О.В. Коробань

ПРОГРАМУВАННЯ НА МОВІ С

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УМАНСЬКИЙ ДЕРЖАВНИЙ ПЕДАГОГІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ПАВЛА ТИЧИНИ
ФАКУЛЬТЕТ ІНЖЕНЕРНО-ПЕДАГОГІЧНОЇ ОСВІТИ
КАФЕДРА ПРОФЕСІЙНОЇ ОСВІТИ ТА ТЕХНОЛОГІЙ ЗА ПРОФІЛЯМИ

ПРОГРАМУВАННЯ НА МОВІ С

Навчальний посібник

Умань - 2023

УДК 519.682.2

Коробань О.В. Програмування на мові С. Навчальний посібник для студентів освітнього ступеня «бакалавр» спеціальності 015.39 «Професійна освіта. Цифрові технології» / Укл.: О.В. Коробань. Умань: Візаві, 2023. 130 с.

Укладач: Коробань О.В., старший викладач кафедри професійної освіти та технологій за профілями

Рецензенти:

Марія Медведєва – к.пед.н, доцент, зав.кафедрою інформатики та інформаційно-комунікаційних технологій УДПУ імені Павла Тичини

Олексій Мельник – к.т.н., доцент, зав.кафедрою професійної освіти та технологій за профілями УДПУ імені Павла Тичини

Рекомендовано Вченою радою
факультету інженерно-педагогічної освіти
Уманського державного педагогічного університету
імені Павла Тичини

Навчальний посібник «Програмування на мові С» рекомендований для бакалаврів освітньо-професійної програми «Професійна освіта. Комп'ютерні технології» та є важливим джерелом інформації для тих, хто бажає освоїти мову програмування С. Цей посібник пропонує систематизоване та детальне вивчення різних аспектів мови С, починаючи з основних концепцій і закінчуючи більш складними темами, такими як використання вказівників для роботи з динамічними структурами даних. Такий підхід сприяє кращому засвоєнню матеріалу студентами та простішому розумінню послідовності концепцій. Загалом, навчальний посібник "Програмування на мові С" є цінним ресурсом для початківців у програмуванні, які бажають отримати тверді основи з мови С.

ЗМІСТ

1. ВСТУП

1.1 Історія розвитку обчислювальної техніки

Вважається, що перша електронна обчислювальна машина була винайдена в 1943 році.

Перший комп'ютер було побудовано на електронних вакуумних лампах.

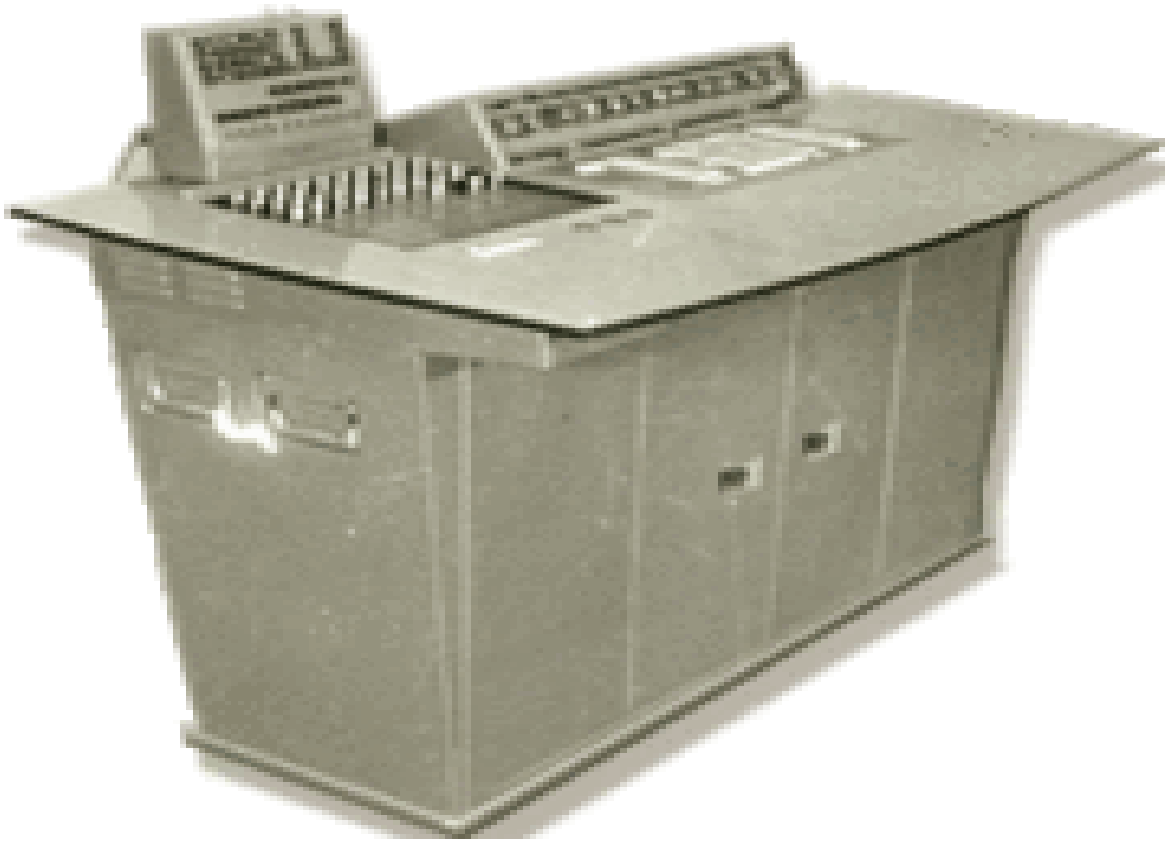
Перші комп'ютери, що були побудовані на електронних вакуумних лампах, належать до першого покоління обчислювальних машин і являли собою досить великі споруди.

Рисунок 1.1 – ЕОМ першого покоління Урал-4

До другого покоління електронних обчислювальних відносять такі, у яких деякі частини мали власні контролери, а також програмне забезпечення, що спрощувало роботу з ЕОМ. У машинах другого покоління стали використовувати транзисторну елементну базу

Рисунок 1.2 – ЕОМ другого покоління Мінськ-32

У цей же час з’явились перші міні-ЕОМ, рисунки 1.3-1.4.



,

Рисунок 1.3 – Міні ЕОМ «Промінь»

Рисунок 1.4 – Міні ЕОМ «МИР»

Третє покоління електронних обчислювальних машин створювалося з використанням інтегральних схем. Це дозволило зробити їх порівняно компактними і знизити витрати на виробництво. У цей період часу створюються перші серійні комп'ютери IBM 360/370.

У Радянському Союзі починають випускати ЕОМ серії ЕС.

Рисунок 1.5 –ЕОМ 3-го покоління «ЕС-1020»

Сучасні комп'ютери є представниками ЕОМ четвертого покоління.

Елементною базою цих комп'ютерів є ВІС(великі інтегральні схеми) Використання ВІС суттєво зменшила габарити ЕОМ та їх вартість.

1.2 Досягнення вітчизняних вчених та інженерів у цій галузі.

В Україні перший комп’ютер було збудовано під Києвом, у Феофанії. 4 січня 1952 р. Президія АН СРСР заслухала доповідь Сергія Лебедєва про введення в експлуатацію малої електронно-цифрової обчислювальної машини "МЭСМ".

В 1952 р. "МЭСМ" була практично єдиною в країні ЕОМ, на якій проводились обчислення найважливіших науково-технічних задач в галузі термоядерних процесів, космічних польотів та ракетної техніки.

Перші радянські міні ЕОМ також були зроблені в Україні. Це були ЕОМ «Промінь» та «Мир» (машина інженерних розрахунків)

1.3 Системи і середовища програмування

Системи програмування – це інтегровані середовища розробки програм, до складу яких входять редактори текстів, дебагери, транслятори и т.п.

- Delphi
- Eclipse
- QT-creator

1.4 Класифікація мов програмування

Мови поділяють на універсальні та спеціалізовані.

Універсальні дозволяють вирішувати широке коло задач.

Спеціалізовані використовуються для вирішення конкретних задач.

Мови прикладного програмування використовуються для розробки програм, що використовуються в економіці, техніці та інших сферах діяльності

людини

Мови системного програмування використовуються для створення програм, які забезпечують керування такими компонентами комп'ютерної системи як процесор, оперативна пам'ять, пристрої введення-виведення, мережеве обладнання.

Такі програми створюють «інтерфейс», з одного боку якого апаратура, а з іншого – застосування користувача

- Асемблер
- Бейсік
- Фортран
- Алгол
- Паскаль
- Си
- Ява …

1.5 Вибір мови програмування

Мов дуже багато. Вибір залежить перш за все від задачі, яка вирішується, а окрім того від традицій та вподобань керівництва фірми-розробника, наявності ліцензій на використання систем програмування, кваліфікації та уподобань програміста.

2 БАЗОВІ ЕЛЕМЕНТИ МОВИ С

2.1 Поняття про програму на мові С.

Мова С створювалася інженерами лабораторії Bell Кеном Томпсоном і Денісом Рітчі на протязі 1969-1973 років. Результати були опубліковані у 1979 році.

Програма на мові С – це файл, який має розширення .c, або .cpp, якщо програма написана на С++.

Файл програми являє собою сукупність різноманітних оголошень та функцій, які розташовуються певним чином, рисунок 2.1.

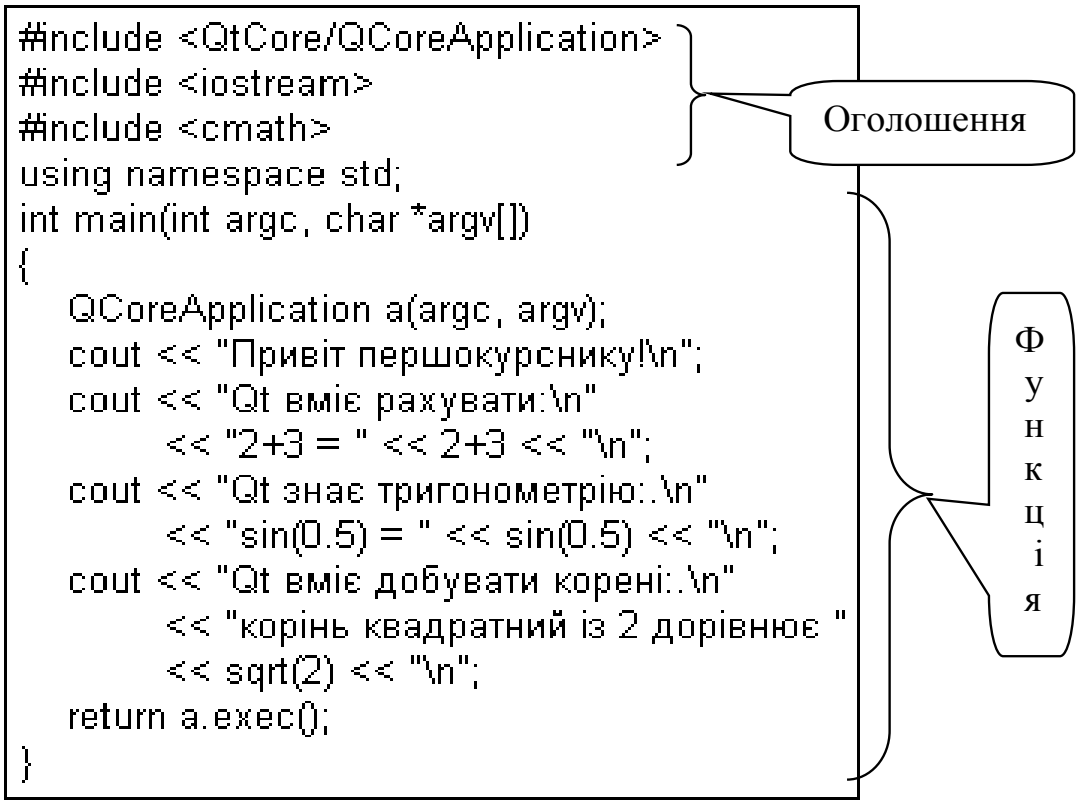


Рисунок 2.1 – Файл програми на С

2.2 Основні риси мови програмування С.

Мову С найчастіше використовують системні програмісти, хоча нею можна написати будь які програми.

Програмування на С потребує від програміста уважності і обережності. Brian W. Kernighan, автор першої книги з Сі так висловився про цю мову: «Сі — інструмент, гострий, як бритва: за його допомогою можна створити і елегантну програму, і кроваве місиво».

Засоби мови дозволяють писати компактні програми, але читати такі тексти не завжди просто.

Складовими частинами мови є:

- символи (алфавіт мови) - це основні неподільні знаки, за допомогою яких пишуться всі тексти на мові програмування;
- лексеми (слова) - мінімальні одиниці мови, які мають самостійний зміст;
- вирази - задають правило обчислення деякого значення;
- оператори (інструкції) задають опис деякої дії.

2.3 Алфавіт мови

- великі та малі латинські літери: A-Z, a-z.
- арабські цифри;
- символи: графічні та ескейп-послідовності (символи табуляції, символ переходу на наступний рядок тощо);
- символи , . ; : ? ' ! | / \ ~ ( ) [ ] { } < > # % ^ & - + * =

2.4 Лексеми.

Лексеми – це слова мови. До лексем відносять ключові слова, ідентифікатори та константи.

2.4.1 *Ключові слова*

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**

case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

2.4.2 *Ідентифікатори*

Ідентифікатори – це імена змінних, функцій та деяких інших об'єктів програми. Основні правила такі:

- мають починатися з букви;
- можуть містити цифри;
- мають писатися без пропусків.

У Сі великі і маленькі літери, що використовуються для імен, розрізняються! Але,у Сі дуже полюбляють знак підкреслення.

max_int_number замість **maxIntNumber**

2.4.3 *Константи*

2.4.3.1 Константи цілочислові

Десяткові: 12345 -17 +1098

Вісімкові: 0110 077 01234567

Шістнадцяткові: 0x1230xF3a2E

2.4.3.2 Дійсні константи

З фіксованою крапкою: 1.2345 -0.017 +1098.2345

З плаваючою крапкою: 123.45e-2 1.0982345E3

2.4.3.3 Символьні константи

Прості символи: ’A’ ’b’ ’щ’ ’[’ ’\’

Ескейп-послідовності ’\n’ ’\a’ ’\’’ ’\t’

2.4.3.4 Рядки символів

”Привіт першокурснику!\n”

”Ми вивчаємо курс \”Програмування на С\”!”

2.4.3.5 Знаки операцій, роздільники, коментарі

```
/* Односимвольні операції:*/
        +   *     <     ^     /     …
// Багатосимвольні операції:
        ++  ||    >>    +=    <=    <<=
// Роздільники:
        ( )  [ ]   { }   ,     ;     :     =     *     #
```

2.5 Структура програми на С.

Програма на мові С являє собою текстовий файл, що містить сукупність різноманітних оголошень та функцій, які розташовуються певним чином, рисунок 2.1.

Оголошення – це рядок тексту, який містить службову.

У наведеній на рисунку 2.1 програмі оголошення using namespace std підключає нашу програму до простору імен компілятора С++.

До розділу оголошень входять також директиви препроцесору, що починаються символом #

Кожна програма на С та С++обов’язково повинна мати у своєму складі функцію з назвою main(). Це головна функція програми з якої починається виконання кожної програми.

2.6 Поняття про бібліотечні функції та заголовні файли.

На сучасних комп'ютерах будь яка програма не може працювати самостійно. Навіть у найпростіших програмах використовується багато допоміжного коду. Цей код зберігається в бібліотеках програм у вигляді функцій, класів та інших конструкцій.

Коли у програмі з'являється, наприклад, звернення до бібліотечної функції, компілятор має перевірити, чи дійсно існує така функція і чи можна до неї так звертатися. Ця проблема вирішується у різних мовах програмування по різному.

У C та C++ цю проблему вирішують за допомогою заголовних файлів. Ці файли містять не самі функції, а тільки їх заголовки, що називають прототипами функцій. Наявність прототипу дає можливість компілятору перевірити правильність використання функції

Окрім прототипів функцій в заголовні файли виносять, шаблони структур, оголошення користувацьких типів, визначення макросів тощо. Сформовані заголовні файли підключають до кожного програмного файлу, що дає змогу автономно компілювати кожен з них.

Заголовні при'єднуються до програми за допомогою директив препроцесора #include.

2.7 Директиви препроцесора

Препроцесор – це спеціальна програма, що опрацьовує текстовий файл програми на C, C++ перед компіляцією. Він входить як обов'язковий компонент до складу компілятора. Результатом роботи препроцесора є готовий до компіляції текст програми, в якому реалізовані всі задані директиви (інструкції) попереднього опрацювання програми.

Найчастіше на етапі препроцесорної обробки виконують такі дії:

– включення до складу програми текстів заданих файлів;

– вилучення з тексту програми тих частин, які не повинні компілюватись у даній реалізації (умовна компіляція);

– заміна в програмі макроімен на задані вирази підстановки;

– реалізація макропідстановок (макросів) з параметрами.

Кожна директива препроцесора записується в окремому рядку і починається зі знаку #. Крапка с комою після директив не ставиться.

Однією з найпоширеніших директив є директива включення #include. Вона призначена для приєднання до програми текстового файлу, ім’я якого задається в цій директиві. Текст файлу вставляється в програму в тому місці, де була записана директива.

Директива допускає дві форми запису імен файлів:

– #include < ім’я_файлу > // ім’я в кутових дужках

– #include “ім’я_файлу ” // ім’я в лапках

Форма запису імені файлу визначає, де відбувається його пошук. Якщо ім’я файлу записане в кутових дужках, то такий файл повинен зберігатись в спеціальному каталозі середовища програмування (цей каталог часто називають INCLUDE) або в іншому визначеному для компілятора каталозі. Якщо ім’я файлу задається в лапках, то більшість компіляторів починають пошук цього файлу починається з поточного активного каталогу. Якщо файл там не знайдено, то пошук продовжується у каталозі, що відповідає варіанту кутових дужок. Здебільшого в кутових дужках вказують імена стандартних заголовних файлів бібліотечних функцій, а в лапках – імена користувацьких файлів, які повинні бути приєднані до даної програми.

2.8 Етапи виконання програми.

Безпосередньому виконанню програми, що написана на мові С, С++, передує декілька етапів перетворення її програмного коду. Схематично цей процес зображено на рисунку 2.2.

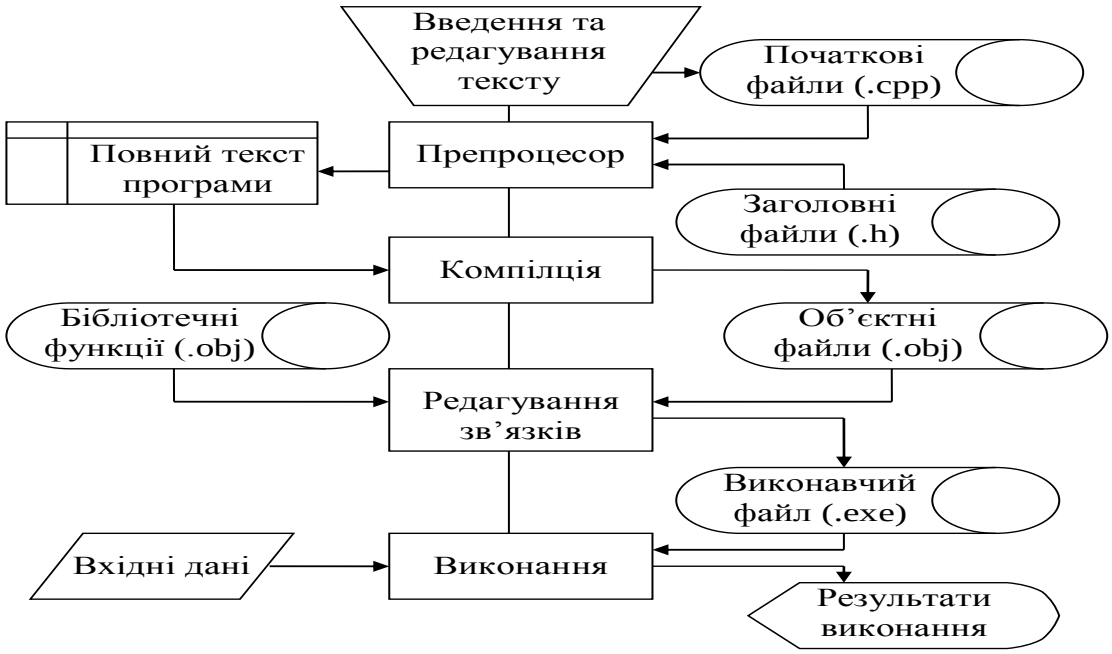


Рисунок 2.2 – Етапи виконання програми

Основними з цих етапів є: препроцесорне опрацювання, компіляція програмних елементів, компонування об'єктних кодів складових частин програм у єдиний виконавчий код у вигляді файлу .exe..

2.9 Використання об'єкту cin для введення даних

У мові С++ введення даних з консолі розглядається як потік символів з клавіатури у програму. Об'єкт **cin** і представляє цей потік у програмі.

Операція >> бере інформацію з цього потоку і перетворює її відповідно до типу змінної, що отримує цю інформацію.

Оператори введення інформації з консолі можуть виглядати так:

```
cin >> x;
cin>>a>>b
      >>c;
```

У першому випадку об'єкт забезпечує введення даних у змінну x.

У другому – до змінних a,b,c.

Об’єкт починає працювати після того, як користувач введе дані з клавіатури і натисне клавішу “Enter”. Вхідні дані розділяють пробілами, табуляцією або переходом до нового рядка.

2.10 Використання об’єкту cout для виведення даних

Об’єкт cout мови С++ являє собою потік, що відповідає за виведення інформації на консоль.

Операція << спрямовує інформацію що підлягає виведенню у потік cout, попередньо перетворивши її у символьну форму. Оператори введення інформації з консолі можуть виглядати так:

```
cout << “Прівіт першокурснику!/n”;
cout << “2 + 3 = “ << 2+3 << “\n”;
```

У подвійних лапках наводиться текст, що підлягає виведенню. Символи «\n» означають, що після виведення тексту слід перейти до нового рядка.

Результатом виконання першого рядка буде поява на консолі тексту "Привіт першокурснику!".

Результатом виконання другого рядка буде поява на консолі тексту "2 + 3 = 5”.

2.11 Створення простішого діалогу з консоллю.

```
1 #include <QtCore/QCoreApplication>
2 #include <iostream>
3 using namespace std;
4 int main(int argc, char *argv[])
5 {
6     QCoreApplication app(argc, argv);
7     cout<< "Hello! I am QT, and what is you name?\n";
8     string name;
9     cin >> name;
10     cout<< "I am glad to see you, "<<name<<"!\n";
11     cout<< "I can calculate. "
12             <<"Enter two number by whitespace, please!\n";
13     float x,y;
14     cin>> x >> y;
15     cout << x <<"+"<<y<<"="<<x+y<<endl;
16     return app.exec();
17 }
```

3 ТИПИ ДАНИХ

3.1 Класифікація типів даних.

Будь-які дані в пам'яті комп'ютера зберігаються як послідовності нулів та одиниць, але для того, щоб визначити, що означає така послідовність, необхідно знати, до якого типу вона відноситься. Тому у мовах C, C++ тип кожної змінної повинен бути обов'язково зазначений при оголошенні.

Наприклад, нехай у пам'яті комп'ютера записана послідовність нулів і одиниць 0100 0010 0100 0011 0100 0100 0000 0000. Якщо розглядати її як ціле число, то це буде 1111704576, а якщо припустити, що це рядок символів, то отримаємо «BCD».

З наведеного прикладу випливає, що тип даних визначає спосіб кодування інформації.

Окрім того, для кожного типу визначено свій набір допустимих операцій над даними та спосіб їх виконання.

Так, наприклад, якщо для розглянутої послідовності виконати операцію «скласти саме із собою», то ця операція буде виконуватися за різними правилами.

Числа будуть складатися за правилами арифметики, і отриманий результат буде виглядати так: 1000 0100 1000 0110 1000 1000 0000 0000 (2223409152).

Рядки ж будуть «склеюватися» і результат буде таким: 0100 0010 0100 0011 0100 0100 0100 0010 0100 0011 0100 0100 0000 0000 («BCDBCD»).

Мова C++ надає програмісту можливість використовувати декілька різновидів типів.

Ці типи можна поділити на такі:

– скалярні, які поділяються на арифметичні, переліки та вказівники;

– тип функція;

– агреговані, що складаються за певними правилами із даних скалярних типів та, можливо, функцій.

Поки що ми розглянемо тільки скалярні арифметичні типи даних.

3.2 Арифметичні типи

До арифметичного типу даних у C++ відносять дані, до яких можна застосовувати усі арифметичні операції.

Ці типи поділяють на основні та модифіковані.

3.2.1 *Основні арифметичні типи*

Основними типами є такі:

char – тип для символів;

int – тип для цілих чисел;

float – тип для дійсних чисел;

double – дійсні числа подвійної точності;

bool – логічний тип, що може приймати значення 0 або 1.

У читача може створитися враження, що тут помилка. Дійсно дивно, що типи char та bool віднесені до арифметичного типу. Але помилки тут нема. Вираз 'Z'*true є допустимим у C і його можна обчислити. Результатом буде число 90, бо true перетворюється у 1, а код символу ’Z’ дорівнює 90. Тобто логічну змінну і символ можна розглядати як цілі числа.

Слід також прийняти до уваги, що тип bool з’явився тільки у мові C++. А в мові C число 0 розглядалося як true, а будь яке інше число розглядалося як false.

3.2.1.1 Модифіковані арифметичні типи

Модифіковані типи отримують за допомогою модифікаторів. Значення модифікаторів та типи, до яких їх можна застосовувати показані у таблиці 3.1.

Таблиця 3.1 – Модифікатори для типів мови C++

Модифікатор	Значення	Застосовують до типів
signed	із знаком	char, int
unsigned	без знаку	char, int
long	довгий	int, long int, double
short	короткий	int

Як бачимо, використання модифікаторів обмежено, за виключенням типу int, до якого можна застосувати будь який модифікатор. Та на практиці область використання ще вужча. Модифікатор signed практичного сенсу не має, бо за принципом замовчування усі типи мають знак. Для знаку виділяється старший біт коду числа. Якщо значення цього біту нуль, то число додатне, якщо одиниця - то від'ємне.

Модифікатор short зменшує удвічі довжину типу int, а модифікатор long збільшує удвічі довжину основного типу. Модифікатор long для типу int можна використовувати двічі. У цьому разі розмір типу int збільшується у чотири рази. Ці модифікатори можна комбінувати з модифікатором unsigned.

У Qt можливі і скорочення, що визначені у заголовному файлі <qtglobal>. Наприклад, замість типу unsigned int можна писати uint. Перелік скорочених назв типів наведено у таблиці 2.2. У цій же таблиці наведено перелік відповідних типів бібліотеки QT, які також визначені у заголовному файлі <qtglobal>.

3.2.1.2 Граничні значення даних цілочислових типів даних

Граничні значення різних типів даних можуть залежати від програмного середовища та типу комп'ютера. Для того щоб зменшити цю залежність, у заголовному файлі <climits> записано набір констант, значення яких дорівнюють мінімальним та максимальним значенням кожного цілочислового типу. Мінімальне значення усіх без знакових констант дорівнює 0.

Таблиця 3.2 – Скорочені назви та граничні константи для типів C++

Повна назва типу	Скорочена назва	Тип Qt	Граничні константи
char	char	qint8	CHAR_MIN
char	char	qint8	CHAR_MAX
unsigned char	uchar	quint8	UCHAR _MAX
short	short	qint16	SHRT_MIN
short	short	qint16	SHRT_MAX

unsigned short	ushort	quint16	USHRT_MAX
int	int	qint32	INT_MIN
int	int	qint32	INT_MAX
unsigned int	uint	quint32	UINT_MAX
long int	long	qint64	LONG_MIN
long int	long	qint64	LONG_MAX
unsigned long int	ulong	quint64	ULONG_MAX
long long int	long long	qlonglong	LLONG_MIN
long long int	long long	qlonglong	LLONG_MAX
unsigned long long int	?	qulonglong	ULONG_LONG_MAX

3.3 Переліки

Переліки – це типи, яким відповідають набори цілочислових констант, кожна з яких має унікальне ім'я.

Оголошення переліку має такий вигляд:

```
enum <ім'я переліку> {<список іменованих констант>};
```

Наприклад, якщо ми захочемо перефарбувати вікно консолі, або змінити колір тексту, що виводиться на консоль, нам потрібно буде оперувати з поняттям колір. Кожний колір для консолі кодується цілим числом. Нулю відповідає чорний, одиниці – синій і т.д. Але пам'ятати номери кольорів не дуже зручно. Зручніше користуватися іменами з такого переліку:

```
enum consoleColors{
   BLACK, BLUE, GREN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY,  LIGHTBLUE,
   LIGHTGREN, LIGHTCYAN, LIGHTRED,  LIGHTMAGENTA, YELLOW, WHITE
};
```

Стандартно перший елементу переліку має значення 0, а кожний

наступний має значення на 1 більше, ніж попередній.

У разі потреби елементам можна присвоювати інші значення.

3.4 Описи змінних

Усі змінні у програмі на мові С мають бути описані, (оголошені). Оголошення змінної – це інструкція, в якій зазначається ім'я змінної та її тип.

У простому випадку інструкція оголошення змінної виглядає так:

```
<тип> <имя>;
```

де: <ім'я> - ім'я змінної;

<тип> - ім'я типу для оголошуваної змінної.

В одній інструкції можна оголошувати декілька змінних одного типу. У цьому випадку імена змінних розділяються комами.

У розділі може бути і декілька інструкцій оголошення, наприклад:

```
int i, j, k ;
float x;
```

Описуючи змінну, можна одразу надати їй значення. Це зветься ініціалізацією. Інструкція ініціалізації змінної виглядає так:

```
<тип> <имя> = < вираз ініціалізації >;
```

У якості виразу ініціалізації може бути константа, змінна, яка вже має значення, або вираз.

Якщо значення, що прийняла змінна після ініціалізації не повинно змінюватися, то для таких змінних використовують кваліфікатор const. Використання такого кваліфікатора перетворює змінну у константу. Нижче наведено приклад оголошення константи 2π.

```
const double twoPi=2*3.1425926;
```

3.4.1 *Макроконстанти*

У мові С++ існує можливість задавати константи за допомогою

макропідстановок або макросів, які визначаються за допомогою директиви #define препроцесора, яка виглядає так:

```
#define <ім'я макросу> < текст заміни>
```

Ця директива виконує заміну *ім'я макросу* на *текст заміни* в усьому тексті програми, починаючи з наступного за директивою рядка.

За звичай для імен макросів використовують заголовні літери, щоб відрізняти їх від звичайних імен.

Наприклад:

```
#define PI    3.1415927
#define TWOPI   (2*PI)
```

3.5 Операція розміру sizeof

Операція sizeof повертає обсяг пам'яті, яку займає або потребує операнд цієї операції, тобто розмір операнда. В ролі операнда може бути змінна, вираз або назва типу. Розмір визначається в байтах. Байт – це основна одиниця виміру обсягу пам'яті в інформатиці. Один байт дорівнює 8 біт. Біт це найменша одиниця виміру обсягу інформації, що відповідає одному розряду двійкового коду і може приймати значення нуль або один.

Операція sizeof має декілька стандартних форм:

```
sizeof (тип)
sizeof  імя змінної
sizeof  (вираз)
```

Розмір операндів різних типів у мові С не є визначеним і залежить від системи програмування та типу комп'ютера.

3.6 Бібліотека математичних функції

Бібліотека математичних функцій cmath надає програмісту можливість використовувати різні функції для обробки даних.

Звернення до цих функцій використовуються у виразах і розглядаються як операнди. Для виклику функції необхідно записати її ім'я і далі в дужках, через кому, перерахувати необхідні параметри. Наприклад, щоб обчислити квадратний корінь із змінної n, достатньо записати sqrt (n).

При роботі з функціями слід враховувати тип значення, що повертається і типи параметрів. В одних випадках виконується автоматичне приведення типів даних, в інших виникає невідповідність і компілятор виводить повідомлення про помилку.

У таблиці 3.3 наведено деякі функції бібліотеки cmath.

Таблиця 3.3 Функції бібліотеки cmath

Приклад звернення до функції	Що обчислює функція
fabs(вираз)	Модуль числа
sqrt(вираз)	Квадратний корінь
pow(вираз, ступінь)	Піднесення до ступеню
exp(вираз для степеню числа e)	Експонента
log(вираз)	Натуральний логарифм
log10(вираз)	Логарифм за основою 10
sin(вираз)	Синус
cos(вираз)	Косинус
tan(вираз)	Тангенс
asin(вираз)	Арксинус
acos(вираз)	Арккосинус
atan(вираз)	Арктангенс
sinh(вираз)	Гіперболічний синус
cosh(вираз)	Гіперболічний косинус
tanh(вираз)	Гіперболічний тангенс
celi(вираз)	Округлення до найближчого більшого
floor(вираз)	Округлення до найближчого меншого
fmod(вираз ділене, вираз дільник)	Залишок від ділення з плаваючою точкою

4 ВИРАЗИ

4.1 Арифметичні та порозрядні операції

4.1.1 Арифметичні операції

Перелік арифметичних операцій C++ наведено у таблиці 4.1. Усі наведені операції, окрім операції %, виконуються над даними усіх числових типів. Операція % має сенс тільки для цілих чисел.

Таблиця 4.1- Арифметичні операції C++

Операція	Опис
*	Множення
/	Ділення
%	Остача від цілочислового ділення
+	Додавання
-	Віднімання

Слід також мати на увазі, що операція ділення для цілих чисел повертає також ціле число. Для того, щоб результат ділення був дійсним числом треба, щоб хоча б один операнд був також дійсним числом. Наприклад, 12 / 5 буде 2, але 12.0 / 5 буде 2.4.

4.2 Операції порівняння та логічні операції

Операції порівняння наведені в таблиці 4.2.

Таблиця 4.2 – Операції порівняння

Операція	Опис
==	Дорівнює
>	Більше

<	Менше
>=	Більше або дорівнює
<=	Менше або дорівнює
!=	Не дорівнює

4.3 Логічні операції

Логічні операції наведено в таблиці 4.3

Таблиця 4.3 – Логічні операції

Операція	Опис
||	Або
&&	і
!	Ні
(a||b) && !(a&&b)	Виключно або (xor)
b>=4 && b<5	Дужки не потрібні

4.4 Порозрядні логічні операції

Порозрядні логічні операції наведено в таблиці 4.4

Таблиця 4.4 – Порозрядні логічні операції

Операція	Опис
|	Або
&	і
!	Ні
^	Виключно або (xor)

4.5 Операції присвоєння, комбіновані присвоєння

4.5.1 Проста операція присвоєння

Операція присвоєння є однією із основних операцій у будь якій мові програмування. У мові С++ вона виглядає так:

```
<змінна> = <вираз>;
```

де: *<змінна>* - ім'я змінної, значення якої змінюється в результаті виконання інструкції присвоювання;

= знак операції присвоювання.

<вираз> – вираз, значення якого присвоюється змінній, ім'я якої вказане ліворуч від символу присвоювання.

Присвоювання виконується наступним чином:

– спочатку обчислюється значення виразу, який знаходиться праворуч від знаку операції присвоювання;

– потім отримане значення записується у змінну, ім'я якої стоїть ліворуч від символу присвоювання.

Операція присвоювання вважається вірною, якщо тип значення, що повертає вираз, відповідає або може бути приведений до типу змінної, яка отримує це значення. Наприклад, змінній типу double можна присвоїти значення виразу, тип якого float або int.

Особливість операції присвоювання у мові С полягає у тому, що ця операція **повертає результат**, що дорівнює значенню виразу у правій частині. Тому цю операцію можна використовувати у виразах. Слід тільки мати на увазі, що ця операція має низький пріоритет. Тому присвоєння у виразах слід брати в дужки. Наприклад, вираз a + (b = c+d) є допустимим у мові С.

4.5.2 Комбіновані присвоєння

Окрім простого оператора присвоєння у мові С завжди були ще і так звані комбіновані оператори присвоєння. Згодом вони з’явилися і у інших мовах.

Ці оператори використовують у тих випадках, коли ліворуч і праворуч від

знака операції присвоєння знаходиться той самий операнд.

Наприклад, замість присвоєння number = number + d можна записати number += d.

Таке присвоєння можна використовувати у комбінації з будь якою арифметичною операцією та багатьма іншими, наприклад, *=, /=, %=, тощо.

Комбіновані присвоєння скорочують запис і вважається, що вони роблять запис більш наочним і скорочують процес обчислення результату. Стиль написання програм на мові С схвалює використання таких присвоєнь.

4.5.3 Унарні присвоєння

Ці присвоєння є окремим випадком комбінованого присвоєння. Вони використовуються тоді, коли до змінної треба додати одиницю або відняти її. Відповідно до цього маємо операції інкременту та декременту.

Запис операції унарного присвоєння ще простіший ніж комбінованого. Для того щоб, наприклад, збільшити змінну number на одиницю, можна написати number++, або ++ number. Так само можна і зменшувати значення на одиницю: number--, або --number.

Операції ++ та -- називають префіксними, якщо знаки цих операцій записуються перед змінною. Якщо ж знаки операції записуються після змінної то операції називають постфіксними.

Різниця між префіксними та постфіксними операціями проявляється у тих випадках, коли ці операції використовуються у виразах разом з іншими операціями. Справа у тому, що префіксні операції мають найвищий пріоритет, а постфіксні – найнижчий. Хай, наприклад, змінна x має значення 5. Тоді значення виразу (3+ ++x) буде дорівнювати 9, а значення виразу (3+x++) буде дорівнювати 8, хоча «x» у обох випадках отримає значення 6.

4.6 Умовна операція (теренарний оператор)

Назва походить від слова три. До складу оператора входить три вирази та два символи (?:).

Синтаксис такий:

Вираз_1 ? Вираз_2 : Вираз_3;

Наприклад:

stip = ball<4.0 ? 0 : ball<5 ? 600 : 800;

4.7 Порядок виконання операцій

Порядок виконання операцій визначається їх пріоритетами. У таблиці 4.1 наведено пріоритети операцій С++. Чим менший номер пріоритету, тим вищий пріоритет. Деякі операції, можливо, поки що незнайомі, та в подальшому вони будуть розглянуті.

Таблиця 4.1 – Пріоритети операцій С++

Операції	Пріоритет
Префіксні ++ -- ( ) [ ] . ->	1
! ~ +a -a (type) sizeof *a &a	2
* / % (арифметичні)	3
+ - (арифметичні)	4
<< >> (зсуви ліворуч і праворуч)	5
< > <= >= (порівняння)	6
= = != (порівняння)	7
& ^ | (порозрядні логічні операції)	8,9,10
&& || (логічні операції)	11,12
? : (теренарна операція)	13
= += -= *= /= %= &= ^= |= >>= (присвоєння)	14
Постфіксні ++ -- , (операція кома)	15

4.8 Узгодження типів операндів у виразах

У мові С++ є допустимою ситуація, коли операнди операції мають різний тип. У цьому випадку компілятор перевіряє сумісність операндів і встановлює

для них спільний тип. Цей же тип і буде типом результату операції. Таке перетворення виконується компілятором автоматично, тому і перетворення звуться автоматичними.

Є дві схеми автоматичного узгодження типів: арифметичні перетворення і перетворення під час присвоєнь.

4.8.1 Арифметичні перетворення

Ці перетворення виконуються у арифметичних та порозрядних операціях та операціях порівняння.

Під час цих перетворень перш за все виконується так званий integral promotion: типи bool, char і short перетворюються у int .

bool, char, short → int

Зокрема, перетворення bool → int перетворює true в 1, a false у 0.

Далі використовується принцип ”найбільшого операнду”, відповідно до ряду:

long double, double, float, usigned long, long, unsigned int, int

Тобто якщо, наприклад, у виразі є операнди типу float, int та double то усі операнди будуть приведені до типу double.

4.8.2 Перетворення типів в операціях присвоєння

У результаті операції присвоєння результат отриманий у правій частині приводиться до типу лівої частини. При цьому:

– якщо тип змінної ліворуч знаку операції присвоєння вище типу результату обчислення правої частини, то результат обчислення перетворюється за принципом ”найбільшого операнду”;

– якщо тип змінної ліворуч знаку операції присвоєння нижче типу результату обчислення правої частини, то результат обчислення обтинається. Це полягає у тому, що може бути зменшено число розрядів мантиси (double->float), або відкинуто дробову частину (float->int). Але внаслідок таких перетворень можливо і спотворення результату. Наприклад, перетворення long -> int може призвести до втрати старших біт результату.

4.8.3 Явне перетворення типів

Мова C++ дозволяє і явне перетворення даних одного типу у інший. Для цього можна використовувати такі конструкції:

(*тип*) *вираз*

тип (*вираз*)

Останній варіант з’явився у C++ і схожий на виклик функції. Але на відміну від виклику функції тут перед круглими дужками рекомендують ставити пробіл.

5 ФУНКЦІЇ

5.1 Структура та правила написання функцій

Всі функції мови С мають однакову структуру, що виглядає таким чином:

```
тип_значення_що_повертається ім'я_функції(список_формальних_параметрів)
{
   тіло_функції
}
```

Початкова частина функції, яка складається з типу значення, що повертається, імені функції та списку оголошень формальних параметрів, записаного у круглих дужках, називається заголовком функції або сигнатурою.

Тип значення, що повертається, це будь який можливий у мові С тип. Якщо функція нічого не повертає, записується тип void.

Ім'я функції має задовольняти вимогам мови С до написання ідентифікаторів. Після імені функції обов'язково мають бути круглі дужки, незалежно від наявності формальних параметрів.

Список параметрів складається із оголошень формальних параметрів, відокремлених комами. Оголошуються параметри так само, як і звичайні змінні (тип та ім'я), але кожен з параметрів оголошується окремо, навіть, якщо типи сусідніх параметрів співпадають.

Тіло функції, що знаходиться у фігурних дужках, складається з описів внутрішніх змінних та операторів, що реалізують дії, які повинна виконувати функція. Змінні , що оголошені у тілі функції, називають локальними, тому що вони доступні тільки у межах своєї функції. Вони мають бути оголошені до першого звертання.

Повернення результату роботи функції реалізується через оператор return, який має такий синтаксис:

```
   return вираз ;
```

Вираз, що розташований після слова return обчислюється, і його значення є результатом, що повертає функція. Оператор return завершує роботу функції, незалежно від того де він розташований у тілі функції. Якщо алгоритм виконання функції передбачає наявність декількох варіантів завершення, то тіло функції може включати і декілька операторів return.

Якщо функція нічого не повертає, оператор return записується без виразу, або взагалі не використовується.

Нижче, як приклад, наведено опис функції, що повертає суму двох чисел:

```
float  sumTwoNumber (float a,  float b)
{float z;
   z=a+b;
   return z;
}
```

У тілі функції визначено локальну змінну z, яка за межами функції недоступна. Ця змінна приймає результат додавання змінних a та b. Отримане значення функція повертає через оператор return.

Ту саму функцію можна записати і коротше:

```
float  sumTwoNumber (float a,  float b) { return a+b;}
```

5.2 Виклик функції

Оператори функції виконуються тільки тоді, коли здійснюється звертання до даної функції. Таке звертання називають викликом функції. Виклик функції виглядає так:

ім'я_функції(*список_фактичних_параметрів*)

Список фактичних параметрів являє собою послідовність виразів, які задають значення формальних параметрів, що перелічені в описі функції. Фактичні параметри обов'язково мають відповідати формальним за кількістю, порядком розташування та типом. **Але не за назвами!**

Виклик функції, що повертає якесь значення, можна використовувати як вираз, а також в якості операнда виразу. Значенням такого операнда буде значення, що повертає функція.

Приклад звернень до функції, опис якої наведено у попередньому пункті, представлено нижче:

У цьому прикладі перше звернення до нашої функції відбувається під час визначення змінної y. Фактичними параметрами у цьому випадку є константи 4.2 та 5.4, а результатом число 9.6. Друге звернення до нашої функції відбувається у виразі, що передається об'єкту cout. Фактичними параметрами у цьому випадку є змінна x та вираз y/2, а результатом буде число 5.5.

```
…
float  x=2.3;
float  y=sumTwoNumber(4.2, 5.4);
cout << 3+sumToNumber(x, y/3);
```

5.3 Прототип функції

Для того, щоб компілятор мав можливість перевірити правильність звертання до функції та використання її результату, опис функції має бути наведено раніше, ніж звертання до неї. Але розташувати таким чином усі функції програми не завжди можливо, а окрім того, і недоречно. Найкраще першою розташовувати функцію main(), далі записувати функції, які послідовно розкривають алгоритм вирішення задачі, а у кінці наводити функції, що деталізують окремі кроки алгоритму.

Для того, щоб у програмах на мові С можна було використовувати довільний порядок розташування функцій використовують прототипи функцій. Прототип функції співпадає із її заголовком і не має тіла, тобто має таку форму:

тип_значення_ що_повертається ім'я_функції(*список_формальних_параметрів*);

Прототип містить усю інформацію, що потрібна компілятору для перевірки правильності застосування функції, і таким чином може замінити для

компілятора опис функції, якщо цей прототип розташувати перед викликом функції. Найчастіше прототипи функцій записують перед функцією main(), що є початком програми.

Слід зазначити, що у прототипі можна навіть не наводити імен формальних параметрів, достатньо лише вказати їх типи.

Так, прототип функції, що розглядалася вище як приклад виглядає так:

```
float  sumTwoNumber (float a,  float b);
```

Той же прототип може виглядати і так:

```
float  sumTwoNumber (float,  float);
```

5.4 Способи передачі параметрів до функцій

Існує два способи передачі параметрів у функції - передача за значенням (by value) і передача через посилання (by reference). Спосіб передачі вказується при оголошенні параметра у списку формальних параметрів.

За замовчуванням передбачається, що параметри звичайних типів, наприклад, float, double, int, char передаються за значенням, а параметри таких типів як масиви передаються через посилання. Якщо виникає необхідність вказати, що параметр передається через посилання, то перед ім'ям параметра, пишеться символ &.

5.4.1 Передача параметрів за значенням

Передача параметрів за значенням передбачає, що під час виклику функції у пам'яті буде виділена спеціальна область для запису копій значень фактичних параметрів, з якими і буде працювати функція.

Такий спосіб передачі захищає змінні, передані у функцію в якості параметрів, від непередбачуваних змін, оскільки функція працює з копіями. Крім того, такий спосіб дозволяє у якості фактичних параметрів задавати вирази. При передачі параметрів буде обчислено значення виразу і передано у функцію.

Недолік такого способу передачі полягає у тому, що для параметрів, які

займають багато пам'яті, наприклад, великі масиви чисел або довгі рядки символів, копії займають багато місця у пам'яті і потребують багато часу для пересилання даних з одного місця пам'яті у інше.

Вище, у прикладі з пункту 3.1.1, параметри до функції передаються по значенню.

5.4.2 Передача параметрів через посилання

У разі передачі параметрів через посилання до функції передаються адреси фактичних параметрів. Тому такий спосіб передачі називається ще передачею параметрів за адресою.

При такому способі передачі в якості фактичних параметрів можуть бути тільки змінні. Вираз і навіть окреме число або символ передати через посилання неможливо.

Передача параметрів через посилання заощаджує пам'ять і скорочує час звернення до функції. Однак це має і побічний ефект. Адже функція працює безпосередньо з фактичними параметрами, і будь-яка зміна формального параметру є зміною фактичного параметру. Для запобігання такому ефекту використовують кваліфікатор const.

Але побічний ефект має і позитивну сторону. Передачу параметрів через посилання можна використовувати для повернення результатів роботи функції через фактичні параметри. Такий спосіб повернення особливо ефективний, коли потрібно повернути декілька параметрів. Адже функція повертає тільки одне значення.

Розглянемо приклад використання передачі параметрів через посилання для повернення результатів роботи функції.

У наведеній нижче функції формальні параметри обмінюються значеннями і фактичні параметри теж обміняються значеннями.

```
void swap(float & a., float & b)
{
      float  temp  = a;  a = b; b = temp;
}
```

5.5 Рекурсивні функції

Функцію називають рекурсивною, якщо вона звертається сама до себе.

Необхідною умовою працездатності функції є наявність виходу з неї без звертання до себе.

Як приклад можна навести функцію обчислення числа Фібоначі за заданим номером. Число Фібоначі обчислюється як сума двох попередніх, а перші два числа дорівнюють 0 та 1.

```
int fibo(uint n){
   if (n==0 || n==1) return n;
   return fibo(n-1) + fibo(n-2);
}
```

5.6 Області оголошення та доступу до імен

Область оголошення – це частина програми, в якій можуть здійснюватися оголошення імен змінних, функцій, тощо. Область доступу до імені – це частина програми, у межах якої це ім’я доступно програмісту.

Доступ до імен простягається від точки, в якій це ім’я оголошено, до кінця області оголошення. Незважаючи на те, що оголошення змінних можна розміщувати будь де у межах області оголошення, краще оголошення згрупувати і розмістити на початку області оголошення.

5.6.1 Глобальні та локальні змінні

Найширшою областю оголошення є файл, який містить нашу програму. Змінні та константи, що оголошені у файлі, але за межами будь-якої функції називаються глобальними. Глобальні змінні автоматично ініціалізуються нульовим значенням.

Більш вузькою областю оголошення є функція. Імена, що оголошені у функції доступні тільки у межах цієї функції. Змінні, що оголошені у функції називають локальними. Глобальні змінні теж доступні у функціях. Але якщо ім’я

локальної змінної у функції співпадає з ім’ям глобальної змінної, то перевага віддається локальній змінній. Тобто локальна змінна перекриває глобальну у межах своєї області доступу.

Найвужчою областю оголошення є блок коду між фігурними дужками. Змінні, що оголошені у блоках теж вважаються локальними.

Локальні змінні автоматично не ініціалізуються. Тому якщо локальна змінна оголошується без ініціалізації (тобто їй не надано початкового значення), то значення цієї змінної може бути будь яким, так зване «сміття».

Глобальні змінні існують поки виконується програма. Пам'ять їм виділяється компілятором і тому їх ще називають статичними.

Локальні змінні створюються тільки після виклику функції та існують тільки до завершення роботи функції, або блоку. Пам'ять для цих змінних автоматично виділяється, а потім звільнюється у процесі виконання програми. Тому такі змінні ще називають автоматичними.

5.6.2 Глобальна чи локальна змінна?

На перший погляд глобальні змінні здаються більш привабливими, оскільки до них мають доступ усі функції. Однак такий легкий доступ до змінних знижує надійність програми.

У більшості випадків слід користуватися локальними змінними і передавати їх іншим функціям тільки через параметри і тільки в разі необхідності.

Однак іноді глобальні змінні доцільно використовувати, наприклад, для зберігання постійних даних, що використовуються декількома функціями. Для запобігання небажаним змінам таких даних можна скористатися ключовим словом const.

5.6.3 Специфікатор statsc

Мова С дозволяє, у разі потреби, зберегти значення локальної змінної, яка була обчислена у функції. Це може бути потрібно у тих випадках, коли функція використовує значення, що були обчислені під час попереднього виклику цієї

функції. Саме так, наприклад, обчислюються випадкові числа. Кожне наступне підраховується на підставі попереднього. Нижче наведено функцію, що формує цілі випадкові числа за формулою x=a*x+b. На перший погляд тут складається враження, що ніякої випадковості нема, але слід враховувати, що константа «a» дуже велика. Результат множення змінної «x» на таке велике число майже завжди буде перевищувати максимально можливе значення для типу unsigned long, внаслідок чого старші біти результату будуть втрачатися. Таким чином результат стає ніби то випадковим.

```
#include <ctime>
#include <iostream>
using namespace std;
// Функція генерації випадкових чисел
unsigned long amkm() {
    unsigned long a = 0xBL, b = 0x5DEECE66DL;
    static unsigned long x = time(NULL);
    return x=a + b * x;
}
// Тестування функції amkm
int main() {
    m:    cout << amkm() << "\n";
          goto m;
          return (0);
}
```

Константи «a» та «b» задано у 16-річному форматі.

Початкове значення змінної «x» дорівнює значенню поточного часу, що повертає функція time(). Потім воно змінюється виразом x=a*x+b. Отримане значення зберігається до наступного виклику функції завдяки тому що змінна «x» оголошена із специфікатором **statsc**.

У функції main() функція amkm() безперервно викликається завдяки використанню мітки «m:» та оператора переходу goto.

5.7 Макроси з параметрами

Макроси з параметрами дають змогу здійснювати макропідстановки, подібні до викликів функцій. Формальні параметри, вказані в списку #define, під час препроцесування замінюються фактичними виразами, заданими у звертанні до макросу. Використання макросів замість функцій має дві переваги. По-перше, макрос вбудовується в тіло програми на етапі препроцесування, отже під час виконання програми не витрачається час на виклик функції. По-друге, аргументи, що вказуються у звертанні до макросів, можуть мати довільний тип.

Синтаксис макросу з параметрами такий:

```
#define ім'я_макросу(список_ параметрів) вираз_для_заміни
```

У виразі для заміни кожен з параметрів і весь вираз слід брати у дужки.

Як приклад, наведемо макрос для піднесення числа a до ступеня b:

```
#define POW(a,b)  ( exp( (b) * log(a) ) )
```

Звернення до цього макросу у програмі може виглядати так:

```
cout <<  POW( 3 + x , 4.5 ) << "\n"
```

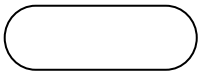
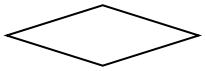
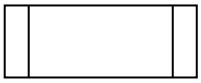
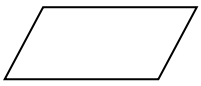
6 ОПЕРАТОРИ МОВИ С

6.1 Поняття про схеми алгоритмів та їх різновиди

При розробці алгоритмів корисно зображувати схеми алгоритмів, використовуючи спеціальні позначення. Схема алгоритму являє собою як би план написання програми. Складання таких схем не вимагає багато часу, але істотно підвищує продуктивність праці програміста.

Деякі умовні позначення, що використовуються при складанні схем алгоритмів, наведені в таблиці 6.1. При зображенні алгоритму окремі блоки нумеруються і з'єднуються лініями. Стрілки використовуються тільки для зворотних напрямків (вгору і ліворуч).

Таблиця 6.1 - Умовні позначення для схем алгоритмів

	Начало и кінець алгоритму
	Обробка інформації, наприклад, розрахунок за формулою
	Перевірка умови і прийняття рішення. Після цього блоку можливі різні шляхи продовження виконання алгоритму
	Зумовлений процес, наприклад, звернення до функції.
	Виведення або введення інформації.

В якості прикладу розглянемо схеми алгоритму розв’язання квадратного рівняння $ax^2 + bx + c = 0$.

На рисунку 6.1 зображено схему алгоритму, де аналізується перший

коефіцієнт рівняння і приймається рішення, чи є рівняння квадратним, або воно лінійне. У першому випадку буде викликана процедура вирішення квадратного рівняння, у другому - процедура рішення лінійного рівняння.

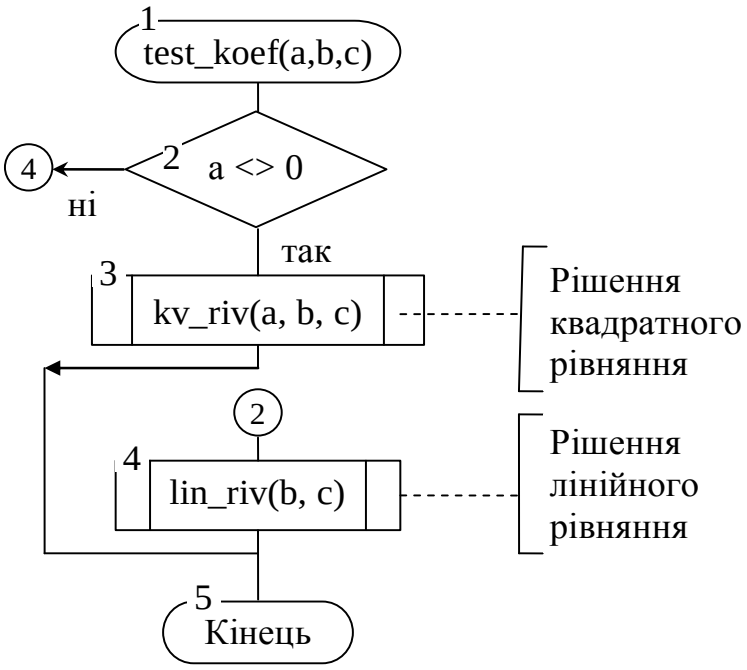


Рисунок 6.1- Схема алгоритму аналізу коефіцієнтів квадратного рівняння

6.2 Оператори-вирази: присвоєння, виклик функції, порожній оператор

Операція простого присвоєння використовується для заміни значення лівого операнда, значенням правого операнда. При присвоюванні відбувається перетворення типу правого операнда до типу лівого операнда.

Приклад.Операція простого присвоєння

```
int t;
char f ='a';
long z = 10;
t = f + z; /*t = 107*/
```

Значення змінної f перетворюється до типу long, обчислюється вираз f+z, результат перетворюється до типу int і потім присвоюється змінній t.

6.3 Умовні оператори: if, switch

Умовні оператори if та switch використовуються для програмування алгоритмів з розгалуженнями. У повсякденному житті ми часто зустрічаємося з такими видами алгоритмів. Наприклад, якщо йде дощ, ми беремо парасольку, а якщо мороз, надягаємо теплу куртку. Таким чином, у алгоритмі, що має розгалуження, деякі дії можуть виконуватися тільки за певних умов.

6.3.1 Оператор розгалуження if…else

Оператор if ... else дозволяє вибрати один з двох можливих варіантів виконання програми. Синтаксис запису цього оператора представлений на рисунку 6.2.

```
if ( <умова> )
        <оператор 1>
else    <оператор 2>
```

Рисунок 6.2– Синтаксис оператора **if … else**

Виконується цей оператор наступним чином:

– обчислюється значення умови (умова - вираз, результат якого приводиться до логічного типу);

– якщо результат приймає значення “true”, то виконується <оператор 1>;

– якщо результат приймає значення “false”, то виконується <оператор 2>, що розташований за словом else.

Як приклад використання цього оператора можна привести програмну реалізацію схеми алгоритму зображеного на рисунку 6.1.

```
void test_koef(float a, float b, float c) {
  if (a==0)
     // Звернення до функції рішення лінійного
     lin_riv(b,c);
  else
```

```
    // Звернення до функції рішення квадратного рівняння
    kv_riv(a,b,c);
}
```

Якщо замість одного оператора потрібно виконати декілька, їх слід об’єднати у блок за допомогою фігурних дужок. У цьому випадку крапка з комою після закриваючої дужки не ставиться.

У деяких випадках, при невиконанні умови, що перевіряється, робити нічого не треба, тобто <оператор 2> не потрібен. У цьому випадку оператор if можна застосовувати в скороченій формі. Синтаксис скороченої форми має вигляд, представлений на рисунку 6.3.

```
if (<умова>)
        <оператор>
```

Рисунок 6.3 – Скорочена форма оператора **if**

Оператор **if**, як повний так і скорочений, може бути вкладений у інший оператор **if**. При цьому слід пам’ятати, що кожна else-частина пов’язується з найближчим **if**. Якщо ж else-частина відноситься до одного з попередніх **if**, то слід застосовувати фігурні дужки. Приклад використання вкладених операторів **if** наведено на рисунку 6.4.

Стандарт мови С гарантує можливість 15 рівнів вкладення для оператора **if**. Проте не слід цим зловживати. Вкладені оператори **if** виглядають зазвичай досить заплутано і часто є причиною виникнення логічних помилок.

```
if  <умова1>
{
        < оператор 1.1>;
        if  <умова2>
        {
                <оператор 2.1>;
        }
        else // для умови 2
        {
                <оператор 2.2>;
        }
        < оператор 1.2>;
}
else // для умови 1
}
        < оператор 1.3>;
}
```

Рисунок 6.4– Приклад використання вкладених операторів **if**

Більш зручними і наочними є ланцюжки повних операторів **if**, в яких після кожного слова **else** (за винятком останнього) знову йде повний оператор **if,** рисунок 6.5. Такі ланцюжки зручно використовувати при вирішенні багатоваріантних задач. Вони досить легко аналізуються, за структурою подібні до розглянутого нижче оператору **swith**, і їх можна застосовувати, на відміну від нього, для перевірки будь-яких умов.

```
if  <умова1>
        <оператор 1>
else if  <умова2>
        <оператор 2>
else if  <умова3>
        <оператор 3>
…
else <оператор N>
```

Рисунок 6.5 – Приклад ланцюжка повних операторів **if**

6.3.2 Умовна операція

Цю операцію називають ще теренарною операцією. Вона виконується над трьома операндами і записується так, як показано на рисунку 6.6.

<умова> ? < вираз 1 > : < вираз 2 >

Рисунок 6.6– Синтаксис теренарного оператора

Тут умова - це вираз, результат якого приводиться до логічного типу. Якщо результат обчислення умови приймає значення “true”, то результатом операції буде значення виразу 1. Якщо результат обчислення умови приймає значення “false”, то результатом операції буде значення виразу 2. Нижче наведено приклад використання вкладеного теренарного оператора у функції для розрахунку стипендії.

```
float stip(bool budget, float ball){
    return !budget || ball<4 ? 0 : ball<5 ? 600 : 800;
}
```

.

6.3.3 Оператор вибору swith

Конструкція **swith** дозволяє ефективно реалізувати численні розгалуження у тих випадках, коли вибір визначається значеннями змінної порядкового типу (int, char).

Синтаксис оператору **swith** представлено на рисунку 6.7.

```
switch ( <вираз> ) {
        case <константа 1>:
        <оператор 1>
        case < константа 2>:
        <оператор 2>;
…
        case< константа n>:
        <оператор n>;
default:
        <оператор n+1>;
}
```

Рисунок 6.7 – Синтаксис оператора **swith**

У цьому описі < вираз > - це вираз, значення якого визначає подальше виконання оператора. В окремому випадку < вираз > може бути просто змінною. Результат обчислення виразу може бути тільки ціле число або символ.

<константа > – це мітка даного варіанту, яка може бути константою або виразом. Значенням константи або результатом обчислення виразу також має бути ціле число або символ. Порядок запису міток довільний, але вони мають бути різними. Останній варіант, що починається словом default, не є обов'язковим. Мітка відділяється від оператора двокрапкою.

< оператор > визначає дії, які повинні бути виконані, якщо < вираз > приймає значення константи. В якості оператора може використовуватися і складений оператор.

Порядок виконання оператора **swith** описаний нижче:

– спочатку обчислюється значення виразу;

– далі, отримане значення послідовно порівнюється зі значеннями констант;

– якщо значення виразу збігається з однією із міток, то виконується оператор цього варіанту і усі наступні за ним до default, якщо раніше не зустрівся оператор переривання break;

– якщо значення виразу не збігається з жодною із міток, то виконується оператор після слова defaul, якщо ця частина присутня.

В якості прикладу розглянемо функцію, що опрацьовує номер варіанту вибраного з меню і відповідно до цього задає коефіцієнти квадратного рівняння. Ця функція може стати у нагоді під час тестування програми обчислення коренів квадратного рівняння. Внаслідок того, що кожний із варіантів виконується окремо і не пов'язаний з іншими, виникає необхідність кожен варіант завершувати оператором break.

Варіант default у цьому прикладі не використовується.

```
void run_variant(int v){
     switch(v) {
     case 0:// Завершити програму
            exit(0); break;
     case 1:// Задати коефіцієнти користувача
            user_koef(); break;
     case 2:// Дійсні корені
            test_koef(4, 5, -3); break;
     case 3:// Комплексні корені
            test_koef(4, 2, 3); break;
     case 4:// Лінійне рівняння
            test_koef(0, 2, 3); break;
     case 5:// Будь-яке рішення
            test_koef(0, 0, 0); break;
     case 6:// Нема розв'язку
            test_koef(0, 0, 5);
     }
}
```

6.4 Оператори циклу: while, do-while, for

6.4.1 Циклічні алгоритми

Алгоритми вирішення багатьох задач є циклічними, тобто для досягнення результату певна послідовність дій повинна бути виконана декілька разів.

Наприклад, для того щоб знайти прізвище людини у списку, треба перевірити перше прізвище списку, потім друге, третє і т. д. доти, поки не буде знайдено потрібне прізвище або не закінчиться список.

Алгоритм, в якому є послідовність операцій (група операторів), яка повинна бути виконана декілька разів, називається циклічним, а сама послідовність операцій іменується тілом циклу.

У програмах на мові С цикл може бути реалізований за допомогою операторів while, do...while і for. Цикл, який створюється за допомогою оператора for, буде розглянуто в наступній роботі. Поки ж ми розглянемо оператора while і do... while.

Особливість циклів, що створюються за допомогою цих операторів, в тому, що в них заздалегідь не відомо, скільки разів буде виконуватися тіло циклу. Виконання повторюється, поки задовольняється деяка умова. А параметри, що впливають на умову змінюються у тілі циклу.

Типовими прикладами використання таких циклів є обчислення із заданою точністю, пошук у масиві або у файлі.

6.4.2 Оператор while

Особливість ццого оператора полягає у тому, що умова перевіряється перед виконанням тіла циклу, тому цикл while називають циклом з передумовою.

В узагальненому вигляді оператор while записується так (рис.6.8).

```
while( умова виконання )
{
        оператори тіла циклу;
}
```

Рисунок 6.8 – Синтаксис оператора **while**

На цьому рисунку <умова виконання > - це вираз логічного типу, що визначає умову за якої виконуються <оператора тіла циклу >.

Загалом, оператор **while** виконується у такий спосіб:

– обчислюється значення виразу <умова виконання>;

– якщо значення виразу <умова виконання > дорівнює **false** або 0, тобто умова не виконується, виконання <операторів тіла циклу > припиняється;.

– якщо значення виразу <умова виконання дорівнює **true** або не 0 (умова виконується), то виконуються <оператора тіла циклу >, розташовані між фігурними дужками;

– після цього знову все повторюється.

Слід зауважити, що для того щоб цикл завершився, потрібно щоб послідовність операторів, розташованих між фігурними дужками, впливала на значення <умови виконання >.

На рисунку 6.9 представлено схему алгоритму виконання цього циклу.

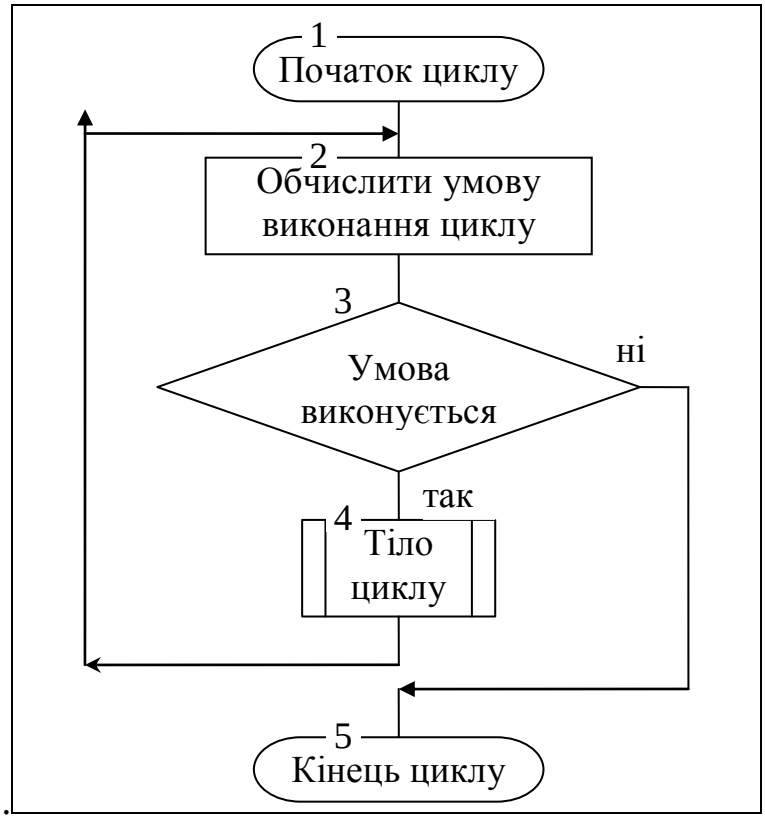


Рисунок 6.9 – Схема алгоритму виконання циклу while

Як приклад розглянемо функцію, що підраховує суму цифр цілого числа. Виділяти окремі цифри числа, починаючи з останньої можна за допомогою операції %10. Далі число можна поділити на 10, що призведе до втрати останньої цифри і передостання цифра стане останньою. Після цього операцію %10 можна повторити, і так далі, доки число після чергового ділення на 10 не стане нулем.

Текст цієї функції наведено нижче:

```
uint sumFig(int x){
    int sum=0;
    while(fabs(x)>0){
        sum+=x%10;
        x/=10;
    }
    return sum;
}
```

У мові С цикл while може виглядати дещо незвично, якщо у виразі для перевірки умови використовувати операцію присвоєння, або операцію «,» (кома). Операція кома пов'язує декілька виразів, що розглядаються як один. Результатом такої послідовності буде результат обчислення останнього виразу.

Використання таких можливостей може призвести до скорочення тіла циклу і навіть до його зникнення, так як це відбулося у наступній функції, яка робить те саме, що й попередня.

```
uint sumFig(int x){
    int sum=0;
    while(sum+=x%10, fabs(x/=10)>0);
    return sum;
}
```

6.4.3 Оператор do…while

Особливість цього оператора полягає у тому, що умова перевіряється після виконання тіла циклу, внаслідок чого <оператори тіла циклу > виконуються, принаймні, один раз. Тому цикл **do…while** називають циклом з постумовою.

У мові програмування С оператор **do…while** виглядає таким чином (рисунок 6.10):

```
do {
        оператори тіла циклу
} while (умова повтору тіла циклу);
```

Рисунок 6.10 – Синтаксис оператора **do…while**

На рисунку <умова повтору тіла циклу > - це вираз логічного типу, що визначає умову за якої будуть знов виконані <оператора тіла циклу >.

Цикл виконується таким чином:

– спочатку виконуються <оператори тіла циклу >, розташовані між фігурними дужками;.

– потім обчислюється значення виразу <умова повтору тіла циклу >.

– якщо значення виразу <умова повтору тіла циклу > дорівнює **true** або не 0 (умова виконується), то знову виконуються <оператори тіла циклу >, розташовані між фігурними дужками;

– якщо значення виразу <умова виконання > дорівнює **false** або 0, тобто умова не виконується, виконання <операторів тіла циклу > припиняється.

Схема алгоритму виконання цього циклу представлена на рисунку 6.11.

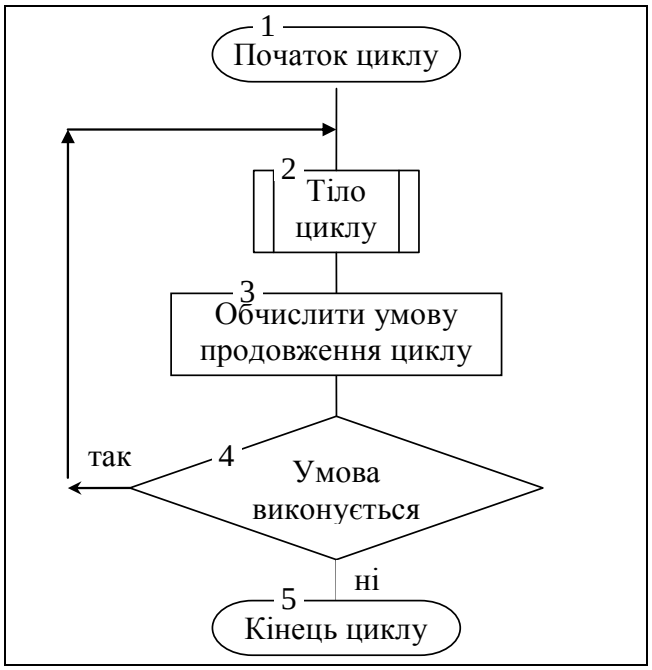


Рисунок 6.11 – Схема алгоритму виконання циклу **do…while**

Як приклад використання циклу **do…while** розглянемо програму, вхід до

якої контролюється паролем. Цикл **do…while** тут буде доречним, бо спочатку потрібно ввести пароль, а потім перевірити його правильність. Якщо пароль не правильний, то введення паролю слід повторити.

Текст цієї програми наведено нижче.

```
string p="qwerty";
string inpt;
do{
   cout<<"/nВведіть пароль ";
   cin>>inpt;
}while(p!=inpt);
cout<<"OK!\n";
 …
```

6.4.4 Оператор циклу for

Оператор **for** у мові С - це дуже потужний інструмент для організації циклів. За його допомогою можна запрограмувати більшість циклічних процесів, але він не такий прозорий, як оператор **while.**

У загальному вигляді оператор **for** записується таким чином:

for (<вираз1>; <вираз2>; <вираз3>)
<тіло циклу>*;*

Рисунок 6.12 – Синтаксис оператора **for**

На цьому рисунку < вираз1> - це вираз ініціалізації, який встановлює початкові значення змінних циклу; <вираз2> - це вираз умови, що задає умову виконання тіла циклу; <вираз3> - це вираз ітерації, який виконує зміну значень змінних циклу; <тіло циклу> - це оператор або блок, який задає дії, що мають повторюватися.

Оператор циклу **for** виконується наступним чином:

1) Обчислюється < вираз1>, внаслідок чого змінні циклу приймають початкові значення. Цей вираз обчислюється тільки один раз на початку роботи циклу.

2) Обчислюється <вираз2>, що є умовою продовження виконання циклу.
3) Якщо умова хибна, то цикл закінчується.
4) Виконується тіло циклу.
5) Обчислюється < вираз3>, який використовують для зміни параметрів циклу.
6) Виконання продовжується з пункту 2.

Параметри циклу можна оголошувати як у виразі1, так і за межами циклу. Але у першому випадку параметри циклу будуть доступні тільки в межах циклу

Як приклад використання циклу **for** розглянемо функцію обчислення числа Фібоначчі із заданим номером. Це послідовність чисел у якій нульове число дорівнює 0 і перше дорівнює 1. Решта чисел обчислюється як сума двох попередніх. Тобто послідовність має такий вигляд 0 1 1 2 3 5 8 13 21 34 …

У цій програмі у тілі циклу послідовно обчислюються числа Фібоначчі починаючи з другого і до заданого, з номером n.

```
uint fibo(uint n){
   if(n==0 || n==1) return n;
   uint  f, f0=0, f1=1; //Початкові значення двох попередніх чисел
   for(uint i = 2; i<=n; i++){
       f=f0+f1; //Обчислюємо число Фібоначчі
       f0=f1;   f1=f;  // Змінюємо значення двох попередніх чисел
    }
   return f;
}
```

У наступному прикладі цикл for використовується для підрахунку факторіала числа n.

```
unsigned long long fact(uint n) {
   unsigned long long f=1;
   for( uint i = 1; i<=n; i++ )
```

```
    f*=i;
  return f;
}
```

6.4.5 Особливості використання циклу for

У попередньому пункті були розглянуті приклади використання циклу **for** де використовувався усього один параметр циклу, усі три вирази і тіло циклу. Але можливі і інші варіанти використання цього циклу.

Так, наявність кожного з трьох виразів заголовку циклу не обов'язкова. Може навіть не бути жодного з них. Таким чином отримуємо безкінечний цикл. Наприклад, розглянуту вище функцію обчислення факторіалу можна записати і таким чином:

```
unsigned long long fact(uint n) {
  unsigned long long f=1;
  uint i = 1;
  for( ; ; ){
    f*=i;
    if(i>=n) return f;
    i++;
  }
}
```

Не важко здогадатися, що коли цикл має тільки вираз перевірки умови, то він буде еквівалентом циклу while.

У програмі підрахунку суми цифр цілого числа, що була розглянута у попередній лабораторній роботі, цикл while можна замінити циклом for і програма буде виглядати так:

```
  cout<<"/nВведіть ціле число\n ";
  int x;
  cin>>x;
  int sum=0;
  for( ; x!=0; ){
```

```
    sum+=x%10;
    x/=10;
  }
  cout<<"/nСума цифр у числі дорівнює "<<sum<<"\n ";
```

Ще одна особливість циклу **for** пов'язана з операцією кома «,». Ця операція пов'язує декілька виразів, що розглядаються як один. Результатом такої послідовності буде результат обчислення останнього виразу.

Можливість використання операції кома дозволяє використовувати декілька параметрів циклу, а використання цієї операції у ітераційному виразі може призвести до зникнення тіла циклу.

Ось як, наприклад може виглядати функція обчислення факторіалу:

```
unsigned long long fact1(uint n) {
   unsigned long long f=1;
   for( uint i = 1; i<=n; f*=i, i++ );
   return f;
}
```

А так може виглядати функція обчислення числа Фібоначчі:

```
int fibo1(unsigned int n){
   if(n==0 || n==1) return n;
   unsigned int f=0;
   for(int i=2,f0=0,f1=1; i<=n; f=f1+f0, f0=f1, f1=f, i++);
   return f;
}
```

6.5 Оператори goto, break, continue, return

6.5.1 Оператор переходу goto

Цей оператор забезпечує перехід до іншого оператора, що позначений заданою міткою. Мітка, до якої відбувається перехід може бути розташована як

до так і після оператора **goto**. Синтаксис оператора **goto** наведено на рисунку 6.13.

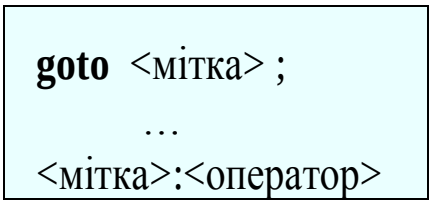


Рисунок 6.13– Синтаксис оператора **goto**

Оператор **goto** використовується рідко, тому що він заплутує програму і робить її малозрозумілою.

6.5.2 Переривання циклу

Під час програмування деяких задач виникає потреба перервати виконання циклу, не чекаючи виконання умов виходу з циклу. Така можливість забезпечується оператором **break**. Цей оператор може бути розташований у будь-якому місці тіла циклу. В результаті цикл відразу ж припиняється.

На відміну від **break** оператор **continue** перериває виконання тільки тіла циклу і одразу ж переходить до обчислення умови продовження циклу.

Крім цих операторів можна використовувати оператор **return**, який перериває виконання не тільки циклу, але і всієї функції.

Можна використовувати і оператор **exit**, але в цьому випадку завершиться робота головної програми.

7 ОДНОВИМІРНІ МАСИВИ

7.1 Оголошення та ініціалізація масивів

Масив являє собою сукупність даних, що організована певним чином, тобто структуру даних. Основні особливості структури даних, що зветься масивом полягають у наступному:

– масив складається з елементів, які мають однаковий тип;

– елементи масиву послідовно розташовані в одній ділянці оперативної пам’яті без проміжків між елементами;

– кожен з елементів масиву має свій порядковий номер, що зветься індексом;

– нумерація елементів починається з 0;

– до елементів масиву можна звертатися використовуючи ім’я масиву і індекс;.

– масив може бути одновимірним, або багатовимірним, тобто таким у якого кожний елемент є також масивом;

– у мові С, С++ ім’я масиву зберігає адресу першого елементу цього масиву, тобто є вказівником на його початок.

Оголошення одновимірного масиву має вигляд, представлений на рисунку 7.1.

<тип елементів> <ім’я масиву> [<кількість елементів>] ;

Рисунок 7.1 – Синтаксис оголошення одновимірного масиву

Тип елементів визначає тип елементів, з яких складається масив. Це може бути будь який допустимий у мові тип, простий або складений

Ім’я масиву – це ідентифікатор написаний за правилами запису імен у мові С С++

Кількість елементів – це константа, або константний вираз, що визначає

розмір даного масиву.

Приклад оголошення масиву з ім'ям A, що складається з десяти елементів цілого типу наведено нижче:

```
int A [10] ;
```

Як і у випадках із оголошенням простих змінних, під час оголошення масиву можна ініціалізувати його усі елементи або тільки декілька початкових. Приклад оголошення масиву з ініціалізацією трьох елементів із десяти наведено нижче:

```
int A [10] ={2,5,10};
```

Незалежно від того, скільки елементів масиву було ініціалізовано при оголошенні, пам’ять виділяється під усі елементи. Значення елементів масиву, що не були ініціалізовані, невизначені («сміття») або дорівнюють нулю, якщо масив визначений як глобальний чи статичний.

Якщо у оголошенні масиву ініціалізуються усі елементи (повна ініціалізація), то кількість елементів у оголошенні можна не показувати, хоча квадратні дужки залишаються. Приклад такого оголошення наведено нижче:

```
int A [] ={2, 5, 10, 3, 6, 0, 9, 4, 5, 7};
```

Проініціалізований таким чином масив, для наочності представимо рисунком 7.2.

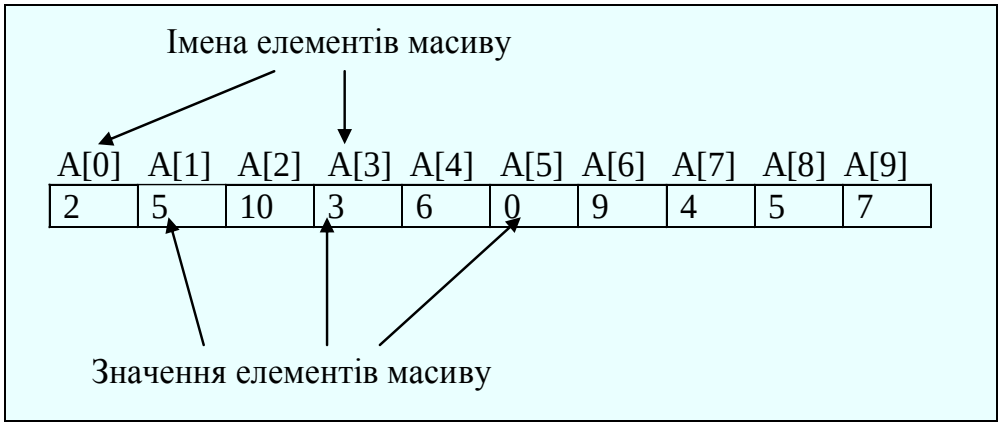


Рисунок 7.2 – Схема одновимірного масиву

7.2 Звертання до елементів масиву через індекси

Для доступу до елементів масиву використовується синтаксична конструкція, що складається з імені масиву та індексу, який записується у квадратних дужках. Тобто доступ до елементів цього масиву забезпечується виразом: A[i]. Індекс і в даному прикладі є цілим числом у діапазоні від 0 до 9. Таким чином, A [0] - це ім'я першого елемента, і т.д., A [9] - ім'я останнього елементу.

Індексовані елементи масиву можуть бути використані так само, як і прості змінні. Наприклад, вони можуть перебувати у виразах як операнди, їм можна привласнювати будь-які значення, відповідні їх типу.

Працюючи з масивами у програмах на мові С слід пам’ятати, що ніякого контролю за значеннями індексів, що використовуються для доступу до елементів масиву нема. Ви можете звернутися до «елементу масиву» з будь яким номером, але отримаєте невідомо що. Ще гірше, якщо ви зміните значення цього «елементу масиву». Це може призвести до катастрофічних наслідків для вашої програми.

Нижче наведено приклад створення масиву для збереження чисел Фібоначчі. Два перших числа Фібоначчі рівні 0 та 1, а кожне наступне дорівнює сумі двох попередніх. У програмі перші 2 елементи масиву заповнюються до циклу, решта - в циклі.

```
int a[10];
a[0] = 0;
a[1] = 1;
for (i=2 ; i<10; i++)
        a[i] = a[i – 2] + a[ i – 1 ];
```

7.3 Одновимірні масиви як параметри функцій

Якщо формальним параметром функції є масив, то він оголошується майже так само, як і прості змінні, лише після його імені слід поставити пусті

квадратні дужки. Тип масиву при цьому записується таким же як і тип елементів масиву.

Слід також пам’ятати, що хоча перед іменем масиву, як формального параметру, символ & не ставиться, та все одно масив буде передано за посиланням. А символ & ставити не потрібно тому, що ім’я масиву і є адресою першого елементу масиву.

Крім того, слід брати до уваги той факт, що масив «не знає», скільки у нього елементів, тому передаючи масив до функції слід передавати і кількість елементів масиву, що має бути оброблена. Це число, звичайно, не може перевищувати кількість елементів під які виділено пам’ять під час оголошення масиву.

Як приклад розглянемо функцію, що знаходить і повертає максимальний елемент масиву:

```
int max(int m[], int n){
   int mx=INT_MIN;
   for(int i=0; i<n; i++)
      if (m[i]>mx) mx=m[i];
   return mx;
}
```

Слід звернути увагу на те, що у мові С С++ такого типу як «масив», не існує, не можна написати int[]. З цієї причини у функції не можна вказати масив, як тип того, що повертається функцією, але масив можна повернути через параметри функції. Ось, наприклад, як може виглядати функція, що повертає масив чисел Фібоніччі.

```
   void fibo(int a[], int size){
         a[0] = 0;
         a[1] = 1;
         for (i=2 ; i<size; i++)
```

```
        a[i] = a[i – 2] + a[ i – 1 ];
}
```

Як бачимо, тип значення, що повертається тут void, але завдяки тому, що масиви передаються через посилання, функція заповнює числами Фібоначчі масив, посилання на який їй передано через параметр.

Щоправда, слід мати на увазі, що функція може повертати покажчик на масив, і таким чином проблема повернення масиву теж вирішується. Але покажчики ми поки що не розглядаємо. Ця тема буде розглянута трохи пізніше.

7.4 Реалізація простих алгоритмів обробки масивів

Під час роботи з масивами чисел доводиться виконувати ряд специфічних операцій, пов'язаних саме з цією структурою даних числового типу. Найбільш поширеними з них є:

- введення масиву чисел;
- виведення масиву чисел;
- формування масиву випадкових чисел;
- пошук суми елементів масиву;
- пошук максимального та мінімального елементів масиву та їх індексів;
- пошук індексу елементу масиву за його значенням;
- циклічний зсув елементів масиву праворуч або ліворуч;
- видалення елементу з масиву;
- формування масиву накопичених значень елементів.

Нижче ми розглянемо деякі з таких функцій на прикладах обробки цілочислових масивів.

Незважаючи на відмінність завдань, що вирішуються цими функціями, у них буде дві однакові особливості.

Перша полягає в тому, що в кожну з цих функцій буде передаватися ім'я масиву і функція буде обробляти саме цей масив, а не його копію.

Друга особливість полягає у тому, що окрім масиву до функції слід

передавати кількість даних у масиві, бо оголошений розмір масиву зазвичай перевищує кількість даних у ньому.

7.4.1 Функція формування випадкового масиву

Ця функція дещо відрізняється від функції генерації послідовності випадкових чисел, що була розглянута раніше. Різниця полягає у тому, що сформовані числа записуються у елементи масиву. Текст функції наведено нижче.

```
void createRandomArray(int ar[], int size, int modul){
   for(int i=0; i<size; i++)
      ar[i] = rand()%modul;
};
```

7.4.2 Функції виведення масиву на консоль

Ця функція досить проста і не потребує коментарів. Ця функція досить проста і не потребує коментарів. Текст функції наведено нижче.

```
void arToConsole(int ar[], int size){
   for(int i=0; i<size;i++){
      cout<<ar[i];
      if(i<size-1)cout<<" ";
   }
   cout<<endl;
}
```

7.4.3 Функції введення масиву з консолі

Можливі декілька варіантів введення масиву з консолі. Перший полягає у тому, що користувач одразу набирає числа масиву, розділяючи їх пробілами, тобто масив чисел являє собою рядок символів. Цей рядок вводиться за допомогою функції gets(), а далі перетворюється у послідовність чисел. Перевага цього способу полягає у простоті вводу, а недолік у тому, що доводиться перетворювати символьний рядок у масив чисел.

Другий варіант полягає у послідовному введенні розміру та кожного елементу масиву з консолі як чисел за допомогою об’єкту cin. Тут вже не потрібно турбуватися про перетворення символів у числа, але сама процедура введення ускладнена.

Нижче ми розглянемо обидва варіанти.

7.4.3.1 Функція введення масиву як рядка символів

Сама функція, що наведена нижче, виглядає досить просто і фактично забезпечує тільки введення рядка символів, а для перетворення рядка символів у масив використовується функція strToArr(). Текст функції введення масиву як рядка символів наведено нижче.

```
void getArFromConsole1(int ar[], int &size){
char str[80];
cout<<"Введіть елементи масиву через пробіли"<<endl;
gets(str);
strToArr(str, ar, size);
}
```

Далі наведено функцію перетворення рядка символів у масив.

```
void strToArr(char str[], int ar[], int &size){
  int i (0);  size=0;
  for( ; ; ){
    while( str[i]!='\0'&&  str[i]==' ')i++;//ігноруємо пробіли
    if(str[i]=='\0')return;
    ar[size]=0;//Формуємо наступне число
    while(str[i]!='\0' && str[i]!=' '){
      if(!isdigit(str[i] )){
        cout<<str[i]<<" не цифра!  \n";
        return;
      }
      ar[size]=ar[size]*10+(str[i]-48); i++;
    }
    size++;
  }
}
```

У цій функції довжину рядка визначено розміром 90 виходячи з ширини

консолі. Для введення рядка використовується функція gets() а не об’єкт cin, тому що цей об’єкт буде сприймати пробіл, як кінець рядка. Використання функції gets() потребує підключення заголовного файлу <cstdio> або < stdio.h>.

Роботу по перетворенню рядка символів у масив виконує функція strToArr():

У цій функції для перевірки, чи є символ цифрою, використовується функція isdigit(), що визначена у заголовному файлі <cctype> або <ctype.h>.

Для перетворення символу цифри у число використовується тай факт, що код цифри на 48 більший ніж значення цієї цифри.

Значення усього числа накопичується за схемою Горнера, відповідно до якої, наприклад, число 12345 обчислюється так: (((1*10+2)*10+3)*10+4)*10+5 .

7.4.4 Функція введення масиву по елементам

Текст функції наведено нижче.

```
void getArFromConsole2(int ar[], int &size) {
   cout<<"\n Введіть кількість елементів масиву ";
   cin>>size;
   for( int i=0; i<=size; i++){
      system("cls");
      cout<<" Усього кількість елементів масиву "<<size<<"\n";
      if (i>0){
         cout<<" Введені елементи масиву: \n";
         for(int j=0; j<i; j++){ cout<<ar[j]<<" "; }
         cout<<"\n";
      }
      if(i<size){
         cout<<"Введіть елемент масиву № "<<i<<": ";
         cin>>ar[i];
      }
   }
}
```

У цій функції кожен елемент масиву вводиться за допомогою об’єкту cin

у циклі for. До початку цього циклу слід ввести кількість елементів масиву.

Функція могла б виглядати так само просто як і функція виведення масиву, що наведена у пункті 9.2.1, і ви можете її саме так і написати, але у наведеній нижче реалізації додано операції очищення екрану і виведення елементів, що були вже введені, і очищення екрану для того, щоб кожен елемент масиву вводився на тому ж самому місці, що й попередній.

Для того, щоб функція працювала, необхідно до складу проекту додати директиву включення заголовний файлу <windows.h> .

7.4.5 Функція вилучення елементу з масиву

У цій функції елементи масиву перебираються, поки не буде знайдено заданий елемент. Після цього усі наступні елементи зсуваються вліво на одну позицію, таким чином займаючи місце елементу, що видаляється. Розмір масиву зменшується на 1. Після цього пошук триває, поки не буде досягнутий кінець масиву.

Для перебору елементів масиву у функції використовується цикл **while,** тому що індекс елементу у разі зсуву елементів ліворуч не потрібно змінювати. Текст функції наведемо нижче.

```
void delElement(int element, int ar[], int &size){
  int i=0;
  while(i < size){
     if(ar[i] == element) {
        size = size - 1;
        for(int j = i; j<size; j++)
           ar[j] = ar[j+1];
     }
     else  i++;
  }
}
```

7.4.6 Функція перевороту масиву

Текст функції наведено нижче.

```
void transAr(int ar[], int size){
   for(int i=0; i<size/2; i++){
      int tmp = ar[i];
      ar[i]=ar[size-i-1];
      ar[size-i-1]=tmp;
   }
}
```

У цій функції елементи масиву, симетрично розташовані відносно середини, міняються місцями. Тому треба перебирати тільки половину масиву.

7.4.7 Функція формування масиву накопичених значень

```
void accumAr(int ar[], int ars[], int size){
  ars[0]=ar[0];
  for(int i=1; i<size; i++){
     ars[i]=ars[i-1]+ar[i];
  }
}
```

Ця функція створює новий масив, такої ж довжини, як і вихідний, але в цьому масиві кожен елемент дорівнює сумі елементів вихідного масиву від першого до поточного елементу.

7.5 Масиви символів

У мові С не було спеціального типу для оголошення рядків символів, у С++ і Qt такі типи є, але ми не будемо поки що їх розглядати.

Коли у тексті програми на мові С з’являлася константа, охоплена подвійними лапками, наприклад, “Невже це не рядок символів?”, то вона розглядалася, як масив. Для такого масиву в оперативній пам’яті виділялася ділянка, розмір якої був на один байт більший, ніж кількість символів у рядку. Цей додатковий байт і досі використовується для збереження ознаки кінця рядка.

У якості такої ознаки використовується символ ‘\0’.

Символьні рядки, які є змінними програми, оголошуються, як звичайні масиви, рисунок 7.3.

```
char   <ім’я символьного рядка> [<кількість байтів >] ;
```

Рисунок 7.3 – Синтаксис оголошення рядка символів

Приклад оголошення рядка символів з ім'ям str, для якого у памяті буде виділено 100 байтів наведено нижче:

```
char  str[100] ;
```

Як і у випадках із оголошенням масивів чисел, масиви символів можна ініціалізувати. Незалежно від того, скільки символів було ініціалізовано при оголошенні, пам’ять виділяється під усі елементи. Значення символів, що не були ініціалізовані, невизначені («сміття») або дорівнюють нулю, якщо масив визначений як глобальний чи статичний.

Приклади оголошення рядків символів з одночасною ініціалізацією наведено нижче:

```
char  str[10] ={‘H’, ‘e’, ‘l’, ‘l’, ‘o’,  ‘!’, ‘\0’} ;
char  str[10] =”Hello!”;
```

Зверніть увагу, якщо рядок ініціалізується як класичний масив, з окремих символів, то слід не забути про ознаку кінця рядка. Та на щастя масив символів можна ініціалізувати за допомогою рядка символів у подвійних лапках. Ознака кінця рядка у цьому випадку додається автоматично.

Якщо у оголошенні рядка символів ініціалізуються усі елементи (повна ініціалізація), то кількість елементів у оголошенні можна не показувати, хоча квадратні дужки залишаються. Приклад такого оголошення наведено нижче:

```
char  str[] =”Hello!”;
```

Що стосується обробки рядків (масивів символів) то усе залишається

таким самим, як і у випадку числових масивів.

7.6 Реалізація алгоритмів обробки рядків символів

Незважаючи на те що за своєю структурою рядки символів майже нічим не відрізняються від масивів чисел, завдання функцій з обробки рядків суттєво відрізняються від завдань функцій з обробки масивів чисел. Так, завдання пошуку суми елементів рядка, або пошук максимального значення елементу рядка не мають ніякого сенсу.

Завдання функцій з обробки рядків зовсім інші, нижче наведено деякі з них:

- копіювання частини рядка;
- вставка рядка у рядок;
- вилучення частини рядка;
- заміна одних символів іншими;
- вилучення деяких символів;
- визначення позиції групи символів у рядку.

7.6.1 Функція копіювання частини рядка

```
void subStr(char str[], char sub[], uint index, uint count){
   if (index>strlen(str))count=0;
   if(index+count-1>strlen(str))count=strlen(str)+1-index;
   for(int i=0, j=index; i<count; i++, j++)
      sub[i]=str[j];
   sub[count]='\0';
}
```

7.6.2 Функція знаходження частини рядка у рядку

```
int posInStr(char str[], char sub[]){
    for(int i=0; i<=strlen(str)-strlen(sub); i++){
      int j=0;
```

```
    while(str[i+j]==sub[j] && j<strlen(sub))j++;
    if (j==strlen(sub))return i;
  }
  return -1;
}
```

7.7 Сортування масивів

При роботі з масивами часто виникає необхідність розміщення елементів у зростаючому або спадаючому порядку. Уявіть, наскільки важко було б користуватися словником, якби слова в ньому не розташовувалися в алфавітному порядку. Так само і швидкість та простота алгоритмів обробки масивів багато в чому залежать від порядку, в якому їх елементи зберігаються в пам'яті комп'ютера.

Сортування масиву – це процес перестановки елементів масиву з метою розміщення елементів масиву в певному порядку.

Наприклад, якщо сортується масив A чисел по зростанню, то після сортування цього масиву буде виконуватися умова:

A[0] < A[1] < A[2] < A[3] < …….< A[n]

Найчастіше задачі сортування вирішуються в інформаційних пошукових системах, бо пошук у впорядкованих масивах проводиться набагато швидше, ніж у невпорядкованих.

Існують різні методи сортування масивів. Тут ми розглянемо найпростіші з них, а саме:

- сортування методом вибору.
- сортування методом обміну (метод бульбашки);
- сортування методом вставки.

7.7.1 Сортування вибором

Алгоритм сортування елементів масиву у зростаючому порядку за методом вибору можна описати так:

1. Серед усіх елементів масиву, починаючи з першого, шукають мінімальний елемент.
2. Знайдений мінімальний елемент міняють місцями з першим елементом.
3. Переглядають масив від другого елементу, і знаходять мінімальний серед цих елементів.
4. Знайдений елемент міняють місцями з другим елементом.
5. Далі те саме з третім елементом, і так далі до останнього елементу.

Аналіз описаних вище дій показує, що для програмної реалізації цього методу сортування буде потрібно два цикли for.

У зовнішньому циклі повинен змінюватися номер елементу, куди буде заноситися черговий мінімальний елемент. Номери цих елементів мають змінюватися від першого до передостаннього. Цей цикл буде визначати кількість проходів по масиву.

Внутрішній цикл повинен забезпечити послідовне порівняння елементу, зафіксованого першим циклом, з усіма елементами, які слідують в масиві за ним. У тілі внутрішнього циклу відбувається порівняння елементів, індекси яких задаються параметрами зовнішнього і внутрішнього циклу. Якщо в результаті порівняння знаходиться елемент, що менший ніж зафіксований, то порівнювані елементи міняються місцями.

Схема алгоритму сортування масиву методом вибору показана на рисунку 7.4.

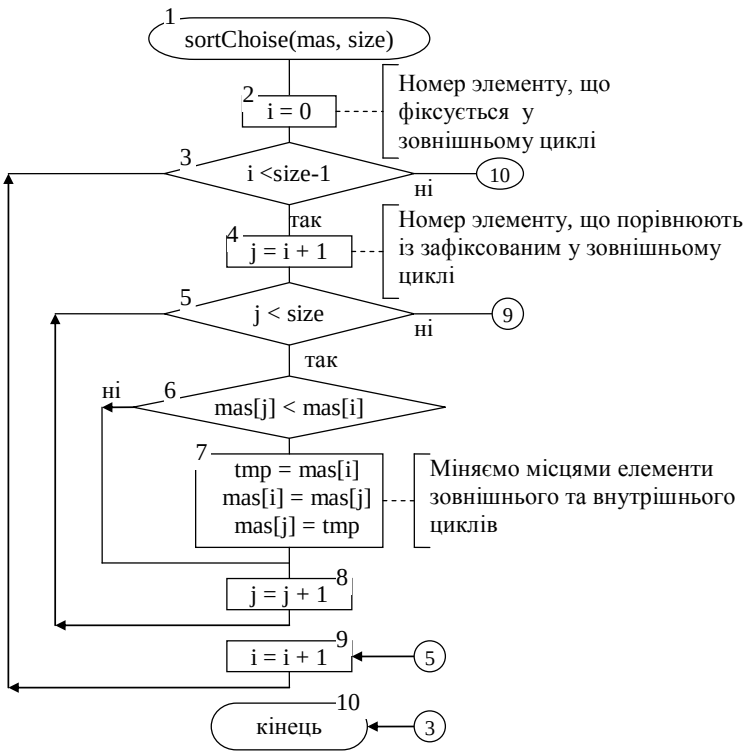


Рисунок 7.4 - Алгоритм сортування елементів масиву у зростаючому порядку за методом вибору

Вихідними даними для алгоритму є: сортований масив mas і кількість елементів у цьому масиві size.

7.7.1.1 Функція сортування масиву методом вибору

Функція, представлена на рисунку 10.7 реалізує за допомогою циклів for алгоритм, зображений на рисунку 7.5.

```
void sortChoice(int mas[], int size){
  for(int i=0; i<size-1;i++){
     for(int j = i + 1; j<size; j++) {
        if (mas[j] < mas[i] ){
           int tmp = mas[i];
           mas[i] = mas[j];
           mas[j] = tmp;
        }
     }
  }
}
```

Рисунок 7.5 – Функція сортування елементів масиву у зростаючому порядку за методом вибору

Наведена функція не єдиний варіант реалізації методу вибору.

Для зменшення кількості операцій перезапису можна у внутрішньому циклі тільки запам’ятовувати індекс мінімального елементу. А обмін робити після завершення внутрішнього циклу. Саме такий спосіб слід реалізувати у проекті до лабораторної роботи.

7.7.2 Сортування обміном (метод бульбашки)

Алгоритм заснований на принципі порівняння та обміну значеннями (у разі необхідності) між сусідніми елементами. Кожен елемент масиву, починаючи з першого, порівнюється з наступним, і якщо він більше наступного, то елементи міняються місцями. В результаті після першого проходу, при сортуванні на зростання, найбільший елемент виявиться на останньому місці.

У наступному проході все повторюється заново і найбільший серед решти елементів виявляється на передостанньому місці.

Схему алгоритму сортування елементів масиву у зростаючому порядку за методом обміну показана на рисунку 7.6.

Цей метод отримав ще назву «метод бульбашки». Назва стає зрозумілою, якщо розташовувати масиви вертикально У процесі реалізації алгоритму елементи з меншим значенням просуваються до початку масиву (спливають), а елементи з великим значенням переміщуються у кінець масиву (тонуть).

Кожен прохід по масиву від його початку упорядковує щонайменше один елемент у хвості масиву, тому кожного наступного проходу доводиться розглядати менше пар елементів ніж на попередньому проході, бо останні елементи вже впорядковані.

Інколи початкове розташування елементів може бути таким, що масив може бути відсортований за невелику кількість проходів. Ознакою впорядкованості масиву є відсутність обмінів. Тому для скорочення кількості проходів по масиву, слід після закінчення кожного з них перевіряти, чи був зроблений хоч один обмін значень елементів.

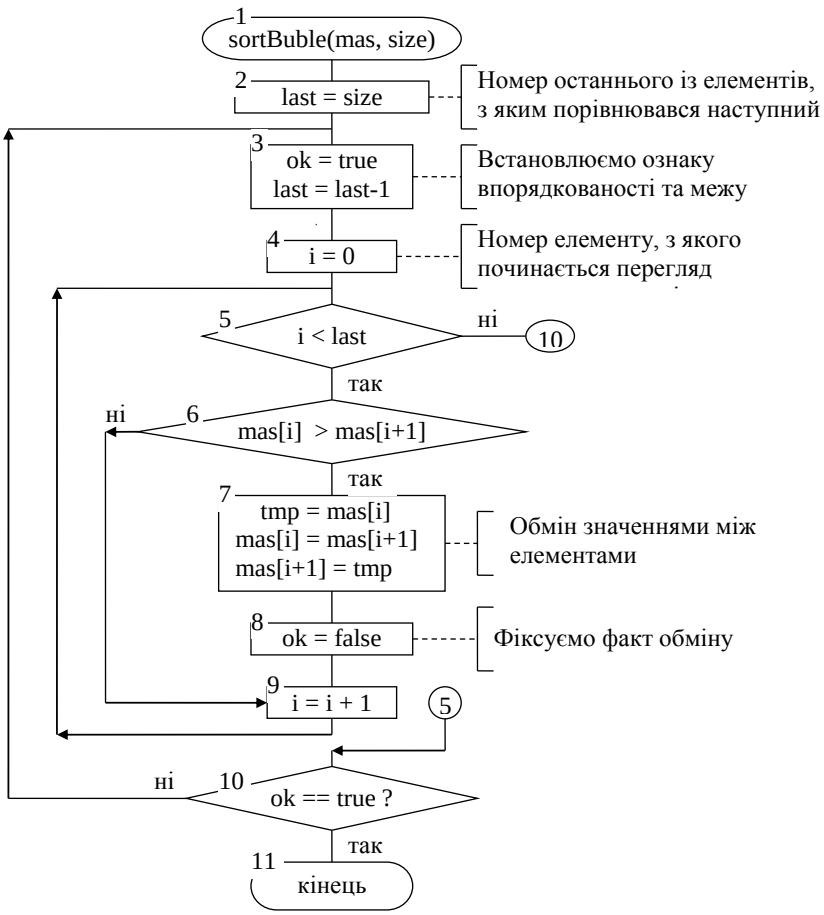


Рисунок 7.6 - Алгоритм сортування елементів масиву у зростаючому порядку за методом обміну

У алгоритмі, що зображений на рисунку 10.8, сортування реалізується за допомогою двох циклів. Зовнішній цикл виконується поки по завершенні чергового проходу по масиву не виявиться, що перестановок елементів не було. У цьому випадку змінна ok збереже значення true, і це означатиме, що масив відсортований. Наявність обмінів між елементами при проході по масиву можна перевірити тільки після завершення проходу, тому зрозуміло, що в даному випадку доречним буде цикл do ... while.

У внутрішньому циклі послідовно змінюється індекс елементів, що порівнюються. Тому для організації проходу по масиву використовується цикл for. Максимальне значення параметра цього циклу у кожному проході зменшується на 1.

7.7.2.1 Функція сортування масиву методом обміну

Функція, представлена на рисунку 10.16 реалізує за допомогою циклів for алгоритм, зображений на рисунку 10.1.

```
void sortBubl(int ar[], int size){
int  last=size; bool ok;
do{
   last=last-1;
   ok = true;
   for(int i= 0; i<last; i++){
      if(ar[i] > ar[i+1]){
         int x = ar[i];
         ar[i] = ar[i+1];
         ar[i+1] =  x;
         ok =  false;
      }
   }
}while(!ok);
}
```

Рисунок 7.7 - Функція сортування елементів масиву у зростаючому порядку за методом обміну

Роботу функції, представленої на рисунку 7.7 можна прискорити, якщо після обміну елементів фіксувати номер елементу, що останнім змінювався (значення змінної i). Це значення можна використовувати і як ознаку завершення сортування, так і в якості максимального значення параметру внутрішнього циклу. Це удосконалення алгоритму дозволяє дещо зменшити кількість циклів сортування.

7.7.3 Сортування вставкою

Суть алгоритму полягає у наступному. На кожному кроці елементи масиву поділені на впорядковану частину A[1], A[2], ..., A[i-1], яка розташовується на початку масиву, та невпорядковану частину A [i], ... , a [N], і перший елемент з невпорядкованої частини вставляється в упорядковану. На початку сортування впорядкована частина складається всього з одного, першого елемента, а всі інші

елементи розташовуються в другій частині масиву. Схема цього алгоритму наведена на рисунку 7.8.

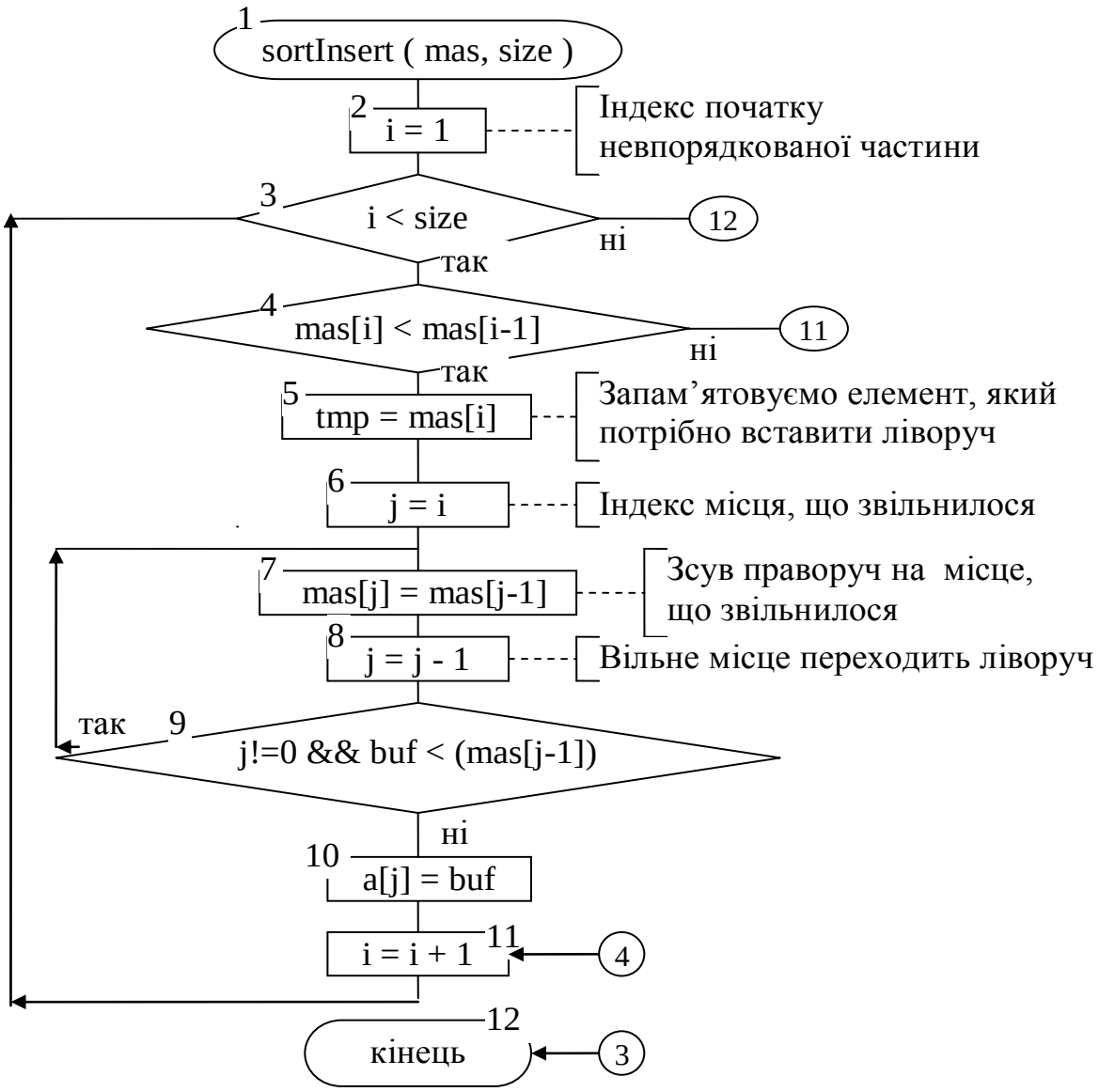


Рисунок 7.8 - Алгоритм сортування елементів масиву у зростаючому порядку за методом вставки

Послідовні кроки алгоритму сортування полягають у тому, що перший елемент з невпорядкованої частини порівнюється з останнім елементом впорядкованої послідовності. Якщо виявляється, що порядок розташування цих елементів не відповідає вимогам сортування, то елемент з невпорядкованої частини вилучається і переноситься в упорядковану частину. Місце для цього елементу звільняється шляхом зсуву упорядкованих елементів вправо на місце витягнутого елемента. Зсув упорядкованих елементів на одну позицію вправо

триває, поки не буде знайдено місце для елемента, вилученого з невпорядкованої послідовності.

Як видно з рисунку 7.8, в алгоритмі є два цикли.

У зовнішньому циклі послідовно змінюється номер лівої межі невпорядкованою області від другого елементу (i=1) до кінця масиву. У тілі цього циклу порівнюються елементи, що знаходяться по обидва боки від межі, що розділяє впорядковану і невпорядковану частини. Якщо порядок порушений, то перший елемент невпорядкованою послідовності запам'ятовується у змінній tmp, внаслідок чого звільняється місце для зсувів упорядкованих елементів вправо.

Внутрішній цикл забезпечує послідовні зсуви упорядкованих елементів вправо, починаючи з останнього, доти, поки не буде знайдено місце для першого елемента з невпорядкованою області, що зберігався у змінній tmp.

Можливість вставки елементу визначається однією з двох умов.

- (mas[j-1] <= buf < mas[j]),якщо (1 < j < i), тобто знайдено місце всередині впорядкованої послідовності.
- j=1 , тобто. tmp менше ніж усі елементи впорядкованої частини і має бути поставлений на перше місце у впорядкованій частині масиву.

Після виконання однієї з цих умов цикл зсувів завершується і елемент масиву із змінної tmp переноситься на знайдене місце у впорядкованій послідовності.

7.7.3.1 Функція сортування масиву за методом вставки

Програмна реалізація алгоритму за методом вставки наведена на рисунку 7.9.

Ефективність роботи цієї функції можна дещо підвищити за рахунок зменшення кількості порівнянь під час пошуку місця для елементу із буфера. Зважаючи на те, що ліва частина масиву вже впорядкована, місце вставки для елементу із буфера можна знайти методом дихотомії. Цей метод буде розглянуто у підрозділі 10.3. Суть методу полягає в тому, що елемент, який знаходиться в

середині області пошуку, порівнюється із елементом у буфері. В результаті порівняння можна визначити, в якій з половин області пошуку знаходиться місце для елементу із буфера, праворуч або ліворуч. Таким чином, в результаті одного порівняння область пошуку звужується наполовину.

Місце вставки буде знаходитися за останнім значенням правої межі області пошуку.

Визначивши місце вставки, можна робити зсув елементів лівої частини масиву праворуч і вставляти елемент із буфера.

Саме таку реалізацію алгоритму слід зробити у проекті до лабораторної роботи.

```
void sortInsert(int ar[], int size){
   for(int i=1; i<size; i++){
      //Порівнюємо елементи на межі між частинами масиву
      if(ar[i] < ar[i-1] ){
         int buf = ar[i];   // Порядок порушено, переносимо i-й елемент у буфері
         // Починаємо зсуви праворуч на місце i-го елементу
         int j = i;   // j – індекс вільного місця
         do{
               ar[j]=ar[j-1];   // зсув праворуч
               j=j-1;
               // поки ліворуч число більше,  або дійшли до початку масиву
           }while (j != 0 && buf <ar[j-1] );
         //Вставляємо елемент, що мав номер i на нове місце з индексом j
         ar[j] = buf;
      }
   }
}
```

Рисунок 7.9 – Функція сортування елементів масиву у зростаючому порядку за методом вставки

7.8 СОРТУВАННЯ ЗА УСКЛАДНЕНИМИ ПРАВИЛАМИ

Вище ми розглянули різні методи сортування масивів, в яких порядок розташування елементів визначався застосуванням найпростішої операції порівняння «більше» або «менше». В результаті застосування цих операцій ми отримуємо сортування на зростання або на спадання.

Однак у деяких випадках необхідно використовувати більш складні критерії порівняння елементів, ніж просте порівняння. У цих випадках доцільно написати функцію порівняння елементів по заданому правилу, яка буде повертати true, якщо порядок проходження елементів не порушений і false у противному випадку. Цю функцію слід викликати в тих місцях процедури сортування, де потрібно порівняння елементів.

Наприклад, нехай правило сортування масиву цілих чисел сформульовано таким чином: «Спочатку парні числа за спаданням, а потім непарні за зростанням».

Функцію, яка порівнює елементи таким чином, наведено нижче:.

```
bool goodDisposition(int x1, int x2){
   if (x1 % 2 != x2 % 2)
        //Якщо одне число парне а друге непарне, то меншим буде парне
        return x1 % 2 < x2 % 2;
   else if( x1 % 2 == 0)  // Числа парні, має бути спадання
            return x1>x2;
      else     // Числа непарні, має бути зростання
            return x1 <  x2;
}
```

Тепер за допомогою створеної функції можна відсортувати масив в потрібному порядку. Метод сортування може бути будь-яким. Для прикладу використаємо сортування вибором:

```
void sortChoise(int ar[], int size){
   for(int i=0; i<size-1;i++){
      for(int j = i + 1; j<size; j++) {
```

```
        if (!goodDisposition(ar[i] ,ar[j])){
            int x =  ar[i];
            ar[i] =  ar[j];
            ar[j] =  x;
        }
      }
   }
}
```

7.9 Реалізація алгоритмів роботи з упорядкованими масивами

Впорядковані масиви найчастіше використовуються як сховища деякої інформації. Найчастіше зустрічаються такі завдання, пов'язані з їх обробкою:

- пошук позиції елемента в масиві;
- вставка елементу у масив, без порушення порядку;
- видалення елемента з масиву;
- об'єднання двох масивів в один зі збереженням порядку;

Нижче розглядаються функції, які вирішують ці завдання.

7.9.1 Пошук позиції елемента у впорядкованому масиві

Для пошуку позиції елемента у впорядкованому масиві можна використовувати метод дихотомії (ділення області пошуку навпіл). У цьому методі елемент, який знаходиться в середині області пошуку, порівнюється із зразком, який потрібно знайти. Якщо він не відповідає зразку, то по його значенню можна визначити, в якій з половин області пошуку може знаходитися потрібний елемент, праворуч або ліворуч. Таким чином, в результаті одного порівняння область пошуку звужується наполовину.

Цикл пошуку повторюється доти, поки потрібний елемент не буде знайдений, або область пошуку звузиться до одного елемента, що буде свідчити про те, що потрібного елементу в масиві немає.

Функція, яка повертає номер позиції елементу у впорядкованому масиві

наведено нижче. Якщо елемент не знайдено, функція повертає -1.

```
int findPos(int element, int ar[], int size){
   // left, right – лівая та права межі області пошуку
   int left = 0, right = size-1;
   while (left <= right){
      // Обчислюємо індекс середини області пошуку
      int pos=(right + left)/2;
      if(ar[pos] == element)
         return pos; //елемент знайдено
      if  (element < ar[pos])
         right = pos-1; // елемент знаходиться ліворуч
      else left = pos + 1; //елемент знаходиться праворуч
   }
   // Елемент не знайдено
   return -1;
}
```

7.9.2 Вставка елементу до впорядкованого масиву

Алгоритм вставки елементу до впорядкованого масиву вже розглядався, як частина алгоритму сортування вставкою. Він полягає у послідовному аналізі елементів масиву, починаючи з останнього. Якщо елемент масиву більший ніж той, що потрібно вставити, елементи масиву переміщують вправо на одну позицію, для того, щоб звільнити місце для елемента, що вставляється. Зсуви проводяться доти, поки не буде знайдено місце, відповідне значенню елемента, що вставляється.

Якщо всі елементи масиву більше ніж елемент, що додається, то всі елементи масиву перемістяться праворуч, а новий буде поставлений на перше місце. Зрозуміло, що кількість елементів масиву при цьому збільшується на один.

Функцію вставки елемента до впорядкованого масиву таким методом, наведено нижчен.

```
void addToSortAr(int element, int ar[], int &size){
   int i = size - 1;
   while( ar[i]>element &&  i>=0 ){
      ar[i+1] = ar[i];
      i--;
   }
   ar[i+1] = element;
   size += 1;
}
```

Але цю функцію можна дещо прискорити за рахунок зменшення кількості порівнянь, використавши метод дихотомії для пошуку місця вставки.

Такий варіант функції вставки наведено далі. Як бачимо, спроба підвищити ефективність алгоритму приводить до його ускладнення.

```
void addToSortArray(int element, int ar[], int &size){
   // left і right лівая і права границі області пошуку
   int left = 0, right = size-1;
   while (left <= right){
      // Обчислюємо індекс середини області пошуку
      int pos=(right + left)/2;
      if  (element < ar[pos] )
         right = pos-1;// потрібна позиція заходиться ліворуч
      else left = pos + 1;//потрібна позиція заходиться праворуч
   }
   //Позиція вставки знаходиться після елементу з індексом right
   int i;
   for( i=size; i>right; i--)
        ar[i+1]=ar[i];
   ar[i+1]=element;
   size+=1;
}
```

7.9.3 Видалення елементу з упорядкованого масиву

В алгоритмі видалення елементу з упорядкованого масиву можна використовувати розглянуту вище функцію для пошуку індексу елементу, що видаляється. Після визначення цього індексу, елемент видаляється шляхом зсуву

ліворуч на одну позицію всіх елементів, що знаходяться за тим, який потрібно видалити. Значення змінної, в якій зберігається кількість елементів, у кожному циклі зменшується на одиницю.

Функцію видалення елементу з упорядкованого масиву наведено нижче. У цій функції передбачено, що у масиві можуть бути однакові елементи.

```
bool delFromSortAr(int element, int ar[], int &size){
   int pos;
   while(( pos = findPos(element, ar,size))!= -1) {
      size--;
      for(int i=pos;i<size;i++)
         ar[i]=ar[i+1];
   }
   return pos!=-1 ? true : false;
}
```

7.9.4 Злиття двох впорядкованих масивів

Функція злиття масивів наведена нижче.

```
void joinSortArrays(int ar1[], int ar2[], int ar3[], int size1, int size2, int &size3){
   size3 = size1 + size2;
   int k=0, i=0, j=0; // поточні індекс для першого, другого та третього масивів
   while (i < size1 && j < size2) {
      if (ar1[i] < ar2[j])  // Добавляємо елемент із першого масиву
         ar3[k++] = ar1[i++];
      else  // Добавляємо елемент із другого масиву
         ar3[k++] = ar2[j++];
    }
   for(i= i;i< size1; i++, k++)  // Додаємо залишок з першого массиву, якщо він є
      ar3[k] = ar1[i];
   for(j = j; j<size2; j++, k++) // Додаємо залишок з другого массиву, якщо він є
      ar3[k] = ar2[j];
}
```

У алгоритмі, що реалізує наведена функція, поточні елементи вихідних

масивів порівнюються, і у новий масив переноситься менший елемент. При цьому поточна позиція масиву, з якого був переписаний елемент, переміщується до наступного елементу. Цикл порівнянь продовжується поки не закінчяться елементи одного з масивів. Після цього елементи, що залишилася у другому масиві просто дописуються у новий масив.

Зверніть увагу на те, як у функції використовуються операція постфіксного інкременту в операціях присвоєння. Для присвоєння використовуються старі значення індексів, а після присвоєння значення індексів збільшуються на 1.

Після того, як один із масивів вичерпався, дописування решти з другого масиву здійснюється за допомогою циклу for, у якому одночасно змінюються обидва індекси і в якості перших індексів використовуються ті самі змінні і їх останні значення у попередньому циклі.

8 БАГАТОВИМІРНІ МАСИВИ

8.1 Оголошення та ініціалізація багатовимірних масивів

Багатовимірний масив містить елементи, розташування яких визначається декількома індексами. Найчастіше використовуються двовимірні масиви, які ще називають матрицями.

Матриця - це структура даних, якій відповідає таблиця. При роботі з матрицями оперують поняттями ім'я матриці, стовпець, номер стовпця, рядок, номер рядка, елемент, розмір матриці, наприклад 3 x 4.

Кожен елемент матриці має своє значення, але тип усіх елементів матриці однаковий. Для ідентифікації елементів матриці використовують індекси. З поняттям індексу ми вже зустрічалися в попередній роботі, де за допомогою індексу ідентифікувалися елементи одновимірного масиву.

У матриці, для ідентифікації елементу, потрібно два індекси. Це нібито координати елементу в матриці. При зверненні до елементу його індекси вказують після імені матриці в квадратних дужках, розділяючи їх комою. Наприклад, якщо ім'я матриці matrix, то matrix [2,3] - це елемент матриці з координатами 2 і 3. Перший індекс задає номер рядка, а другий - номер стовпця.

Нумерація рядків і стовпців починається з 0.

Оголошення матриці має вигляд, представлений на рисунку 8.1.

<тип елементів> <ім’я матриці> [<кількість рядків>] [<кількість стовпців>];

Рисунок 8.1 – Синтаксис оголошення матриці

Тип елементів визначає тип елементів, з яких складається матриця. Це може бути будь який допустимий у мові тип, простий або складений

Ім’я матриці – це ідентифікатор написаний за правилами запису імен у мові С С++

Кількість рядків та кількість стовпців – це константи, або константні

вирази, що визначають розмір матриці.

Приклад оголошення матриці з ім'ям A, що складається з десяти рядків по шість елементів цілого типу у кожному рядку наведено нижче:

```
int A [10] [6];
```

Як і у випадках із одновимірними масивами, під час оголошення матриць можна ініціалізувати усі її елементи або тільки декілька початкових. Приклад оголошення матриці з неповною ініціалізацією тільки трьох рядків із десяти наведено нижче:

```
int A [10][6] = {{2,5,10}, {6,12}, {7}};
```

Цей приклад показує, що фактично матриця являє собою масив, кожен елемент якого теж є масивом.

Незалежно від того, скільки елементів матриці було ініціалізовано при оголошенні, пам’ять виділяється під усі елементи. Значення елементів, що не були ініціалізовані, невизначені («сміття») або дорівнюють нулю, якщо матриця визначена як глобальна чи статично.

Якщо у оголошенні матриці ініціалізуються усі елементи (повна ініціалізація), то елементи матриці можна записувати не поділяючи їх на рядки. Приклад такого оголошення наведено нижче:

```
int A [3][4] ={2, 5, 10, 3, 6, 0, 9, 4, 5, 7, 12, 1};
```

У оголошенні матриці при повній ініціалізації кількість рядків можна не показувати, бо їх кількість може бути обчислена виходячи із загальної кількості елементів та розміру рядка. Приклад такого оголошення наведено нижче:

```
int A [][4] ={2, 5, 10, 3, 6, 0, 9, 4, 5, 7, 12, 1};
```

8.2 Звертання до елементів багатовимірних масивів через індекси

Для доступу до елементів матриці використовується синтаксична конструкція, що складається з імені матриці та пари індексів, кожен з яких

записується у своїх квадратних дужках. Тобто доступ до елементів цього матриці з наведеного вище прикладу забезпечується виразом: A[i][j]. Індекс і в даному прикладі є цілим числом у діапазоні від 0 до 2, а індекс j – це ціле число від 0 до3. Таким чином, A [0][0] - це ім'я першого елемента, а A [2][3] - ім'я останнього елементу.

Індексовані елементи матриці можуть бути використані так само, як і прості змінні. Наприклад, вони можуть перебувати у виразах як операнди, їм можна привласнювати будь-які значення, відповідні їх типу.

Як і у випадках із одновимірними масивами слід пам’ятати, що ніякого контролю за значеннями індексів, що використовуються для доступу до елементів матриці нема. Ви можете звернутися до «елементу матриці» з індексами, значення яких перевищують граничні, але отримаєте невідомо що. Ще гірше, якщо ви зміните значення цього «елементу масиву». Це може призвести до катастрофічних наслідків для вашої програми.

8.3 Матриці як параметри функцій

Якщо формальним параметром функції є матриця, то він оголошується майже так само, як і одновимірний масив. Різниця полягає у тому, що обов’язково потрібно вказувати кількість елементів для другого виміру, тобто довжину рядка матриці.

Кількість елементів у рядку, що наведена у заголовку функції обов’язково має бути тотожною кількості елементів у рядку для матриць, що оголошені у програмі, яка викликає цю функцію, і які передаються у функцію як параметри. Тому, для того щоб уникнути помилок, доцільно у таких програмах оголошувати глобальні константи, що визначають розмір матриці, у вигляді макросів за допомогою директиви #define.

Слід також пам’ятати, що хоча перед іменем матриці, як формального параметру, символ & не ставиться, та все одно матрицю буде передано за посиланням. А символ & ставити не потрібно тому, що ім’я матриці і є адресою першого її елементу.

Крім того, слід брати до уваги той факт, що матриця «не знає», скільки рядків у неї реально заповнено, і скільки елементів у рядку, тому передаючи масив до функції слід передавати і кількість рядків і кількість елементів у рядку, що мають бути оброблені. Ці числа, звичайно, не можуть бути більшими ніж значення під які було виділено пам’ять під час оголошення матриці.

Якщо матриця є результатом, який має повернути функція, то такий результат, як і у випадках із масивами ми поки що будемо повертати через параметри функції.

8.4 Реалізація алгоритмів обробки багатовимірних масивів

Приклади функцій, що працюють з матрицями розглянуто у наступних пунктах.

8.4.1 Формування та виведення матриць з використанням консолі

Для виведення матриці на консоль можна використовувати функцію, що зображена на рисунку 8.2. Перед описом функції наведені директиви #define, що визначають максимально можливий розмір матриці.

Для введення матриці з консолі можна скористатися функцією, що наведена на рисунку 8.3.

У цій функції матриця вводиться рядками, але особливість функції полягає у тому, що у разі неоднакової кількості елементів у рядках, що вводяться, кількість елементів у кожному рядку матриці буде дорівнювати кількості елементів останнього рядка. Зайві елементи інших рядків будуть ігноруватися, а ті що не вводилися будуть дорівнювати 0, бо уся ділянка пам’яті, що відведена для матриці перед початком вводу заповнюється нулями.

```
#define ROWS 10
#define COLS 10
...
void printMatrix(int matr[][COLS], int nRow, int nCol) {
    cout << endl;
    for(int i = 0; i < nRow; i++) {
        for(int j = 0; j < nCol; j++) {
            cout << setw(6) << matr[i][j];
        }
        cout << endl;
    }
}
```

Рисунок 8.2 – Функція виведення матриці на консоль

```
void getMatrixFromConsole(int matr[][COLS], int &nRow, int &nCol){
    char s[80];   //Масив символів для введення рядка матриці
    int ar[COLS]; // Масив чисел для поточного рядка матриці
    //Обнуляємо ділянку пам'яті для матриці
    for(int i = 0; i < ROWS; i++)
        for(int j = 0; j < COLS; j++)
            matr[i][j]=0;
    cout<<"Введіть кількість рядкв матриці: ";
    cin>>nRow;
    gets(s);//Забираємо "Enter", що не забрав cin
    for( int i=0; i<=nRow; i++){
        system("cls"); //чистимо екран
        cout<<"Усього рядків у матриці "<<nRow<<"\n";
        if (i>0){
            cout<<"Введені рядки матриці:";
            printMatrix(matr, i, nCol);
        }
        if(i<nRow){
            cout<<"Введіть рядок № "<<i<<endl;
            gets(s);//Читаємо рядок символів
            strToArr(s,ar,nCol); //Перетворюємо символи у масив чисел
            // Записуємо у матрицю черговий рядок
            for(int k=0; k<nCol;k++){
                matr[i][k]=ar[k];
            }
        }
    }
}
```

Рисунок 8.3 – Функція введення матриці з консолі

Для перетворення рядка символів у масив використовується функція strToArr, розглядалася раніше.

8.4.2 Тотальна обробка даних у матрицях

Тотальна обробка передбачає виконання однакових операцій для всіх елементів матриці.

Нижче перераховані деякі завдання тотальної обробки матриць:

– заповнення матриці випадковими або іншими числами;

– пошук суми всіх елементів матриці;

– пошук максимального або мінімального елемента в матриці;

– множення матриці на число;

– отримання суми двох матриць однакового розміру.

Тотальна обробка зазвичай реалізується за допомогою двох вкладених циклів for, параметрами яких є індекси матриці. Якщо матриця обробляється по рядках, то заголовок зовнішнього циклу записується для першого індексу, а якщо по стовпцях, то перший індекс повинен змінюватися у внутрішньому циклі.

Прикладом тотальної обробки матриці може служити функція заповнення елементів числової матриці нулями, що наведена нижче.

До функції передається матриця із зазначенням заявленої кількості елементів у рядку, а також кількість рядків (nRow) та кількість елементів у рядку (кількість стовпців nCol), що підлягають опрацюванню.

Перед описом функції наведені директиви #define, що визначають максимально можливий розмір матриці.

Всі завдання тотальної обробки мають подібну структуру. Відрізняються вони тільки інструкціями всередині циклу.

```
#define ROWS 10
#define COLS 8
…
void matrixToZero(int matr[][COLS], int nRow, int nCol){
   for(int i = 0; i < row; i++)
      for(int j = 0; j < col; j++)
         matr[i][j]=0;
}
```

8.4.3 Вибіркова обробка матриць

При вибірковій обробці матриці можуть вирішуватися ті ж завдання що і при тотальній, але тільки для частини елементів матриці. Ось приклади

можливих варіантів групування елементів матриці:

– елементи головної діагоналі квадратної матриці;

– елементи допоміжної діагоналі квадратної матриці;

– елементи деякого стовпця або рядка матриці;

– елементи матриці, розташовані по її кромці;

– елементи квадратної матриці, розташування яких відповідає розташуванню білих або чорних клітин шахової дошки.

Для перших трьох з перерахованих вище варіантів обробка здійснюється в одному циклі, бо під час перебору елементів тільки один індекс є незалежним.

У першому з наведених варіантів у оброблюваних елементів, які розташовані на головній діагоналі матриці, номери рядка і стовпця співпадають. Тому в циклі можна змінювати номер стовпця, а рядку привласнювати той же самий номер.

У другому випадку, на допоміжній діагоналі, номер стовпця j зв'язаний з номером рядка і співвідношенням j = nCol -1-i. У цьому співвідношенні nCol - це кількість елементів у рядку (кількість стовпців).

У разі обробки одного із стовпців матриці, його номер фіксований, і єдиний цикл достатньо організувати тільки за елементами рядка. Аналогічно оброблюється і окремий рядок.

При обробці елементів матриці розташованих по її кромках можна організувати чотири цикли, що виконуються послідовно один за одним. У кожному циклі обробляються елементи чергового рядка або стовпця, що утворюють кромку матриці. Обробляти слід всі елементи за винятком того, який буде початком наступної послідовності, тобто останнього або першого.

При «шаховому» групуванні, можна організувати подвійний цикл, як при тотальній обробці, але обробляти тільки ті елементи, сума індексів яких є парною (або непарною).

8.4.4 Перестановки елементів матриці

Є завдання, де потрібно міняти місцями елементи матриці. Нижче перераховані деякі з таких завдань:

- транспонування матриці (поворот навколо головної діагоналі);
- поворот навколо допоміжної діагоналі;
- поворот навколо горизонтальної осі;
- поворот навколо вертикальної осі.

Для вирішення всіх цих завдань потрібен подвійний цикл. Зовнішній цикл зазвичай перебирає усі номери рядків або стовпців, іноді, за винятком першого чи останнього номера. Внутрішній же цикл забезпечує опрацювання тільки половини елементів, які розташовані з одного боку від осі повороту. Тіло циклу містить оператори, які забезпечують обмін значеннями між поточним елементом матриці і симетричним йому. Для організації обміну найзручніше використовувати проміжну змінну.

Основні труднощі, що виникають при вирішенні цих завдань, це визначення індексів елемента, симетричного поточному елементу.

У таблиці 8.1 наведені параметри циклів, що забезпечують перевороти елементів матриці навколо різних осей симетрії. Індексні вирази, наведені в таблиці, складені в припущенні, що нижні індекси рядків і стовпців рани одиниці, верхні індекси стовпців і рядків відповідно рівні nCol і nRow, а m - ім'я матриці.

Таблиця 8.1 – Параметри циклів при перестановках елементів матриці

Лінія симетрії у повороті	Зовнішній цикл, за рядками (індекси рядка)	Внутрішній цикл, за елементами рядка (індекси стовпця)	Індекси елементів	
			поточні	симетричні
Головна діагональ	від i=1; i < n	від j = 0; j< i	[i, j]	[j, i]
Допоміжна діагональ	від i=0 ; i < n -1	від j=0 ; j<n -1- i	[i, j]	[n-1-j, n-1-i]

Горизонтальна вісь	від i=0; i<nRow/2	від j=0; j< nCol	[i, j]	[nRow -1–i, j]
Вертикальна вісь	від i=0; i<nRow	від j = 0; j< nCol / 2	[i, j]	[i, nCol -1- j]

8.4.5 Видалення та вставка елементів матриці

Тут мається на увазі видалення стовпця або рядка матриці, бо видалити окремий елемент матриці ми не можемо, не порушивши прямокутну структуру.

Видалення або вставка рядка чи стовпця подібно видаленню або вставці елементу масиву. При видаленні потрібно зрушити наступні рядки вліво або стовпці вгору, на місце видаленого компонента, і зменшити відповідний розмір. При вставці потрібно зрушити наступні рядки вправо або стовпці вниз, звільнивши місце для компоненту що вставляється, і збільшити відповідний розмір матриці.

8.4.6 Сортування елементів матриці

Є два різновиди задачі сортування матриць.

Перший з них полягає у тому, що сортуються окремі частини матриці, незалежно від інших частин. До цього різновиду можна віднести наступні задачі:

– сортування кожного рядка або кожного стовпця незалежно від інших;

– сортування однієї з діагоналей матриці;

– сортування рядків або стовпців матриці за значенням першого елементу групи;

– сортування рядків матриці по сумі елементів рядка;

– сортування стовпців за сумою елементів стовпця.

Вирішення цих завдань не набагато складніше сортування масиву.

Для вирішення першого завдання із наведеного списку, сортування масиву слід застосувати послідовно для кожного рядка або стовпця.

У другій задачі діагональ розглядається як одновимірний масив, і важливо тільки не заплутатися в індексах, якщо потрібно відсортувати допоміжну діагональ.

При вирішенні третього завдання додаткова проблема виникає тільки в тому, що потрібно міняти місцями цілі рядки або стовпці. Для цього можна написати окрему функцію.

Для вирішення останнього завдання потрібно буде ще написати функцію обчислення суми елементів стовпця або рядка.

Другий різновид завдань сортування матриць полягає у тому, що в одній процедурі сортування беруть участь всі елементи матриці

Таке сортування забезпечує розташування елементів матриці за зростанням або спаданням в напрямку заданого способу обходу матриці.

Варіанти обходу елементів матриці можуть бути найрізноманітніші, але більшість з них має тільки навчальну користь. Деякі з них наведені нижче:

– по рядках зверху вниз і зліва направо, або знизу вгору і справа наліво, або змійкою;

– те ж саме по стовпцях;

– те ж саме по лініях паралельним головній або допоміжній діагоналях;

– кутом, вершина якого знаходиться на головній або допоміжній діагоналі.

Яким би не був спосіб обходу елементів матриці, саму процедуру сортування завжди можна звести до сортування масиву. Для цього слід переписати елементи матриці в масив. Потім масив відсортувати одним з відомих способів. Після цього елементи масиву потрібно знову переписати в матрицю, обходячи її за відповідним маршрутом.

Написати функцію переписування матриці в масив не складає труднощів, ця функція наведена на рисунку 8.4.

```
void matrixToArr(int matr[][COLS], int ar[], int nRow, int nCol){
   int n = 0;
   for(int i = 0; i< nRow; i++)
      for( int j = 0; j<nCol; j++)
         ar[n++]=matr[i][j];
}
```

Рисунок 8.4 – Функція переписування матриці у масив

Проблема сортування масиву нами теж вже вирішена. Можна використовувати будь-який з раніше розглянутих методів.

Таким чином, проблема сортування матриці зводиться до написання процедури, що забезпечує обхід елементів матриці у відповідності з необхідним маршрутом.

Підходи до створення процедур маршрутизації можуть бути різними.

Для простих варіантів сортування, коли напрям сортування збігається зі стовпчиками або рядками, можна використовувати розрахунковий спосіб визначення координат необхідного елемента.

Як приклад використання розрахункового способу, розглянемо маршрут обходу матриці по рядках зверху вниз і справа наліво для всіх рядків.

Припустимо, що матриця має 3 рядки і 4 стовпця, і обхід матриці відбувається по рядках зліва направо. Тоді елементи матриці умовно слід перенумерувати так, як показано на рисунку 8.5.

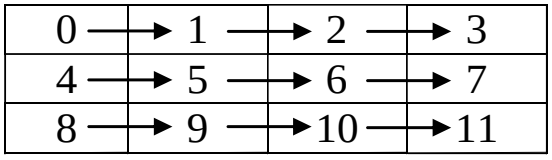


Рисунок 8.5 – Схема маршруту обходу матриці при сортуванні по рядках

Нехай номер елементу в масиві дорівнює 7. Координати цього елемента в матриці будуть такими.

Номер рядка буде 7 / 4 = 1.

Номер стовпця буде 7 % 4 = 3.

На рисунку 11.5 наведено функцію, що реалізує запис елементів масиву в матрицю і використовує наведені розрахункові формули.

Вихідними даними для процедури є:

– matr - матриця, яку потрібно заповнити по рядках;

– nCol - кількість стовпців матриці;

– nRow - кількість рядків матриці;

– ar - упорядкований масив, з якого беруться дані заповнення матриці.

У функції обчислюються номер рядка “r” та номер стовпця “c” матриці, що відповідають номеру “i” елемента в масиві, і у відповідності зі значеннями цих координат елемент масиву переписується в матрицю.

```
void fillMatrFromArHrzLine(int matr[][COLS], int nRow, int nCol, int ar[] ){
   for(int i = 0; i < nRow*nCol; i++){
      int r = i / nCol;
      int c = i % nCol;
      matr[r][c] = ar[i];
   }
}
```

Рисунок 8.6 – Функція переписування впорядкованого масиву до матриці по рядкам

У тих випадках, коли маршрут сортування складний, розрахунковий спосіб непридатний.

Для реалізації складних маршрутів сортування можна рекомендувати метод моделювання маршруту. Суть цього методу, полягає у створенні алгоритму, який для визначення номера рядка і стовпця моделює проходження маршруту сортування від початку до останнього елемента.

Алгоритм маршрутизації на кожному кроці повинен оцінювати положення поточного елемента і визначати, виходячи з цього, координати наступного. Таким чином, в процесі роботи алгоритму послідовно обчислюються координати всіх елементів маршруту, поки не буде досягнутий останній.

В якості прикладу можна навести процедуру маршрутизації для сортування квадратної матриці «Кутом, зверху - вниз - ліворуч, від початку головної діагоналі». Схема маршруту реалізованого в процедурі наведена на рисунку 8.7.

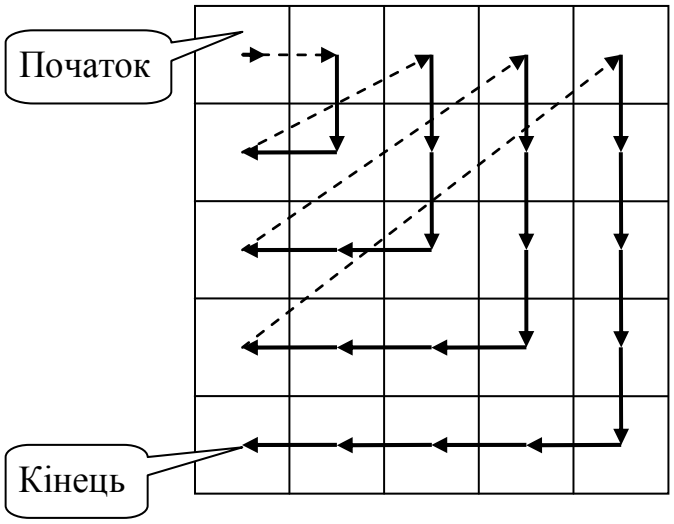


Рисунок 8.7 – Схема маршруту обходу матриці кутом униз і ліворуч, у напрямку головної діагоналі

Наведена схема реалізована у функції, що зображена на рисунку 8.8.

```
void fillMatrFromArCornerTopLeft(int matr[][COLS],int nRow, int nCol,int ar[]){
   int r=0, c=0; // Координати початку маршруту
   for(int i=0; i<nRow*nCol; i++){
      matr[r][c]=ar[i];
      if(c == 0){
         c = r + 1;
         r = 0;
      }
      else if( r < c)  r++;
      else c--;
   }
}
```

Рисунок 8.8 – Функція реалізації маршруту обходу матриці кутом униз і ліворуч, у напрямку головної діагоналі

Назви формальних параметрів у цій функції такі ж, як і в попередньому прикладі.

У тілі функції організовано цикл послідовного формування номерів стовпця і рядка для елементів відсортованого масиву. При кожному повторенні

циклу обчислюються координати чергової точки маршруту. Спосіб обчислення координат наступної точки змінюється при досягненні лівого краю матриці або головної діагоналі.

Якщо поточний елемент знаходиться у лівому стовпцю, то наступний елемент потрапляє на перший рядок.

При переході через головну діагональ змінюється правило зміни координат елементів матриці. Вище діагоналі збільшується на одиницю номер рядка (рух униз). Нижче діагоналі слід зменшувати на одиницю номер стовпця (рух ліворуч).

9 СТРУКТУРИ

9.1 Оголошення та ініціалізація структур

Структура - це тип даних, якому відповідає суміш елементів даних різних типів. Кожен з таких елементів, що входять до складу структури, називають полем. При роботі зі структурами оперують поняттями ім'я структури і ім'я поля. Імена структурам та їх полям присвоюються у відповідності зі стандартними правилами конструювання імен ідентифікаторів. Програміст може оперувати як з усією структурою, так і з окремими полями - це залежить від розв'язуваної задачі і операторів, що використовуються.

Для структури використовується її ім'я, а для ідентифікації її складових частин використовується складене ім'я, яке складається з імені структури та імені поля, розділених крапкою.

Для конструювання структури використовується поняття шаблону структури.

Синтаксис оголошення шаблону структури виглядає так, як показано на рисунку 9.1.

```
struct
        < тег шаблону структури >{
                <тип поля1>  <ім’я поля1>;
                < тип поля2>  <ім’я поля2>;
                 ...
                <тип поляN>  <ім’я поляN>;
        };
```

Рисунок 9.1 – Синтаксис оголошення шаблону структури

На цьому рисунку:

– struct – службове слово, що використовується для визначення структур;

– тег шаблону структури – ім’я, яким позначають структури даної конструкції, фактично відіграє роль типу структури, хоча поняття тип не використовується;

– { } – дужки, що обмежують перелік полів структури;

– тип поля та ім’я поля – стандартне оголошення кожного з полів.

В якості прикладу розглянемо шаблон структури для збереження результатів атестації студентів. Структура буде містити чотири поля - прізвище студента з ініціалами, назва групи, в якій навчається студент, середній бал поточної успішності та кількість незадовільних оцінок.

Нижче наведено цей шаблон.

```
struct stud {
    char fio[20];
    char gr[10];
    float srBall;
    int nzd;
};
```

Оголосити змінні, що мають відповідну структуру можна так:

```
struct stud s1, s2, s3;
```

Слово struct у оголошенні таких змінних використовувати не обов’язково:

```
stud s1, s2, s3;
```

Змінні можна оголошувати і одночасно із оголошенням шаблону:

```
struct stud {
    char fio[20];
    char gr[10];
    float srBall;
    int nzd;
} s1, s2, s3;
```

При оголошенні структур їх можна ініціалізувати. Список полів обмежується фігурними дужками, а поля розділяються комами:

```
stud  bestStud = {Гетьман Т.Г., КІ-131, 5.00, 0};
```

Полями структури можуть бути також і структури.

9.2 Операція присвоєння для структур

Значення полів можна визначати за допомогою оператора присвоєння:

```
s1.fio =’Петренко А.П.’;
s1.gr = ‘КС041’;
s1.srBall = ‘3.2’;
s1.nzd = ‘1’;
```

Оператор присвоювання можна використовувати і для запису в цілому. У прикладі, наведеному нижче, всім полям запису s2 будуть присвоєні значення полів запису s1.

```
s2 = s1;
```

9.3 Звертання до полів структури

Програміст може оперувати як з усією структурою, так і з окремими полями - це залежить від розв'язуваної задачі і операторів, що використовуються.

Для структури використовується її ім'я, а для ідентифікації її складових частин використовується складене ім'я, яке складається з імені структури та імені поля, розділених крапкою.

При організації вводу-виводу варто мати на увазі, що безпосередньо структуру ввести або вивести, без використання спеціально написаних процедур, не можна. Можна вводити або виводити окремі поля структури, або елементи полів, якщо поля є теж структурами.

Введення і виведення структур схоже на введення та виведення матриць, але різниця тут у тому, що стовпчики можуть мати різні типи, і доступ до елементів різних стовпців проводиться не за індексом, а за назвою.

9.4 Масиви структур

Із структур однакового тегу може бути створений масив, точно так само як з даних інших типів. Наприклад, якщо ми хочемо мати результати атестації для всіх студентів, можна створити масив, що містить дані про кожного студента у вигляді структури і частково його проініціалізувати:

```
#define FIO_SIZE 15
#define GR_SIZE 7
#define AR_SIZE 30
struct stud {char fio[FIO_SIZE];
        char gr[GR_SIZE];
        float srBall;
        int nzd;
};
stud ar[AR_SIZE]={
              {"Чуб П.П.","КС051", 4.55, 0},
            {"Гай А.Л.","КС052", 1.55, 3},
            {"Кит А.В.","КС051", 2.45, 1},
            {"Кіт С.В.","КС052", 1.25, 3},
            {"Бут К.Л.","КС052", 4.65, 0}
};
int size=5;
```

При зверненні до елементу масиву структур також використовуються індекси та квадратні дужки. При цьому слід враховувати, що елементом масиву є структура і, тому, квадратні дужки ставляться після імені масиву, а потім, через крапку, записується ім'я поля.

Наприклад, вище наведений масив можна доповнити ще однією структурою.

```
ar[size].fio =’Петренко А,П,’;
ar[size].gr = ‘КС041’;
ar[size].srBall = ‘3.2’;
ar[size].nzd = 1;
```

Якщо полем структури є масив, то у зверненні до елементу масиву квадратні дужки вже будуть після імені поля. Так, наприклад, якщо ми хочемо у вищенаведеному масиві структур змінити прізвище студента «Гай» на «Гак», то слід написати так:

```
ar[1].fio[2] = 'к';
```

9.4.1 Сортування масивів структур

Сортування масиву структур проводиться так само, як і сортування масиву чисел, тобто структури порівнюються і, якщо необхідно, переставляються. Єдина проблема тут полягає в тому, що способів порівняння записів, а, отже, і варіантів сортування, може бути багато, і кожен з них може знадобитися. Так для масиву, розглянутого в попередньому прикладі, записи можна сортувати за прізвищем студента або за середнім балом, за кількістю незадовільних оцінок і таке інше. Більш того, можливі і складніші випадки сортування, наприклад, по групі, а в межах однієї групи на прізвище студента.
У такій ситуації доводиться для кожного варіанту сортування писати функції сортування, які будуть відрізнятися тільки способом порівняння структур.

9.5 Декларація іменування типу typedef

До складу мови С++ входить спеціальна декларація typedef, яка дає програмісту змогу надавати власні імена типам даних програми. Синтаксис перейменування типів такий:

typedef тип користувацьке_ім’я_типу;

Декларація typedef не створює нового типу, а тільки надає можливість звертатися до існуючого типу за допомогою нового імені. За допомогою цієї декларації можна змінювати довгі імена типів даних на більш короткі і зручні:

```
typedef unsigned int UINT;
UINT a = 23, b;
```

В цьому прикладі ми перейменували тип даних unsigned int в UINT. Після цього нове ім’я (UINT) можна використовувати в програмі так само, як і старе (unsigned int).

Всередині typedef користувацьке ім’я типу записується в тому місці, де для звичайного оголошення вказується ім’я змінної. Наступне оголошення можна використовувати для оголошення символьних рядків:

```
typedef char STRING[ 80 ];
STRING s = "a string";
cout << s << endl;
```

Використання декларації typedef сприяє підвищенню наочності імен типів даних та скорочує їх запис. Це передусім стосується таких типів даних, як структури, для яких typedef можна використовувати одночасно з оголошенням шаблону. Наприклад:

```
typedef struct student {
char name[ 40 ];
int number;
int rating;
} STUDENT;
STUDENT s = { "James Bond", 12, 80 };
```

Крім наочності та простоти іменування типів декларація typedef дає програмістові змогу створювати машино незалежні типи даних. Насамперед це стосується тих типів, параметри яких залежать від апаратних особливостей комп'ютера. Якщо такому типові надати певне typedef – ім'я і використовувати це ім'я всюди в тексті програми, то в новому середовищі достатньо змінити рядок декларації typedef, щоб встановити нові параметри для перейменованого типу.

10 БАЗОВІ ПОНЯТТЯ ПРО ВКАЗІВНИКИ

10.1 Оголошення вказівників

Покажчик або вказівник – це особливий тип даних, значенням якого є адреса певного байту оперативної пам'яті. Цей тип даних найчастіше використовується у системному програмуванні, але і прикладні програмісти успішно використовують цей інструмент. Слід сказати, що не усі мови програмування надають можливість використовувати покажчики, але у таких популярних мовах програмування як С та Паскаль робота з покажчиками можлива. Більш того, можна сказати, що покажчики – це візитна картка мови С.

Вважається, що використання покажчиків дає змогу скоротити текст програми та підвищити її ефективність. Але тут є і зворотна сторона. Покажчики – це не зовсім простий у використанні засіб програмування і часто стають джерелом помилок, виявити які у програмі дуже складно. Тому у деяких мовах прикладного програмування (наприклад, Java) принципово відмовилися від покажчиків.

Оголошення вказівника має вигляд, представлений на рисунку 10.1.

<тип даних> *<ім'я покажчика> ;

Рисунок 10.1 – Синтаксис оголошення покажчика

У цьому оголошенні тип даних визначає тип елементу програми, адресу якого може зберігати даний покажчик. Це може бути будь який допустимий у мові тип, простий або складений

Ім'я покажчика – це ідентифікатор написаний за правилами запису імен у мові С.

* – це ознака того, що наступне ім'я є покажчиком.

Приклад оголошення покажчика з ім'ям px, на дані типу int наведено нижче:

```
int *px ;
```

Слід мати на увазі, що символ * відноситься не до типу, а до імені покажчика, тому в разі оголошення декількох покажчиків символ * треба ставити перед кожним іменем. Нижче наведено приклад оголошення двох покажчиків pw, pz одночасно із оголошенням простої змінної pv:

```
int  *pw, *pz, pv ;
```

Під час оголошення покажчика його можна ініціалізувати константою цілого типу, але цю константу слід явно привести до типу покажчика. У прикладі наведеному нижче покажчику pd на дані типу double присвоюється значення 1024 (нагадаємо, що значення покажчика – це адреса даних):

```
double  *pd = (double*)1024;
```

Зверніть увагу, у конструкції, що використовується для приведення до типу покажчика символ * відноситься вже до назви типу даних.

Покажчик можна проініціалізувати і адресою деякої, вже оголошеної змінної. Для отримання адреси змінної використовується операція визначення адреси - &. Нижче наведено приклад такої ініціалізації:

```
double  z, *pz = &z;
```

У цьому прикладі оголошені проста змінна z, і покажчик pz, який проініціалізований адресою змінної z. Таким чином вказівник pz містить адресу змінної z.

Якщо покажчик не проініціалізовано, то його значенням буде «сміття», тобто будь яка адреса. Для того, щоб не мати справи із «сміттям», у тих випадках, коли невідомо, як ініціалізувати покажчик, йому присвоюють значення NULL. Вважається, що покажчик, значенням якого є NULL, ні на що не посилається і його називають порожнім покажчиком. Нижче наведено приклад такої ініціалізації:

```
int  *pw = NULL ;
```

Покажчики можна оголошувати з кваліфікатором const. При цьому можливі три варіанти оголошення.

Перший варіант полягає у тому, що кваліфікатор const забороняє змiювати дані на які він посилається. Для реалізації цього варіанту слово const має бути записано перед типом покажчика:

```
const int z = 100;
const int *ptc = &z;
```

Зверніть увагу, ініціалізувати покажчик на константні дані можна тільки адресою константи.

Покажчики на константні дані часто використовуються в оголошеннях параметрів функцій.

Другий варіант використання кваліфікатора const передбачає незмінність самого покажчика. Для реалізації цього варіанту слово const має бути записано перед іменем покажчика:

```
int  *const cpt = &z;
```

Оголошений таким чином покажчик cpt не може змінювати свого значення, а змінна z на яку він посилається змінюватися може. Такі покажчики називають константними.

Третій варіант використання кваліфікатора const передбачає незмінність як самого покажчика, так і даних, на які він посилається. Для реалізації цього варіанту слово const має бути записано двічі, перед типом та перед іменем покажчика:

```
const int z = 100;
const int *const ptc = &z;
```

10.2 Звертання до даних через вказівники

Якщо відома адреса якихось даних, то мабуть можна отримати і значення цих даних. Операцію отримання даних через їх адресу називають операцією розадресації і позначають знову ж таки символом *, що ставиться перед ім’ям покажчика. Звертання до даних через покажчик можна використовувати всюди,

де можна звертатися до даних через ім’я змінної.

Нехай оголошено зміну z і проініціалізовано покажчик на неї pz:

```
double  z, *pz = &z;
```

У цьому випадку вирази

```
z = 12.37;  z  += 2.5;  z *= 2;
```

можна записати і так:

```
*pz = 12.37;  *pz  += 2.5;  *pz *= 2;
```

Як бачимо з цього прикладу, вирази &z та pz позначають адресу змінної z, а вирази z та *pz позначають поточне значення змінної z.

Працюючи з покажчиками у програмах на мові С слід пам’ятати, що ніякого контролю за значеннями покажчиків, що використовуються для доступу до даних нема. Ви можете звернутися до «своїх даних» з будь якою адресою, але отримаєте невідомо що. Ще гірше, якщо ви зміните значення цих «своїх даних». Це може призвести до катастрофічних наслідків для вашої програми.

10.3 Адресна арифметика

Над покажчиками можливі такі операції:

– присвоєння;
– порівняння;
– збільшення/зменшення;
– віднімання.

За допомогою операції присвоєння покажчику можна присвоїти значення адресної константи, адресу якоїсь змінної або результат обчислення виразу, що знаходиться праворуч від знака присвоєння. Необхідною умовою операції присвоєння для покажчиків є однаковість базових типів вказівника і значення, що йому присвоюється.

Для порівняння покажчиків можна використовувати звичайні операції порівняння. Найчастіше з цих операцій використовуються операції == та !=.

До значення покажчиків можна додавати (або віднімати) цілі числа. При цьому значення покажчика збільшується на величину числа, що додається, помножену на кількість байтів, що займає у пам'яті елемент даних відповідного типу. У результаті виконання наведеного нижче прикладу значення покажчика ptn буде на 20 більшим, ніж значення покажчика ptz. Тут мається на увазі що елемент даних типу int займає 4 байти.

```
int z = 100;
int  *ptn, *ptz = &z;
ptn = ptz +5 ;
```

Таким чином, вираз, у якому збільшується або зменшується значення покажчика, наприклад, на величину k, формує адресу елементу даних, розташованого на k елементів правіше, або лівіше у разі операції віднімання.

Для зміни значень покажчиків можна також застосовувати операції інкременту та декременту. Операція інкременту зміщує покажчик до наступного елементу, а декременту – до попереднього. Найчастіше такі операції використовуються під час роботи з масивами.

Операція віднімання, що виконується над двома покажчиками повертає кількість елементів базового, типу що може розміститися між адресами, на які вказують перший та другий операнди операції.

10.4 Вказівники void та типізація вказівників

У мові С можна використовувати покажчики, які не пов'язані з якимось конкретним типом і сумісні покажчиками на будь які типи даних. Використання безтипових покажчиків дозволяє підвищувати ефективність створюваних програм. Безтипові покажчики оголошуються із ключовим словом void:

```
void *pz;
```

Хоча значення без типових покажчиків можна прямо присвоювати вказівникам на будь які типи даних, усе ж доцільно у таких присвоєннях використовувати явне приведення типу, бо це підвищує надійність програм.

Розглянемо приклад, який дозволяє отримати доступ до двох половин числа типу long (long удвічі більший за int).

```
ulong number = 0x12AB34DC;
uint  part1, part2;
void *p= &number;
part1=*(uint*)p;
part2=*((uint*)p+1)
```

У цьому прикладі ми розрізали число number типу long на числа part1та part2 типу int.

11 ВИКОРИСТАННЯ ВКАЗІВНИКІВ ДЛЯ РОБОТИ З МАСИВАМИ

11.1 Звертання до елементів масиву через вказівники

Можливість використання покажчиків для звертання до елементів масивів пов'язана з тим, що ім'я масиву є константним покажчиком на перший елемент масиву. Таким чином, якщо маємо оголошення масиву:

```
int a[10];
```

то звертання *a дає той самий результат, що й a[0]. Звертання *(a+3) дасть той самий результат, що й a[3].

11.2 Реалізація простих алгоритмів обробки масивів з використанням вказівників

Нижче наведено приклад створення масиву для збереження чисел Фібоначчі, який розглядався раныше. Два перших числа Фібоначчі рівні 0 та 1, а кожне наступне дорівнює сумі двох попередніх. У програмі перші 2 елементи масиву заповнюються до циклу, решта - в циклі.

```
int a[10];
*a = 0;
   *(a+1) = 1;
   for (i=2 ; i<10; i++)
      *(a+i) = *(a + i – 2) + *(a + i – 1);
```

У цьому прикладі доступ до елементів масиву здійснюється за допомогою покажчиків, але самі покажчики формуються за допомогою операцій адресної арифметики з використанням адреси масиву та індексу елементів і, що формується у циклі for.

Але частіше для обробки елементів масиву використовуються додаткові покажчики. За звичай один такий покажчик вказує на кінець масиву, а другий,

поточний, формується у циклі for.

Нижче наведено такий варіант формування масив чисел Фібоначчі.

```
int n=10, a[n],  *pCurrent, *pEnd;
pEnd = a+n;
pCurrent = a;
*pCurrent++ = 0;
*pCurrent++ = 1;
for ( ; pCurrent <pEnd; pCurrent ++)
      * pCurrent  = *( pCurrent – 2) + *( pCurrent – 1);
```

11.3 Використання вказівників для роботи з рядками символів

Так само як и у випадку числового масиву, ім'я символьного масиву є константним покажчиком на перший символ. Тому для обробки символьних рядків теж можна використовувати покажчики, так само, як і для масивів чисел. Обробка навіть дещо спрощується завдяки тому, що кожний рядок закінчується спеціальним символом '\0'.

У якості прикладу розглянемо функцію пошуку позиції, з якої комбінація символів sub знаходиться у рядку str:

```
int posInStr( char *str,  char *sub){
   char *p1, *p2, *p;
   for(p1=str; ; p1++){
      for(p=p1,p2=sub; *p != '\0' && *p2 != '\0' && *p == *p2 ; p++, p2++);
      if(*p2=='\0')return p1-str;
      if(*p=='\0')return -1;
   }
}
```

В якості параметрів до цієї функції передаються покажчики на рядок і

комбінацію символів.

Зовнішній цикл for забезпечує послідовне переміщення по символах рядка.

Внутрішній цикл, який не має тіла, забезпечує порівняння символів комбінації із символами рядка, починаючи з поточного. Він має три умови завершення.

Якщо поточні символи комбінації і рядка не співпадають, відбувається перехід до зовнішнього циклу.

Якщо під час цього порівняння знайдено кінець комбінації то це означає, що комбінація входить до складу рядка, і функція повертає відстань між поточним символом рядка і його початком.

Якщо знайдено кінець рядка, то повертається ознака того, що комбінація не входить до рядка.

11.4 Бібліотечні функції для роботи із символами та символьними рядками

Стандартні бібліотеки мови С надають користувачеві можливість використовувати ряд функцій для опрацювання символьних рядків, а також функцій перетворень рядків символів у числа та зворотних перетворень. Ці функції докладно описані у підручнику [4], стр.152-154. Ознайомтесь із цими функціями і перевірте їх роботу. Майте на увазі, у підручнику є помилка, яку не важко помітити.

11.5 Масиви покажчиків на рядки символів

У практиці програмування зустрічаються задачі в яких доводиться працювати з масивами рядків символів. Прикладом такого масиву може бути список студентів групи. Особливість таких масивів полягає у тому, що майже кожний елемент має свою довжину. Якщо оголошувати такий масив як двовимірний (матриця символів), то ширину такої матриці доведеться вибирати виходячи з найдовшого прізвища. Це призведе до нераціонального використання

пам’яті.

Значно краще створити одновимірний масив покажчиків на прізвища і з ним працювати. Нижче наведено приклад, у якому створюється масив покажчиків на рядки. Після ініціалізації масив упорядковується і виводиться на консоль.

```
//Створення масиву покажчиків на рядки
const char *arrPtrToFio[]={"Жванецький М.В.","Вовк А.П.","Карась П.С."};
//Сортування покажчиків методом вибору
for(int i=0; i<2;i++){
   for(int j=1;j<3;j++){
      //Порівняння прізвищ
        if(strcmp(arrPtrToFio[i], arrPtrToFio[j])>0){
           // Обмін покажчиків у масиві
           const char *p=arrPtrToFio[i];
         arrPtrToFio[i]=arrPtrToFio[j];
         arrPtrToFio[j]=p;
      }
   }
}
//Виведення на консоль
for(int i=0;i<3;i++){
   cout<<arrPtrToFio[i]<<endl;
}
```

Зверніть увагу, під час сортування обмін відбувається тільки між покажчиками, а рядки залишаються на місцях.

Ілюстрації до прикладу можна знайти у підручнику [4], стр.159-160.

11.6 Покажчики на структури

Змінні, яким відповідають структури, розглядаються у мові С як звичайні змінні. До цих змінних можна застосовувати операцію &, яка поверне адресу першого байту ділянки оперативної пам’яті, що займає відповідна структура.

Можна оголошувати покажчик на структуру, використовуючи назву шаблону структури в якості базового типу покажчика.

Використовуючи операції розадресації слід для структур слід пам’ятати, що операція «крапка» має вищий пріоритет, ніж операція «зірочка». Тому можна використовувати такий вираз для отримання першого символу рядка name, що є полем структури: *bestStud.name.

До полів структури можна звертатися і використовуючи покажчик на цю структуру. Але у цьому разі замість операції «крапка» слід використовувати операцію ->. Слід зазначити, що Qt автоматично змінює операцію «крапка» на операцію -> у разі помилки з вибором операції.

Покажчики на структури можна використовувати і в якості полів структури, у тому числі і у структурах того самого типу. Такі поля є невід’ємною частиною динамічних інформаційних структур – списків та дерев.

12 ВИКОРИСТАННЯ ВКАЗІВНИКІВ ДЛЯ РОБОТИ З ДИНАМІЧНИМИ СТРУКТУРАМИ ДАНИХ

12.1 Стандартні функції динамічного виділення пам'яті

Динамічне виділення пам'яті необхідно для того, щоб пам'ять виділялася тільки тоді, коли нам треба, і при необхідності звільнялася.

Розглянемо, наприклад, ситуацію, коли необхідно сформувати одновимірний масив з наперед невизначеним числом елементів. У цьому випадку, якщо ми впевнені, що кількість елементі ніколи не буде більшою за 100, можна визначити масив таким чином - int y[100]. Але випадку, коли кількість елементів буде менше 100, то частина пам’яті, що була виділена для масиву буде витрачена зайво. Ще гірше буде у випадку коли кількість елементів масиву буде більше 100, тоді програма буде працювати некоректно.

Саме для коректного вирішення таких ситуацій і використовується динамічне виділення пам’яті (виділяється стільки скільки потрібно).

Функції malloc() та free() є основними функціями для роботи з пам’ятю в С.

Функція malloc() виділяє пам’ять. Це означає, що при кожному виклику malloc() виділяється порція вільної пам’яті.

Функція free() звільняє пам’ять. При кожному виклику free() виділена пам’ять повертається в систему.

Кожна програма, яка використовує ці функції повинна містити заголовочний файл stdlib.h в якому зберігаються прототипи цих функцій.

Прототип функції malloc() вглядає так:

void* malloc() (unsigned число_байт).

Функція повертає вказівник типу void, що означає, що його можна привести до будь-якого типу. Після успішного виклику malloc() поверне вказівник на перший байт області пам’яті виділеногї з “кучі”. В протилежному

випадку функція поверне значення NULL.

Для визначення точного числа байтів потрібного для кожного типу даних слід використати функцію sizeof().

Прототип функції free()вглядає так:

void free(void*p);

Дуже важливо викликати free() з правильним значенням аргументу, тому що інакше виклик функції приведе до непередбачуваних дій комп'ютера.

Наступна коротка програма виділяє пам'ять для 10 цілих чисел, друкуючи їх значення і звільняючи пам'ять для подальшого використання:

```
int*p, t, n=10;
p=(int*)malloc(n*sizeof(int));
if(!p) /* чи достатньо пам'яті */
        cout<<"out of memory\n";
else {
        for (t=0; t<n; t++)
                *(p+t) = t;
        for (t=0; t<n; t++)
                cout<<*(p+t)<<endl;
        free(p);
    }
```

Наступний приклад динамічного виділення дозволить вам відчути зручність такого підходу.

```
int  n;
cout<<"Введіть кількість чисел\n";
cin>>n;
// виділення місця для одномірного масиву
float *p;
```

```
p=(float*)malloc(n*sizeof(float));
if(!p) /* чи достатньо пам’яті */
      cout<<"out of memory\n";
else {
   float avg=0;
   for(int i=0; i<n;i++){
      cout<<i+1<<"-е число = ";
      cin >> p[i];
      avg+=p[i];
   }
   cout<<"середнє = "<<avg/n<<endl;
   free(p); /*звільнення*/
}
```

Наступний більш складний приклад демонструє використання динамічного виділення пам’яті для двомірного масиву, який широко використовується в програмуванні

```
int **p; /*вказівник на масив вказівників*/
int i, j, maxI, maxJ;
cout<<"Введіть кількість рядків : ";
cin>>maxI;
cout<<"Введіть кількість стовпчиків : ";
cin>>maxJ;
//Виділяємо пам'ять для масиву вказівників на рядки
p=(int**)malloc(maxI * sizeof(int*));
//формуємо масив вказівників на рядки
if(p==NULL){
   cout<<"allocation error";
```

```
    exit (1);
}
for(i=0; i < maxI; i++){
    //Виділяємо пам'ять для і-го рядка
    p[i] =(int*) malloc(maxJ* sizeof(int));
    if(p[i] == NULL){
        cout<<"allocation error";
        exit (1);
    }
}
//Память для масиву виділено, заповнюємо числами
for(i=0; i < maxI; i++)
    for(j=0;j<maxJ; j++)
        *(p[i]+j)=10*i+j;
//Виводимо числа на консоль
for(i=0; i < maxI; i++){
    for(j=0;j<maxJ; j++)
        cout<<*(p[i]+j)<<"  ";
    cout<<endl;
}
//звільнення пам'яті
for(i=0; i < maxI; i++)
    free( p[i]);
free(p);
```

12.2 Динамічні списки

Односпрямований лінійний список - це структура впорядкована даних, що складається з окремих елементів. Порядок елементів забезпечується тим, що кожний елемент має покажчик на наступний, а у останнього елементу значення покажчика на наступний елемент дорівнює NULL.

Односпрямований кільцевої список - це список, де покажчиком на наступний елементом для останнього елементу є покажчик на початок списку.

Двоспрямований лінійний список – це список, я якому кожний елемент мае покажчик не тільки на наступний елемент, а і на попередній.

Ідентифікується список покажчиком на перший елемент.

Елементами списку є структури, які мають дві складові - інформаційну і службову. Інформаційна складова – це поля з даними, які зберігаються у списках. Службова складова – це поля, що містять вказівники на сусідів по списку.

Таким чином, для того щоб працювати зі списком перш за все треба визначити шаблон структури для його елементів та змінну, типу вказівник на визначену структуру, яка буде містити вказівник на перший елемент списку.

Найпростіша структура для односпрямованого лінійного списку може виглядати так:

```
struct element{
   char name[20];
   element *next;
};
```

Визначення вказівника на початок списку може виглядати так:

```
element* next=NULL; //Вказівник на список
```

Далі потрібно створити функції для додавання елементів у список, їх вилучення, виведення списку для перегляду та інші.

12.3 Простіші функції для роботи із списками

12.3.1 Функція створення нового елементу для списку

Створюючи нові елементи списку слід не забувати, що це динамічна структура і кожен елемент потребує виділення пам’яті із кучі.

```
element* newElement(){
   cout<<"Введіть name  ";
   //Виділяємо пам'ять для нового елементу
   element* p =(element*) malloc(sizeof(element));
   if(p == NULL){
      cout<<"allocation error";
      exit (1);
   }
   p->next=NULL;
   cout<<"Введіть name  ";

   cin.getline(p->name,20);
   return p;
}
```

12.3.2 Функція для вставки елемента у початок списку

```
void addFirst(element* &pList, element* pNew){
   if(pList==NULL)pList=pNew;
   else {
      pNew->next=pList;
      pList=pNew;
   }
}
```

12.3.3 Функція для вставки елемента у кінець списку

```
void addLast(element* &pList, element* pNew){
    if(pList==NULL)pList=pNew;
   else {
      element* p = pList;
      while(p->next!=NULL)p=p->next;
      p->next=pNew;
   }
```

12.3.4 Функція виведення списку на консоль

```
void showList(element* pBegin){
   if(pBegin==NULL){
      cout<< "List is empty"<<endl;
      system("pause");
   }
   cout << "This is the list" <<endl;
   element *p = pBegin;
   for(int i=1; p!=NULL; i++,p=p->next){
      cout.width(3); cout<<i; cout<<". ";
      cout<<p->name<<endl;
   }
}
```

12.3.5 Функція вилучення елемента

```
bool delElement(element* &begin, char name[]){
   if(begin==NULL){
      cout<< "List is empty"<<endl;
      system("pause");
```

```
    }
    if(!strcmp(begin->name,name)){
        element* deleted = begin;
        begin = begin->next;
        free(deleted);
        return true;
    }
    element* prev=begin;
    for(element* p = begin->next; p!=NULL;prev=p, p=p->next){
        if(!strcmp(p->name,name)){
            prev->next=p->next;
            free(p);
            return true;
        }
    }
    cout << "Not found" <<endl;
    return false;
}
```

12.3.6 Функція звільнення пам’яті, що займав список

```
void freeMemo(element* &pBegin){
    while(pBegin!=NULL){
      delElement(pBegin,pBegin->name);
    }
}
```

ЛІТЕРАТУРА

1. Бородкіна І. Л. Теорія алгоритмів: посіб. для студентів вищих навчальних закладів / І. Л. Бородкіна, Г. О. Бородкін. — Київ : Національний університет біоресурсів та природокористування України, 2018. — 231 с

2. Глибовець М. М. Основи комп'ютерних алгоритмів / М. М. Глибовець. — Київ : Видавничий дім «КМ Академія», 2003. — 452 с.

3. Горлова Т. М. Теорія алгоритмів: конспект лекцій [для студентів напряму підготовки 6.050101 «Комп'ютерні науки» денної та заочної форм навчання] [Електронний ресурс] / Т. М. Горлова, К. Є. Бобрівник, Н. В. Лімaнська. — Київ : НУХТ, 2015. — 95 с. — Режим доступу: http://library.nuft.edu.ua/ebook/file/ M51.21.pdf

4. Грицюк Ю.І., Рак Т.Є. Програмування мовою С++ : навч. посіб. Львів: Вид-во Львівського ДУ БЖД, 2011. 292 с. URL: http://xn--e1ajqk.kiev.ua/wpcontent/uploads/2019/12/Griczyuk-C.pdf

5. Кублій Л. І. Алгоритмізація та програмування. Практикум : навч. посіб. [для здобувачів ступеня бакалавра за спеціальністю 122 «Комп'ютерні науки»] [Електронний ресурс] / Л. І. Кублій. — Київ : КПІ ім. Ігоря Сікорського, 2019. — 209 с. — Режим доступу: ela.kpi.ua/handle/123456789/28216

6. Кузнєцов М.С. Процедурне програмування з використанням мови С: навч. посіб. Дніпропетровськ : НМетАУ, 2005. 84 с.

7. Селін О. Алгоритми та структури даних: конспект лекцій [Електронний ресурс] / О. Селін. — Київ : Національний технічний університет України "Київський політехнічний інститут", 1998. — Режим доступу: http://mmsa.kpi.ua/sancho/ASD_HTM/index.html

8. Сліпченко В. Г. Структури даних мови Паскаль / В. Г. Сліпченко, Н. В. Ревинська. — Київ : КПІ, 1998. — 200 с.

9. Ткачук В. М. Алгоритми і структури даних: навч. посіб. / В. М. Ткачук. — ІваноФранківськ: Вид-во Прикарпатського національного університету ім. Василя Стефаника, 2016. — 286 с.

10. Шпак З.Я. Програмування мовою С : навч. посіб. Львів : Оріяна-Нова, 2006. 431 с.