ЖИТОМИРСЬКА ПОЛІТЕХНІКА
ДЕРЖАВНИЙ УНІВЕРСИТЕТ

А.В. Морозов, Г.В. Марчук, В.Л. Левківський, А.Ю. Левченко

C: теорія та практика

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
Державний університет «Житомирська політехніка»

C: теорія та практика

Навчальний посібник

Житомир 2025

УДК 004.4
С11

Затверджено Вченою радою
Державного університету «Житомирська політехніка»
протокол № 6 від 24 березня 2025 року

Рецензенти:

Молодецька Катерина Валеріївна – докторка технічних наук, професорка, професорка кафедри менеджменту організацій Києво-Могилянської бізнес-школи НаУКМА

Колос Катерина Ростиславівна – докторка педагогічних наук, професорка, ад'юнктка кафедри кібернетики, нанотехнологій і баз даних відділу автоматики, електроніки та інформатики Сілезької політехніки, Польща

Сугоняк Інна Іванівна – кандидатка технічних наук, доцентка, доцентка кафедри комп'ютерних наук Державного університету «Житомирська політехніка»

С11 С: теорія та практика : навчальний посібник / А.В. Морозов, Г.В. Марчук, В.Л. Левківський, А.Ю. Левченко – Житомир : Державний університет «Житомирська політехніка», 2025. – 268 с.

ISBN 978-966-683-697-0

У навчальному посібнику викладено основи алгоритмічної мови С. На простих прикладах показано можливості застосування мови до розв'язання практичних задач з програмування. Весь теоретичний виклад супроводжується прикладами. Наведено довідково-інформаційні дані для розв'язання задач (таблиці, схеми тощо).

Навчальний посібник призначено для здобувачів вищої освіти різних форм навчання галузі знань «Інформаційні технології»

УДК 004.4

ISBN 978-966-683-697-0

Зміст

ПЕРЕДМОВА **8**
1. МОВА ПРОГРАМУВАННЯ С **9**
Історична довідка 9
Стандарти 9
Структура програми 10
Контрольні запитання та завдання 12
2. ЛЕКСИЧНІ СТРУКТУРИ МОВИ С **14**
Алфавіт 14
Ідентифікатори 14
Коментарі 15
Ключові (зарезервовані) слова 15
Константи 16
Контрольні запитання та завдання 19
3. ТИПИ ДАНИХ, ВВЕДЕННЯ-ВИВЕДЕННЯ ІНФОРМАЦІЇ **21**
Базові типи даних 21
Перетворення типу 24
Операція sizeof() 25
Функції введення та виведення 26
Контрольні запитання та завдання 31
4. ОСНОВНІ ОПЕРАЦІЇ **34**
Арифметичні операції 34
Операції присвоювання 37
Операції порівняння 39
Логічні операції 40
Порозрядні операції (побітові операції) 41
Операція слідування (кома) 44
Контрольні запитання та завдання 45
5. ОСНОВИ АЛГОРИТМІЗАЦІЇ **47**
Алгоритми та їх властивості 47

Блок-схеми ... 48
Базові алгоритмічні конструкції ... 49
Контрольні запитання та завдання ... 51
6. УПРАВЛЯЮЧІ СТРУКТУРИ ... 53
Оператори розгалуження ... 53
Оператор розгалуження if ... 54
Оператор switch ... 57
Тернарний оператор ... 61
Оператори циклів ... 63
Цикл з передумовою while ... 63
Цикл з постумовою do..while ... 66
Цикл for ... 68
Оператор розриву break ... 72
Оператор продовження continue ... 73
«Порожній» оператор ... 75
Контрольні запитання та завдання ... 75
7. ПОКАЖЧИКИ І МАСИВИ ... 78
Покажчики ... 78
Основні відомості про покажчики ... 78
Основні операції над покажчиками ... 80
Багаторівнева непряма адресація ... 86
Операції над покажчиками ... 88
Проблеми, пов'язані з покажчиками ... 92
Масиви ... 95
Основні поняття ... 95
Оголошення та звертання в одновимірних масивах ... 100
Оголошення та звертання до багатовимірних масивів ... 103
Рядки ... 108
Основні відомості про представлення рядків ... 108
Функції роботи з рядками ... 109
Методи сортування масивів ... 116

Сортування методом обміну (бульбашкове сортування) *116*
Сортування методом вибору *117*
Сортування вставками *119*
Швидке сортування *120*
Контрольні запитання та завдання 122
8. ФУНКЦІЇ 125
Передача параметрів 129
Рекурсивні функції 134
Покажчики на функції 138
Класи пам'яті 141
Додаткові можливості функції main() 149
Контрольні запитання та завдання 151
9. СТРУКТУРИ 154
Оголошення структури 154
Масиви структур 158
Бітові поля 162
Об'єднання (union) 163
Тип перерахування enum 165
Контрольні запитання та завдання 168
10. ФАЙЛОВІ ПОТОКИ 170
Текстові файли 171
Бінарні файли 174
Контрольні запитання та завдання 179
11. ДИРЕКТИВИ ПРЕПРОЦЕСОРА 180
Директива #include 180
Директива #define 181
Директива #undef 187
Директиви #if, #elif, #else, #endif 188
Директиви #ifdef і #ifndef 190
Директива #line 192
Контрольні запитання та завдання 193

12. ДИНАМІЧНІ СТРУКТУРИ ДАНИХ 196
Лінійні списки 196
Стеки 210
Контрольні запитання та завдання 212
13. ДОВІДКОВО-ІНФОРМАЦІЙНІ ДАНІ 214
Таблиця символів ASCII 214
Розширені коди клавіатури 215
Функції стандартної бібліотеки 217
Функції I/O (stdio.h) 217
Математичні функції (math.h) 219
Функції для роботи з рядками (string.h) 221
Функції для роботи із символами 222
Функції для роботи з графічним режимом (graphics.h) 223
ТЕСТИ ДЛЯ САМОКОНТРОЛЮ 227
ВІДПОВІДІ НА ТЕСТИ ДЛЯ САМОКОНТРОЛЮ 254
ЛІТЕРАТУРА 266
ДЛЯ НОТАТОК 267

Передмова

Видання даного посібника було зумовлене необхідністю підготовки здобувачів вищої освіти, що навчаються за спеціальностями галузі знань «Інформаційні технології».

На даний час існує багато мов програмування і тому гостро стоїть питання про вибір мови для навчання. Оскільки С дає основу для інших мов таких як С++, С #, Java тощо, тому рекомендується вивчити С, перш ніж перейти до вивчення інших мов програмування. С – універсальна мова, ефективна як для вирішення завдань системного програмування, так і для створення прикладних програм.

Зручність мови С полягає в тому, що вона є мовою високого рівня з повним набором структурних конструкцій програмування, які підтримують модульність, блочну структуру програм і можливість індивідуальної компіляції модулів. Водночас мова С має набір низькорівневих засобів, які забезпечують зручний доступ до апаратного забезпечення комп'ютера і до кожного біта пам'яті.

Посібник містить стислий, та водночас досить повний виклад мови С у відповідності до її стандарту. На простих прикладах показано засоби застосування мови для розв’язання практичних задач. Усі теоретичні відомості супроводжуються простими та зрозумілими прикладами. Викладення матеріалу відбувається за зростанням від простого до більш ускладненого, що допоможе краще оволодіти можливостями даної мови програмування.

1. Мова програмування С

Історічна довідка

С – типізована мова програмування загального призначення, розроблена в 1969-1973 роках співробітником Bell Labs Деннісом Рітчі. Спочатку мова розроблялася для написання операційної системи Unix. Згодом вона була портована на багато інших операційних систем і стала одною з найбільш використовуваних мов програмування.

С цінують за її ефективність, вона є найпопулярнішою мовою для створення системного програмного забезпечення. ЇЇ також часто використовують для створення прикладних програм. Незважаючи на те, що С не розроблялася для новачків, вона активно використовується для навчання програмуванню. Надалі синтаксис мови С став основою для багатьох інших мов таких як С ++, С #, Java тощо.

Для мови С характерні лаконічність, сучасний набір конструкцій управління потоком виконання, структур даних і великий набір операцій.

Серед переваг мови С потрібно відзначити основні:

- компактність та універсальність коду;
- швидкість виконання програм;
- гнучкість мови;
- висока структурованість.
- проста мовна база;
- доступ до пам'яті через використання покажчиків.

Стандарти

Стандарти мови програмування С розробляються для забезпечення її стабільності, сумісності та розширення функціональності. На даний час доступно декілька стандартів мови С.

С89 (ANSI C) – перший офіційний стандарт, опублікований у 1989 році Американським національним інститутом стандартів (англ. American national standards institute, ANSI). Стандарт визначає базову функціональність мови С.

С90 – стандарт ANSI C з невеликими змінами був прийнятий Міжнародною організацією зі стандартизації (англ. International

Organization for Standardization, ISO) як ISO / IEC 9899: 1990. C89 і C90 вважаються ідентичними.

C99 – оновлений у 1999 році стандарт C90, який було опубліковано як ISO/IEC 9899:1999. Він включає нові можливості, такі як змінні, які можна оголошувати будь-де у коді, підтримка нових типів даних (наприклад, long long int), і розширена робота з масивами та математичними функціями.

C11 – затверджений у 2011 році як ISO/IEC 9899:2011. Основнми нововведеннями є підтримка багатопоточності, нові типи даних, розширені засоби роботи з пам'яттю та функціональність для роботи з Unicode.

C17 – у 2017 році стандарт C11 було удосконалено та створено стандарт C17 (ISO/IEC 9899:2017).

C18 – офіційно опублікований у 2018 році як ISO/IEC 9899:2018. Основна мета – усунення помилок і недоліків у попередньому стандарті C17 без додавання нових функцій. Стандарти C17 і C18 часто використовуються як синоніми.

Кожен новий стандарт базується на попередньому, зберігаючи зворотну сумісність, що дозволяє поступово впроваджувати нові можливості, зберігаючи стабільність коду.

Структура програми

Усі програми, написані на мові *C*, повинні містити в собі хоча б одну функцію. Функція *main()* – вхідна точка будь-якої програмної системи, причому немає різниці, де її розміщувати. Але потрібно пам'ятати наступне: якщо вона буде відсутня, завантажувач не зможе зібрати програму, про що буде виведене відповідне попередження(лістинг 1.1). Перший оператор програми повинен розміщуватися саме в цій функції.

Лістинг 1.1. Мінімальна програма на мові C

```
main() {        }
```

Деякі компілятори можуть не компілювати цей варіант. Тому цей самий код може бути представлений по іншому (лістинг 1.2).

Лістинг 1.2. Мінімальна програма на мові C

```
int main() {
```

```
        return 0;
}
```

Функція починається з імені. В даному прикладі вона не має параметрів, тому за її ім'ям розташовуються порожні круглі дужки *()*. Далі фігурні дужки {...} позначають блок або складений оператор.

Мінімальна програма має лише один оператор – оператор повернення значення *return* (лістинг 1.2). Виконання навіть цієї найпростішої програми, як і решти багатьох, проходить у декілька етапів (рис 1.1).

Рис. 1.1. Етапи виконання програми на мові С

Оператор *return* завершує виконання програми та повертає деяке ціле значення (ненульове значення свідчить про помилки в програмі, нульове про успішне її завершення). Результат виконання програми:

```
Process exited with return value 0
Press any key to continue . .
```

З мовою С можна працювати в різних середовищах та операційних системах і програмний код може бути більш складним (лістинг 1.3).

Лістинг 1.3. Приклад коду на мові С

```
#include<stdio.h>

int sum (intn, inta[] ){
   int i, s=0;
   for( i=0; i<n; i++ )
      s+=a[i];
return s;
}
void main(){
int a[]={ 3, 5, 7, 9, 11, 13, 15 };
int s = sum( 7, a );
   printf("sum=%i\n",s);
}
```

Результат виконання програми:

```
sum=63
```

У перших двох рядках коду наведено спеціальні інструкції для компілятора, які називаються директивами препроцесора. Препроцесор підготовлює код перед його компіляцією. Директиви починаються зі символу "#" і виконуються ще до того, як компілятор розпочне перетворення коду на машинний. Директива *#include* використовується для підключення зовнішніх файлів (бібліотек). В розділі 11 це буде розглянуто більш детально.

В наведеному прикладі (лістинг 1.3) *#include <stdio.h>* – директива підключає стандартну бібліотеку введення-виведення, яка необхідна для використання функції printf, яка виводить дані на екран.

Рядок коду *int sum (int n, int a[])* – це оголошення функції sum, яка приймає два аргументи. Функція рахує суму елементів масиву та повертає її у функцію main().

void main() – оголошення головної функції програми. Саме з цієї функції починається виконання програми. За стандартом C99, *main()* повинна повертати int, хоча деякі компілятори допускають void.

У функції *main()* проведено оголошення і ініціалізація масивуи *a[]*. Компілятор автоматично визначає розмір масиву за кількістю заданих елементів. Також оголошено змінну *s* в яку буде повернено з функції *sum()* значення суми елементів масиву. Для виведення значення змінної *s* на екран використано оператор printf().

Більш детальніше складові програмного коду будуть розглянуті в наступних розділах посібника.

Контрольні запитання та завдання

1. Яка була основна мета створення мови С?
2. Кого вважають засновником мови С та де вона була створена?
3. Що таке стандарт мови програмування і чому він важливий?
4. Які існують основні версії стандарту мови С?
5. Які основні відмінності між стандартами мови С?
6. Які основні переваги використання мови С?
7. Який внесок мови С зробила в розвиток інших мов програмування?
8. Які основні компоненти має програма на мові С?

9. Назвіть основні етапи виконання програми на мові С?
10. Що таке препроцесор в мові С і які його основні директиви?
11. Що таке функція main і чому вона є обов'язковою в кожній програмі на С?
12. Яку роль відіграють фігурні дужки {} в структурі програми?
13. Що таке ключове слово return і для чого воно використовується?
14. Що означає ненульове значення, повернене оператором return у результаті виконання програми?
15. Чому мова С залишається популярною протягом багатьох десятиліть?
16. В яких середовищах і операційних системах можна працювати з мовою С?
17. Як можна перевірити, який стандарт С використовується в даному проекті?
18. Чи можуть бути проблеми при переході на новий стандарт С?
19. Які переваги використання сучасних стандартів мови С?
20. Як стандарти С впливають на написання портативного коду?
21. Який стандарт С є найбільш актуальним на сьогоднішній день?
22. Які основні відмінності між стандартами С89 та С99?
23. Які основні відмінності між стандартами С11 та С17?
24. Як забезпечити сумісність коду з різними стандартами С?
25. Для чого використовується макрос __STDC_VERSION__?

2. Лексичні структури мови С

Будь-яка мова (українська, англійська, французька тощо) складається з декількох основних елементів – символів, слів, словосполучень і речень. В алгоритмічних мовах програмування існують аналогічні структурні елементи, тільки слова називають лексемами, словосполучення – виразами, а речення – операторами.

Лексеми в свою чергу утворюються із символів, вирази – із лексем і символів, оператори – із символів, лексем і виразів.

- *Алфавіт мови*, або її символи – це основні неподільні знаки, за допомогою яких пишуться всі тексти на мові програмування.
- *Лексема*, або елементарна конструкція – мінімальна одиниця мови, яка має самостійний зміст.
- *Вираз* задає правило обчислення деякого значення.
- *Оператор* задає кінцевий опис деякої дії.

Алфавіт

Алфавіт мови С включає :

- великі та малі літери латинської абетки: A…Z, a…z;
- арабські цифри: 0…9;
- пробільні символи : пробіл, символи табуляції, символ переходу на наступний рядок тощо;
- символи , . ; : ? ‘ ! “ | / \ ~ ( ) [ ] { } < > # % ^ & - + * =.

Ідентифікатори

Ідентифікатори використовуються для іменування різних об’єктів : змінних, констант, міток, функцій тощо. При записі *ідентифікаторів* можуть використовуватися великі та малі літери латинської абетки, арабські цифри та символ підкреслення. Ідентифікатор не може починатися з цифри і не може містити пробілів.

Формальне визначення ідентифікатора можна записати наступним чином:

[<буква > | <підкреслення>]{< буква > | <підкреслення> | <цифра >}

Компілятор мови С розглядає літери верхнього та нижнього регістрів як різні символи. Тому можна створювати ідентифікатори, які співпадають орфографічно, але відрізняються регістром літер.

Приклад.

Кожний з наступних ідентифікаторів унікальний:

```
Sum   sum   sUm   SUM   sUM
```

Правильно записані ідентифікатори:

```
F, f, max, Min, MaX, _min, sum1, min_znach
```

Неправильно записані ідентифікатори:

```
1sum, max-znach, min znach, a..b
```

Слід також пам'ятати, що ідентифікатори не повинні співпадати з ключовими словами.

Коментарі

Текст на С, що міститься у дужках /* та */ ігноруватиметься компілятором, тобто вважатиметься коментарем до програми. Такі коментарі можуть розміщуватися в будь-якому місці програми.

Коментарі здебільшого використовуються для документування коду програми та під час їх відлагодження. В програму бажано вміщувати текст, що хоч якось пояснює її роботу та призначення. Проте не слід надто зловживати коментарями, а використовувати більш розумні форми найменування змінних, констант, функцій тощо. Якщо, наприклад, функція матиме назву *add_matrix*, очевидно не зовсім раціональним буде включення у програму після її заголовної частини коментар про те, що:

```
/*функція обчислює суму матриць */
```

У цьому випадку ім'я функції пояснює її призначення. У більш сучасних версіях С широко застосовується так званий угорський запис імен, коли ім'я змінної містить в собі інформацію про її призначення і тип.

Ключові (зарезервовані) слова

Ключові слова – це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора. Їх використання суворо регламентоване. Імена змінних, констант, міток, типів тощо не можуть співпадати з ключовими словами.

Наводимо перелік ключових слів мови С :

auto	***continue***	***float***	***interrupt***	***short***	***unsigned***
asm	***default***	***for***	***long***	***signed***	***void***
break	***do***	***far***	***near***	***sizeof***	***volatile***
case	***double***	***goto***	***pascal***	***static***	***while***
cdecl	***else***	***huge***	***switch***	***struct***	
char	***enum***	***if***	***register***	***typedef***	
const	***extern***	***int***	***return***	***union***	

Константи

Константами називають сталі величини, тобто такі, які в процесі виконання програми не змінюються. В мові С існує чотири типи констант : цілі, дійсні, рядкові та символьні.

1. Константи цілого типу можуть задаватися у десятковій, вісімковій або шістнадцятковій системі числення.

Десяткова константа – послідовність десяткових цифр (від 0 до 9), яка починається не з нуля якщо це число не нуль. Приклади десяткових констант : 10, 132, 1024.

Вісімкові константи починаються з символу 0, після якого розміщуються вісімкові цифри (від 0 до 7). Наприклад : 023. Запис константи вигляду 08 буде сприйматися компілятором як помилка, так як 8 не є вісімковою цифрою.

Шістнадцяткові константи починаються з символів 0x або 0X, після яких розміщуються шістнадцяткові цифри (від 0 до 9 і від A до F, можна записувати їх у верхньому чи нижньому регістрах). Наприклад : 0XF123. У лістингу 2.1 продемонстровано використання констант цілого типу.

Лістинг 2.1. Десяткова, вісімкова, шістнадцяткова константи

```
#include <stdio.h>
main() {
    const int a = 10, b = 010, c = 0x10;
    printf("\na=%d b=%d c=%d", a, b, c);
}
```

Результат виконання програми:

```
a=10 b=8 c=16
```

2. Константи дійсного типу складаються з цілої частини, десяткової крапки, дробової частини, символу експоненти (e чи E) та показника степеня. Дійсні константи мають наступний формат представлення :

[ціла_частина][. дробова_частина][E [–] степінь]

У записі константи можуть бути опущені ціла чи дробова частини (але не обидві разом), десяткова крапка з дробовою частиною чи символ E (e) з показником степеня (але не разом). Приклади констант дійсного типу:

2.2 , 220e–2, 22.E–1, .22E1

Якщо потрібно сформувати від'ємну цілу або дійсну константу, то перед константою необхідно поставити знак унарного мінуса. У лістингу 2.2 продемонстровано використання констант дійсного типу.

Лістинг 2.2. Константи дійсного типу

```
#include <stdio.h>
main() {
    const float a = 10.25, b = 247, c = 0.1258;
    printf("\na=%e b=%f c=%f", a, b, c);
}
```

Результат виконання програми:

```
a=1.025000e+01 b=247.000000 c=0.125800
```

3. Символьні константи. Символьна константа – це значення символу, який написано в поодиноких лапках. Символьні константи можуть мати тип char, short або int, що відповідає коду символа у ASCII таблиці.

Деякі неграфічні символи, такі як одинарні лапки, зворотна коса риска (зворотний слеш) можуть бути представлені двома символами. Послідовності символів, які починаються з символу \ називаються керуючими або escape-послідовностями (таблиця 1.1).

Таблиця 1.1.

Escape-послідовності

Спеціальний символ	Шістнадцят-ковий код	Значення
\a	07	звуковий сигнал
\b	08	повернення на 1 символ
\f	0C	переведення сторінки
\n	0A	перехід на наступний рядок
\r	0D	повернення каретки
\t	09	горизонтальна табуляція
\v	0B	вертикальна табуляція
\\	5C	символ \
\’	27	символ ‘
\”	22	символ “
\?	3F	символ ?
\0	00	нульовий символ
\0ddd	–	вісімковий код символу
\0xddd	ddd	десятковий код символу

В лістингу 2.2 наведено приклад використання символьних констант.

Лістинг 2.2. Використання символьних констант

```
#include <stdio.h>
main() {
const char a ='g', b ='*', c ='\\', d = '3', e = 36, f =97;
printf("\na=%c b=%c c=%c d=%c e=%c f=%c",a,b,c,d,e,f);
}
```

Результат виконання програми:

```
a=g b=* c=\ d=3 e=$ f=a
```

4. Рядкові константи записуються як послідовності символів, заключених в подвійні лапки.

“Це рядковий літерал!\n”

Для формування рядкових констант, які займають декілька рядків тексту програми використовується символ \ (зворотний слеш):

```
"Довгі рядки можна розбивати на \
частини"
```

Загальна форма визначення іменованої константи має вигляд :

const тип ім'я = значення;

Модифікатор *const* попереджує будь-які присвоювання даному об'єкту, а також інші дії, що можуть вплинути на зміну значення.

Приклад оголошення констант.

```
const float  pi = 3.14l5926;
const maxint = 32767;
char *const str="Hello,P...!"; /* покажчик-константа */
char const *str2= "Hello!"; /*покажчик на константу */
```

Використання одного лише модифікатору *const* еквівалентно *const int*.

Контрольні запитання та завдання

1. Що таке лексична структура мови програмування? Яку роль вона відіграє у процесі компіляції програми?
2. Які є основні лексичні елементи мови С? Наведіть приклади.
3. Які символи входять до алфавіту мови С?
4. Чи є різниця між великими та малими літерами в мові С?
5. Що таке ідентифікатор в мові С?
6. Яка максимальна довжина ідентифікатора в мові С?
7. Які правила створення ідентифікаторів в мові С?
8. Що називають коментарем в мові програмування С?
9. Для чого використовують коментарі в програмуванні?
10. Які символи використовуються для позначення коментарів в мові С?
11. Що таке ключове слово у мові програмування? Наведіть приклади ключових слів мови С.

12. Чи можна ключові слова використовувати як імена змінних? Чому?
13. Яке слово в мові С називається ключовим (зарезервованим)?
14. Що називають константою у мові програмування С?
15. Які розрізняють типи констант?
16. Як відбувається оголошення констант?
17. Поясніть рядок програмного коду: const float GRAVITY = 9.81;
18. Поясніть рядок програмного коду: const int MAX = 30;
19. Поясніть рядок програмного коду: const char FIRST_LETTER = 'A';
20. Як оголосити константу з плаваючою крапкою?
    - `const float PI = 3.14159;`
    - `const double E = 2.71828;`
    - `const long double G = 6.674e-11;`
21. Як оголосити символьну константу?
    - `const char LETTER = 'A';`
    - `const char ESCAPE = '\n';`
    - `const char NULL_CHAR = '\0';`
22. Для чого використовуються Escape-послідовності?
23. Як здійснити перехід на наступний рядок за допомогою escape-послідовності?
24. Як виконати табуляцію за допомогою escape-послідовності?
25. Яке призначення escape-послідовності \a?

3. Типи даних, введення-виведення інформації

Базові типи даних

Будь-яка програма передбачає виконання певних операцій з даними. Тип даних визначає, як саме здійснюватимуться операції та як буде реалізовано алгоритм.

Що таке тип даних? Сформулювати це поняття можна наступним чином: *множина значень плюс перелік дій або операцій, які можна виконати над кожною змінною даного типу.* Вважається, що змінна або вираз належить до конкретного типу, якщо його значення лежить в області допустимих значень цього типу.

Арифметичні типи даних об'єднують цілі та дійсні. Цілі у свою чергу містять декілька різновидів цілих та символьних типів даних. Скалярні типи включають в себе арифметичні типи, покажчики та перелічувані типи. Агрегатні або структуровані типи містять в собі масиви, структури та файли.

Базові типи даних С можна перерахувати у наступній послідовності:

1. ***char – символ***

Тип може використовуватися для зберігання літери, цифри або іншого символу з множини символів ASCII. Значенням об'єкта типу *char* є код символу. Тип *char* інтерпретується як однобайтове ціле з областю значень від –128 до 127.

2. ***int – ціле***

В операційних середовищах Windows використовуються 32- або 64-розрядні цілі, що дозволяє розширити діапазон їх значень. Як різновиди цілих чисел, у деяких версіях компіляторів існують *short* - коротке ціле (2 байти) та *long long int* (8 байти) – довге довге ціле. Хоча синтаксис мови не залежить від операційних систем, розмірність цих типів може коливатися від конкретної реалізації. Гарантовано лише, що співвідношення розмірності є наступним:

short ≤ int ≤ long ≤ long long int

3. ***float – число з плаваючою комою одинарної точності***

Тип призначений для зберігання дійсних чисел. Може представляти числа як у фіксованому форматі (наприклад число пі - 3.14159), так і в експоненціальній формі – 3.4E+8.

4. double - число з плаваючою комою подвійної точності

Має значно більший діапазон значень, порівняно з типом *float*: $\pm(1.7\cdot10^{-308} \ldots 1.7\cdot10^{308})$.

У мові С використовується префіксний запис оголошення, на початку вказується тип змінної, а потім її ім'я. Змінні повинні бути описаними до того моменту, як вони будуть використовуватися у програмі. Ніяких додаткових ключових слів при цьому не пишуть.

Приклад.

```
int name;
float var, var1;
double temp;
char ch;
long height;
```

Змінні можна ініціалізувати (присвоювати їм початкові значення) безпосередньо у місці їх опису.

Приклад.

```
int height = 33 ;
float income = 2834.12 ;
char val = 12 ;
```

Для виведення інформації на екран використаємо функцію printf() (детально про операції введення-виведення значень змінних йтиметься у розділі "Операція sizeof()".

Крім того, цілі типи char, short, int, long можуть використовуватися з модифікаторами signed (із знаком) та unsigned (без знаку). Цілі без знаку (unsigned) не можуть набувати від'ємних значень, на відміну від знакових цілих (signed). За рахунок цього дещо розширюється діапазон можливих додатних значень типу (таблиця 3.1.).

Таблиця 3.1.

Діапазони значень простих типів даних

Тип	Діапазон значень	Розмір (байт)
char	-128 … 127	1
short	-32768 ... 32767	2
int		2 або 4
long	-2,147,483,648 ... 2,147,483,647	4
unsigned char	0 ... 255	1
unsigned short	0 … 65535	2
unsigned		2 або 4
unsigned long	0 ... 4,294,967,295	4
float	$\pm(3.4\cdot10^{-38} \ldots 3.4\cdot10^{38})$	4
double	$\pm(1.7\cdot10^{-308} \ldots 1.7\cdot10^{308})$	8
long double	$\pm(3.4\cdot10^{-4932} \ldots 3.4\cdot10^{4932})$	10

В лістингу 3.1 наведено приклад використання різних типів даних в програмному коді.

Лістинг 3.1. Використання різних типів даних

```
#include <stdio.h>
main() {
     int a = 9;
     int b = -9;
     int c = 89U;
     long int d = 99998L;
     float e = 17.25;
     double f = 258.235064;
     printf("a = %d\n", a);
     printf("b = %d\n", b);
     printf("c = %u\n", c);
     printf("d = %ld\n", d);
     printf("e = %f\n", e);
     printf("f = %f\n", f);
 }
```

Результат виконання програми:

```
a = 9
b = -9
c = 89
d = 99998
e = 17.250000
f = 258.235064
```

Перетворення типу

Компілятор С виконує автоматичне перетворення типів даних, особливо в математичних виразах, коли найчастіше цілочисельний тип перетворюється у тип з плаваючою комою, причому значення типу *char* та *int* в арифметичних виразах змішуються: кожний з таких символів автоматично перетворюється в ціле. Взагалі, якщо операнди мають різні типи, перед тим, як виконати операцію, молодший тип "підтягується" до старшого. Отже,

– *char* та *short* перетворюються в *int*;

– *float* перетворюється в *double*;

– якщо один з операндів *long double*, то і другий перетворюється в *long double*;

– якщо один з операндів *long*, тоді другий перетворюється відповідно до того ж типу, і результат буде *long*;

– якщо один з операндів *unsigned*, тоді другий перетворюється відповідно до того ж типу, і результат буде *unsigned*.

Приклад.

```
double ft, sd;
unsigned char ch;
unsigned long in;
int i;
/* ... */
sd = ft*(i+ch/in);
```

При виконанні оператора присвоювання в даному прикладі правила перетворення типів будуть використані наступним чином. Операнд *ch* перетворюється до *unsigned int*. Після цього він

перетворюється до типу *unsigned long*. За цим же принципом *i* перетворюється до *unsigned long* і результат операції, що розміщена в круглих дужках буде мати тип *unsigned long*. Потім він перетворюється до типу *double* і результат всього виразу буде мати тип *double*.

Взагалі, тип результату кожної арифметичної операції виразу є тип того операнду, який має у відповідності більш високий тип приведення.

Але, окрім цього в С, з'являється можливість і примусового перетворення типу, щоб дозволити явно конвертувати (перетворювати) значення одного типу даних в інший. Загальний синтаксис перетворення типу має два варіанти :

1. (новий_тип) вираз ;
2. новий_тип (вираз) .

Приклад.

```
char letter = 'a';
int nasc = int (letter);
long iasc = (long) letter;
```

Операція sizeof()

Дана операція обчислює розмір пам'яті, необхідний для розміщення в ній виразів або змінних вказаних типів.

Операція має дві форми:

1) ім'я_типу A;
sizeof A;
2) sizeof (ім'я_типу);

Операцію *sizeof*() можна застосовувати до констант, типів або змінних, у результаті чого буде отримано число байт, що відводяться під операнд. Приміром, *sizeof*(*int*) поверне число байт для розміщення змінної типу *int* (лістинг 3.2).

Лістинг 3.2. Використання операції *sizeof()*

```
#include <stdio.h>
int main(){
    printf("Розмір типу char: %d байт\n", sizeof(char));
    printf("Розмір типу short: %d байт\n", sizeof(short));
```

```
    printf("Розмір типу int: %d байт\n", sizeof(int));
    printf("Розмір типу long: %d байт\n", sizeof(long));
    printf("Розмір типу long long: %d байт\n", sizeof(long
long));
    printf("Розмір типу float: %d байт\n", sizeof(float));
    printf("Розмір типу double: %d байт\n",
sizeof(double));
    printf("Розмір типу long double: %d байт\n",
sizeof(long double));
return 0;
}
```

Результат виконання програми:

```
Розмір типу char: 1 байт
Розмір типу short: 2 байт
Розмір типу int: 4 байт
Розмір типу long: 4 байт
Розмір типу long long: 8 байт
Розмір типу float: 4 байт
Розмір типу double: 8 байт
Розмір типу long double: 8 байт
```

Функції введення та виведення

В мові С на стандартні потоки введення-виведення завжди вказують імена *stdin* та *stdout*. Обробку цих потоків здійснюють функції, визначені в заголовочному файлі *stdio.h*.

До функцій введення-виведення можна віднести *getchar()*, *putchar()*.

Функція *getchar()* зчитує і повертає черговий символ з послідовності символів вхідного потоку. Якщо цю послідовність вичерпано, то функція *getchar()* повертає значення –1 (цьому значенню відповідає константа *EOF*).

Функція *putchar(аргумент)*, де аргументом є вираз цілого типу, виводить у стандартний вихідний потік значення аргументу, перетворене до типу *char*.

В лістингу 3.3 наведено приклад використання функцій введення-виведення в програмному коді.

Лістинг 3.3. Використання функцій введення-виведення

```
#include <stdio.h>
int main(){
       char ch;
       ch = getchar();
       putchar(ch);
return 0;
}
```

Результат виконання програми:

```
f
f
```

Для введення та виведення більш складної інформації використовуються функції *scanf()* та *printf()*.

Функція *printf()* призначена для виведення інформації за заданим форматом. Синтаксис функції *printf()*:

printf("Рядок формату"[, аргумент1[, аргумент2, [...]]]);

Першим параметром даної функції є «рядок формату», який задає форму виведення інформації. Далі можуть розташовуватися арифметичні вирази або рядки. Аргументи в списку відокремлюються комами. Функція *printf()* перетворює значення аргументів до вигляду, поданого у рядку формату і виводить одержану послідовність символів у стандартний потік виведення.

Рядок формату складається з об'єктів двох типів: звичайних символів, які з рядка копіюються в потік виведення, та специфікацій перетворення. ***Кількість специфікацій у рядку формату повинна дорівнювати кількості аргументів.***

В лістингу 3.4 наведено приклад використання функції *printf()* в програмному коді.

Лістинг 3.4. Використання функції *printf()*

```
#include <stdio.h>
int main(){
   int a = 10, b = 20, c = 30;
```

```
    printf(" a == %d \n b == %d \n c == %d \n", a, b, c);
return 0;
}
```

Результат виконання програми:

```
a == 10
b == 20
c == 30
```

Специфікації перетворення для функції *printf()*:

%d – десяткове ціле;
%i – десяткове ціле;
%o – вісімкове ціле без знаку;
%u – десяткове ціле без знаку (unsigned);
%x – шістнадцяткове ціле без знаку;
%f – представлення величин float та double з фіксованою точкою;
%e або %E – експоненціальний формат представлення дійсних величин;
%g – представлення дійсних величин як f або E в залежності від значень;
%c – один символ (char);
%s – рядок символів;
%p – покажчик;
%n – покажчик
%ld – long (в десятковому вигляді);
%lo – long (у вісімковому вигляді);
%p – виведення покажчика в шістнадцятковій формі;
%lu – unsigned long.

Можна дещо розширити основне визначення специфікації перетворення, помістивши модифікатори між знаком % і символами, які визначають тип перетворення (таблиця 3.2.).

Таблиця 3.2.

Значення основних модифікаторів форматованого виведення

Модифікатор	Значення
–	Аргумент буде друкуватися починаючи з лівої позиції поля заданої ширини. Звичайно друк аргументу закінчується в самій правій позиції поля. Приклад : %-10d

Рядок цифр	Задає мінімальну ширину поля. Поле буде автоматично збільшуватися, якщо число або рядок не буде вміщуватися у полі. Приклад : %4d
Цифри.цифри	Визначає точність: для типів даних з плаваючою комою - число символів, що друкуються зліва від десяткової коми; для символьних рядків – максимальну кількість символів, що можуть бути надруковані. Приклад : %4.2f

В лістингу 3.5 наведено приклад використання форматованого виведення для даних цілого типу.

Лістинг 3.5. Використання форматованого виведення для даних цілого типу

```
#include <stdio.h>
int main(){
       printf("/%d/ \n", 336);
       printf("/%2d/ \n", 336);
       printf("/%10d/ \n", 336);
       printf("/%-10d/ \n", 336);
return 0;
}
```

Результат виконання програми:

```
/336/
/336/
/     336/
/336      /
```

В лістингу 3.6 наведено приклад використання форматованого виведення для даних дійсного типу.

Лістинг 3.6. Використання форматованого виведення для даних дійсного типу

```
#include <stdio.h>
int main(){
       printf(" / %f / \n", 1234.56);
       printf(" / %e / \n", 1234.56);
       printf(" / %4.2f / \n", 1234.56);
```

```
        printf(" / %3.1f / \n", 1234.56);
        printf(" / %10.3f / \n", 1234.56);
        printf(" / %10.3e / \n", 1234.56);
return 0;
    }
```

Результат виконання програми:

```
/ 1234.560000 /
 / 1.234560e+03 /
 / 1234.56 /
 / 1234.6 /
 /   1234.560 /
 /  1.235e+03 /
```

Для введення інформації зі стандартного потоку введення використовується функція *scanf()*.

Синтаксис :

scanf(“Рядок формату”,&аргумент1[,&аргрумент2[, ...]]);

Так, як і для функції *printf()*, для функції *scanf()* вказується рядок формату і список аргументів. Суттєва відмінність у синтаксисі цих двох функцій полягає в особливостях даного списку аргументів. Функція *printf()* використовує імена змінних, констант та вирази, в той час, як для функції *scanf* () вказується тільки покажчики на змінні.

Поширеною помилкою використання *scanf()* у початківців є звертання: *scanf(“%d”, n)* замість *scanf(“%d”, &n)*. ***Параметри цієї функції обов’язково повинні мати &!***

Функція *scanf()* використовує практично той же набір символів специфікації, що і функція *printf()*.

В лістингу 3.7 наведено приклад використання функції введення даних.

Лістинг 3.7. Використання функції введення даних

```
#include <stdio.h>
int main(){
        int a, b, c;
        printf("A = ");
```

```
        scanf("%d", &a);
        printf("B = ");
        scanf("%d", &b);
        c = a + b;
        printf("A + B = %d", c);
return 0;
}
```

Результат виконання програми:

```
A = 7
B = 2
A + B = 9
```

Контрольні запитання та завдання

1. Яка різниця між змінною та константою у мові С?
2. Який з наведених типів даних не є стандартним типом мови програмування С: char, double, int, bool, float.
3. Який тип даних використовується для оголошення символьних змінних?
4. Для чого використовуються модифікатори типів даних?
5. Який модифікатор використовується для оголошення беззнакових змінних?
6. Яка різниця між типами даних float та double?
7. Який розмір кожного з базових типів даних у байтах?
8. Які є типи даних для зберігання цілих чисел у мові С?
9. Який тип даних використовується для зберігання чисел з плаваючою комою?
10. Який тип даних використовується для зберігання окремих символів?
11. Як оголосити змінну певного типу даних у мові С?
12. Як ініціалізувати змінну під час оголошення?
13. Як присвоїти значення змінній?
14. Який вираз відповідає оголошенню змінній цілого типу з ініціалізацією значення?

int x;

int x = 1;

float y;

int v=(float) y;

char ch='A';

15. Тип даних int дозволяє зберігати у пам'яті значення:

дійсного числа

цілого числа

рядкового числа

символьного числа

додатнього цілого числа

16. Який тип даних з нижче перерахованих дозволяє зберігати невід'ємні цілі числа?

long

unsigned long

signed long

long long

int

17. Визначити чому буде дорівнювати b

```
float a = 241.5;
int b = (int)a % 2;
```

18. Який синтаксис операції sizeof()?
19. Як можна отримати розмір типу даних?
20. Порівняйте розміри типів даних мови С.
21. Що виведе на екран даний програмний код:

```
int x = 5;
printf("int = %d\n", sizeof(int));
```

22. Опишіть структуру функцій printf() та scanf()?
23. Завдання на явне перетворення типів. Напишіть програму, яка оголошує змінну типу double та присвоює їй значення 25.407. Використайте явне перетворення типу для перетворення значення на int. Виведіть на екран початкове значення типу double та перетворене значення типу int. Поясніть, що відбувається з десятковою частиною числа під час перетворення.

24. Завдання на неявне перетворення типів. Напишіть програму, яка оголошує змінні різних типів (наприклад, int, float, char). Виконайте арифметичні операції з цими змінними (наприклад, додавання, множення). Поясніть, які неявні перетворення типів відбуваються під час виконання операцій.
25. Напишіть програму, яка запитує у користувача два цілих числа та обчислює їх середнє арифметичне значення. Використайте неявне перетворення типів для отримання результату з плаваючою комою.
26. Напишіть програму для підрахунку виразу за формулою:

    $((c/a-d))/(a/4+d)-(5d-b)/(c-a)^2$

    Всі змінні приймають дійсне значення.
27. Дано рядок, що містить число. Напишіть програму, яка перетворює цей рядок у ціле або дробове число, залежно від того, чи є в ньому десяткова крапка.
28. Напишіть програму для підрахунку виразу за формулою:

    ((c/b-d/a))/(dc^2)-(d+5)/c

    Всі змінні приймають дійсне значення.
29. Дано значення кута в градусах ($0<\alpha<360$). Визначити значення кута у радіанах, якщо 180 градусів дорівнює π радіанів.
30. Зробіть форматоване виведення на консоль фрагмента вірша:

    Сонце низенько, вечір близенько,
    Іди до мене, моє серденько!
    Ой вийди, вийди, та не барися,
    Моє серденько, розвеселися.

    Іван Котляревський

4. Основні операції

В програмуванні операції – це дії, які виконуються над даними для досягнення певного результату. Вони є ключовими елементами будь-якого алгоритму. Більшість операцій мають два операнди, один з яких розташовується перед знаком операції, а інший – після. Наприклад, два операнди має операція додавання A+B.

Операції, які мають два операнди називаються *бінарними*. Існують і *унарні* операції, тобто такі, які мають лише один операнд. Наприклад, запис –A означає застосування до операнду A операції унарного мінуса. А три операнди має лише одна операція – ?:. Це єдина *тернарна* операція мови С.

У складних виразах послідовність виконання операцій визначається дужками, старшинством операцій, а при однаковому старшинстві – асоціативністю.

Операції поділяються на кілька основних типів залежно від їхнього призначення:

- арифметичні операції;
- операції присвоювання;
- операції відношення;
- логічні операції;
- порозрядні операції;
- операція обчислення розміру *sizeof()*;
- операція слідування (кома).

Арифметичні операції

До арифметичних операцій належать відомі всім бінарні операції додавання, віднімання, множення, ділення та знаходження залишку від ділення (таблиця 4.1).

Таблиця 4.1.

Бінарні арифметичні операції

Операція	Значення	Приклад
+	Додавання	a+b
–	Віднімання	a–b
*	Множення	a*b
/	Ділення	a/b
%	Залишок від ділення	a%6

Для наведених арифметичних операцій діють наступні правила:

- бінарні операції додавання (+) та віднімання (–) можуть застосовуватися до цілих та дійних чисел, а також до покажчиків;
- в операціях множення (*) та ділення (/) операнди можуть бути будь-яких арифметичних типів;
- операція «залишок від ділення» застосовується лише до цілих операндів.

Операції виконуються зліва направо, тобто спочатку обчислюється вираз лівого операнда, потім вираз, що стоїть справа від знака операції. Якщо операнди мають однаковий тип, то результат арифметичної операції має той же тип. Тому, коли операції ділення / застосовується до цілих або символьних змінних, залишок відкидається. Так, вираз 11/3 буде рівний 3, а вираз 1/2 буде рівним нулю.

В мові С визначені також і унарні арифметичні операції (таблиця 4.2).

Таблиця 4.2.

Унарні арифметичні операції

Операція	*Значення*	*Приклад*
+	Унарний плюс (підтвердження знака)	+5
–	Унарний мінус (зміна знака)	–x
++	Операція інкременту (збільшення на 1)	i++, ++i
—	Операція декременту (зменшення на 1)	j—, —j

Операція інкременту (++) збільшує операнд на одиницю, а операція декременту (--) відповідно зменшує операнд на одиницю. Ці операції виконуються швидше, ніж звичайні операції додавання одиниці (a=a+1;) чи віднімання одиниці (a=a–1;).

Існує дві форми запису операцій інкременту та декременту. Якщо операція інкременту (декременту) розміщена перед змінною, то це префіксна форма запису інкременту (декременту). Якщо операція інкременту (декременту) записана після змінної, то це постфіксна форма запису. У префіксній формі змінна спочатку збільшується (зменшується) на одиницю, а потім її нове значення використовується у виразі. При постфіксній формі у виразі спочатку використовується поточне значення змінної, а потім відбувається збільшення (зменшення) цієї змінної на одиницю.

У мові С операції мають різні рівні пріоритету. Пріоритет визначає порядок виконання операцій у виразах, а асоціативність визначає, у якому напрямку виконується операція, якщо два або більше операторів мають однаковий пріоритет (таблиця 4.3).

Таблиця 4.3.

Пріоритети операцій

Операції (від вищого пріоритету до нижчого)	Асоціативність
( ) [ ] –> . ::	Зліва направо
! ~ ++ –– & * (тип) sizeof new delete	Справа наліво
* / %	Зліва направо
+ –	Зліва направо
<< >>	Зліва направо
< <= > >=	Зліва направо
== !=	Зліва направо
&	Зліва направо
^	Зліва направо
|	Зліва направо
&&	Зліва направо
||	Зліва направо
?:	Справа наліво
= += –= *= /= %= <<= >>= ^= |= &=	Справа наліво
,	Зліва направо

Приклад, який демонструє роботу операції інкременту наведено в лістингу 4.1.

Лістинг 4.1. Використання операції інкременту

```
#include <stdio.h>
void main()
{
    int x = 3, y = 3;
    printf("Значення префіксного виразу : % d\n", ++x);
    printf("Значення постфіксного виразу : % d\n", y++);
    printf("Значення x після інкременту : % d\n", x);
    printf("Значення y після інкременту : % d\n", y);
}
```

Результат виконання програми:

```
Значення префіксного виразу :  4
Значення постфіксного виразу :  3
Значення x після інкременту :  4
Значення y після інкременту :  4
```

Приклад, який демонструє роботу операції інкременту та декременту наведено в лістингу 4.2.

Лістинг 4.2. Використання операції інкременту та декременту

```
#include<stdio.h>
void main()
{
   int m = 7, n = 4, yz;
   printf ("n =%d, m =%d \n", n, m);
   yz = ++ n + m;
   printf ("++ n * ++ m =%d \n", yz);
   printf ("n =%d, m =%d \n", n, m);
   yz = m-- <n;
   printf ("m -- < n =%d \n", yz);
   printf ("n =% d, m =%d \n", n, m);
}
```

Результат виконання програми:

```
n =4, m =7
++ n * ++ m =12
n =5, m =7
m -- < n =0
n = 5, m =6
```

Операції присвоювання

В мові С знак = означає операцію присвоювання деякого значення змінній. Для рядка вигляду *vr1=1024;* правильно казати *«присвоїти змінній vr1 значення 1024»*.

Перелік операцій присвоювання мови С ілюструє таблиця 4.4.

Таблиця 4.4.

Операції присвоювання

Операція	*Значення*
a = b	присвоювання значення *b* змінній *a*
a += b	додавання з присвоюванням, означає a = a + b
a –= b	віднімання з присвоюванням, означає a = a – b
a *= b	множення з присвоюванням, означає a = a * b
a /= b	ділення з присвоюванням, означає a = a / b
a %= b	залишок від ділення з присвоюванням, означає a = a % b
a <<= b	зсув вліво з присвоюванням, означає a = a << b
a >>= b	зсув вправо з присвоюванням, означає a = a >> b
a &= b	порозрядне І з присвоюванням, означає a = a & b
a |= b	порозрядне АБО з присвоюванням, означає a = a | b
a ^= b	побітове додавання за МОД2 з присвоюванням, означає a = a ^ b

Операція присвоювання повертає присвоєне значення як результат. Завдяки цій особливості можливий такий запис:

```
a=(b=c=1)+1;
```

Але таке використання операції присвоювання потрібно застосовувати обережно, щоб уникати ускладнення коду. Розглянемо приклад, який демонструє використання таких присвоювань (лістинг 4.3.).

Лістинг 4.3. Використання операції інкременту та декременту

```
#include <stdio.h>
main()
{ int data1, data2, data3;
  data1 = data2 = data3 = 68;
  printf("\ndata1 = %d\ndata2 = %d\ndata3 = %d",
    data1, data2, data3);
}
```

Результат виконання програми:

```
data1 = 68
data2 = 68
data3 = 68
```

Присвоювання відбувається справа наліво: спочатку змінна *data3* отримує значення 68, потім змінна *data2* і нарешті *data1*.

Операції порівняння

Операції порівняння (таблиця 4.5) здебільшого використовуються в умовних виразах. Результатом виконання таких операцій є логічне значення істина (true) або хиба (false). В мові С немає логічного (булевого) типу. Тому результатом умовного виразу є цілочисельне арифметичне значення. «Істина» – це ненульова величина, а «хиба» – це нуль. В більшості випадків в якості ненульового значення «істина» використовується одиниця.

Таблиця 4.5.

Операції порівняння

Операція	*Значення*
<	Менше
<=	менше або рівно
==	перевірка на рівність
>=	більше або рівно
>	Більше
!=	перевірка на нерівність

Приклад, наведений в лістингу 4.4, демонструє роботу операцій порівняння.

Лістинг 4.4. Використання операцій порівняння

```
#include <stdio.h>
main()
{   int tr, fal;
    tr = (111 <= 115); /* вираз істинний */
    fal = (111 > 115); /* вираз хибний */
    printf("true - % d false - % d \n", tr, fal);
}
```

Результат виконання програми:

```
data1 = 68
data2 = 68
data3 = 68
```

Логічні операції

Логічні операції &&, ||, ! використовуються здебільшого для «об'єднання» виразів порівняння у відповідності з правилами логічного І, логічного АБО та логічного заперечення (таблиця 4.6).

Таблиця 4.6.

Логічні операції

Операція	*Значення*
&&	логічне І (and)
||	логічне АБО (or)
!	логічне заперечення (not)

В таблиці 4.7 наведено результати логічних операцій для логічного І, логічного АБО та логічного заперечення.

Таблиця 4.7.

Таблиця істинності логічних операцій

E1	E2	E1&&E2	E1||E2	!E1
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Розглянемо приклад обчислення складних логічних виразів. Логічний вираз

```
(A<=B) && (B<=C)
```

складається з двох порівнянь, поєднаних логічною операцією «І» (&&).

Спочатку виконується порівняння A <= B. Якщо результат false (A > B), вираз одразу завершує обчислення і повертає false. Якщо результат true, виконується друге порівняння B <= C. Після обчислення обох частин результати зводяться за допомогою логічної операції &&. Приклад обчислення складного логічного виразу наведено в таблиці 4.8.

Таблиця 4.8.

Логіка обчислення складного логічного виразу

A	B	C	A <= B	B <= C	(A <= B) && (B <= C)
1	3	4	true	true	true
5	1	4	false	true	false
2	3	2	true	false	false
3	3	3	true	true	true

Приклад, наведений в лістингу 4.5, демонструє роботу логічних операцій.

Лістинг 4.5. Використання логічних операцій

```
#include <stdio.h>
main()
{
    int a=5, b=2, c=5;
    printf("a>b = %d\n",  a>b);
    printf("a<=c = %d\n", a<=c);
    printf("a!=c = %d\n", a!=c);
    printf("a==c = %d\n", a==c);
    printf("a>b && a>c = %d\n",  a>b && a>c);
    printf("a>b || a>c = %d\n",  a>b || a>c);
}
```

Результат виконання програми:

```
a>b = 1
a<=c = 1
a!=c = 0
a==c = 1
a>b && a>c = 0
a>b || a>c = 1
```

Порозрядні операції (побітові операції)

Порозрядні операції застосовуються тільки до цілочисельних операндів і «працюють» з їх двійковими представленнями. Ці операції неможливо використовувати із змінними типу *double*, *float*, *long double* (таблиця 4.9).

Таблиця 4.9.

Порозрядні операції

Операція	*Значення*
~	порозрядне заперечення
&	побітова кон'юнкція (побітове І)
|	побітова диз'юнкція (побітове АБО)
^	побітове додавання за МОД2

<<	зсув вліво
>>	зсув вправо

В таблиці 4.10 наведено результати логічних порозрядних операцій.

Таблиця 4.10.

Таблиця істинності логічних порозрядних операцій

E1	E2	E1&E2	E1^E2	E1|E2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Операція & записує в біт результат 1 тільки в тому випадку, якщо обидва порівнюваних біта дорівнюють 1, як показано в таблиці 4.9. Ця операція часто використовується для маскування окремих бітів числа.

Приклад:

```
0xF1 & 0x 35 = 0x 31
(10010011) & (00111101) == (00010001)
```

Операція | записує в біт результат 1 в тому випадку, якщо хоч би один з порівнюваних бітів дорівнює 1, як показано в таблиці 4.9. Ця операція часто застосовується для установки окремих бітів числа.

Приклад:

```
0x F1 | 0x 35 = 0x F5
(10010011) | (00111101) == (10111111)
```

Операція ^ записує в біт результату 1 в тому випадку, якщо порівнювані біти відрізняються один від одного, як показано в таблиці 4.9. Ця операція часто застосовується при виведенні зображень на екран, коли відбувається накладення декількох графічних шарів.

Приклад:

```
0x F1 ^ 0x 35 = 0x C4
(10010011) ^ (00111101) == (10101110)
```

На операції побітового додавання МОД2 ґрунтується метод обміну значень двох цілочисельних змінних.

Приклад:

```
a^=b^=a^=b;
```

Порозрядне заперечення ! заміняє кожну 1 на 0, а 0 на 1.

Приклад:

```
~(10011010) == (01100101)
```

Операція зсуву вліво (вправо) переміщує розряди першого операнду вліво (вправо) на число позицій, яке задане другим операндом. Позиції, що звільняються, заповнюються нулями, а розряди, що зсуваються за ліву (праву) границю, втрачаються.

Приклад:

```
(10001010) << 2 == (00101000)
(10001010) >> 2 == (00100010)
```

Програмний код, наведений в лістингу 4.6, демонструє роботу логічних порозрядних операцій. Значення змінних представлено в десятковій системі числення.

Лістинг 4.6. Використання логічних порозрядних операцій

```
#include <stdio.h>
main()
{
    int a, b;
    a = 25;
    b = 10;
    printf("a&b = %d\n", a & b);
    printf("a|b = %d\n", a | b);
    printf("a^b = %d\n", a ^ b);
    printf("~b = %d\n", ~b);
    printf("a<<b = %d\n", a << b);
    printf("a>>b = %d\n", a >> b);
```

```
    printf("!a = %d\n", !a);
}
```

Результат виконання програми:

```
a&b = 8
a|b = 27
a^b = 19
~b = -11
a<<b = 25600
a>>b = 0
!a = 0
```

Операція слідування (кома)

Операція «кома» (,) називається операцією слідування, яка «зв'язує» два довільних вирази. Список виразів, розділених між собою комами, обчислюються зліва направо.

Оператор «кома» використовується для зв'язки декількох виразів. Ліва сторона оператора «кома» завжди обчислюється як void (тобто не повертає значення). Це означає, що значення виразу, що знаходиться з правого боку, стане значенням розділеного комами виразу.

Приклад:

```
int a = 1, b = 3, c;
c = (b + a++, a += b);
```

Операція слідування використовується найчастіше в операторах циклів (оператори циклів буде розглянуто в розділі 6).

Програмний код, наведений в лістингу 4.7, демонструє роботу операції слідування «кома».

Лістинг 4.7. Використання операції слідування «кома»

```
#include <stdio.h>
main()
{
    int a = 3, b = 8, c;
    printf("-----  c=(a++, a+b) -----------\n");
```

```
    c = (a++, a + b);
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
    printf("-----  a=(b--,c) -----------\n");
    a = (b--, c);
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
}
```

Результат виконання програми:

```
----- c=(a++, a+b) -----------
a = 4
b = 8
c = 12
-----  a=(b--,c) -----------
a = 12
b = 7
c = 12
```

Контрольні запитання та завдання

1. Які основні операції є в мові програмування С?
2. Які є арифметичні операції?
3. Поясніть, що таке оператор присвоєння.
4. Чому буде дорівнювати зачення змінної x після виконання програмного коду:

   int a=015, b=100, c=1010, d=0x1F;

   int x=d/(c-b-a);
5. Яка операція використовується для присвоєння значень змінним?
6. Як позначається операція перевірки рівності двох виразів у мові С?
7. Як позначається операція перевірки нерівності двох виразів?
8. Яким буде значення змінної a у виразі int a = 9/2+9%4?
9. Яким буде значення виразу int a =14/2+31%5?
10. Визначте значення кожної змінної після операції, якщо на початку операції всі змінні мають значення рівне 5.

p *= x++;

q /= ++x;

w = (--x) + (w--);

k += ((--x)--) + 10;

11. Назвіть пріоритети виконання арифметичних операцій.
12. Обрахуйте значення виразів:

    10%2=?

    10%3=?

    3%11=?
13. Дайте визначення поняттю «порозрядна операція».
14. Які є порозрядні операції в мові програмування С?
15. Поясніть сенс поразрядной операції «логічне І».
16. Поясніть сенс поразрядной операції «логічне АБО».
17. За допомогою якої поразрядной операції здійснюється зсув ліворуч?
18. Що означає операція x<<n?
19. Що таке операція слідування (кома)?
20. Поясніть результат роботи фрагменту програмного коду:

    int a=1, b=7, c;

    c = (a, b++);
21. Напишіть програму, яка приймає рік як вхідні дані та виводить 1, якщо рік високосний, і 0 в іншому випадку.
22. Напишіть програму, яка приймає три числа з плаваючою крапкою як вхідні дані та виводить 1, якщо вони утворюють зростаючу послідовність, і 0 в іншому випадку.
23. Напишіть програму, яка приймає два цілих числа як вхідні дані та виводить 1, якщо обидва числа парні, і 0 в іншому випадку.
24. Напишіть програму, яка використовує порозрядну операцію & для перевірки, чи є введене користувачем число парним, і виводить відповідне повідомлення.
25. Напишіть програму, яка використовує порозрядну операцію | для встановлення певного біта в числі. Користувач повинен ввести число, номер біта (починаючи з 0) та значення, яке потрібно встановити (0 або 1).

5. Основи алгоритмізації

Алгоритми та їх властивості

Алгоритм – це чітко визначена для конкретного виконавця послідовність дій, які спрямовані на досягнення поставленої мети або розв'язання задачі певного типу.

У 820 році нашої ери в Бухарі був написаний підручник «Аль-Джабр Ва-аль-Мукабала» («Наука виключення скорочення»), в якому були описані правила виконання чотирьох арифметичних дій над числами в десятковій системі числення. Автором підручника був арабський математик Мухаммед Бен Муса аль-Хорезмі. Від слова «альджебр» у назві підручника пішло слово «алгебра», а від імені аль-Хорезмі – слово «алгоризм», що пізніше перейшло в слово «алгоритм».

Властивості алгоритмів :

1. *Зрозумілість*. В алгоритмі повинні бути лише операції, які знайомі виконавцеві. При цьому виконавцем алгоритму може бути: людина, комп'ютер, робот тощо.
2. *Масовість*. За допомогою складеного алгоритму повинен розв'язуватися цілий клас задач.
3. *Однозначність*. Будь-який алгоритм повинен бути описаний так, щоб при його виконанні у виконавця не виникало двозначних вказівок. Тобто різні виконавці згідно з алгоритмом повинні діяти однаково та прийти до одного й того ж результату.
4. *Правильність*. Виконання алгоритму повинно давати правильні результати.
5. *Скінченність*. Завершення роботи алгоритму повинно здійснюється в цілому за скінченну кількість кроків.
6. *Дискретність*. Алгоритм повинен складатися з окремих завершених операцій, які виконуються послідовно.
7. *Ефективність*. Алгоритм повинен забезпечувати розв’язання задачі за мінімальний час з мінімальними витратами оперативної пам’яті.

Способи представлення алгоритмів. Алгоритми можуть бути представлені: у вигляді таблиці, описані як система словесних правил (лексикографічний або словеснокроковий спосіб запису алгоритму), представлені алгоритмічною мовою у вигляді послідовності операторів (операторний спосіб), або з допомогою графічного зображення у формі блок-схем (графічний або геометричний спосіб запису алгоритму).

Слід зауважити, що графічному способу подання алгоритмів надається перевага через його простоту, наочність і зручність. *Блок-схема* алгоритму зображає послідовність блоків, з'єднаних між собою стрілками, які вказують послідовність виконання і зв'язок між блоками. Всередині блоків записується їх короткий зміст.

Блок-схеми

Блок-схема – це спосіб представлення алгоритму в графічній формі, у вигляді геометричних фігур, сполучених між собою лініями (стрілками). Форма блока визначає тип дії, а текст всередині блоку дає детальне пояснення конкретної дії. Стрілки на лініях, що сполучають блоки схеми, вказують послідовність виконання команд, передбачених алгоритмом. Блок-схеми, за рахунок наочності спрощують створення ефективних алгоритмів, розуміння роботи вже створених, а як наслідок і їх оптимізацію. Існуючі стандарти на типи блоків дозволяють легко адаптувати алгоритми, створені у вигляді блок-схем до будь-яких існуючих на сьогоднішній день мов програмування.

Зображення блоків у алгоритмі, їх розміри, товщина ліній, кут нахилу ліній тощо, регламентуються Державним стандартом ISO 5807:2016 (ISO 5807:1985, IDT) «Символи та угоди щодо документації стосовно даних, програм та системних блок-схем, схем мережевих програм та схем системних ресурсів».

Блоки у блок-схемі з'єднуються лініями потоків. У кожен блок може входити не менше однієї лінії, з блоку ж (окрім логічного) може виходити лише одна лінія потоку. З логічного блоку завжди виходять дві лінії потоку: одна у випадку виконання умови, інша – при її невиконанні. Бажано, щоб лінії потоку не перетинались.

Алгоритм може бути детальним, або спрощеним (деякі зрозумілі блоки можуть не записуватись, інакше алгоритм збільшується в розмірі).

Основні види блок-схем:

- лінійні (нерозгалужені);
- розгалужені;
- циклічні;
- з підпрограмами;
- змішані.

Основні графічні елементи блок-схем зображені на рисунку 5.1.

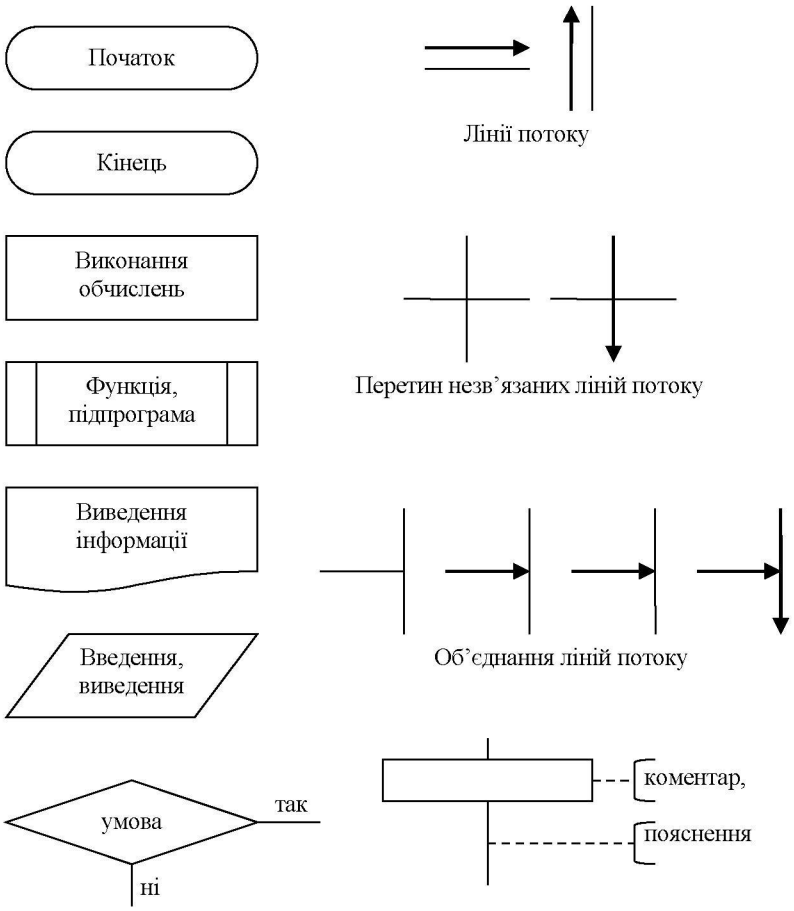


Рисунок 5.1 – Основні графічні елементи блок-схем

Базові алгоритмічні конструкції

Базові алгоритмічні конструкції – це способи управління процесами обробки даних. Виділяють три базові алгоритмічні конструкції:

1. лінійні алгоритми;
2. алгоритми розгалуженої структури;
3. алгоритми циклічної структури.

Лінійні алгоритми (рис. 5.2). Алгоритм називається *лінійним*, якщо блоки алгоритму виконуються один за одним. Алгоритми лінійної структури не містять умовних і безумовних переходів, циклів.

Алгоритми розгалуженої структури (рис. 5.3). Якщо вибраний метод розв'язання задачі передбачає виконання різних дій в залежності від значень будь-яких змінних, але при цьому кожна гілка алгоритму в процесі розв'язання задачі виконується не більше одного разу, алгоритм називається *розгалуженим*.

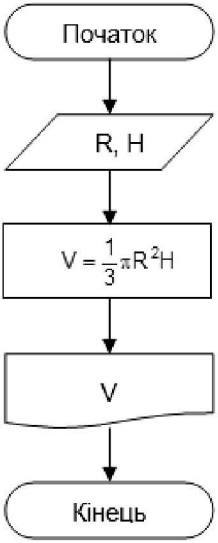


Рисунок 5.2 – Приклад лінійного алгоритму

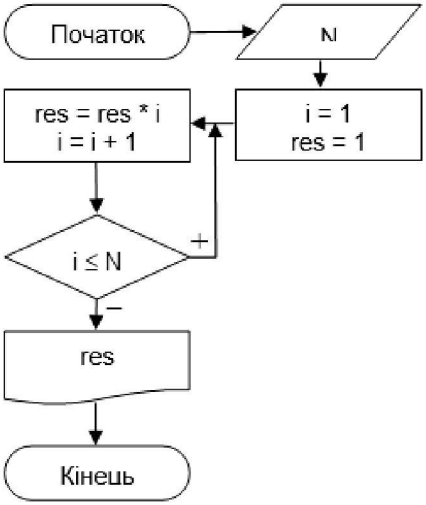


Рисунок 5.3 – Приклад розгалуженого алгоритму

Алгоритми циклічної структури (рис. 5.4). *Цикл* - це команда виконавцеві (компілятору) багаторазово повторити послідовність певних команд.

При багатократному проходженні деяких ділянок алгоритму в процесі виконання алгоритм називається *циклічним*. Кількість проходжень циклу повинна бути повністю визначена алгоритмом розв'язання задачі, інакше виникає "зациклювання", при якому процес розв'язання задачі не може завершитися.

Алгоритми розв'язку задач циклічної структури можуть бути такими, що при однократному проході циклу деякі ділянки алгоритму виконуються неодноразово, тобто всередині циклу існують інші цикли.

Алгоритми такої структури називаються *алгоритмами з вкладеними циклами*.

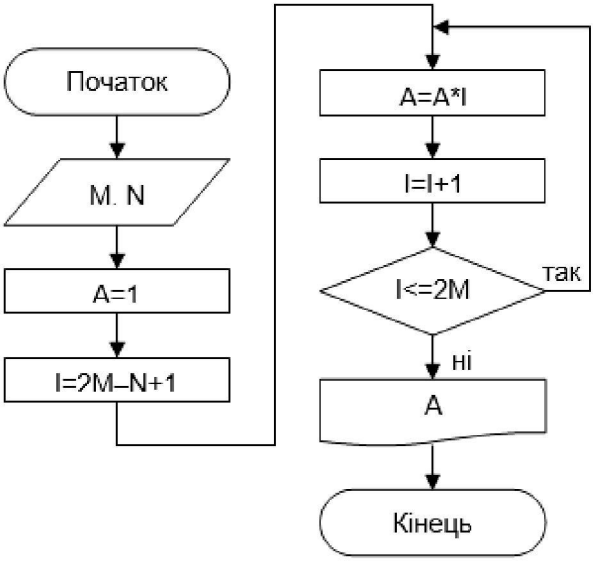


Рисунок 5.4 – Приклад циклічного алгоритму

Контрольні запитання та завдання

1. Поясніть поняття «алгоритм»?
2. Назвіть основні властивості алгоритмів?
3. Які способи представлення алгоритмів існують?
4. Чи регламентовано представлення алгоритмів нормативними документами? Якщо так то якими?
5. Чому важливо дотримуватись державних стандартів при створенні блок-схем?
6. Що таке блок-схема і для чого вона використовується?
7. Які основні елементи використовуються при створенні блок-схем?
8. Які існують типи блок-схем і в чому їх відмінності?
9. Які основні правила створення блок-схем?
10. Які програми та інструменти можна використовувати для створення блок-схем?
11. Який блок використовується для початку алгоритму?

12. Який блок використовується для завершення алгоритму?
13. Який блок використовується для представлення обчислень або присвоєння значень?
14. Який блок використовується для умовних переходів?
15. Який блок використовується для введення та виведення даних?
16. Як відобразити цикли в блок-схемі?
17. Як створити блок-схему для складного алгоритму?
18. Як блок-схема допомагає в розробці програмного забезпечення?
19. Як блок-схема допомагає в аналізі алгоритмів?
20. Як блок-схема допомагає в документуванні процесів?
21. Зобразіть блок-схему яка перевіряє чи введене користувачем число є парним.
22. Створіть блок-схему для перевірки чи є введене користувачем число простим.
23. Побудуйте блок-схему для знаходження найбільшого числа з трьох заданих чисел.
24. Створіть блок-схему для обчислення суми елементів заданого масиву цілих чисел.
25. Зобразіть блок-схему для програми, яка виводить таблицю множення для заданого числа.

6. Управляючі структури

Оператори розгалуження

Оператори – це основні елементи, з яких «будуються» програми на будь-якій мові програмування. Більшість операторів складаються з виразів.

Вираз представляє собою об'єднання операцій і операндів. Найпростіший вираз складається з одного операнду.

Приклади виразів:

```
5
-7
10+21
a*(b+d*1)-1
x=++a%3
a>3
```

Неважко помітити, що операнди можуть бути константами, змінними, їх об'єднаннями. Деякі вирази складаються з менших виразів. Дуже важливою особливістю мови С є те, що кожний вираз має значення.

Приклад виразів і їх значень:

–5+7	2
1<2	1
6+(a=1+2)	9
a=1+2	3

Оператором-виразом називається вираз, який закінчується крапкою з комою. Всі оператори можна поділити на декілька груп:

- оператори присвоювання;
- оператори розгалуження;
- оператори циклів.

Проте, оператори найчастіше відносяться до більш ніж однієї групи. Наприклад, оператор *if ((a=b+c)>d)* складається з представників наступних груп: присвоювання та розгалуження. У тому і є гнучкість

С, що є можливість поєднання операторів різних груп. Але не варто цим зловживати – програма може правильно працювати, проте надто заплутаною та нечитабельною.

Оператор розгалуження if

Оператор *розгалуження (if - else)* призначений для виконання дій в залежності від істинності або хибності умови. Умова, що перевіряється, повинна розміщуватись у круглих дужках.

Синтаксис оператора:

```
if (<умова>)
    <оператор1>;
[else
    <оператор2;>]
```

Рисунок 6.1 – Графічне представлення синтаксису оператора if

Конструкція *else* в операторі *if* є необов'язковою. Оператор *if* може працювати самостійно. Розгалуження часто зображають за допомогою блок-схем. Це допомагає візуалізувати логіку алгоритму.

Приклад, наведений в лістингу 6.1, демонструє роботу повної форми оператора *if - else*.

Лістинг 6.1. Використання оператора *if – else*

```
#include <stdio.h>
main()
{
    int a = 5, b = 10, c;
    if (a > 10)
        c = a - b;
    else
        c = a + b;
    printf("c = %d", c);
}
```

Результат виконання програми:

```
c = 15
```

Приклад, наведений в лістингу 6.2, демонструє роботу не повної форми оператора *if - else*.

Лістинг 6.2. Використання оператора *if*

```
#include <stdio.h>
main()
{
    int a = 5, b = 10, c;
    if (a != 0)
        c = b / a;
    printf("c = %d", c);
}
```

Результат виконання програми:

```
c = 2
```

Умова хибна, якщо вона дорівнює нулю, в інших випадках вона істинна. Це означає, що навіть від'ємні значення розглядаються як істинні. До того ж, умова, що перевіряється, повинна бути скалярною, тобто зводитися до простого значення, яке можливо перевірити на рівність нулю. Взагалі не рекомендується використання змінних типу *float* або *double* в логічних виразах перевірки умов з причини недостатньої точності подібних виразів. Більш досвідчені програмісти скорочують оператори.

Приклад:

if (вираз!=0) оператор;

або

if (вираз) оператор;

Обидва логічні вирази функціонально еквівалентні, тому що *будь-яке ненульове значення розцінюється як істина*. Це продемонстровано програмно на лістингах 6.3 – 6.4.

Лістинг 6.3. Використання оператора *if - else*

```
   /* програма виводить результат ділення двох дійсних
чисел */
#include<stdio.h>
main()
{
   float a, b, c;
   printf("Введіть число a :\n");
   scanf("%f", &a);
   printf("Введіть число b :\n");
   scanf("%f", &b);
   if (b == 0)
      printf("Ділення да нуль !\n");
   else
   {
      c = a / b;
      printf("a : b == %g", c);
      }
   }
```

Результат виконання програми:

```
Введіть число a :
28
Введіть число b :
4
a : b == 7
```

Лістинг 6.4. Використання оператора *if*

```
/* застосування умовного оператора для перевірки
коректності введення певних значень */
#include <stdio.h>
main()
{
   int number;
   int ok;
   printf("Введіть число з інтервалу 1..100 : ");
```

```
    scanf("%d", &number);
    ok = (1 <= number) && (number <= 100);
    if (!ok)
          printf("Не коректно !!\n");
}
```

Результат виконання програми:

```
Введіть число з інтервалу 1..100 : -5
Не коректно !!
```

Змінній *ok* присвоюється значення результату виразу: ненульове значення, якщо істина, і в протилежному випадку – нуль. Умовний оператор *if(!ok)* перевіряє, якщо *ok* дорівнюватиме нулю, то *!ok* дасть позитивний результат і буде отримано повідомлення про некоректність.

Оператор switch

Оператор *switch* призначений для вибору одного з декількох альтернативних шляхів виконання програми, що є ефективним способом реалізації складних логічних розгалуджень. Він дозволяє оптимітизуати програмний код, який написаний з використанням багатьох вкладених операторів *if-else*.

Синтаксис оператора *switch*:

```
switch(<вираз цілого типу>)
{
  case <значення_1>:
           <послідовність_операторів_1>;
           break;
  case <значення_2>:
           <послідовність_операторів_2>;
           break;
  ...
  case <значення_n>:
           <послідовність_операторів_n>;
           break;
  [default:
            <послідовність_операторів_n+1>;]
}
```

Виконання оператора *switch* починається з обчислення значення виразу у круглих дужках. Після цього управління передається одному з блоків *case*, значення якого співпадає зі значенням виразу у круглих дужках. Блок *default* в операторі *switch* використовується для визначення блоку коду, який буде виконаний, якщо жоден із варіантів case не відповідає значенню виразу. *default* не є обовязковим для використання. Треба також зазначити обов'язкове застосування оператора *break* у кожному з *case*-блоків, що негайно передасть керування у точку програми, що слідує відразу за останнім оператором у *switch*.

Блоки *case* та *default* можуть розташовуватись у будь-якому порядку. Якщо *break* відсутній, тоді програмний код продовжує виконуватись, навіть, якщо далі знаходить наступний блок *case*.

В лістингу 6.5 продемонстровано роботу оператора *switch* без застосування *break*.

Лістинг 6.5. Використання оператора *switch*

```
#include <stdio.h>
main()
{
    int day;
    printf("Day = ");
    scanf("%d", &day);
    switch (day)
    {
    case 6: printf("субота\n");
    case 1: printf("понеділок\n");
    default:printf("помилка\n");
    case 4: printf("четвер\n");
    case 7: printf("неділя\n");
    case 2: printf("вівторок\n");
    case 3: printf("середа\n");
    case 5: printf("п'ятниця\n");
}
```

Результат виконання програми 1:

```
Day = 1
понеділок
помилка
четвер
```

```
неділя
вівторок
середа
п'ятниця
```

Результат виконання програми 2:

```
Day = 9
помилка
четвер
неділя
вівторок
середа
п'ятниця
```

Результат виконання демонструє роботу оператора *switch* при введенні значення «1» в змінну Day. Якщо в змінну Day буде введено некоректне значення, наприклад «9», то спрацює блок *default*, а також всі наступні оператори *case*.

В лістингу 6.6 продемонстровано роботу оператора *switch* з застосуванням *break*.

Лістинг 6.6. Використання оператора *switch*

```
#include <stdio.h>
main()
{   int day;
    printf("Day = ");   scanf("%d", &day);
    switch (day)
    {
    case 6: printf("субота\n"); break;
    case 1: printf("понеділок\n"); break;
    default:printf("помилка\n"); break;
    case 4: printf("четвер\n"); break;
    case 7: printf("неділя\n"); break;
    case 2: printf("вівторок\n"); break;
    case 3: printf("середа\n"); break;
    case 5: printf("п'ятниця\n"); break;
    }
}
```

Результат виконання програми 1:

```
Day = 1
Понеділок
```

Результат виконання програми 2:

```
Day = 9
Помилка
```

Використання *break,* дозволяє зупинити виконання подальших блоків, що надає коректності роботи оператора *switch*.

В лістингу 6.7 продемонстровано, як працює оператор switch, коли значення виразу може бути додатнім або від'ємним.

Лістинг 6.7. Використання оператора *switch*

```
#include <stdio.h>
main()
{
    int i, n = 2, z = 4, p = 7;
    printf("i = ");
    scanf("%d", &i);
    switch (i)
    {
    case -1:
        n++;
        break;
    case 0:
        z++;
        break;
    case 1:
        p++;
        break;
    }
    printf("n= %d ", n);
    printf("z= %d ", z);
    printf("p= %d ", p);
}
```

Результат виконання програми:

```
i = -1
n= 3 z= 4 p= 7
```

В лістингу 6.8 продемонстровано роботу оператора *switch* для завдання «абетка» з використанням значення виразу символьного типу даних.

Лістинг 6.8. Використання оператора *switch*

```
#include <stdio.h>
main() {
    char c;
    printf("Symbol = ");
    scanf("%c", &c);
    switch (c)
    {
    case 'A':
    case 'a': printf("Antelope"); break;
    case 'B':
    case 'b': printf("Beaver"); break;
    default:
       printf("Input error"); break;
    }
}
```

Результат виконання програми:

```
Symbol = b
Beaver
```

Тернарний оператор

Операція ?: – єдина тернарна операція в мові С. Її синтаксис :

умова ? вираз_1 : вираз_2

Принцип роботи цієї операції схожий на оператор *if – else*. Спочатку обчислюється вираз умови, якщо цей вираз має ненульове значення, то обчислюється *вираз_1*. Результатом операції ?: в даному

випадку буде значення *виразу_1*. Якщо вираз умови рівний нулю, то обчислюється *вираз_2* і його значення буде результатом операції. В будь-якому випадку обчислюється тільки один із виразів (*вираз_1* або *вираз_2*).

Для прикладу застосуємо оператор для знаходження найбільшого з двох чисел (лістинг 6.9 - 6.10).

Лістинг 6.9. Використання тернарного оператора

```
#include <stdio.h>
main() {
    int x = 5, y = 2;
    x > y ? printf("%d", x) : printf("%d", y);
}
```

Результат виконання програми:

```
max = 20
```

Лістинг 6.10. Використання тернарного оператора

```
#include <stdio.h>
main()
{
    int a = 10, b = 20;
    int max = (a > b) ? a : b;
    printf("max = %d", max);
}
```

Результат виконання програми:

```
max = 20
```

Також тернарний оператор може застосовуватись у більш складних випадках. У лістингу 6.11 наведено приклад вкладення тернарних операторів.

Лістинг 6.11. Використання тернарного оператора

```
#include <stdio.h>
main() {
    unsigned day;
    printf("day: ");
    scanf("%u", &day);

    day == 1 ? printf("Monday") :
       day == 2 ? printf("Tuesday") :
       day == 3 ? printf("Wednesday") :
       day == 4 ? printf("Thursday") :
       day == 5 ? printf("Friday") :
       day == 6 ? printf("Saturday") :
       day == 7 ? printf("Sunday") :
       printf("Wrong day of week");
}
```

Результат виконання програми:

```
day: 4
Thursday
```

Оператори циклів

Цикл з передумовою while

Оператор *while* використовується для організації циклічного виконання оператора або серії операторів, поки виконується певна умова.

Синтаксис :

while (<логічний вираз>)
оператор;

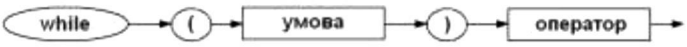


Рисунок 6.2 – Синтаксис оператора while

Цикл закінчується у наступних випадках:

1. умовний вираз у заголовку приймає нульове значення;
2. у тілі циклу досягнуто місця, де розташований оператор *break*;
3. у тілі циклу виконаний оператор *return*;

У перших двох випадках керування передається оператору, розташованому безпосередньо за циклом, у третьому випадку активна на той момент функція завершує свою роботу, повертаючи якесь значення.

Приклад використання оператора циклу *while* для виведення на екран послідовності чисел від 0 до 9 наведено на лістингу 6.12.

Лістинг 6.12. Використання оператора циклу *while*

```
#include <stdio.h>
main()
{
    int i = 0;
    while (i < 10) {
        printf("%d ", i);
        i++;
    }
}
```

Результат виконання програми:

```
0 1 2 3 4 5 6 7 8 9
```

Нерідкою помилкою програмістів є використання замість оператора порівняння *(==)* оператора присвоювання *(=)*. Наприклад, наступний фрагмент створить нескінчений цикл (лістинг 6.13).

Лістинг 6.13. Некоректне використання оператора циклу *while*

```
#include <stdio.h>
main()
{
    int j = 5;
    while (j = 5)    /* змінній j присвоїти значення 5 */
    {
        printf("%d\n", j);
```

```
        j++;
    }
}
```

Результат виконання програми:

```
5
5
5
5
5
5
5
....
```

Компілятор мови С попередить про некоректне присвоювання в даному випадку, виправити яке особливих труднощів не викличе. Втім, часто такий цикл використовується для перевірки відповіді користувача на питання з програми ("так чи ні ?") (лістинг 6.14).

Лістинг 6.14. Використання оператора циклу *while*

```
#include <stdio.h>
main() {
    char ch;
    printf("Підтверджуєте ? Так чи ні ?(y/n);");
    scanf("%c", &ch);
    while (ch != 'y' && ch != 'n')
    {
        printf("\n Відповідайте так чи ні . . (y/n);");
        scanf("%c", &ch);
    }
}
```

Результат виконання програми:

```
Відповідайте так чи ні . . (y/n);f

Відповідайте так чи ні . . (y/n);h
```

```
Відповідайте так чи ні . . (y/n):g

Відповідайте так чи ні . . (y/n):n
```

Тіло циклу почне виконуватися, якщо користувач введе будь-який символ, відмінний від *y* або *n*. Цикл виконується доти, доки користувач не введе *y* або *n*.

Цікаво розглянути й наступний приклад, що застосовує оператор *while* для підрахунку факторіалу (лістинг 6.15).

Лістинг 6.15. Використання оператора циклу *while*

```
#include <stdio.h>
main() {
    long total, number = 5;
    printf("factorial %d = ", number);
    total = number;
    while (--number)
       total *= number;
    printf("%d", total);
}
```

Результат виконання програми:

```
factorial 5 = 120
```

Цикл з постумовою do..while

Оператор *do..while* використовується для організації циклічного виконання оператора або серії операторів, які називаються тілом циклу, до тих пір, поки умова не стане хибною.

Синтаксис:

```
do
   <оператор>;
while (<логічний_вираз>);
```

Рисунок 6.3 – Синтаксис оператора *do..while*

Особливістю циклу є те, що спочатку виконується тіло циклу, а потім перевіряється умова; тіло циклу виконається мінімум один раз (навіть якщо умова одразу буде хибною).

Для прикладу розглянемо задачу: дано ціле число N > 0, потрібно знайти суму 1 + 1/2 + 1/3 + ... + 1 / N. Змоделюємо два варіанти розв'язку використовуючи цикли *do..while* та *while* (лістинг 6.16 - 6.17).

Лістинг 6.16. Використання оператора циклу *do..while*

```
#include <stdio.h>
main()
{
    int N = 10, j = 1;
    float Sum = 0;
    do {
        Sum += 1.0 / j;
        j++;
    } while (j <= N);

    printf("Sum=%f\n", Sum);
}
```

Результат виконання програми:

```
Sum=2.928968
```

Лістинг 6.17. Використання оператора циклу *while*

```
#include <stdio.h>
main()
{
    int N = 10, j = 1;
    float Sum = 0;
    while (j <= N) {
        Sum += 1.0 / j;
        j++;
    }
    printf("Sum = %f\n", Sum);
}
```

Результат виконання програми:

```
Sum = 2.928968
```

Розглянемо приклад, що застосовує оператор *do..while* для підрахунку факторіалу (лістинг 6.18).

Лістинг 6.18. Використання оператора циклу *do..while*

```
#include <stdio.h>
main()
{
    int n, i;
    float fact;
    printf("Програма обчислення n!\n");
    printf("Введіть число n: ");
    scanf("%d", &n);
    i = 1;
    fact = 1;
    do {
        fact *= i;
        i++;
    } while (i <= n);
        printf("n! = %g", fact);
}
```

Результат виконання програми:

```
Програма обчислення n!
Введіть число n: 5
n! = 120
```

Цикл for

Оператор *for* забезпечує циклічне повторення деякого оператора певне число разів. Оператор, який повторюється називається тілом циклу. Повторення циклу звичайно здійснюється з використанням деякої змінної (лічильника), яка змінюється при кожному виконанні тіла циклу. Повторення завершується, коли лічильник досягає заданого значення.

Синтаксис оператора:

for ([ініціалізація];[перевірка_умови];[лічильник])
оператор ;

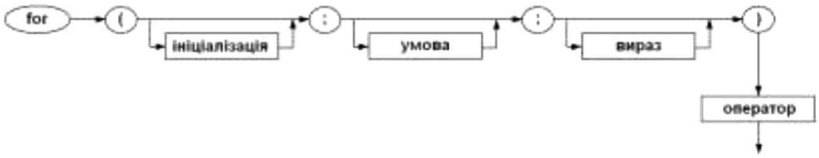


Рисунок 6.4 – Синтаксис оператора for

В операторі *for* присутні три складових частини – ініціалізація, умова, лічильник. Перша складова служить для ініціалізації лічильника, друга – для перевірки кінця циклу, а третя – для зміни значення лічильника. Кожена з трьох складових частин може бути відсутня.

Схема роботи циклу *for*:

1. Здійснюється ініціалізація лічильника циклу.
2. Перевіряється істинність умови.
3. Якщо умова істинна – виконується тіло циклу (оператор), якщо хибна – вихід з циклу.
4. Здійснюється зміна лічильника і перехід до пункту 2.

В лістингу 6.19 наведено приклад використання циклу for для виведення парних чисел у проміжку від 1 до 16 з використанням умовного оператора *if* та операції залишку від ділення.

Лістинг 6.19. Використання оператора циклу *for*

```
#include <stdio.h>
main()
{
    int n = 16;
    for (int i = 1; i <= n; i++)
    {
        if (i % 2 == 0) {
            printf("%d ", i);
        }
    }
}
```

Результат виконання програми:

```
2 4 6 8 10 12 14 16
```

В лістингу 6.20 наведено приклад використання циклу for для виведення парних чисел у проміжку від 16 до 1 використовуючи зміну лічильника.

Лістинг 6.20. Використання оператора циклу *for*

```
#include <stdio.h>
main() {
    int i;
    for (i = 16; i > 0; i = i - 2)
          printf(" %d", i);
}
```

Результат виконання програми:

```
16 14 12 10 8 6 4 2
```

Для того, щоб продемонструвати гнучкість даного різновиду циклу, розглянемо інший варіант рішення цієї ж задачі. Продемонструємо весь перелік обчислень в одному операторі *for*.

Лістинг 6.21. Використання оператора циклу *for*

```
#include <stdio.h>
main()
{
    int i;
    for (i = 16; i > 0; printf(" %d", i), i = i - 2);
}
```

Результат виконання програми:

```
16 14 12 10 8 6 4 2
```

Кожен програміст може використовувати власний стиль написання програм та надати перевагу більш стислому викладанню або взагалі скористатися іншим оператором. Цикли *for* та *while* є взаємозамінними.

```
for (ініціалізація; перевірка_умови; зміна_лічильника)
{
  оператор;
}
```

Аналогічний цикл *while* буде виглядати наступним чином:

```
ініціалізація;
while (перевірка_умови)
{
 оператор;
 зміна_лічильника;
}
```

Наведені приклади на лістингах 6.22 та 6.23 демонструють підрахунок кількості чисел кратних трьом на проміжку від 3 до 100.

Лістинг 6.22. Використання оператора циклу *for*

```
#include <stdio.h>
main() {
    int result = 0;
    for (int i = 3; i <= 100; i += 3)
        result++;
    printf("result = %d", result);
    }
```

Результат виконання програми:

```
result = 33
```

Лістинг 6.23. Використання оператора циклу *while*

```
#include <stdio.h>
main() {
    int result = 0, i = 3;
    while (i <= 100)
```

```
    {
        result++;
        i += 3;
    }
        printf("result = %d", result);
}
```

Результат виконання програми:

```
result = 33
```

Не завжди гнучкість переважає стисліть та навпаки. Справа за конкретною ситуацією. Зрештою, вибір циклу може бути справою смаку конкретного програміста – саме йому вирішувати, які оператори застосувати для запису того чи іншого алгоритму.

Оператор розриву break

Оператор розриву *break* використовується для переривання виконання операторів *do..while*, *for*, *while* або *switch*. В операторі *switch* він використовується для завершення блоку *case*.

Синтаксис:

```
break;
```

В операторах циклу для негайного завершення використовується оператор *break*. Коли він зустрічається всередині оператора циклу, то здійснюється негайний вихід з циклу і перехід до виконання наступного оператору.

В лістингу 6.24 наведено приклад підрахунку суми всіх введених з клавіатури чисел доки не буде введено 0. В даному прикладі цикл *while* є нескінченним і для закінчення його роботи використано оператор *break*. Цикл називається нескінченним, коли не вказано умову закінчення його роботи (умова завжди істинна) і постійно виконується тіло циклу. Цикли for (;;){ } та while (1){ } є синтаксично коректними, але є нескінченними.

Лістинг 6.24. Використання оператора розриву *break*

```
#include <stdio.h>
main() {
```

```
    int number, sum = 0;
    printf("Введіть числа (0 для завершення):\n");
    while (1)
    {
        scanf("%d", &number);
        if (number == 0)
        {
            break;
        }
        sum += number;
    }
        printf("Сума введених чисел: %d\n", sum);
}
```

Результат виконання програми:

```
Введіть числа (0 для завершення):
5
6
5
4
2
0
Сума введених чисел: 22
```

Оператор продовження continue

Оператор *continue* передає управління на наступну ітерацію в операторах циклу *do..while*, *for*, *while*. Він може розміщуватися тільки в тілі цих операторів. В операторах *do..while* і *while* наступна ітерація починається з обчислення виразу умови. Для оператора *for* наступна ітерація починається з обчислення зміни значення лічильника.

Синтаксис :

```
continue;
```

В лістингу 6.25 наведено приклад вирішення задачі табуляції функції y=1/x коли x змінюється на проміжку від -5 до 5 з кроком 2,5. Для уникнення помилки ділення на 0 застосовується оператор продовження *continue*. Коли значення змінної x буде дорівнювати 0 відбудеться перехід на зміну лічильника.

Лістинг 6.25. Використання оператора продовження *continue*

```
#include <stdio.h>
main() {
    double a = -5, b = 5, h = 2.5, res;
    for (int i = a; i <= b; i += h)
    {
        if (i == 0)
            continue;
        res = 1.0 / i;
        printf("1 / %g = %g\n", i, res);
    }
}
```

Результат виконання програми:

```
1 / -0.2 = -5
1 / -0.5 = -2
1 / 0.5 = 2
1 / 0.25 = 4
```

В лістингу 6.26 наведено приклад вирішення задачі знаходження квадрату п'яти додатніх чисел. При введенні від'ємного числа оператор *continue* передає управління на зміну лічильника.

Лістинг 6.26. Використання оператора продовження *continue*

```
#include <stdio.h>
main() {
    int x;
    for (int i = 0; i < 5; ++i) {
        printf("x = "); scanf("%d", &x);
        if (x < 0) continue;
        printf("x*x =%d\n", x * x);
    }
}
```

Результат виконання програми:

```
x = 2
```

```
x*x =4
x = -5
x = 6
x*x =36
x = -3
x = 25
x*x =625
```

Оператор continue служить ефективним інструментом для керування потоком виконання в циклах. Він сприяє спрощенню та гнучкості коду, але його використання потребує обережності, щоб не ускладнити розуміння логіки програми.

«Порожній» оператор

Порожній оператор – це оператор що складається лише з крапки з комою. Він може використовуватися в будь-якому місці програми, де за правилами мови програмування очікується оператор, але виконувати дії немає потреби.

Синтаксис:

```
;
```

Приклад:

```
for (i=0; i<10; printf("%d\n",i))  ;
```

Контрольні запитання та завдання

1. Які оператори розгалуження існують в мові програмування С?
2. Для чого використовується оператор if-else?
3. Яка структура оператора if-else?
4. Які типи даних можна використовувати в умовах оператора if-else?
5. Як працюють оператори порівняння в умовах?
6. Поясніть фрагмент коду. Яке значення матиме змінна y?

```
int x = 4, y;
if(x <= 4) y = 2*x;
else y = x;
```

7. Яке основне призначення оператора switch?
8. Який тип даних може бути використаний у виразі оператора switch?
9. Як працює блок case оператора switch?
10. Яка роль блоку default оператора switch?
11. Чи можна робити вкладення оператора switch в інший switch?
12. Вкажіть що виведе на екран поданий фрагмент коду?

```
int a = 1;
switch (a)
{ case 1: case 2: case 3:printf("Three"); }
```

13. Що таке тернарний оператор і як він записується?
14. Які переваги тернарного оператора перед оператором if?
15. Коли краще використовувати тернарний оператор, а коли – if?
16. Поясніть фрагмент коду. Чому буде дорівнювати змінна max?

```
int x = 10, y = 20;
int max = (x > y) ? x : y;
```

17. Поясніть фрагмент коду.

```
int num = 15;
printf("%s\n", (num % 2 == 0) ? "Парне" : "Непарне");
```

18. Що буде виведено на екран при виконанні фрагменту коду:

```
int a=2, b=5, y;
y = (a > b) ? a : b;
printf("%d", y);
```

19. Які види циклів існують у мові програмування С?
20. Як працює цикл for?
21. Опишіть синтаксис та особливості циклу do-while?
22. Яка різниця між циклами while та for?
23. Коли краще використовувати while, а коли do-while? Наведіть приклади.
24. Наведіть приклади використання оператора continue.
25. Як можна перервати цикл достроково?
26. Що таке нескінченний цикл?
27. Що буде виведено на екран при виконанні фрагменту коду:

```
for(int i = 3; i > 0; i--)
    printf("%d ",2*i);
```

28. Що буде виведено на екран при виконанні фрагменту коду:

```
int sum=0, i=2;
while(i<4) {
sum+=i; i++;
}
printf("%d",sum);
```

29. Що буде виведено на екран при виконанні фрагменту коду

```
int n = 1;
for(int i = 1; i < 6; i++)
  if(i < 3) n *= i;
printf("%d ", n);
```

30. Чому буде дорівнювати змінна sum при викнанні програмного коду?

```
int i = 0, sum = 0;
while(i <7) {
 if(i % 2 == 0)  sum += i; i++; }
printf("%d  ", sum);
```

31. Напишіть програму, яка запитує користувача ввести довжини трьох сторін трикутника та визначає, чи є він рівностороннім, рівнобедреним або різностороннім.
32. Напишіть програму виведення послідовності фібоначі до n-го елементу, n –ціле число, яке вводить користувач.
33. Напишіть програму, яка запитує користувача ввести кількість балів (від 0 до 100) та виводить на екран відповідну оцінку (наприклад, 90-100 - "A", 80-89 - "B" тощо).
34. Напишіть програму, яка запитує користувача ввести рік та визначає, чи є він високосним. (Рік є високосним, якщо він ділиться на 4, але не ділиться на 100, або якщо він ділиться на 400).
35. Напишіть програму, яка запитує введення у користувача символа і числа та виводить на консоль зображення трикутника заданого розміру та вказаним символом.

7. Покажчики і масиви

Покажчики

Основні відомості про покажчики

В результаті процесу компіляції всі імена змінних замінюються на адреси комірок пам'яті, в яких зберігаються відповідні дані. У машинному коді замість імен змінних використовуються їхні адреси. Це називається прямою адресацією, коли значення отримуються за вказаною адресою. Наприклад, в операторі присвоювання k = j на машинному рівні відбувається копіювання значення з області оперативної пам'яті, що відповідає змінній j, до області оперативної пам'яті, відведеної для змінної k. Таким чином, під час виконання машинної програми операції виконуються над операндами — значеннями змінних, які зберігаються за відповідними адресами в пам'яті. Імена змінних у командах машинного рівня не використовуються; їх компілятор замінює на адреси. Проте програміст не має прямого доступу до цих адрес, якщо не використовує покажчики.

Покажчики в *C* використовується набагато інтенсивніше, аніж в інших мовах програмування, тому що деякі обчислення виразити можливо лише за їх допомогою, а частково й тому, що з ними утворюються більш компактні та ефективніші програми. Навіть існує твердження: аби стати знавцем *C*, потрібно бути спеціалістом з використання покажчиків.

Покажчик (вказівник) – це змінна або константа стандартного типу даних для збереження адреси змінної визначеного типу. Значення покажчика – це беззнакове ціле, воно повідомляє, де розміщена змінна, і нічого не говорить про саму змінну.

Покажчик може вказувати на дані різних типів: простих (int, char, float), складних (структури, об'єднання) або навіть загального типу void. Покажчик типу void є універсальним і може посилатися на будь-яку область пам'яті, але не дозволяє безпосередньо доступатися до значень, які там зберігаються. Щоб працювати з такими даними, потрібно явно перетворити покажчик на конкретний тип. Розмір пам'яті для самого покажчика і формат збереженої адреси (вмісту покажчика) залежить від типу комп'ютера та обраної моделі пам'яті. Константа *NULL* зі стандартного файлу *stdio.h* призначена для ініціалізації покажчиків нульовим (незайнятим) значенням адреси.

Змінна типу покажчик оголошується подібно звичайним змінним із застосуванням унарного символу «*». Форма оголошення змінної типу покажчик наступна:

тип [модифікатор] * ім'я-покажчика ;

Тип – найменування типу змінної, адресу якої буде містити змінна-покажчик.

Ім'я-покажчика – ідентифікатор змінної типу покажчик.

Модифікатор необов'язковий компонент і може мати значення:

- *near* – ближній, 16-бітний покажчик (встановлюється за замовчуванням), призначений для адресації 64-кілобайтного сегмента оперативної пам'яті;
- *far* – дальній, 32-бітний покажчик, містить адресу сегмента і зсув у ньому: може адресувати оперативну пам'ять обсягом до 1 Мб;
- *huge* – це модифікатор, який розширює можливості стандартних покажчиків, дозволяючи їм вказувати на більший адресний простір.

Значення змінної-покажчика – це адреса, ціле значення без знака. Покажчик містить адресу першого байту змінної визначеного типу. Тип змінної, на яку посилається покажчик, визначає об'єм оперативної пам'яті, що виділяється змінній. Для того, щоб машинний код обробив (наприклад, прочитати або записати) значення змінної за допомогою покажчика, треба знати адресу її початкового (нульового) байта та кількість байтів, що займає ця змінна. Покажчик містить адресу нульового байту цієї змінної, а тип змінної, що адресується, визначає, скільки байтів, починаючи з адреси, визначеної покажчиком, займає це значення.

Приклад деяких можливих оголошень покажчиків:

```
int *pi;   /* - покажчик - змінна на дані типу int */
float *pf; /* - покажчик - змінна на дані типу float */
int ml [5];  /* - ім'я масиву  на 5 значень типу int;
   ml - покажчик-константа, про це йтиметься згодом */
int *m2[10]; /* m2 - ім'я масиву на 10 значень типу
     покажчик на  значення   типу int, m2 - покажчик-
       константа */
int (*m3)[10]; /* - покажчик на масив з 10 елементів
             типу int; m3 - покажчик-константа */
```

Зверніть увагу на те, що у трьох з наведених оголошень ім'я масиву є константою – покажчиком! (Про це йтиметься далі)

За допомогою покажчиків можна:

1. обробляти одновимірні та багатовимірні масиви, рядки, символи, структури і масиви структур;
2. динамічно створювати нові змінні в процесі виконання програми;
3. обробляти зв'язані структури: стеки, черги, списки, дерева, мережі;
4. передавати функціям адреси фактичних параметрів;
5. передавати функціям адреси функцій в якості параметрів.

Застосування покажчиків критикується через те, що в силу їх природи неможливо визначити, на яку змінну вказує в даний момент покажчик, якщо не повертатися до того місця, де покажчику востаннє було присвоєно значення. Це робить доведення правильності програми дещо ускладненим. Програміст, що добре володіє С, повинен насамперед знати, що таке покажчики, та вміти їх використовувати.

Основні операції над покажчиками

Мова С надає можливість використання адрес змінних за допомогою основних операцій – & та *. За допомогою основних операцій можна отримати значення адреси змінної а використовуючи непряму адресацію – одержати значення змінної за її адресою.

Призначення операцій:

& ім'я змінної – одержання адреси; визначає адресу розміщення значення змінної визначеного типу;

** ім'я-покажчика* – отримання значення визначеного типу за вказаною адресою; визначає вміст змінної, розміщеної за адресою, що міститься у даному покажчику; це – непряма адресація (інші назви – «зняття значення за покажчиком» або «розіменування»).

Оператор присвоювання значення адреси покажчику має вигляд:

Ім'я_змінної_покажчика = & ім'я змінної;

Приклад:

```
int i, *pi;    /* pi –змінна покажчик   */
pi = &i;       /* pi одержує значення адреси 'i' */
```

Операція & – визначення адреси змінної повертає адресу оперативної пам'яті свого операнда. Операндом операції & повинне бути ім'я змінної того ж типу, для якого визначений покажчик, що одержує значення цієї адреси. У вищенаведеному прикладі це тип *int*.

Операції * і & можна писати впритул до імені операнду або через пробіл.

Приклад:

&i, * pi

Непряма адресація змінної за допомогою операції * здійснює доступ до змінної за покажчиком, тобто повернення значення змінної, розташованої за адресою, що міститься у покажчику. Операнд операції * обов'язково повинен бути типу покажчик. Результат операції * – це значення, на яке вказує (адресує, посилається) операнд. Тип результату – це тип, визначений при оголошенні покажчика.

У загальному вигляді оператор присвоювання, що використовує ім'я покажчика та операцію непрямої адресації, можна представити у вигляді:

ім'я змінної * ім'я-покажчика;

де ім'я-покажчика – це змінна або константа, що містить адресу розміщення значення, необхідного для змінної лівої частини оператора присвоювання.

Приклад:

```
i= *pi; /* 'i' одержує значення, розташоване за
           адресою, що міститься в покажчику 'pi' */
```

Як і будь-які змінні, змінна pi типу покажчик має адресу і значення. Операція & над змінною типу покажчик: &pi – дає адресу місця розташування самого покажчика, pi – ім'я покажчика визначає його значення, а *pi – значення змінної, що адресує покажчик. Звичайно, усі ці значення можна надрукувати (лістинг 7.1).

Лістинг 7.1. Робота з покажчиками

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

```
main()
{
    char c = 'A';
    int i = 7776;
    int* pi = &i;
    char* pc = &c;
    printf("pi=%u,*pi=%d, &pi=%u\n", pi, *pi, &pi);
    printf("pc=%u, *pc=%c, &pc=%u\n", pc, *pc, &pc);
}
```

Результат виконання:

```
pi=1592212568,*pi=7776, &pi=1592212560
pc=1592212575, *pc=A, &pc=1592212552
```

Одне з основних співвідношень при роботі з покажчиками - це симетричність операцій адресації та непрямої адресації, що продемонстровано в прикладі.

Приклад:

```
int *pi, i = 123, j;
pi = &i; /*- присвоювання покажчику значення адреси i */
j = *pi; /* - присвоювання j вмісту за адресою pi */
```

Змінна *j* отримує вміст, розташований за адресою змінної *i*, тобто значення змінної, що адресує покажчик *pi*: *j = * pi = * &i = i;*. Два останніх вищенаведених оператора виконують те саме, що один оператор: *j = i*.

Для повного остаточного розуміння процесів, що відбувається у пам'яті при маніпуляції з покажчиками, розглянемо код наведений в лістингу 7.2.

Лістинг 7.2. Робота з покажчиками

```
#include <stdio.h>
main()
{
    int x;
    int* pi; /* px - покажчик на змінну типу int*/
```

```
    pi = &x; /* адреса змінної x заноситься в px*/
    *pi = 77;  /* число зберігається за адресою, на яку
                вказує px */
    printf("pi=%u\n *pi=%d\n &pi=%u\n", pi, *pi, &pi);
}
```

Результат виконання програми:

```
pi=3108255372
 *pi=77
 &pi=3108255360
```

Зобразимо цей приклад графічно (рисунок 7.1): програма займає область пам'яті, починаючи з адреси *0x100*, *x* знаходиться за адресою *0x102*, а *pi – 0x106*. Тоді перша операція присвоювання, коли значення *&x(0x102*) зберігається в *pi,* матиме вигляд, зображений на рисунку 7.1 зліва.

Наступну операцію, коли число 77 записується за адресою, яка знаходиться в *pi* та дорівнює *0x102* (адреса x), зображений на рисунку 7.1 справа. Запис **pi* надає доступ до вмісту комірки, на яку вказує *pi*.

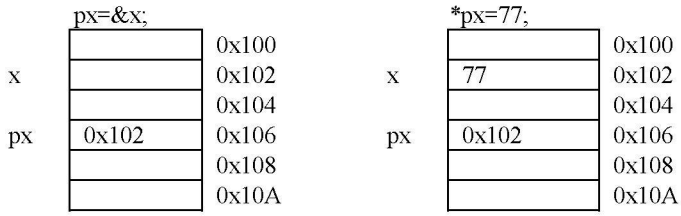


Рисунок 7.1 – Схематичне представлення значень в оперативній пам'яті

Лістинг 7.3 демонструє приклад програми виводу значень покажчика і вмісту, розташованого за адресою, що зберігає покажчик.

Лістинг 7.3. Приклад роботи з покажчиками

```
#include<stdio.h>
void main()
{
```

```
    int i = 123, * pi = &i; /* pi-покажчик на значення типу
int */
    printf("розмір покажчика pi = %d\n", sizeof(pi));
    printf("адреса розміщення покажчика pi=%u\n", &pi);
    printf("адреса змінної i = %u\n", &i);
    printf("значення покажчика pi = %u\n", pi);
    printf("значення за адресою pi = %d\n", *pi);
    printf("значення змінної i = %d\n", i);
}
```

Результат виконання програми:

```
розмір покажчика pi = 2
адреса розміщення покажчика pi = 65522
адреса змінної i = 65524
значення покажчика pi = 65524
значення за адресою pi = 123
значення змінної i = 123
```

Покажчики можна використовувати:

1. У виразах, наприклад, для одержання значень, розташованих за адресою, що зберігається у покажчику.
2. У лівій частині операторів присвоювання, наприклад:
   a. для одержання значення адреси, за якою розташоване значення змінної;
   b. для одержання значення змінної.

Якщо *pi* – покажчик цілого значення (змінної *i*), то **pi* можна використовувати в будь-якому місці програми, де можна є значення цілого типу.

Приклад:

```
int i = 123, j, *pi;
pi = &i; /*pi у лівій частині оператора присвоювання */
j = *pi + 1; /*-це еквівалентно: j = i + 1;
   pi-у виразі правої частини оператора присвоювання*/
```

Виклик значення за покажчиком можна використовувати також як фактичні параметри при звертанні до функцій.

Приклад:

```
d = sqrt ((double) *pi);/* *pi - фактичний параметр */
fscant (f, "%d", pi );   /* pi - фактичний параметр */
printf ("%d\n", *pi );        /* *pi - фактичний параметр */
```

У виразах унарні операції & і *, пов'язані з покажчиками, мають більший пріоритет, ніж арифметичні.

Приклад:

```
*px = &x;
y = 1 + *px;  /* спочатку виконується '*', потім '+'    */
              /* останній оператор еквівалентний:  y = 1 + x;    */
```

Доступ до значення через змінну-покажчик можна використовувати в операторі присвоювання там, де зазвичай застосовується ім'я змінної. Після виконання оператора: *px = &x;* цілком еквівалентними є описи наведені в прикладі.

Приклад:

Оператор:	*Його еквівалент:*	*Або:*
**px =0;*	*x = 0;*	
**px += 1;*	**px = *px + 1;*	*x = x + 1;*
*(*px)++ ;*	**px = *px + 1;*	*x = x + 1;*
*(*px)--;*	**px = *px – 1;*	*x = x - 1;*

Програма представлена в лістингу 7.4 демонструє найпростіше практичне використання покажчиків, виводячи звичайну послідовність літер алфавіту.

Лістинг 7.4. Використання покажчиків для виведення літер алфавіту

```
#include <stdio.h>
char c;  /* змінна символьного типу*/
main()
{
```

```
    char* pc; /* покажчик на змінну символьного типу*/
    pc = &c;
    for (c ='A'; c <='Z'; c++)
        printf("%3c", *pc);
}
```

Результат виконання програми:

```
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 операторі *printf("%3c",*pc)* відбувається розіменування покажчика *(*pc)* – передача у функцію значення, що зберігається за адресою, яка міститься у змінній *pc*. Щоб дійсно довести, що *pc* є псевдонімом *c*, спробуємо замінити **pc* на *c* у виклику функції – і після заміни програма працюватиме абсолютно аналогічно. Оскільки покажчики обмежені заданим типом даних, типовою серйозною помилкою їх використання буває присвоєння адреси одного типу даних покажчика іншого типу, на що компілятор видасть помилку.

Багаторівнева непряма адресація

У мові С можна використовувати багаторівневу непряму адресацію, тобто непряму адресацію на перший, другий і подальші рівні. При цьому для оголошення і звертання до значень за допомогою покажчиків можна використовувати відповідно кілька символів зірочка (*). Зірочки при оголошенні вказують на призначення імені змінної, визначаючи рівень непрямої адресації для доступу до значень через ці покажчики.

Приклад оголошення змінної і покажчиків для багаторівневої непрямої адресації:

```
int     i = 123  /* де: i - ім'я змінної  */
int    *pi = &i;  /* pi - покажчик на змінну i  */
int  **ppi = &pi; /* ppi - покажчик на покажчик на змінну pi */
int ***pppi = &ppi; /* pppi - покажчик на 'покажчик на 'покажчик на
змінну ppi' */
```

Для звертання до значень за допомогою покажчиків можна прийняти наступне правило, що жорстко зв'язує форму звертання з оголошенням цих покажчиків:

- повна кількість зірочок непрямої адресації, яка відповідає числу зірочок при оголошенні покажчика, визначає значення змінної;
- зменшення кількості зірочок непрямої адресації додає до імені змінної слово "покажчик", причому цих слів може бути стільки, скільки може бути рівнів непрямої адресації для цих імен покажчиків, тобто стільки, скільки зірочок стоїть в оголошенні покажчика.

Приклад:

---

int i, *pi=&i;
****pi - визначає значення змінної,***
pi - покажчик на змінну i.

А при звертанні до змінних можна використовувати різну кількість зірочок для різних рівнів адресації.

Приклад:

---

pi, ppi, pppi	***– 0-й рівень адресації, пряма адресація;***
****pi, *ppi, *pppi***	***– 1-й рівень непрямої адресації***
*****ppi, **pppi***	***– 2-й рівень непрямої адресації***
******pppi***	***– 3-й рівень непрямої адресації***

Таким чином, до покажчиків 1-го і вище рівнів непрямої адресації можливі звертання і з меншою кількістю зірочок непрямої адресації, аніж задано при оголошенні покажчика. Ці звертання визначають адреси, тобто значення покажчиків визначеного рівня адресації. Відповідність між кількістю зірочок при звертанні за допомогою покажчика і призначенням звертання за покажчиком для наведеного прикладу ілюструє таблиця 7.1 (де Р.н.а. – рівень непрямої адресації).

Таблиця 7.1.

Відповідність між кількістю уточнень (*) і результатом звертання за допомогою покажчика

Звертання	*Результат звертання*	*Р.н.а.*
i	значення змінної i	1
*pi pi	значення змінної, на яку вказує pi покажчик на змінну типу int, значення pi	1 0

**ppi *ppi ppi	значення змінної типу int покажчик на змінну типу int покажчик на "покажчик на змінну типу int', значення покажчика ppi	2 1 0
***pppi **pppi *pppi pppi	значення змінної типу int; покажчик на змінну типу int покажчик на 'покажчик на змінну типу int' покажчик на 'покажчик на 'покажчик на змінну типу int', значення покажчика pppi	3 2 1 0

Операції над покажчиками

Мова С надає можливості виконання над покажчиками операцій присвоювання, цілочисельної арифметики та порівнянь, тобто:

- присвоїти покажчику значення адреси даних, або нуль;
- збільшити (зменшити) значення покажчика;
- додати або відняти від значення покажчика ціле число;
- скласти або відняти значення одного покажчика від іншого;
- порівняти два покажчики за допомогою операцій відношення.

Змінній-покажчику можна надати певне значення за допомогою одного із способів:

1. присвоїти покажчику адресу змінної, що має місце в оперативній пам'яті, або нуль;

Приклад:

```
pi = &j;
pi = NULL;
```

2. оголосити покажчик поза функцією (у тому числі поза *main*()) або у будь-якій функції, додавши до нього *static*; при цьому початковим значенням покажчика є нульова адреса (NULL);
3. присвоїти покажчику значення іншого покажчика, що до цього моменту вже має визначене значення;

Приклад:

```
pi = pj;  /* це - подвійна вказівка однієї і тієї ж змінної; */
```

4. присвоїти змінній-покажчику значення за допомогою функцій *calloc*() або *malloc*() - функцій динамічного виділення оперативної пам'яті.

Розглянемо кілька простих прикладів дій над покажчиками. Зміну значень покажчика можна робити за допомогою операцій: +, ++, –, – –. Бінарні операції (+ та –) можна виконувати над покажчиками, покажчиками одного типу, тому що об'єм оперативної пам'яті для різних типів даних різний.

Наприклад, значення типу *short* займає 2 байти, а типу *float* – 4 байти. При додаванні одиниці до покажчика додається «квант пам'яті», тобто кількість байтів, яку займає одне значення відповідного типу. Для покажчика на елементи масиву це означає перехід до адреси наступного елемента масиву, а не до наступного байта. Таким чином, значення покажчика при переході між елементами масиву типу short збільшується на 2 байти, а типу *float* – на 4 байти. У мові С результат обчислення покажчиків визначається як значення типу int.

Приклад програми зміни значення покажчика на 1 квант пам'яті за допомогою операції «++» і визначення результату обчислення покажчиків продемонстрований в лістингу 7.5.

Лістинг 7.5. Операції над покажчиками

```
#include <stdio.h>
main()
{
    short shortArray[3] = { 1, 2, 3 };
    float floatArray[3] = { 1.1, 2.2, 3.3 };

    // Покажчики на перший елемент кожного масиву
    short* shortPtr = shortArray;
    float* floatPtr = floatArray;

    printf("Початкові адреси:\n");
    printf("Адреса shortPtr: %d\n", shortPtr);
    printf("Адреса floatPtr: %d\n", floatPtr);

    // Зміщення покажчиків
    shortPtr++;
    floatPtr++;

    printf("\nАдреси після збільшення покажчиків:\n");
```

```
    printf("Адреса shortPtr: %d (збільшилась на %zu
байтів)\n", shortPtr, sizeof(short));
    printf("Адреса floatPtr: %d (збільшилась на %zu
байтів)\n", floatPtr, sizeof(float));

    // Різниця між покажчиками
    int shortOffset = shortPtr - shortArray;
    int floatOffset = floatPtr - floatArray;

    printf("\nРізниця між початковими та новими адресами (у
байтах):\n");
    printf("Для shortPtr: %d квант ОП типупу short\n",
shortOffset);
    printf("Для floatPtr: %d квант ОП типупу float\n",
floatOffset);
}
```

Результат виконання програми:

```
Початкові адреси:
Адреса shortPtr: 360544930
Адреса floatPtr: 360544916

Адреси після збільшення покажчиків:
Адреса shortPtr: 360544932 (збільшилась на 2 байтів)
Адреса floatPtr: 360544920 (збільшилась на 4 байтів)

Різниця між початковими та новими адресами (у байтах):
Для shortPtr: 1 квант ОП типупу short
Для floatPtr: 1 квант ОП типупу float
```

Розглянемо приклад програми (лістинг 7.6) для виведення значень індексів елементів масивів, адрес першого байта оперативної пам'яті для їх розміщення та значень елементів масивів. В мові С ім'я масиву еквівалентно адресі його нульового елемента, тобто $x = \&x[0]$. Покажчики *pi* і *pf* спочатку містять значення адрес нульових елементів масивів, а при виведенні складаються з *i*-номером елемента масиву, визначаючи адресу *i*-елемента масиву. Для одержання адрес елементів масивів у програмі використовується додавання покажчиків-констант *x* та *y*, та змінних-покажчиків *pi* і *pf* з цілим значенням змінної *i*. Зміна

адрес у програмі дорівнює кванту оперативної пам'яті для даних відповідного типу, для цілих – 2 байти, для дійсних – 4 байти.

Лістинг 7.6. Виведення значень індексів елементів масивів

```
#include<stdio.h>
void main(){
    int x[4], * pi = x, i;
    float y[4], * pf = y;
    printf("\nномер елемента адреси елементів масивів : \n
i    pi + i      x + i      &x[i]          pf + i      y +
i     &y[i]\n");
    for (i = 0; i < 4; i++)
        printf("  %d : %d %d %6d     %6d %6d %6d\n",
i, pi + i, x + i, &x[i], pf + i, y + i, &y[i]);
}
```

Результат виконання програми:

```
номер елемента адреси елементів масивів:
 i     pi + i          x + i          &x[i]          pf + i          y + i
&y[i]
 0 : 1605827096 1605827096 1605827096    1605827208 1605827208
1605827208
 1 : 1605827100 1605827100 1605827100    1605827212 1605827212
1605827212
 2 : 1605827104 1605827104 1605827104    1605827216 1605827216
1605827216
 3 : 1605827108 1605827108 1605827108    1605827220 1605827220
1605827220
```

Мовою С можна визначити адреси нульового елемента масиву *x* як *x* або *&x[0]*: *x = &x[0]*. Краще і стисло використовувати просто *x* – це базова адреса масиву. Ту саму адресу елемента масиву можна представити у вигляді *x + 2 = &x[2]; x + i = &x[i]*.

Приклад:

(x + 0) == *x == x[0]** - ***значення нульового елемента масиву x;
(x + 2) == x[2]** - ***значення другого елемента масиву x;
(x + i) == x[i]** - ***значення i-го елемента масиву x.

А операції над елементами масиву x можна представити у вигляді:
****x + 2== x[0] +2; *(x + i) - 3 == x[i] - 3;***

Проблеми, пов'язані з покажчиками

Проблеми, пов'язані з покажчиками, виникають при некоректному використанні покажчиків. Усі застереження щодо некоректного використання покажчиків відносяться до мови С так само, як і до багатьох інших низькорівневих мов програмування. Некоректним використанням покажчиків може бути:

- спроба працювати з неініціалізованим покажчиком, тобто з покажчиком, що не містить адреси, що виділена змінній;
- втрата значення покажчика через присвоювання йому нового значення до звільнення оперативної пам'яті, яку він адресує;
- незвільнення оперативної пам'яті, що виділена за допомогою функції *malloc()*.

Запит на виділення оперативної пам'яті з купи робиться за допомогою функцій *calloc()* та *malloc()*. Звільнення оперативної пам'яті робиться за допомогою функції *free()*.

При оголошенні покажчика на скалярне значення будь-якого типу оперативна пам'ять для значення, що адресується, не резервується. Виділяється тільки оперативна пам'ять для змінної-покажчика, але покажчик при цьому не має значення. Якщо покажчик має специфікатор static, то ініціюється початкове значення покажчика що дорівнює нулю.

Приклад ініціалізації покажчиків нульовими значеннями при їх оголошенні:

```
static int *pi, *pj; /* pi = NULL; pj= NULL; */
```

Розглянемо приклад, що містить грубу помилку: спробу працювати з непроініціалізованим покажчиком.

Приклад:

```
int *x;      /* змінній-покажчику 'x' виділена ОП, але  'x'
                не містить значення адреси ОП для змінної */
*x = 123;  /* - груба помилка!  */
```

Таке присвоювання помилкове, тому що змінна-покажчик *x* не має значення адреси, за яким має бути розташоване значення змінної. Компілятор видасть попередження. При цьому випадкове (непроініціалізоване) значення покажчика (сміття) може бути неприпустимим адресним значенням! Наприклад, воно може збігатися з адресами розміщення програми або даних користувача, або даних операційної системи. Запис цілого числа 123 за такою адресою може порушити працездатність програми користувача або самої операційної системи.

Виправити ситуацію можна за допомогою функції *malloc()*. Форма звертання до функції *malloc()* наступна:

ім'я-покажчика = (тип-покажчика) malloc (об'єм - ОП) ;

де ім'я-покажчика – ім'я змінної-покажчика; тип-покажчика - тип значення, що повертається функцією malloc; об'єм-ОП – кількість байтів оперативної пам'яті, що виділяються змінній, яка адресується.

Приклад:

---

x = (int *) malloc (sizeof (int));

При цьому з купи виділяється 2 байти оперативної пам'яті для цілого значення, а отримана адреса його розміщення заноситься в змінну-покажчик *x*. Значення покажчика гарантовано не збігається з адресами, що використовуються іншими програмами, у тому числі програмами ОС. Параметр функції *malloc* визначає об'єм оперативної пам'яті для цілого значення за допомогою функції *sizeof(int)*. Запис (int *) означає, що адреса, що повертається функцією *malloc()*, буде розглядатися як покажчик на змінну цілого типу. Це операція приведення типів. Таким чином, помилки не буде у випадку використання операторів.

Приклад:

---

int *x; /* x - ім'я покажчика, він одержав ОП */
x = (int *) malloc (sizeof(int));
/* Виділена ОП цілому значенню, на яке вказує 'x' */
****x = 123; /* змінна, на яку вказує 'x', одержала значення 123 */***
Повернення (звільнення) ОП у купі виконує функція free(). Її аргументом є ім'я покажчика, що посилається на пам'ять, що звільняється. Наприклад:
free (x);

Щоб уникнути помилок при роботі з функціями не слід повертати як результат їхнього виконання адреси локальних змінних функції. Оскільки при виході з функції пам'ять для всіх локальних змінних звільняється, повернута адреса може бути використаною системою й інформація за цією адресою може бути невірною. Можна повернути адресу оперативної пам'яті, що виділена з купи.

Одна з можливих помилок – подвійна вказівка на дані, розташовані у купі, і зменшення об'єму доступної оперативної пам'яті через незвільнення отриманої оперативної пам'яті. Це може бути для будь-якого типу даних, у тому числі для скаляра або масиву.

Приклад:

```
/* Виділення оперативної пам'яті динамічним змінним x, y и z: */
    int *x = (int *) malloc ( sizeof(int)),
    *y = (int *) malloc ( sizeof(int)),
    *z = (int *) malloc ( sizeof(int));
 /* Ініціалізація значення покажчиків x, y, z;*/
  *x = 14; *y = 15; *z = 17;
 /*Динамічні змінні одержали конкретні цілі значення*/
 y=x; /* груба помилка - втрата покажчика на динамічну
      змінну в без  попереднього звільнення її ОП */
```

У наведеному прикладі не оголошуються імена змінних, а використовуються лише покажчики. Після виконання операції *y = x*, покажчики *x* і *y* вказують на ту саму область оперативної пам'яті, пов'язану зі змінною **x*, тобто **x = 14*, а **y = 15*. При цьому 2 байти пам'яті, виділені для змінної, на яку раніше вказував *y (*y)*, стають недоступними (втраченими), оскільки адреса, що зберігалася в *y*, була замінена адресою *x*. У результаті ці 2 байти в купі залишаються зайнятими, зменшуючи її доступний обсяг. Це призводить до скорочення доступної оперативної пам'яті, що небажано і потребує уникнення. Щоб уникнути такої помилки треба попередньо звільнити оперативну пам'ять, виділену змінній **y*, а потім виконати присвоювання значення змінній *y*.

Приклад:

```
free (y);  /* звільнення ОП, виділеної змінної '*y' */
y = x;   /* присвоювання нового значення змінній 'y' */
```

Грубою помилкою є присвоєння змінній-покажчику значення адреси в операторі оголошення.

Приклад:

```
int *x = 12345;
```

В даному прикладі *12345* – це константа цілого типу, а значенням покажчика *x* може бути тільки адреса, тобто байт в оперативній пам'яті. Тому компілятор при цьому видасть повідомлення про помилку.

В наведеному нижче прикладі присвоювання не викличе помилки.

Приклад:

```
int a[5], *x = a;
```

Масиви

Основні поняття

Між покажчиками і масивами існує тісний взаємозв'язок. Будь-яка дія над елементами масивів, що досягається індексуванням, може бути виконана за допомогою покажчиків (посилань) і операцій над ними. Програма виконана з використанням покажчиків буде виконуватись швидше, але для розуміння вона є складнішою.

Покажчики частіше використовуються з масивами, рідше зі скалярними змінними. Покажчики дають можливість застосовувати адреси приблизно так, як це робиться на машинному рівні. Це дозволяє ефективно організувати роботу з масивами.

Для роботи з масивом необхідно:

1. визначити ім'я масиву, кількість елементів масиву, його вимірність;
2. виділити оперативну пам'ять для його розміщення.

У мові С можна використовувати статичні та динамічні масиви даних будь-якого типу. Для статичних масивів оперативна пам'ять виділяється в стеку. Для динамічних оперативна пам'ять виділяється з

купи в процесі виконання програми, за допомогою функцій *malloc*() або *calloc*().

Динамічні змінні використовують, якщо розмір масиву невідомий до початку роботи програми і визначається в процесі її виконання, наприклад за допомогою обчислення або введення.

Розмір статичного масиву визначається при оголошенні, оперативна пам'ять виділяється до початку виконання програми, ім'я масиву є покажчиком, кількість елементів масиву визначається явно або неявно, при ініціалізації елементів масиву.

Приклад:

```
int a[] = { 1, 2, 3 };
int b[10];
```

Розмір динамічних масивів визначається у процесі виконання програми, оперативна пам'ять для них запитується і виділяється динамічно, з купи, ім'я покажчика на масив – це змінна.

Приклад:

```
int   *m1 = (int * ) malloc ( 100 * sizeof (int)) ;
float *m2 = (float * ) malloc ( 200 * sizeof (float)) ;
```

де *m1* – змінна-покажчик на масив 100 значень типу *int*;

m2 – змінна-покажчик на масив 200 значень типу *float*.

Звільнення виділеної оперативної пам'яті відбувається за допомогою функції

free (покажчик-змінна) ;

Приклад:

```
free(m1);
free(m2);
```

Масиви можуть бути одновимірними і багатовимірними, при цьому визначається кількість елементів й оперативна пам'ять для всього масиву.

Масиви також можуть бути вільні (спеціальні двовимірні), при цьому визначається кількість рядків і кількість елементів кожного рядка, і оперативна пам'ять запитується і виділяється для елементів кожного рядка масиву в процесі виконання програми. При використанні вільних масивів використовують масиви покажчиків.

Розмір масиву можна не вказувати. В цьому разі необхідно вказати порожні квадратні дужки, якщо при оголошенні ініціалізується значення його елементів.

Приклад:

```
int a[] = {1, 2, 3};
char b[] = "Відповідь:";
```

Масив може бути оголошено глобально, тоді в функції робиться посилання на масив.

Приклад:

```
int a[5];  /* оголошення глобального масиву  */
main ()
{
 extern int a[];  /*посилання на глобальний масив */
}
```

В усіх можливих оголошеннях масиву ім'я – це покажчик-константа! Для формування динамічного масиву може використовуватися тільки ім'я покажчика на масив – це покажчик-змінна.

Звертання до елементів масивів *m1* і *m2* відбувається з використанням імені масиву та індексу елемента.

Приклад:

```
m1[i]
m2[j]
```

Стандартних засобів для копіювання масивів у мові С немає. Але можна зробити копію масива поелементно або сумістити масиви в оперативній пам'яті, давши їм практично те саме ім'я.

Приклад:

```
int *m1 = (int *) malloc(100 * sizeof(int));
int *m2 = (int *) malloc(100 * sizeof(int));
```

Для копіюваня елементів одного масиву в інший можна використати оператор циклу.

Приклад:

```
for (i = 0; i < 100; i++ ) m2[i] = ml [i];
```

Замість *m2[i] = m1 [i];* можна використовувати покажчики.

Приклад:

```
*m2++ = *ml++;
*(m2 + i) = *(ml + i) ;
```

Маніпулюючи покажчиками, ми можемо змусити два масиви працювати з однією й тою самою областю пам'яті. Це досягається шляхом звільнення пам'яті, зайнятої одним масивом, та присвоєння його покажчику адреси іншого масиву. Однак такий підхід не завжди коректний, особливості коли масиви розташовані в різних сегментах пам'яті, наприклад, у стеку та купі.

Приклад:

```
free(m2);    m2 = ml ;
```

Операції *free(m2)* та *m2 = ml* призводять до того, що покажчик *m2* тепер вказує на початок масиву *m1*. З цього моменту будь-які зміни через *m2* мають вплив на дані в m1. В лістингу 7.7 розглянуто приклад використання спільної області пам’яті двома покажчиками.

Лістинг 7.7. Використання спільної області пам’яті двома покажчиками

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

```
void main()
{
    int* m1 = (int*)malloc(5 * sizeof(int));
    int* m2 = (int*)malloc(5 * sizeof(int));

    for (int i = 0; i < 5; i++)
    {
        m1[i] = 2 * i;
        m2[i] = m1[i] - 4;
    }
    printf("\n m1 = ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", m1[i]);
    }
    printf("\n m2 = ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", m2[i]);
    }
    free(m2);
    m2 = m1;
    printf("\n\n m1 = ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", m1[i]);
    }
    printf("\n m2 = ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", m2[i]);
    }
}
```

Результат виконання програми:

```
m1 = 0 2 4 6 8
m2 = -4 -2 0 2 4

m1 = 0 2 4 6 8
m2 = 0 2 4 6 8
```

Приклад розглянутий в лістингу 7.7 незавжди можна використовувати, наприклад, коли масиви розташовані в різних типах оперативної пам'яті: один – у стеку, інший – у купі.

Приклад:

```
int *m1 = (int *) malloc(100* sizeof(int));
int m2[100] ;
```

У вищенаведеному прикладі m1 – пакажчик-змінна, і масив *m1* розташований у купі, m2 – покажчик-константа, і масив *m2* розташований у стеку. У цьому випадку код *m2 = m1*; є помилковим, тому що *m2* – це покажчик-константа. Але застосувавши оператор *free(m1)* код *m2 = m1*; є припустимим.

Для доступу до частин масивів і до елементів масивів використовується індексування (індекс). Індекс – це вираз, що визначає адресу значення або групи значень масиву, наприклад адреса значень чергового рядка двовимірного масиву. Індексування можна застосовувати до покажчиків-змінних на одновимірний масив – так само, як і до покажчиків-констант.

Індексний вираз обчислюється шляхом додавання адреси початку масиву з цілим значенням для одержання адреси необхідного елемента або частини масиву. Для одержання значення за індексним виразом до результату – адреси елемента масиву застосовується операція непрямої адресації (*), тобто одержання значення за заданою адресою. Відповідно до правил обчислення адреси цілочисельний вираз, що додається до адреси початку масиву, збільшується на розмір кванта оперативної пам'яті типу даних, що адресується покажчиком.

Оголошення та звертання в одновимірних масивах

Форма оголошення одновимірного масиву з явною вказівкою кількості елементів масиву:

тип ім'я_масиву [кількість-елементів-масиву];

Звертання до елементів одновимірного масиву в загальному випадку можна представити індексуванням, тобто у вигляді

ім'я-масиву [вираз];

де ім'я-масиву – покажчик-константа;

вираз – індекс, число цілого типу, що визначає зсув – збільшення адреси заданого елемента масиву щодо адреси нульового елемента масиву.

Елементи одновимірного масиву розташовуються в оперативній пам'яті підряд: нульовий, перший тощо.

Приклад:

```
int a[10];
int *p = a;        /* - p одержує значення  a */
```

При цьому компілятор виділяє масив в стеку оперативної пам'яті розміром *(sizeof(Type) * розмір-масиву)* байтів.

У вищенаведеному прикладі це 2 * 10 = 20 байтів. Причому *a* - покажчик-константа, адреса початку масиву, тобто його нульового елемента. Змінній-покажчику *p* можна присвоїти значення різними способами.

Приклад:

```
p = a;
p = &a[0];
p = &a[i];
```

де *&a[i] дорівнює (a + i)* – адреса *i*-того елемента масиву.

Відповідно до правил перетворення типів значення адреси *i*-того елемента масиву на машинному рівні формується таким чином:

```
&a[i]= a + i * sizeof(int);
```

Наведемо приклад еквівалентних співвідношень звернення до елементів масиву за допомогою покажчика.

Приклад:

```
&a ⇔ a+0 ⇔ &a[0] – адреса a[0] - нульового елемента масиву;
a+2 ⇔ &a[2]– адреса a[2] - другого елементи масиву;
a+i ⇔ &a[i] – адреса a[i] - i-го елемента масиву;
*a ⇔ *(a+0) ⇔ *(&a[0]) ⇔ a[0] – значення 0-ого елемента масиву;
*(a + 2) ⇔ a[2] – значення a[2] - другого елементи масиву;
*(a + i) ⇔ a[i] – значення a[i] - i-го елемента масиву;
```

a + 2 ⇔ a[0] + 2 – сума значень a[0] і 2.

Якщо *p* є покажчиком на масив *a*, то *a* та *p* будуть взаємозамінними.

Приклад:

---

p ⇔ &a[0] ⇔ a + 0;
p+2 ⇔ &a[2] ⇔ a + 2;
****(p + 2) ⇔ (&a[2]) ⇔ a[2] ⇔ p[2];***
****(p + i) ⇔ (&a[i]) ⇔ a[i] ⇔ p[i];***

Для *a* та *p* еквівалентні всі звертання до елементів *a*.

Приклад:

---

a[i], *(a+i), *(i+a), i[a], та
p[i], *(p+i), *(i+p), i[p]

В лістингу 7.8 розглянуто приклад звернення до елементів масиву за допомогою покажчика.

Лістинг 7.8. Приклад звернення до елементів масиву за допомогою покажчика

```
#include<stdio.h>

void main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    int* p = arr;
    printf("%d\n", arr[3]);
    printf("%d\n", *(arr + 3));
    printf("%d\n", 3[arr]);
    printf("%d\n", *(p + 3));
    printf("%p\n", &arr);
    printf("%p\n", arr);
}
```

Результат виконання програми:

```
4
4
4
4
000000B5929DF838
000000B5929DF838
```

Оголошення та звертання до багатовимірних масивів

У мові С багатовимірний масив розглядається як набір масивів меншої розмірності. Наприклад, двовимірний масив являє собою набір одновимірних масивів (його рядків), тривимірний масив складається з набору матриць, матриці є набором рядків, а рядок – це набір елементів одновимірного масиву.

Елементи масиву в пам'яті розміщуються лінійно, починаючи з елемента з найменшим індексом. Для двовимірних масивів елементи розташовуються спочатку по першому індексу (рядках), потім по другому (стовпцям) і так далі.

Для звертання до елементів багатовимірного масиву можна використовувати декілька індексів (індексних виразів), відповідно до вимірності масиву.

ім'я-масиву [індекс1][індекс2] ...

Наприклад, для звертання:

- до одновимірного масиву можна використовувати одно-індексний вираз (індекс);
- до двовимірного – одно- або двоіндексний вираз;
- до тривимірного – одно-, дво- або трьохіндексний вираз тощо.

При звертанні до багатомірних масивів одержання значення елемента масиву можливо тільки після визначення адреси елемента масиву, тобто при повній кількості індексів. При цьому обчислюються індексні вирази зліва на право, і доступ до значення виконується після обчислення останнього індексного виразу.

Двовимірний масив оголошується з використанням двох індексів типу *int*.

Приклад:

int a[m][n] ;

Масив *a[m][n]* складається з *m* одновимірних масивів (рядків), у кожному з яких утримується *n* елементів (стовпців). При роботі з цим двовимірним масивом можна використовувати одно- або двоіндексний вираз.

Якщо робота з масивом відбувається з використанням двох індексів *a[i][j],* це звернення до елемента і-рядка, j-стовпця масиву, де визначається адреса елемента масиву і вилучається його значення. Якщо робота з масивом відбувається з використанням одного індексу *a[i],* то визначається адреса одновимірного масиву (адреса початку і-рядка масиву). Ім'я масиву *a* – не містить індексу і визначає адресу масиву, його нульового елемента. Таким чином, звертання до двовимірних масивів за допомогою імені і тільки одного індексу визначає покажчик на початок відповідного рядка масиву (адреси його нульового елемента).

Приклад:

a[0] ⇔ &a[0][0] ⇔ a+0*n*sizeof(int);
a[1] ⇔ &a[1][0] ⇔ a+1*n*sizeof(int);
a[i] ⇔ &a[i][0] ⇔ a+i*n*sizeof(int);

Розглянемо приклад де задано матрицю (двовимірний масив) *a*. Необхідно вивести на екран елементи головної діагоналі, першого рядка і першого стовпця, застосувавши для цього покажчики (лістинг 7.9).

Лістинг 7.9. Приклад звернення до елементів двовимірного масиву за допомогою покажчиків

```
#include<stdio.h>
void main()
{
    int a[3][3] = { {10,20,30},
                    {40,50,60},
                    {70,80,90} };
   int* pa[3] = { a[0], a[1], a[2] };
```

```
    int* p = a[0];
    printf("Елементи головної діагоналі ");
    for (int i = 0; i < 9; i += 4)
       printf("%4d", *(p + i)); printf("\n");
    printf("Елементи першого рядка ");
    for (int i = 0; i < 3; i++)
       printf("%4d", p[i]); printf("\n");
    printf("Елементи першого стовпця ");
    for (int i = 0; i < 3; i++)
       printf("%4d", *pa[i]);  printf("\n");
}
```

Результат виконання програми:

```
Елементи головної діагоналі   10  50  90
Елементи першого рядка   10  20  30
Елементи першого стовпця   10  40  70
```

Оголошення тривимірного масиву відбувається з використанням трьох індексів.

Приклад:

int a[k][m][n];

В наведеному прикладі *k* – кількість матриць з m рядками і n стовпцями; *m* – кількість рядків (одновимірних масивів) у матриці; *n* – кількість стовпців (елементів у рядку) матриці.

Звернення до елементів тривимірного масиву можна виконувати:

a[l][i][j] – містить 3 індекси, які використовується для звертання до елемента l-матриці, i-рядка, j-стовпця масиву, при цьому обчислюються індексні вирази, визначається адреса елемента масиву і вилучається його значення;

a[k][i] – визначає одновимірний масив – адреса початку i-рядка; k - матриці;

a[k] – визначає двовимірний масив – адреса початку k - матриці, тобто нульового елемента його нульового рядка;

a – адреса початку масиву, нульового елемента нульового рядка нульової матриці.

Приклад:

```
int b[3][4][5];
int i, *ip, *ipp;
i = b[0][0][1];
ip = b[2][0];
ipp = b[2];
```

В наведеному прикладі *ip*, *ipp* – покажчики на значення типу *int*.

Після виконання коду *ip = b[2][0];* *ip* буде покажчиком на елемент 0-рядка 0-го стовпця 2-гої матриці масиву, тобто *b[2][0][0]*.

Після виконання коду *ipp = b[2];* *ipp* адресує 0-й рядок 2-ї матриці масиву, тобто містить адреса *b[2][0][0]*.

Для звертання до багатовимірного масиву *a* можна використовувати співвідношення.

Приклад:

```
&a ⇔ a ⇔ &a[0][0] ⇔ *a
    адреса a[0][0] - елемента 0-ого рядка 0-ого стовпця масиву a;
**a ⇔ *(&a[0][0]) ⇔ a[0][0]
    значення елемента нульового рядка нульового стовпця масиву a;
a[i] ⇔ (a + i) ⇔ *(a + i) ⇔ &a[i][0]
    адреса елемента i-рядка 0-стовпця;
*a[i] ⇔ **(a + i) ⇔ *(&a[i]) ⇔ a[i][0]
    значення 0-го елемента i-рядка;
a[i][j] ⇔ *(*(a + i) + j) ⇔ *(a[i] + j) ⇔ a[i][j]
    значення елемента i-рядка j-стовпця масиву a;
де:
(a + i) ⇔ *(a + i) ⇔ a[i]
    адреса 0-го елемента i-рядка = &a[i][0];
(*(a + i) + j)
    адреса j-елемента i-рядка = &a[i][j];
*(*(a + i) + j)
    значення j-елемента i-рядка = a[i][j].
```

Значення адреси початку i-рядка (адреси 0-елемента i-рядка) на машинному рівні формується у виді:

a[i] = a + i == (a+i*n*sizeof(int))

де n – кількість значень в одному рядку.

Таким чином, адреса (i+1)-рядка знаходяться на відстані від i-рядка на *(n*sizeof(int))* байтів, тобто на відстань одного рядка масиву.

Вираз *a[i][j]* компілятор С переводить в еквівалентний вираз **((*a + i) + j)*, але запис *a[i][j]* більш традиційний у математиці і більш наочний.

До елементів двовимірного масиву можна звернутися і за допомогою скалярного покажчика на масив.

Приклад:

int a[m][n], *p = a;
****(p+i*n+j) - значення j - елемента i-рядка ;***

де: n - кількість елементів у рядку;
i*n + j – змішання a[i][j] – елемента відносно початку масиву a.

В лістингу 7.10. наведено приклад роботи з двовимірним масивом за допомогою покажчиків, де звернення до елементів максиву здійснюється за допомогою **(p+i*n+j)*.

Лістинг 7.10. Робота з двовимірним масивом за допомогою покажчиків

```
#include<stdio.h>
#include <time.h>
void main()
{
    srand(time(NULL));
    int* a, n=5, m=5;
    a = (int*)malloc(n * m * sizeof(int));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            *(a + i * m + j) = rand() % 11;
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            printf("%5d ", *(a + i * m + j));
        printf("\n");
    }
    free(a);
}
```

Результат виконання програми:

```
9   7   1   4   1
10  7   4   6   9
10  8   0   10  0
3   5   4   8   1
0   8   2   3   7
```

Рядки

Основні відомості про представлення рядків

Символьний рядок представляє собою набір з одного або більше символів. В мові С немає спеціального типу даних, який можна було б використовувати для опису рядків. Замість цього рядки представляються у вигляді масиву елементів типу *char*. Це означає, що символи рядка розташовуються в пам'яті в сусідніх комірках, по одному символу в комірці.

Рисунок 7.2 – Представлення рядка у вигляді масиву символів

В мові С використовується так звана концепція «zero termination string». Це означає, що для позначення закінчення кожного рядка (масиву типу char) використовується нуль-символ ('\0').

Нульовий символ – це не цифра 0, він не виводиться на друк і в таблиці символів ASCII має номер 0. Наявність нульового символу передбачає, що кількість комірок масиву повинна бути принаймні на одну більше, ніж число символів, які необхідно розміщувати в пам'яті.

Приклад:

```
char str[10];
```

В прикладі наведено рядок розміром 10 символів, але насправді він може містити максимум 9 символів. Існують різні методи ініціалізації символьних рядків.

Приклад:

```
char str1[]= "ABCdef";
char str2[]={'A', 'B', 'C', 'd', 'e', 'f', '\0'};
char str3[100];
gets(str3);
char str4[100];
scanf("%s", str4);
```

Усі рядки-константи в програмі, навіть якщо вони записані однаково, зберігаються за окремими адресами в статичній пам'яті. Для кожного такого рядка створюється сталий покажчик, що вказує на його перший символ. Фактично, рядок-константа є виразом типу «покажчик на char» із фіксованим значенням – адресою першого символу.

Вираз *p="ABC"*, де *p* – покажчик на *char*, встановлює покажчик *p* на символ *'A'*; значенням виразу **(p+1)* є символ *'B'*. Елементи рядків доступні через покажчики на них, тому будь-який вираз типу «покажчик на *char*» можна вважати рядком.

Слід пам'ятати, що рядок виду *"x"* не є тим самим, що й символ *'x'*. Перша відмінність: *'x'* – це об'єкт основного типу даних у мові С (типу char), тоді як *"x"* – це об'єкт похідного типу (масиву елементів типу char). Друга відмінність: "x" фактично складається з двох символів – символу 'x' і нуль-символу *('\0')*.

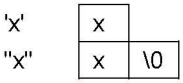


Рисунок 7.2 – Різниця між представленням 'x' та "x"

Функції роботи з рядками

Функції введення рядків.

Прочитати рядок із стандартного потоку введення можна за допомогою функції *gets*(). Функція читає символи до тих пір, поки їй не зустрінеться символ нового рядка '\n', який генерується натисканням клавіші ENTER. Функція зчитує всі символи до нового рядка, додаючи до них нульовий символ '\0' в кінець рядка.

Синтаксис :

```
char *gets(char *buffer);
```

Для читання рядків із стандартного потоку введення можна використовувати також функцію *scanf*() зі специфікатором формату *%s*. Основна відмінність між *scanf*() і *gets*() полягає у способі визначенні досягнення кінця рядка, *scanf*() призначена для читання слова, а не рядка. Функція *scanf()* може використовуватися в двох варіантах. У кожному випадку читання рядка починається з першого непорожнього символу. При використанні формату *%s* рядок зчитується до першого порожнього символу (пробілу, табуляції або символу нового рядка), але цей символ не включається. Якщо задати розмір поля, наприклад, *%10s*, то *scanf()* зчитує не більше 10 символів або до першого порожнього символу.

Функції виведення рядків.

Для виведення рядків можна використовувати функції *puts*() і *printf*(). Синтаксис функції *puts*():

```
int puts(char *string);
```

Ця функція виводить всі символи рядка у стандартний потік виведення. Виведення завершується переходом на наступний рядок.

Функція *puts*() автоматично додає символ нового рядка (\n) після виведення. Функція printf() є більш універсальною, яка використовується для форматованого виведення.

Стандартна бібліотека мови програмування С містить клас функцій для роботи з рядками, і всі вони починаються з літер *str*. Для того, щоб використовувати одну або декілька функції необхідно підключити файл *string.h*.

```
#include<string.h>
```

Визначення довжини рядка.

Для визначення довжини рядка використовується функція *strlen*(). Синтаксис:

```
size_t strlen(const char *s);
```

Функція strlen() повертає довжину рядка *s*, при цьому завершуючий нульовий символ не враховується.

В лістингу 7.11 наведено приклад виведення рядка в консоль та визначення довжини рядка.

Лістинг 7.11. Робота з рядками

```
#include<stdio.h>
void main()
{
```

```
    char s[100] = "Чого Івась не навчиться, того й Іван не
знатиме";
    puts(s);
    int len;
    len = strlen(s);
    printf("Розмір рядка = %d\n", len);
}
```

Результат виконання програми:

```
Чого Івась не навчиться, того й Іван не знатиме
Розмір рядка = 47
```

Копіювання рядків.

Оператор присвоювання для рядків не визначений. Тому, якщо *s1* і *s2* – символьні масиви, то неможливо скопіювати один рядок в інший за допомогою оператора присвоювання.

Приклад:

```
char s1[100];
char s2[100];
s1 = s2; /* помилка */
```

Щоб скопіювати один рядок в інший необхідно викликати функцію копіювання рядків *strcpy*().

Синтаксис:

```
char *strcpy(char *dest, const char *src);
```

Функція копіює посимвольно значення рядка *s2* в *s1*, включаючи завершуючий нуль-символ.

Приклад:

```
strcpy(s1, s2);
```

Для копіювання рядків можна використовувати і функцію *strncpy*(), яка дозволяє обмежувати кількість символів, що копіюються.

Синтаксис:

char *strncpy(char *dest, const char *src, size_t maxlen);

Приклад:

strncpy(s1, s2, 10);

Наведений оператор скопіює 10 символів із рядка *s2* в рядок *s1*. Якщо символів в рядку *s2* менше, ніж вказане число символів, що копіюються, то байти, що не використовуються встановлюються рівними нулю.

Функції роботи з рядками, в імені яких міститься додаткова літера *n* мають додатковий числовий параметр, що певним чином обмежує кількість символів, з якими працюватиме функція.

В лістингу 7.12 наведено приклади роботи функцій *strcpy(), strncpy()*.

Лістинг 7.12. Використання функцій *strcpy(), strncpy()*

```
#include<stdio.h>
#include <string.h>
void main()
{
char s1[100] = "Чого Івась не навчиться, того й Іван не
знатиме";
    char s2[100] = "Пан з паном, а Іван з Іваном";
    char s3[100] = "Хто про Хому, а він про Ярему";
    printf("Вхідні рядки:\n");
    puts(s1);
    puts(s2);
    puts(s3);
    printf("Вихідні рядки:\n");
    strcpy(s1, s2);
    puts(s1);
    strncpy(s2, s3, 12);
    puts(s2);
    puts(s3);
}
```

Результат виконання програми:

```
Вхідні рядки:
Чого Івась не навчиться, того й Іван не знатиме
Пан з паном, а Іван з Іваном
Хто про Хому, а він про Ярему
Вихідні рядки:
Пан з паном, а Іван з Іваном
Хто про Хому, а Іван з Іваном
Хто про Хому, а він про Ярему
```

Конкатенація рядків.

Конкатенація двох рядків означає їх об'єднання, при цьому створюється новий, більш довгий рядок. Для цього призначена функція *strcat()*.

Синтаксис:

```
char *strcat(char *dest, const char *src);
```

Приклад:

```
char first[]= "Один ";
strcat(first, "два три чотири!");
```

В наведеному прикладі функція *strcat()* перетворить рядок *first* в рядок "Один два три чотири".

При викликанні функції *strcat(s1,s2)* потрібно впевнитися, що перший аргумент типу char * ініціалізований і має достатньо місця щоб зберегти результат. Якщо *s2* адресує нульовий рядок, то оператор *strcat(s1,s2)* перезапише рядок *s1*, викликавши помилку.

Функція *strcat()* повертає адресу рядка результату (що співпадає з її першим параметром), що дає можливість виконувати вкладення декількох викликів функцій.

Приклад:

```
strcat(strcat(s1, s2), s3);
```

Наведений приклад можна виконати почерговим викликом двох функцій *strcat()*.

Приклад:

strcat(s1,s2);
strcat(s1,s3);

В лістингу 7.13 наведено приклади роботи функції *strcat()*.

Лістинг 7.13. Використання функції *strcat()*

```
#include<stdio.h>
#include <string.h>
void main()
{
    char s1[200] = "Чого Івась не навчиться, того й Іван не
знатиме";
    char s2[100] = "Пан з паном, а Іван з Іваном";
    char s3[100] = "Хто про Хому, а він про Ярему";
    printf("Вхідні рядки:\n");
    puts(s1); puts(s2); puts(s3);
    printf("Вихідні рядки:\n");
    strcat(s2, s1);
    puts(s2);
    printf("----------------\n");
    strcat(strncat(s1, s2, 4), s3);
    puts(s1);
}
```

Результат виконання програми:

```
Вхідні рядки:
Чого Івась не навчиться, того й Іван не знатиме
Пан з паном, а Іван з Іваном
Хто про Хому, а він про Ярему
Вихідні рядки:
Пан з паном, а Іван з ІваномЧого Івась не навчиться, того й Іван не
знатиме
----------------
```

Порівняння рядків. Функція *strcmp()* призначена для порівняння двох рядків. Синтаксис функції:

```
int strcmp(const char *s1, const char*s2);
```

Функція *strcmp()* порівнює рядки *s1* і *s2* і повертає значення типу int, яке дорівнює 0, якщо *s1* і *s2* ідентичні (рядки рівні); значення менше 0, якщо *s1* лексикографічно менший за *s2*; значення більше 0, якщо *s1* лексикографічно більший за *s2*.

В лістингу 7.14 наведено приклади роботи функції *strcmp()*.

Лістинг 7.14. Використання функції *strcmp()*

```
#include<stdio.h>
#include <string.h>
void main()
{
    char str1[] = "Олесь";
    char str2[] = "Орест";
    int result = strcmp(str1, str2);

    if (result < 0) {
        printf("лексикографічно %s < %s \n", str1, str2);
    }
    else if (result > 0) {
        printf("лексикографічно %s > %s \n", str1, str2);
    }
    else {
        printf("%s і %s лексикографічно рівні\n", str1,
str2);
    }
}
```

Результати виконання програми:

```
лексикографічно Олесь < Орест
```

Методи сортування масивів

Сортування масиву – один із найпоширеніших процесів обробки даних, що дозволяє впорядковувати об'єкти у заданому порядку, наприклад, числа за зростанням або спаданням, прізвища в алфавітному порядку тощо. Існує багато різноманітних методів сортування, серед яких: сортування обміном (метод «бульбашки»), сортування вибором, вставками, швидке сортування, сортування Шелла, пірамідальне сортування тощо.

Сортування методом обміну (бульбашкове сортування)

Метод «бульбашкового сортування» ґрунтується на перестановці сусідніх елементів. Для впорядкування елементів масиву здійснюються повторні проходи по масиву.

Переміщення елементів масиву здійснюється таким чином : масив переглядається зліва направо, здійснюється порівняння пари сусідніх елементів; якщо елементи в парі розміщені в порядку зростання, вони лишаються без змін, а якщо ні – міняються місцями.

В результаті першого проходу (якщо сортування здійснюється за зростанням) найбільше число буде поставлено в кінець масиву. При другій ітерації операції виконуються над елементами з першого до (N-1)-ого, у третій – від першого до (N-2)-ого тощо. Впорядкування масиву буде закінчено, якщо під час ітерації не виконається жодної перестановки елементів масиву. Факт перестановки фіксується за допомогою змінної, яка на початку має значення 0 і набуває значення 1 тоді, коли виконається перестановка в якій-небудь парі.

Таблиця 7.2

Сортування методом обміну

Масив до впорядкування	22	20	–1	–40	88	–75	–22
Перша ітерація	20	–1	–40	22	–75	–22	88
Друга ітерація	–1	–40	20	–75	–22	22	88
Третя ітерація	–40	–1	–75	–22	20	22	88
Четверта ітерація	–40	-75	–22	–1	20	22	88
П'ята ітерація	–75	-40	–22	–1	20	22	88

В лістингу 7.15 наведено приклад сортування масиву методом обміну.

Лістинг 7.15. Сортування масиву методом обміну

```
#include<stdio.h>
#include <string.h>
void main()
{
     int n = 7; int tmp, fl;
    int array[] = { 22, 20, -1, -40, 88, -75,-22 };
    printf("\n Початковий масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);

    do {
        fl = 0;
        for (int i = 1; i < n; i++)
            if (array[i - 1] > array[i]) {
                tmp = array[i];
                array[i] = array[i - 1];
                array[i - 1] = tmp;
                fl = 1;
            }
    } while (fl);
    printf("\nВідсортований масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
}
```

Результат виконання програми:

```
Початковий масив
  22  20  -1 -40  88 -75 -22
Відсортований масив
 -75 -40 -22  -1  20  22  88
```

Сортування методом вибору

Алгоритм роботи методу вибору передбачає наступні дії: при першій ітерації здійснюється знаходження мінімального елементу масиву, який міняється місцями з першим елементом. Наступні ітерації алгоритму сортування базуються на ітеративному пошуку мінімуму у невідсортованій частині масиву та подальшому обміні його місцями з

першим елементом цієї частини. Процес продовжується до повного сортування масиву.

Таблиця 7.3

Сортування методом вибору

Масив до впорядкування	22	20	–1	–40	88	–75	–22
Перша ітерація	**–75**	20	–1	–40	88	22	–22
Друга ітерація	**–75**	**–40**	–1	20	88	22	–22
Третя ітерація	**–75**	**–40**	**–22**	20	88	22	–1
Четверта ітерація	**–75**	**–40**	**–22**	**–1**	88	22	20
П'ята ітерація	**–75**	**–40**	**–22**	**–1**	**20**	22	88
Шоста ітерація	**–75**	**–40**	**–22**	**–1**	**20**	**22**	88

В лістингу 7.16 наведено приклад сортування масиву методом вибору.

Лістинг 7.16. Сортування масиву методом вибору

```
#include<stdio.h>
#include <string.h>
void main(){
   int n = 7, imin, a;
    int array[] = { 22, 20, -1, -40, 88, -75,-22 };
    printf("\n Початковий масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
    for (int i = 0; i < n - 1; i++)
    {
        imin = i;
        for (int j = i + 1; j < n; j++)
            if (array[j] < array[imin]) imin = j;
        a = array[i];
        array[i] = array[imin];
        array[imin] = a;
    }
    printf("\nВідсортований масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
}
```

Результат виконання програми:

```
Початковий масив
 22  20  -1 -40  88 -75 -22
Відсортований масив
 -75 -40 -22  -1  20  22  88
```

Сортування вставками

Алгоритм сортування вставками працює по принципу поступової побудови відсортованої послідовності. На *i*-му етапі відбувається «вставка» *i*-ого елемента a[i] у потрібну позицію серед вже впорядкованих елементів. Після цієї вставки перші *i* елементів будуть впорядковані. Це відбувається доки останній елемент не буде поставлено в потрібну позицію.

Таблиця 7.4

Сортування вставками

Масив до впорядкування	**22**	20	–1	–40	88	–75	–22
Перша ітерація	**20**	**22**	-1	-40	88	-75	-22
Друга ітерація	**-1**	**20**	**22**	-40	88	-75	-22
Третя ітерація	**-40**	**-1**	**20**	**22**	88	-75	-22
Четверта ітерація	**-40**	**-1**	**20**	**22**	**88**	-75	-22
П'ята ітерація	**-75**	**-40**	**-1**	**20**	**22**	**88**	-22
Шоста ітерація	**-75**	**-40**	**-22**	**-1**	**20**	**22**	**88**

В лістингу 7.17 наведено приклад сортування масиву вставками.

Лістинг 7.17. Приклад сортування вставками

```
#include<stdio.h>
#include <string.h>
void main()
{
   int n = 7, c;
   int array[] = { 22, 20, -1, -40, 88, -75,-22 };
   printf("\n Початковий масив \n");
   for (int i = 0; i < n; i++)
```

```
        printf("%4d", array[i]);
    for (int i = 1; i < n; i++)
    {
        c = array[i];
        for (int j = i - 1; j >= 0 && array[j] > c; j--)
        {
            array[j + 1] = array[j];
            array[j] = c;
        }
    }
    printf("\nВідсортований масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
}
```

Результат виконання програми:

```
Початковий масив
  22  20  -1 -40  88 -75 -22
Відсортований масив
 -75 -40 -22  -1  20  22  88
```

Швидке сортування

Алгоритм швидкого сортування можна описати наступним чином:

1. Обирається опорний елемент *p*

2. Масив розділяється на дві частини за опорним елементом: усі елементи, менші або рівні опорному, переміщуються ліворуч від нього, а всі елементи, більші за опорний – праворуч.

3. Якщо підмасиви ліворуч та праворуч від *p* містять більше одного елемента, рекурсивно виконуються пункт 1-2 для підмасивів.

Час роботи алгоритму швидкого сортування в гіршому випадку складає $O(n^2)$, але на практиці цей алгоритм виявляється одним із найшвидших.

Таблиця 7.5

Швидке сортування (QuickSort)

Масив до впорядкування	22	20	–1	**–40**	88	–75	–22
Перша ітерація	-75	-40	-1	20	88	22	-22

Друга ітерація	-75	-40	-1	20	-22	22	88
Третя ітерація	-75	-40	-1	-22	20	22	88
Четверта ітерація	-75	-40	-22	-1	20	22	88

В лістингу 7.18 наведено приклад впорядкування елементів масиву методом швидкого сортування.

Лістинг 7.18. Приклад швидкого сортування

```
#include<stdio.h>
#include <string.h>
void qs(int* arr, int first, int last) {
    if (first < last) {
        int left = first, right = last, middle = arr[(left +
right) / 2];
        do {
            while (arr[left] < middle) left++;
            while (arr[right] > middle) right--;
            if (left <= right) {
                int tmp = arr[left]; arr[left] = arr[right];
                arr[right] = tmp;
                left++; right--;
            }
        } while (left <= right);
        qs(arr, first, right); qs(arr, left, last);
    }
}
int main() {
int n = 7; int array[] = { 22, 20, -1, -40, 88, -75,-22 };
    printf("\n Початковий масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
    qs(array, 0, n - 1);
    printf("\nВідсортований масив \n");
    for (int i = 0; i < n; i++)
        printf("%4d", array[i]);
    return 0;
}
```

Результат виконання програми:

```
Початковий масив
 22  20  -1 -40  88 -75 -22
Відсортований масив
-75 -40 -22  -1  20  22  88
```

Контрольні запитання та завдання

1. Що таке покажчик і для чого він використовується в мові С?
2. Як оголосити покажчик на певний тип даних?
3. Яка різниця між змінною та змінною-покажчиком?
4. Як отримати адресу змінної за допомогою оператора &?
5. Як розіменувати покажчик, щоб отримати значення, на яке він вказує?
6. Які основні арифметичні операції можна виконати з покажчиками?
7. Яка різниця між масивом і покажчиком на перший елемент масиву?
8. Як динамічно виділити пам'ять за допомогою функції malloc?
9. Як звільнити динамічно розділену пам'ять за допомогою функції free?
10. Як скопіювати один масив в іншому за допомогою покажчиків?
11. Що таке покажчик на покажчик?
12. Що буде виведено:

```
int A = 100; int *a = &A;
double B = 2.3; double *b = &B;
printf(" %d\n", sizeof(A));
printf(" %d\n", sizeof(a));
printf(" %d\n", sizeof(B));
printf(" %d\n", sizeof(b));
```

13. Що буде виведено:

```
int *p;
int arr[8] = {5,7,-3,0,15,21,4,2};
p = &arr[0];
```

```
printf(" %d\n", *p);
p++;
printf(" %d\n", *p);
p = p + 3;
printf(" %d\n", *p);
```

14. Що таке масиви?
15. Як розташовуються елементи масивів у пам'яті?
16. Які бувають масиви за розмірністю та способом виділення їм пам'яті?
17. Як оголошуюсь статичні масиви?
18. Як звернутись до першого та останнього елементу масиву?
19. Яким чином обчислюється кількість байтів пам'яті, виділених масиву?
20. Наведіть фрагмент програми (за допомогою операторів for, while та do-while), що обчислює суму, добуток, кількість парних елементів одновимірного масиву, найбільше та найменше значення?
21. Що таке одновимірний масив?
22. Що таке елемент масиву та індекси?
23. Як описується одновимірний масив?
24. Які існують способи ініціалізації одновимірних масивів?
25. Що таке двовимірний масив?
26. Які дані можуть бути записані у масив?
27. Які оператори використовуються для опису двовимірних масивів?
28. Яким чином здійснюється доступ до елементів двовимірного масиву?
29. Наведіть приклади з життя, в яких можна було б використовувати двовимірні масиви?
30. Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?

```
int ar[10]; int sum = 0;
for (int i = 0; i<10; i++)
ar[i] = i + 1;
for (int i = 0; i<10; i++)
if (i % 2) sum += ar[i];
```

```
printf("%d ", sum);
```

31. Як оголосити і ініціалізувати рядок?
32. Чому рядок в С має фіксовану довжину?
33. Які є способи введення і виведення рядків?
34. Як визначити довжину рядка?
35. Як скопіювати один рядок в інший?
36. Як порівняти два рядки?
37. Чому не можна просто присвоїти один рядок іншому?
38. Дано два рядки s1 і s2. Яким чином у мові С правильно записати перевірку умови їх рівності (рядок s1 дорівнює рядку s2)?
    - if(strcmp(s1,s2)==0) { … }
    - if(!strcmp(s1,s2)) { … }
    - if (s1 == s2) { … }
    - if (s1 != s2) { … }
    - if (s1 <= s2) { … }
39. Що таке сортування масиву і чому воно важливе?
40. Які основні алгоритми сортування ви знаєте?
41. Напишіть програму, яка приймає рядок і підраховує кількість входжень в нього заданого символу.
42. Напишіть програму, яка перетворює всі символи рядка в верхній або нижній регістр.
43. Напишіть програму, яка розбиває рядок на слова, використовуючи пробіли як роздільники.
44. Напишіть програму, яка видаляє всі дублікати символів з рядка.
45. Напишіть програму яка створює динаічний масив цілих чисел та сортує його по зростанню методом швидкого сортування.

8. Функції

При створенні програм використовують різні концепції, ідеї та методи, що визначають стиль або підхід до вирішення завдань програмування. Це дозволяє правильно організувати структуру програми та процес її виконання. Підхід до побудови програм на мові С, у якому головну роль відіграють процедури (або функції) називається процедурним програмуванням. Методи, які ґрунтуються на моделі процедурного програмування описують підходи до розробки, структурування та реалізації функцій, що формують програму.

Основу процедурного програмування на будь-якій мові програмування складає процедура (походить від назви) або функція (як різновид, що саме відповідає мові програмування С). Структура кожної функції співпадає зі структурою головної функції програми *main()*. Функції іноді ще називають підпрограмами.

Функція – це фрагмент програмного коду, до якого можна звернутися з іншого місця програми. Її розробка та реалізація у програмі може розглядатися як побудова операцій, що вирішують конкретну задачу (підзадачу).

Функції мають параметри, тому їх операції узагальнені для використання будь-якими фактичними аргументами відповідного типу. Вхідними даними для неї є аргументи та глобальні структури даних, що використовуються функцією. Вихідними даними є ті значення, які функція повертає, а також зміни глобальних даних, модифікації.

Будь-яка програма на мові С складається з функцій, причому одна з яких обов'язково повинна мати ім'я *main()*.

Синтаксис опису функції:

```
тип_поверт_значення ім'я_функції ([список_аргументів])
{
    оператори тіла функції
}
```

Слід чітко розрізняти поняття опису та представлення функцій. Опис функції задає її ім'я, тип значення, що повертається та список параметрів. Він дає можливість організувати доступ до функції (рис. 8.1).

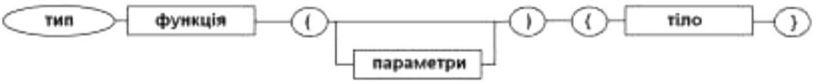


Рисунок 8.1 – Синтаксис опису функції

Функція може повертати або не повертати значення результату своєї роботи. Якщо функція повертає значення, то тип даних має бути визначений перед її ім'ям. Якщо функція не повертає значення, то вказується тип *void*.

Програма в мові С завжди починає роботу з функції *main()*. Для передачі управління функціям потрібно здійснити їх виклик, вказавши їх ім'я. По завершенні роботи функції, управління повертається в ту ж позицію, звідки було здійснено виклик.

В лістингу 8.1 наведено приклад роботи функцій типу *void*, які не повертають значення. Програма починає виконання з функції main(), де першим оператором для виконання є виклик функції func1(). Під час виконання функція виводить текст у консоль "Процедура 1". По завершенні роботи, управління передається *main()* і виконується наступний оператор, який є викликом функції *func2()*. Під час виконання *func2()* у консоль виводиться текст "Процедура 2". Після завершення func2() управління повертається у функцію *main()* і так як відсутні наступні оператори для виконання, програма закінчує роботу.

Лістинг 8.1. Приклад роботи функцій типу *void*

```
#include<stdio.h>
void func1();
void func2();

void main()
{
    func1();
    func2();
}

void func1()
{
    printf("Процедура 1\n");
}

void func2()
{
    printf("Процедура 2\n");
}
```

Результат виконання програми:

```
Процедура 1
Процедура 2
```

Звернемо увагу на те, що текст програми починається з оголошення прототипів функцій – схематичних записів, що повідомляють компілятору ім'я та форму кожної функції у програмі.

Приклад оголошення прототипу функцій:

```
void func1();
void func2();
```

Будь-яка невідповідність між прототипом (оголошенням) функції та її визначенням (заголовком) призведе до помилки компіляції. Кожна з оголошених функцій має бути визначена у програмі, тобто заповнена операторами, що її виконують. Спочатку йтиме заголовок функції, який повністю співпадає з оголошеним раніше прототипом функції, але без заключної крапки з комою. Фігурні дужки обмежують тіло функції. В середині функцій можливий виклик будь-яких інших функцій, але неможливо оголосити функцію в середині тіла іншої функції.

В лістингу 8.2 розглянемо завдання обчислення коренів квадратного рівняння із застосуванням функціонального підходу.

Лістинг 8.2. Приклад використання функцій для обчислення коренів квадратного рівняння

```
#include <stdio.h>
#include <math.h>
float A, B, C;

void GetData();
void Run();

void main()
{
   GetData();
   Run();
}
```

```
void GetData()
{
    printf("A,B,C:");
    scanf("%f%f%f", &A, &B, &C);
}
void Run()
{
    float D;
    float X1, X2;
    if ((A == 0) && (B != 0))
    {
        X1 = (-C) / B;
        printf("X1: %f\n", X1);
        exit(0);
    }
    D = B * B - 4 * A * C;
    if (D < 0) printf("\nNo roots...");
    if (D == 0)
    {
        X1 = (-B) / (2 * A);
        printf("X1=X2=%f\n", X1);
    }
    if (D > 0)
    {
        X1 = (-B + sqrt(D)) / (2 * A);
        X2 = (-B - sqrt(D)) / (2 * A);
        printf("X1: %f\nX2: %f\n", X1, X2);
    }
}
```

Результат виконання програми:

```
A,B,C:-1  2 4
X1: -1.236068
X2: 3.236068
```

В лістингу 8.2 для вирішення завдання використано дві функції (процедури) типу *void*. *GetData()* використовується для введення значень коефіцієнтів квадратного рівняння *A, B, C*, які описано як глобальні змінні. Процедура *Run()* здійснє обчислення коренів квадратного рівняння. В функції *main()* здійснюється виклик цих двох процедур.

Якщо в описі функції не вказується її тип, то за замовчуванням функція буде типу *int. GetData()* та *Run()* мають явно визначений тип *void,* що означає, що вони не генерують значення для повернення.

Передача параметрів

У попередньому підрозділі було розглянуто один з різновидів функцій – процедури. Але також вирізняють функції, які можуть отримувати значення та повертати результат.

Значення, які передаються у функцію називають параметрами. Усі параметри, крім вказівників і масивів, передаються за значенням. Це означає, що під час виклику функції їй передаються лише копії значень змінних.

Сама функція не в змозі змінити цих значень у викликаючій функції. Наступний приклад це демонструє:

В лістингу 8.3 розглянемо приклад передачі параметрів у функцію.

Лістинг 8.3. Приклад передачі параметрів у функцію

```
#include<stdio.h>
void test(int a)
{
    a = 15;
    printf("    in test : a==%d\n", a);
}
void main()
{
    int a = 10;
    printf("before test : a==%d\n", a);
    test(a);
    printf("after test  : a==%d\n", a);
}
```

Результат виконання програми:

```
before test : a==10
    in test : a==15
 after test  : a==10
```

При передачі параметрів за значенням у функції утворюється локальна копія, що приводить до збільшення об'єму необхідної пам'яті. При виклику функції стек відводить пам'ять для локальних копій параметрів, а при виході з функції ця пам'ять звільняється. Це не впливає на значення параметрів, що були передані у функцію. Цей спосіб використання пам'яті не тільки потребує додаткового її об'єму, але й займає додатковий час для зчитування.

Лістинг 8.4 ілюструє механізм передачі параметрів у функцію. Параметри *one*, *two* передаються за значенням, *three* – передається за допомогою покажчика, забезпечуючи прямий доступ до змінної.

Лістинг 8.4. Приклад передачі параметрів у функцію

```
#include<stdio.h>
void test(int one, int two, int* three)
{
    printf("\nАдреси параметрів у функції test");
    printf("\nАдреса one дорівнює %d", &one);
    printf("\nАдреса two дорівнює %d", &two);
    printf("\nАдреса three дорівнює %d", three);
    *three += 1;
}
int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int a1, b1;
    int c1 = 42;
    printf("\nАдреси змінних у функції main");
    printf("\nАдреса a1 дорівнює %d", &a1);
    printf("\nАдреса b1 дорівнює %d", &b1);
    printf("\nАдреса c1 дорівнює %d", &c1);
    test(a1, b1, &c1);
    printf("\nЗначення c1 = % d\n", c1);
}
```

Результат виконання програми:

```
Адреси змінних у функції main
Адреса a1 дорівнює 1264515252
Адреса b1 дорівнює 1264515284
Адреса c1 дорівнює 1264515316
```

```
Адреси параметрів у функції test
Адреса one дорівнює 1264515216
Адреса two дорівнює 1264515224
Адреса three дорівнює 1264515316
Значення c1 =  43
```

В лістингу 8.4 значення змінної *c1* передається у функцію за покажчиком **three*. В тілі функції *test()* збільшується на одиницю значення на яке вказує покажчик. Таким чином змінюється значення змінної *c1* у функції *main()*.

Для повернення результатів обчислень з функції використовується оператор *return*. Тип значення що повертається з функції визначається при оголошенні функції. Функція може повертати лише одне значення через оператор *return*. Лістинг 8.5 демонструє повернення значення з функції.

Лістинг 8.5. Приклад повернення значення з функції

```
#include<stdio.h>

int Star(int);

int main()
{
    int n = 7;
    while ( n>0 )
    {
        n = Star(n);
    }
    return 0;
}

int Star(int n)
{
    for (int i = 0; i < n; i++)
        printf("%c", '*');
    printf("\n");
    return n - 2;
}
```

Результат виконання програми:

```
*******
*****
***
*
```

В функції *main()* в відбувається виклик функції *Star(int n),* з передачею значення змінної *n.* У функції *Star(int n)* отриманий параметр використовується для виведення рядка символів '*'. Оператор return повертає результат зменшення змінної *n – 2.* Це значення передається назад у функцію *main()* і стає вхідним параметром для наступного виклику функції *Star(int n).*

В лістингу 8.6 розглянемо програму, що відшукує та видаляє коментарі з програмного коду. Код програми складається з чотирьох функцій (*main, echo_quote, in_comment, rcomment*).

Лістинг 8.6. Приклад використання функцій для пошуку та видалення коментарів з програмного коду

```
#include <stdio.h>
void rcomment(int c);
void in_comment(void);
void echo_quote(int c);

int main()
{
    int c, d;
    while ((c = getchar()) != EOF)
        rcomment(c);
    return 0;
}

void echo_quote(int c)
{
    int d;
    putchar(c);
    while ((d = getchar()) != c)
    {
        putchar(d);
        if (d == '\\')
            putchar(getchar());
```

```
    }
    putchar(d);
}

void in_comment(void)
{
    int c, d;
    c = getchar();
    d = getchar();
    while (c != '*' || d != '/')
    {
        c = d;
        d = getchar();
    }
}

void rcomment(int c) {
    int d;
    if (c == '/')
        if ((d = getchar()) == '*')
            in_comment();
        else if (d == '/') {
            putchar(c);
            rcomment(d);
        }
        else {
            putchar(c);
            putchar(d);
        }
    else if (c == '\'' || c == '"') echo_quote(c);
    else
        putchar(c);
}
```

Результат виконання програми:

```
void (*efct)(char* s); /* змінній-покажчику виділена ОП*/
void (*efct)(char* s);

efct("Error"); /* груба помилка */
efct("Error");
```

Функція *rcomment(int c)* виконує роль лексичного аналізатора. Коли зустрічається початок коментаря, функція делегує завдання пошуку його кінця функції *in_comment()*. Це гарантує, що коментарі будуть повністю ігноруватися при подальшому аналізі. Функція також відстежує одинарні та подвійні лапки. При виявленні одинарних чи подвійних лапків, вона передає управління функції *echo_quote(int c)*, яка гарантує, що інформація всередині лапок відображається точно та не приймається помилково за коментар. Функція *echo_quote(int c)* не вважає лапки, що слідують за зворотнім слешем, заключними. Будь-який інший символ виводяться на консоль. Програма завершується, коли *getchar()* повертає символ кінця файла.

Рекурсивні функції

Рекурсія – це спосіб організації обчислювального процесу, при якому функція в ході виконання операторів звертається сама до себе. Функція називається *рекурсивною*, якщо під час її виконання можливий повторний її виклик.

Прямою (безпосередньою) *рекурсією* називається рекурсія, при якій всередині тіла деякої функції міститься виклик тієї ж функції.

Приклад:

```
void fn(int i)
{
  /* ... */
  fn(i);
  /* ... */
}
```

Непрямою рекурсією називається рекурсія, що здійснює рекурсивний виклик функції шляхом ланцюга викликів інших функцій. При цьому всі функції ланцюга, що здійснюють рекурсію, вважаються також рекурсивними.

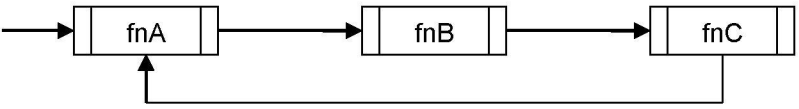


Рисунок 8.2 – Непряма рекурсія

Приклад:

```
void fnA(int i);
void fnB(int i);
void fnC(int i);
void fnA(int i) {
  /* ... */
  fnB(i);
  /* ... */
}
void fnB(int i) {
  /* ... */
  fnC(i);
  /* ... */
}
void fnC(int i) {
  /* ... */
  fnA(i);
  /* ... */
}
```

Якщо функція викликає сама себе, то в стеку створюється копія значень її параметрів, як і при виклику звичайної функції, після чого управління передається першому оператору функції. При повторному виклику цей процес повторюється.

В лістингу 8.7 продемонстровано приклад прямої рекурсивної функції виведення послідовності чисел. При виклику у функцію передається початкове значення (5) і функція викликає сама себе зменшуючи значення на одиницю. Умова завершення виконання рекурсивної функції $a<1$. В результаті роботи в консоль буде виведено послідовність чисел від 5 до 1.

Лістинг 8.7. Приклад прямої рекурсії

```
#include <stdio.h>
int funA(int);

int main() {
    funA(5);
    return 0;
}
```

```
int funA(int a) {
    if (a < 1) return 0;
    printf("%d\n", a);
    a--;
    return funA(a);
}
```

Результат виконання програми:

```
5
4
3
2
1
```

Класичний приклад рекурсивної функції – обчислення факторіалу, тобто добутку натуральних чисел від 1 до N. N! = 1* 2* 3* … *N

Факторіал визначається рівнянням

n! = n ⋅ (n − 1) ⋅ (n −2) …3 ⋅ 2 ⋅ 1.

n! для будь - якого позитивного цілого числа n дорівнює n, помноженому на (n −1)!:

n! = n ⋅ [(n −1) ⋅ (n −2) … 3 ⋅ 2 ⋅ 1] = n ⋅ (n −1)!

Приклад:

0!=1

5!=5*4!=5*4*3!=5*4*3*2!=5*4*3*2*1!=5*4*3*2*1

Лістинг 8.8. Приклад прямої рекурсії

```
#include <stdio.h>
int factorial(int n);

int main() {
    int n;
    printf("n="); scanf_s("%d", &n);
```

```
    printf("factorial %d = %d\n", n, factorial(n));
    return 0;
}

int factorial(int n) {
    return ((n == 1) ? 1 : n * factorial(n - 1));
}
```

Результат виконання програми:

```
n=6
factorial 6 = 720
```

Роботу рекурсивної функції *factorial*() розглянемо на прикладі n=6! За рекурентним співвідношенням $6! = 5! \cdot 6$. Таким чином, щоб обчислити 6! ми спочатку повинні обчислити 5!. Використовуючи співвідношення, маємо, що $5! = 4! \cdot 5$, тобто необхідно визначити 4!. Продовжуючи процес, отримаємо :

1) $6! = 5! \cdot 6$ 2) $5! = 4! \cdot 5$ 3) $4! = 3! \cdot 4$ 4) $3! = 2! \cdot 3$ 5) $2! = 1! \cdot 2$ 6) $1! = 1$

В кроках 1-5 обчислення не завершується, а відкладається до наступного кроку, шостий крок є ключовим. Відповідно, обчислення проводиться від 6-ого кроку до 1-ого:

factorial(6);

6 * factorial(5);

6 *5 * factorial(4);

6 *5* 4 * factorial(3);

6 *5* 4* 3 * factorial(2);

6 *5* 4* 3 * 2 * factorial(1);

6 *5*4*3*2*1;

6 *5*4*3*2;

6 *5*4*6;

6 *5*24;

6 *120;

720;

Важливим для розуміння ідеї рекурсії є те, що в рекурсивних функціях можна виділити дві серії кроків.

Перша серія – це *кроки рекурсивного занурення функції* в саму себе до тих пір, поки вибраний параметр не досягне граничного значення. Ця важлива вимога завжди повинна виконуватися, щоб функція не створила нескінченну послідовність викликів самої себе. Кількість таких кроків називається глибиною рекурсії.

Друга серія – це *кроки рекурсивного виходу* до тих пір, поки вибраний параметр не досягне початкового значення. Вона, як правило забезпечує отримання проміжних і кінцевих результатів.

Покажчики на функції

На функцію в мові С можна створити покажчик. Покажчики на функції широко використовується для передачі функцій як параметрів іншим функціям.

Приклад:

```
float (*func)(float a, float b);
/* покажчик на функцію, що приймає два параметри типу float і повертає
значення типу float  */
```

Покажчик на функцію містить адресу першого байта або слова виконуваного коду функції. Над покажчиками на функцію заборонені арифметичні операції.

Покажчик на функцію перед використанням має бути проініціалізованим. При спробі працювати з непроініціалізованим покажчиком виникають грубі помилки.

Приклад:

```
#include<stdio.h>
#include<conio.h>
void main(void)
{
  void (*efct)(char *s); /* змінній-покажчику виділена ОП, але
                            efct не містить значення адреси ОП для функції */
  efct("Error"); /* груба помилка – спроба працювати з
                    неініціалізованим покажчиком*/
}
```

Для того, щоб можна було використовувати покажчик на функцію потрібно спочатку присвоїти йому значення адреси пам'яті, де розташована функція (лістинг 8.9).

Лістинг 8.9. Приклад використання покажчика на функцію

```
#include <stdio.h>

void print(const char* s)
{
    puts(s);
}

void main(void)
{
    void (*efct)(const char* s);
    efct = &print;
    (*efct)("Function Print!");
}
```

Результат виконання програми:

```
Function Print!
```

Для отримання значення адреси функції необов'язково використовувати операцію &, тому можуть використовуватись два варіанти ініціалізації покажчиків.

Приклад:

```
1). efct=&print;
2). efct=print;
```

Операція розіменування покажчика на функцію * також є необов'язковою.

Приклад:

```
1). (*efct)("Function Print!");
2). efct("Function Print!");
```

Покажчикам на функції можна присвоювати адреси стандартних бібліотечних функцій (лістинг 8.10-8.11).

Лістинг 8.10. Приклад використання покажчика на функцію

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

void main(void)
{
    double (*fn)(double x);
    float y, x = 1;
    fn = sin;
    y = fn(x);
    printf("sin(%g)==%g\n", x, y);
    fn = cos;
    y = fn(x);
    printf("cos(%g)==%g\n", x, y);
}
```

Результат виконання програми:

```
sin(1)==0.841471
cos(1)==0.540302
```

Лістинг 8.11. Приклад використання покажчика на функцію

```
#include<stdio.h>

void main(void)
{
    int (*func_ptr)(const char*, ...) = printf;
    func_ptr("function pointer\n");
}
```

Результат виконання програми:

```
function pointer
```

Покажчики на функції можуть також виступати в якості аргументів функцій (лістинг 8.12).

Лістинг 8.12. Приклад використання покажчика на функцію

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

double fn(double (*pfn)(double x), double x)
{
    double y = pfn(x);
    printf("y==%g\n", y);
    return y;
}

double sin_cos(double x)
{
    return sin(x) * cos(x);
}

void main(void)
{
    fn(sin, 1);
    fn(&cos, 1);
    fn(&sin_cos, 1);
}
```

Результат виконання програми:

```
y==0.841471
y==0.540302
y==0.454649
```

Класи пам'яті

Будь-яка змінна та функція, описана у програмі на С, належить до конкретного класу пам'яті, що визначає час її існування та область видимості. *Час існування змінної* – це період, протягом якого змінна існує в пам'яті, а *область видимості* (область дії) – це частина програми, в якій змінна може використовуватися.

В мові С існує чотири специфікатори класу пам'яті: *auto*, *register*, *extern* і *static*.

Таблиця 8.1.

Область дії та час існування змінних різних класів пам'яті

Клас пам'яті	*Ключове слово*	*Час існування*	*Область дії*
Автоматичний	auto	тимчасово	блок
Регістровий	register	тимчасово	блок
Статичний локальний	static	постійно	блок
Статичний глобальний	static	постійно	файл
Зовнішній	extern	постійно	програма

Клас пам'яті для функції завжди *external*, якщо перед її описом не стоїть специфікатор *static*. Клас пам'яті конкретної змінної залежить або від місця розташування її опису, або задається явно за допомогою спеціального специфікатору класу пам'яті, що розташовується перед описом функції. Усі змінні мови *C* можна віднести до одного з наступних класів пам'яті:

1) auto (автоматична, локальна)

Ключове слово *auto* використовується рідко. Кожна змінна, описана в тілі функції (в середині блоку), обмеженого фігурними дужками, відноситься до класу пам'яті автоматичних (локальних) змінних:

Приклад:

```
int anyfunc(void)
{
  char item;
  ........
}
або
for (int i = 0; i < 10; i++)
{ ... }
```

Область дії локальної змінної *item* та *i* поширюється лише на блок, в якому вони оголошені. Пам'ять відводиться під змінну динамічно, під час виконання програми при вході у функцію (блок), в якому описана відповідна змінна. Локальна змінна тимчасово зберігається в стеку, коли функція (блок) починає свою роботу і автоматично

звільняється по закінченню роботи функції (блоку), тому декілька функцій безконфліктно можуть оголошувати локальні змінні з ідентичними іменами.

Область видимості такої змінної розпочинається з місця її опису і закінчується в кінці блоку, в якому змінна описана. Доступ до таких змінних із зовнішнього блоку неможливий.

Застосування автоматичних змінних в локальних блоках дозволяє наближати опис таких змінних до місця їх розташування.

Лістинг 8.13 демонструє приклад використання класу пам'яті *auto*.

Лістинг 8.13. Приклад використання класу пам'яті *auto*

```
#include <stdio.h>
int main()
{
    int i = 125;
    for (int i = 0; i < 5; i++)
    {
        printf("\n%d", i);
    }
    printf("\n % d", i);
    return 0;
}
```

Результат виконання програми:

```
0
1
2
3
4
 125
```

2) register (регістрова)

Register може використовуватися лише для автоматичних змінних або для формальних параметрів функції. Він вказує компілятору на те, що користувач бажає розмістити змінну не в оперативній пам'яті, а на одному з швидкодіючих регістрів комп'ютеру, від чого програма виконуватиметься більш ефективніше. Звісно, це стосується перш за

все саме тих змінних, звертання до яких у функції виконуватиметься найчастіше. На практиці на цей тип змінних накладаються деякі обмеження, що відображають реальні можливості конкретної машини. У випадку надлишкових та недопустимих описів подібний специфікатор просто ігнорується.

Лістинг 8.14 демонструє приклад виконання підрахунків не використовуючи клас пам'яті *register*, а лістинг 8.15 виконує те ж завдання з використанням класу пам'яті *register*. За результатами виконання програм можна стверджувати, що використовуючи клас пам'яті *register* швидкодія збільшується.

Лістинг 8.14. Приклад виконання підрахунків не використовуючи клас пам'яті *register*

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

int main()
{
    int i, sum = 0;
    clock_t start, end;

    start = clock();
    for (i = 0; i < 1000000; ++i)
    {
        sum += i;
    }
    printf("Sum: %d\n", sum);
    end = clock();

    double time_spent = (double)(end - start) /
CLOCKS_PER_SEC;
    printf("Time: %f sek\n", time_spent);
    return 0;
}
```

Результат виконання програми:

```
Sum: 1783293664
Time: 0.001175 sek
```

Лістинг 8.15. Приклад виконання підрахунків використовуючи клас пам'яті *register*.

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

int main()
{
    register int i;
    int sum = 0;
    clock_t start, end;

    start = clock();
    for (i = 0; i < 1000000; ++i)
    {
        sum += i;
    }
    printf("Sum: %d\n", sum);

    end = clock();

    double time_spent = (double)(end - start) /
CLOCKS_PER_SEC;
    printf("Time: %f sek\n", time_spent);
    return 0;
}
```

Результат виконання програми:

```
Sum: 1783293664
Time: 0.000425 sek
```

3) extern (зовнішня, глобальна)

Будь-яка змінна, описана не в тілі функції (без специфікатору класу пам'яті), по замовчуванню відноситься до *extern* – змінних (глобальних). Глобальні змінні продовжують існувати протягом усього життєвого циклу програми. У випадку, коли користувач не визначить початкове значення змінної, компілятор автоматично присвоїть їй нульове значення за замовчуванням. Найчастіше оголошення таких змінних розташовується безпосередньо перед *main*() (лістинг 8.16).

Лістинг 8.16. Приклад використання глобальної змінної.

```
#include <stdio.h>
int globalnaZminna = 10;
void someFunction();

int main()
{
    int lokalnaZminna = 5;

    printf("Значення глобальної змінної: %d\n",
globalnaZminna);
    printf("Значення локальної змінної: %d\n",
lokalnaZminna);

    someFunction();
    printf("Значення глобальної змінної після роботи
функції: %d\n", globalnaZminna);
    return 0;
}

void someFunction() {
    globalnaZminna = 20;
    printf("Значення глобальної змінної всередині функції:
%d\n", globalnaZminna);
}
```

Результат виконання програми:

```
Значення глобальної змінної: 10
Значення локальної змінної: 5
Значення глобальної змінної всередині функції: 20
Значення глобальної змінної після роботи функції: 20
```

У лістингу 8.16 змінна *globalnaZminna* оголошена як ціле число *(int)* і розміщена перед функцією *main()*. Для неї вказано початкове значення 10, але після відпрацювання функції *someFunction()* її значення змінилося.

Глобальні змінні завжди залишаються під контролем завантажувача програми, що здійснює збірку програми із множини obj-файлів. Саме завдяки цьому до зовнішніх змінних можливий доступ з

інших файлів. Для використання таких змінних слід задати специфікатор *extern*.

Приклад:

```
/*file1.c*/
#include<stdio.h>
int globalnaZminna = 20;
void main()
{
  printf("globalnaZminna : %d", globalnaZminna);
}

/*file2.c*/
#include<stdio.h>
void main()
{
  extern globalnaZminna;
  globalnaZminna=25
  printf("globalnaZminna : %d", globalnaZminna);
}
```

Extern вказує компілятору, що змінна, яку він бачить, визначена в іншому місці, найчастіше в іншому файлі. Це дозволяє програмістам розділити код на модулі та ефективніше управляти великими проектами.

4) static (статична)

Для збереження значень змінних між викликами функцій та обмеження доступу до них, використовуються статичні зміни. Вони можуть бути оголошені як всередині функції (внутрішні), так і поза нею (зовнішні). Внутрішні статичні зміни, нараховані від автоматичних, зберігають своє значення при виконанні всіх програм, навіть якщо функції, в яких вони оголошені, зараховуються багаторазово. Таким чином, вони надають функцію власної, постійної області пам'яті.

Приклад:

```
int funct(void)
{
  static int value=20;
  ...
}
```

Компілятор виділяє для змінної value окрему область пам'яті та ініціалізує її значенням. Ініціалізація виконується лише один раз і не повторюється при кожному виклику функції. У подальшому змінна зберігає значення, яке отримала під час останнього виконання функції. Варто зазначити, що така змінна залишається недоступною для завантажувача програми. Її область видимості обмежується функцією, у якій вона була оголошена. Інші функції не можуть отримати доступ до статичних змінних, оголошених в інших функціях.

Зовнішні статичні об'єкти доступні лише у файлі, де вони описані, але залишаються невидимими для інших файлів. Це забезпечує спосіб ізоляції даних і їх обробки підпрограмами таким чином, щоб інші підпрограми або дані не могли викликати конфлікти.

В лістингу 8.17 наведено приклад використання різних класів пам'яті – глобальна змінна, локальна автоматична змінна, локальна статична змінна.

Лістинг 8.17. Приклад використання різних класів пам'яті.

```
#include <stdio.h>
int x = 10; // глобальна змінна
void myFunction() {
    int y = 20; // локальна автоматична змінна
    static int z = 30; // локальна статична змінна

    printf("x = %d, y = %d, z = %d \n", x, y, z);
    z++;
}
int main() {
    myFunction();
    myFunction();
    return 0;
}
```

Результат виконання програми:

```
x = 10, y = 20, z = 30
x = 10, y = 20, z = 31
```

Додаткові можливості функції main()

Функція *main()* може як повертати деяке значення в операційну систему, так і приймати параметри.

```
тип main(int argc, char* argv[], char *env[]){/*…*/}
```

Імена параметрів мають назви *argc, argv* та *env,* їх дозволяється перейменовувати. Параметр *argc* містить ціле число аргументів командного рядка, *argv* – це масив покажчиків на рядки. Значення *argv[0]* – повний шлях до виконуючого файлу (.exe) програми, *argv[1]* та *argv[2]* містять перший та другий після імені програми параметри командного рядка, *argv[argc-1]* вказує на останній аргумент, *argv[argc]* містить *NULL*. Параметр *env* – це масив покажчиків на рядки, які є в середовищі в якому виконується програма

Лістинг 8.18 демонструє приклад використання аргументів, що передаються в функцію *main()*.

Лістинг 8.18. Приклад передачі аргументів у функцію *main()*.

```
#include <stdio.h>

void main(int argc, char* argv[], char* env[])
{
    int i;
    printf("Значення argc = %d \n\n", argc);
    printf("В командному рядку міститься %d параметрів \n",
argc);
    for (i = 0; i < argc; i++)
        printf(" argv[%d]: %s\n", i, argv[i]);

    printf("Середовище містить наступні рядки:\n");
    for (i = 0; env[i] != NULL; i++)
        printf(" env[%d]: %s\n", i, env[i]);
}
```

Результат виконання програми:

```
C:\Users\admin>C:\Debug\ConsoleApplication1.exe 1_st_arg "2_arg " 3 4
"dummy" stop!
Значення argc = 7

В командному рядку міститься 7 параметрів
 argv[0]: C:\Debug\ConsoleApplication1.exe
 argv[1]: 1_st_arg
 argv[2]: 2_arg
 argv[3]: 3
 argv[4]: 4
 argv[5]: dummy
 argv[6]: stop!
Середовище містить наступні рядки:
 env[0]: ALLUSERSPROFILE=C:\ProgramData
 env[1]: APPDATA=C:\Users\admin\AppData\Roaming
 env[2]: CommonProgramFiles=C:\Program Files\Common Files
 env[3]: CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
 env[4]: CommonProgramW6432=C:\Program Files\Common Files
 env[5]: COMPUTERNAME=DESKTOP-HMV7GRF
…
```

Для виконання програми з командного рядка необхідно вказати шлях до exe-файлу та вхідні параметри для функції *main()*.

Приклад:

```
C:\Users\admin>C:\Debug\ConsoleApplication1.exe 1_st_arg "2_arg " 3 4
"dummy" stop!
```

В лістингу 8.19 наведено приклад передачі аргументів у функцію *main()* та подальшого їх використання в програмі.

Лістинг 8.19 демонструє приклад використання аргументів, що передаються в функцію *main()*.

Лістинг 8.19. Приклад передачі аргументів у функцію *main()*.

```
#include <stdio.h>
int main(int argc, char* argv[]) {
```

```
        int i;
        printf("Кількість аргументів: %d\n", argc);

        for (i = 0; i < argc; i++) {
            printf("Аргумент %d: %s\n", i, argv[i]);
        }
        int k = atoi(argv[3]);
        for (i = 0; i < k; i++) {
            printf("%s %s\n", argv[1], argv[2]);
        }
        return 0;
}
```

Результат виконання програми:

```
C:\Users\admin>C:\Users\admin\source\repos\ConsoleApplication1\x64\De
bug\ConsoleApplication1.exe Hello world 3
Кількість аргументів: 4
Аргумент 0:
C:\Users\admin\source\repos\ConsoleApplication1\x64\Debug\ConsoleAppl
ication1.exe
Аргумент 1: Hello
Аргумент 2: world
Аргумент 3: 3
Hello world
Hello world
Hello world
```

В результаті виконання програми було отримано чотири аргументи, три з них, а саме «Hello», «world», «3» було використано для подальшого виведення текстового повідомлення на консоль «Hello world».

Контрольні запитання та завдання

1. Що таке функція в мові С і для чого вона використовується?
2. Який синтаксис функції в мові С?
3. Яка різниця між оголошенням і визначенням функції?
4. Що таке аргументи функції?
5. Що таке сигнатура функції?

6. Які бувають типи функцій за поверненням значення?
7. Які є способи передачі аргументів у функції?
8. Коли використовувати передачу за значенням, а коли за посиланням?
9. Що таке прототип функції?
10. Опишіть роботу функції:

```
int sum(int a, int b)
{ return a + b; }
```

11. Опишіть роботу функції:

```
int factorial(int n)
{ if (n == 0) { return 1; }
else { return n * factorial(n - 1); } }
```

12. Опишіть роботу функції:

```
void increment(int *ptr) { (*ptr)++; }
```

13. Які є способи передачі даних між функціями в мові С?
14. Як використовуються покажчики для передачі даних між функціями?
15. Як передаються масиви в функції?
16. Як передаються структури в функції?
17. Що таке рекурсія? Наведіть приклад рекурсивної функції в мові С.
18. Які основні компоненти рекурсивної функції?
19. Які переваги та недоліки використання рекурсії?
20. Як працює стек викликів під час виконання рекурсивних функцій?
21. Як запобігти переповненню стека при використанні рекурсії?
22. Коли варто використовувати рекурсію, а коли краще віддати перевагу ітеративним алгоритмам?
23. Які переваги та недоліки використання глобальних змінних?
24. Які інструменти можна використовувати для аналізу використання пам'яті?
25. Що таке клас пам'яті в мові С?
26. Які основні класи пам'яті існують у мові С?
27. Яка різниця між змінними з автоматичним і статичним класом пам'яті?
28. Для чого використовується специфікатор register?

29. Яка різниця між глобальними та локальними змінними?
30. Коли слід використовувати клас пам'яті auto?
31. Коли слід використовувати клас пам'яті extern?
32. Які переваги і недоліки використання змінних з класом пам'яті register?
33. Як ініціалізуються статичні змінні?
34. Чи можна змінювати значення статичної змінної всередині функції?
35. Що таке область видимості змінної?
36. Напишіть функцію, яка приймає динамічний масив цілих чисел і повертає новий динамічний масив, що містить елементи в зворотному порядку.
37. Напишіть функцію, яка приймає два динамічні масиви цілих чисел і повертає новий динамічний масив, що містить всі елементи з обох масивів.
38. Напишіть функцію, яка приймає динамічний масив цілих чисел і повертає новий динамічний масив, що містить лише унікальні елементи (тобто, кожен елемент має зустрічатися лише один раз).
39. Напишіть функцію, яка приймає масив цілих чисел і його розмір, і змінює порядок елементів на зворотний. Використовуйте покажчики для доступу до елементів масиву.
40. Напишіть функцію, яка приймає масив дійсних чисел і його розмір, і знаходить максимальний та мінімальний елементи. Використовуйте покажчики для обходу масиву.

9. Структури

Оголошення структури

Структури дозволяють об'єднувати в єдиному об'єкті сукупність значень, які можуть мати різні типи. Оголошення структури здійснюється за допомогою ключового слова *struct*.

Синтаксис опису структури:

```
struct [ім'я_структури]
{
   тип1 елемент1;
   тип2 елемент2;
   ......................
   типN елементN;
} [список описів];
```

В прикладі розглянемо представлення структури date, що складається з декількох полів: числові – день, місяць, рік, текстові – назва тижня та місяця.

Приклад:

```
struct date {
  int day ;
  int month ;
  int year;
  char day_name[15];
  char mon_name[14];
} arr[100],*pd, data, new_data;
```

В даному прикладі оголошуються:

data, new_data – змінні типу структури *date*;
pd – покажчик на тип *data*
arr – масив із 100 елементів, кожний елемент якого має тип *date*.

Ключове слово *typedef* дозволяє задавати нові імена для типів даних, що робить код більш зручним і зрозумілим. Воно також може бути використане для спрощення оголошення структур.

Приклад:

```
typedef struct mystruct {
```

```
  int year;
  char size;
  float field;
} MYSTRUCT;

MYSTRUCT s;  /* те саме, що й struct mystruct s; */
```

Приклад:

```
typedef struct dataTypes {
    float aFloat;
    int anInt;
    char aString[8];
    char aChar;
    char aLong;
  } DataTypes;

  DataTypes data;
```

Пам'ять розподіляється у структурі покомпонентно, зліва-направо, від молодших до старших адрес пам'яті (рис. 9.1).

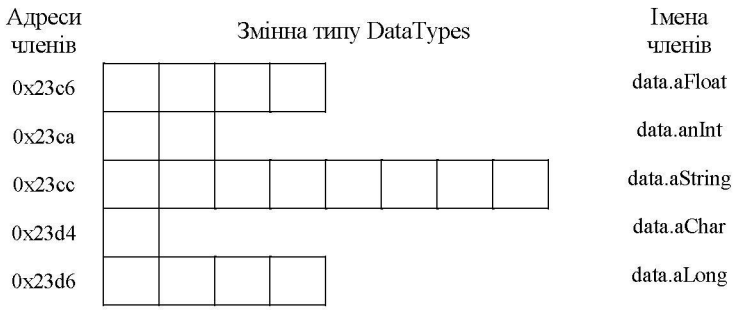


Рисунок 9.1 - Зберігання елементів структури у пам'яті

Потрібно відзначити, що на відміну від описів інших типів даних, опис структури не виділяє місця у пам'яті під елементи структури. Її опис визначає лише так званий шаблон, що описує характеристики змінних, що будуть розміщуватися у конкретній структурі. Для використання структур в програмі, потрібно створити змінну типу

структури. Це можна зробити двома способами: після фігурної дужки, що завершує опис структури, вказати список ідентифікаторів, або окремо оголосити змінні використавши ім'я структури в коді програми.

Доступ до окремого елемента структури забезпечується операторами:

. (прямий селектор)

-> (непрямий селектор).

Приклад:

```
struct mystruct {
  int i;
  char str[21];
  double d;
} s,*sptr=&s;

s.i =3;
sptr->d = 1.23;
```

Ініціалізація структури подібна до тієї, що є у масивах, але з урахуванням розміщення даних різного типу.

Приклад:

```
struct person {
  char frnm[20];
  char nm[30];
  int year;
  char s;
};
struct person poet = {"Taras", "Shevtchenko", 1814, 'M'};
struct person classics[] = {{"Alfred", "Aho", 1939, 'M'},
            {"Seimour", "Ginzburg",}, /* … */
            {"Jeffrey", "Ulman", 1938, 'M'}};
```

У вищенаведеному прикладі ініціалізується змінна *poet* і масив структур *classics*. Значення classics[1].year і classics[1].s мають значення відповідно 0 і '\0'.

Для порівняння змінних типу структури необхідно перевіряти рівність відповідних полів.

Приклад:

```
struct point {
  float x,y;
  char c;
} point1, point2;
if (point1.x == point2.x && point1.y == point2.y && point1.c == point2.c)
{  /* … */  };
```

В лістингу 9.1 наведено приклад створення структури *book*, внесення даних та виведення на консоль.

Лістинг 9.1. Робота зі структурою

```
#include<stdio.h>
#define MAXTIT 41
#define MAXAUT 31

struct book{
    char title[MAXTIT];
    char author[MAXAUT];
    float value;
};

main()
{
 struct book libry;
 printf("Введіть назву книги ");
 gets(libry.title);
 printf("Введіть прізвище автора ");
 gets(libry.author);
 printf("Введіть ціну ");
 scanf("%f", &libry.value);
 printf("\n%s '%s',%g грн.\n", libry.author,
        libry.title, libry.value);
}
```

Результат виконання програми:

```
Введіть назву книги Солодка Даруся
Введіть прізвище автора Марія Матіос
Введіть ціну 380

Марія Матіос 'Солодка Даруся', 380 грн.
```

Кожний опис структури вводить унікальний тип структури, тому в наступному прикладі змінні a і a1 мають однаковий тип struct A, але об'єкти a і b мають різні типи структури. Структурам можна виконувати присвоювання тільки в тому випадку якщо і вихідна структура, і структура, які присвоюється мають один і той же тип.

Приклад:

```
struct A {
  int i, j;
  double d;
} a, a1;
struct B {
  int i, j;
  double d;
} b;
a = a1; /*можна виконати, так як a і a1 мають однаковий тип */
a = b; /* помилка */
```

Масиви структур

З масивами структур можна працювати як і зі звичайними масивами. Розглянемо приклад, який ілюструє оголошення масиву структур *arr*, що складається зі 100 елементів, кожний з яких має тип *Data*. Кожний елемент масиву – це окрема змінна типу *Data*, що складається із трьох полів цілого типу – *d*, *m*, *y*.

Приклад:

```
typedef struct Date
{
  int d; /* день */
```

```
  int m; /* місяць */
  int y; /* рік */
} Date;

Date arr[100];
  arr[25].d=24;  /* доступ до полів структури */
  arr[12].m=12;
```

Доступ до полів структури аналогічний доступу до звичайних змінних, плюс використання індексу номеру елементу у квадратних дужках.

Лістинг 9.2. демонструє реалізацію роботи зі структурою *Data*. Окремими функціями реалізуємо ініціалізацію елементів структури, додавання нового значення, виведення дати на екран, визначення високосного року.

Лістинг 9.2. Робота зі структурою

```
#include<stdio.h>
typedef struct Date
{
    int d; /* день */
    int m; /* місяць */
    int y; /* рік */
}Date;

void set_date_arr(Date* arr, Date value, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        arr[i].d = value.d;
        arr[i].m = value.m;
        arr[i].y = value.y;
    }
}

void print_date_arr(Date* arr, int n)
{
    int i;
    for (i = 0; i < n; i++)
```

```
    {
        printf("%d.%d.%d\n", arr[i].d, arr[i].m, arr[i].y);
    }
}

void print_date(Date& d) {/* виведення на екран дати */

    printf("%d.%d.%d\n", d.d, d.m, d.y);
}

void init_date(Date& d, int dd, int mm, int yy) {
    /* ініціалізація структури типу Date */
    d.d = dd;
    d.m = mm;
    d.y = yy;
}

int leapyear(int yy) {/* визначення, чи високосний рік */
    if ((yy % 4 == 0 && yy % 100 != 0) || (yy % 400 == 0))
return 1;
    else return 0;
}

void add_year(Date& d, int yy) {/* додати yy років до
дати*/
    d.y += yy;
}

void add_month(Date& d, int mm) {
    /*додати mm місяців до  дати */
    d.m += mm;
    if (d.m > 12)
    {
        d.y += d.m / 12;
        d.m = d.m % 12;
    }
}

void add_day(Date& d, int dd) {
    /* додати dd днів до дати */
    int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };
    d.d += dd;
    if (leapyear(d.y)) days[1] = 29;
```

```
    while ((d.d > days[d.m - 1]))
    {
        if (leapyear(d.y)) days[1] = 29;
        else days[1] = 28;
        d.d -= days[d.m - 1];
        d.m++;
        if (d.m > 12)
        {
            d.y += d.m % 12;
            d.m = d.m / 12;
        }
    }
}

void main(void)
{
    Date date1, date2;
    Date array[10] = { {12,11,1980},{15,1,1982},{8,6,1985},
                  {8,8,1993},{20,12,2002},{10,1,2003} };

    init_date(date1, 15, 12, 2002);
    add_day(date1, 16);
    print_date(date1);
    puts("");
    init_date(date2, 1, 1, 2003);
    add_month(date2, 10);
    print_date(date2);
    puts("");
    print_date_arr(array, 6);
}
```

Результат виконання програми:

```
31.12.2002

1.11.2003

12.11.1980
15.1.1982
8.6.1985
8.8.1993
```

20.12.2002
10.1.2003

Бітові поля

Бітові поля (*bit fields*) – це спеціальний вид полів у структурах, що дозволяє визначати кількість бітів, відведених для зберігання значень цілих типів. Вони забезпечують ефективне використання пам'яті, оскільки дозволяють зберігати дані в мінімально необхідній кількості бітів.

При оголошенні бітового поля після типу елемента ставиться двокрапка (:) і вказується ціле число, яке визначає розмір поля (кількість бітів). Цей розмір має бути константою, що лежить у межах від 0 до максимальної кількості бітів, яку може зберігати відповідний тип даних.

В лістингу 9.3 наведено приклад роботи зі структурою в якій використано бітові поля. Дану структуру описано чотирма бітовими полями, які можуть зберігати значення типу int. Для ефективного використання пам'яті кожне з полів має встановлений власний розмір (кількість бітів).

Лістинг 9.3. Робота зі структурою

```
#include <stdio.h>
struct bit_field {
    int bit_1 : 1;
    int bits_2_to_5 : 4;
    int bit_6 : 1;
    int bits_7_to_16 : 10;
} bit_var;

int main() {
    bit_var.bit_1 = 0;
    bit_var.bits_2_to_5 = 7;
    bit_var.bit_6 = 1;
    bit_var.bits_7_to_16 = 512;

    printf("bit_1 = %d\n", bit_var.bit_1);
    printf("bits_2_to_5 = %d\n", bit_var.bits_2_to_5);
    printf("bit_6 = %d\n", bit_var.bit_6);
    printf("bits_7_to_16 = %d\n", bit_var.bits_7_to_16);
```

```
    return 0;
}
```

Результат виконання програми:

```
bit_1 = 0
bits_2_to_5 = 7
bit_6 = -1
bits_7_to_16 = -512
```

Об’єднання (union)

Об’єднання (union) у мові С – це спеціальний тип даних, який дозволяє зберігати в одній і тій самій області пам’яті кілька змінних, але тільки одну з них можна використовувати в певний момент часу. Розмір об’єднання визначається розміром найбільшого з його полів, оскільки всі поля накладаються одне на одне в пам’яті. Використання об’єднань вимагає додаткової уваги, контроль за тим, значення якого типу зберігається в даний момент в об’єднанні покладається на програміста.

Усі елементи об'єднання знаходяться в одній і тій самій області пам’яті та мають одну й ту саму адресу. При присвоєнні значення одному з елементів об'єднання, значення іншого елемента автоматично втрачається.

Синтаксис:

```
union [ім’я_об’єднання]
{
  тип1 елемент1;
  тип2 елемент2;
  ..........
  типN елементN;
} [список описів];
```

Приклад:

```
union sign
{
  int svar;
  unsigned uvar;
} number;
```

В наведеному прикладі оголошується змінна типу об'єднання з ім'ям *number*. Список оголошень елементів об'єднання містить дві змінні: *svar* типу *int* і *uvar* типу *unsigned*. Це об'єднання дозволяє запам'ятати ціле значення в знаковому або в без знаковому вигляді. Тип об'єднання має ім'я *sign*.

Приклад:

```
union
{
  char *a,b;
  float f[20];
} var;
```

В наведеному прикладі оголошується змінна типу об'єднання з ім'ям *var*. Список оголошень елементів містить три оголошення: покажчик *a* та змінна *b* типу *char* і масив *f* типу *float* з 20 елементів. Дане об'єднання не має власного імені. Обсяг пам'яті, виділений для змінної var, дорівнює розміру, необхідному для зберігання масиву f, оскільки це найбільший елемент об'єднання.

Лістинг 9.4 демонструє приклад створення та виведення даних на консоль зі змінної *union*.

Лістинг 9.4. Робота з об'єднанням (*union*)

```
#include <stdio.h>
union Data {
    int intVal;
    double dblVal;
    char charVal[4];
};

int main() {
    union Data data;
    printf("union: %d byte\n", sizeof(Data));

    data.intVal = 65;
    printf("intVal: %d\n", data.intVal);
    data.dblVal = 3.14;
    printf("dblVal: %.2f\n", data.dblVal);
    data.charVal[0] = 'A';
    data.charVal[1] = '\0';
```

```
    printf("charVal: %s\n", data.charVal);
    return 0;
}
```

Результат виконання програми:

```
union: 8 byte
intVal: 65
dblVal: 3.14
charVal: A
```

Тип перерахування enum

При написанні програм часто виникає необхідність визначити декілька іменованих констант, які мають різні значення. Для цього зручно скористатися типом даних *enum* (enumeration – перерахування), значення якого описуються цілочисельними константами.

Синтаксис :

```
enum [ ім'я_типу ] { список_констант };
```

При необхідності визначати змінні даного типу в програмі йому задається ім'я. Компілятор забезпечує, щоб ці змінні приймали значення тільки із вказаного списку констант.

Приклад:

```
enum {mRead, mEdit, mWrite, mCreate } Mode;
```

Змінна *Mode* типу *enum* описує константи *mRead*, *mEdit*, *mWrite*. В момент оголошення змінна ініціалізується значенням першої константи *(mRead)*. Їй можна присвоювати будь-які допустимі значення.

Приклад:

```
Mode = mCreate;
```

Змінну *Mode* можна використовувати в операторі *switch*.

Приклад:

```
switch(Mode)
{
  case mRead:   /* … */
                break;
  case mEdit:   /* … */
                break;
  case mWrite:  /* … */
                break;
  case mCreate: /* … */
                break;
}
```

За замовчуванням значення, задані в enum, інтерпретуються як цілі числа: перше має значення 0, друге — 1 і так далі. Однак це значення можна змінити, вказавши після імені константи знак рівності та бажане ціле число.

Приклад:

```
enum {mRead = –1, mEdit, mWrite = 2, mCreate } Mode;
```

В наведеному прикладі не всім константам задано значення, тому вони набудуть значення на один більшого, ніж попередня константа.

Приклад:

```
mRead = –1
mEdit = 0
mWrite = 2
mCreate = 3
```

Лістинг 9.5 демонструє використання перерахування enum для визначення днів тижня.

Лістинг 9.5. Робота з перерахуванням *enum*

```
#include <stdio.h>
enum Day {
```

```
    MONDAY,    /* 0 */
    TUESDAY,   /* 1 */
    WEDNESDAY, /* 2 */
    THURSDAY,  /* 3 */
    FRIDAY,    /* 4 */
    SATURDAY,  /* 5 */
    SUNDAY     /* 6 */
};

void printDay(enum Day day)
{
    switch (day) {
    case MONDAY:
        printf("Monday\n");
        break;
    case TUESDAY:
        printf("Tuesday\n");
        break;
    case WEDNESDAY:
        printf("Wednesday\n");
        break;
    case THURSDAY:
        printf("Thursday\n");
        break;
    case FRIDAY:
        printf("Friday\n");
        break;
    case SATURDAY:
        printf("Saturday\n");
        break;
    case SUNDAY:
        printf("Sunday\n");
        break;
    default:
        printf("Invalid day\n");
    }
}

int main()
{
    enum Day today;
    today = WEDNESDAY;
    printf("Today is: ");
    printDay(today);
```

```
    printf("Tomorrow is: ");
    printDay(THURSDAY);
    return 0;
}
```

Результат виконання програми:

```
Today is: Wednesday
Tomorrow is: Thursday
```

Контрольні запитання та завдання

1. Що таке структура в мові С і яка її основна мета?
2. Як оголосити структуру? Наведіть приклад.
3. Які типи даних можуть використовуватись для створення структури?
4. Як ініціалізувати структуру при оголошенні?
5. Яка різниця між структурою та масивом?
6. Як оголосити масив структур?
7. Як отримати доступ до елементів масиву типу структури?
8. Як визначити розмір структури в байтах?
9. Яка різниця між структурою і об'єднанням?
10. Як порівняти дві структури?
11. Що таке бітові поля в структурах?
12. Чим відрізняється бітове поле від звичайного поля структури?
13. Яка основна мета використання бітових полів?
14. В яких ситуаціях бітові поля особливо корисні?
15. Які типи даних можна використовувати для бітових полів?
16. Як оголосити бітове поле в структурі?
17. Що таке union у мові С? Коли використовується union?
18. Який синтаксис union? Які обмеження union?
19. Поясніть фрагмент коду:

```
enum Colors {
  Red = 1, Green, Blue = 4
};
```

20. Поясніть фрагмент коду:

    union Data { int i; float f; char c; };

21. Створіть структуру Book, що містить поля: title (назва книги), author (автор), year (рік видання), price (ціна). Напишіть функцію, яка приймає масив книг і виводить інформацію про книги, видані після заданого року.

22. Створіть структуру Student, що містить поля: name (ім'я), group (група), grades (масив оцінок). Напишіть функцію, яка приймає масив студентів і виводить інформацію про студентів, середній бал яких вище заданого значення.

23. Створіть структуру Point, що містить поля: x (координата x), y (координата y). Напишіть функцію, яка обчислює відстань між двома точками.

24. Створіть структуру Rectangle, що містить поля: topLeft (ліва верхня точка), bottomRight (права нижня точка). Напишіть функцію, яка обчислює площу прямокутника.

25. Створіть структуру Employee, що містить поля: name (ім'я), position (посада), salary (зарплата). Створіть масив типу структури Employee розміром 10 елементів. Напишіть функцію, яка приймає масив працівників і виводить інформацію про працівників, зарплата яких вище заданого значення.

26. Створіть структуру Car, що містить поля: brand (марка), model (модель), year (рік випуску), price (ціна). Створіть масив типу структури Car розміром 5 елементів. Напишіть функцію, яка приймає масив автомобілів і виводить інформацію про автомобілі, ціна яких знаходиться в заданому діапазоні.

27. Створіть структуру Fraction, що містить поля: numerator (чисельник), denominator (знаменник). Напишіть функцію, яка додає два дроби.

28. Створіть структуру Date, що містить поля: day (день), month (місяць), year (рік). Напишіть функцію, яка перевіряє, чи є заданий рік високосним.

29. Напишіть програму, яка використовує union для зберігання цілого числа, дійсного числа та символу в одній змінній. Додайте можливість виводу значення кожного типу.

30. Створіть програму, яка демонструє різницю у використанні пам’яті між struct та union. Виведіть розмір структури та об'єднання, що містять ті самі поля.

10. Файлові потоки

Файли в програмуванні використовуються для зберігання інформації. В мові С файли представлені як *потік* (*stream*), що являє послідовність байтів. При цьому потік не інтерпретує байти як певні типи даних. Розшифровка змісту написаних у ньому байтів лежить на програмі.

Інформація про потік заноситься в структуру *FILE*, яка визначена у файлі *stdio.h*. Ця структура містить інформацію про відкритий файл, наприклад, його буфер, позицію у файлі та стан помилок, тощо. Щоб відкрити файл використовується функція *fopen*, яка повертає покажчик на структуру типу *FILE*. Дана функція відкриває файл із заданим ім'ям і зв'язує з ним потік.

Приклад:

```
typedef struct
{
  short level;              /*рівень буферу */
  unsigned flags;           /*статус файлу */
  char fd;                  /*дескриптор файлу */
  char hold;                /*попередній символ, якщо немає буферу */
  short bsize;              /*розмір буферу */
  unsigned char *buffer;    /*буфер передавання даних */
  unsigned char *curp;      /*поточний активний покажчик */
  short token;              /*перевірка коректності */
} FILE;
```

Синтаксис функції *fopen*:

```
FILE *fopen(const char *filename, const char *mode);
```

Функція *fopen* приймає два аргументи: **filename* – ім'я файлу, **mode* – режим відкриття файлу. Файл можна відкривати в декількох режимах (таблиця 10.1).

Таблиця 10.1.

Значення аргументу mode функції fopen

"r"	відкриття файлу без дозволу на модифікацію, файл відкривається лише для читання.
"w"	створення нового файлу тільки для запису, якщо файл із вказаним ім'ям вже існує, то він перезапишеться.
"a"	відкриття файлу тільки для додавання інформації в кінець файлу,

	якщо файл не існує, він створюється.
"r+"	відкриття існуючого файлу для читання та запису.
"w+"	створення нового файлу для читання та запису, якщо файл із вказаним ім'ям вже існує, то він перезаписується.
"a+"	відкриває файл у режимі читання та запису для додавання нової інформації у кінець файлу; якщо файл не існує, він створюється.

До вказаних режимів доступу в кінці або перед символом "+" може додаватися символ "t" (текстовий файл), або "b" (бінарний, двійковий файл).

Текстові файли

В лістингу 10.1 наведено приклад відкриття текстового файлу *test.txt* в режимі відкриття файлу "r".

Лістинг 10.1. Приклад відкриття текстового файлу

```
#include<stdio.h>
int main()
{
    FILE* f;
    if ((f = fopen("test.txt", "r")) == NULL)
    {
        printf("There are an error\n");
        return;
    }

    printf("file open\n");
fclose(f);
return 0;
}
```

Результат виконання програми:

```
There are an error
```

У цьому прикладі змінна *f* асоціюється з файлом *"test.txt"*, який відкривається у режимі лише для читання. Це дозволяє зчитувати дані з файлу. Після завершення роботи файл слід закрити за допомогою функції *fclose()*.

Якщо відкрити файл командою *fopen("test.txt", "rt+");*, то з'являється можливість не тільки зчитувати дані, але й записувати. Читання з текстового файлу може здійснюватися по рядках, символах або за форматом.

Записування символу в файловий потік здійснюється за допомогою функції *putc*().

```
int putc(int ch, FILE *f);
```

Читання рядка здійснюється за допомогою функції *fgets()*.

```
char *fgets(char *s,int n,FILE *stream);
```

Аргументами функції *fgets()* є *s* – покажчик на буфер, в який читається рядок, *n* – кількість символів, *stream* – покажчик на файл. Зчитування з файлу відбувається до символу кінця рядка «*\n*», або читається *n–1* символ. В кінці прочитаного рядка записується нуль-символ.

В лістингу 10.2 продемонстровано зчитування рядка з файлу та виведення його на консоль.

Лістинг 10.2. Зчитування рядка з файлу

```
#include<stdio.h>
#include<string.h>
main() {
    char s[80];
    FILE* f;
    if ((f = fopen("1.txt", "r")) == NULL) {
        printf("There are an error\n");
        return 0;
    }
    do {
        fgets(s, 80, f);
        printf("%s", s);
    } while (!feof(f));
    fclose(f);
}
```

Результат виконання програми:

```
Good morning!!
```

Функція *feof()* використовується для перевірки, чи досягнуто кінець файлу під час читання. Якщо кінець файлу досягнуто, то *feof()* повертає ненульове значення і цикл завершується. Синтаксис:

```
int feof(FILE *stream);
```

Для зчитування з файлу форматованих даних використовується функція *fscanf()*. Синтаксис:

```
int fscanf(FILE *stream, const char *format[, address, …]);
```

Аргументи функції: *stream* - покажчик на відкритий файл, *format* – типи зчитуваних даних, змінна, у яку будуть записані зчитані значення.

Під час форматованого зчитування можуть виникати помилки через досягнення кінця файлу або невідповідність формату записаних у файлі даних. Переконатися в успішності зчитування можна за значенням, яке повертає функція *fscanf()*. Якщо зчитування пройшло успішно, функція повертає кількість правильно зчитаних полів. Тому процес читання даних доцільно організовувати відповідним чином.

Приклад:

```
if (fscanf(f,"%d%d%d", &a, &b, &c) != 3)
{
   printf("Помилка читання!\n");
};
```

Для запису даних у текстовий файл найчастіше використовуються функції *fgetc*(), *fputs()* та *fprintf()*. Функція *fgetc*() використовується для читання чергового символу з потоку, відкритого функцією *fopen*(). Синтаксис функції:

```
int fgetc(FILE *f);
```

Функції *fprintf()* є аналогом функції *printf()*, але вона приймає додатковий аргумент – посилання на файл, він є першим у списку аргументів. Синтаксис функції *fprintf()*:

```
int fprintf(FILE *stream, const char *format[,argument,…]);
```

В лістингу 10.3 наведено приклад зчитування та запису даних до текстового файлу. Функцією *fopen()* було відкрито файл *age.txt* в режимі читання. Функцією fscanf() зчитано дані цілого типу та занесено до змінної *age*. На консоль було виведено значення «54», яке

отримано з файлу. Файл було закрито та повторно відкрито в режимі дозапису. Інкремент змінної *age* за допомогою функції *fprintf()* записано у файл *age.txt*, що продемонстровано на рисунку 10.1. По завершенні роботи файл було закрито.

Лістинг 10.3. Зчитування та запис даних до файлу

```
#include<stdio.h>
int main() {
    FILE* fi;
    int age;
    fi = fopen("age.txt", "r"); /* відкриття файлу для
читання */
    fscanf(fi, "%d", &age);/*читання з файлу числового
значення */
    printf("%d", age);
    fclose(fi);  /* закриття файлу */

    fi = fopen("age.txt", "a"); /* відкриття файлу для
додавання інформації в кінець */
    fprintf(fi, "\n%d", ++age); /* запис рядка в файл  */
       fclose(fi);  /* закриття файлу */
   }
```

Результат виконання програми:

```
54
```

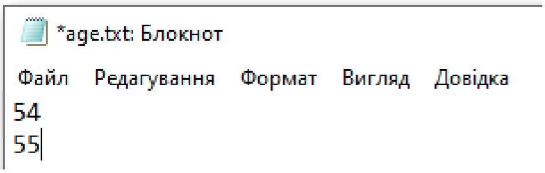


Рисунок 10.1 – Вміст файлу age.txt

Бінарні файли

Бінарні файли використовуються для збереження та зчитування даних у двійковому форматі, що дозволяє ефективніше працювати з великими обсягами інформації.

Бінарні файли мають перевагу над текстовими при збереженні числових даних. Операції запису та читання з таких файлів відбуваються значно швидше, оскільки немає потреби у форматуванні (перетворенні даних у текстовий вигляд і навпаки). Крім того, бінарні файли зазвичай займають менше місця, ніж їхні текстові аналоги. Вони дозволяють довільно переміщатися всередині файлу для читання або запису даних у будь-якому порядку, тоді як у текстових файлах інформація обробляється, як правило, послідовно.

Відкриття бінарних файлів здійснюється аналогічно до текстових файлів. Функції *fread* і *fwrite* використовуються для читання та запису даних у файл.

Синтаксис функції *fread()*:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Синтаксис функції *fwrite()*:

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);
```

Обидві функції опрацьовують дані за допомогою покажчика *ptr*. Розмір даних які читаються або записуються вказується за допомогою параметра *size* та задається в байтах.

В лістингу 10.4 наведено приклад створення та запису даних в бінарний файл.

Лістинг 10.4. Запис даних до бінарного файлу

```
#include <stdio.h>
struct myStruct
{
    int i;
    char ch;
};

int main()
{
    FILE* stream;
    struct myStruct mystruct;
    if ((stream = fopen("test.bin", "wb")) == NULL)
    {
        fprintf(stderr, "Неможливо відкрити файл\n");
        return 1;
    }
```

```
    mystruct.i = 0;
    mystruct.ch = 'A';
    fwrite(&mystruct, sizeof(mystruct), 1, stream);
    fclose(stream);
    return 0;
}
```

В результаті виконання програми буде створено бінарний файл test.bin та записано в нього інормацію.

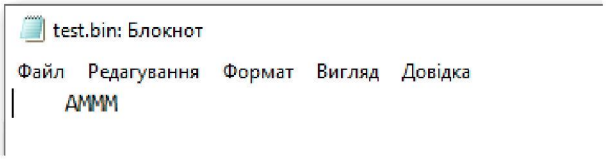


Рисунок 10.2 – Вміст файлу test.bin

При роботі з бінарними файлами є деякі особливості запису і читання рядків.

Приклад:

```
char s[10];
strcpy(s, "Example");
…
fwrite(s, strlen(s)+1, sizeof(char), stream);
```

Запис рядків відбувається за допомогою функції *fwrite()* посимвольно. Функція *strlen()* визначає довжину рядка *s* без врахування нуль-символа, тому в прикладі використано запис *strlen(s)+1*.

Читання даних здійснюється за допомогою функції *fread()* посимвольно. Аргументи функції *fread()* аналогічні до функції *fwrite()*.

Приклад:

```
fread(s, strlen(s)+1, sizeof(char), stream);
```

У випадку роботи з рядками різної довжини, зручним є внесення в файл розміру рядка перед самим рядком. Це забезпечує можливість безпомилкової ідентифікації початку та кінця рядка у файлі.

Приклад:

```
…
int i = strlen(s)+1;
fwrite(&i, 1, sizeof(int), stream);
fwrite(s, i, 1, stream);
…
fread(&i, 1, sizeof(int), stream);
fread(s, i, 1, stream)
```

У всіх наведених вище прикладах дані зчитувалися послідовно. Однак при роботі з бінарними файлами можливе довільне зчитування інформації. Це досягається завдяки використанню «покажчика файлу» (курсора), який визначає поточну позицію в файлі. Під час читання курсор автоматично зміщується на кількість прочитаних байтів. Дізнатися поточну позицію курсора у файлі можна за допомогою функції *ftell()*. Синтаксис функції *ftell()*:

```
long ftell(FILE *stream);
```

Для встановлення курсору у файлі в поточну позицію використовується функція *fseek()*. Синтаксис функції *fseek()*:

```
int fseek(FILE *stream, long offset, int whence);
```

Функція *fseek()* змінює поточну позицію у файлі на задану кількість байт (*offset*) від певної точки відліку, яка визначається параметром *whence*. Параметр *whence* може приймати значення 0, 1, 2 (таблиця 10.2).

Таблиця 10.2.

Можливі значення параметра whence функції fseek

Константа	*whence*	*Точка відліку*
SEEK_SET	0	Початок файлу
SEEK_CUR	1	Поточна позиція
SEEK_END	2	Кінець файлу

Якщо *whence=1*, то *offset* може приймати додатне або від'ємне значення, тобто зсув може здійснюватись вперед або назад по файлу.

Для переміщення курсору на початок файлу використовують функцію *rewind()*. Синтаксис функції *rewind()*:

```
void rewind(FILE *stream);
```

Переміщення курсору на початок файлу можна виконати і за допомогою функції *fseek()*.

Приклад:

```
fseek(stream, 0L, SEEK_SET);
```

В лістингу 10.5 наведено приклад застосування функцій для роботи з бінарним файлом.

Лістинг 10.5. Робота з бінарним файлом

```
#include<stdio.h>
long filesize(FILE* stream);

int main()
{
    FILE* stream;
    stream = fopen("test.bin", "wb+");
    fprintf(stream, "This is a test1");
    printf("Розмір файла test.bin рівний %ld байт\n",
filesize(stream));
    fclose(stream);
    return 0;
}

long filesize(FILE* stream)
{
    long curpos, length;
    curpos = ftell(stream);
    fseek(stream, 0L, SEEK_END);
    length = ftell(stream);
    fseek(stream, curpos, SEEK_SET);
    return length;
}
```

Результат виконання програми:

```
Розмір файла test.bin рівний 15 байт
```

Контрольні запитання та завдання

1. Які існують режими відкриття файлів в мові С?
2. Як закрити файл в мові програмування С?
3. Як прочитати один символ з текстового файлу?
4. Як прочитати рядок з текстового файлу?
5. Як записати символ у текстовий файл?
6. Як записати рядок у текстовий файл?
7. Як перевірити, чи досягнуто кінець файлу?
8. Як переміститися на початок файлу?
9. Як отримати поточну позицію у файлі?
10. Як зчитати дані певного типу з файлу за допомогою форматованого введення?
11. Як записати дані певного типу у файл за допомогою форматованого виведення?
12. Як обробити помилку відкриття файлу?
13. Як обробити помилку читання або запису файлу?
14. Що таке бінарний файл і чим він відрізняється від текстового?
15. Які переваги використання бінарних файлів?
16. Які функції використовуються для роботи з бінарними файлами?
17. Які режими відкриття файлу використовуються для бінарних файлів?
18. Як відкрити бінарний файл для читання?
19. Поясніть фрагмент коду:

    ```
    FILE *file = fopen("filename.bin", "a+");
    ```
20. Як записати дані в бінарний файл?
21. Для чого використовується функція fseek()?
22. Для чого використовується функція ftell()?
23. Як записати структуру в бінарний файл?
24. Поясніть фрагмент коду:

    ```
    fwrite(&data, sizeof(data), 1, file);
    ```
25. Поясніть фрагмент коду:

    ```
    fread(&data, sizeof(data), 1, file);
    ```

11. Директиви препроцесора

Директиви препроцесора в мові С – це спеціальні інструкції, які не є частиною виконуваного коду, а обробляються до компіляції програми. На етапі попередньої обробки можуть виконуватись наступні дії: підключення інших файлів до компільованого, визначення глобальних констант і макросів, налаштування умовної компіляції та виконання директив препроцесора. Директиви починаються з символу #. У рядку перед директивою можуть бути тільки пробіли, після директиви крапка з комою не ставиться.

Директива #include

Директива #include використовується для включення в код іншого файлу. Це може бути стандартна бібліотека або власний файл. Синтаксис:

```
#include "ім'я_файла"

#include <ім'я_файла>
```

Директива "ім'я_файла" використовується для включення файлів заголовків створених користувачем. Пошук файлу здійснюється в поточній директорії, а потім в каталогах включення.

Директива <ім'я_файла> використовується для включення файлів заголовків, які містять стандартні функції та оголошення. Пошук препроцесором заданого файла в каталогах визначається встановленими каталогами включення (include directories).

Текст файлу, що включається, може містити інші директиви препроцесора, зокрема і директиву #include. Це дозволяє вкладати директиви #include одну в одну. Кількість рівнів такої вкладеності обмежена конкретною реалізацією компілятора.

Приклад:

```
#include <stdio.h>  /* приклад 1*/
#include "defs.h"   /* приклад 2*/
```

В першому прикладі за допмогою директиви *#include* до основного файлу програми підключається стандартний заголовочний файл з ім'ям *stdio.h*, який містить функції для забезпечення введення-виведення.

В другому прикладі до програми підключається файл з ім'ям *defs.h*. Використання подвійних лапок вказує, що під час пошуку цього файлу спочатку слід перевірити директорію, яка містить поточний файл.

У мові С не можна безпосередньо використовувати іменовану константу або змінну для визначення шляху до файлу в директиві #include, оскільки директиви препроцесора обробляються до компіляції, а значення іменованих констант або змінних визначаються під час виконання програми.

Директиви препроцесора, такі як #include, працюють з константними значеннями або літералами, а не з змінними або константами, визначеними під час виконання. Однак, можна використовувати макроси для визначення шляху до файлу, якщо шлях є статичним і відомий на момент компіляції.

Приклад:

```
#define myincl "c:\test\my.h"
#include myincl
```

У наведеному прикладі *myincl* є макросом, який визначає шлях до файлу, і препроцесор замінить *myincl* на відповідне значення.

Директива #define

Синтаксис:

```
#define   ідентифікатор   текст

#define   ідентифікатор   (список_параметрів)   текст
```

Директива *#define* замінює всі входження ідентифікатора у коді на текст, що йде після нього в директиві. Цей процес називається макропідстановкою. Заміна відбувається лише тоді, коли ідентифікатор виступає окремою лексемою; якщо він є частиною рядка або довшого ідентифікатора, підстановка не виконується. Якщо за ідентифікатором слідує список параметрів, то директива визначає макровизначення з параметрами, приклад якого наведено в лістингу 11.1.

Лістинг 11.1. Макровизначення з параметрами

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

```
#define SQUARE 25

int main() {
    int a = 5;
    int b = SQUARE * a;
    printf("b = %d", b);
}
```

Результат виконання програми:

```
b = 125
```

Текст являє собою набір лексем, таких як ключові слова, константи, ідентифікатори або вирази. Від ідентифікатора (або параметрів у дужках) його має відокремлювати хоча б один пробільний символ. Якщо текст не поміщається в один рядок, його можна перенести на наступний, поставивши в кінці рядка символ зворотного слеша \, після чого натиснути Enter (лістинг 11.2).

Лістинг 11.2. Макровизначення з параметрами

```
#include <stdio.h>

#define PRINT() \
  printf("Визначення макросів.\n"); \
  printf("Багаторядкове макровизначення.\n")

int main() {
    PRINT();
}
```

Результат виконання програми:

```
Визначення макросів.
Багаторядкове макровизначення.
```

Якщо текст, пов'язаний з ідентифікатором, відсутній, цей ідентифікатор видаляється з програми. Проте, під час перевірки умови

#if, цей ідентифікатор вважається визначеним та дає значення 1 (лістинг 11.3).

Лістинг 11.3. Макровизначення з параметрами

```
#include <stdio.h>
#define PRINT

int main() {
    #if defined(PRINT)
           printf("true1\n");
    #endif

    #ifdef PRINT
           printf("true2\n");
    #endif
}
```

Результат виконання програми:

```
true1
true2
```

Макровизначення може містити список параметрів з одного або декількох значень, які слід розділяти комами у разі множинного використання, вони повинні бути унікальними, а їх область дії обмежується межами макроса. Список параметрів записується в круглих дужках. У тексті макровизначення імена формальних параметрів позначають місця, де будуть підставлятися фактичні аргументи під час виклику макросу. Кожен формальний параметр може зустрічатися у тексті необмежену кількість разів (лістинг 11.4).

Лістинг 11.4. Макровизначення зі списком параметрів

```
#include <stdio.h>
#define DOB(x, y) ((x) * (y))
int main() {
    int a = 5, b=15;
    printf("Rezult = %d", DOB(a, b));
}
```

Результат виконання програми:

```
Rezult = 75
```

В макровиклику вслід за *ідентифікатором DOB* в круглих дужках записується список фактичних аргументів *(x, y)*. *Текст* модифікується шляхом заміни кожного формального параметра на відповідний фактичний параметр *(a, b)*. Списки фактичних і формальних параметрів повинні мати одне і те ж число елементів.

У мові С макроси, визначені через *#define*, просто підставляють код без перевірки на логіку виконання. Це може призводити до помилок, якщо передані аргументи мають «побічні ефекти» – тобто змінюють значення змінних або виконують операції, що впливають на стан програми (лістинг 11.5).

Лістинг 11.5. Приклад макровизначення з «побічним ефектом»

```
#include <stdio.h>
#define SQUARE(x) (x * x)

int main() {
    int a = 3;
    printf("Rezult = %d\n", SQUARE(a + 1));
    return 0;
}
```

Результат виконання програми:

```
Rezult = 7
```

В результаті виконання макропідстановки в лістингу 11.5 очікуємо, що вираз (3+1) * (3+1) дасть значення рівне 16. Однак після підстановки макроса обчислення виглядатиме наступним чином (a + 1 * a + 1). Операція множення має вищий пріоритет, тому фактично виконується a + (1 * a) + 1, що дає 7, а не 16.

Макроси можуть створювати вкладені структури, тобто одні макроси можуть бути визначені в межах іншого макросу. Після того, як виконана макропідстановка, отриманий рядок знову переглядається для пошуку інших імен констант і макровизначень. При повторному перегляді не розглядається ім'я раніше проведеної макропідстановки.

Тому директива ***#define a a*** не призведе до зациклювання препроцесора.

Приклад:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

В наведеному прикладі ідентифікатор WIDTH визначається як ціла константа яка дорівнює 80, а ідентифікатор LENGTH включає в себе (WIDTH+10). Кожне входження LENGTH буде замінено на (WIDTH+10), який перетвориться на вираз (80+10). Дужки дозволяють уникнути помилок (лістинг 11.6).

Лістинг 11.6. Приклад вкладених макросів

```
#include <stdio.h>

#define WIDTH 80
#define LENGTH (WIDTH+10)

int main() {
    int val = LENGTH * 20;
    printf("val = %d\n", val);
    return 0;
}
```

Результат виконання програми:

```
val = 1800
```

В лістингу 11.6 якщо будуть відсутні дужки LENGTH WIDTH+10 значення *val* буде дорівнювати 280 (***val=80+10*20).***

Приклад:

```
#define MAX(x,y) ((x)>(y))?(x):(y)
```

В наведеному прикладі визначається макрос MAX. Кожне входження ідентифікатора MAX в тексті програми буде замінено на

вираз ((x)>(y))?(x):(y), в якому замість формальних параметрів *x* та *y* підставляються фактичні, де MAX(1,2) заміниться на вираз ((1)>(2))?(1):(2).

В лістингу 11.7 продемонстровано використання препроцесора С для визначення констант, макросів та умовної компіляції.

Лістинг 11.7. Приклад використання різних типів макросів

```
#include <stdio.h>

#define PI 3.1416

#define SQUARE(x) ((x) * (x))
#define CUBE(x) ((x) * (x) * (x))

#define MIN(a, b) ((a) < (b) ? (a) : (b))

#ifdef DEBUG
#define DEBUG_PRINT(msg) printf("DEBUG: %s\n", msg)
#else
#define DEBUG_PRINT(msg)
#endif

int main() {
    printf("PI = %f\n", PI);
    printf("SQUARE(5) = %d\n", SQUARE(5));
    printf("CUBE(3) = %d\n", CUBE(3));

    DEBUG_PRINT("Program started");

    printf("MIN(10, 20) = %d\n", MIN(10, 20));
    return 0;
}
```

Результат виконання програми:

```
PI = 3.141600
SQUARE(5) = 25
CUBE(3) = 27
MIN(10, 20) = 10
```

Розглянемо детальніше наведений приклад (лістинг 11.7). Макрос *#define PI 3.1416* використовується для визначення константи PI зі значенням 3.1416.

Макрос *#define MIN(a, b) ((a) < (b) ? (a) : (b))* використовується для знаходження мінімального числа з двох введених. Макроси *#define SQUARE(x) ((x) * (x))* і *#define CUBE(x) ((x) * (x) * (x))* визначено для обчислення квадрата числа та куба числа. Зверніть увагу на використання дужок *((x) * (x))*, вони важливі для запобігання неочікуваним результатам, якщо аргумент макросу є виразом.

Макрос DEBUG_PRINT дозволяє включати або виключати налагоджувальні повідомлення залежно від режиму компіляції. #ifdef DEBUG – це директива препроцесора, яка перевіряє, чи визначено макрос DEBUG. Якщо DEBUG визначено, то виконується код між #ifdef та #else. Якщо DEBUG не визначено, то виконується код між #else та #endif. У цьому випадку, якщо DEBUG визначено, то макрос DEBUG_PRINT виводить повідомлення налагодження. Якщо DEBUG не визначено, то DEBUG_PRINT не робить нічого. Це дозволяє додавати код налагодження, який буде скомпільований лише тоді, коли буде явно вказано програмістом.

Директива #undef

Синтаксис :

```
#undef  ідентифікатор
```

У мові С визначення символічних констант та макросів можуть бути скасовані за допомогою директиви #undef. Це означає, що область дії (життєвий цикл) символічної константи або макросу обмежена, вона починається з моменту їх оголошення (#define) і завершується або командою скасування (#undef), або досягає кінця файлу. Після того, як константу або макрос було скасовано (#undef), ідентифікатор може бути використаний знову і оголошений за допомогою #define (лістинг 11.7).

Лістинг 11.7. Приклад використання директиви #undef

```
#include <stdio.h>

#define WIDTH 80
#define LENGTH (WIDTH+10)
```

```
int main() {
    int val = LENGTH * 20;
    printf("val = %d\n", val);

#undef WIDTH
#define WIDTH 20
    val = LENGTH * 20;
    printf("val = %d\n", val);
    return 0;
}
```

Результат виконання програми:

```
val = 1800
val = 600
```

Директиви #if, #elif, #else, #endif

Умовна компіляція дозволяє програмісту контролювати виконання директив препроцесора та компіляцію коду. Кожна умовна директива препроцесора оцінює значення цілочисельного константного виразу. Директива препроцесора #if подібна до умовного оператора if і має наступний синтаксис:

```
#if  умова
...
[ #elif  умова
...]
[ #elif  умова
...]
[ #else
...]
#endif
```

Умова – це цілочисельний вираз. Якщо цей вираз повертає не нуль (істинна), то фрагмент коду, що розташований між *#if* і *#endif*, компілюється. Якщо вираз повертає нуль (хиба), то цей фрагмент коду ігнорується препроцесором і компілятором.

В умовах, окрім звичайних виразів, можна використовувати конструкцію *defined (ідентифікатор)*, яка повертає 1, якщо вказаний ідентифікатор раніше був визначений *#define*, і 0 – в протилежному випадку.

Директива *#elif* може зустрічатись кілька разів. Директива *#elif* після директиви *#else* не використовується, неї слідує тільки директива *#endif*.

Приклад:

```
#if defined (CREDIT)
    credit();
#elif defined (DEBIT)
    debit();
#else
    printerror();
#endif
```

В наведеному прикладі директиви препроцесора #if, #elif, #else, #endif використовуються для вибору однієї з трьох викликів функцій. Виклик функції *credit*() відбудеться, якщо визначена іменована константа CREDIT. Якщо визначена іменована константа DEBIT, то скомпілюється виклик функції *debit*(). Якщо жодна із наведених іменованих констант не визначена, то скомпілюється виклик функції *printerror*().

Приклад:

```
#if DLEVEL>5
#define SIGNAL 1
#if STACKUSE == 1
#define STACK 200
#else
#define STACK 100
#endif
#else
#define SIGNAL 0
#if STACKUSE == 1
#define STACK 100
#else
#define STACK 50
#endif
#endif
```

В наведеному прикладі продемонстровано вкладені умовні директиви #if, #else, #endif. Перший набір директив оброблюється,

якщо значення DLEVEL більше за 5. В протилежному випадку оброблюється другий набір.

Директиви #ifdef і #ifndef

Директиви препроцесора #ifdef і #ifndef використовуються для умовної компіляції коду. #ifdef перевіряє, чи визначений макрос, #ifndef перевіряє, чи не визначений макрос. Синтаксис:

```
#ifdef  ідентифікатор
#ifndef ідентифікатор
```

Директиви використовуються для включення або виключення певних фрагментів коду або захисту заголовочних файлів від повторного включення. Приклад використання директиви #ifdef наведено в лістингу 11.8.

Лістинг 11.8. Приклад використання директиви #ifdef

```
#include <stdio.h>
#define DEBUG

int main() {
    #ifdef DEBUG
       printf("Режим налагодження увімкнено.\n");
    #endif
       printf("Програма виконується.\n");
     return 0;
}
```

Результат виконання програми:

```
Режим налагодження увімкнено.
Програма виконується.
```

Коли препроцесор оброблює директиву #ifdef, він перевіряє, чи визначений в даний момент вказаний ідентифікатор. Якщо так, то умова вважається істинною, якщо ні – хибною.

Директива #ifndef протилежна за своєю дією директиві #ifdef. Якщо ідентифікатор не був визначений директивою #define, або його дія відмінена директивою #undef, то умова вважається істинною. В протилежному випадку умова хибна.

Приклад використання директиви #ifndef наведено в лістингу 11.9.

Лістинг 11.9. Приклад використання директиви #ifndef

```
#include <stdio.h>
#define RELEASE

int main() {
    #ifndef DEBUG
        printf("Режим налагодження вимкнено.\n");
    #endif
        printf("Програма працює у стандартному режимі.\n");
     return 0;
}
```

Результат виконання програми:

```
Режим налагодження вимкнено.
Програма працює у стандартному режимі.
```

В лістингу 11.9 директива #ifndef перевіряє, чи не визначений макрос DEBUG. Якщо додати #define DEBUG, то printf не відпрацює і повідомлення "Режим налагодження вимкнено." виведено не буде.

Директиви #ifdef і #ifndef працюють аналогічно директиві #if, #elif, вони можуть використовуватись разом та створювати вкладені структури. Набір директив повинен закінчуватися директивою #endif (лістинг 11.10).

Лістинг 11.10. Приклад використання вкладених директив

```
#include <stdio.h>
#define WINDOWS

int main()
{
    #ifdef WINDOWS
    #define OS_TYPE "Windows"
```

```
    #elif defined(LINUX)
    #define OS_TYPE "Linux"
    #else
    #define OS_TYPE "Unknown"
    #endif
     printf("Операційна система: %s\n", OS_TYPE);
     return 0;
}
```

Результат виконання програми:

```
Операційна система: Windows
```

В лістингу 11.10 використовується умовна компіляція. Оскільки визначений макрос WINDOWS, OS_TYPE присвоюється "Windows". Якщо б був визначений макрос LINUX, то було б присвоєно OS_TYPE – "Linux". Якщо жоден із них не був би визначений, OS_TYPE був би визначений як "Unknown".

Директива #line

Директива *#line* використовується для зміни номерів рядків та імен файлів у повідомленнях компілятора. Ці зміни впливають лише на те як обробляє компілятор код, не змінюючи самої програми. Синтаксис:

```
#line  константа  ["ім'я_файла"]
```

Директива *#line* дозволяє керувати відображенням помилок та діагностичних повідомлень, що корисно при генерації коду або налагодженні. Після обробки чергового рядка лічильник номера рядка збільшується на одиницю. У випадку зміни номера рядка й імені файла програми директивою #line, компілятор продовжує роботу вже з новими значеннями.

Поточний номер рядка і ім'я файла програми доступні в програмі через псевдозмінні з іменами __LINE__ і __FILE__. Ці псевдозмінні можуть використовуватися для виведення під час виконання програми повідомлень про точне місцезнаходження помилки. Значенням __FILE__ зберігає рядок з шляхом до виконуваного файлу та його ім'я. В лістингу 11.11 продемонстровано використання псевдозмінних __LINE__ і __FILE__.

Лістинг 11.11. Приклад використання псевдозмінних __LINE__ і __FILE__

```
#include <stdio.h>
void print_location(const char* message)
{
    fprintf(stderr, "Помилка: %s, файл: %s, рядок: %d\n",
message, __FILE__, __LINE__);
}

int main()
{
    float result = 0;
    for (int i = -3; i < 3; i++)
    {
        if (i == 0) {
        print_location("Ділення на нуль");
        continue;
        }
        result += 1.0 / i;
        printf("Програма продовжує виконання.\n");
    }
    printf("result = %f\n", result);
    return 0;
}
```

Результат виконання програми:

```
Програма продовжує виконання.
Програма продовжує виконання.
Програма продовжує виконання.
Помилка: Ділення на нуль, файл:
C:\Users\admin\source\repos\ConsoleApplication1\ConsoleApplication1\Co
nsoleApplication1.cpp, рядок: 18
Програма продовжує виконання.
Програма продовжує виконання.
result = -0.333333
```

Контрольні запитання та завдання

1. Що таке макрос в мові С, та як його визначити?

2. Що таке директива #include?
3. Який синтаксис директиви #include?
4. Як працює директива #include під час компіляції?
5. Яка різниця між підключенням файлів заголовків <name1.h> та "name2.h"?
6. Що таке директива #define?
7. Який синтаксис #define?
8. Які переваги використання #define?
9. Як відрізнити макрос від функції?
10. Для чого використовується #undef?
11. Чи можна використовувати #undef для стандартних макросів?
12. Що станеться, якщо включити один і той самий файл кілька разів?
13. Що таке препроцесор і як він пов'язаний з директивами #if, #elif, #else, #endif?
14. Опишіть синтаксис використання директив #if, #elif, #else, #endif?
15. Що таке умовна компіляція і як вона працює?
16. Які оператори можна використовувати в умовах директив #if та #elif?
17. Чи можна використовувати макроси в умовах директив #if та #elif?
18. Чи можна вкладати директиви #if, #elif, #else, #endif одна в одну?
19. Що таке директива #ifndef?
20. Який синтаксис використання #ifdef та #ifndef?
21. Чи можна використовувати #ifdef та #ifndef з #else?
22. Де найчастіше використовуються #ifdef та #ifndef?
23. Які переваги використання #ifdef та #ifndef?
24. Які недоліки використання #ifdef та #ifndef?
25. Чи існують альтернативи використанню #ifdef та #ifndef?
26. Напишіть програму в якій створіть макрос PRINT_VALUES, який приймає змінну кількість аргументів та виводить їх значення.
27. Визначте макроси для числа Пі (π), числа Ейлера (e) та швидкості світла (c). Використайте ці макроси для обчислення площі кола, об'єму кулі та енергії фотона.
28. Визначте макроси для обчислення квадрата числа, куба числа та максимального з двох чисел. Використайте ці макроси для обчислення значень функцій.

29. Визначено два макроси:

    #define byte unsigned char

    #define word unsigned short

    Поясніть чи можливе використання цих макросів в програмі? Якщо так, то для чого?

30. Створіть макрос IS_EVEN(x), який повертає 1, якщо число x парне, і 0, якщо непарне.

31. Визначте макрос AREA_CIRCLE(r), який обчислює площу кола з радіусом r.

32. Визначте макрос SWAP(a, b), який міняє місцями значення двох змінних a та b.

33. Визначте макрос PRINT_ARRAY(arr, size), який виводить на екран елементи масиву arr розміром size.

34. Визначте макрос PRINT_VAR(var), який виводить на екран ім'я змінної var та її значення.

35. Визначте макрос CONCAT(a, b), який об'єднує два ідентифікатори a та b в один.

36. Визначте макрос PRINT_MATRIX(matrix, rows, cols), який виводить на екран матрицю matrix розміром rows і cols.

37. Визначте макрос DEBUG, який дозволяє вмикати/вимикати виведення налагоджувальної інформації.

38. Використайте макрос DEBUG, щоб вивести значення змінної лише в режимі налагодження.

39. Визначте макрос OS_WINDOWS, який дозволяє компілювати код лише для операційної системи Windows.

40. Напишіть програму, яка використовує макроси для обчислення:

    - квадрата числа;

    - максимального значення між двома числами;

    - мінімального значення між двома числами.

    Використовуйте #define з inline-виразами.

12. Динамічні структури даних

Попри те, що терміни «тип даних» і «структура даних» мають схоже звучання, їх значення суттєво відрізняються. Як було зазначено раніше, тип даних визначає множину значень, які може приймати змінна певного типу. Натомість структури даних є організованими наборами даних, які можуть включати елементи різних типів, об'єднані певним чином.

Основним компонентом структури даних є елемент (вузол), що зберігає конкретний тип даних. Якщо елементи пов'язані між собою за допомогою покажчиків, така організація даних отримує назву динамічних структур, оскільки їх розмір може змінюватися під час виконання програми.

До найбільш популярних динамічних структур відносяться лінійні списки, стеки, черги та бінарні дерева.

Лінійні списки

Лінійні списки - це динамічні структури даних, які дозволяють ефективно зберігати та обробляти набори елементів. На відміну від статичних масивів, розмір лінійних списків може змінюватися під час виконання програми, що робить їх дуже гнучкими. Основна одиниця лінійного списку - вузол, яка містить дані та посилання на наступний вузол. Кількість елементів у лінійному списку називається довжиною списку.

Приклад:

```
F=(1,2,3,4,5,6)  /* довжина лінійного списку F дорівнює 6 */
```

Основні операції з лінійними списками:

- додавання елемента в початок списку;
- вилучення елемента з початку списку;
- додавання елемента в будь-яке місце списку;
- вилучення елемента з будь-якого місця списку;
- перевірка, чи порожній список;
- очистка списку;
- друк списку.

Основні способи зберігання лінійних списків можна поділити на послідовне та зв'язане зберігання.

Послідовне зберігання списків передбачає використання масиву елементів певного типу, а також змінної, яка містить поточну кількість елементів у списку.

Приклад:

```
#define MAX 100        /* максимально можлива довжина списку */
typedef struct
{
  int x; /* тут потрібно описати структуру елементів списку*/
} elementtype;


typedef struct
{
  elementtype elements[MAX];
  int count;
} listtype;
```

У прикладі описуються типи даних *elementtype* (визначає структуру елемента списку) та *listtype* (містить масив для зберігання елементів та змінну для зберігання поточного розміру списку).

Розглянемо приклади реалізації основних операцій над списками.

1. Ініціалізація списку.

Приклад:

```
void list_reset(listtype *list)
{
  list->count=0;
};
```

2. Додавання нового елементу у кінець списку.

Приклад:

```
void list_add(listtype *list, elementtype element)
{
  if (list->count==MAX) return;
```

```
 list->elements[list->count++]=element;
};
```

3. Додавання нового елементу в позицію pos.

На рисунку 12.1 показано масив з 15 елементів. Щоб звільнити місце для нового елемента, елементи, що знаходяться після позиції вставки, були зсунуті на одну позицію вправо, на рисунку це показано стрілками. Після додавання елемента розмір масиву збільшився на 1, тепер список містить 16 елементів.

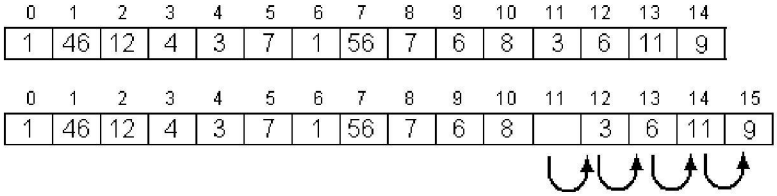


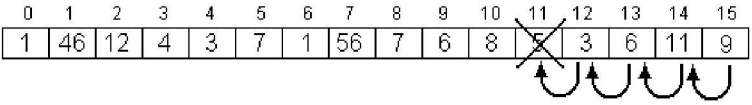
Рисунок 12.1. Додавання нового елемента в задану позицію

Приклад:

```
void list_insert(listtype *list,int pos,elementtype element)
{
 int j;
 if (pos<0||pos>list->count||pos>=MAX) return;
 for (j=list->count;j>pos;j--)
 {
  list->elements[j]=list->elements[j-1];
 };
 list->elements[j]=element;
 list->count++;
};
```

4. Вилучення елемента з номером pos.

На рисунку 12.2 зображено вилучення елемента під індексом 11 зі списку. Вилучення елемента зі списку передбачає зсув елементів в ліво на одну позицію.



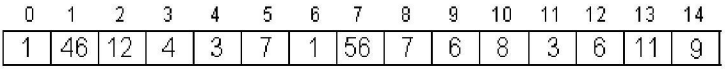


Рисунок 12.2. Схема вилучення елемента зі списку

Приклад:

```
void list_delete(listtype *list,int pos)
{
  int j;
  if (pos<0||pos>list->count) return;
  for (j=pos+1;j<list->count;j++)
  {
    list->elements[j-1]=list->elements[j];
  };
  list->count--;
};
```

5. Отримання елемента з номером pos.

Приклад:

```
int list_get(listtype *list,int pos,elementtype *element)
{
  if (pos<0 || pos>list->count)
  {
    return 0;
  };
  *element=list->elements[pos];
  return 1;
};
```

В лістингу 12.1 наведено приклад роботи з лінійними списками при послідовному зберіганні елементів.

Лістинг 12.1. Робота з лінійними списками – послідовне зберігання

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

#define MAX 100

typedef struct {
```

```
    int x;
} elementtype;

typedef struct {
    elementtype elements[MAX];
    int count;
} listtype;

void list_reset(listtype* list) {
    list->count = 0;
}

void list_insert(listtype* list, int pos, elementtype
element) {
    int j;
    if (pos < 0 || pos > list->count || pos >= MAX) return;
    for (j = list->count; j > pos; j--) {
        list->elements[j] = list->elements[j - 1];
    }
    list->elements[pos] = element;
    list->count++;
}

void list_delete(listtype* list, int pos) {
    int j;
    if (pos < 0 || pos >= list->count) return;
    for (j = pos + 1; j < list->count; j++) {
        list->elements[j - 1] = list->elements[j];
    }
    list->count--;
}

int list_get(listtype* list, int pos, elementtype* element)
{
    if (pos < 0 || pos >= list->count) {
        return 0;
    }
    *element = list->elements[pos];
    return 1;
}

void list_print(listtype* list) {
    int i;
    printf("Список: [");
```

```
    for (i = 0; i < list->count; i++) {
        printf("%d", list->elements[i].x);
        if (i < list->count - 1) {
            printf(", ");
        }
    }
    printf("]\n");
}

int main() {
    listtype my_list;
    elementtype element;
    int pos;
    list_reset(&my_list);

    // Додавання елементів
    element.x = 10;
    list_insert(&my_list, 0, element);
    element.x = 20;
    list_insert(&my_list, 1, element);
    element.x = 30;
    list_insert(&my_list, 2, element);
    element.x = 40;
    list_insert(&my_list, 1, element); // Вставка в
середину

    list_print(&my_list);

    // Отримання елемента
    pos = 2;
    if (list_get(&my_list, pos, &element)) {
        printf("Елемент на позиції %d: %d\n", pos,
element.x);
    }
    else {
        printf("Помилка: позиція %d недійсна\n", pos);
    }

    // Видалення елемента
    pos = 1;
    list_delete(&my_list, pos);

    list_print(&my_list);
```

```
    // Спроба отримати елемент за недійсним індексом
    pos = 10;
    if (list_get(&my_list, pos, &element)) {
        printf("Елемент на позиції %d: %d\n", pos,
element.x);
    }
    else {
        printf("Помилка: позиція %d недійсна\n", pos);
    }
    return 0;
}
```

Результат виконання програми:

```
Список: [10, 40, 20, 30]
Елемент на позиції 2: 20
Список: [10, 20, 30]
Помилка: позиція 10 недійсна
```

При послідовному зберіганні списків за допомогою масивів елементи зберігаються в одному масиві. Така організація дозволяє легко переглядати вміст списку та додавати нові елементи в його кінець. Однак операції вставки елементів у середину списку або видалення елементів з середини вимагають зсуву всіх наступних елементів. Зі збільшенням кількості елементів у масиві зростає і кількість операцій, необхідних для оновлення списку.

Зв'язане зберігання лінійних списків. Найпростіший спосіб зв'язати елементи – це додати посилання кожного елемента на наступний. Такий список називається односпрямованим (однозв'язаним). Якщо додати посилання і на попередній елемент, отримуємо двозв'язаний список. Коли перший і останній елементи списку зв'язані між собою, це вже кільцевий список.

Існує три основних типи лінійних списків односпрямовані, двоспрямовані та кільцеві (рис. 12.3). Кожен елемент односпрямованого списку (рис. 12.3 (а)) містить два поля: дані та покажчик на наступний елемент. Останній елемент містить покажчик що вказує на NULL. Переміщення в такому списку можливе лише в одному напрямку.

Кожен елемент двоспрямованого списку (рис. 12.3 (б)) містить три поля: дані, покажчик на наступний елемент та покажчик на попередній елемент. В даному списку можливе переміщення в обох напрямках.

Кожен елемент кільцевого списку (рис. 12.3 (в)) містить два поля: дані та покажчик на наступний елемент. В даному списку можливо переміщуватись по колу, тобто останній елемент містить покажчик що вказує на перший елемент.

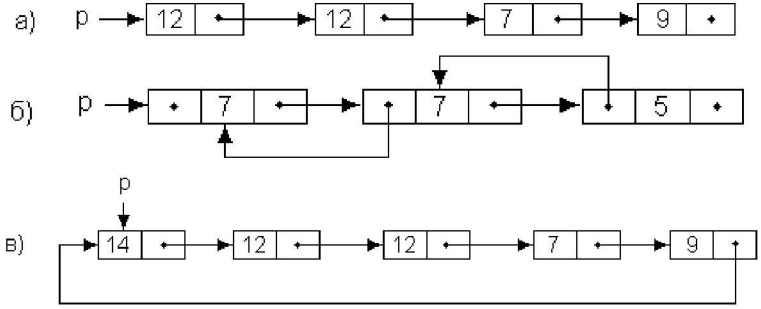


Рисунок 12.3. Схематична структура односпрямованого (а), двоспрямованого (б), та кільцевого списків (в)

Структуру даних для зберігання односпрямованого лінійного списку у вигляді програмного коду наведено в прикладі.

Приклад:

```
typedef long elemtype;
typedef struct node
{
  elemtype val;
  struct node *next;
} list;
```

В даному фрагменті програми описуються декілька типів даних:

elemtype – визначає тип даних лінійного списку. Можна використовувати будь-який стандартний тип даних, включаючи структури.

list – визначає структуру елемента лінійного списку (*val* – значення, яке зберігається у вузлі, *next* – покажчик на наступний вузол).

Схематично лінійний односпрямований список представлено на рисунку 12.4.

Рисунок 12.4. Схематичне зображення односпрямованого лінійного списку

Розглянемо приклади реалізації основних операцій над односпрямованим лінійним списком.

1. Включення елемента в початок списку.

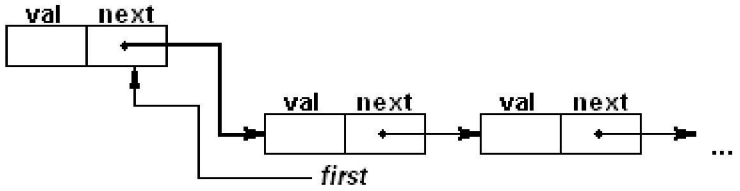


Рисунок 12.5. Схема дії операції включення елемента в початок списку

Приклад:

```
list *addbeg(list *first, elemtype x){
    list *vsp;
    vsp = (list *) malloc(sizeof(list));
    vsp->val=x;
    vsp->next=first;
    first=vsp;
     return first;
}
```

2. Видалення елемента з початку списку.

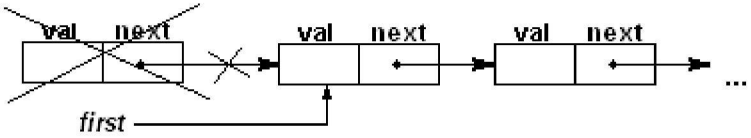


Рисунок 12.5. Схема дії операції видалення елемента

Приклад:

```
list *delbeg(list *first)
{    list *vsp;
     vsp=first->next;
     free(first);
     return vsp;
}
```

3. Включення нового елемента у список.

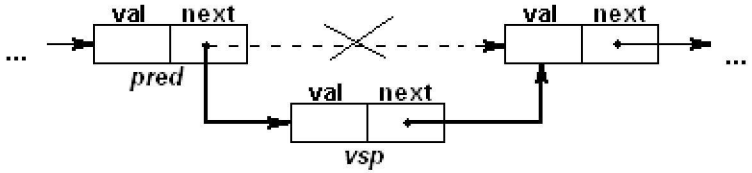


Рисунок 12.5. Схема включення нового елемента у список

Приклад:

```
list *add(list *pred, elemtype x)
{  list *vsp;
   vsp = (list *) malloc(sizeof(list));
   vsp->val=x;
   vsp->next=pred->next;
   pred->next=vsp;
   return vsp;
}
```

4. Видалення елемента зі списку.

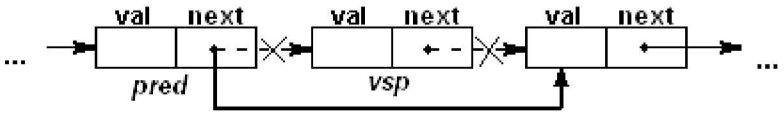


Рис. 1.31. Схема вилучення елемента зі списку

Приклад:

```
elemtype del(list *pred)
{  elemtype x;
   list *vsp;
   vsp=pred->next;
   pred->next=pred->next->next;
   x=vsp->val;
   free(vsp);
   return x;
}
```

5. Друк значень списку.

Приклад:

```
void print(list *first)
{ list *vsp;
  vsp=first;
  while (vsp)
  {
    printf("%i\n",vsp->val);
    vsp=vsp->next;
  }
}
```

6. Перевірка, чи порожній список

Приклад:

```
int pust(list *first)
{
     return !first;
}
```

7. Знищення списку

Приклад:

```
list *kill(list *first) {
```

```
  while (!pust(first)) first=delbeg(first);
  return first;
}
```

В лістингу 12.2 наведено приклад роботи з односпрямованим лінійним списком.

Лістинг 12.2. Робота з односпрямованим лінійним списком

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

typedef int elemtype; // Визначення типу елементів списку
typedef struct list {
    elemtype val;
    struct list* next;
} list;

// Функція для додавання елемента на початок списку
list* addbeg(list* first, elemtype x) {
    list* vsp;
    vsp = (list*)malloc(sizeof(list));
    vsp->val = x;
    vsp->next = first;
    first = vsp;
    return first;
}

// Функція для видалення елемента з початку списку
list* delbeg(list* first) {
    list* vsp;
    if (first == NULL) return NULL; // Перевірка на
порожній список
    vsp = first->next;
    free(first);
    return vsp;
}

// Функція для додавання елемента після заданого вузла
list* add(list* pred, elemtype x) {
    list* vsp;
    vsp = (list*)malloc(sizeof(list));
    vsp->val = x;
```

```
    vsp->next = pred->next;
    pred->next = vsp;
    return vsp;
}

// Функція для видалення елемента після заданого вузла
elemtype del(list* pred) {
    elemtype x;
    list* vsp;
    if (pred == NULL || pred->next == NULL) return -1; //
Перевірка на коректність вузла
    vsp = pred->next;
    pred->next = pred->next->next;
    x = vsp->val;
    free(vsp);
    return x;
}

// Функція для виведення списку
void print(list* first) {
    list* vsp;
    vsp = first;
    while (vsp) {
        printf("%d ", vsp->val);
        vsp = vsp->next;
    }
    printf("\n");
}

// Функція для перевірки, чи порожній список
int pust(list* first) {
    return !first;
}

// Функція для знищення списку
list* kill(list* first) {
    while (!pust(first)) first = delbeg(first);
    return first;
}

// Функція для вставки елемента в задану позицію
list* insertAtPosition(list* first, elemtype x, int
position) {
    if (position < 0) return first; // Некоректна позиція
```

```
    list* newNode = (list*)malloc(sizeof(list));
    if (newNode == NULL) return first; // Перевірка на
виділення пам'яті

    newNode->val = x;

    if (position == 0) {
        newNode->next = first;
        return newNode;
    }

    list* current = first;
    for (int i = 0; i < position - 1 && current != NULL;
i++) {
        current = current->next;
    }

    if (current == NULL) {
        free(newNode); // Позиція за межами списку
        return first;
    }

    newNode->next = current->next;
    current->next = newNode;
    return first;
}

int main() {
    list* myList = NULL;
    myList = addbeg(myList, 5);
    myList = addbeg(myList, 10);
    myList = addbeg(myList, 15);

    printf("Список до вставки: ");
    print(myList);

    myList = insertAtPosition(myList, 20, 1); // Вставляємо
20 у позицію 1
    printf("Список після вставки: ");
    print(myList);

    myList = insertAtPosition(myList, 30, 4); // Вставляємо
30 у позицію 4
```

```
    printf("Список після вставки: ");
    print(myList);
    myList = kill(myList); // Очищаємо список
    return 0;
}
```

Результат виконання програми:

```
Список до вставки: 15 10 5
Список після вставки: 15 20 10 5
Список після вставки: 15 20 10 5 30
```

Стеки

Стек – це динамічна структура даних, що являє собою впорядковану сукупність елементів, у якій додавання нових і видалення існуючих елементів здійснюється лише з одного кінця, що називається вершиною стека.

Стек працює за принципом LIFO (Last In – First Out, «останнім зайшов – першим вийшов»). Яскравим прикладом такої організації є дитяча пірамідка, де кільця знімаються і додаються зверху, дотримуючись цього принципу.

Основні операції, що виконуються над стеком:

- додавання елемента;
- видалення елемента;
- перевірка, чи стек порожній;
- перегляд верхнього елемента без його видалення;
- очищення стека.

Стек створюється аналогічно лінійному списку, оскільки є його окремим випадком і базується на односпрямованій структурі. В лістингу 12.3 наведено приклад роботи зі стеком.

Лістинг 12.2. Робота зі стеком

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

typedef long elemtype;
```

```
typedef struct node {
    elemtype val;
    struct node* next;
} stack;

// Початкове формування стеку
stack* first(elemtype d) {
    stack* pv = (stack*)calloc(1, sizeof(stack));
    if (pv == NULL) {
        fprintf(stderr, "Помилка виділення пам'яті.\n");
        exit(1);
    }
    pv->val = d;
    pv->next = NULL;
    return pv;
}

// Занесення елемента в стек
void push(stack** top, elemtype d) {
    stack* pv = (stack*)calloc(1, sizeof(stack));
    if (pv == NULL) {
        fprintf(stderr, "Помилка виділення пам'яті.\n");
        exit(1);
    }
    pv->val = d;
    pv->next = *top;
    *top = pv;
}

// Вилучення елемента зі стека
elemtype pop(stack** top) {
    if (*top == NULL) {
        fprintf(stderr, "Стек порожній.\n");
        exit(1);
    }
    elemtype temp = (*top)->val;
    stack* pv = *top;
    *top = (*top)->next;
    free(pv);
    return temp;
}
```

```
int main() {
    stack* my_stack = NULL;

    // Додавання елементів до стеку
    push(&my_stack, 10);
    push(&my_stack, 20);
    push(&my_stack, 30);

    // Вилучення елементів зі стеку та виведення їх
    printf("Вилучений елемент: %ld\n", pop(&my_stack));
    printf("Вилучений елемент: %ld\n", pop(&my_stack));
    printf("Вилучений елемент: %ld\n", pop(&my_stack));
    // Спроба вилучення з порожнього стеку (виведе помилку)
    pop(&my_stack);
    return 0;
}
```

Результат виконання програми:

```
Вилучений елемент: 30
Вилучений елемент: 20
Вилучений елемент: 10
Стек порожній.
```

Контрольні запитання та завдання

1. Що таке лінійний список і які його основні типи?
2. Як реалізувати однозв'язний список мовою С?
3. Як реалізувати двозв'язний список мовою С?
4. Як реалізувати кільцевий список мовою С?
5. Як додати елемент на початок лінійного списку?
6. Як додати елемент в кінець лінійного списку?
7. Як додати елемент в задану позицію лінійного списку?
8. Як видалити елемент з початку лінійного списку?
9. Як видалити елемент з кінця лінійного списку?
10. Як видалити елемент з заданої позиції лінійного списку?
11. Як очистити лінійний список?
12. Що таке стек і які основні операції з ним пов'язані?

13. Як перевірити, чи стек порожній?
14. Як реалізувати функцію push для додавання елемента в стек?
15. Як реалізувати функцію pop для видалення елемента зі стека?
16. Як реалізувати функцію peek для перегляду верхнього елемента стека без його видалення?
17. Чим відрізняються лінійний список і стек?
18. Напишіть функцію, яка приймає рядок, що містить дужки (), [], {}. Функція повинна використовувати стек для перевірки, чи збалансовані дужки (кожна відкриваюча дужка повинна мати відповідну закриваючу). Поверніть 1, якщо дужки збалансовані, і 0 в іншому випадку.
19. Створіть структуру даних «лінійний список», використовуючи вузли (структури з даними та покажчиком на наступний вузол). Реалізуйте функції insert (вставити елемент), delete (видалити елемент), search (пошук елемента) та printList (вивести список). Протестуйте список, додавши, видаливши та знайшовши кілька елементів.
20. Створіть мінікалькулятор використовуючи структуру даних «стек».

13. Довідково-інформаційні дані

Таблиця символів ASCII

Для позначення символів використовується американський національний стандартний код для обміну інформацією ASCII (American Standard Code for Information Interchange). Відповідно до нього код символу зберігається в одному байті, тому коди символів можуть приймати значення від 0 до 255. Всього існує 256 символів (таблиця 13.1)

Таблиця 13.1.

Таблиця символів ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	☺ 1	☻ 2	♥ 3	♦ 4	♣ 5	♠ 6	• 7	◘ 8	○ 9	◙ 10	♂ 11	♀ 12	♪ 13	♫ 14	☼ 15
1	► 16	◄ 17	↕ 18	‼ 19	¶ 20	§ 21	▬ 22	↨ 23	↑ 24	↓ 25	→ 26	← 27	∟ 28	↔ 29	▲ 30	▼ 31
2	32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	– 45	. 46	/ 47
3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
6	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	⌂ 127
8	А 128	Б 129	В 130	Г 131	Д 132	Е 133	Ж 134	З 135	И 136	Й 137	К 138	Л 139	М 140	Н 141	О 142	П 143
9	Р 144	С 145	Т 146	У 147	Ф 148	Х 149	Ц 150	Ч 151	Ш 152	Щ 153	Ъ 154	Ы 155	Ь 156	Э 157	Ю 158	Я 159
A	а 160	б 161	в 162	г 163	д 164	е 165	ж 166	з 167	и 168	й 169	к 170	л 171	м 172	н 173	о 174	п 175
B	░ 176	▒ 177	▓ 178	│ 179	┤ 180	╡ 181	╢ 182	╖ 183	╕ 184	╣ 185	║ 186	╗ 187	╝ 188	╜ 189	╛ 190	┐ 191
C	└ 192	┴ 193	┬ 194	├ 195	─ 196	┼ 197	╞ 198	╟ 199	╚ 200	╔ 201	╩ 202	╦ 203	╠ 204	═ 205	╬ 206	╧ 207
D	╨ 208	╤ 209	╥ 210	╙ 211	╘ 212	╒ 213	╓ 214	╫ 215	╪ 216	┘ 217	┌ 218	█ 219	▄ 220	▌ 221	▐ 222	▀ 223
E	р 224	с 225	т 226	у 227	ф 228	х 229	ц 230	ч 231	ш 232	щ 233	ъ 234	ы 235	ь 236	э 237	ю 238	я 239
F	Ё 240	ё 241	Є 242	є 243	Ї 244	ї 245	÷ 246	≈ 247	° 248	∙ 249	· 250	√ 251	ⁿ 252	² 253	■ 254	255

Розширені коди клавіатури

Як відомо, функція *getch()* повертає код натиснутої клавіші. Нульове значення сигналізує про натискання спеціальної клавіші або комбінації спеціальних клавіш. Отже, якщо *getch()* повертає нуль, то в такому разі при наступному звертанні до функції *getch()* за кодом можна визначити, яка саме клавіша була натиснутою.

Таблиця 13.2.

Значення допоміжного байта для функціональних клавіш

		SHIFT	CTRL	ALT
F1	59	84	94	104
F2	60	85	95	105
F3	61	86	96	106
F4	62	87	97	107
F5	63	88	98	108
F6	64	89	99	109
F7	65	90	100	110
F8	66	91	101	111
F9	67	92	102	112
F10	68	93	103	113
F11	133	135	137	139
F12	134	136	138	140

Таблиця 13.3.

Значення допоміжного байта для комбінацій клавіш з ALT

ALT-1	120	**ALT-A**	30	**ALT-K**	37	**ALT-U**	22
ALT-2	121	**ALT-B**	48	**ALT-L**	38	**ALT-V**	47
ALT-3	122	**ALT-C**	46	**ALT-M**	50	**ALT-W**	17
ALT-4	123	**ALT-D**	32	**ALT-N**	49	**ALT-X**	45
ALT-5	124	**ALT-E**	18	**ALT-O**	24	**ALT-Y**	21
ALT-6	125	**ALT-F**	33	**ALT-P**	25	**ALT-Z**	44
ALT-7	126	**ALT-G**	34	**ALT-Q**	16	**ALT-мінус**	74
ALT-8	127	**ALT-H**	35	**ALT-R**	19	**ALT-плюс**	78
ALT-9	128	**ALT-I**	23	**ALT-S**	31	**ALT - ***	55
ALT-0	129	**ALT-J**	36	**ALT-T**	20	**ALT - =**	131

Таблиця 13.4.

Значення допоміжного байта для інших комбінацій клавіш

ALT-	43	←	75
Insert	82	↑	72
Home	71	→	77
PgUp	73	↓	80
PgDn	81	**CTRL -** ←	115
End	79	**CTRL -** →	116
Delete	83	**CTRL-END**	117
5 (цифрова)	76	**CTRL-Home**	119
Shift-Tab	15	**CTRL-PgDn**	118

Приклад реалізації обробки відслідковування натискань спеціальних клавіш.

```
char c=getch();
if (c==0)
{
   c=getch();
   switch(c)
   {
    case 75  :/* натиснута стрілка вліво */ break;
    case 77  :/* натиснута стрілка вправо */ break;
    case 72  :/* натиснута стрілка вгору */ break;
    case 80  :/* натиснута стрілка вниз */ break;
    case 103 :/* натиснута комбінація CTRL-F10 */
              break;
    case 46  :/* натиснута комбінація ALT-C*/ break;
   }
}
```

Функції стандартної бібліотеки

Функції I/O (stdio.h)

Таблиця 13.5.

Функції I/O (stdio.h)

clearerr(FILE *stream);	Очистка прапорця помилок для вказаного потоку.	void
fclose(FILE *stream);	Закриття потоку.	int
fcloseall(void);	Закриття всіх відкритих (на верхньому рівні) файлів (потоків).	int
feof(FILE *stream);	Перевірка на кінець потоку.	int
ferror(FILE *stream);	Перевірка прапорця помилок потоку.	int
fflush(FILE *stream);	Запис даних з буфера у потік.	int
fgetc(FILE *stream);	Читання символу з потоку.	int
fileno(FILE *stream);	Отримання дескриптора, зв'язаного з потоком.	int
fgetchar(void);	Читання символу із стандартного потоку введення	int
fgetpos(FILE *stream, fpos_t *pos);	Повертає поточну позицію у файлі.	int
fgets(char *s, int n, FILE *stream);	Читання рядка з потоку.	char *
fdopen(int handle, char *type);	Відкриття потоку (відкрити файл і зв'язати його з потоком).	FILE*
fprintf (FILE *stream, const char *format [, argument, ...]);	Запис даних в потік за форматом.	int
fputc(int c, FILE *stream);	Запис символу в потік.	int
fputchar(int c);	Запис символа в стандартний потік виведення.	int
fputs(const char *s, FILE *stream);	Запис рядка в потік.	int

fread(void *ptr, size_t size, size_t n, FILE *stream);	Читання даних з потоку.	size_t
freopen(const char *filename, const char *mode, FILE *stream);	Повторне відкриття потоку в новому режимі.	FILE *
fscanf (FILE *stream, const char *format [, address, ...]);	Читання даних з потоку за рядком формату.	int
fseek(FILE *stream, long offset, int whence);	Зміна позиції покажчика файла.	int
fsetpos(FILE *stream, const fpos_t *pos);	Переміщення вказівника файла відносно початку файла.	int
ftell(FILE *stream);	Повертає поточну позицію вказівника файла.	long
fwrite(const void *ptr, size_t size, size_t n, FILE*stream);	Запис даних із заданого буфера в потік.	size_t
getc(FILE *stream);	Читання символа з потоку.	int
getchar(void);	Читання символа з потоку stdin.	int
gets(char *s);	Читанн рядка із потоку stdin.	char*
getw(FILE *stream);	Читання слова (двох байт) із потоку.	int
printf (const char *format [, argument, ...]);	Запис даних в потік stdout за форматом.	int
putc(int c, FILE *stream);	Запис символа в потік.	int
putchar(int c);	Запис символа в потік stdout.	int
puts(const char *s);	Запис рядка в потік.	int
putw(int w, FILE *stream);	Запис слова (двох байт) в потік.	int
remove(const char *filename);	Знищення файла.	int
rename(const char *oldname, const char *newname);	Переіменування файла.	int
rewind(FILE *stream);	Встановлення вказівника файла на його початок.	void
scanf (const char *format [, address, ...]);	Читання даних з потоку stdin за форматом.	int

setbuf(FILE *stream, char *buf);	Встановлення буферизації потоку.	void
setvbuf(FILE *stream, char *buf, int type, size_t size);	Встановлення буферизації і розміру потока.	int
sprintf (char *buffer, const char *format [, argument, ...]);	Запис даних в рядок за форматом.	int
sscanf (const char *buffer, const char *format [, address, ...]);	Читання даних із рядка за форматом.	int
tempnam(char *dir, char *prefix);	Згенерувати ім'я тимчасового файла в заданому каталозі.	char *
ungetc(int c, FILE *stream);	Повертає символ в потік.	int
vfscanf(FILE *stream, const char *format, va_list arglist);	Читання даних з потоку з використанням списку аргументів.	int
vprintf (const char *format, va_list arglist);	Запис даних в стандартний потік виведення за форматом.	int
vsprintf(char *buffer, const char *format, va_list arglist);	Виведення рядка параметріву визначеному форматі.	int
vsscanf(const char *buffer, const char *format, va_list arglist);	Читає рядок, використовуючи список аргументів.	int

Математичні функції (math.h)

Таблиця 13.6.

Математичні функції (math.h)

abs(int x);	Повертає модуль цілого числа.	int
acos(double x); acosl(long double (x));	Повертає арккосинус аргумента.	double long double
asin(double x); asinl(long double (x));	Повертає арксинус аргумента.	double long double
atan(double x); atanl(long double (x));	Повертає арктангенс аргумента.	double long double

atan2(double y, double x); atan2l(long double(y), long double (x));	Повертає арктангенс відношення аргументів.	double long double
ceil(double x); ceill(long double (x));	Заокруглення до найменшого цілого, більшого або рівного заданому числу.	double long double
cos(double x); cosl(long double x);	Обчислення косинуса.	double long double
cosh(double x); coshl(long double (x));	Обчислення гіперболічного косинуса.	double long double
exp(double x); exp(long double (x));	Повертає степінь числа e.	double long double
fabs(double x); fabsl(long double @E(x));	Повертає модуль числа (для дійсних чисел).	double long double
floor(double x); floorl(long double (x));	Заокруглення до найменшого цілого, меньшого або рівного заданому числу.	double long double
fmod(double x, double y); fmod(long double (x), long double (y));	Повертає залишок від ділення аргументів.	double long double
frexp(double x, int *exponent); frexp(long double (x), int *(exponent));	Виділяє з числа мантису і експоненціальну частину.	double long double
ldexp(double x, int expon); ldexpl(long double (x), int (expon));	Перетворює мантису і показник степеня в число.	double long double
log(double x); logl(long double (x));	Обчислює натуральний логарифм.	double long double
log10(double x); log10l(long double (x));	Обчислює десятковий логарифм.	double long double
modf(double x, double *ipart); modfl(long double (x), long double *(ipart));	Розбиває число на цілу і дробову частини.	double long double
pow(double x, double y); pow(long double (x), long double (y));	Підносить число до вказаного степеня.	double long double

sinl(long double x) sin(double x);	Обчислює синус аргумента.	long double double
sinh(double x); sinhl(long double (x));	Обчислює гіперболічний синус аргумента.	double long double
sqrt(double x); sqrtl(long double @E(x));	Обчислює квадратний корінь аргумена.	double long double
tan(double x); tanl(long double x);	Обчислює тангенс аргумента.	double long double
tanh(double x); tanhl(long double (x));	Обчислює гіперболічний тангенс аргумента.	double long double

Функції для роботи з рядками (string.h)

Таблиця 13.7.

Функції для роботи з рядками (string.h)

strcat(char *dest, const char *src);	Об'єднання рядків.	char *
strchr(const char *s, int c);	Пошук символа у рядку.	char *
strcmp(const char *s1, const char*s2);	Порівняння рядків.	int
strcpy(char *dest, const char *src);	Копіювання одного рядка в інший.	char *
strcspn(const char *s1, const char *s2);	Знайти перше входження символа із заданого набору символів в рядку.	size_t
strdup(const char *s);	Дублювання рядка.	char *
strerror(int errnum);	Повертає покажчик на рядок з описом помилки.	char *
strlen(const char *s);	Повертає довжину рядка.	size_t
strlwr(char *s);	Перетворити рядок у нижній регістр.	char *
strncat(char *dest, const char *src, size_t maxlen);	Об'єднує один рядок з n символами іншого.	char *
strncmp (const char *s1, const char *s2, size_t maxlen);	Порівнює один рядок з n символами іншого.	int

strncpy(char *dest, const char *src, size_t maxlen);	Копіює перші n символів одного рядка в інший.	char *
strnset(char *s, int ch, size_t n);	Заповнити n символів рядка в задане значення.	char *
strpbrk(const char *s1, const char *s2);	Знайти перше входження будь-якого символа із заданого набору в рядку	char *
strrchr(const char *s, int c);	Пошук першого входження заданого символа в рядку.	char *
strrev(char *s);	Інвертувати рядок.	char *
strncat(char *dest, const char *src, size_t maxlen);	Встановити всі символи рядка в задане значення.	char *
strspn(const char *s1, const char *s2);	Шакує перший символ одного рядка, відсутній в іншому.	size_t
strstr(const char *s1, const char *s2);	Шукає частину рядка в іншому рядку.	char *
strupr(char *s);	Перетворити рядок у верхній регістр.	char *

Функції для роботи із символами

Таблиця 13.8.

Функції для роботи із символами

Функція	*Опис*	*Тип повернення*
isalnum(int c);	Перевірка, чи є символ літерою або цифрою.	int
isalpha(int c);	Перевірка, чи є символ літерою.	int
iscntrl(int c);	Перевірка, чи є символ керуючим.	int
isdigit(int c);	Перевірка, чи є символ десятковою цифрою.	int
isgraph(int c);	Перевірка, чи є символ видимим.	int
islower(int c);	Перевірка, чи є символ літерою нижнього регістру.	int
ispunct(int c);	Перевірка, чи є символ знаком пунктуації.	int
isspace(int c);	Перевірка, чи є символ пробільним.	int
isupper(int c);	Перевірка, чи є символ літерою верхнього регістру.	int
isxdigit(int c);	Перевірка, чи є символ шістнадцятковою цифрою.	int

tolower(int c);	Перетворення символу в нижній регістр.	int
toupper(int c);	Перетворення символу у верхній регістр.	int

Функції для роботи з графічним режимом (graphics.h)

Таблиця 13.9.

Функції для роботи з графічним режимом (graphics.h)

bar(int left, int top, int right, int bottom);	Малює зафарбований прямокутник.	void far
arc(int x, int y, int stangle, int endangle, int radius);	Малює дугу.	void far
bar3d(int left, int top, int right, int bottom, int depth, int topflag);	Вимальовує трьохвимірний стовпець.	void far
circle(int x, int y, int radius);	Малює коло.	void far
cleardevice(void);	Очищає екран.	void far
clearviewport(void);	Очищає графічне вікно.	void far
closegraph(void);	Закриває графічний режим.	void far
detectgraph(int far *graphdriver, int far *graphmode);	Повертає тип графічного драйвера.	void far
drawpoly(int numpoints, int far *polypoints);	Вимальовує ламану лінію.	void far
ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);	Малює еліптичну дугу від початкового кута до кінцевого.	void far
fillellipse(int x, int y, int xradius, int yradius);	Малює заштрихований еліпс.	void far
fillpoly(int numpoints, int far *polypoints);	Малює і штрихує багатокутник.	void far
floodfill(int x, int y, int border);	Заштриховує замкнену область.	void far
getaspectratio(int far *xasp, int far *yasp);	Повертає відношення сторін графічного екрану.	void far

getbkcolor(void);	Повертає поточний колір фону.	int far
getcolor(void);	Повертає поточний колір.	int far
getfillpattern(char far *pattern);	Повертає поточний тип штриховки.	void far
getfillsettings (struct fillsettingstype far *fillinfo);	Повертає поточний тип і колір штриховки.	void far
getimage(int left, int top, int right, int bottom, void far *bitmap);	Зберегти бітовий образ частини екрана.	void far
getlinesettings(struct linesettingstype far *lineinfo);	Повертає поточний стиль, шаблон і товщину штриховки.	void far
getmaxcolor(void);	Повертає максимальний колір, який можна задавати в параметрах.	int far
getmaxx(void); getmaxy(void);	Повертають відповідно максимальну X-координату та Y-координати екрана.	int far int far
getpixel(int x, int y);	Повертає колір пікселя з координатами (x,y)	unsigned far
gettextsettings(struct textsettingstype far *texttypeinfo);	Повертає поточний шрифт, розмір та вирівнювання тексту.	void far
getx(void); gety(void);	Повертають відповідно X- та Y-координати поточного вказівника.	int far int far
graphresult(void);	Повертає код помилки для останньої графічної операції.	int far
imagesize(int left, int top, int right, int bottom);	Повертає число байт, що необхідні для зберігання прямокутної частини екрана.	unsigned far
initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);	Ініціалізація графічного режиму роботи адаптера.	void far

line(int x1, int y1, int x2, int y2);	Малює лінію від точки (x1,y1) до точки (x2,y2).	void far
linerel(int dx, int dy);	Малює лінію від поточного положення вказівника до точки, заданої приростом координат.	void far
lineto(int x, int y);	Малює лінію від поточного положення вказівника до заданої точки.	void far
moverel(int dx, int dy);	Переміщує вказівник до точки, заданої приростом координат.	void far
moveto(int x, int y);	Переміщує вказівник до точки з заданими координатами.	void far
outtext(char far *textstring);	Виводить текстовий рядок на екран.	void far
outtextxy(int x, int y, char far *textstring);	Виводить текстовий рядок в задане місце екрана.	void far
pieslice(int x, int y, int stangle, int endangle, int radius);	Малює і штрихує сектор кола.	void far
putimage(int left, int top, void far *bitmap, int op);	Виводить бітовий образ на екран.	void far
putpixel(int x, int y, int color);	Виводить точку з заданими координатами і кольором.	void far
rectangle(int left, int top, int right, int bottom);	Малює прямокутник.	void far
sector(int x, int y, int stangle, int endangle, int xradius, int yradius);	Штрихує сектор еліпса.	void far
setaspectratio(int xasp, int yasp);	Змінює масштабний коефіцієнт відношення сторін екрана.	void far
setbkcolor(int color);	Встановлює колір фону.	void far

setcolor(int color);	Встановлює поточний колір	void far
setfillpattern(char far *upattern, int color);	Встановлює тип штриховки (довільний).	void far
setfillstyle(int pattern, int color);	Встановлює тип і колір штриховки.	void far
setlinestyle(int linestyle, unsigned upattern, int thickness);	Встановлює товщину і стиль лінії.	void far
settextjustify(int horiz, int vert);	Встановлює вирівнювання тексу.	void far
settextstyle(int font, int direction, int charsize);	Встановлює поточний шрифт, стиль і розмір тексту.	void far
setviewport(int left, int top, int right, int bottom, int clip);	Визначає вікно для графічного виводу.	void far
textheight(char far *textstring);	Повертає висоту рядка у пікселях.	int far
textwidth(char far *textstring);	Повертає довжину рядка у пікселях.	int far

Тести для самоконтролю

№ з/п	Текст завдання	Варіанти відповідей
Типи даних в мові С. Операції та оператори.		
1.	Який тип даних використовується для оголошення символьних змінних?	А. short Б. long В. char Г. float Д. double
2.	Який модифікатор використовується для оголошення беззнакових змінних?	А. char Б. float В. double Г. unsigned Д. void
3.	Яка операція використовується для присвоєння значень змінним?	А. % Б. = В. == Г. >= Д. <=
4.	Яка операція використовується для об'єднання двох байтів по логіці « I »?	А. != Б. | В. || Г. & Д. &&
5.	Яка операція використовується для віднімання з присвоюванням значення змінній?	А. += Б. -= В. *= Г. /= Д. %=
6.	Який вираз відповідає оголошенню змінній цілого типу з ініціалізацією значення?	А. int x; Б. int x = 1; В. float y; Г. int v=(float) y; Д. char ch='A';
7.	Який вираз відповідає операції інкремента в префіксній формі?	А. int v=m[2]; Б. i++; В. i--; Г. ++i; Д. --i;

8.	Вираз i++ у мові С еквівалентний виразу:	А. i = i – 1 Б. i = 1 В. i = i + 1 Г. i = -1 Д. i -= 1
9.	Як позначається операція перевірки рівності двох виразів?	А. = Б. == В. != Г. >= Д. <=
10.	Як позначається операція побітового зсуву на 1 розряд вліво?	А. <1 Б. 1<= В. <<1 Г. -<1 Д. =<1
11.	Тип даних int дозволяє зберігати у пам'яті значення:	А. дійсного числа Б. цілого числа В. рядкового числа Г. символьного числа Д. додатнього цілого числа
12.	Тип даних double дозволяє зберігати у пам'яті значення:	А. дійсного типу Б. цілого типу В. рядкового типу Г. символьного типу Д. короткого цілого типу
13.	Яким буде значення змінної a у виразі int a = 9/2+9%4 у мові С?	А. 5 Б. 4 В. 5. 5 Г. 3.75 Д. 6
14.	Як позначається операція побітового "І"?	А. | А. || В. & Г. ==

		Д. &&
15.	Яка з нижче перерахованих операцій є унарною?	А. ++ Б. && В. & Г. || Д. ==
16.	Як позначається операція побітового "АБО"?	А. ++ Б. || В. | Г. == Д. &&
17.	Вираз i-- у мові С еквівалентний виразу:	А. i = i + 1 Б. i = 1 В. i = i - 1 Г. i = -1 Д. i += 1
18.	Який тип даних з нижче перерахованих дозволяє зберігати невід'ємні цілі числа?	А. long Б. unsigned long В. signed long Г. long long Д. int
19.	Яким буде значення виразу int a =14/2+31%5 у мові С?	А. 3 Б. 3.133333 В. 3.333333 Г. 8 Д. 6
20.	Яким символом позначається операція взяття адреси змінної?	А. & Б. % В. * Г. @ Д. &&
21.	Тип даних char дозволяє зберігати у пам'яті значення:	А. дійсного типу Б. рядкового типу В. цілого символьного типу Г. об`єктного типу Д. довгого цілого типу

22.	Як у мові С записується операція зсуву вправо на 3 розряди?	А. =>3 Б. 3>= В. ->3 Г. >>3 Д. 3>>
23.	Як у мові С записується операція зсув вліво на 2 розряди?	А.2 = Б. <=2 В. <-2 Г.<<2 Д. 2<<

24.	Якщо є код:
	int x = 5;
	x++;
	Яке значення матиме x?
		А. 0
		Б. 1
		В. 2
		Г. 5
		Д. 6

25.	Якщо є код: int x = 5; x--; Яке значення матиме x?	А. 0 Б. 1 В. 2 Г. 4 Д. 5

26.	Якщо є код:
	int x = 5;
	x += 3;
	Яке значення матиме x?
		А. 5
		Б.. 6
		В. 7
		Г. 8
		Д. 3

27.	Якщо є код:
	int x = 5;
	int y;
	y = x % 2;
	Яке значення матиме y?
		А. 0
		Б. 1
		В. 2
		Г. 3
		Д. 5

28.	Якщо є код:
	int x = 5;
	int y;
	y = abs(x);
	Яке значення матиме y?
		А. 0
		Б. 1
		В. 5
		Г. -5
		Д. 2

29.	Якщо є код:
	float x = 4;
	float y;
	y = sqrt(x);
	Яке значення матиме y?
		А. 0
		Б. 1
		В. 2
		Г. 3
		Д. 4

30.	Якщо є код:
	float x = 3;
	float y;
	y = pow(x, 2);
	Яке значення матиме y?
		А. 6
		Б. 9
		В. 3
		Г. 2
		Д. 1

31.	Дано фрагмент коду на мові С:
	float f = 32.51;
	printf("%2.1f", f);
	Що буде виведено на екран?
		А. 32.5
		Б. 32.56
		В. 032.56
		Г. 032.5
		Д. 32.560

32.	Який результат виконання операції 0x5 & 0x6?	А. 0 Б. 4 В. 2 Г. 3 Д. 5
33.	Який результат виконання операції 0x3 | 0x4?	А. 0 Б. 1 В. 3 Г. 7 Д. 5
34.	Який результат виконання операції (0010) << 1?	А.0000 Б.0001 В. 0011 Г. 0010 Д. 0100
35.	Який результат виконання операції (1000) >>2?	А. 0000 Б. 0100 В. 0010 Г. 1100 Д. 0110
36.	Який розмір пам'яті потрібний для збереження змінної типу int (для 16-разрядних платформ)?	А. 1байт Б. 2 байта В. 3 байт Г. 6 байт Д. 8 байт
37.	Який розмір пам'яті потрібний для збереження змінної типу char?	А. 1байт Б. 2 байта В. 4 байта Г. 6 байт Д. 8 байт

38.	Який розмір пам'яті потрібний для збереження змінної типу float?	А. 1байт Б. 2 байта В. 4 байта Г. 6 байт Д. 10 байт
39.	Яким буде значення змінної a у виразі int a = 27/2-13%4 у мові С?	А. 4.75 Б. 6 В. 3 Г. 12. 5 Д. 12
40.	Яким буде значення змінної a у виразі int a =12/2+32%5 ?	А. 3 Б. 3.133333 В. 3.333333 Г. 8 Д. 6
41.	Який тип даних з перерахованих займає 1 байт пам'яті?	А. long Б. unsigned В. double Г. char Д. float
42.	Яка з нижче перерахованих операцій є бінарною?	А. ?: Б. & В. ++ Г. > Д. --

43.	Дано фрагмент коду. Визначити, чому дорівнює змінна a?
	int a=3;
	a *= (a *=a);
		А. 9
		Б. 3
		В. 27
		Г. 81
		Д. 18

44.	Дано фрагмент коду. Визначити, чому дорівнює змінна x? x = (y = 3, y + 1);	А. 3 Б. 1 В. 5 Г. 4 Д. 2

45.	Дано фрагмент коду. Визначити, чому дорівнює змінна a?
	int a=1, b=3, c;
	c =(b+a++, a+--b);
		А. 1
		Б. 2
		В. 4
		Г. 3
		Д. 5

46.	Дано фрагмент коду. Визначити, чому дорівнює змінна b?
	int a=3, b=8, c;
	c=(a++, a+b);
	(b--,c)*=3;
		А. 8
		Б. 11
		В. 3
		Г. 7
		Д. 12

47.	Визначитизначення змінної p після операції, якщо на початку операції всі змінні мають значення рівне 5: p*=x++?	А. 25 Б. 5 В. 36 Г. 30 Д. 6
	Розгалудження та цикли.	

48.	Якщо є код:
	int x = 4;
	int y;
	if (x < 4) y = 2*x; else y = x;
	Яке значення матиме y?
		А. 2
		Б. 4
		В. 6
		Г. 8
		Д. 1

49.	Для заданного виразу знайти тотожний: if ((x == 0) && (y == 0)) ...	А. if (x!0&& y!0) ... Б. if (x=0&& y=0) ... В. if (x && y) ... Г. if (!x && !y) ... Д. if (x&&y=0) ...

50.	Що буде виведено на екран при виконанні фрагмента коду:
	for (int i = 3; i > 0; i--)
	printf("%d ",2*i);
		А. 0 1 2
		Б. 1 2 3
		В. 2 3 4
		Г. 6 4 2
		Д. 2 4 6

51.	Що буде виведено на екран при виконанні фрагмента коду:
	for (int i=0; i<3; i++)
	printf("%d ", 2*i);
		А. 0 1 2
		Б. 1 2 3
		В. 0 2 4
		Г. 2 3 4
		Д. 0 4 2

52.	Що буде виведено на екран при виконанні фрагмента коду:
	int x=7;
	if (x%2==0) printf("0");
	else printf("1");
		А. 0
		Б. 1
		В. 2
		Г. 3
		Д. 5

53.	Що буде виведено на екран при виконанні фрагмента коду: int a=2, b=5, y;	А. 0 Б. 1 В. 2

	y = (a > b) ? a : b; printf("%d", y);	Г. 3 Д. 5

54.	Що буде виведено на екран при виконанні фрагмента коду:
	int sum=0, i=2;
	while(i<4)
	{sum+=i; i++;}
	printf("%d", sum);
		А. 1
		Б. 2
		В. 5
		Г. 4
		Д. 3

55.	Що буде виведено на екран при виконанні фрагмента коду:
	int sum=0, i=2;
	do{sum+=i; i++;}
	while(i<4);
	printf("%d",sum);
		А. 2
		Б. 3
		В. 4
		Г. 5
		Д. 8

56.	Що буде виведено на екран при виконанні фрагмента коду:
	int sum=0;
	for (int i=1; i<8; i++)
	{sum+=i; if(i==4) break;}
	printf("%d",sum);
		А. 2
		Б. 4
		В. 10
		Г. 7
		Д. 8

57.	Якщо є код:
	int x = 0;
	int y;
	while(x < 4) {y = 2*x; x++ ;}
	Яке значення матиме y?
		А. 0
		Б. 2
		В. 4
		Г. 6
		Д. 8

58.	Якщо є код:
	int x = 4;
	int y;
	while(x > 0) {y = 3*x; x-- ;}
	Яке значення матиме y?
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 6

59.	Що буде виведено на екран при виконанні фрагмента коду:
	int x = 4 , y;
	if(x > 3) y = x*x; else y = x;
	printf("%d",y);
		А. 3
		Б. 6
		В. 8
		Г. 16
		Д. 12

60.	Що буде виведено на екран при виконанні фрагмента коду:
	int n = 0;
	for(int i = 0; i < 5; i++)
	if(i % 2 == 0) n++;
	printf("%d ", n);
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 5

61.	Що буде виведено на екран при виконанні фрагмента коду:
	int n = 1;
	for(int i = 1; i < 5; i++)
	n = n * i;
	printf("%d ", n);
		А. 12
		Б. 14
		В. 16
		Г. 24
		Д. 28

62.	Що буде виведено на екран при виконанні фрагмента коду:
	int i = 4, sum = 0;
	while(i >0) {
	if(i % 2 == 0) sum += i; i--; }
	printf("%d ", sum);
		А. 4
		Б. 6
		В. 8
		Г. 10
		Д. 12

63.	Дано фрагмент коду на мові С:
	int v = 0X24;
	printf("%d", v);
	Що буде виведено на екран?
		А. 39
		Б. 0x25
		В. 36
		Г. 0X25
		Д. 25

64.	Дано фрагмент коду на мові С:
	int i = 0,n = 0;
	while(i< 7) {
	if(i % 2 == 0) n++; i++; }
	printf("%d ", n);
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 5

65.	Що буде виведено на екран при виконанні фрагмента коду:
	int i = 4, n = 0;
	while(i >0) {
	if(i % 2 == 0) n ++; i--; }
	printf("%d ", n);
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 5

66.	Визначити скільки коренів матиме квадратне рівняння, якщо a=1, b=-5, c=4:
	double d = b*b-4*a*c;
	int n=(d<0.0)?0:(d>0.0)?2:1;
	printf(" корней: %d\n",n);
		А. 1
		Б. 0.0
		В. 2.0
		Г. 2
		Д. 9

67.	Вкажіть що виведе на екран поданийфрагмент коду?
	for (int i = 1; i<3; i++)
	for (int j = 3; j>0; j--){
	printf("%d ", i+j) ;
	}
		А. 4 3 2 1 2 3
		Б. нічого
		В. 4 3 2 5 4 3
		Г. помилка в третій частині циклу for
		Д. 5 4 3

68.	Вкажіть що виведе на екран поданийфрагмент коду?
	int ch = 1, k = 10, sum = 0;
	while (ch < k){
	sum += ch;
	ch++;
	printf("%d ", ch); }
		А. 0 1 2 3 45 67 8
		Б. нічого
		В. 2 3 45 67 8 9 10
		Г. 1 2 3 45 67 8 9
		Д. 5 4 3

69.	Вкажіть що виведе на екран поданийфрагмент коду?
	int a = 1;
	switch (a){
	case 1:
	case 2:
	case 3:printf("Three");
	}
		А. Three Three Three
		Б. Three Three
		В. Three
		Г.One Two Three
		Д. Помилка, код не відпрацює

70.	Вкажіть що виведе на екран поданийфрагмент коду?
	int a = 1;
	switch (a){
	case 1:printf("One");
	case 2:printf("Two");
	case 3:printf("Three");
	}
		А. One
		Б. Two
		В. Three
		Г.One Two Three
		Д. Помилка, код не відпрацює

71.	Вкажіть що виведе на екран поданийфрагмент коду?
	doublea=1;
	switch (a){
	case 1:printf("One"); break;
	case 2:printf("Two"); break;
	case 3:printf("Three"); break;
	}
		А. One
		Б. Two
		В. Three
		Г.One Two Three
		Д. Помилка, код не відпрацює

72.	Вкажіть що виведе на екран поданийфрагмент коду?
	if (5 = 5)
	printf("Yes");
	else
	printf("No");
		А. Yes
		Б. No
		В. Yes No
		Г.NoYes
		Д. Помилка, код не відпрацює

73.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int a = 10;
	if (a = 5)
		А. Yes
		Б. No
		В. Yes No
		Г.NoYes

	printf("Yes");
	else
	printf("No");
		Д. Помилка, код не відпрацює

74.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int x = 5, y = 10;
	if (x<10){
	if (y >= 10)
	printf("1");
	printf("2");
	}
	elseprintf("3");
		А. 1
		Б. 2
		В. 3
		Г.12
		Д. Помилка, код не відпрацює

75.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int num = 200;
	if (num < 750)
	printf("num < 750");
	else if (num < 450)
	printf("num < 450");
	else
	printf("num < 150");
		А. num < 750
		Б. num < 450
		В. num < 150
		Г.num <200
		Д. Помилка, код не відпрацює

76.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int ch = 0, k = 4;
	while (-3){
	if (ch >= k) break;
	printf("%d ", ch);
	ch += 3;
	}
		А. 1 2 3
		Б. 1 2 3 4
		В. 0 3
		Г. 3 0
		Д. Помилка, код не відпрацює

77.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int i = 4, k = -2;
	do {
	printf("%d ", k);
	i += 2;
	} while (k >= i);
		А. 2
		Б. -2
		В. 4
		Г. -2 0 2 4
		Д. Помилка, код не відпрацює

78.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int i = 4;
	do {
	printf("Hello");
		А. 2
		Б. -2
		В. Hello 4
		Г. Hello
		Д. Помилка, код

	i++;
	if (i > 0) continue;
	printf("%d ", i);
	} while (i<3);
		не відпрацює

79.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	inti = 4;
	for(; ;){
	printf("good!");
	if (3<=5) break;
	}
		А. good!
		Б. good! good!
		В. good!
		good!good!
		Г. Відпрацює, але нічого не виведе
		Д. Помилка, код не відпрацює

80.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int i = 0, a=4;
	while (i<a){
	printf("%d",a-i);
	i++;
	}
		А. 1 2 3 4
		Б. 4 3 2 1
		В. 3 2 1 0
		Г. Відпрацює, але нічого не виведе
		Д. Помилка, код не відпрацює

81.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int ar[10];
	int sum = 0;
	for (int i = 0; i<10; i++)
	ar[i] = i + 1;
	for (int i = 0; i<10; i++)
	if (i % 2)
	sum += ar[i];
	printf("%d ", sum);
		А. 10
		Б. 20
		В. 30
		Г. 40
		Д. Помилка, код не відпрацює

82.	Чи виконається поданийфрагмент коду, якщо так – що відбудеться в результаті його виконання?
	int size = 10;
	float arr[size] = { 0 };
	for (int i = 0; i < size; i++)
	printf("%d ", arr[i]);
		А. Буде створено масив з 10 елементів
		Б. Буде створено масив розміром size
		В. Буде створено масиврозміром 0
		Г. Буде створено масив і

		заповнено нулями Д. Помилка, код не відпрацює
	Функції.	

83.	Вкажіть що виведе на екран поданийфрагмент коду?
	int sum(int a, int b){
	return a + b;
	}
	void main(){
	printf("%d ",sum(10, 12));
	}
		А. 10
		Б. 12
		В. 22
		Г. 10+12
		Д. Помилка, код не відпрацює

84.	Вкажіть що виведе на екран поданийфрагмент коду?
	void sum(int a, int b){
	return a + b;
	}
	void main(){
	printf("%d ",sum(10, 12));
	}
		А. 10
		Б. 12
		В. 22
		Г. a + b
		Д. Помилка, код не відпрацює

85.	Вкажіть що виведе на екран поданийфрагмент коду?
	void Show(char y, int z = 5, float x = 7.7){
	printf("%d %.1f %c", z,x,y);
	}
	void main(){
	Show('W');
	}
		А. W
		Б. 5 7.7 W
		В. 5 7.7
		Г.W5 7.7
		Д. Помилка, код не відпрацює

86.	Яка функція призначена для читання рядків із потоку stdin у мові С?	А. getch( ) Б. gets( ) В. puts( ) Г. cin>> Д. cout<<
87.	Яка з перерахованих функцій повертає значення дійсного типу?	А. void func(char x); Б. void func(int x); В. double func(int x); Г. int func(float

		x); Д. int func(int x);
88.	Яка з перерахованих функцій нічого не повертає?	А. double func(); Б. float func(int x); В. void func(float x); Г. int func(float x); Д. char func(float x);
89.	Яка з перерахованих функцій повертає значення символьного типу?	А. void func(char x); Б. float func(int x); В. void func(int x); Г. int func(double x); Д. char func(float x);
90.	Яка з перерахованих функцій приймає значення дійсного типу?	А. void func(float x); Б. void func(int x); В. float func(int x); Г. int func(long x); Д. int func(char x);
91.	Яка з перерахованих функцій повертає значення цілого типу?	А. double func(); Б. float func(int x); В. void func(float x); Г. int func(int x); Д. char func(float x);
92.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?	А. 10 Б. 9

	int var = 10;
	int* pVar = &var;
	*pVar = 9;
	printf("%d ", *pVar);
		В. Невідоме значення
		Г.Адресу змінної var
		Д. Помилка, код не відпрацює

93.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int var = 5;
	int* pVar = &var;
	*pVar = 9;
	printf("%d ", pVar);
		А. 10
		Б. 9
		В. Невідоме значення
		Г.Адресу змінної var
		Д. Помилка, код не відпрацює

94.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int z[5] = { 1, 2, 3, 4, 5 };
	int* zPtr = &z;
	for (int i = 0; i<5; i++)
	printf("%d ", *(zPtr + i)) ;
		А. 1 2 3 4 5
		Б. 1 2 3 4
		В. 15
		Г. Адресу масиву z
		Д. Помилка, код не відпрацює

95.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int num = 100;
	int*numPtr = #
	++*numPtr;
	printf("%d ", num);
		А. 101
		Б. 100
		В. 10000
		Г.Адресу змінної num
		Д. Помилка, код не відпрацює

96.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int num = 100, rez = 10;
	int*numPtr = #
	rez *= *numPtr;
	printf("%d ", rez);
		А. 1010
		Б. 1000
		В. Невідоме значення
		Г.Адресу змінної rez
		Д. Помилка, код не відпрацює

97.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int z[5] = { 1, 2, 3, 4, 5 };
	int* zPtr = &z[0];
	for (int i = 0; i<5; i++)
		А. 1 2 3 4 5
		Б. 1 2 3 4
		В. 15
		Г. Адресу масиву z

	printf("%d ", *(zPtr + i)) ;	Д. Помилка, код не відпрацює
Масиви. Структури та об'єднання.		
98.	Який вираз відповідає оголошенню двовимірного статичного масиву?	А. char ch=65; Б. int *m; В. int m[10]; Г. double m[5][4]; Д. m[3]=234;
99.	Що буде виведено на екран при виконанні фрагмента коду: char str[] = "Test"; printf("%d\n", strlen(str));	А. 5 Б. 0 В. 3 Г. 4 Д. 8
100.	Виберіть правильну форму оголошення рядка довжиною 32:	А. char s[32]; Б. char s[] = 32; В. char 32[s]; Г. char [32] s; Д. s= char [32];
101.	Який розмір пам'яті потрібний для збереження одновимірного масиву цілих чисел(int для 16-разрядних платформ) із 20 елементів?	А. 10 байт Б. 40 байт В. 50 байт Г. 60 байт Д. 20 байт
102.	Який розмір пам'яті потрібний для збереження одновимірного масиву дійсних чисел типа float із 10 елементів?	А. 10 байт Б. 20 байт В. 40 байт Г. 60 байт Д. 100 байт
103.	Який розмір пам'яті потрібний для збереження одновимірного масиву дійсних чисел типа double із 10 елементів?	А. 10 байт Б. 20 байт В. 40 байт Г. 60 байт Д. 80 байт

104.	Що обчислює цей фрагмент коду:
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i = 0; i<5; i++)
	if(arr[i] % 2 != 0) sum=sum + arr[i];
		А. Кількість елементів масиву
		Б. Суму елементів масиву
		В. Суму парних елементів масиву
		Г. Додаток

		елементів масиву Д. Суму непарних елементів масиву

105.	Що обчислює цей фрагмент коду:
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(arr[i] % 2 == 0) sum=sum + arr[i];
		А. Кількість елементів масиву
		Б. Суму елементів масиву
		В. Суму парних елементів масиву
		Г. Додаток елементів масиву
		Д. Суму непарних елементів масиву

106.	Що обчислює цей фрагмент коду:
	int i, p =1;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  p=p * arr[i];
		А. Кількість елементів масиву
		Б. Суму елементів масиву
		В. Суму парних елементів масиву
		Г. Добуток елементів масиву
		Д. Суму непарних елементів масиву

107.	Що обчислює цей фрагмент коду:
	int i, p =1;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(arr[i] % 2 == 0) p=p * arr[i];
		А. Кількість елементів масиву
		Б. Суму елементів масиву
		В. Добуток парних елементів масиву
		Г. Додаток елементів масиву
		Д. Суму непарних елементів масиву

108.	Що обчислює цей фрагмент коду: int i, p =1; int arr[ ] = { 3,5,2,1,4 };	А. Кількість елементів масиву Б. Суму

	for(i=0; i<5; i++) if(arr[i] % 2 != 0) p=p * arr[i];	елементів масиву В. Добуток непарних елементів масиву Г. Додаток елементів масиву Д. Суму непарних елементів масиву

109.	Що буде виведено на екран при виконанні фрагмента коду
	int i, p =1;
	int arr[ ] = { 3,1,2,1,4 };
	for(i=0; i<5; i++) p=p * arr[i];
	printf("%d ", p);
		А. 6
		Б. 12
		В. 18
		Г. 24
		Д. 28

110.	Що буде виведено на екран при виконанні фрагмента коду
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i = 0; i<5; i++) sum=sum + arr[i];
	printf("%d ", sum);
		А. 10
		Б. 15
		В. 20
		Г. 12
		Д. 16

111.	Що буде виведено на екран при виконанні фрагмента коду
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 == 0) sum=sum + arr[i];
	printf("%d ", sum);
		А. 2
		Б. 4
		В. 6
		Г. 8
		Д. 15

112.	Що буде виведено на екран при виконанні фрагмента коду
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 != 0) sum=sum + arr[i];
	printf("%d ", sum);
		А.11
		Б. 12
		В. 15
		Г. 9
		Д. 8

113.	Що буде виведено на екран при виконанні фрагмента коду
	int i, p =1;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 != 0) p=p * arr[i];
		А. 8
		Б. 10
		В. 12
		Г. 14
		Д. 15

	printf("%d ", p);	

114.	Що буде виведено на екран при виконанні фрагмента коду
	int i, p =1;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 == 0) p=p * arr[i];
	printf("%d ", p);
		А. 2
		Б. 4
		В. 6
		Г. 8
		Д. 10

115.	Що буде виведено на екран при виконанні фрагмента коду
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] > 2) sum=sum + arr[i];
	printf("%d ", sum);
		А. 8
		Б. 10
		В. 12
		Г. 14
		Д. 15

116.	Що буде виведено на екран при виконанні фрагмента коду
	int i, sum =0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] < 3) sum=sum + arr[i];
	printf("%d ", sum);
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 5

117.	Що буде виведено на екран при виконанні фрагмента коду int i; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) printf("%d ", arr[i]);	А.1 2 3 4 5 Б. 3 5 1 2 4 В. 3 4 2 1 5 Г. 3 5 2 1 4 Д. 5 4 3 2 1

118.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 == 0)
	printf("%d ", arr[i]);
		А. 1 2 3 4 5
		Б. 3 5 1 2 4
		В. 2 4
		Г. 1 2 4
		Д. 1 3 4

119.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] % 2 != 0)
	printf("%d ", arr[i]);
		А. 1 2 3 4 5
		Б. 3 5 1 2 4
		В. 1 2 4
		Г. 3 5 1
		Д. 1 3 4

120.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(arr[i] > 2)
	  printf("%d ", arr[i]);
		А. 1 2 3 4 5
		Б. 1 3 4
		В. 1 2 4
		Г. 2 3 4
		Д. 3 5 4

121.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(arr[i] <= 3)
	  printf("%d ", arr[i]);
		А. 3 2 1
		Б. 1 3 4
		В. 1 2 4
		Г. 3 4 5
		Д. 2 1 4

122.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(i % 2 == 0)
	  printf("%d ", arr[i]);
		А. 3 5 2
		Б. 3 2 4
		В. 5 2 1
		Г. 2 4
		Д. 3 2

123.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(i % 2 != 0)
	  printf("%d ", arr[i]);
		А. 3 5 1
		Б. 5 2 1
		В. 2 1 3
		Г. 5 1
		Д. 1 4 2

124.	Що буде виведено на екран при виконанні фрагмента коду
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	  if(i > 1)
	  printf("%d ", arr[i]);
		А. 3 5 2
		Б. 5 2 1
		В. 3 5 1
		Г. 5 1 4
		Д. 2 1 4

125.	Що буде виведено на екран при виконанні фрагмента коду:
	int i;
	int arr[ ] = { 3,5,2,6,4 };
	for(i=0; i<5; i++)
	  if(i <= 2)
		А. 3 5 2
		Б. 2 6 4
		В. 5 2 6
		Г. 3 5 4
		Д. 5 6 4

	printf("%d", arr[i]);	

126.	Що буде виведено на екран при виконанні фрагмента коду:
	int i;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(i > 0 && i <=3)
	printf("%d ", arr[i]);
		А. 3 5 2
		Б. 3 2 1
		В. 5 1 4
		Г. 5 2 1
		Д. 3 2 4

127.	Що буде виведено на екран при виконанні фрагмента коду:
	int i, sum = 0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(i > 1) sum += arr[i];
	printf("%d ", sum);
		А. 3
		Б. 5
		В. 7
		Г. 9
		Д. 15

128.	Що буде виведено на екран при виконанні фрагмента коду:
	int i, sum = 0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(i < 3) sum += arr[i];
	printf("%d ", sum);
		А.12
		Б. 15
		В. 8
		Г. 10
		Д. 14

129.	Що буде виведено на екран при виконанні фрагмента коду:
	int i, n = 0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] > 3) n ++;
	printf("%d ", n);
		А. 1
		Б. 2
		В. 3
		Г. 4
		Д. 5

130.	Що буде виведено на екран при виконанні фрагмента коду
	int i, n = 0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
	if(arr[i] < 4) n ++;
	printf("%d ", n);
		А. 1
		Б. 2
		В. 4
		Г. 3
		Д. 5

131.	Що буде виведено на екран при виконанні фрагмента коду:
	int i, n = 0;
	int arr[ ] = { 3,5,2,1,4 };
	for(i=0; i<5; i++)
		А. 1
		Б, 2
		В. 3
		Г. 4
		Д. 5

	if(arr[i] >1 && arr[i] < 5) n ++; printf("%d ", n);	
132.	Для оголошення об'єднань використовується ключове слово:	А. union Б. struct В. class Г. void Д. enum
133.	Для оголошення структур використовується ключове слово:	А. struct Б. unoin В. STRUCT Г. Structura Д. enum
134.	Є оголошення: struct Person { int n; char name[10]; } p; Який розмір пам'яті потрібний для збереження змінної p?	А. 10 байт Б. 12 байт В. 20 байт Г. 16 байт Д. 18 байт
135.	Є оголошення: struct Person { int n; char name[10]; } P[10]; Який розмір пам'яті потрібний для збереження змінної P?	А. 10 байт Б. 100 байт В. 120 байт Г. 200 байт Д. 160 байт
136.	Є оголошення: struct Point { double x; double y; } point; Який розмір пам'яті потрібний для збереження змінної point?	А. 10 байт Б. 12 байт В.14 байт Г. 16 байт Д. 18 байт
137.	Визначити діапазон генерації псевдовипадкових чисел: rand()%41-20;	А. [0, 40) Б. [0, 40] В. [-20, 20) Г. [-20, 20] Д. (10, 40)
138.	Визначити діапазон генерації псевдовипадкових чисел: 0.01*(rand()%101);	А. [10, 100) Б. [0.01, 1] В. [1, 100) Г. [0.01, 1) Д. [0.1, 100)
139.	Визначити діапазон генерації псевдовипадкових чисел: 80 + rand()%(100 - 80 + 1)	А. [0; 20] Б. (21; 101) В. (20; 100) Г. [20; 100]

		Д. [80; 100]
140.	Визначити діапазон генерації псевдовипадкових чисел: (double) rand()/RAND_MAX	А. [0,1] Б. [0, 32767] В. [0.45,100000] Г. [0,10.67] Д.(0, 32767)
141.	Визначити діапазон генерації псевдовипадкових чисел: 1 + rand() % 10	А. (0, 10] Б. [0,10) В. (0,10) Г. [1,10] Д. (1, 11]

142.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	struct test {
	int a;
	double b;
	char arr[3];
	};
	void main(){
	printf("%d ", sizeof(test));}
		А. 8
		Б. 16
		В. 24
		Г. 32
		Д. Помилка, код не відпрацює

143.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	struct person {
	char name[20], surname[20];
	int age;
	} one = { "Ivan", "Sidorov", 50 }, *second = &one;
	void main(){
	strcpy_s(one.surname, "Ivanov");
	printf("%s ", (second->name));
	printf("%s ", second->surname);
	printf("%d ", second->age);
	}
		А. Ivanov
		Б. IvanSidorov 50
		В. IvanIvanov 50
		Г. Адресу полів name, surname, age
		Д. Помилка, код не відпрацює

144.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран?
	int ar[2][2] = { { 1,2 },{ 3,4 } };
	int** mas = ar;
	for (int i = 0; i<2; i++)
	for (int j = 0; j<2; j++)
	printf("%d ", mas[i][j]);
		А. 1 23 4
		Б. 4 3 2 1
		В. 0 1 1 0
		Г. 3 4 3 4
		Д. Помилка, код не відпрацює

145.	Чи виконається поданийфрагмент коду, якщо так – що відбудеться в результаті його виконання?
	int n=5;
	int *a =(int *) malloc(n*sizeof(int));
	for(int i=0; i<n; i++)
	scanf("%d", &a[i]);
		А. Буде створено двумірний масив на nрядків tan стовпців
		Б. Буде створено одновимірний масив на nелементів
		В. Буде створено трьохмірний масив
		Г. Масив не буде створений
		Д. Помилка, код не відпрацює

146.	Чи виконається поданийфрагмент коду, якщо так – що відбудеться в результаті його виконання?
	int n=5;
	int *a =(int *) calloc(n,sizeof(int));
	for(int i=0; i<n; i++)
	scanf("%d", &a[i]);
		А. Буде створено одновимірний масив на n елементів
		Б. Буде створено двумірний масив на nрядків tan стовпців
		В. Буде створено трьохмірний масив
		Г. Масив не буде створений
		Д. Помилка, код не відпрацює

	Директиви препроцесора	
147.	Що таке препроцесор в мові С?	А) Компілятор Б) Програма, яка обробляє код до компіляції В) Редактор коду Г) Дебагер Д)Інтерпетатор
148.	Яка директива використовується для включення заголовкового файлу?	А) include Б) define В) ifdef

		Г) pragma Д) endif
149.	Як правильно включити стандартний заголовковий файл <stdio.h>?	А) include "stdio.h" Б) include <stdio.h> В) #include "stdio.h" Г) #include <stdio.h> Д) #define <stdio.h>
150.	Як правильно визначити макрос PI зі значенням 3.14159?	А) #define PI = 3.14159 Б) #define PI 3.14159 В) define PI = 3.14159 Г) define PI 3.14159 Д) define PI : 3.14159
151.	Як перевірити, чи визначений макрос DEBUG?	А) #if DEBUG Б) #ifdef DEBUG В) #if defined(DEBUG) Г) Всі варіанти вірні Д) Всі варіанти невірні
152.	Яка директива використовується для завершення умовної компіляції?	А) #end Б) #endif В) #terminate Г) #finish Д) #stop
153.	Яка директива використовується для виведення повідомлення про помилку під час компіляції?	А) #error Б) #warning В) #message Г) #print Д) #finish

154.	Що робить директива #include <filename.h>	А) Включає файл filename.h з поточного каталогу Б) Включає файл filename.h з системного каталогу В) Включає файл filename.h з будь-якого каталогу Г) Не робить нічого Д) Виключає файл filename.h з будь-якого каталогу

155.	Який результат виконання коду:
	#define SUM(a, b) a + b
	int main() {
	int x = SUM(2, 3) * 4;
	printf("%d", x);
	return 0;
	}
		А) 20
		Б) 14
		В) 11
		Г) 9
		Д) 24

156.	Що робить директива #define SQUARE(x) x * x?	А) Визначає функцію SQUARE Б) Визначає макрос SQUARE В) Визначає змінну SQUARE Г) Визначає тип даних SQUARE Д) Визначає процедуру SQUARE

157.	Який результат виконання коду:
	#define MAX(a, b) (a > b ? a : b)
	int main() {
	int x = MAX(5, 10);
	printf("%d", x);
		А) 5
		Б) 10
		В) 15
		Г) 0
		Д) 20

	return 0; }	
158.	Яка директива використовується для визначення макросу з аргументами?	А) #define macro(arg1, arg2) Б) #define macro arg1 arg2 В) #macro arg1 arg2 Г) #macro(arg1, arg2) Д) #define arg1 arg2
159.	Яка директива використовується для видалення макросу?	А) #undef Б) #delete В) #remove Г) #clear Д) #delete macro
160.	Яка директива використовується для виведення попередження під час компіляції?	А) #error Б) #warning В) #message Г) #print Д) #message error

Відповіді на тести для самоконтролю

№ з/п	Питання	Правильна відповідь
Типи даних в мові С. Операції та оператори.		
1.	Який тип даних використовується для оголошення символьних змінних?	В. char
2.	Який модифікатор використовується для оголошення беззнакових змінних?	Г. unsigned
3.	Яка операція використовується для присвоєння значень змінним?	Б. =
4.	Яка операція використовується для об'єднання двох байтів по логіці «І»?	Г. &
5.	Яка операція використовується для віднімання з присвоюванням значення змінній?	Б. -=
6.	Який вираз відповідає оголошенню змінної цілого типу з ініціалізацією значення?	Б. int x = 1;
7.	Який вираз відповідає операції інкремента в префіксній формі?	Г. ++i;
8.	Вираз i++ у мові С еквівалентний виразу:	В. i = i + 1
9.	Як позначається операція перевірки рівності двох виразів?	Б. ==
10.	Як позначається операція побітового зсуву на 1 розряд вліво?	В. <<1
11.	Тип даних int дозволяє зберігати у пам'яті значення:	Б. цілого числа
12.	Тип даних double дозволяє зберігати у пам'яті значення:	А. дійсного типу
13.	Яким буде значення змінної a у виразі int a = 9/2+9%4 у мові С?	А. 5
14.	Як позначається операція побітового "І"?	В. &
15.	Яка з нижче перерахованих операцій є унарною?	А. ++
16.	Як позначається операція побітового "АБО"?	**В.
17.	Вираз i-- у мові С еквівалентний виразу:	В. i = i - 1
18.	Який тип даних дозволяє зберігати невід'ємні цілі числа?	Б. unsigned long
19.	Яким буде значення виразу int a = 14/2+31%5 у мові С?	Д. 6

20.	Яким символом позначається операція взяття адреси змінної?	А. &
21.	Тип даних char дозволяє зберігати у пам'яті значення:	В. цілого символьного типу
22.	Як у мові С записується операція зсуву вправо на 3 розряди?	Г. >>3
23.	Як у мові С записується операція зсув вліво на 2 розряди?	Г. <<2
24.	Якщо є код int x = 5; x++; яке значення матиме x?	Д. 6
25.	Якщо є код int x = 5; x--; яке значення матиме x?	Г. 4
26.	Якщо є код int x = 5; x += 3; яке значення матиме x?	Г. 8
27.	Якщо є код int x = 5; int y; y = x % 2; яке значення матиме y?	Б. 1
28.	Якщо є код int x = 5; int y; y = abs(x); яке значення матиме y?	В. 5
29.	Якщо є код float x = 4; float y; y = sqrt(x); яке значення матиме y?	В. 2
30.	Якщо є код float x = 3; float y; y = pow(x, 2); яке значення матиме y?	Б. 9
31.	Дано float f = 32.51; printf("%2.1f", f); що буде виведено на екран?	А. 32.5
32.	Який результат виконання операції 0x5 & 0x6?	Б. 4
33.	Який результат виконання операції `0x3	0x4`?
34.	Який результат виконання операції (0010) << 1?	Д. 0100
35.	Який результат виконання операції (1000) >>2?	В. 0010
36.	Який розмір пам'яті потрібний для int (16-разрядні платформи)?	Б. 2 байта
37.	Який розмір пам'яті потрібний для char?	А. 1 байт
38.	Який розмір пам'яті потрібний для float?	В. 4 байта
39.	Яким буде значення змінної a у int a = 27/2-13%4?	Б. 6
40.	Яким буде значення змінної a у int a = 12/2+32%5?	Д. 6
41.	Який тип даних займає 1 байт пам'яті?	Г. char

42.	Яка з перерахованих операцій є бінарною?	Б. &
43.	Чому дорівнює a у int a=3; a *= (a *=a);?	Г. 81
44.	Чому дорівнює x у x = (y = 3, y + 1);?	Г. 4
45.	Чому дорівнює c у int a=1, b=3, c; c =(b+a++, a+--b);?	Г. 3
46.	Чому дорівнює b у int a=3, b=8, c; c=(a++, a+b); (b--,c)*=3;?	Б. 11
47.	Визначити p після p*=x++, якщо всі змінні мали значення 5	А. 25
	Розгалудження та цикли.	
48.	Якщо є код: int x = 4; int y; if (x < 4) y = 2*x; else y = x;. Яке значення матиме y?	Б. 4
49.	Для заданого виразу знайти тотожний: if ((x == 0) && (y == 0)) ...	Г. if (!x && !y) ...
50.	Що буде виведено на екран при виконанні фрагмента коду: for (int i = 3; i > 0; i--) printf("%d ", 2*i);	Г. 6 4 2
51.	Що буде виведено на екран при виконанні фрагмента коду: for (int i=0; i<3; i++) printf("%d ", 2*i);	В. 0 2 4
52.	Що буде виведено на екран при виконанні фрагмента коду: int x=7; if (x%2==0) printf("0"); else printf("1");	Б. 1
53.	Що буде виведено на екран при виконанні фрагмента коду: int a=2, b=5, y; y = (a > b) ? a : b; printf("%d", y);	Д. 5
54.	Що буде виведено на екран при виконанні фрагмента коду: int sum=0, i=2; while(i<4) {sum+=i; i++;} printf("%d", sum);	Г. 4
55.	Що буде виведено на екран при виконанні фрагмента коду: int sum=0, i=2; do {sum+=i; i++;} while(i<4); printf("%d",sum);	Г. 5
56.	Що буде виведено на екран при виконанні фрагмента коду: int sum=0; for (int i=1; i<8; i++) {sum+=i; if(i==4) break;} printf("%d",sum);	В. 10
57.	Якщо є код: int x = 0; int y; while(x < 4) {y = 2*x; x++;}. Яке значення матиме y?	Г. 6
58.	Якщо є код: int x = 4; int y; while(x > 0) {y = 3*x; x-- ;}. Яке значення матиме y?	Д. 6

59.	Що буде виведено на екран при виконанні фрагмента коду: int x = 4 , y; if(x > 3) y = x*x; else y = x; printf("%d",y);	Г. 16
60.	Що буде виведено на екран при виконанні фрагмента коду: int n = 0; for(int i = 0; i < 5; i++) if(i % 2 == 0) n++; printf("%d ", n);	Б. 2
61.	Що буде виведено на екран при виконанні фрагмента коду: int n = 1; for(int i = 1; i < 5; i++) n = n * i; printf("%d ", n);	Г. 24
62.	Що буде виведено на екран при виконанні фрагмента коду: int i = 4, sum = 0; while(i >0) { if(i % 2 == 0) sum += i; i--; } printf("%d ", sum);	Б. 6
63.	Дано фрагмент коду на мові С: int v = 0X24; printf("%d", v);. Що буде виведено на екран?	В. 36
64.	Дано фрагмент коду на мові С: int i = 0,n = 0; while(i< 7) { if(i % 2 == 0) n++; i++; } printf("%d ", n);	Б. 2
65.	Що буде виведено на екран при виконанні фрагмента коду: int i = 4, n = 0; while(i >0) { if(i % 2 == 0) n++; i--; } printf("%d ", n);	Б. 2
66.	Визначити скільки коренів матиме квадратне рівняння, якщо a=1, b=-5, c=4: double d = b*b-4*a*c; int n=(d<0.0)?0:(d>0.0)?2:1; printf(" корней: %d\n",n);	Г. 2
67.	Вкажіть що виведе на екран поданийфрагмент коду: for (int i = 1; i<3; i++) for (int j = 3; j>0; j--){ printf("%d ", i+j); }	В. 4 3 2 5 4 3
68.	Вкажіть що виведе на екран поданийфрагмент коду: int ch = 1, k = 10, sum = 0; while (ch < k){ sum += ch; ch++; printf("%d ", ch); }	Г. 1 2 3 4 5 6 7 8 9
69.	Вкажіть що виведе на екран поданийфрагмент коду: int a = 1; switch (a){ case 1: case 2: case 3: printf("Three"); }	В. Three
70.	Вкажіть що виведе на екран поданийфрагмент коду: int a = 1; switch (a){ case 1: printf("One"); case 2: printf("Two"); case 3: printf("Three"); }	Г. One Two Three
71.	Вкажіть що виведе на екран поданийфрагмент коду: double a = 1; switch (a){ case 1: printf("One"); break; case 2: printf("Two"); break; case 3: printf("Three"); break; }	А. One

72.	Вкажіть що виведе на екран поданийфрагмент коду: if (5 = 5) printf("Yes"); else printf("No");	Д. Помилка, код не відпрацюється
73.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int a = 10; if (a = 5) printf("Yes"); else printf("No");	А. Yes
74.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int x = 5, y = 10; if (x<10){ if (y >= 10) printf("1"); printf("2"); } else printf("3");	Г. 12
75.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int num = 200; if (num < 750) printf("num < 750"); else if (num < 450) printf("num < 450"); else printf("num < 150");	А. num < 750
76.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int ch = 0, k = 4; while (-3){ if (ch >= k) break; printf("%d ", ch); ch += 3; }	В. 0 3
77.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int i = 4, k = -2; do { printf("%d ", k); i += 2; } while (k >= i);	Б. -2
78.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int i = 4; do { printf("Hello"); i++; if (i > 0) continue; printf("%d ", i); } while (i<3);	В. Hello 4
79.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int i = 4; for(; ;){ printf("good!"); if (3<=5) break; }	А. good!
80.	Чи виконається поданийфрагмент коду, якщо так – вкажіть що виведена екран? int i = 0, a = 4; while (i < a){ printf("%d", a-i); i++; }	Б. 4 3 2 1
81.	Чи виконається поданийфрагмент коду, якщо так – що відбудеться в результаті його виконання? `int ar[10]; int sum = 0; for (int i = 0; i<10; i++) ar[i] = i	Б. 20
82.	Чи виконається поданийфрагмент коду, якщо так – що відбудеться в результаті його виконання? int size = 10; float arr[size] = { 0 }; for (int i = 0; i < size; i++) printf("%d ", arr[i]);	Д. Помилка, код не відпрацює

	Функції.	
83.	Вкажіть що виведе на екран поданий фрагмент коду? int sum(int a, int b){ return a + b; } void main(){ printf("%d ",sum(10, 12)); }	В. 22
84.	Вкажіть що виведе на екран поданий фрагмент коду? void sum(int a, int b){ return a + b; } void main(){ printf("%d ",sum(10, 12)); }	Д. Помилка, код не відпрацюється
85.	Вкажіть що виведе на екран поданий фрагмент коду? void Show(char y, int z = 5, float x = 7.7){ printf("%d %.1f %c", z,x,y); } void main(){ Show('W'); }	Б. 5 7.7 W
86.	Яка функція призначена для читання рядків із потоку stdin у мові С?	Б. gets( )
87.	Яка з перерахованих функцій повертає значення дійсного типу?	В. double func(int x);
88.	Яка з перерахованих функцій нічого не повертає?	В. void func(float x);
89.	Яка з перерахованих функцій повертає значення символьного типу?	Д. char func(float x);
90.	Яка з перерахованих функцій приймає значення дійсного типу?	А. void func(float x);
91.	Яка з перерахованих функцій повертає значення цілого типу?	Г. int func(int x);
92.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? int var = 10; int* pVar = &var; *pVar = 9; printf("%d ", *pVar);	Б. 9
93.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? int var = 5; int* pVar = &var; *pVar = 9; printf("%d ", pVar);	Г. Адресу змінної var
94.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? int z[5] = { 1, 2, 3, 4, 5 }; int* zPtr = &z; for (int i = 0; i<5; i++) printf("%d ", *(zPtr + i));	А. 1 2 3 4 5
95.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? int num = 100; int*numPtr = # ++*numPtr; printf("%d ", num);	А. 101
96.	Чи виконається поданий фрагмент коду, якщо	Б. 1000

	так – вкажіть що виведена екран? int num = 100, rez = 10; int*numPtr = # rez *= *numPtr; printf("%d ", rez);	
97.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? int z[5] = { 1, 2, 3, 4, 5 }; int* zPtr = &z[0]; for (int i = 0; i<5; i++) printf("%d ", *(zPtr + i));	А. 1 2 3 4 5
	Масиви. Структури та об'єднання	
98.	Який вираз відповідає оголошенню двовимірного статичного масиву?	Г. double m[5][4];
99.	Що буде виведено на екран при виконанні фрагмента коду: char str[] = "Test"; printf("%d\n", strlen(str));	А. 5
100.	Виберіть правильну форму оголошення рядка довжиною 32:	А. char s[32];
101.	Який розмір пам'яті потрібний для збереження одновимірного масиву цілих чисел(int для 16-разрядних платформ) із 20 елементів?	Б. 40 байт
102.	Який розмір пам'яті потрібний для збереження одновимірного масиву дійсних чисел типа float із 10 елементів?	Б. 20 байт
103.	Який розмір пам'яті потрібний для збереження одновимірного масиву дійсних чисел типа double із 10 елементів?	Д. 80 байт
104.	Що обчислює цей фрагмент коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i = 0; i<5; i++) if(arr[i] % 2 != 0) sum=sum + arr[i];	Д. Суму непарних елементів масиву
105.	Що обчислює цей фрагмент коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 == 0) sum=sum + arr[i];	В. Суму парних елементів масиву
106.	Що обчислює цей фрагмент коду: int i, p = 1; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) p=p * arr[i];	Г. Добуток елементів масиву
107.	Що обчислює цей фрагмент коду: int i, p = 1; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 == 0) p=p * arr[i];	В. Добуток парних елементів масиву
108.	Що обчислює цей фрагмент коду: int i, p = 1;	В. Добуток

	int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 != 0) p=p * arr[i];	непарних елементів масиву
109.	Що буде виведено на екран при виконанні фрагмента коду: int i, p = 1; int arr[] = { 3,1,2,1,4 }; for(i=0; i<5; i++) p=p * arr[i]; printf("%d ", p);	Г. 24
110.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) sum=sum + arr[i]; printf("%d ", sum);	Б. 15
111.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 == 0) sum=sum + arr[i];	В. 6
112.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 != 0) sum=sum + arr[i];	Г. 9
113.	Що буде виведено на екран при виконанні фрагмента коду: int i, p = 1; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 != 0) p=p * arr[i];	Д. 15
114.	Що буде виведено на екран при виконанні фрагмента коду: int i, p = 1; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 == 0) p=p * arr[i];	Г. 8
115.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] > 2) sum=sum + arr[i];	Г. 14
116.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] < 3) sum=sum + arr[i];	А. 1
117.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) printf("%d ", arr[i]);	Г. 3 5 2 1 4
118.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 == 0) printf("%d ",	В. 2 4

	arr[i]);	
119.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] % 2 != 0) printf("%d ", arr[i]);	Г. 3 5 1
120.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] > 2) printf("%d ", arr[i]);	Д. 3 5
121.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] <= 3) printf("%d ", arr[i]);	А. 3 2 1
122.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i % 2 == 0) printf("%d ", arr[i]);	Д. 3 2
123.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i % 2 != 0) printf("%d ", arr[i]);	Г. 5 1
124.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i > 1) printf("%d ", arr[i]);	Б. 5 2 1
125.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,6,4 }; for(i=0; i<5; i++) if(i <= 2) printf("%d", arr[i]);	А. 3 5 2
126.	Що буде виведено на екран при виконанні фрагмента коду: int i; int arr[] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i > 0 && i <= 3) printf("%d ", arr[i]);	Г. 5 2 1
127.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i > 1) sum += arr[i]; printf("%d ", sum);	В. 7
128.	Що буде виведено на екран при виконанні фрагмента коду: int i, sum = 0; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(i < 3) sum += arr[i]; printf("%d ", sum);	В. 8
129.	Що буде виведено на екран при виконанні	Б. 2

	фрагмента коду: int i, n = 0; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] > 3) n ++; printf("%d ", n);	
130.	Що буде виведено на екран при виконанні фрагмента коду: int i, n = 0; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] < 4) n ++; printf("%d ", n);	Г. 3
131.	Що буде виведено на екран при виконанні фрагмента коду: int i, n = 0; int arr[ ] = { 3,5,2,1,4 }; for(i=0; i<5; i++) if(arr[i] >1 && arr[i] < 5) n ++; printf("%d ", n);	В. 3
132.	Для оголошення об'єднань використовується ключове слово:	А. union
133.	Для оголошення структур використовується ключове слово:	А. struct
134.	Є оголошення: struct Person { int n; char name[10]; } p; Який розмір пам'яті потрібний для збереження змінної p?	Г. 16 байт
135.	Є оголошення: struct Person { int n; char name[10]; } P[10]; Який розмір пам'яті потрібний для збереження змінної P?	В. 120 байт
136.	Є оголошення: struct Point { double x; double y; } point; Який розмір пам'яті потрібний для збереження змінної point?	Г. 16 байт
137.	Визначити діапазон генерації псевдовипадкових чисел: rand()%41-20;	В. [-20, 20)
138.	Визначити діапазон генерації псевдовипадкових чисел: 0.01*(rand()%101);	Б. [0.01, 1]
139.	Визначити діапазон генерації псевдовипадкових чисел: 80 + rand()%(100 - 80 + 1)	Г. [20; 100]
140.	Визначити діапазон генерації псевдовипадкових чисел: (double) rand()/RAND_MAX	А. [0,1]
141.	Визначити діапазон генерації псевдовипадкових чисел: 1 + rand() % 10	Г. [1,10]
142.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? struct test { int a; double b; char arr[3]; }; void main(){ printf("%d ", sizeof(test)); }	В. 24

143.	Чи виконається поданий фрагмент коду, якщо так – вкажіть що виведена екран? struct person { char name[20], surname[20]; int age; } one = { "Ivan", "Sidorov", 50 }, *second = &one; void main(){ strcpy_s(one.surname, "Ivanov"); printf("%s ", (second->name)); printf("%s ", second->surname); printf("%d ", second->age); }	Б. IvanSidorov 50
144.	Чи виконається поданий фрагмент коду, якщо так – що відбудеться в результаті його виконання? int ar[2][2] = { { 1,2 },{ 3,4 } }; int** mas = ar; for (int i = 0; i<2; i++) for (int j = 0; j<2; j++) printf("%d ", mas[i][j]);	Д. Помилка, код не відпрацює
145.	Чи виконається поданий фрагмент коду, якщо так – що відбудеться в результаті його виконання? int n=5; int *a =(int *) malloc(n*sizeof(int)); for(int i=0; i<n; i++) scanf("%d", &a[i]);	Б. Буде створено одновимірний масив на n елементів
146.	Чи виконається поданий фрагмент коду, якщо так – що відбудеться в результаті його виконання? int n=5; int *a =(int *) calloc(n,sizeof(int)); for(int i=0; i<n; i++) scanf("%d", &a[i]);	А. Буде створено одновимірний масив на n елемент
	Директиви препроцесора	
147.	Що таке препроцесор в мові С?	Б. Програма, яка обробляє код до компіляції
148.	Яка директива використовується для включення заголовкового файлу?	А. include
149.	Як правильно включити стандартний заголовковий файл <stdio.h>?	Г. #include <stdio.h>
150.	Як правильно визначити макрос PI зі значенням 3.14159?	Б. #define PI 3.14159
151.	Як перевірити, чи визначений макрос DEBUG?	Г. Всі варіанти вірні
152.	Яка директива використовується для завершення умовної компіляції?	Б. #endif
153.	Яка директива використовується для виведення повідомлення про помилку під час компіляції?	А. #error

154.	Що робить директива #include <filename.h>?	Б. Включає файл filename.h з системного каталогу
155.	Який результат виконання коду: #define SUM(a, b) a + b int main() { int x = SUM(2, 3) * 4; printf("%d", x); return 0; }	Б. 14
156.	Що робить директива #define SQUARE(x) x * x?	Б. Визначає макрос SQUARE
157.	Який результат виконання коду: #define MAX(a, b) (a > b ? a : b) int main() { int x = MAX(5, 10); printf("%d", x); return 0; }	Б. 10
158.	Яка директива використовується для визначення макросу з аргументами?	А. #define macro(arg1, arg2)
159.	Яка директива використовується для видалення макросу?	А. #undef
160.	Яка директива використовується для виведення попередження під час компіляції?	Б. #warning

Література

1. Aditya Y. Bhargava. Grokking Algorithms, Second Edition. Manning; 2nd edition (March 26, 2024). – 320 p.

2. Aria Thane. C Programming: All-in-One Resource for C Programming, Comprehensive Tutorials, Expert Tips, and a Wide Range of Exercises for All Skill Levels. Aria Thane; 1st edition (December 30, 2023) – 207 p.

3. Mailund T. Pointers in C Programming. A Modern Approach to Memory Management, Recursive Data Structures, Strings, and Arrays / Thomas Mailund. – New York: Apress, 2021. – 552 c.

4. Horton I. Beginning C: From Beginner to Pro 7th ed. Edition / I. Horton, G. Gonzalez-Morris. – New York: Apress, 2024. – 726 c.

5. Preschern C. Fluent C: Principles, Practices, and Patterns / Christopher Preschern. – New York: O\'Reilly Media, 2022. – 304 c.

6. Sandler N. Writing a C Compiler: Build a Real Programming Language from Scratch / Nora Sandler., 2024. – 792 p.

7. Szuhay Jeff. Learn C Programming - Second Edition: A beginner's guide to learning the most powerful and general-purpose programming language with ease 2nd ed. Edition. 2022 – 742 p.

8. ДСТУ ISO 5807:2016 (ISO 5807:1985, IDT). Символи та угоди щодо документації стосовно даних, програм та системних блок-схем, схем мережевих програм та схем системних ресурсів. [Чинний від 10.10.2016]. К.: ДП "УкрНДНЦ", 2016. 30 с.

9. Войтенко В. В. С/С++: Теорія та практика: навч. посіб. / В. В. Войтенко, А. В. Морозов. Житомир: ЖДТУ, 2004. – 324 с

10. Мартін Р. Чиста архітектура / Роберт Сесіл Мартін. – Харків: Фабула, 2019. – 359 с.

Для нотаток

Навчальне видання

Морозов Андрій Васильович

Марчук Галина Вікторівна

Левківський Віталій Леонідович

Левченко Антон Юрійович

С: теорія та практика

Навчальний посібник

Редактори: Морозов А.В., Марчук Г.В., Левківський В.Л.,
Левченко А.Ю.
Комп'ютерний набір та верстка Марчук Г.В., Левківський В.Л.

Підписано до видання 24.03.2025 Формат 60x84 1/16.
Гарнітура Times New Roman. Ум. друк. арк. 8,55.

Редакційно-видавничий відділ
Державного університету «Житомирська політехніка»
10005, м. Житомир, вул. Чуднівська, 103.