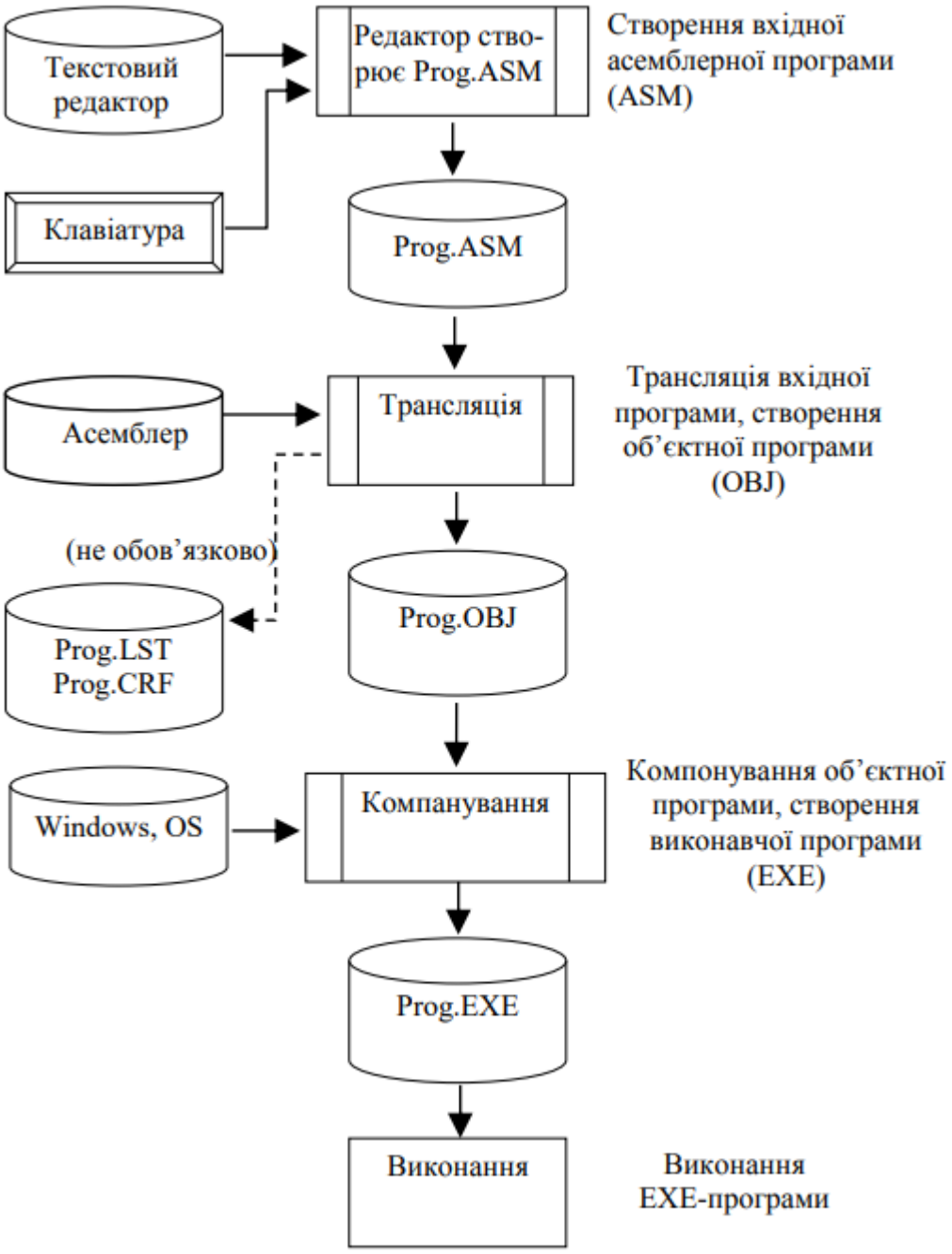
Схема стандартна: вставив своє питання - під ним написав відповідь. Після того виділив зеленим кольором виконане питання у таблиці для прозорості.

Бать Тарас	1, 11, 21, 31, 41, 51, 61
Войтович Ярослав	2, 12, 22, 32, 42, 52, 62
Вівчар Назар	3, 13, 23, 33, 43, 53, 63
Наконечний Лев	4, 14, 24, 34, 44, 54, 64
Пастернак Андрій	5, 15, 25, 35, 45, 55, 65
Сорокопуд Назарій	6, 16, 26, 36, 46, 56, 66
Кузьмич Макс	7, 17, 27, 37, 47, 57, 67
Лисак Роман	8, 18, 28, 38, 48, 58, 68
Дяків Юра	9, 19, 29, 39, 49, 59, 69
Горошко Юра	10, 20, 30, 40, 50, 60

Список питань до розділу "Асемблери"

1. Схема трансляції, компонування і виконання програми.



У процесі налагодження програми виділяються етапи:

- трансляція вихідного тексту програми;
- компонування програми;
- виконання програми з метою визначення логічних помилок;
- тестування програми

Трансляція, компіляція, інтерпретація, лінкування

Трансляція програми - перетворення програми, представленої на одній з мов програмування, в програму на іншій мові і, в певному сенсі, рівносильну першою. При трансляції виконується переклад програми, зрозумілою людині, на мову, зрозумілу комп'ютеру. Виконується спеціальними програмними засобами (транслятором).

Перекладачі реалізуються у вигляді компіляторів або інтерпретаторів. З точки зору виконання роботи компілятор і інтерпретатор істотно розрізняються. Якщо мета трансляції - перетворення всього вихідного тексту на внутрішній мова комп'ютера (тобто одержання певного нового коду) і тільки, то така трансляція називається також **компіляцією.** Оригінальний текст називається також вихідною програмою або вихідним модулем, а результат компіляції - об'єктним кодом або об'єктним модулем. Якщо ж трансляції піддаються окремі оператори вихідних текстів і при цьому отримані коди відразу виконуються, така трансляція називається **інтерпретацією.** Оскільки трансляція виконується спеціальними програмними засобами (трансляторами), останні носять назву компілятора або інтерпретатора, відповідно.

Мета трансляції - перетворити текст з однієї мови на іншу, яка зрозуміла адресату тексту. У випадку програм-трансляторів, адресатом є технічний пристрій (процесор) або програма-інтерпретатор.

Компіляція - перетворення програмою-компілятором вихідного тексту програми, написаного на мові високого рівня в машинну мову, в мову, близький до машинного, або в об'єктний модуль. Результатом компіляції є об'єктний файл з необхідними зовнішніми посиланнями для Компоновнику.

Компілятор читає всю програму цілком, робить її переклад і створює закінчений варіант програми на машинній мові, який потім і виконується.

Види компіляції

- *Пакетна.* Компіляція кількох вихідних модулів в одному пункті завдання.
- *Прогресивний.* Те ж, що і інтерпретація.
- ***Умовна.*** Компіляція, при якій транслюється текст залежить від умов, заданих у вихідній програмі. Так, в залежності від значення деякої константи, можна включати або вимикати трансляцію частини тексту програми.

Інтерпретація - процес безпосереднього покомандного виконання програми без попередньої компіляції, «на льоту»; в більшості випадків інтерпретація набагато повільніше роботи вже компільованою програми, але не вимагає витрат на компіляцію, що у випадку невеликих програм може підвищувати загальну продуктивність.

Типи інтерпретаторів

Простий інтерпретатор аналізує і тут же виконує (власне інтерпретація) програму покомандно (або підрядник), у міру надходження її вихідного коду на вхід інтерпретатора. Його перевага - миттєва реакція. Недолік - такий інтерпретатор виявляє помилки у тексті програми тільки при спробі виконання команди (або рядка) з помилкою.

Інтерпретатор компилирующего типу - це система з компілятора, який переводить вихідний код програми в проміжне представлення, наприклад, в байт-код або р-код, і власне інтерпретатора, який виконує отриманий проміжний код (так звана віртуальна машина). Його перевага - більшу швидкодію виконання програм (за рахунок виносу аналізу вихідного коду в окремий, разовий прохід, і мінімізації цього аналізу в інтерпретаторі). Недоліки - більше вимога до ресурсів і вимогу на коректність вихідного коду.

Алгоритм роботи простого інтерпретатора

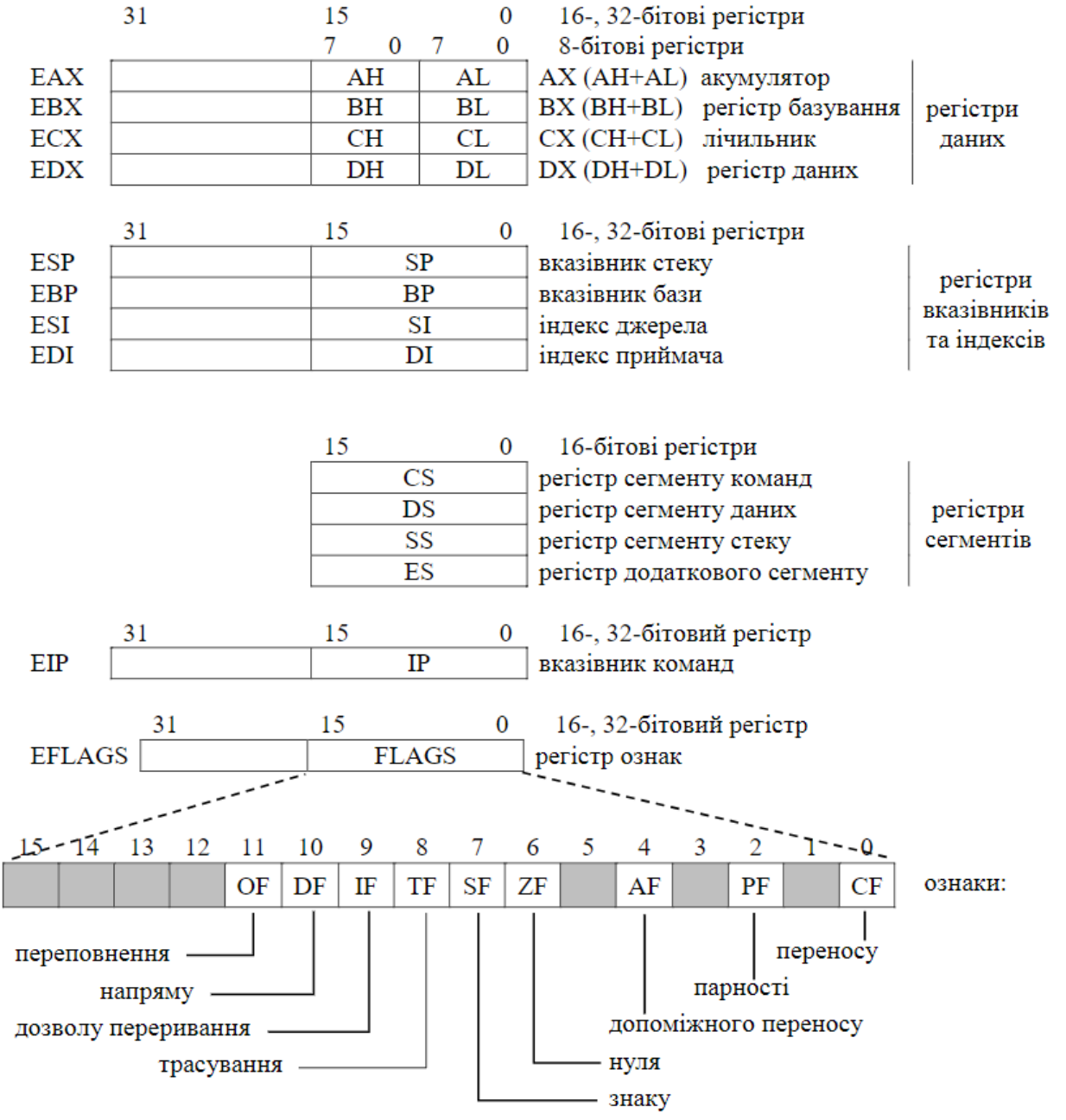
1. прочитати інструкцію;
2. проаналізувати інструкцію і визначити відповідні дії;
3. виконати відповідні дії;
4. якщо не досягнуто умова завершення програми, прочитати таку інструкцію і перейти до пункту 2.

Лінкування (компонування) - це процес, при якому все "недокомпілірованние" частини програми доводяться до кінця і зв'язуються між собою у виконуваний файл (або файли) формату, зрозумілого даної операційної системи. У підсумку, ми отримуємо виконувану програму.

2. Основні регістри мікропроцесора.

За функціональним призначенням регістри поділяють на:

- **Регістри даних** - для збереження цілочисельних даних;
- **Адресні (індексні, базові) регістри -** зберігають адреси в пам'яті і використовуються для операцій з пам'яттю;
- **Регістри загального призначення** - для адрес і даних;
- **Регістри спеціального призначення** - лічильник команд, вказівник стеку, регістр стану процесора, регістр ознак.



Короткі пояснення функціонального призначення регістрів:

Регістри загального призначення

Вісім регістрів загального призначення мають довжину в 32 біт і містять адреси або дані. Вони підтримують операнди-дані довжиною 1, 8, 16, 32 і 64 біт: бітові поля від 1 до 32 біт: операнди-адреси довжиною 16 і 32 біт. Ці регістри називаються EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. Доступ до молодших 16 біт цих регістрів виконується незалежно. Це робиться в більшості асемблерів при використанні 16-розрядних імен регістрів: AX, BX, CX, DX, Sl, Dl, BP, SP.

Регістр ознак

Регістр EFLAGS управляє вводом-висновком, що маскується перериваннями, налагодженням, перемиканням завдань і включенням виконання в режимі віртуального МП 8086 в захищеній багатозадачному середовищі - все це на додаток до прапорів стану, які відображають результат виконання команди. Молодші 16 біт його представляють собою 16-розрядний регістр прапорів і стану МП 80286, званий FLAGS, який найбільш корисний при виконанні програм для МП 8086 і 80286.

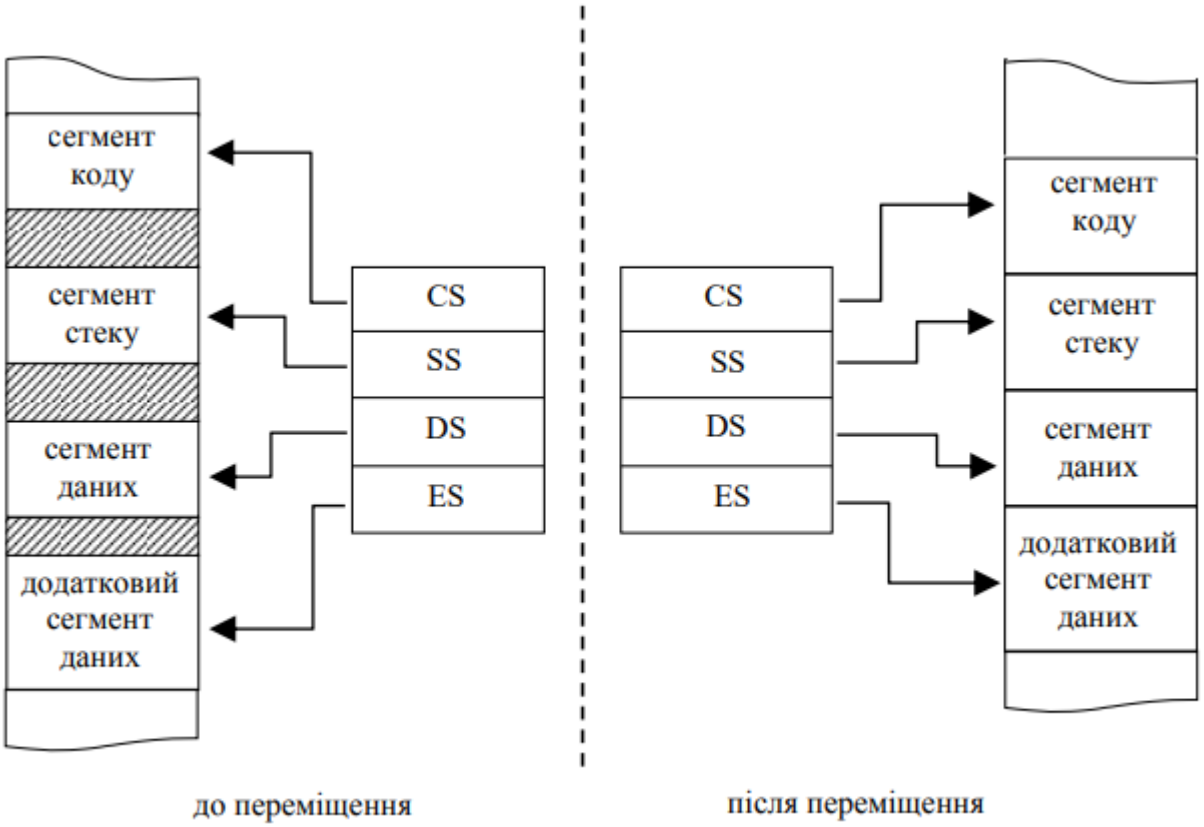
Регістри сегментів

Шість 16-розрядних регістрів містять значення селекторів сегментів, які вказують на поточні адресовані сегменти пам'яті. Нижче перераховані ці регістри. Регістр сегмента програми (CS) - вказує на сегмент, який містить поточну послідовність виконуваних команд. Процесор вибирає всі команди з цього сегменту, використовуючи вміст лічильника команд як відносний адресу. Вміст CS змінюється в результаті виконання внутрішньосегментних команд управління потоком, переривань і виключень. Він не може бути завантажений явним способом. Регістр сегмента стека (SS). Виклики підпрограм, запису параметрів та активізація процедур зазвичай вимагають області пам'яті, що резервується під стік. Всі операції зі стеком використовують регістр SS при зверненні до стека. На відміну від регістру CS регістр SS може бути завантажений явно за допомогою команди програми. Решта чотири регістри є регістрами сегментів даних (DS, ES, FS, GS), кожен з яких адресується поточної виконуваною програмою. Доступ до чотирьох роздільних областей даних має на меті підвищити ефективність програм, дозволяючи їм звертатися до різних типів структур даних. Вміст цих регістрів може бути замінено під керуванням програми.

Вказівник команд

Розширений покажчик команд (EIP) є 32-розрядним регістром. Він містить відносну адресу наступної команди, яка підлягає виконанню. Відносний адресу відраховується від початку сегменту поточної програми. Покажчик команд безпосередньо не доступний програмістові, але він управляється явно командами управління потоком, перериваннями і виключеннями. Молодші 16 біт регістра EIP називаються IP і можуть бути використані процесором незалежно. Це властивість корисно при виконанні команд МП 8086 і 80286, які мають лише регістр IP.

3. Позиційна незалежність програми в однопрограмних ОС.



У ранніх комп'ютерах код залежав від позиції його виконання: кожна програма була побудована для завантаження і запуску з певної адреси у пам’яті. Щоб виконати кілька завдань з використанням окремих програм одночасно, оператор повинен був ретельно планувати завдання так, щоб жодне з двох одночасних завдань не виконувало програми, що вимагають однакових адрес завантаження.

Наприклад, якщо програма розрахунку заробітної плати, і програма розрахунків з дебіторами були створені для роботи за адресою 32K, оператор не міг запускати їх одночасно. Іноді оператор тримав кілька версій програми, кожна з яких була побудована для іншої адреси завантаження, щоб розширити свої можливості.

Щоб зробити процес виконання програм більш гнучкими, був винайдений позиційно-незалежний код. Позиційно-незалежний код може запускатися з будь-якої адреси, за яким оператор вирішив його завантажити. Позиційно-незалежний код використовувався не тільки для координації роботи додатків користувацького рівня, але і всередині операційних систем.

Позиційно незалежний код програми може бути виконаний за будь-якою адресою пам'яті без змін. Це відрізняє його від абсолютного коду, який повинен бути завантажений у певному місці, щоб нормально функціонувати та коду, який можна локалізувати під час завантаження (LTL), в якому лінкер або завантажувач програми модифікує програму перед виконанням, таким чином його можна запустити лише з певного місця пам'яті. Генерування коду, незалежного від позиції, часто є поведінкою за замовчуванням для компіляторів, але вони можуть обмежувати використання деяких мовних функцій, таких як заборона використання абсолютних адрес (код, незалежний від позиції, повинен використовувати відносну адресацію). Інструкції, що посилаються безпосередньо на конкретні адреси пам'яті, іноді виконуються швидше, і заміна їх на еквівалентні інструкції відносної адреси може призвести до дещо повільнішого виконання, хоча для сучасних процесорів цією різницею можна знехтувати.

4. Режими адресування операндів в однопрограмних ОС.

Режими адресування та формат операндів в однопрограмних ОС:

1) Регістровий режим адресування - формат операнда регістр;
2) Безпосередній режим адресування - формат операнда число;
3) Прямий режим адресування - формат операнда мітка та зміщення;
4) Неявний регістровий режим адресування - формат операнда BX, BP, DI, SI;
5) По базі режим адресування - формат операнда BX + зміщення, BP + зміщення;
6) Прямий з індексуванням режим адресування - формат операнда DI + зміщення, SI + зміщення;
6) По базі з індексуванням режим адресування - формат операнда BX DI + зміщення, BX SI + зміщення, BI DI + зміщення, BI SI + зміщення;

Зміщення при адресуванні по базі з індексуванням є необов'язкове. Операнд <регістр>може бути будь-яким 8-або 16-бітовим регістром, крім IP. Операнд <число>може бути 8-або 16-бітовим значенням константи. Операнд <зміщення>може бути 8-або 16-бітовим значенням зміщення зі знаком.

5. Моделі адресування операндів в командах мікропроцесора.

Регістрове адресування

При регістровому адресуванні мікропроцесор вибирає операнд з регістра абозавантажує (записує) операнд в регістр:

MOV AX , CX

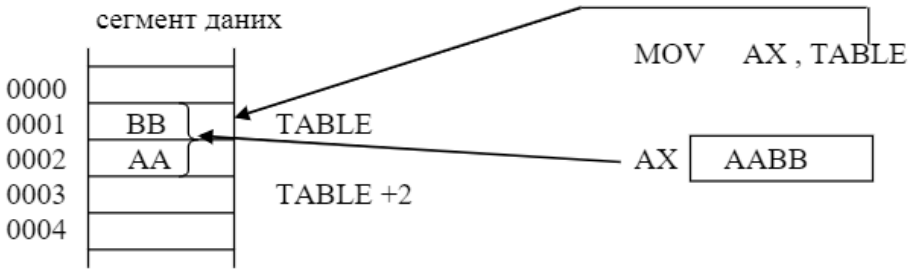
Безпосереднє адресування

Дозволяє вказувати значення константи, як операнда джерела. Така константа записана в самій команді :

MOV CX , 500 K EQV 1024
MOV AL , -30 MOV CX , K

Пряме адресування

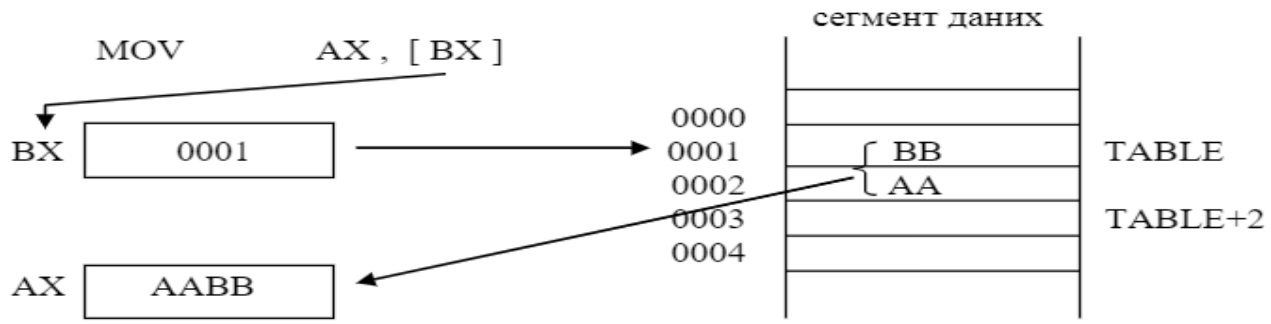
Найчастіше застосовують, коли операндом є окрема комірка пам'яті (проста змінна або константа в пам'яті) :



Неявне регістрове адресування

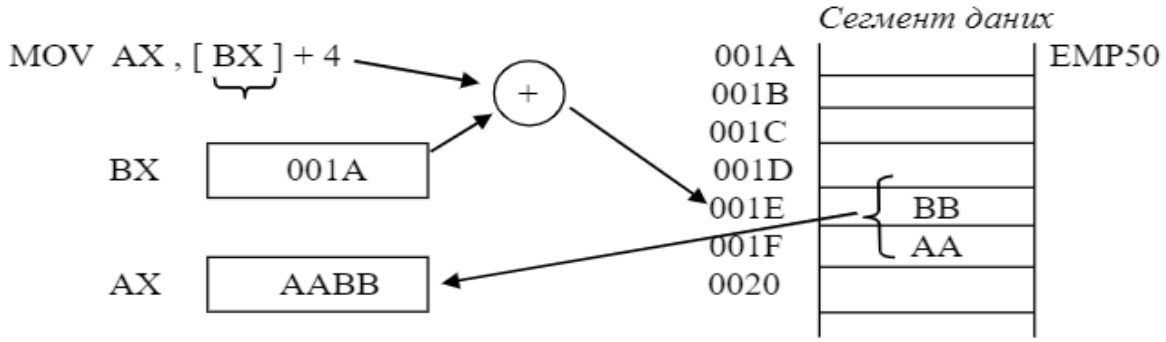
Адреса операнда міститься в одному з регістрів BX, BP, SI, DI :

MOV BX , OFFSET TABLE
MOV AX , [BX]



Адресування по базі

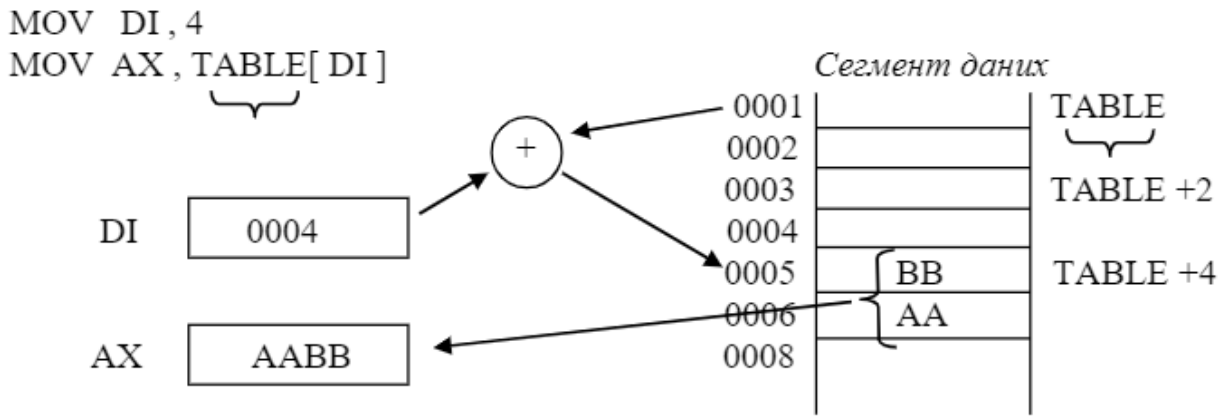
Адреса є сумою вмісту регістра BX або BP та зміщення. Зручно використовувати для доступу до структурованих записів даних, розташованих в різних ділянках пам'яті:



Пряме адресування з індексуванням

Адреса є сумою вмісту регістра SI або DI та зміщення.

Зручно використовувати для доступу до елементів таблиці (вектора):



Адресування по базі з індексуванням

Зручно використовувати для двовимірніх масивів: базовий регістр – початкова адреса, індексний регістр – зміщення до початку відповідного рядка, зміщення – зміщення від початку рядка до потрібного елементу (або зміщення, базовий регістр, індексний регістр відповідно).

Адреса є сумою вмісту базового регістру, індексного регістру і, можливо, зміщення.

Зручно також використовувати для адресування масиву, розташованого в стеку.

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

Головна програма і підпрограма

Головна програма і підпрограма - класичний шаблон програмування. Розділяючи функціонал на модулі (підпрограми), архітектура розділяється на малі частини меншої складності, що полегшує роботу. Головна програма керує основним потоком виконання і може за потреби викликати підпрограми.

Підпрограма — частина програми, яка реалізує певний алгоритм і дозволяє звернення до неї з різних частин загальної (головної) програми.

Підпрограма часто використовується для скорочення розмірів програм в тих задачах, в процесі розв'язання яких необхідно виконати декілька разів однаковий алгоритм при різних значеннях параметрів. Інструкції (оператори, команди), які реалізують відповідну підпрограму, записують один раз, а в необхідних місцях розміщують інструкцію виклику підпрограми.

Набір найбільш уживаних підпрограм утворює бібліотеку стандартних підпрограм.

В більшості мов програмування високого рівня, підпрограми називаються процедурами та функціями. В залежності від мови програмування, терміни «процедура» та «функція» можуть розрізнятися (як правило, процедурою називають підпрограму, що не повертає результату, тоді як функція має результат і може використовуватись як частина виразу) чи розглядатись як синоніми (зокрема, в мові C, де в початковому варіанті всі підпрограми могли повертати результат, їх здебільшого називають функціями). У об'єктно-орієнтованому програмуванні функції-члени класів називають методами.

Приклад програми:

```
<оператор EXTRN, якщо потрібний>

STACK SEGMENT      PARA   STACK 'STACK' ; сегмент стеку
     DB            64  DUP ('STACK _ _ _')        ; ділянка пам'яті стеку
STACK  ENDS

DSEG   SEGMENT PARA        PUBLIC    'DATA'       ; сегмент даних
<      тут записують дані  >
DSEG   ENDS

;      основна програма
CSEG   SEGMENT      PARA   PUBLIC  'CODE'  ; сегмент команд
     ASSUME  CS:CSEG,  DS: DSEG,  SS: STACK  ; зв'язування сегм. регістрів
ENTRY  PROC  FAR    ; функція: точка входу в програму

; формування стеку для правильного повернення в DOS
     PUSH  DS
     SUB  AX,AX
     PUSH  AX
; встановити адресу сегменту даних
     MOV  AX , DSEG
```

```
    MOV  DS,AX
<   тут записують команди програми  >

    RET         ;       повернення  в  DOS
ENTRY ENDP ;        кінець функції
CSEG   ENDS  ;      кінець сегменту команд
    END  ENTRY   ; кінець головної програми і точка входу
```

Приклад підпрограми:

```
PUBLIC  PNAME
<   оператор PUBLIC для змінних сегменту даних, якщо потрібно >

DSEG   SEGMENT PARA  PUBLIC  'DATA'   ; сегмент даних (продовження)
< тут записують дані >
DSEG   ENDS
CSEG   SEGMENT PARA  PUBLIC     'CODE'   ; сегмент команд (продовження)
    ASSUME  CS : SCEG,  DS : DSEG   ; зв'язування сегм. регістрів
PNAME  PROC  NEAR  ; процедура ближнього типу NEAR
<тут записують команди підпрограми >
    RET         ; повернути  в  основну програму
PNAME  ENDP
CSEG   ENDS
    END         ; кінець підпрограми
```

COM-програма

.COM (англ. command) — розширення файлу, що використовувалось у деяких комп'ютерних системах у різних цілях. Програми формату .COM не підтримуються 64-розрядними версіями Windows. В такому випадку, для їхнього запуску можна використати емулятор DOS. COM-програми зазвичай є невеликими застосунками, системними утилітами або невеликими резидентними програмами.

.COM — один із найпростіших форматів виконуваних файлів для процесорів сімейства x86. Програма, завантажена в пам'ять для виконання, є точною копією файла на диску.
Запуск COM-програми в MS-DOS відбувається наступним чином:

1. Система виділяє вільний сегмент пам'яти і заносить його адресу до всіх сегментних регістрів (CS, DS, ES та SS).
2. У перші 256 байтів цього сегмента записується PSP.
3. Безпосередньо за ним завантажується вміст COM-файла без змін.
4. Вказівник стеку (регістр SP) встановлюється на кінець сегмента.
5. У стек записується 0000h (адреса повернення для команди ret).
6. Керування передається за адресою CS:0100h, де знаходиться перший байт виконуваного файла.

Модель пам'яті, використовувану COM-програмами, коли код програми, всі її дані, PSP і стек розміщені в одному сегменті, компілятори високорівневих мов називають TINY (крихітна).

Приклад COM-програми:

```
CODESG          SEGMENT  PARA  'CODE'
```

```
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG,ES:CODESG
        ORG     100H    ;  резерв для PSP-префікса прогр. сегмента
BEGIN:  JMP     MAINPROG    ;       обхід даних
MESS1 DB        'Тестовий приклад: введіть рядок'
        DB      '$'
MAINPROG    PROC  NEAR  ;       процедура ближнього типу  NEAR
        MOV     AH,09  ;        друкувати на екран рядок до '$'
        LEA     DX,MESS1    ; адреса початку рядка запрошення
        INT     21H
        CALL    READLINE    ;  виклик підпрограми
        CALL    WRITELN
        RET
MAINPROG    ENDP

;       процедура читання рядка
READLINEPROC   NEAR
        MOV     AH,0AH      ; прочитати з клавіатури рядок до Enter
        LEA     DX,MAXLEN   ; адреса списку параметрів
        INT     21H     ; виконати DOS-функцію  0A (10)
        RET     ;
READLINE    ENDP

;       пам'ять та дані  для процедури читання рядка
MAXLEN      DB      30          ;  максимальна довжина введеного рядка
REALLEN     DB      ?           ;  кількість фактично введених символів
POLE   DB       30      DUP  '_'    ;  поле ( 30 літер ) для введених символів

;       процедура виведення на екран
WRITELN     PROC  NEAR
        MOV     AH,09
        LEA     DX,MESS2    ;  рядок-повідомлення
        INT     21H
        MOV     AH,40H      ;  друкувати рядок вказаної довжини
        MOV     BX,1        ;  на екран
        MOV     CX,30       ;  30 символів
        LEA     DX,POLE     ;  адреса початку рядка
        INT     21H
        RET
MESS2 DB        'Ви ввели такий рядок : '
        DB      '$'
WRITELN     ENDP
CODESG      ENDS
        END     BEGIN       ;  кінець програми і точка входу
```

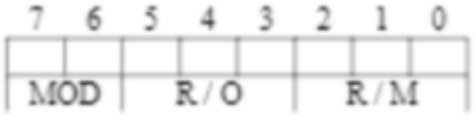
7. Загальний формат бітової структури команди процесора Intel. Поля коду команди.

Загальний формат команди процесора Intel

Команда може мати до шести полів:
1. Префікси – від нуля до чотирьох однобайтових префіксів.
2. Код – один або два байти, що визначають команду.
3. ModRM – 1 байт способу адресування (якщо він потрібний), який описує операнди (нумерація бітів справа наліво починаючи з нуля):
біти 7-6: поле MOD – режим адресування;
біти 5-3: поле R/O – або визначає регістр, або є продовженням коду команди;
біти 2-0: поле R/M – або визначає регістр, або разом з MOD – режим адресування.
4. SIB – 1 байт, якщо він потрібний (розширення ModRM для 32-бітового адресування):
біти 7-6: S – коефіцієнт масштабування;
біти 5-3: I – індексний регістр;
біти 2-0: B – регістр бази.
5. Зміщення – 0, 1, 2 або 4 байти.
6. Безпосередній операнд – 0, 1, 2 або 4 байти.
Значення полів коду команди
У кодах деяких команд зустрічаються спеціальні біти і групи бітів, котрі позначають w, s, d, reg, sreg, cond:
w = 0, якщо команда працює з байтами;
w = 1, якщо команда працює з словами або подвійними словами;
s = 0, якщо безпосередній операнд записаний в команді повністю;
s = 1, якщо безпосередній операнд є молодшим байтом більшого операнда і має розглядатись як число зі знаком;
d = 0, якщо код джерела записаний в полі R/O, а приймача – в полі R/M;
d = 1, якщо код джерела записаний в полі R/M, а приймача – в полі R/O.
Поле reg визначає потрібний регістр і має довжину 3 біти:
Поле sreg визначає потрібний сегментний регістр:
Поле cond визначає умову для команд Jcc, CMOVcc, SETcc, FCMOVcc.

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

Значення полів байта ModRM способу адресування



Поле R/O(біти 5-3) має або додаткові три біти коду команди, або код операнда, який є регістром. В таблицях команд другий випадок позначають reg, а в першому записують потрібні біти.

Поля MOD(біти 7-6) і R/M(біти 2-0) визначають операнд, який може бути як регістром, так і коміркою пам’яті:

MOD= 11, якщо використовується регістрове адресування і поле R/M має код регістра reg;

MOD= 00, якщо використовується адресування в пам’яті без зміщення ([BX+SI]або [EDX]), тобто неявне адресування через регістри;

MOD= 01, якщо використовується адресування в пам’яті з 8-бітовим зміщенням (var[BX+SI]чи [BX+SI]+число);

MOD= 10, –те ж саме, що й MOD=01, тільки зміщення 16-бітове або 32-бітове.

Значення поля R/M відрізняється в 16-і 32-бітових режимах.

R/M в 16-бітовому режимі:

000 –[BX + SI]

001 –[BX + DI]

010 –[BP + SI]

011 –[BP + DI]

100 –[SI]

101 –[DI]

110 –[BP](крім MOD=00–в цьому випадку після байта ModRM записується 16-бітове зміщення, тобто використовується пряме адресування: ADD DX, Table)

111 –[BX]

R/M в 32-бітовому режимі:

000 –[EAX]

001 –[ECX]

010 –[EDX]

011 –[EBX]

100 –використовується байт SIB

101 –[EBP](крім MOD=00–в цьому випадку після байта ModRM записується 32-бітове зміщення, тобто використовується пряме адресування: ADD EDX, Table)

110 –[ESI]

111 –[EDI]

r/m	mod=00	mod=01	mod=10	mod=11 w=0	mod=11 w=1
000	BX+SI	BX+SI+disp	BX+SI+disp	AL	AX
001	BX+DI	BX+DI+disp	BX+DI+disp	CL	CX
010	BP+SI	BP+SI+disp	BP+SI+disp	DL	DX
011	BP+DI	BP+DI+disp	BP+DI+disp	BL	BX
100	SI	SI+disp	SI+disp	AH	SP
101	DI	DI+disp	DI+disp	CH	BP
110	Direct	BP+disp	BP+disp	DH	SI
111	BX	BX+disp	BX+disp	BH	DI

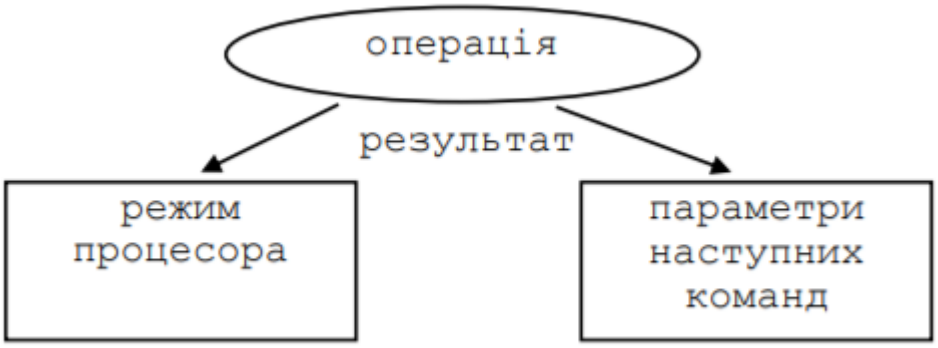
(r/m=110, mod=00) - для прямого адресування типу MOV AX, TABLE

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

Схеми виконання команд мікропроцесора

Команда може мати нуль, один, два або три операнди. Загальні правила визначають способи адресування операндів і місце для результату виконання команди.

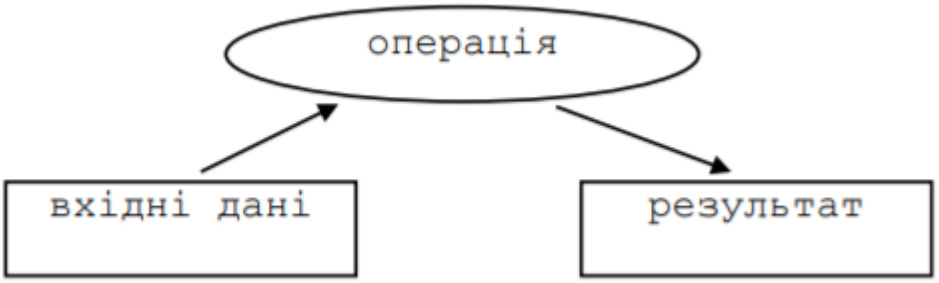
1) Команди без операндів (нуль-операндні). Такі команди призначені для керування режимом роботи цілого процесора або способом виконання наступних команд.



Приклади команд:

REP	префікс; означає повторення рядкової операції наступної команди ECX разів; різновиди: REPZ (REPE) – виконувати так само, поки виконується умова рівності (ZF=1), закінчити повторення, якщо після чергової ітерації ZF=0 (до знайдення іншого значення); REPNZ (REPNE) – так само, але закінчити при ZF≠0 (до знайдення такого самого значення)
CLD	ознака напряму DF регістра ознак встановлюється в нуль; це означає автоматичне збільшення SI і DI на 1 за кожною ітерацією операцій над рядками
STD	DF=1, SI і DI зменшуються на 1 за кожною ітерацією

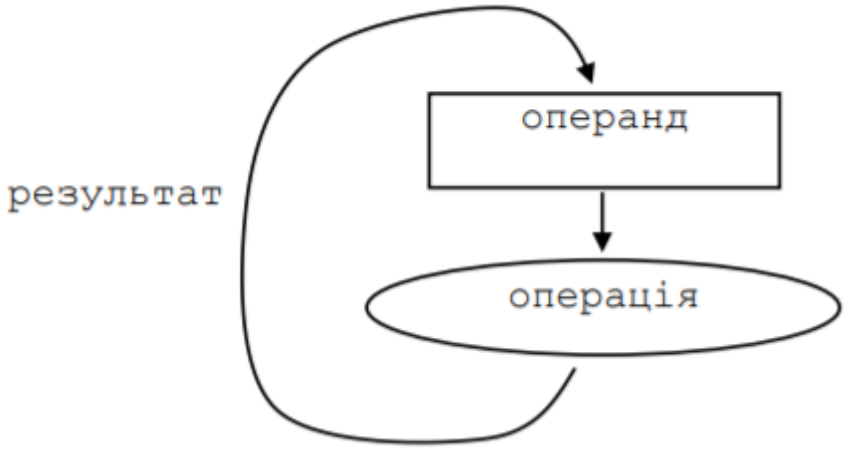
2) Нуль-операндні команди з неявними операндами, визначеними окремо. Такі команди виконують групу елементарних операцій за одне виконання, самі операнди визначають попередньо або зафіксовані командою.



Приклади команд:

XLAT	завантажити в AL байт з таблиці сегменту даних, на початок якої вказує EBX (BX), а початкове значення AL є зміщенням; тобто, це є заміна одного коду іншим за таблицею
PUSHAD	записати в стек регістри EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
CWD	розширення слова (AX) до подвійного слова (DX:AX) з копіюванням знакового біту
CWDE	розширення слова (AX) до подвійного слова (EAX) з копіюванням знакового біту

3) Команди операцій з одним операндом. Команда опрацьовує заданий операнд відповідно до свого призначення, результат записує на місце операнда. Отже, після виконання команди операнд буде модифікований.



Приклади команд:

INC EAX	інкремент операнда
NEG testvalue	змінити знак операнда

4) Команди дії за одним операндом. Такі команди виконують задану дію, використовуючи операнд. Сам операнд не міняється.

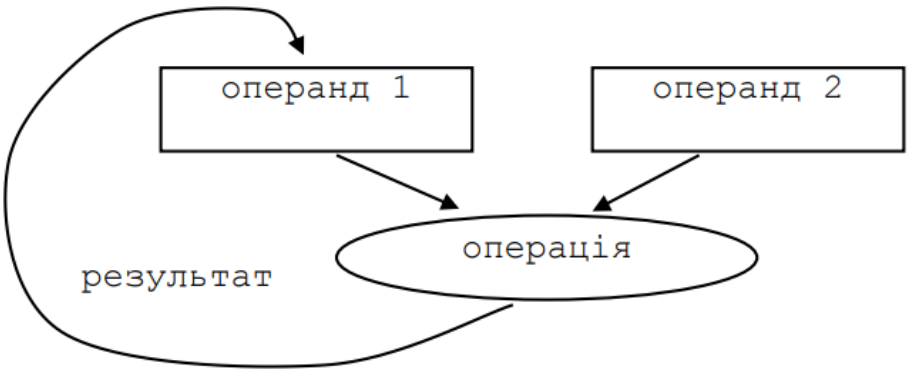
Приклади команд:

PUSH EDI	записати в стек значення регістра
POP memx	прочитати з стеку слово чи подвійне слово
JMP target	перейти в програмі до вказаної адреси (різні форми адресування)
JGE / JNL target	перейти, якщо більше або рівне
CALL target	передача керування процедурі за вказаною адресою; в стек зберігається адреса команди, наступної за CALL-командою
INT n	двобайтова команда; спочатку в стек записує регістр ознак, після нього – повну адресу повернення; крім того, скидає ознаку TF; після цього виконує неявний перехід через n-й елемент дескрипторної таблиці переривань

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

Команди з двома операндами:

Результат записують на місце першого операнда, отже, він набуває нового значення, тому порядок запису операндів команди є важливий

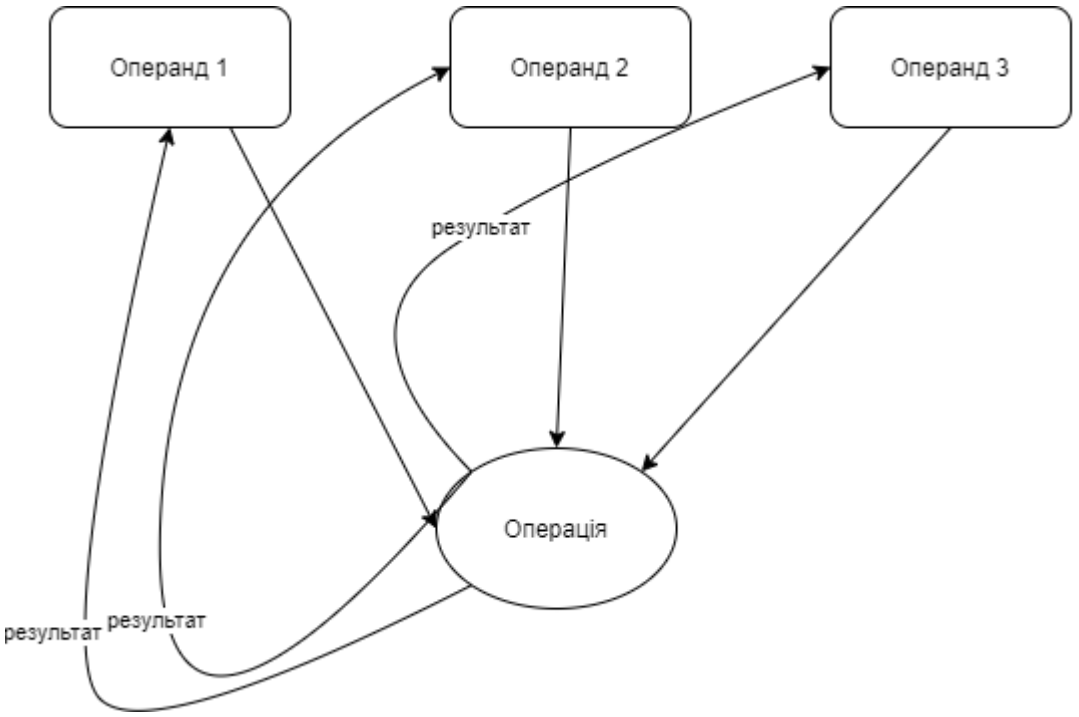


Приклад команд:

ADD EDX,alfa	додавання, результат – в регістрі EDX
OR beta,ECX	логічне порозрядне «або», результат – в пам’яті за адресою beta

З трьома операндами:Такі команди є лише для окремих різновидів операцій.

Результат буде на місці операнда, визначеного структурою команди
Схему я сам придумав, в лекції не було, хз чи правильна:



IMUL regDWdst, regDWsrc | memDWsrc, immDW	множення зі знаком
SHLD / SHRD regDWdest | memDWdest, regDWsrc, count	зсув вліво / вправо
`regDWdest ← (regDWdest regDWsrc) SHLD на count`	спочатку об’єднати біти

11. Директиви асемблера та їх застосування.

Директиви призначені для керування процесом асемблювання і формування листингу. Вони діють тільки в процесі асемблювання програми і не переводяться в машинні коди. Мова асемблеру містить такі основні директиви:

- керування моделями пам’яті MODEL та сегментами DATA, CONST, STACK, CODE;
- початку і кінця сегмента SEGMENT та ENDS;
- початку і кінця процедури PROC та ENDP;
- визначення пам’яті та призначення сегментів ASSUME;
- початку ORG;
- визначення даних DB, DW, DD, DF, DP, DQ, DT;
- символьних констант: = , EQU, TEXTEQU;
- завершення програми END;
- відзначення LABEL.

12. Трансляція програм ASM за першим і другим переглядом.

За першим переглядом транслятор послідовно рядок за рядком переглядає текст програми. При цьому виконує таку роботу:

1) керує лічильником поточної адреси LOCCTR;

2)обчислює довжину кожної команди чи даних в байтах;

3)відстежує адресу (зміщення) кожної команди і даних в сегменті;

4) виконує основне перетворення в машинні коди за кожною командою чи даними;

5) будує таблицю символів SYMTAB з адресами всіх іменованих ділянок пам'яті, які будуть дописані в коди команд за другим переглядом. За другим переглядом транслятор виконує підстановку обчислених адресів операндів пам'яті, які побудовані в таблиці SYMTAB в результаті першого перегляду.

Коротко, при першому перегляді здійснюється розподіл пам'яті і надання значень символічним іменам, при другому — формується робоча програма у вигляді об'єктного файлу.

13. Алгоритм першого перегляду асемблера при трансляції.

Асемблер - це програма для перетворення інструкцій, написаних низькорівневим кодом збірки, у машинний код, що переміщується, та генерування інформації для навантажувача.

Асемблер генерує інструкції шляхом оцінки мнемотехніки (символів) в операційному полі та знаходить значення символу та літералів для створення машинного коду. Якщо асемблер робить всю цю роботу за одне сканування, це називається однопрохідним асемблером, інакше, якщо він виконує багаторазове сканування, то називається багатопрохідним асемблером. Ми розглянемо другий варіант, при якому асемблер виконує роботу за два проходи.

Прохід 1:

1. Визначає символи та літерали та записує їх у таблиці символів та таблиці літералів відповідно.
2. Відстежує лічильник місцезнаходжень
3. Обробляє псевдо операції

Прохід 2:

1. Генерує об'єктний код, перетворивши символічний код-код у відповідний числовий код-код
2. Створює дані для літералів та шукає значення символів

Примітка: Якщо під час першого проходу виявлено помилку, процес складання завершується і не продовжується до другого проходу. Якщо це трапляється, список асемблера містить лише помилки та попередження, створені під час першого проходження асемблера. Нижче подано код для виконання першого проходу із коментарями:

```
begin { pass 1 }
  записати 0 в LOCCTR; OKEY:=true прочитати перший
рядок тексту програми while(OPCODE<>"END" and
OKEY) do begin
    if це не рядок коментар then begin if є поле мітки
    then begin

       пошук мітки в SYMTAB
       if знайшли then begin
          встановити ознаку помилки-повторне визначення OKEY:=false
        end
```

```
        else занести (< мітка >,LOCCTR) в SYMTAB end
    пошук OPCODE в OPTAB
    if знайшли then begin
        аналіз поля операндів
        if формат правильний then begin визначення формату та
            довжини команди додати до LOCCTR довжину
            команди
        end
        else OKEY:=false (помилка воператорах)
    if OKEY then перевірка наявності та птавильності комантаря end { знайшли код
    операції }
    else if OPCODE="ORG" then LOCCTR:=< число >{ операнд ORG } else if
    OPCODE="DW" then додати 2 до LOCCTR
    else if OPCODE="DB"          then begin
            аналіз поля операндів DB, визначення довжини додати
            довжину до LOCCTR
            end
    else встановити ознаку помилки - неправильний код операції (OKEY:=false)
    end { if це не рядок-коментар }
    записати рядок впроміжний файл
    прочитати наступний рядок тексту програми
end { while }
if not OKEY then сформувати діагностику про помилку
              else begin записати останій рядок в проміжний файл запамятати
                LOCCTR як довжину програми
                end
end { pass1 }
```

14. Алгоритм другого перегляду асемблера при трансляції.

За другим переглядом транслятор виконує підстановку обчислених адресів операндів пам’яті, які побудовані в таблиці SYMTAB в результаті першого перегляду.

```
begin { pass2 }

сформувати запис-заголовок в об'єктній програмі прочитати
перший рядок з проміжного файлу ініціалізувати перший запис
тіла програми OKEY:=true

      while(OPCODE<>"END") and OKEY do begin

     if це не рядок-коментар then

        begin пошук OPCODE в OPTAB

       if знайшли then begin

       аналіз поля операндів

          if в полі операндів є імя then begin пошук імені в
            SYMTAB
```

if знайшли **then** взяти з SYMTAB адресу **else**
OKEY:=**false**(не визначене імя)

end

перевести команду в об'єктне зображення

end (**if** знайшли)

else if OPCODE="DW" або "DB" **then**

перетворити константу або рядок в об'єктне зображення занести об'єктний код в запис тіла програми ініціалізувати новий запис тіла програми

end { **if** це не рядок-коментар }

записати у файл для роздруку черговий рядок прочитати
наступний рядок з промідного файлу

end { **while** }

if not OKEY **then** сформувати дфагностику про помилку

else begin

занести запис-кінець в об'єктну програму надрукувати
останній рядок в файл роздруку

end
end { pass2 }

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

Статичне компонування виконуваних файлів має низку недоліків.

- Якщо декілька різних прикладних програм використовують спільний код (наприклад, коди функцій бібліотеки мови С),кожний виконуваний файл буде мати окрему копію цього коду; в результаті такі файли займатимуть більше місця на диску і в пам’яті.Варто мати можливість зберігати на диску і завантажувати в пам’ять лише одну копію спільного коду.
- У разі кожного оновлення прикладну програму потрібно повністю перекомпілювати, перекомпонувати, повторно тестувати і перевстановити.
- Неможливо реалізувати динамічне завантаження програмного коду під час виконання, наприклад, якщо потрібно реалізувати модульну структуру програми, подібно до того, як це зроблено в ядрі Linux .

Для вирішення цих і подібних проблем було запропоновано концепцію динамічного компонування з використанням динамічних або розділюваних бібліотек (dynamic - link libraries (DLL), shared libraries).

Динамічне компонування у Windows. Особливості реалізації

- У Windows динамічне компонування здійснює система, а не окремий динамічний компонувальник; очевидно, що всередині виконуваного файла ім'я такого компонувальника не задають.

- Окрема секція.edata виділена для опису експортованих символів. Це досить важлива відмінність від ELF, для якого всі символи за замовчуванням є експортованими.
- Відмінності має і секція імпортованих функцій. Тут створена спеціальна таблиця, що визначає всі імпортовані функції; після запуску її заповнюють адресами функцій з DLL. Таку таблицю називають таблицею імпорту адрес (IAT). Використання IAT дає змогу звести всю інформацію про імпортовані адреси в одне місце у файлі.

Зазначимо, що як і для ELF, відмінностей між динамічними бібліотеками і виконуваними файлами із погляду формату файла немає, фактично їх розрізняють за значенням поля типу в заголовку.

Виконувані файли в Linux. Формат ELF

Об'єктні файли містять також спеціальну секцію .reloc з даними для налаштування адрес. Крім цього, можна виокремити кілька секцій, що зберігають інформацію для системного завантажувача і динамічного компонувальника, що буде розглянуто окремо. Застосування можуть задавати і свої власні секції. Зазначимо, що в термінології ELF секція означає поіменований розділ у виконуваному файлі; після відображення у пам'ять секції відповідає сегмент.

16. Компонувальники і принципи їх роботи.

Компілятор не розв’язує зовнішні посилання, а отже, не може створити виконуваний файл. Це робота компонувальника (linker).

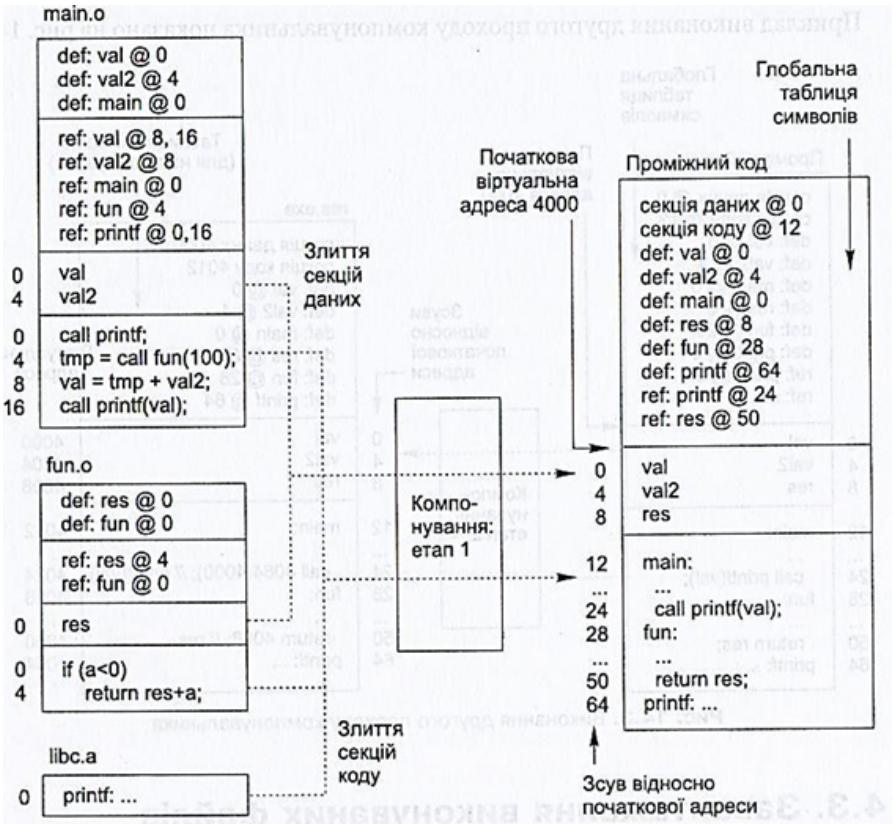
Основні функції компонувальника:

- об’єднати всі частини програми у виконуваний файл;
- зібрати разом код і дані секцій однакового призначення з різних об’єктних файлів;
- визначити остаточно всі адреси для коду і даних, розв’язуючи при цьому зовнішні посилання.

В результаті за статичного компонування на диск записують виконуваний файл, готовий до запуску (EXE-файл), за динамічного – виконуваний файл так само буде створений, але йому для виконання потрібні додаткові файли.

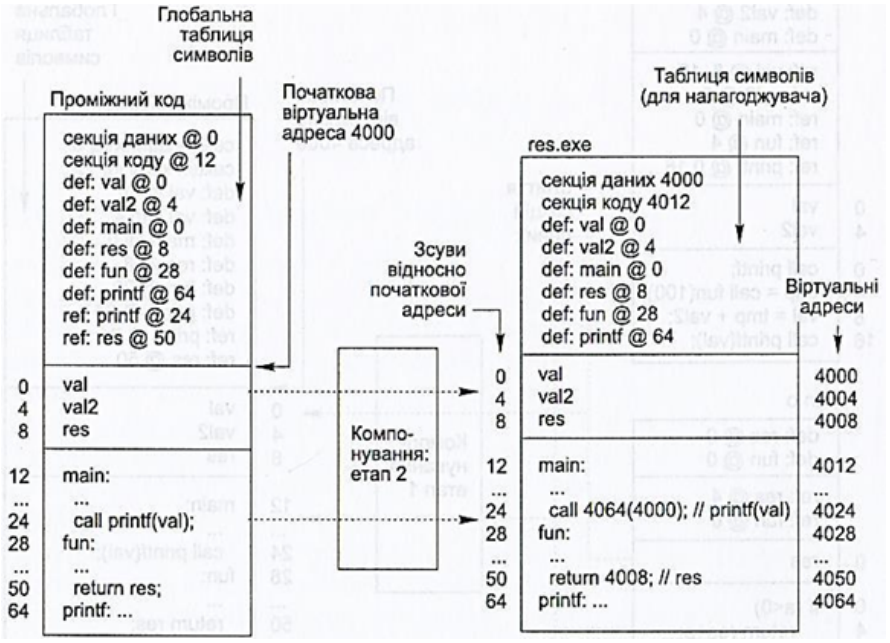
Головне завдання першого проходу компонувальника – визначити відносні адреси, за якими треба розташувати задані в коді об’єкти.

Приклад виконання першого проходу показано на рисунку:



Головне завдання другого проходу компонувальника – корекція всіх адрес в об'єктному коді та розв'язання всіх зовнішніх посилань. Перевіряє, щоб кожний символ мав тільки одне визначення (використати можна багато разів), і виконує корекцію посилань фактичними адресами, з врахуванням отриманої початкової віртуальної адреси.

Приклад виконання другого проходу показано на рисунку:



Список питань до розділу "Бібліотеки DLL"

17. Переваги і недоліки статичного компонування програм.

Статичне компонування виконуваних файлів має низку недоліків.

1) Якщо декілька різних прикладних програм використовують спільний код (наприклад, коди функцій бібліотеки мови С), кожний виконуваний файл буде мати окрему копію цього коду; в результаті такі файли займатимуть більше місця на диску і в пам’яті. Варто мати можливість зберігати на диску і завантажувати в пам’ять лише одну копію спільного коду.

2) У разі кожного оновлення прикладну програму потрібно повністю перекомпілювати, перекомпонувати, повторно тестувати і перевстановити.

3) Неможливо реалізувати динамічне завантаження програмного коду під час виконання, наприклад, якщо потрібно реалізувати модульну структуру програми, подібно до того, як це зроблено в ядрі Linux.

Переваги:

1) Всі компоненти програми знаходяться в одному місці, не потрібно шукати реалізацію мевної функції в інших проектах.
2) Займає менше дискового простору
3) Менший час завантаження застосування, адже відображення кожного файла в адресному просторі забирає час
4) Не потрібно боятися проблеми зворотної сумісності динамічних бібліотек

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

Динамічна бібліотека – набір функцій та програмних ресурсів, скомпонованих разом в бінарний файл, який можна динамічно завантажити в адресний простір процесу, що використовує ці функції.

Динамічне завантаження (dynamic loading) – завантаження під час виконання процесу, зазвичай реалізоване як відображення файла бібліотеки в його адресний простір.

Динамічне компонування (dynamic linking) – компонування образу виконуваного файла під час виконання процесу з використанням динамічних бібліотек.

19. Переваги і недоліки використання динамічних бібліотек.

Переваги і недоліки використання динамічних бібліотек

Переваги

- Оскільки бібліотечні функції містяться в окремому файлі, розмір виконуваного файла стає меншим. Якщо врахувати, що є динамічні бібліотеки, які використовують майже всі застосування у системі (стандартна бібліотека мови С у Linux, бібліотека підсистеми Win32 у Windows), то очевидно, що так заощаджують дуже багато дискового простору.

- Якщо динамічну бібліотеку використовують кілька процесів, у пам'ять завантажують лише одну її копію, після чого сторінки коду бібліотеки відображаються в адресний простір кожного з цих процесів. Це дає змогу ефективніше використовувати пам'ять.

- Оновлення застосування може бути зведене до встановлення нової версії динамічної бібліотеки без необхідності перекомпонування тих його частин, які не змінилися.

- Динамічні бібліотеки дають змогу застосуванню реалізувати динамічне завантаження модулів на вимогу. На базі цього може бути реалізований розширюваний API застосування. Для додавання нових функцій до такого API стороннім розробникам достатньо буде створити і встановити нову динамічну бібліотеку, яка підлягає певним правилам.

- Динамічні бібліотеки дають можливість спільно використовувати ресурси застосування (наприклад, така бібліотека може містити спільний набір піктограм), крім того, вони дають змогу спростити локалізацію застосування (якщо всі рядки, які використовуються програмою, помістити в окрему DLL, для заміни мови застосування достатньо буде замінити тільки цю DLL).

- Оскільки динамічні бібліотеки є двійковими файлами, можна організувати спільну роботу бібліотек, розроблених із використанням різних мов програмування і програмних засобів, що спрощує створення застосувань на основі програмних компонентів (отже, динамічне компонування лежить в основі компонентного підходу до розробки програмного забезпечення).

Недоліки

Динамічні бібліотеки не позбавлені недоліків (хоча вони тільки в окремих випадках виправдовують використання статичного компонування).

- Використання DLL сповільнює завантаження застосування. Що більше таких бібліотек потрібно процесу, то більше файлів треба йому відобразити у свій адресний простір під час завантаження, а відображення кожного файла забирає час. Для прискорення завантаження рекомендують укрупнювати DLL, об'єднуючи кілька взаємозалежних бібліотек в одну загальну.

- У деяких ситуаціях (наприклад, під час аварійного завантаження системи із дискети) використання спільних системних DLL неприйнятне через нестачу дискового простору для їхнього зберігання (такі системні DLL, подібно до стандартної бібліотеки мови C, можуть займати кілька мегабайтів дискового простору, при цьому застосування часто потребують усього по кілька функцій із них). У такій ситуації найчастіше використовують версії застосувань, статично скомпоновані таким чином, щоб у їхні виконувані файли був включений зі стандартних бібліотек код лише тих функцій, які їм потрібні.

- Найбільшою проблемою у використанні динамічного компонування є проблема зворотної сумісності динамічних бібліотек. Розглянемо її окремо.

20. Зворотня сумісність динамічних бібліотек.

Динамічні бібліотеки не мають вбудованого механізму зворотної сумісності. Проблеми можуть виникнути коли програма встановлює нову версію DLL поверх попередньої. Якщо нова версія бібліотеки не має зворотньої сумісності з попередніми, така програма може перестати працювати.Досягти такої сумісності досить складно, особливо коли попередня версія містила відомі помилки, і прикладні програми, що використовують бібліотеку, розробили код їхнього обходу –виправлення помилки у бібліотеці може зробити код прикладної програми невірним (кажуть, що в цьому разі порушується сумісність за помилками –bug-to-bug compatibility).

Проблеми також виникають в тому, що ОС Windows не дозволяє одночасно завантажувати в пам’ять різні версії однієї бібліотеки. Для всіх завантажених бібліотек виділяли спільний каталог, через що динамічні бібліотеки перезаписували одна одну, можливим був навіть перезапис новішої версії старішою.Не зберігали в динамічних бібліотеках і застосуваннях інформації про точні версії бібліотек, від яких вони залежать. Проте варто зазначити що дану проблему вирішено в програмах, що працюють на .NET, оскільки .NET використовує так званий GAC (Global Assembly Cache, глобальний кеш збірок). Він дозволяє тримати різні версії однієї бібліотеки і використовувати потрібну.

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

Є два основні способи завантаження динамічних бібліотек в адресний простір процесу – неявне і явне зв'язування (implicit і explicit binding). Неявне – основний спосіб завантаження динамічних бібліотек у сучасних ОС. При цьому бібліотеку завантажують автоматично до початку виконання застосування під час завантаження виконуваного файла, за це відповідає завантажувач виконуваних файлів ОС. У деяких системах такий завантажувач є частиною ядра ОС, у деяких – окремим застосуванням. Список бібліотек, потрібних для завантаження, зберігають у виконуваному файлі. До переваг цього методу належать: ► простота і прозорість з погляду програміста (йому не потрібно писати код завантаження бібліотек, достатньо у налаштуваннях компонувальника вказати список бібліотек, які йому потрібні); ► висока ефективність роботи процесу після початкового завантаження (усі необхідні бібліотеки до цього часу вже завантажені у його адресний простір). Недоліком неявного зв'язування можна вважати зниження гнучкості (так, наприклад, якшо хоча б однієї з необхідних бібліотек не буде на місці, процес завантаження не обійдеться без проблем, навіть коли для виконання конкретної задачі ця бібліотека не потрібна). Крім того, збільшуються час завантаження і початковий обсяг необхідної пам'яті.
Альтернативним для неявного є явне зв'язування, коли динамічну бібліотеку завантажують в адресний простір процесу виконанням системного виклику з його коду. Після цього, використовуючи інший системний виклик, застосування отримує адресу необхідної йому функції бібліотеки і може її викликати. Після використання бібліотеку можна вилучити з пам'яті. Компонувальних при цьому нічого про неї не знає, завантажувач ОС автоматично бібліотек не завантажує (здебільшого неявне зв'язування зводиться до автоматичного виконання тих самих викликів, які сам програміст виконує за явного). Такий підхід вимагає від програміста додаткових зусиль, але має більшу гнучкість.

22. Взаємодія динамічної бібліотеки з адресним просторостанм процесу. Особливості об'єктного коду динамічних бібліотек.

Після відображення динамічної бібліотеки в адресний простір процесу вона стає повністю видимою для виконуваного програмного коду. Функції бібліотеки доступні для всіх потоків процесу. Під час відображення бібліотеки у пам'ять використовують технологію копіювання під час записування, тому кожен процес матиме свою копію стека і даних бібліотеки.
Для коду функцій з бібліотеки доступні ресурси процесу такі як дескриптори відкритих файлів та стек потоку, що викликав цю функцію. Код динамічної бібліотеки у багатопотокових застосунках повинен бути безпечним з точки зору потоків: не рекомендується звільняти пам’ять, що виділена поза бібліотечним кодом за допомогою цього бібліотечного коду.
Особливості об'єктного коду динамічних бібліотек:
Код динамічних бібліотек зазвичай зберігають у виконуваних файлах формату, стандартного для цієї ОС (далі побачимо, що виконувані файли і файли DLL можуть відрізнятися тільки одним бітом у заголовку), але з погляду характеру цього коду є одна важлива відмінність між кодом DLL і кодом звичайних виконуваних файлів. Вона полягає в тому, що код DLL в один і той самий час повинен мати

можливість завантажуватися за різними адресами. Для того щоб це було можливо, такий код потрібно робити позиційно-незалежним. Позиційно-незалежний код завжди використовує відносну адресацію (базову адресу додають до зсуву). Базову адресу налаштовують у момент завантаження DLL в адресний простір процесу і називають також базовою адресою бібліотеки; такі адреси відрізняються для різних процесів. Зсув у цьому разі називають внутрішнім зсувом об'єкта.

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

Одна із функцій динамічної бібліотеки може бути позначена як її точка входу. Така функція автоматично виконуватиметься завжди, коли цю DLL відображають в адресний простір процесу (явно або неявно); у неї можна поміщати код ініціалізації структур даних бібліотеки. Багато систем дають змогу задавати також і функцію, що буде викликана в разі вивантаження DLL із пам'яті

функцію точки входу можна використовувати для ініціалізації або знищення структур даних відповідно до вимог DLL. Крім того, якщо програма багатопотокова, то вона дозволяє використовувати локальну пам’ять потоку (TLS). Для виділення пам’яті, приватну для кожного потоку, також зручно використовувати функцію точки входу.

Точку входу в DLL у Win32 API описують як функцію:
BOOL WINAPI DllMain(HINSTANCE libh, DWORD reason, LPVOID reserved);

Першим параметром для неї є дескриптор екземпляра бібліотеки, другим – індикатор причини виклику (DLL_PROCESS_ATTACH – під час завантаження бібліотеки, DLL_PROCESS_DETACH – у разі її вивантаження), третій параметр не використовують.

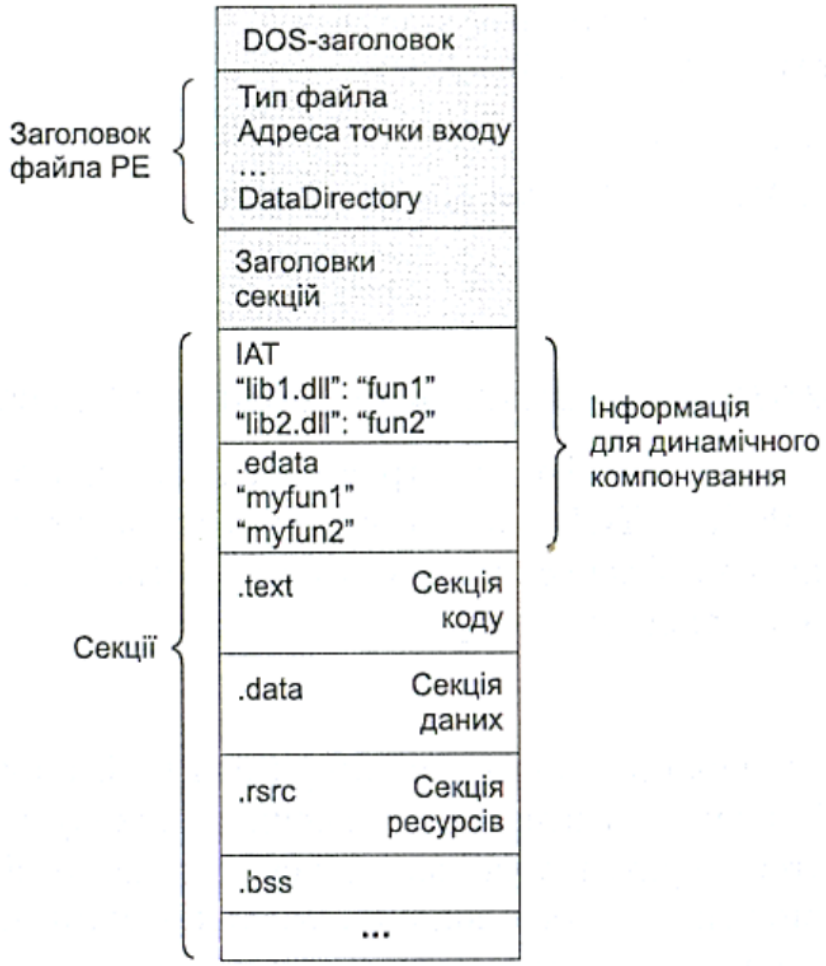
```
BOOL WINAPI DllMain ( HINSTANCE libh, DWORD reason, LPVOID reserved ) {
    switch (reason) {

            case DLL_PROCESS_ATTACH: printf("завантаження DLL\n"); break;
            case DLL_PROCESS_DETACH: printf("вивантаження DLL\n"); break;
            case DLL_THREAD_ATTACH: printf("процес створює новий потік DLL\n"); break;
            case DLL_THREAD_DETACH: printf("потік виходить без помилок\n"); break;
    }

    return TRUE;
}
```

Функція точки входу повинна виконувати лише прості завдання ініціалізації та не повинна викликати будь-які інші функції завантаження або завершення DLL. Наприклад, у функції точки входу не слід прямо чи опосередковано викликати функцію LoadLibrary або функцію LoadLibraryEx. Крім того, не слід викликати функцію FreeLibrary, коли процес завершується.

ПОПЕРЕДЖЕННЯ - У багатопотокових програмах необхідно переконатися, що доступ до глобальних даних DLL синхронізований (безпечний для потоків), щоб уникнути можливого пошкодження даних. Для цього потрібно використовувати TLS, щоб надати унікальні дані для кожного потоку.

24. Структура виконуваних файлів для Windows. Формат PE.

Файли формату PE (Portable Executable) з'явилися разом із Windows NT. Вони мають багато спільного із традиційними форматами UNIX-систем (об'єктні файли, на основі яких будують такий файл, можуть бути у форматі COFF –одному зі стандартних форматів об'єктних файлів у UNIX).Загальна структура файла в форматі PE наведена на рисунку.



Основна подібність між двома форматами полягає в їхній секційній організації та описі списку секцій таблицею секцій. Зазначимо, що навіть назви і призначення багатьох основних секцій збігаються –це стосується .text, .data, .bss, .reloc. Окрема секція, як і в ELF, виділена під опис списку імпорту. Основні відмінності між двома форматами наведені нижче. Перед основним заголовком PE-файла поміщають так званий DOS-заголовок. Він містить невелику програму, яка у разі запуску під MS-DOS виводить повідомлення і завершується. Цей заголовок залишився відтоді, коли PE-файли часто намагалися запускати під керуванням MS-DOS або Windows 3.x. Під час звичайного запуску (під Windows) керування негайно передають за адресою початку виконання, заданою в основному заголовку. Інформацію, необхідну для виконання файла, не виносять в окрему структуру даних, а зберігають у стандартних структурах –основному заголовку, заголовках секцій тощо. Для зручності доступу адреси найважливіших об'єктів усередині файла (секцій експорту, імпорту, ресурсів тощо) утримують у заголовку файла як окремий масив DataDirectory. Окрема секція .rsrc зарезервована для зберігання ресурсів програми. Ресурси всередині цієї секції мають деревоподібну організацію з каталогами і підкаталогами.

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

Для реалізації неявного зв'язування все, що потрібно від розробника застосування,–це вказати компонувальнику список необхідних DLLi впевнитися, що під час виконання всі вони можуть бути знайдені. Завантажувач ОС забезпечить пошук і виконання потрібного коду. Зауважимо, що якщо під час завантаження DLL виникла помилка, весь процес завантаження переривається. До початку виконання основного потоку процесу всі необхідні DLL мають бути відображені в його адресний простір.
У розробці DLL, на відміну від UNIX, основним завданням програміста є задання списку експортованих функцій. Цього можна домогтися такими способами:

- створити спеціальний файл із розширенням.DEF, у якому перелічити всі такі функції,іпередати його компонувальнику;
- у разі використання засобів компілятора C++ скористатися спеціальною конструкцією declspec(dllexport), яку поміщають перед оголошенням експортованої функції:

```
_declspec ( dllexport ) DWORD fun() {
        printf("Виклик fun()\n");
        return 100;
}
```

При цьому додатково до створення DLL компонувальник згенерує спеціальний файл заглушок–статичну бібліотеку з розширенням.LIB, яку компонують з клієнтським застосуванням і яка містить код заглушок для створення зв'язків з DLL під час завантаження.Ім'я цієї бібліотеки має бути явно задана як один із параметрів виклику компонувальника підчас компонування клієнтського застосування.
У разі використання функцій з DLL їх, на відміну від UNIX, потрібно імпортувати. Це може мати такий вигляд:

```
_declspec(dllimport) DWORD fun();
void main(){
    printf("%d\n",fun()):
}
```

Тоді під час компонування будуть узяті функції з.LIB-файла, а в разі виконання заглушки звертатимуться до справжніх функцій з DLL.

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

Альтернативним для неявного є явне зв'язування, коли динамічну бібліотеку завантажують в адресний простір процесу виконанням системного виклику з його коду. Після цього, використовуючи інший системний виклик, застосування отримує адресу необхідної йому функції бібліотеки і може її викликати. Після використання бібліотеку можна вилучити з пам'яті. Компонувальник при цьому нічого про неї не знає, завантажувач ОС автоматично бібліотек не завантажує, отже компіляція і запуск застосування відбуваються без клопотів щодо пошуку і підключення бібліотеки.
Явне зв'язування здебільшого зводиться до ручного виконання тих самих викликів, які виконуються автоматично за неявного. Такий підхід вимагає від програміста додаткових зусиль, але має більшу гнучкість.

Для реалізації явного зв’язування треба виконати такі кроки:

Крок 1. Визначити місце розташування та ім'я бібліотеки DLL:

```
char AllocDLL[] = "d:\\MyDLLfun\\";
```

```
char LibName[] = "DLLCreate.dll";
```

Крок 2. Оголошення типів вказівників на зовнішні функції DLL:

```
typedef int (CALLBACK* TReplLt)(char*, char, char);

typedef double (CALLBACK* TMinDset)
                    (int, double[], double[], double, double,int*);

//     #define  CALLBACK    __stdcall
```

Крок 3. Завантажити і перевірити наявність бібліотеки:

```
HINSTANCE hDLL;     // дескриптор бібліотеки

int buffersize = strlen(AllocDLL) + strlen(LibName) + 2;

// розмір пам'яті для повного імені DLL


char * FullName = new char[buffersize]; // місце для імені DLL strcpy_s(FullName, buffersize,
AllocDLL); // копіювати шлях до файла strcat_s(FullName, buffersize, LibName);

// додати ім'я файла = повне ім'я

hDLL = LoadLibrary(FullName);   // завантаження бібліотеки

// перевірка присутності

if (hDLL == NULL) {    // якщо немає

      cout << "DLL not found.\n";

        system("pause");  return 0;          //          то припинити програму

    }
```

Крок 4. Побудувати вказівники на функції і перевірити наявність функцій в бібліотеці:

```
TReplLt ReplLt = (TReplLt)GetProcAddress(hDLL, "_ReplLt@12");

//  ім'я змінено

if (!ReplLt) {

        cout << "Function ReplLt not found.\n"; FreeLibrary(hDLL);
        system("pause"); return 0;
        // якщо немає - то вихід

    }
```

Тут потрібно зробити пояснення щодо імен функцій бібліотеки DLL. Компілятори мови C++ (а також деякі компілятори інших мов, наприклад, MASM) перетворюють імена функцій так, щоб відобразити використаний спосіб передавання параметрів. Так, до імен всіх функцій, які використовують конвенцію C, дописують спереду знак підкреслення. В кінці імені додають знак @ і розмір ділянки стеку в байтах,

яку займають параметри. Отже, ім’я "ReplLt" буде перетворене до "_ReplLt@12". Аналогічно будуть перетворені імена інших функцій.

Щоб точно знати, як виглядають перетворені імена, можна скористатись окремими засобами перегляду файлів. Переглядати можна файл *.dll або файл *.lib, секції Imports/Exports або Library Header відповідно.

Крок 5. В процесі виконання програми викликати функції як звичайно:

```
char one[] = "abcdabcdaarrrad";

int k = ReplLt(one, 'a', '+');
```

Звернемо увагу, що тут ReplLt() є не іменем функції, а оголошеним вказівником типу TReplLt на функцію. Проте, мова C++ не робить різниці між іменами і вказівниками на функції.

Крок 6. В кінці виконання програми звільнити бібліотеку:

```
FreeLibrary(hDLL);
```

Повний приклад програми з явним зв’язуванням

```
#include <iostream>

#include <Windows.h>

using namespace std;

//  місце розташування та ім'я бібліотеки DLL char AllocDLL[] =
"d:\\MyDLLfun\\";

char LibName[] = "DLLCreate.dll";

//  оголошення типів вказівників на зовнішні функції DLL typedef int
(CALLBACK* TReplLt)(char*, char, char);

typedef double (CALLBACK* TMinDset)(int, double[], double[], double,

                                double, int*);

      //      #define  CALLBACK   __stdcall

     //  тестування

int main()

{

    //  завантажити і перевірити наявність бібліотеки HINSTANCE hDLL;
    // дескриптор бібліотеки

    int buffersize = strlen(AllocDLL) + strlen(LibName) + 2;

      //  розмір пам'яті для повного імені DLL
```

```
    char * FullName = new char[buffersize]; // місце для імені DLL strcpy_s(FullName, buffersize,
    AllocDLL); // копіювати шлях до файла strcat_s(FullName, buffersize, LibName);

      //  додати ім'я файла = повне ім'я

    hDLL = LoadLibrary(FullName);   // завантаження бібліотеки

      //  перевірка присутності

    if (hDLL == NULL) {          // якщо немає

        cout << "DLL not found.\n";

        system("pause");  return 0;          //          то припинити програму

    }

//  побудувати вказівники на функції

//     і перевірити наявність функцій в бібліотеці

//  ім'я змінено

    if (!ReplLt) {

        cout << "Function ReplLt not found.\n"; FreeLibrary(hDLL);
        system("pause"); return 0; // якщо немає - то вихід

    }

    TMinDset MinDset = (TMinDset)GetProcAddress(hDLL, "_MinDset@32");

      //  ім'я змінено

    if (!MinDset) {

        cout << "Function MinDset not found.\n"; FreeLibrary(hDLL);
        system("pause"); return 0; // якщо немає - то вихід

    }

//  тепер можна викликати функції як звичайно

//  тестування ReplLt()

cout << "test ReplLt():\n";

char one[] = "abcdabcdaarrrad";

cout << one << endl;

int k = ReplLt(one, 'a', '+');

cout << one << endl << k << endl;

//  тестування MinDset()
```

```
cout << "\ntest MinDset():\n";

double xi[6] = { -0.15, 7.6, 2.4, 2.5, -4.4, 2.3 }; double yi[6] = { -0.58, -0.01, 3.4, 2.9,
1.4, -6.0 }; double x0 = 1.0; double y0 = 1.3;
int N; // номер найближчої точки з масивів xi[],yi[] double minDist;

minDist = MinDset(6, xi, yi, x0, y0, &N);

cout << "Min= " << minDist << '\t' << "NumP= " << N << endl;

//  в кінці звільнити бібліотеку

    FreeLibrary(hDLL); delete[] FullName;
    system("pause"); return 0;

}
```

Приклад отриманих результатів:

```
test ReplLt():
abcdabcdaarrrad
+bcd+bcd++rrr+d
5

test MinDset():
Min= 2.19317    NumP= 3
```

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

Передавання параметрів за значенням. Функції передають власне значення параметра. При цьому фактично копіюють значення параметра, а функція використовує його копію.

Передавання параметрів за посиланням (за адресою). Функції передають не значення змінної, а її адресу, за якою функція сама прочитає значення пераметра

Передавання параметрів за поверненим значенням. Цей механізм об'єднує передавання за значенням і за посиланням. Функції передають адресу змінної, а функція будує локальну копію параметра, після цього працює з нею, а в кінці записує локальну копію назад за переданою адресою.

Передавання параметрів за результатом. Цей механізм відрізняється від попереднього лише тим, що при виклику функції значення параметра не отримують, функція самостійно будує результат, а передану адресу використовують лише для записування за нею результату.

Передавання параметрів за іменем. Цей механізм використовують у макровизначеннях, в директиві EQU і в препроцесорі C в часі опрацювання команди #define. При реалізації цього механізму в компільованій мові програмування доводиться виконати заміну параметра за іменем іншими механізмами, зокрема, за допомогою макровизначень.

Передавання параметрів відкладеним обчисленням. Як і в попередньому випадку, викликана функція отримує адресу іншої функції, що обчислює значення параметра.

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

Передавання параметрів в регістрах. Якщо функція має невелику кількість параметрів та їх розміри не більші чотирьох байтів, то найкращим місцем є регістри загального призначення. Прикладами можуть бути майже всі виклики переривань DOS і BIOS. Мови високого рівня використовують регістр EAX для того, щоб повернути результат роботи функції, а також регістр ST(0) співпроцесора для повернення результату дійсного числа.

Передавання параметрів в глобальних змінних. Якщо не вистарчає регістрів, один з способів обійти таке обмеження – записати параметр у величину, до якої пізніше буде звертатись функція, подібним механізмом, як було показано при передаванні параметрів за поверненим значенням. Проте такий метод не можна використати для рекурсивних функцій чи функцій з повторним викликом.

Передавання параметрів в стеку. Це є основний спосіб зв'язку між функціями, який придатний для функцій різної будови і різних операційних систем. Параметри записують в стек безпосередньо перед викликом функції. Такий спосіб використовують мови високого рівня компільованого типу. Для читання параметрів з стеку використовують не команду POP, а регістр EBP, основне призначення якого в архітектурі Intel якраз полягає в забезпеченні роботи з стеком як зі звичайною ділянкою пам'яті.

Погодження	Параметри	Очищення стеку	Регістри
pascal (конвенція мови Паскаль)	зліва направо	функція	немає
register (швидкий, або регістровий виклик)	зліва направо	функція	задіяні три регістри (EAX, EDX, ECX), після цього стек
cdecl (конвенція C)	справа наліво	викликаюча функція	немає
stdcall (стандартний виклик)	справа наліво	функція	немає

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

Розглянемо детальніше на прикладі взаємодію викликаючої і викликаної функції за конвенцією stdcall, бо саме така прийнята за основу в більшості реальних проектів.

Нехай маємо деяку функцію з трьома параметрами, двома локальними змінними і одним поверненим результатом:

```
int _ _stdcall Fun(char * STRN, char LA, int COUNT)
{
    int cnt=0; int test;
    // . . . . . .
    return cnt;
}
```

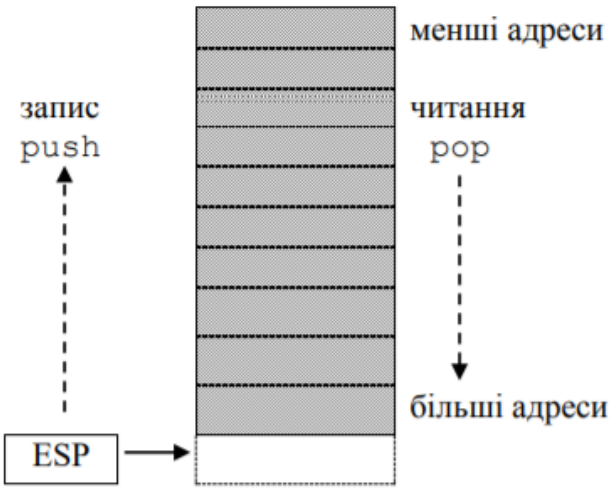
Тоді схема виклику і виконання функції буде такою:

```
[ . . . . . ] ; попередні команди викликаючої функції
```

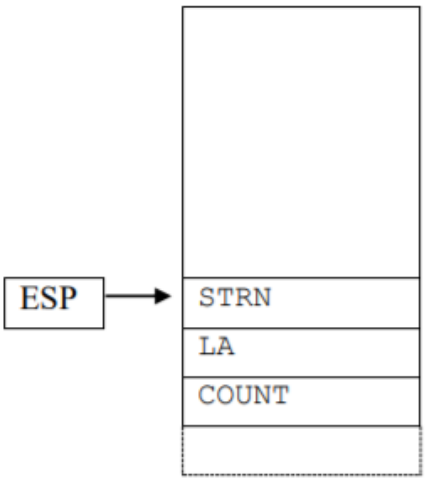
```
push parameterCOUNT
push parameterLA
push parameterSTRN ; параметри в стек в оберненому порядку
call Fun ; викликаємо функцію Fun
; сюди буде повернення після виконання Fun
 [ . . . . . ] ; наступні команди викликаючої функції
Fun PROC EXPORT STRN:dword,LA:dword,COUNT:dword
; це розширений заголовок для узгодження імені функції з мовами програмування
; в тому числі – розміри параметрів
push EBP ; попереднє значення EBP
mov EBP,ESP ; базова адреса для читання параметрів «кадр стеку»
; будуємо позначення адресів параметрів
STRN equ dword ptr [EBP+8] ; адреса першого параметра
LA equ byte ptr [EBP+12] ; другого
COUNT equ dword ptr [EBP+16] ; третього
; резервуємо в стеку 8 байтів для локальних змінних
sub ESP,8
; і будуємо позначення адресів для них
cnt equ dword ptr [EBP-4]
test equ dword ptr [EBP-8]
[ . . . . . ] ; команди функції Fun
; вихід з функції
mov EAX,cnt ; результат стандартно повертають через EAX
mov ESP,EBP ; відновити ESP, викресливши з стеку локальні змінні
pop EBP ; відновити EBP функції, яка викликала
ret 12 ; повернення з викресленням параметрів з стеку
; ( mov ESP,EBP ) + ( pop EBP ) = LEAVE
Fun ENDP ; кінець визначення функції
```

Зауважимо, що тут подана саме схема компіляції з мов високого рівня, у випадку безпосереднього програмування на макроасемблері команди входу і виходу з функції макроасемблер додає сам автоматично.

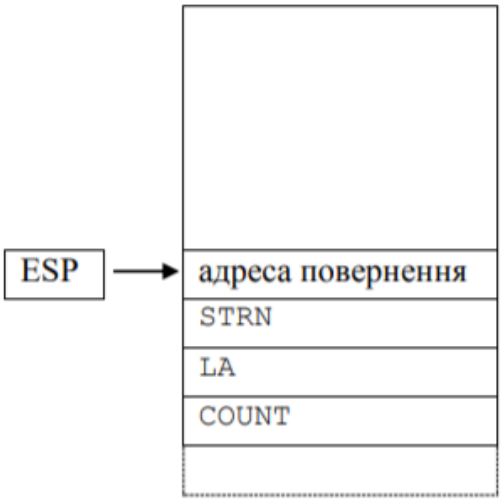
Тепер розглянемо на рисунках кроки виклику і виконання функції.



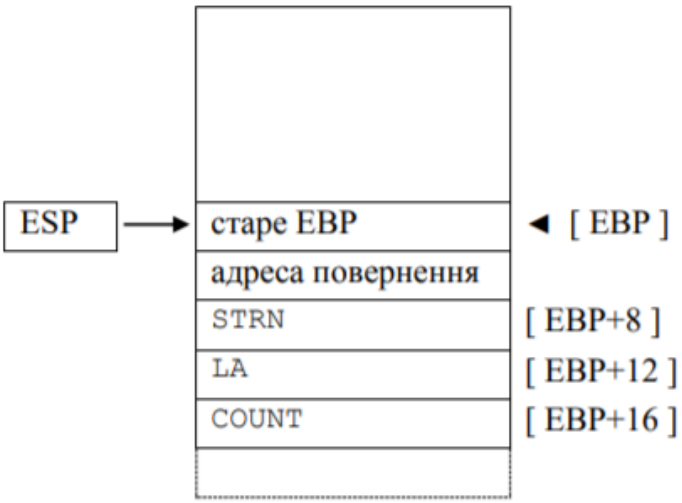
1) початковий стан стеку



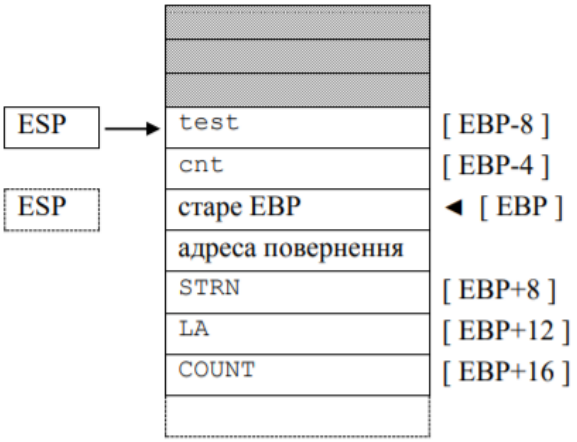
2) в стек записали параметри `push`



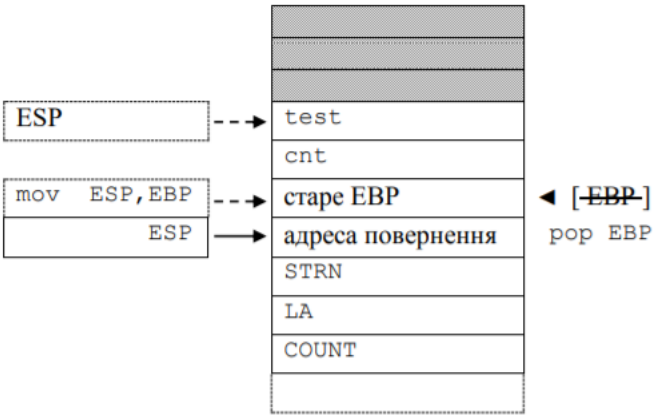
3) викликали функцію `call`



4) налаштували «кадр стеку»



5) місце локальних змінних `sub ESP,8`



6) вихід з функції – відновили ESP і EBP

7) повернення з викресленням параметрів з стеку: ret 12 - стан 1.

30. Стандартні типи даних Windows та їх еквіваленти в мові С.

Windows	означення	C
BOOL	логічний (4-байтовий) має два значення - 0 або 1. При використанні WINAPI прийнято вживати замість 0 специфікатор NULL	bool (1-байтовий)
BYTE	байт, або восьмибітне беззнакове ціле число	unsigned char
DWORD, UINT	32-бітове беззнакове ціле	unsigned long int
INT, LONG	32-бітове ціле	long int
DOUBLE	8-байтове дійсне число	double
NULL	нульовий вказівник (4-байтовий) або ціле число 0;	void * NULL=0; int NULL=0;
HANDLE	дескриптор – ідентифікатор якого-небудь об'єкта (4-байтовий); для різних типів об'єктів існують різні дескриптори	
HCURSOR	дескриптор курсора (4-байтовий)	
HDC	дескриптор контексту пристрою	
HINSTANCE	дескриптор екземпляра додатка (програми)	
HWND	дескриптор вікна (4-байтовий)	
LPINT	вказівник на ціле (4-байтовий)	int *
LPSTR	вказівник на будь-який рядок, що закінчується нуль-кодом (4-байтовий)	char * (4-байтовий)
LPTSTR	вказівник на рядок без юнікоду; це надбудова функції LPSTR	char *
LPWSTR	вказівник на UNICODE рядок; це надбудова функції LPSTR	wchar_t * (4-байтовий)
CHAR, TCHAR	символьний тип (1-байтовий)	char

WCHAR	символьний тип (2-байтовий)	wchar_t (2-байтовий)
LPARAM	довгий параметр (4-байтовий); використовують разом з WPARAM в деяких функціях	
WPARAM	параметр-слово (4 байти); використовують разом з LPARAM в деяких функціях	
LRESULT	значення, яке повертає віконна процедура (4-байтове)	long

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

Першим кроком треба виконати проектування, програмування і тестування функцій, які будуть поміщені в DLL. Для цього можна створити проект типу “Empty Project”, який не зв’язаний з конкретними типами проектів.
Приклад. Функція виконує заміну в рядку літер STRN кожної букви LA на LB. Крім того, обчислює кількість виконаних замін.

Крок 1. Тестування з використанням типів C++.

Виконуємо розробку і тестування в межах одного середовища програмування C++. В цьому разі достатньо використати стандартні типи даних мови C++, бо виклики функцій відбуваються так само в межах мови C++, і проблеми узгодження типів параметрів не виникає.

```
/ прототип функції
 int (__stdcall ReplLt) (char * STRN, char LA, char LB);

// повні визначення функції
 int(__stdcall ReplLt)(char * STRN, char LA, char LB) {
int cnt = 0; int i = 0;
while (STRN[i]) {
 if (STRN[i] == LA) {
STRN[i] = LB; cnt++;
} i++; }
 return cnt; }
```

Крок 2. Тестування з використанням типів Windows.
Як було зазначено раніше, функції, поміщені в DLL, в загальному випадку розраховані на використання в різномовних середовищах програмування. Тому варто організувати повторне тестування тих самих функцій, використавши для параметрів стандартні типи Windows. При цьому треба одночасно замінити типи деяких локальних змінних функцій. Прототипи функцій за кроком 2 зазвичай подають як інструкцію до використання функцій нашої бібліотеки. Тоді користувач має узгодити типи даних іншої А-мови програмування за схемою:

типи Windows ↔ типи А–мови

Кроку 2 можна не виконувати, тоді інструкцією до бібліотеки будуть прототипи функцій мовою C++, і узгодження відбувається за схемою:

типи C++ ↔ типи A–мови

```
// прототип функції
INT (__stdcall ReplLt) (LPTSTR STRN, CHAR LA, CHAR LB);
// повні визначення функції за типами Windows
INT (__stdcall ReplLt)(LPTSTR STRN, CHAR LA, CHAR LB) {
INT cnt = 0; int i = 0;
while (STRN[i]) {
if (STRN[i] == LA) {
STRN[i] = LB; cnt++;
} i++;
}
return cnt; }
```

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

Розглянемо процес компіляції функцій у динамічну бібліотеку за допомогою середовища Visual Studio.

1) Створюємо новий проект типу динамічної бібліотеки:

New Project → Windows Desktop → Dynamic Link Library (DLL) → визначити Name і Location → Create directory for solution → OK;

2) Вносимо код функцій та директиви експорту функцій:

Нехай ім'я Name файла проекту MyDLL.cpp. У MyDLL.cpp друкуємо тексти функцій та додатково директиви експорту функцій. При цьому вирішуємо, чи додати в бібліотеку функцію входу DllMain. Ця функція має шаблон визначення в зв'язаному файлі dllmain.cpp, який редагуємо за потреби.
Приклад директив експорту функцій:

```
extern "C" __declspec(dllexport) int(__stdcall A) (char * param1, char param2, char param3);
extern "C" __declspec(dllexport) double(__stdcall B) (int param1, double param2[], double param3[], double param4, double param5, int * param6);
```

3) **Компілюємо DLL : Build → Build Solution.**

33. Використання DLL в прикладних програмах методом явного зв'язування (на прикладі мови C чи іншої).

Для Використання DLL в прикладних програмах методом явного зв'язування існує три основних методи:

1. LoadLibrary - Метод завантажує DLL за іменем і повертає дескриптор
2. GetProcAddress - Метод знаходить об'єкт / функцію / змінну за іменем у DLL
3. FreeLibrary - Метод вивантажує DLL, коли ми закінчимо з нею

Нижче наведено приклади коду для створення DLL та її використання. В усіх прикладах використовується код, написаний мовою програмування C++.

Код самих функцій не відрізняється від такого, який може застосовуватися не тільки для створення DLL, а й для звичайного виконання

```
int passwordToHashCode(const char* password) {
   int hash = 0;

   for (; *password; ++password)
   {
      hash += *password;
      hash += (hash << 10);
      hash ^= (hash >> 6);
   }

   hash += (hash << 3);
   hash ^= (hash >> 11);
   hash += (hash << 15);

   return hash;
}

bool comparePassword(const char* potentialPassword, int actualPasswordHash) {
   int potentialPasswordHash = passwordToHashCode(potentialPassword);

   return potentialPasswordHash == actualPasswordHash;
}

char* generateRandomPassword(const int passwordLength) {
   const char alphanum[] =
"0123456789!@#$%^&*abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
   int string_length = sizeof(alphanum) - 1;

   char* password = new char[passwordLength + 1];


   for (int i = 0; i < passwordLength; i++) {
      password[i] = alphanum[rand() % string_length];
   }

   password[passwordLength] = '\0';

   return password;
}

void printPassword(char* password) {
   for (; *password; ++password) {
      cout << *password;
   }

   cout << endl;
}
```

Для того щоб вказані вище функції можна було застосувати як підключені функції за доопмогою явного зв’язування необхідно створити файл-заголовків із описом функцій:

```
//Налаштовує імпорт та експорт складників DLL
#ifdef PASSWORD_EXPORTS
#define PASSWORD_API __declspec(dllexport)
#else
#define PASSWORD_API __declspec(dllimport)
#endif


//Нижче описано функції, які будуть доступні для використання при імпорту DLL

// Returns hash code for password
extern "C" PASSWORD_API int passwordToHashCode(const char* password);

// Compares raw password with hash code.
extern "C" PASSWORD_API bool comparePassword(const char* potentialPassword, int actualPasswordHash);

// Generates random password.
extern "C" PASSWORD_API char * generateRandomPassword(const int passwordLength);

// Prints password.
extern "C" PASSWORD_API void printPassword();
```

Після коректного оголошення елементів DLL та його збірки, можна застосувати отримані ресурси для явного зв’язування та використання функцій в інших прикладних програмах.
Для викорситання цих функції необхідно спершу оголосити їх сигнатури. Це робиться за допомогою наступного коду:

```
typedef int (_cdecl* tPasswordToHashCode)(const char* password);
typedef bool (_cdecl* tComparePassword)(const char* potentialPassword, int actualPasswordHash);
typedef char* (_cdecl* tGenerateRandomPassword)(const int passwordLength);
typedef void(_cdecl* tPrintPassword)();

tPasswordToHashCode passwordToHashCode;
tComparePassword comparePassword;
tGenerateRandomPassword generateRandomPassword;
tPrintPassword printPassword;
```

Після оголошення сигнатур функцій, можна створити функцію-помічник яка явно завантажить функції з DLL та присвоїть посилання на них вказаним раніше змінним. Така функція може виглядати так:

```
int loadDllExplicitly() {
   cout << "**** Starting loading dynamic library ****" << endl;

   //Змінна для доступу до DLL
   HINSTANCE hDLL;
   //Двійковий буфер для зчитування DLL
```

```
    int buffersize = strlen(AllocDLL) + strlen(LibName) + 2;
    char* fullName = new char[buffersize];
    strcpy_s(fullName, buffersize, AllocDLL);
    strcat_s(fullName, buffersize, LibName);

    hDLL = LoadLibraryA(fullName);

    // Перевірка чи було знайдено DLL за адресою, вказаною у змінній fullName
    if (hDLL == NULL) {
        cout << "DLL not found.\n"; system("pause");  return 0;
    }

    // Намагаємося отримати функцію passwordToHashCode з DLL
    passwordToHashCode = (tPasswordToHashCode)GetProcAddress(hDLL, "passwordToHashCode");

    // Якщо не вадлося отримати функцію passwordToHashCode - виводимо повідомлення про помилку,
вивантажуємо DLL, завершуємо виконання функції
    if (!passwordToHashCode) {
        cout << "Function passwordToHashCode not found.\n"; FreeLibrary(hDLL); system("pause");
        return 0;
    }

    cout << "Function passwordToHashCode was successfully initiliazed"  << endl;

    // Намагаємося отримати функцію comparePassword з DLL
    comparePassword = (tComparePassword)GetProcAddress(hDLL, "comparePassword");

    // Якщо не вадлося отримати функцію comparePassword - виводимо повідомлення про помилку,
вивантажуємо DLL, завершуємо виконання функції
    if (!comparePassword) {
        cout << "Function comparePassword not found.\n"; FreeLibrary(hDLL); system("pause");
        return 0;
    }

    cout << "Function comparePassword was successfully initiliazed" << endl;


    // Намагаємося отримати функцію generateRandomPassword з DLL
    generateRandomPassword = (tGenerateRandomPassword)GetProcAddress(hDLL,
"generateRandomPassword");

    // Якщо не вадлося отримати функцію generateRandomPassword - виводимо повідомлення про помилку,
вивантажуємо DLL, завершуємо виконання функції
    if (!generateRandomPassword) {
        cout << "Function generateRandomPassword not found.\n"; FreeLibrary(hDLL); system("pause");
        return 0;
    }

    cout << "Function generateRandomPassword was successfully initiliazed" << endl;
```

```
    // Намагаємося отримати функцію printPassword з DLL
    printPassword = (tPrintPassword)GetProcAddress(hDLL, "printPassword");

    // Якщо не вадлося отримати функцію printPassword - виводимо повідомлення про помилку,
вивантажуємо DLL, завершуємо виконання функції
    if (!comparePassword) {
        cout << "Function printPassword not found.\n"; FreeLibrary(hDLL); system("pause");
        return 0;
    }

    //Вивантажуємо DLL після завершення роботи з нею
    FreeLibrary(hDLL);

    /*Якщо функція не завершила виконання раніше, значить вдалося завантажити усі цільові функції з
DLL.
    У такому випадку виводимо повідомлення про успіх завантаження та виходимо з функції з кодом
успіху*/
    cout << "Function printPassword was successfully initiliazed" << endl;

    cout << "**** All functions have been successfully initialized ****" << endl << endl;

    return 1;
}
```

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

Неявне зв'язування - операційна система завантажує бібліотеку DLL в той момент, коли вона використовується виконуваним файлом. Виконуваний файл клієнта викликає експортовані функції бібліотеки DLL так само, як статично скомпоновані і включені до складу самого виконуваного файлу функції. Процес неявного зв'язування також іноді називають статичної завантаженням або динамічної компонуванням часу завантаження.

Мова C# забезпечує неявне зв’язування з бібліотеками DLL.Для цього потрібно оголосити кожну функцію окремо як зовнішню за допомогою тегів DllImport:

```
[DllImport("CryptoLib.dll")]
public static extern void ShiftEncode(String text, Int32 shift,StringBuilder result);
[DllImport("CryptoLib.dll")]
public static extern void ShiftDecode(String text, Int32 shift,StringBuilder result);
```

а далі використовувати як звичайно. Звернути увагу на співставленні типів параметрів функцій DLL, написаних на C++, і типів параметрів C# при оголошенні функцій. Самі бібліотеки необхідно або копіювати в папку проекту, або визначити за інструкцією місце розташування бібліотеки DLL. Отже, використання бібліотек мовою C# зводиться до оголошення функцій як зовнішніх.

Список питань до розділу "Графічні редактори"

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

Піксель – елементарна одиниця, зафарбовані у будь - який колір. Зауважимо, що пікселі певних типів дисплеїв можуть не бути точними квадратами, однак переважно все ж таки є квадратними.
Роздільна здатність – кількістю пікселів на одиницю довжини (переважно на дюйм: DPI – dots per inch). Чим більша роздільна здатність, тим кращу якість картинки можна отримати. Переважно користуються стандартними розмірами (роздільними здатностями) екрана, наприклад: 640 x 480, 800 x 600, 1024 x 768, 1152 x 864, 1280 x 1024 тощо.
Сучасні комп'ютери зазвичай використовують **палітру кольорів**, яка може мати 256 x 256 x 256 різних значень, однак використовують також палітру з 256 - ти кольорів і навіть з 16 - ти кольорів.
Малювання на екрані полягає у зафарбовуванні певної групи пікселів. Оскільки вони квадратної форми, то зображення завжди є апроксимацією (наближенням) реальної картинки

36. Графічні примітиви. Графічні бібліотеки. Принципи використання.

Графічні бібліотеки

Графічний інтерфейс в різних системах програмування реалізують за допомогою графічних бібліотек. Графічні бібліотеки, як правило, є частиною системи програмування.

System.Drawing (ООП-"обгортка" для GDI и GDI+) - стандартна бібліотека для програмування графіки мовою C#. Надає набір класів та засобів для створення графічних програм.

Qt - графічна бібліотека, доступна для мови C++, Python та інших. Надає набір інструментів для швидкого і зручного проектування GUI. Є кросплатформенною, доступна як на Windows, Linux і Mac OS, так і на мобільних платформах - Windows Mobile, Android та iOS.
Tkinter - графічна бібліотека для мови Python. Більшість програм, які використовують бібліотеку tkinter, будують загальний сценарій за такою схемою:
1.Завантажити клас віджета (віконного елемента) з модуля tkinter.
2.Створити екземпляр імпортованого класу.
3.Розташувати (запакувати) новий об'єкт на батьківському елементі.
4.Викликати функцію mainloop, щоб показати вікно і розпочати цикл подій tkinter.

Растрова графіка

Двовимірну графічну поверхню, на якій здійснюється відображення графічної інформації, можна представити у вигляді прямокутного растра кольорових квадратних пікселів. Завданням малювання, таким чином, є надання потрібним пікселям заданого кольору. Нумерацію пікселів на площині зручно здійснювати в декартовій системі координат x, y. За замовчуванням початок відліку (піксель з координатами x = 0, y = 0) розміщується в лівому верхньому куті графічної поверхні, і вертикальна вісь y спрямована вниз.

Крім вказування координат пікселів для малювання найпростіших фігур потрібно визначити перо (Pen) для задання кольору і деяких інших властивостей контуру фігури або пензель (Brush) - для задання кольору пікселів, що заповнюють фігуру. Наприклад, наступний код створює в «керованій купі» і повертає вказівники на дескриптор пера чорного кольору товщиною три пікселі і «твердої» кисті (SolidBrush) червоного кольору:

```
Pen blackPen = new Pen(Color.Black, 3);
SolidBrush redBrush = new SolidBrush(Color.Red);
```

Малювання виконують за допомогою графічних примітивів - операцій, таких, як Ellipse(x1,y1,x2,y2), LineTo(x,y), Rectangle(x1,y1,x2,y2), FloodFill(x,y,Color,FillStyle), TextOut(x,y,Text), Draw(x,y,Graphic) тощо. Сукупність таких операцій реалізують як окремі модулі чи бібліотеки, що належать до складу системи програмування. Якщо примітиви відсутні, то доцільно спершу створити їхню бібліотеку, а потім розробляти графічний редактор.

Відрізок прямої

Відрізок прямої на графічній поверхні можна намалювати, викликавши метод DrawLine для графічного об'єкта Graphics. Буде потрібно перо, а також піксельні координати початку відрізка (x1, y1) і кінця (x2, y2):

```
e.Graphics.DrawLine(blackPen, x1, y1, x2, y2);
```

Прямокутник

Метод DrawRectangle малює контур прямокутника, а метод FillRectangle зафарбовує прямокутну область. Для першого методу необхідно вказати перо, а для другого - кисть. Сама прямокутна область задається координатами її лівого верхнього кута, шириною (уздовж осі x) і висотою (уздовж осі y), вимірюваними в пікселях:

```
e.Graphics.DrawRectangle(blackPen, x, y, width, height);
e.Graphics.FillRectangle(redBrush, x, y, width, height);
```

Одиничний піксель

Зафарбувати один піксель з координатами (x, y) найпростіше за допомогою методу FillRectangle, вказавши одиничну ширину и висоту:

```
e.Graphics.FillRectangle(redBrush, x, y, 1, 1);
```

Еліпс

Малюється і зафарбовується методами DrawEllipse і FillEllipse, відповідно. Координатами задається прямокутна область, в яку еліпс буде вписано:

```
e.Graphics.DrawEllipse(blackPen, x, y, width, height);
e.Graphics.FillEllipse(redBrush, x, y, width, height);
```

37. Графічний інтерфейс редактора з користувачем. Клавіатурний інтерфейс редактора з користувачем.

Інтерфейс з користувачем

1. Графічний інтерфейс.

Частина екрана пристосована під панель інструментів – зображення графічних операцій (примітивів), якими можна користуватися. Команди редактора (інструменти) можна вибирати як мишкою, так і комбінацією клавіш зі стрілками і, можливо, інших.

Порядок виконання однієї операції зазвичай є таким: вибір (мишею) виду операції → фіксація місця початку фігури (верхнього лівого кутка) → фіксація місця закінчення фігури (правого нижнього кутка). При фіксуванні не обов'язково вибирати кутки як зазначено – достатньо обрати два протилежні кутки у довільному порядку.

При малюванні декількох однотипних фігур повторний вибір зазвичай непотрібний.

До моменту закінчення малювання фігури бажано мати змогу бачити прообраз майбутньої фігури.

Остаточно фігура фіксується або при відпусканні клавіші миші, або при повторному її натисненні – залежно від реалізації.

2. Клавіатурний інтерфейс.

Графічні операції обирають натисканням певних клавіш, наприклад: E → Ellipse, C → Circle, R → Rectangle і т.д. У цьому випадку відображення панелі інструментів на екрані не є обов'язковим, але необхідно мати в інформаційній частині повідомлення про вибрану команду.

Фіксування місця та розмірів фігури можна виконати клавішами зі стрілками, або мишкою – так само, як при графічному інтерфейсі.

38. Прокручування поля, багатовіконність редактора, коректне припинення роботи.

Прокручування поля малювання. Малюнок може бути довільних розмірів, зокрема, більшим, ніж поле малювання. У цьому випадку необхідне горизонтальне та вертикальне прокручування поля малювання за допомогою смуг перегляду. В кожен момент на полі малювання, як у вікні, відображається частина малюнка як на рис.9.

Рис.9. Прокручування поля малювання

Весь малюнок зберігається як єдине ціле в пам'яті. Варто подумати про максимально припустимі розміри малюнка.

Багатовіконність. Це є можливість одночасної роботи з декількома малюнками – кожен у своєму вікні. Переключення між вікнами реалізується через меню або комбінацією клавіш, наприклад, Ctrl+F6. Для ефективної роботи необхідно мати команди (функції) для копіювання фрагментів і цілих малюнків у

внутрішній буфер графічного редактора та вставляння з буфера. Графічні редактори можуть використовувати одночасно свій власний

внутрішній буфер та буфер Windows, при цьому копіювання у різні буфери необхідно здійснювати різними командами, і так само вставляння.

Закінчення роботи. Перевіряємо збереження малюнків у файлі, виконуємо очищення буферів пам'яті, звільняємо надану під час роботи пам'ять, закриваємо файли, викреслюємо тимчасові файли і т.п.

39. 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

1. Розбиття множини усіх припустимих для редактора команд на окремі логічні групи. Перемикання між групами.
2. Збільшення малюнка (розтягання), зменшення (стиснення), а також повороти на кут $\pm n(\pi/2)$, $n=1,2,3$.
3. Маштабування малюнка з метою точного виконання графічних операцій.
4. Реалізація контекстного (наприклад, за клавішею F1) і випливаючого меню (наприклад, за правою клавішею мишки). Вибір та виконання команд через меню.
5. Підказки про події на екрані.
6. Згладжування ліній та інші подібні методи геометричного опрацювання (рис.10).

Рис.10. Приклад згладжування ліній

7. Згладжування межі зміни кольору (плавний перехід одного кольору в інший) та інші способи фільтрування малюнка (рис.11).

Рис.11. Приклад згладжування межі кольорів

8. Формування у файлах набору демонстраційних малюнків.
9. Наявність навчальної підсистеми – опис редактора, команд, особливостей використання

40. Графічний програмний інструментарій: перо, пензель, шрифт. Загальні характеристики.

За пензель відповідає клас Brush, Об’єкти класу Brush (пензлі) використовують для заповнення внутрішнього простору

замкнених фігур. Властивості класу можуть бути такими:

property Color	Колір пензля
property Style	Стиль пензля – спосіб зафарбування внутрішньої ділянки фігури (див. нижче)
property Bitmap	Визначає зовнішнє растрове зображення, котре буде використане пензлем для заповнення. Якщо ця властивість визначена, властивості Color і Style ігноруються
property Handle	Дескриптор пензля. Використовується при безпосередньому звертанні до API-функцій Windows

Тип стилю пензля визначається так:
enum TBrushStyle = { bsSolid, bsClear, bsHorizontal, bsVertical,
bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross };
Перо:
З допомогою класу Pen створюється об’єкт-перо, який використовують для малювання ліній. Подаємо огляд потенційних параметрів класу. В різних системах програмування можуть бути реалізовані такі самі параметри або подібні за змістом. Остаточна реалізація залежить від можливостей бібліотеки графічних функцій

property Color	Колір ліній, які креслить перо
property Width	Товщина ліній у пікселах екрана
property Style	Визначає стиль ліній (див. нижче). Враховується тільки для ліній товщиною в 1 піксел. Для товстих ліній завжди psSolid (суцільна)
property Mode	Визначає спосіб взаємодії ліній з тлом (див. нижче)
property Handle	Дескриптор пера. Використовується при беспосередньому звертанні до функцій API

Тип стилю лінії Style визначається так: *enum TPenStyle = { psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame };*
Тип взаємодії лінії з тлом визначається так: enum *TPenMode = { pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor };*
Шрифт:
За шрифт відповідає клас Font, він має наступні характеристики:

Bold	Містить значення, яке вказує, чи жирний цей шрифт.
FontFamily	Містить FontFamily, пов’язану з цим шрифтом.
GdiCharSet	Містить байтове значення, яке вказує набір символів GDI, який використовує цей шрифт.
GdiVerticalFont	Містить булеве значення, яке вказує, чи цей шрифт походить від вертикального шрифту GDI.
Height	Містить міжрядковий інтервал цього шрифту.
IsSystemFont	Містить значення, що вказує, чи є шрифт членом SystemFonts.
Italic	Містить значення, яке вказує, чи застосовується цей шрифт курсивом.
Name	Містить ім'я цього шрифта.
OriginalFontName	Містить назву шрифту, вказаного спочатку.
Size	Містить em-розмір цього шрифта, виміряний в одиницях, визначених властивістю Unit.
SizeInPoints	Містить розмір em у шрифтах у точках.
Strikeout	Містить значення, чи є шрифт закресленим
Style	Містить інформацію про стиль для цього шрифта.
SystemFontName	Містить ім'я системного шрифту, якщо властивість IsSystemFont повертає true.
Underline	Містить значення, яке вказує, чи підкреслено цей шрифт.
Unit	Містить одиницю виміру для цього шрифту.

41. Операції читання/запису для графічних зображень. Формати збереження у файлах.

JPEG / JFIF

JPEG (Joint Photographic Experts Group) - метод стиснення втрат. Стислі зображення JPEG зазвичай зберігаються у форматі файлів JFIF (JPEG File

Interchange Format). Розширення назви файлів JPEG / JFIF - JPG або JPEG. Майже кожен цифровий фотоапарат може зберігати зображення у форматі JPEG / JFIF, який підтримує восьмибітні зображення сірого кольору та 24-бітні кольорові зображення (вісім біт для червоного, зеленого та синього). JPEG застосовує стиснення втрат до зображень, що може призвести до значного зменшення розміру файлу. Програми можуть визначати ступінь стиснення, що застосовується, а кількість стиснення впливає на візуальну якість результату. Якщо не надто велике, стиснення непомітно впливає на якість зображення чи не погіршує його, але JPEG- файли зазнають деградації поколінь при повторному редагуванні та збереженні. (JPEG також забезпечує зберігання зображень без втрат, але версія без втрат не підтримується широко.)

PNG

Формат файлу PNG (Portable Network Graphics) був створений як безкоштовна

альтернатива GIF з відкритим кодом. Формат файлу PNG підтримує восьмибітні палітри зображень (з додатковою прозорістю для всіх кольорів палітри) та 24-бітну truecolor (16 мільйонів кольорів) або 48-бітну truecolor з альфа-каналом і без нього - в той час як GIF підтримує лише 256 кольорів і один прозорий колір.

Порівняно з JPEG, PNG є кращим, якщо зображення має великі, рівномірні кольори. Навіть для фотографій - де JPEG часто є вибором для остаточного розповсюдження, оскільки техніка стиснення зазвичай дає менші розміри файлів - PNG все ще добре підходить для зберігання зображень під час процесу редагування через стиснення без втрат.

PNG забезпечує безпатентну заміну GIF (хоча GIF сам по собі зараз не є патентом), а також може замінити багато поширених видів використання TIFF. Підтримуються зображення з індексованим кольором, відтінками сірого та справжнім кольором, плюс додатковий альфа-канал. Adam7 переплетення дозволяє ранній попередній перегляд, навіть коли був переданий лише невеликий відсоток даних зображення. PNG може зберігати дані гамми та кольоровості для поліпшення відповідності кольорів на неоднорідних платформах.

PNG розроблений для того, щоб добре працювати в онлайнових програмах перегляду, таких як веб-браузери, і може бути повністю потоково переданий за допомогою прогресивної опції відображення. PNG надійний, забезпечує як повну перевірку цілісності файлів, так і просте виявлення поширених помилок передачі.

GIF

GIF (формат графічного обміну) в звичайному використанні обмежується 8-бітною палітрою, або 256 кольорів (в той час як 24-бітна глибина кольору технічно можлива). GIF найбільше придатний для зберігання графіки з кількома кольорами, наприклад простими діаграмами, формами, логотипами та зображеннями в стилі мультфільму, оскільки використовує стиснення без втрат LZW, що є більш ефективним, коли великі площі мають один колір, і менш ефективні для фотографічних або пофарбованих зображень. Завдяки простоті та віку GIF досяг майже універсальної програмної підтримки. Завдяки анімаційним можливостям він все ще широко використовується для надання ефектів анімації зображень, незважаючи на низький коефіцієнт стиснення порівняно з сучасними відеоформатами.

BMP

Формат файлу BMP (растрова карта Windows) обробляє графічні файли в операційній системі Microsoft Windows. Зазвичай файли BMP не стискаються, а значить, великі та без втрат; їх перевагою є їх проста структура та широке сприйняття в програмах Windows.

TIFF

Формат TIFF (Tagged Image File Format) - гнучкий формат, який зазвичай зберігає вісім біт або шістнадцять біт на колір (червоний, зелений, синій) для 24-бітних і 48-бітових зображень відповідно, зазвичай використовуючи розширення імені файлу TIFF, або TIF. Ця структура була розроблена так, щоб вона легко розширювалася, і багато виробників ввели власні власні теги спеціального призначення - в результаті чого ніхто не читає будь-який варіант файлу TIFF. TIFF можуть бути втратними або без втрат, залежно від метода, обраного для зберігання пікселів зображення. Дехто пропонує порівняно гарну компресію без втрат для дворівневих (чорно-білих) зображень. Деякі цифрові камери можуть зберігати зображення у форматі TIFF, використовуючи алгоритм стиснення LZW для зберігання без втрат. Формат зображення TIFF не підтримується широко веб-браузерами. TIFF залишається широко прийнятим як стандарт фотофайлів у поліграфічній справі. TIFF може обробляти кольорові простори, характерні для пристрою, такі як CMYK, визначений певним набором фарб друкарського друку. Програмні пакети OCR (Optical Character Recognition) зазвичай генерують певну форму зображення TIFF (часто однотонне) для сканованих текстових сторінок.

SVG

SVG (масштабована векторна графіка) - це відкритий стандарт, створений та

розроблений Всесвітнім консорціумом веб-сторінок для вирішення потреб (та спроб декількох корпорацій) в універсальному, доступному для написання та універсальному векторному форматі для Інтернету та іншим способом. Формат SVG не має власної схеми стиснення, але через текстовий характер XML графіку SVG можна стиснути за допомогою програми, такої як gzip. Через свій сценарій сценаріїв, SVG є ключовим компонентом у веб-додатках: інтерактивні веб-сторінки, які виглядають і діють як програми.

CGM

CGM (Computer Graphics Metafile) - це формат файлів для двовимірної векторної графіки, растрової графіки та тексту, визначається ISO/IEC 8632. Усі графічні елементи можна вказати у текстовому вихідному файлі, який можна скласти у двійковий файл або в одне з двох текстових подань. CGM забезпечує засіб обміну графічними даними для комп’ютерного представлення 2D графічної інформації, незалежної від будь-якого конкретного додатку, системи, платформи чи пристрою. Вона певною мірою була прийнята в галузі технічної ілюстрації та професійного дизайну, але значною мірою його витіснили формати, такі як SVG та DXF .

ODF

Формат Open Document для додатків Office, (ODF), також відомий як OpenDocument , є ZIP стиснутим XML-форматом файла для електронних таблиць, діаграм, презентацій і обробки текстів документів. Він був розроблений з метою надання відкритої специфікації формату файлів на основі XML для офісних програм. Стандарт був розроблений технічним комітетом в консорціумі Організації просування структурованих інформаційних стандартів (OASIS). Він базувався на специфікації Sun Microsystems для OpenOffice.org XML, форматі за замовчуванням для OpenOffice.org та LibreOffice. Спочатку було розроблений для StarOffice, "щоб забезпечити відкритий стандарт для офісних документів". Крім того ,

що стандарт OASIS, він був опублікований як ISO/IEC міжнародного стандарту ISO/IEC 26300 - Формат Open Document для додатків Office (OpenDocument).

EPS

Інкапсульований PostScript (EPS) - це конвенція про структурування документів - відповідність (DSC) формату документа PostScript, яку можна використовувати у форматі графічного файлу. Файли EPS - це більш-менш автономні, досить передбачувані документи PostScript, що описують зображення або малюнок, і їх можна помістити в інший документ PostScript. Файл EPS - це по суті програма PostScript, яка зберігається як єдиний файл, який включає попередній перегляд із низькою роздільною здатністю, "інкапсульований" всередині документа, що дозволяє деяким програмам відображати попередній перегляд на екрані.

Файл EPS містить коментар DSC BoundingBox, що описує прямокутник, який містить зображення, описане у файлі EPS. Програми можуть використовувати цю інформацію для викладення сторінки, навіть якщо вони не в змозі безпосередньо відобразити PostScript всередині.

XML

Як можна використовувати графіку в XML? Зробити посилання на HTML або використати XLink. Або вбудувати SVG.

Графіка традиційно є лише посиланнями, які, як правило, мають файл зображення в результаті опрацювання, а не інший фрагмент тексту. Тому вони можуть бути реалізовані будь-яким способом, який підтримується специфікаціями XLink та XPointer, включаючи використання аналогічного синтаксису для існуючих зображень HTML. На них також можна посилатися, використовуючи вбудований механізм NOTATION та ENTITY XML аналогічно стандартному SGML, як і зовнішні нерозділені об'єкти.

Однак масштабована векторна графіка (специфікація XML для векторної графіки) дозволяє використовувати розмітку XML для малювання об'єктів векторної графіки безпосередньо у вашому XML-файлі. Це забезпечує величезну потужність для включення портативної графіки, особливо інтерактивної або анімованої послідовності, і тепер вона стає підтримуваною в браузерах, і їх можна експортувати зі стандартних графічних програм (малюнків), таких як GIMP.

Характеристики посилань на XML для зовнішніх зображень дають набагато кращий контроль над переходом та активацією посилань, тому автор може, наприклад, показувати зображення під час завантаження сторінки чи вибору мишкою в окремому вікні, не вдаючись до сценаріїв. Сам XML не передбачає або обмежує формати графічних файлів: GIF, JPG, TIFF, PNG, CGM, EPS та SVG. Проте векторні формати (EPS, SVG), як правило, необхідні для нефотографічних зображень (діаграм).

Вбудована двійкова графіка

Не можна вбудовувати необроблений бінарний графічний файл (або будь-який інший бінарний [нетекстовий]) безпосередньо у XML-файл, оскільки будь-який байт, що нагадує розмітку, буде неправильно інтерпретований: потрібно зробити на нього посилання. Однак можна включити текстово-кодовану трансформацію бінарного файлу як розділ, позначений CDATA, використовуючи щось на зразок UUencode з символами розмітки], та >, і викреслити з карти, щоб вони не були подані як помилка CDATA і припинення послідовності та бути неправильно інтерпретованим. Ви навіть можете використовувати просте шістнадцяткове кодування, як використовується у PostScript. Однак для векторної графіки рішення - використовувати SVG.

42. Растрові і векторні методи малювання. Зберігання растрових і векторних зображень. Формати растрові і векторні.

Растрова графіка складається з крихітних квадратів, званих пікселями. Після створення растрової графіки певного розміру (тобто фіксованої кількості пікселів) її неможливо змінити без втрати якості зображення. Чим більша кількість пікселів у зображенні, тим більший розмір файлу - між цими величинами є прямопропорційна залежність, оскільки комп'ютеру потрібно зберігати інформацію про кожен піксель. Широко використовуються формати растрових файлів: .jpg, .png, .gif, .bmp та .tiff.

Векторна графіка використовує математичні рівняння, щоб намалювати свої проекти. Ці математичні рівняння переводяться в точки, які з'єднані або лініями, або кривими, також відомими як векторні контури, і вони складають усі різні фігури, які ви бачите у векторній графіці. Це дозволяє масштабувати векторну графіку до будь-якого розміру без шкоди для якості зображення, а також підтримувати невеликий розмір файлу. Поширені векторні формати файлів: .svg, .cgm, .odg, .eps і .xml.

43. Принципи будови графічних редакторів на основі растрових і на основі векторних зображень. Характеристика особливостей растрового і векторного малювання, переваги і недоліки кожного методу.

Растрова графіка складається з крихітних квадратів, званих пікселями. Після створення растрової графіки певного розміру (тобто фіксованої кількості пікселів) її неможливо змінити без втрати якості зображення. Чим більша кількість пікселів у зображенні, тим більший розмір файлу - вони позитивно співвідносяться, оскільки комп'ютеру потрібно зберігати інформацію про кожен піксель. Широко використовуються формати растрових файлів: .jpg, .png, .gif, .bmp та .tiff. *Векторна графіка* використовує математичні рівняння, щоб намалювати свої проекти. Ці математичні рівняння переводяться в точки, які з'єднані або лініями, або кривими, також відомими як векторні контури, і вони складають усі різні фігури, які ви бачите у векторній графіці. Це дозволяє масштабувати векторну графіку до будь-якого розміру без шкоди для якості зображення, а також підтримувати невеликий розмір файлу. Поширені векторні формати файлів: .svg, .cgm, .odg, .eps і .xml.

Растрові графічні редактори оптимальні для цифрового редагування фотографій, оскільки растрова графіка здатна відображати кращу глибину кольорів. Кожен піксель може бути будь-яким із 16 мільйонів різних кольорів. Але якщо ви не працюєте з цифровими фотографіями, редактори векторної графіки були б найкращими для всіх інших типів редагування дизайну, тим більше, що векторну графіку можна чітко масштабувати та маніпулювати будь-яким розміром. Також важливо враховувати розмір файлу.

Якщо ви шукаєте менший розмір файлу, дотримуйтесь векторної графіки. Файли растрових зображень можуть бути досить великими, оскільки комп'ютер повинен запам'ятати інформацію про кожен піксель. Вибір графічного типу залежить від того, який тип дизайну ви створюєте.

Порівняння можна звести до наступної таблиці:

Растрова графіка	Векторна графіка

Растрові зображення також відомі як растрові зображення і складаються з маленьких крапок, відомих як пікселі.	Векторні зображення складаються з ліній, заливок та кривих.
У Растрі пікселі одного кольору або різного кольору розташовуються близько один до одного, щоб людське око сприймало їх як картинки, а не як окремі крапки.	Векторні зображення - це колекції таких елементів, як лінії, прямокутники, кола та квадрати. Кожен векторний елемент має власні координати і його можна змінити.
Растрове зображення зберігає колір кожного окремого пікселя.	Вектор складається з інструкцій, які вказують де розмістити його компоненти.
Растр має лише два кольори, чорний або білий. Зі збільшенням складності він може містити більше кольорів.	Вектори використовують математичні формули при описі кольорів, фігур та розміщення.
Растр має простий вихід, якщо RIP або принтер потребує пам'яті.	Вектори - це невеликі файли, що містять інформацію про криві Безьє, що формують креслення.
Растрові зображення мають широкий діапазон кольорових градацій і менш масштабовані, ніж векторні зображення.	Векторні зображення набагато масштабніші, ніж растрові.
Коли користувач малює растрове зображення, це схоже на занурення пензля у фарбу та малювання зображень.	Коли користувач малює векторне зображення, малюється лише контур фігури. Подібно до створення зображення з різними плитками різної форми та розміру.
При збільшенні растрового зображення без зміни кількості пікселів зображення виглядає розмитим. Збільшення файлу зі збільшенням кількості пікселів, що може дати кращі результати порівняно.	При збільшенні векторного зображення математичні формули залишаються незмінними, масштабованими до будь-якого розміру.
Растрові програми найкраще підходять для редагування фотографій, створення безперервного тонованого зображення з м'яким кольором.	Вектори найкраще підходять для креслень, створення логотипів, ілюстрацій та технічних креслень
Величезні розміри та чітке детальне зображення створюють великий розмір зображення.	Вектори залежать від роздільної здатності, їх можна друкувати будь-якого розміру або роздільної здатності.
Важко друкувати растрові зображення з використанням обмежених точкових кольорів.	У векторному режимі кількість кольорів для друку можна збільшувати або зменшувати.
Перетворення растра у вектор забирає багато часу та засноване на складності.	Векторне зображення можна легко перетворити на растрове зображення.

Растр має високу швидкість обробки, що використовується від значків до плакатів.	У векторному обсязі дані залежать від фактичної вартості об'єкта.
Своєрідний растровий файл можна легко перетворити на інший.	Вектори важко модифікувати або відобразити, якщо вони не відкриваються в програмах, що розуміють вектор.
Розширення форматів файлів для растру: .tif .tiff: Tagged Image File Format .jpg .jpeg: Joint Photographic Experts Group .psd: Photoshop Document .gif: Graphics Interchange Format .png: Portable Network Graphics	Розширення формату файлу для вектора: .eps: Encapsulated PostScript .ai: Adobe Illustrator Artwork .cdr: CorelDraw .svg: Scalable Vector Graphics .pdf: Portable Document Format

Основною перевагою розробки додатків для роботи із растровою графікою є відносна простота та очевидність програмування, адже додаток завждт редагуватиме конкретні пікселі конкретним чином. З іншого боку, окремі операції для таких додатків розробити складно через те, що растрові файли займають дуже багато пам'яті.

Основною перевагою розробки додатків для роботи із векторною графікою є легкість відміни операції’ чи збереження їх операції, адже ці кроки вимагають збереження відносно невеликої кількості інстррукці’, а розмір результуючого файлу виходить відносно невеликим. З іншого боку, програмування коректної роботи інтерфейсу для малювання в таких додатках є значно складнішим чим в додатках для роботи з растровою графікою

44. Події Windows, зв'язані з малюванням і відновленням зображень у вікні. Перелік подій та їх характеристика.

45. Стандартні класи системи програмування для малювання. Класи без зберігання малюнка (без поля пам'яті) і класи зі зберіганням малюнка у власній пам'яті.

Клас Pen

З допомогою класу Pen створюється об’єкт - перо, який використовують для малювання ліній. Подаємо огляд потенційних параметрів класу. В різних системах програмування можуть бути реалізовані такі самі параметри або подібні за змістом. Остаточна реалізація залежить від можливостей бібліотеки графічних функцій.

Клас Brush

Об’єкти класу Brush (пензлі) використовують для заповнення внутрішнього простору замкнених фігур.

Клас Font

Визначає конкретний формат тексту, включаючи шрифт, його розмір і атрибути стилю.

Клас FontFamily

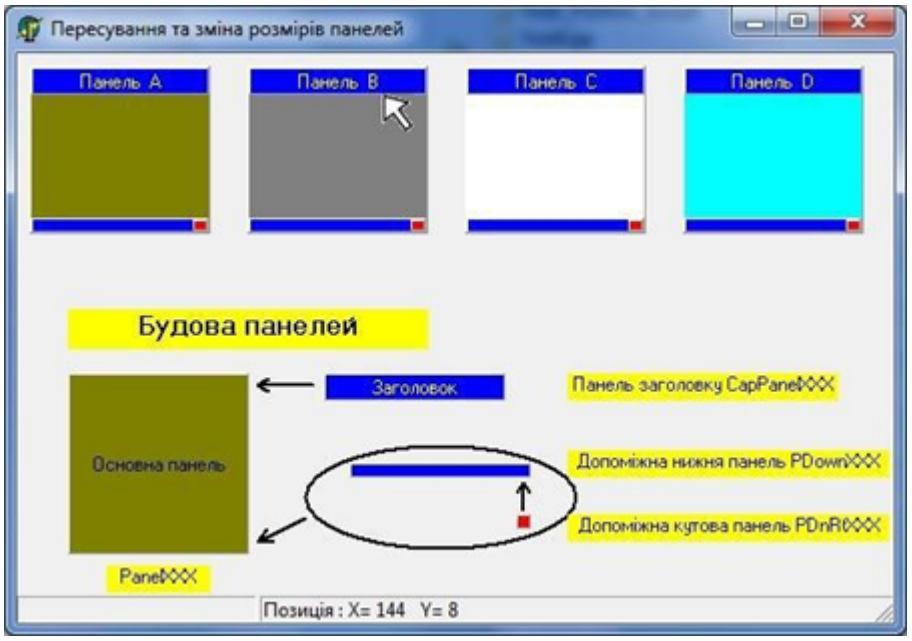
Визначає групу гарнітур шрифту зі схожим базовим макетом та певними відмінностями у стилі.

46. Поняття про пересування та зміну розмірів видимих елементів вікна під час виконання програми.

Якщо потрібний код:
https://www.codeproject.com/Tips/709121/Move-and-Resize-Controls-on-a-Form-at-Runtime-With

Розглянемо деякі прийоми програмування щодо пересування та зміни розмірів видимих елементів форми програми, стандартні для Windows. Зазначимо, що мова йде про пересування та зміну розмірів під час виконання програми за нашими командами, наприклад, від мишки. Для програмування треба вибрати за основу деякі події. У випадку мишки найкраще скористатись подіями MouseDown, MouseMove, MouseUp.

Нехай, наприклад, маємо проєкт форми програми, як показано на рисунку:



Зауваження до проєктування. Розташовуємо на формі для прикладу 4 панелі поряд (PanelA, PanelB, ...). Зверху на кожну панель накладаємо іншу - меншу, яка відіграватиме роль заголовка (CapPanelA, CapPanelB, ...). Знизу на кожну панель також накладаємо вузеньку допоміжну панель (PDownA, PDownB, ...), а на цю допоміжну справа - ще одну допоміжну кутову, за допомогою якої змінюватимемо розміри основної панелі (PDnRtA, PDnRtB, ...). Колір заголовної панелі - синій, підпис на ній - білий, тоді така панель є дуже подібною до заголовка звичайного вікна Windows. Допоміжні панелі знизу мають бути невеликого розміру по висоті, їхній колір можна обрати довільно. Важливо для допоміжної кутової панелі визначити вид курсора як діагональної двоспрямованої стрілки, тоді під час виконання програми будемо бачити момент можливої зміни висоти та ширини основної панелі.

Зв'язування панелей між собою можна здійснити так: під час проектування розташувати їх на формі де завгодно, а на початку виконання програми (за подією, наприклад, OnCreate форми) визначити для всіх допоміжних панелей відповідні їм батьківські панелі (Parent) та спосіб вирівнювання Align щодо батьківської. В такому разі події одних елементів (дочірніх, надбудованих, зверху) можна передавати

(транслювати) до батьківських елементів, а вже батьківські елементи будуть реагувати і виконувати потрібні операції.

На формі знизу розташуємо панель статусу StatusBar класу TStatusBar, з вирівнюванням alBottom, де записуватимемо позицію панелі, яка пересувається чи змінює розміри. А до панелі StatusBar додаємо через Object Inspector і властивість Panels два елементи класу TStatusPanel – окремі поля цілої панелі для різних даних.

47. Події миші і клавіатури, які можна використати для пересування і зміни розмірів видимих елементів вікна під час виконання програми. Параметри подій.

Для програмування треба вибрати за основу деякі події. У випадку мишки найкраще скористатись подіями MouseDown, MouseMove, MouseUp.

MouseDown

Ця подія відбувається при натисканні користувачем кнопки миші, коли курсор миші знаходиться на елементі управління. Оброблювач цієї події бере аргумент типу MouseEventArgs.

MouseMove

Ця подія виникає при переміщенні вказівника миші на елемент керування. Оброблювач цієї події бере аргумент типу MouseEventArgs.

MouseUp

Ця подія виникає, коли вказівник миші знаходиться на елементі управління і користувач відпускає кнопку миші. Оброблювач цієї події бере аргумент типу MouseEventArgs.

MouseWheel

Ця подія виникає, коли користувач обертає коліщатко миші, коли фокус знаходиться на елементі управління. Оброблювач цієї події бере аргумент типу MouseEventArgs. Для визначення того, наскільки прокручено коліщатко миші, можна використовувати властивість Delta елемента MouseEventArgs.

У випадку клавіатури:

KeyDown

Ця подія виникає, коли користувач натискає фізичну клавішу. Оброблювач KeyDown отримує: Параметр KeyEventArgs, який надає властивість KeyCode (вказує на фізичну клавішу клавіатури). Властивість Modifiers (SHIFT, CTRL або ALT). Властивість KeyData (яке об'єднує код клавіші і модифікатор). Параметр KeyEventArgs також надає: Властивість Handled, яке може бути задано для запобігання отримання коду клавіші базовим елементом управління. Властивість SuppressKeyPress, яке може використовуватися для придушення подій KeyPress і KeyUp для даного натискання клавіші.

KeyPress

Ця подія виникає якщо в результаті натискання клавіші або клавіш виходить символ. Наприклад, користувач натискає клавіші SHIFT і малу "a", в результаті виходить символ великої літери "A".

KeyPress виникає після KeyDown. Оброблювач KeyPress отримує: Параметр KeyPressEventArgs, який містить код символу натиснутоюклавіші. Цей код є унікальним для кожної комбінації клавіш символу і модифікатора. Наприклад клавіша "A" створить - код символу 65, якщо вона натиснута, утримуючи клавішу "SHIFT"

- Або клавіша CAPS LOCK поверне код 97, якщо вона натиснута сама по собі, - І код 1, якщо вона натиснута спільно з клавішею CTRL.

KeyUp

Ця подія виникає, коли користувач відпускає фізичну клавішу.

Оброблювач KeyUp отримує:

Параметр KeyEventArgs, - який надає властивість KeyCode (вказує на фізичну клавішу клавіатури).

- Властивість Modifiers (SHIFT, CTRL або ALT).

- Властивість KeyData (яке об'єднує код клавіші і модифікатор).

48. Об'єкт пересування та зміни розміру, зміщення об'єкта в процесі пересування, зовнішні розміри об'єкта, поточна канва малювання.

Щоб перемістити об'єкт:

Використовуйте клавіші зі стрілками на клавіатурі, щоб перемістити об’єкт горизонтально або вертикально. Об'єкт рухатиметься на 1 піксель у напрямку натиснутої клавіші зі стрілкою. Ви також можете виконати ці дії, утримуючи клавішу Shift. Це призведе до переміщення об’єкта з кроком 10 пікселів у обраному напрямку.

Клацніть на об'єкт і, утримуючи ліву кнопку миші, перетягніть об'єкт на нове місце. Щоб перемістити об’єкт у нове положення, яке знаходиться безпосередньо над, під або з будь-якої сторони його поточного положення, утримуючи клавішу Shift, перетягуючи елемент у нове місце. Це призведе до того, що об’єкт рухатиметься прямо, вгору, вниз, вліво або вправо.

Вручну встановіть положення об’єкта, змінивши положення X та Y на вкладці Позиція та розмір властивостей об’єкта.

Щоб змінити розмір об’єкта:

Клацніть на будь-який кут або сторону об'єкта, і, утримуючи ліву кнопку миші, перетягніть кут або сторону об'єкта, щоб змінити його розмір. Утримання клавіші Shift на клавіатурі під час виконання цих кроків забезпечить пропорційний розмір об’єкта.

Встановіть розмір об’єкта вручну, змінивши Ширину та Висоту об’єкта на вкладці Позиція та Розмір властивостей об’єкта.

49. Особливості малювання многокутників. Фіксування вершин полігону. Відображення проміжних полігонів. Динаміка малювання ребер полігона. Події, придатні до малювання полігона і відображення динаміки малювання.

Розглянемо кроки малювання стандартних фігур на прикладі прямокутника. Початок фігури фіксується у момент натискання на клавішу мишки (рис.1, а). При пересуванні мишки з натиснутою клавішею малюємо прямокутник від точки початку фігури до точки поточного розташування мишки (рис.1, б). При подальшому пересуванні мишки стираємо попередній прямокутник і малюємо його у новому місці (рис.1, в) – така операція виконується багатократно, при кожній зміні позиції мишки. Остаточний прямокутник фіксується у момент відпускання клавіші мишки (рис.1, г).

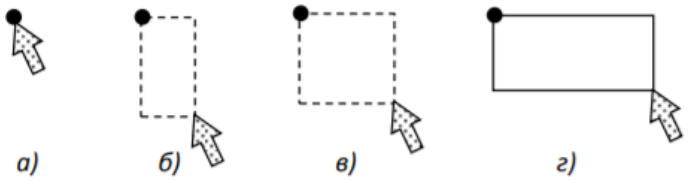


Рис. 1. Кроки малювання стандартної фігури

Особливості малювання многокутника

У загальному випадку замкнений многокутник (в іншій термінології – полігон) можна малювати описаним вище методом. Однак усі проміжні полігони до остаточного вибору не показують перспективи продовження шляхом додавання нових ланок і дають неправильну уяву про кінцевий результат. Тому варто у процесі побудови малювати лише окремі ребра, які вже вибрано (Polyline). Необхідно домогтися фіксування вершин полігона. З цією метою можна використати подію OnClick. Наприклад, на рис.2, а зазначено вже побудовану частину полігона. Наступне ребро для продовження полігона малюємо подібно, як звичайну лінію, описаним вище методом (рис.2, б, в), лише пересувати мишку можна й без натисненої клавіші. Наступне ребро фіксується подальшим натисненням клавіші мишки (подія OnClick) (рис.2, г). Закінчення побудови полігона – за подією OnDblClick, при цьому остання вершина сполучається з найпершою (рис.2, д).

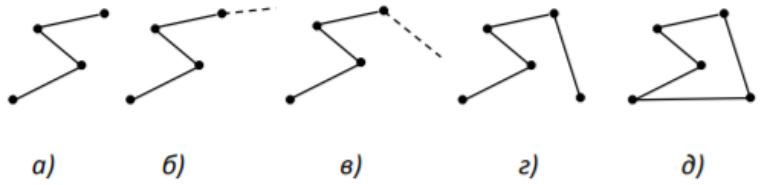


Рис. 2. Кроки малювання полігона

50. Масштабування зображень в процесі малювання. Проблеми, які можуть виникати при ручному програмуванні масштабування.

Масштабування зображення відбувається методом копіювання певної ділянки зображення і “розтягування” його до потрібного розміру.

Розглянемо приклад:
```
//завантажимо зображення:
Bitmap myBitmap = new Bitmap("Image.jpg");

//створимо прямокутник в який покладемо оригінальне зображення
Rectangle sourceRectangle = new Rectangle(80, 70, 80, 45);

//створимо прямокутник в який покладемо збільшене зображення
Rectangle destRectangle2 = new Rectangle(200, 40, 200, 160);

//Малюємо оригінальне зображення:
myGraphics.DrawImage(myBitmap, 0, 0);

//Малюємо масштабоване зображення:
myGraphics.DrawImage(myBitmap, destRectangle2, sourceRectangle, GraphicsUnit.Pixel);
```

Однією з основних проблем при масштабуванні залишається співвідношення сторін, потрібно вручну вирахувати співвідношення для вхідного зображення. Після цього треба масштабувати лише один вимір (наприклад висоту), а ширину масштабувати помноживши висоту на коефіцієнт співвідношення.

51. Опрацювання зображень способом фільтрування. Приклади фільтрів та їх формули. Застосування фільтрів до частини загального зображення.

52. Контекстні підказки про хід виконання графічних операцій. Перемикання контекстом підказки

Список питань до розділу "Електронні таблиці"

53. Означення термінів "активна комірка", "впливаюча комірка", "залежна комірка", "ряд даних", "властивості комірки". Алгоритм розпізнавання статусу комірки.

Активна комірка - це комірка, у яку вводяться дані і яка виділена чорною рамкою
Впливаюча комірка - це комірка, на яку посилається одна або більше формул
Наприклад, якщо клітинка D10 містить формулу =B5, то клітинка B5 впливаюча для клітинки B5.
Залежна комірка – це комірка, яка містить містять формули що посилаються на інші клітинки.
Наприклад, якщо клітинка D10 містить формулу =B5, то клітинка D10 залежна від клітинки B5.
Ряд даних – це рядок або стовпець чисел, які вводяться на аркуші та наносяться на діаграму, наприклад список квартальних прибутків.
Властивості комірки - загальні особливості комірки та тексту в ній. До таких властивостей можна віднести наприклад шрифт та колір тексту, стиль та колір межі, колір фону, вирівнювання та форматування чисел.

54. Формули в комірках ЕТ. Структура формули на рівні змістового призначення. Типи можливих синтаксичних і семантичних помилок в формулах. Критерії визначення помилки.

Формули:
<формула> ::= = <вираз>
<вираз> ::= <доданок> | <вираз> + <доданок> <вираз> -<доданок>
<доданок>::= <множник> | <доданок> * <множник> | <доданок> / <множник>
<множник> ::= <число> | <комірка> | <функція> | (<вираз>)
<комірка> ::= <букви><цифри>
<букви> ::= буква | буква буква
<цифри> ::= цифра | цифра цифра
<функція> ::= SUM (<ряд>) | MAX (<ряд>) | MIN (<ряд>) |
ABS (<комірка>) | INT (<комірка>)
<ряд> ::= <комірка> : <комірка>
Знаки (однолітерні розділювачі) кодуємо числами, починаючи від 101. Імена функцій кодуємо числами, починаючи від 201. Сканер фактично повинний розпізнавати лише такі групи лексем: однолітерні розділювачі (знаки); ідентифікатори (імена функцій і комірок); числа (різних форматів); літерали в одинарних лапках; літерали в подвійних лапках. Все решту вважається помилкою запису формули.Головне правило для розпізнавання лексемє таке: групу, до якої належить поточна лексема.
if (цифра) { група чисел }
else if (буква) { група ідентифікаторів }
else if (допустимий знак) { група однолітерних розділювачів }
else if (одинарний апостроф) { літерал в одинарних лапках }
else if (подвійний апостроф) { літерал в подвійних апострофах }
else { помилка: недопустима літера }

55. Граматики для розпізнавання типу комірки. Алгоритм розпізнавання.

Перевірте якщо мб хтось бачив щось схоже

Розпізнавання типу комірки

```
struct ETindex // індекс таблиці
{
int row,col; // індекси рядка і стовпця електронної таблиці
};
struct Ttoken  // визначення лексеми
{
int token;    // тип лексеми
// уточнення змісту лексеми
union // спільна ділянка пам'яті (накладання)
{
ETindex index; // індекси рядка і стовпця
double cellsvalue; // або значення числа в комірці
char * lit;     // текстовий рядок (літерал)
};
};
```

Величина token є головною і визначає тип лексеми. Величини row і col, які входять в структуру ETindex, є допоміжними і визначають уточнюючі параметри лексеми , а саме – номер рядка і номер стовпця комірки електронної таблиці, на яку є посилання в даній лексемі.Величина cellsvalue визначає безпосередньо число, якщо воно є значенням лексеми. З метою економії пам'яті використовуємо накладання величин (union) index і cellsvalue , враховуючи ту обставину, що лексема не може бути одночасно адресою комірки і числом. Величина lit є вказівником на текстовий рядок, що є значенням лексеми, у випадку, якщо лексема є літералом.

Алгоритм розпізнання

Головне правило для розпізнавання лексем є таке : групу, до якої належить поточна лексема, можна визначити за першою літерою лексеми:

if (цифра) { група чисел }
else if (буква) { група ідентифікаторів }
else if (допустимий знак) { група однолітерних розділювачів }
else if (одинарний апостроф) { літерал в одинарних лапках }
else if (подвійний апостроф) { літерал в подвійних апострофах }
else { помилка: недопустима літера }

56. Поняття лексеми формули. Перелік лексем на прикладі граматики формули. Формальне визначеня лексем.

Лексема (токен) - кожен окремий компонент (слово) певного виразу, послідовність символів граматики якоїсь мови, що мають певне сукупне значення.

Правила будови синтаксичних визначень

Для точного визначення будови деякої синтаксичної конструкції приймемо такі позначення. Їх використання є загальноприйнятим, і має основою формули БНФ:
- нетермінали позначаємо звичайним ідентифікатором, наприклад number;
- термінали позначаємо лапками, наприклад "+", "(", "sqrt";

- круглі дужки використовуємо подібно, як в алгебрі, для групування інших конструкцій, наприклад ("+" | "-" | "*" | "/"); самі дужки в такому записі не є частиною конструкції;
- вертикальна риска | означає "або", тобто вибір однієї з альтернатив;

- знак зірочки * означає повторення нуль або більше разів конструкції, записаної перед зірочкою, наприклад cipher * ; сама зірочка не є частиною конструкції;

- квадратні дужки [ ] означають або відсутність, або однократне входження в синтаксичний запис, наприклад [prompt] ;

- знак ::= читаємо "за означенням є".

Інтерпретація формул загальних алгебраїчних правил

Розглянемо записи формул, виконані відомими алгебраїчними правилами Операції поділені на ранги: 1-й ранг "множення" і "ділення", другий ранг "додавання" і віднімання". Будемо для початку вважати, що інших операцій немає. Запис круглих дужок означає зміну порядку виконання операцій. Операції однакового рангу виконують зліва направо.

Для першого вивчення розглядаємо формули, які складаються лише з цілих чисел, наприклад:

49 – 108 / (6 + 11) * (100 – 94)

Тепер треба дати точне синтаксичне означення правил запису формули, врахувавши загальні алгебраїчні правила. Граматика матиме наступний вигляд:

arith_expr ::= term (("+" | "-") term) *
term ::= factor (("*" | "/") factor) *
factor ::= number | "(" arith_expr ")"
number ::= cipher cipher *

cipher ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Аксіомою цієї граматики є arith_expr. З точки зору правил граматичного розбору і будови виведень в граматиках приорітет буде зростати при переході до правила, яке використовують для підстановки. Наприклад, якщо маємо 20+10*4, тоді виведення виглядає так в лівосторонньому порядку (для спрощення запису лапки не показуємо, деякі підстановки об'єднуємо):

arith_expr => term + term => factor + term => number + term => 20 + term =>

=> 20 + factor * factor => 20 + number * factor => 20 + 10 * factor => 20 + 10 * number => =>20+10*4

Операції будуть виконані в оберненому порядку до кроків виведення. Спочатку буде виконана операція множення, після неї – операція додавання. Операцію можна виконати лише тоді, коли обидва операнди обчислені.

Реалізацію граматики виразів і розпізнавання виразу реалізуємо методом нисхідного граматичного розбору за алгоритмом рекурсивного спуску. З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил. Для нашого випадку ця вимога виконана.

За методом рекурсивного спуску до кожного нетермінального символа будують окрему функцію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для arithexpr буде викликати функцію для term, функція для term буде викликати функцію для factor, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, arithexpr term factor (arithexpr) … .

Вхідними даними для програми обчислення виразу є масив лексем, побудований сканером. Арифметичний вираз може мати довільний вигляд в межах записаних вище синтаксичних правил. Тому доцільно підготувати список лексем в формі, зручній для синтаксичного розбору. Кожну лексему можна подати у вигляді (код,значення), де код – числове позначення лексеми кожного виду. Приймемо такі позначення лексем:

number	1	значення числа
openbracket	3	"("
closebracket	4	")"
add	5	"+"
subtract	6	"-"
multiply	7	"*"
divide	8	"/"

Список лексем легко отримати, переглядаючи синтаксичні правила визначення формули.

Маючи записані раніше правила для арифметичних формул, їх можна розширювати для отримання додаткових можливостей. Наприклад, ми хочемо мати ще таке:

1) бінарні операції ділення на ціло //, остача від ділення %;

2) піднесення до степеня **;

3) додати функцію sin(x);

4) унарні операції + і -;

5) крім цілих чисел дозволити дійсні числа фіксованої крапки і в експоненціальній формі, наприклад 45.02, 1.0e-5; зафіксуємо вимогу, що запис будь-якого числа завжди починається з цифри.

Змінимо правила так, щоб зберегти раніше визначену частину правил, додати нові операції і врахувати приорітети нових операцій. Тоді нова граматика буде мати такий вигляд:

```
arith_expr ::=   term     ( ( "+" | "-" )     term ) *
# term ::=       factor   ( ( "*" | "/" )     factor ) *          # було
term ::=factor   ( ( "*" | "/" | "//" | "%" )  factor ) *
# factor ::=     number |          "(" arith_expr ")"          # було
factor ::= [( + | - )] ( number | "(" arith_expr ")" | function ) [( "**" factor )*]
number ::=       cipher cipher * [ . cipher cipher * ] [ (e|E) [(+|-)] cipher cipher * ]
function ::=     "sin(" arith_expr ")"
cipher ::=       "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Множником (factor) тепер вважаємо елемент (число, вираз в дужках, функція), який може мати операцію ** піднесення до степеня. Показник степеня знову трактуємо як множник. Зауважимо, що при такому визначенні декілька підряд записаних операцій піднесення до степеня без дужок будуть виконані справа наліво: 2**3 **4 означає 2**(3 **4), а не (2**3)**4

Крім того, додаємо можливість унарної операції + або – перед множником.

57. Загальна схема сканера. Визначення структур або класів для розпізнавання лексем сканером. Перелік параметрів лексем для їх опрацювання.

    Сканер – це програма, яка текст однієї формули ділить на окремі лексеми і кодує лексеми певним способом, наприклад, цілими числами. Отже, замість тексту формули будемо мати масив цілих чисел, що дозволить більш ефективно будувати програми перетворення і обчислення формул.

    Наприклад, якщо маємо формулу = B5+B6-SUM(C4:C10)/2

    то поділ на лексеми мав би виглядати так:

    Щоб програмувати сканер, потрібно укласти повний список всіх допустимих лексем і визначити таблицю кодування лексем. В основі таблиці кодування має бути деякий принцип чи загальний підхід. Наприклад, для кожної лексеми можна надати три специфікатори: перший визначає тип

    лексеми, а один або два наступні уточнють зміст лексеми. Отже, для однієї лексеми можна визначити такі структури:

    struct ETindex // індекс таблиці

    {

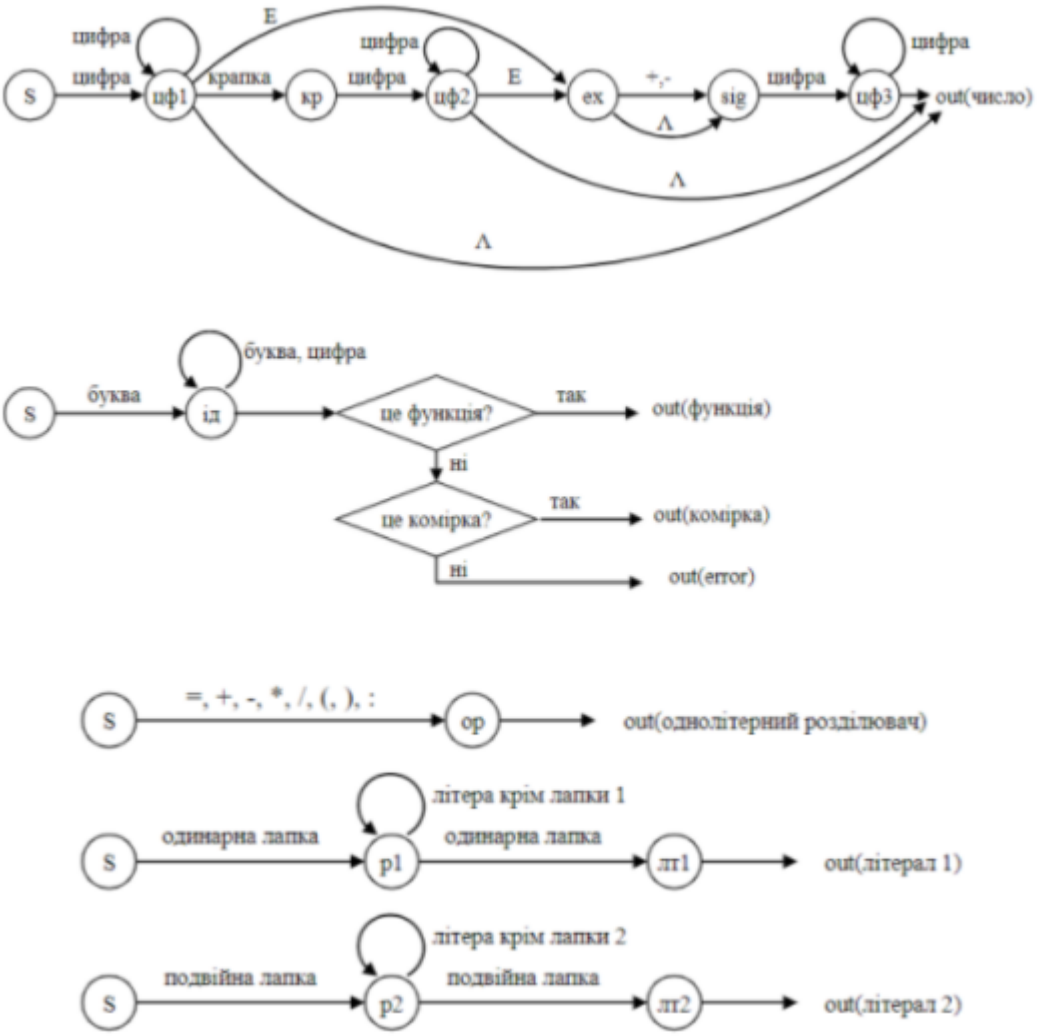
        int row, col; // індекси рядка і стовпця електронної таблиці

```
};
struct Ttoken // визначення лексеми
{
	int token; // тип лексеми
	union// спільна ділянка пам'яті (накладання)
	{
		ETindex index; // індекси рядка і стовпця
		double cellsvalue; // або значення числа в комірці
		char * lit; // текстовий рядок (літерал)
	};
};
```

Величина token є головною і визначає тип лексеми. Величини row і col, які входять в структуру ETindex, є допоміжними і визначають уточнюючі параметри лексеми, а саме – номер рядка і номер стовпця комірки електронної таблиці, на яку є посилання в даній лексемі. Величина cellsvalue визначає безпосередньо число, якщо воно є значенням лексеми. З метою економії пам'яті використовуємо накладання величин (union) index і cellsvalue, враховуючи ту обставину, що лексема не може бути одночасно адресою комірки і числом. Величина lit є вказівником на текстовий рядок, що є значенням лексеми, у випадку, якщо лексема є літералом.

58. Алгоритми розпізнавання лексем формул у вигляді діаграми станів скінченного автомата.

Оскільки для розпізнавання лексем потрібно аналізувати по черзі кожну окрему літеру формули, то алгоритми розпізнавання лексем доцільно подати у формі діаграм станів детермінованого скінченного автомата. Нижче подано відповідні діаграми станів. Позначення станів є умовними. Знаком "Λ" позначають пусту літеру, іншими словами, це є перехід без розпізнавання будь-якої літери за умови, що інші переходи неможливі. Out означає вихід з відповідною розпізнаною лексемою.



59. Схема основного циклу сканування формули і кодування лексем.

Сканер може виявляти помилки, які стосуються окремих лексем, і у випадку помилки повідомляє код помилки і лексему з помилкою. У випадку успішного сканування можна прочитати по черзі всі закодовані лексеми.

// допоміжні функції для сканування:

Init() - знайти першу значущу літеру, починаючи з поточної позиції

AddLetterToToken() - дописати чергову літеру до лексеми

GetNextChar() - перейти до наступної літери формули

FunctionNameCode(string xfn) - кодування імені функції xfn числом

GetCodeTok1L(char t) - кодування однолітерних розділювачів

ConvertA1toR1C1(const string & s) - перетворити буквенний номер s стовпця на числовий

// цикл сканування і кодування лексем;

```
//аналіз виконуємо за поточною літерою letter
while (letter != '\0' && !error)
{
    Init(); // знайти першу значущу літеру від поточної позиції
    if( error|| letter== '\0') break; // помилка функції Init() або кінець формули
    // за першою літерою розпізнаємо групу, до якої належить лексема
    if(letter>='0' && letter<='9') // це є число
    {
    // цифри перед крапкою
    while(isdigit(letter)) { AddLetterToToken(); GetNextChar(); }
    if(letter == '.') // може бути крапка
    {
        AddLetterToToken(); GetNextChar(); // сама крапка
        // цифри після крапки
        if( ! isdigit(letter)) // після крапки немає цифри
        {codeerror=1002; error=true; break; }
        else // прочитати цифри після крапки
        while(isdigit(letter)) { AddLetterToToken(); GetNextChar(); }
    }
    // далі за текстом може бути записаний порядок числа
    if(letter=='E') // порядок присутній
    {
        AddLetterToToken(); GetNextChar(); // дописати букву E
        if(letter=='+' || letter=='-') // може бути знак порядку
        { AddLetterToToken(); GetNextChar(); } // дописати знак
        // далі мають бути цифри порядку
        if( ! isdigit(letter)) // після букви E немає цифри
        { codeerror=1003; error=true; break; }
        // дописати цифри порядку
        while(isdigit(letter)) { AddLetterToToken(); GetNextChar(); }
    }
```

```
// перетворити число з текстової форми в double і записати лексему
double x = atof(tx.c_str());
toklist[N].token=number_et;  toklist[N].cellsvalue=x;
}
else if(letter>='A' && letter<='Z') // це комірка або функція
{
// спочатку прочитати всі букви
while(letter>='A' && letter<='Z') {AddLetterToToken(); GetNextChar();}
// далі можуть бути цифри і букви в довільному порядку
while(isdigit(letter) || letter>='A' && letter<='Z')
{ AddLetterToToken(); GetNextChar(); }
// перевіримо, чи це є іменем функції
int fnc = FunctionNameCode(tx);
if(fnc) { toklist[N].token=fnc; } // знайшли ім'я функції
else // перевіримо, чи це правильне ім'я комірки
{
    // розділити лексему на букви і цифри
    int i=0;
    string tempcol="";
  for(; i<tx.length() && tx[i]>='A'&&tx[i]<='Z'; i++) {tempcol+= tx[i];}
    string temprow="";
    for(; i<tx.length() && isdigit(tx[i]); i++)  {temprow+= tx[i];}
    int numrow=atoi(temprow.c_str());
    Int numcol=ConvertA1toR1C1(tempcol);
    if(numrow>=1 && numrow<=99 && numcol>=1 && numcol<=256&& i>=tx.length())
    { // правильна лексема
    toklist[N].token=cell_et;
    toklist[N].index.col=numcol;
    toklist[N].index.row=numrow;
    }
    else // помилка
```

```
    {
    codeerror=1004; error=true; break;
    // неправильне ім'я комірки або функції
    }
}
}
else if(GetCodeTok1L(letter)>0) // це однолітерний розділювач
{
toklist[N].token=GetCodeTok1L(letter);GetNextChar          (); // до наступної літери формули
}
else if(letter=='\'') // це літерал з одинарними апострофами
{
// читати до кінця літерала
GetNextChar(); // наступна літера після апострофа
while(letter != '\'' && letter != '\0') {    AddLetterToToken(); GetNextChar(); }
toklist[N].token=lit1;
toklist[N].lit= new char[tx.length()+1]; // надати пам'ять
strcpy(toklist[N].lit,tx.c_str()); // копіювати літерал
GetNextChar(); // перейти до наступної літери за апострофом
}
else if(letter=='\"') // літерал з подвійними апострофами
{ // читати до кінця літерала
GetNextChar(); // наступна літера після апострофа
while(letter != '\"' && letter != '\0') { AddLetterToToken(); GetNextChar(); }
toklist[N].token=lit2;
toklist[N].lit= new char[tx.length()+1]; // надати пам'ять
strcpy(toklist[N].lit,tx.c_str()); // копіювати літерал
GetNextChar(); // перейти до наступної літери за апострофом
}
else // помилка -невизначена лексема
{
```

```
    AddLetterToToken(); // запам'ятати недопустиму літеру в лексемі
    codeerror=1001; error=true;  // невизначена лексема (недопустима літера);
    }
}
```

60. Програмовані функції для кодування і декодування адресів комірок ЕТ. Зображення комірок як лексем.

Отже подивимось на реалізацію методів для кодування і декодування лексеми. Реалізацію деяких окремих функцій упущено та додано коментарі.
Це копіпаст з лекції Черкаша
Кодування лексем:

```
while (letter != '\0' && !error)
{
        Init(); // знайти першу значущу літеру від поточної позиції
        if ( error || letter == '\0') break; // помилка функції Init()
        // або кінець формули
        // за першою літерою розпізнаємо групу, до якої належить лексема
        if(letter>='0' && letter<='9') // це є число
        {
                // цифри перед крапкою
                while(isdigit(letter))
                {
                        AddLetterToToken();
                        GetNextChar();
                }
        if(letter == '.') // може бути крапка
        {
                AddLetterToToken(); GetNextChar(); // сама крапка
                // цифри після крапки
                if( ! isdigit(letter)) // після крапки немає цифри
                {
                        codeerror=1002; error=true; break; // після крапки немає цифри
                }
                else // прочитати цифри після крапки
                        while(isdigit(letter))
                        {
                                AddLetterToToken();
                                GetNextChar();
                        }
        } // if(letter == '.')
        // далі за текстом може бути записаний порядок числа
        if(letter=='E') // порядок присутній
        {
                AddLetterToToken();
                GetNextChar(); // дописати букву E
                if(letter=='+' || letter=='-') // може бути знак порядку
```

```
                {
                        AddLetterToToken();
                        GetNextChar();
                } // дописати знак
                // далі мають бути цифри порядку
                if( ! isdigit(letter)) // після букви E немає цифри
                {
                        codeerror=1003;
                        error=true; break; // після букви E немає цифри
                }
                // дописати цифри порядку
                while(isdigit(letter))
                {
                        AddLetterToToken();
                        GetNextChar();
                }
        } // порядок присутній
        // перетворити число з текстової форми в double і записати лексему
        double x = atof(tx.c_str());
        toklist[N].token=number_et; toklist[N].cellsvalue=x;
} // це є число
else if(letter>='A' && letter<='Z') // це комірка або функція
{
        // спочатку прочитати всі букви
        while(letter>='A' && letter<='Z')
        {
                AddLetterToToken();
                GetNextChar();
        }
        // далі можуть бути цифри і букви в довільному порядку
        while(isdigit(letter) || letter>='A' && letter<='Z')
        {
                AddLetterToToken();
                GetNextChar();
        }
        // перевіримо, чи це є іменем функції
        int fnc = FunctionNameCode(tx);
        if(fnc)
        {
                toklist[N].token=fnc;
        } // знайшли ім'я функції
        else // перевіримо, чи це правильне ім'я комірки
        {
                // розділити лексему на букви і цифри
                int i=0;
                string tempcol="";
                for(; i<tx.length() && tx[i]>='A'&&tx[i]<='Z'; i++) tempcol+= tx[i];
                string temprow="";
                for(; i<tx.length() && isdigit(tx[i]); i++) temprow+= tx[i];
                int numrow=atoi(temprow.c_str());
```

```
                        int numcol=ConvertA1toR1C1(tempcol);
                        if(numrow>=1 && numrow<=99 && numcol>=1 && numcol<=256 &&
i>=tx.length())
                        { // правильна лексема
                                toklist[N].token=cell_et;
                                toklist[N].index.col=numcol; toklist[N].index.row=numrow;
                        }
                        else // помилка
                        {
                                codeerror=1004; error=true; break;
                                // неправильне ім'я комірки або функції
                        }
                } // перевірка правильності імені комірки
        } // це комірка або функція
        else if(GetCodeTok1L(letter)>0) // це однолітерний розділювач
        {
                toklist[N].token=GetCodeTok1L(letter);
                GetNextChar(); // до наступної літери формули
        } // це однолітерний розділювач
        else if(letter=='\") // це літерал з одинарними апострофамиТема 12. Сканер формул електронної
таблиці. [Ч.В.В.] 9
        { // читати до кінця літерала
                GetNextChar(); // наступна літера після апострофа
                while(letter != '\" && letter != '\0')
                {
                        AddLetterToToken(); GetNextChar();
                }
                toklist[N].token=lit1;
                toklist[N].lit= new char[tx.length()+1]; // надати пам'ять
                strcpy(toklist[N].lit,tx.c_str()); // копіювати літерал
                GetNextChar(); // перейти до наступної літери за апострофом
        } // це літерал з одинарними апострофами
        else if(letter=='\"') // літерал з подвійними апострофами
        { // читати до кінця літерала
                GetNextChar(); // наступна літера після апострофа
                while(letter != '\"' && letter != '\0')
                {
                        AddLetterToToken(); GetNextChar();
                }
                toklist[N].token=lit2;
                toklist[N].lit= new char[tx.length()+1]; // надати пам'ять
                strcpy(toklist[N].lit,tx.c_str()); // копіювати літерал
                GetNextChar(); // перейти до наступної літери за апострофом
        } // літерал з подвійними апострофами
        else // помилка - невизначена лексема
        {
                AddLetterToToken(); // запам'ятати недопустиму літеру в лексемі
                codeerror=1001; // невизначена лексема (недопустима літера)
                error=true;
        } // помилка - невизначена лексема
```

```
}
```

Декодування:

```
const Ttoken& GetToken(int k) // прочитати лексему номер k
{
    //повертаємо лексему з списку
    if(isscanning && k>=0 && k<=N) return toklist[k];
    else return NulTok; // фіктивно
} // const Ttoken& GetToken(int k)
```

Виклик:

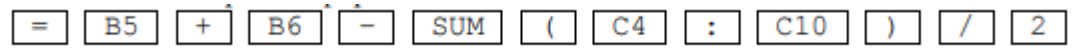
```
int k;
Ttoken leks;
for(k=0; k<Scanner.GetNumberToken(); k++)
{
    leks=Scanner.GetToken(k); // читаємо чергову лексему
    Print1Token(leks); // і друкуємо
} // for
cout << endl;
```

61. Перетворення формул ЕТ з інфіксної форми у постфіксну. Правила перетворення. Алгоритми, необхідні для перетворення.

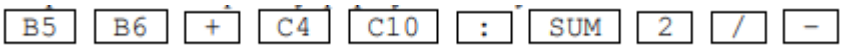
Перетворення виразів – це є реалізація програми, яка інфіксне зображення однієї формули електронної таблиці перетворює на постфіксне. Отже, та сама формула набуває постфіксного вигляду, що дозволяє ефективно реалізувати обчислення формул. Наприклад, якщо маємо формулу:

= B5+B6-SUM(C4:C10)/2

то поділ на лексеми і інфіксна форма виглядали би так:



Після перетворення в постфіксну форму запис буде таким:



В постфіксній формі операція має бути записана за операндами, до яких вона стосується. Знак "=", який є на початку формули, можна викреслити з постфіксної форми, бо він не є операцією, а лише ознакою формули в комірці таблиці. Дужки в постфіксній формі відсутні, бо всі операції виконують зліва направо в порядку записування

Постфіксну форму виразу можна отримати за схемою, наведеною раніше – за допомогою процедур генерування постфіксної форми на основі граматики виразів. Реалізацію граматики виразів і розпізнавання виразу реалізуємо методом нисхідного граматичного розбору за алгоритмом рекурсивного спуску. З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил. При цьому використовують символи дужок "{", "}", "(", ")", "[", "]" як метасимволи, тобто для зображення самої формули. Якщо ж будь-яка дужка є елементом самої формули, то дужку записують в апострофах, наприклад '('

Значення дужок як метасимволів є таким. Фігурні дужки {*} означають, що заключена в них конструкція може повторюватись нуль або більше разів. Квадратні дужки [*] означають, що заключена в них конструкція може бути або відсутня, або присутня лише один раз, тобто альтернатива присутності. Круглі дужки (*) вживаються тоді, коли потрібно зробити вибір однієї з альтернатив, записаних в круглих дужках

Отже, визначені раніше граматичні правила для формул перепишемо в такому вигляді:

Формули:

```
<формула> ::=    = <вираз>
<вираз> ::=  <доданок>  { ( + | - ) <доданок> }
<доданок> ::=  <множник>  {  ( * | / )  <множник>  }
<множник> ::=  <число>  |  <комірка>  |  <функція>  |  '(' <вираз> ')'
<комірка>  ::=  <букви> <цифри>
<букви> ::= буква | буква буква        не використовуємо, бо вже розпізнано
<цифри> ::= цифра | цифра цифра
<функція> ::=  ( SUM | MAX | MIN ) '(' <ряд> ')'
             | ( ABS | INT ) '(' < вираз > ')'
<ряд> ::=  <комірка> : <комірка>
```

За методом рекурсивного спуску до кожного нетермінального символа будують окрему функцію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для <вираз> буде викликати функцію для <доданок>, функція для <доданок> буде викликати функцію для <множник>, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, <вираз> <доданок> <множник> <вираз> … .

Вхідними даними для програми перетворення з інфіксної форми в постфіксну є масив лексем, побудований сканером. Отже, всі лексеми вже мають однаковий числовий формат, що значно спрощує програмування такого Ну я перетворення. Відповідно, вихідними даними цієї програми є перебудований масив лексем в тому самому числовому форматі.

Головна ідея генерування постфіксної форми полягає в правильному визначенні послідовності записування елементів виразу в постфіксну форму. Постфіксна форма – це є спільний масив (потік) для всіх функцій розпізнавання нетермінальних символів. Функції почергово записують в такий спільний масив елементи виразу, які вони розпізнали. Якщо маємо, наприклад, два доданки виразу за формулою

```
<вираз> ::=  <доданок>  + <доданок>
```

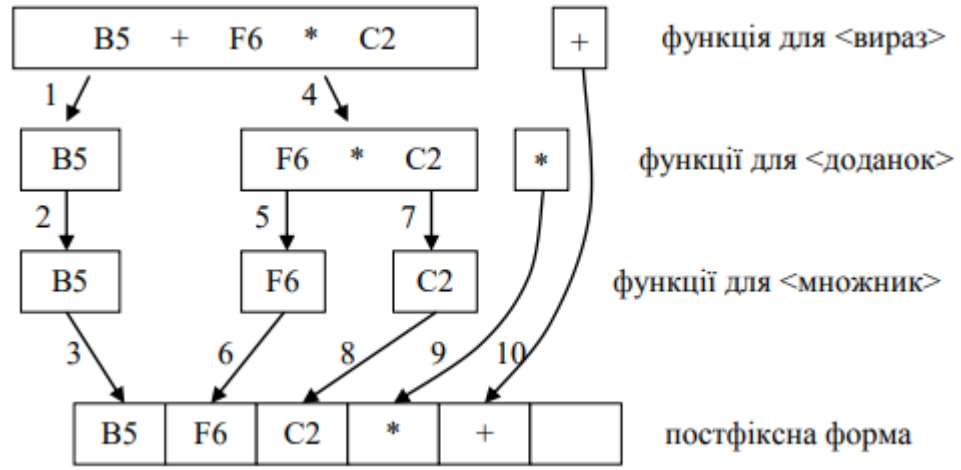
то функція для <вираз> повинна спочатку викликати функцію <доданок> для обчислення першого доданка, потім ще раз цю саму функцію для обчислення другого доданка, після чого записати до постфіксної форми знак операції '+'. Але функції для <доданок> першими запишуть до постфіксної форми свої доданки. Отже, якщо мали, наприклад, вираз B5+F6 , то в результаті вийде B5,F6,+ . Якщо ж вираз буде мати вигляд

B5 + F6 * C2

то будуть використані формули у вигляді

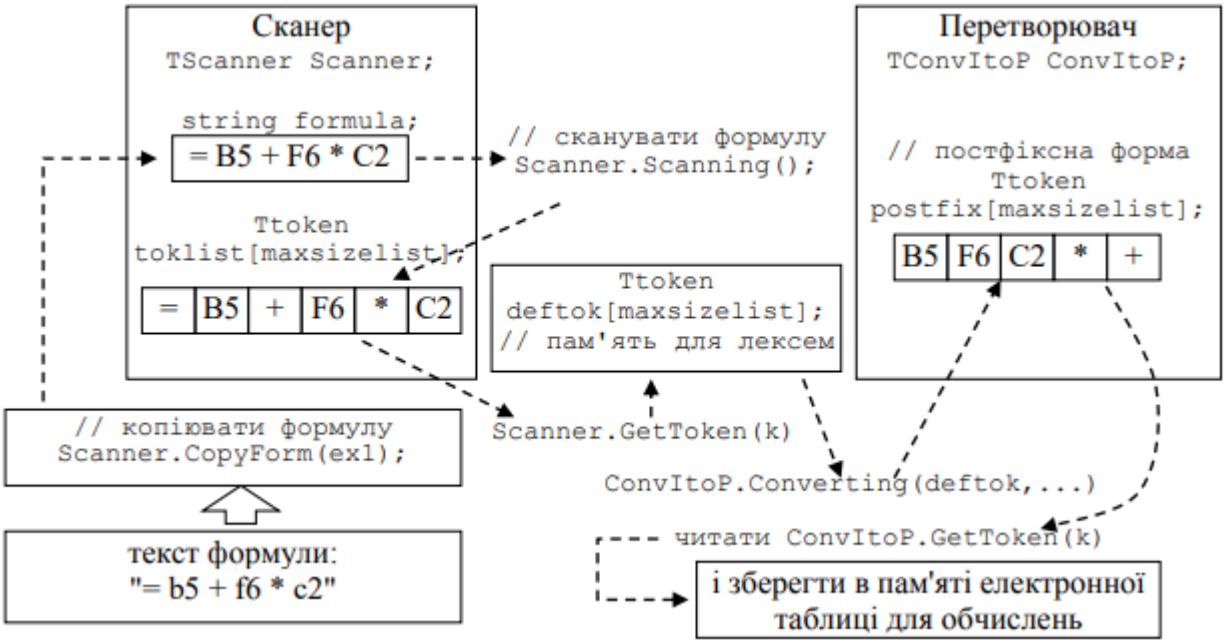
```
<вираз>  ::=   <доданок>   +  <доданок>
<доданок>  ::=   <множник>   |   <множник> *  <множник>
<множник>  ::=    <комірка>
```

В результаті побудова постфіксної форми буде виконана за схемою, як на рисунку.



В процесі перетворення в постфіксну форму можуть бути виявлені різні помилки, пов'язані з неправильним записом самого виразу, наприклад: між двома іменами комірок немає знака операції; відсутня закриваюча кругла дужка у виразі; у тексті формули знак операції записаний двічі підряд '++'; і інші. У випадку знаходження помилки процес перетворення припиняється і повідомляється зміст помилки та її імовірна позиція

Загальну схему взаємодії і використання сканера і перетворювача формул до постфіксної форми можна подати такою діаграмою:



62. Побудова постфіксної форми виразів на основі граматики виразів. Нисхідний граматичний розбір виразів методом рекурсивного спуску.

Єдина інформація, яка є по першому реченню - це те саме що у 61 питанні.

Нисхідний граматичний розбір :

Зазвичай в компіляторах використовують або низхідний, або висхідний методи. В обох методах вхідні дані синтаксичний аналізатор опрацьовує зліва направо. Основний принцип роботи даних методів можна зрозуміти з їх назв: низхідний синтаксичний аналізатор працює зверху-вниз, висхідний – навпаки.

Низхідний синтаксичний аналіз – побудова дерева розбору починаючи із кореня та подальше створення вузлів дерева. До алгоритмів даного типу аналізу належить метод рекурсивного спуску (recursive-descent parsing). Цей метод більш потужний та частіше використовується в реальних компіляторах. Програма синтаксичного аналізу методом рекурсивного спуску складається із набору процедур, по одній для кожного нетерміналу. Робота програми починається із виклику процедури для початкового символу і успішно завершується у випадку сканування всієї вхідної стрічки. В загальному випадку для рекурсивного спуску може виникнути необхідність у поверненні – повторне сканування вхідних даних.

Варто зауважити, що ліворекурсивна граматика може призвести до безкінечного циклу, якщо синтаксичний аналізатор працює методом рекурсивного спуску. Це відбувається тому, що ми щоразу намагатимемося розгорнути нетермінал, який в свою чергу знову ж таки буде намагатися розгорнути цей ж нетермінал. Тоді виникне зациклювання.

63. Модифікація граматичних правил виразів для випадку розбору методом рекурсивного спуску. Питання ліворекурсивних правил і факторизації.

За методом рекурсивного спуску до кожного нетермінального символа будують окрему функцію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для arithexpr буде викликати функцію для term, функція для term буде викликати функцію для factor, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, arithexpr term factor (arithexpr) … . Вхідними даними для програми обчислення виразу є масив лексем, побудований сканером. Арифметичний вираз може мати довільний вигляд в межах записаних вище синтаксичних правил. Тому доцільно підготувати список лексем в формі, зручній для синтаксичного розбору. Кожну лексему можна подати у вигляді (код,значення), де код – числове позначення лексеми кожного виду.

З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил.

Ліворекурсивні правила - це такі правила граматики, які містять виклик самих себе без просування по формулі вперед. Ліворекурсивним правилом можна задати наприклад добування кореня парного степеня, якщо задати задати рекурсію коли для добування корення вищого степення викликається видобування кореня меншого ступеня аж до кореня 1-го степеня.

В цілому ряді програм потрібно, щоб граматика розглянутого мови не містила лівої рекурсії. Наявність леворекурсівних правил у вихідній граматиці не фатальна, так як для будь-якої граматики існує еквівалентна граматика без лівої рекурсії. Процес перетворення граматики із леворекурсивної до звичайної називають **лівою фактаризацією.**

Граматичні правила, які будуть задовольняти цю умову та дозволять використовувати метод рекурсивного спуску можуть бути наступними:

```
arith_expr ::= term ( ( "+" | "-" ) term ) *
term ::= factor ( ( "*" | "/" ) factor ) *
factor ::= number | "(" arith_expr ")"
number ::= cipher cipher *
cipher ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

64. Алгоритм взаємодії функцій розпізнавання елементів виразів для перетворення формул ЕТ з інфіксної форми у постфіксну.

Реалізацію розпізнавання виразу реалізуємо методом нисхідного граматичного розбору за алгоритмом рекурсивного спуску. З теорії формальних мов і граматик відомо, що для реалізації методу рекурсивного спуску потрібно переписати граматичні правила у вигляді, де немає ліворекурсивних правил. При цьому використовують символи дужок "{", "}", "(", ")", "[", "]" як метасимволи, тобто для зображення самої формули. Якщо ж будь-яка дужка є елементом самої формули, то дужку записують в апострофах, наприклад '('.Значення дужок як метасимволів є таким. Фігурні дужки {*} означають, що заключена в них конструкція може повторюватись нуль або більше разів. Квадратні дужки [*] означають, що заключена в них конструкція може бути або відсутня, або присутня лише один раз, тобто альтернатива присутності. Круглі дужки (*) вживаються тоді, коли потрібно зробити вибір однієї з альтернатив, записаних в круглих дужках.

ію, яка повинна розпізнати праву частину граматичного правила. При цьому функція може викликати в потрібні моменти інші функції для розпізнавання частини конструкції. Наприклад, функція для <вираз> буде викликати функцію для <доданок>, функція для <доданок> буде викликати функцію

для <множник>, і т.д. Врешті за такою схемою можливі непрямі рекурсії, наприклад, <вираз><доданок><множник><вираз>... .

65. Алгоритм фіксування помилок у формулах і друкування діагностики в процесі перетворення до постфіксної форми.

Це копіпаст з лекції Черкаша

В процесі перетворення в постфіксну форму можуть бути виявлені різні помилки, пов'язані з неправильним записом самого виразу, наприклад: між двома іменами комірок немає знака операції; відсутня закриваюча кругла дужка у виразі; у тексті формули знак операції записаний двічі підряд '++'; і інші. У випадку знаходження помилки процес перетворення припиняється і повідомляється зміст помилки та її імовірна позиція.

```
class TConvItoP  // клас перетворювача
{
protected:
Ttoken * toklist; // вказівник на масив лексем в інфіксній формі
int size;  // кількість лексем в інфіксному масиві
int ff;       // номер поточної лексеми в інфіксному масиві
Ttoken * token; // вказівник на поточну лексему в інфіксному масиві
Ttoken postfix[maxsizelist];  // масив лексем в постфіксній формі
int N;       // номер останньої лексеми в постфіксному масиві (0<=N<100)
Ttoken * errcontext; // лексема, імовірно до якої стосується помилка
bool error; // ознака наявності помилки при перетворенні
int codeerror;  // номер помилки за класифікацією перетворювача
// допоміжні функції для побудови постфіксної форми
void GetNextTok() // перейти до наступної лексеми інфіксної форми
{
      if(ff < size -1) { ff++; token= &toklist[ff]; }
      else { token= &NulTok; } // список лексем вичерпаний
} // void GetNextTok()
      void SaveTok(Ttoken * t) // записати лексему t в постфіксний масив
{
      N++; postfix[N]= *t; // копія лексеми t
} // void SaveTok(Ttoken * t)
. . . . . .
void Formula(); // функція для <формула>
void Vyraz();   // функція для <вираз>
void Dodanok(); // функція для <доданок>
void Mnoznyk(); // функція для <множник>
void ETFunction(); // функція для <функція>
void ETRjad();  // функція для <ряд>
public:
. . . . . .
// визначення методів класу: стартова функція для перетворення
int TConvItoP::Converting(Ttoken * t, int c, Ttoken * badtoken, Ttoken *current)
{ // перетворити в постфіксну форму
// t - масив лексем в інфіксній формі; c - кількість лексем
// badtoken - може означати лексему, до якої стосується помилка
// current - лексема на момент виявлення помилки
toklist=t;  size=c; // запам'ятати параметри
ff =-1; //найперша лексема в інфіксному масиві
N=-1;  // порожній масив лексем постфіксної форми
```

```
error=false; // ознака наявності помилки
errcontext = &NulTok; // імовірна позиція помилки
// побудова дерева синтаксичного розбору з реагуванням на помилки
try
{
// всі функції викликаються при першій лексемі, яка до них належить
GetNextTok(); // вибрати найпершу лексему
Formula();  // запуск дерева розбору
}
catch (int ne) // номер (код) виявленої помилки
{
error=true; codeerror=ne;
}
if(!error) re
turn 0; // помилок не було
else
{ // у випадку помилки передається лексема з помилкою
//і повертається номер помилки
*badtoken = *errcontext; // лексема, до якої може стосуватися помилка
*current = *token; // лексема на момент виявлення помилки
return codeerror; // номер знайденої помилки
}
} // int TConvItoP::Converting(Ttoken * t, int c, Ttoken & badtoken, Ttoken *current)
. . . . . .
void TConvItoP::Formula() // функція для <формула>
{
if(*token != s_equal)
{ errcontext = token;
throw 2001; // немає знака = на початку формули
}
GetNextTok(); // знак = не зберігаємо
Vyraz(); // перехід до аналізу виразу
// чи всі лексеми були проаналізовані?
if(*token != end_list)
{ errcontext=token; // позиція помилки
throw 2002; // відсутній або неправильний знак операції
}
} // void TConvItoP::Formula()
void TConvItoP::Vyraz() // функція для <вираз>
{
Ttoken * opr; // вказівник на лексему зі знаком
Dodanok();  // найперший доданок виразу
while(*token == s_plus || *token == s_minus)
{
opr = token; // запам'ятати лексему зі знаком
GetNextTok(); // вибрати наступну лексему після знака
Dodanok();  // наступний доданок виразу
SaveTok(opr); // записати в постфіксну форму знак
}
} // void TConvItoP::Vyraz()
```

```
. . . . . .
void TConvItoP::Mnoznyk() // функція для <множник>
{
if(*token == number_et || *token == cell_et) // це число або комірка
{ SaveTok(token); // записати в постфіксну форму число або ко
мірку
GetNextTok(); // вибрати наступну лексему після числа або комірки
}
else if
(*token >= fsum && *token <= fint) // це функція
{ ETFunction(); // перейти до аналізу функції
}
else if
(*token == openbracket) // це вираз в дужках
{
GetNextTok
(); // наступна лексема після відкриваючої дужки
Vyraz(); // аналіз виразу в дужках
if(*token!=closebracket)
{ errcontext = token;
throw 2003; // немає закриваючої дужки для виразу
}
GetNextTok(); // наступна лексема після закриваючої дужки
}
else
{ errcontext = token;
throw 2004; // неправильний запис елемента виразу
}
} // void TConvItoP::Mnoznyk()
```

66. Інтерпретація формул ЕТ на основі постфіксної форми зображення. Структура основного алгоритму інтерпретації.

67. Алгоритм застосування стеку і правила опрацювання постфіксної форми виразів в процесі інтерпретації.

68. Алгоритм опрацювання зв'язаних формул в процесі інтерпретації (залежна-впливаюча).

69. Організація алгоритму інтерпретації пряморекурсивної формули. Необхідні допоміжні параметри для коректної інтерпретації та їх використання.