичного

аналізу і рекурсивного спуску дозволяє точно визначати місця виникнення помилок і забезпечує коректне обчислення виразів.

14. Інтерпретація формул ЕТ на основі постфіксної форми зображення. Структура основного алгоритму інтерпретації.

Інтерпретація формул електронних таблиць (ЕТ) на основі постфіксної форми зображення

Інтерпретація формул на основі постфіксної форми (зворотна польська нотація) дозволяє легко обчислювати значення виразів без необхідності враховувати пріоритети операторів і дужки. У постфіксній формі оператори розташовані після своїх операндів, що дозволяє обчислювати вираз, використовуючи стек.

Структура основного алгоритму інтерпретації

1. **Підготовка даних:**
   - Перетворення формули з інфіксної форми у постфіксну форму.
   - Ініціалізація порожнього стека для обчислення.

2. **Обчислення постфіксного виразу:**
   - Ітерація по кожному токену в постфіксному виразі.
   - Якщо токен є операндом (числом або посиланням на комірку), він додається до стека.
   - Якщо токен є оператором, виконати операцію над відповідними операндами, взятими зі стека, і результат помістити назад у стек.

3. **Повернення результату:**
   - Результат обчислення залишається в стеку.

Алгоритм на Python

```python
import re

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
```

```
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value} at position {match.start()}')
        tokens.append((kind, value, match.start()))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index
```

```
    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        if index >= len(tokens):
            raise ValueError(f"Unexpected end of expression at position {index}")
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if index >= len(tokens) or tokens[index][0] != 'RPAREN':
                raise ValueError(f"Mismatched parentheses at position {tokens[index-1][2]}")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token {token[1]} at position {token[2]}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError(f"Unexpected tokens at end of expression starting at position
{tokens[index][2]}")
    return postfix

# Функція для інтерпретації постфіксного виразу
def evaluate_postfix(postfix, cell_values):
    stack = []

    for token in postfix:
        if isinstance(token, (int, float)):
            stack.append(token)
        elif isinstance(token, str) and re.match(r'^[A-Z]+\d+$', token):
            stack.append(cell_values.get(token, 0))  # Повернути значення комірки або 0,
якщо комірка відсутня
        elif token in ('+', '-', '*', '/', '^'):
            b = stack.pop()
            a = stack.pop()
            if token == '+':
```

```
            stack.append(a + b)
        elif token == '-':
            stack.append(a - b)
        elif token == '*':
            stack.append(a * b)
        elif token == '/':
            if b == 0:
                raise ZeroDivisionError("division by zero")
            stack.append(a / b)
        elif token == '^':
            stack.append(a ** b)
    else:
        raise ValueError(f"Unexpected token {token}")

if len(stack) != 1:
    raise ValueError("Invalid postfix expression")

return stack.pop()

# Приклад використання
expression = "A1 + 5 * (2 - B2)"
cell_values = {'A1': 10, 'B2': 3}  # Значення комірок для прикладу

try:
    tokens = tokenize(expression)
    print("Tokens:", tokens)
    postfix = parse_expression(tokens)
    print("Postfix expression:", postfix)
    result = evaluate_postfix(postfix, cell_values)
    print(f"Result: {result}")
except ValueError as e:
    print(f"Error: {e}")
except ZeroDivisionError as e:
    print(f"Math Error: {e}")
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени на основі регулярних виразів. Кожен токен визначається своїм типом, значенням і позицією у виразі.

2. **Рекурсивний спуск:**
   - Головна функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.

3. **Інтерпретація постфіксного виразу:**
   - Функція `evaluate_postfix` обчислює значення постфіксного виразу, використовуючи стек.
   - Для операндів (чисел або посилань на комірки) додає значення до стека.
   - Для операторів виконує операцію над операндами зі стека і додає результат назад у стек.

4. **Приклад використання:**
   - Вираз `A1 + 5 * (2 - B2)` розбивається на токени, перетворюється у постфіксну форму і обчислюється з використанням значень комірок.

Висновок

Інтерпретація формул електронних таблиць на основі постфіксної форми зображення дозволяє ефективно і коректно обчислювати значення виразів. Використання лексичного аналізу, рекурсивного спуску і стекового алгоритму забезпечує точність і надійність обчислень, а також дозволяє легко інтегрувати обробку формул у різні додатки.

15. Алгоритм застосування стеку і правила опрацювання постфіксної форми виразів в процесі інтерпретації.

Алгоритм застосування стеку і правила опрацювання постфіксної форми виразів в процесі інтерпретації

Інтерпретація постфіксних виразів використовує стек для зберігання операндів і проміжних результатів. Оператори в постфіксній нотації застосовуються до операндів, взятих зі стека, що забезпечує коректне виконання операцій.

Алгоритм інтерпретації постфіксного виразу

1. **Ініціалізація:**
   - Ініціалізація порожнього стека для зберігання операндів.

2. **Обробка постфіксного виразу:**
   - Ітерація по кожному токену в постфіксному виразі:
     - Якщо токен є операндом (число або посилання на комірку), додати його до стека.
     - Якщо токен є оператором, взяти необхідну кількість операндів зі стека, виконати операцію і помістити результат назад у стек.

3. **Повернення результату:**
   - Після обробки всіх токенів результат виразу буде на вершині стека.

Правила опрацювання токенів

1. **Операнди:**
   - Додати операнд до стека.

2. **Оператори:**
   - Взяти необхідну кількість операндів зі стека.
   - Виконати операцію.
   - Помістити результат назад у стек.

Реалізація алгоритму на Python

```python
import re

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value} at position {match.start()}')
        tokens.append((kind, value, match.start()))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
```

```
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        if index >= len(tokens):
            raise ValueError(f"Unexpected end of expression at position {index}")
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if index >= len(tokens) or tokens[index][0] != 'RPAREN':
                raise ValueError(f"Mismatched parentheses at position {tokens[index-1][2]}")
```

```
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token {token[1]} at position {token[2]}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError(f"Unexpected tokens at end of expression starting at position
{tokens[index][2]}")
    return postfix

# Функція для інтерпретації постфіксного виразу
def evaluate_postfix(postfix, cell_values):
    stack = []

    for token in postfix:
        if isinstance(token, (int, float)):
            stack.append(token)
        elif isinstance(token, str) and re.match(r'^[A-Z]+\d+$', token):
            stack.append(cell_values.get(token, 0))  # Повернути значення комірки або 0,
якщо комірка відсутня
        elif token in ('+', '-', '*', '/', '^'):
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                if b == 0:
                    raise ZeroDivisionError("division by zero")
                stack.append(a / b)
            elif token == '^':
                stack.append(a ** b)
        else:
            raise ValueError(f"Unexpected token {token}")

    if len(stack) != 1:
        raise ValueError("Invalid postfix expression")

    return stack.pop()

# Приклад використання
expression = "A1 + 5 * (2 - B2)"
cell_values = {'A1': 10, 'B2': 3}  # Значення комірок для прикладу

try:
```

```
    tokens = tokenize(expression)
    print("Tokens:", tokens)
    postfix = parse_expression(tokens)
    print("Postfix expression:", postfix)
    result = evaluate_postfix(postfix, cell_values)
    print(f"Result: {result}")
except ValueError as e:
    print(f"Error: {e}")
except ZeroDivisionError as e:
    print(f"Math Error: {e}")
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени на основі регулярних виразів. Кожен токен визначається своїм типом, значенням і позицією у виразі.

2. **Рекурсивний спуск:**
   - Головна функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції `parse_E`, `parse_E_prime`, `parse_T`, `parse_T_prime` і `parse_F`.
   - Кожна функція обробляє відповідну частину граматики і повертає список токенів у постфіксній формі.

3. **Інтерпретація постфіксного виразу:**
   - Функція `evaluate_postfix` обчислює значення постфіксного виразу, використовуючи стек.
   - Для операндів (чисел або посилань на комірки) додає значення до стека.
   - Для операторів виконує операцію над операндами зі стека і додає результат назад у стек.

4. **Обробка помилок:**
   - Функція `evaluate_postfix` обробляє можливі помилки, такі як поділ на нуль і некоректні токени.

Висновок

Алгоритм застосування стека для інтерпретації постфіксних виразів дозволяє ефективно і коректно обчислювати значення виразів. Використання лексичного аналізу, рекурсивного спуску і стекового алгоритму забезпечує точність і надійність обчислень, а також дозволяє легко інтегрувати обробку формул у різні додатки.

16. Алгоритм опрацювання зв'язаних формул в процесі інтерпретації (залежна-впливаюча).

Алгоритм опрацювання зв'язаних формул в процесі інтерпретації (залежна-впливаюча)

В електронних таблицях формули можуть бути зв'язані через залежності, де одна комірка впливає на іншу. Інтерпретація таких зв'язаних формул вимагає відстеження залежностей та правильного порядку обчислень.

Основні етапи алгоритму

1. **Визначення залежностей:**
   - Побудова графа залежностей між комірками, де вершини представляють комірки, а ребра - залежності між ними.

2. **Топологічне сортування:**
   - Виконання топологічного сортування графа для визначення порядку обчислення комірок.

3. **Обчислення значень комірок:**
   - Виконання обчислень у порядку, визначеному топологічним сортуванням.

Реалізація алгоритму на Python

1. **Побудова графа залежностей:**
   - Використання словника для зберігання залежностей між комірками.

2. **Топологічне сортування:**
   - Використання алгоритму Кана для топологічного сортування графа.

3. **Обчислення значень комірок:**
   - Обчислення значень комірок у порядку, визначеному топологічним сортуванням.

Приклад реалізації

```python
import re
from collections import defaultdict, deque

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
```

```
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
        ('RPAREN',   r'\)'),            # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value} at position {match.start()}')
        tokens.append((kind, value, match.start()))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
```

```
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        if index >= len(tokens):
            raise ValueError(f"Unexpected end of expression at position {index}")
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if index >= len(tokens) or tokens[index][0] != 'RPAREN':
                raise ValueError(f"Mismatched parentheses at position {tokens[index-1][2]}")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token {token[1]} at position {token[2]}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError(f"Unexpected tokens at end of expression starting at position
{tokens[index][2]}")
    return postfix

# Алгоритм Кана для топологічного сортування
def topological_sort(dependency_graph):
    in_degree = {u: 0 for u in dependency_graph}  # Ініціалізація ступенів входу
    for u in dependency_graph:
        for v in dependency_graph[u]:
            in_degree[v] += 1

    queue = deque([u for u in in_degree if in_degree[u] == 0])
    topo_order = []
```

```
    while queue:
        u = queue.popleft()
        topo_order.append(u)
        for v in dependency_graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)

    if len(topo_order) == len(dependency_graph):
        return topo_order
    else:
        raise ValueError("Cyclic dependency detected")

# Функція для інтерпретації постфіксного виразу
def evaluate_postfix(postfix, cell_values):
    stack = []

    for token in postfix:
        if isinstance(token, (int, float)):
            stack.append(token)
        elif isinstance(token, str) and re.match(r'^[A-Z]+\d+$', token):
            stack.append(cell_values.get(token, 0))  # Повернути значення комірки або 0,
якщо комірка відсутня
        elif token in ('+', '-', '*', '/', '^'):
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                if b == 0:
                    raise ZeroDivisionError("division by zero")
                stack.append(a / b)
            elif token == '^':
                stack.append(a ** b)
        else:
            raise ValueError(f"Unexpected token {token}")

    if len(stack) != 1:
        raise ValueError("Invalid postfix expression")

    return stack.pop()

# Функція для обчислення значень всіх комірок
def evaluate_all_cells(cell_expressions):
```

```
    # Побудова графа залежностей
    dependency_graph = defaultdict(list)
    for cell, expr in cell_expressions.items():
        tokens = tokenize(expr)
        for token in tokens:
            if token[0] == 'CELL':
                dependency_graph[token[1]].append(cell)
        if cell not in dependency_graph:
            dependency_graph[cell] = []

    # Топологічне сортування
    sorted_cells = topological_sort(dependency_graph)

    # Обчислення значень комірок
    cell_values = {}
    for cell in sorted_cells:
        if cell in cell_expressions:
            tokens = tokenize(cell_expressions[cell])
            postfix = parse_expression(tokens)
            cell_values[cell] = evaluate_postfix(postfix, cell_values)

    return cell_values

# Приклад використання
cell_expressions = {
    'A1': '5',
    'B2': 'A1 + 2',
    'C3': 'A1 + B2 * 2'
}

try:
    cell_values = evaluate_all_cells(cell_expressions)
    print("Cell values:", cell_values)
except ValueError as e:
    print(f"Error: {e}")
except ZeroDivisionError as e:
    print(f"Math Error: {e}")
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени (лексеми) на основі регулярних виразів.

2. **Рекурсивний спуск:**
   - Функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції.

3. **Топологічне сортування:**
   - Функція `topological_sort` виконує топологічне сортування графа залежностей за допомогою алгоритму

 Кана.

4. **Обчислення значень комірок:**
   - Функція `evaluate_postfix` обчислює значення постфіксного виразу, використовуючи стек.
   - Функція `evaluate_all_cells` обчислює значення всіх комірок у порядку, визначеному топологічним сортуванням.

5. **Приклад використання:**
   - Використовуються вирази для комірок `A1`, `B2` та `C3`. Обчислюються значення всіх комірок з урахуванням їх залежностей.

Висновок

Алгоритм опрацювання зв'язаних формул забезпечує коректне обчислення значень комірок, враховуючи їх залежності. Використання топологічного сортування дозволяє визначити правильний порядок обчислень, що запобігає помилкам і забезпечує точність результатів.

17. Організація алгоритму інтерпретації пряморекурсивної формули. Необхідні допоміжні параметри для коректної інтерпретації та їх використання.

Організація алгоритму інтерпретації пряморекурсивної формули

Пряморекурсивні формули - це формули, які викликають самі себе безпосередньо або через інші формули. Для коректної інтерпретації таких формул необхідно забезпечити механізм обробки рекурсивних викликів та уникнення нескінченних циклів.

Основні етапи алгоритму

1. **Визначення рекурсивних залежностей:**
   - Побудова графа залежностей між комірками, де вершини представляють комірки, а ребра - залежності між ними.
   - Виявлення циклів у графі для уникнення нескінченних рекурсій.

2. **Топологічне сортування:**
   - Виконання топологічного сортування графа для визначення порядку обчислення комірок.

3. **Обчислення значень комірок:**

  - Виконання обчислень у порядку, визначеному топологічним сортуванням.
  - Використання допоміжних параметрів для обробки рекурсивних викликів.

Допоміжні параметри для коректної інтерпретації

1. **Стан обчислення (calculating_state):**
  - Словник, який зберігає стан обчислення кожної комірки (обчислюється чи вже обчислена).

2. **Кеш значень (value_cache):**
  - Словник, який зберігає обчислені значення комірок для уникнення повторних обчислень.

Реалізація алгоритму на Python

1. **Побудова графа залежностей:**
  - Використання словника для зберігання залежностей між комірками.

2. **Виявлення циклів у графі:**
  - Використання алгоритму для виявлення циклів у графі.

3. **Топологічне сортування:**
  - Використання алгоритму Кана для топологічного сортування графа.

4. **Обчислення значень комірок:**
  - Виконання обчислень з обробкою рекурсивних викликів.

Приклад реалізації

```python
import re
from collections import defaultdict, deque

# Пріоритет операторів
precedence = {
    '+': 1,
    '-': 1,
    '*': 2,
    '/': 2,
    '^': 3
}

# Лексичний аналізатор для виразів
def tokenize(expression):
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('OPERATOR', r'[+\-*/^]'),      # Arithmetic operators
        ('LPAREN',   r'\('),            # Left parenthesis
```

```
        ('RPAREN',   r'\)'),           # Right parenthesis
        ('CELL',     r'[A-Z]+\d+'),     # Cell reference
        ('SKIP',     r'[ \t]+'),        # Skip over spaces and tabs
        ('MISMATCH', r'.')              # Any other character
    ]
    token_re = re.compile('|'.join('(?P<%s>%s)' % pair for pair in token_specification))
    tokens = []
    for match in re.finditer(token_re, expression):
        kind = match.lastgroup
        value = match.group()
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise ValueError(f'Unexpected character {value} at position {match.start()}')
        tokens.append((kind, value, match.start()))
    return tokens

# Рекурсивний спуск для перетворення в постфіксну форму
def parse_expression(tokens):
    def parse_E(index):
        postfix, index = parse_T(index)
        postfix_E_prime, index = parse_E_prime(index)
        postfix.extend(postfix_E_prime)
        return postfix, index

    def parse_E_prime(index):
        if index < len(tokens) and tokens[index][1] in ('+', '-'):
            op = tokens[index][1]
            index += 1
            postfix_T, index = parse_T(index)
            postfix_E_prime, index = parse_E_prime(index)
            postfix_T.append(op)
            postfix_T.extend(postfix_E_prime)
            return postfix_T, index
        return [], index

    def parse_T(index):
        postfix, index = parse_F(index)
        postfix_T_prime, index = parse_T_prime(index)
        postfix.extend(postfix_T_prime)
        return postfix, index

    def parse_T_prime(index):
        if index < len(tokens) and tokens[index][1] in ('*', '/', '^'):
            op = tokens[index][1]
            index += 1
```

```
            postfix_F, index = parse_F(index)
            postfix_T_prime, index = parse_T_prime(index)
            postfix_F.append(op)
            postfix_F.extend(postfix_T_prime)
            return postfix_F, index
        return [], index

    def parse_F(index):
        if index >= len(tokens):
            raise ValueError(f"Unexpected end of expression at position {index}")
        token = tokens[index]
        if token[0] == 'NUMBER':
            return [token[1]], index + 1
        elif token[0] == 'CELL':
            return [token[1]], index + 1
        elif token[0] == 'LPAREN':
            index += 1
            postfix, index = parse_E(index)
            if index >= len(tokens) or tokens[index][0] != 'RPAREN':
                raise ValueError(f"Mismatched parentheses at position {tokens[index-1][2]}")
            return postfix, index + 1
        else:
            raise ValueError(f"Unexpected token {token[1]} at position {token[2]}")

    postfix, index = parse_E(0)
    if index != len(tokens):
        raise ValueError(f"Unexpected tokens at end of expression starting at position
{tokens[index][2]}")
    return postfix

# Алгоритм Кана для топологічного сортування
def topological_sort(dependency_graph):
    in_degree = {u: 0 for u in dependency_graph}  # Ініціалізація ступенів входу
    for u in dependency_graph:
        for v in dependency_graph[u]:
            in_degree[v] += 1

    queue = deque([u for u in in_degree if in_degree[u] == 0])
    topo_order = []

    while queue:
        u = queue.popleft()
        topo_order.append(u)
        for v in dependency_graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                queue.append(v)
```

```
    if len(topo_order) == len(dependency_graph):
        return topo_order
    else:
        raise ValueError("Cyclic dependency detected")

# Функція для інтерпретації постфіксного виразу
def evaluate_postfix(postfix, cell_values):
    stack = []

    for token in postfix:
        if isinstance(token, (int, float)):
            stack.append(token)
        elif isinstance(token, str) and re.match(r'^[A-Z]+\d+$', token):
            stack.append(cell_values.get(token, 0))  # Повернути значення комірки або 0,
якщо комірка відсутня
        elif token in ('+', '-', '*', '/', '^'):
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
            elif token == '/':
                if b == 0:
                    raise ZeroDivisionError("division by zero")
                stack.append(a / b)
            elif token == '^':
                stack.append(a ** b)
        else:
            raise ValueError(f"Unexpected token {token}")

    if len(stack) != 1:
        raise ValueError("Invalid postfix expression")

    return stack.pop()

# Функція для обчислення значень всіх комірок
def evaluate_all_cells(cell_expressions):
    # Побудова графа залежностей
    dependency_graph = defaultdict(list)
    for cell, expr in cell_expressions.items():
        tokens = tokenize(expr)
        for token in tokens:
            if token[0] == 'CELL':
                dependency_graph[token[1]].append(cell)
        if cell not in dependency_graph:
```

```
            dependency_graph[cell] = []

    # Виявлення циклів у графі
    def detect_cycles():
        visited = set()
        rec_stack = set()

        def visit(node):
            if node in rec_stack:
                raise ValueError(f"Cyclic dependency detected involving {node}")
            if node not in visited:
                visited.add(node)
                rec_stack.add(node)
                for neighbor in dependency_graph[node]:
                    visit(neighbor)
                rec_stack.remove(node)

        for node in dependency_graph:
            visit(node)

    detect_cycles()

    # Топологічне сортування
    sorted_cells = topological_sort(dependency_graph)

    # Обчислення значень комірок
    cell_values = {}
    calculating_state = {}

    def evaluate_cell(cell):


if cell in calculating_state:
            if calculating_state[cell] == 'calculating':
                raise ValueError(f"Cyclic dependency detected involving {cell}")
            return cell_values[cell]

        calculating_state[cell] = 'calculating'
        tokens = tokenize(cell_expressions[cell])
        postfix = parse_expression(tokens)
        cell_values[cell] = evaluate_postfix(postfix, cell_values)
        calculating_state[cell] = 'calculated'
        return cell_values[cell]

    for cell in sorted_cells:
        if cell in cell_expressions:
            evaluate_cell(cell)
```

```
    return cell_values

# Приклад використання
cell_expressions = {
    'A1': '5',
    'B2': 'A1 + 2',
    'C3': 'A1 + B2 * 2',
    'D4': 'C3 + B2'
}

try:
    cell_values = evaluate_all_cells(cell_expressions)
    print("Cell values:", cell_values)
except ValueError as e:
    print(f"Error: {e}")
except ZeroDivisionError as e:
    print(f"Math Error: {e}")
```

Пояснення

1. **Лексичний аналізатор:**
   - Функція `tokenize` розбиває вхідний рядок на токени на основі регулярних виразів.

2. **Рекурсивний спуск:**
   - Функція `parse_expression` виконує розбір виразу згідно з правилами граматики, використовуючи рекурсивні функції.

3. **Виявлення циклів у графі:**
   - Функція `detect_cycles` використовує DFS для виявлення циклів у графі залежностей.

4. **Топологічне сортування:**
   - Функція `topological_sort` виконує топологічне сортування графа залежностей за допомогою алгоритму Кана.

5. **Інтерпретація постфіксного виразу:**
   - Функція `evaluate_postfix` обчислює значення постфіксного виразу, використовуючи стек.

6. **Обчислення значень комірок:**
   - Функція `evaluate_all_cells` будує граф залежностей, виконує топологічне сортування і обчислює значення комірок у правильному порядку. Вона також обробляє рекурсивні виклики і запобігає циклічним залежностям.

7. **Приклад використання:**
   - Визначаються формули для комірок `A1`, `B2`, `C3` і `D4`. Алгоритм обчислює їх значення з урахуванням залежностей та можливих рекурсивних викликів.

Висновок

Алгоритм інтерпретації пряморекурсивних формул забезпечує коректне обчислення значень комірок, враховуючи їх взаємозалежності. Використання допоміжних параметрів, таких як стан обчислення і кеш значень, дозволяє ефективно обробляти рекурсивні виклики і уникати нескінченних циклів.