Частина 2. Алгоритмізація і програмування

Тема 9. Основи алгоритмізації і програмування
Тема 10. Інтегроване середовище Borland Pascal
Тема 11. Базові елементи мови Pascal
Тема 12. Базові оператори Pascal
Тема 13. Структуризація програм. Процедури і функції
Тема 14. Масиви Pascal
Тема 15. Обробка символьної інформації
Тема 16. Множини Pascal
Тема 17. Записи Pascal
Тема 18. Файли Pascal
Тема 19. Модулі Pascal
Тема 20. Графічне програмування у Pascal

Тема 9. Основи алгоритмізації і програмування

План

1. Етапи розробки програми
2. Основи алгоритмізації
3. Контрольні запитання

Дана тема є оглядом основ алгоритмізації і програмування. Алгоритмізація є першим питанням даного розділу і першим питання вирішення задачі будь-якого рівня складності. Відомості, які будуть наведені тут потрібні для вирішення усіх лабораторних робіт і курсової роботи. Ми розглянемо терміни і етапи, які складають основу розробки програм, з'ясуємо властивості та способи представлення алгоритмів, познайомимось із основами структурного програмування.

1. Етапи розробки програми

Перш за все визначимо основні терміни, з точки зору яких ми будемо розглядати розробку програмних систем.

Задача - процес, який може бути формалізований і вирішений на комп'ютері. Кожна задача, у випадку її складності може бути розбита на менш складні задачі.

Алгоритм - послідовність правил і операцій для вирішення певної задачі.

Програма - послідовність інструкцій, написаних мовою програмування, яка реалізує алгоритм вирішення задачі. Програма може бути простою (реалізованою в одному модулі) або складною. В останньому випадку кажуть про **програмну систему**, яка складається з

кількох **модулів**, що ієрархічно зв'язані між собою інформаційними зв'язками.

У процесі створення будь-якої програми, чи то лабораторна робота, яку треба виконати і продемонструвати, чи великий проект, у виконанні якого беруть участь десятки або сотні програмістів, можна виділити кілька етапів. Витрати праці і часу на їхнє виконання різняться, різняться витрати і для різних програм, деякі етапи можна "пропустити", однак аналіз процесу розробки призводить до висновку про те, що майже завжди, приходиться виконувати такі **етапи розробки програми**:

- Постановка задачі;
- Аналіз, формалізований опис задачі, вибір моделі;
- Вибір і розробка алгоритму вирішення задачі;
- Проектування загальної структури програми;
- Кодування;
- Налагоджування і верифікація програми;
- Отримання результату, його інтерпретація і, можливо, наступна модифікація моделі;
- Публікування або передача замовнику результату роботи;
- Супровід програми.

Розглянемо докладніше зміст кожного з етапів.

Постановка задачі здійснюється замовником, яким може бути зовнішня організація, організація, у якій працює програміст, сам програміст, викладач. На цьому етапі задача, яку необхідно вирішити шляхом створення програми, формулюється природною мовою. При цьому така постановка може бути простою, як у завданнях до лабораторних робіт, більш складною, як у завданні до учбової курсової роботи, або високої складності, як це відбувається у реальних комерційних програмних системах.
Важливо уявляти, чи є вирішення програми за допомогою комп'ютера найкращим способом отримання результату. Адже при вирішенні обчислювальної задачі може статися, що рівняння, що описують математичну модель об'єкту або процесу, можуть бути точно вирішені аналітичним способом. У такому випадку комп'ютер не потрібен.
Якщо ставиться задача розробки складної програмної системи слід також оцінити раціональність такої розробки. При цьому слід звернути увагу на такі питання:

- Чи є вже на ринку готові програмні системи, які реалізують поставлену задачу?
- Якщо такі системи є, чи не краще буде купити готові, ніж витрачати гроші і час на розробку нової системи?
- Якщо готові системи є, але вигідніше розробляти аналогічні самим, то за якими параметрами ваш продукт буде кращим за аналогічні і наскільки він буде дорожчим (дешевшим) ніж у конкурентів?
- Якщо готових систем немає, чи є реальним досягнення мети в принципі і для вашої організації зокрема?
- Чи має ваша система перспективу? Чи не вирішує вона завдання, які були актуальними у минулому, а сьогодні не є цікавими?

Аналіз задачі включає визначення вихідних даних і результатів, виявлення можливих обмежень і звичайно завершується формалізованим описом задачі, яке найчастіше припускає її математичне формулювання. Якщо мова йде про моделювання певних об'єктів або процесів, на цьому етапі розробляється математична модель об'єкту (процесу). У такому випадку визначаються фактори, які відіграють суттєву роль і відкидаються другорядні.

Вибір або розробка алгоритму або числового методу вирішення задачі мають непересічне значення для успішної роботи над програмою. Ретельно опрацьований алгоритм - запорука ефективної роботи по вирішенню поставленої задачі. Основи створення алгоритмів будуть

розглядатись у наступному питанні.

Проектування загальної структури програми. На цьому етапі відбувається опрацювання проекту, під час якої визначаються складові елементи програмної системи, ієрархічна підпорядкованість і взаємозв'язок окремих модулів, визначаються способи збереження інформації.

Кодування - процес запису програми мовою програмування. На цьому етапі відповідно до правил конкретної мови програмування відбувається переведення алгоритму у програмний код. Якщо алгоритм є ретельно проробленим, то кількість помилок при кодуванні суттєво знижується, а ефективність цього процесу підвищується.

Налагоджування і верифікація програми являють собою дуже важливу частину процесу розробки програми. Налагоджування полягає в усуненні помилок кодування, а верифікація - у перевірці алгоритму і подоланні його помилок. Верифікація - доведення того, що програма працює коректно і видає правильні результати.
Для цього розробляється і проводиться тестування, яке може полягати у спеціально підібраних прикладах, рішення яких або відомі заздалегідь, або можуть бути отримані точно за допомогою інших пристроїв. Якщо результат такого тестування співпадає з очікуваним, є можливість вважати, що програма працює коректно. Це зовсім не означає, що ваша програма при усіх можливих варіантах вихідних даних буде видавати вірний результат. Насправді програму можна вважати готовою, якщо розробник зміг довести собі і замовнику, що результат роботи програми і є рішенням поставленої задачі.

Отримання результату, його інтерпретація і, можливо, наступна модифікація моделі. На цьому етапі слід ретельно проаналізувати результат роботи програми. Для об'єкту або процесу, який моделювався, слід порівняти результати розрахунків і результати спостережень. Якщо результати не співпадають, слід змінювати саму модель, робити її більш точною і реалістичною.

Публікування або передача замовнику результату роботи. Закінчену і налагоджену програму слід передати замовнику. Якщо дві сторони - замовник і виконавець - дійшли згоди в оцінці системи, робота вважається виконаною. У такому випадку, у комерційному проекті складаються відповідні акти і реалізується оплата, в учбовому проекті викладач виставляє студентові оцінку. Певні результати розробки і експлуатації програмних систем можуть публікуватись у періодичних паперових або електронних виданнях.

Супровід системи полягає у консультаціях замовника по роботі із програмою, виправлення виявлених в процесі експлуатації недоліків або помилок, навчання користувачів роботі з програмою. Для великих комерційних проектів цей етап має дуже важливе значення.

2. Основи алгоритмізації

Алгоритм - правила перетворення інформації, що містять усі необхідні вказівки про послідовність і характер дій по обробці даних для забезпечення отримання необхідного результату.

Такі вказівки мають вигляд однозначних правил виконання перетворень при усіх варіантах вихідних даних. Алгоритм складається з найпростіших дій, серед яких можна виділити:

- операції введення-виведення;
- операції перевірки умови і передачі управління;
- процедури обчислень;

- операції початку-кінця перетворювань.

Алгоритм повинен мати такі *фундаментальні властивості*:

- **Однозначність** - правила повинні бути визначені точно і однозначно, виключаючи довільне трактування;
- **Масовість** - означає можливість застосування алгоритму для вирішення певного кола задач при початкових даних, які змінюються у визначених межах;
- **Результативність** алгоритму означає можливість отримати кінцевий результат за кінцеву кількість кроків і у придатний час;

З урахуванням властивостей алгоритму можна дати ще й таке визначення алгоритму.

Алгоритм - кінцева послідовність правил, які однозначно визначають процес перетворювання вихідних та проміжних даних у кінцевий результат вирішення певного кола задач.

Розрізняють *два способи запису алгоритму*:

- **Описовий** (словами). Є достатньо компактним і зручним для подальшого кодування програми але важко сприйнятливим при високій складності алгоритму.
- **Графічний** (у вигляді блок-схем). Є більш наочною, хоча й менш компактною формою подання алгоритму. Для позначення дій використовуються спеціальні графічні фігури, вигляд і призначення яких регламентуються стандартами (ГОСТ 19002-80 і ГОСТ 19003-80). Деякі знаки блок схем наведені у таблиці 9.1.

Таблиця 9.1. Зображення і функціональний зміст деяких символів блок-схем

Символ	Найменування	Функція
	Пуск / Зупинка	Початок (кінець) програми, модулю, процедури або функції. Такому символу відповідають оператори `begin..end` мови Pascal.
	Процес	Виконання операції або групи послідовних лінійних операцій перетворювання даних. Такому символу відповідає оператор присвоювання мови Pascal.
	Введення-виведення	Загальний оператор введення-виведення (без вказівки джерела або приймача інформації). Такому символу відповідають оператори `write, writeln, read, readln` мови Pascal.
	Рішення	Перевірка певної умови і прийняття рішення про напрямок продовження програми в залежності від результату перевірки. Такому символу відповідають оператори `if, case` мови Pascal. Використовується також як елемент структури циклу з використанням операторів

		`repeat,while`.
	Підпрограма	Виконання (виклик) раніше створених або окремо сформованих алгоритмів. Таким символом позначаються процедури.
	Виведення на принтер	Різновид оператора виведення, який означає виведення результатів на принтер (отримання твердої копії результату).
	Модифікація	Означає циклічне змінювання певної змінної (параметру циклу у визначених межах. Такий символ використовується як елемент структури циклу з використанням оператора `for`.
	Введення-виведення на магнітний диск	Різновид оператора введення-виведення із зчитуванням-записуванням інформації з/на магнітний диск. Використовується у операторах `write, writeln, read, readln`, коли перший параметр є файловою змінною.
	Виведення на монітор	Різновид оператора виведення, який означає виведення на екран монітору.
	З'єднувач	Поєднує інформаційні зв'язки у межах однієї сторінки. Вихідний і вхідний символи, які слід з'єднувати, повинні мати однакові цифри.
	Міжсторінковий поєднувач	Поєднує інформаційні зв'язки на різних сторінках. Використовується у складних, багатосторінкових алгоритмах. У символі вказується номер блоку і номер сторінки з якого приходить інформація або на який уходить.

Створення алгоритму є творчим процесом і тут не існують жорсткі рекомендації, дотримуючись яких можна створити алгоритм (у такому випадку алгоритми вже давно складав би комп'ютер). Одну і ту саму задачу можна вирішувати різними способами. Звичайно треба намагатися обирати спосіб, який забезпечить найефективнішу роботу як алгоритму, так і програми. У випадку, коли поставлена задача є складною, її треба розбити на кілька простіших, незалежних підзадач.

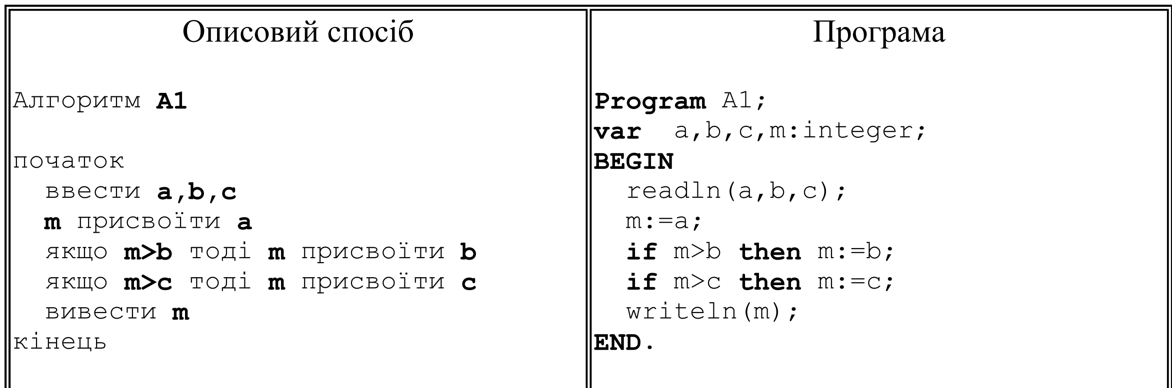
Розглянемо приклад створення двох алгоритмів знаходження найменшого з трьох чисел. Алгоритми будуть представлені у описовому виді, у вигляді блок-схеми і, для підкреслення збіжностей між описовим способом і програмою, написаною мовою Pascal, буде наведена відповідна програма. У загальному випадку алгоритм складається з трьох частин: спочатку треба ввести з клавіатури (присвоїти) значення трьох змінних (констант); на другому етапі

слід знайти найменше значення із цих трьох; на третьому етапі слід вивести результат на екран. Перший і третій етапи не представляють якихось труднощів. Основа цих алгоритмів - у другому етапі.

Суть першого алгоритму полягає у наступному: спочатку ми вважаємо, що перше число є найменшим і присвоюємо його значення змінній m. Далі ми перевіряємо, чи є друге число меншим за m. У випадку вірної відповіді ми наново присвоюємо змінній m значення вже другого числа. І, наприкінці, перевіряємо, чи є третє число меншим за m. У випадку вірної відповіді ми наново присвоюємо змінній m значення третього числа. Таким чином після трьох підкроків ми у змінній m отримаємо значення найменшого з трьох чисел.

Суть другого алгоритму полягає у тому, що ми одразу порівнюємо між собою два перших числа. Те число, яке є більшим, у подальшому порівнянні участі не бере, а те яке виявилося меншим - порівнюється із третім числом. Фактично у цьому алгоритмі ми за два кроки визначаємо, яке із чисел є найменшим і його значення присвоюємо змінній m. Зазначимо, що існують і інші алгоритми знаходження найменшого із трьох чисел.

Перший алгоритм



Описовий спосіб	Програма

Алгоритм **A1**
 	
початок
  ввести **a**,**b**,**c**
  m присвоїти **a**
  якщо **m>b** тоді **m** присвоїти **b**
  якщо **m>c** тоді **m** присвоїти **c**
  вивести **m**
кінець
	Program A1;
	var a,b,c,m:integer;
	BEGIN
	  readln(a,b,c);
	  m:=a;
	  if m>b **then** m:=b;
	  if m>c **then** m:=c;
	  writeln(m);
	END.

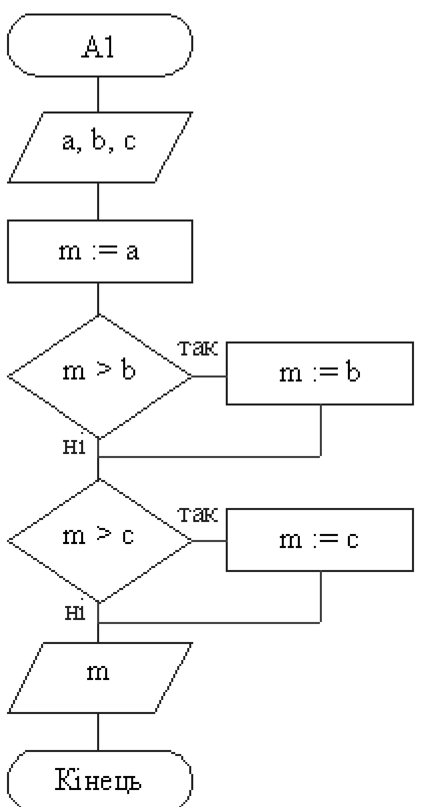


Рис. 9.1. Блок-схема першого алгоритму

Другий алгоритм

Описовий спосіб	Програма

Алгоритм **A2**
 	
початок
  ввести **a,b,c**
  якщо **a<b** тоді, якщо **a<c** тоді **m** присвоїти **a**
                          інакше **m** присвоїти **c**
           інакше, якщо **b<c** тоді **m** присвоїти **b**
                          інакше **m** присвоїти **c**
  вивести **m**
кінець
	Program A2;
	var a,b,c,m:integer;
	BEGIN
	  readln(a,b,c);
	  if a<b **then** if a<c **th**
	                      el
	        else if b<c **th**
	                      el
	  writeln(m);
	END.

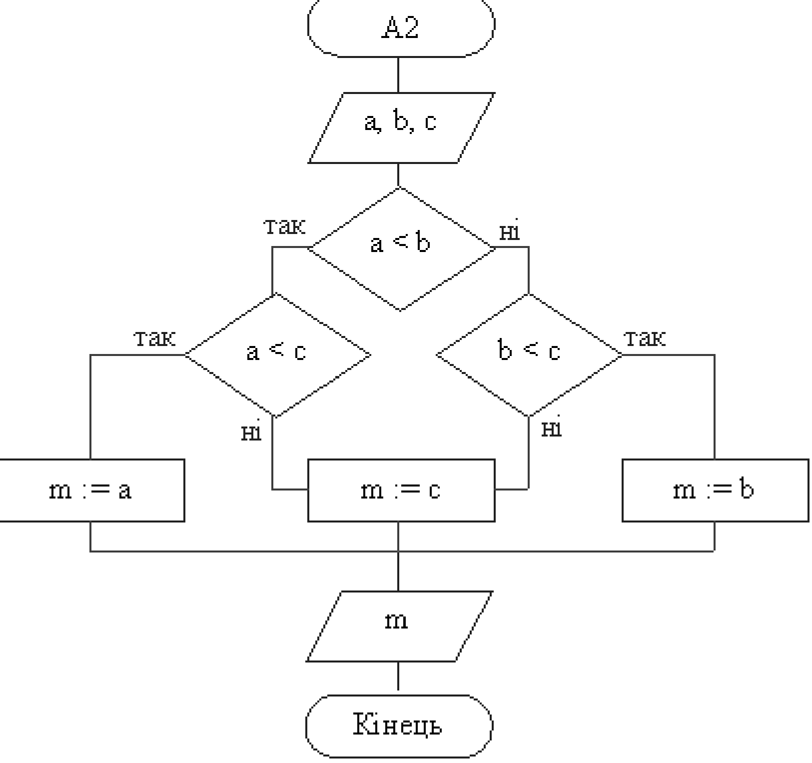


Рис. 9.2. Блок-схема другого алгоритму

Відзначимо, що розглядаючи питання програмування ми будемо обов'язково торкатися питань алгоритмізації. Розбираючи приклади, звертайте увагу на те, у якій послідовності і якими діями вирішуються ті чи інші задачі.

У своїх задачах і вправах ми будемо дотримуватись **принципів структурного програмування**. Такі принципи базуються на використанні у програмах трьох структур:

- лінійної;
- розгалуженої;
- циклічної.

Лінійний алгоритм має один інформаційний потік і може бути зображений такою схемою.

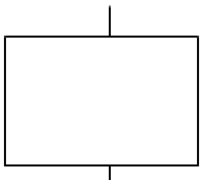


Рис. 9.3. Лінійний алгоритм

Розгалужений алгоритм передбачає вибір одного з кількох можливих варіантів. При використанні розгалуженого алгоритму інформаційний потік розгалужується на два потоки, але самі потоки не повертаються назад. Розгалужений алгоритм може бути представлений такими схемами.

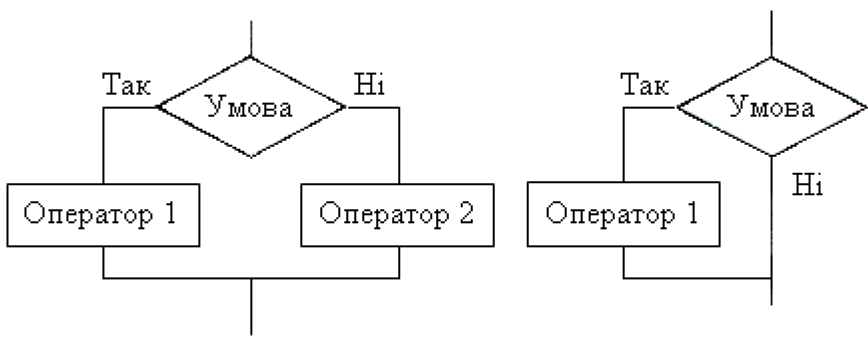


Рис. 9.4. Розгалужені алгоритми

Циклічні алгоритми передбачають багаторазове виконання визначеної послідовності дій над вихідними або поточними даними. Інформаційний потік у циклі розгалужується на два потоки, один з яких входить у тіло циклу, а другий обходить його. Зверніть увагу на те, що інформаційний потік тіла циклу повертається назад.
За способом організації виходу із циклу циклічні алгоритми поділяються на:

- цикли з відомою кількістю повторювань (див. рис. 9.5, а);
- ітераційні цикли з умовою завершення циклу (див. рис. 9.5, б);
- ітераційні цикли з умовою продовження циклу (див. рис. 9.5, в);

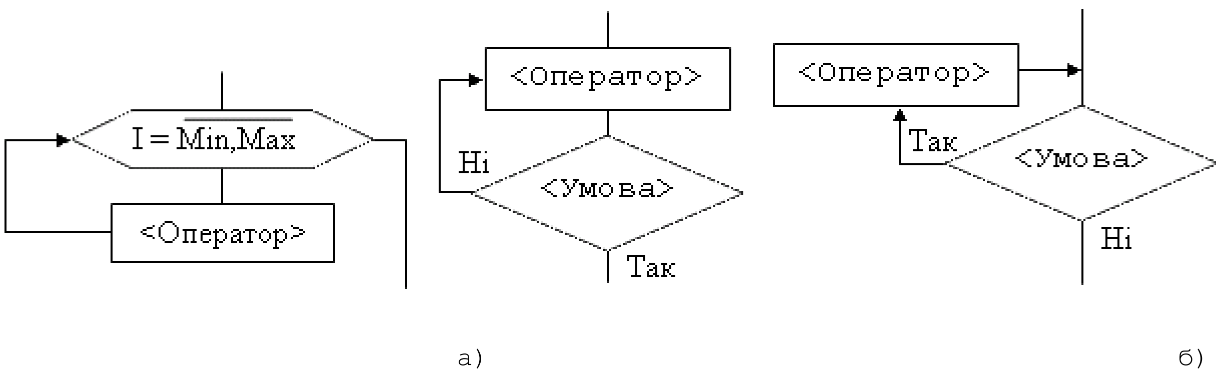


Рис. 9.5. Циклічні алгоритми

Загальний алгоритм програми складається із будь-якої кількості будь-яких з цих шести основних структур шляхом з'єднання виходу попередньої структури із входом наступної. Більш складні алгоритми утворюють шляхом розбиття задачі на підзадачі, кожна з яких може бути представленою своїм власним алгоритмом (підпрограмою).

3. Контрольні запитання

1. Охарактеризуйте поняття задача, алгоритм, програма.
2. Назвіть основні етапи розробки програми.
3. Охарактеризуйте зміст етапу постановки задачі.
4. У чому полягає суть аналізу задачі?
5. У чому полягає суть проектування загальної структури програми?
6. Що таке кодування?
7. У чому полягає суть налагодження і верифікації програми?
8. Як відбувається аналіз результатів програми і їхня інтерпретація?
9. Що таке супровід програмної системи?
10. Що таке алгоритм? Що є його змістом?
11. Які властивості повинен мати алгоритм?
12. Які основні способи запису алгоритму ви знаєте?
13. Які символи для графічного зображення алгоритму ви знаєте?
14. У чому полягають принципи структурного програмування? Які типи структур ви знаєте?
15. Розкажіть про розгалужені структури.
16. Розкажіть про циклічні структури.

Тема 10. Інтегроване середовище Borland Pascal

План

1. Інтерфейс Borland Pascal і команди роботи з файлами
2. Редагування програм
3. Компіляція і виконання програм
4. Налагодження програм
5. Використання довідкової служби і прикладів
6. Контрольні запитання

Ця тема є вступом до програмування у одному з найрозповсюдженіших і найпопулярніших середовищ розробки програмного забезпечення - Borland Pascal. Ми розглянемо інтерфейс системи, дізнаємось про те, яки виконувати основні операції з файлами, познайомимось із прийомами редагування програм. Ця тема містить відомості про суть процесу компіляції та виконання програм, а також про те, як ці дії виконуються у Borland Pascal. Будуть наведені рекомендації щодо налагодження програм і використання довідкової служби. Суттєву допомогу складуть додатки, у яких перелічені основні клавіші та комбінації клавіш, коди помилок з поясненнями їхньої суті та рекомендації щодо їх усунення.

1. Інтерфейс Borland Pascal і команди роботи з файлами

Розглянемо роботу з версією 7.0 інтегрованого середовища Borland Pascal. Будемо вважати, що таке програмне забезпечення вже встановлене на вашому ПК. Для того, щоб запустити інтегроване середовище ви повинні викликати один із файлів:

- **turbo.exe** - Turbo Pascal 7.0 - інтегроване середовище, яке працює у реальному режимі MS-DOS і генерує програми виключно для реального режиму роботи центрального процесора. Нагадаємо, що режими роботи розглядалися у темі 4;
- **bp.exe** - Borland Pascal 7.0 - інтегроване середовище, яке працює у захищеному режимі

MS-DOS і здатне генерувати програми для реального і захищеного режимів MS-DOS, а також програми для Windows. За своїм виглядом інтегровані середовища Turbo Pascal 7.0 і Borland Pascal 7.0 різняться лише у частині компіляції програм;
- **bpw.exe** - Borland Pascal for Windows 7.0 - інтегроване середовище, яке підтримує інтерфейс і роботу у Windows і здатне генерувати програми для реального і захищеного режимів MS-DOS, а також програми для Windows. Зазначимо, що таке інтегроване середовище робить більш зручною роботу у Windows, зокрема не виникають проблеми зі шрифтами, але містить обмежений набір можливостей при налагоджуванні програм.

Ми будемо орієнтуватися на компілятор Turbo Pascal, як на такий, що забезпечує створення програм на будь-якому ПК.

1.1. Інтерфейс системи

При запуску файла **turbo.exe**, або ярлика, який посилається на нього, перед вами з'явиться вікно, зображене на рис. 10.1.

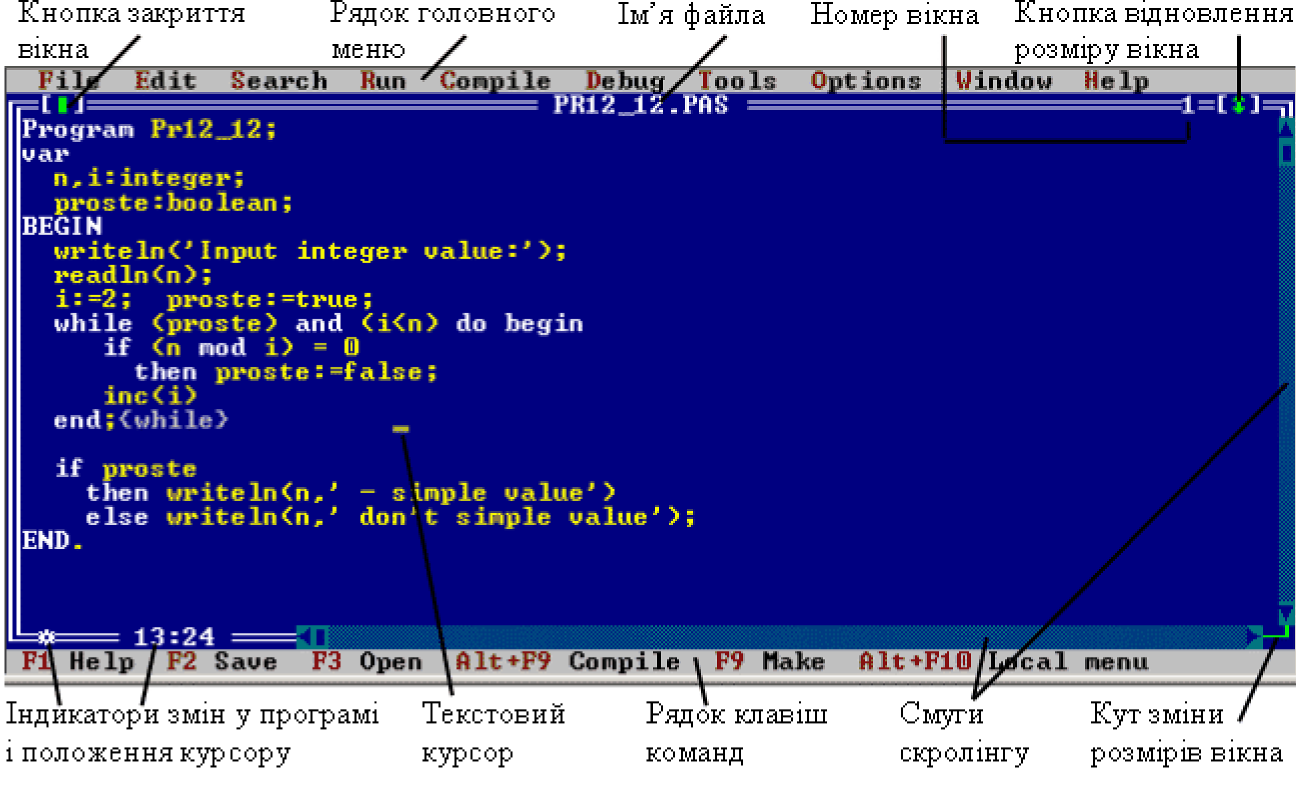


Рис. 10.1. Робочий екран інтегрованого середовища Turbo Pascal

Рядок головного меню надає доступ до усіх можливостей інтегрованого середовища Turbo Pascal. Він може бути активним, (активізація відбувається натисканням клавіші F10) і тоді ви маєте можливість виконати будь-яку команду головного меню, або неактивним, наприклад, під час редагування тексту програми. Виконувати команди меню можна також і мишею. Існує спосіб швидкого виконання команд головного меню - комбінація гарячих клавіш. Для цього слід утримуючи натиснутою клавішу `Alt` натиснути символ, який виділений у назві кожного з пунктів меню.
Кожному з пунктів головного меню підпорядкована група команд, з якими зв'язані підпункти меню, причому існує кілька різновидів команд:

- **Виклик вікна діалогу** відбувається, якщо після тексту команди слідує три крапки "...", наприклад "`File / Open ...`";

- **Виклик вкладеного меню** відбувається, якщо після тексту команди вказано стрілку "➔", наприклад, "Options / Environment ➔";
- **Виклик команди** - в усіх інших випадках.

Посередині рамки вікна буде відображатись **ім'я файла** у якому зберігається ваша програма. У випадку, якщо файл зберігається не у поточному каталозі - буде виводитись і шлях до вашого файла відносно поточного каталогу Turbo Pascal. Зазначимо, що рекомендується працювати таким чином, щоб ваш каталог був поточним - так компілятор буде швидше відшукувати файли, які зв'язані з вашою програмою і так само легше буде відкривати ваші нові файли.
Зазначимо також, що **категорично не рекомендується** відкривати у різних вікнах один і той самий файл (у такому випадку після імен файлів через двокрапку буде ставитись номер версії відкритого файла, наприклад "Lab_01.pas:1" і "Lab_01.pas:2"). Це є небезпечна ситуація, оскільки існує можливість закрити останнім файл з невиправленою версією програми.

Інтегроване середовище Turbo Pascal дозволяє працювати з кількома різними програмами (модулями) одночасно, але редагування відбувається лише у вікні однієї (активної) програми. Кожне вікно має свій номер. Переключити (робити активними) вікна можна натисканням клавіші Alt разом із номером відповідного вікна.

Основна робота при створенні програми відбувається у вікні, де **текстовий курсор** вказує позицію введення тексту програми. Дізнатися про місце знаходження курсору (номер рядка і номер стовпчика) можна за допомогою **індикатора положення курсору** в нижній лівій частині вікна. Там саме **індикатор змін** відображає, чи були внесені зміни до активного документу. Якщо текст програми не вміщується у вікні. то за допомогою **смуг скролінгу** можна переглянути будь-яку частину коду програми.

Під час роботи ви маєте можливість змінювати розміри вікна, збільшувати активне вікно на весь екран, відновлювати його попередній розмір, закривати його. Відповідні елементи управління зображені на рис. 10.1.

Для завершення роботи з інтегрованим середовищем Turbo Pascal слід виконати команду головного меню "**File / Exit**" або натиснути комбінацію клавіш Alt+X.

1.2. Основні команди роботи із вікнами (файлами)

Створення нових програм

Як правило під час першого виклику інтегрованого середовища Turbo Pascal автоматично створюється нове вікно з ім'ям "NONAME00.PAS", у якому ви одразу можете починати писати текст програми. У випадку, якщо працюючи над однією програмою ви вирішили створити вікно для нової програми - слід виконати команду головного меню "**File / New** ".
Рекомендується одразу після початку роботи над програмою зберегти її на жорсткому диску вашого ПК. Зберігати програми тільки на дискетах **не рекомендується** .

У додатку А після контрольних запитань наведені основні клавіші і комбінації клавіш, які діють в інтегрованому середовищі Turbo Pascal.

Збереження програм

Для того, щоб записати програму на диск, слід виконати команду головного меню "**File / Save ...**", або ж натиснути функціональну клавішу **F2** (рекомендується це робити через кожні 10-15 хвилин роботи). При першому виконанні такої команди (коли ім'я файла вказане як `NONAME00.PAS`" перед вами з'явитися вікно діалогу (див. рис. 10.2).
При збереженні програми, яка вже має своє власне ім'я і місце на диску - запис відбувається без появи вікна діалогу.
У тому випадку, коли ви бажаєте створити копію програми, слід скористатися командою головного меню "**File / Save as ...**". Відмітимо, що при роботі у цьому і інших вікнах діалогу без миші перехід по позиціях відбувається натисканням клавіш `Tab` - у прямому порядку і комбінації клавіш `Shift+Tab` - у зворотному порядку.

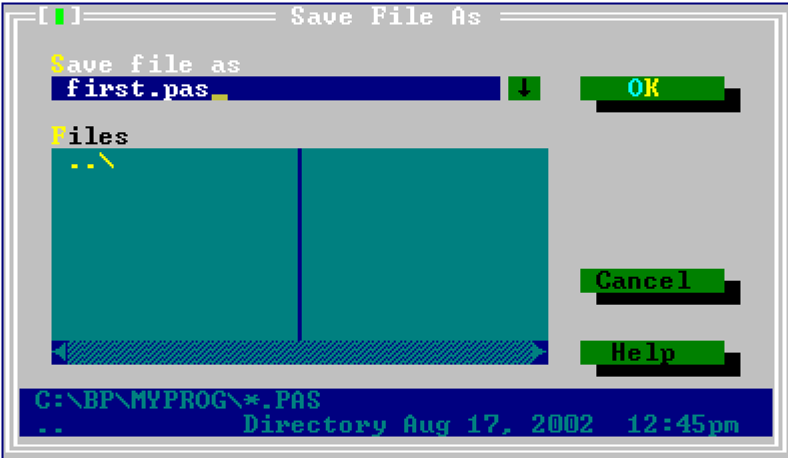


Рис. 10.2. Вікно діалогу збереження файла

Для коректного збереження файла ви повинні зробити наступне:

- У рядку "`Save file as ...`" ввести ім'я файла. Для сумісності із будь-якою операційною системою рекомендується дотримуватись правил запису імен файлів для операційної системи MS-DOS (див. тему 7);
- У інформаційному рядку вікна проконтролювати і запам'ятати місце збереження файла. У нашому прикладі це "`C:\BP\MYPROG`". У випадку, якщо вас не влаштовує місце, яке

пропонується, його слід змінити, вказавши це за допомогою зони "Files";

- Переконавшись у правильності введеної інформації натиснути кнопку "Ok" або клавішу Enter.

Примітка. Пам'ятайте, що команда "Save" це запис змісту вікна Turbo Pascal на диск. Не намагайтеся вибрати ім'я файла, який вже існує (імена таких файлів будуть висвітлюватися у зоні "Files") ім'ям нового файла. Після попередження англійською мовою про заміну старого файла (див. рис. 10.3) і натискання клавіші Enter, цей старий файл буде знищено, а під його ім'ям буде записано новий файл.

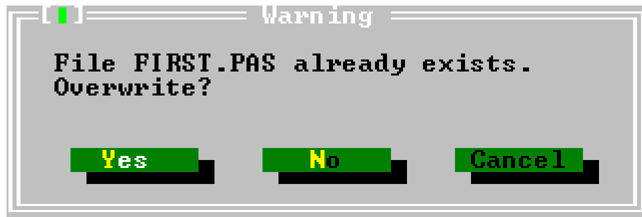


Рис. 10.3. Попередження про заміну файла

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
[■]                    NONAME00.PAS                    1=[↕]
Program First;
BEGIN
  Write('Hello!');
END.

  4:1
F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Відкриття файла

Для того, щоб відкрити файл (зчитати раніше створений файл з диска) слід скористатись командою "**File / Open ...**" або натиснути функціональну клавішу **F3**. Перед вами відкриється вікно діалогу (див. рис. 10.4), яке зовні є схожим на вікно збереження файла.

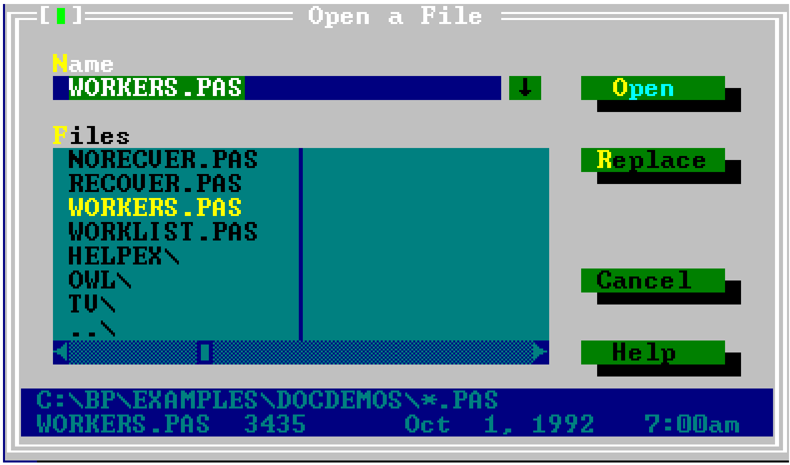


Рис. 10.4. Вікно відкриття файла

Коректне відкриття файла повинно виконуватись у такій послідовності:

- У зоні "`Files`" слід вибрати потрібну папку і побачити у цій зоні ім'я потрібного файла. Зазначимо, що папки у цій зоні мають після імені символ "`\`", а перехід на один рівень наверх відбувається через папку "`..\`". Контролювати повний шлях до файла можна в інформаційному рядку цього вікна;
- Вибрати у зоні "`Files`" потрібний файл, при цьому його ім'я повинно з'явитись у рядку "`Name`";
- Натиснути кнопку "`Open`" або клавішу `Enter`.

У випадку, коли необхідно перейти на інший логічний диск, слід у рядку "`Name`" ввести ім'я логічного диска, і вже після цього вибирати потрібну папку у зоні "`Files`".
Зручною є можливість вибору файла із переліку тих файлів, які вже відкривались, викликати такий список можна натисканням стрілки униз праворуч рядка "`Name`" або на клавіатурі. Вигляд списку наведений на рис. 10.5.

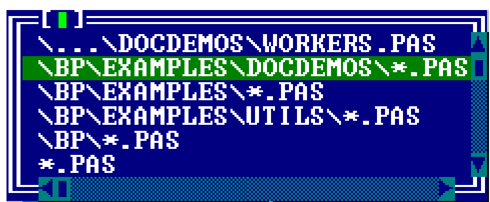
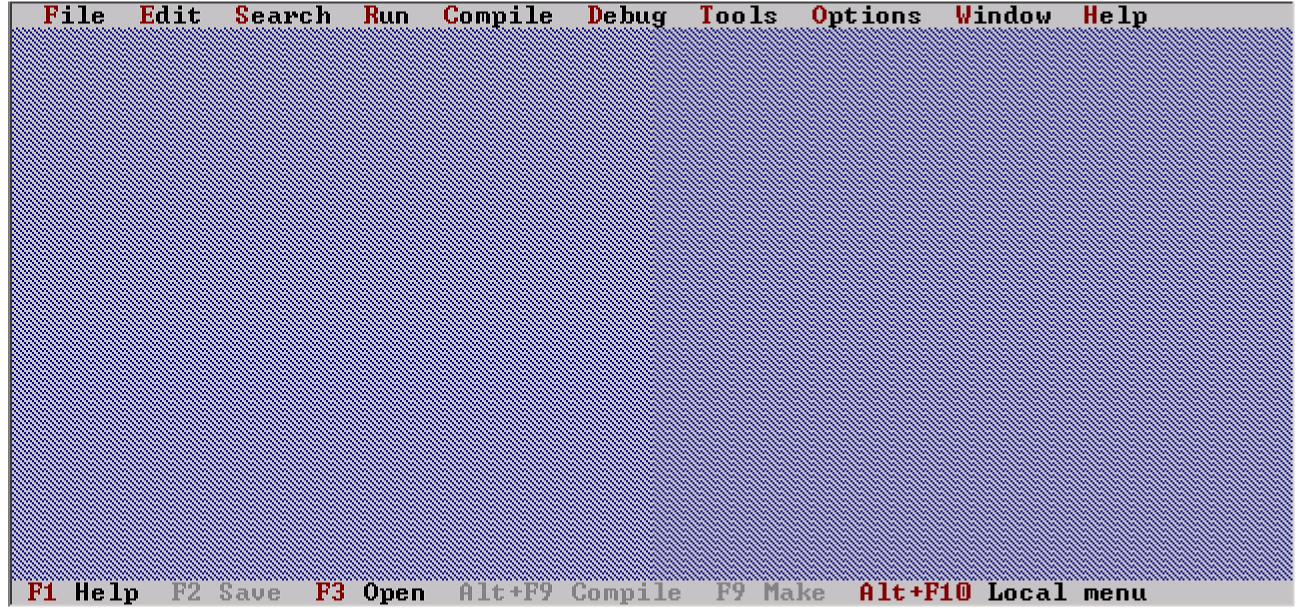


Рис. 10.5. Вибір імені файла із списку



Закриття файла

Після остаточного або тимчасового завершення роботи з файлом його слід закрити. Зробити це можна натисканням **кнопки закриття вікна** або комбінацією клавіш `Alt+F3`. Автоматичне закриття файла відбувається також при завершенні роботи із Turbo Pascal. Якщо перед закриттям файла його зміст не був збережений, на екрані з'явиться попередження (див. рис. 10.6). Відповідні кнопки вікна призведуть до таких дій:

- "`Yes`" - збереження файла на диску і закриття вікна;
- "`No`" - закриття вікна без збереження файла;
- "`Cancel`" - відміна команди закриття вікна. Файл записаний не буде і ви повернетесь у режим редагування файла.

Рис. 10.6. Інформація при завершення роботи з файлом, зміни у якому не були зафіксовані

Настроїти конфігурацію інтегрованого середовища Turbo Pascal можна через команди пункту "`Options`" головного меню (див. рис. 10.7).

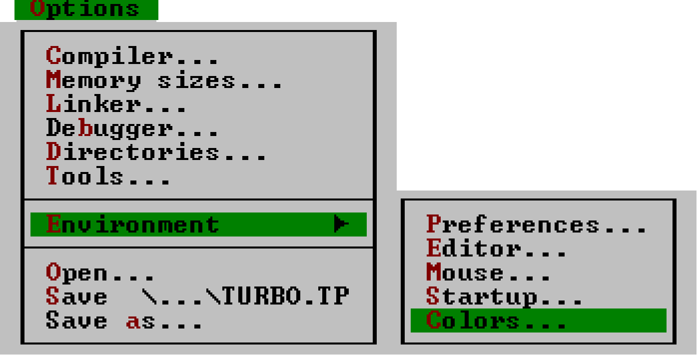


Рис. 10.7. Команди меню "`Options`"

2. Редагування програм

2.1. Загальні прийоми набору і редагування тексту

Набір і редагування тексту програми відбувається засобами вбудованого багатовіконного редактора. При набиранні програм слід дотримуватись загальноприйнятого стилю написання програм мовою Pascal. Такий стиль передбачає використання відступів у 1-2 позиції для позначення підпорядкованості певних частин програми і написання програми рядковими літерами з виділенням початку слів у ідентифікаторах великими літерами. Сам стиль не впливає на процес виконання програми, але значно покращує сприйняття програми самим автором та іншими програмістами.

У додатку А наведені основні клавіші і комбінації клавіш, що діють в інтегрованому середовищі Turbo Pascal.

При набиранні програми текстовий курсор переміщується управо до тих пір, поки програміст не натисне клавішу `Enter`. Наступний рядок розпочнеться з тим саме відступом, що й попередній, полегшуючи тим самим формування підпорядкованої структури програми. У випадку, якщо вам потрібно підвищити пріоритет рядка, тобто змістити його уліво, - слід натиснути клавішу `BackSpace`.
Щоб вставити декілька пустих рядків слід кілька разів натиснути клавішу `Enter`. Для того, щоб знищити пусті рядки слід або підвести текстовий курсор у кінець рядка після якого слід знищити рядки і натискати клавішу `Delete`, або підвести текстовий курсор у початок рядка перед яким слід знищити рядки і натискати клавішу `BackSpace`.

Відмінити дії, які виконував програміст під час набирання і редагування тексту можна командою "**Edit / Undo**" (комбінація клавіш `Alt+BackSpase`). Таким способом можна повернутись на декілька десятків кроків назад. Для того щоб повторити відмінені дії можна скористатись командою "**Edit / Redo**".

2.2. Копіювання і переміщення фрагментів

При формуванні тексту програми доволі часто виникає потреба скопіювати або перемістити фрагмент коду програми з одного місця у інше. Зазначимо, що такі маніпуляції можуть відбуватись як у межах одного вікна так і у різних вікнах. Turbo Pascal надає можливість виконувати такі маніпуляції і ними слід активно користуватись.
У будь-якому випадку спочатку слід виділити фрагмент і вже потім виконувати певні маніпуляції із ним.
Для виділення фрагменту зручно утримуючи натиснутою клавішу `Shift` за допомогою клавіш управління курсором визначити фрагмент (сам фрагмент буде виділено сірим фоном).
Існує ще один прийом виділення: ви підводите курсор на початок фрагменту і натискаєте `Ctrl+K`, потім - `B`, далі підводите курсор у кінець фрагменту і натискаєте `Ctrl+K`, потім `K`.
У межах одного вікна виділений фрагмент можна:

- **Скопіювати** командою `Ctrl+K, C`;
- **Перемістити** командою `Ctrl+K, V`;
- **Видалити** командою `Ctrl+K, Y`;

З одного вікна у інше виділений фрагмент можна:

- **Скопіювати**. Послідовність дій: команда меню "**Edit / Copy**" (клавіші `Ctrl+Ins`), потім у новому вікні - "**Edit / Paste**" (Клавіші `Shift+Ins`);
- **Перемістити**. Послідовність дій: команда меню "**Edit / Cut**" (клавіші `Shift+Del`), потім у новому вікні - "**Edit / Paste**" (Клавіші `Shift+Ins`);
- **Видалити**. Команда "**Edit / Clear**" (клавіші `Ctrl+Del`).

Відмітимо, що після чергового розміщення фрагменту в буфер (команди "Edit / Copy" або "Edit / Cut" або відповідні комбінації клавіш) вставляти фрагмент можна багаторазово аж до розміщення у буфері іншого фрагменту.

2.3. Пошук і заміна тексту

Зручні можливості надає Turbo Pascal для пошуку і заміни тексту. Використання цього прийому стає актуальним у великих програмах. Команди пошуку і заміни тексту зосереджені у пункті "**Search**" головного меню (див. рис. 10.8).

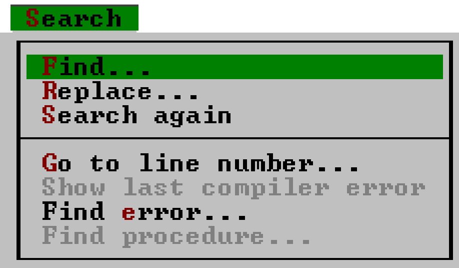


Рис. 10.8. Команди меню "Search"

Команда "**Find ...**" активізує вікно діалогу (див. рис. 10.9) і дозволяє знайти потрібний текст у програмі. У рядку "**Text to find**" слід набрати тест або його частину, яку ви бажаєте відшукати.

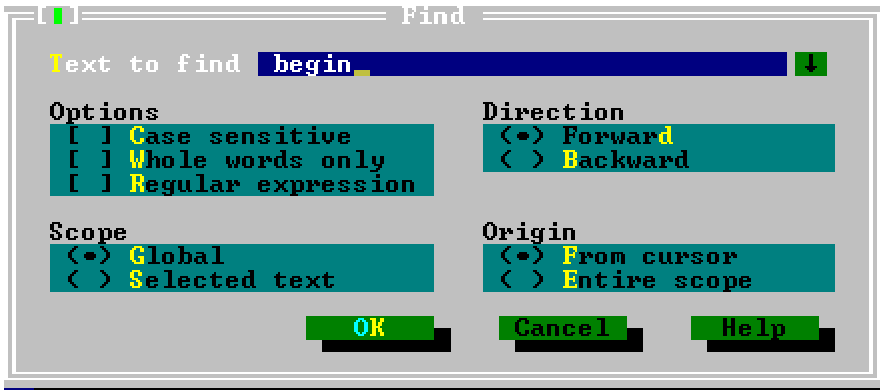


Рис. 10.9. Вікно діалогу "Find..."

Параметри вікна діалогу означають наступне:

- **Case sensitive** - шукати з урахуванням регістру. При активній опції слова `"Begin"` і `"begin"` будуть вважатись різними, при відключеній - однаковими;
- **Whole word only** - тільки ціле слово. При активній опції пошук буде відбуватись для умови повного збігу тексту, при відключеній - вказаний текст може бути частиною більшого слова;
- **Regular expression**. Активізована опція дозволяє відшукувати і спеціальні символи у тексті, формуючи таким чином шаблони слів;

- **Global**. Пошук буде проводитись по всьому файлу;
- **Selected text**. Пошук буде проводитись тільки у попередньо виділеному фрагменту;
- **Forward**. Пошук буде проводитись від початку до кінця;
- **Backward**. Пошук буде проводитись від кінця до початку;
- **From cursor**. Пошук буде проводитись від позиції курсору;
- **Entire scope**. Пошук буде проводитись від початку (кінця) програми.

```
  File   Edit   Search   Run   Compile   Debug   Tools   Options   Window   Help
 =[■]================== \BP\EXAMPLES\KURS\KURS.PAS ===================1=[↕]=
program Kurs;
uses Crt,Graph,MyUnit,PVMath,PVServis,PVEdit,PVMenu,PVGr,PVMech;
BEGIN
 repeat
   TextAttr := $07;
   clrscr;
   CursorOff;
   SMenu(KodMain,1,1,6,MenuMain,g,$07,$07,$70,2,0);
   case KodMain of
        1: repeat
             SMenu(KodProces,1,4,2,MenuProces,v,$07,$07,$70,1,0);
             case KodProces of
                  1: InputDatas;
                  2: Result;
             end; {case}
           until KodProces=0;
        2: repeat
             SMenu(KodEskiz,18,4,2,MenuEskiz,v,$07,$07,$70,1,0);
             case KodEskiz of
                  1: Shema;
                  2: Mechanizm;
 ========= 1:17 =========
 F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Для пошуку і заміни певного фрагменту програми на інший слід активізувати команду меню "**Search / Replace ...**". Вікно діалогу заміни (див. рис. 10.10) багато в чому нагадує вікно діалогу пошуку.

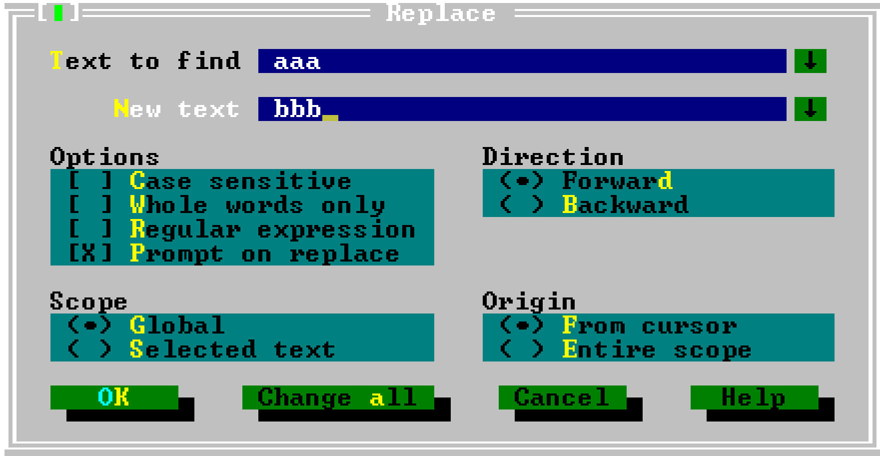


Рис. 10.10. *Вікно діалогу "Replace..."*

Новими можливостями вікна є:

- **New text**. Рядок у якому слід увести текст на який буде замінено текст рядка "`Text to find`";
- **Prompt on replace**. При активізованій опції перед заміною буде видаватись на екран запит на підтвердження заміни;
- **Change all**. Команда, яка дозволяє замінити усі знайдені слова.

Для більш досвідчених користувачів можна порекомендувати використовувати й такі

можливості команди меню "**Search**":

- **Go to line number** - перейти у рядок з певним номером;
- **Show last compiler error** - показати останню помилку, яка виникла під час компіляції програми;
- **Find error** - знайти помилку за відомою адресою її виникнення;
- **Find procedure** - знайти процедуру або функцію під час налагожування програми.

3. Компіляція і виконання програм

Трансляція - процес перетворення програми, написаної мовою програмування високого рівня у програму в машинних кодах. Трансляція може відбуватись як інтерпретація або компіляція.

Інтерпретація - процес трансляції, який відбувається рядок за рядком, причому для кожного рядка спочатку виконується перевірка синтаксису, а потім,- переведення його у машинний код.

Компіляція - процес трансляції, при якому спочатку для усієї програми перевіряється синтаксис, і у випадку відсутності помилок,- відбувається переведення усієї програми у машинний код.

Turbo Pascal має вбудований компілятор - один із найкращих компіляторів мов програмування. Схема роботи компілятора наведена на рис. 10.11.

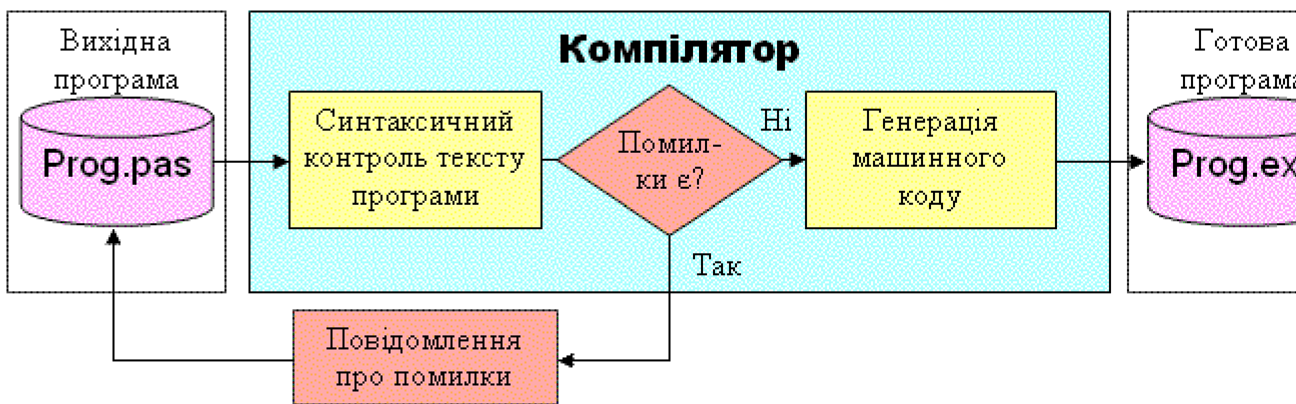


Рис. 10.11. Схема роботи компілятора Turbo Pascal

Компіляція може проводитись однією з команд меню "**Compile**" (див. рис. 10.12):

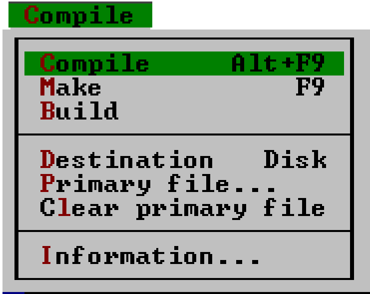


Рис. 10.12. Команди меню "Compile"

- **Compile**. Забезпечує "просту" компіляцію тільки активного документу. Така команда

виконується найчастіше комбінацією клавіш `Alt+F9`.

- **Make**. Забезпечує перевірку, чи проводились зміни у модулях, які використовує активна програма і, якщо такі зміни проводились, то перед компіляцією програми відбувається перекомпіляція модулів. Така команда використовується найчастіше клавішею `F9`.
- **Build**. Забезпечує "повну" компіляцію усіх модулів, які використовує активна програма і після цього - компіляцію самої програми.

Під час компіляції на екрані з'являється вікно (див. рис. 10.13), у якому відображається інформація про етапи цього процесу, зокрема: ім'я головного файлу, напрямок компіляції або режим, загальна кількість рядків програми і номер рядка, який компілюється. У випадку безпомилкової компіляції виводиться повідомлення "`Compile successful: Press any key`" - Компіляція успішна, натисніть будь-яку клавішу.

```
Compiling
Main file: \BP\EXAMPLES\DOS\BGI\ARTY.PAS
Done.

Destination:  Disk      Line number:      0
Free memory:   299K     Total lines:    379

    Compile successful: Press any key
```

Рис. 10.13. Вікно компіляції програми

У випадку, якщо під час компіляції була знайдена помилка, у верхньому рядку вікна програми з'явиться повідомлення про код помилки з коротким поясненням, курсор переміститься у рядок з помилкою і процес компіляції перерветься. На рис. 10.13 наведено приклад помилки, яка виникла у 15-й позиції 33-го рядка і полягає у невірному записі зарезервованого слова **`array`**. Така помилка має код 3 - невідомий ідентифікатор.

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
                              FIRST.PAS                         1
 [■]            \BP\EXAMPLES\DOS\BGI\ARTY.PAS                   2=[↑]
 Error 3: Unknown identifier.
}

uses
  Crt, Graph;

const
   Memory  = 100;
   Windows =   4;

type
  ResolutionPreference = (Lower, Higher);
  ColorList = aray [1..Windows] of integer;

var
  Xmax,
  Ymax,
  ViewXmax,
  ViewYmax : integer;
 *      33:15
 F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Рис. 10.14. Повідомлення про помилку під час компіляції

Коди помилок, повідомлення і деякі пояснення наведені у додатку Б. Отримати коротку довідку про помилку можна натиснувши клавішу **F1**.

Додатковими можливостями, які надає інтегроване середовище Turbo Pascal є:

- **Distantion** - встановлює режим компіляції на диск - опція "**Disk**" або в оперативну пам'ять - опція "**Memory**". При встановленні компіляції на диск, поряд із файлом програми, який має розширення `pas` утворюється файл програми з розширенням `exe` або файл модуля з розширенням `tpu`.
  Для компілятора Borland Pascal замість цієї опції встановлено опцію "**Target**", яка дозволяє визначити режим компіляції **Real** - компіляція у реальному режимі, **Protected** - компіляція у захищеному режимі і **Windows** - компіляція програм під Windows;
- **Primary file** - встановлює файл, який повинен компілюватись перед активною програмою. Вказівка такого файла є корисною при відладці модуля. У такому випадку першим повинен компілюватися файл, який цей модуль використовує;
- **Clear primary file** - відключає режим з попередньою компіляцією неактивного файла
- **Information** - виводить докладну інформацію про відкомпільований файл (див. рис. 10.15).

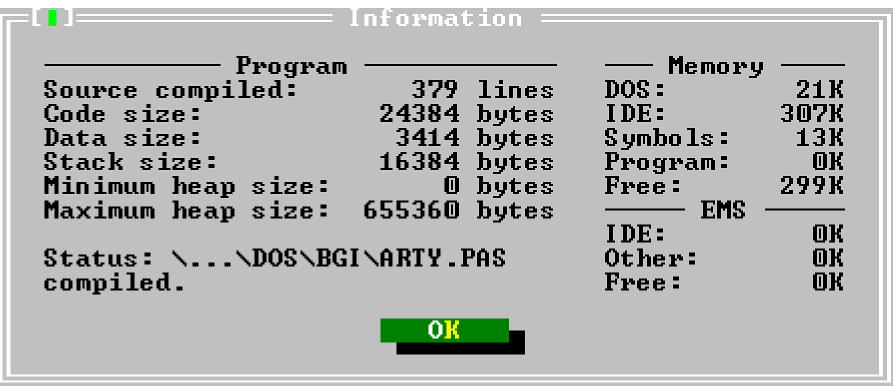


Рис. 10.15. Докладна інформація про відкомпільований файл

У випадку успішної компіляції можна виконати програму. Усі можливості виконання програми, повністю або по кроках зосереджені у меню "**Run**" (див. рис. 10.16). Під час виконання програми оточення інтегрованого середовища Turbo Pascal тимчасово зникає і на екрані відображаються лише ті результати, які передбачені алгоритмом програми. Після завершення роботи програми відбувається автоматична активізація інтегрованого середовища. Переглянути наслідки роботи програми можна натисканням комбінації клавіш **Alt+F5**.

Рис. 10.16. Команди меню "Run"

Команда "**Run**" (виконати) є основною командою, яка дозволяє виконати всю програму в цілому. Найчастіше ця команда активізується комбінацією клавіш `Ctrl+F9`. Якщо програму слід виконати із деякими параметрами, їх можна вказати у вікні діалогу, яке з'явиться при виконанні команди "**Parameters**".
Решта команд меню "**Run**" призначена для виконання програми по кроках і буде розглянута у наступному питанні.

Таким чином типова послідовність дій при створенні програми має вигляд циклу, який повторюється доти, доки не буде отримано програму, що працює коректно. Сама

послідовність дій циклу така:

1. Редагування програми;
2. **`Alt+F9`** - компіляція програми;
3. **`Ctrl+F9`** - виконання програми;
4. **`Alt+F5`** - перегляд результатів.

4. Налагодження програм

Під час створення програми виникають помилки кількох типів.

Помилки першого типу - це синтаксичні помилки, пов'язані із неправильним застосуванням різних елементів мови програмування Pascal. Причиною виникнення таких помилок звичайно є недостатнє знання мови програмування і механічні помилки при набиранні тексту. Усі такі помилки виявляються компілятором і суттєвої небезпеки не несуть. Важливу роль тут відіграє те, що Pascal є мовою із суворим контролем типів. У додатку Б наведені коди синтаксичних помилок і певні рекомендації щодо їх подолання.

Помилки другого типу містять більше неприємностей і пов'язані з періодом виконання програми. Така помилка може виникати під час виконання програми і може залежати від конкретних значень вихідних даних. Повідомлення про таку помилку має вигляд:

```
Run-time error <код помилки> at <адреса>
```

де *код* помилки вказує яка саме помилка виникла (коди помилок періоду виконання програми наведені у додатку В), а *адреса* вказує у якій області оперативної пам'яті виникла помилка. Найчастіше виникає помилка при спробі ділення на нуль або при зацикленні програми. Виникнення помилок такого типу пов'язане із тим, що програма веде себе не так, як припускає програміст. З'ясувати дійсну причину такої помилки можна шляхом налагоджування програми.

Помилки третього типу пов'язані з невірним алгоритмом вирішення задачі.

Для подолання помилок слід використовувати таку можливість інтегрованого середовища Turbo Pascal, як режим виконання програми по кроках разом із налагоджувачем.

Керувати послідовністю виконання програми можна командами меню "**Run**" (див. рис. 10.16):

- **`Step over`** (клавіша **F8**). Виконання програми по кроках без входження у процедури або функції, які використовуються у програмі. Такий режим налагоджування можна рекомендувати, коли ви впевнені у правильності процедур і функцій, або не використовуєте їх.
- **`Trace into`** (клавіша **F7**). Виконання програми по кроках із входженням у процедури або функції, які використовуються у програмі. Такий режим налагоджування рекомендується для перевірки програми разом із усіма блоками. Він є найдовшим, але забезпечує можливість перевірити усе.
- **`Go to cursor`** (клавіша **F4**). Виконати програму до рядка у якому знаходиться курсор. Якщо ви впевнені у тому, що початкова частина програми працює вірно, або вже перевіряли її - ставте курсор у той рядок з якого слід почати налагоджування і викликайте цю команду.

Зазначимо, що ці три способи можна комбінувати між собою, наприклад, першу частину програми виконати до курсору, далі перейти у режим виконання по кроках, заходячи у ті

процедури або функції, у роботі яких ви не впевнені і обминаючи ті, які працюють коректно або є запозиченими.

Якщо програма є доволі великою, можна проставити на ній точки переривання роботи програми (`Breakpoint`). Для того, щоб створити таку точку або додати її до вже існуючих точок, слід скористатись командою меню "**Debug / Add breakpoint ...**". У відповідному вікні діалогу можна визначити умову ("**Conditions**") переривання, кількість проходів ("**Pass count**") після яких програма буде зупинятися, вказується ім'я файла і номер рядка у програмі, де розміщена точка зупинки.

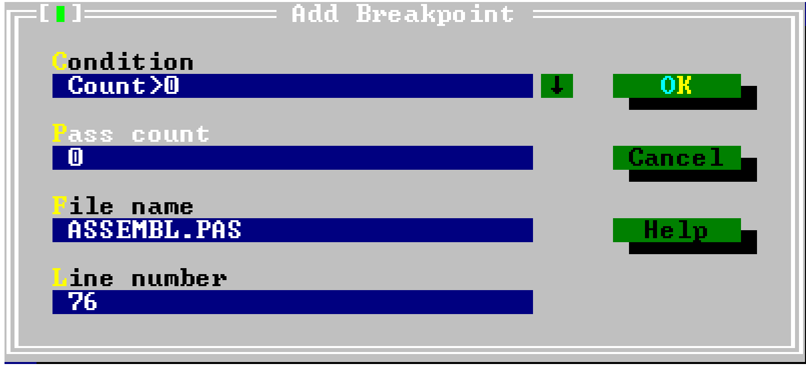


Рис. 10.17. Вікно діалогу "Add breakpoint"

Переглянути інформацію про встановлені точки переривання програми можна командою меню "**Debug / Breakpoints**". Результатом цієї команди буде активізація вікна (див. рис. 10.18). Тут можна відредагувати ("**Edit**") характеристики точки переривання, видалити одну точку ("**Delete**") або усі ("**Clear all**"), переглянути ("**View**") місце у програмі, де розміщена точка зупинки.

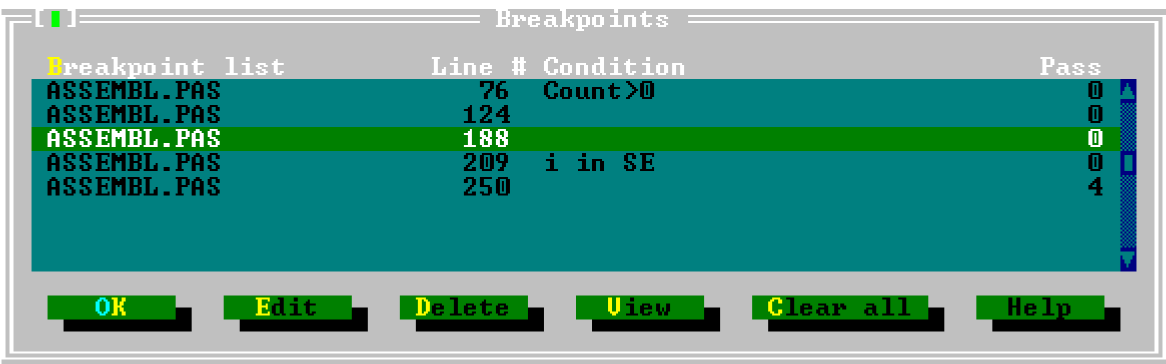


Рис. 10.18. Вікно з інформацією про точки переривання програми

Перервати режим налагоджування програми можна командою меню "**Run / Program reset**" або ж натисканням комбінації клавіш `Ctrl+F2`.

Після виконання чергового кроку або діставшись до чергової точки переривання програми ви можете за допомогою налагоджувача, команди якого активізуються через меню "**Debug**" (див. рис. 10.19), перевірити стан певних елементів вашої програми (констант, змінних тощо).

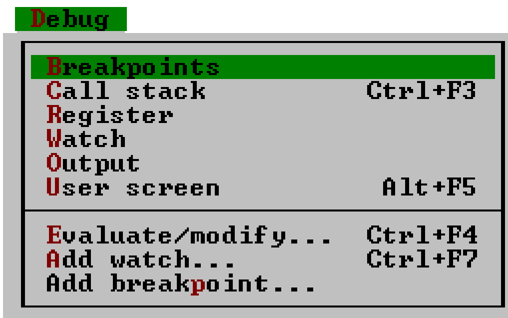


Рис. 10.19. Команди меню "Debug"

Якщо ви відслідковуєте поодинокі значення одного елемента, можна скористатися командою "**Evaluate / modify ...**" (Розрахувати/змінити - комбінація клавіш **Ctrl+F4**), яка дозволяє вивести на екран значення певного елементу програми і, навіть, у разі потреби змінити його поточне значення. Причому, якщо курсор зупинився біля певного елементу програми, то у полі "**Expression**" вікна діалогу (див. рис. 10.20) автоматично з'явиться ім'я цього елемента, у полі "**Result**" його значення, а у полі "**New value**" буде можливість замінити поточний результат на новий. У випадку подальшого натискання кнопки "**Evaluate**" залишиться дійсним розраховане значення, а у випадку натискання кнопки "**Modify**" - буде прийнято нове значення.

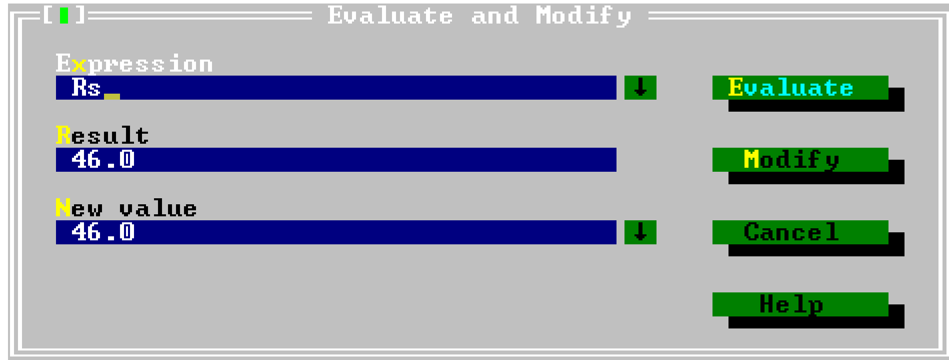


Рис. 10.20. Вікно діалогу "Evaluate and Modify"

У тому випадку, коли вам потрібно відслідковувати значення кількох елементів програми, зручно відкрити спеціальне вікно перегляду значень. Активізується таке вікно командою "**Add watch ...**" (додати до вікна перегляду - комбінація клавіш `Ctrl+F7`). Таке вікно (див. рис. 10.21) займає як правило кілька рядків і розміщується у нижній частині екрану. Активізувати таке вікно можна також командою меню "**Debug / Watch**". Коли вікно є активним до нього можна додавати елементи програми (клавіша `Ins`), вилучати непотрібні елементи (клавіша `Delete`) або редагувати певний елемент - для цього слід виділити рядок з цим елементом і натиснути клавішу `Enter`.

Рис. 10.21. Вікно перегляду значень елементів програми

Під час налагоджування процедур і функцій, особливо рекурсивних, зручним буде

використання вікна перегляду змісту стеку (див. рис. 10.22), яке активізується командою "**Debug / Call stack**". Нагадаємо, що через стек передаються параметри у процедури і функції.

```
[■]════════════════ Call stack ════════════════3=[↑]
FindDirect((...),[1..12],(...),0,[])
AnalysSyntes

F1 Help  ◄┘ Go to source  F7 Trace  F8 Step  F10 Menu
```

Рис. 10.22. Вікно перегляду стеку

Під час програмування на рівні команд процесору потрібно знати стан регістрів центрального процесора. Зробити це можна активізувавши вікно "CPU" (див. рис. 10.23) командою "**Debug / Register**".

```
[■]═ CPU ═══4
AX 0101 DX 0001
CX 0100 BX 0007
IP 0155 CS 5536
SI 3D0E DS 57E5
DI 0001 ES 57E5
SP 3CCE SS 58E2
BP 3EE2
c=0 z=0 s=0 o=0
p=0 i=1 a=0 d=0
```

Рис. 10.23. Вікно перегляду стану регістрів центрального процесора

Якщо потрібно одночасно бачити перед собою вікно програми і вікно коду програми - можна активізувати вікно виведення (див. рис. 10.24) командою "**Debug / Output**".

```
 File  Edit  Search  Run  Compile  Debug  Tools  Options  Window  Help
[■]═══════════════════ UNASS.PAS ═══════════════════2=[↑]
    if (PtrPart(N) and (not ZapretMontag(CurPart,SB1))
      then begin
        dec(PtrPart);
        AddPart(PtrPart,NaprAss,Napr,NaprP,SB1);
        DeletePartFromMatrix(CurPart);  end
      else begin
        dec(PtrPart);
        NaprAss := NaprAss + ' ) => ( ';
      end;
end; {Procedure AddPart;}


BEGIN
END.
      669:18
──────────────────────── Output ────────────────────────5
 Матрица сборки
 X+      0   0   0   0   0   0   0   0   0   0   0   0
 X-      3   0   0   0   0   1   0  12   0   6   7   8
 Y+      3   9   0   2   0   0   0   0  11   0   0   0
 Y-      3   9   0   0   4   0   5   0   0   0   0   0
F1 Help  F2 Save  F3 Open  Alt+F9 Compile  F9 Make  Alt+F10 Local menu
```

Рис. 10.24. Одночасний перегляд вікна коду програми і вікна виведення результатів програми

5. Використання довідкової служби і прикладів

Під час написання програми доволі часто виникає необхідність отримати довідку з певного питання. Можна скористатись підручником або довідником з програмування, який не завжди

буває поруч, а можна - вбудованою довідковою службою Turbo Pascal. І хоча за умови володіння англійською мовою ефективність використання такого сервісу суттєво збільшується, використовувати допомогу слід у будь-якому випадку, адже там зосереджена велика кількість прикладів фрагментів програм.
Довідкова система Turbo Pascal є контекстною, тобто інформація, яка виводиться, залежить від того, де у момент її виклику розміщувався курсор. При натисканні клавіші `F1` виводиться інформація, яка відноситься до редагування файла або ж до активного пункту меню. Якщо ви при компіляції отримали повідомлення про синтаксичну помилку - натискайте клавішу `F1` і отримаєте коротку допомогу з поясненнями про суть помилки. Так на рис. 10.25 показано вікно довідкової служби, викликане при активній команді "**Compile**" головного меню.

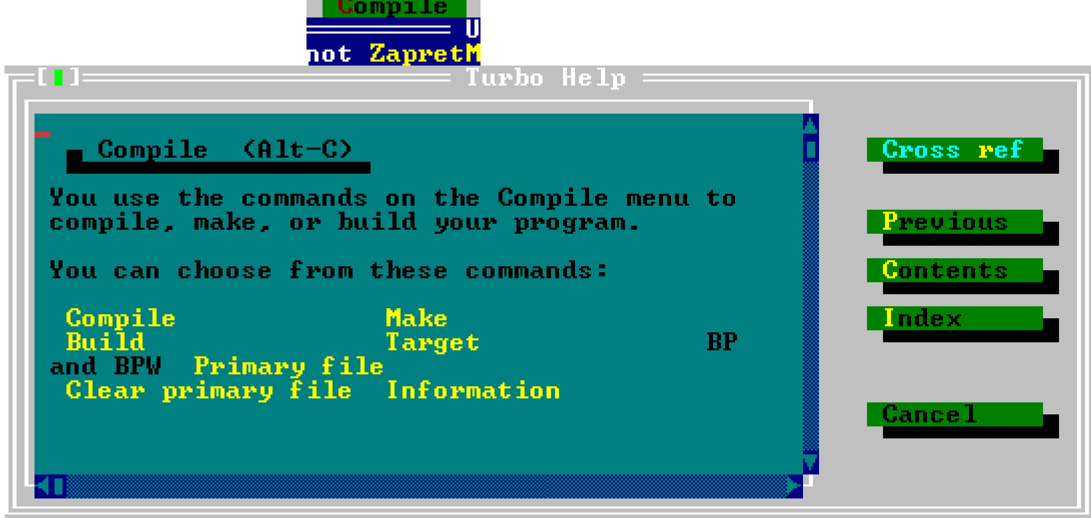


Рис. 10.25. Довідка по команді меню "Compile"

Зверніть увагу на те, що деякі слова виділені жовтим (світлим) кольором. Ці слова є набором команд меню "Compile". Якщо ви захочете отримати довідку по цих командах - підведіть курсор до потрібної команді і натисніть кнопку "**Cross ref**" або клавішу `Enter`.
Практично усі вікна допомоги мають такі кнопки:

- **Previous** - повернення до попереднього вікна довідки;
- **Contents** - виклик змісту допомоги (див. рис. 10.26).
  Активізується також із середовища Turbo Pascal командою меню "**Help / Contents**";
- **Index** - виклик індексної допомоги (див. рис. 10.27).
  Активізується також із середовища Turbo Pascal командою меню "**Help / Index**".

Зміст усіх розділів допомоги можна викликати командою меню "**Help / Contents**". Тут (див. рис. 10.26) уся довідкова інформація розбита на категорії, Обираючи потрібну і уточнюючи, про що саме ви хочете дізнатись, можна отримати допомогу з будь-якого питання використання інтегрованого середовища Turbo Pascal і мови програмування.

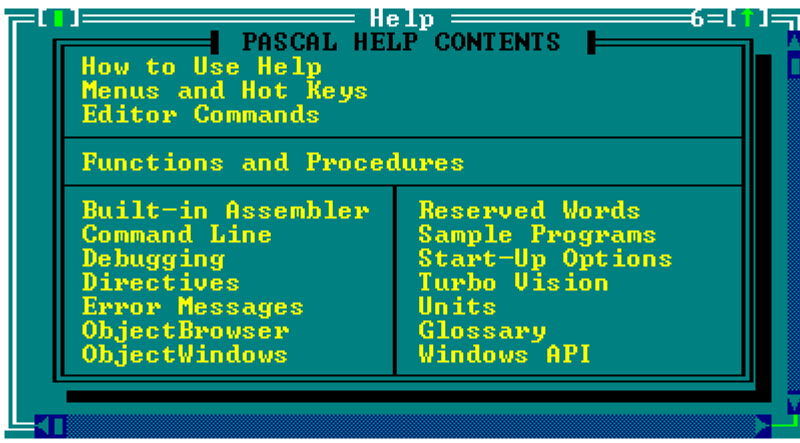


Рис. 10.26. Зміст довідкової служби

Зручним є використання індексної допомоги, яка викликається командою меню "**Help / Index**" або комбінацією клавіш `Shift+F1` і у якій (див. рис. 10.27) усі розділи довідкової служби розміщені у алфавітному порядку. Причому, дістатись до потрібного пункту можна як використанням смуг скролінгу, так і набираючи послідовно перші літери того пункту, по якому вам потрібна довідка. Останній прийом дозволяє дуже швидко знайти потрібну допомогу.

```
Help
Blue                                          Built-in-Assembler
Boolean                                       ButtonCount
Boolean-operators                             ButtonState,TCluster
Borland-Pascal-language-element               BYTE
BottomText                                      TURBO
Break                                           TURBO
breakpoint,conditions

C

C-(Character-format-specifier)                ColorGroup
CalcBounds,TView                              ColorIndexes
CalcWidth,TTerminal                           ColorItem
calling-TEMC                                  ColorTable(v.3.0)
CanInsert,TTerminal                           Command,TButton
CanMoveFocus,TProgram                         Command-line
```

Рис. 10.27. Індексна допомога

У тому випадку, коли вам потрібно отримати конкретну допомогу по одному з операторів мови програмування Pascal, зручно скористатись контекстною довідкою мови програмування. Для цього слід підвести курсор до того оператора у програмі, який вас цікавить і викликати команду меню "**Help / Topic Search**" або натиснути комбінацію клавіш **Ctrl+F1**. На рис. 10.28 наведений результат таких дій у тому випадку, коли курсор знаходився біля оператора `repeat`.

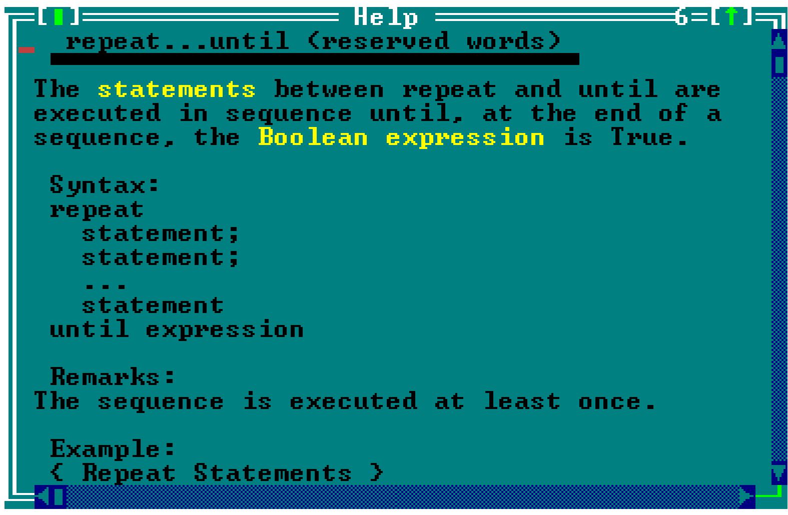


Рис. 10.28. Виклик контекстної допомоги по оператору `repeat`

Зверніть також особливу увагу на те, що у багатьох пунктах довідки є **приклади** (**Example**) використання того чи іншого елементу мови програмування Pascal. Ви маєте змогу скопіювати такий приклад і вставити його у вікно своєї програми. **Звертайте увагу** на такі приклади, дивіться, як формуються стандартні конструкції мови, використовуйте їх у своїх програмах.

Перелік комбінацій клавіш, які використовуються для виклику довідкової служби наведений у додатку А

6. Контрольні запитання

1. Охарактеризуйте інтерфейс інтегрованого середовища Turbo Pascal.
2. Як створити нове програмне вікно і записати його зміст у файл?
3. Як відкрити файл з програмою?
4. Як коректно закрити вікно і завершити роботу з інтегрованим середовищем Turbo Pascal?
5. Охарактеризуйте основні прийоми набору і редагування тексту.
6. Продемонструйте копіювання і переміщення фрагментів.
7. Як відбувається пошук і заміна тексту?
8. Охарактеризуйте поняття трансляція, інтерпретація, компіляція.
9. Як відкомпілювати програму в інтегрованому середовищі Turbo Pascal?
10. Як слід реагувати на повідомлення про помилки, які виникають під час компіляції?
11. Як додаткові можливості при компіляції програм у Turbo Pascal ви знаєте?
12. Як виконати програму з інтегрованого середовища Turbo Pascal?
13. Які типи помилок можуть виникати при написанні програм?
14. Які способи виконання програм по кроках реалізовані в інтегрованому середовищі Turbo Pascal?
15. Як визначити значення певного елементу програми під час її виконання?
16. Як реалізувати багаторазове відстеження значень кількох елементів під час виконання програми?
17. Як переглянути стан стеку і регістрів центрального процесора?
18. Як організувати одночасний перегляд вікна з текстом програми і вікна результатів?
19. Розкажіть про можливості довідкової служби Turbo Pascal.

Тема 11. Базові елементи мови Pascal

План

1. Алфавіт і словник мови
2. Типізація даних
3. Вирази, операнди, операції
4. Структура Pascal-програми
5. Введення-виведення даних
6. Контрольні запитання

У цій темі ми з вами познайомимось із основними елементами, з яких складаються Pascal-програми. Тут не буде жодного прикладу закінченої програми, але тут будуть розглядатися дуже важливі питання, без яких не можна написати навіть найпростішу програму.
Ми познайомимось із алфавітом мови, дізнаємось для чого призначені окремі символи та їхні групи. Зустрівшись із ідентифікаторами ви дізнаєтесь про те, як правильно записувати їхні імена.
Тут вперше ви дізнаєтесь про те, якими типами даних може оперувати Pascal-програма, що є спільного у цих типах і чим вони різняться, дізнаєтесь про те, як створювати нові типи даних.
Одним із важливих елементів програмування є вміння грамотно створювати з операндів і операцій прості і складні вирази. Ви познайомитесь із арифметичними і логічними операціями, а також із операціями відношення.
Ви дізнаєтесь про те, що Pascal-програма має певну структуру, якої слід обов'язково дотримуватись. Ви навчитесь підключати модулі до своєї програми, описувати мітки, типи, константи, змінні.
Наприкінці ви дізнаєтесь про те, як вводити значення з клавіатури і виводити результати на екран у певному форматі.
Давши вірні відповіді на контрольні запитання ви зможете переконатися у тому, що засвоїли пройдений матеріал.

1. Алфавіт і словник мови

При записуванні алгоритму вирішення задачі мовою програмування Pascal слід знати правила запису і використання елементарних інформаційних і мовних одиниць.

Програма мовою Pascal формується за допомогою кінцевого набору знаків, які утворюють алфавіт мови.

Алфавіт мови Pascal складається з:

- букв;
- десяткових і шістнадцяткових цифр;
- спеціальних символів.

У якості **букв** використовуються великі й малі літери латинського алфавіту, а також символ підкреслювання:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

У якості десяткових і шістнадцяткових **цифр** використовуються:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

При написанні програм також використовуються **спеціальні символи**:

Символ	*Значення*	*Символ*	*Значення*
+	плюс	.	крапка
-	мінус	,	кома
*	зірочка	:	двокрапка
/	риска дробу	;	крапка з комою
=	дорівнює	'	апостроф
>	більше	#	номер
<	менше	@	комерційне а
[ ]	квадратні дужки	^	тильда
( )	круглі дужки	$	символ долару
{ }	фігурні дужки		пробіл (без позначення)

Комбінації спеціальних символів можуть утворювати **складені символи**:

Складений символ	*Значення*
:=	присвоювання
<>	не дорівнює
..	діапазон значень
(* *)	аналогічно до { } (коментар)
<=	не більше
>=	не менше
(. .)	аналогічно до [ ] (індекс масиву)

Зверніть увагу на те, що не можна ставити пробіли всередині складаних символів.

Неподільні послідовності знаків алфавіту утворюють **слова**, які відділяються одне від одного розділювачами і мають певний сенс у програмі. **Розділювачем** можуть бути пробіл, кінець рядка, коментар. Слова поділяються на:

- **Зарезервовані слова**, які є складовою частиною мови програмування, мають фіксований напис і певний сенс. Наприклад, `program, const, begin, end` тощо;
- **Стандартні ідентифікатори**, які слугують визначенню міток, типів даних, констант, змінних, процедур і функцій. На відміну від зарезервованих слів, значення стандартних ідентифікаторів можна перевизначити. Наприклад, можна перевизначити дію стандартної функції `sin( )`. Перевизначення значень стандартних ідентифікаторів найчастіше призводить до помилок, тому цього слід уникати.
- **Ідентифікатори користувача**, які застосовуються для визначення міток, констант, змінних, процедур і функції, що визначаються самим користувачем. Вдало підібране ім'я ідентифікатора користувача зменшує кількість помилок у програмі.

Загальні **правила і рекомендації** щодо запису ідентифікаторів користувача:

- Ідентифікатор повинен починатися із букви;
- Ідентифікатор може складатися із букв і цифр. Використання спеціальних символів, у тому числі й пробілу, в ідентифікаторах є неприпустимим;
- Між двома ідентифікаторами повинен бути принаймні один розділювач;

- Максимальна довжина ідентифікатора - 127 символів, але значущими є лише перші 63 символи;
- Рекомендується, щоб імена ідентифікаторів мали сенс і, у випадку, коли воно складається з кількох слів, бажано першу літеру слова робити великою (без пробілів!!!). Наприклад, `FirstProgram, OldDay` тощо.

2. Типізація даних

2.1. Загальні відомості

Кожен елемент даних повинен відноситися до одного з типів, який є припустимим конкретною версією мови програмування. Усі типи даних поділяються на дві групи: **скалярні** (прості) і **структуровані** (складені). Скалярні типи даних у свою чергу поділяються на **стандартні** і типи **користувача**. До стандартних типів належать **цілочислові**, **дійсні**, **символьні**, **бульові (логічні)** і **покажчики**.

Дані цілочислових типів можуть бути представлені як у десятковій, так і у шістнадцятковій системах. В останньому випадку перед числом без пробілу ставиться символ $.
У десятковій системі числа можуть бути записаними двома способами: з фіксованою точкою і з плаваючою точкою.
Дійсні десяткові числа у форматі з фіксованою точкою записуються за правилами арифметики, причому ціла частина відділяється від дробової точкою. Якщо дробова частина числа відсутня, то число вважається цілим. Перед числом може знаходитись знак "+" або "-". Якщо знак відсутній, число приймається за додатне.

Приклади запису цілих і дійсних чисел у форматі з фіксованою точкою:

`125`	– ціле десяткове число
`$1F`	– ціле шістнадцяткове число
`12.246`	– дійсне число
`-0.07`	– від'ємне дійсне число

Дійсні десяткові числа у форматі з плаваючою точкою представляються у експоненціальному виді:

mE*p,

де
`m` – мантиса (ціле або дробове число),
`E` – означає "десять у степені",
`*` – знак степені числа десять (плюс або мінус),
`p` – показник степені (ціле число до 4-х розрядів).

Приклади дійсних чисел у форматі з плаваючою точкою:

Формат з плаваючою точкою	***Алгебраїчний еквівалент***	***Десятковий еквівалент***
`2.1342E+02`	$2,1342*10^{2}$	`213.42`
`-1E-05`	$-1*10^{-5}$	`-0.00001`

Типи користувача - перелічуваний і інтервальний - створюються самим програмістом.

Структуровані типи даних у своїй основі мають один або декілька скалярних типів даних. До структурованих типів даних відносяться рядки, масиви, множини, записи, файли, процедурні типи, об'єктові типи тощо. Структуровані типи даних більш детально будуть розглядатися далі.

2.2. Скалярні типи даних

Цілочислові типи даних. Object Pascal має сім цілих типів даних, діапазон і формат яких наведений далі у таблиці. Найбільш універсальним типом даних, який підтримується усіма компіляторами є тип `integer`.

Тип	*Діапазон*	*Формат (біт)*
Byte	0..255	8
ShortInt	-128..127	8
Word	0..65535	16
SmallInt	-32768..32767	16
LongWord	0..4294967295	32
LongInt(Integer)	-2147483648.. 2147483647	32
Int64	$-2^{63}..2^{63}-1$	64

Обираючи будь-який тип даних, а особливо один із числових типів даних, слід бути обережним, оскільки усі діапазони є кінцевими. Це входить у протиріччя із математичними поняттями чисел, де множина числових значень, цілочислових і дійсних, є нескінченною. Призначаючи у програмі певній змінній певний тип Object Pascal слід передбачити можливі наслідки. Наприклад, нехай ми маємо справу із змінною `B` типу `byte`, тоді дії, що наведені нижче призведуть до помилки.

```
B:=250;
Writeln(B+10);
```

Адже, замість очікуваного значення 260 ви побачите на екрані, як це не дивно, значення 4. При додаванні відбудеться циклічний зсув, втрата старшого розряду і виведення невірного значення. У тому випадку, коли для змінної `B` обрати інший тип, наприклад `Word` або `SmallInt`, результат буде коректним.

Приклади описувань цілочислових змінних:

```
var
  n:byte;
  i,j,k:integer;
  Num:LongWord;
```

Дійсні типи даних. Object Pascal має шість дійсних типів даних, які різняться між собою діапазоном і кількістю значущих цифр. Далі наведені характеристики цих шести дійсних типів. Найбільш універсальним типом, який підтримується усіма компіляторами, є тип `real`.

Тип	*Діапазон*	*Кількість значущих*	*Формат (байт)*

		цифр	
Single	$1.5x10^{-45}$ .. $3.4x10^{38}$	7-8	4
Real48 (Real)	$2.9x10^{-39}$ .. $1.7x10^{34}$	11-12	6
Double	$5.0x10^{-324}$ .. $1.7x10^{308}$	15-16	8
Extended	$3.6x10^{-4951}$ .. $1.1x10^{4932}$	19-20	10
Comp	$-2^{-63}+1$ .. $2^{63}-1$	19-20	8
Currency	-922337203685477.5808 .. 922337203685477.5807	19-20	8

Так само, як і у випадку з цілими числами, слід приділяти увагу вибору діапазону значень. Скінчена множина значень Pascal ставиться у відповідність нескінченній множині дійсних чисел. Нехай вас не вводять в оману величезні степені числа 10, чим більшим або меншим є число, тим з меншою точністю воно може бути представлене. Після віддалення від нульового значення на 20 цифр обов'язково виникають "дірки" у представленні чисел. Наприклад, якщо ви захочете точно зберегти число 0.123456789012345678901234567890123456789, то повністю вам це не вдасться, оскільки залишаться лише перші 20 цифр включаючи десяткову точку, а саме 0.12345678901234567 8.

Крім того, обираючи відповідний тип, не варто обирати тип з найбільшим діапазоном. Треба реально оцінити потреби конкретної задачі і обрати тип з найменшими витратами пам'яті. Особливо ця вимога є актуальною при використанні великих багатовимірних масивів, коли об'єми інформації, що зберігається, суттєво збільшуються.

Приклади описувань дійсних змінних:

```
var
  x1,y1,x2,y2:real;
  x,y,z:extended;
  Pay:currency;
```

Символьний тип даних. Object Pascal підтримує два символьних типи: `AnsiChar` (`Char`) і `WideChar`.
`AnsiChar` - це символи у кодуванні `ANSI`, яким відповідають числа у діапазоні від 0 до 255. Для збереження даних такого типу відводиться один байт пам'яті. Різновид набору `ANSI`, який підтримує символи кирилиці у Windows має назву `Windows-1251`.
Тип `WideChar` - це символи у кодуванні `Unicode`. Для збереження даних у такому виді відводиться два байти пам'яті, а самим символам відповідають числа у межах від 0 до 65535.

Приклади описувань символьних змінних:

```
var
  c:char;
  Kod:WideChar;
  N:AnsiChar;
```

Бульові (логічні) дані. Логічна величина описується типом `Boolean`, може мати одне із двох значень `true` (істина) або `false` (неправда), і займає у пам'яті один байт.

Приклади описувань логічних змінних:

```
var
```

```
  Umova,Solution,B:boolean;
```

2.3. Типи користувача

Крім стандартних типів, Object Pascal підтримує використання скалярних типів, зміст яких визначає сам програміст. До таких типів відносяться перелічуваний та інтервальний типи. У пам'яті комп'ютера вони займають 1 байт і тому не можуть містити більше ніж 256 елементів. Застосування типів користувача підвищує наочність програми, робить більш легким пошук помилок, економить пам'ять.

Перелічуваний тип визначається безпосереднім переліком усіх можливих значень, які може приймати змінна цього типу. Окремі значення вказуються через кому, а весь список береться у круглі дужки. Формуючи перелічуваний тип, можна або створити новий тип і після цього визначити змінні нового типу (такому способу слід надавати перевагу), або безпосередньо при описуванні змінної перелічити можливі її значення.

Приклади описування даних перелічуваного типу:

```
type
  Comandy = (Dynamo, Shakhtar, Metalist);
var
  C1,C2:Comandy;
  Season:(Winter,Spring,Summer,Autum);
```

Слід зазначити, що для перелічуваних типів даних немає стандартних процедур введення-виведення, тому програміст сам повинен потурбуватися про такі процедури. Для роботи з перелічуваними типами даних у Object Pascal реалізовані стандартні підпрограми `succ`, `pred`, `ord`.

Інтервальний тип дозволяє вказати дві константи, що визначають межі діапазону значень даної змінної. Компілятор при кожній операції із змінною інтервального типу генерує підпрограми перевірки, у яких визначається приналежність результату вказаному діапазону. Константи, які визначають межі діапазону повинні належати до одного зі скалярних типів даних за винятком дійсного. Значення першої константи повинно бути меншим за друге.

Приклади описування даних інтервального типу:

```
type
  Diametr = 2..100;
var
  d1,d2,d3:Diametr;
```

2.4. Тотожність і сумісність типів

Для запису вірних виразів і особливо для коректного використання процедур і функцій слід з'ясувати поняття *тотожності* і *сумісності* типів.

Два типи є **тотожними**, якщо вони описані разом або для їхнього визначення використано один і той самий ідентифікатор типу. Вимога тотожності типів висувається при передачі параметрів у процедури або функції.

Сумісності типів слід дотримуватись у виразах відношення і операторах присвоювання.

У **виразах відношення** два типи вважаються сумісними, якщо:

- Обидва типи є однаковими або тотожними.
  Тобто, можна порівнювати `Real` і `Real`, `Byte` і `Byte`, `Boolean` і `Boolean` тощо;
- Обидва типи належать до загального цілочислового типу
  Тобто, можна порівнювати `Byte` і `Integer`, `Integer` і `LongWord` тощо;
- Обидва типи належать до загального дійсного типу.
  Тобто, можна порівнювати `Real` і `Single`, `Double` і `Comp` тощо;
- Обидва типи належать до символьного або рядкового типів.
  Тобто можна порівнювати `String` і `Char`;
- Один тип є піддіапазоном іншого.
  Тобто, якщо ви визначили власний тип, як піддіапазон цілочислового значення, то ви маєте право порівнювати його з цілочисловими типами.

В **операціях присвоювання** два типи вважаються сумісними, якщо:

- Обидва типи є тотожними;
- Обидва типи є сумісними скалярними типами, причому значення другого даного потрапляють у діапазон можливих значень першого. Наприклад, типові `Integer` можна присвоювати значення типа `Byte`, типові `LongWord` - типа `Word` або `Byte`, типові `Double` - типа `Single` тощо. Зазначимо, що зворотні присвоювання можуть призвести до помилки;
- Перший тип є дійсним, а другий - цілочисловим. Наприклад, типові `Real` можна присвоювати тип `Word` тощо;
- Перший тип є рядковим, а другий символьним. Тобто, типові `String` можна присвоювати тип `Char` тощо;

Існують ще деякі ознаки сумісності типів, які стосуються множинних, об'єктових типів і покажчиків.

3. Вирази, операнди, операції

Вираз визначає порядок дій над елементами даних і складається з операндів і операцій.
Операндом може бути: константа, змінна, виклик функції, вираз у круглих дужках.
Операції визначають дії, які треба виконати над операндами.

```
Result := (2 + y) * sin(x);
           К О З        З
           \___/     \___/
             В     О   Ф
           \_______________/
                   В
```

```
К – операнд-константа
З – операнд-змінна
Ф – операнд-функція
В – операнд-вираз
О – операція
```

Операції у Pascal розділяють на арифметичні, відношення, логічні (бульові) та ін. Вирази, відповідно, називають арифметичними, відношення, логічними. Операції мають різний пріоритет виконання, яким можна керувати за допомогою круглих дужок.

Операції можуть бути унарними і бінарними. У першому випадку операція відноситься до одного операнда і завжди записується перед ним, у другому випадку операція виражає відношення між двома операндами і записується між ними.

Наприклад:

унарні операції	бінарні операції

```
-a              X*y
not vv          a-2.4
```

3.1. Арифметичні вирази і операції

Арифметичний вираз породжує ціле або дійсне значення. Найпростішими формами арифметичних виразів є: ціла або дійсна константа без знака; ціла або дійсна змінна; елемент масиву цілого або дійсного типу; функція, яка приймає ціле або дійсне значення. Інші арифметичні вирази утворюються з вищезгаданих шляхом використання круглих дужок і арифметичних операцій. При формуванні арифметичних виразів у Pascal важливо "розтягнути" алгебраїчний вираз у один рядок.

Арифметичні операції виконують арифметичні дії у виразах над значеннями операндів цілочислових та дійсних виразів. Основні арифметичні операції і тип результату їхньої дії наведені нижче.

Операція	*Дія*	*Тип операндів*	*Тип виразу*
Унарні операції			
-	Заміна знаку	integer real	integer real
not	Арифметичне заперечення	integer	integer
Бінарні операції			
+	Додавання	real, real real, integer integer, integer	real real integer
-	Віднімання	real, real real, integer integer, integer	real real integer
*****	Множення	real, real real, integer integer, integer	real real integer
/	Ділення	real, real real, integer integer, integer	real real real
div	Ділення без остачі	integer, integer	integer
mod	Залишок від ділення	integer, integer	integer
and	Арифметичне І	integer, integer	integer
or	Арифметичне АБО	integer, integer	integer
xor	Арифметичне виключне АБО	integer, integer	integer

shl	Зсув уліво по бітах	integer, integer	integer
shr	Зсув управо по бітах	integer, integer	integer

При використанні унарної **операції заміни знаку** відбувається перетворення додатного цілого або дійсного числа у від'ємне, а від'ємного - у додатне.

Арифметичне заперечення проводиться над цілими числами не в цілому а побітно (див. тема 2). При цьому відбувається заміна нулів одиницями і одиниць нулями у межах кожного розряду числа. Кінцевий результат суттєво залежить від типу операнда.

Приклад

тип даних - byte	тип даних - word
$22_{10} = 00010110_2$,	$22_{10} = 0000000000010110_2$,
not 22 = not 00010110 = 11101001	not 22 = not 0000000000010110 = 1111111111101001
$11101001_2 = 231_{10}$	$11101001_2 = 65511_{10}$

Бінарні операції додавання, віднімання і множення виконуються звичайним чином і не потребують особливих коментарів. При використанні слід лише дивитися за тим, щоб результат такої операції знаходився у межах діапазону відповідного типу даних. Недотримання цієї вимоги може призвести до помилки у найпростіших операціях. Також слід брати до уваги те, що у випадку, коли один з операндів має дійсний тип, то і результат буде мати дійсний тип.

Бінарна операція ділення потребує більшої уваги, адже у випадку, коли навіть обидва операнди мають цілочисловий тип, результат все рівно буде мати дійсний тип.
Тобто, $4 / 2 \neq 2$, а $4 / 2 = 2.000$
Це може не мати принципового значення для користувача програми, але програміст повинен ретельно доглядати за відповідністю типів даних.

Операція цілочислового ділення може застосовуватися лише для цілочислових операндів. Проте її результат завжди має також цілочисловий тип. Слід звернути увагу на те, що результат не округлюється до ближчого значення, а береться ціла частина результату (залишок відкидається).

Приклад:

```
123 div 4 = 30
```

Операція знаходження залишку від ділення також завжди оперує з цілочисловими операндами. І результат її дії також має цілочисловий тип. Зверніть увагу на те, що результатом є не дробова частина, що залишається після ділення, а саме ціле число, яке вже не можна розділити на дільник.

Приклад:

```
123 mod 4 = 3
```

Бінарна арифметична операція І виконується над цілими операндами по бітах. Правила формування результату є такими: якщо у відповідних бітах двох операндів одночасно присутня 1, тоді і відповідний біт результату отримає значення 1. У всіх інших випадках результат відповідного біту буде дорівнювати 0. Причому, на відміну від унарної операції арифметичного заперечення, тут кінцевий результат не залежить від базового типу операндів.

Правила формування результату:

I1	0	0	1	1
I2	0	1	0	1
I1 and I2	0	0	0	1

Приклад

тип даних - `byte`

22_{10} = 00010110_2, 12_{10} = 00001100_2,
12 and 22 = 00001100 and 00010110 = 00000100
00000100_2 = 4_{10}
тобто, 12 and 22 = 4

тип даних - `word`

22_{10} = 0000000000010110_2, 22_{10} = 0000000000010110_2,
12 and 22 = 0000000000010110 and 0000000000010110 =
0000000000000100
0000000000000100_2 = 4_{10}
тобто, 12 and 22 = 4

Бінарна арифметична операція АБО виконується над цілими операндами по бітах. Правила формування результату є такими: якщо у відповідних бітах двох операндів присутня хоча б одна 1, тоді і відповідний біт результату отримає значення 1. У всіх інших випадках результат відповідного біту буде дорівнювати 0. Кінцевий результат не залежить від типу операнда.

Правила формування результату:

I1	0	0	1	1
I2	0	1	0	1
I1 or I2	0	1	1	1

Приклад

тип даних - `byte`

22_{10} = 00010110_2, 12_{10} = 00001100_2,
12 or 22 = 00001100 or 00010110 = 00011110
00011110_2 = 30_{10}

тобто, 12 or 22 = 30

Бінарна арифметична операція виключного АБО виконується над цілими операндами побітно. Правила формування результату є такими: якщо у відповідних бітах двох операндів присутня тільки одна 1, тоді і відповідний біт результату отримає значення 1. У всіх інших випадках результат відповідного біту буде дорівнювати 0. Кінцевий результат не залежить від типу операнда.

Правила формування результату:

I1	0	0	1	1
I2	0	1	0	1
I1 xor I2	0	1	1	0

Приклад

тип даних - byte

$22_{10} = 00010110_2$, $12_{10} = 00001100_2$,
12 xor 22 = 00001100 xor 00010110 = 00011010
$00011010_2 = 26_{10}$
тобто, 12 xor 22 = 26

Операція зсуву ліворуч по бітах застосовується для двох цілочислових операндів, причому перший операнд формує початкове значення, а другий операнд вказує на скільки позицій треба здійснити зсув уліво. Нові позиції справа заповнюються нулями. Кінцевий результат, як і для унарної операції арифметичного заперечення залежить від типу операндів.

Приклад

тип даних - byte

$22_{10} = 00010110_2$,
22 shl 4 = 01100000
$01100000_2 = 96_{10}$
тобто, 22 shl 4 = 96

тип даних - word

$22_{10} = 0000000000010110_2$,
22 shl 4 = 0000000101100000
$0000000101100000_2 = 4_{10}$
тобто, 22 shl 4 = 352

Операція зсуву праворуч по бітах застосовується для двох цілочислових операндів, причому перший операнд формує початкове значення, а другий операнд вказує на скільки позицій треба здійснити зсув управо. Нові позиції зліва заповнюються нулями. Кінцевий результат не залежить від типу операндів.

Приклад

тип даних - byte

тип даних - word

$22_{10} = 00010110_2$,
22 shr 2 = 00000101
$00000101_2 = 5_{10}$
тобто, 22 shr 2 = 5

$22_{10} = 0000000000010110_2$,
22 shr 2 = 0000000000000101
$0000000000000101_2 = 4_{10}$
тобто, 22 shr 2 = 5

3.2. Вирази і операції відношення

Операції відношення виконують порівняння двох операндів. Результат такого порівняння завжди має логічний тип. Величини, що порівнюються, можуть належати до будь-якого скалярного чи перелічуваного типу даних.

Операція	*Назва*	*Вираз*	*Результат*
=	дорівнює	**a = b**	true, якщо a дорівнює b
<>	не дорівнює	**a <> b**	true, якщо a не дорівнює b
>	більше	**a > b**	true, якщо a більше b
<	менше	**a < b**	true, якщо a менше b
>=	не менше	**a >= b**	true, якщо a не менше b
<=	не більше	**a <= b**	true, якщо a не більше b
in	належність	**a in b**	true, якщо a знаходиться у b

3.3. Логічні вирази і операції

Наслідком виконання логічного виразу є логічне значення true або false. Операндами логічних виразів можуть бути тільки логічні дані. Дозволяється застосовувати логічні операції і до цілочислових даних, однак результатом їхньої дії у такому випадку буде ціле число, тому вони віднесені до арифметичних операцій.

Логічними операціями є:

- **not** - логічне заперечення;
- **and** - логічне І;
- **or** - логічне АБО;
- **xor** - логічне виключне АБО.

Результат дії таких функцій наведений у таблиці.

A	**B**	**not A**	**A and B**	**A or B**	**A xor B**
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Дуже часто при складанні логічних виразів використовуються операції відношення, у яких об'єднуються операнди різних типів. Причому тут слід пам'ятати, що операції відношення мають найнижчий пріоритет і тому вирази, які вони утворюють слід обов'язково брати у круглі дужки, наприклад:

```
(x>=0) and (x<=200)
(Name='Ivan') or (Name='Alex')
```

Опустивши круглі дужки у таких виразах, ми отримаємо помилковий запис, оскільки першою буде виконуватися операція `and(or)`, а вже після того,- операції порівняння, де і виникне помилка.

3.4. Пріоритет виконання операції

Порядок виконання операцій, що задаються операторами, визначається у відповідності до порядку, в якому вони наведені у даному переліку:

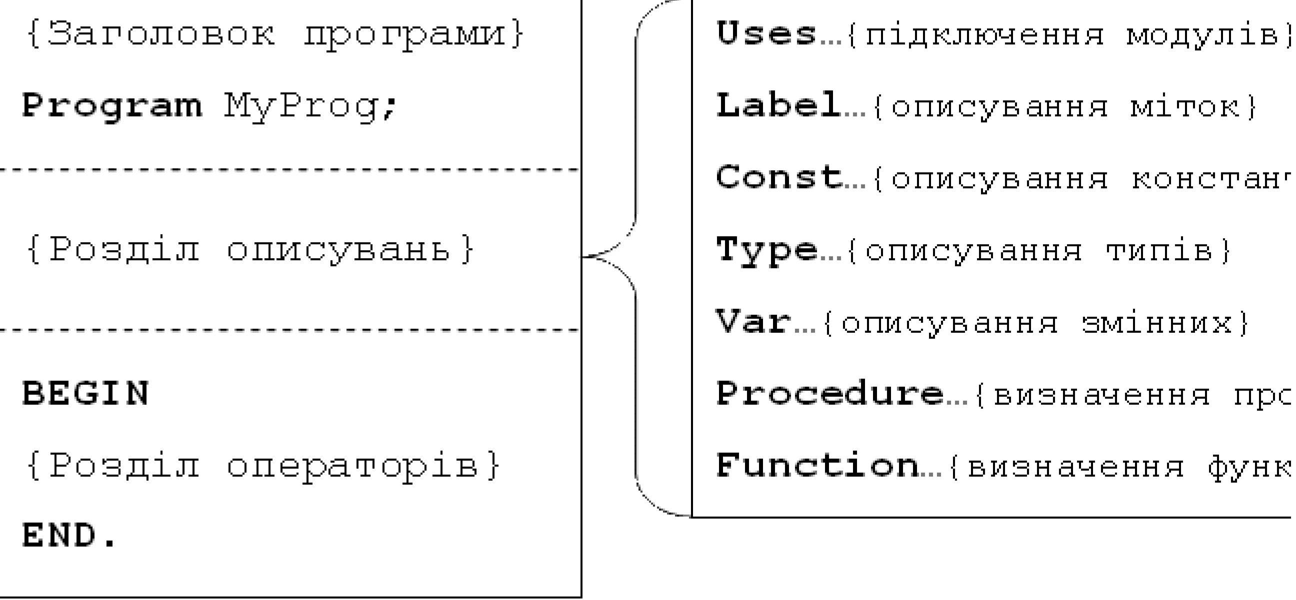
1. функція;
2. одинарний мінус (мінус перед одним оператором);
3. операція `not` (заперечення);
4. операції типу множення: `*, /, div, mod, and, shl, shr`;
5. операції типу додавання: `+, -, or, xor`;
6. операції відношення: `=, <>, <, >, <=, >=, in`.

Для визначення першочерговості виконання операцій існують кілька **правил**:

1. Операнд, який знаходиться між двома операціями з різними пріоритетами, зв'язується з операцією, яка має більш високий пріоритет.
2. Операнд, який знаходиться між двома операціями з рівними пріоритетами, зв'язується з тією операцією, яка знаходиться ліворуч.
3. Вираз у круглих дужках перед виконанням обчислюється як окремий операнд.
4. Операції з однаковим пріоритетом виконуються зліва направо з можливим регулюванням порядку обчислення круглими дужками.

4. Структура Pascal - програми

Pascal-програма складається із заголовка, розділу описувань і розділу операторів.



Заголовок програми розпочинається ключовим словом `Program`, після якого ви самостійно визначаєте ім'я своєї програми. Заголовок виконує допоміжну функцію і ніяк не впливає на хід виконання програми. Заголовок не є обов'язковим елементом програми. Якщо ви вже записуєте заголовок, то він повинен мати певний сенс.

Приклади:

```
Program First;
Program TestOfEnglish;
```

Розділ описувань ще до початку виконання програми визначає, які модулі, мітки, константи, типи, змінні, процедури і функції можуть у ній використовуватися. Далі можна детально ознайомитися із структурою цього розділу.

Розділ операторів є останнім розділом програми. Він визначає дії, які повинна виконувати програма. Розділ операторів починається ключовим словом `BEGIN`. За ним наводяться оператори, які відокремлюються між собою крапкою з комою. В кінці розділу ставиться ключове слово `END` з крапкою.

Структура розділу описування

Розділ описування складається з шести необов'язкових підрозділів:

1. Підрозділ підключення модулів;
2. Підрозділ описування міток;
3. Підрозділ визначення констант;
4. Підрозділ визначення типів;
5. Підрозділ описування змінних;
6. Підрозділ визначення процедур і функції користувача.

Pascal дозволяє, щоб кожен з підрозділів зустрічався будь-яку кількість разів і у будь-якій послідовності. Винятком є підрозділ підключення модулів, який обов'язково повинен бути першим. Також слід дивитися за тим, щоб елементи, на які є посилання, наприклад константи для визначення типу даних, були описані раніше.

Підключення модулів відбувається шляхом переліку через кому їхніх імен. Розпочинається цей розділ зарезервованим словом `Uses`. Більш детальну інформацію про модулі ви можете отримати у темі 19.

Приклади підключення модулів:

```
uses Crt, Graph;
uses MyMath, PVGraph, Service;
```

Описування міток розпочинається ключовим словом `Label`, після якого перелічуються ідентифікатори міток, розділені комами. Мітка може ставитись перед будь-яким оператором програми, що дозволяє здійснити перехід на цей оператор при виконанні оператора `goto`. Мітка складається з ідентифікатора, за яким слідує двокрапка. Використання міток суперечить ідеям структурного програмування. Міток повинно бути якнайменше.

Приклади міток:

```
Label
  Quit, err, L11;
```

Визначення констант розпочинається ключовим словом `Const`, за яким подається перелік виразів, у яких відповідним ідентифікаторам (іменам констант) присвоюються сталі значення (числові, рядкові, логічні тощо). Кожен вираз складається з ідентифікатора, за яким після знаку рівності вказується його значення у програмі. Елементи списку констант розмежовуються крапкою з комою.

Приклади визначення констант:

```
Const
  Limit = 255;
  Name = 'Anton';
  Vysota = 120.5;
```

Зверніть увагу на те, що тип константи визначається автоматично. Присвойте константі значення 25, і вона отримає тип `byte`, присвойте 25.0 - отримає тип `real`.

Особливим видом констант є **типізовані константи**. З точки зору компілятора вони є не константами, а змінними із початковими значеннями. Використання типізованих констант дозволяє з одного боку чітко визначити приналежність константи до конкретного типу даних, а з іншого боку, - об'єднати описування змінної із оператором присвоювання їй початкового значення.

Формат описування типізованої константи нагадує поєднання описувань константи і змінної:

```
Const
  Limit: byte = 255;
  Name:string = 'Anton';
  Vysota:real = 120.5;
```

Визначення типів користувача розпочинається із зарезервованого слова `type`. Кожному новому типові слід надати своє ім'я (ідентифікатор) і після знаку рівності визначити зміст типу. Слід зазначити, що нестандартний тип даних може бути введений і без попереднього визначення. У такому випадку його записують у розділі описування змінних. Вважається добрим стилем програмування, коли новий тип має своє ім'я.

Приклади визначення типів:

```
Type
  StHelp = string[80];
  M1 = array[1..25] of real;
```

Описування змінних розпочинається зарезервованим словом `var`. Змінна - це область пам'яті, у якій знаходяться дані, з якими оперує програма. Для того, щоб програма мала змогу звернутися до змінної (області пам'яті), така змінна повинна мати унікальне у межах програми ім'я. Таке ім'я призначає програміст. При створенні нових імен слід дотримуватися певних правил, це суттєво підвищить наочність програми і знизить кількість помилок.

Кожна змінна, яка зустрічається у програмі повинна буди описаною. Можна описувати кожну змінну окремо, а можна описувати одразу групи змінних одного типу. У першому випадку вказується ім'я (ідентифікатор) змінної і через двокрапку її тип (стандартний або тип користувача). У другому випадку наводиться список імен змінних, розділених між собою комами, і далі, через двокрапку, вказується тип групи змінних.

Приклади описування змінних:

```
Var
  Summa, dlina,x1      : real;
  Helpscreen : array [1..25] of StHelp;
  Valid : boolean;
  I,j,k : integer;
  Buffer,b1 : array [1..127] of byte;
```

Описування процедури (функції) починається із зарезервованого слова `procedure` `(function)`. Докладніше використання процедур (функцій) буде розглянуто далі у темі 13.

Коментарі до програм

Коментар - пояснювальний текст оточений у дужки коментарю { } або (* *) і розміщений у будь-якому місці програми, де дозволяється ставити пробіл. Текст коментарю може містити майже усі символи кодової таблиці комп'ютера за винятком службових символів і символів коментарів. Обмежень на довжину коментарів немає. Вкладати один коментар у інший можна лише використовуючи різні групи дужок коментарів.

Коментарі слід обов'язково використовувати, оскільки з часом забуваються прийоми і рішення, які були знайдені. Вважається добрим стилем, коли 20-30% тексту програми складають саме коментарі. Пояснювати можна призначення програми в цілому, або окремих її блоків, розшифровувати змінні, константи, типи, виокремлювати логічні розгалуження програми, пояснювати найбільш складні моменти алгоритму тощо.

Рекомендується фігурні дужки використовувати для пояснень до програми, а комбінацію круглої дужки із зірочкою - для тимчасового вилучення фрагменту програми (включаючи і коментарі-пояснення). У такий спосіб ви швидко знайдете частину програми, яку тимчасово вилучили з алгоритму.

Приклад використання коментарів у програмі:

```
{ коментар до програми в цілому }
Program . . .
var . . .
  { пояснення до змінних, констант . . }
BEGIN
```

```
  { пояснення до частин програми }
  <оператор>
  (* початок частини, яка тимчасово вилучена
  { пояснення до блоку }
  <блок операторів>
  завершення частини, яка тимчасово вилучена
  *)
  { прикінцевий коментар }
END.
```

5. Введення-виведення даних

5.1. Введення даних з клавіатури

Введення даних у Pascal-програмі здійснюється за допомогою процедур `read` або `readln`. Формат використання процедур у випадку введення даних з клавіатури має вигляд:

```
Read(v1,v2,v3,...,vn);
Readln(v1,v2,v3,...,vn); ,
```

де

`v1,v2,v3,...,vn` - змінні допустимих типів даних.

Значення `v1,v2,v3,...,vn` уводяться через один пробіл з клавіатури і висвітлюються на екрані. При введенні даних необхідно слідкувати за відповідністю типів списку змінних і типів значень, які вводяться. Після набору даних для однієї процедури `read` натискається клавіша <Enter>

Різниця між процедурами `read` і `readln` полягає в тому, що при використанні процедури `readln` після зчитування останнього у списку значення однієї процедури дані для наступної процедури будуть зчитуватись, починаючи з нового рядка , а при використанні процедури `read` - введення буде продовжуватись у тому ж рядку.

Приклад:

```
Var
  N 1, N2, N3: real;
  Ch1,ch2: char;
  . . . . . . . . .
  Read(n1);
  read(n2,n3);
  Readln(ch1);
  readln(ch2);
```

При наявності в програмі такого фрагмента потрібно з клавіатури вводити дані.

```
23.88 <Enter> 2.12e-02 56.0 <Enter>y<Enter>
N<Enter>
```

5.2. Виведення результатів на екран

Виведення даних у Pascal-програмі здійснюється за допомогою процедур `write` або `writeln`. Формат використання процедур у випадку виведення даних на екран монітора має вигляд:

```
Write(v1,v2,v3,...,vn);
Writeln(v1,v2,v3,...,vn);,
```

де

`v1,v2,v3,...,vn` - список виразів припустимих типів.

Різниця між процедурами write і writeln полягає в тому, що при використанні процедури writeln після виведення на екран останнього елемента в списку значень курсор переводиться на новий рядок, де продовжується виведення значень нової процедури. При використанні процедури write після закінчення виведення списку значень однієї процедури виведення списку значень наступної процедури продовжується в тому ж рядку. Якщо при виведенні інформації на екран курсор дійшов до кінця рядка, то він автоматично перейде до наступного рядка, а якщо ми досягнемо кінця екрану, то усі рядки змістяться на один рядок уверх і виведення буде продовжено від початку останнього рядка. Процедура writeln, яка записана без параметрів, викликає переведення курсору у початок наступного рядка.
Приклад:

```
Const Misto = 'Київ';
  Var x,y : real;
  . . . . . . . . . . . . . . . .
  x:=12.0; y:=3.0;
  Write(Misto);Writeln(2002);
  Writeln(x*(y+1),x/y);
  Writeln;
  Write('Кінець');
```

Результат виведення на екран

```
Київ2002
4.8000000000E+0001 6.0000000000E+0000

Кінець
```

Зручними засобами є використання **процедури переведення курсору** у визначену позицію екрану. Для використання такої можливості слід перш за все ознайомитись із системою координат у текстовому режимі. При виведенні інформації на екран у текстовому режимі увесь екран поділений на знакомісця, у кожному з яких може розміститися тільки один символ або пробіл. Кількість рядків і стовпчиків на екрані є сталою і складає, відповідно, 25 і 80. Найвище знакомісце зліва має найменші координати (1,1). Найнижче знакомісце справа має найбільші координати (80,25). Тобто, якщо вказуються координати екрану (10,15), це означає, що мова йде про 15-й рядок (зверху) і про 10-у позицію (зліва).

Для переведення курсору у відповідну позицію слід скористатися процедурою GotoXY (Col,Row:integer); зі стандартного модулю Crt. Для коректної роботи процедури модуль Crt слід підключити до програми. Також корисною процедурою з модулю Crt є процедура очищення екрану ClrScr. Далі наведений фрагмент програми, у якому до неї підключається згадуваний вище модуль, очищується екран і курсор переводиться у 40-у позицію 12-го рядка екрану. Починаючи з цієї позиції виводиться текст "Hello !".

```
uses Crt;
  BEGIN
  ClrScr;
  GotoXY(40,12); write('Hello !');
  END.
```

5.3. Формати виведення виразів

В процедурах виведення write і writeln є можливість записувати вирази, які визначають ширину полів виведення. У наведених нижче форматах використовуються такі позначення:
i,p,q - цілочислові вирази;
r - дійсні вирази;
b - логічні вирази;

ch - символьні вирази;
s - рядкові вирази;
- цифра;
* - знак "+"або "-";
_ - пробіл (на екран не виводиться).

При використанні у списку значень процедур write або writeln цілочислових, логічних, символьних і рядкових виразів виведення здійснюється починаючи з поточної позиції курсору. При виведенні дійсних виразів у поле, яке має ширину 17 символів, виводиться дійсне число в форматі з плаваючою точкою.
Якщо r>=0.0, використовується формат _#.##########E*##.
Якщо r<0.0, використовується формат -#.##########E*##.

Приклади:

Значення	***Вираз***	***Результат***
i := 2233;	write(i);	2233
i := 123;	write(i,i,i);	123123123
r := 456.789;	write(r);	_4.5678900000E+02
r := -0.0000444;	write(r);	-4.4400000000E-05
ch:= 'y';	write(ch);	y
ch:= '&';	write(ch,ch);	&&
s := 'прізвище';	write(s);	прізвище
s := 'день';	write(s,s,s);	деньденьдень
b := true;	write(b);	true
b := false;	write(b,**not** b);	falsetrue

Якщо після цілочислового, дійсного, логічного, символьного або рядкового виразу через двокрапку вказати цілочисловий вираз p, то виведення буде здійснюватись у крайні праві позиції поля, ширина якого p. Виведення дійсних виразів здійснюється в форматі з плаваючою точкою, при цьому вираз p не може бути меншим, ніж 8.

Приклади:

Значення	***Вираз***	***Результат***
i := 2233;	write(i:6);	__2233
i := 123;	write(i+i:3);	246
r := 456.789;	write(r:11);	_4.5678900000E+02
r := -0.0000444;	write(r:8);	-4.4400000000E-05
ch:= 'y';	write(ch:4);	___y
ch:= '&';	write(ch:3,ch:5);	__&____&
s := 'прізвище';	write(s:11);	___прізвище
s := 'день';	write(s:4,s:5,s:6);	день_день__день
b := true;	write(b:6);	__true

```
b := false;      write(b:6,not b:5); _false_true
```

Якщо після дійсного виразу r через двокрапки вказати два цілочислових вирази p і q, то в крайні праві позиції поля, ширина якого p, буде виводитись вираз r в форматі з фіксованою точкою, причому після десяткової точки виводиться q цифр (0<=q<=24), які представляють дробову частину числа.
Якщо q=0, ні дробова частина, ні десяткова точка не виводяться.
Якщо q>24, то при виводі використовується формат з плаваючою точкою.
Якщо залишок позицій для цілої частини виразу менший, ніж число позицій у виразі, значення p автоматично збільшується на необхідну кількість позицій.

Приклади:

Значення	***Вираз***	***Результат***
`r := 456.789;`	`write(r:5:1);`	`456.7`
`r := -0.0000444;`	`write(r:11:6);`	`___0.000044`
`r := 1234.56789;`	`write(r:5:2);`	`1234.57`

Рекомендується при виведенні дійсних чисел використовувати формат із фіксованою точкою.

Закріпити навички використання операторів введення-виведення а також форматів виведення ви зможете виконавши лабораторну роботу №1.

6. Контрольні запитання

1. Охарактеризуйте алфавіт мови програмування Pascal.
2. Які складені символи можна використовувати у Pascal-програмі?
3. Що таке зарезервовані слова, стандартні ідентифікатори та ідентифікатори користувача у Pascal-програмі?
4. Вкажіть правила запису ідентифікаторів у Pascal-програмі.
5. Охарактеризуйте типи даних Pascal.
6. Які існують форми запису цілочислових і дійсних даних у Pascal-програмі?
7. Охарактеризуйте цілочислові типи даних.
8. Охарактеризуйте дійсні типи даних.
9. Охарактеризуйте символьні і логічні типи даних.
10. Охарактеризуйте перелічуваний тип даних.
11. Охарактеризуйте інтервальний тип даних.
12. Охарактеризуйте поняття тотожності і сумісності типів.
13. Охарактеризуйте поняття виразу, операнду, операції.
14. Охарактеризуйте загальні правила створення арифметичних виразів і операцій у Pascal-програмі.
15. Особливості застосування операцій `+, -, *, /`.
16. Особливості застосування операцій цілочислового ділення DIV і залишку від ділення MOD.
17. Особливості застосування логічних операції над цілими числами NOT, AND, OR, XOR.
18. Особливості застосування операцій циклічного зсуву SHL і SHR.
19. Охарактеризуйте вирази і операції відношення у Pascal-програмі.
20. Охарактеризуйте логічні вирази і операції у Pascal-програмі.
21. Вкажіть пріоритет виконання операцій.
22. Охарактеризуйте структуру Pascal-програми.
23. Яку структуру має розділ описування у Pascal-програмі?
24. Як підключаються модулі до Pascal-програми?

25. Як описуються мітки у Pascal-програмі?
26. Як визначаються константи у Pascal-програмі?
27. Як визначаються типи даних у Pascal-програмі?
28. Як описуються змінні у Pascal-програмі?
29. З якою метою і як використовуються коментарі у Pascal-програмі?
30. Яким чином здійснюється введення даних у Pascal-програмі?
31. Яким чином здійснюється виведення даних у Pascal-програмі?
32. Як зручно переводити курсор у певно позицію на екрані?
33. Охарактеризуйте формати виведення символьної і логічної інформації.
34. Охарактеризуйте формати виведення цілочислової інформації.
35. Охарактеризуйте формати виведення інформації дійсного типу.

Тема 12. Базові оператори Pascal

План

1. Прості і структуровані оператори
2. Оператори перевірки умови
3. Оператори повторювання
4. Контрольні запитання

У цій темі ми з вами познайомимось із основами створення простих програм. Алгоритми, які реалізуються за допомогою операторів, що розглядаються далі, є основою будь-якої програми. Адже практично у кожній з програм є і лінійний алгоритм, і розгалужений, і циклічний. Без детального вивчення цих алгоритмів і прийомів програмування не варто сподіватися на подальші успіхи. У цій темі буде розглянуто 12 прикладів завершених програм, які ілюструють використання базових операторів Pascal. Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторні роботи №2-7.

Формати операторів наводяться за такими домовленостями:

- зарезервовані слова наводяться напівжирним шрифтом.
  Наприклад, **`Const, Procedure`**;
- Необов'язкові елементи наводяться у прямих дужках.
  Наприклад, `|`**`else`**` ...|;`
- Частина, яка повинна бути представлена певним елементом мови Pascal, але у форматі наводиться українською мовою, виводиться у кутових дужках.
  Наприклад, замість `<оператор>`, можна написати будь-який оператор Pascal.

1. Прості і структуровані оператори

Основна частина pascal-програми є послідовністю операторів, кожен з яких виконує певні дії над даними. Оператори виконуються послідовно у тому порядку, в якому вони записані в тексті програми. Розділяються оператори крапкою з комою. Всі оператори поділяються на дві групи: прості і структуровані.

1.1. Прості оператори

Оператори, які не містять в собі ніяких інших операторів, називаються **простими**. До цих операторів належать оператори присвоювання, безумовного переходу, викликання процедури і пустий оператор.

Оператор присвоювання :=
Вираховує вираз, який задано у правій частині оператора, і результат надає змінній, ідентифікатор якої розташований у лівій частині. Змінна і вираз повинні мати сумісні типи.

Формат:

```
<ідентифікатор> := <вираз>;
```

Приклади:

```
Sort := 1;
suma := a+b;
Cena := 12.20;
name := 'nick';
```

Оператор безумовного переходу goto
Реалізує команду переходу до певної частини програми, яка помічена міткою. У відповідності до принципів структурного програмування міток повинно бути якомога менше (краще взагалі уникати використання міток і операторів безумовного переходу). Як виняток можна використовувати цей оператор для примусового завершення операторів повторювання.

Формат:
```
goto <мітка>;
```

Приклад:
```
goto m1;
```

При використанні цього оператора слід пам'ятати про таке:

- мітка, на яку передається управління, повинна бути описаною у розділі описування міток тієї програми, процедури, або функції, де вона використовується;
- областю дії мітки є той блок, у якому вона описана і перехід на неї можливий лише у межах блоку;
- спроба вийти за межі блоку або увійти до підлеглого блоку призведе до помилки.

Оператор викликання процедури
Служить для активізації попередньо визначеної користувачем або стандартної процедури (див. тема 13).

Формат:

```
<ім'я процедури>|(<список фактичних параметрів>)|;
```

Пустий оператор
Не містить ніяких символів і не виконує ніякі дії. Може бути розташований у будь-якому місці програми, де синтаксис програми припускає наявність оператора. Фактично використання пустого оператора знімає певні обмеження на використання зайвих крапки з комою, наприклад перед оператором `end`.

Закріпити використання простих операторів ви зможете виконавши лабораторні роботи №2 і №3.

Прості оператори дозволяють реалізувати лінійний алгоритм виконання програми. Далі ми

розглянемо приклади кількох програм, які реалізують лінійний алгоритм.

Приклад 12.1. Введення виведення даних

Завдання: Створити програму, яка буде запитувати у користувача його ім'я і виводити персональне привітання.

Рішення: Для реалізації програми визначимо константу `Vitanya` з текстом привітання, змінну `Name`, у якій буде зберігатися введене з клавіатури ім'я. Вирішення задачі буде відбуватися у три етапи: спочатку треба вивести на екран текст запиту; далі слід організувати введення з клавіатури імені користувача; наприкінці слід вивести на екран персональне привітання.

Текст програми:

```
{Програма введення-виведення}
Program Pr12_01;
const {Опис константи із текстом привітання}
  Vitanya='Вітаю Вас, ';
var   {Описування рядкової змінної}
  Name:string;
BEGIN {Початок блоку операторів}
  write(' Як Вас звуть? ');   { Виведення на екран запиту }
  readln(Name);{Зчитування з клавіатури імені користувача }
  writeln(Vitanya,Name,'!');{Виведення на екран привітання}
  readln;
END.{Завершення блоку операторів}
```

Результат виконання програми
(напівжирним шрифтом виділені дані, які уводяться з клавіатури):

```
Як Вас звуть? Віталій
Вітаю Вас, Віталій!
```

Перегляньте роботу готової програми.

Приклад 12.2. Лінійний алгоритм. Числові дані

Завдання: Знайти площу трапеції, якщо відомі дві основи трапеції та її висота.

Рішення: При вирішенні задачі слід спочатку згадати формулу для обчислення площі трапеції. Формула має вигляд: $S = h*(a+b)/2$. Для реалізації програми слід на першому етапі увести з клавіатури дійсні значення двох основ і висоти трапеції. Далі слід за допомогою операції присвоювання знайти площу трапеції і наприкінці вивести знайдений результат на екран. Зазначимо, що для коректної роботи програми для основ, висоти і площі трапеції слід обрати дійсні змінні

Текст програми:

```
Program Pr12_02;
var
  a,b,h,s:real;{ Описування змінних }
BEGIN
  { Етап 1. Введення даних з клавіатури }
  write('Нижня основа трапеції  (мм) = ');readln(a);
  write('Верхня основа трапеції (мм) = ');readln(b);
```

```
  write('Висота трапеції       (мм) = ');readln(h);
  { Етап 2. Розрахунок площі }
  s := h*(a+b)/2;
  { Етап 3. Виведення результату на екран }
  writeln('Площа трапеції = ',s:8:2,' кв.мм');
  readln;
END.
```

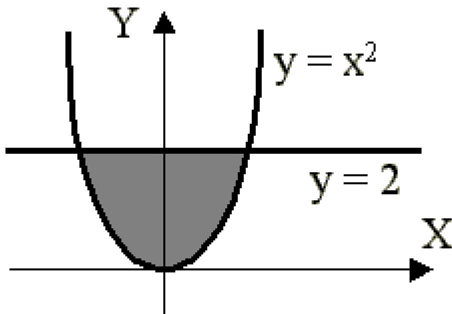
Результат виконання програми:

```
Нижня основа трапеції (мм) = 23
Верхня основа трапеції (мм) = 35
Висота трапеції (мм) = 10
Площа трапеції = 290.00 кв.мм
```

Перегляньте роботу готової програми.

Приклад 12.3. Лінійний алгоритм. Числові дані - логічний результат

Завдання: З'ясувати, чи потрапляє точка з координатами (X,Y) до заштрихованої зони, яка зображена на рисунку. Результат присвоїти логічній змінній Target.



Рішення: Алгоритм вирішення складається із трьох етапів. На першому етапі слід увести дійсні змінні X і Y, у яких будуть зберігатися координати точки. На другому етапі слід визначити логічне значення: потрапляє чи ні точка з координатами (X,Y) до заштрихованої зони. Тут слід звернути увагу на те, що для параболи умова потрапляння до заштрихованої зони буде мати такий вигляд: $y>=x^2$, а для прямої:- y<=2. Об'єднавши ці умови за допомогою логічної операції AND ми отримаємо рішення. На третьому етапі залишається лише вивести логічний результат (true або false) на екран.
Зверніть увагу на те, що ця програма вирішена без застосування оператора перевірки умови. Це зроблено з метою показати, що логічні дані можна обробляти так само, як і числа.

Текст програми:

```
Program Pr12_03;
var
  x,y:real;        { Описування змінних }
  Target:boolean;
BEGIN
  { Етап 1. Введення координат з клавіатури }
  write('X = ');readln(x);
  write('Y = ');readln(y);
  { Етап 2. Розрахунок значення }
  Target := (y<=sqr(x)) and (y<=2);
  { Етап 3. Виведення результату на екран }
  writeln('Умова потрапляння: ',Target);
```

```
  readln;
END.
```

Результати виконання програми:

```
X = 1
Y = 1
Умова потрапляння: TRUE

X = 1.5
Y = 0.8
Умова потрапляння: FALSE
```

Перегляньте роботу готової програми.

Приклад 12.4. Лінійний алгоритм. Символьні дані - логічний результат

Завдання: З клавіатури уводиться символ. Вважається, що він може бути представленням цілого числа у шістнадцятковій формі. З'ясувати так це чи ні. Результат присвоїти логічній змінній `Hex`.

Рішення: Алгоритм вирішення задачі складається із трьох етапів. На першому етапі слід увести з клавіатури символ. На другому етапі слід визначити логічне значення: потрапляє чи ні уведений символ до переліку символів, з яких може складатися шітнадцяткове число. Найпростішим, але не найкращим способом, було б перелічування усіх 16-ти можливих значень шістнадцяткового числа. Кращим рішенням буде спроба розрахувати логічну умову через порівняння символу з граничними значеннями і подальшого об'єднання умови за допомогою логічних операції. На третьому етапі залишається лише вивести логічний результат (`true` або `false`) на екран.
Зверніть увагу на те, що як і у попередньому прикладі, ця програма вирішена без застосування оператора перевірки умови.

Текст програми:

```
Program Pr12_04;
var
  Ch:char;        { Описування змінних }
  Hex:boolean;
BEGIN
  { Етап 1. Введення символу з клавіатури }
  write('Уведіть символ: ');readln(Ch);
  { Етап 2. Розрахунок значення }
  Hex := ((Ch>='0') and (Ch<='9')) or
         ((Ch>='A') and (Ch<='F'));
  { Етап 3. Виведення результату на екран }
  writeln('Умова приналежності: ',Hex);
  readln;
END.
```

Результати виконання програми:

```
Уведіть символ: F
Умова приналежності: TRUE

Уведіть символ: R
Умова приналежності: FALSE
```

Перегляньте роботу готової програми.

1.2. Структуровані оператори

Такі оператори являють собою структури, які побудовані з інших операторів за визначеними правилами. Всі структуровані оператори поділяються на три групи: складені, перевірки умови і повторювання.

Складені оператори
Складені оператори являють собою групу з будь-якої кількості операторів, розділених крапкою з комою і обмежених операторними дужками `begin` і `end`. Складений оператор сприймається як єдине ціле і може знаходитись у будь-якому місці програми.

Формат:

```
begin
   <оператор>;
   ...
   <оператор>
end;
```

Приклад:

```
begin
   A := A*B+(N-2*D);
   Rez := A+B*Pi;
   writeln(Rez:12:5)
end;
```

2. Оператори перевірки умови

Забезпечують виконання або невиконання оператора, групи операторів або блоку в залежності від логічної умови. Pascal дозволяє використання двох операторів перевірки умови: **`if`** і **`case`**.

2.1 Оператор перевірки умови `IF`

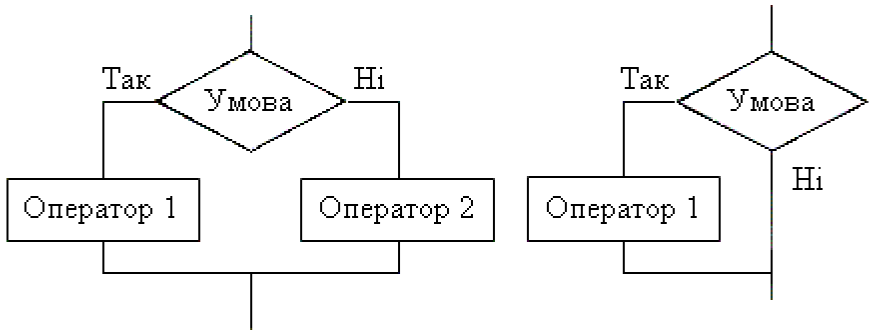
Цей оператор є найбільш популярним засобом, який дозволяє змінити лінійний порядок виконання програми і реалізує розгалужений алгоритм.

Формат:

```
if <умова>
  then <оператор 1>
 |else <оператор 2>|;
```

Слова `if, then, else` перекладаються так: "якщо", "тоді", "інакше".

Оператор `if` реалізує такі два варіанти алгоритмів:



Умова - це логічна змінна, простий або складний логічний вираз. Складні умови утворюються за допомогою логічних операції. У тому випадку, **якщо** *умова* є вірною (має логічне значення `true`), **тоді** виконується *оператор 1*, **в іншому випадку** - виконується *оператор 2*. Оператор перевірки умови може мати спрощений формат, коли частина `else` відсутня. Зверніть увагу на те, що оператор перевірки умови є суцільним, тому категорично забороняється ставити крапку з комою перед `else`.

Приклади:

```
if Um then writeln('Умова вірна')
      else writeln('Умова невірна');

if (A<B) and (B<C)
  then writeln ('Число A - найменше');
```

У тому випадку, коли при виконанні або невиконанні умови треба виконати не один а кілька операторів, слід використовувати складений оператор, оточуючи усе потрібне в операторні дужки `begin...end`.

Приклад:

```
if (X>0)
then begin
  z:=x-y; b:= k-z end
else begin
  z:=2*x; b:=z-k end ;
```

При використанні оператора перевірки умови досить часто виникають випадки, коли в середину одного оператора треба вкласти інший оператор. У такому випадку слід уважно дивитися, до якого оператора `if` відноситься та чи інша частина `then` або `else`. Тут як ніколи рекомендується чітко формувати запис програми із відступами і пам'ятати про те, що слово `else` має відношення до найближчого оператора `if`.

Приклад:

```
if U1
then if U2
     then writeln('U1 і U2 - виконуються')
     else writeln('U1 - виконується, U2 -ні')
else if U3
     then writeln('U3 - виконується, U1 -ні')
     else writeln('U1 і U3 - не виконуються');
```

Ще раз зверніть увагу на те, що крапку з комою ми поставили тільки наприкінці всього складеного оператора перевірки умови.

Закріпити використання оператора `if` ви зможете виконавши лабораторну роботу №4.

2.2. Оператор варіанту case

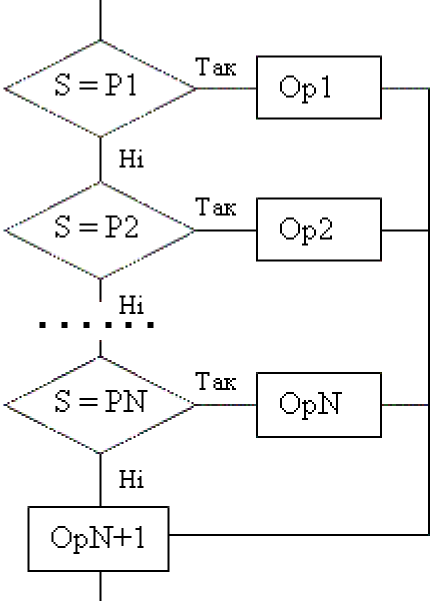
Оператор варіанту `case` є узагальненням оператора `if` з певними обмеженнями. Він дозволяє зробити вибір з великої кількості варіантів можливих значень.

Формат:

```
case <селектор> of
     <список 1>:<оператор 1>;
     <список 2>:<оператор 1>;
     . . . . . . . . .
     <список N>:<оператор N>
     |else      <оператор N+1>|
end;
```

Слово `case` перекладається як "випадок". *Селектором* може бути змінна або вираз будь-якого скалярного типу за винятком дійсного. Списки визначають перелік значень, яким може дорівнювати селектор. Вони складаються з будь-якої кількості констант або діапазонів значень, розділених між собою комами. Типи значень у списках повинні бути сумісними із типом даних селектора. Зверніть також увагу на те, що оператор `case` має свій власний `end`, тобто в цьому випадку ми зустрічаємо `end` без відповідного йому `begin`.

Оператор `case` реалізує такий алгоритм:



Оператор `case` працює таким чином. Спочатку вираховується значення селектора. Зауважимо, що краще це зробити до оператора `case`. Далі послідовно перевіряється умова, чи потрапляє значення селектора до списків 1, 2, 3 і т.д. Знайшовши перший збіг, компілятор буде виконувати той оператор, який вказаний після відповідного списку. Якщо компілятор не знайде жодного збігу, а в операторі `case` присутня необов'язкова частина `else`, тоді буде виконаний оператор, що слідує за нею. У випадку відсутності частини `else`, і знову ж таки, коли значення селектора не співпало з будь-яким списком, жоден з операторів не буде виконаний.
Слід уникати випадків, коли у списках будуть однакові елементи або діапазони, оскільки компілятор виконує тільки один оператор, який знайде першим .

Приклад:

```
case Sel of
     1,2,3:writeln('Числа 1, 2 або 3');{список-перелік}
     4..20:writeln('Числа від 4 до 20');{список-діапазон}
        21:writeln('Число 21');{список-константа}
     else :writeln('Число <1 і >21');{частина else}
end;{завершення оператора case}
```

Закріпити використання оператора `case` ви зможете виконавши лабораторну роботу №5.

Розглянемо кілька прикладів програм, які реалізують розгалужені алгоритми за допомогою операторів `if` і `case`.

Приклад 12.5. Розгалужений алгоритм. Простий оператор IF

Завдання: Написати програму для розрахунку опору електричного ланцюга, який містить два резистори, що можуть бути з'єднані послідовно або паралельно.

Рішення: Перед вирішенням задачі слід згадати, що загальний опір ланцюга, в залежності від типу з'єднання, розраховується по різним формулам. Для послідовного з'єднання: `Rs=R1+R2`; для паралельного з'єднання: `Rs=(R1*R2)/(R1+R2)`. Для вирішення задачі потрібно спочатку організувати введення з клавіатури значень опорів резисторів R1 і R2 (дійсні змінні), а також типу з'єднання (ціла змінна TZ). Результат буде присвоєний дійсній змінній Rs, значення якої буде розраховуватися за різними залежностями. Наприкінці буде виведено розраховане значення.

Текст програми:

```
program Pr12_05;
var
  R1,R2:real;{Величини опору резисторів}
  Rs   :real;{Опір ланцюга}
  TZ   :byte;{Тип з"єднання}
BEGIN
  {Етап 1. Введення даних}
  write('Опір першого резистора (Ом) = ');readln(R1);
  write('Опір другого резистора (Ом) = ');readln(R2);
  write('Тип з"єднання (1-послідовне, 2-паралельне)');
  readln(TZ);
  {Етап 2. Розрахунок}
  if TZ = 1
    then Rs:=R1+R2
    else Rs:=R1*R2/(R1+R2);
  {Виведення результату}
  writeln('Опір ланцюга = ',Rs:6:2,' Ом');
  readln;
END.
```

Результат виконання програми:

```
Опір першого резистора (Ом) = 12
Опір другого резистора (Ом) = 18
Тип з"єднання (1-послідовне, 2-паралельне) 2
Опір ланцюга = 7.20 Ом
```

Перегляньте роботу готової програми.

Приклад 12.6. Розгалужений алгоритм. Складений оператор IF

Завдання: Написати програму для визначення найбільшого з трьох дійсних чисел значення яких уводяться з клавіатури.

Рішення: Цю задачу можна вирішити різними способами, і тут для порівняння будуть наведені три таких способи. Вони реалізують різні алгоритми але розраховують однаковий результат. У першому випадку ми маємо один головний оператор `if` і два підлеглі оператори, які вкладені до частин `then` і `else` головного. У другому випадку є один головний і один підлеглий оператор (вкладений у частину `else`). У третьому випадку ми маємо три окремих оператори а результат отримуємо завдяки складним умовам, у яких використовуються логічні операції.

Текст програми 1:

```
program Pr12_061;
var
  A,B,C:real;
BEGIN
  write(' A = ');readln(A);
  write(' B = ');readln(B);
  write(' C = ');readln(C);
  if A>B
  then if A>C
       then writeln('Число A - найбільше.')
       else writeln('Число C - найбільше.')
  else if B>C
       then writeln('Число B - найбільше.')
       else writeln('Число C - найбільше.');
  readln;
END.
```

Перегляньте роботу готової програми 1.

Текст програми 2:

```
program Pr12_062;
var
  A,B,C:real;
BEGIN
  write(' A = ');readln(A);
  write(' B = ');readln(B);
  write(' C = ');readln(C);
  if (A>B)and(A>C)
  then writeln('Число A - найбільше.')
  else if B>C
       then writeln('Число B - найбільше.')
       else writeln('Число C - найбільше.');
  readln;
END.
```

Перегляньте роботу готової програми 2.

Текст програми 3:

```
program Pr12_063;
var
  A,B,C:real;
BEGIN
```

```
  write(' A = ');readln(A);
  write(' B = ');readln(B);
  write(' C = ');readln(C);
  if (A>B)and(A>C)then writeln('Число A - найбільше.');
  if (B>A)and(B>C)then writeln('Число B - найбільше.');
  if (C>B)and(C>A)then writeln('Число C - найбільше.');
  readln;
END.
```

Перегляньте роботу готової програми 3.

Результат виконання програм 1, 2, 3:

```
A = 12.4
B = -1.8
C = 21
Число C - найбільше.
```

Приклад 12.7. Розгалужений алгоритм. Оператор case

Завдання: Написати програму для виведення пори року в залежності від номера місяця.

Рішення: Для вирішення цієї задачі зручно використовувати оператор `case` що дозволяє уникнути підвищеної складності запису операторами `if`. Алгоритм складається з двох етапів. На першому етапі з клавіатури вводиться номер місяця, а на другому - аналізується номер місяця і одразу виводиться пора року до якої він відноситься. У випадку, якщо номер уведений невірно, на екран виводиться відповідне повідомлення.

Текст програми:

```
Program Pr12_07;
var
  Month:byte;{ Номер місяця }
BEGIN
  write('Введіть номер місяця: ');readln(Month);
  case Month of
    12,1,2:writeln('Зима');
    3..5  :writeln('Весна');
    6..8  :writeln('Літо');
    9..11 :writeln('Осінь')
    else writeln('Невірний номер місяця.')
  end;{case}
  readln;
END.
```

Результати виконання програми:

```
Введіть номер місяця = 4
Весна

Введіть номер місяця = 13
Невірний номер місяця.
```

Перегляньте роботу готової програми.

3. Оператори повторювання

Оператори повторювання використовуються при організації циклів. **Цикл** - це послідовність операторів, яка може виконуватись більше одного разу. Якщо кількість повторень відома заздалегідь, використовується оператор `for`, якщо кількість повторів невідома - використовуються оператори `repeat` або `while`.

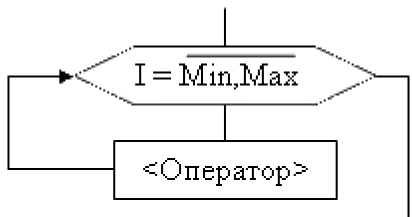
3.1. Оператор циклу for

Оператор `for` використовується для організації циклів з фіксованою кількістю повторювань і може бути представленим у двох форматах:

```
for <параметр циклу> := <PMin> to <PMax> do <оператор>;
for <параметр циклу> := <PMax> downto <PMin> do <оператор>;
```

Параметр циклу - змінна одного зі скалярних типів за винятком дійсного. Категорично забороняється впливати на значення параметра циклу (примусово змінювати значення відповідної змінної) у середині циклу.
Pmin, Pmax - граничні значення (початкове і кінцеве) параметру циклу. Саме ними визначається кількість повторювань циклу. Граничними значеннями можуть бути константи, змінні або вирази відповідних типів (без використання функцій), але обов'язково їхні значення повинні бути визначені до початку роботи циклу.
`to, downto` - визначають напрямок зміни параметру циклу при кожному повторюванні. Причому `to` визначає збільшення на 1, а `downto` - зменшення на 1.
Оператор - простий або складений оператор, який називають тілом циклу.
Сам оператор циклу можна прочитати так:
Для (`for`) параметру циклу, який змінюється від початкового (`PMin`) до кінцевого (`PMax`) значення з кроком одиниця робити (`do`) таке (`оператор`).

Оператор циклу `for` реалізує такий алгоритм.



Приклади операторів циклу:

```
for i := 1 to 10 do write('*');
```

Тут `i` - параметр циклу, що збільшується з кожним повторюванням на 1 і змінює своє значення від 1 (мінімальне значення) до 10 (максимальне значення). Граничні значення вказані як константи.

Наступний приклад виводить на екран 10 зірочок.

```
M1:= 10;
M2 := M1 div 10;
for k := M1 downto M2 do begin
  Sound(k*200); Delay(2000);
  NoSound;      Delay(2000)
end;
```

Тут M1 і M2 - цілочислові змінні, причому перша із них отримує стале значення, а друга вираховується через вираз. Параметр циклу k змінюється від 10 до 1 з кроком -1 (на це вказує зарезервоване слово downto). Тілом циклу є складений оператор, який оточений у операторні дужки begin...end; Наведений фрагмент програми виводить на внутрішній динамік системного блоку ПК звукові сигнали частотою від 2 КГц до 200 Гц протягом 2 с (2000 мс) кожний із паузою між ними 2 с. Зверніть увагу на те, що граничні значення параметру циклу задаються як змінні.

Дозволяється також вказувати граничні значення як вирази, тип яких відповідає типові параметру циклу, наприклад:

```
for j :=N+1 to N*2 do ...
```

Допускається і досить часто використовується вкладання всередину одного оператора циклу іншого. У такому випадку кажуть про вкладені оператори циклу і параметр циклу внутрішнього циклу буде стільки разів змінюватися від початкового до кінцевого значення, скільки це вказано граничними значеннями у зовнішньому операторі циклу. Причому граничні значення внутрішнього оператора циклу можна зв'язати із поточним значенням зовнішнього оператору циклу (зворотній зв'язок заборонений). Варіант такого підходу дивись у прикладі 12.9.

У випадку, коли треба передчасно перервати роботу оператора циклу, наприклад, коли перебираючи елементи масиву знайдено елемент, який відшукувався, слід скористатись процедурою Break або Exit. Не слід штучно створювати мітки для того щоб вийти або, ще гірше, - не можна намагатися примусово змінити значення параметру циклу. Наведені далі приклади програм ілюструють механізм передчасного завершення циклу. Зверніть лише увагу на те, що процедура Break завершує роботу тільки того циклу, в тілі якого вона знаходиться, а процедура Exit виходить з усіх вкладених один до одного циклів (стрілки вказують напрямки переходів.

```
for i := IMin to IMAx do begin
   {Оператори циклу по i}

   for j:= JMin to JMax do begin
      {Оператори циклу по j}
      if <умова> then break;
      {Оператори циклу по j}
   end; {Завершення циклу по j}

   {Оператори циклу по i}
end;{Завершення циклу по i}

{Продовження основної програми}
```

```
for i := IMin to IMAx do begin
   {Оператори циклу по i}

   for j:= JMin to JMax do begin
      {Оператори циклу по j}
      if <умова> then exit;
      {Оператори циклу по j}
   end; {Завершення циклу по j}

   {Оператори циклу по i}
end;{Завершення циклу по i}

{Продовження основної програми}
```

Закріпити використання оператора `for` ви зможете виконавши лабораторну роботу №6.

Дуже часто оператори циклу із фіксованою кількістю повторювань використовуються для виведення таблиць (див. приклад далі). Розглянемо декілька прикладів використання операторів циклу `for`.

Приклад 12.8. Циклічний алгоритм. Оператор FOR

Завдання: Написати програму для виведення на екран таблицю квадратів і кубів цілих чисел в межах від -5 до 5.

Рішення: Вирішення цієї задачі здійснюється у два етапи. На першому етапі виводиться заголовок таблиці, а на другому - одним оператором циклу виводиться: значення базового числа, його квадрат та його куб. Усі значення визначаються як константи, тому в програмі відсутня частина введення початкових даних з клавіатури.

Текст програми:

```
Program Pr12_08;
var i:integer; {параметр циклу}
BEGIN
  writeln('   x     x^2    x^3');
  for i := -5 to 5 do
    writeln(i:6,i*i:6,i*i*i:6);
  readln;
END.
```

Результати виконання програми:

```
     x   x^2    x^3
    -5    25   -125
    -4    16    -64
    -3     9    -27
    -2     4     -8
    -1     1     -1
     0     0      0
     1     1      1
     2     4      8
     3     9     27
     4    16     64
```

```
  5   25   125
```

Перегляньте роботу готової програми.

Приклад 12.9. Циклічний алгоритм. Оператор FOR

Завдання: Написати програму для виведення на екран зірочками рівнобічного прямокутного трикутника. Довжина катета (у кількості зірочок) вводиться користувачем.

Рішення: Оскільки у завданні нічого не вказано про орієнтацію трикутника, який треба вивести, то програма може мати багато варіантів розв'язку, які пов'язані саме із орієнтацією. Далі наведено три варіанти програми. Перший і другий варіанти мало різняться між собою і демонструють лише особливості використання оператору повторювання у випадку збільшення або зменшення значень параметрів циклу. При цьому вирішення задачі складається з чотирьох етапів: введення довжини катету; виведення зірочки (для другої програми - всього катета); виведення "тіла" трикутника (три оператори, один із яких є вкладеним оператором циклу); виведення катету (для другої програми - останньої зірочки). Третя програма більшеі відрізняється від двох попередніх, оскільки вимагає введення додаткового етапу (див. етап 2), коли розраховується довжина гіпотенузи. Виведення "тіла" трикутника складається вже з чотирьох етапів, два з яких є вкладеними циклами.
В усіх прикладах, а особливо у третьому, активно використовуються вирази для обчислення граничних значень параметрів циклу.

Текст програми 1:

```
Program Pr12_09_1; {1 варіант. Катет знизу}
var
  i,j,       {параметри циклу}
  k:integer; {довжина катета}
BEGIN
  {Етап 1. Введення довжини катетів}
  write('Уведіть довжину катета від 2 до 20');
  readln(k);
  {Етап 2. Виведення на екран першої зірочки}
  writeln('*');
  {Етап 3. Виведення на екран "тіла" трикутника}
  for i := 2 to k-1 do begin
    write('*');
    for j := 2 to i-1 do
      write(' ');
    writeln('*');
  end;
  {Етап 4. Виведення на екран катету трикутника}
  for i:= 1 to k do
    write('*');
  readln;
END.
```

Перегляньте роботу готової програми 1.

Текст програми 2:

```
Program Pr12_09_2;{2 варіант. Катет зверху}
var
  i,j,       {параметри циклу}
  k:integer; {довжина катета}
BEGIN
  {Етап 1. Введення довжини катетів}
```

```
  write('Уведіть довжину катета від 2 до 20');
  readln(k);
  {Етап 4. Виведення на екран катету трикутника}
  for i:= 1 to k do
    write('*');
  writeln;
  {Етап 3. Виведення на екран "тіла" трикутника}
  for i := k-1 downto 2 do begin
    write('*');
    for j := i-1 downto 2 do
      write(' ');
    writeln('*');
  end;
  {Етап 4. Виведення на екран останньої зірочки}
  writeln('*');
  readln;
END.
```

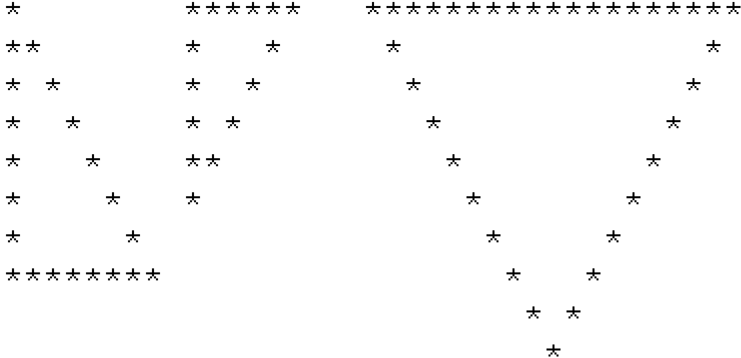
Перегляньте роботу готової програми 2.

Текст програми 3:

```
Program Pr12_09_3;{3 варіант. Гіпотенуза зверху}
var
  i,j,       {параметри циклу}
  k:integer; {довжина катета}
  g:integer; {довжина гіпотенузи}
BEGIN
  {Етап 1. Введення довжини катетів}
  write('Уведіть довжину катета від 2 до 10');
  readln(k);
  {Етап 2. Розрахунок довжини гіпотенузи}
  g:=k*2-1;
  {Етап 2. Виведення на екран гіпотенузи трикутника}
  for i:= 1 to g do
    write('*');
  writeln;
  {Етап 3. Виведення на екран "тіла" трикутника}
  for i := k-1 downto 2 do begin
    for j := 1 to k-i do
      write(' ');
    write('*');
    for j := (i-1)*2 downto 2 do
      write(' ');
    writeln('*');
  end;
  {Етап 4. Виведення на екран останньої зірочки}
  for i:= 1 to k-1 do
    write(' ');
  writeln('*');
  readln;
END.
```

Перегляньте роботу готової програми 3.

Результати виконання програм 1, 2 і 3, відповідно для довжин катетів 8, 6 і 10:



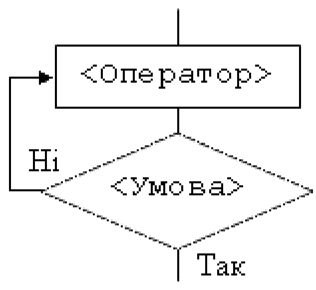
3.2. Оператор циклу `Repeat...Until`

Оператор `repeat...until` використовується для організації циклів з наперед невідомою кількістю повторювань і може бути представленим у такому форматі:

```
repeat
  <оператор>;
  . . . .
  <оператор>
untlil <умова завершення>;
```

`Repeat` перекладається як "повторювати", а `until` - "доки не стане"
Умова завершення - логічна змінна, функція або вираз, які визначають за якої умови виконання циклу буде завершене. Оператори всередині називають тілом циклу. Зверніть увагу на те, що додатково ставити операторні дужки `begin...end;` немає потреби. Тіло оператора виконується до того, поки умова завершення не прийме значення `true`. Крім того, тіло циклу виконується принаймні один раз, причому, якщо умова одразу приймає значення `true`, все рівно усі оператори будуть один раз виконані. Для коректного завершення роботи цього оператора слід щоб всередині тіла знаходився оператор, який впливає на значення умови, або всередині самої умови була функція, яка здатна змінити її значення. У іншому випадку ви ризикуєте отримати нескінченний цикл, перервати роботу якого можна буде тільки аварійно.

Оператор циклу `repeat...until` реалізує такий алгоритм.



Приклади операторів циклу.

```
D:=1; S:=0;
repeat
  S:=S+D;
until (D>100);
{Нескінченний цикл,
оскільки немає
зміни значення D}
```

```
D:=1; S:=0;
repeat
  S:=S+D;
  inc(D);
until (D>100);
{Цикл буде завершений, оскільки
у тілі є вплив на змінну D}
```

Приклад зліва є помилковим, оскільки в умові завершення фігурує змінна `D`, яка при початковому значенні "1" визначає значення умови завершення `false`. Оскільки у тілі циклу на цю змінну немає ніякого впливу, то цикл не завершиться ніколи. Приклад праворуч є коректним, оскільки процедура `inc(D)` при кожному повторюванні збільшує значення змінної `D` на "1".

```
repeat until KeyPressed;
```

У цьому прикладі оператор циклу не має тіла, тобто він є пустим. На умову його завершення впливає значення функції, яка реагує на натискання будь-якої клавіші на клавіатурі. Тобто фактично наведений оператор є призупиненням програми до тих пір, поки користувач не підтвердить продовження програми натисканням будь-якої клавіші на клавіатурі.
Примітка. Для коректної роботи функції `KeyPressed` слід підключити програму до модуля `Crt`. Не слід зловживати такою конструкцією, оскільки незважаючи на "пустоту" тіла циклу процесор залишається завантаженим виконанням циклу, а при розташуванні кількох подібних конструкцій працює лише перша з них.

Цикл `repeat...until` використовується, як правило, для організації наближених обчислень, у задачах пошуку і обробки даних, які уводяться з клавіатури або з файла.
Закріпити використання оператора `repeat...until` ви зможете виконавши лабораторну роботу №7.
Розглянемо два приклади програм з використанням циклів `repeat...until`.

Приклад 12.10. Циклічний алгоритм. Оператор REPEAT...UNTIL

Завдання: Написати програму для обчислення суми ряду $1+1/2^2+1/3^2+1/4^2+...$ з точністю, коли додавання чергового члена буде збільшувати його на величину меншу, ніж ту, яку введе користувач з клавіатури. Вивести також на екран кількість елементів, які потрапили до суми ряду.

Рішення: Оскільки заздалегідь кількість членів ряду невідома, доцільним буде використання оператору `repeat...until`. Вихідним даним буде лише точність, значення якої треба буде увести з клавіатури (дійсна змінна `t`). Для накопичення суми введемо змінну `s`, а змінна `i` буде слугувати для розрахунку чергового члену ряду. Перед початком циклу слід ініціювати початкове значення суми - `s:=0;` і номер елемента ряду `i:=1;`. Сам цикл складається з двох операторів. У першому - шляхом додавання значення чергового члена ряду до попереднього значення накопичується сума, а у другому - номер члена ряду збільшується на 1. Умова завершення вказує на те, що припинити підсумовування слід у тому випадку, коли наступний `(i+1)` член ряду є меншим за точність `t`, яку ввів користувач.
Наприкінці залишається лише вивести на екран значення знайденої суми і кількість доданих членів із відповідними поясненнями.

Текст програми:

```
Program Pr12_10;
var
  t:real;{точність розрахунку}
  s:real;{сума ряду}
  i:longint;{номер елементу ряду}
BEGIN
  {Введення початкових даних}
  write('Уведіть точність обчислення: ');readln(t);
  {Ініціювання початкових значень}
  i:=1; s:=0;
```

```
  repeat
    s:=s+1/sqr(i); {Додавання чергового члену ряду}
    inc(i);    {Збільшення i на 1}
  until (1/(sqr(i+1)))<t; {умова завершення циклу}
  {Виведення результату}
  writeln('Сума ряду = ',s:12:9);
  writeln('Підсумовано ',i,' членів ряду');
  readln;
END.
```

Результати роботи програми:
(напівжирним шрифтом вказані дані, що уводяться із клавіатури)

```
Уведіть точність обчислення: 0.001
Сума ряду = 1.612150118
Підсумовано 31 членів ряду

Уведіть точність обчислення: 0.000005
Сума ряду = 1.642694426
Підсумовано 447 членів ряду
```

Примітка. Розглянутий приклад ілюструє класичний алгоритм знаходження будь-якої суми ряду значень.

Перегляньте роботу готової програми.

Приклад 12.11. Циклічний алгоритм. Оператор REPEAT...UNTIL

Завдання: Написати програму яка запрошує користувача увести з клавіатури ціле число в межах від 0 до 120 і продовжує свою роботу тільки у випадку, коли користувач виконав умову. В протилежному випадку слід повторювати запит і очікувати введення коректного значення.

Рішення: Значення з клавіатури повинно бути введене принаймні один раз, але у випадку помилки або багаторазових помилок слід повторювати одне і те саме невизначену кількість разів. Для вирішення задачі найкраще підійде оператор циклу `repeat...until`. У програмі визначимо цілочислову змінну `k` і реалізуємо цикл уведення її значення з контролем.

Текст програми:

```
Program Pr12_11;
var
  k:integer;
BEGIN
  repeat
    write('Уведіть ціле число в межах від 0 до 120:');
    readln(k);
  until (k>=0) and (k<=120); {умова завершення циклу}
  readln;
END.
```

Результат роботи програми:

```
Уведіть ціле число в межах від 0 до 120: 547
Уведіть ціле число в межах від 0 до 120: -23
Уведіть ціле число в межах від 0 до 120: 77
```

Примітка: запропоноване рішення не є досконалим, оскільки дозволяє аналізувати тільки

числові значення. У випадку введення символів програма буде аварійно завершувати свою роботу. Це є суттєвим недоліком. Після ознайомлення із можливостями обробки символьної інформації ми повернемося до цього завдання і вирішимо його на більш високому рівні.

Перегляньте роботу готової програми.

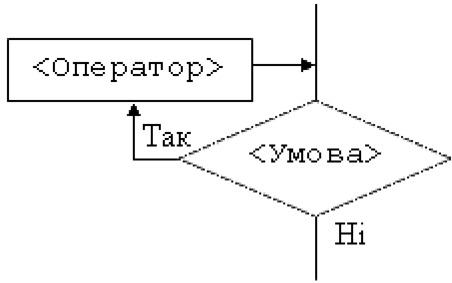
3.3. Оператор циклу While

Оператор while, так само, як і попередній оператор repeat...until, використовується для організації циклів з наперед невідомою кількістю повторювань і може бути представленим у такому форматі:

```
While <умова початку> do <оператор>;
```

While перекладається "поки", а do - "робити".
Умова початку - логічна змінна, функція або вираз, які вказують за якої умови слід починати виконувати тіло циклу (*оператор*). Оператор може бути простим або складеним. Якщо умова початку приймає значення true, оператор виконується. Завершується виконання оператора у випадку, коли умова початку приймає значення false. У випадку, коли при підході до оператора циклу умова початку одразу має значення false, тіло циклу не буде виконуватися жодного разу. Цю особливість оператора while і його відмінність від repeat...until слід враховувати. Так само слід враховувати, що для коректного завершення роботи цього оператора слід щоб всередині тіла знаходився оператор, який впливає на значення умови, або всередині самої умови була функція, яка здатна змінити її значення. У іншому випадку ви ризикуєте отримати нескінченний цикл, перервати роботу якого можна буде тільки аварійно.

Оператор while реалізує такий алгоритм:



Приклади операторів циклу:

```
D:=1; S:=0;
while (D<100) do
  S:=S+D;

{Нескінченний цикл,
оскільки немає
зміни значення D}
```

```
D:=1; S:=0;
while (D<100) do begin
  S:=S+D;
  inc(D);
end);
{Цикл буде завершений, оскільки
 у тілі є вплив на змінну D}
```

Наведений приклад навмисно дуже схожий на приклад у операторі repeat...until, адже ці два оператори є дуже схожими. Порівняйте їх, знайдіть відмінності та особливості.

```
while not KeyPressed do write('*');
```

У цьому прикладі на екран будуть виводитись зірочки до тих пір, поки користувач не натисне

на будь-яку клавішу.

Цикл `while` так саме, як і цикл `repeat...until` використовується для організації наближених обчислень, у задачах пошуку і обробки даних, які уводяться з клавіатури або з файла. Закріпити використання оператора `while` ви зможете виконавши лабораторну роботу №7. Розглянемо приклад програми з використанням циклу `while`.

Приклад 12.12. Циклічний алгоритм. Оператор WHILE

Завдання: Написати програму яка запрошує користувача увести з клавіатури ціле число і перевіряє, чи є це число простим.

Рішення: Згадаємо, що простим є число, яке ділиться без остачі на одиницю і саме на себе. Перевірити таку умову можна послідовним діленням уведеного числа на `i=2,3,4,...,n` з перевіркою остачі після кожного такого ділення. Спочатку будемо вважати, що число є простим (`Proste:=true`). Якщо після чергової спроби ми отримали остачу 0, це означає, що знайдено число `i`, на яке `n` ділиться без остачі, тобто число `n` не є простим (`Proste:=false`). У такому випадку далі перевіряти нема чого і цикл слід завершувати. В залежності від знайденого значення логічної змінної `Proste` на екран буде виведене відповідне повідомлення.

Текст програми:

```
Program Pr12_12;
var
  n,i:integer;
  proste:boolean;
BEGIN
  writeln('Уведіть ціле число:');
  readln(n);
  i:=2;  proste:=true;
  while (proste) and (i<n) do begin
     if (n mod i) = 0
       then proste:=false;
     inc(i)
  end;{while}
  if proste
    then writeln(n,' - просте число')
    else writeln(n,' не є простим числом');
  readln;
END.
```

Результати роботи програми:

```
Уведіть ціле число: 123
123 не є простим числом

Уведіть ціле число: 127
127 - просте число
```

Перегляньте роботу готової програми.

4. Контрольні запитання

1. Які оператори належать до простих?
2. Як записується і як працює оператор присвоювання?

3. Охарактеризуйте призначення і особливості оператора безумовного переходу.
4. Що таке пустий оператор і де він може використовуватись?
5. Які структуровані оператори ви знаєте?
6. Охарактеризуйте призначення і запис складеного оператора.
7. Як записується і як працює оператор перевірки умови?
8. Як працюють вкладені оператори перевірки умови?
9. Як записується і як працює оператор варіанту?
10. Які оператори повторювання ви знаєте?
11. Охарактеризуйте дві форми запису оператору циклу з фіксованою кількістю повторювань.
12. Як працюють вкладені оператори повторювання?
13. Яким чином можна достроково перервати роботу циклу?
14. Дайте приклад фрагменту програми для виведення таблиці.
15. Охарактеризуйте призначення і запис операторів циклу з післяумовою.
16. Як реалізуються алгоритми знаходження суми і добутку?
17. Охарактеризуйте призначення і запис операторів циклу з передумовою.
18. Як за допомогою операторів циклу реалізувати контроль введення даних?

Тема 13. Структуризація програм. Процедури і функції

План

1. Поняття підпрограми
2. Стандартні процедури і функції
3. Процедури і функції користувача
4. Рекурсія і випереджуюче описування
5. Контрольні запитання

У цій темі ми з вами дізнаємось про те, що для підвищення ефективності створення програм у Turbo Pascal реалізований механізм підпрограм. Такі підпрограми дозволяють більш чітко структурувати програми, зменшити її об'єм і кількість помилок. Ви дізнаєтесь про те, що підпрограми поділяються з одного боку на процедури і функції, а також на стандартні підпрограми і підпрограми користувача, причому останні вимагають обов'язкового попереднього описування у тому блоці, який їх використовує.
При розгляді стандартних підпрограм ви познайомитесь із використанням різних груп, серед яких будуть процедури управління процесом виконання програми, математичні функції, скалярні процедури і функції, функції перетворення, рядкові процедури і функції, процедури і функції динамічного розподілу пам'яті, функції для роботи із покажчиками і адресами, процедури і функції введення-виведення тощо.
Особливу увагу ми приділимо питанням створення і використання підпрограм користувача. Тут увагу слід буде звернути на особливості описування і використання процедур і функцій, на визначення і використання параметрів, на локалізацію ідентифікаторів. Наприкінці ми розглянемо питання створення рекурсивних процедур і функцій.
Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторні роботи №8 і №9.

1. Поняття підпрограми

При програмуванні нерідко бувають випадки, коли групу операторів, що реалізують певну послідовність дій, необхідно повторити без змін у кількох інших місцях програми. Зазначимо,

що мова йде не про оператори циклу, а саме про відокремлені іншими операторами групи операторів. Для того щоб уникнути нераціонального повторювання коду програми, уведений механізм **підпрограм**.

Підпрограмою називається логічно завершена група операторів, яку можна викликати для виконання по імені будь-яку кількість разів з будь-якого місця зовнішнього блоку (**блоком** ми будемо називати програми, процедури і функції у випадку, коли для викладення це є непринциповим). У Turbo Pascal за способом організації підпрограм виділяють **процедури** і **функції**.

Процедура - це незалежна пойменована частина програми, яка призначена для виконання певних дій. За своєю структурою процедура повторює програму, тобто вона складається із заголовка, розділу описування і розділу операторів (тіла процедури). Процедуру можна викликати по імені з будь-якої частини розділу операторів зовнішнього блоку. Після того, як процедура виконає дії, які реалізовані у ній, основний блок продовжить свою роботу з оператора, який слідує за оператором виклику процедури. Ім'я процедури не може входити у вираз.

Функція схожа на процедуру і багато в чому її призначення співпадає із призначенням процедури, але є дві відмінності:

- функція завжди повертає у точку свого виклику результат своєї роботи;
- ім'я функції може входити у вираз як операнд.

Усі процедури і функції Turbo Pascal розділяють на дві групи - **вбудовані** і **визначені користувачем**.
Вбудовані (стандартні) процедури і функції є частиною мови, описані в одному з модулів Turbo Pascal і їх можна використовувати без попереднього описування.
Процедури і функції користувача організовуються програмістом самостійно у відповідності до синтаксису Turbo Pascal і являють собою локальний блок, який діє тільки у тій програмі, у який він описаний (виняток становлять блоки, які описані у модулях). Попереднє описування процедур і функцій користувача обов'язкове.

2. Стандартні процедури і функції

Стандартні процедури і функції, як вже зазначалося вище, є частиною мови і описані в одному з модулів Turbo Pascal. Їх можна використовувати без попереднього описування. Можна нічого не знати про те, як у процедурі або функції реалізована та чи інша дія, треба лише знати ім'я, призначення і можливі параметри, які слід передати у процедуру або функцію. Реалізація усіх стандартних процедур і функцій зосереджена у модулях. Базові процедури і функції зосереджені у модулі System, який автоматично підключається до кожної програми. Багато інших процедур і функцій розміщується в інших модулях. Так, наприклад, процедури і функції роботи з екраном і клавіатурою у текстовому режимі зосереджені у модулі Crt, а підпрограми графічного програмування - у модулі Graph. Використання таких процедур і функцій вимагає попереднього підключення до програми відповідних модулів. У цій темі ми розглянемо лише деякі базові процедури і функції, які можна умовно розділити на такі категорії:

- Процедури управління процесом виконання програми;
- Математичні функції;
- Скалярні процедури і функції;
- Функції перетворення;
- Рядкові процедури і функції;
- Процедури і функції динамічного розподілу пам'яті;

- Функції для роботи із покажчиками і адресами;
- Процедури і функції введення-виведення;
- Інші процедури і функції.

Процедури управління процесом виконання програми

`procedure` **`Break;`**
Перериває роботу операторів циклу `for, repeat, while`. `Break` одразу завершує роботу най внутрішнього циклу, у якому містився виклик даної процедури. Якщо `Break` знаходиться не у тілі циклу, компілятор видасть повідомлення про помилку.

`procedure` **`Continue;`**
Примушує най внутрішній цикл `for, repeat, while` одразу перейти до наступної ітерації. Якщо `Continue` знаходиться не у тілі циклу, компілятор видасть повідомлення про помилку.

`procedure` **`Exit;`**
Виконує терміновий вихід із поточного блоку. Коли процедура `Exit` виконується у підпрограмі, тоді відбувається вихід із підпрограми у блок вищого рівня. Коли ця процедура виконується у операторній частині програми, вона викликає завершення роботи програми.

`procedure` **`Halt`**`|(ExitCode:word)|;`
Припиняє виконання програми і повертає управління операційній системі. `ExitCode` - необов'язковий параметр, який містить код повернення. `Halt` без параметрів відповідає виклику `Halt(0)`. Код повернення може перевірятись за допомогою функції `DosExitCode` модулю Dos, або за допомогою перевірки `ErrorLevel` у пакетному файлі Dos.

`procedure` **`RunError`**`[(ErrorCode:byte)];`
Припиняє виконання програми і генерує помилку часу виконання. `ErrorCode` - необов'язковий параметр, який містить номер помилки. Дія цієї процедури аналогічна дії процедури `Halt`, однак у доповнення вона генерує на поточному операторі помилку часу виконання, яку можна проаналізувати за допомогою налагоджувача.

Математичні функції

`function` **`Abs`**`(X);`
Повертає абсолютне значення (модуль) числа. `X` - вираз цілочислового або дійсного типу. Результат має той самий тип, що й `X`.
Приклади:
`writeln(abs(2-3));`
результат: **1**
`writeln(abs(-7.18):6:4);`
результат: **7.1800**

`function` **`ArcTan`**`(X:real):real;`
Повертає арктангенс аргументу. `X` - вираз дійсного типу. Результатом функції є головне значення арктангенсу `X` (у радіанах).
Приклади:
`writeln(arctan(1):6:4);` {обчислення арктангенсу 1, результат у радіанах}
результат: **0.7854**
`writeln(arctan(1)*180/Pi):8:4);` {обчислення арктангенсу 1, результат переводиться у градуси }

результат: **45.0000**

`function` **`Cos`**`(X:real):real;`
Обчислює косинус аргументу. `X` - величина кута у радіанах.
Приклади:
`writeln(cos(1):6:4);` {обчислення косинусу 1 радіану}
результат: **0.5403**
`writeln(cos(30*Pi/180):6:4);` {обчислення косинусу 30 градусів}
результат: **0.8660**

`function` **`Exp`**`(X:real):real;`
Повертає експоненту від аргументу. `X` - показник експоненти.
Приклади:
`writeln(exp(0):6:4);` {обчислення експоненти 0}
результат: **1.0000**
`writeln(exp(1):6:4);` {обчислення експоненти 1}
результат: **2.7183**

`function` **`Frac`**`(X:real):real;`
Повертає дробову частину аргументу. `X` - будь-який дійсний вираз.
Приклади:
`writeln(frac(11.875):6:4);`
результат: **0.8750**
`writeln(frac(-11.875):6:4);`
результат: **-0.8750**

`function` **`Int`**`(X:real):real;`
Повертає цілу частину числа. `X` - будь-який дійсний вираз. Результат також має дійсний тип.
Приклади:
`writeln(int(11.875):8:4);`
результат: **11.0000**
`writeln(int(-11.875):8:4);`
результат: **-11.0000**

`function` **`Ln`**`(X:real):real;`
Повертає натуральний логарифм аргументу. `X` - будь-який дійсний, більший за нуль вираз.
Приклади:
`writeln(ln(2.71):6:4);`
результат: **0.9969**
`writeln(ln(-1):6:4);`
результат: **помилка**

`function` **`Pi`**`;`
Повертає значення числа π (3.1415926535897932385).
Приклад:
`writeln(Pi:6:4);`
результат: **3.1416**

`function` **`Sin`**`(X:real):real;`
Обчислює синус аргументу. `X` - величина кута у радіанах.
Приклади:
`writeln(sin(1):6:4);` {обчислення синусу 1 радіану}
результат: **0.8415**

`writeln(sin(30*Pi/180):6:4);` {обчислення синусу 30 градусів}
результат: **0.5000**

`function` **`Sqr`**`(X);`
Повертає квадрат аргументу. `X` - будь-який цілочисловий або дійсний вираз. Результат має той самий тип, що й аргумент.
Приклади:
`writeln(sqr(2));` {обчислення квадрату 2 }
результат: **4**
`writeln(sqr(1.41):6:4);` {обчислення квадрату числа 1.41}
результат: **1.9881**

`function` **`Sqrt`**`(X:real):real;`
Повертає квадратний корінь додатного аргументу.
Приклади:
`writeln(sqrt(2):8:6);` {обчислення квадратного кореня з 2 }
результат: **1.414214**
`writeln(sqrt(-1):6:4);` {обчислення квадратного кореня з -1}
результат: **помилка**

Скалярні процедури і функції

`procedure` **`Dec`**`( var X|;N:longint|);`
Зменшує значення змінної `X` на `N`. При відсутності необов'язкового параметра `N` значення `X` зменшується на 1. `Dec(X)` відповідає оператору `X:=X-1;` а `Dec(X,N)` - оператору `X:=X-N`, однак процедура `Dec` породжує оптимізований код і є корисною у великих циклах.
Приклади:
`writeln(dec(10,2));` {зменшення 10 на 2 }
результат: **8**
`writeln(dec(-5));` {зменшення -5 на 1}
результат: **-6**

`procedure` **`Inc`**`( var X|;N:longint|);`
Збільшує значення змінної `X` на `N`. При відсутності необов'язкового параметра `N` значення `X` збільшується на 1. `Inc(X)` відповідає оператору `X:=X+1;` а `Inc(X,N)` - оператору `X:=X+N`, однак процедура `Inc` породжує оптимізований код і є корисною у великих циклах.
Приклади:
`writeln(inc(10,2));` {збільшення 10 на 2 }
результат: **12**
`writeln(inc(-5));` {збільшення -5 на 1}
результат: **-4**

`function` **`Odd`**`(X:longint):boolean;`
Логічна функція, яка перевіряє, чи є аргумент `X` непарним числом. Функція повертає логічний результат `true`, якщо `X` - непарне, і `false` - у протилежному випадку.
Приклади:
`writeln(odd(10));`
результат: **false**
`writeln(odd(-5));`
результат: **true**

`function` **`Pred`**`(X);`

Повертає значення, яке передує аргументові. X - вираз перелічуваного типу. Результат має той самий тип, що й X і є величиною, яка йому передує.

function **Succ**(X);
Повертає значення, яке слідує за аргументом. X - вираз перелічуваного типу. Результат має той самий тип, що й X і є величиною, яка слідує за ним.
Приклад:

```
type
  Colors = (RED,BLUE,GREEN);
begin
  Writeln('Числу 5 передує число: ', Pred(5));
  Writeln('За числом 10 слідує число: ', Succ(10));
  Writeln('Перед кольором BLUE йде колір ',Pred(BLUE),', а за ним слідує колір ',
end.
```

Результат:
Числу 5 передує число 4
За числом 10 слідує число 11
Перед кольором BLUE йде колір RED, а за ним слідує колір GREEN

Функції перетворення

function **Chr**(X:byte):char;
Повертає символ, що відповідає значенню X у кодовій таблиці ASCII.
Приклади:
writeln(chr(70));
результат: **F**
writeln(chr(102));
результат: **f**

function **Hight**(X);
Повертає найбільшу величину у діапазоні аргументу. X може бути порядкового типу, масивом або рядком. Для порядкового типу функція видає найбільшу величину із діапазону цього типу, для масивів - найбільше значення індексу для цього типу масиву, для рядків видається оголошена довжина рядка. Для "відкритих" масивів і рядків функція видає величину типу word, яка містить кількість елементів параметра мінус одиниця.
Приклад. Функція обчислення суми елементів масиву дійсних чисел.

```
function Sum(var X: array of Real): Real;
var
  I: Word;
  S: Real;
begin
  S := 0;
  for I := 0 to High(X) do S := S + X[I];
  Sum := S;
end;
```

function **Low**(X);
Повертає найменшу величину у діапазоні аргументу. X може бути порядкового типу, масивом або рядком. Для порядкового типу функція видає найменшу величину із діапазону цього типу, для масивів - найменше значення індексу для цього типу масиву, для рядків видається нуль. Для "відкритих" масивів і рядків функція видає нуль.
Приклади.

```
{Присвоєння усім елементам одновимірного масиву нульових значень}
var
  A: array[1..100] of integer;
  I: integer;
begin
  for I := Low(A) to High(A) do A[I] := 0;
end;
```

`function Ord(X):longint;`
Повертає порядковий номер для значення перелічуваного типу.
Приклади:
`writeln(ord('F'));`
результат:**70**
`writeln(ord('f'));`
результат: **102**

`function Round(X:real):longint;`
Округлює значення дійсного типу до найближчого цілого. Якщо `X` знаходиться точно посередині між двома цілими числами, тоді результатом буде число із більшим абсолютним значенням. Якщо округлення виходить за діапазон `longint`, відбувається помилка періоду виконання програми.
Приклади:
`writeln(round(15.7));`
результат: **16**
`writeln(round(-4.9));`
результат: **-5**

`function Trunc(X:real):longint;`
Відкидає дробову частину дійсного числа і перетворює його у цілочислове. Якщо результат відкидання виходить за діапазон `longint`, відбувається помилка періоду виконання програми.
Приклади:
`writeln(trunc(15.7));`
результат: **15**
`writeln(trunc(-4.9));`
результат: **-4**

Рядкові процедури і функції

До рядкових процедур і функцій відносяться: `Concat, Copy, Delete, Insert, Length, Pos, Str, Val`. Детально використання цих процедур і функцій буде розглянуто окремою темою 15.

Процедури і функції динамічного розподілу пам'яті

До процедур і функцій динамічного розподілу пам'яті відносяться: `Dispose, FreeMem, GetMem, Mark, MaxAvail, MemAvail, New`. У цій версії видання процедури і функції цього типу не розглядаються.

Функції для роботи із покажчиками і адресами

До функцій роботи із покажчиками і адресами відносяться: `Addr, Assigned, CSeg, DSeg, Ofs, Ptr, Seg, SPtr, SSeg.` У цій версії видання функції цього типу не розглядаються.

Процедури і функції введення-виведення

До процедур і функцій введення-виведення відносяться: Append, Assign. BlockRead, BlockWrite, ChDir, Close, EoF, EoLn, Erase, FilePos, FileSize, Flush, GetDir, IOResult, MkDir, Read, Readln, Rename, Reset, Rewrite, RmDir, Seek, SeekEoF, SeekEoLn, SetTextBuf, Truncate, Write, Writeln. Частина з них вже була розглянута у темі 11, а більшість буде розглянута у темі 18.

Інші процедури і функції

function **Random**|(Range: word)|;
Видає випадкове число. Range - необов'язковий параметр, який визначає діапазон випадкових чисел. Якщо він не заданий, результатом буде дійсне число у діапазоні 0<=X<1. Якщо параметр Range вказаний, результатом буде ціле число, типу word у діапазоні 0<=X<Range. Якщо Range=0, результат завжди буде нульовим. Генератор випадкових чисел завжди повинен ініціюватись процедурою Randomize, інакше при кожному наступному виклику програми буде ініціюватись одна і та сама послідовність випадкових чисел.

procedure **Randomize**;
Ініціює генератор випадкових чисел випадковою величиною, яка отримується із системного таймера. Випадкова величина зберігається у системній змінній RandSeed типу longint.

Процедури Exclude і Include розглядатимуться у темі 16. Інші процедури і функції: FillChar, Hi, Lo, Move, ParamCount, ParamStr, SizeOf, Swap, TypeOf, UpCase у цьому виданні не розглядаються.

3. Процедури і функції користувача

Незважаючи на велику кількість стандартних процедур і функцій, дуже частими є випадки, коли необхідно створити свою процедуру або функцію для вирішення локальних підзадач. Переваги такого підходу полягають у можливості багаторазового використання одного й того ж фрагменту (а це як скорочення самого коду програми, так і кількості помилок) і більш чіткій структуризації програми (таку програму легше писати і налагоджувати). У Turbo Pascal процедури і функції мають багато спільного.

3.1. Функції користувача

Функція користувача вимагає обов'язкового попереднього описування, яке повинно проводитись у розділі описування зовнішнього блоку. Тобто, якщо наша функція є вкладеною у програму - у розділі описування програми, якщо наша функція є вкладеною у іншу функцію - у розділі описування цієї функції.

Формат описування функції:

```
function  <ім'я функції> |(<список формальних параметрів>)|:<тип результату>;
  <розділ описувань>
begin
  <розділ операторів>;
  <ім'я функції>:=...;
end;
```

`<ім'я функції>` - будь-який припустимий ідентифікатор, який буде позначати вашу функцію.
`<список формальних параметрів>` - не є обов'язковим (можуть бути функції без параметрів). Якщо він присутній. то складається з переліку імен параметрів, які функція використовує для отримання даних із зовнішніх блоків. Після списку імен параметрів через двокрапку вказується тип цих параметрів. Якщо є кілька груп параметрів різного типу, вказується тип для кожної групи.
`<тип результату>` - обов'язковий елемент, який вказує на те, який результат буде мати єдине значення функції. Зазначимо, що допустимими типами результату функції є: усі скалярні типи (цілочислові, логічні, символьні, перелічувані, діапазонні); усі різновиди дійсних чисел; рядковий тип; покажчик.

Внутрішня структура функції подібна до структури програми - спочатку у ній розміщені усі описування а потім оператори. На відміну від програми, функція завершується крапкою з комою після `end`. У розділі операторів (тілі функції) обов'язково повинен бути принаймні один оператор, який імені функції присвоює значення виразу. Дуже часто такий оператор є останнім оператором функції.

Виклик функції відбувається шляхом вказівки її імені і, якщо потрібно, фактичних параметрів у складі виразу. При цьому слід дотримуватись правил сумісності типів, тобто тип результату функції повинен бути припустимим у даному місці виразу.

Створення і використання функцій користувача проілюструємо трьома прикладами.

Приклад 13.1. Арифметична функція користувача

Завдання: Створити функцію піднесення числа X у степінь Y.

Рішення: Оскільки у Pascal стандартної функції піднесення числа X у степінь Y не існує, а така задача постає досить часто, то доцільно створити свою власну функцію. Для цього слід згадати, що степінь числа можна представити через експоненту і натуральний логарифм (тобто $x^y = e^{y*\ln(x)}$), які є серед стандартних математичних функцій Turbo Pascal. Для того, щоб зробити функцію універсальною, у ній будуть присутні перевірки знаку і величини числа і степені.

Текст програми:

```
program Pr13_01;
var a,b:real;
{Описування функції}
function Stepen(X,Y:real):real;
begin
  if X>0
  then Stepen:=exp(Y*ln(X))
  else if X<0
       then Stepen:=exp(Y*ln(abs(X)))
       else if Y=0
            then Stepen:=1
            else Stepen:=0
end; {кінець описування функції}
BEGIN  {початок програми}
  write('Уведіть дійсне число: '); readln(a);      {Використання}
  write('Уведіть показник степені: ');readln(b);   {функції}
  writeln(a:8:3,' у степені ',b:8:3,' дорівнює ',Stepen(a,b):10:3);
  readln;
END.
```

Результати виконання програми:

```
Уведіть дійсне число: 2
Уведіть показник степені: 3
   2.000 у степені    3.000 дорівнює    8.000
Уведіть дійсне число: 2
Уведіть показник степені: 0
   2.000 у степені    0.000 дорівнює    1.000
```

Перегляньте роботу готової програми.

Приклад 13.2. Логічна функція користувача

Завдання: Створити логічну функцію, яка видає логічне значення `true`, якщо символ, переданий у функцію, як аргумент, є голосною латинською літерою.

Рішення: Визначимо логічну функцію з ім'ям `Golosn`, яка буде видавати логічне значення `true` лише у випадку, якщо переданий ній параметр є голосною латинською літерою. З огляду на те, що великі і малі символи є різними символами у кодовій таблиці, перелічимо кожен символ двічі. Для скорочення запису використаємо операцію приналежності `in`. Зазначимо, що можна було у функції першим оператором використати стандартну функцію `UpCase`, яка б перевела символ у верхній регістр, і вже далі аналізувати тільки великі літери. Сама програма використовує цю функцію і у випадку, якщо уведений з клавіатури символ є голосною латинською літерою, на екран виводиться відповідне повідомлення. В усіх інших випадках - введення приголосної літери, цифри, символів кирилиці - на екран ніякі повідомлення не виводяться.

Текст програми:

```
Program Pr13_02;
var c:char;
function Golosn(a:char):boolean;{описування функції}
begin
  Golosn := a in ['A','a','E','e','I','i','O','o','U','u','Y','y']
end;{кінець описування функції}
BEGIN {початок програми}
  write('Уведіть один символ'); readln(c);
  if Golosn(c)       {Використання функції}
    then writeln('Уведений символ - голосна латинська літера');
  readln;
END.
```

Результати виконання програми:

```
Уведіть один символ: u
Уведений символ - голосна латинська літера

Уведіть один символ: 2
```

Перегляньте роботу готової програми.

Приклад 13.3. Функція користувача, яка видає символьний результат

Завдання: Створити функцію, яка порівнює два цілих числа і результат порівняння видає у вигляді одного з символів: "<", ">" або "=".

Рішення: Визначимо функцію з ім'ям `Poriv`, яка за допомогою вкладених операторів

перевірки умов буде визначати символ, який відповідає співвідношенню двох аргументів. Зверніть увагу, що функції присвоюється саме символ.
У програмі вводяться два числа, які передаються у функцію як фактичні аргументи.

Текст програми:

```
Program Pr13_03;
var k,n:integer;
function Poriv(a,b:integer):char; {Описування функції}
begin
  if a<b then Poriv:='<'
         else if a<b then Poriv:='<'
                     else Poriv:='='
end;{Кінець описування функції}
BEGIN
 write('Уведіть перше число: ');readln(k);
 write('Уведіть друге число: ');readln(n);
 writeln(k,Poriv(k,n),n);
 readln;   {Використання функції}
END.
```

Результати виконання програми:

```
Уведіть перше число: 2
Уведіть друге число: 3
2<3
Уведіть перше число: -5
Уведіть друге число: -5
-5=-5
Уведіть перше число: -12
Уведіть друге число: -13
-12>-13
```

Перегляньте роботу готової програми.

Закріпити створення функцій користувача ви зможете виконавши лабораторну роботу №8.

3.2. Процедури користувача

Іноді потрібно, щоб результатом роботи підпрограми було багато однотипних або різнотипних значень, наприклад масив або запис. Також буває, що підпрограма не видає ніяких значень, а просто виконує певні дії, наприклад, виводить послідовність звукових сигналів, переводить курсор у певну позицію екрану тощо. У таких випадках зручніше використовувати **процедури**.

Процедура користувача, так саме, які і функція користувача вимагає обов'язкового попереднього описування, яке повинно проводитись у розділі описування зовнішнього блоку.

Формат описування процедури:

```
procedure  <ім'я процедури> |(<список формальних параметрів>)|;
  <розділ описувань>
begin
  <розділ операторів>;
end;
```

<ім'я процедури> - будь-який припустимий ідентифікатор, який буде позначати процедуру.

<список формальних параметрів>, якщо він присутній, складається з переліку імен параметрів, які функція використовує для отримання даних із зовнішніх блоків і які передає назад. Після списку імен параметрів через двокрапку вказується тип цих параметрів. Якщо є кілька груп параметрів різного типу, вказується тип для кожної групи. Перед параметрами, значення яких планується не тільки передавати у процедуру, а й повертати назад у зовнішній блок, слід ставити ключове слово var. Зверніть увагу на те, що після списку формальних параметрів у процедурі, на відміну від функції, не ставиться її тип.

Виклик процедури здійснюється окремим оператором. Для цього у програмі вказується ім'я процедури із списком фактичних параметрів, кількість, тип і послідовність яких повинні співпадати із оголошеними. Завершується такий оператор крапкою з комою.

Створення і використання процедур користувача проілюструємо трьома прикладами.

Приклад 13.4. Процедура користувача, яка видає кілька числових значень

Завдання: Створити процедуру обчислення площі бічної поверхні, об'єму і маси паралелепіпеда.

Рішення: Для визначення вказаних у завдання параметрів потрібно знати довжини трьох сторін паралелепіпеда (позначимо їх a,b,c у параметрах процедури) і густину матеріалу, з якого зроблений паралелепіпед (позначимо її gust). З процедури ми отримуємо площу бічної поверхні (Sb), об'єм (Vol) і масу (Masa). Останні три параметри повинні передаватись, як параметри-змінні. Самі розрахунки є простими, тому не потребують коментарів.
У програмі визначаються відповідні змінні L,B,H,G,S,V,M, перші чотири з яких передають у процедуру фактичні значення, а останні три - отримують результати розрахунку.

Текст програми:

```
Program Pr13_04;
var L,B,H,G,S,V,M:real;
{Описування процедури}
procedure Paralelepiped(a,b,c,gust:real; var Sb,Vol,Masa:real);
     {дані передаються у процедуру          видаються процедурою}
begin
  Sb:=2*(a*b+b*c+a*c);  {Обчислення площі бічної поверхні}
  Vol:=a*b*c;           {Обчислення об'єму}
  Masa:=Vol*gust;       {Обчислення маси}
end; {Кінець описування процедури}
BEGIN     {Початок програми}
  write('Уведіть довжину паралелепіпеда (см): '); readln(L);
  write('Уведіть ширину паралелепіпеда (см) : '); readln(B);
  write('Уведіть висоту паралелепіпеда (см) : '); readln(H);
  write('Уведіть густину матеріалу(г/куб.см): '); readln(G);
     {Виклик процедури з фактичними параметрами}
  Paralelepiped(L,B,H,G,S,V,M);
     {Виведення результату}
  writeln('Площа бічної поверхні = ',S:8:0,' кв.см');
  writeln('Об''єм паралелепіпеда = ',V:8:2,' куб.см');
  writeln('Маса паралелепіпеда   = ',M:8:2,' г');
  readln;
END.
```

Результат виконання програми:

```
Уведіть довжину паралелепіпеда (см): 2
Уведіть ширину паралелепіпеда (см) : 3
```

```
Уведіть висоту паралелепіпеда (см)  : 4
Уведіть густину матеріалу(г/куб.см): 7.8
Площа бічної поверхні =        52 кв.см
Об'єм паралелепіпеда  =     24.00 куб.см
Маса паралелепіпеда   =    187.20 г
```

Перегляньте роботу готової програми.

Приклад 13.5. Процедура користувача без параметрів і результатів

Завдання: Створити процедуру подачі трьох коротких звукових сигналів

Рішення: Оскільки у завданні не вказана частота звукових сигналів і їхня тривалість, а також чітко встановлена кількість, ми можемо створити процедуру з ім'ям `Sound3`. Для коректної роботи процедури, зокрема для використання процедур `Sound, Delay, NoSound` підключимо нашу програму до модулю `Crt`. Кількість звукових сигналів визначається циклом, для реалізації якого у процедурі визначена локальна змінна `i`. У прикладі число `1000` - частота звукового сигналу, а `3000` - величина паузи.
Використання процедури складається із вказівки імені процедури у тілі програми.

Текст програми:

```
Program Pr13_05;
uses Crt;     {підключення модулю Crt}
procedure Sound3; {описування процедури}
var i:integer;    {описування внутрішньої змінної}
begin
  for i :=1 to 3 do begin
      {подача звукового сигналу}
    Sound(1000); Delay(3000);
      {відключення звукового сигналу}
    NoSound; Delay(3000);
  end
end; {Кінец описування процедури}
BEGIN
  Sound3;
END.
```

Перегляньте роботу готової програми. Програма не має виведення на екран. Запустивши її ви тільки почуєте три звукових сигнали.

Приклад 13.6. Процедура користувача, яка виконує дії

Завдання: Створити процедуру для виведення на екран вказану кількість разів рядка тексту, який передається як параметр.

Рішення: Для реалізації процедури, яку ми назвемо `WriteText` слід передати у неї кількість повторювань тексту - `Num` і сам текст - `Text`. Виведення тексту буде починатися з поточної позиції курсору на екрані. Після досягнення курсором кінця екрану виведення буде продовжуватись з нового рядка.

Текст програми:

```
Program Pr13_06;
        {описування процедури}
procedure WriteText(Num:integer;Text:string);
var i:integer;  {локальна змінна}
```

```
begin
 for i:=1 to Num do
   write(Text);
end;   {кінець описування процедури}
BEGIN  {початок програми}
  WriteText(5,'Hello! '); {використання процедури}
  readln;
END.
```

Результат роботи програми:

```
Hello! Hello! Hello! Hello! Hello!
```

Перегляньте роботу готової програми.

Закріпити створення процедур користувача ви зможете виконавши лабораторну роботу №9.

3.3. Параметри процедур і функцій. Локалізація імен

Ви вже звернули увагу на те, що переважна більшість процедур і функцій використовується із параметрами. Саме вони надають підпрограмам гнучкості і дозволяють не переписуючи зміст реалізовувати різні дії. Розглянемо більш детально використання параметрів у підпрограмах.

Параметри, які вказуються у заголовку підпрограми, є ідентифікаторами змінних і слугують обміну інформацією між підпрограмою і блоком, який її викликає. Такі змінні називаються **формальними параметрами** і їх можна використовувати у підпрограмах без додаткового описування. Більше того, не можна всередині підпрограми визначати змінні з тими самими іменами, що й імена формальних параметрів. При визначенні формальних параметрів процедур існує одна особливість - можна створювати параметри, які будуть не тільки передавати значення у процедуру, а й повертати розраховані значення назад, такі параметри слід описувати із зарезервованим словом `var` (див. приклад 13.4). Зверніть також увагу на те, що параметри, описані у підпрограмах і змінні зовнішнього блоку - *це зовсім різні змінні, навіть, якщо у них і співпадають імена*.

З іншого боку, при кожному зверненні до підпрограми до неї можуть передаватись значення різних змінних. Такі змінні, імена яких підставляються в оператор виклику процедури замість формальних, називають **фактичними параметрами**. Механізм заміни параметрів діє так: кожній змінній формального параметра присвоюється те значення змінної-фактичного параметра, яке вона мала у момент виклику підпрограми. У період роботи підпрограми усі дії через імена формальних параметрів виконуються із значеннями фактичних параметрів.

Turbo Pascal дозволяє визначати формальні параметри чотирма способами, а саме через:

- параметри-значення;
- параметри-змінні;
- не типізовані параметри;
- параметри-константи.

Формальний **параметр-значення**, при викликанні підпрограми отримує своє початкове значення шляхом **копіювання** відповідного фактичного параметра. При зміні формального параметра-значення **фактичний параметр не змінюється**. Фактичне значення параметра-значення, повинно бути виразом, тип якого відповідає типові формального параметра-значення. Перед списком параметрів-значень у заголовку програми будь-яке ключове слово відсутнє, а після нього вказується тип, наприклад

```
procedure Test1(p1,p2,p3:real);
```

Формальний **параметр-змінна** використовується у тому випадку, коли значення повинно передаватися із процедури або функції у блок, який викликав її. Відповідний фактичний параметр в операторі виклику підпрограми повинен бути **посиланням на змінну**. При викликанні процедури або функції формальний параметр-змінна заміщується фактичною змінною. При будь-яких змінах значення формального параметра-змінної **зміниться і фактичний параметр**. Тип фактичного параметра повинен співпадати з типом формального параметра-змінної. У заголовку підпрограми перед списком параметрів-змінних вказується зарезервоване слово var, а після списку вказується тип, наприклад

```
function Detection(var a,b,c:byte):boolean;
```

Не типізований формальний параметр являє собою **посилання** на будь-яку змінну, незалежно від її типу. У заголовку підпрограми перед групою не типізованих параметрів ставиться ключове слово var, а після списку тип не вказується, наприклад

```
procedure Test2(var p1,p2,p3);
```

Формальний параметр-константа дозволяє передати фактичне значення не створюючи копію параметра. Значення таких параметрів не можуть змінюватись у підпрограмі, що забезпечує певну безпеку по відношенню до випадкових присвоювань значень, а також дозволяє дещо зекономити пам'ять, що може бути актуальним при передачі великого об'єму значень. У заголовку підпрограми перед списком формальних параметрів-констант вказується ключове слово `const`, а після нього вказується тип, наприклад

```
procedure Add(const X,Y:real; var A:real);
```

При викликанні підпрограм фактичні параметри можуть передаватись також по різному. Так для формальних параметрів-значень і параметрів-констант передавати фактичні параметри можна, як

- **безпосередні значення**, наприклад
  ```
  Test1(12.4,18.02,-0.7);
  Add(7.22,-3.42,Sum);
  ```
- **ідентифікатори констант**, наприклад (`x,y,z` - константи)
  ```
  Test1(x,y,z);
  Add(x,y,Sum);;
  ```
- **ідентифікатори змінних**, приклади є аналогічними до попередніх, але `x,y,z` визначені як змінні;
- **вирази**, які складаються із значень, констант і змінних, наприклад
  ```
  Test1(x+2.1,(y+x)*2,z/3);
  Add(x+z,y-z,Sum);
  ```

Для параметрів-змінних і не типізованих параметрів передавати фактичні параметри можна тільки як ідентифікатори змінних, наприклад

```
if Detection(x1,x2,x3) then Test2(x1,x2,x3);
```

Одним із найважливіших питань у використанні підпрограм є *визначення області дії ідентифікаторів*. Кожна підпрограма може мати свої внутрішні розділи описувань (див. приклади 13.5 і 13.6). Крім того вони можуть мати свої формальні параметри.
Усі описування всередині підпрограми і її формальні параметри утворюють **локальні** ідентифікатори підпрограми. Такі локальні змінні, константи, типи, вкладені підпрограми **діють тільки всередині підпрограми**. Ніякого зв'язку між ними і ідентифікаторами зовнішніх блоків, які мають збіжні імена, немає.

З іншого боку, у підпрограмі можна використовувати ідентифікатори, описані не тільки всередині, а й ті, які описані у зовнішніх блоках. Сенс і значення таких ідентифікаторів у зовнішньому блоці і у підпрограмі є однаковим. Такі ідентифікатори називають

глобальними.

Пояснимо локалізацію ідентифікаторів за допомогою схеми. зображеної на рис. 13.1.

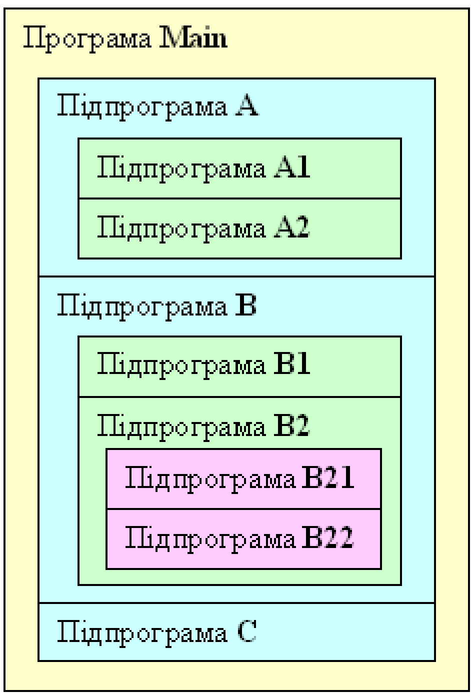


Рис. 13.1. Визначення області дії ідентифікаторів

Розглянемо ієрархічно підпорядковані блоки. Найвищий рівень має програма **Main**. Усі ідентифікатори, які описані у розділі описування цієї програми діють в усіх інших блоках. З головної програми можна викликати тільки підпрограми **A**, **B**, **C.**
Програмі **Main** підпорядковані підпрограми **A**, **B** і **C**. У цих підпрограмах діють тільки локальні ідентифікатори кожної з них і глобальні ідентифікатори програми **Main**. Тобто, підпрограми **A**, **B** і **C** нічого не знають про ідентифікатори одна одної.
Підпрограмі **A** підпорядковані дві підпрограми - **A1** і **A2**, які вона може викликати. Всередині підпрограми **A1** діють її власні ідентифікатори, ідентифікатори підпрограми **A** і ідентифікатори програми **Main**. Всередині підпрограми **A2** діють її власні ідентифікатори, ідентифікатори підпрограми **A** і ідентифікатори програми **Main**.
Підпрограмі **B** підпорядковані дві підпрограми - **B1** і **B2**, які вона може викликати. Всередині підпрограми **B1** діють її власні ідентифікатори, ідентифікатори підпрограми **B** і ідентифікатори програми **Main**. Всередині підпрограми **B2**, також, діють її власні ідентифікатори, ідентифікатори підпрограми **B** і ідентифікатори програми **Main**. Також всередині **B2** можна викликати підпорядковані ній підпрограми **B21** і **B22**.
У підпрограмах найнижчого рівня **B21** і **B22** діють їхні локальні ідентифікатори, ідентифікатори підпрограм **B2**, **B** і програми **Main**.
Зазначимо, що підпорядковані блоки одного рівня можуть викликати попередньо описані блоки. Так підпрограма **B** може викликати підпрограму **A** і не може викликати підпрограму **C.** А підпрограма **C** може посилатися і на **A** і на **B**.
Підпрограма **A2** може викликати **A1**, а **A1** викликати **A2** не може.

Допомогти запам'ятати правило для визначення області видимості може таке порівняння. Процедура нагадує автомобіль, у якого затінені вікна. Той, хто знаходиться в автомобілі (елементи підпрограми), бачить все всередині і ззовні, а той, хто знаходиться на вулиці (у зовнішньому блоці) - бачить тільки автомобіль, але не бачить, хто всередині і що там відбувається. Автомобіль, який рухається попереду (підпрограма описана раніше) добре видимий тим, хто слідує за ним (наступні підпрограми), а йому самому для того, щоб

побачити задніх потрібні дзеркала заднього виду (випереджуюче описування).

4. Рекурсія і випереджуюче описування

У відповідності до базового положення мови Turbo Pascal не можна використовувати елементи, які ще не були описані. А як же бути, коли алгоритм задачі вимагає такого підходу, адже у деяких випадках підпрограма повинна посилатися на саму себе або на підпрограму, яка описана після неї. Для реалізації такої можливості у Turbo Pascal уведено механізм **рекурсії**.
Підпрограма називається рекурсивною, якщо викликає саму себе (**пряма рекурсія**). Рекурсивною також може бути процедура, яка викликає іншу процедуру, котра, в свою чергу, викликає першу процедуру (**непряма рекурсія**). Можливими є і більш складні конструкції, коли посилання зациклюються.

При написанні програм з рекурсивними підпрограмами слід дотримуватись певних **правил безпеки**. Рекомендується компілювати програму із директивою {$S+}, яка включає перевірку переповнення стеку. Корисним буде також включення директиви {$R+}, яка активізує перевірку діапазону змінних. На початку кожної рекурсивної підпрограми можна розмістити рядок `if KeyPressed then Halt;`. У такому випадку при зацикленні рекурсії замість перезавантаження комп'ютера достатньо буде натиснути будь-яку клавішу.

Кількість рекурсивних викликів називають **глибиною рекурсії**. Глибина рекурсії повинна бути кінцевою, про що повинен потурбуватись сам програміст.

Використання непрямої рекурсії на практиці є рідшим за пряму рекурсію. У такому випадку слід організовувати **випереджуюче описування** процедури. Для цього у мові Turbo Pascal передбачена директива випереджуючого описування `forward`.
Попереднє описування складається із вказівки заголовка процедури, після якого вказується слово `forward`.
Приклад випереджуючого описування

```
Program DemoForward;
procedure A2(<формальні параметри процедури A2>); forward;
procedure A1(<формальні параметри процедури A1>);
  begin
    A2(<фактичні параметри виклику процедури A2>);
  end;
procedure A2; {список формальних параметрів не повторюється}
  begin
    A1(<фактичні параметри виклику процедури A1>);
  end;
BEGIN
  A1(<фактичні параметри виклику процедури A1>);
END.
```

Розглянемо три приклади програм із використанням прямої рекурсії.

Приклад 13.7. Пряма рекурсія

Завдання: Написати програму, яка розраховує факторіал цілого числа, значення якого вводиться з клавіатури.

Рішення: Відмітимо, що знаходження факторіала є класичною задачею, яка ефективно вирішується за допомогою рекурсії. Про факторіал відомо, що це є добуток усіх цілих чисел від 1 до числа, факторіал якого треба знайти. Для організації рекурсії скористаємось

залежністю, що `k!=k*(k-1)!`, а також тим, що при `k=1` факторіал дорівнює 1.

Текст програми:

```
Program Pr13_07;
        {описування рекурсивної функції}
function Factorial(k:integer):longint;
begin
  if k=1
    then Factorial:=1
    else Factorial:=k*Factorial(k-1)
end;                  {рекурсивний виклик функції}
var
  n:integer; {змінна, яка визначатиме основу факторіала}
BEGIN
  write('Уведіть основу факторіала: ');readln(n);
  writeln('Факторіал числа ',n,' дорівнює ',Factorial(n));
  readln;                                  {виклик функції}
END.
```

Результати роботи програми:

```
Уведіть основу факторіала: 5
120

Уведіть основу факторіала: 9
362880
```

Перегляньте роботу готової програми.

Приклад 13.8. Пряма рекурсія

Завдання: Написати програму, яка розраховує опір електричного ланцюга, схема якого наведена на рис. 13.2. Значення опору окремих резисторів R1, R2, R3, а також кількість резисторів R2 вводяться з клавіатури під час роботи програми

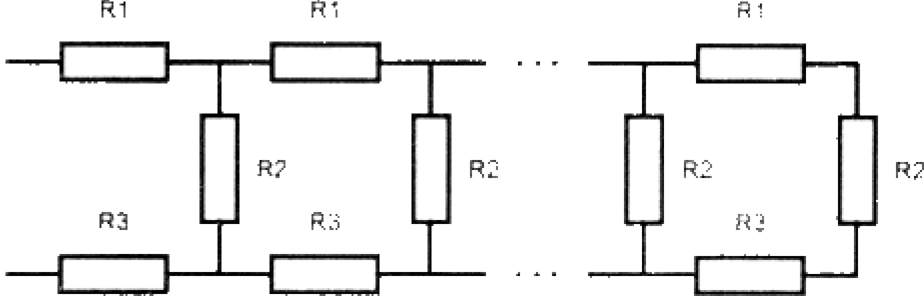


Рис. 13.2. Схема електричного ланцюга

Рішення: Скористаємось тим, що при кількості резисторів `R2=1` загальний опір ланцюга буде складати `Rs1=R1+R2+R3` (ланцюг перетворюється у послідовний). При збільшенні порядку, ланцюг перетворюється у паралельний, тому для `R2=2` буде `Rs2=R2*Rs1/(R2+Rs1)+R1+R3`, для `R2=3` буде `Rs3=R2*Rs2/(R2+Rs2)+R1+R3` і т.д.

Текст програми:

```
Program Pr13_08;
var
```

```
 rr1,rr2,rr3:real;{величини опору резисторів}
 nn:integer;      {порядок ланцюга}
        {описування функції}
function Opir(n:integer;R1,R2,R3:real):real;
var Rs:real;    {локальна змінна - опір ланцюга}
begin           {n-1-го порядку                 }
  if n=1
    then Opir:=R1+R2+R3
    else begin
      Rs:=Opir(n-1,R1,R2,R3);  {рекурсивний виклик}
      Opir:=R2*Rs/(R2+Rs)+R1+R3{остаточний опір}
    end
end; {кінець описування функції}
BEGIN
   {введення даних}
  Write('Уведіть порядок ланцюга  : ');readln(nn);
  Write('Уведіть опір резистора R1: ');readln(rr1);
  Write('Уведіть опір резистора R2: ');readln(rr2);
  Write('Уведіть опір резистора R3: ');readln(rr3);
    {виведення результату}
  writeln('Опір ланцюга = ',Opir(nn,rr1,rr2,rr3):8:2,' Ом');
  readln;                  {виклик функції}
END.
```

Результати роботи програми:

```
Уведіть порядок ланцюга  : 4
Уведіть опір резистора R1: 7.2
Уведіть опір резистора R2: 4.1
Уведіть опір резистора R3: 9
Опір ланцюга =    19.59 Ом
```

Перегляньте роботу готової програми.

Приклад 13.9. Пряма рекурсія

Завдання: Написати програму, яка по введених координатах точок вершин трикутника на площині визначає, на яку мінімальну кількість трикутників можна його розбити, щоб довжини їхніх сторін не перевищували встановлену межу (така задача може бути окремим етапом при розбитті профілів у задачах аналізу методом кінцевих елементів).

Рішення: Алгоритм вирішення цієї задачі може бути таким. На першому етапі ми за координатами точок вершин трикутника визначаємо довжини його сторін. На другому етапі, якщо хоча б одна зі сторін перевищує встановлену межу, починаємо процес розділення трикутника. Для того, щоб розділення відбувалося максимально рівномірно, будемо ділити навпіл найдовшу зі сторін. На третьому етапі знайдемо координати точки, яка лежить посередині найдовшої сторони і двічі рекурсивно викличемо процедуру аналізу трикутника. На завершальному етапі, коли занурюючись у процедуру аналізу трикутника, ми дійшли до того моменту, коли розбиття проводити вже не потрібно, слід збільшити лічильник кількості трикутників на 2, адже один трикутник розбивається на два.

Текст програми:

```
Program pr13_09;
var     { описування глобальних змінних програми }
  CountTriangle:integer;           {кількість трикутників}
  px1,py1,px2,py2,px3,py3,lg:real; {координати точок і межа}
         {загальна процедура аналізу трикутника}
procedure LLL(x1,y1,x2,y2,x3,y3,L:real);
```

```
  {вкладена процедура обчислення координат нової точки,
   яка лежить посередині відрізка з координатами xx1,yy1,xx2,yy2}
  procedure NewPoint(xx1,yy1,xx2,yy2:real; var xxn,yyn:real);
  begin
      xxn:=(xx1+xx2)/2;     yyn:=(yy1+yy2)/2;
  end;  {завершення процедури NewPoint}
  {вкладена функція обчислення довжини відрізка
   з координатами xx1,yy1,xx2,yy2}
  function LengthLine(xx1,yy1,xx2,yy2:real):real;
  begin
      LengthLine:=sqrt(sqr(xx1-xx2)+sqr(yy1-yy2));
  end; {завершення функції LengthLine}
var   {описування локальних змінних процедури LLL }
  L12,L23,L31,  {довжини відрізків}
  xn,yn:real;   {координати нової точки}
begin  { початок реалізації процедури LLL }
     {знаходження довжин сторін трикутника
      за координатами його стрін}
  L12:=LengthLine(x1,y1,x2,y2);
  L23:=LengthLine(x2,y2,x3,y3);
  L31:=LengthLine(x3,y3,x1,y1);
     {перевірка умови "Чи яка небудь зі сторін трикутників
      є більшою за встановлене значення?" }
  if (l12>l)or(l23>l)or(l31>l)
  then begin
      {є сторони більші за встановлене значення}
    if (l12>l23) and (l12>l31)
    then begin
      {сторона l12 є найбільшою, тому розбиваємо її}
      NewPoint(x1,y1,x2,y2,xn,yn);
      LLL(x1,y1,xn,yn,x3,y3,l);
      LLL(x2,y2,xn,yn,x3,y3,l);      end
    else if (l23>l31)
         then begin
           {сторона l23 є найбільшою, тому розбиваємо її}
           NewPoint(x2,y2,x3,y3,xn,yn);
           LLL(x1,y1,xn,yn,x3,y3,l);
           LLL(x1,y1,xn,yn,x2,y2,l);  end
         else begin
           {сторона l13 є найбільшою, тому розбиваємо її}
           NewPoint(x3,y3,x1,y1,xn,yn);
           LLL(x2,y2,xn,yn,x3,y3,l);
           LLL(x2,y2,xn,yn,x1,y1,l);  end;     end
   else {усі сторони є меншими за встановлену межу,
         далі розбивати нічого, збільшуємо на 2
         кількість трикутників}
     inc(CountTriangle,2);
end; {завершення процедури LLL}
BEGIN {початок програми }
  CountTriangle:=0;
  write('Координата X першої точки  = ');readln(px1);
  write('Координата Y першої точки  = ');readln(py1);
  write('Координата X другої точки  = ');readln(px2);
  write('Координата Y другої точки  = ');readln(py2);
  write('Координата X третьої точки = ');readln(px3);
  write('Координата Y третьої точки = ');readln(py3);
  write('Гранична довжина сторони   = ');readln(lg);
    { виклик процедури аналізу трикутника }
  LLL(px1,py1,px2,py2,px3,py3,lg);
    { виведення результатів на екран }
  writeln('Кількість трикутників = ',CountTriangle);
  readln;
END.
```

Результат роботи програми:

```
Координата X першої точки  = 0
Координата Y першої точки  = 0
Координата X другої точки  = 20
Координата Y другої точки  = 0
Координата X третьої точки = 0
Координата Y третьої точки = 20
Гранична довжина сторони   = 10
Кількість трикутників = 16
```

Перегляньте роботу готової програми.

На рис. 13.3 наведена схема розбиття трикутника із параметрами, вказаними у результаті роботи програми. Вертикальні і горизонтальні сторони трикутників дорівнюють 5, а нахилені - 7,07, тобто усі вони є меншими за 10. Зверніть увагу на те, що у випадку умови "<=" кількість трикутників скоротилась би у два рази, оскільки припустимими були б сторони із довжиною 10.

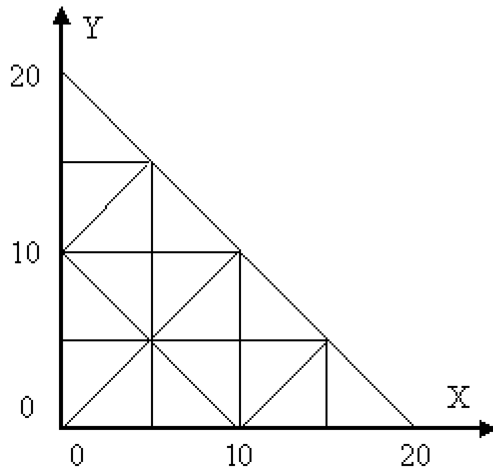


Рис. 13.3. Ілюстрація до результатів роботи програми 13.9

5. Контрольні запитання

1. Охарактеризуйте поняття підпрограма, процедура, функція.
2. Поясніть особливості використання стандартних процедур і функцій. Які групи стандартних процедур і функцій ви знаєте?
3. Поясніть використання стандартних процедур управління процесом виконання програм.
4. Поясніть використання стандартних математичних функцій.
5. Поясніть використання стандартних скалярних процедур і функцій.
6. Поясніть використання стандартних функцій перетворення.
7. Поясніть особливості використання функцій користувача.
8. Поясніть особливості використання процедур користувача.
9. Що таке формальні і фактичні параметри?
10. Якими способами можна визначати формальні параметри?
11. Якими способами можна передавати у підпрограми фактичні параметри?
12. Що таке глобальні і локальні ідентифікатори?
13. Поясніть, як визначається область дії ідентифікаторів.
14. Що таке рекурсія, якою вона буває, коли і як використовується?

Тема 14. Масиви Pascal

План

1. Визначення масивів
2. Основні операції з масивами
3. Пошуку інформації у масивах
4. Сортування інформації у масивах
5. Контрольні запитання

У цій темі ми познайомимось з масивами - одним із найважливіших типів структурованих даних. Масиви широко використовуються як у обчислювальних програмах, так і у програмах обробки інформації. Ви дізнаєтесь про те, як описувати масиви і використовувати їхні переваги у програмах, познайомитесь із прийомами виконання основних операції з масивами. Класичною сферою застосування масивів є сортування і пошук інформації. У цій темі ми розглянемо декілька способів вирішення таких задач.
Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторну роботу №10.

1. Визначення масивів

Введення таких структурованих даних, якими є масиви пов'язана із тим, що доволі часто при вирішення практичних задач виникає потреба обробити велику кількість даних одного і того самого типу, які найчастіше утворюють цілісну сукупність. Так сукупність студентів утворює групу, сукупність місяців утворює рік, сукупність чисел може утворювати вектор або матрицю. Звичайно незручно визначати велику кількість даних різними змінними з різними іменами. Ще складніше буде складати з такими змінними ефективні алгоритми їхньої обробки.
Значно зручнішим було б присвоїти усім даним одне ім'я і додатково вказувати індекс елемента (порядковий номер студента у групі, номер місяця, індекси елементу матриці тощо). Саме з метою полегшення обробки такої структурованої інформації у синтаксичні конструкції мови програмування Turbo Pascal, як і у інші мови, і були введені масиви.

Масив - це структурований тип даних, який складається з фіксованої кількості елементів одного типу, які мають спільне ім'я. Усі елементи масиву пронумеровані і звернутися до окремого елемента можна вказавши один або кілька індексів. При визначенні масивів можна використовувати будь-який раніше описаний тип. Елементами масиву можуть дані якого завгодно типу, включаючи структуровані. Індекси елементів масиву є виразами будь-якого скалярного типу за винятком дійсного. Кількість елементів масиву визначається діапазоном при описуванні і в процесі виконання програми не може бути зміненою.

Формат визначення масиву:

```
type
  <ім'я типу> = array [діапазон] of <тип елементів>;
var
  <ідентифікатор, ...>:<ім'я типу>;
      або
var
  <ідентифікатор, ...>:array [діапазон] of <тип елементів>;
```

Якщо при описуванні масиву використовується один індекс, масив називається одномірним (вектором), якщо два індекси - двомірним (матрицею), якщо n індексів - n-мірним. Хоча

кількість незалежних індексів масиву (розмірність масиву) обмежується лише об'ємом пам'яті комп'ютера, на практиці дуже рідко застосовуються масиви з розмірністю понад 3-4.

Для визначення масивів зручно використовувати константи. Ті самі константи будуть межами зміни параметрів циклів при обробці таких масивів і, у випадку необхідності зміни цих меж, потрібно буде змінити у програмі лише значення констант.

Приклади описування масивів:

```
const
  Hor=6; Ver=8;
type
  Month = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
  Matr  = array[1..10,1..20] of integer;
var
  M1,M2,M3  :Matr;
  MT21,MT22 :array[1..30] of string;
  YearProfit:array[Month] of real;
  Box       :array[0..Hor,0..Ver] of byte;
  Cube      :array[1..3,1..3,1..3] of boolean;
```

Так змінні `M1,M2,M3` є цілочисловими матрицями розміром 10x20, `MT21,MT22` - одномірним масивом рядків, які містять рядкову інформацію, `YearProfit` - одномірний масив дійсних чисел у якому індексом виступає перелічуваний тип `Month`, `Box` - матриця, розміри якої визначаються константами `Hor` і `Ver`, `Cube` - тримірний масив логічних значень розміром 3x3x3.

Ще одним зручним і компактним способом визначення масивів з одночасним присвоєнням усім елементам масиву початкових значень є використання механізму *типізованих констант*. У такому випадку початкові значення повинні бути відомими ще до початку виконання програми. Нагадаємо, що значення типізованих констант можна змінювати під час роботи програми.

Приклад визначення масиву як типізованої константи:

```
const
  Matrix:array[1..3,1..5] of integer = ((27, 11, 64, 33, -9),
                                         (-2, 97, 71,-54,  0),
                                         (19, 42, 83,-90, 39));
  Team:array[1..8] of string[8] = ('Динамо','Шахтар','Металіст','Дніпро',
                                   'Нива','Металург','Оболонь','Зоря');
```

Зверніть увагу на те, що кількість і порядок елементів списку повинні відповідати розмірності масиву. Причому у випадку багатомірних масивів, найшвидше змінюється най останній індекс.

У програмі усі операції виконуються не з усім масивом одразу а з окремими елементами. Для того щоб дістатися до певного елемента масиву слід вказати ім'я масиву і його індекс (індекси). Кількість і порядок таких індексів мають значення і не повинні виходити за межі діапазону, що встановлені при описуванні масиву. Важливим є те, що вказувати індекси можна як значення, як іменовані константи, як змінні, як вирази. При цьому слід лише забезпечити відповідність типів індексу і значення, та не допустити вихід за межі діапазону.

Приклади використання елементів масиву, описаних вище::

```
{Коректне використання елементів масивів}
```

```
  M1[1,1]:=-2312;
  MT21[14]:='Коваленко Василь Володимирович';
  i:=3;j:=5
  M3[i,j]:=M1[i,j]*M2[j,i];
  YearProfit[apr]:= 46117,23;
  write(Team[4]:12,Team[7]:12);
  read(Matrix[2,4]);

{Помилкові дії із масивами}
  M3:=18; {Спроба присвоїти значення усьому масиву а не елементу}
  Box[10,6]:=0;{Вихід індексу (10) за межі діапазону}
  Cube[2,2]:=true;{Невірна розмірність масиву}
  Write(Cube); {Спроба вивести на екран весь масив а не елемент }
```

У Turbo Pascal існує можливість одним оператором присвоювання передати значення елементів від одного масиву до іншого. Такий оператор виглядає як `M1:=M2`, причому ці два масиви повинні мати тотожний тип.

2. Основні операції з масивами

Основними операціями з масивами є

- ініціювання початкових значень масиву, яке полягає у наданні кожному елементу масиву одного і того ж значення, яке відповідає базовому типу;
- введення-виведення значень масиву, яке полягає у організації такої структури циклів, коли значення усіх елементів масиву у визначеній послідовності будуть введені з клавіатури (з файлу) або виведені на екран (у файл).

Ініціювання початкових значень елементів масиву зручно використовувати у циклі або вкладених циклах. Розглянемо приклади ініціювання масивів, що описані вище.

```
{Ініціювання одномірних масивів}
  for i:=1 to 30 do MT21[i]:='';
  for m:=Jan to Dec do YearProfit[m]:=0.0;
{Ініціювання двомірних масивів}
  for i:=1 to 10 do
    for j:= 1 to 20 do begin
       M1[i,j]:=0; M2[i,j]:=0; M3[1,1]:=1;
  end;
  for i:= 0 to Hor do
    for j:=0 to Ver do
      Box[i,j]:=255;
{Ініціювання тримірного масиву}
  for i:=1 to 3 do   {Увесь куб заповнимо значеннями false}
    for j:=1 to 3 do
      for k:=1 to 3 do
        Cube[i,j,k]:=false;
  for i:=1 to 3 do {головну діагональ кубу заповнимо значеннями true}
    Cube[i,i,i]:=true;
```

Введення-виведення можна організовувати в різному порядку, але при цьому слід забезпечити, щоб значення, яке присвоюється було присвоєно саме тому елементу, на який розраховує користувач. Тому зручно при виконанні цих дій виводити додаткову інформацію про індекси відповідних елементів. Найчастіше для введення-виведення масивів використовуються цикли. Розглянемо приклади введення-виведення масивів, описаних вище.

```
{Введення значень масивів. При введенні конкретних значень
 з клавіатури слід дотримуватись діапазону базового типу масиву}
{одномірні масиви}
```

```
for i:=1 to 30 do begin
  write('Уведіть відомості про студента під № ',i);
  readln(MT21[i]); end;
for m:=Jan to Dec do begin
  write('Уведіть прибуток за місяць ',m);
  readln(YearProfit[m]); end;
{двомірні масиви}
for i:= 1 to 10 do
  for j:=1 to 20 do begin
    write('M1[',i,',',j,']=');readln(M1[i,j]); end;
for i:= 0 to Hor do
  for j:=0 to Ver do begin
    write('Box[',i,',',j,']=');readln(Box[i,j]); end;
{тримірні масиви}
for i:= 1 to 3 do
  for j:=1 to 3 do
    for k:=1 to 3 do begin
      write('Cube[',i,',',j,',',k,']=');
      readln(Cube[i,j,k]); end;

{Виведення значень масивів. При введенні значень елементів
 масивів рекомендується використовувати формати}

{Одномірні масиви}
writeln('Список студентів групи МТ21');
for i:=1 to 30 do
  writeln(i:2,'.',MT21[i]);
Результат:
Список студентів групи МТ21
 1.Авраменко Андрій Олексійович
 2.Баранов Юрій Петрович
. . . .
30.Якименко Роман Павлович

writeln('Прибуток за рік по місяцях');
for m:=Jan to Dec do
  writeln('Місяць ',m:2,': 'YearProfit[m]10:2,' грн.');
Результат:
Прибуток за рік по місяцях
Місяць  1:   54472.22 грн.
Місяць  2:   49964.17 грн.
. . . .
Місяць 12:   71113.02 грн.

{Двомірні масиви}
{виведення двомірного масиву у вигляді таблиці}
writeln('   '); for j:= 1 to 20 do write(j:3); writeln;
for i:= 1 to 10 do begin
  write(i:3);
  for j:=1 to 20 do write(M1[i,j]:3);
  writeln; end;
Результат:
     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
  1  7  0  0  9  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  2  0  0  0  0  0  0  0 11  0  0  0 23  0  0  0  0  0  0  0  0
  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  3  0
  4  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  5  0  7  0  0  0 47  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  6  0  0  0  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0  0  0
  7  0  0  0  0  0  0  0  0  9  0  0  0  0  0  0  0  0  0  0  0
  8  0  0  0 43  0  0 57  0  0  0  0  0  0  0  0  0  0 12  0  0
  9  6  0  0  0  0  0  0  0  0  0  0  0  0  0 17  0  0  0  0  0
 10  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0  0  0  0 99

{Виведення двомірного масиву у стовпчик}
for i:= 0 to Hor do
  for j:=0 to Ver do
    writeln('Box[',i,',',j,']=',Box[i,j]:4);
Результат:
Box[1,1]=   0
```

```
Box[1,2]= 127
. . . .
Box[6,8]= 255

{Тримірний масив. Виведення у вигляді трьох таблиць по шарах}
for k:= 1 to 3 do begin
  writeln('Шар № ',k);
  for i:=1 to 3 do begin
    for j:=1 to 3 do write(Cube[i,j,k]:6);
    writeln;
  end;
end;
Результат:
Шар № 1
  true false false
 false false false
 false false false
Шар № 2
 false false false
 false  true false
 false false false
Шар № 3
 false false false
 false false false
 false false  true
```

3. Пошук інформації у масивах

При вирішенні багатьох задач виникає потреба визначити, містить масив певну інформацію чи ні. Наприклад, чи є у матриці нульові елементи. Задачі такого типу називають пошуком у масиві. Для організації пошуку у масиві можна використовувати різні алгоритми. Найбільш простий, але не самий продуктивний - алгоритм простого (повного) перебору. Пошук здійснюється послідовним порівнянням до тих пір, поки не буде знайдений елемент, який задовольняє умові пошуку, або не будуть перевірені усі елементи масиву. Цей алгоритм застосовується коли елементи масиву не упорядковані.

Приклад 14.1. Пошук у числовому масиві

Завдання: Створити програму, яка в одномірному цілочисловому масиві буде відшукувати число, значення якого користувач уводить з клавіатури.

Рішення: У програмі визначимо типізовану константу-масив цілих чисел. Змінна `i` буде параметром циклу, `n` - значення, яке вводиться з клавіатури і відшукується у масиві, `Spiv` - логічна змінна, що визначає збіг числа `n` з елементом масиву `M`.
Пошук відбувається у циклі `repeat`, у тілі якого оператор `if` порівнює поточний елемент масиву із уведеним числом. Якщо такий збіг буде знайдено, логічній змінній `Spiv` буде присвоєне логічне значення `true`, в іншому випадку буде збільшено значення параметру циклу і повторена перевірка умови із новим елементом.
Цикл завершується, якщо у масиві знайдено елемент, який дорівнює зразку, або у випадку, коли перебрані усі елементи масиву. Після завершення циклу змінна `Spiv` визначає, було знайдено збіг чи ні, а у випадку позитивної відповіді змінна `i` зберігатиме номер позиції у якій знайдено збіг. Зверніть також увагу на те, що програма відшукає лише першу появу числа у масиві а не усі позиції, як це могло бути із числом 27.

Текст програми:

```
Program Pr14_01;
const
  M:array[1..10] of integer =
    (27,12,132,-8,94,0,-35,27,93,-41);
```

```
var
  i,n:integer;
  Spiv:boolean;
BEGIN
  write('Уведіть число, яке слід знайти: ');readln(n);
  Spiv:=false;  i:=1;{Ініціювання початкового стану}
  repeat {цикл пошуку}
    if M[i]=n
      then Spiv:=true
      else inc(i)
  until (Spiv)or(i>10); {Умова завершення циклу}
  if Spiv    {Виведення результату пошуку}
    then writeln('Уведене число співпадає з елементом  № ',i)
    else writeln('Збігу із уведеним числом немає.');
  readln;
END.
```

Результати виконання програми:

```
Уведіть число, яке слід знайти: 11
Збігу із уведеним числом немає.
Уведіть число, яке слід знайти: 27
Уведене число співпадає з елементом № 1
Уведіть число, яке слід знайти: -35
Уведене число співпадає з елементом № 7
```

Перегляньте роботу готової програми.

Приклад 14.2. Підрахунок у числовому масиві

Завдання: Створити програму, яка у двомірному масиві цілих чисел від -99 до 99 буде підраховувати кількість нульових елементів, кількість однорозрядних і дворозрядних чисел.

Рішення: Задача підрахунку кількості елементів, значення яких відповідають певній умові є різновидом задачі пошуку. У такому випадку слід створити спеціальні змінні, у яких буде накопичуватись кількість таких елементів.

У масиві `M`, який визначений як типізована константа, зберігаються числа. Змінні `Zero`, `OneDigit`, `TwoDigit` будуть відповідно накопичувати кількість нульових одно- і дворозрядних чисел. Присвоївши їм на початку роботи програми нульові значення, увійдемо у цикли повного перебору всіх значень масиву `M`. Приналежність числа до нульового, одно- або дворозрядного визначається оператором `case`, у якому поступово збільшуються лічильники кількості чисел. Після завершення циклу залишається лише вивести отримані значення на екран.

Текст програми:

```
Program Pr14_02;
const   {Визначення та ініціювання матриці}
  M:array[1..5,1..6] of integer =
    ((  4, 25, -1,  0, 77, -12),
     (-99, -3,  1,  5, 54,  81),
     (  0, 28,-85,-13, 41,  93),
     ( 37, -5,  0,  2, 68,   7),
     ( 14,  0, -1,  0,  0, -54));
var
  Zero,OneDigit,TwoDigit,
  i,j:byte;
BEGIN
```

```
  {Ініціювання лічильників чисел}
  Zero:=0; OneDigit:=0; TwoDigit:=0;
  {Підрахунок кількості чисел}
  for i:=1 to 5 do
    for j:=1 to 6 do
      case M[i,j] of
           0:inc(Zero);
           -9..-1,1..9:inc(OneDigit);
           -99..-10,10..99:inc(TwoDigit)
      end;{case}
  {Виведення результату}
  writeln('Кількість нульових елементів  = ',Zero);
  writeln('Кількість однорозрядних чисел = ',OneDigit);
  writeln('Кількість дворозрядних чисел  = ',TwoDigit);
  readln;
END.
```

Результати виконання програми:

```
Кількість нульових елементів  = 6
Кількість однорозрядних чисел = 9
Кількість дворозрядних чисел  = 15
```

Перегляньте роботу готової програми.

Приклад 14.3. Пошук максимального (мінімального) елемента масиву

Завдання: Створити програму, яка у двомірному масиві цілих чисел від -99 до 99 буде знаходити мінімальний елемент. На екран вивести значення цього елемента і його координати у матриці.

Рішення: Алгоритм пошуку мінімально або максимального елемента масиву є доволі очевидним: слід припустити, що перший елемент масиву є мінімальним після чого порівнювати його із рештою елементів, присвоюючи нові значення у випадку знаходження ще меншого елемента.

Текст програми:

```
Program Pr14_03;
const   {Визначення та ініціювання матриці}
  M:array[1..5,1..6] of integer =
    ((  4, 25, -1,  0, 77, -12),
     (-99, -3,  1,  5, 54,  81),
     (  0, 28,-85,-13, 41,  93),
     ( 37, -5,  0,  2, 68,   7),
     ( 14,  0, -1,  0,  0, -54));
var
  Min,MinI,MinJ,i,j:integer;
BEGIN
  {Ініціювання початкової умови}
  Min:=M[1,1]; MinI:=1;MinJ:=1;
  {Пошук мінімального елемента}
  for i:=1 to 5 do
    for j:=1 to 6 do
      if M[i,j]<Min
      then begin
         Min:=M[i,j];  MinI:=i;  MinJ:=j;
      end;
  {Виведення результату}
  writeln('Мінімальний елемент має значення ',Min);
```

```
  writeln('Він знаходиться у рядку ',MinI,
          ' на позиції ',MinJ);
  readln;
END.
```

Результати виконання програми:

```
Мінімальний елемент має значення -99
Він знаходиться у рядку 2 на позиції 1
```

Перегляньте роботу готової програми.

Приклад 14.4. Бінарний пошук

Завдання: Є масив цілих чисел, відсортований у порядку збільшення. З клавіатури вводиться ціле число. Перевірити, чи є уведене число у масиві.

Рішення: Для пошуку інформації в упорядкованому масиві застосовують інші, більш ефективні у порівнянні з методом прямого перебору алгоритми, одним із яких є **метод бінарного пошуку**.
В упорядкованому масиві мінімальним (`Min`) елементом буде перший елемент у масиві, а максимальним (`Max`) - останній. На першому етапі слід порівняти зразок (`Find`) із середнім за номером (`Sered`) елементом масиву (див. рис. 14.1, *а*). Якщо трапилось, що зразок дорівнює середньому елементу, тоді задача вирішена.
Якщо зразок є більшим за середній елемент, це означає, що відшукуваний елемент розташований нижче. У такому випадку мінімальним елементом, від якого слід шукати число буде елемент з номером `Sered+1`, а максимальним залишиться те саме значення (див. рис. 14.1, *б*).
Якщо зразок є меншим за середній елемент, це означає, що відшукуваний елемент розташований вище. У такому випадку максимальним елементом, до якого слід шукати елемент буде елемент з номером `Sered-1`, а мінімальним залишиться те саме значення (див. рис. 14.1, *в*).
Після визначення частини (верхня або нижня) масиву, у якій слід продовжувати пошук, значення `Sered` повинно лежати на середині відстані між `Min` і `Max`, тобто, `Sered:=(Max-Min)/2+Min`.
Алгоритм бінарного пошуку, блок-схема якого представлена на рис. 14.2, закінчує свою роботу, якщо відшукуваний елемент знайдено, або у випадку, коли перед виконанням чергового циклу отримуємо `Min>Max`.

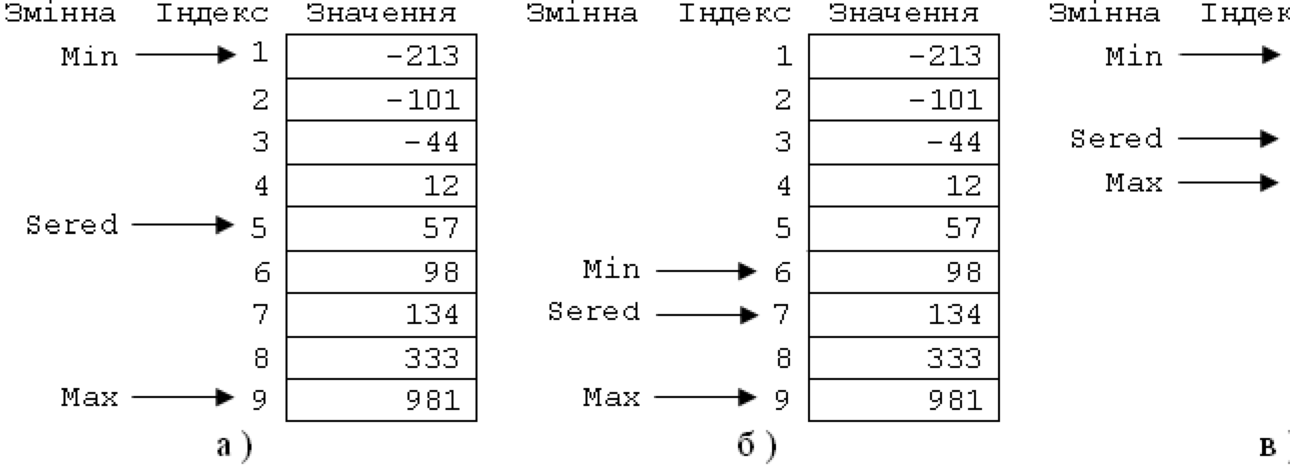


Рис. 14.1. Вибір чергового елемента при бінарному пошуку

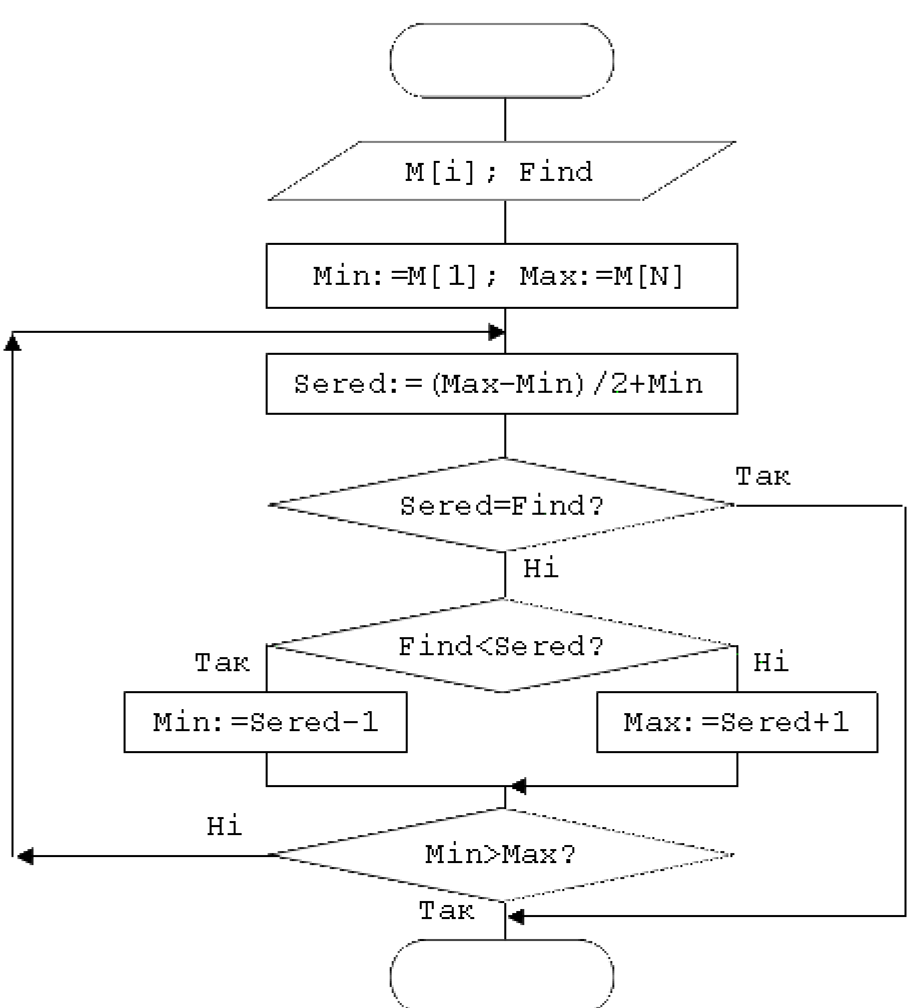


Рис. 14.2. Алгоритм бінарного пошуку у масиві, упорядкованому за збільшенням

Текст програми:

```
Program Pr14_04;
const  M:array[1..9] of integer =
         (-213,-101,-44,12,57,98,134,333,981);
var    Find,Min,Max,Sered:integer;
BEGIN
  write('Уведіть число, яке слід знайти:');readln(Find);
  Min:=1; Max:=9;
  repeat
    {Знаходження індексу середнього елементу}
    Sered:=round((Max-Min)/2)+Min;
    if Find=M[Sered] {перевірка співпадіння}
    then begin
      writeln('Значення ',Find,' знайдено');Exit  end;
    if Find<M[Sered]{Розрахунок нових індексів пошуку}
      then Max:=Sered-1
      else Min:=Sered+1;
  until Max<Min;
  writeln('Значення ',Find,' не знайдено');
  readln;
END.
```

Результати виконання програми:

```
Уведіть число, яке слід знайти:-101
```

```
Значення -101 знайдено
Уведіть число, яке слід знайти:111
Значення 111 не знайдено
```

Перегляньте роботу готової програми.

4. Сортування інформації у масивах

Задачі сортування масивів розповсюджені в інформаційних системах і використовуються як попередній етап задач пошуку, оскільки пошук в упорядкованому масиві відбувається набагато швидше, аніж у неупорядкованому. Під сортуванням масиву розуміється процес перестановки елементів масиву, метою якого є розміщення елементів масиву у певному порядку, тоді:

```
a[1]<=a[2]<=...<=a[N] - збільшення елементів
a[1]>=a[2]>=...>=a[N] - зменшення елементів
```

Існує декілька методів сортування масивів. Розглянемо два з них: метод повного перебору і метод прямого обміну.

Приклад 14.5. Сортування методом прямого перебору

Завдання: Є масив випадкових цілих чисел. Відсортувати масив у порядку збільшення, використовуючи метод прямого перебору.

Рішення: Алгоритм сортування масиву у порядку збільшення методом прямого перебору може бути представлений у такій послідовності дій. Спочатку, переглядаючи масив від першого до останнього елемента, знайти мінімальний елемент, поставити такий елемент на перше місце, а перший - на місце мінімального. Далі переглянути масив від другого елемента до останнього, у цій групі знайти найменший елемент, поставити цей елемент на друге місце а другий - на місце знайденого. Таку послідовність дій слід виконувати до передостаннього елемента.
Зазначимо, що час роботи алгоритму не залежить від початкового стану масиву. Працює алгоритм довго, навіть якщо для остаточного сортування масиву слід виконати одну перестановку, все рівно буде виконано `N-1` циклів.

Текст програми:

```
Program Pr14_05;
const
  N=10;
var
  M:array[1..N] of byte;
  min,i,j,k,temp:byte;
BEGIN
  {Ініціювання і виведення початкового масиву}
  writeln('Початковий масив:');
  randomize;
  for i:=1 to N do begin
     M[i]:=random(255);
     write(M[i]:4); end; writeln;
  writeln('Сортування масиву:');
  for i:=1 to N-1 do begin
    min:=i;
    for j:=i+1 to N do
      if M[j]<M[min] then Min:=j;
        Temp:=M[i];
```

```
        M[i]:=M[min];
        M[min]:=temp;
      {Виведення поточного стану масиву}
      for j:=1 to N do write(M[j]:4);writeln;
  end;
  writeln('Масив відсортований!');
  readln;
END.
```

Результат виконання програми:*(стрілки на екран не виводяться)*

```
Початковий масив:

107 233 109 234   9  76  42  66 116 122

Сортування масиву:
  9 233 109 234 107  76  42  66 116 122
  9  42 109 234 107  76 233  66 116 122
  9  42  66 234 107  76 233 109 116 122
  9  42  66  76 107 234 233 109 116 122
  9  42  66  76 107 234 233 109 116 122
  9  42  66  76 107 109 233 234 116 122
  9  42  66  76 107 109 116 234 233 122
  9  42  66  76 107 109 116 122 233 234
  9  42  66  76 107 109 116 122 233 234
Масив відсортований!
```

Перегляньте роботу готової програми.

Приклад 14.6. Сортування методом прямого обміну ("бульбочковий" метод)

Завдання: Є масив випадкових цілих чисел. Відсортувати масив у порядку зменшення, використовуючи метод прямого обміну.

Рішення: В основі алгоритму лежить обмін сусідніх елементів масиву. При сортуванні у порядку зменшення обмін буде відбуватись за умови, що наступний елемент є більшим за поточний. Таким чином менші ("легші") числа начебто спливають у кінець масиву. Такий процес нагадує процес підняття бульбашок у рідині, тому такий алгоритм ще називають "бульбочковим". Тривалість алгоритму залежить від початкового стану масиву, чим далі будуть "легкі" елементи від кінця і чим невпорядкованішим буде масив, тим довшою буде робота з упорядкування.
У програму введена логічна змінна `changed`, яка перед виконанням циклу отримує значення `false`. Процес сортування завершується, якщо після чергового циклу жоден з елементів не може бути обмінений місцями із сусіднім.

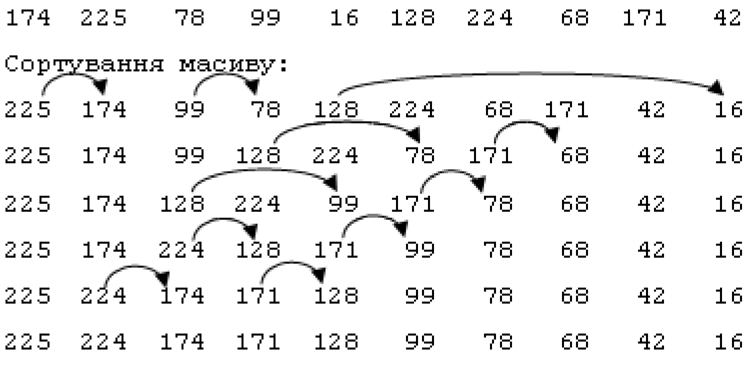
Текст програми:

```
Program Pr14_06;
const
  N=10;
var
  M:array[1..N] of byte;
  i,j,temp:byte;  changed:boolean;
BEGIN
  {Ініціювання і виведення початкового масиву}
```

```
  writeln('Початковий масив:');
  randomize;
  for i:=1 to N do begin
     M[i]:=random(255);
     write(M[i]:4); end; writeln;
  writeln('Сортування масиву:');
  {Сортування масиву}
  repeat
    changed:=false;
    for j:=1 to N-1 do
      if M[j]<M[j+1] then begin {Обмін елементів}
        Temp:=M[j];
        M[j]:=M[j+1];
        M[j+1]:=temp;
        changed:=true;
      end;
      {Виведення поточного стану масиву}
      for j:=1 to N do write(M[j]:4);
      writeln;
  until not changed;
  writeln('Масив відсортований!');
  readln;
END.
```

Результат виконання програми:*(стрілки на екран не виводяться)*

```
Початковий масив:
```



```
174  225   78   99   16  128  224   68  171   42
Сортування масиву:
225  174   99   78  128  224   68  171   42   16
225  174   99  128  224   78  171   68   42   16
225  174  128  224   99  171   78   68   42   16
225  174  224  128  171   99   78   68   42   16
225  224  174  171  128   99   78   68   42   16
225  224  174  171  128   99   78   68   42   16
```

```
Масив відсортований!
```

Перегляньте роботу готової програми.

5. Контрольні запитання

1. Що таке масив? Поясніть.
2. Як визначаються масиви? Наведіть приклади.
3. Як визначати масиви-типізовані константи?
4. Як масиви використовуються в операторній частині програми?
5. Наведіть приклади ініціювання масивів.
6. Наведіть приклади введення значень масивів.
7. Наведіть приклади виведення значень масивів.
8. Поясніть використання алгоритму повного перебору при пошуку інформації у масиві.
9. Яким чином можна підрахувати кількість елементів масиву, які задовольняють певній умові?
10. Як у неупорядкованому масиві відшукати максимальний (мінімальний) елемент?
11. Розкажіть про бінарний метод пошуку інформації у масиві.
12. Поясніть суть сортування елементів масиву.

13. Поясніть суть алгоритму сортування інформації у масиві методом повного перебору.
14. Поясніть суть алгоритму сортування інформації у масиві методом прямого обміну.

Тема 15. Обробка символьної інформації

План

1. Визначення даних рядкового типу. Структура представлення інформації
2. Процедури і функції роботи з даними рядкового типу
3. Контрольні запитання

У цій темі ми познайомимось із визначенням і використанням даних рядкового типу, дізнаємось про те, як утворюються рядкові вирази і які операції можна застосовувати до рядкових даних, які процедури і функції суттєво полегшують обробку символьної інформації. Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторну роботу №11.

1. Визначення даних рядкового типу. Структура представлення інформації

Рядок - це послідовність символів кодової таблиці комп'ютера, його можна розглядати, як структуру, утворену із простих даних типу `char`. Кількість символів у рядку (довжина рядка) може динамічно змінюватись від 0 до 255. Для визначення даних рядкового типу використовується зарезервоване слово `string`, за яким у квадратних дужках може вказуватись максимальна довжина рядка. Якщо таке значення не вказане, вважається, що максимальна довжина рядка складає 255 символів. (Відмітимо, що у наступних реалізаціях компілятора Pascal, зокрема у середовищі Delphi, обмеження на довжину рядка вже зняте шляхом введення додаткових типів `LongString` і `WideString`).

Змінну рядкового типу можна визначити або через попередньо описаний рядковий тип, або безпосередньо у розділі описування змінних. У програмах можна використовувати і рядкові константи, однак, зміст рядкової константи у тексті програми повинен розміщуватись у межах одного рядка коду програми.

Формат визначення даних рядкового типу:

```
type
  <ім'я типу>= string|[максимальна довжина рядка]|;
var
  <ідентифікатор,...>:<ім'я типу>;
      або
var
  <ідентифікатор,...>:string|[максимальна довжина рядка]|;
```

Приклади:

```
 const
  Adresa = 'Пр. Перемоги, 37'; {визначення рядкової константи}
type
  HelpStr = string[80]; {описування рядкового типу}
var
  Help1,Help2:HelpStr; {описування змінних з використанням типу}
  St         :string; {змінна із максимальною довжиною у 255 символів}
```

```
  Name,Prim  :string[15];{явне визначення максимальної довжини рядка}
```

Рядкові дані у пам'яті комп'ютера займають на один байт більше, ніж максимальна довжина рядка, вказана при визначенні. Цей додатковий (нульовий) байт зберігає значення поточної довжини рядка. При обробці рядкових даних ми можемо дістатися кожного окремого символу за його номером. Пояснимо останні твердження прикладом. Нехай ми визначили рядкову змінну `Name` із максимальною довжиною 15 символів (див. попередній приклад), а значення, яке матиме змінна під час роботи програми складає 'Сергій'. Тоді розміщення у пам'яті комп'ютера цієї змінної буде таким.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	байти
6	С	е	р	г	і	й										Значення

Таким чином значення `Name[0]` буде містити значення 6 - поточну довжину рядка. При зверненні до будь-якого іншого байта, буде видаватись його вміст, наприклад оператор `writeln(Name[3])` виведе на екран символ 'р'.

Для присвоювання рядковій змінній результату рядкового виразу використовується **оператор присвоювання (:=)**.

Приклади:

```
Gr1:='Група МТ-11';
PIB:='Коваленко В.В.';
```

Якщо довжина змінної після виконання оператора присвоювання або об'єднання рядків перевищує оголошену довжину, усі зайві символи відкидаються.

Рядкові вирази складаються з рядкових констант, змінних, ідентифікаторів функцій та знаків операцій. Над рядковими даними припустимі операції з'єднання і відношення.

Операція з'єднання (+) використовується для послідовного об'єднання кількох рядків у один.

Приклади:

```
        Вираз              Результат
'I'+'B'+'M'+' '+'PC'        'IBM PC'
'A'+'B'+'C'+'D'+'E'         'ABCDE'
```

Операції відношення (=, >, <, >=, <=, <>) виконують порівняння двох рядкових операндів. Це порівняння провадиться зліва направо до першого незбіжного символу. Більшим вважається той рядок, у якому цей символ має більший номер у кодовій таблиці комп'ютера., тому `'a'>'A'`, а не навпаки. Якщо рядки мають різну довжину, але повністю співпадають у спільній частині, більшим вважається довший рядок. Рядки вважаються рівними, якщо вони є рівними по довжині і містять усі однакові символи на відповідних місцях. При цьому враховуються і пробіли, у тому числі і ті, які стоять перед або за видимими символами. Важливо пам'ятати, що результат операції відношення над рядковими даними, як результат будь-якого відношення, завжди має логічний тип.

Приклади:

```
                 Вираз                       Результат
'Turbo Pascal 6.0' < 'Turbo Pascal 7.0'        true
```

```
'computer' > 'COMPUTER'            true
'ABCDEFGH' > 'ABCDEF'              true
    'ABC ' < 'ABC'                 false
   'Група' = 'Група'               true
```

2. Процедури і функції роботи з даними рядкового типу

Для ефективної роботи з рядковими даними можна використовувати такі процедури і функції

Процедура вилучення символів

`delete(St,Pos,N);`

`St` - змінна рядкового типу із якої вилучаються символи і яка містить результат процедури,
`Pos` - змінна або константа - номер позиції, з якої починається вилучення,
`N` - змінна або константа - кількість символів, які вилучаються.

```
Значення St   Процедура         Результат
'abcdefgh'    delete(St,4,2);   'abcfgh'
'Група МТ-11' delete(St,1,6);   'МТ-11'
```

Процедура вставки рядка

`insert(St1,St2,Pos);`

`St1` - змінна або константа - рядок, який вставляється,
`St2` - змінна рядкового типу, у яку вставляється St1 і яка містить результат процедури,
`Pos` - змінна або константа - номер позиції, починаючи з якої відбувається вставка.

```
Значення St1   Значення St2    Процедура            Результат
'МТ-'          'Група 11'      insert(St1,St2,7);   'Група МТ-11'
'В.В.'         'Коваленко '    insert(St1,St2,11);  'Коваленко В.В.'
```

Процедура перетворення числового значення у рядок

`str(Value,St);`
`Value` - змінна або константа цілочислового, дійсного або логічного типу. Може вказуватись разом із форматом,
`St` - змінна рядкового типу, якій присвоюється результат перетворення.

```
Числове значення     Процедура          Результат
i:=1450;            str(i:6,St);       '  1450'
r:=4.845E+01;       str(r:8:3,St);     '  48.450'
b:=true;            str(b:8,St);       '    true'
```

Зверніть увагу на принципову відмінність перетворених даних від початкових. Навіть у випадку повного збігу зовнішнього представлення числового значення і результату, останній все ж таки є рядковим даним, а тому і операції до нього слід застосовувати як для рядкових даних, а не чисел.

Процедура перетворення рядкового значення у числове

`val(St,Value,Code);`
`St` - змінна або константа рядкового типу, яку слід спробувати перетворити у число,

`Value` - змінна цілочислового, дійсного або логічного типу, яка буде зберігати числове значення після вдалої спроби перетворення.
`Code` - цілочислова змінна, яка після перетворення буде зберігати код перетворення. У випадку вдалого перетворення буде мати значення 0, у випадку помилки - номер позиції першого помилкового символу.

Значення St	Процедура	Результат	Code
'1450'	val(St,i,Code);	1450	0
'77.22E+02'	val(St,r,Code);	7722.0	0
'122.5H12-0.05'	val(St,r,Code);	-	6

Зверніть увагу на те, що не кожний рядок можна перетворити у число і, у випадку невдалої спроби такого перетворення,- значення числової змінної не буде визначене.

Функція копіювання частини рядка

`copy(St,Pos,N):`**`string`**
`St` - змінна або константа рядкового типу з якої виділяється частина,
`Pos` - змінна або константа - номер позиції символу, з якої починається копіювання. Якщо `Pos` перевищує поточну довжину рядка, функція повертає пробіл. Якщо `Pos>255` виникає помилка,
`N` - змінна або константа - кількість символів, яку слід скопіювати. Якщо `N` перевищує кількість символів, яка залишилася до кінця, функція видаляє тільки ті символи, які були у `St`.

Значення St	Функція	Результат
'ABCDEFG'	copy(St,2,3);	'BCD'
'ABCDEFG'	copy(St,6,10);	'EFG'
'ABCDEFG'	copy(St,10,6);	' '

Функція конкатенації (зчеплення) рядків

`concat(St1,St2,...,StN):`**`string`**

`St,St2,...,StN` - змінні або константи рядкового типу, які слід зчепити між собою. Порядок рядків визначається порядком їхнього переліку у фактичних параметрах функції.Довжина зчеплених рядків не повинна перевищувати 255 символів, інакше зайві символи будуть відкинуті.

Значення	Функція	Результат
St1:='Кафедра'; St2:='TM';	concat(St1,' ',St2);	'Кафедра TM'

Функція визначення поточної довжини рядка

`length(St):integer`
`St` - змінна або константа рядкового типу, поточну довжину якої слід визначити. Результат функції має цілочисловий тип.

Значення St	Функція	Результат
'Технологія машинобудування';	length(St);	26
'987654321'	length(St)	9

Функція пошуку рядка у рядку

`pos(St1,St2):integer`

`St1` - змінна або константа рядкового типу, яку слід відшукати,
`St2` - змінна або константа рядкового типу, у якій слід відшукати рядок `St1`,

Результат функції має цілочисловий тип, який дорівнює номеру позиції першої появи рядка `St1` у рядку `St2`. Якщо таку появу не знайдено, функція видає нульовий результат.

Значення St1	Значення St2	Функція	Результат
'abcdef'	'de'	Pos(St2,St1);	4
'abcdef'	're'	Pos(St2,St1);	0

Після розгляду прикладів програм, які ілюструють обробку символьної інформації, ви зможете закріпити отримані знання, виконуючи лабораторну роботу № 11.

Приклад 15.1. Обробка символьної інформації. Маніпулювання із рядками

Завдання: Реалізувати програму виведення на екран рядка тексту, що рухається.

Рішення: Для вирішення цієї задачі спочатку представимо по етапах, що повинно відображатися на екрані для імітації переміщення рядка. Нехай нам потрібно вивести рядок символів ABC починаючи від десятої позиції екрану (без урахування положення по висоті). Тоді послідовна імітація рухомого рядка буде мати такий вигляд:

```
  A   - виведення першого символу, починаючи з позиції 10
 AB   - виведення першого і другого символу, починаючи з позиції 9
ABC   - виведення усіх символів, починаючи з позиції 8
BC    - виведення другого і третього символів, починаючи з позиції 8
C     - виведення останнього символу, починаючи з позиції 8
```

Як видно, загальну послідовність слід розбити на два етапи. Перший етап повинен виводити символи починаючи від 1 і до усіх, причому позиція, з якої відбувається виведення постійно зменшується на одиницю. Другий етап виводить символи від усіх до останнього, причому позиція виведення не змінюється.
З метою структуризації програми введемо загальну процедуру виведення рядка тексту у певну позицію екрану (процедура `GoString` у прикладі), а всередині неї визначимо підлеглу процедуру виведення рядка тексту у певну позицію по горизонталі (процедура `OutR`).
Всередині процедури `GoString` визначимо локальну змінну `S1`, яка на відміну від повного рядка тексту зберігає його частину, яку треба вивести на даному конкретному етапі.
Суть процедури `OutR` полягає у переведенні курсору у певну позицію екрану (`GotoXY (Xv,Y)`), виведення рядка (`write(S1)`), паузі у роботі програми (`delay(1000)`) і видаленні з екрану усього рядка (`DelLine`). Зазначимо, що процедури `GotoXY`, `delay` і `DelLine` запозичені зі стандартного модулю `Crt`.
Суть основної процедури `GoString` полягає у наявності двох послідовних циклів, відповідно до того, як ми це визначили на етапі аналізу задачі. Всередині таких циклів спочатку формується значення локальної змінної `S1` і викликається процедура `OutR`. Формування змінної `S1` на першому етапі відбувається шляхом додавання по одному символу з повного рядка `S`. Формування змінної `S1` на другому етапі відбувається послідовним видаленням із неї першого символу.

Текст програми:

```
Program Pr15_01;
uses Crt;
type Ryadok=string[160]; {тип, який обмежує довжину рядка}
procedure GoString(X,Y:byte;S:Ryadok);
{процедура, яка реалізує виведення на екран рухомого рядка.
```

```
 X,Y - координати позиції екрану для виведення рядка S}
var
  S1:Ryadok; {перетворений рядок для виведення}
  i :byte;
  procedure OutR(Xv:byte);
  {вкладена процедура виведення рядка}
  begin
    GotoXY(Xv,Y); write(S1);{перевести курсор і вивести рядок S1}
    delay(10000); DelLine {пауза і знищення рядка}
  end;
begin {початок процедури GoString}
  S1:='';
  clrscr;
  for i:=1 to length(S) do  begin
  {перший етап виведення рядка від першого до всіх символів}
    S1:=S1+copy(S,i,1);
    OutR(X-i)
  end;
  for i:= 1 to length(S) do begin
  {другий етап виведення рядка від усіх символів до останнього}
    delete(S1,1,1);
    OutR(X-length(S))
  end
end; {завершення процедури GoString}
BEGIN
  GoString(40,12,'Attention! Error!');
END.
```

Результат виконання програми:

Результатом виконання програми із параметрами, які вказані у прикладі буде імітація переміщення уліво рядка тексту "`Attention! Error!`". Виведення буде починатися із 40-ї позиції 12-го рядка екрану.

Перегляньте роботу готової програми.

Приклад 15.2. Обробка символьної інформації. Пошук символів у рядку

Завдання: З клавіатури уводиться кілька слів, розділених принаймні одним пробілом. Реалізувати програму виділення кожного слова і формування із них масиву. Вивести на екран у стовпчик усі слова і кількість виділених слів.

Рішення: Основною задачею завдання є виділення слів, причому вказано, що слова розділяються між собою принаймні одним пробілом. Тобто, пробілів між словами може бути і більше, крім того можуть бути пробіли на початку початкового рядка і наприкінці.
Після введення з клавіатури послідовності слів (змінна `InSt`) додамо у кінець рядка один пробіл. Таким чином ми уникнемо невизначеності при завершенні аналізу. Основу такого аналізу складає цикл, який виконується до тих пір, поки у рядку `InSt` залишаються символи. Всередині такого циклу аналіз виконується по таких етапах. Спочатку вилучаються усі лідируючі пробіли. Така дія пов'язана із тим, що стандартна функція `pos` може відшукати позицію тільки першого пробілу у рядку. На другому етапі, який виконується тільки у випадку, якщо після видалення лідируючих пробілів у рядку `InSt` ще щось залишилось, відбувається збільшення лічильника слів, пошук позиції пробілу після слова, копіювання слова у масив і видалення його з початкового рядка `InSt`. Таким чином, знайшовши слово ми фактично переносимо його у масив слів.
Виведення результату відбувається у циклі і не являє собою якихось складнощів.

Текст програми:

```
Program Pr15_02;
var
  InSt:string;                   {початковий рядок}
  ArrSt:array[1..124] of string; {масив слів}
  i,n :byte;                     {параметр циклу і лічильник}
BEGIN
  writeln('уведіть набір слів, розділяючи їх принаймні одним пробілом');
  readln(InSt);
  {доточення одного пробілу у кінець рядка та ініціювання лічильника}
  InSt:=InSt+' ';    n:=0;
  {цикл виділення слів і формування масиву}
  while length(InSt)>0 do begin
    while  (InSt[1]=' ')and(length(InSt)<>0) do
      delete(InSt,1,1);{вилучення лідируючих пробілів}
    if length(InSt)<>0 then begin
      inc(n);{збільшення лічильника слів}
      i:=pos(' ',InSt); {пошук позиції першого пробілу після слова}
      ArrSt[n]:=copy(InSt,1,i-1);{копіювання слова у масив}
      delete(InSt,1,i);{видалення проаналізованого слова}
    end;
  end;
  {Виведення результатів роботи програми}
  for i := 1 to n do writeln(ArrSt[i]);
  writeln('Виділено ',n,' слів.');
  readln;
END.
```

Результат виконання програми:

```
уведіть набір слів, розділяючи їх принаймні одним пробілом
технологія   верстат     пристрій важіль редуктор
технологія
верстат
пристрій
важіль
редуктор
Виділено 5 слів.
```

Перегляньте роботу готової програми.

Приклад 15.3. Обробка символьної інформації. Перетворення рядків у число

Завдання: Реалізувати процедуру аналізу коректності уведеної цілочислової інформації. Така процедура повинна у випадку невірного введення даних видавати на екран повідомлення про помилку і пропонувати повторити введення. Помилкою вважається вихід за межі припустимого діапазону і введення замість числа символів.

Рішення: Задача аналізу введеної інформації - обов'язкова складова усіх сучасних систем и програм. Не можна допускати ситуації, коли при навмисному або ненавмисному введенні невірних даних програма аварійно перериває свою роботу. Вважається необхідним забезпечити контроль введення, обробити результати і у випадку помилки - видати відповідне повідомлення і повторити запит. Найчастіше трапляються помилки при введенні числової інформації, особливо дійсних чисел.
У програмі створимо процедуру `InputInt`, якій через параметри будемо передавати таке: `Prompt` - запит, який повинен з'явитись на екрані перед введенням; `Col`, `Row` - стовпчик і рядок, у яких слід починати введення; `IMin`, `IMax` - граничні значення; `I` - змінна, яка у випадку коректного введення отримає числове значення і передасть його назад у зовнішній блок.
Реалізація процедури полягає у наступному. Перед введенням даних встановити стан помилки

у false. Тоді при першому входженні у цикл repeat фрагмент обробки помилкової ситуації виконуватись не буде. Далі слід вивести у певне місце екрану запит, зчитати значення з клавіатури і присвоїти його не одразу цілочисловій змінній I, а локальній змінній рядкового типу IStr. Наступним кроком буде спроба перетворити рядкове значення у числове. Завершальним етапом буде визначення стану помилки за результатами перетворення і аналізу потрапляння числа у припустимий діапазон. Завершиться наш цикл тільки у випадку відсутності будь-якої помилки. У протилежному випадку цикл почне повторюватись спочатку, але вже із значенням стану помилки true. Тоді фрагмент обробки помилкової ситуації переведе курсор у позицію введення числа, виведе на певний час повідомлення 'Error!' і знищить весь рядок. Таким чином доки не буде з клавіатури уведене вірне цілочислове значення, доти буде продовжуватись виконання процедури із повторними запитами.

Текст програми:

```
Program Pr15_03;
uses Crt;
var n:integer;
procedure InputInt(Prompt:string;
          Col,Row,Imin,IMax:integer;var I:integer);
var
  IStr:string;    {Рядкова зміна для введення}
  C:integer;      {Змінна для коду помилки}
  Err:boolean;    {Логічна змінна стану помилки}
begin
  Err:=false;
  repeat
    {Фрагмент обробки помилкової ситуації}
    if Err then begin
      GotoXY(Col+length(Prompt),Row);
      write('Error!');
      delay(5000);DelLine;
    end;
    {Виведення запиту і зчитування значення}
    GotoXY(Col,Row);write(Prompt);
    readln(IStr);
    {Спроба перетворити рядок IStr у число I}
    val(IStr,I,C);
    {Визначення стану помилки}
    Err:=(C<>0) or (I>Imax) or (I<IMin)
  until not Err;
end;
BEGIN
  InputInt('Input integer value from 0 to 10:',1,1,0,10,n);
END.
```

Результат виконання програми (по етапах):

```
Input integer value from 0 to 10:12
Input integer value from 0 to 10:Error!
Input integer value from 0 to 10:Два
Input integer value from 0 to 10:Error!
Input integer value from 0 to 10:5
```

Перегляньте роботу готової програми.

Приклад 15.4. Обробка символьної інформації. Шифрування і дешифрування тексту

Завдання: Реалізувати дві програми. Перша програма повинна шифрувати рядок тексту

введений з клавіатури і виводити на екран його у зашифрованому виді. Друга програма повинна дешифрувати зашифрований попередньою програмою рядок, зміст якого вводиться з клавіатури.

Рішення: Будь-який процес кодування базується на певному методі або методах. Ці методи успішно розвиваються вже понад три тисячі років, тому сьогодні їх відомо кілька сотень. У нашій задачі розглянемо простий, але достатньо ефективний у повсякденні метод кодування за допомогою логічної операції "виключного або" - `XOR`. Цей метод базується на залежності: `A` **`xor`** `Any` **`xor`** `Any = A`, тобто, якщо до певного числа `A` двічі застосувати операцію `xor` із будь-яким постійним числом `Any`, то ми отримаємо початкове число.
Таким чином процес кодування і декодування у даному випадку - це два абсолютно ідентичних процеси (Якщо браузер дозволяє, розмістіть тексти програм поряд і переконайтеся у цьому). Місцями міняються лише оригінальний і кодований тексти. Суть алгоритму кодування полягає у переведенні тексту по символах у числові значення (функція `ord`), застосування до двох чисел (коду символу і ключа), логічної операції "виключного або", переведення результату і символьний вид (функція `chr`).

Тексти програм:

```
Program Pr15_041;
var
  TextOrig,TextCod:string;
  i,Key:byte;
BEGIN
  {Введення тексту і ключа для шифрування}
  writeln('Уведіть текст, який слід зашифрувати:');
  readln(TextOrig);
  writeln('Уведіть ключ для шифрування (1-255):');
  readln(Key);
  TextCod:='';
  {Шифрування тексту за допомогою операції xor}
  for i:=1 to length(TextOrig) do
    TextCod:=TextCod+chr(Key xor ord(TextOrig[i]));
  {Виведення зашифрованого тексту}
  writeln('Зашифрований текст:');
  writeln(TextCod);
  readln;
END.
```

Перегляньте роботу готової програми шифрування.

```
Program Pr15_042;
var
  TextOrig,TextCod:string;
  i,Key:byte;
BEGIN
  {Введення зашифрованого тексту і ключа}
  writeln('Уведіть зашифрований текст:');
  readln(TextCod);
  writeln('Уведіть ключ для дешифрування (1-255):');
  readln(Key);
  TextOrig:='';
  {Дешифрування за допомогою логічної операції xor}
  for i:=1 to length(TextCod) do
    TextOrig:=TextOrig+chr(Key xor ord(TextCod[i]));
  {Виведення результату дешифрування}
  writeln('Розшифрований текст:');
  writeln(TextOrig);
  readln;
END.
```

Перегляньте роботу готової програми дешифрування.

Результат виконання програми шифрування:

```
Уведіть текст, який слід зашифрувати:
Kafedra TM
Уведіть ключ для шифрування:
17
Зашифрований текст:
Zpwtucp1E\
```

Результат виконання програми дешифрування:

```
Уведіть зашифрований текст:
Zpwtucp1E\
Уведіть ключ для дешифрування:
17
Розшифрований текст:
Kafedra TM
```

3. Контрольні запитання

1. Охарактеризуйте дані рядкового типу. З чого вони можуть складатись і які існують при цьому обмеження?
2. Як визначаються дані рядкового типу? Наведіть приклади.
3. Як розміщуються у пам'яті комп'ютера дані рядкового типу?
4. Як утворюються вирази за участю рядкових даних?
5. Поясніть використання операцій відношення між даними рядкового типу.
6. Поясніть використання процедури вилучення символів.
7. Поясніть використання процедури вставки символів.
8. Поясніть використання процедури перетворення числової інформації у рядок символів.
9. Поясніть використання процедури перетворення символьної інформації у числову.
10. Поясніть використання функції копіювання символів.
11. Поясніть використання функції зчеплення рядків.
12. Поясніть використання функції визначення поточної довжини рядка.
13. Як не через функцію можна дізнатися про поточну довжину рядка?
14. Поясніть використання функції пошуку символів у рядку.
15. Поясніть суть алгоритму створення рядка, що рухається по екрану.
16. Як реалізувати коректне введення початкових даних з контролем?
17. Поясніть суть найпростіших алгоритмів шифрування і дешифрування текстів.

Тема 16. Множини Pascal

План

1. Визначення даних типу „множина"
2. Операції з множинами
3. Особливості введення виведення множин
4. Контрольні запитання

У цій темі ми познайомимось із визначенням і використанням даних типу "множина", дізнаємось про те, як формуються значення множин, які операції можна застосовувати до них і яким чином організовувати введення і виведення множин.
Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці

теми, а також виконавши лабораторну роботу №12.

1. Визначення даних типу „множина"

Множина - структурований тип даних, які є набором взаємозв'язаних за будь-якою ознакою або групою ознак частин, які можна розглядати як єдине ціле. Кожну частину в множині називають *елемент множини*. Усі елементи множини повинні належати до одного зі скалярних типів, за винятком дійсного. Цей тип називається *базовим типом* множини і визначається діапазоном або переліком. Кількість елементів множини не може перевищувати 256 і, відповідно, номери значень базового типу повинні знаходитись у діапазоні 0..255. Один елемент множини займає у пам'яті 1 біт.

У виразах Pascal значення елементів множини вказуються у квадратних дужках. Якщо множина не має елементів, вона називається пустою множиною.

Приклади множин:

`[1,3,5,7]`	- множина непарних цілих чисел від 1 до 7
`['a','f','k']`	- множина символів
`[0..9]`	- множина цілих чисел від 0 до 9
`[Red,Yellow,Green]`	- множина з трьох кольорів
`[]`	- пуста множина

Для визначення множин використовується словосполучення `set of` (множина з). Як і інші типи даних, визначити множину можна або визначивши тип і потім - змінну цього типу, або безпосередньо у розділі визначення змінних.

Формати визначення множин:

```
type
  <ім'я типу> = set of <елемент1,...,елементN>;
var
  <ідентифікатор,...>:<ім'я типу>
   або
var
   <ідентифікатор,...>:set of <елемент1,...,елементN>;
```

Приклади визначення множин:

```
type
  Digits  = set of 0..9;
  Symbols = set of 'A'..'Z';
  Colors = (Red,Green,Blue,Yellow,Magenta,Cyan,White,Black);
const
  EvenDigits: Digits = [0, 2, 4, 6, 8];
  Vowels: Symbols = ['A', 'E', 'I', 'O', 'U', 'Y'];
  HexDigits: set of '0'..'z' = ['0'..'9', 'A'..'F', 'a'..'f'];
var
  X,Y:Digit;
  C :set of Colors;
  Bukva :set of 'a'..'z';
```

Тип `Digits` визначає множину десяткових цифр, тип `Symbols` є множиною великих латинських символів, тип `Colors` є множиною кольорів.
Константа `EvenDigits` є множиною десяткових чисел, константа `Vowels` є множиною

великих латинських голосних символів, типізована константа HexDigits може містити множину символів від '0' до 'z', а при ініціюванні містить множину символів від '0' до '9', від 'A' до 'F' і від 'a' до 'f'.
Змінні X,Y можуть містити набір цифр, змінна C може містити набір кольорів, змінна Bukva може містити набір символів від 'a' до 'z'.

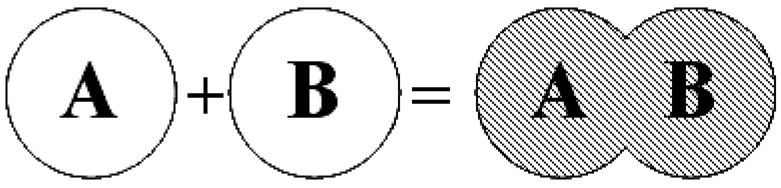
Використання в програмі множин надає низку переваг: значно спрощуються логічні умови у складних операторах if; підвищується наочність програми і зрозумілість алгоритму, економиться оперативні пам'ять комп'ютера і підвищується швидкість обчислення логічних виразів. У той же час, через відсутність у Pascal засобів введення-виведення та невелику можливу кількість елементів множин, їхнє застосування має певні обмеження.

2. Операції з множинами

Для множин припустимими є операції присвоювання, об'єднання, перетину, різниці та операції відношення.

Присвоювання. Присвоювання множині певного значення (набору) виконується за допомогою оператора присвоювання із такими застереженнями. Значення повинні подаватися у квадратних дужках, причому можливі значення контролюються компілятором лише для перелічуваних типів. Тому, якщо ви будете присвоювати значення множині, базовим типом якої є цілі числа або символи, будьте пильними. Так наприклад при намагання виконати оператор присвоювання X:=[0,1,12]; компілятор не видасть повідомлення про невідповідність значення 12 базовому типові, але при намаганні виконати присвоювання C:=[Red,Brown]; помилкова ситуація буде ідентифікована, адже значення Brown немає у базовому типі Colors.

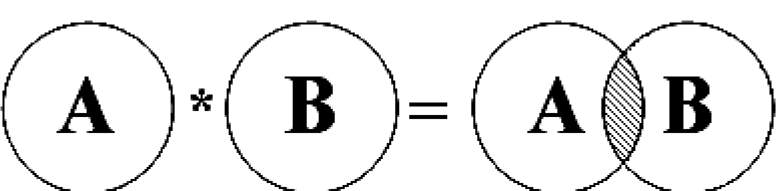
Об'єднання множин - операція "+". Результатом об'єднання двох множин буде третя множина, яка отримує елементи обох множин (див. рис.). Пам'ятайте, що об'єднувати можна лише множини із сумісними базовими типами.



Приклади виконання операції об'єднання множин:

Значення A	Значення B	Вираз	Результат
[1,2,3]	[2,4,6]	A + B	[1,2,3,4,6]
['a'..'d']	['e'..'z']	A + B	['a'..'z']

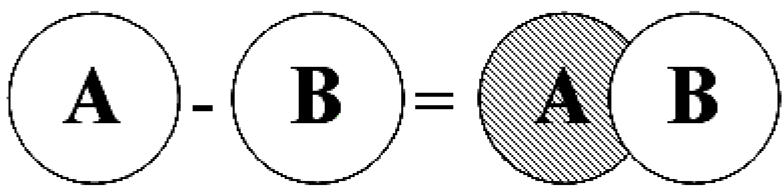
Перетин множин - операція "*". Результатом перетину двох множин є третя множина, яка отримує елементи, що одночасно належали двом вихідним множинам (див. рис.).



Приклади виконання операції перетину множин:

Значення A	Значення B	Вираз	Результат
`[1,2,3]`	`[2,4,6]`	`A * B`	`[2]`
`['a'..'x']`	`['e'..'z']`	`A * B`	`['e'..'x']`
`['a'..'c']`	`['e'..'z']`	`A * B`	`[ ]`

Різниця множин - операція "-". Результатом різниці двох множин є третя множина, яка отримує елементи першої множини, як не входять до другої множини (див. рис.).



Приклади виконання операції різниці множин:

Значення A	Значення B	Вираз	Результат
`[1,2,3]`	`[2,4,6]`	`A - B`	`[1,3]`
`['a'..'x']`	`['e'..'z']`	`A - B`	`['a'..'d']`
`['a'..'c']`	`['a'..'z']`	`A - B`	`[ ]`

Операції відношення. При роботі з множинами припустимі такі операції відношення: `=`, `<>`, `>=`, `<=`, `in`. Наслідком виразів із використанням операцій відношення є логічне значення `true` або `false`.

Операція "дорівнює" (=). Дві множини вважаються однаковими, якщо вони складаються з однакових елементів. Порядок розташування елементів у множині значення не має.

Приклади виконання операції порівняння "дорівнює":

Значення A	Значення B	Вираз	Результат
`[1,2,3,4]`	`[1,2,4,3]`	`A = B`	`true`
`['a','b','c']`	`['a','c']`	`A = B`	`false`
`['a'..'z']`	`['z'..'a']`	`A = B`	`true`

Операція "не дорівнює" (<>). Дві множини вважаються не однаковими, якщо вони відрізняються за потужністю або за значенням хоча б одного елемента.

Приклади виконання операції порівняння "не дорівнює":

Значення A	Значення B	Вираз	Результат
`[1,2,3]`	`[3,1,2,4]`	`A <> B`	`true`
`['a'..'z']`	`['b'..'z']`	`A <> B`	`true`
`[0..9]`	`[0..9]`	`A <> B`	`false`

Операція *"не менше" (>=)*. Множина A вважається не меншою за множину B, якщо всі елементи множини B містяться у множині A.

Приклади виконання операції порівняння "не менше":

Значення A	Значення B	Вираз	Результат
`[1,2,3,4]`	`[2,4,3]`	`A >= B`	`true`
`['a','b','c']`	`['a','c']`	`A >= B`	`true`
`['a'..'z']`	`['b'..'y']`	`A >= B`	`true`

Операція *"не більше" (<=)*. Множина A вважається не більшою за множину B, якщо всі елементи множини A містяться у множині B.

Приклади виконання операції порівняння "не більше":

Значення A	Значення B	Вираз	Результат
`[2,3]`	`[1,2,4,3]`	`A <= B`	`true`
`['a']`	`['a','b','c']`	`A <= B`	`true`
`['e'..'z']`	`['a'..'y']`	`A <= B`	`false`

Операція приналежності (in). Використовується для перевірки належності будь-якого значення певній множині. Саму множину попередньо описувати не обов'язково. Операція `in` дозволяє ефективно і наочно проводити складні перевірки. Наприклад, вираз "`if (a=1) or (a=2) or (a=3) or (a=4) or (a=5) or (a=6) then ...`" можна замінити коротшим виразом "`if a in [1..6] then ...`".

3. Особливості введення і виведення множин

Ми вже відзначали вище, що відсутність у Pascal стандартних процедур введення-виведення даних множинного типу створює певну перешкоду для їхнього використання. Тому, якщо є можливість організувати роботу з множинами без введення або виведення їхніх значень, це бажано робити. Якщо такої можливості немає, слід самостійно писати відповідні коди програми а при потребі частого виконання таких дій - писати відповідні процедури. Далі наведено фрагменти коду програми, які ілюструють, як можна організовувати введення-виведення даних множинного типу.

Спочатку наведемо фрагмент коду, у якому визначаються типи і змінні. які будуть використовуватись далі.

```
uses Crt;
type
  Digits  = set of 0..9;
  Colors = (Red,Green,Blue);
var
  x,y:Digits;
  Col:Colors;
  C:set of Colors;
  Bukva:set of 'a'..'z';
  a,b,i:byte;
  symbol:char;
  S:string;

BEGIN
```

Найпростішою є **робота із цілочисловими множинами**. Послідовність дій тут повинна бути такою. Спочатку з клавіатури вводиться число і присвоюється змінній байтового типу. Далі

число додається (або виконується інша операція) вже як елемент множини.
Виведення цілочислових множин повинно проводитись у циклі, який починається із мінімального елемента множини і закінчується максимальним. Кожне число перевіряється на присутність у множині (див. оператор `if`) і виводиться на екран при позитивному значенні цієї умови.

```
X:=[0,1]; { Формування початкової множини }
write('Digits = '); readln(a);{ Введення числа  }
y:=x+[a];{ Додавання числа, як елемента множини }
for i:= 0 to 9 do { Виведення числа, якщо воно є у множині }
  if i in y then writeln(i);
```

Схожою із числовими множинами є **робота із символьними множинами**. Послідовність дій тут повинна бути тією самою. Спочатку з клавіатури вводиться символ і присвоюється відповідній змінній. Далі символ передається до множини.
Виведення символьних множин також проводиться у циклі від найменшого до найбільшого елемента множини. Кожний символ перевіряється на присутність у множині (див. оператор `if`) і виводиться на екран при позитивному значенні цієї умови.

```
Bukva := ['a','b','c']; { Формування початкової множини }
write(' Bukva = '); readln(Symbol); { Введення символу }
Bukva := Bukva + [Symbol];{ Додавання символу, як елемента множини }
for Symbol:= 'a' to 'z' do { виведення символу, якщо він є у множині }
  if Symbol in Bukva then writeln(Symbol);
```

Найскладнішою є **робота із множинами перелічуваного типу**. Тут значення повинно спочатку вводитись як рядок символів. Для того щоб додати відповідний елемент множини слід через оператори перевірки умови `if` перебрати усі можливі варіанти введення значення. Для того щоб скоротити кількість варіантів (адже користувач може вводити значення у різних регістрах) можна скористатися стандартною функцією UpCase, яка переводить один символ у верхній регістр.
При виведенні множини слід у циклі перебрати усі можливі значення і при наявності елемента у множині за допомогою оператора `case` вивести відповідні рядки на екран.
Незручність полягає у тому, що при великій кількості елементів множини суттєво збільшується частина формування самої множини (якою є максимальна кількість елементів множини, такою буде і кількість операторів `if`) і збільшується довжина оператора `case` при остаточному виведенні результатів.

```
  C:=[Red,Green]; { Формування початкової множини }
  Write(' Input color (Red,Green or Blue) = '); readln(S); { введення рядка }
  for i:=1 to length(S) do s[i] := UpCase(s[i]); { Зробити усі символи великими }
  { доповнити множину значенням в залежності від рядка }
  if S='RED'   then C:=C+[Red];
  if S='GREEN' then C:=C+[Green];
  if S='BLUE'  then C:=C+[Blue];
  { Виведення множини }
  for Col:=Red to Blue do
    { Якщо значення є у множині, тоді виводити відповідний йому текст}
    if Col in C then case Col of
                        Red   : writeln('Red');
                        Green : writeln('Green');
                        Blue  : writeln('Blue');
                     end;
 readln;
END.
```

Наприкінці розглянемо приклад програми із використанням множин.

Приклад 16.1. Пошук у множині

Завдання: На певних підприємствах України випускається визначена номенклатура виробів. Створити програму, яка визначає всю номенклатуру виробів, які не випускаються на жодному з підприємств, а також визначити підприємства, які випускають певний вид продукції.

Рішення: У програмі визначимо два перелічуваних типи - перелік підприємств (`Pidpryemstvo`) і перелік товарів (`Products`), а також два множинні типи, що є відповідно множинами товарів (`Asortyment`) і множинами підприємств (`Vyrobnyki`). Тоді всю промисловість можна визначити як масив підприємств, що випускають певний асортимент продукції - змінна `Promyslovist`. Для виведення назв підприємств і назв товарів створені дві процедури, які по їхнім значенням у множині виводять відповідні тексти українською мовою.
Перша частина основної програми полягає у визначення того, яку продукцію випускають підприємства.
При визначенні номенклатури товарів, які не випускає жодне із підприємств, спочатку формується повна множина товарів, після чого перебираються усі підприємства і від початкової множини віднімаються ті товари, які на них випускаються. Ті товари, які залишаться у множині після перебору усіх підприємств не випускаються на жодному із них.
При визначенні того, які підприємства випускають світильники, спочатку формується пуста множина таких підприємств. Далі перебираються усі підприємства і, якщо серед їхнього асортименту є світильники, додаються до множини підприємств. Після перебору усіх підприємств буде сформована множина виробників даної продукції.

Текст програми:

```
Program pr16_01;
type
   Products = (Automobil,Motocycl,Velosiped,Traktor,Kholodylnik,
               Pylesos,Praska,PralnaMachina,GazovayaPlita,
               Chaynik,Kavovarka,Mixer,KuchonnyjKombajn,Korabl,
               Litak,Svetilnik,Photoaparat,Binokl,Televisor);
   Asortyment = set of Products;
   Pidpryemstvo = (ANTK_Antonov, Aviant, Artem, Radar, Lepse,
                   Motozavod, Rostok, Radiozavod, KhTZ,
                   Velozavod, Nord, Arsenal, AutoZAZ);
var
   Promyslovist: array [Pidpryemstvo] of Asortyment;
   Factory : Pidpryemstvo;
   As : Asortyment;
   Product : Products;
   Vyrobnyki :set of Pidpryemstvo;

{ Процедура виведення назви підприємства за його значенням у множині }
procedure OutPidpryemstvo(Nazva:Pidpryemstvo);
   begin
     case Nazva of
          ANTK_Antonov :writeln('АНТК ім. Антонова');
          Aviant       :writeln('Авіант');
          Artem        :writeln('Артем');
          Radar        :writeln('Радар');
          Lepse        :writeln('Завод ім. Лепсе');
          Motozavod    :writeln('Мотозавод');
          Rostok       :writeln('Росток');
          Radiozavod   :writeln('Радіозавод');
          KhTZ         :writeln('Харківський тракторний завод');
          Velozavod    :writeln('Харківський велозавод');
          Nord         :writeln('Норд');
          Arsenal      :writeln('Арсенал');
```

```
          AutoZAZ      :writeln('АвтоЗАЗ');
      end;
   end;

{ Процедура виведення назви товару за його значенням у множині }
procedure OutProduct(Nazva:Products);
   begin
      case Nazva of
          Automobil         :writeln('Автомобіль');
          Motocycl          :writeln('Мотоцикл');
          Velosiped         :writeln('Велосипед');
          Traktor           :writeln('Трактор');
          Kholodylnik       :writeln('Холодильник');
          Pylesos           :writeln('Пилесос');
          Praska            :writeln('Праска');
          PralnaMachina     :writeln('Пральна машина');
          GazovayaPlita     :writeln('Газова плита');
          Chaynik           :writeln('Чайник');
          Kavovarka         :writeln('Кавоварка');
          Mixer             :writeln('Міксер');
          KuchonnyjKombajn  :writeln('Кухонний комбайн');
          Korabl            :writeln('Корабель');
          Litak             :writeln('Літак');
          Svetilnik         :writeln('Світильник');
          Photoaparat       :writeln('Фотоапарат');
          Binokl            :writeln('Бінокль');
          Televisor         :writeln('Телевізор');
      end;
   end;

BEGIN
   { Формування множин асортименту по підприємствах }
   Promyslovist[ANTK_Antonov] := [Litak,PralnaMachina,Velosiped];
   Promyslovist[Aviant]        := [Litak];
   Promyslovist[Artem]         := [Pylesos,Svetilnik];
   Promyslovist[Radar]         := [Kavovarka,Svetilnik,GazovayaPlita];
   Promyslovist[Lepse]         := [Traktor];
   Promyslovist[Motozavod]     := [Motocycl];
   Promyslovist[Rostok]        := [Praska,Chaynik,Kavovarka,Mixer,KuchonnyjKombajn
   Promyslovist[Radiozavod]    := [Televisor,Svetilnik];
   Promyslovist[KhTZ]          := [Traktor];
   Promyslovist[Velozavod]     := [Velosiped];
   Promyslovist[Nord]          := [Kholodylnik,GazovayaPlita];
   Promyslovist[Arsenal]       := [Photoaparat,Binokl];
   Promyslovist[AutoZAZ]       := [Automobil];

 { Пошук товарів, які не випускаються на жодному з підприємств }
   As:=[Automobil,Motocycl,Velosiped,Traktor,Kholodylnik,
        Pylesos,Praska,PralnaMachina,GazovayaPlita,
        Chaynik,Kavovarka,Mixer,KuchonnyjKombajn,Korabl,
        Litak,Svetilnik,Photoaparat,Binokl,Televisor]; {повний асортимент}
   for Factory := ANTK_Antonov to AutoZAZ do { перебирати усі підприємства }
       {віднімати від усього асортименту асортимент кожного підприємства }
       As:=As-Promyslovist[Factory];
   writeln('Не випускаються на жодному підприємстві такі товари:');
   for Product := Automobil to Televisor do { Перебирату всю продукцію}
      {Якщо виріб у залишковому асортименті - виводити його назву}
      if Product in As then OutProduct(Product);

   { Пошук заводів, які випускають світильники }
   Vyrobnyki := []; { Початкова пуста множина }
   for Factory := ANTK_Antonov to AutoZAZ do { перебирати усі підприємства }
      if Svetilnik in Promyslovist[Factory] { якщо світильник є в асортименті під
```

```
      then Vyrobnyki := Vyrobnyki+[Factory]; { додавати підприємство до множини в
   writeln('Випускають світильники такі підприємства:');
   for Factory := ANTK_Antonov to AutoZAZ do { перебирати усі підприємства }
     { якщо підприємство у множині виробників - виводити його назву}
     if Factory in Vyrobnyki then OutPidpryemstvo(Factory);
   readln;
END.
```

Результат виконання програми:

```
Не випускаються на жодному підприємстві такі товари:
Корабель
Випускають світильники такі підприємства:
Артем
Радар
Радіозавод
```

Перегляньте роботу готової програми.

4. Контрольні запитання

1. З даних якого типу можна утворювати множини?
2. Яка максимальна кількість елементів множин припустима у Pascal?
3. Як визначаються дані множинного типу?
4. Як виконується присвоювання значень даним множинного типу, на що слід звертати особливу увагу?
5. Що є результатом виконання операцій об'єднання, різниці та перетину множин?
6. Як виконуються операції відношення для даних множинного типу?
7. Як виконується введення-виведення для даних множинного типу, елементи якої є числами або символами?
8. Як виконується введення-виведення для даних множинного типу, елементи якої є даними перелічуваного типу?

Тема 17. Записи Pascal

План

1. Визначення даних типу „запис"
2. Робота із записами
3. Контрольні запитання

У цій темі ми познайомимось із призначенням і визначенням даних типу "запис", дізнаємось про те, яким чином організовувати введення-виведення і обробка записів.
Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторну роботу №13.

1. Визначення даних типу „запис"

До сих пір, розглядаючи структуровані типи даних, ми зазначали, що структурувати у межах однієї змінної можна дані тільки одного типу. Наприклад, рядки завжди є набором символів, усі елементи одного масиву або множини можуть бути тільки одного типу - або числом, або символом, або перелічуваним типом.
На практиці доволі часто виникає потреба у вирішенні задач, де сукупність даних про певний

об'єкт складається із даних різного типу. Саме для таких задач створений тип "запис".

Запис - структурований тип даних, який складається з фіксованої кількості компонентів одного або кількох типів.

Для визначення записів використовується конструкція, що розпочинається із слова `record` (запис) і закінчується `end`. Між цими словами записуються компоненти (поля) запису. Компонент запису подається як ім'я і тип.

Формати визначення записів:

```
type
  <ім'я типу> = record
                  <ім'я поля>:< тип поля>;
                  . . . .
                  <ім'я поля>:< тип поля>
                end;
var
  <ідентифікатор,...>:<ім'я типу>

   або

var
   <ідентифікатор,...>:record
                  <ім'я поля>:< тип поля>;
                  . . . .
                  <ім'я поля>:< тип поля>
                end;
```

Приклади визначення множин:

```
type
  FormType = (hatchback, sedan, universal, cabriolet);
             { типи кузовів }
  FuelType = (petrol, diesel, gas);
             { типи палива }
  Auto = record
           Marka  : string[20]; { марка автомобіля    }
           Price  : real;       { вартість автомобіля }
           Form   : FormType;   { тип кузова          }
           EngTyp : FuelType;   { тип двигуна         }
           EngVol,              { об'єм двигуна       }
           EngPow : real;       { потужність двигуна  }
           Year   : integer;    { рік випуску         }
           ABS    : boolean;    { наявність АБС       }
         end;
var
  MyAuto,FriendAuto : Auto;
  BaseOfAuto        : array [1..100] of Auto;
```

Найчастіше записи використовуються для організації баз даних. У наведеному прикладі запис містить 8 полів різних типів (перелічувані, числові, рядковий і логічний) і визначає інформаційну структуру даних про автомобіль. Поля одного типу можна визначати як окремо, так і через кому.
Визначати змінні типу "запис" можна як поодинокі (`MyAuto`), так і у вигляді масивів (`BaseOfAuto`). Ідентифікатори полів у межах запису повинні бути унікальними, в той же час у різних записах можуть бути поля із однаковими іменами.

Для того, щоб звернутися у програмі до даного типу "запис" слід спочатку вказати ім'я

змінної, значення індексу масиву (для масивів) і після коми - ім'я поля, *наприклад*: `MyAuto.Marka, MyAuto.EngVol, BaseOfAuto[1].Year, BaseOgAuto [27].Marka`.
Такі конструкції слід використовувати при введенні з клавіатури значень полів, в операторах присвоювання, у виразах із участю записів, при виведенні тощо.

2. Робота із записами

Робота із записами поділяється на введення-виведення даних та їх обробку.

При введенні значень змінних типу "запис" окремо вводяться значення кожного поля. Порядок переліку полів при введенні не має значення, але при цьому необхідно організовувати відповідні пояснення, щоб користувач орієнтувався у тому, яку інформацію він уводить. Далі розглянемо приклади організації введення даних для одиничної змінної та для масиву.

```
{ Введення значень для одиничної змінної }
  writeln(' Вкажіть параметри автомобіля:');
  write  ('Марка              :'); readln(MyAuto.Marka);
  write  ('Вартість           :'); readln(MyAuto.Price);
  write  ('Об"єм двигуна      :'); readln(MyAuto.EngVol);
  write  ('Потужність двигуна :'); readln(MyAuto.EngPow);
  write  ('Рік випуску        :'); readln(MyAuto.Year);
  { Введення значень для масиву змінних }
  writeln(' Заповніть базу даних про автомобілі');
  for i := 1 to 100 do begin
    writeln(' Вкажіть параметри автомобіля за номером ',i:3);
    write  (' Марка              :'); readln(BaseOfAuto[i].Marka);
    write  (' Вартість           :'); readln(BaseOfAuto[i].Price);
    write  (' Об"єм двигуна      :'); readln(BaseOfAuto[i].EngVol);
    write  (' Потужність двигуна :'); readln(BaseOfAuto[i].EngPow);
    write  (' Рік випуску        :'); readln(BaseOfAuto[i].Year);
  end;
```

У прикладі не наводиться введення полів перелічуваних і логічного типів. Введення даних такого типу потребує створення окремих процедур. Схожі процедури наведені нижче у прикладі організації виведення даних типу "запис".

```
  writeln(' Відомості про автомобіль:');
  writeln('Марка              :', MyAuto.Marka);
  writeln('Вартість           :', MyAuto.Price);
  write  ('Тип кузова         :');OutFormType (MyAuto.Form);
  write  ('Тип пального       :');OutFuelType (MyAuto.EngTyp);
  write  ('Об"єм двигуна      :', MyAuto.EngVol);
  write  ('Потужність двигуна :', MyAuto.EngPow);
  write  ('Рік випуску        :', MyAuto.Year);
  OutABS(MyAuto.ABS);
```

Для організації виведення значень полів перелічуваного типу були створені такі процедури:

```
procedure OutFormType (Form:FormType);
begin
  case Form of
        hatchback :writeln(' Хетчбек ');
        sedan     :writeln(' Седан ');
        universal :writeln(' Універсал ');
        cabriolet :writeln(' Кабріолет ');
  end;
end;
```

```
procedure OutFuelType(Fuel:FuelType);
begin
  case Fuel of
        petrol :writeln(' Бензин ');
        diesel :writeln(' Дизельне паливо ');
        gas    :writeln(' Газ ');
  end;
end;
procedure OutABS (ABS:boolean);
begin
  case ABS of
        true  :writeln(' АБС ');
        false :writeln(' Без АБС');
  end;
end;
```

Обробка даних типу "запис" полягає у створенні виразів, операндами яких є поля записів, а також присвоювання полям записів відповідних значень.

```
  MyAuto.Marka  := 'Таврія';
  MyAuto.Price  := 13990.00;
  MyAuto.Form   := Hatchback;
  MyAuto.EngTyp := petrol;
  MyAuto.EngVol := 1.2;
  MyAuto.EngPow := 51;
  MyAuto.Year   := 2003;
  MyAuto.ABS    := false;
```

Якщо змінні типу "запис" мають один і той самий тип, можна використовувати оператор присвоювання для змінної в цілому, наприклад:

```
 FriendAuto:=MyAuto;
```

Звернення до полів запису при введенні, виведенні або при присвоюванні через потребу кожного разу записувати префікс (ім'я змінної) є не досить зручним. Для вирішення цієї проблеми у `Pascal` введений оператор `with`, який дозволяє один раз вказати ім'я змінної і у межах дії цього оператора більше її не повторювати.

Формат оператора `with`:

```
with <змінна типу "запис" > do < оператор >;
```

Наведений вище фрагмент коду програми із використанням оператора `with` виглядає більш компактним без втрати наочності присвоювання.

```
  with MyAuto do begin
     Marka  := 'Таврія';
     Price  := 13990.00;
     Form   := Hatchback;
     EngTyp := petrol;
     EngVol := 1.2;
     EngPow := 51;
     Year   := 2003;
     ABS    := false;
   end;
```

У `Pascal` є можливість організовувати вкладені записи (тобто поле у записі може бути в свою чергу записом. Для полегшення роботи із такими типами даних можна використовувати вкладені оператори `with`.

Дані типу "запис" можна використовувати для організації роботи із динамічними структурами даних, для роботи із файлами тощо.

Перегляньте роботу готової програми, у якій наведено усі основні етапи створення і використання бази даних.

Використання даних тип "запис" може бути ефективним і зручним не тільки при створення баз даних. Основна вимога до інформації, яку планується представляти як запис - структурованість. Так у прикладі, що буде розглянутий далі, структурованість комплексних чисел дозволяє використати запис для їхньої обробки.

Приклад 17.1. Обробка комплексних чисел

Завдання: Створити програму обробки (додавання, віднімання, множення і ділення) комплексних чисел.

Рішення: Враховуючи можливу потребу багаторазової обробки комплексних чисел, створимо окремі процедури для кожної з операції. Зміст таких процедур полягає у визначенні окремо значення дійсної і уявної частини комплексного результату.
Основна програма складається із присвоювання початковим даним відповідних значень і наступного виклику відповідних процедур в комплексі з операторами виведення результату.

Текст програми:

```
Program pr17_01;
type
  Complex = record { Визначення комплексного числа }
              Re,Im : real;
            end;
var
  A,B,C : Complex;
{ Процедура додавання двох комплексних чисел }
procedure AddComplex(x,y:Complex; var z:Complex);
begin
   z.Re:=x.Re+y.Re;
   z.Im:=x.Im+y.Im
end;
{ Процедура віднімання двох комплексних чисел }
procedure SubComplex(x,y:Complex; var z:Complex);
begin
   z.Re:=x.Re-y.Re;
   z.Im:=x.Im-y.Im
end;
{ Процедура множення двох комплексних чисел }
procedure MulComplex(x,y:Complex; var z:Complex);
begin
   z.Re:=x.Re*y.Re-x.Im*y.Im;
   z.Im:=x.Re*y.Im+x.Im*y.Re
end;
{ Процедура ділення двох комплексних чисел }
procedure DivComplex(x,y:Complex; var z:Complex);
var
   temp:real;
begin
   temp:=sqr(y.Re)+sqr(y.Im);
   z.Re:=(x.Re*y.Re+x.Im*y.Im) / temp;
   z.Im:=(x.Re*y.Im-x.Im*y.Re) / temp;
end;
{ Основна програма }
```

```
BEGIN
   a.Re := 1; a.Im := 1;
   b.Re := 1; b.Im := 2;
   AddComplex(a,b,c);
   writeln('Результат додавання  = ',c.Re:6:2,c.Im:6:2,'i');
   SubComplex(a,b,c);
   writeln('Результат віднімання = ',c.Re:6:2,c.Im:6:2,'i');
   MulComplex(a,b,c);
   writeln('Результат множення   = ',c.Re:6:2,c.Im:6:2,'i');
   DivComplex(a,b,c);
   writeln('Результат ділення    = ',c.Re:6:2,c.Im:6:2,'i');
   readln;
END.
```

Результат роботи програми:

```
Результат додавання  =  2.00  3.00i
Результат віднімання =  0.00 -1.00i
Результат множення   = -1.00  3.00i
Результат ділення    =  0.60  0.20i
```

Перегляньте роботу готової програми.

3. Контрольні запитання

1. Чим була викликана потреба у введенні типу "запис"?
2. Як визначити у програмі тип "запис"?
3. Як у програмі звернутися до поля даного тип "запис"?
4. Поясніть, які типи даних можна використовувати для визначення полів у записах?
5. Наведіть приклади введення значень записів з клавіатури?
6. Які особливості введення з клавіатури значень полів перелічуваних і логічних типів?
7. Наведіть приклади виведення записів на екран?
8. Як організувати виведення полів перелічуваного типу?
9. Як утворюються вирази за участю записів?
10. Призначення і формат оператора `with`?

Тема 18. Файли Pascal

План

1. Типи файлів та їх визначення
2. Підготовчі і завершальні дії з файлами
3. Запис і зчитування інформації з файлів
4. Переміщення по файлах
5. Спеціальні операції із фалами
6. Особливості роботи з текстовими файлами
7. Особливості роботи з типізованими файлами
8. Особливості роботи з не типізованими файлами
9. Контрольні запитання

У цій темі ми дізнаємось про призначення і типи файлів, які можна використовувати у Pascal. З'ясуємо , які дії слід виконати для коректної роботи із файлами і яка небезпека є на цьому шляху. Ми побачимо як відбувається запис інформації у файл і зчитування із нього. Вивчимо особливості роботи із текстовими, типізованими і не типізованими файлами.

Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторну роботу №14.

1. Типи файлів та їх визначення

Виконавши у програмі розрахунки ми до сих пір могли вивести їх на екран і там переглянути. Після завершення роботи програми ці результати вже були недоступними. В той же час доволі часто трапляються випадки, коли результати роботи однієї програми є вихідними даними для іншої, тому, якщо не мати засобу зберегти результати і потім прочитати їх, тоді слід усі дані спочатку записати на папері а потім заново вводити їх з клавіатури. Такий підхід є неприйнятним із декількох причин. По-перше, введення даних в ручному режимі є джерелом помилок і, по-друге, навіщо робити зайву роботу, інформація вже була отримана, і була отримана на комп'ютерів, слід тільки навчитися передавати таку інформацію із однієї програми до іншої.
Ще одним аргументом на користь використання файлів є те, що результати роботи програми ми можемо побачити лише на тому комп'ютері, де вона виконувалася. Записавши результати у файл, їх можна розіслати комп'ютерною мережею або надрукувати, що суттєво розширює коло можливих споживачів результатів.

Переконавшись у доцільності використання файлів, з'ясуємо, які типи файлів можна використовувати у Pascal. Практично усі компілятори Pascal підтримують роботу із текстовими, типізованими і не типізованими файлами. Спочатку пригадайте, що називається файлом з точки зору операційної системи. У Pascal файл сприймається так само, як іменована область зовнішньої пам'яті (диска, дискети тощо).
При організації роботи із файлами, кожному імені файлу ставиться у відповідність файлова змінна, тобто, ім'я фала - це його адреса у файловій системі комп'ютера, а файлова змінна - представник цього фала у програмі під час її роботи. Кожен файл містить у собі інформацію будь-якого типу Pascal за винятком файлового (тобто не може існувати "файл файлів"). Особливістю файлів є також те, що їхня довжина ніяк не вказується при створенні, а під час роботи із ними обмежується лише наявною ємністю зовнішньої пам'яті.

Файловий тип у Pascal можна визначити одним із трьох способів:

```
type
  <ім'я типу 1> = text;           {тип - текстовий файл      }
  <ім'я типу 2> = file of <тип>; {тип - типізований файл    }
  <ім'я типу 3> = file;           {тип - не типізований файл }
var
  <ім'я файлової змінної 1>:<ім'я типу 1>; {змінна - текстовий файл      }
  <ім'я файлової змінної 2>:<ім'я типу 2>; {змінна - типізований файл    }
  <ім'я файлової змінної 3>:<ім'я типу 3>; {змінна - не типізований файл }
```

або безпосередньо у розділі визначення змінних

```
var
  <ім'я файлової змінної 1>:text;           {змінна - текстовий файл      }
  <ім'я файлової змінної 2>:file of <тип>; {змінна - типізований файл    }
  <ім'я файлової змінної 3>:file;           {змінна - не типізований файл }
```

Приклади визначення файлів:

```
type
  product = record
              Name :string;
              Code :word;
              Cost :real;
```

```
  end;
  Text80 = file of string[80];
var
  FC :file of char;
  t  :text;
  f  :file;
  FH :Text80;
  P  :file of Product;
```

У наведеному прикладі: `FC`, `FH`, `P` - типізовані файли; `t` - текстовий файл; `f` - не типізований файл.
Зверніть увагу на те, що Pascal не має засобів контролю виду раніше створеного файлу. При визначенні файлової змінної до файла, який був створений раніше, сам програміст повинен прослідкувати про відповідність виду описування характеру змісту файла.

2. Підготовчі і завершальні дії з файлами

Робота із файлами вимагає певної послідовності. Перед початком зчитування інформації із файла або запису інформації до нього слід виконати підготовчі дії. Суть цих дій полягає у тому, що треба встановити відповідність між файловою змінною (фактично це область оперативної пам'яті комп'ютера) і самим файлом на зовнішньому носії інформації (область фізичної зовнішньої пам'яті). Після встановлення такої відповідності слід відкрити канал зв'язку між змінною і файлом, вказавши одночасно і напрямок передачі інформації. Підготовчі дії реалізуються процедурами `assign`, `rewrite`, `append`, `reset`.
Після завершення роботи із файлом слід виконати завершальні дії, суть яких полягає в очищенні тимчасового буферу передачі інформації, закритті каналу передачі інформації і руйнуванні зв'язку між змінною і файлом. Завершальні дії реалізуються процедурами `flush`, `close`.

Розглянемо докладно використання таких процедур.

Процедура `assign`
Реалізує зв'язок між файловою змінною у програмі і конкретним файлом на зовнішньому носії.

Формат:

```
assign(<ім'я файлової змінної>,<ім'я файла>);
```

Нагадаємо, що імена файлів будуються за загальним правилами (див. тему 7). Якщо ім'я логічного пристрою не вказане - вважається що файл розташований на поточному пристрої, якщо не вказується шлях до файла - вважається що файл розташований у поточні папці (там саме, де і основна програма).

Приклади:

```
assign(FC,'A:\Base\Group.dat');
assign(t,'result.txt');
```

Тепер у програмі усі дії, які будуть застосовуватись до змінної `FC` будуть фактично виконуватись з файлом `group.dat` папки `Base` у дисководі `A:`, а усі дії стосовно змінної `t` будуть мати безпосереднє відношення до файла `result.txt` поточної папки.
Остерігайтеся застосовувати процедуру `assign` до вже відкритих файлів.
Відмітимо, що другий параметр може бути іменем послідовного фізичного пристрою (`CON` -

консоль (клавіатура при введенні і монітор при виведенні), `LPT1(PRN),LPT2,LPT3` - паралельні порти, `COM1(AUX),COM2` - послідовні порти, `NUL` - нульовий (неіснуючий) пристрій).

Процедура rewrite

Реалізує відкриття каналу зв'язку у напрямку від файлової змінної у програмі до файла на зовнішньому носії. Після таких дій можна у файл записувати інформацію.

Формат:

```
rewrite(<ім'я файлової змінної>);
```

Сама файлова змінна повинна бути попередньо зв'язана з певним файлом (див. процедуру `assign`).

Приклади:

```
rewrite(FC);
rewrite(t);
```

Після виконання процедур, що наведені у прикладі, файлові змінні `FC` і `t` можна використовувати для запису інформації у відповідні ним файли.
Відкриття файла для запису інформації фактично означає створення нового файла на зовнішньому носії, активізацію спеціальних системних буферів для обміну інформацією і встановлення поточного покажчика на початок файла. При використанні процедури `rewrite` існує небезпека знищення попереднього файла із тим самим ім'ям, адже система, створюючи новий файл псує старий. реальний вихід з цієї ситуації полягає у попередньому створенні резервних копій тих файлів, для яких застосовуються подібні критичні дії.

Процедура append

Дія цієї процедури в цілому є аналогічною до дії процедури `rewrite`, тобто ця процедура відкриває файл для запису. Але при відкритті файла такою процедурою його зміст не очищується, а нова інформація дописується у його кінець. Ще однією специфікою використання процедури `append` є можливість застосовувати її виключно до текстових файлів.

Формат:

```
append(<ім'я файлової змінної>);
```

Приклади:

```
append(t);
```

Процедура reset

Реалізує відкриття каналу зв'язку у напрямку від файла на зовнішньому носії до файлової змінної у програмі. Після таких дій із файла можна зчитувати інформацію.

Формат:

```
reset(<ім'я файлової змінної>);
```

Сама файлова змінна повинна бути попередньо зв'язана з певним файлом (див. процедуру

assign).

Приклади:

```
reset(FC);
reset(t);
```

Після виконання процедур, що наведені у прикладі, файлові змінні `FC` і `t` можна використовувати для зчитування інформації із відповідних ним файлів.
Відкриття файла для запису інформації фактично означає пошук файла на зовнішньому носії, активізацію спеціальних системних буферів для обміну інформацією і встановлення поточного покажчика на початок файла.
Важливо, щоб файл, до якого звертається процедура `reset`, існував на зовнішньому носії, бо інакше виникне помилка. Далі наведений фрагмент коду програми, у якому обробляється небажана ситуація. Суть фрагменту полягає у тому, що перед процедурою відключається контроль введення-виведення, а після її виконання включається і аналізується, виникла помилка чи ні. У випадку помилки програма примусово завершує свою роботу за допомогою процедури `Halt`.

```
...
assign(f,'myfile.dat');{Зв'язати файлову змінну із файлом}
{$I-}                  {Директива відключення контролю введення-виведення}
reset(f);              {Відкрити файл для зчитування }
{$I+}                  {Директива включення контролю введення-виведення}
if IOResult <> 0       {Аналіз значення функції результату введення-виведення}
  then Halt(1);        {У випадку помилки примусово завершити роботу }
...
```

Визначити причину помилкового зчитування із файла можна проаналізувати код помилки. Можливі помилки наведені у додатку В і, зокрема, коди, які повертає функція IOResult.

Процедура `Close`
Реалізує закриття файла після завершення усіх операцій із ним. Під закриттям файла слід розуміти остаточний запис у файл усієї інформації із системного буфера, якщо інформація записувалась у файл, закриття каналу зв'язку між файлом і файловою змінною та ліквідацію системних буферів обміну інформацією.

Формат:

```
close(<ім'я файлової змінної>);
```

Приклади:

```
close(FC);
close(t);
```

Використання процедури `flush` доцільно лице при записі інформації у файл і дозволяє очистити системний буфер обміну інформації, тобто реально записати дані на диск. Ця процедура не розриває зв'язок між файловою змінною і файлом, тобто після цього ще можна виконувати операції запису в файл.

3. Запис і зчитування інформації з файлів

Запис або зчитування інформації з файла даних повинно виконуватися тільки для вже

відкритих з цією метою файлів. Ці операції потребують підвищеної уваги з боку програміста, оскільки вони є потенційним джерелом помилок.

Основними процедурами запису-зчитування у Pascal є `write` і `read`. Для текстових файлів додатково можна використовувати процедури `writeln` і `readln`. Для не типізованих файлів використовуються процедури `BlockWrite` і `BlockRead`.

3.1. Зчитування інформації з файла

Процедура `read`
Реалізує зчитування з текстового або типізованого файла порції даних і присвоювання відповідній змінній або списку змінних.

Формат:

```
read(<ім'я файлової змінної>,<список змінних>);
```

До файлової змінної попередньо повинна бути застосована процедури `assign` і `reset`. У списку змінних розміщуються їхні ідентифікатори. Цим змінним будуть присвоюватись значення, зчитані з файла.
Виконання процедури `read` відбувається таким чином. Починаючи з поточної позиції покажчика файла, будуть зчитуватися значення, які містяться у файлі. Кожне зчитане значення буде присвоюватись черговій змінній у списку. Після кожного акту зчитування покажчик файла буде зміщуватися на наступну позицію. Якщо в процесі виконання процедури `read` поточний покажчик буде встановлено на позицію, яка не містить інформацію (тобто досягнуто кінець файла), то зчитування буде припинено.

На рис. 18.1 наведена імітація зв'язування файлової змінної з файлом, відкриття файла для зчитування, зчитування інформації з типізованого файла і закриття файла. Змінні `a,b,c,d` - символьного типу. Змінна `f` - файл символів.

<table>
<tr><th>Програма</th><th>Магнітний диск
(зміст файла "data")</th><th colspan="5">Оперативна пам'ять</th></tr>
<tr><td rowspan="2">assign(f,'data');
reset(f);
read(f,a,b,c,d);
close(f);</td><td rowspan="2">| 3 | F | D | 5 |</td><td>f</td><td>a</td><td>b</td><td>c</td><td>d</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Рис. 18.1. Зчитування інформації з файла

3.2. Запис інформації у файл

Процедура `write`
Реалізує запис даних у текстовий або типізований файл.

Формат:

```
write(<ім'я файлової змінної>,<список виведення>);
```

До файлової змінної попередньо повинна бути застосована процедури `assign` і `rewrite`. При виконанні процедури `write` значення чергового виразу (тільки для текстових файлів)

або змінної зі списку виведення записується у файл у місце, яке помічене поточним покажчиком. Після цього поточний покажчик переміщується на одну позицію далі і аналогічні дії виконуються для наступного елементу списку виведення.

На рис. 18.2 наведена імітація запису в типізований файл.

Програма	Магнітний диск (зміст файла "data")	Оперативна пам'ять				

assign(f,'data');
rewrite(f);
write(f,a,b,c,d);
close(f);
		f	a	b	c	d

			3	F	D	5

Рис. 18.2. Запис інформації у файл

Приклад 18.1. Створення текстового і типізованого файла.

Завдання. Створити програму, яка створює на зовнішніх носіях текстовий і типізований файл. Записати у ці файли тестову інформацію і закрити їх.

Рішення. На рис. 18.3. наведена програма, яка містить повний набір процедур, які необхідні для відкриття файла (`assign`, `reset`, `rewrite`), запису (`write`) і закриття (`close`) файлів та пояснення до її роботи.

Програма	Магнітний диск	Оперативна пам'ять

<pre>**Program** Pr_18_01;
var
  i,j :integer;
  ft :text;
  fi :**file of** integer;
BEGIN
  i:=2;
  j:=3;
  assign(ft,'A:\textfile.txt');
  assign(fi,'intfile.dat');
  rewrite(ft);
  rewrite(fi);
  write(ft,'Its text file ');
  write(ft,i+j);
  write(fi,j,i);
  close(ft);
  close(fi);
END.</pre>
	Поточний каталог
	файл "intfile.dat"
	
	Гнучкий диск
	файл "textfile.txt"	

Рис. 18.3. Створення файлів і запис інформації

Зверніть увагу, що числові дані записуються по байтах, тому ви не побачите самі числа переглянувши зміст типізованого файла.

Перегляньте роботу готової програми.

4. Переміщення по файлах

При відкритті файла поточний покажчик автоматично встановлюється на початок файла. При зчитуванні або при запису даних покажчик переміщується. Доволі часто слід мати можливість визначити позицію цього покажчика (поточну або кінцеву), а також перемістити його у певну позицію. Для реалізації таких можливостей у Pascal є дві процедури (`seek, truncate`) і три функції (`FileSize, FilePos, EoF`), які працюють із поточним покажчиком.

Процедура `seek`
Дозволяє змінити місце поточного покажчика, встановивши його на елемент файла із вказаним номером.

Формат:

```
seek(<ім'я файлової змінної>,<номер елемента>);
```

Номер елемента має тип `longint` і вказує на яку позицію слід перемістити покажчик. Усі подальші процедури зчитування і запису у файл будуть виконуватись, починаючи із встановленої позиції.

На рис. 18.4. наведена імітація використання процедури переміщення вздовж файла.

Програма	Файл на диску								Оперативна пам‘ять		

read(f,a);
seek(f,4);
read(f,b,c);
seek(f,2);
read(f,a);
	1	2	3	4	5	6	7	8	a	b	c

Рис. 18.4. Переміщення вздовж файла

Процедура `truncate`
Призначена для знищення кінцевої частини файла, починаючи із поточної позиції покажчика.

Формат:

```
truncate(<ім'я файлової змінної>);
```

На рис. 18.5. наведена імітація використання процедури знищення кінцевої частини файла.

Програма	Файл на диску							
seek(f,4); truncate(f);	1	2	3	4	5	6	7	8

Рис. 18.5. Знищення кінцевої частини файла

Функція `FileSize`
Визначає загальну довжину файла у байтах. Результат функції має тип `longint`. Для

типізованих файлів за допомогою цієї функції можна визначити кількість елементів у файлі, для цього слід розділити довжину файла у байтах на кількість байт, потрібних для збереження одного елемента відповідного типу. Для стандартних типів даних їхня довжина відома (див. тему 11). Для структурованих даних зручно скористатися стандартною функцією `SizeOf`.

Формат:

FileSize(<ім'я файлової змінної>);

Функція FilePos
Визначає позицію положення поточного покажчика. Результат функції має тип `longint`.

Формат:

FilePos(<ім'я файлової змінної>);

Функція EoF
Визначає чи знаходиться покажчик у позиції кінця файла. Результат функції має тип `boolean`. Функція повертає логічне значення `true`, якщо поточний покажчик знаходиться у кінці файла і значення `false` - у протилежному випадку.

Формат:

EoF(<ім'я файлової змінної>);

На рис. 18.6 наведено приклади використання функцій `FileSize, FilePos, EoF`.

Програма	Файл на диску	Екран монітор

writeln(FileSize(f));
writeln(FilePos(f));
writeln(EoF(f));
seek(FileSize(f));
writeln(EoF(f));
	Т е к с т о в и й ф а й л
	⇧	

Рис. 18.6. Визначення параметрів файла та позиції покажчика

Процедури `seek` і `truncate` а також функції `FileSize, FilePos, EoF` дозволяють створювати ефективні алгоритми обробки інформації з файлів.

5. Спеціальні операції із файлами

Для отримання доступу до можливостей операційної системи по роботі із файлами можна скористатися спеціальними процедурами і функціями. Ми не будемо детально розглядати роботу з ними, а обмежимось лише форматами та призначенням.

Формат функції	**Призначення**
`Erase(f);`	- видаляє файл, який зв'язаний із файловою змінною `f`.
`Rename(f,NewName);`	- надає файлові, який зв'язаний із файловою змінною `f`

	нове ім'я - NewName.
ChDir(Dir);	- встановлює новий поточний каталог - Dir.
MkDir(Dir);	- створює новий каталог - Dir.
RmDir(Dir);	- видаляє пустий каталог - Dir.
GetDir(Drive,Dir);	- визначає ім'я поточного каталогу у пристрої Drive. Пристрій Drive може мати такі значення: 0 - поточний; 1 - дисковід А; 2 - дисковід В; 3 - диск С). Ім'я поточного каталогу присвоюється змінній Dir.
IOResult	- визначає код помилки введення-виведення при роботі із файлом. Для роботи функції необхідно, щоб була відключена директива компілятору про контроль операцій введення-виведення {$I-}.
DiskFree(Drive);	- визначає об'єм вільного місця у байтах на диску Drive. Можливі значення параметру Drive аналогічні функції GetDir.
DiskSize(Drive);	- визначає об'єм диску Drive у байтах . Можливі значення параметру Drive аналогічні функції GetDir.
FindFirst(Path,Attr,f);	- відшукує перший файл, що відповідає шаблону Attr у каталогу Path. Ім'я знайденого файла зберігається у змінній f.
FindNext(f);	- відшукує наступний файл, відповідно до умов попередньої процедури.
GetFTime(f,Time);	- визначає дату і час створення файла.
SetFTime(f,Time);	- встановлює нову дату і час створення файла.
GetFAttr(f,Attr);	- визначає атрибути файла.
SetFAttr(f,Attr);	- встановлює атрибути файла.
FSearch(Path,DirList);	- відшукує файл у списку каталогів.
FSplit(Path,Dir,Name,Ext);	- розщеплює повне ім'я файла на шлях, ім'я і тип.
FExpand(Path);	- доповнює скорочене ім'я файла до повного.

6. Особливості роботи з текстовими файлами

Коли ми переглядаємо текстовий файл у будь-якому редакторі, від найпростішого Notepad до потужного Microsoft Word, ми бачимо, що текст складається із послідовності символів, рядків, параграфів, розділів тощо. Тобто текстовий файл має логічну структуру і представляється у вигляді принаймні кількох рядків. У той же час файл, який зберігається на диску має послідовну структуру, його можна уявити, як дуже довгий ланцюг символів. Тоді виникає питання, як перейти від послідовної структури до логічно підпорядкованої. Для цього були ведені спеціальні символи, які не відображаються і не друкуються, але вказують на ті місця, де слід перейти до нового рядка, абзацу, розділу.

На рис. 18.7 наведено приклад того, як простий текстовий файл відображається на екрані і як він записаний у файлі.

Відображення файла на екрані

```
Текстовий
файл у
три рядки
```

Послідовність символів у файлі на диску

Т	е	к	с	т	о	в	и	й	#13	#10	ф	а	й	л	#0	у	#13	#10	т	р	и	#0

Рис. 18.7. Представлення текстових файлів

Як видно з рисунку, після кожного рядка ставляться два спеціальні символи: #13 - символ повернення каретки і #10 - символ переходу до наступного рядка. Сукупність таких символів утворює символ переходу до нового рядка, який у Pascal позначається ім'ям EoLn (*End of Line*).
Весь файл завершується символом кінця файла EoF (*End of File*) - #26.

Зверніть також увагу, що кожен символ, навіть якщо він ніяк не відображається на екрані має свій код, так символ "пробіл" має код "0" і вставляється всюди там, де при перегляді ви бачите пробіл.

Окрім вищезазначених процедур і функцій Seek, Truncate, FileSize, FilePos, EoF, для текстових файлів можна застосовувати функції пошуку кінця рядка (SeekEoLn) і кінця файла (SeekEoF).

Функція SeekEoLn
Визначає чи знаходиться покажчик у позиції кінця рядка. Результат функції має тип boolean. Функція повертає логічне значення true, якщо поточний покажчик знаходиться у кінці рядка і значення false - у протилежному випадку.

Формат:

```
SeekEoLn(<ім'я файлової змінної>);
```

Функція також видає значення true, якщо покажчик встановлено на завершальні у рядку пробіли або символи табулювання.

Функція SeekEoF
Визначає чи знаходиться покажчик у позиції кінця текстового файла. Результат функції має тип boolean. Функція повертає логічне значення true, якщо поточний покажчик знаходиться у кінці файла і значення false - у протилежному випадку.

Формат:

```
SeekEoLn(<ім'я файлової змінної>);
```

Текстові файли припускають лише один вид операції введення виведення - в одному сеансі роботи із файлом ви можете або тільки записувати у нього інформацію, або тільки зчитувати.

Запис інформації у текстовий файл. Окрім стандартної процедури write для текстових файлів можна застосовувати процедуру writeln, особливість якої полягає у тому, що вона при виведенні автоматично вставляє символи завершення рядка (пару символів з кодами #13 і #10). Перед записом інформації у файл, відповідна файлова змінні повинна бути зв'язана із конкретним файлом (процедура assign) і відкрита для нового запису (процедура rewrite)

або для допису в кінець існуючого файла (процедура append).

Процедури виведення (write, writeln) передбачають автоматичне перетворення скалярних типів даних у послідовність символів. Крім того, для покращення представлення виведеної інформації доцільно використовувати формати виведення, розглянуті у темі 11.

Рис. 18.8 ілюструє використання процедур write і writeln при записі інформації у текстовий файл.

Програма

```
write(f,'D = ',d2:6:2);
writeln(' мм');
writeln(D>=100);
write('Київ':6);
```

Текстовий файл

Рис. 18.8. Запис інформації у текстовий файл

Зчитування інформації з текстового файла. Окрім стандартної процедури read для текстових файлів можна застосовувати процедуру readln, особливість якої полягає у тому, що вона при зчитуванні порції даних переходить до нового рядка, навіть у випадку, коли у поточному рядку ще залишилися дані. Процедура readln без параметрів дозволяє зчитати символ кінця рядка (пару символів з кодами #13 і #10).

Слід пам'ятати, що зчитування з текстового файла може бути причиною великої кількості помилок, що пов'язані як із наявністю і типом самого файла, так і зі змістом файла. Тому така дія повинна виконуватися особливо обережно.

Зчитування числової інформації із текстового файла продемонструємо на прикладі програми, яка зчитує із тестового файла значення цілочислової матриці розміром 2x3.

Приклад 18.2. Зчитування числової інформації з текстового файла

Завдання. Нехай у текстовому файлі 'data.txt' міститься числова інформація про матрицю розміром 2x3, причому елементи матриці у рядку розділені між собою пробілам, а кожен рядок матриці представлений окремим рядком файла. Створити програму, яка читатиме інформацію з файла і виводитиме її для контролю на екран монітора.

Рішення. Маючи відомості про кількість елементів масиву і про форму запису фрагмент зчитування даних із файла можна реалізувати через вкладені цикли for.

Текст програми:

```
program Pr_18_02;
var
  t:text;
  m:array[1..2,1..3] of integer;
  i,j:integer;
begin
  assign(t,'data.txt'); {зв'язати файлову змінну t з файлом data.txt}
  {$I-}      {відключити контроль помилок введення-виведення}
  reset(t);  {відкрити файл для зчитування інформації}
```

```
  {$I+}      {включити контроль помилок введення-виведення  }
  if IOResult <> 0
    then begin     {обробка помилки відкриття файла }
       writeln(' Помилка при відкритті файла! ');
       readln;
       Halt(1);    end;
  {зчитування даних із текстового файла}
  for i:=1 to 2 do begin
    for j:=1 to 3 do
      read(t,m[i,j]);
    readln(t);
  end;
  close(t);  {закриття файла}
  {виведення контрольної інформації на екран монітора}
  for i:=1 to 2 do begin
    for j:=1 to 3 do
      write(m[i,j]:6);
    writeln;
  end;
  readln;
end.
```

Результат виконання програми:

```
   5     6     1
   2     9     3
```

Перегляньте роботу готової програми.

Часто, зчитуючи інформацію з текстового файла, ми наперед не знаємо загальну кількість рядків, з якої він складається. У такому випадку зручно використати цикл `while` і виконувати його до тих пір, поки не зустрінеться символ кінця файла. Використання такого прийому наведено у прикладі 18.3.

Приклад 18.3. Зчитування інформації з текстового файла невідомої довжини

Завдання. Організувати зчитування інформації із текстового файла і виведення його на екран.

Рішення. Основу програми складає цикл `while`, який виконуватиметься до моменту зчитування символу кінця файла. Всередині циклу перший оператор зчитує один рядок із файла, а другий оператор - виводить цей рядок на екран.

Текст програми:

```
program pr18_03;
var
  t:text;
  s:string;
BEGIN
  assign(t,'textfile.txt');
  {$I-}
  reset(t);
  {$I+}
  if IOResult<>0 then Halt(1);
  while not EoF(t) do begin {виконувати до кінця файла}
    readln(t,s);  {прочитати рядок із файла}
    writeln(s);   {вивести рядок на екран}
  end;
  readln;
  close(t);
```

```
END.
```

Перегляньте роботу готової програми.

Ще одна ситуація, яка доволі часто трапляється при зчитуванні інформації із тестового файла,- виділення із послідовності рядків окремих слів. У такому випадку слід формувати слова у межах рядка за наявністю пробілів між ними. Виділення слів з текстового файла проілюстровано у прикладі 18.4.

Приклад 18.4. Формування масиву слів з текстового файла невідомої довжини

Завдання. Організувати зчитування інформації із текстового файла і формування масиву слів. Словом вважати послідовність сімволів, розділених одне від одного принаймні одним пробілом.

Рішення. Основу програми складають два цикли `while`. Зовнішній цикл виконуватиметься до моменту зчитування символу кінця файла, а внутрішній - до моменту зчитування символів кінця рядка. Якщо при зчитуванні чергового символу з'ясовується, що це пробіл, відбувається перехід до формування нового елементу масиву рядків. Така ж дія виконується і при переході до зчитування нового рядка. Наприкінці програми відбувається у циклі `for` виведення масиву слів на екран. Відзначимо, що подана програма є спрощеною і не враховує того, що слова можуть відділятися одне від одного не одним, а кількома пробілами.

Текст програми:

```
program pr18_04;
var
  t:text;
  ch:char;
  s:string;
  a:array[1..100] of string;
  i,j:integer;
BEGIN
  assign(t,'textfile.txt');
  {$I-}
  reset(t);
  {$I+}
  if IOResult<>0 then Halt(1);
  i:=1;
  while not EoF(t) do begin    {виконувати до кінця файла}
    while not EoLn(t) do begin {виконувати до кінця рядка}
      read(t,Ch);              {прочитати символ із файла}
      if ch<>' '               {якщо цей символ не є пробілом}
        then a[i]:=a[i]+ch     {додати його до поточного рядка}
        else inc(i);           {перейти до формування нового рядка}
    end;
    readln(t);             {прочитати із файла символи кінця рядка}
    inc(i);                {перейти до формування нового рядка}
  end;
  for j:= 1 to i do {вивести масив рядків на екран}
   writeln(a[j]);
  readln;
  close(t);
END.
```

Перегляньте роботу готової програми.

7. Особливості роботи з типізованими файлами

Нагадаємо, що до типізованих належать файли, усі елементи яких мають наперед визначений тип. Тому, якщо у текстових фалах зміст розглядається як послідовність символів, то у типізованих файлах зміст є послідовністю записів певного типу. Одиницею вимірювання такого набору є сам запис, а його довжину у байтах можна визначити за допомогою стандартної функції `SizeOf`.

Важливою особливістю типізованих файлів є те, що вони дозволяють в одному сеансі роботи з ними і записувати і зчитувати інформацію. Ця обставина дуже часто відіграє вирішальну роль у виборі, якому типові файлів, текстовому або типізованому, надати перевагу при вирішенні конкретної задачі.

Ще однією особливістю типізованих файлів є те, що усі записи у них мають однакову довжину, і якщо реальна інформація записана у них є коротшою, відбувається доповнення її пробілами. Тобто, якщо елементом типізованого файла є рядок довжиною 20-и символів, а реальний зміст елемента складається із 3-х символів, то 17-ть пробілів будуть автоматично дописані у файл. З огляду на цю обставину, слід ретельно планувати структуру запису, оскільки недоліки цього процесу призведуть до надлишкових витрат дискового простору на носії інформації.

При відкритті типізованого файла покажчик файла встановлюється на початок першого файла, який має номер "0", тобто номер фізичного запису є на одиницю більшим від номеру логічного запису. Дуже обережно слід поводитись із процедурою переміщення вздовж файла `Seek`, оскільки невірне позиціювання не викликає до помилок введення-виведення з точки зору операційної системи, але суттєво впливає на зміст інформації, що зчитується.

Дуже часто типізовані файли використовуються для організації роботи із базами даних, де переваги цих фалів є незаперечними. У прикладі 18.5 наведено програму створення і обробки бази даних металорізальних верстатів. Відзначимо, що для скорочення місця, структура самої бази даних суттєво спрощена.

Приклад 18.5. Створення і використання бази даних за допомогою типізованого файла

Завдання. Організувати створення і обробку інформації бази даних токарних металорізальних верстаів *[Справочник технолога-машиностроителя. В 2-х т., Т.2 / Под. ред. А.Г. Косиловой и Р. К. Мещерякова.- 4-е изд., перераб. и доп.- М.: Машиностроение, 1986. 496 с., ил.]*.

Рішення. Основою бази даних є тип `Machine`, який містить записи, відповідно до структури інформації про токарні МРВ. Для заповнення і доповнення бази даних використовується процедура `AddRec`, яка організовує зчитування з клавіатури відповідних параметрів верстата і записує їх у типізований файл "`tokar.dat`". Для виведення поточного запису на екран створена процедура `OutRec`, а виведення усієї бази даних виконується за допомогою процедури `OutAllRec`.

Текст програми:

```
Program Pr18_05;
type Machine = record
                 Model        :string[10];{Модель верстата               }
                 DMax         :integer;   {Найбільший діаметр заготовки    }
                 LMax         :integer;   {Найбільша довжина заготовки     }
                 SpindlMin    :real;      {Найм. частота обертання шпинделя}
                 SpindlMax    :real;      {Найб. частота обертання шпинделя}
                 NumbSpeed    :byte;      {Кількість швидкостей шпинделя   }
                 FeedProdMin  :real;      {Найменша поздовжня подача       }
                 FeedProdMax  :real;      {Найбільша поздовжня подача      }
```

```
                  FeedPoperMin :real;      {Найменша поперечна подача       }
                  FeedPoperMax :real;      {Найбільша поперечна подача      }
                  NumbFeed     :byte;      {Кількість ступенів подач        }
                  MainPower    :real;      {Потужність головного приводу    }
                  L            :integer;   {Довжина верстата                }
                  B            :integer;   {Ширина верстата                 }
                  H            :integer;   {Висота верстата                 }
                  Mass         :integer;   {Маса верстата                   }
                end; { Machine }
var
  MashFile  :file of Machine;{Змінна для файла з записами  }
  M         :Machine;        {Змінна для доступу до запису }
  i         :byte;
  Answer    :char;

procedure OutputRec;
   {Процедура виведення поточного запису на екран}
begin
  read(MashFile,M);
  with M do begin
    writeln('Модель верстата                  ',Model );
    writeln('Найбільший діаметр заготовки     ',DMax );
    writeln('Найбільша довжина заготовки      ',LMax );
    writeln('Найм.частота обертання шпинделя ',SpindlMin );
    writeln('Найб.частота обертання шпинделя ',SpindlMax );
    writeln('Кількість швидкостей шпинделя    ',NumbSpeed );
    writeln('Найменша поздовжня подача        ',FeedProdMin );
    writeln('Найбільша поздовжня подача       ',FeedProdMax );
    writeln('Найменша поперечна подача        ',FeedPoperMin );
    writeln('Найбільша поперечна подача       ',FeedPoperMax );
    writeln('Кількість ступенів подач         ',NumbFeed );
    writeln('Потужність головного приводу     ',MainPower );
    writeln('Довжина верстата                 ',L );
    writeln('Ширина верстата                  ',B );
    writeln('Висота верстата                  ',H );
    writeln('Маса верстата                    ',Mass );
  end;
end; {procedure OutputRec}

procedure OutAllRec;
   {Процедура виведення на екран усіх записів файла}
begin
  Seek(MashFile,0); {Встановлення покажчика на перший запис}
  writeln(' Виведення всієї бази даних на екран ');
  while (not EoF(MashFile)) do OutputRec;
end; {procedure OutAllRec}

procedure AddRec;
   {Процедура зчитування з клавіатури і додавання запису у файл}
begin
   writeln('Заповніть відомості:');
   with M do begin
     write('Модель верстата                   ');readln(Model );
     write('Найбільший діаметр заготовки      ');readln(DMax );
     write('Найбільша довжина заготовки       ');readln(LMax );
     write('Найм. частота обертання шпинделя ');readln(SpindlMin );
     write('Найб. частота обертання шпинделя ');readln(SpindlMax );
     write('Кількість швидкостей шпинделя     ');readln(NumbSpeed );
     write('Найменша поздовжня подача         ');readln(FeedProdMin );
     write('Найбільша поздовжня подача        ');readln(FeedProdMax );
     write('Найменша поперечна подача         ');readln(FeedPoperMin);
     write('Найбільша поперечна подача        ');readln(FeedPoperMax);
     write('Кількість ступенів подач          ');readln(NumbFeed);
```

```
      write('Потужність головного приводу     ');readln(MainPower);
      write('Довжина верстата                 ');readln(L );
      write('Ширина верстата                  ');readln(B );
      write('Висота верстата                  ');readln(H );
      write('Маса верстата                    ');readln(Mass);
      write(MashFile,M);
    end;
end; {procedure AddRec}

BEGIN
   {Програма Pr18_05 містить приклади роботи з базою даних}
   assign(MashFile,'tokar.dat'); { створюється новий файл }
   rewrite(MashFile);            { з ім'ям tokar.dat }

   { Початкове заповнення бази даних }
   repeat
     AddRec;
     write('Додати ще один запис (Y/N) і');readln(Answer);
   until not((Answer = 'Y') or (Answer = 'y'));
   writeln('Створення бази даних завершено. Натисніть <Enter>.');
   readln;

   { Виведення змісту запису за його номером }
   repeat
     write('Введіть номер запису бази даних.');readln(i);
     seek(MashFile,i-1);
     if (not EoF(MashFile))
       then OutputRec;
     write('Вивести ще один запис (Y/N) і');readln(Answer);
   until not((Answer = 'Y') or (Answer = 'y'));
   writeln('Натисніть клавішу <Enter>.');
   readln;

   { Виведення всієї бази даних на екран }
   OutAllRec;
   close(MashFile);
END.
```

Перегляньте роботу готової програми.

8. Особливості роботи з не типізованими файлами

Не типізований файл розглядається в Pascal як сукупність символів або байтів. Таке представлення затирає різницю між файлами незалежно від типу їх оголошення. При програмуванні це призводить до того, що будь-який файл (текстовий або типізований) можна відкрити і почати з ним працювати, як з не типізованим набором даних. Для не типізованих файлів немає потреби витрачати час на перетворення типів і пошук управляючих послідовностей, необхідно лише зчитати зміст файла у визначену область пам'яті. Не типізовані файли є файлами прямого доступу, що вказує на можливість одночасного виконання операцій зчитування і записування.

При відкритті файла процедурами `reset` або `rewrite` додатково другим параметром вказується довжина запису файла на сеанс роботи. Використання для базових операцій введення-виведення з не типізованими файлами стандартних процедур `read` і `write` не може дати високої швидкості передачі даних. Тому тільки для не типізованих файлів у Pascal введені дві процедури, які підтримують операції введення-виведення з більшою швидкістю.

Процедура `BlockRead` зчитує з файла визначену кількість блоків у буфер, що призначений для накопичування інформації з файла.

Формат:

```
BlockRead (<файлова змінна>,<буфер>,<Count>|,<Result>|);
```

де
`Count` - кількість блоків, що будуть зчитуватись при одному звертанні до диска;
`Result` - необов'язковий параметр, який містить після викликання процедури реальну кількість зчитаних записів.

Процедура `BlockWrite` призначена для швидкого передавання у файл визначеної кількості записів з буфера.

Формат:

```
BlockWrite (<файлова змінна>,<буфер>,<Count>|,<Result>|);
```

Окрім високої швидкості обміну інформацією, переваги наведених процедур полягають у можливості користувача самостійно визначати об'єм буфера для файлових операцій. Ця можливість відіграє вирішальну роль у тих задачах, де необхідно жорстке планування ресурсів.

9. Контрольні запитання

1. Які переваги надає використання файлів у програмах?
2. Як сприймається файл у Pascal і що таке файлова змінна?
3. Як визначаються файли у Pascal?
4. Які підготовчі і завершальні дії треба виконати при роботі з файлами у Pascal?
5. Призначення, формат і приклади процедури `assign`.
6. Призначення, формат і приклади процедури `rewrite`.
7. Призначення, формат і приклади процедури `append`.
8. Призначення, формат і приклади процедури `reset`.
9. Як організувати контроль коректності відкриття файла процедурою `reset`?
10. Призначення, формат і приклади процедури `close`.
11. Призначення, формат і приклади процедури `read`.
12. Призначення, формат і приклади процедури `write`.
13. Призначення, формат і приклади процедури `seek`.
14. Призначення, формат і приклади процедури `truncate`.
15. Призначення, формат і приклади функції `FileSize`.
16. Призначення, формат і приклади функції `FilePos`.
17. Призначення, формат і приклади функції `EoF`.
18. Які процедури і функції відносять до спеціальних про роботі із файлами?
19. Які особливості представлення текстових файлів?
20. Які додаткові функції можна використовувати при роботі з текстовими файлами?
21. Наведіть приклади запису інформації у текстовий файл.
22. Наведіть приклади зчитування числових даних з текстового файла.
23. Наведіть приклади зчитування слів з текстового файла.
24. Які особливості роботи з типізованими файлами у Pascal ви знаєте?
25. Які особливості роботи з не типізованими файлами у Pascal ви знаєте?

Тема 19. Модулі Pascal

План

1. Поняття про модулі та їхнє призначення
2. Стандартні модулі. Використання їхніх елементів
3. Модулі користувача. Структура, створення, тестування, використання
4. Контрольні запитання

У цій темі ви дізнаєтесь про те, які використовувати потужний механізм роботи з модулями Pascal. Модулі вам обов'язково знадобляться при створення програм середнього і високого рівня складності. Саме у цій темі наведені відомості про стандартні модулі та їхні елементи, про створення і використання модулів користувача.
Закріпити отримані знання ви зможете давши відповіді на контрольні запитання наприкінці теми, а також виконавши лабораторну роботу №15.

1. Поняття про модулі та їхнє призначення

Створюючи певні елементи у програмі (типи, константи, змінні, процедури, функції) ми маємо змогу використовувати їх лише всередині цієї програми. Тому, якщо виникає потреба скористатися одним із них в іншій програмі, слід мати вихідний код програми і скопіювати ці елементи. Очевидно, що такий шлях є хибним, адже не завжди є сам початковий код і, відверто кажучи, не хотілось би копіювати цей елемент. На цьому шляху нас чекають можливі помилки, іноді - невиправні, ми можемо скопіювати не все, що потрібно для роботи із певним елементом, або набагато більше ніж було потрібно, перекрити власні елементи у випадку збігу імен, помилитися у місці вставлення тощо. Крім того, при такому копіюванні відбувається дублювання інформації, що обов'язково призведе до невиправданих витрат дискового простору. Отже, ми одразу відкидаємо такий спосіб запозичення елементів із інших програм і будемо шукати інші шляхи вирішення цієї проблеми.

Аналогом вдалого рішення у повсякденному житті є бібліотеки, адже не обов'язково мати дома усі книжки, особливо у випадку, коли ці книжки бувають потрібні одноразово чи епізодично. Достатньо записатися у бібліотеку і користуватися усім тим, що є у ній. Так само можуть користуватися усіма її книжками і інші користувачі.

І у програмуванні є можливість створити бібліотеку, складові елементи якої будуть доступними для інших програм. Причому в даному випадку, можна навіть не мати початкового коду програми і нічого не знати про алгоритм конкретного елемента, - достатньо лише знати їхні імена і параметри виклику. Таку бібліотеку можна відкомпілювати автономно (незалежно від основної програми), але виконати бібліотеку самостійно неможливо (так саме публічна бібліотека сама книжок не читає, читають - користувачі).

Бібліотеки компонентів у Pascal називають модулями (англ. - *unit*). Кожен модуль має власне ім'я. Для того, щоб отримати можливість використовувати усі компоненти модуля необхідно підключитися до нього (записатися у бібліотеку). Робиться це просто - достатньо лише у розділі `uses` основної програми вказати ім'я цього модулю, наприклад

```
uses Crt, Graph, PVMech;
```

Підключення до модуля дозволяє, але не вимагає використовувати його елементи. Умовою компіляції програми є те, що компілятор повинен знати, де знаходиться цей модуль. Цікаво, що до модуля може підключатися не тільки програма, і й інший модуль. Таким чином можна створювати ланцюги підключених модулів і усі вони будуть доступними автономно.

Усі модулі поділяються на дві групи:

1. ***Стандартні модулі*** - ті, які поставляються разом із компілятором;
2. ***Модулі користувача*** - ті, які створюються власне будь-яким програмістом.

Використання елементів зі стандартних модулів і модулів користувача не має ніякої різниці, головне мати сам модуль.

2. Стандартні модулі. Використання їхніх елементів

До інсталяційного пакету Pascal входить велика кількість модулів. Вони мають різне призначення і допомагають у вирішенні певних задач. Звичайно, що усі модулі та їхні компоненти ми розглянути не в змозі. Проведемо короткий огляд лише основних (системних) модулів Pascal, які складають основу усіх програм та інших модулів.

Так до складу бібліотеки модулів `turbo.tpl` входять п'ять модулів: `System`, `Crt`, `Dos`, `Printer`, `Overlay`. У файлі `graph.tpu` міститься модуль `Graph`.

Використання деяких процедур і функцій базових модулів наведено у прикладах програм до різних тем, тому тут обмежимось лише загальним оглядом призначення модулів та їхніх компонентів.

2.1. Призначення базових модулів

Модуль `System`

Містить усі процедури і функції стандартної мови Pascal та вбудовані процедури і функції Turbo Pascal (основні із цих елементів були розглянуті у темі 13). Цей модуль автоматично підключаєтся до кожної програми і тому немає потреби вказувати його ім'я у розділі `uses`.

Модуль `Crt`

Містить набір типів, констант, змінних, процедур і функцій, які розширюють можливості роботи у текстовому режимі. призначення основних процедур і функцій цього модулю будуть розглянуті далі.

Модуль `DOS`

Містить засоби доступу до усіх можливостей операційної системи MS-DOS, зокрема:

- процедури роботи з системними перериваннями - `Intr, MsDos, GetIntVec, SetIntVec, SwapVectors`;
- процедури обробки системної дати і часу - `GetDate, SetDate, GetTime, SetTime`;
- сервісні процедури і функції - `DosVersion, EnvCount, EnvStr, GetEnv, FSplit, FExpand, FSearch, GetVerify, GetCBreak`;
- процедури роботи із логічними дисками і файлами - `DiskSize, DiskFree, GetFAttr, SetFAttr, GetFTime, SetFTime, PackTime, UnpackTime, FindFirst, FindNext`;
- процедуру запуску інших програм - `Exec`.

Модуль `Printer`

Забезпечує зв'язок між процесором і матричним принтером. Для виведення інформації на

папір використовується стандартний логічний файл `Lst`, яка вказується першою у процедурах виведення, наприклад:

```
writeln(Lst,'Виведення інформації на папір');
```

Модуль `Overlay`

Містить набір елементів для організації сегментів програм. Такі дії можуть бути викликані тим, що створена програма є занадто великою. І хоча об'єми оперативної пам'яті сучасних комп'ютерів суттєво перевищують потреби DOS-програм, організація оверлеїв є важливим засобом структурування великих програм.

Модуль `Graph`

Містить великий набір елементів для роботи у графічному режимі. Докладно можливості модуля `Graph` будуть розглянуті у темі 20.

2.2. Компоненти модулю `Crt`

Усі процедури і функції модуля можна умовно розділити на такі групи:

- управління зображенням;
- управління текстовим курсором;
- управління клавіатурою;
- управління звуком.

До процедур управління зображенням відносяться:

`ClrScr` - процедура очищення екрану або поточного вікна (встановленого процедурою `window`) і переведення курсору у позицію (1,1) екрану або поточного вікна.

`ClrEoL` - процедура видалення усіх символів у рядку, починаючи від позиції курсору і до кінця рядка.

`DelLine` - процедура видалення всього рядка, у якому знаходиться курсор. Усі рядки, що знаходились нижче, переміщуються на один рядок вверх.

`InsLine` - процедура вставлення пустого рядка у позиції курсору. Усі рядки, що знаходились у поточній позицій і нижче, зміщуються на один рядок униз.

`TextColor(Color)` - процедура встановлення кольору, яким виводитимуться символи.

`TextBackGround(Color)` - процедура встановлення кольору фону, на якому виводитимуться символи.

Можливі варіанти кольорів у текстовому режимі наведені у таблиці.

Число	Константа	Колір	Зразок
0	Black	Чорний	
1	Blue	Темно-синій	

2	Green	Зелений	
3	Cyan	Морської хвилі	
4	Red	Темно-червоний	
5	Magenta	Фіолетовий	
6	Brown	Оливковий	
7	LightGray	Світло-сірий	
8	DarkGray	Темно-сірий	
9	LightBlue	Синій	
10	LightGreen	Світло-зелений	
11	LightCyan	Блакитний	
12	LightRed	Світло-червоний	
13	LightMagenta	Світло-фіолетовий	
14	Yellow	Жовтий	
15	White	Білий	

Window(x1,y1,x2,y2**)** - процедура встановлення розмірів вікна виведення інформації. Дозволяє організувати виведення інформації тільки у певну частину екрану, розміри і положення якої визначаються координатами верхнього лівого (x1,y1) і правого нижнього (x2,y2) кута. Значення параметрів x1 і x2 повинно знаходитись в межах від 1 до 80, а значення y1 і y2 - від 1 до 25. Саме вікно на екрані ніяк не окреслюється, але усі процедури і функції write|ln|, read|ln|, GotoXY, ClrScr, InsLine, DelLine, WhereX, WhereY виконують відповідні дії відносно встановленого вікна.

До процедур і функцій управління курсором відносяться:

GotoXY(X,Y**)** - процедура переміщення курсору у позицію, що визначається координатами X (стовпчик) і Y (рядок) активного вікна. Верхній лівий кут повного екрану має координати (1,1), правий нижній - (80,25).

WhereX - функція визначення поточного стовпчика положення курсору відносно поточного вікна.

WhereY - функція визначення поточного рядка положення курсору відносно поточного вікна.

До функцій управління клавіатурою відносяться:

ReadKey - функція дозволяє без додаткового натискання клавіші <Enter> зчитати значення натиснутої клавіші. Функція видає результат типу char, і дозволяє розпізнати натискання не тільки символьних клавіш, а й спеціальних.

KeyPressed - функція видає логічний результат, який має значення true, якщо до моменту виклику цієї функції була натиснута будь-яка клавіша на клавіатурі, в іншому випадку видається значення false.

До процедур управління звуком відносяться:

`Sound(Duration)` - процедура активізує вбудований динамік персонального комп'ютера. Цілочислове значення `Duration` вказує частоту звуку у герцах. Звук буде генеруватися до того моменту, поки у програмі не з'явиться процедура `NoSound`. Для того, що визначити певну тривалість звучання між процедурами `Sound` і `NoSound` ставлять процедуру `Delay (t)`, яка визначає паузу в мілісекундах.

`NoSound` - процедура відміняє дію процедури `Sound`.

3. Модулі користувача. Структура, створення, тестування, використання

Створення модуля потребує певної організації з використанням зарезервованих слів `Unit, Interface, Implementation, begin, end`. У випадку наявності вказаного сполучення слів компілятор сформує бібліотечний модуль - файл з розширенням `*.tpu` (Turbo Pascal Unit). Інакше буде створений готовий до виконання по виклику з DOS-файл, який матиме розширення `*.exe` (Executable).

3.1. Загальна структура модуля

```
Unit <ім`я модуля>;
Interface {Інтерфейсна (зовнішня)секція}
uses <список використаних модулів>;
type <визначення глобальних типів модуля>;
const <описування глобальних констант модуля>;
var <описування глобальних змінних модуля>;
procedure <заголовки процедур з параметрами>;
function <заголовки функцій з параметрами>;

Implementation {Секція реалізації}
type <визначення локальних типів модуля>;
const <описування локальних констант модуля>;
var <описування локальних змінних модуля>;
procedure <заголовки без параметрів та зміст процедур>;
function <заголовки без параметрів та зміст функцій>;

begin {Секція ініціювання}
<оператор>;
..........
<оператор>
end.
```

Розділ `Unit` містить ім`я бібліотечного модуля. Воно обов`язково повинно збігатися з ім`ям дискового файла, в якому зберігається текст модуля. Наприклад, якщо файл називається `Stat.pas`, то модуль повинен мати ім`я `Stat` (`Unit Stat;`).

В **інтерфейсній секції** описуються глобальні типи, константи, змінні, процедури і функції, тобто всі елементи, які повинні бути доступними основній програмі. Основна програма може використовувати всі ці елементи так, ніби вони описані у її секції описування.

У **секції реалізації** можуть бути описані локальні типи, константи та змінні, тобто елементи, які недоступні основній програмі, а потрібні лише для реалізації внутрішніх алгоритмів модуля. У цій секції також обов`язково повинні бути описані тіла всіх процедур і функцій, які були оголошені у інтерфейсній частині.

Секція ініціювання повинна бути останньою секцією модуля. Вона може складатися або з слів `begin end` (у такому випадку модуль не містить коду ініціювання), або з операторів, які треба виконати для ініціювання модуля (наприклад, відкрити файли, розрахувати початкові параметри тощо). Якщо в модулі є розділ ініціювання, то всі оператори цього розділу будуть виконані перед початком виконання основної програми, у якій використовується даний модуль. Якщо програма використовує кілька модулів, їхні розділи ініціювання будуть виконані в порядку переліку у розділі `uses`.

3.2. Створення модуля

Розглянемо типову ситуацію створення бібліотечного модуля користувача. Нехай потрібно сформувати на диску бібліотеку `MyLib`, у якій будуть зберігатися деякі необхідні процедури і функції. Тоді слід виконати такі дії:

1. Завантажити файл `MyLib.pas`, у якому повинні знаходитись тексти процедур та функцій, які призначені для зберігання у бібліотеці (будемо вважати, що тексти процедур і функцій не мають помилок).
2. Сформувати структуру `Unit, Interface, Implementation, begin, end`. (див. 3.1).
3. Встановити режим: `Compile: Destination - Disk`
4. Відкомпілювати файл (`Alt+F9`).
5. Перевірити наявність файла `MyLib.tpu`.

3.3. Використання елементів модуля

Для того, що отримати у програмі MyProg доступ до компонентів модуля MyLib слід виконати такі дії:

1. Завантажити програму `MyProg`, у якій планується використовувати компоненти модуля `MyLib`.
2. В розділі `uses` цієї програми вказати ім`я модуля `MyLib: uses Mylib`;
3. В програмі використати необхідні типи, константи, змінні, процедури і функції модуля `MyLib`.
4. Вибрати режим меню: `Options\Directories\UnitDirectories` і вказати в ньому шлях до модуля, наприклад:

5. Відкомпілювати програму `MyProg` і переконатися у правильності підключення модуля. (При компіляції на диск з`явиться файл `MyProg.exe`).

3.4. Пошук модулів при компіляції

При трансляції програми або модуля, що використовує інші модулі, компілятор послідовно відшукує файли з кодами необхідних модулів, щоб підключити їх до програми. Пригадайте,

які можливі варіанти компіляції (тема 10) існують і якими є їхні особливості. Модулі, що підключаються відшукуються у наступному порядку:

1. Компілятор переглядає зміст системного бібліотечного файла `turbo.tpl` (Turbo Pascal Library).
2. Якщо модуль не знайдено у `turbo.tpl`, компілятор шукає його у поточному каталогу.
3. Якщо модуль не знайдено у поточному каталогу, то пошук продовжується у каталогах, які вказані у параметрі: `Options\Directories\UnitDirectories`
4. Якщо на етапах 1..3 модуль не знайдено, то компілятор завершує роботу і видає повідомлення про помилку.

Тобто модуль повинен знаходитись або у файлі `turbo.tpl`, або у поточному каталогу, або у каталогах, що вказані у параметрі "`Options\Directories\UnitDirectories`".

Файл `turbo.tpl` має спеціальну структуру і призначений для компактного збереження і швидкого доступу до включених у нього модулів. Звичайно в цьому файлі знаходяться кілька системних (стандартних) модулів, однак з допомогою утиліти `tpumover` можна включати до файла `turbo.tpl` потрібні модулі і вилучати з нього непотрібні.

Створення і використання модулів розглянемо на прикладі.

Приклад 19.1. Створення і використання модуля користувача

Завдання: Створити модуль користувач, який міститиме функцію піднесення дійсного числа у дійсний степінь, і програму, яка підключившись до модуля, буде цю функцію використовувати.

Рішення: Відповідно до рекомендацій п.3, створимо загальну структуру модуля на ім'я `math`, який запишемо у файл `math.pas.` і сформуємо у ньому функцію піднесення дійсного числа у дійсний степінь (функція `Power`).
Далі створимо програму, яка використовуватиме означену функцію з модулю `Math`.

Текст модуля користувача:

```
Unit Math;
Interface
function Power(a,b:real):real;
Implementation
function Power;
begin
  if a>0 then Power := exp(b*ln(a))
         else if a<0 then Power := exp(b*ln(abs(a)))
                     else if b=0 then Power := 1
                                 else Power := 0
end; { Power }
Begin end.
```

Текст програми:

```
Program Pr19_01;
uses Math;
var
x,y :real;
BEGIN
  write('Уведіть дійсне число      = ');readln(x);
  write('Уведіть показник степеня = ');readln(y);
```

```
  writeln(X:8:2,' у степені ',Y:8:2,' = ',Power(x,y):8:2)
END.
```

Результати виконання програми:

```
Уведіть дійсне число      = 2
Уведіть показник степеня = 3
    2.00 у степені     3.00 =     8.00

Уведіть дійсне число      = 18.4
Уведіть показник степеня = 0
   18.40 у степені     0.00 =     1.00
```

Перегляньте роботу готової програми.

4. Контрольні запитання

1. Назвіть причини, які призвели до створення модулів у Pascal.
2. На які групи поділяються модулі і як можна до них підключитись?
3. Назвіть основні модулі Pascal і коротко охарактеризуйте призначення кожного із них.
4. Охарактеризуйте процедури управління зображенням модуля Crt.
5. Охарактеризуйте процедури управління курсором модуля Crt та наведіть приклади їхнього використання.
6. Охарактеризуйте процедури і функції управління клавіатурою і звуком модуля Crt.
7. Яку структуру повинен мати модуль користувача? Охарактеризуйте призначення секцій такого модуля.
8. У якій послідовності слід створювати власний модуль?
9. Як використовувати елементи власного модуля у програмі?
10. Де і як компілятор відшукує модулі при компіляції?

Тема 20. Графічне програмування у Pascal

План

1. Особливості графічного режиму
2. Ініціювання графічного режиму та підключення драйверів
3. Система координат і поточний покажчик. Виведення точок
4. Виведення ліній. Встановлення типу ліній
5. Виведення тексту в графічному режимі. Параметри виведення тексту
6. Побудова прямокутників і багатокутників
7. Виведення криволінійних примітивів
8. Визначення кольорів графічних примітивів
9. Маніпулювання графічними зображеннями
10. Контрольні запитання

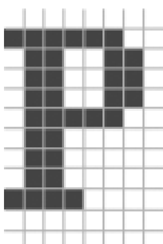
Ця тема є останньою, найбільшою і найцікавішою серед тем курсу. Ми познайомимось із основами графічного програмування у Pascal, яке надає широкі можливості у створенні різноманітного програмного забезпечення. при вивченні цієї теми прийдеться використовувати усі знання, отримані у попередніх темах, адже графічне програмування вимагає використання більш складних алгоритмів. Тут знадобиться все, і чітке розуміння типів даних, перевірка умов і цикли, взаємне перетворення чисел і рядків, створення власних процедур і функцій тощо.
У цій темі ви дізнаєтесь, які відмінності є між текстовим і графічним режимами роботи

монітора, як переключатися між цими режимами, якою є система координат у них.
Створення графічних зображень розпочнемо із точок, продовжимо лініями, прямокутниками і багатокутниками, колами, дугами і еліпсами, а завершимо виведенням тексту і встановленням кольорів графічних примітивів. Окремо розглянемо прийоми анімації графічних об'єктів.
Закріпити отримані знання вам суттєво допоможе виконання спрощеного і повного ескізу, а також імітація роботи механізму при виконанні курсової роботи.

1. Особливості графічного режиму

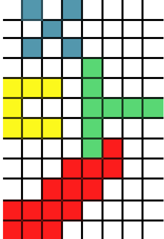
Відеомонітор комп`ютера може працювати у текстовому або графічному режимі.

У ***текстовому режимі*** ми можемо виводити на екран символи, що визначаються кодовою таблицею комп`ютера. Це, зокрема, символи латиниці і кирилиці, цифри, спеціальні символи (`! @ # # $ % ^ &` ....) та символи псевдографіки. Такий символ представляється у вигляді матриці розміром 8x8 (8x14, 8x16).



На кольоровому відеомоніторі ми можемо всі символи відображати одним з 16 кольорів на одному з 8 кольорів фону, а також, забезпечувати мерехтіння символів. При цьому можливості текстового режиму майже не залежать від типу відеомонітора.

У ***графічному режимі*** ми можемо виводити на екран будь-що, керуючи кольором кожної окремої точки на екрані, яка називається **піксель**. Роздільна здатність (кількість точок, що вміщуються на екрані відеомонітора по горизонталі і вертикалі) і кількість кольорів суттєво залежать від типу відеомонітора.



Перш ніж приступати до роботи у графічному режимі, треба визначити, яким відеомонітором та адаптером оснащений ваш комп`ютер. Нагадаємо, що ***адаптером*** (контролером, відеокартою) називається спеціальна електронна плата, яка підключається до системної плати комп`ютера і керує роботою відеомонітора. В залежності від типу відеомонітора вам буде потрібно підключити той чи інший ***драйвер*** - спеціальну програму, яка керує адаптером. Для роботи у графічному режимі у Pascal розроблено кілька драйверів для основних типів відеомоніторів. Ці драйвери зберігаються у файлах з розширенням `*.BGI` (*Borland Graphics Interface*).
Щоб отримати доступ до можливостей графіки Pascal, треба підключити модуль `Graph`, який зберігається у файлі `Graph.tpu` і являє собою комплекс засобів для створення професійних графічних програм. Він містіть багато процедур, функцій, констант і типів, які значно полегшують програмування у графічному режимі.

Підключення модуля `Graph` здійснюється в розділі `uses`.

```
uses Graph;
```

З цього моменту усі графічні засоби є доступними для користувача.

2. Ініціювання графічного режиму та підключення драйверів

Ініціювання графічного режиму означає перехід від текстового режиму відображенні до графічного (див. вище). Таке ініціювання здійснюється процедурою `InitGraph`, яка встановлює один з можливих відеорежимів.
Формат:

```
InitGraph(var GraphDriver:Integer; var GraphMode:Integer; PathToDriver: string);
```

Перший параметр - `GraphDriver` потребує вказати код (ім'я) графічного драйвера. Другий параметр - `GraphMode` потребує вказати код (ім'я) графічного режиму для конкретного графічного драйвера. Можливі варіанти графічних драйверів і графічних режимів наведені у табл. 20.1. Третій параметр потребує вказати місце на зовнішньому носії, де файл графічного драйвера встановлений. Якщо файл драйвера знаходиться у поточному каталогу, шлях можна не вказувати (ставляться пусті апострофи).

Таблиця 20.1. Графічні драйвери і графічні режими

Драйвер		Файл драйвера	Відеорежим		Роздільна здатність
Константа	Число		Константа	Число	
CGA	1	cga.bgi	CGAC0 CGAC1 CGAC2 CGAC3 CGAHi	0 1 2 3 4	320 x 200 320 x 200 320 x 200 320 x 200 640 x 200
MCGA	2		MCGAC0 MCGAC1 MCGAC2 MCGAC3 MCGAMed MCGAHi	0 1 2 3 4 5	320 x 200 320 x 200 320 x 200 320 x 200 640 x 200 640 x 480
EGA	3	egavga.bgi	EGALo EGAHi	0 1	640 x 200 640 x 350
EGA64	4		EGA64Lo EGA64Hi	0 1	640 x 200 640 x 350
EGAMono	5		EGAMonoHi	3	640 x 350
IBM8514	6	ibm8514.bgi	IBM8514Lo IBM8514Hi	0 1	640 x 480 1024 x 768
HercMono	7	herc.bgi	HercMonoHi	0	720 x 348
			ATT400C0 ATT400C1 ATT400C2	0 1 2	320 x 200 320 x 200 320 x 200

ATT400	8	att.bgi	ATT400C3 ATT400Med ATT400Hi	3 4 5	320 x 200 640 x 200 640 x 480
VGA	9	egavga.bgi	VGALo VGAMed VGAHi	0 1 2	640 x 200 640 x 350 640 x 480
PC3270	10	pc3270.bgi	PC3270Hi	0	720 x 350

При ініціюванні графічного режиму необхідно уникнути критичних помилок. Нижче наведений фрагмент програми коректного використання процедури `InitGraph`.

```
uses Graph;
var
  GraphDriver,GraphMode,ErrorCode   :integer;
BEGIN
  GraphDriver := Detect; {Функція Detect автоматично визначає тип драйвера}
  InitGraph(GraphDriver, GraphMode, 'D:\TP\BGI');{Ініціювання графічного режиму}
  ErrorCode := GraphResult;{Визначення коду помилки ініціювання}
  if ErrorCode <> grOk  then begin {Обробка помилкової ситуації}
       writeln('Графічна помилка :',GraphErrorMsg(ErrorCode));
       Halt(1){Примусове завершення роботи у випадку помилки}
     end;
  { Будь-які графічні та інші процедури, функції  тощо}
  . . . . . . . .
  CloseGraph;{Закриття графічного режиму - повернення у текстовий режим}
END.
```

У наведеному прикладі змінній `ErrorCode` надається значення функції `GraphResult`, яка дозволяє визначити причину помилки і проаналізувати її. У табл. 20.2 наведені коди і значення помилок роботи у графічному режимі.

Таблиця 20.2. Коди помилок ініціювання графічного режиму

Константа	Значення	Пояснення
grOk	0	Без помилок
grNoInitGraph	-1	Графіка не ініційована
grNotDetected	-2	Графічний пристрій не знайдено
grFileNotFound	-3	Файл драйвера не знайдено
grInvalidDriver	-4	Невірний файл драйвера пристрою
grNoLoadMem	-5	Недостатньо пам`яті для драйвера
grNoScanMem	-6	Вихід за межі пам`яті при заповненні (scan fill)
grNoFloodMem	-7	Вихід за межі пам`яті при заповненні (flood fill)
grFontNotFound	-8	Файл шрифту не знайдено
grNoFontMem	-9	Недостатньо пам`яті для шрифту
grInvalidMode	-10	Невірний графічний режим
grError	-11	Графічна помилка
grIOError	-12	Помилка графічного введення-виведення
grInvalidFont	-13	Невірний файл шрифту
grInvalidFontNum	-14	Невірний номер шрифту

Закриття графічного режиму здійснюється викликанням процедури `CloseGraph`, яка звільняє пам`ять, яку займали драйвери, шрифти, поточні дані, та поновлює режим роботи адаптера, в якому він знаходився до ініціювання графічного режиму.

Інколи потрібно тимчасово переключатися у текстовий режим і повертатися у графічний без його закриття і наступного ініціювання. Зробити це можна з допомогою процедур `RestoreCrtMode` та `SetGraphMode`, приклад використання яких наведений нижче.

```
. . . .
RestoreCrtMode;
writeln(' Текстовий режим. ');
. . . .
SetGraphMode(GraphMode);
OutTextXY(300,250,' Графічний режим. ');
. . . .
```

Деякі типи адаптерів дозволяють формувати кілька відеосторінок. У будь-який момент часу не екрані може відображатися тільки одна сторінка - та, що ми бачимо. У той же час ми можемо формувати зображення на іншій сторінці, яка називається активною. Для роботи з відеосторінками існує дві процедури `SetActivePage` та `SetVisualPage`.

```
. . . .
SetVisualPage(0);   {Видима сторінка - 0}
SetActivePage(1);   {Активна сторінка - 1}
Line(10,10,200,100);{Виведення лінії на активну невидиму сторінку 1}
SetVisualPage(1);   {Відображення сторінки 1. Лінія стане видимою}
. . . .
```

3. Система координат і поточний покажчик. Виведення точок

Стандартна система координат

Для побудови зображень на екрані у графічному режимі використовується така система координат: відлік починається від верхнього лівого кута екрана, який має координати (`0,0`); координата `X` збільшується зліва направо, координата `Y` збільшується зверху вниз. Координати точок можуть мати лише цілі значення у межах, що визначаються роздільною здатністю відеорежиму (див. табл. 20.1). Найбільші значення координат на одиницю менші за роздільну здатність, наприклад, для роздільної здатності 640 x 480 точка з максимальними координатами (`639,479`).

Для автоматичного визначення максимальних координат екрана вздовж осі X і вздовж осі Y можна використовувати функції `GetMaxX` і `GetMaxY`. Обидві функції мають однакові формати і відрізняються лише виміром екрану, значення якого вони повертають у програму.

Формат:

```
GetMaxX: integer;
GetMaxY:integer;
```

Система координат користувача

Використання функцій `GetMaxX` і `GetMaxY` дозволяє створювати програми, у яких розміри графічних зображень не будуть залежати від типу адаптера. Це можливо зробити шляхом

визначення власної системи координат. Нижче наведений фрагмент програми визначення власної системи координат з такими параметрами:
- початок системи координат у лівому верхньому куті екрана;
- максимальне значення координати X - 400 (збільшується зліва направо);
- максимальне значення координати Y - 300 (збільшується зверху вниз).

```
uses Graph;
var
   mx, my :real; { описування масштабних коефіцієнтів }
function X(r:real):integer;
begin
   X := round( r * mx )
end;
function Y(r:real):integer;
begin
   Y := GetMaxY - round( r * my )
end;
BEGIN
   Gd := Detect;
   InitGraph(Gd, Gm, '');
   if GraphResult <> grOk then Halt(1);
   mx := GetMaxX / 400 ; {значення масштабного коефіцієнта mx}
   my := GetMaxY / 300 ; {значення масштабного коефіцієнта my}
   { після цього можна використовувати всі можливості модуля Graph,
   але аргументи всіх процедур слід передавати як результати функцій X і Y}
   MoveTo(X(200),Y(150)); { переміщення поточного покажчика у центр екрана }
   . . . . .
   CloseGraph;
END.
```

Поточний покажчик і його переміщення

Щоб створювати графічні зображення, необхідно принаймні вказати точку початку виведення. У графічному режимі курсор не виводиться на екран, але існує ***поточний покажчик CP*** (*Current Pointer*). Фактично він відіграє ту саму роль, що й курсору в текстовому режимі. На екрані поточний покажчик не відображається, але його можна переміщувати з допомогою процедур `MoveTo` і `MoveRel`.

Процедура `MoveTo` переміщує поточний покажчик у точку з координатами (`X,Y`).

Формат:

```
MoveTo(X,Y: integer);
```

Процедура `MoveRel` переміщує поточний покажчик на вказану кількість пікселів по горизонталі і вертикалі.
Формат:

```
MoveRel(dX,dY: integer);
```

При використанні процедур `MoveTo` і `MoveRel` слід дивитися, щоб координати поточного покажчика не вийшли за межі, припустимі для встановленого відеорежиму. Так для процедури `MoveTo` неприйнятними є від'ємні значення координат X і Y, а для процедури `MoveRel` вони можуть бути як додатними, так і від'ємними, але не повинні перевищувати відстані до відповідних сторін екрану.

Далі наведений фрагмент програми із використанням процедур переміщення поточного

покажчика і далі, на рис. 20.1 наведено ілюстрацію їхньої дії. Відповідні вузли помічені хрестиками з номерами.

```
MoveTo(GetMaxX div 2, GetMaxY div 2); {1}
MoveRel(-50,100);                     {2}
MoveTo(100,100);                      {3}
MoveTo(50,200);                       {4}
MoveRel(100,0);                       {5}
```

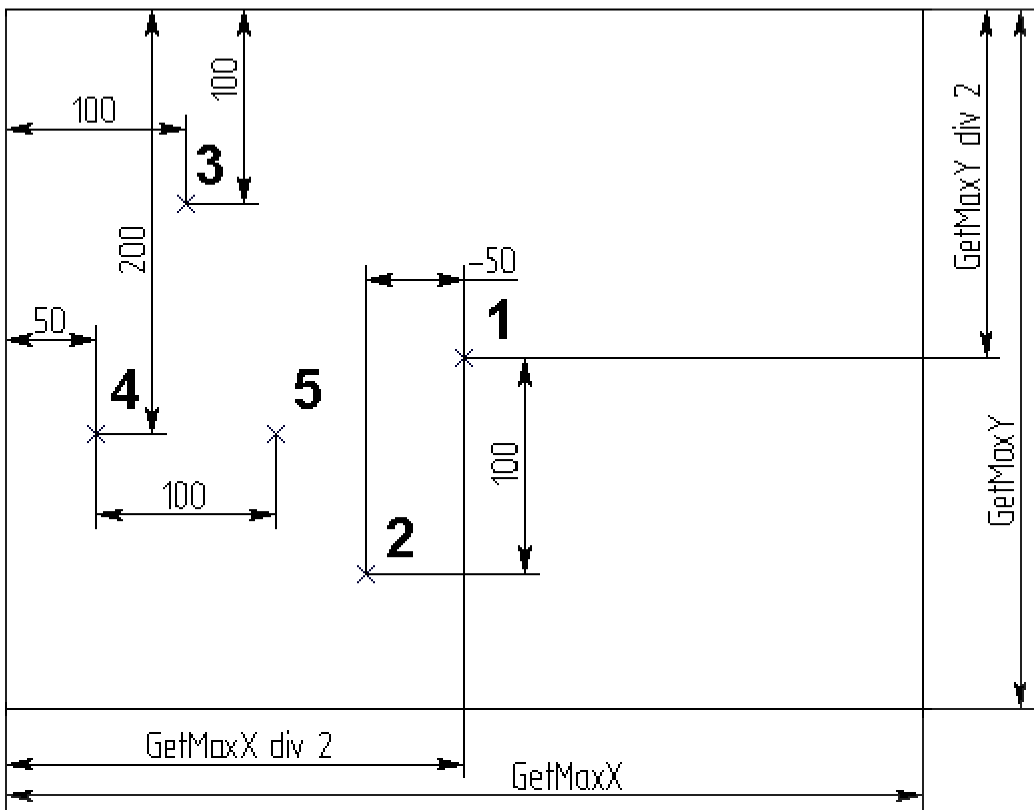


Рис. 20.1. Ілюстрація до використання процедур `MoveTo` *і* `MoveRel`

У багатьох програмах буває необхідно визначати місце розташування поточного покажчика. Для цього слід використовувати функції `GetX:integer` і `GetY:integer`, які повертають у програму координати `X` і `Y` поточного покажчика.

Вікна у графічному режимі

Процедура `SetViewPort` дозволяє виводити інформацію лише на частину екрана в активне прямокутне вікно.

Формат:

```
SetViewPort(x1,y1,x2,y2:integer; Clip:boolean);
```

де
`x1,y1` – координати верхнього лівого кута вікна;
`x2,y2` – координати нижнього правого кута;
`Clip` – визначає, буде (`Clip=true`) чи ні (`Clip=false`) відсікатися малюнок при виведенні за межі вікна.

Параметри встановленого вікна можна визначити з допомогою наступної процедури.

Формат:

```
GetViewSettings(var ViewPort: ViewPortType);
```

де
`ViewPortType` - стандартний тип модуля `Graph`.

```
type ViewPortType = record
                      x1,y1,x2,y2: integer;
                      Clip       : boolean;
                    end;
```

Для того, щоб очистити весь екран у графічному режимі, слід скористатися процедурою `ClearDevice`, а очищення встановленого вікна здійснюється з допомогою процедури `ClearViewPort`, При очищенні вікон у графічному режимі слід пам`ятати, що, на відміну від текстових вікон, графічні вікна після встановлення фону `SetBkColor` змінюють фон разом із загальним фоном екрана.

Виведення точок

Які б зображення не виводились на екран, всі вони, в решті решт, складаються з точок (пікселів). Маючи засіб виведення у потрібне місце екрана точки визначеного кольору, теоретично можна створити будь-яке зображення, аж до картини.

Процедура `PutPixel` призначення для виведення точки у графічному режимі.

Формат:

```
PutPixel(X,Y :integer; Color :word);
```

де
`X,Y` - координати точки на екрані;
`Color` - колір, яким виводитиметься точка.

Для визначення кольору точки, яка вже виведена на екран, можна скористатися процедурою `GetPixel(x,y:integer):word`.

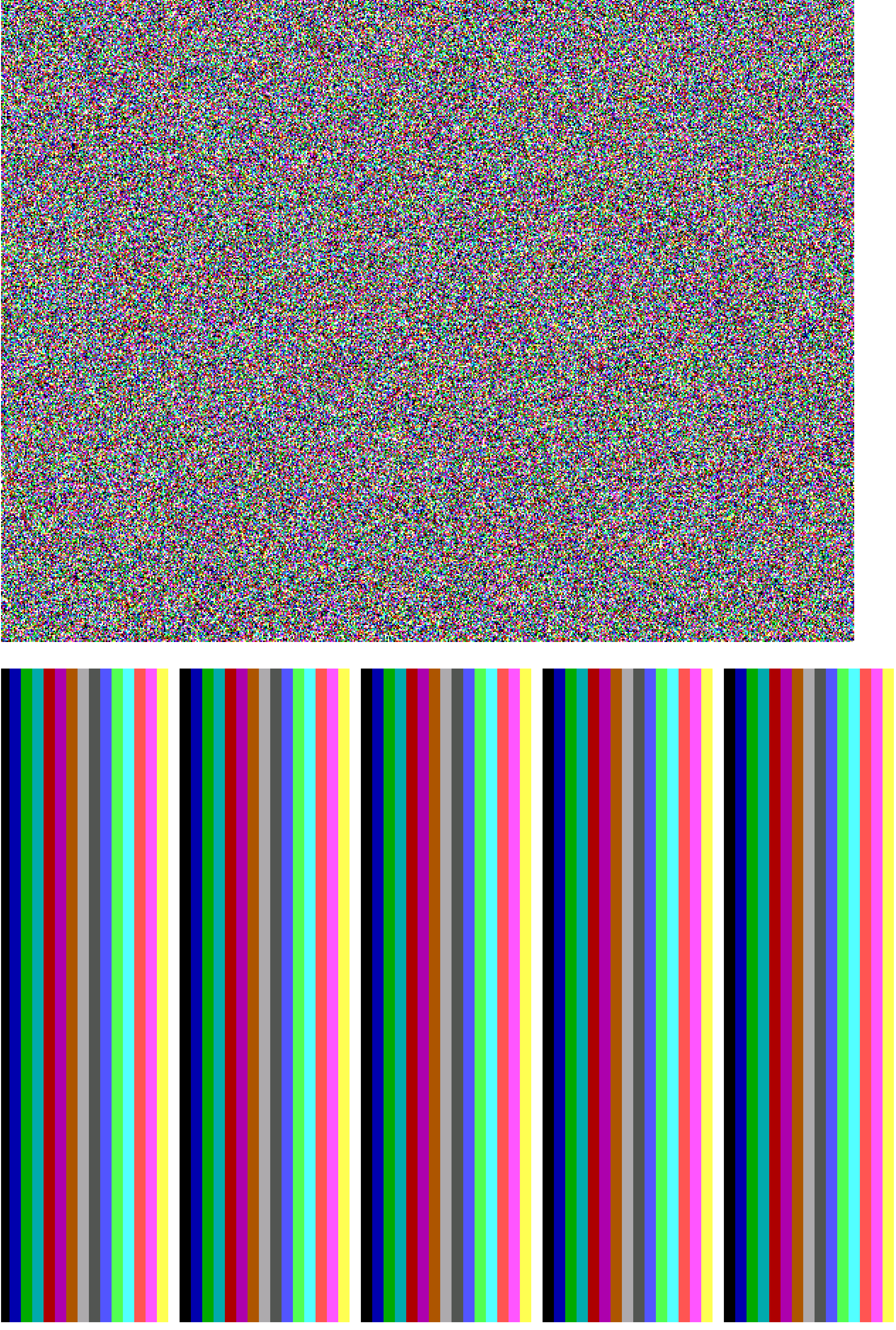
Приклад 20.1. Виведення точок

Завдання. Створити програму, яка виключно за допомогою оператора виведення точок послідовно сформує екрани, заповнені точками різного кольору, вертикальними та горизонтальними смугами різного кольору.

Рішення. Першими етапом є ініціювання графічного режиму, яке виконується за стандартним алгоритмом.
Далі, використовуючи вкладені цикли, екран послідовно заповнюється точками, колір яких є випадковим.
Наступний етап розпочинається після натискання користувачем клавіші `<Enter>`. Виведення смуг також виконується у двох вкладених циклах, і колір точок, які виводяться зв'язаний із параметром внутрішнього циклу, розділивши який без остачі на 8, ми отримаємо зміну кольору через 8 рядків.

Останній етап також розпочинається після натискання користувачем клавіші <Enter>. Виведення вертикальних смуг також виконується у двох вкладених циклах, але колір точок, які виводяться, зв'язаний вже із параметром зовнішнього циклу. Цілочислове ділення на 16 забезпечує ширину смуг у 16 пікселів.

```
Program Pr20_01;
uses Graph;
var
  Gd,Gm,Ec :integer;
  i,j,Color:integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  { виведення точок випадковим кольором }
  for j:=0 to GetMaxY do
    for i:=0 to GetMaxX do
      PutPixel(i,j,random(16));
  readln;
  { виведення вертикальних смуг товщиною 8 пікселів }
  ClearDevice;
  for j:=0 to GetMaxY do
    for i:=0 to GetMaxX do
      PutPixel(i,j,i div 8);
  readln;
  { виведення горизонтальних смуг товщиною 16 пікселів }
  ClearDevice;
  for j:=0 to GetMaxY do
    for i:=0 to GetMaxX do
      PutPixel(i,j,j div 16);
  readln;
  closeGraph;
END.
```



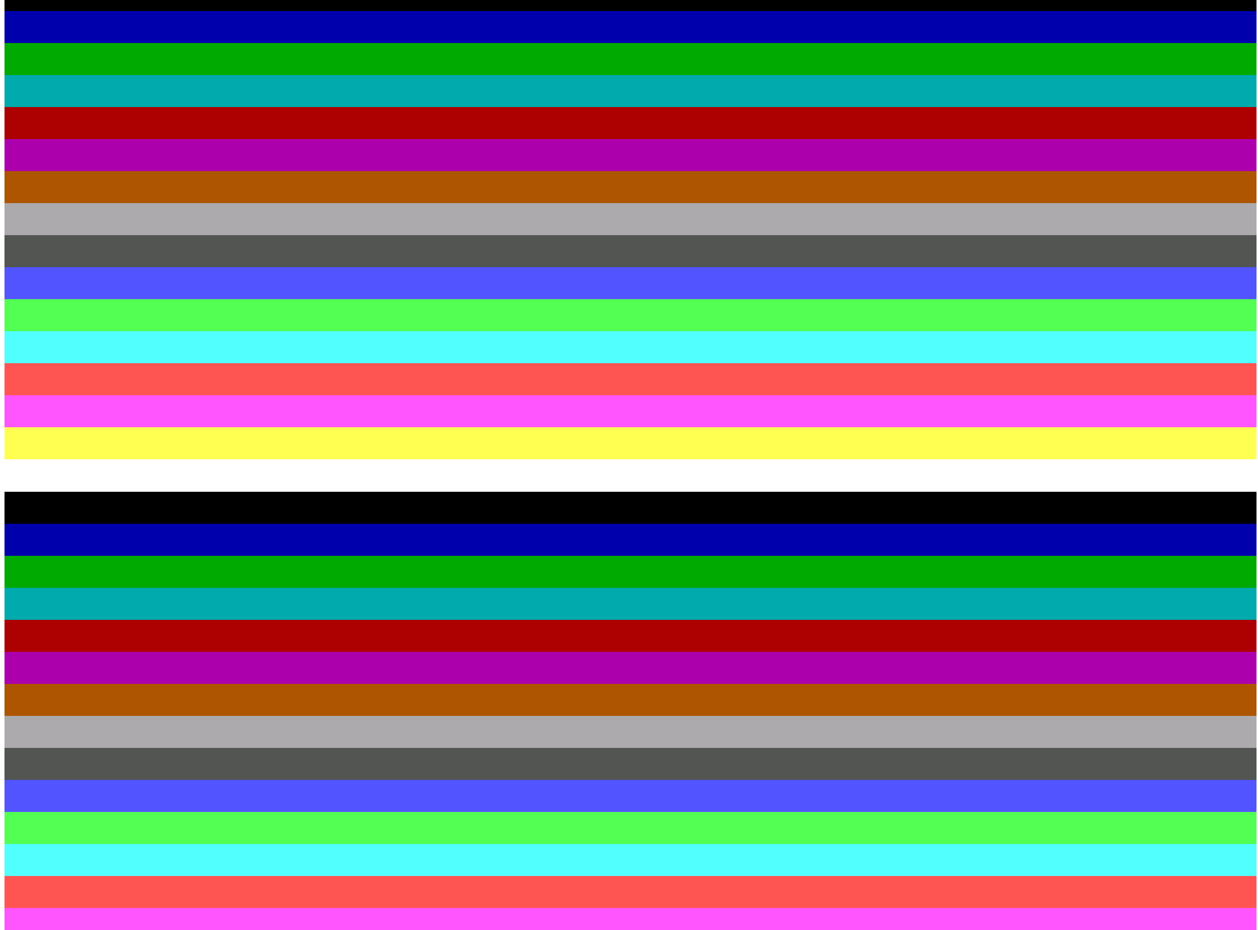


Рис. 20.2. Фрагменти копій екрану етапів роботи програми;
а) точки випадкового кольору; б) вертикальні смуги; в) горизонтальні смуги

Перегляньте роботу готової програми.

4. Виведення ліній. Встановлення типу ліній

Багато графічних об'єктів можна побудувати за допомогою ліній. У Pascal з цією метою можне використовувати процедури `Line`, `LineTo`, `LineRel`.

Формати:

```
Line (X1,Y1,X2,Y2: integer);
LineTo (X,Y:integer);
LineRel (dX,dY:integer);
```

Перша процедура виводить лінію від точки екрана з координатами (`X1,Y1`) у точку з координатами (`X2,Y2`). Друга процедура виводить лінію від місця розташування поточного покажчика у точку екрана з координатами (`X,Y`). Третя процедура виводить лінію від місця розташування поточного покажчика у точку, яка лежить на відстані (`dX,dY`) від неї. Значення `dX` і `dY` можуть бути додатними або від`ємними, значення `X,Y,X1,Y1,X2,Y2` - лише додатними.

Pascal дозволяє виводити най різноманітніші лінії: товсті і тонкі, штрихові, пунктирні тощо. Визначення стилю лінії здійснюється процедурою `SetLineStyle`.

Формат:

```
SetLineStyle(LineStyle,Pattern,Thinkness:word);
```

Можливі значення параметрів процедури `SetLineStyle` наведені у табл. 20.3.

Таблиця 20.3. Параметри процедури `SetLineStyle`

Константа	Значення	Пояснення	Зразок
Параметр `LineStyle`			
`SolidLn`	0	Суцільна лінія	———————
`DottedLn`	1	Лінія з точок	....
`CenterLn`	2	Осьова лінія	—·—·—·—·—
`DashedLn`	3	Пунктирна лінія	— — — — — —
`UserBitLn`	4	Лінія користувача	
Параметр `Thickness`			
`NormWidth`	1	Тонка лінія (1 піксель)	———————
`ThickWidth`	3	Товста лінія (3 пікселі)	━━━━━━━

Якщо для певної задачі слід створити лінію власного стилю, тоді слід параметр `LineStyle` встановити у значення `UserBitLn`, а параметром `Pattern` визначити як саме повинна виглядати лінія. Цей параметр є шістнадцятковим числом, бітовий запис якого визначає стиль лінії. Наприклад, якщо встановити `Pattern=$44`, тоді лінія буде формуватися такими бітами `01000100` (пригадайте переведення шістнадцяткових чисел у двійкові) і матиме такий вид (збільшено) □■□□□■□□ - лінія з рідких точок.

Далі наведено приклади програм, у яких використовуються різні стилі ліній і різні оператори для їх виведення.

Приклад 20.2. Виведення ліній за допомогою оператора `Line`

Завдання. Створити програму, яка за допомогою оператора `Line` вимальовує на екрані спрощене (без цифр і пояснень) зображення циферблату годинника.

Рішення. Після ініціювання графічного режиму визначаємо точку центру циферблату (`xc,yc`) і величину зовнішнього радіусу циферблату (`Rn`). Виведення ліній відбувається у циклі, в якому параметр циклу `i` змінюється із кроком `6` (це відповідає куту однієї секунди або хвилини). Далі в залежності від поточного кута визначаються координати (xn,yn) точки, яка є зовнішньою точкою кожної риски циферблата. Наступним кроком є визначення умови, коли слід виводити довші риски - це відповідає куту 30 градусів (кожна година). Якщо така умова виконується, тоді встановлюється величина внутрішнього радіусу циферблату `Rk` на 40 пікселів меншою за `Rn`, в іншому випадку різниця складає 20 пікселів. Далі розраховується координата внутрішньої точки риски циферблату і виводиться відповідна лінія. Після виведення кожної лінії збільшується параметр циклу, а сам цикл завершується коди пройдено все коло.

```
Program Pr20_02;
uses Crt,Graph;
var
```

```
  Gd,Gm,Ec :integer;
  xc,yc,xn,yn,xk,yk,Rn,Rk,i:integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2; {Визначення координат центру екрану}
  Rn:=(GetMaxY div 2)-20;          {Визначення величини зовнішнього радіусу }
  i:=0;                            {Ініціювання лічильника циклу     }
  repeat
    xn:=xc-round(Rn*cos(i*Pi/180));{Визначення координат точки       }
    yn:=yc-round(Rn*sin(i*Pi/180));{на зовнішньому радіусі           }
    if (i mod 30)=0                {Умова для виведення довгих рисок }
      then  Rk:=(GetMaxY div 2)-60 {Внутрішній радіус довгих рисок   }
      else  Rk:=(GetMaxY div 2)-40;{Внутрішній радіус коротших рисок }
    xk:=xc-round(Rk*cos(i*Pi/180));{Визначення координат точки       }
    yk:=yc-round(Rk*sin(i*Pi/180));{на внутрішньому радіусі          }
    Line(xn,yn,xk,yk);             {Виведення лінії                  }
    inc(i,6);                      {збільшення лічильника на 6       }
  until i>=360;                    {Умова завершення циклу           }
  repeat until KeyPressed;
  CloseGraph;
END.
```

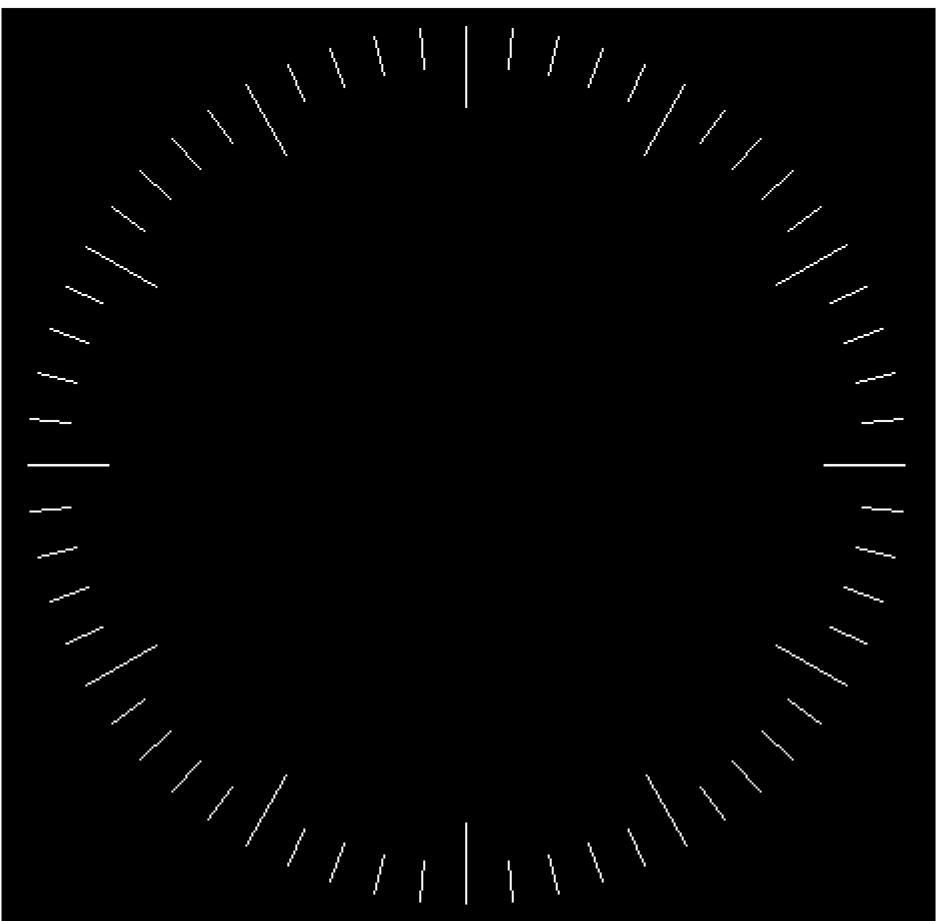


Рис. 20.3. Результат роботи програми `Pr20_02`

Перегляньте роботу готової програми.

Приклад 20.3. Виведення ліній за допомогою оператора `LineRel`

Завдання. Створити програму, яка виводить на екран хаотичні ламані відрізки, причому початок наступного відрізка співпадає із початком наступного.

Рішення. Для вирішення цієї задачі зручним буде використання оператора LineRel. Суть програми полягає у тому, що після ініціювання графічного режиму поточний покажчик переміщується у середину екрану. Далі після ініціювання генератора початкових чисел (а це потрібно для того, щоб при кожному запуску програми лінії виводились по різному) розпочинається цикл виведення відрізків. Зміщення кінцевої точки на (`dx,dy`) знаходиться як випадкове число, причому саме зміщення може бути як додатним, так і від'ємним, і за допомогою оператора `LineRel` виводиться черговий відрізок. Цикл завершується коли користувач натискає будь-яку клавішу на клавіатурі.
Відзначимо, що через певний час координати чергового відрізка вийдуть за межі екрану і можуть вже ніколи не потрапити у зону, обмежену (`0,0`) - (`GetMaxX,GetMaxY`).

```
Program Pr20_03;
uses Crt, Graph;
var
  Gd,Gm,Ec :integer;
  Dx,Dy:integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  MoveTo(GetMaxX div 2,GetMaxY div 2); {Перемістити покажчик у центр екрану}
  randomize;                  {Ініціювання генератора випадкових чисел}
  repeat
    Dx:=random(64)-32;        {Випадкове зміщення вздовж X}
    Dy:=random(48)-24;        {Випадкове зміщення вздовж Y}
    LineRel(Dx,Dy);           {Виведення лінії             }
    Delay(500);               {Пауза                       }
  until KeyPressed;           {Умова завершення циклу      }
  CloseGraph;
END.
```

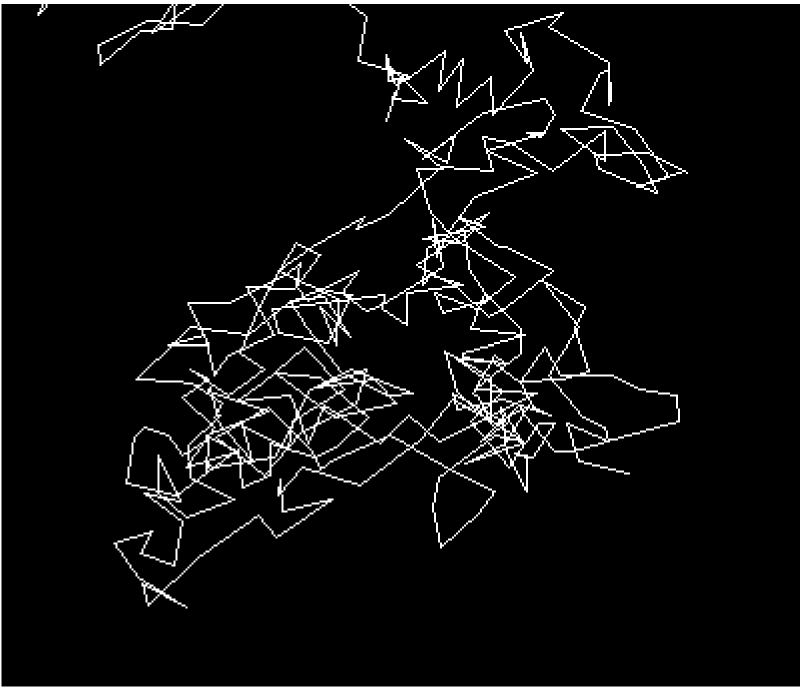


Рис. 20.4. Один із варіантів роботи програми `Pr20_03`

Перегляньте роботу готової програми.

Приклад 20.4. Виведення ліній за допомогою оператора LineTo

Завдання. Створити програму, яка від різних точок екрану і різним кольором виводить випадкові лінії до центру екрану.

Рішення. Для вирішення цієї задачі зручним буде використання оператора `LineTo`. Суть алгоритму вирішення цієї задача полягає у тому, що координати точки від якої буде починатися виведення лінії є випадковими в межах роздільної здатності екрану. Колір лінії також вибирається випадково із 16-ти можливих кольорів. У циклі, який може завершити користувач натисканням будь-якої клавіші відбувається переміщення поточного покажчика у випадкову точку екрану, з якої виводиться лінія у центр екрану.

```
Program Pr20_04;
uses Crt, Graph;
var
  Gd,Gm,Ec,w,h :integer;
  xc,yc,x,y:integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2;{Визначення координат центру вікна}
  w:=GetMaxX; h:=GetMaxY;{Визначення ширини і висоти вікна          }
  randomize;             {Ініціювання генератора випадкових чисел }
  repeat
    x:=random(w);        {Випадкове значення координати X           }
    y:=random(h);        {Випадкове значення координати Y           }
    SetColor(random(16));{Встановлення випадкового кольору          }
    MoveTo(x,y);         {Перемістити покажчика у випадкову точку }
    LineTo(xc,yc);       {Провести лінію у центр екрану             }
    Delay(500);          {Пауза                                     }
  until KeyPressed;      {Умова завершення циклу                    }
  CloseGraph;
END.
```

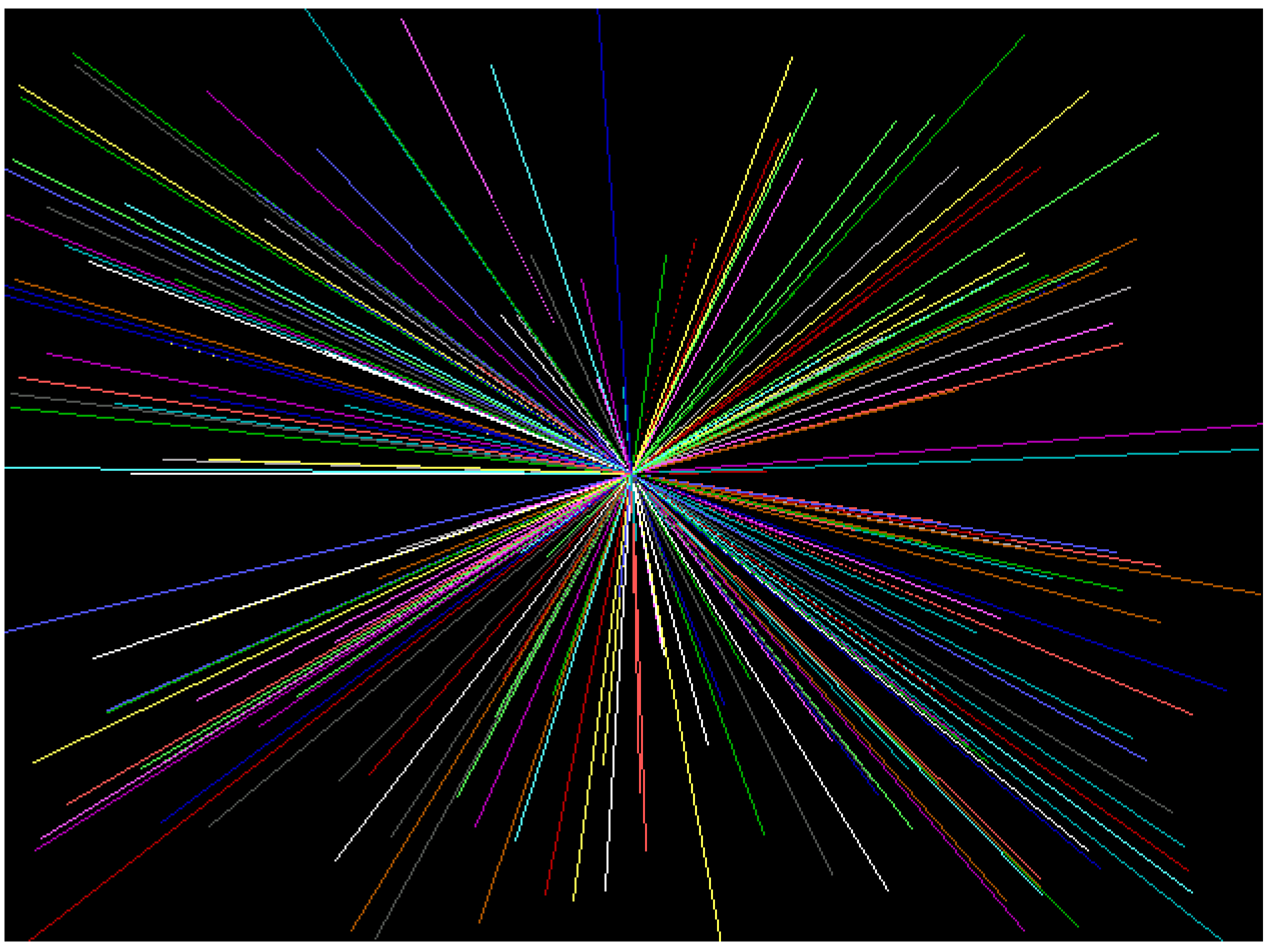


Рис. 20.5. Один із варіантів роботи програми `Pr20_04`

Перегляньте роботу готової програми.

5. Виведення тексту в графічному режимі. Параметри виведення тексту

5.1. Виведення інформації у графічному режимі

На відміну від текстового режиму, де для виведення використовувались оператори `write` або `writeln`, у графічному режимі слід користуватися процедурами `OutText` і `OutTextXY`, причому виводити можна лише рядкову інформацію.

Процедура `OutText` виводить текст, починаючи з місця розташування поточного покажчика.

Формат:

```
OutText(Text:string);
```

Процедура `OutTextXY` виводить текст, починаючи з позиції (`X`, `Y`) екрану

Формат:

```
OutTextXY(X,Y:integer; Text:string);
```

Приклади:

```
OutText('Текст виводиться з поточної позиції покажчика.');
OutTextXY(50,50,'Текст виводиться від точки (50,50).');
```

Виведення числової інформації у графічному режимі потребує певних попередніх дій, адже процедури `OutText` і `OutTextXY` дозволяють виводити лише текстову інформацію. Тому для виведення чисел треба спочатку перетворити їх у рядок символів з допомогою процедури `Str`, а потім операцією конкатенації рядків підключити результат до рядка, який виводиться процедурою, наприклад:

```
S := 0.12;
str(S:4:2,Sstr); {результат перетворення знаходиться у Sstr}
OutTextXY(50,50,' Подача = ' + Sstr + ' мм/об ');
```

У разі потреби можна створити процедури виведення дійсних або цілих чисел на екран у графічному режимі. Далі наведений текст процедури виведення цілих чисел у графічному режимі.

```
procedure OutIntegerXY(X,Y,Znach,Format:    integer);
var SS:string;
begin
   str(Znach:Format,SS);
   OutTextXY(X,Y,SS);
end;
```

5.2. Визначення шрифтів

Якісне виведення інформації у графічному режимі потребує використання різноманітних шрифтів. Встановити потрібний шрифт та його параметри можна з допомогою процедури `SetTextStyle`.

Формат:

```
SetTextStyle(Font, Direction, Size :word);
```

де,
`Font` – код або константа шрифту (див. табл. 20.4);
`Direction` – напрямок виведення тексту (див. табл. 20.4);
`Size` – розмір шрифту (від 1 до 10).

Таблиця 20.4. Шрифти та напрямки виведення тексту

Константа	Файл шрифту	Значення	Пояснення	Зразок шрифту
Параметр `Font`				
`DefaultFont`	-	0	Стандартний шрифт	Example
`TriplexFont`	`trip.chr`	1	Векторний шрифт	Example
`SmallFont`	`litt.chr`	2	Маленький шрифт	Example

SansSefirFont	sans.chr	3	Прямий шрифт	Example
GothicFont	goth.chr	4	Готичний шрифт	Example
Параметр Direction				
HorizDir		0	Виведення по горизонталі (зліва - направо)	
VertDir		1	Виведення по вертикалі (знизу - вверх)	

Слід зазначити, що файли шрифтів повинні знаходитись або у поточному каталозі, або у каталозі, який було вказано при ініціюванні графічного режиму процедурою `InitGraph`. У протилежному випадку може виникнути помилка, код якої можна визначити з допомогою функції `GraphResult`. Окрім зазначених у табл. 20.4 стандартних шрифтів певні версії компілятора Pascal можуть містити й додаткові шрифти. Дізнатися про наявність таких шрифтів можна переглянувши папку "`..\tp\bgi`". Такі файли мають розширення `*.chr`, а їхні номери продовжуються із номера 5.

Наступний приклад ілюструє використання різних шрифтів і дозволяє дізнатися про те, які шрифти встановлені на вашому комп'ютері

Приклад 20.5. Виведення тексту

Завдання. Створити програму, яка виводить на екран приклад тексту різними шрифтами і різними кольорами.

Рішення. Після ініціювання графічного режиму в циклі встановлюємо черговий за номером шрифт розміром 4, встановлюється черговий колір і, починаючи від точки, що кожного разу зміщується вниз на 40 пікселів, виводиться приклад тексту.

Текст програми:

```
Program Pr20_05;
uses Crt, Graph;
var
  Gd,Gm,Ec,i :integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  for i:=0 to 9 do begin {Цикл виведення тексту         }
    SetTextStyle(i,0,4); {Визначення шрифту із розмірами}
    SetColor(i+1);       {Визначення кольору тексту     }
    OutTextXY(0,i*40,'Example');{Виведення тексту       }
  end;
  repeat until KeyPressed;
  CloseGraph;
END.
```

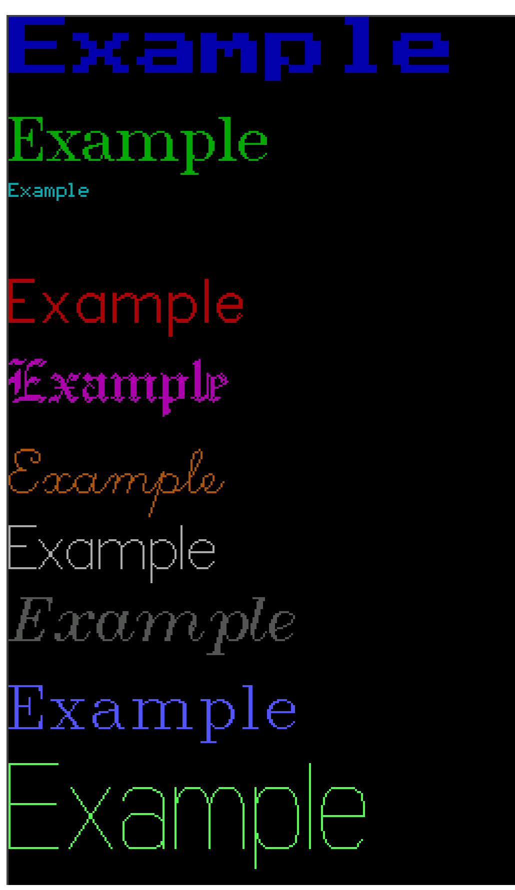


Рис. 20.6. Результат роботи програми Pr20_05

Перегляньте роботу готової програми.

Потрібний розмір шрифту, окрім явного визначення через параметр `Size` процедури `SetTextStyle`, можна встановити з допомогою процедури `SetUserCharSize`.

Формат:

```
SetUserCharSize(multX,divX,multY,divY:word);
```

Перші два параметри визначають горизонтальний розмір шрифту, два останні - вертикальний. Якщо взяти за одиницю ширину стандартного шрифту, то відношення `multX/divX` буде визначати ширину нового шрифту. Так само відношення `multY/divY` буде визначати висоту нового шрифту.

Приклади визначення розмірів шрифтів:

```
SetTextStyle(TriplexFont, HorizDir, 4);
OutText('Нормальний розмір');
SetUserCharSize(1,1,3,1);
OutText('Текст утричі вищий за нормальний ');
SetUserCharSize(2,1,1,1);
OutText('Текст удвічі ширший за нормальний ');
```

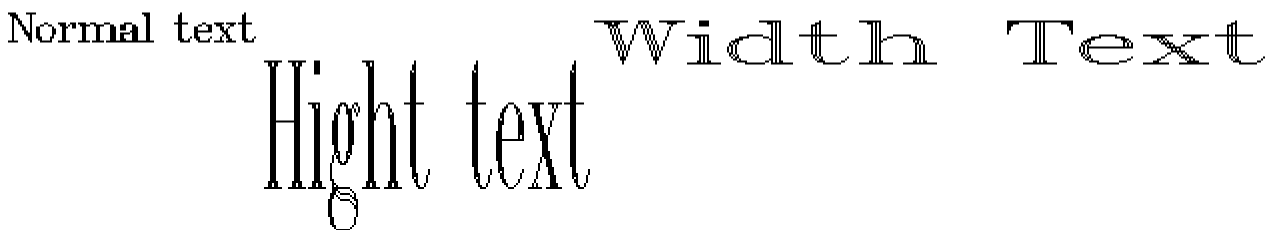


Рис. 20.7. Зміна пропорцій шрифтів

Щоб дізнатися про вертикальний і горизонтальний розміри символу або рядка у пікселах, можна скористатися функціями `TextHeight` і `TextWidth`.

Формат:

```
TextHeight(Text:string):word;
TextWidth (Text:string):word;
```

5.3. Визначення параметрів вирівнювання шрифтів

У деяких задачах потрібно змінювати положення тексту відносно поточного покажчика. Вирівнювання тексту виконується процедурою `SetTextJustify`.

Формат:

```
SetTextJustify(Horiz,Vert:word);
```

де
`Horiz` і `Vert` - параметри вирівнювання тексту, можливі значення яких наведені у табл. 20.5.

Таблиця 20.5. Параметри вирівнювання тексту

Вирівнювання по горизонталі			**Вирівнювання по вертикалі**		
Параметр `Horiz`	**Код**	**Положення базової точки**	**Параметр `Vert`**	**Код**	**Положення базової точки**
`LeftText`	0	Ліворуч тексту	`BottomText`	0	Вгорі тексту
`CenterText`	1	По центру	`CenterText`	1	По центру
`RightText`	2	Праворуч тексту	`TopText`	2	Внизу тексту

Використання різного за вирівнюванням виведення тексту проілюстровано наступним прикладом.

Приклад 20.6. Управління параметрами вирівнювання тексту

Завдання. Удосконалити програму 20.2 таким чином, щоб виводився циферблат годинника із підписами біля відповідних годин, а також, щоб години 3, 6, 9 і 12 виводились збільшеним шрифтом та іншим кольором.

Рішення. Додатково до програми 20.2 встановимо наступне. Колір фону екрану встановимо білим, при виведенні великих рисок встановлюємо зелений колір, а при виведенні малих рисок - синій. Перший оператор `case` в залежності від цифри, яку слід виводити, встановлює

відповідні параметри вирівнювання тексту. Другий оператор `case` визначає шрифт, яким буде виводитись текст (для цифр 3,6,9,12 - `TrilpexFonr`, для решти цифр - `DefaultFont`). Саме виведення тексту вимагає попереднього переведення числа у рядок, адже оператор `OutTextXY` дозволяє виводити тільки рядкову інформацію. Виведення всіх рисок і тексту відбувається у циклі.

Текст програми:

```
Program Pr20_06;
uses Crt, Graph;
var
  Gd,Gm,Ec :integer;
  xc,yc,xn,yn,xk,yk,Rn,Rk,i:integer;
  s_time:string;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2;
  Rn:=(GetMaxY div 2)-40;
  SetBkColor(White);{Встановлення білого кольору циферблату}
  i:=0;
  repeat
    xn:=xc+round(Rn*sin(i*Pi/180));
    yn:=yc-round(Rn*cos(i*Pi/180));
    if (i mod 30)=0
      then  begin Rk:=(GetMaxY div 2)-80;{Внутрішній радіус великих рисок}
                  SetColor(Green);       {Колір великих рисок}
            end
      else  begin Rk:=(GetMaxY div 2)-60;{Внутрішній радіус малих рисок}
                  SetColor(Blue);        {Колір малих рисок}
            end;
    xk:=xc+round(Rk*sin(i*Pi/180));
    yk:=yc-round(Rk*cos(i*Pi/180));
    Line(xn,yn,xk,yk);{Виведення рисок}
    SetColor(Green);{Встановлення кольору для виведення цифр}
    case (i div 30) of{Встановлення параметрів вирівнювання}
      1,2  :SetTextJustify(LeftText,BottomText);
      3    :SetTextJustify(LeftText,CenterText);
      4,5  :SetTextJustify(LeftText,TopText);
      6    :SetTextJustify(CenterText,TopText);
      7,8  :SetTextJustify(RightText,TopText);
      9    :SetTextJustify(RightText,CenterText);
      10,11:SetTextJustify(RightText,BottomText);
      12   :SetTextJustify(CenterText,BottomText);
    end;
    case (i div 30) of {Встановлення параметрів шрифту}
       3,6,9,12: SetTextStyle(TriplexFont,0,5);
       else      SetTextStyle(DefaultFont,0,2);
    end;
    if ((i mod 30)=0) and (i<>0){Умова виведення цифр }
    then begin
       str(i div 30,s_time);{Переведення числа у рядок}
       OutTextXY(xn,yn,s_time);{Виведення рядка}
    end;
    inc(i,6);
  until i>360;
  repeat until KeyPressed;
  CloseGraph;
END.
```

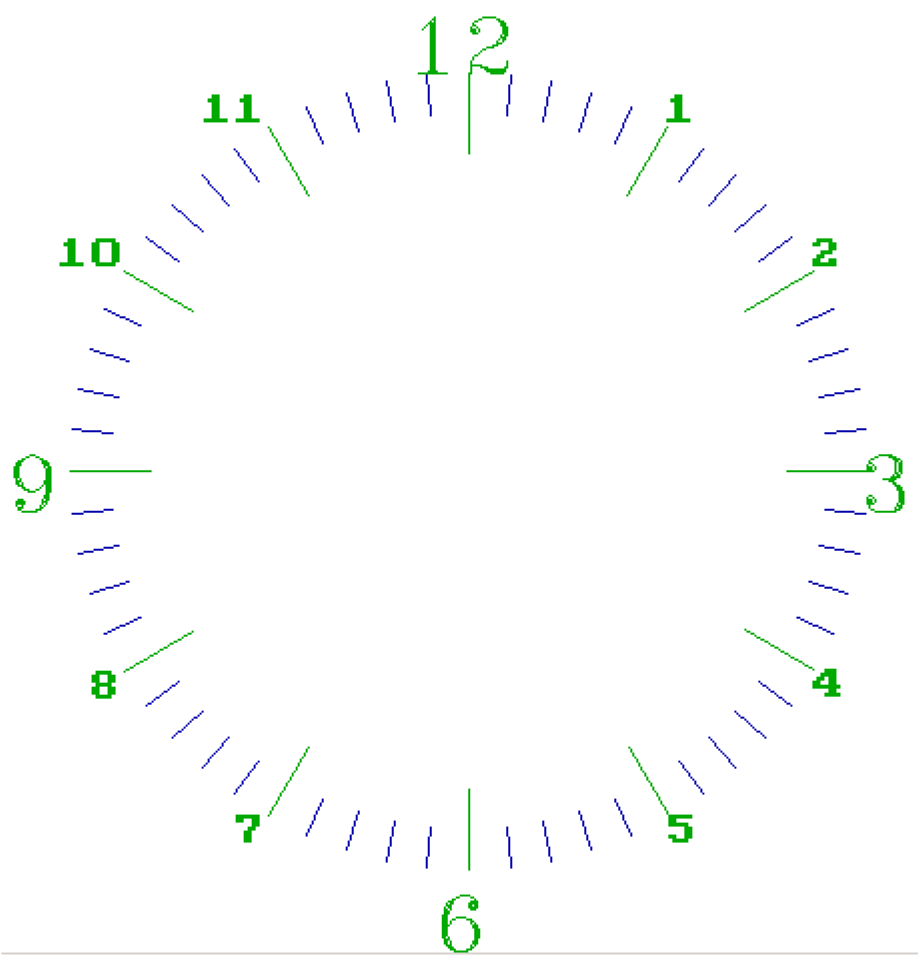


Рис. 20.8. Результат роботи програми `Pr20_06`

Перегляньте роботу готової програми.

6. Побудова прямокутників і багатокутників

Pascal має додаткові засоби для виведення зображень, які утворюються сукупністю відрізків. До таких процедур належать процедури виведення прямокутників (`Rectangle, Bar, Bar3D`) і полігонів (`DrawPoly, FillPoly`).

6.1. Виведення прямокутників

Процедура `Rectangle` виводить зовнішній контур прямокутника, сторони якого є паралельними сторонам екрану.

Формат:

```
Rectangle(X1,Y1,X2,Y2:integer);
```

де
`X1,Y1` – координати верхнього лівого кута прямокутника;
`X2,Y2` – координати правого нижнього кута прямокутника.

Процедура `Bar` виводить прямокутник, внутрішня частина якого заповнюється відповідно до параметрів процедури `SetFillStyle` (див. п.8). Параметри процедури аналогічні

параметрам попередньої процедури Rectangle.

Формат:

```
Bar(X1,Y1,X2,Y2:integer);
```

Процедура Bar3D виводить тривимірний стовпчик, із заповненням внутрішньої частини.

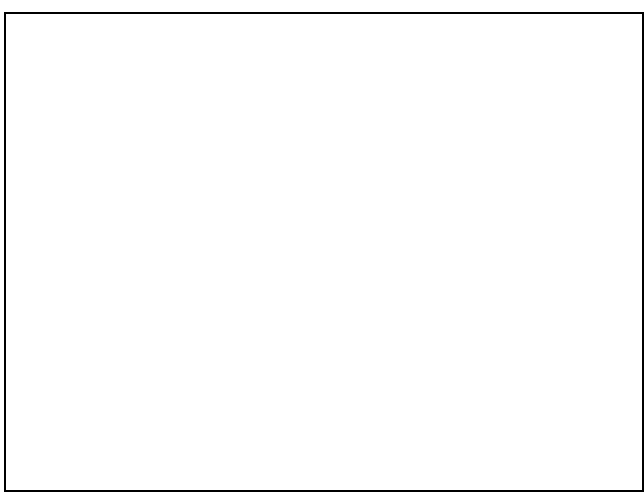
Формат:

```
Bar3D(X1,Y1,X2,Y2:integer; Depth:word; Top:boolean);
```

де
X1,Y1 – координати верхнього лівого кута прямокутника;
X2,Y2 – координати правого нижнього кута прямокутника;
Depth – "глибина" прямокутника;
Top – ознака виведення верхньої частини ("даху") прямокутника (true - виводити "дах", false - не виводити)

Наведемо кілька прикладів використання процедур побудови прямокутників та результати їхньої роботи.

```
Rectangle(0,0,GetMaxX div 2,GetmaxY div 2);{Обводить рамкою верхню ліву чверть ек
Bar(0,GetMaxY,20,GetMaxY-20);{Виводить у лівому нижньому куті екрана замальований
                              прямокутник розміром 20 на 20 пікселів
Bar3D(GetMaxX div 2,GetMaxY,GetMaxX div 2+40,GetMaxY-100,10,true);
     {Виводить тривимірний  прямокутник в нижній частині
      екрана глибиною 10 пікселів з верхньою стороною   }
```



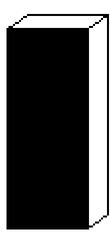


Рис. 20.9. Результати роботи фрагменту програми

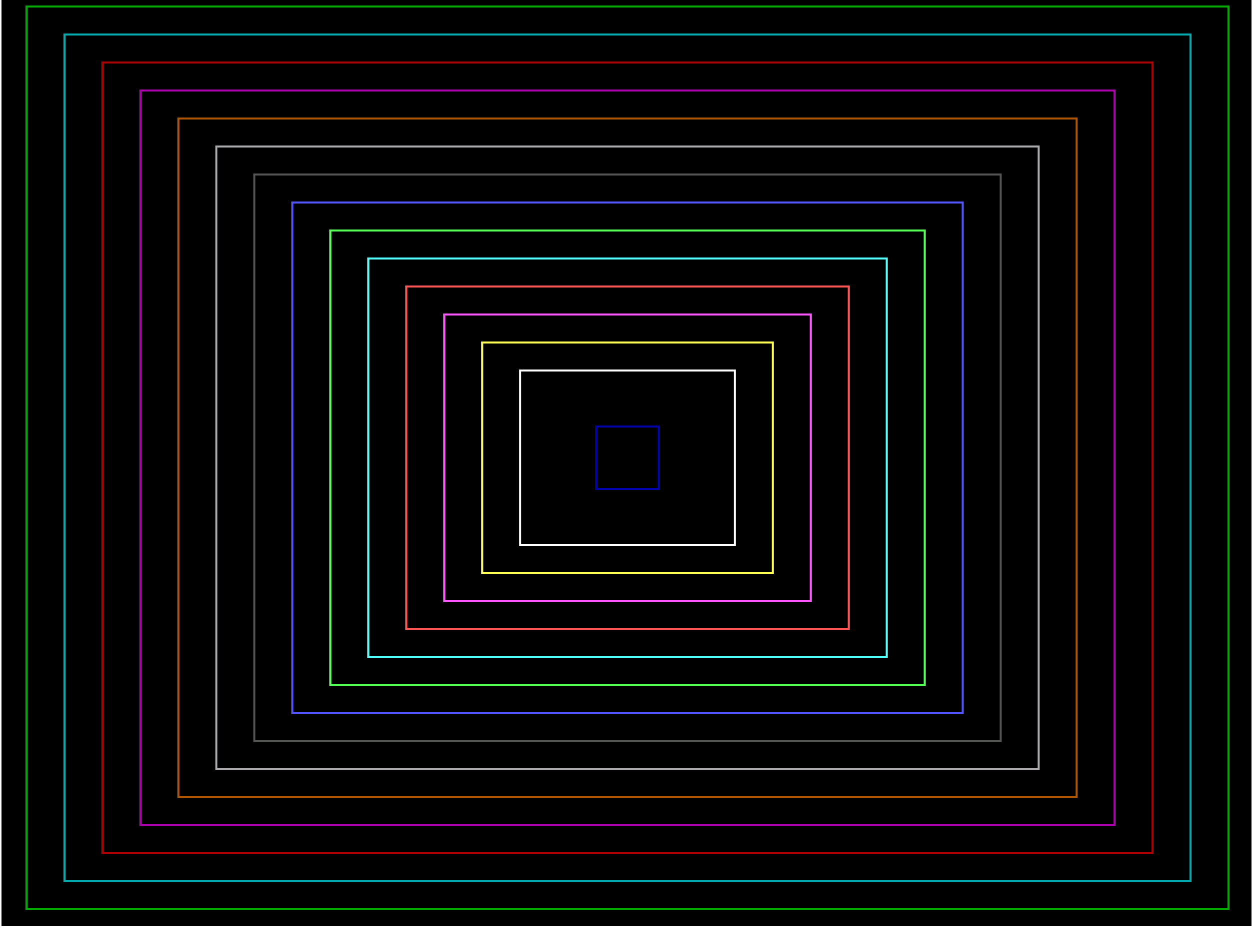
Приклад 20.7. Виведення прямокутників за допомогою процедур `Rectangle` і `Bar`.

Завдання. Створити програму, яка спочатку виводитиме набір пустих прямокутників різного кольору, які послідовно сходяться до центру екрану, а далі виводитиме такі саме прямокутники, але заповнені допустимими типами штрихувань.

Рішення. У першому циклі визначаються координати початкової і кінцевої точок прямокутників, встановлюється черговий колір і виводиться безпосередньо прямокутник. У другому циклі формуються і виводить замальовані різним кольором і стилем прямокутники (стиль заповнення буде розглядатися у п.8)

```
Program Pr20_07;
uses Crt, Graph;
var
  Gd,Gm,Ec :integer;
  x1,y1,x2,y2,i :integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  for i:= 0 to 16 do begin         {Цикл виведення пустих прямокутників}
    x1:=(GetMaxX div 32)*i;        {Визначення координат}
    y1:=(GetMaxY div 32)*i;        {початкової точки}
    x2:=GetMaxX-(GetMaxX div 32)*i;{Визначення координат}
    y2:=GetMaxY-(GetMaxY div 32)*i;{кінцевої точки}
    SetColor(i+1);                 {Встановлення кольору}
```

```
    Rectangle(x1,y1,x2,y2);          {Виведення пустого прямокутника}
  end;
  readln;
  for i:= 0 to 12 do begin           {Цикл виведення штрихованих прямокутників}
    x1:=(GetMaxX div 24)*i;          {Визначення координат}
    y1:=(GetMaxY div 24)*i;          {початкової точки}
    x2:=GetMaxX-(GetMaxX div 24)*i;{Визначення координат}
    y2:=GetMaxY-(GetMaxY div 24)*i;{кінцевої точки}
    SetFillStyle(i,i+1);             {Встановлення параметрів штрихування}
    Bar(x1,y1,x2,y2);                {Виведення штрихованого прямокутника}
  end;
  readln;
  CloseGraph;
END.
```



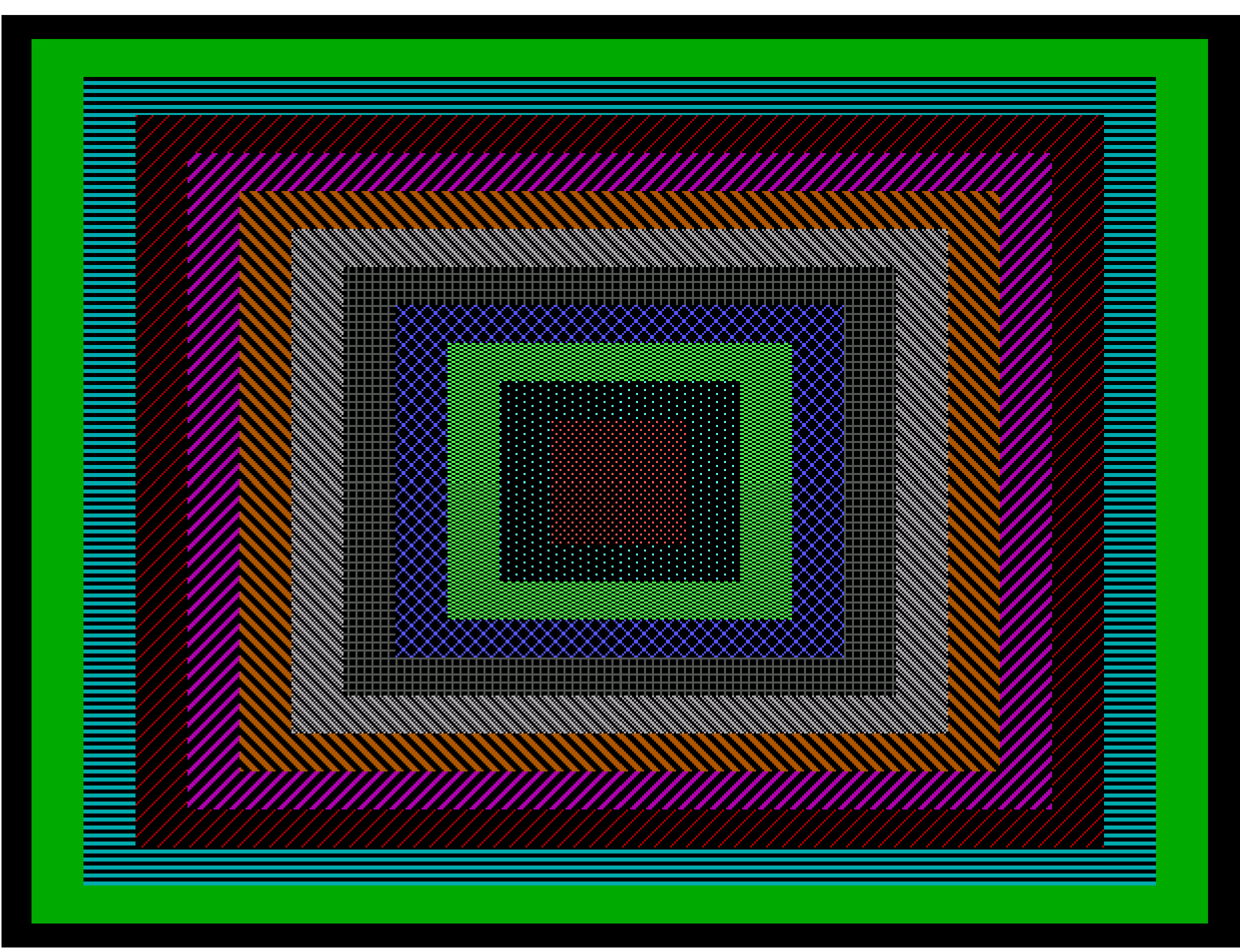


Рис. 20.10. Результати роботи програми Pr20_07:
а) пусті прямокутники; б) штриховані прямокутники

Перегляньте роботу готової програми.

Приклад 20.8. Виведення тривимірних прямокутників.

Завдання. Створити програму, яка по масиву дійсних значень, які є результатами роботи двох відділів за чотири квартали року, дозволяє вивести стовпчикову діаграму. Така діаграма повинна ілюструвати результати роботи кожного відділу окремо і двох відділів разом по кварталах. Біля кожного стовпчика вивести числове значення результату роботи і підсумок роботи за квартал.

Рішення. Результати робот двох відділів по кварталах будемо зберігати у типізованій константі-матриці `Sale`.
Першим етапом після ініціювання графічного режиму є цикл виведення нижнього ряду стовпчикової діаграми, який є результатами роботи першого відділу по кварталах, і виведення підписів до кожного зі стовпчиків. Для таких стовпчиків останній параметр у процедурі `Bar3D` має значення `false`, це пояснюється тим, що над ним буде "надбудовуватись" ще один ряд. Зверніть увагу на перетворення числової інформації у рядкову для виведення підписів.
Другим етапом є цикл виведення верхнього ряду діаграми, який є результатами роботи другого відділу по кварталах, і виведення підписів до кожного зі стовпчиків. Для стовпчиків цього ряду останні параметр процедури `Bar3D` має значення `true`. Зверніть увагу на те, що координати `y` початкової точки цього ряду збігаються із координатами відповідної кінцевої точки першого ряду.
Третім етапом виводяться підсумкові результати роботи двох відділів по кварталах.

Текст програми:

```
Program Pr20_08;
uses Crt, Graph;
var
  Gd,Gm,Ec,i,j:integer;
  S_sale:string;
const
  {Типізована константа із показниками роботи по відділах і кварталах}
  Sale :array[1..2,1..4] of real=
        ((50.4,44.9,62.2,74.3),
         (28.8,29.1,36.5,42.2));
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  SetTextJustify(RightText,TopText);{Параметри вирівнювання тексту для підписів}
  {Цикл виведення нижнього ряду стовпців}
  for i:= 1 to 4 do begin
      SetColor(i);  SetFillStyle(1,i);{Колір і стиль заповнення стовпців}
      Bar3D(i*100,GetMaxY, {Виведення i-го стовпця нижнього ряду}
            i*100+25,GetMaxY-round(Sale[1,i]),
            20,false);
      str(Sale[1,i]:6:2,S_sale);{Перетворення числа у рядок}
      OutTextXY(i*100,GetMaxY-round(Sale[1,i]),S_Sale);{Виведення тексту}
  end;
  {Цикл виведення верхнього ряду стовпців}
  for i:= 1 to 4 do begin
      SetColor(i+4); SetFillStyle(1,i+4);{Колір і стиль заповнення стовпців}
      Bar3D(i*100,GetMaxY-round(Sale[1,i]),{Виведення i-го стовпця верхнього ряду
            i*100+25,GetMaxY-round(Sale[1,i])-round(Sale[2,i]),
            20,true);
      str(Sale[2,i]:6:2,S_sale);{Перетворення числа у рядок}
      OutTextXY(i*100,GetMaxY-round(Sale[1,i])-round(Sale[2,i]),S_Sale);{Виведенн
  end;
  {Параметри виведення підсумкових значень}
  SetTextJustify(LeftText,BottomText);
  SetColor(Yellow);
  {Цикл виведення підсумкових значень}
  for i:=1 to 4 do begin
      str(Sale[2,i]+Sale[1,i]:5:2,S_sale);{Перетворення значення суми у рядок}
      OutTextXY(i*100,GetMaxY-round(Sale[1,i])-round(Sale[2,i]),S_Sale);{Виведенн
  end;
  repeat until KeyPressed;
  CloseGraph;
END.
```

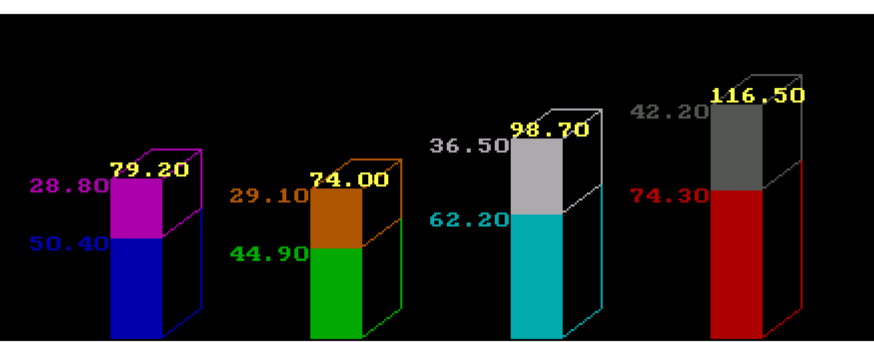


Рис. 20.11. Результат роботи програми `Pr20_08`

Перегляньте роботу готової програми.

6.2. Виведення полігонів

Полігонами є замкнені або відкриті, замальовані або контурні фігури, які складаються із відрізків. Такі фігури можна формувати за допомогою процедури `DrawPoly`, яка дозволяє будувати будь-які багатокутники лініями поточного типу, стилю і ширини.

Формат:

```
DrawPoly(NumbPoints:word; var PolyPoints);
```

де

`NumbPoints` – кількість вершин з яких складається фігура. При виведенні замкненої полігональної фігури його значення повинно на одиницю перевищувати кількість вершин, а координати вершини з номером `N+1` повинні збігатися з координатами вершини `1`.
`PolyPoints` – не типізований параметр, який містить координати кожної вершини фігури.

Якщо треба вивести на екран замальовану полігональну фігуру, скористайтеся процедурою `FillPoly`. Значення всіх параметрів цієї процедури повністю збігаються з параметрами попередньої процедури `DrawPoly`.

Формат:

```
FillPoly(NumbPoints:word; var PolyPoints);
```

Приклад 20.9. Виведення полігонів

Завдання. Створити програму, яка за допомогою процедур `DrawPoly` і `FillPoly` схематично виводить на екран токарний різець.

Текст програми:

```
Program Pr20_09;
uses Crt, Graph;
const {Типізовані константи з координатами базових точок}
  Tool_1 :array[1..6] of PointType =
          ((X:340; Y:180),(X:340; Y:115),(X:320; Y:100),
           (X:335; Y:80 ),(X:370; Y:105),(X:370; Y:180));
  Tool_2 :array[1..6] of PointType =
          ((X:400; Y:380),(X:340; Y:380),(X:330; Y:390),
           (X:320; Y:380),(X:320; Y:360),(X:400; Y:360));
var
  Gd,Gm,Ec:integer;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  DrawPoly(6,Tool_1); {Виведення контуру різця}
  FillPoly(6,Tool_2); {Виведення замальованого різця}
  repeat until KeyPressed;
  CloseGraph;
END.
```

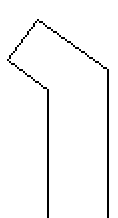


Рис. 20.12. Результат роботи програми Pr20_09

Перегляньте роботу готової програми.

7. Виведення криволінійних примітивів

До криволінійних примітивів, які можна виводити стандартними процедурами Pascal відносяться кола, еліпсів та їхні дуги. Координати центрів задаються у пікселях. В усіх процедурах, де треба визначати кути, використовується полярна система координат. Нульовий кут збігається з додатним напрямком осі X. Відлік кутів ведеться проти годинникової стрілки. Значення кутів повинні бути цілими числами. При виведенні кіл еліпсів та їх дуг використовуються встановлені заздалегідь колір, тип лінії, заповнення.

Процедура `Circle` призначена для виведення контуру кола.

Формат:

```
Circle(X,Y:integer; Radius:word);
```

де
`X,Y` – координати центра кола;
`Radius` – радіус кола.

Процедура `Arc` призначена для виведення дуги кола.

Формат:

```
Arc(X,Y:integer; StartAng,EndAng,Radius :word);
```

Додаткові параметри `StartAng` і `EndAng` визначають початковий і кінцевий кут дуги кола.

Процедура `Ellipse` призначена для побудови еліпсів та їхніх дуг.

Формат:

```
Ellipse(X,Y:integer; StartAng,EndAng,RadX,RadY:word);
```

Параметри `RadX` і `RadY` визначають значення радіуса еліпса вздовж осі X і вздовж осі Y відповідно. Якщо `StartAng=0`, а `EndAng=360`, на екран буде виведено повний еліпс.

Процедура `GetArcCoords` дозволяє отримати інформацію про параметри останньої виведеної дуги кола або еліпса.

Формат:

```
GetArcCoords (var ArcCoords:ArcCoordsType);
```

Змінна `ArcCoords` належить до типу `ArcCoordsType`.

```
type
  ArcCoordsType = record
                    x,y            :integer;
                    xStart,yStart :integer;
                    xEnd,yEnd      :integer
                  end;
```

Процедура повертає змінну `ArcCoords` із значеннями координат центру дуги (`x,y`), початкової (`xStart,yStart`) і кінцевої (`xEnd,yEnd`) точки останньої процедури `Arc` або `Ellipse`.

Процедура `FillEllipse` дозволяє вивести замальований еліпс.

Формат:

```
FillEllipse(X,Y:integer; RadX,RadY:word);
```

Параметри цієї процедури аналогічні до параметрів процедури `Ellipse`. Стиль і колір замальовування визначаються процедурою `SetFillStyle`.

Процедура `PieSlice` дозволяє вивести замальований сектор кола.

Формат:

```
PieSlice(X,Y:integer; StartAng,EndAng,Radius:word);
```

Параметри цієї процедури аналогічні до параметрів процедури `Arc`. Стиль і колір замальовування визначаються процедурою `SetFillStyle`.

Процедура `Sector` дозволяє вивести замальований сектор еліпса.

Формат:

```
Sector(X,Y:integer; StartAng,EndAng,xRad,yRad :word);
```

Параметри цієї процедури аналогічні до параметрів процедури `Ellipse`. Стиль і колір замальовування визначаються процедурою `SetFillStyle`.

Розглянемо два приклади, у яких використовуються процедури роботи з криволінійними

примітивами.

Приклад 20.10. Створення криволінійних примітивів

Завдання. Створити програму виведення малого Герба України (тризубу). Допускається певне спрощення деяких елементів тризубу.

Рішення. На схемі (рис.20.13) наведені позначення вузлових точок, відносно яких будуються елементи зображення. Оскільки зображення Герба є симетричним відносно вертикальної осі, - побудуємо його ліву половину, а праву частину отримаємо віддзеркаленням.

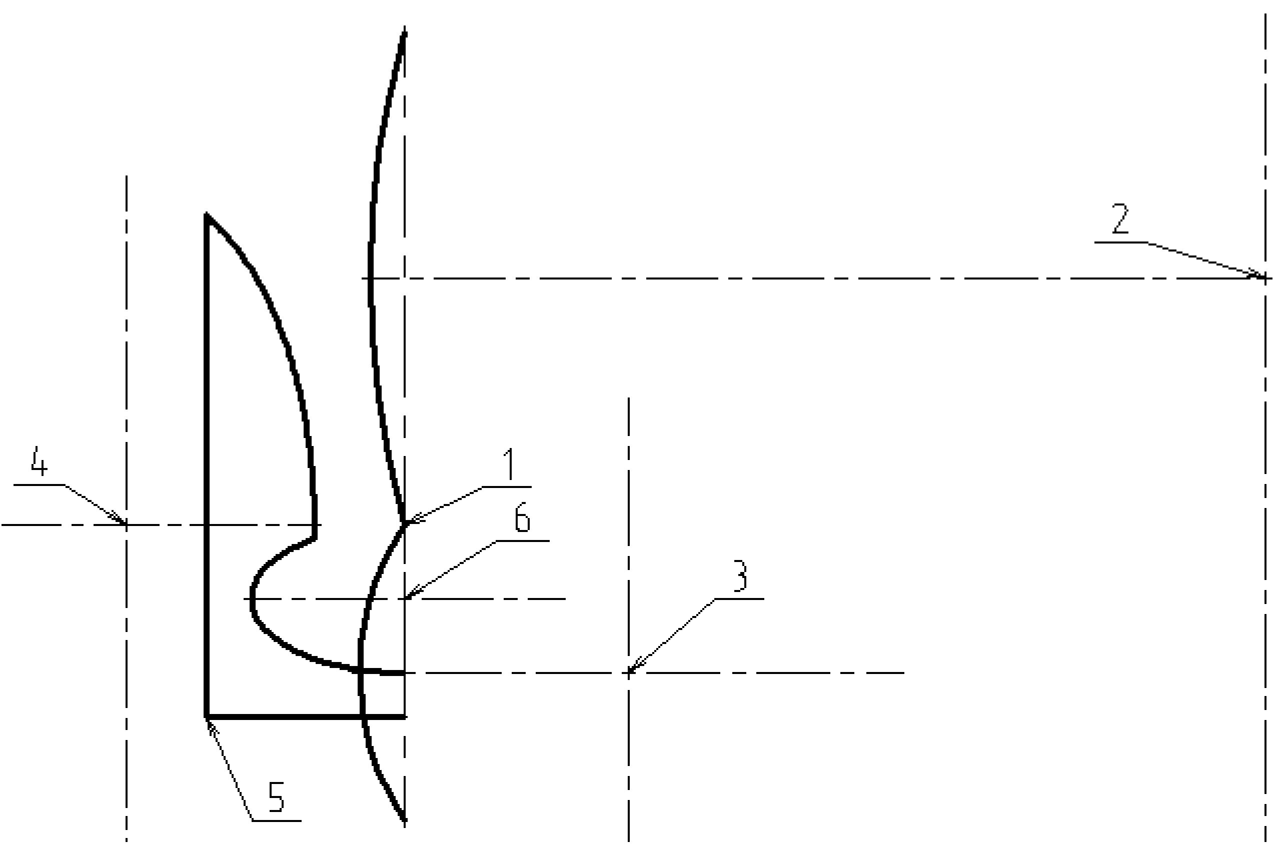


Рис. 20.13. Розрахункова схема для малювання половини малого Герба України

Текст програми:

```
Program Pr20_10;
uses Crt, Graph;
var
  Gd,Gm,Ec,i,j:integer;
  x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6:integer;
  AC:ArcCoordsType;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  SetBkColor(Blue);  {Колір фону}
  SetColor(Yellow);  {Основний колір}
  SetLineStyle(SolidLn,0,ThickWidth);{Стиль лінії: суцільна, товста}
  x1:=GetMaxX div 2; y1:=GetMaxY div 2;{Координати центру екрану}
  x2:=x1+288; y2:=y1-80;
  arc(x2,y2,164,196,300);
```

```
  x3:=x1+82; y3:=y1+60;
  arc(x3,y3,145,215,100);
  x4:=x1-190; y4:=y1;
  ellipse(x4,y4,0,50,120,180);
  GetArcCoords(AC);{Визначення координат останнього еліпсу}
  Line(AC.XEnd,AC.YEnd,AC.XEnd,AC.YEnd+210);{Лінія від точки закінчення еліпсу}
  x5:=AC.XEnd; y5:=AC.YEnd+210;
  Line(x5,y5,x1,y5);
  x6:=x1; y6:=y1+20;
  ellipse(x6,y6,135,270,100,30);
  {Віддзеркалюванні правої половини}
  for i := 0 to GetMaxY do
    for j:= 0 to x1 do
      PutPixel(j+x1,i,GetPixel(x1-j,i));
  repeat until KeyPressed;
  CloseGraph;
END.
```

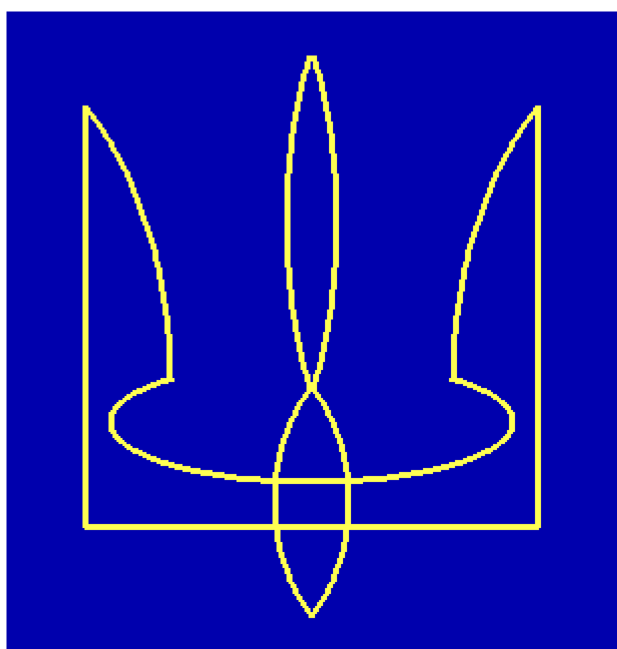


Рис. 20.14. Результат роботи програми PR20_10

Перегляньте роботу готової програми.

Приклад 20.11. Створення криволінійних примітивів із заливкою

Завдання. Створити програму, яка виводитиме на екран емблему кафедри технології машинобудування НТУУ "КПІ". Допускаються певні спрощення в зображенні емблеми.

Рішення. На схемі (рис. 20.15) зображені основні елементи, відносно яких будуються елементи зображення емблеми кафедри технології машинобудування.

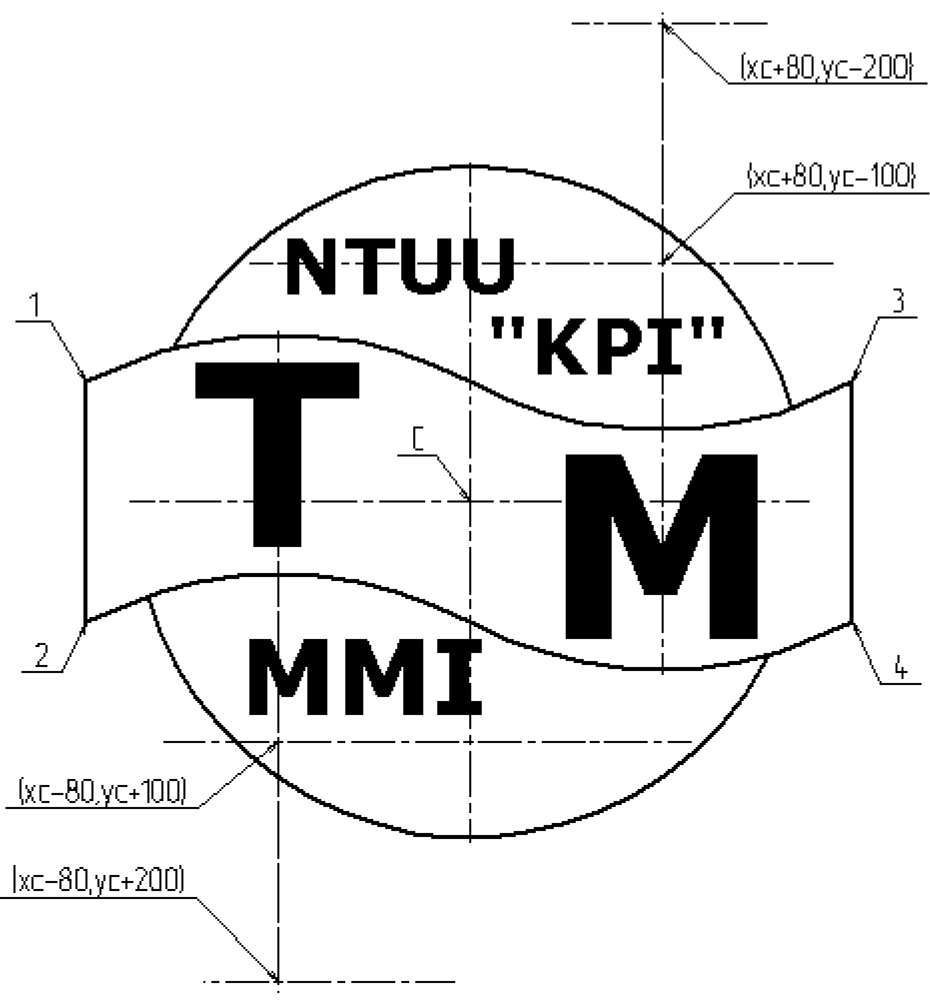


Рис. 20.15. Розрахункова схема для емблеми

Текст програми:

```
Program Pr20_11;
uses Crt, Graph;
var
  GD,GM:integer;
  xc,yc,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6:integer;
  AC:ArcCoordsType;
BEGIN
  GD := Detect;
  InitGraph(GD,GM,'');
  if GraphResult<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2;
  SetLineStyle(SolidLn,0,ThickWidth);
  SetColor(Blue);
  Circle(xc,yc,140);{Виведення кола}
  SetFillStyle(1,Blue);{Визначення стилю заповнення}
  FloodFill(xc,yc,Blue);{Заповнення кола}
  SetColor(Yellow);
  {Формування прапорця}
  arc(xc-80,yc+100,60,120,170);
  GetArcCoords(AC);
  x1:=AC.XEnd; y1:=AC.YEnd;
  arc(xc+80,yc-200,240,300,170);
  GetArcCoords(AC);
  x3:=AC.XEnd; y3:=AC.YEnd;
  arc(xc-80,yc+200,60,120,170);
  GetArcCoords(AC);
```

```
  x2:=AC.XEnd; y2:=AC.YEnd;
  arc(xc+80,yc-100,240,300,170);
  GetArcCoords(AC);
  x4:=AC.XEnd; y4:=AC.YEnd;
  Line(x1,y1,x2,y2);
  Line(x3,y3,x4,y4);
  SetFillStyle(1,Yellow); {Стиль заповнення прапорця}
  FloodFill(xc,yc,Yellow);{Заливка прапорця}
  {Виведення текстової інформації}
  SetColor(Blue);
  SetTextStyle(TriplexFont,0,10);
  SetTextJustify(BottomText,RightText);
  OutTextXY(xc+30,yc-65,'M');
  SetTextJustify(TopText,LeftText);
  OutTextXY(xc-30,yc+25,'T');
  SetColor(Yellow);
  SetTextStyle(DefaultFont,0,3);
  OutTextXY(xc+30,yc-85,'NTUU');
  OutTextXY(xc+110,yc-50,'"KPI"');
  OutTextXY(xc+10,yc+100,'MMI');
  repeat until KeyPressed;
  CloseGraph
END.
```

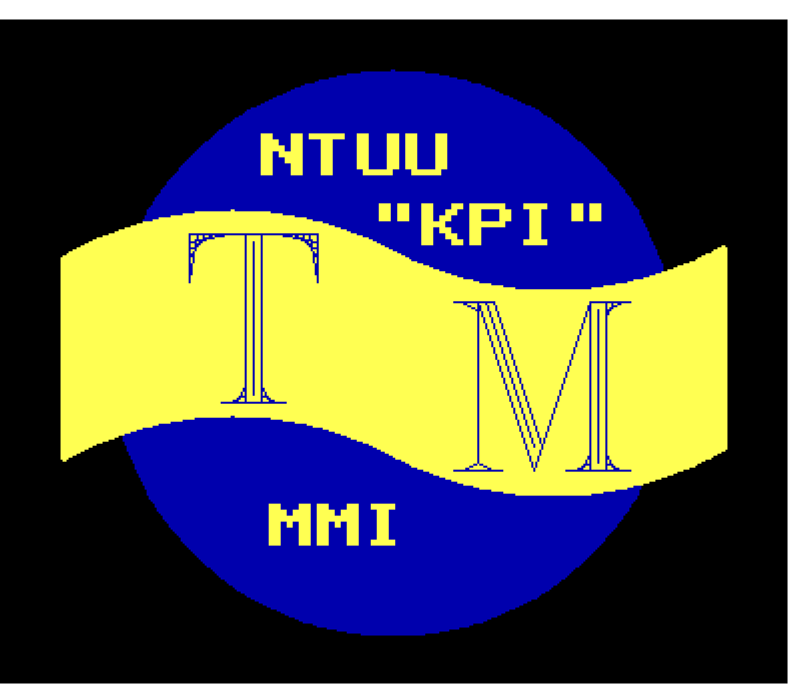


Рис. 20.16. Результат роботи програми PR20_11

Перегляньте роботу готової програми.

8. Визначення кольорів графічних примітивів

8.1. Встановлення і визначення кольорів

Сприймання графічного зображення залежить перш за все від кольору зображення та фону, на якому воно створювалось. Pascal має дві процедури `SetColor` і `SetBkColor` для встановлення цих параметрів.

Формат:

```
SetColor(Color:word);
SetBkColor(Color:word);
```

Процедура SetColor встановлює колір для використання процедурами графічного виведення. Процедура SetBkColor встановлює колір фону, на якому будуть виводитись графічні елементи. До того моменту, поки колір не визначено, для виведення використовується колір, що має максимальний номер, і фон, що має мінімальний номер. Якщо в процедурах SetColor і SetBkColor параметр Color визначає неприпустимий колір, поточний колір залишається без зміни. Для адаптерів EGA/VGA (за винятком режиму EGAHI) значення параметра Color встановлювати константами або значеннями у відповідності до табл. 20.5, або числом від 0 до 63.

Таблиця 20.6. Кольори у графічному режимі

Темні кольори		**Світлі кольори**	
Константа	**Число**	**Константа**	**Число**
EGABlack	0	EGADarkGray	56
EGABlue	1	EGALightBlue	57
EGAGreen	2	EGALightGreen	58
EGACyan	3	EGALightCyan	59
EGARed	4	EGALightRed	60
EGAMagenta	5	EGALightMagenta	61
EGABrown	20	EGAYellow	62
EGALightGray	7	EGAWhite	63

Після ініціювання графічного режиму вся інформація про встановлену палітру знаходиться у змінній стандартного типу PaletteType.

```
type PaletteType = record
                     Size     :byte;
                     Colors   :array[0..MaxColors] of shortint
                   end;
```

де
Size – кількість кольорів у палітрі,
Colors – значення кольорів у регістрах палітри, елементи масиву Colors - цілі числа, що визначають коди кольорів.

Процедура GetDefaultPalette дозволяє отримати інформацію про поточну палітру кольорів.

Формат:

```
GetDefaultPalette(var Palette:PaletteType);
```

Процедура SetPalette дозволяє змінити розташування одного або кількох кольорів у палітрі у тому випадку, коли стандартний порядок з якихось причин не задовольняє програміста

Формат:

```
SetPalette(ColorNum:word; Color:shortint);
```

де
`ColorNum` – номер кольору в оригінальній палітрі,
`Color` – нове значення кольору.

Для визначення кольору можна використовувати такі функції: `GetColor:word` - видає номер поточного основного кольору; `GetBkColor:word` - видає номер поточного кольору фону; `GetMaxColor:word` - видає максимальну кількість кольорів для встановленого режиму.

8.2. Атрибути графічних фігур

Якісне виведення фігур потребує попереднього визначення основного кольору, кольорів фону і заповнювача. Для зручності до модуля `Graph` включена група заздалегідь визначених (стандартних) комбінацій символів-заповнювачів для заповнення внутрішніх і зовнішніх областей графічних фігур. Будемо називати їх маскою.

Процедура `SetFillStyle` дозволяє встановити маску і кольору для заповнювача.

Формат:

```
SetFillStyle(Pattern,Color :word);
```

де
`Pattern` – маска, можливі значення якої наведені у табл. 20.7;
`Color` – колір.

Таблиця 20.7. Стандартні маски заповнювача

Константа	Код	Пояснення	Зразок
EmptyFill	0	Заповнення кольором фону	
SolidFill	1	Заповнення поточним кольором	
LineFill	2	Заповнення символами --, колір - color	
LtSlashFill	3	Заповнення символами // нормальної товщини, колір - color	
SlashFill	4	Заповнення символами // подвійної товщини, колір - color	
BkSlashFill	5	Заповнення символами \\ подвійної товщини, колір - color	
LtBkSlashFill	6	Заповнення символами \\ нормальної товщини, колір - color	
HatchFill	7	Заповнення горизонтально-вертикальним штрихуванням тонкими лініями, колір - color	
		Заповнення штрихуванням навхрест по діагоналі	

XHatchFill	8	"нечастими" тонкими лініями, колір - color	
InterLiaveFill	9	Заповнення штрихуванням навхрест по діагоналі "частими" тонкими лініями, колір - color	
WideDotFill	10	Заповнення "нечастими" точками	
CloseDotFill	11	Заповнення "частими" точками	
UserFill	12	Заповнення по масці, що визначена користувачем, колір - color	

Процедура GetFillSettings дозволяє отримати інформацію про встановлений стиль заповнювача.

Формат:

```
GetFillSettings(var Inf:FilSettingsType);
```

Інформація буде зберігатися у змінній Inf такого типу:

```
type FillSettingsType = record
                          Pattern :word;
                          Color   :word
                        end;
```

Процедура SetFillPattern дозволяє визначити власну маску для заповнювача.

Формат:

```
SetFillPattern(Pattern:FillPatternType; Color:word);
```

де
Pattern – нова маска,
Color – колір маски.

Маска користувача визначається матрицю 8x8 (64 пікселі). Для визначення маски використовується 8 байт, причому кожен байт визначає стан відповідного рядка. Для визначення власних масок використовується стандартний масив типу FillPatternType.

```
type FillPatternType=array[1..8] of byte;
```

Приклад 20.12. Визначення власної маски для заповнювача

Завдання. Створити дві власні маски для заповнення геометричних фігур. Перша маска повинна реалізувати заповнення навхрест, а друга - хвилястими горизонтальними лініями.

Рішення. Оскільки матриця заповнення має розміри 8x8, створимо зображення і представимо їх у шістнадцятковому вигляді. Для першої маски зображення і числа будуть такими.

Маска	Число
	$18
	$24
	$42
	$81
	$81
	$42
	$24
	$18

Для другої маски зображення і числа будуть такими.

Маска	Число
	$60
	$99
	$06
	$00
	$60
	$99
	$06
	$00

Сама програма за допомогою типізованих констант визначає маски, ініціює графічний режим, встановлює потрібні маски і виводить по одному прямокутнику із прикладами нових заповнювань.

Текст програми:

```
Program Pr20_12;
uses Crt, Graph;
const
  MP1:FillPatternType =
         ($18,$24,$42,$81,$81,$42,$24,$18);
  MP2:FillPatternType =
         ($60,$99,$06,$00,$60,$99,$06,$00);
var
  GD,GM:integer;
BEGIN
  GD := Detect;
  InitGraph(GD,GM,'');
  if GraphResult<>grOk then Halt(1);
  SetFillPattern(MP1,Green);
  Bar(10,10,50,100);
  SetFillPattern(MP2,Red);
  Bar(100,100,200,200);
  repeat until KeyPressed;
  CloseGraph
END.
```

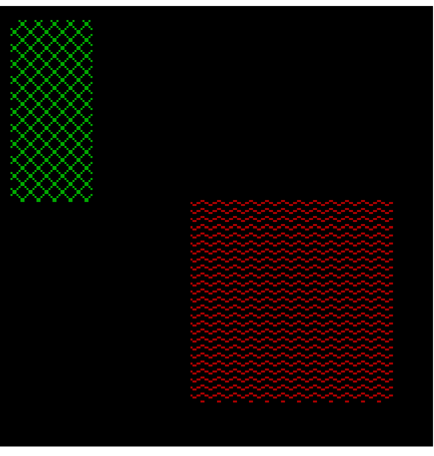


Рис. 20.17. Результат роботи програми `Pr20_12`

Перегляньте роботу готової програми.

Процедура `FloodFill` дозволяє заповнити замкнену фігуру поточною маскою.

Формат:

```
FloodFill(X,Y:integer; Border:word);
```

де
`X,Y` – координати точки всередині або ззовні фігури,
`Border` – колір, до зустрічі з яким буде відбуватися заповнення.

Якщо точка з координатами (`X,Y`) знаходиться всередині замкненої області, то буде заповнюватись внутрішня частина. Якщо вказана точка знаходиться ззовні, то буде заповнюватись весь екран за винятком зони, яка обмежена кольором `Border`.

Приклад 20.13. Заповнення замкненої області

Завдання. Створити програму, яка виводитиме у центрі екрану 12 трикутників різного кольору і заповнює їх всередині штрихуваннями різних типів.

Рішення. Виведення трикутників будемо виконувати у циклі, з параметром якого зв'яжемо кут повороту кожного нового трикутника, його колір і стиль його штрихування. Спочатку розраховуються координати вершин трикутника (`x1,y1`) і (`x2,y2`). Далі за допомогою трьох операторів `line` виводиться черговий трикутник і заповнюється відповідним штрихуванням.

Текст програми:

```
Program pr20_13;
uses Crt, Graph;
var
  GD,GM,i:integer;
  xc,yc,x1,y1,x2,y2,r:integer;
BEGIN
  GD:=Detect;
  InitGraph(GD,GM,'');
  if GraphResult<>grOk then Halt(1);
  xc:=GetMaxX div 2;
  yc:=GetMaxY div 2;
  r := yc;
```

```
  for i:=0 to 11 do begin
    SetFillStyle(i,i);
    SetColor(i);
    x1:=xc+round(R*sin(i*30*(Pi/180)));
    y1:=yc-round(R*cos(i*30*(Pi/180)));
    x2:=xc+round(R*sin((i+1)*30*(Pi/180)));
    y2:=yc-round(R*cos((i+1)*30*(Pi/180)));
    line(xc,yc,x1,y1);
    line(xc,yc,x2,y2);
    line(x1,y1,x2,y2);
    FloodFill((xc+x1+x2) div 3,(yc+y1+y2) div 3, i);
  end;
  repeat until KeyPressed;
  CloseGraph
END.
```

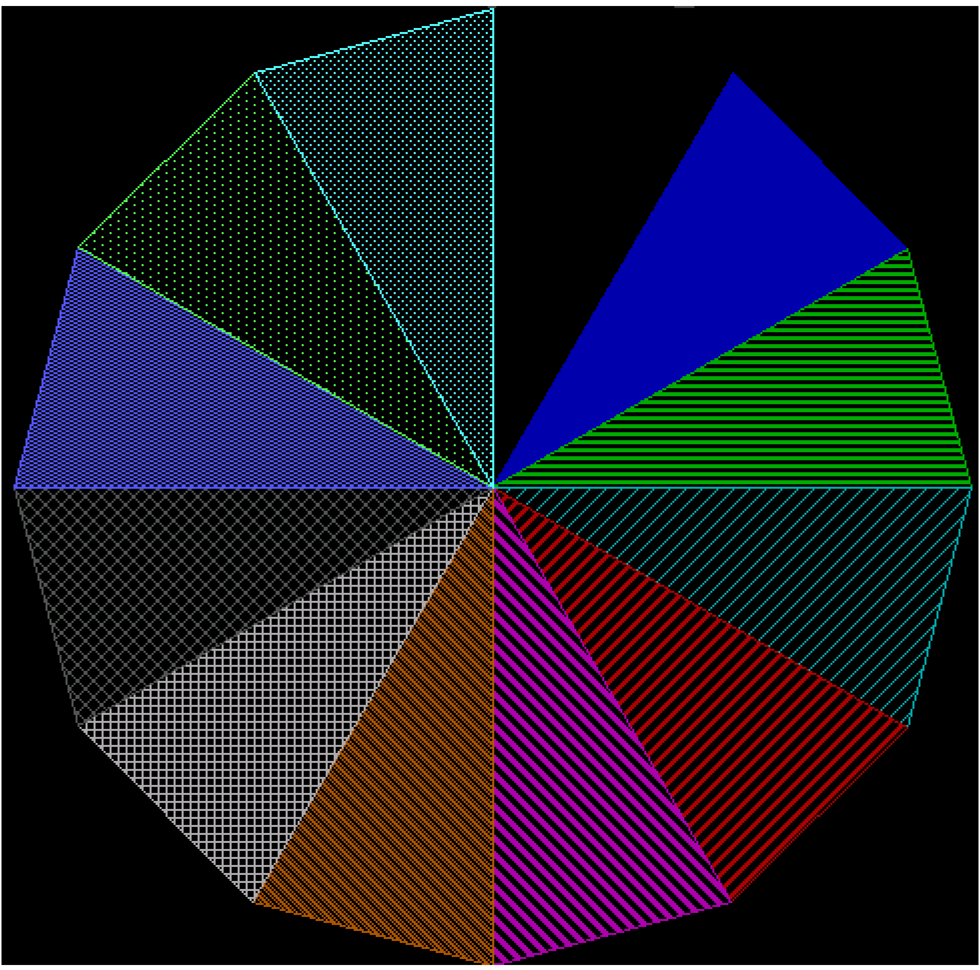


Рис. 20.18. Результат роботи програми Pr20_13

Перегляньте роботу готової програми.

9. Маніпулювання графічними зображеннями

Маніпулювати графічними зображеннями у Turbo Pascal можна різними способами. Можна перемальовувати через певні відрізки часу окремі елементи зображення, можна запам'ятовувати окремі зони екрану і переміщувати їх у нове місце, можна будувати математичну модель зображення і перемальовувати весь екран.

Розглянемо три приклади програм, у яких реалізується маніпулювання графічними

зображеннями.

Приклад 20.14. Електронний годинник зі стрілками

Завдання. Створити програму, яка виводить на екран системний час у вигляді годинника зі стрілками.

Рішення. Скористаємось попередніми програмами (приклади 20.2 і 20.6), у яких вже було створено циферблат і підписи до нього. Допоміжними у цій програмі будуть процедури виведення стрілки годин (`DrawHour`), стрілки хвилин (`DrawMinute`) і стрілки секунд (`DrawSecond`), які по певній годині (хвилині, секунді) та координаті початкової точки виводять відповідну стрілку на екран.
Після виведення циферблату і підписів до нього, визначається системний час комп'ютера (процедура `GetTime`) і виводяться відповідні стрілки годинника.
Далі основу програми складає цикл `repeat..until`, який оновлює зображення на екрані кожну секунду. Вкладений цикл `repeat..until` визначає момент такої події.
Зверніть увагу на те, що процедури виведення стрілок використовуються двічі - для виведення попередньої стрілки (`*Old`) білим кольором і виведення нової стрілки (`*New`) зеленим кольором.
До основного циклу включено також виведення текстів посередині циферблату. Це зроблено для того, що стрілки підчас руху не затирали надписи.

Програма:

```
Program Pr20_14;
uses Crt,Graph,DOS;
var
  Gd,Gm,Ec :integer;
  xc,yc,xn,yn,xk,yk,Rn,Rk,i:integer;
  s_time:string;
  HourOld,MinuteOld,SecondOld,HourNew,MinuteNew,SecondNew,S100:word;

{Процедура виведення стрілки годин}
procedure DrawHour(Hour:word;xc,yc:integer);
var x1,y1,x2,y2:integer;
begin
  SetLineStyle(SolidLn,0,ThickWidth);
  x1:=xc+round(100*sin(Hour*30*Pi/180));
  y1:=yc-round(100*cos(Hour*30*Pi/180));
  x2:=xc-round(10*sin(Hour*30*Pi/180));
  y2:=yc+round(10*cos(Hour*30*Pi/180));
  Line(x1,y1,x2,y2);
end;

{Процедура виведення стрілки хвилин}
procedure DrawMinute(Minute:word;xc,yc:integer);
var x1,y1,x2,y2:integer;
begin
  SetLineStyle(SolidLn,0,ThickWidth);
  x1:=xc+round(150*sin(Minute*6*Pi/180));
  y1:=yc-round(150*cos(Minute*6*Pi/180));
  x2:=xc-round(8*sin(Minute*6*Pi/180));
  y2:=yc+round(8*cos(Minute*6*Pi/180));
  Line(x1,y1,x2,y2);
end;

{Процедура виведення стрілки секунд}
procedure DrawSecond(Second:word;xc,yc:integer);
var x1,y1,x2,y2:integer;
```

```
begin
  SetLineStyle(SolidLn,0,1);
  x1:=xc+round(150*sin(Second*6*Pi/180));
  y1:=yc-round(150*cos(Second*6*Pi/180));
  x2:=xc-round(8*sin(Second*6*Pi/180));
  y2:=yc+round(8*cos(Second*6*Pi/180));
  Line(x1,y1,x2,y2);
end;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2;
  Rn:=(GetMaxY div 2)-40;
  SetBkColor(White);{Фоновий колір циферблату}
  i:=0;
  repeat {Цикл виведення циферблату і підписів до нього}
    xn:=xc+round(Rn*sin(i*Pi/180));
    yn:=yc-round(Rn*cos(i*Pi/180));
    if (i mod 30)=0
      then  begin Rk:=(GetMaxY div 2)-80;
                  SetColor(Green);
            end
      else  begin Rk:=(GetMaxY div 2)-60;
                  SetColor(Blue);
            end;
    xk:=xc+round(Rk*sin(i*Pi/180));
    yk:=yc-round(Rk*cos(i*Pi/180));
    Line(xn,yn,xk,yk);
    SetColor(Green);
    case (i div 30) of
      1,2  :SetTextJustify(LeftText,BottomText);
      3    :SetTextJustify(LeftText,CenterText);
      4,5  :SetTextJustify(LeftText,TopText);
      6    :SetTextJustify(CenterText,TopText);
      7,8  :SetTextJustify(RightText,TopText);
      9    :SetTextJustify(RightText,CenterText);
      10,11:SetTextJustify(RightText,BottomText);
      12   :SetTextJustify(CenterText,BottomText);
    end;
    case (i div 30) of
       3,6,9,12: SetTextStyle(TriplexFont,0,5);
       else      SetTextStyle(DefaultFont,0,2);
    end;
    if ((i mod 30)=0) and (i<>0)
    then begin
       str(i div 30,s_time);
       OutTextXY(xn,yn,s_time);
    end;
    inc(i,6);
  until i>360;

  {Визначення системного часу}
  GetTime(HourOld,MinuteOld,SecondOld,S100);
  {Виведення стрілок відповідно до часу}
     DrawHour(HourOld,xc,yc);
     DrawMinute(MinuteOld,xc,yc);
     DrawSecond(SecondOld,xc,yc);

  {Цикл виведення стрілок і внутрішніх надписів кожну секунду}
  repeat
     repeat {Цикл, який визначає момент змін}
        GetTime(HourNew,MinuteNew,SecondNew,S100);
```

```
      until SecondOld<>SecondNew;
      SetColor(White);DrawHour(HourOld,xc,yc);{Видалення попередньої стрілки годин
      SetColor(Green);DrawHour(HourNew,xc,yc);{Виведення нової стрілки годин}
      SetColor(White);DrawMinute(MinuteOld,xc,yc);{Видалення попередньої стрілки х
      SetColor(Green);DrawMinute(MinuteNew,xc,yc);{Виведення нової стрілки хвилин}
      SetColor(White);DrawSecond(SecondOld,xc,yc);{Видалення попередньої стрілки с
      SetColor(Green);DrawSecond(SecondNew,xc,yc);{Виведення нової стрілки секунд}
      {Запам'ятовування поточного часу}
      HourOld:=HourNew;
      MinuteOld:=MinuteNew;
      SecondOld:=SecondNew;
      {Виведення внутрішніх надписів}
      SetColor(Blue);
      SetTextStyle(DefaultFont,0,1);
      SetTextJustify(CenterText,BottomText);
      OutTextXY(Xc,round(Yc*0.75),'National Technical University of Ukraine');
      SetTextJustify(CenterText,TopText);
      OutTextXY(Xc,round(Yc*0.75),'<<Kiev Polytechnic Institute>>');
      OutTextXY(Xc,round(1.25*Yc),'Machinebuilding Department');
   until KeyPressed;
   CloseGraph;
END.
```

Рис. 20.19. Результат роботи програми PR20_14

Перегляньте роботу готової програми.

Модуль Graph містить кілька процедур і функцій, які дозволяють зберігати та поновлювати окремі фрагменти зображення на екрані.

Для маніпулювання фрагментом треба перш за все визначити його розмір. Зробити це можна

з допомогою функції `ImageSize`.

Формат:

```
ImageSize(x1,y1,x2,y2:integer):word;
```

де
`x1,y1,x2,y2` – координати (у пікселях) верхнього лівого та правого нижнього кута прямокутної області екрана. Функція визначає розмір вказаного фрагмента в байтах.

Збереження образу здійснюється процедурою `GetImage`.

Формат:

```
GetImage(x1,y1,x2,y2:integer; var BitMap);
```

де `BitMap` - додатковий не типізований параметр, який повинен дорівнювати або бути більшим за 6 плюс розмір пам`яті, що відведена для області екрана. Для визначення розміру пам`яті, яка необхідна для параметра `BitMap`, використовуйте описану вище функцію `ImageSize`.

Повернути образ фрагмента на екран можна процедурою `PutImage`.

Формат:

```
PutImage(x,y:integer; var BitMap; Mode:word);
```

де
`x, y` – координати точки екрана, від якої вправо і вниз буде виводитись образ.
`BitMap` – змінна, у якій зберігається образ.
`Mode` – визначає режим виведення фрагмента.

Можливі значення `Mode` наведені у табл. 20.8. Кожна константа відповідає двійковій операції, яка визначатиме правила накладання зображень та утворюваних кольорів.
Режим `XorPut` забезпечує виведення і видимість фрагмента на будь-якому фоні. Цікавою особливістю цього режиму є те, що повторне виведення цього ж фрагмента поновлює початковий стан екрана. Це пояснюється наступним рівнянням:
`A xor B xor B = A`.

Таблиця 20.8. Режими виведення прямокутних фрагментів

Константа	Число	Операція	Опис стилю заповнення
NormalPut (CopyPut)	0	MOV	Нове зображення повністю замінює попереднє
XORPut	1	XOR	Код кольору пікселів утворюється за допомогою операції логічного виключного "АБО"
OrPut	2	OR	Код кольору пікселів утворюється за допомогою логічного "АБО"
AndPut	3	AND	Код кольору пікселів утворюється за допомогою логічного "І"

NotPut	4	NOT	Код кольору пікселів утворюється за допомогою логічного "Ні" (інвертування)

Використання вищезазначених процедур проілюструємо наступним прикладом.

Приклад 20.15. Рухома кулька

Завдання. Створити програму, в якій по екрану буде переміщуватись і відбиватись від боків екрану кулька. При відскакуванні кулька повинна змінювати свою швидкість. По правому і лівому боках повинні бути розташовані "ворота". Програма повинна додатково підраховувати і виводити скільки раз кулька потрапила до воріт.

Рішення. Після ініціювання графічного режиму встановлюються основний і фоновий кольори і у центрі екрану формується зображення кульки, яка буде переміщуватись.
Далі відбувається запам'ятовування зображення кульки. Для цього спочатку за допомогою функції ImageSize визначається розмір зображення. Далі, за допомогою функції GetMem виділяється відповідні ділянка оперативної пам'яті для зображення. Останнім кроком цього етапу є передача зображення вказаної області до змінної Emblema^, яка є покажчиком на область оперативної пам'яті.
Далі виводяться ліві і праві ворота, а також визначаються початкові напрямки переміщення кульки (Right,Top) і швидкості (SpeedHor,SpeedVer), з якими вона буде переміщуватись.
Наступний етап є основним. Цикл repeat..until виконується до натискання користувачем будь-якої клавіші на клавіатурі. Всередині цього циклу спочатку на попереднє місце вставляється (PutImage))оригінальне зображення кульки (фактично у цей момент вона зникає з екрану). Далі перевіряється, чи не виходить кулька за ліву або праву межу. Якщо одна з таких умов виконується, то змінюється напрямок горизонтального (Right) переміщення і аналізується, чи не потрапила кулька у ліві або праві ворота. У випадку потрапляння реалізуються відповідні блоки, які збільшують лічильники м'ячів (BallRed,BallGreen) і виводять нові значення на екран. Також тут розраховується координата x нової точки і значення нової горизонтальної швидкості (SpeedHor). Далі перевіряється, чи не дісталась кулька верхнього або нижнього боку екрану. Якщо це так, змінюється напрямок вертикального (Top) переміщення, змінюється вертикальна швидкість (SpeedVer) і розраховуються нова координата y точки виведення кульки. Якщо ж кулька переміщується всередині екрану, то для неї відповідно до поточних напрямків переміщення (Right і Top) розраховуються координати нової точки.
Перед завершенням основного циклу виводиться нова кулька (PutImage) і через певну паузу (Delay) цикл починається знову.

Текст програми:

```
Program Pr20_15;
uses Crt,Graph,DOS;
var
  Gd,Gm,Ec :integer;
  xc,yc:integer;
  Emblema:pointer; {Покажчик на область пам'яті}
  Size:word;{Розмір зображення}
  Right,Top:boolean;{Напрямок переміщення}
  SpeedVer,SpeedHor:integer;{Швидкості по горизонталі і вертикалі}
  BallRed,BallGreen:Word;{Лічильники голів}
  s:string;
BEGIN
  GD:=Detect;
  InitGraph(Gd,GM,'');
```

```
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  xc:=GetMaxX div 2;  yc:=GetMaxY div 2;
  SetColor(Blue);
  SetFillStyle(1,Yellow);

  {Формування кульки-емблеми}
  Circle(xc+20,yc+20,20);
  FloodFill(xc+20,yc+20,Blue);
  SetTextStyle(DefaultFont,0,2);
  SetTextJustify(CenterText,CenterText);
  OutTextXY(xc+20,yc+20,'TM');

  {Запам'ятовування зображення}
  Size := ImageSize(xc-20,yc-20,xc+20,yc+20); {Визначення розміру зображення}
  GetMem(Emblema, Size);{Виділення пам'яті під зображення}
  GetImage(xc,yc,xc+40,yc+40,Emblema^);{Запам'ятовування зображення}
  {Ліві ворота}
  SetFillStyle(1,Red);
  Bar(0,GetMaxY div 4,20, 3*GetMaxY div 4);
  {Праві ворота}
  SetFillStyle(1,Green);
  Bar(GetMaxX-20,GetMaxY div 4,GetMaxX, 3*GetMaxY div 4);

 {Початкові напрямки переміщення кульки}
  Right:=true;
  Top:=true;
  randomize; {Ініціювання датчика випадкових чисел}

 {Формування початкових швидкостей переміщення}
  SpeedVer:=random(10)+1;
  SpeedHor:=random(20)+1;
  SetFillStyle(0,1);
  SetTextJustify(LeftText,TopText);

  {Основний цикл програми}
  repeat
    PutImage(xc,yc,Emblema^,XorPut);{Вставити зображення }
    {Перевірка на вихід за межі екрану по горизонталі}
    if (xc>(GetMaxX-40)) or (xc<=0)
      then begin
             Right := not Right; {Зміна напрямку переміщення по горизонталі}
             {Перевірка на попадання у ворота}
             if (yc>GetMaxY div 4 - 20) and (yc< 3*GetMaxY div 4 - 20)
             then if xc<=0 {Попадання у ліві ворота}
                  then begin
                    inc(BallRed); {Збільшити лічильник голів у ліві ворота}
                    Bar(0,0,50,20);{Витерти попередній рахунок}
                    str(BallRed:3,s);{Перетворити число у рядок}
                    OutTextXY(0,0,s);{Вивести кількість голів у ліві ворота}
                  end
                  else begin {Попадання у праві ворота}
                    inc(BallGreen);{Збільшити лічильник голів у праві ворота}
                    Bar(GetMaxX,0,GetMaxX-50,20);{Витерти попередній рахунок}
                    str(BallGreen:3,s);{Перетворити число у рядок}
                    OutTextXY(GetMaxX-50,0,s);{Вивести кількість голів у праві во
             end;
             {Розрахувати координату x нової точки для виведення кульки}
             if Right then xc:=xc+SpeedHor
                      else xc:=xc-SpeedHor;
             speedHor := random(20)+1;{Визначити нову швидкість}
           end;
    {Перевірка на вихід за межі екрану по вертикалі}
    if (yc>(GetMaxY-40)) or (yc<=0)
```

```
      then begin
             Top := not Top;{Зміна напрямку переміщення по вертикалі}
             {Розрахувати координату у нової точки для виведення кульки}
             if Top then yc:=yc-SpeedVer else yc:=yc+SpeedVer;
             speedVer:= random(10)+1;{Визначити нову швидкість}
           end;
    {Розрахунок координат нової точки під час руху всередині екрану}
    if Right then xc:=xc+SpeedHor else xc:=xc-SpeedHor;
    if Top   then yc:=yc-SpeedVer else yc:=yc+SpeedVer;
    PutImage(xc,yc,Emblema^,XorPut);{Вивести кульку}
    Delay(500); {Пауза}
  until KeyPressed; {Завершення основного циклу програми}
  CloseGraph;
END.
```

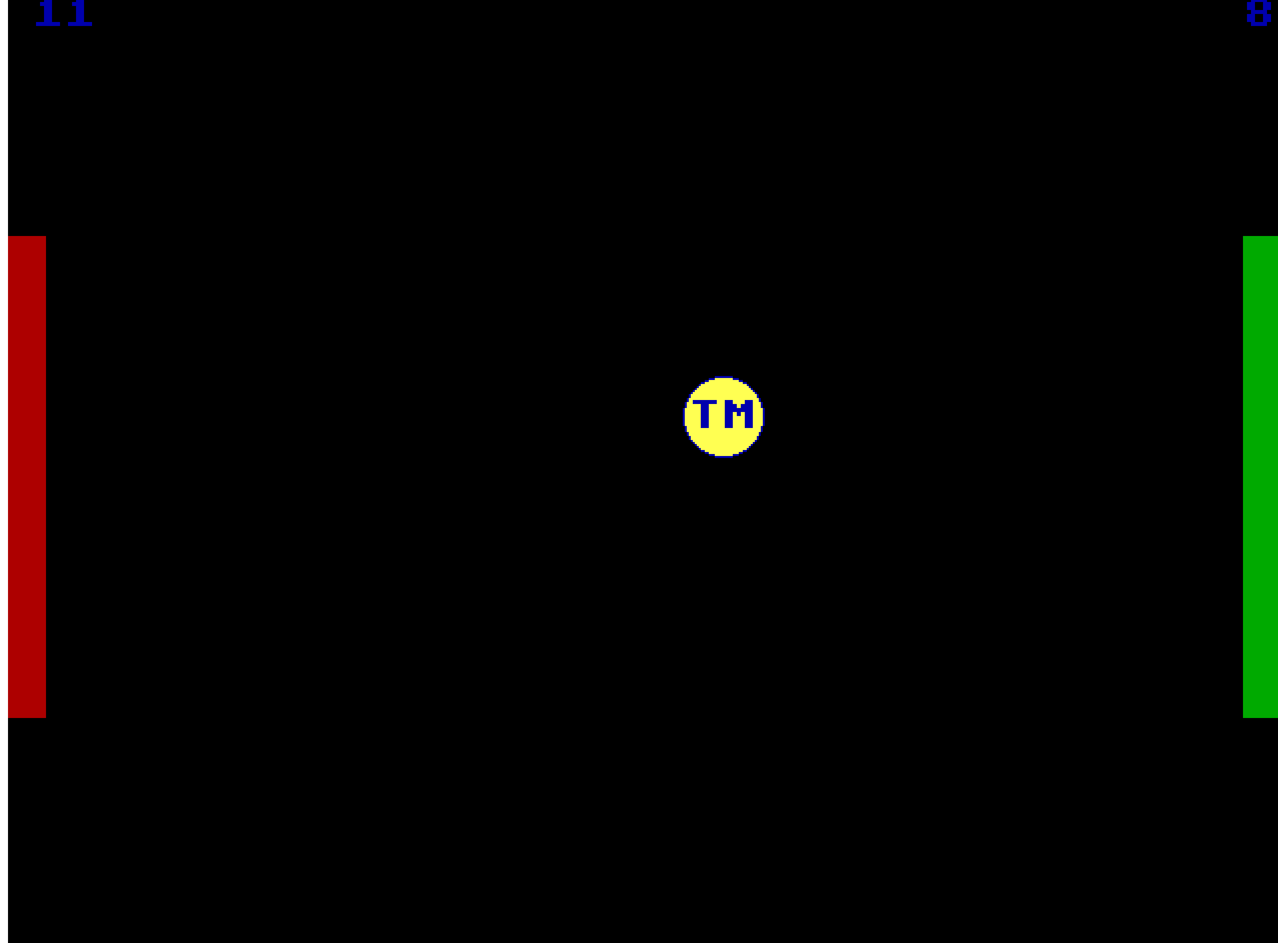


Рис. 20.20. Фрагмент роботи програми PR20_15

Перегляньте роботу готової програми.

Часто при створенні анімацій є можливість створити математичну модель об'єкту. У такому випадку слід визначити координати вузлових точок моделі у системі координат екрану і вивести елементи об'єкту, прив'язуючись до таких вузлових точок. Розглянемо приклад побудови математичної моделі механізму і імітацію його роботи на наступному прикладі.

Приклад 20.16. Моделювання роботи механізму

Завдання. Створити програму для імітації роботи кривошипно-повзунного механізму, який містить два повзуни, що переміщуються у взаємно перпендикулярних площинах. Довжини кривошипу та шатунів вводяться на початку програми.

Рішення. Для реалізації програми спочатку слід створити і проаналізувати розрахункову схему (модель) механізму (рис. 20.21).

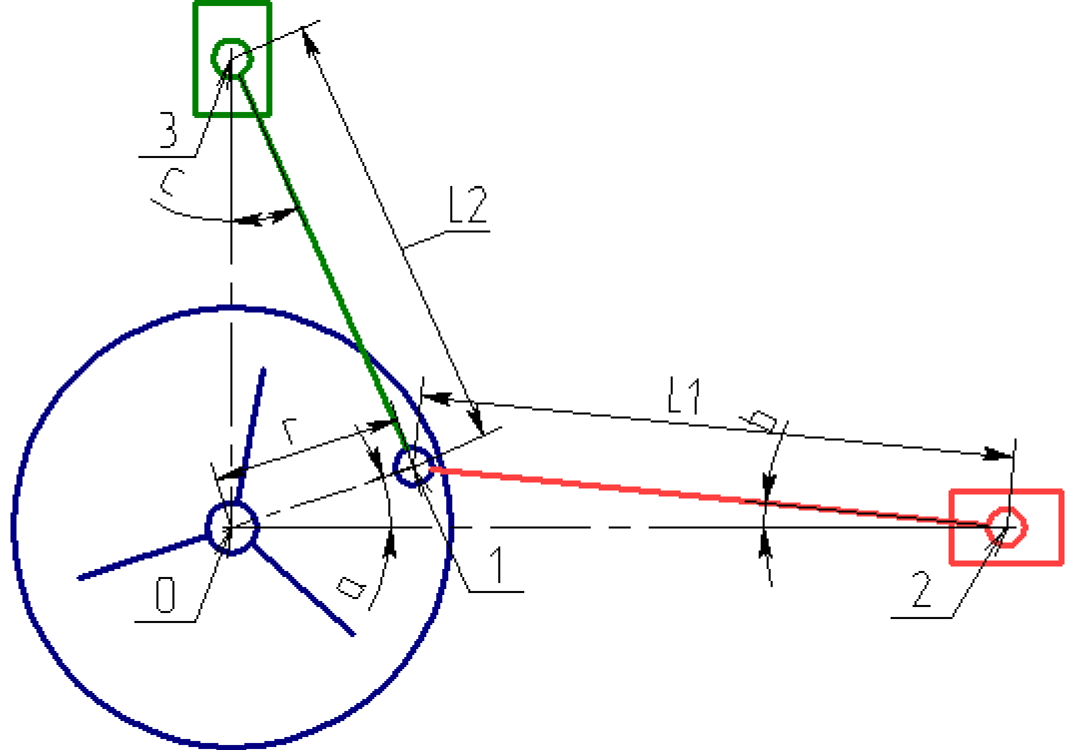


Рис. 20.21. Розрахункова схема механізму

Нерухомою точкою є точка `0`, розміщувати яку на екрані будемо ближче до нижнього лівого кута.
Відносно точки `0` зі сталою швидкістю обертається точка `1`. Положення точки `1` визначається радіусом кривошипу `r` і кутом повороту `a`. Розрахункові залежності для координат точки `1` будуть такими: `x1=x0+r*cos(a); y1=y0-r*sin(a)`.
Положення точки `2` залежить від довжини кривошипу `r`, довжини шатуна `L1`, кута повороту кривошипу `a` і кута повороту шатуна `b`. Розрахункові залежності для координат точки `2` будуть такими: `x2=x0+r*cos(a)+L1*cos(b); y2=y0`, де кут `b=arcsin(-r*sin(a)/L1)`.
Положення точки `3` залежить від довжини кривошипу `r`, довжини шатуна `L1`, кута повороту кривошипу `a` і кута повороту шатуна `c`. Розрахункові залежності для координат точки `3` будуть такими: `x3=x0; y3=y0-r*cos(90-a)-L2*cos(c)`, де кут `c=arcsin(-r*sin(a)/L2)`.

Для реалізації програму в ній створена внутрішня процедура (`ReadReal`) безпомилкового введення дійсних даних і функція `arcsin`, якої немає серед стандартних функцій. Після введення параметрів механізму та ініціювання графічного режиму визначаються координати базової точки `0`. Ця точка є єдиною нерухомою під час роботи механізму точкою. Координати решти точок будемо визначати під час анімації.
Сама анімація виконується у циклі по параметру `i`, перед початком якого ініціюється початкове значення кута повороту кривошипу `a`.
На початку циклу встановлюється синій колір, визначаються поточні координати точки `1`, виводяться усі елементи кривошипу, в тому числі і три радіальні риски (цикл `for`).
Наступним кроком є визначення кута `b` нахилу горизонтального кривошипу, координат його точки 2 і виведення червоним кольором усіх його елементів. Аналогічні дії виконуються і для

вертикального повзуна, усі елементи якого виводяться зеленим кольором.
Наприкінці циклу робиться невелика пауза, після якої очищується екран, розраховується нове значення кута повороту кривошипу a і цикл повторюється до натискання користувачем будь-якої клавіші на клавіатурі.

Текст програми:

```
Program Pr20_16;
uses Crt, Graph;
var
  Gd,Gm,Ec,i:integer;
  x0,y0,x1,y1,x2,y2,x3,y3:integer;
  xc1,yc1,xc2,yc2:array [1..3] of integer;
  a,b,c,speed:real;
  R,L1,L2,D:real;

procedure ReadReal(Name:string;RMin,RMax:real;var R:real);
var x,y,Code:integer;
    s:string;
begin
  x:=WhereX;  y:=WhereY;
  repeat
    GotoXY(x,y); DelLine;
    write(Name,' (',RMin:4:0,' ..',RMax:4:0,') = ');
    readln(s);
    val(s,R,Code);
  until (Code=0) and (R>=RMin) and (R<=RMax)
end;

function arcsin(x:real):real;
begin
  arcsin:=arctan(x/sqrt(1-sqr(x)));
end;{arcsin}

BEGIN
  clrscr;
  {Введення параметрів механізму}
  writeln('Input mechanizm parameters, please...');
  ReadReal('R    ',10,80,R);
  D:=2*R+20;
  ReadReal('L1   ',D,2*D,L1);
  ReadReal('L2   ',D,1.5*D,L2);
  ReadReal('Speed',-20,20,speed);

  {Перехід у графічний режим}
  GD:=Detect;
  InitGraph(Gd,GM,'');
  EC:=GraphResult;
  if EC<>grOk then Halt(1);
  {Визначення початкових параметрів}
  x0:=Round(D/2)+10;
  Y0:=GetMaxY-round(D/2)-10;
  a:=0;
  SetFillStyle(0,0);

  {Цикл анімації роботи механізму}
  repeat
    {Визначення координат точки 1 і виведення кривошипу}
    SetColor(Blue);
    x1:=x0+round(R*cos(a*Pi/180));
    y1:=y0-round(R*sin(a*Pi/180));
    circle(x0,y0,Round(D/2));
    circle(x0,y0,3);
```

```
    circle(x1,y1,3);
    {Цикл виведення рисок кривошипу}
    for i:= 1 to 3 do begin
     xc1[i]:=x0+round(10*cos(2*Pi/3*(i-1)+a*Pi/180));
     yc1[i]:=y0-round(10*sin(2*Pi/3*(i-1)+a*Pi/180));
     xc2[i]:=x0+round((R-10)*cos(2*Pi/3*(i-1)+a*Pi/180));
     yc2[i]:=y0-round((R-10)*sin(2*Pi/3*(i-1)+a*Pi/180));
     Line(xc1[i],yc1[i],xc2[i],yc2[i]);
    end;
    {Виведення горизонтального повзуна і шатуна}
    SetColor(Red);
    b:=arcsin(-R*sin(a*Pi/180)/L1);{Визначення кута нахилу шатуна}
    {Визначення координат точки 2}
    x2:=x0+round(R*cos(a*Pi/180)+L1*cos(b));
    y2:=y0;
    Line(x1,y1,x2,y2);{Шатун}
    rectangle(x2-20,y2-10,x2+20,y2+10);{Повзун}
    circle(x2,y2,3);{Коло в центрі повзуна}
    {Виведення вертикального повзуна і шатуна}
    SetColor(Green);
    c:=arcsin(-R*sin(Pi/2-a*Pi/180)/L1);{Визначення кута нахилу шатуна}
    {Визначення координат точки 3}
    y3:=y0-round(R*sin(a*Pi/180)+L2*cos(c));
    x3:=x0;
    Line(x1,y1,x3,y3);{Шатун}
    rectangle(x3-10,y3-20,x3+10,y3+20);{Повзун}
    circle(x3,y3,3);{Коло в центрі повзуна}
    {Завершельна частина циклу анімації}
    delay(1000);{Пауза}
    ClearDevice;{Очищення екрану}
    a:=a+round(speed);{визначення наступного кута повороту кривошипу}
  until KeyPressed;
  CloseGraph;
END.
```

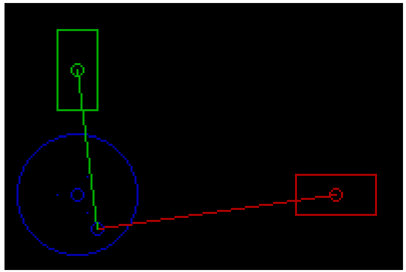


Рис. 20.22. Фрагмент роботи програми PR20_16

Перегляньте роботу готової програми.

10. Включення драйверів і шрифтів у програму

Звичайно графічний драйвер (файл з розширенням `*.bgi`) та графічні шрифти (файли з розширенням `*.chr`) не входять до складу програмного коду (файла з розширенням `*.exe`), а завантажуються як оверлеї під час виконання процедур `InitGraph` та `SetTextStyle`. Це не завжди буває зручно, особливо коли програма вже готова. Turbo Pascal дає змогу підключити до готової програми необхідні драйвери та шрифти.

Робити це слід у такому порядку:

- За допомогою утиліти `BinObj.exe` потрібні файли драйверів та шрифтів перетворити

у об`єктні файли з розширенням `*.obj`;
- Об`єднати результуючі `obj`-файли з вашою програмою;
- Перед викликанням процедури `InitGraph` зареєструвати файл драйвера та файли шрифтів.

Реєстрація драйверів та шрифтів здійснюється, відповідно, процедурами `RegisterBGIDriver` і `RegisterBGIFont`.

Формати:

```
RegisterBGIDriver(Driver:pointer):integer;
RegisterBGIFont(Font:pointer):integer;
```

Якщо вас не турбує розмір `exe`-файла, тоді доцільно створити дві `tpu`-бібліотеки, у яких будуть зберігатися усі драйвери і шрифти, які можуть знадобитися. Перед створенням бібліотек потрібно з допомогою утиліти `BinObj.exe` перетворити усі потрібні файли драйверів та шрифтів у `obj`-формат.

Текст модуля драйверів для адаптерів EGA і VGA:

```
Unit BGIDriv;
INTERFACE
  procedure EgaVgaProc;
IMPLEMENTATION
  procedure EgaVgaProc;external;
  {$L  EGAVGA.obj}
END.
```

Текст модуля для шрифтів `BoldFont`, `GothicFont` і `TriplexFont`:

```
Unit BGIFont;
INTERFACE
  procedure BoldProc;
  procedure GothProc;
  procedure TripProc;
IMPLEMENTATION
  procedure BoldProc;external;
  {$L  Bold.obj}
  procedure GothProc;external;
  {$L  Goth.obj}
  procedure TripProc;external;
  {$L  Trip.obj}
END.
```

Підключивши бібліотеки та провівши реєстрацію, можна користуватися будь-яким драйвером та шрифтом. Нижче наведений фрагмент програми, яка ілюструє такі дії.

```
Program Pr20_17;
uses Graph,BGIDriv,BGIFont;
var GraphDriver,GraphMode,Error :integer;
procedure Abort(Message :string);
begin
  writeln(Message,' : ',GraphErrorMsg(GraphResult));
  Halt(1)
end; {Abort}
procedure Registration;
begin
  if RegisterBGIDriver(@EGAVGAProc) < 0 then Abort('EGAVGA');
```

```
  if RegisterBGIFont(@BoldProc)     < 0 then Abort('Bold');
  if RegisterBGIFont(@GothProc)     < 0 then Abort('Goth');
  if RegisterBGIFont(@TripProc)     < 0 then Abort('Trip');
end; {Registration}
BEGIN
  Registration;
  GraphDriver := Detect;
  InitGraph(GraphDriver,GraphMode,'');
  if GraphResult <> grOk then Halt(1)
  SetTextStyle(GothicFont, HorizDir, 2);
  OutTextXY(20, 40, 'Готичний шрифт');
  SetTextStyle(TriplexFont, HorizDir, 4);
  OutTextXY(80, 120, 'Векторний шрифт');
  Readln;
  CloseGraph;
END.
```

10. Контрольні запитання

1. Поясніть особливості текстового і графічного режимів роботи монітора.
2. У чому полягає ініціювання графічного режиму і як воно виконується?
3. Які помилки можуть виникати під час ініціювання графічного режиму та як їх уникнути?
4. Яким чином можна швидко переключатись між текстовим і графічним режимами роботи монітора?
5. Як використовуються відеосторінки у графічному режимі?
6. Якою є система координат у графічному режимі, як визначити поточну роздільну здатність відеорежиму?
7. Як створити і використовувати власну систему координат у графічному режимі?
8. Що таке поточний покажчик і як його можна переміщувати по екрану?
9. Як встановлюються вікна у графічному режимі та як можна визначити параметри поточних вікон?
10. Як у графічному режимі виводяться точки? Проілюструйте прикладами.
11. Як у графічному режимі виводяться лінії? Проілюструйте прикладами.
12. Як визначити стиль лінії та які параметри можна при цьому вказувати?
13. Як визначити власний шаблон для виведення ліній?
14. Як у графічному режимі вивести текстову інформацію?
15. Як у графічному режимі вивести числову інформацію?
16. Як визначити шрифт для виведення тексту? Які параметри можна при цьому вказувати?
17. Як змінити стандартні розміри та пропорції шрифтів?
18. Як дізнатися про ширину і висоту тексту?
19. Як встановлювати вирівнювання тексту? Які параметри можна при цьому вказувати?
20. Як вивести контурний прямокутник? Проілюструйте прикладами.
21. Як вивести замальований прямокутник? Проілюструйте прикладами.
22. Як вивести тривимірний прямокутник? Проілюструйте прикладами.
23. Як вивести контурний полігон? Проілюструйте прикладами.
24. Як вивести замальований полігон? Проілюструйте прикладами.
25. Як вивести коло або дугу кола? Проілюструйте прикладами.
26. Як вивести еліпс або дугу еліпса? Проілюструйте прикладами.
27. Як визначити параметри дуги кола або еліпса?
28. Як вивести замальований сектор кола або еліпса? Проілюструйте прикладами.
29. Як визначається основний і фоновий колір у графічному режимі? Які параметри при цьому можна встановлювати?
30. Як отримати інформацію про поточну палітру кольорів та як внести до неї зміни?
31. Як визначити поточний основний і фоновий колір, а також максимальну кількість кольорів?

32. Як визначити маску для заповненні фігур? Які параметри можна при цьому вказувати?
33. Як створити власну маску для заповнення фігур? Проілюструйте прикладами.
34. Як заповнювати область екрану поточною маскою? Проілюструйте прикладами.
35. Як організувати анімацію у графічному режимі через перемальовування окремих елементів зображення? Проілюструйте прикладами.
36. Які процедури і функції є у графічному режимі для роботи із областями екрану? Які режими існують для виведення зображень?
37. Як організувати анімацію у графічному режимі через запам'ятовування і виведення певних ділянок екрану? Проілюструйте прикладами.
38. Як організувати анімацію у графічному режимі через побудову математичної моделі? Проілюструйте прикладами.
39. Як включити до завершеної програми графічні драйвери?
40. Як включити до завершеної програми векторні шрифти?