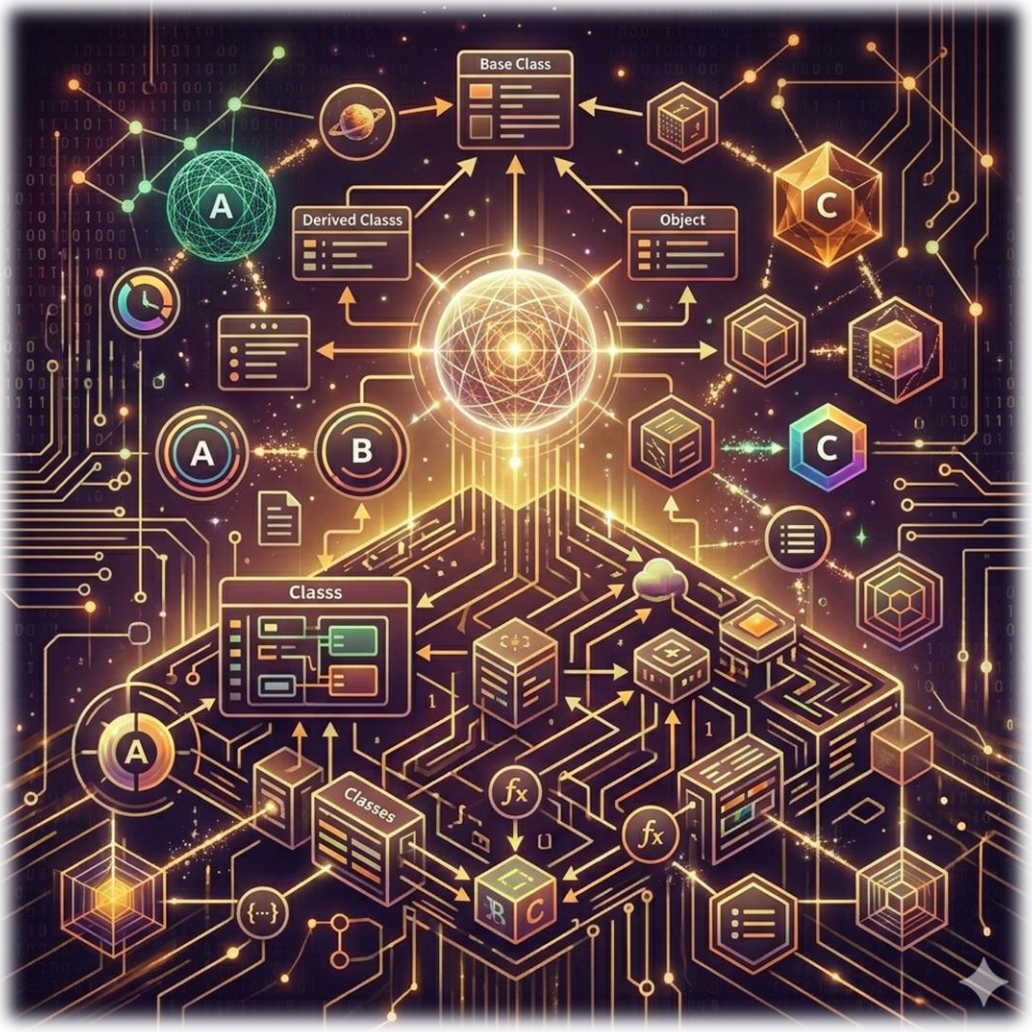
МИКОЛАЇВСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ МЕНЕДЖМЕНТУ
КАФЕДРА ЕКОНОМІЧНОЇ КІБЕРНЕТИКИ,
КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

конспект лекцій

для змішаного навчання здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спеціальності F3(122) «Комп'ютерні науки» денної та заочної форми здобуття вищої освіти



Миколаїв
2025

УДК 004.43:004.42
О-13

Друкується за рішенням науково-методичної комісії факультету менеджменту Миколаївського національного аграрного університету (протокол № 1 від 28 серпня 2025 року)

Укладачі:

С. І. Ємельянов – PhD, старший викладач кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

С. І. Тищенко – в.о. завідувача кафедри, к.п.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. Ю. Пархоменко – к.ф.-м.н., доцент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. О. Жебко – асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету;

О. Є. Богатєнкова – асистент кафедри економічної кібернетики, комп'ютерних наук та інформаційних технологій Миколаївського національного аграрного університету

Рецензенти:

Махровська Н. А. – кандидат фізико-математичних наук, доцент кафедри теорії й методики природничо-математичної освіти та інформаційних технологій Миколаївський обласний інститут післядипломної педагогічної освіти

Садовий О. С. - канд. техн. наук, доцент, завідувач кафедри агроінженерії Миколаївського національного аграрного університету

О-13 **Об'єктно-орієнтоване** програмування : конспект лекцій для змішаного навчання здобувачів першого (бакалаврського) рівня вищої освіти ОПП «Комп'ютерні науки» спеціальності F3(122) «Комп'ютерні науки» денної та заочної форми здобуття вищої освіти / уклад. С. І. Ємельянов, С. І. Тищенко, О. Ю. Пархоменко, О. О. Жебко, О. Є. Богатєнкова. Миколаїв : МНАУ, 2025. 200 с.

УДК 004.43:004.42

ЗМІСТ

ПЕРЕДМОВА
ЗМІСТОВИЙ МОДУЛЬ 1
ІСТОРІЯ ТА ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ С++
Тема 1.1. Основні поняття мови програмування С++. Типи даних. Структура програми на С++
Тема 1.2. Розгляд опису змінних та різних типів даних, включаючи цілі, дійсні, символьні та логічні типи даних
Тема 1.3. Вивчення різних операцій та їх використання в мові С++
ЗМІСТОВИЙ МОДУЛЬ 2
БАЗОВІ АЛГОРИТМІЧНІ СТРУКТУРИ У МОВІ С++
Тема 2.1 Розгляд конструкцій умовних операторів, розгалуження та циклів у програмуванні
Тема 2.2. Вивчення вказівників та їх ролі в мові С++. Робота з масивами
Наприклад
ЗМІСТОВИЙ МОДУЛЬ 3
РОЗШИРЕНІ ОПЕРАЦІЇ З ДАНИМИ У МОВІ С++
Тема 3.1. Дослідження функцій, їх створення та використання. Робота з посиланнями
Тема 3.2. Вивчення роботи з багатовимірними масивами та методи динамічного виділення пам'яті для масивів
Тема 3.3. Дослідження операцій та роботи з рядками в мові програмування С++
ЗМІСТОВИЙ МОДУЛЬ 4
ВСТУП ДО ООП ТА БАЗОВІ КОНСТРУКЦІЇ КЛАСІВ
Тема 4.1. Вступ до об’єктно-орієнтованого програмування. Поняття класу та об’єкту
Тема 4.2. Конструктори та деструктор класу.
ЗМІСТОВИЙ МОДУЛЬ 5
РОБОТА З ОБ'ЄКТАМИ, МАСИВАМИ ТА КОМПОЗИЦІЯМИ КЛАСІВ
Тема 5.1. Робота із масивами об’єктів
Тема 5.2. Види класів. Вкладені класи
ЗМІСТОВИЙ МОДУЛЬ 6
ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ ТА ОДИНАРНЕ УСПАДКУВАННЯ
Тема 6.1. Перевантаження операторів
Тема 6.2. Успадкування. Одинарне успадкування
ЗМІСТОВИЙ МОДУЛЬ 7

МНОЖИННЕ УСПАДКУВАННЯ ТА МОДУЛЬНА ОРГАНІЗАЦІЯ КОДУ .....
Тема 7.1. Множинне успадкування. Механізми успадкування декількох базових класів.....
Тема 7.2. Оголошення класів у заголовочних файлах .....
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....

ПЕРЕДМОВА

Конспект лекцій розроблений для здобувачів першого (бакалаврського) рівня 3-го року денної форми здобуття вищої освіти з дисципліни «Об’єктно-орієнтоване програмування» спеціальності 122 «Комп’ютерні науки».

Конспект лекцій містить матеріал, необхідний для формування у студентів знань і практичних навичок створення програмного забезпечення з використанням об’єктно-орієнтованої парадигми.

Основна мета дисципліни – навчити студентів принципам побудови програмних систем на основі понять класів, об’єктів, спадкування, інкапсуляції та поліморфізму, а також методам проектування та реалізації програм із використанням сучасних мов програмування.

У результаті вивчення курсу студенти здобудуть уміння моделювати предметні області, розробляти масштабовані й підтримувані програмні рішення, застосовувати шаблони проектування та принципи SOLID. Це дозволить їм створювати якісне програмне забезпечення, ефективно працювати у командах розробників і підготує до подальшого вивчення більш поглиблених технологій програмування та розробки програмних систем.

Цей матеріал рекомендується використовувати для вивчення дисципліни «Об’єктно-орієнтоване програмування».

ЗМІСТОВИЙ МОДУЛЬ 1
ІСТОРІЯ ТА ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ С++

Тема 1.1. Основні поняття мови програмування С++. Типи даних. Структура програми на С++

Історія виникнення та особливості мови С++

Мова програмування С була розроблена у 1972 році Денісом Рітчі у Bell Telephone Laboratories з метою написання нею операційної системи UNIX. Але, незважаючи на те, що мова С була розроблена для написання системного програмного забезпечення, наразі вона досить часто використовується для написання прикладного програмного забезпечення. Мова С здійснила великий вплив на інші мови програмування, особливо на об'єктно-орієнтовану мову програмування С++, яка спочатку проектувалася, як розширення для С, а також на Java та C#, які запозичили у мови С синтаксис.

Мова програмування С – це мінімалістична мова програмування. Серед її головних цілей: можливість прямолінійної реалізації компіляції, використовуючи відносно простий компілятор, забезпечення низькорівневого доступу до оперативної пам'яті і досить невелика динамічна підтримка. У результаті, код С придатний для більшості системного програмного забезпечення, яке традиційно писалося на асемблері. Що стосується граматики і синтаксису, то С є структурною мовою програмування.

С++ – універсальна мова програмування високого рівня з підтримкою декількох парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. При її створенні розробники прагнули зберегти сумісність з мовою С. Більшість програм на С працюватимуть і з компілятором С++. С++ має синтаксис, заснований на синтаксисі С. Мова програмування С++ була створена на початку 1980-х років, її творець – співробітник тієї ж фірми Bell Laboratories – Бьйорн Страуструп. Вона розроблялась як надбудова над С, використовуючи всі можливості останньої і додавши можливість роботи з класами та об'єктами. У 1983 році відбулося переймену- вання мови з «Сі з класами» в «мову програмування С++».

З 90-х років С++ стає однією з найуживаніших мов програмування загального призначення. Мову використовують для системного програмування, розробки програмного забезпечення, написання драйверів, потужних серверних та клієнтських програм, а також для створення різного роду розважальних програм.

Нащадками мови С++ є мови Java (Джава) та C# (Сі-шарп), спеціалізовані для програмування в сучасних комп'ютерних мережах. Ці мови за структурою схожі на С++.

Основні поняття мови програмування С++.

Склад мов програмування

У тексті на будь-якій природній мові можна виділити чотири основні елементи:

- символи,
- слова,
- словосполучення
- речення.

Алгоритмічна мова також містить такі елементи, тільки слова називають лексемами (елементарними конструкціями), словосполучення – виразами, речення – операторами.

Лексеми утворюються з символів, вирази з лексем і символів, оператори з символів виразів і лексем.

Таким чином, елементами алгоритмічної мови є:

- алфавіт мови,
- лексеми,
- вирази,
- оператори.

Програми складаються з синтаксичних конструкцій, які називаються *командами* (інші назви – оператори, вказівники, речення). Команди будуються з *лексем* – неподільних елементів мови: слів, чисел, символів, операцій.

Алфавіт мови програмування С++

Алфавіт мови – це той набір символів (знаків), що є допустимим у даній мові.

Алфавіт мови С ++ включає

- великі та малі літери латинського алфавіту: 'A', ..., 'Z', 'a', ..., 'z';
- цифри `0, 1, ..., 9`;
- спеціальні символи: " ( ) ' [ ] { } < > . , ; : ? ! * + – = / ₩ | # % $ & ~ @ та символ підкреслення _;
- пробільні символи (пробіл, символ табуляції, символи переходу на новий рядок);
- інші символи можна використовувати лише в коментарях до тексту програми.

Лексеми: класифікація.

Текст програми представляє собою послідовність рядків, які складаються з символів, що входять до алфавіту мови. Рядки програми завершуються спеціальними символами. Максимальна довжина рядка становить приблизно 2048 байтів. Символи з алфавіту мови використовуються для побудови базових елементів програм – лексем. Нагадаємо, що лексема це мінімальна неподільна одиниця мови, що має самостійний зміст і при компіляції сприймається як

єдине ціле.

Розглядають наступні класи лексем:

- Спеціальні символи
- Зарезервовані (ключові, службові) слова (зарезервовані ідентифікатори)
- Ідентифікатори (імена)
- Стандартні (визначені) ідентифікатори
- Ідентифікатори директив
- Ідентифікатори стандартних функцій
- Ідентифікатори користувача
- Знаки операцій
- Константи (літерали)
- Числа
- Десяткові
- Вісімкові
- Шістнадцяткові
- Символи
- Рядки
- Символи-роздільники (дужки, крапка, кома, пробільні символи)

Межі лексем визначаються іншими лексемами, такими, як роздільники або знаки операцій.

До *спеціальних символів* відносяться наступні:

+ - * / < = > ( ) { } [ ] . , : ; '^ @ # $.

Також використовуються комбінації спецсимволів: <= >= := .. (* *) (. .).

Ключові слова.

Зарезервовані (ключові, службові) слова – це зарезервовані ідентифікатори, які складають фіксований словник мови програмування. Зміст і спосіб їх використання строго визначений в описі мови. Такі слова не можна використовувати в якості імен програмних об'єктів (змінних, процедур, функцій та ін.).

Зверніть увагу, що при написанні ключових слів великі та маленькі літери розрізняються і це треба враховувати при написанні програм, оскільки від регістру символів повністю змінюється їх зміст.

До зарезервованих слів відносяться наступні:

alignas*	alignof*	and	and_eq	asm
auto(1)	bitand	bitor	bool	break
case	catch	char	char16_t*	char32_t*
class	compl	const	constexpr*	const_cast
continue	decltype*	default(1)	delete(1)	do
double	dynamic_cast	else	enum	explicit

export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept*
not	not_eq	nullptr*	operator	or
or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof
static	static_assert*	static_cast	struct	switch
template	this	thread_local*	throw	true
try	typedef	typeid	typename	union
unsigned	using(1)	virtual	void	volatile
wchar_t	while	xor	xor_eq	

* - починаючи з C++11

(1) – значення в C++11 змінилося відносно початкового Ключові слова поділяють на наступні групи:

- специфікатори типів: char, double, enum, float, int, long, short struct, signed, union, unsigned, void, typedef;
- кваліфікатори типів: const, volatile;
- кваліфікатори класів пам'яті: auto, extern, register, static;
- оператори мови та ідентифікатори спеціального призначення: break, continue, do, for, goto, if, return, switch, while; default, cas else,sizeof;
- модифікатори і псевдозмінні: privat, protected.

Ідентифікатори та імена користувача.

Ідентифікатор (ім'я) – *послідовність латинських букв та цифр, що починається з букви.*

Нагадаємо, що знак підкреслювання також вважається буквою.

Складаючи програму, користувач описує різні об'єкти і надає їм імена на свій розсуд. Тут простежується аналогія з математикою та фізикою, де різні величини позначають різними буквами, наприклад: *a, b, c* – довжини сторін трикутника, *h* – висота, *s* – шлях чи площа. В алгоритмічних мовах дане, яке міститиме в собі інформацію про висоту, можна назвати різними способами: *h, high, vysota* чи ще інакше.

Правила утворення імен користувача (*користувацьких ідентифікаторів*):

- ім'я може складатися лише з латинських літер, цифр та символу підкреслення – «_» (риска знизу);
- цифра не може бути першим символом в імені;
- літери можуть бути малими або великими;
- може бути будь-якої довжини (обмежується лише доступною

пам’яттю, але насправді такі імена сенсу не мають);

- пропуски в іменах не допускаються;
- два різні об’єкта не можна позначати одним і тим же іменем в одній програмі.

В іменах великі і малі букви розрізняються: імена *A* та *a* (або *MyName* та *my_name*) – це різні імена з точки зору мови програмування C++.

Довжина ідентифікатора за стандартом не обмежена, але деякі компілятори і компонувальники накладають на неї обмеження. Ідентифікатор створюється на етапі оголошення змінної, функції, типу тощо. Після цього його можна використовувати в подальших операторах програми. При виборі ідентифікатора необхідно мати на увазі наступне:

- ідентифікатор не повинен збігатися з ключовими словами та іменами використовуваних стандартних об'єктів мови;
- не рекомендується починати ідентифікатори з символу підкреслення, оскільки вони можуть збігтися з іменами системних функцій або змінних, і, крім того, це знижує мобільність програми;
- на ідентифікатори, використовувані для визначення зовнішніх змінних, накладаються обмеження компонувальника.

На відміну від математики та фізики в інформатиці використовуються довгі імена, наприклад, *high*, *myName*, *MyNumber*, *my_program* тощо, які мають логічний смисл і дають розуміння призначення даного ідентифікатору.

Приклад. Наступні користувацькі ідентифікатори утворені правильно: *a, b, c, x, z, a1, a2, ... , a100, alpha, cat, my_number*. А такі імена утворені неправильно: *10a*, *11b*, *a+2*, *a?*, оскільки при їх записі не дотримано правил, що наведені вище.

Щоб текст програми був більш зрозумілим, рекомендується дотримуватися загальноприйнятих угод про імена об'єктів.

- ім'я змінної пишеться малими літерами, наприклад index;
- з великої літери починаються імена функцій, класів, типів – Index;
- повністю великими літерами позначаються константи – INDEX;
- якщо ім’я складається з декількох слів, як, наприклад, birth_date, то прийнято або розділяти слова символом підкреслювання (birth_date), або писати кожне наступне слово з великої літери (birthDate). Який з двох способів краще – це справа власних уподобань програміста;
- не бажано використовувати скорочення або акроніми, особливо, якщо вони не є загальновідомими, оскільки це призводить до труднощів у розумінні програми;
- імена функцій та методів бажано задавати дієсловами;
- для створення ідентифікаторів слід використовувати англійську мову.

Існує угода про правила створення імен, названа угорською нотацією (оскільки запропонував її співробітник компанії Microsoft угорець за національністю), за якою кожне слово, з якого складається ідентифікатор, починається з великої літери, а спочатку ставиться префікс, який відповідає

типу величини, наприклад, iMaxLength, ipfnSetFirstDialog. Така нотація стала внутрішнім стандартом Microsoft. Префікси зарані обговорюються і є загальними для компанії.

Проте на сьогодні специфікація мови програмування С++ не рекомендує програмістам застосовувати таку нотацію.

Стандартні ідентифікатори.

При створенні програм доводиться використовувати ряд стандартних функцій, які є загальновизнаними, як то виведення інформації на екран, відкриття файлів, застосування математичних функцій, назви бібліотек, тощо. Саме тому у мові програмування завжди є набір стандартних ідентифікаторів, які мають своє призначення і широко використовуються за умови підключення відповідних бібліотек. Як правило, стандартні імена описані у бібліотеках, які входять до стандарту мови. Стандарт періодично оновлюється і до мови додаються нові стандартні ідентифікатори.

До *стандартних* (*визначених*) *ідентифікаторів* відносяться:

- імена вбудованих у мову функцій (наприклад, *cin*, *cout*, *sin*, *sqrt*),
- імена директив (наприклад, *#include*, *#define*);
- імена класів, що входять до стандарту мови, тощо.

Використання таких ідентифікаторів в якості імені змінної допускається, однак у цьому випадку їхня стандартна дія для даної програми буде втрачена. Проте деякі з них можна перевизначати, тобто дещо уточнювати, або змінювати їх призначення.

Знаки операцій.

Знак операції – це один або більше символів, що визначають дію над операндами. Усередині знаку операції пропуски не допускаються. Операції поділяються на унарні, бінарні та тернарну за кількістю операндів, що беруть участь у операції. Знаки операцій розглянемо далі.

Один і той же знак операції може інтерпретуватися по-різному в залежності від контексту. Всі знаки операцій за винятком [], (), ? і : є окремими лексемами.

Більшість стандартних операцій можна перевизначити (перевантажити), тобто визначити для них нове розуміння або взагалі змінити їх дію.

Константи (літерали) в С++.

Будь-які значення, які використовуються у програмах – це або значення змінних, або константи (літерали). Принципова відмінність літерала – для нього не виділяється окрема пам'ять. Літерал є частиною коду програми, тому в процесі виконання програми його значення змінитись не може.

Константа у тексті програми може бути лексемою одного з двох видів: або це деяке фіксоване значення, або ідентифікатор, оголошений як константа. Частіше *літералом* називають фіксоване значення, а ідентифікатор, значення якого не можна змінити, *константою*. Нагадаємо, що константу, описану

ідентифікатором прийнято оголошувати повністю великими літерами.

Коли в програмі зустрічається певне число, наприклад 15, то це число називається літералом, або літеральної константою. Константою, тому що ми не можемо змінити його значення, і літералом, тому що його значення фігурує в тексті програми. Літерал є безадресною величиною: хоча реально він, звичайно, зберігається в пам'яті машини, немає ніякого способу дізнатися його адресу. Кожен літерал має певний тип, який визначається компілятором за замовчуванням, або можна задати його тип явно за потребою.

Компілятор виділяє літерал, як лексему і відносить її до тієї чи іншої групи, а потім всередині групи до певного типу по її формі запису в тексті програми і по числовому значенню.

Константа (літерал) *– це лексема, що представляє зображення фіксованого числового, рядкового або символьного значення.*

Якщо значення деякої величини (даного) не змінюватиметься протягом виконання всієї програми, то таке дане варто задати як постійну (константу).

Константи бувають не типізовані (літеральні) і типізовані, тобто під час оголошення ідентифікатора константи можна вказати її тип.

Літерали діляться на 6 груп:

- цілі;
- дійсні (з фіксованою та плаваючою крапкою);
- логічні;
- символьні;
- рядкові;
- типізовані.

Цілі літерали.

Цілі константи можуть бути десятковими, вісімковими і шістнадцятковими.

Десяткові константи визначаються як послідовність десяткових цифр, що починається не з 0, якщо це не саме число 0 (приклади: 8, 0, 192345).

Вісімкові константи – це константи, які завжди починаються з 0. За 0 слідують вісімкові цифри (приклади: 01, 016 – десяткове значення 14,).

Шістнадцяткові константи – послідовність шістнадцяткових цифр, яким передують символи 0x або 0X (приклади: 0xA, 0X00F).

Залежно від значення цілого літерала компілятор по-різному представить його в пам'яті комп'ютера (тобто компілятор сам визначить тип даних цього літерала).

Наприклад, одне й те ж саме число 20 у різних системах числення має такий вигляд

Десяткова	20
Вісімкова	024
Шістнадцяткова	0x14

За замовчуванням всі цілі літерали мають тип signed int. Можна явно визначити цілий літерал як має тип long long, приписавши в кінці числа букву L

(можна використовувати як прописну літеру L, так і рядкову – l, однак для зручності читання не слід вживати малу: її легко переплутати з цифрою 1).

Буква U (або u) поставлена після числа визначає літерал як unsigned int, а дві букви – UL або LU – як тип unsigned long. Наприклад:

128u 1024UL 1L 8Lu

Дійсні літерали.

Дійсні константи мають іншу форму внутрішнього представлення в пам'яті комп'ютера. Компілятор розпізнає такі константи за їхнім виглядом. Дійсні константи можуть мати дві форми подання: з фіксованою точкою і з плаваючою точкою.

Літерали з фіксованою точкою мають такий вигляд:

[Цифри].[цифри]

Наприклад, 5.7, 0, .0001, 41.

Як видно з прикладів, така форма запису допускає відсутність або цілої частини, або дробової, але не двох одразу.

Вид константи з плаваючою точкою (експоненціальна або показникова форма):

[цифри] [цифри]{E | e} [+ | -] [цифри]

або вигляд

mEn

Наприклад, 0.5e5, .11e-5, 5E3.

У цій формі запису числа можна виділити такі основні характеристики:

- знак числа (додатнє чи від'ємне),
- m – мантиса числа – для нормалізованої форми $m \in [1;10)$,
- e – експонента,
- знак порядку,
- n – порядок числа (виражає степінь основи числа, на яке множиться мантиса).

Зазначені характеристики дійсного числа зберігаються у пам'яті комп'ютера. Число у показниковій формі може бути представлено, наприклад, так: 1.0123E-10. Мантиса записується ліворуч від знаку експоненти(***E чи e***), порядок – праворуч. Символ **E(e)** означає основу степеню – 10, і компілятор розпізнає цей запис як форму представлення дійсного числа. Символ пропуску всередині числа не допускається, а для відділення цілої частини від дробової використовується не кома, а крапка. При додатних значеннях числа і мантиси знак «+» можна не вказувати.

Експоненціальна форма дійсного числа використовується для запису дуже великих або дуже малих чисел, для яких задавати зайві нулі не зовсім зручно, наприклад:

$1.0123*10^{20}$=1.0123e20=10123000000000000000,

$1.0123*10^{-10}$=1.0123E-10=0.0000010123.

Експоненціальна форма запису також допускає відсутність цілої або дробової частини.

За замовчуванням всі дійсні константи мають тип **double** – подвійну точність, що найчастіше займає в пам’яті 64 біти, тобто 8 байтів. Але у випадку, якщо програміста не влаштовує тип за замовчуванням, його можна вказати явно за допомогою спеціальних літер. Так, додавши літеру **f чи F**, константі надають дійсний тип **float** з одинарною точністю, наприклад, 8.5f. Якщо в представленні константи використовується літера **L чи l**, то вона має тип **long double**.

Зображення від’ємної цілої чи дійсної константи вважається константним виразом, що складається зі знаку унарної операції зміни знаку (-) та константи. Наприклад, -273, -2730.e-1, -273L.

Логічні літерали.

Логічні константи представлені двома словами, що позначають логічні значення – *true* (істина) і *false* (неправда).

Символьні константи.

Символьні константи – це один або два символи, які заключено в одинарні прямі лапки.

Наприклад, 's', '2', '#', 'c', ' ' (пробіл).

Символьні константи, що складаються з одного символу, мають тип *char* і займають в пам'яті один байт, символьні константи, що складаються з двох символів, мають тип *int* і займають два байти. Символьні константи мають цілий тип і їх можна використовувати як цілочислові операнди у виразах.

Всі символьні константи та їх відповідні числові значення (коди) представлені у стандартній таблиці ASCII (таблиця кодування). Причому числові значення для зручності представлені у десятковому, вісімковому та шістнадцятковому кодах.

Окремої уваги заслуговують послідовності, що починаються зі зворотної риски \ (back-slash, бек-слеш, обернений слеш). Вони називаються керуючими або ***escape***-послідовностями, вони використовуються:

Для представлення символів, які не мають графічного відображення, наприклад:

\a – звуковий сигнал,

\v – вертикальна табуляція,

\n – перехід на новий рядок,

\t – горизонтальна табуляція.

Для представлення (екранування) символів \ , ' , ? , ", тобто \\, \', \? ,\".

Для представлення будь-яких символів за допомогою шістнадцяткових (\0x2b – '+') або вісімкових (\053 – '+') кодів.

escape -послідовності – це комбінації символів, що представляють розділові (CR, Tab та ін.) і нетрадиційні символи.

Керуюча послідовність інтерпретується як одиночний символ. Але в останніх стандартах мови програмування С++ керуючу послідовність потрібно заключать у подвійні прямі лапки для їх коректної роботи. Якщо безпосередньо за косою рискою знаходиться символ, який не передбачено стандартними

escape-комбінаціями, то результат інтерпретації непередбачуваний, але частіше всього бек-слеш буде просто проігнорований.

Порожня символьна константа недопустима.

Символьний літерал може містити префікс L (наприклад, L'a'), що означає спеціальний тип ***wchar_t*** – двобайтний символьний тип, який застосовується для зберігання символів національних алфавітів, як що вони не можуть бути представлені звичайним типом *char*, такі як, наприклад, китайські або японські літери.

Рядкові константи.

Рядкова константа – це послідовність символів, яку заключено в подвійні прямі лапки. У кінець кожного рядкового літералу компілятором автоматично додається нуль-символ, який представляється керуючою послідовністю "\0". Саме тому навіть порожній рядок "" має довжину 1 байт.

Існує суттєва різниця між символом 'ф' і рядком "ф", оскільки символ має 1 байт, а рядок 2 байти.

Приклад рядка: "Студенти – веселий народ."

Усередині рядків також можуть використовуватися керуючі символи. Наприклад, рядок "\nМова програмування" – напис «Мова програмування» на екрані виводиться з нового рядка.

Якщо всередині рядкового літерала потрібно записати подвійну пряму лапку, то перед нею ставиться бек-слеш, за яким компілятор відрізняє її від подвійної лапки, яка обмежує (закінчує) рядок.

Наприклад, рядок "Ми вивчаємо \"Мову програмування С++\"." на екрані має вигляд

```
Ми вивчаємо "Мову програмування С++".
```

Рядки, які записані у програмі підряд при компіляції конкатенуються («склеюються»). Тобто послідовність двох рядків

"Літерали – це фіксовані значення." "Вони застосовуються в програмах."

На екрані виглядають наступним чином:

```
Літерали - це фіксовані значення. Вони застосовуються в
програмах.
```

Довгу рядкову константу можна розмістити в декількох рядках, використовуючи в якості знаку переносу бекслеш. При виведенні ці символи ігноруються компілятором, а наступний рядок сприймається як продовження попереднього.

Наприклад, рядок "Ми розглядаємо\
типи та види літералів\
у мові програмування"

є повністю еквівалентним рядку

«Ми розглядаємо типи та види літералів у мові програмування».

У C++ існують стандартні константи, описані у різних бібліотеках. Наприклад, математичні константи: число π позначається M_PI, π/2 – M_PI_2, ln2 – M_LN2 тощо. Їх можна безпосередньо використовувати в програмі, попередньо підключивши бібліотеку math.h або cmath.

Максимальна допустима довжина рядкового літерала в Microsoft C++ приблизно 2048 байтів. Однак якщо рядковий літерал складається з двох частин, взятих в подвійні лапки, препроцесор об'єднує ці частини в один рядок, і для кожного об'єднаного рядка додає додатковий байт до загальної кількості байтів.

Наприклад, припустимо, що рядок складається з 40 рядків з 50 символами в кожному рядку (2000 символів) і одного рядка з 7 символами і що кожен рядок заключений в подвійні лапки. Це означає, що додається до 2007 байтів плюс один байт для завершального нуль-символу, тобто всього 2008 байтів. В об'єднанні додатковий символ додається для кожного з перших 40 рядків. В результаті ми отримуємо 2048 байтів. Зверніть увагу, що якщо замість подвійних лапок використовуються продовження рядків (\), препроцесор не додає додатковий символ для кожного рядка. І хоча окремі рядки в лапках не можуть бути довшими 2048 байтів, об'єднавши рядки, можна створити рядковий літерал, що складається приблизно з 65535 байтів.

Типізовані константи.

Типізовані константи використовуються як змінні, значення яких не може бути змінено після ініціалізації. Під час виконання програми значення констант змінювати не можна. Типізована константа оголошується за допомогою ключового слова const, за яким слід вказати тип константи, але, на відміну від змінних, константи завжди повинні бути ініціалізовані, тобто їм має бути присвоєне конкретне значення.

Тема 1.2. Розгляд опису змінних та різних типів даних, включаючи цілі, дійсні, символьні та логічні типи даних

Структура програми на C++.

Структура програми – це розмітка робочого коду з метою точного визначення меж основних блоків програми та синтаксису. Структура програми дещо відрізняється в залежності від обраного середовища програмування.

Загалом, програма мовою C++ складається з

- директив препроцесора,
- функцій,
- описів,
- коментарів.

Одна з функцій обов’язково повинна мати ім’я ***main*** і саме з неї починається виконання програми.

Програма на мові С++ має наступну структуру:

```
1.  //program.cpp – файл, який містить код програми
2.  #include <назва бібліотечного файлу1>
3.  #include <назва бібліотечного файлу N>
4.  <інші директиви препроцесора>
5.  <описи>
6.  <оголошення глобальних змінних>
7.  <оголошення глобальних констант>;
8.  <оголошення і створення функцій користувача>;
9.  int main()
10. {
11. <оператори, які складають тіло програми (функції)>;
12. }
```

Спершу перерахуємо елементи цієї структури.

- У першому рядку міститься коментар, який починається символом //.
- З другого по четвертий рядки після символу # розташовуються директиви препроцесора.
- У 9 рядку – заголовок головної функції.
- Фігурні дужки {} у 10 та 12 рядках означають початок та кінець функції *main*. Оператори у тілі функції розділяються символом **;** (крапка з комою).
- З 5 по 8 рядки можуть міститися описи та оголошення, які не є обов'язковими.

Коментарі.

Коментар – це фрагмент тексту програми, який служить для пояснення призначення програми або окремих команд і не впливає на виконання команд.

Коментарі потрібні для кращого розуміння програми як для самого розробника, так і для сторонніх людей, яким потрібно вникати в суть написаної програми. При створенні великих проектів групою розробників коментарі набувають особливого значення, оскільки документований код спрощує роботу колективу в цілому.

У мові С++ є два види коментарів – *однорядкові* та *багаторядкові*. Їх записують так:

```
// однорядковий коментар
/* багаторядковий коментар*/
```

У першому випадку коментар повинен бути або в кінці рядка, або єдиним

у рядку. Він є однорядковим і діє тільки до кінця рядка. Отже, коментарі такого виду можуть знаходитися лише праворуч від оператора. Вкладення однорядкових коментарів один в один не має сенсу, оскільки лівий обмежувач вкладеного коментарю при цьому нічим не відрізняється від інших вже за коментованих символів.

Другий спосіб більш універсальний: багаторядковий коментар можна записувати будь-де. Компілятор ігнорує все, що міститься між символами /* і */, включаючи самі ці символи, і отже, між ними можна вставляти скільки завгодно рядків тексту. Вкладення багаторядкових коментарів не використовується, оскільки перший символ */ у вкладеному коментарі завершить коментар взагалі і це проведе до помилки компіляції, оскільки подальший текст коментаря буде сприйняте компілятором як фрагмент основного коду програми.

Як правило, однорядкові коментарі пояснюють зміст окремого рядка програми. Багаторядкові коментарі можуть містити умову задачі, пояснення призначення та параметрів функцій або класів, тощо. Крім того, коментарі дозволяють локалізувати помилки при налагодженні програми адже, за коментувавши підозрілий фрагмент програми, можна досить швидко знайти місце можливої помилки. Таким чином, багаторядкові коментарі доцільно застосовувати для тимчасового виключення блоків при налагодженні програми.

Наведемо кілька порад стосовно раціонального застосування коментарів:

- коментарі можна записувати будь-якою мовою, враховуючи аудиторію для якої вони призначені;
- коментарі бажано записувати правильними та логічними реченнями, за правилами мови коментаря і використовувати правильну пунктуацію
- розміщувати в коментарях тільки потрібну для супроводу інформацію;
- пропуск рядка – один з найбільш ефективних коментарів, що значно поліпшує розуміння програми, видимо розбиваючи її на фрагменти;
- штрихові лінії коментаря або порожні рядки застосовуються для поділу функцій та інших логічно завершених фрагментів програм.

Директиви препроцесора.

Препроцесор – це програма, яка обробляє директиви на першому етапі компіляції. Вона виконує макропідстановку, умовну компіляцію і включення іменованих файлів. Кожна з цих операцій кодується у виді особливого оператора (*директиви*), що починається символом #.

Директиви препроцесора – це команди компілятора відповідної мови програмування, які виконуються до початку компіляції програми.

Препроцесор не виконує синтаксичний аналіз тексту. Він просто розпізнає макропідстановки і виконує їх. Директиви препроцесора не залежать від синтаксису мови, за одним виключенням – їх імена чуттєві до регістра букв.

У мові C++ визначено такі директиви

#define	#elif	#else	#endif	#error	#if
#ifdef	#ifndef	#include	#line	#pragma	#undef

Кожна директива повинна розташовуватися в окремому рядку програми. При цьому необхідно уважно стежити за пробілами всередині директив. Деякі з них не мають значення, а інші приводять до помилок. Як правило, в кінці рядка з директивою не ставиться ніяких символів-роздільників.

Директиви препроцесора можна розташовувати в довільному місці програми.

Директива #include.

Директивою ***#include*** до тексту програми включаються заголовочні або інші файли. Фактично необхідно приєднати програмний код із файлу, зазначеного після цієї директиви.

Заголовні файли мають розширення **.h** Їх також називають файлами заголовків (*header* -файлами, бібліотеками, модулями). У таких файлах зазвичай оголошують константи і змінні, заголовки (сигнатури) функцій тощо. У файлах заголовків визначені всі стандартні команди і функції мови С++.

Ім’я файлу може бути вказане двома способами:

#include <fileName.h> #include "OwnFile.h"

Тобто після директиви #include потрібно в кутових лапках <…> записати назву файлу та вказати його розширення. Кутові лапки означають, що заданий файл є стандартним і компілятор шукає його у визначених для даного середовища програмування місцях, спеціально призначених для зберігання таких файлів. В переважній більшості середовищ всі стандартні бібліотеки розміщені в папці INCLUDE. Тоді стандартний файл підключається таким чином:

наприклад, #include <math.h> або, більш нові формати, #include <cstdio>

Подвійні лапки означають, що заголовочний файл – користувацький, тобто створений програмістом і його пошук компілятором починається з того каталогу, в якому розташований проект (початковий код програми).

Якщо потрібний файл розміщений не в папці проекту, то зазначається повний шлях до файлу, починаючи з диску і т.д. Наприклад, якщо деякий файл MyLib.h є в папці ***stud*** на диску **d**:, то потрібно писати так: **#include** "d:***stud***\MyLib.h".

Згідно з останніми стандартами ISO/ANSI файли заголовків в директиві **#include** прийнято записувати без розширення, наприклад: **#include** <stdlib>, але в такому випадку потрібно вказувати *простір імен*, в якому розв’язується задача. Файли заголовків мови С, які використовуються в С++-програмах, починаються з літери ***c***, наприклад # **include** <cmath>.

Вживання директиви **#include** не підключає відповідну стандартну бібліотеку, а тільки дозволяє вставити в текст програми опису із зазначеного заголовка. Підключення кодів бібліотеки здійснюється на етапі компонування, тобто після компіляції. Хоча в стандартних заголовних файлах містяться всі описи стандартних функцій, в код програми включаються тільки ті функції, які

використовуються в програмі.

Заголовний файл сам по собі може містити директиви #include. Вони називаються *вкладеними директивами* #include. Тому іноді важко зрозуміти, які ж конкретно заголовочні файли включені в даний текст, і, таким чином, деякі заголовочні файли можуть виявитися включеними декілька разів. Запобігти цьому дозволяють *умовні директиви препроцесора*. Розглянемо приклад:

#ifndef MYTEXT_H
#define MYTEXT_H
/*вміст файлу mytext.h*/
#endif

Умовна директива #ifndef перевіряє, чи не було значення MYTEXT_H визначено раніше. MYTEXT_H – це константа препроцесора. Такі константи прийнято записувати великими буквами. Препроцесор опрацьовує наступні рядки аж до директиви #endif. В іншому випадку він пропускає рядки від #ifndef до #endif.

Директива #define MYTEXT_H

визначає константу препроцесора MYTEXT_H. Розмістивши цю директиву безпосередньо після директиви #ifndef, можна гарантувати, що змістовна частина заголовного файлу mytext.h буде включена у текст програми тільки один раз, скільки б разів не включався в текст сам цей файл.

Директива #define.

Директива **#define** слугує для заміни констант, ключових слів, операторів або виразів, які часто застосовуються у програмі деякими ідентифікаторами. Тобто ця директива має подвійне значення:

- визначення констант;
- визначення макросів.

Ідентифікатори, які замінюють текстові або числові значення (константи), називають *іменованими константами*. Ідентифікатори, що замінюють фрагменти програм, називають макровизначеннями, причому макровизначення можуть мати аргументи.

Директивою #**define** можна встановити постійне значення (оголосити іменовану константу). Наприклад, якщо в програмі є рядок #**define** N 25, то *N* під час виконання програми буде мати значення 25, тобто всі входження змінної *N* буде замінено на 25. У програмі змінювати це значення не можна, тобто, якщо вже *N* оголошено, як іменовану константу, то операції типу *N*++ або *N**=7 викличуть помилку компіляції.

Крім того директива #**define** дає можливість описати макроси (макровизначення) – короткі команди (перевизначити команди) або записати функції.

Наприклад,

```
#define D(a, b, c) b*b-4*a*c.
```

Тепер скрізь для обчислення дискримінанту замість команди d = b*b-

4*a*c можна записувати d = D(a, b, c), тобто D(a, b, c) в даній програмі є макросом – скороченим записом команди обчислення виразу.

При створенні макросу з параметрами однією з поширених помилок початківців, є наявність пробілу між іменем макросу і відкритою дужкою при описуванні директиви **#define**. Це неправильно, тобто відкрита дужка повинна розташовуватись впритул до імені макросу, інакше при виконанні програми виникне помилка компіляції, оскільки в такому випадку макрос вважається непараметризованим, а параметри є частиною подальшого виразу.

Також необхідно враховувати, що макрос не є функцією і в якості параметрів йому не можна передавати вирази.

Наприклад, якщо задано макровизначення для знаходження квадрату числа такого виду

#define Mult(x) x*x

то коректним викликом макросу у програмі є, наприклад, такий Mult(c), де c – деяке число (вже відоме на момент виклику). А виклик Mult(c+5) приведе до помилкового результату, оскільки буде виконана підстановка c+5*c+5, що за правилами порядку дії приводить до результату 6*c+5 і є неправильним для нашого макросу.

Щоб запобігти таким проблемам при написанні макросів з параметрами в усіх місцях використання параметрів їх потрібно заключать в круглі дужки, а для підсилення коректності і весь вираз також взяти у круглі дужки. Таким чином більш коректним варіантом наведеного вище прикладу буде наступний:
#define Mult(x) ((x)*(x))

Код програми

Початкова програма, яка підготовлена на C++ у вигляді текстового файлу (*початковий код програми*), проходить 3 етапи обробки:

1) препроцесорне перетворення тексту;
2) компіляція;
3) компонування (редагування зв’язків або збірок).

Після цих трьох етапів формується *виконуваний код програми*. Завдання препроцесора – перетворення тексту програми до її компіляції. Правила препроцесорної обробки визначає програміст за допомогою директив препроцесора.

Як вже було зазначено раніше, суттєвою особливістю мови C++ є те, що програми складаються з функцій, які виконують роль підпрограм в інших мовах. Головна функція, яка повинна бути в кожній програмі, – це функція виду

```
int main()
{
тіло функції (можна з командою return 0;)
}
```

де ***int* main()** – ***заголовок функції.***

Тіло функції – це послідовність оголошень, визначень, описів і

виконуваних операторів, заключених у фігурні дужки {}. Кожне оголошення, визначення, опис або оператор закінчується символом **;** (крапка з комою).

Визначення – задають об'єкти (об'єкт – це іменована область пам'яті; окремий випадок об'єкта – змінна), необхідні для представлення в програмі оброблюваних даних. Напиклад,

const int *y*=10; //іменована константа
float *x*; //змінна.

Описи – повідомляють компілятор про властивості і імена об'єктів і функцій, описаних в інших частинах програми.

Оператори – визначають дії програми на кожному кроці її виконання.

Типи даних

Навколишній світ можна відобразити в програмі у вигляді конкретних даних. Дані мають деяку величину, тобто *величина* – одиниця даних, якими оперує програма.

Величина характеризується

- назвою (ім’я, ідентифікатор);
- типом (залежить від того, що саме ця величина описує і не змінюється протягом виконання програми);
- значенням, яке може змінюватись за час виконання програми безліч разів.

Мета програми полягає в обробці різноманітних даних. Дані можуть бути *сталими* та *змінними*. Константи (літерали) – це сталі значення, які описані у попередній темі. Як правило, літерали вже мають конкретне значення і за їх виглядом *автоматично* (за замовчуванням) визначається їх тип. Змінні величини обов’язково мають ім’я (ідентифікатор), яке не дає жодного уявлення про їх тип. Тому кожен ідентифікатор у програмі мовою С++ повинен мати асоційований з ним тип даних. Дані різних типів зберігаються і обробляються по-різному. Всі змінні, які фігурують у програмі, ретельно класифікують за типами.

Тип даних визначає:

- обсяг оперативної пам'яті, який резервується для зберігання значень зазначеного типу;
- множину значень, які можуть приймати величини цього типу;
- операції і функції, які можна застосовувати до даних цього типу.

Зауваження: Обсяг пам’яті може залежати від різновиду операційної системи комп'ютера.

Залежно від вимог завдання програміст вибирає тип для об'єктів програми.

Типи С++ можна розділити на *прості* (вбудовані) і *складені* (похідні). До простих типів відносять типи, які характеризуються одним значенням. В С++ визначено 7 простих типів даних:

- ***int*** (цілий);
- ***char*** (символьний);

- ***wchar_t*** (розширений символьний);
- ***bool*** (логічний);
- ***float*** (дійсний);
- ***double*** (дійсний з подвійною точністю);
- ***void*** (порожній тип, який не має значення).

На основі цих типів вводиться опис складених типів (масивів, функцій, структур, класів, вказівників, тощо). Оскільки похідні типи будуються на основі простих, то їх може бути велика кількість, тому для початку розглянемо прості вбудовані типи.

Для уточнення внутрішнього подання та діапазону значень стандартних типів мова С++ використовує чотири *специфікатори типу*:

- ***short*** (короткий);
- ***long*** (довгий);
- ***signed*** (знаковий);
- ***unsigned*** (беззнаковий).

Всі величини повинні бути описані до першого їх використання.

Опис змінних

Для зберігання різних даних використовуються змінні. Поняття змінних відоме зі шкільного курсу математики. У програмуванні принципи досить схожі. Величини, які під час виконання програми можуть мати різні значення, називають *змінними*.

Змінна (ідентифікатор користувача) – це іменована область пам'яті, у якій зберігаються дані визначеного типу. Змінна має ім'я, розмір та інші атрибути, такі як область видимості, час існування, тощо. Ім'я змінної служить для звертання до області пам'яті, у якій зберігається її значення. Перед використанням будь-яка змінна повинна бути описана, при цьому для неї резервується деяка область пам'яті, розмір якої залежить від конкретного типу змінної. Під час виконання програми змінна може приймати різні значення.

Ім'я змінній дає програміст під час написання програми.

Кожне ім'я змінної у програмі – це ідентифікатор користувача і для того, щоб його розпізнавав компілятор, змінна повинна бути описана. Для опису вказується її тип та ім'я.

Загальний вигляд опису змінних такий:

тип_змінної ім'я_змінної;

або

тип <список імен змінних, перерахованих через кому>;

де ***тип*** – це коректний тип даних мови С++,

ім'я_змінної – надає програміст на власний розсуд.

Наприклад, змінні оголошують так:

```
int a, dob_numbers;
float minimum, x1, suma;
```

```
char simv;
```

При оголошенні змінних для них виділяється комірка пам’яті, розмір якої залежить від обраного типу даних. Але в цій комірці не має ніякого значення, тобто в пам’яті в якості значення цієї змінної знаходиться так зване «сміття». Автоматично ніяке значення змінній не надається. Тому бажано присвоїти оголошеній змінній значення, тобто *ініціалізувати* її. Якщо деяка змінна ініціалізується, то в списку вона вказується у вигляді:

<ім’я>=<константа>;

або

<ім’я>=<константний_вираз>;

або

<ім’я>=<вираз>;

або

<ім’я>=<інше_ім’я>;

де <ім’я> – ім’я деякої вже описаної змінної;

= – оператор присвоєння;

<константа> – визначає конкретне встановлене значення;

<константний_вираз> – вираз, який складається тільки з констант та знаків операцій, наприклад, 20*3+8;

<вираз> – вираз, який складається з констант, змінних та знаків операцій, наприклад, 25+*x*-2**b*.

```
int a=12;
double x1, dobutok=12*a;
char simb=12;
```

Цілі типи даних

Назва типу	**Назва типу на мові С++**	**Обсяг, байтів**	**Діапазон допустимих значень**
цілі числа	***int***	2 або 4	-32768 ... 32767 або -2147483648 ... 2147483647
короткі цілі числа	***short int***	2	-32768 ... 32767
короткі цілі числа без знаку	***unsigned short int***	2 або 4	0 ... 65535 або 0 ... 4294967295
довгі цілі числа	***long int***	4	-2147483648 ... 2147483647

довгі цілі числа без знаку	***unsigned long int***	4	0 ... 4294967295

Приклад. Оголосимо три змінні цілого типу:

```
int x, b;
short int z;
```

На етапі компіляції для змінних *x, b, z* буде надано певний обсяг оперативної пам'яті. Дати значення цим змінним можна на етапі виконання програми за допомогою команд присвоєння, наприклад, так:

```
x = 157; b = -68; z = 15;
```

У ділянку пам'яті, виділену для змінної *x*, буде записано число 157, для *b* – -68, а для *z* – 15.

Під час виконання програми значення змінних можна змінювати. Наприклад, команда присвоєння *x* = 2003 занесе до відповідної для змінної *x* ділянку пам'яті число 2003 (попереднє значення 157 видаляється автоматично).

Дійсні типи даних

Назва типу	**Назва типу на мові С++**	**Обсяг, байтів**	**Діапазон допустимих значень**
дійсні числа одинарної точності	***float***	4	$\pm 3{,}410^{-38}$ ... $\pm 3{,}410^{38}$; 0
дійсні числа подвійної точності	***double***	8	$\pm 1{,}710^{-308}$ ... $\pm 1{,}710^{308}$; 0
розширення дійсного числа подвійної точності	***long double***	10	$\pm 1{,}1810^{-4932}$ ...$\pm 1{,}1810^{4932}$; 0

У десяткових числах ціла і дробова частини числа відокремлюються крапкою. Внутрішнє представлення дійсного числа складається з 2 частин: мантиси і порядку. У IBM-сумісних ПК величини типу ***float*** займають 4 байта, з яких один розряд відводиться під знак мантиси, 8 розрядів під порядок і 24 - під мантису. Величини типу ***double*** займають 8 байтів, під порядок і мантису відводяться 11 і 52 розряду відповідно. Довжина мантиси визначає точність числа, а довжина близько його діапазону.

Приклад. Розглянемо фрагмент програми

```
float h, pi=3.1415926;
double v=365.976;
const float w=12, s=23.4;
```

Дійсні числа можна записати у форматі з *фіксованою точкою*, наприклад –2.3, 5.0041, або в *науковому форматі* (в форматі с *плаваючою точкою*), наприклад, –0.2e+2 (це є число –20), 3.27e–3 (це є 0.00327).

Запис $\pm me\pm n$ означає множення числа m на 10 в степені $\pm n$, тобто за визначенням $me\pm n=m*10^{\pm n}$, m – це мантисса, а n – це порядок.

Знак «+» можна не ставити, знак «–» писати обов'язково.

Символьний тип даних (char)

Символьний тип – це множина символів кодової таблиці ASCII (Американський стандартний код для міжнародного обміну). Символьна константа – це один символ, взятий в одинарні лапки, або число в 8-, 10- або 16-ковій системі числення, яке є кодом символу в таблиці ASCII. Кожному символу ставиться у відповідність число, яке називається кодом символу. Під величину символьного типу відводиться 1 байт. Символи з кодами від 0 до 31 відносяться до службових і мають самостійне значення тільки в операторах введення-виведення. Величини типу ***char*** також застосовуються для зберігання чисел з діапазонів від -128 до 127 (***signed char***) або від 0 до 255 (***unsigned char***).

Приклад. Розглянемо опис символьних змінних, де змінним *m*1, *m*2, *m*3 і *m*4 дамо значення латинської літери А чотирма способами:

```
char m1='A';
char m2=0101;
char m3=65;
char m4=0x41;
```

Число 65 - це десятковий код символу 'A', 101 – вісімковий, 41 – шістнадцятковий. На початку останніх двох кодів (101, 41) записують префікси "0" або "0x" відповідно.

Приклад. Розглянемо спосіб визначення десяткового ASCII-коду деякого символу, наприклад 'A':

```
char c = 'A';
int n = c;
```

Змінній *n* буде присвоєно значення 65.

Ще в C++ існує тип ***wchar_t***, який призначений для роботи з набором символів, для кодування яких недостатньо 1 байта, наприклад символів таблиці *Unicode*. Розмір цього типу, як правило, відповідає типу ***short***. Рядкові константи такого типу записуються з префіксом *L*: *L*"String #1".

Логічний тип даних (bool)

Логічний тип характеризується двома значеннями даних: *false* (неправда) і *true* (істина). Наприклад,

bool *b=true*.

Змінні цього типу займають 1 байт в пам'яті комп'ютера. У C++ значення

змінних типу ***int*** можна асоціювати з логічними значеннями: нулю відповідає значення *false*, всім іншим числам, відмінним від нуля – *true*.

Зауважимо, що не всі компілятори підтримують тип даних ***bool***. Тому, перед тим як його використовувати, варто з'ясувати можливості компілятора.

Тип void

Тип ***void*** застосовують до функцій, які *не повертають значення в точку виклику* або до функцій без параметрів. Множина значень цього типу – порожня.

Команда присвоєння

Команда *присвоєння* має такий загальний вигляд:

```
<назва змінної>=<вираз>
```

або

```
<ім'я змінної 1>=<ім'я змінної 2>=...=< ім'я змінної N>=<вираз>;
```

Дія команди. Обчислюється вираз і його значення записується в комірку відповідної змінної або декількох змінних одночасно. Вираз призначений для опису формул, за якими будуть виконуватися обчислення. Вираз може містити числа, літерали (константи), змінні, назви функцій, з'єднані символами операцій.

Змінна і вираз не обов'язково повинні бути одного типу. Крім того, в виразі можуть бути дані різних числових типів (змішані вирази). Якщо тип змінної не збігається з типом виразу, то в С ++ відбувається автоматичне перетворення (узгодження, перетворення) типів.

Типи користувача

Крім вищеописаних стандартних типів даних, можна створювати типи користувача

```
typedef <опис типу> <назва нового типу>;
```

Приклад. Опишемо тип *kilkist* для позначення коротких цілих даних без знаку:

```
typedef unsigned short int kilkist;
```

Змінні (*kil*1, *kil*2) цього типу в програмі можна оголосити так:

```
kilkist kil1, kil2;
```

Змінні

Дані, значення яких необхідно ввести з клавіатури або які під час виконання програми можуть мати різні значення, називають змінними. Їх оголошують так:

```
<тип змінних 1> <список змінних 1>;
...
<тип змінних N> < список змінних N>;
```

Елементи списків записують через кому. Наприклад, змінні оголошують так:

```
int a, c;
float b, d, z;
char w;
```

Змінна в С++ – іменована область пам'яті, в якій зберігаються дані певного типу. У змінної є тип, ім'я та значення. Тип визначає розмір комірки пам'яті, виділеної для цієї змінної, ім'я служить для звернення до цієї області пам'яті, в якій зберігається значення. Перед використанням будь-яка змінна повинна бути описана.

Приклади:

```
int a;
float x;
```

Змінним можна надавати початкове значення відразу під час оголошення. Це називається *ініціалізацію* даних.

Наприклад,

```
float b, d=2.5, a=4;
char w='t';
```

Отже, в загальному випадку змінні одного типу можна оголошувати так:

```
<тип змінних> <ім'я змінної 1>=<значення 1>, ..., <ім'я змінної
N>=<значення N>, <список інших змінних>;
```

Перетворення типів

В С ++ існує *явне* і *неявне* перетворення типів.

У загальному випадку неявне перетворення типів зводиться до участі в виразі змінних різного типу (так звана арифметика змішаних типів). Якщо подібна операція здійснюється над змінними базових типів, розглянутих вище, вона може спричинити за собою помилки: у разі, наприклад, якщо результат займає в пам'яті більше місця, ніж відведено під приймаючу змінну, неминуча втрата значущих розрядів.

Для явного перетворення змінної одного типу в інший перед ім'ям змінної в дужках вказується присвоєний їй новий тип:

```
p=(float) k;
```

Тема 1.3. Вивчення різних операцій та їх використання в мові С++

Операції в С++.

Операції представляють собою певну дію над операндом (операндами), результатом якої є значення, що повертається.

Операнд це об'єкт (об'єкти), над яким виконується операція.

Знаки операцій забезпечують формування виразів. Вирази складаються з операндів, знаків операцій та дужок. Кожний операнд є, в свою чергу, виразом або окремим випадком виразу – сталою або змінною.

В залежності від кількості операндів операції поділяються на

- унарні (дія виконується над одним операндом);
- бінарні (у операції приймає участь два операнда);
- тернарні (у операції приймає участь три операнда).

Таблиця 4.1. Операції в С++

Знак	**Зміст операції**	**Приклад застосування**
Унарні операції (дія з одним операндом, знак операції, як правило, ставиться ПЕРЕД операндом)		
&	операція визначення адреси операнда (адреси зображуються 16-річними числами)	**&***a*
*	вказівник (pointer) – засіб визначення адреси (звернення за адресою, розіменування). Надає можливість оперувати не з іменами змінних, а безпосередньо звертатися до областей пам'яті комп'ютера.	**int** **r, *nom;* **float*** *ptrA;*
–	унарний мінус, змінює знак арифметичного операнда.	*–k*
~	порозрядне інвертування внутрішнього двійкового коду цілочисельного операнда (побітове заперечення)	~10011010 = 01100101
!	логічне заперечення (НЕ). В якості логічних значень використовується 0 – неправда і не 0 – істина, запереченням	!18 = 0 !0 = 1

	0 буде 1, запереченням будь-якого ненульового числа буде 0.	
++	операція інкременту (збільшення на одиницю)	**int** *a*=2, *b*; *b*=3*++*a*; *a*=3; *b*=3*3=9 (префіксна)
	префіксна операція – збільшує операнд ДО його застосування, постфіксна операція – збільшує операнд ПІСЛЯ його застосування.	
--	операція декремента (зменшення на одиницю)	**int** *f*=20, *g*; *g*=(*f*--)-10; *g*=20-10=10 ; *f*=19 (постфіксна)
	префіксна операція - зменшує операнд ДО його застосування, постфіксна операція зменшує операнд ПІСЛЯ його застосування.	
Бінарні операції (дія з двома операндами, знак операції ставиться МІЖ операндами)		
Адитивні		
+	бінарний плюс (додавання арифметичних операндів)	3+5=8
–	бінарний мінус (віднімання арифметичних операндів)	7-12=-5
Мультиплікативні		
*	множення операндів арифметичного типу	2*(-5)+4=-6
/	ділення операндів арифметичного типу (якщо операнди цілочисельні, то виконується цілочисельне ділення)	12/4-2=1 18/(6-1)=3 (але не 3.6)
%	отримання остачі від ділення цілочислових операндів	7%3=2 4%13=4
Операції зсуву (визначені тільки для цілочислових операндів).		

Основне призначення цих операторів – швидкі обчислення, так як їх підтримка здійснюється на апаратному рівні (процесор), то алгоритми, виконані з використанням даних операторів, виконуються найбільш високопродуктивно.

<<	зсув вліво бітового представлення значення лівого цілочисельного операнда на кількість розрядів, що дорівнює значенню правого операнда (звільнені розряди обнуляються). *Примітка.* Таким чином, наприклад, виконується швидке піднесення до степеня числа 2. 1<<6=64, т. я. $1_{10}=1_2$, то якщо 1 зсунути на 6 розрядів вліво, то отримаємо 1000000, тоді $1000000_2=2^6=64_{10}$ (таким чином $1<<6=2^6$)	6<<2=24, т. я. $6_{10}=110_2$, то якщо 110 зсунути на 2 розряди вліво, то маємо 11000, тоді $11000_2=24_{10}$
>>	зсув вправо бітового представлення значення правого цілочисельного операнда на кількість розрядів, що дорівнює значенню правого операнда (звільнені зліва розряди обнуляються, а праві розряди зникають, якщо операнд беззнакового типу) і заповнюються знаковим розрядом, якщо знакового.	10>>1=2, т. я. $10_{10}=1010_2$, то 1010 на 1 розряд вправо $=0101_2=5_{10}$
Порозрядні операції		
&	порозрядна кон'юнкція (І) бітових представлень значень цілочисельних операндів (біт =1, якщо відповідні біти обох операндів=1, в інших випадках = 0).	6&5=4, $6_{10}=110_2$, $5_{10}=101_2$, 1&1=1, 1&0=0, 0&1=0, тобто $100_2=4_{10}$
|	порозрядна диз'юнкція (АБО) бітових представлень значень цілочисельних операндів (біт =1, якщо відповідний біт одного з операндів=1, якщо обидва =0, то і біт =0).	6|5=7, $6_{10}=110_2$, $5_{10}=101_2$, 1|1=1, 1|0=1, 0|1=1, тобто $111_2=7_{10}$

^	порозрядне виключне АБО бітових представлень значень цілочисельних операндів(біт =1, якщо відповідний біт тільки одного з операндів=1)	6^5=3, $6_{10}=110_2$, $5_{10}=101_2$, 1^1=0, 1^0=1, 0^1=1, тобто $11_2=3_{10}$
Операції порівняння: результатом є *true*(не 0) або *false*(0)		
<	менше, ніж	7<4=*false*
>	більше, ніж	5>1=*true*
<=	менше чи дорівнює	6<=6=*true*
>=	більше чи дорівнює	
==	дорівнює	7==2=*false*
!=	не дорівнює	5!=6=*true*
Логічні бінарні операції		
&&	кон'юнкція (І) цілочисельних операндів або відношень, істинна лише тоді, коли обидва операнди істинні.	1&&1=1; 1&&0=0; 0&&1=0; 0&&0=0;
||	диз'юнкція (АБО) цілочисельних операндів або відношень, є хибною лише тоді, коли обидва операнди помилкові.	1||1=1; 1||0=1; 0||1=1; 0||0=0;
Операції з присвоєнням		
	=, +=, −=, *=, /=, %=і т.д.	a+=10 ↔ a=a+10

Вирази в С++.

З символів, змінних, роздільників і знаків операцій можна конструювати ***вираз***. Кожен вираз являє собою *правило обчислення нового значення*. Будь-який вираз, за яким йде крапка з комою утворює *пропозицію* або *інструкцію мови*.

Якщо вираз повертає ціле або дійсне число, то він називається арифметичним. Пара арифметичних виразів, об'єднана операцією порівняння, називається ***відношенням***. Якщо відношення має ненульове значення, то воно *істинне*, інакше – *помилкове*.

Пріоритети операцій у виразах наводяться у наступній таблиці.

Пріоритет	**Операції**	**Зміст операції**

1	:: ( ) [ ] . –>	дужки
2	! ~ – ++ -- & * ^	унарні
3	* / %	мультиплікативні бінарні
4	+ –	адитивні бінарні
5	<< >>	порозрядного зсуву
6	< > <= >=	відношення
7	== !=	відношення
8	&	порозрядна кон'юнкція «І»
9	^	порозрядне виключне «АБО»
10	|	порозрядна диз'юнкція«АБО»
11	&&	логічна кон'юнкція «І»
12	||	логічна диз'юнкція «АБО»
13	? :	умовна операція
14	= *= /= %= -= &= ^= |= <<= >>=	операція з присвоєнням
15	,	оператор кома

Всі стандартні математичні функції в С++ описані в бібліотеці *cmath* (або, більш старіша, *math.h*). Тому, якщо в програмі використовуються якісь математичні функції, то на початку такої програми потрібно записати рядок підключення заголовкового файлу

#include<cmath> або, #include <math.h>

Примітка. Бібіліотека *cmath* є більш сучасною, працює стабільніше і містить більш розширений функціонал.

Основні математичні функції бібліотеки *cmath* наведені в наступній таблиці.

Назва функції	**Математичний запис**
***abs*(*x*)**	*x* (цілі числа)
***cos*(*x*)**	*cos x*
***sin*(*x*)**	*sin x*
***tan*(*x*)**	*tg x*

***log*(*x*)**	*ln x*
***pow*(*x*,*y*)**	$_x y$
***sqrt*(*x*)**	$x\sqrt{}$
***exp*(*x*)**	e^x
***pow*10(*x*)**	10^x
***log*10(*x*)**	*lg x*
***int* (*x*)**	відкидає дробову частину числа, але результат дійсний
***fabs*(*x*)**	*x* (дійсні числа)
***acos*(*x*)**	*arccos x*
***asin*(*x*)**	*arcsin x*
***atan*(*x*)**	*arctg x*
***ceil*(*x*)**	округлює число *x* до більшого цілого
***floor*(*x*)**	округлює число *x* до меншого цілого

Всі наведені функції, крім **abs**(*x*) і **pow10**(*x*), мають тип аргумента і результат ***double.*** Для функції **abs**(*x*) і **pow10**(*x*) типом аргумента и результатом є ***int.***

Приклад. Нехай оголошено змінні

```
int x = -2, x1, a = 3;
float pi = 3.1415926, m = 16, kut, k;
```

Тоді в результаті виконання наведених команд змінним *x1*, *a*, *kut*, *k*, *m* будуть присвоєні наступні значення:

x1=**abs**(*x*)	*x1*=**abs**(–2)=|-2|=2;
a=**pow10**(*a*)	*a*=**pow10**(3)=10^3=1000;
kut=**cos**(2**pi*)	*kut*=**cos**(2*π)=1;

Приклад. Нехай у програмі оголошені змінні

```
double b=7.6, b1, b2
```

Тоді після виконання команд

b1=***ceil****(b)*	*b1*=***ceil****(*7.6*)*=8

*b2=**floor**(b)*	*b2=**floor**(7.6)=7*

Всі інші математичні функції можна виразити через основні. Наприклад:

$$ctg\ x = 1/tg\ x,\ \log_b a = \ln a / \ln b$$

Послідовність виконання операцій у виразах така ж, як у математиці, і визначається пріоритетом виконання операцій.

Операції одного рівня виконуються послідовно зліва направо. Для зміни порядку виконання операцій використовують круглі дужки. Спочатку обчислюються вирази в дужках – в першу чергу у внутрішніх, потім – у зовнішніх. Кількість відкритих та закритих дужок у виразі має бути однаковим.

Всі елементи виразів (дроби, показник степеню, індекси) записують в горизонтальні рядки. У багатьох випадках їх беруть у квадратні дужки. Вирази можна записувати в декількох рядках. "Розривати" вирази можна, наприклад, після символу арифметичної операції, але власне символ дублювати не потрібно.

Потоки. Введення й виведення даних

На відміну від класичного стандарту мови, у мові C++ немає вбудованих засобів введення і виведення – він здійснюється з допомогою функцій, типів і об'єктів, які знаходяться в стандартних бібліотеках. Для організації введення-виведення тут реалізована концепція потоків, яка визначена в спеціальних модулях. У модулі ***istream.h*** описані *команди введення*, в модулі ***ostream.h*** – *команди виведення*, а в модулі ***iostream.h*** – команди введення і виведення. Тобто, при використанні бібліотеки класів C++, використовується бібліотечний файл *iostream.h*, в якому визначені стандартні потоки введення даних з клавіатури **cin** і виведення даних на екран монітора **cout**, а також відповідні операції

<< – операція запису даних у потік;
>> – операція читання даних з потоку.

Наприклад:

```
#include <iostream.h>;
. . . . . .
cout << "\n Введіть кількість елементів: ";
cin >> n;
```

Можливе використання традиційних функцій мови С – ***printf*** і ***scanf***. Під потоком розуміють процес вводу-виводу інформації в файл.

Периферійні пристрої вводу-виводу, такі як клавіатура, монітор, принтер, розглядаються як текстові файли. Під час виконання будь-якої програми підключаються стандартні потоки для введення даних з клавіатури (**cin**), виведення на екран (**cout**), виведення повідомлення про помилки (**cerr**) і допоміжний потоці (**clog**).

Стандартні потоки використовують операції введення (>>) і виводу (<<) даних. За замовчуванням стандартним пристроєм для потоків виводу даних і повідомлень про помилки є монітор, а для потоку вводу даних – клавіатура. Однак можна перенаправляти потоки, наприклад, можна зчитувати вхідну інформацію для програми не з клавіатури, а з деякого текстового файлу на диску.

Команда введення даних

Задавати значення змінних можна двома способами: за допомогою *команди присвоювання*, наприклад x=3.1, або команди введення даних з клавіатури. Команда введення даних з клавіатури дає можливість виконувати програму для різних вхідних даних, що робить її більш універсальною (масовою). Команда введення має такий загальний вигляд:

```
cin  >> <змінна>;
```

Дія команди. Виконання програми зупиняється. Система переходить у режим очікування введення даного (екран темний, блимає курсор). Користувач набирає на клавіатурі значення змінної і натискає на клавішу вводу. В результаті виконання цієї команди, змінної буде присвоєно конкретне значення (те, що користувач введе з клавіатури).

Якщо необхідно ввести значення відразу для декількох змінних, можна використовувати кілька потоків вводу, або записати всі змінні в одному потоці **cin**, застосувавши для цього кілька команд «>>», а саме:

```
cin >> < зміна 1> >> < зміна 2> >> ... >> < змінна N>;
```

Значення змінних можна вводити через пробіл або після введення кожного даного натискати клавішу Enter.

Якщо в списку введення (який набрали на клавіатурі через пропуск) даних більше, ніж змінних, то зайві дані будуть зчитані наступною

командою введення. Якщо такої команди в програмі немає, вони будуть проігноровані.

Перед командою введення даних варто записувати команди виводу на екран текстової підказки – повідомлення про те, що саме слід ввести.

Команда виведення даних

Для виводу на екран повідомлень і результатів обчислень використовують стандартний потік виводу **cout** і операцію <<. Команда виведення має такий загальний вигляд:

```
cout << <змінна>; або для виведення декількох змінних ->
cout << <вираз 1> << <вираз 2> << ... << <вираз N>;
```

У списку виводу можуть бути константи, змінні або виразу. Елементи списку в потоці ***cout*** відокремлюють операціями <<.

Дія команди. Константи, значення змінних і виразів, що виводяться на екран у вікно виведення. Це вікно можна переглянути за допомогою команди *Window → User screen* головного вікна компілятора або комбінації клавіш *Alt+F5*.

Текстові повідомлення в команді виведення записують в лапках.

Лапки на екран виводитися не будуть.

Для того, щоб дані виводилися в потрібному для користувача вигляді і не зливалися не екрані, використовують керуючі послідовності.

Керуючі послідовності

Керуючі послідовності (escape-послідовності) – це комбінації спеціальних символів, які використовуються для вводу і виводу даних. Керуюча послідовність складається з символу «зворотний» слеш "\" і наступного за ним символу. Вони призначені для форматного виведення результатів обчислень на екран, наприклад, для переходу на новий рядок, звукового сигналу, а також для виведення на екран деяких спеціальних символів: апостроф, лапки тощо. Основні керуючі послідовності наведені в таблиці.

Символи керуючих послідов-ностей	Коментар
\a, \7	Подати звуковий сигнал
\b	Повернути курсор на один символ назад (знищити по-передній символ)
\f	Перейти на нову сторінку
\n	Перейти на новий рядок
\r	Повернути курсор на початок рядка
\t	Перевести курсор на наступну позицію табуляції
\v	Вертикальна табуляція
\\	Вивести символ похилої риски
\'	Вивести символ апострофа
\"	Вивести символ лапок
\?	Вивести знак питання

Керуючі послідовності разом з коментарями записують в лапках.

Розглянемо дію цих послідовностей на прикладі. Якщо записати команди

```
cout << "Увага! \a\n ";
cout << "Дане \"a\" введено некоректно \n";
cout << "Виконати програму ще раз";
```

то буде поданий звуковий сигнал і на екрані побачимо:

```
Увага!
Дане "a" введено некоректно
Виконати програму ще раз
```

Зауваження 1. Замість керуючої послідовності \n можна використовувати команду **endl** – кінець рядка. Наприклад, ці дві команди рівносильні:

cout << "f=" << *f* << "\n"; і ***cout*** << "f=" << *f* << *endl*;.

Як ви вже знаєте значення дійсних чисел (тип float або double) можна виводити на екран в стандартному або науковому форматах. Якщо значення даного необхідно округляти до n значущих цифр, то перед командою виведення потрібно записати **cout.precision(n).** Ця команда дасть можливість визначити кількість нових знаків після коми. Для подання результатів у науковому форматі необхідно під'єднати заголовковий файл **iomanip.h** і перед командою

виведення записати

```
cout << setiosflags(ios::scientific);
```

Для задання розміру поля, в якому буде показана зміна можна використовувати команду **cout.width(n);** Виведене число буде притискатися до правого краю поля, тобто «зайві» знаки в полі будуть заповнені пробілами перед виведеним числом. Якщо вам потрібна така функція, то її потрібно викликати перед кожним використанням команди ***cout***.

При виведенні на **cout** або **cerr** програми можна вказати ширину виводу кожного числа, використовуючи модифікатор **setw** (установка ширини). З допомогою setw програми вказують мінімальну кількість символів, займане числом. Щоб використовувати модифікатор setw, ваша програма повинна включати заголовковий файл *iomanip.h*. Модифікатор setw(n) використовується в потоці виводу безпосередньо перед виведеним числом.

Класи пам'яті

Загальний вигляд оператора опису змінної:

[клас пам'яті][const]тип ім'я [инициализатор];

[] – такі дужки означають, що даний параметр може бути в цьому місці, але він не обов'язковий.

Клас пам'яті може приймати значення: **auto, extern, static, register**. Клас пам'яті визначає час життя і область видимості змінної. Якщо клас пам'яті не вказаний явно, то компілятор визначає його виходячи з контексту оголошення. Час життя може бути постійним, – протягом виконання програми або тимчасовим – протягом блоку.

Область видимості – частина тексту програми, з якої припустимо звичайний доступ до змінної. Зазвичай область видимості збігається з областю дії. Крім того випадку, коли у внутрішньому блоці існує змінна з таким же ім'ям.

const – показує, що цю змінну не можна змінювати (іменована константа). При описі можна присвоїти початкове значення змінної (ініціалізація).

Класи пам’яті:

auto – автоматична локальна змінна. Специфікатор auto може бути заданий тільки при визначенні об'єктів блоку, наприклад, в тілі функції. Цим змінним пам'ять виділяється при вході в блок і звільняється при виході з нього. Поза блоку такі змінні не існують.

extern – глобальна змінна, вона знаходиться в іншому місці програми (в іншому файлі або далі по тексту). Використовується для створення змінних, які доступні у всіх файлах програми.

static – статична змінна, вона існує тільки в межах того файлу, де визначена змінна.

register – аналогічний *auto*, але пам'ять під такі змінні виділяється в регістрах процесора. Якщо такої можливості немає, то змінні обробляються як auto.

Приклад:

```
int a;    //глобальна зміна
void main()
{
int b;    //локальна змінна
extern int x; //зміна x визначена в іншому місці
static int c; //локальна статична змінна
a=1; //ініціалізація глобальної змінної
int a;    //локальна змінна a
a=2; //ініціалізація локальної змінної
::a=3;    //ініціалізація глобальної змінної
}
int x=4; // оголошення та ініціалізація x
```

У прикладі змінна *a* визначена поза всіх блоків. Областю дії змінної *a* є вся програма, крім тих рядків, де використовується локальна змінна *a*. Змінні *b* і *c* – локальні, область їх видимості – блок. Час життя по різному: пам'ять під *b* виділяється при вході в блок (т. к. за замовчуванням клас пам'яті **auto**), звільняється при виході з нього. Змінна (***static***) існує, поки працює програма.

Якщо при визначенні початкове значення змінним не задається явно, то компілятор обнуляє глобальні та статичні змінні. Автоматичні змінні не ініціалізуються.

Ім'я змінної повинно бути унікальним в своїй області.

Опис змінної може бути виконано або як оголошення, або як визначення. Оголошення містить інформацію про клас пам'яті і тип змінної, визначення разом з цією інформацією дає вказівку виділити пам'ять. У прикладі **extern int x**; – оголошення, а решта – визначення.

Функція стандартного виводу printf()

Функція **printf ()** - функція стандартного виводу. З допомогою цієї функції можна вивести на екран рядок символів, число, значення змінної.

Функція **printf()** має прототип у файлі *stdio.h*, тобто для використання цієї функції потрібно підключити зазначену бібліотеку директиви препроцесора.

Керуючий рядок містить два типи інформації: символи, які безпосередньо виводяться на екран, і специфікатори формату, що визначають, як виводити аргументи.

Функція **printf()** це функція форматованого виводу. Це означає, що в параметрах функції необхідно вказати формат даних, які будуть виводитися. Формат даних вказується специфікаторами формату. Специфікатор формату починається з символу % за яким слід код формату. **Специфікатори формату:**

%c	символ
%d (%i)	ціле десяткове число
%e (%E)	десяткове число у вигляді x.xx e+xx
%f (%F)	десяткове число з плаваючой комою xx.xxxx
%o	восьмирічне число
%u	беззнакове десяткове число
%x (%X)	шістнадцятирічне число
%%	символ %
%p (%n)	вказівник

Крім того, до команд формата можуть бути застосовані модифікатори *l* і *h*.

%ld	друк long int і long long
%hu	друк short unsigned
%Lf	друк long double

У специфікаторі формату, після символу % може бути вказана точність (кількість цифр після коми). Точність визначається наступним чином:

%m.n<код формату>.

де m – загальна кількість позицій виводу, n – число цифр після коми, а <код формату> – один з специфікаторів наведених вище.

Наприклад, є змінна x=10.35635 типу float і потрібно вивести її значення з точністю до 3-х цифр після коми, то потрібно написати:

printf("Змінна x = %.3f", x); Результат: Змінна x = 10.356

І можна вказати мінімальну ширину поля відведеного для друку. Якщо рядок або число більше вказаної ширини поля, рядок або число друкується повністю.

Наприклад, якщо написати: printf("%d 5.0",20); то результат буде

наступним: _ _ _ 20

Зверніть увагу на те, що число 20 надрукувалось не з самого початку рядка. Якщо потрібно щоб невикористані місця поля заповнювалися нулями, то потрібно поставити перед шириною поля символ 0.

Наприклад: printf("%05d",20); Результат: 00020

Крім специфікаторов формату даних у керуючому рядку можуть перебувати керуючі символи:

\b	BS, забій
\f	Нова сторінка, перевід сторінки
\n	Новий рядок, перевід рядка
\r	Повернення каретки
\t	Горизонтальна табуляція
\v	Вертикальна табуляція
\"	Подвійні лапки
\'	Апостроф
\\	Зворотна похила риска
\0	Нульовий символ, нульовий байт
\a	Сигнал
\N	Восьмирічна константа
\xN	Шістнадцятирічна константа
\?	Знак питання

Найчастіше використовується символ \n. За допомогою цього керуючого символу здійснюється перехід на новий рядок.

Приклади програм.

/* **Приклад 1** */

```
#include <stdio.h>
 int main()
{
int a, b, c; a=5; b=6; c=9;
printf("a=%d, b=%d, c=%d", a, b, c);
}
```

```
Результат роботи програми:
a=5, b=6, c=9
```

/* **Приклад 2** */

```
#include <stdio.h>
 int main()
{
float x, y, z;
x=10.5; y=130.67; z=54;
printf("Координати об'єкта: x: %.2f, y:%.2f, z:%.2f", x, y,
z);
}
```

```
Результат роботи програми:
Координати об'єкта: x: 10.50, y:130.67, z:54.00
```

/* **Приклад 3** */

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

```
Результат роботи програми:
x=10
```

/* **Приклад 4** */

```
#include <stdio.h>
int main()
{
printf("\"Текст в лапках\"");
printf("\n Вміст кисню: 100%");
}
```

```
Результат роботи програми:
"Текст в лапках" Вміст кисню: 100%
```

/* **Приклад 5** */

```c
#include <stdio.h>
int main()
{
int a;
a=11;     // 11 в десятковій дорівнює b в шістнадцятковій
printf("a-dec=%d, a-hex=%X", a, a);
}
```

```
Результат роботи програми:
a-dec=11, a-hex=b
```

/* **Приклад 6** */

```c
#include <stdio.h>
int main()
{
char ch1, ch2, ch3; ch1='A';
ch2='B';
ch3='C';
printf("%c%c%c", ch1, ch2, ch3);
}
```

```
Результат роботи програми:
ABC
```

/* **Приклад 7** */

```c
#include<stdio.h>
 int main()
{
char *str="Мій рядок.";
printf("Это %s", str);
```

```
}
```

```
Результат роботи програми:
Мій рядок.
```

/* **Приклад 8** */

```
#include  <stdio.h>
int main()
{
printf("Добрий день!\n"); // Після виводу - перехід на новий рядок
printf("Мене звати Павло."); // Це буде виводитись на новому рядку
}
```

```
Результат роботи програми:
Добрий день! Мене звати Павло.
```

Функція стандартного вводу scanf()

Функція **scanf()** – функція форматованого вводу. З її допомогою можна вводити (зчитувати дані зі стандартного пристрою вводу. Введеними даними можуть бути цілі числа, числа з плаваючою комою, сиволи, рядки і покажчики.

Функція **scanf()** має прототип у файлі stdio.h, тобто його треба підключити.

Керуючий рядок містить три види символів: специфікатори формату, прогалини та інші символи. Специфікатори формату починаються з символу %.

Специфікатори формату:

%c	читання символа
%d (%i)	читання десяткового цілого
%e	читання числа типу float (плаваюча кома)
%h	читання short int
%o	читання восьмирічного числа

%s	читання рядка
%x	читання шістнадцятирічного числа
%p	Читання вказівника
%n	Читання вказівника в збільшеному форматі

При читанні рядка за допомогою функції **scanf()** (специфікатор формату %s), рядок зчитується до першого пробілу!! тобто якщо ви вводите рядок "Привіт, світ!" з використанням функції **scanf()**

```
char str[80]; // масив на 80 символів
scanf("%s",str);
```

Після зчитування результуючий рядок, яка буде зберігатися в масиві str буде складатися з одного слова "Привіт". ФУНКЦІЯ ВВОДИТЬ РЯДОК ДО ПЕРШОГО ПРОБІЛУ! Якщо потрібно зчитувати рядки з пробілами, то використовується функція gets(char *str).

За допомогою функції **gets()** можна зчитувати повноцінні рядки. Функція gets() читає символи з клавіатури до появи символу нового рядка (\n). Сам символ нового рядка з'являється, коли натискається клавіша Enter.

Приклад програми, яка дозволяє ввести цілий рядок з клавіатури і вивести її на екран.

```
#include <stdio.h>
int main()
{
char buffer[100]; // масив (буфер) для рядка, що водиться
gets(buffer); // вводимо рядок и натискаємо Enter
printf("%s", buffer); // вивід введеного рядка на екран
}
```

Ще одне важливе зауваження! Для введення даних за допомогою функції **scanf()**, їй у якості параметрів потрібно передавати адреси змінних, а не самі змінні. Щоб отримати адресу змінної, потрібно поставити перед ім'ям змінної знак &(амперсанд). Знак & означає взяття адреси.

Якщо в програмі є змінна, то вона зберігає своє значення в пам'яті комп'ютера. Адреса, яку ми отримуємо за допомогою & – це адреса в пам'яті комп'ютера, де зберігається значення змінної.

Розгянемо приклад програми, який показує, як використовувати

&:

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

```
int x;
printf("Введіть змінну x:"); scanf("%d",&x);
printf("Змінна x=%d", x);
}
```

Тепер повернемося до керуючої рядку функції scanf().

Символ пробілу в керуючому рядку дає команду пропустити один або більше прогалин у потоці вводу. Крім пробілу може сприйматися символ табуляції або новий рядок. Ненульовий символ вказує на читання і відкидання цього символу.

Роздільниками між двома введеними числами є символи пробілів, табуляції або новий рядок. Знак * після % і перед кодом формату (специфікатором формату) дає команду прочитати дані зазначеного типу, але не присвоювати значення.

Наприклад: **scanf**("%d%*c%d", &i, &j); при введенні 50+20 присвоїть змінній і значення 50, змінної j – значення 20, а символ " + " буде прочитаний і проігнорований.

У команді формату може бути вказана найбільша ширина поля, яка підлягає зчитуванню.

Наприклад: scanf("%5s", str); вказує необхідність прочитати з потоку вводу перші 5 символів. При введенні 1234567890ABC масив str буде містити тільки 12345, решта символи ігноруються. Роздільники: пробіл, символ табуляції, символ нового рядка при введенні символу сприймаються, як і всі інші символи.

Якщо в керуючому рядку зустрічаються якісь інші символи, то вони призначаються для того, щоб визначити і пропустити відповідний символ. Потік символів 10plus20 оператором **scanf**("%dplus%d", &x, &y); присвоїть змінній x значення 10, змінної y – значення 20, а символи plus пропустить, так як вони зустрілися в керуючій рядку.

Однією з потужних особливостей функції **scanf()** є можливість задання множини пошуку (scanset). Множина пошуку визначає набір символів, з якими будуть порівнюватися символи scanf(), що читаються функцією. Функція scanf() читає символи до тих пір, поки вони зустрічаються у безлічі пошуку. Як тільки символ, який введений, не зустрівся в множині пошуку, функція scanf() переходить до наступного специфікатору формату. Множина пошуку визначається списком сиволів, укладених у дужки. Перед відкриваючою дужкою ставитися знак %. Розглянемо це на прикладі.

```
#include <stdio.h>
int main()
{
char str1[10], str2[10];
scanf("%[0123456789]%s", str1, str2);
```

```
printf("\n%s\n%s", str1, str2);
}
```

Введемо набір символів:

```
12345abcdefg456
```

На екрані програма видасть:

```
12345
abcdefg456
```

При заданні безлічі пошуку можна також використовувати символ "дефіс" для завдання проміжків, а також максимальну ширину поля вводу.

```
scanf("%10[A-Z1-5]", str1);
```

Можна також визначити символи, які не входять в безліч пошуку. Перед першим з цих символів ставитися знак ^. Безліч символів розрізняє малі та великі літери.

Нагадаємо, що при використанні функції scanf(), їй у якості параметрів потрібно передавати адреси змінних. Вище був написаний код:

```
char str[80]; // масив на 80 символів
scanf("%s", str);
```

Зверніть увагу на те, що перед str не стоїть символ &. Це зроблено тому, що str є масивом, а ім'я масиву – str є вказівник на перший елемент масиву. Тому знак & не ставиться. Ми вже передаємо функції scanf() адреса.

Приклади програм

Приклад 1. Ця програма виводить на екран запит "Скільки вам років?:" і чекає введення даних. Якщо, наприклад, ввести число 20, то програма виведе рядок "Вам 20 років.". При виклику функції scanf(), перед змінній " age " ми поставили знак &, так як функції scanf() потрібні адреси змінних. Функція scanf() запише введене значення за вказаною адресою. У нашому випадку введене значення 20 буде записано за адресою змінної age.

```
#include <stdio.h>
    int main()
{
int age;
printf("\n Скільки вам років?:");
scanf("%d",&age);
printf("Вам %d років.", age);
}
```

Приклад 2.

Програма-калькулятор. Цей калькулятор може тільки додавати числа. При вводі 100+34 програма видасть результат: 100+34=134.

```
#include <stdio.h>
    int main()
{
int x, y; printf("\nКалькулятор:"); scanf("%d+%d", &x, &y);
printf("\n%d+%d=%d", x, y, x+y);
}
```

Приклад 3.

Цей приклад показує, як встановити ширину поля зчитування. У нашому прикладі ширина поля дорівнює п'яти символів. Якщо вводиться рядок з великою кількістю символів, то всі символи після 5-го будуть відкинуті. Зверніть увагу на виклик функції **scanf().** Знак & не стоїть перед ім'ям масиву *name* так як ім'я масиву *name* є адреса першого елемента масиву.

```
#include <stdio.h>
    int main()
{
char name[5];
printf("\nВведіть ваш логін (не більше 5 символів):");
scanf("%5s", name);
printf("\nВи ввели %s", name);
}
```

Приклад 4.

Приклад показує, як можна використовувати множину пошуку. Після запуска програми вводиться число от 2 до 5.

```
#include <stdio.h>
    int main()
{
char bal;
printf("Ваша оцінка 2,3,4,5:");
scanf("%[2345]", &bal);
printf("\nОцінка %c", bal);
}
```

ЗМІСТОВИЙ МОДУЛЬ 2
БАЗОВІ АЛГОРИТМІЧНІ СТРУКТУРИ У МОВІ С++

Тема 2.1 Розгляд конструкцій умовних операторів, розгалуження та циклів у програмуванні

Розв'язання задач з використанням основних операторів С ++

«Початкові програмісти, особливо студенти, часто пишуть програми так: отримавши завдання, тут же сідають за комп'ютер і починають кодувати ті фрагменти алгоритму, які їм вдається придумати відразу. Змінним дають перші-ліпші імена типу x і y. Коли комп'ютер зависає, робиться перерва, після якого все написане стирається, і все повторюється заново. Періодично висловлюються сумніви в правильності роботи компілятора, комп'ютера і операційної системи. Коли програма доходить до стадії виконання, в неї вводяться довільні значення, після чого екран стає об'єктом пильного здивованого вивчення. «Працює» така програма зазвичай тільки в дбайливих руках господаря на одному наборі даних, а внесення в неї змін може привести автора до втрати віри в себе і ненависті до процесу програмування.

Але завдання полягає в тому, щоб навчитися підходити до програмування професійно. Зрештою, професіонал відрізняється тим, що може досить точно оцінити, скільки часу у нього займе написання програми, яка буде працювати в повній відповідності з поставленим завданням. Крім

«розуму, смаку і терпіння», для цього потрібен досвід, а також знання основних принципів, вироблених програмістами протягом більш, ніж півстоліття розвитку цієї дисципліни. Навіть до написання найпростіших програм потрібно підходити послідовно, дотримуючись певної дисципліни.

Рішення задач з програмування передбачає ряд етапів:

1. Розробка математичної моделі. На цьому етапі визначаються вихідні дані і результати виконання завдання, а також математичні формули, за допомогою яких можна перейти від вихідних даних до кінцевого результату.
2. Розробка алгоритму. Визначаються дії, виконуючи які можна буде від вихідних даних прийти до необхідного результату.
3. Запис програми на деякій мові програмування. На цьому етапі кожному кроці алгоритму ставиться у відповідність конструкція обраної алгоритмічної мови.
4. Виконання програми (вихідний модуль -> компілятор -> об'єктний модуль -> компоновщик -> виконуваний модуль)
5. Тестування і налагодження програми.

При виконанні програми можуть виникнути помилки 3 типів:

a. синтаксичні - виправляються на етапі компіляції;

b. помилки виконання програми (ділення на 0, логарифм від негативного числа і т. п.) - виправляються при виконанні програми;

c. семантичні (логічні) помилки - з'являються через неправильно

зрозуміле завдання, неправильно складений алгоритм.

Щоб усунути ці помилки програма повинна бути виконана на деякому наборі тестів. Мета процесу тестування - визначення наявності помилки, знаходження місця помилки, її причини та відповідні зміни програми - виправлення. Тест - це набір вихідних даних, для яких заздалегідь відомий результат. Тест, який виявив помилку, вважається успішним. Налагодження програми закінчується, коли достатня кількість тестів виповнилася успішно, тобто програма на них видала правильні результати.

Для визначення достатньої кількості тестів існує два підходи. При першому підході програма розглядається як «чорний ящик», в який передають вихідні дані і отримують результати. Пристрій самого ящика невідомо. При цьому підході, щоб здійснити повне тестування, треба перевірити програму на всіх вхідних даних, що практично неможливо. Тому вводять спеціальні критерії, які повинні показати, яка кінцева множина тестів є достатнім для програми. При першому підході найчастіше використовуються наступні критерії:

1. Тестування класів вхідних даних, тобто набір тестів повинен містити по одному представнику кожного класу даних.
2. Тестування класів вихідних даних, набір тестів повинен містити дані достатні для отримання по одному представнику з кожного класу вихідних даних.

При другому підході програма розглядається як «білий ящик», для

якого повністю відомий пристрій. Повне тестування при цьому підході закінчується після перевірки всіх шляхів, що ведуть від початку програми до її кінця. Однак і при такому підході повне тестування програми неможливо, так як шляхів в програмі з циклами безліч. При такому підході використовуються наступні критерії:

1. Тестування команд. Набір тестів повинен забезпечувати проходження кожної команди не менше одного разу.
2. Тестування гілок. Набір тестів в сукупності має забезпечити проходження кожної гілки не менше одного разу. Це найпоширеніший критерій в практиці програмування.

Лексеми

Коли компілятор обробляє програму, він розбиває програму на групи символів, які називаються лексемами. *Лексема* - це мінімальна одиниця тексту програми, яка має певний сенс для компілятора і яка не може бути розбита в подальшому. Операції, константи, ідентифікатори і ключові слова, описані раніше, є прикладами лексем. Знаки пунктуації, такі як квадратні дужки ([]), фігурні дужки ({}), кутові дужки (<>), круглі дужки і коми, також є лексемами. Межі лексем визначаються пробільними символами та іншими лексемами, такими як операції і знаки пунктуації. Щоб попередити неправильну роботу компілятора, забороняються пробільні символи між символами ідентифікаторів, операціями, що складаються з декількох символів і символами ключових слів.

Коли компілятор виділяє окрему лексему, він послідовно об'єднує стільки символів, скільки можливо, перш ніж перейти до обробки наступної лексеми. Тому лексеми, не розділені пробільними символами, можуть бути проінтерпретовані невірно.

Наприклад, розглянемо наступний вираз: i+++j

У цьому прикладі компілятор спочатку створює з трьох знаків плюс найдовшу з можливих операцій (++), а потім обробить знак, що залишився +, як операцію додавання (+). Вираз проінтерпретують як (i ++) + (j), а не як (i) + (++ j). У таких випадках необхідно використовувати пробільні символи або круглі дужки, щоб однозначно визначити ситуацію.

Пробільні символи

Пробіл, табуляція, переведення рядка, повернення каретки, нова сторінка, вертикальна табуляція і новий рядок - це символи, звані пробільними, оскільки вони мають те ж саме призначення, як і пропуски між словами і рядками на друкованій сторінці. Ці символи поділяють об'єкти, визначені користувачем, такі, як константи і ідентифікатори, від інших об'єктів програми.

Компілятор C ++ ігнорує пробільні символи, якщо вони не використовуються як роздільники або як компоненти константи-символу або рядкових літералів. Це потрібно мати на увазі, щоб додатково використовувати пробільні символи для підвищення наочності програми (наприклад, для перегляду редактором текстів).

Порожній оператор

Найпростішою формою оператора є оператор: «;»

Він не робить нічого. Однак він може бути корисний в тих випадках, коли синтаксис вимагає наявність оператора, а вам оператор не потрібен. Він зазвичай використовується в наступних випадках:

- в операторах do, for, while, if в рядках, коли місце оператора не потрібно, але по синтаксису потрібно хоча б один оператор;
- при необхідності позначити фігурну дужку.

Синтаксис мови C++ вимагає, щоб після мітки обов'язково слідував оператор. Фігурна ж дужка не є оператором. Тому, якщо треба передати управління на фігурну дужку, необхідно використовувати порожній оператор.

Приклад:

```
#include <iostream>

int main()
{
    int matrix[3][3] = {
        {1, 2, 3},
```

```
        {4, 5, 6},
        {7, 8, 9} // 7 - це потрібне число
    };

    bool found = false;

    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {

            if (matrix[i][j] % 7 == 0) {
                // Знайшли потрібний елемент!
                std::cout << "Знайдено число, що ділиться на
7, у позиції [" << i << "][" << j << "]" << std::endl;

                // Використовуємо goto для виходу з обох
циклів
                goto end_loops;
            }
        }
    }

    // Мітка end_loops розташована одразу перед фігурною
дужкою, що закривається.
    // Оскільки фігурна дужка {} не є оператором, ми додаємо
порожній оператор ;
    // щоб задовольнити синтаксис C++.
end_loops:;

    // Код, який виконується після дострокового виходу або
нормального завершення циклів
    std::cout << "Програма продовжила виконання після
циклів." << std::endl;

    return 0;
}
```

Блоки

Блок – це список операторів (можливо пустий), укладений у фігурні

дужки:

```
{
a=b+2; b++;
}
```

Блок дозволяє розглядати кілька операторів як один. Крім того, в блоці можна визначити змінну, яка буде автоматично знищена при виході з блоку.

Опис

Опис - це оператор, який запроваджує ім'я в програмі. Опис задає тип цього імені. Тип визначає правильне використання імені або виразу. Опис може також ініціювати об'єкт з цим ім'ям. Виконання опису означає, що коли потік управління доходить до опису, обчислюється ініціалізований вираз (ініціалізатор) і виробляється ініціалізація.

Літерали

Літерали (*literals*) – це постійні значення, такі як 1 або 3.14159 (π). Для кожного типу C ++ існують літерали, включаючи символьний і булевський типи, цілі числа і числа з плаваючою крапкою. Можливі рядкові літерали, хоча типу для зберігання рядків в C ++ не існує.

Деякі приклади:

5	ціла константа
5u	u або U означає без знакові константи
5l	l або L означає **long**
true	логічна константа
5.0	константа з плаваючою точкою, розуміється як **double**
5.0f	f або F — з плаваючою точкою, розуміється як **float**
0.3e-2	константа з плаваючою точкою **double**, e або E відокремлюють експонентну частину
5.0l	l або L в даному випадку розуміється как long double
'd'	символьна константа
"Visual"	рядкова константа

Ви звернули увагу при розгляді типів даних, що серед розглянутих типів даних відсутній «строковий» тип, на відміну від Pascal. Справа в тому, що компілятори C ++ підтримують лише рядкові літерали. З самим **поняттям строковий літерал** ви вже добре знайомі. Наприклад, в операторі cout <<

"Hello, World!"; використовується строковий літерал «Hello, World!». Іншими словами, строковий літерал - це набір довільних символів, укладений в лапки. Компілятор сприймає його саме як набір сиволів і ніяк обробляти його не збирається, навіть якщо в лапках виявляться якісь ключові слова і операції. Винятком є використання **escape**- послідовностей (керуючих послідовностей).

Базові конструкції структурного програмування.

У теорії програмування доведено, що програму для вирішення завдання будь-якої складності можна скласти тільки з трьох структур: лінійної, розгалуженої і циклічної. Ці структури називаються базовими конструкціями структурного програмування.

Лінійною називається конструкція, що представляє собою послідовне з'єднання двох або більше операторів.

Розгалуження - задає виконання одного з двох операторів, в залежності від виконання будь-якого умови.

Цикл - задає багаторазове виконання одного і того ж оператора або групи операторів.

Слідування. Розгалуження. Цикл.

Метою використання базових конструкцій є отримання програми простої структури. Таку програму легко читати, налагоджувати і при необхідності вносити в неї зміни. Структурне програмування також називають програмуванням без goto, так як часте використання операторів переходу ускладнює розуміння логіки роботи програми. Але іноді зустрічаються ситуації, в яких застосування операторів переходу, навпаки, спрощує структуру програми.

Оператори управління роботою програми називають керуючими конструкціями програми. До них відносять:

- складові оператори;
- оператори умови;
- оператори циклів;
- оператори переходу.

Оператор «вираз»

Будь-який вираз, що закінчується крапкою з комою, розглядається як оператор, виконання якого полягає в обчисленні цього виразу. Окремим випадком виразу є *порожній оператор*; (крапка з комою).

Приклади:

i++; *a*+=2; *x*=*a*+*b*;

Оператор вираз

Будь-який вираз, який закінчується крапкою з комою, є оператором. Виконання оператора виразу полягає в обчисленні виразу. Отримане значення виразу ніяк не використовується, тому, як правило, такі вирази викликають побічні ефекти. Зауважимо, що викликати функцію, неповернутих значення можна тільки за допомогою оператора вираження.

Приклади:

++i; – цей оператор представляє вираз, який збільшує значення змінної *i* на одиницю.

a=***cos***(*b**5); – цей оператор представляє вираз, який включає в себе операції присвоювання і виклика функції.

a(*x, y*); – цей оператор представляє вираз, який викликає функцію ***a***.

Оператори умови

Всі приклади, розглянуті раніше, виконувалися в порядку проходження операторів, що виконуються обов'язково по одному разу. Однак на практиці можливості таких програм досить обмежені. Більшість завдань вимагає від програми прийняття рішень в залежності від різних ситуацій. Мова С ++ має багато конструкцій для управління порядком виконання гілок програми. Для здійснення розгалуження використовуються оператори умови.

Оператори умови - це умовний оператор і оператор вибору (селектор). Вони застосовуються для перевірки деяких умов.

Умовний оператор

Умовний оператор має *повну* (оператор **if-else**) та *скорочену* (оператор **if**) форму.

Скорочена форма (оператор **if**): **if (**вираз-умова**)** *оператор*; В якості виразу-умови можуть використовуватися.

- арифметичні вирази,
- відношення з операціями порівняння,
- логічний вираз.

Якщо значення виразу-умови відмінне від нуля (тобто істинно), то виконується *оператор*, в іншому випадку не виконується *нічого* і управління передається наступному після **if** оператору.

Наприклад:

```
if (x <y && x <z) min = x;
```

В даному випадку, якщо x = 5, y = 6, z = 7, тобто x <y і x <z і вираз-умова істинно (приймає значення *true*), тоді виконується оператор *min* = x і змінна min приймає значення 5. якщо ж x = 9, y = 6, z = 11, тобто x не «y, але x <z, тоді вираз-умова помилкова (приймає значення false) і оператор взагалі не виконується, тобто значення min не зміниться.

Оператор може бути *блоковим*, тобто, в залежності від прийнятого рішення виконується не один, а цілий блок операторів. Весь вміст блоку розглядається компілятором мови як один оператор.

Однією з типових помилок при використанні оператора **if** є пропуск фігурних дужок для позначення блоку виконуваних операторів.

```
int main()
{
int x=0;
int y=8;
int z=0;
if(x!=0) z++;
y/=x; // Помилка!!! Ділення на 0!!!
cout<<"x="<<x<<"  y="<<y;
}
```

```
int main()
{
int x=0; int y=8; int z=0;
if(x!=0)
{
z++;
y/=x; // Помилки немає!
}
cout<<"x="<<x<<"   y="<<y;
}
```

У лівій колонці блок після **if** не використовується, тому виконується тільки оператор z ++; і далі, без всяких умов y / = x, що призводить до помилки ділення на нуль. У правій колонці, завдяки блоку, такий розподіл виконується тільки в разі x! = 0, що допомагає уникнути помилки.

Перевірка на нульове значення використовується дуже часто в програмуванні.

```
Повна форма (оператор if-else):
if (вираз-умова) оператор1; else оператор2;
```

Якщо значення виразу-умови істинно (*true*, відмінно від нуля), то виконується *оператор1*, при помилковому (нульовому, *false*) значення виразу-умови виконується *оператор2*.

Слід зазначити, що комбінація **if-else** дозволяє значно спростити код програми.

Наприклад:

```
if (d>=0)
{
x1=(-b-sqrt(d))/(2*a);
x2=(-b+sqrt(d))/(2*a);
cout << "\n x1=" << x1 << "x2=" << x2;
}
else cout << "\nРозв'язків немає";
```

Типовою помилкою програмістів є використання в умовних конструкціях оператора присвоювання замість оператора порівняння (= замість ==), що не приводить до помилки компілятора, але веде до невірного результату.

На практиці часто перевіряється умова являє собою перевірку значення деякої цілочисельний змінної на нульове значення, тому часто зустрічаються записи виду if (! X) ..., що означає **if** (x == 0), або **if** (x) ..., що означає **if** (x ! = 0)

Існує **альтернативна форма** оператора умови (тернарний оператор):

```
(вираз-умова)? оператор1 : оператор2;
```

Його дія аналогічна повній формі розгалуження: якщо значення виразу-умови істинно (відмінно від нуля), то виконується *оператор1*, при помилковому (нульовому) значення виразу-умови виконується *оператор2*.

Повна і альтернативна форми оператора умови абсолютно аналогічні, тому вибір використання тієї чи іншої форми - це особиста справа програміста, але альтернативну форму краще використовувати, якщо входять оператори досить прості.

Наприклад:

```
max=(a>b)? a:b;
```

Оператор вибору switch (селектор)

Ще однією альтернативою умовного оператора може служити оператор вибору **switch**. Він використовується в тих випадках, коли необхідно проводити порівняння деякої змінної з великою кількістю однотипних змінних або літералів. Ця конструкція являє собою своєрідний перемикач. Перемикач визначає множинний вибір.

```
switch (вираз-селектор)
{
case константа1 : оператор1;
case константа2 : оператор2; break;
. . .
default: оператори;
}
```

При виконанні оператора **switch**, аналізується вираз-селектор, записаний після **switch**, він повинен бути цілочисельним.

Отримане значення послідовно порівнюється з константами, які записані слідом за **case**. При першій збіжності виконуються оператори помічені даною міткою. Якщо виконані оператори не містять оператора переходу **breake**, то далі виконуються оператори всіх наступних варіантів, поки не з'явиться оператор переходу або не закінчиться перемикач. Якщо значення виразу, записаного після **switch** не співпало з жодною константою, то виконуються оператори, які слідують за міткою **default**. Мітка **default** може бути відсутньою. Ключове слово **break** не є обов'язковим, але якщо воно відсутнє, то відбувається перегляд всіх варіантів, що сильно уповільнює роботу програми.

Якщо при виконанні різних констант слідує один і той же оператор, то їх можна об'єднувати в такий спосіб:

```
case константа1:
case константа2:
case константа3:
case константа4:
оператор1;
```

Оператори циклів.

Наступним потужним механізмом управління ходом послідовності виконання програми є використання циклів.

Цикл (повторення) - це процес виконання певного набору команд деяку кількість разів. Він має точку входу, перевіряючи умову і (необов'язково) точку виходу. Цикл, що не має точки виходу, називається нескінченним. Для нескінченного циклу перевіряючи умову завжди приймає істинне значення. Цикли можуть бути вкладеними один в одного довільним чином.

У мові С ++ є три оператора циклу - **for** (цикл з параметром), **while** (цикл з передумовою) і **do-while** (цикл з постумовою).

Група дій, що повторюються в циклі, називається його тілом. Одноразове виконання циклу називається його кроком.

Цикл з передумовою (while).

Оператор циклу **while** виконує оператор або блок до тих пір, поки перевіряючи умову (вираз) залишається істинним. Він повинен виглядати так:

```
while (вираз-умова)
оператор;
```

В якості виразу-*умови* може використовуватися:

- відношення,
- логічний вираз,
- константа

– змінна логічного типу.

Якщо воно істинне, тобто не дорівнює 0, то тіло циклу виконується до тих пір, поки вираз-умова *істинна* (не стане *хибним*). Якщо потрібно перевірити кілька умов, то застосовують оператор *«кома».* Оператор в даному циклі може бути порожнім, простим або складеним. Для того, щоб стався вихід з циклу, необхідно *змінювати* значення параметра в циклі в операторі. *Параметр циклу* - це дане, яке входить у вираз-умову.

Дія команди.

1. Обчислюються значення виразу-умови. Якщо значення виразу- умови *істинне*, то переходимо до пункту 2), якщо *хибне* - до пункту 3).
2. Виконується оператор і відбувається перехід до пункту 1).
3. Виконується перехід до наступної після **while** команди.

Циклу **while** необхідний початковий оператор, який ініціалізує змінну управління циклом (параметр, лічильник циклу). Зауважимо також, що всередині циклу **while** повинен знаходиться оператор, що змінює значення змінної управління циклом.

Якщо вираз являє собою константу з *істинним* значенням, тіло циклу буде виконуватися завжди, і, отже, виходить нескінченний оператор. Цикл також виявиться нескінченним, коли умова, визначена в виразі-умові спочатку, *істинно* і ніде далі в тілі циклу не змінюється. Якщо ж перевірочна умова повертає *хибне* значення, здійсниться вихід з циклу і тіло оператора **while** буде пропущено.

Досить часто в якості виразу-умови використовується оператор присвоювання. Так як при цьому повертається деяке число, в операторі **while** фактично проводиться порівняння отриманого значення з нулем (нуль - еквівалент помилкового значення) з подальшим прийняттям рішення про вихід з циклу або про його продовження.

Таким чином, команда **while** може бути виконана один раз, кілька разів або жодного разу, в залежності від виразу-умови.

Приклад.

```
int a, s=0;
while (a!=0)
    {
    cin>>a;
    s+=a;
    }
```

Кроки	*x*	*s*
0	4	0
1	5	4
2	6	9
3	7	15
4	8	22
5	9	30

Приклад.

```
int x=4, s=0;
while (x<=8)
{
s+=x;
x++;
}
```

Після виконання команди *s*=30, *x*=9. В цьому випадку цикл виконується 5 разів.

А ось після виконання команди

```
int x=4, d=1;
while (x>10) d*=x;
```

змінна *d* свого значення не змінить, так як значення виразу *x*>10 одразу хибне і тому команда *d**=*x* в циклі **while** виконуватися не буде. Тобто цикл не виконається ні разу.

Цикл з післяумовою (do-while)

На відміну від оператора **while** цикл **do-while** спочатку виконує тіло (оператор або блок), а потім вже здійснює перевірку вираження на істинність. Синтаксис оператора має вигляд:

```
do
оператор;
while (вираз-умова);
```

Тіло циклу виконується до тих пір, поки вираз-умова є *істинною*. *Дія команди.*

1. Виконується оператор
2. Обчислюється значення виразу-умови.
3. Якщо значення виразу-умови *істинне*, то переходимо до пункту 1), якщо *хибне* - виконується перехід до наступної після **do-while** команди.

Оператор в циклі **do-while**, на відміну від попереднього циклу, буде виконуватися хоча б один раз завжди.

Одне з часто використовуваних застосувань даного оператора - запит до користувача на продовження виконання програми:

```
int main()
{
char ch;
do
{
// Тіло програми
...
cout<<"Продовжувати виконання?";
cin>>ch;
}
while(ch!='N')
}
```

Таким чином, тіло програми буде повторюватися до тих пір, поки користувач на питання «Продовжувати виконання?» не відповість символом *N*. При цьому в якості змінної, використовуваної для зберігання цього значення, виступає *ch*.

Приклад:

```
int a, s=0;
 do
{
cin>>a;   s+=a;
}
while(a!=0);
```

Зверніть увагу, що складовий оператор або блок обов'язково укладається в операторні дужки {}.

Кроки	*y*	*z*	*x*
0	0	1	5
1	5	10	3
2	8	6	1

Приклад.

```
int x=5, y=0;
 do
{
y+=x; z=2*x; x-=2;
}
while(x>1);
```

Після виконання команди *y*=8, z=6, x=9. В цьому випадку цикл виконується 2 рази.

Цикл з параметром (for)

Синтаксис циклу **for** має вигляд:

```
for (вираз_1; вираз-умова; вираз_3)
оператор або блок операторів;
```

вираз 1 і вираз 3 можуть складатися з декількох виразів, розділених комами.

Вираз_1 – найчастіше задає початкові умови для циклу (ініціалізація), воно виконується один раз.

Вираз-умова – визначає умову виконання циклу (умова виходу з циклу), до тих пір, поки воно *істинно* (не дорівнює 0), тіло циклу виконується, а потім обчислюється значення виразу_3.

Вираз_3 – задає зміна параметру циклу або інших змінних (корекція).

Цикл триває доти, поки вираз-умова не стане *хибною* (дорівнює 0).

Будь-який вираз може бути відсутнім, але розділяючи їхні порожні оператори **;** повинні бути обов'язково. Тому найпростіший приклад нескінченного циклу **for** (виконується постійно до примусового завершення програми ззовні) виглядає наступним чином: **for (;;) cout** << "Нескінченний цикл ...";

Дія команди.

1. Обчислюються значення виразу_1.
2. Обчислюються значення виразу-умови.
3. Якщо значення виразу-умови *істинне*, то виконується оператор. Якщо

хибне - програма переходить до наступної після циклу **for** команди.

3. Обчислюються значення виразу_3 і вираз-умова і виконується пункт 3. Типова помилка програмування циклів **for** - зміна значення лічильника як в конструкції (вираз_3), так і в тілі циклу. Це може призводити до таких негативних наслідків, як «випадання» або повтор ітерацій.

Приклади використання циклу з параметром.

1) Зменшення параметра:

```
for(int n=10; n>0; n--)
{
оператор
}
```

2) Зміна кроку коректування:

```
for(n=2; n<60; n+=13)
{
оператор
}
```

3) Можливість перевіряти умову відмінну від умови, що накладається на число ітерацій:

```
for (num=1;num*num*num<216; num++)
{
оператор
}
```

4) Корекція може здійснюватися не тільки за допомогою додавання або віднімання:

```
for (d=100.0; d<150.0;d*=1.1)
{
оператор
}

for (x=1; y<=75; y=5*(x++)+10)
{
оператор
}
```

5) Можна використовувати декілька ініціюючих або коригувальних виразів:

```
for (x=1, y=0; x<10; x++, y+=x);
```

Приклад. Суму цілих чисел з проміжку від 1 до 15 можна обчислити одним із способів:

1) **int** *n*=1, S=0; **for** (; *n*<16; *n*++) *S*+=*n*;
2) **for** (**int** *n*=1, S=0; *n*<16; *n*++) *S*+=*n*;
3) **for** (**int** *n*=1, S=0; *n*<16; *S*+=*n*++);
4) **for** (**int** *n*=1, S=0; *n*<16; *S*+=*n*, *n*++);

Зверніть увагу на те, що в якості тіла циклу в 3) і 4) може використовуватися порожній оператор (;). Його застосування при відсутності

оператора обов'язково, так як компілятор вимагає, щоб тіло циклу було не пустим. В 4) випадку ілюструється застосування оператора «кома».

Приклад. Кількість і добуток всіх парних чисел з проміжку від 4 до 11 можна обчислити так:

```
int n, D, kil;
for ( D=1, kil=0, n=4; n<=11; n+=2)
{
D*=n; kil++;
}
```

Оператори переходу

Оператори переходу виконують передачу управління.

1) **break** – оператор переривання циклу. Якщо потрібно передбачити вихід з оператора циклу в кількох місцях, не чекаючи виконання решти коду, то для цих цілей служить оператор **break**.

```
{
<оператори>
if (<вираз_умова>) break;
<оператори>
}
```

Тобто оператор **break** доцільно використовувати, коли умову продовження ітерацій треба перевіряти в середині циклу. На практиці оператор **break** часто використовують для виходу з нескінченного циклу.

Приклад: пошук суми чисел, що вводяться з клавіатури, до тих пір, поки не буде введено 100 чисел або 0

```
for(s=0, i=1; i<100;i++)
{
cin>>x;
if (x==0) break;   // якщо ввели 0, то підсумовування
s+=x;
}
```

2) **continue** – перехід до наступної ітерації циклу. Так само як і **break**, оператор **continue** перериває виконання тіла циклу, але він наказує програмі перейти на наступної ітерації циклу. Він використовується, коли тіло циклу містить розгалуження.

Приклад: Пошук кількості і суми додатних чисел

```
for( k=0,s=0,x=1; x!=0;)
{
```

```
cin>>x;
if (x<=0) continue;
k++; s+=x;
}
```

3) Оператор безумовного переходу **goto** має формат:

```
goto мітка;
```

У тілі тієї ж функції повинна бути присутня конструкція: *мітка: оператор;*

Мітка - це звичайний ідентифікатор, областю видимості якого є функція, з розташованим за ним символом двокрапки (:). Мітками позначають будь-який оператор, на який в подальшому повинен бути здійснений безумовний перехід. Оператор **goto** передає управління оператору, який стоїть після мітки. Використання оператора **goto** виправдано, якщо необхідно виконати перехід з декількох вкладених циклів або перемикачів вниз по тексту програми або перейти в одне місце функції після виконання різних дій.

Застосування **goto** порушує принципи структурного і модульного програмування, за якими всі блоки, з яких складається програма, повинні мати тільки один вхід і тільки один вихід.

Не можна передавати управління всередину операторів **if**, **switch** і *циклів*. Не можна переходити всередину блоків, що містять ініціалізацію, на оператори, які стоять після ініціалізації.

Взагалі кажучи, використання структурного та об'єктно-орієнтованого підходів до програмування дозволяє повністю відмовитися від застосування операторів безумовного переходу. Однак на практиці часто бувають випадки, коли **goto** значно спрощує код програми. В особливій мірі це твердження стосується вкладених конструкцій **switch-case** і **if-else.**

Приклад:

```
int k;
goto m;
. . .
{
int a=3, b=4; k=a+b;
m: int c=k+1;
. . .
}
```

У цьому прикладі при переході на мітку *m* не виконуватиметься ініціалізація змінних *a, b* і *k*.

4) Оператор **return** - оператор повернення з функції. Він завжди завершує виконання функції і передає управління в точку її виклику. Вид оператора:

```
return [вираз];
```

Перетворення типів

При виконанні операцій відбуваються неявні перетворення типів в наступних випадках:

- при виконанні операцій здійснюються звичайні арифметичні перетворення;
- при виконанні операцій присвоювання, якщо значення одного типу присвоюється змінній іншого типу;
- при передачі аргументів функції.

Крім того, в C ++ є можливість явного приведення значення одного типу до іншого. В операціях присвоювання тип значення, яке присвоюється, перетвориться до типу змінної, яка отримує це значення. Допускається перетворення цілих і плаваючих типів, навіть якщо таке перетворення веде до втрати інформації.

Перетворення цілих типів зі знаком. Ціле зі знаком перетвориться до більш короткого цілого зі знаком, за допомогою усічення старших бітів. Ціле зі знаком перетвориться до більш довгого цілому зі знаком, шляхом розмноження знаку. При перетворенні цілого зі знаком до цілого без знаку, ціле зі знаком перетвориться до розміру цілого без знаку і результат розглядається як значення без знаку.

Перетворення цілого зі знаком до плаваючого типу відбувається без втрати інформації, за винятком випадку перетворення значення типу **long long** або **unsigned long long** до типу **float**, коли точність часто може бути втрачена.

Перетворення цілих типів без знаку. Ціле без знаку перетворюється до більш коротшого цілого без знаку або зі знаком шляхом усічення старших бітів. Ціле без знаку перетворюється до більш довгого цілого без знаку або зі знаком шляхом доповнення нулів зліва. Коли ціле без знаку перетворюється до цілого зі знаком того ж розміру, бітове подання не змінюється. Тому значення, яке воно представляє, змінюється, якщо знаковий біт встановлений (дорівнює 1), тобто коли вихідне ціле без знаку більше ніж максимальне додатнє ціле зі знаком, такої ж довжини. Цілі значення без знаку перетворюються до плаваючого типу, шляхом перетворення цілого без знаку до значення типу **signed long**, а потім значення **signed long** перетворюється в плаваючий тип. Перетворення з **unsigned long** до типу **float**, **double** або **long double** виробляються з втратою інформації, якщо значення, яке потрібно більше, ніж максимальне додатнє значення, яке може бути представлено для типу **long**.

Перетворення плаваючих типів. Величини типу **float** перетворюються до типу **double** без зміни значення. Величини **double** і **long double** перетворюються до **float** с деякою втратою точності. Якщо значення занадто велике для **float**, то відбувається втрата значущості, про що повідомляється під час виконання. При перетворенні величини з плаваючою точкою до цілих типів вона спочатку перетворюється до типу **long** (дрібна частина плаваючою

величини при цьому відкидається), а потім величина типу **long** перетвориться до необхідного цілого типу. Якщо значення занадто велике для **long**, то результат перетворення не визначений.

Перетворення з **float, double** або **long double** до типу **unsigned long** проводиться з втратою точності, якщо значення, яке потрібно більше, ніж максимально можливе додатнє значення, представлене типом **long**.

Перетворення при обчисленні виразів

При виконанні операцій проводиться автоматичне перетворення типів, щоб привести операнди виразів до загального типу або щоб розширити короткі величини до розміру цілих величин, використовуваних в машинних командах. Виконання перетворення залежить від специфіки операцій і від типу операнду або операндів.

Розглянемо загальні арифметичні перетворення.

1. Операнди типу **float** перетворюються до типу **double**.
2. Якщо один операнд **long double**, то другий перетворюються до цього ж типу.
3. Якщо один операнд **double**, то другий також перетворюються до типу **double**.
4. Будь-які операнди типу **char** і **short** перетворюються до типу **int**.
5. Будь-які операнди **unsigned char** або **unsigned short** перетворюються до типу **unsigned int**.
6. Якщо один операнд типу **unsigned long**, то другий перетворюється до типу **unsigned long**.
7. Якщо один операнд типу **long**, то друий перетворюється до типу long.
8. Якщо один операнд типу **unsigned int**, то другий операнд перетворюється до цього ж типу.

Таким чином, можна відзначити, що при обчисленні виразів операнди перетворюються до типу того операнду, який має найбільший розмір.

Тема 2.2. Вивчення вказівників та їх ролі в мові С++. Робота з масивами

Поняття вказівника

Будь-який об'єкт програми (змінна) займає в пам'яті певну область. Місце розташування об'єкта в пам'яті визначається його адресою. При оголошенні змінної для неї резервується місце в пам'яті, розмір якого залежить від типу даної змінної, а для доступу до вмісту об'єкту служить його ім'я (ідентифікатор). При зверненні до змінної визначається її фактичне розташування в пам'яті, і всі дії ведуться вже з цією конкретною ділянкою.

Для вирішення різних завдань програмування (робота з масивами, побудова графічних зображень і використання динамічних ефектів)

потрібно знати не тільки значення деякої змінної, але і її адресу в оперативній пам'яті. Для визначення адреси в пам'яті є унарна операція визначення адреси:

```
&<назва даного>
```

Приклад 1. Розглянемо програму:

```
#include <iostream.h>
#define nr '\n'
int main()
{
int a = 40000;
cout << "Значення змінної a = " << a << nr;
cout << "Адреса змінної a: " << &a << nr;
}
```

В результаті виконання цих команд на екрані отримаємо:

```
Значення змінної a = 25
Адресою змінної a є 0xaf72254
```

Адреси зображуються шістнадцятковими числами. Адреси локальних змінних розміщуються в стеку і йдуть у зворотному порядку (стек росте в напрямі молодших адрес). Результат може відрізнятися навіть при повторному запуску програми, так як неможливо передбачити, за яким адресом почнуть розміщуватися змінні. Різниця в адресах між першою та другою змінними завжди буде однакова і складе розмір однойменних типів. Для різних типів механізм виділення пам'яті більш складний.

Вказівники.

Потужним засобом розробника програмного забезпечення на C++ є можливість здійснення безпосереднього доступу до пам'яті. Для цієї мети передбачається спеціальний тип змінних – вказівник (pointer). Вказівник являє собою змінну, значення якої є адреса комірки пам'яті. Він вказує на початок області оперативної пам'яті комп'ютера, де зберігається дане. Вказівники дають можливість оперувати не з іменами даних, а безпосередньо звертатися до областей пам'яті комп'ютера. Найбільша ефективність застосування вказівників у розробці програм досягається при використанні їх з масивами і символьними рядками.

Оголошення вказівника має наступний синтаксис:

```
<тип об'єкта>* <ім'я вказівника>
```

Тут <тип об'єкта> визначає тип даних, на які посилається вказівник з назвою <ім'я вказівника>. Можна створювати вказівники на константи, змінні,

функції, інші вказівники тощо. Символ «зірочка» повідомляє компілятору, що оголошена змінна є вказівником і належить до типу змінної, тому його рекомендується ставити поряд з типом, а від імені змінної відокремлювати пробілом, за винятком тих випадків, коли описуються кілька вказівників. При описі кількох вказівників знак * ставиться перед ім'ям змінної-вказівника, так як інакше не зрозуміло, що ця змінна також є вказівником.

Приклад 2.

```
int* nomer;
float *rost, *ptrA, *ptrB;
```

Тут оголошений вказівник на цілий тип *nomer* и вказівники на дійсний тип *rost, ptrA, ptrB.*

Базовий тип вказівника визначає тип даних, на які він буде посилатися.

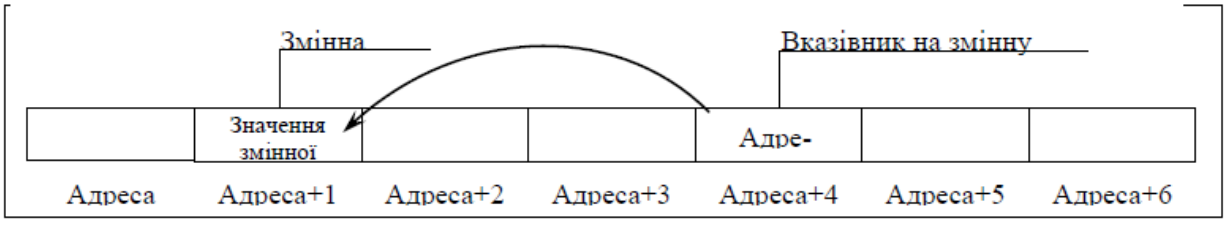
Символ * означає "вказівник на".

Номер байта, з якого починається в пам'яті змінна, називається ***адресом*** змінної. Також про адресу кажуть, що вона ***вказує*** на визначений байт. Таким чином, ***вказівник*** є просто адресом байту пам'яті.

Присвоїти значення вказівниками можна так:

```
<ім'я вказівника>=<адреса змінної>.
```

Оскільки вказівник являє собою посилання на деяку область пам'яті, йому може бути присвоєна тільки адреса деякої змінної (або функції), а не саме значення. У випадку некоректного присвоєння компілятор видає відповідне повідомлення про помилку.




```
int *prt_i; double *ptr_d; char *ptr_c;
cout << "\nНа цілий тип " << sizeof(ptr_i);
cout << "\nНа дійсний тип " << sizeof(ptr_d);
cout << "\nНа символьний тип " << sizeof(ptr_c);
```

3. Нехай в програмі оголошенні змінні:

```
int n = 10;
float stud1, stud2, stud3;
```

Тоді можна визначити вказівник на ці змінні:

```
nomer=&n;
rost=&stud1;
rost=&stud3;
```

Приклад 4. Знайти довжини (в байтах) вказівників на різні типи даних можна так:

```
int *prt_i; double *ptr_d; char *ptr_c;
cout << "\nНа цілий тип " << sizeof(ptr_i);
cout << "\nНа дійсний тип " << sizeof(ptr_d);
cout << "\nНа символьний тип " << sizeof(ptr_c);
```

Якщо виконати цю програму, то буде видно, що всі вище описані вказівники в пам'яті комп'ютера мають однаковий обсяг. Це тому, що для вказівників в залежності від операційної системи і версії компілятора резервується 2 або 4 байту в оперативній пам'яті.

Розіменування вказівників

Вказівники допомагають здійснювати безпосередній доступ до пам'яті і для того, щоб отримати (прочитати) *значення*, записане в області пам'яті, на яку посилається вказівник, використовують операцію непрямого звернення або ***операцію розіменування*** (*). При цьому використовується ім'я вказівника із зірочкою перед ним.

Приклад 5.

```
int x=1, y=22;
int *ptr=&y;
int* ip; ip=&x;
y=*ip;
*ip=*ip+10;
```

У наведеному фрагменті оголошуються дві змінні цілого типу *x* і *y* і два вказівника на тип **int**, причому один з них відразу ініціалізований першим адресом змінної y, а другий відразу оголошується як покажчик на цілий тип, а потім йому присвоюється значення адреси змінної x. Крім того присвоюється нове значення змінної y, застосовуючи операцію роз іменування. Значення *ptr також тепер зміниться, оскільки цей вказівник вказував на змінну y, а її значення в передостанньому рядку змінилося. В останньому рядку демонструється спосіб виконання додавання, використовуючи раз іменований вказівник.

На практиці досить часто застосовується так званий порожній вказівник (типу **void**), який може вказувати на об'єкт будь-якого типу. Для отримання доступу до об'єкта, на який посилається вказівник **void**, його необхідно попередньо привести до того ж типу, що й тип самого об'єкта.

Приклад 6.

```
char Let='T';
int nNum=9;
void *ptr; ptr=&Let;
*(char*)ptr='L';
ptr=&nNum;
*(int*)ptr=43;
```

Спочатку створюються два різнотипних об'єкта *Let* та *nNum* і порожній вказівник *ptr*, який ініціалізується адресом символьної змінної. Далі *ptr* розіменовується, приводиться до типу **char*** і за допомогою непрямої адресації модифікується значення символу *Let*. Аналогічним чином покажчик ініціалізується значенням адреси змінної *nNum*, приводиться до цілочисельного типу, після чого стає можливим зміна вмісту змінної *nNum*.

Таким чином, з покажчиками використовуються два оператора: "**&**" – повертає адресу пам'яті, за яким розташований операнд; "*****" – повертає значення змінної, на яку вказує операнд.

Арифметика вказівників.

Операції, що виконуються за допомогою вказівників, часто називають ***операціями непрямого доступу***, оскільки ми побічно отримуємо доступ до змінної за допомогою іншої змінної.

До вказівників можуть застосовуватися арифметичні операції і операції порівняння.

Компілятор, знаючи тип вказівника, обчислює розмір змінної того ж типу, після чого модифікує адресу, що міститься у вказівнику відповідно до заданої арифметичною операцією, але з урахуванням обчисленого для даного типу розміру. Це означає, що якщо оголошений вказівник типу **double**, що займає 8 байт пам'яті, то операція, наприклад, інкремента вказівника збільшить значення адреси не на один, а на 8 байт.

Вказівники можна віднімати один від одного, тим самим визначаючи кількість елементів (одного і того ж типу, на який вказують вказівники), розташованих між ними. Цей прийом буває корисним при операціях з масивами і символьними рядками.

Операція	Приклади й пояснення
==, !=, >=, <=, >, <	Порівнює значення двох вказівників (адреси, на які вони вказують). *Наприклад*, якщо вказівники вказують на одне і те ж дане, то результатом порівняння *vk1==vk2* буде істина, інакше – брехня.
++, --	Інкремента і декремента вказівників
–	*vk1–vk2*. Використовується для визначення кількості елементів, які знаходяться між двома вказівниками.
+, –	*vk1+k, vk1–k*. Знаходить вказівник, який зміщений відносно даного на *k* одиниць.

Існує можливість використання при оголошенні вказівників ключового слова **const**. При цьому слід брати до уваги, що застосування даного специфікатора трактується наступним чином:

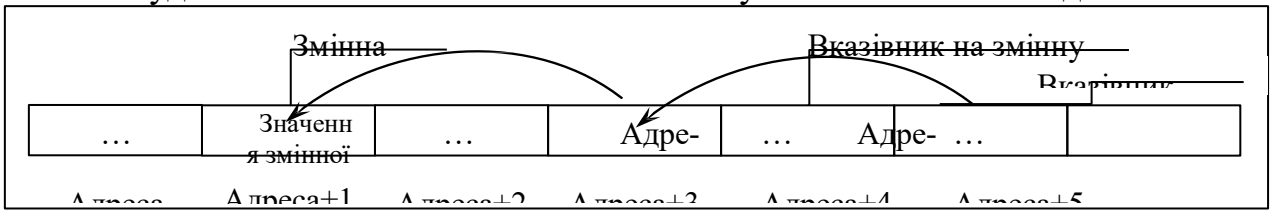
Приклад 7.

```
int* pi=&i;   // вказівник на цілу змінну. const int* pci=&ci; // вказівник на цілу константу. int* const cpi=&i;
    //вказівник-константа на змінну цілого типу.
const int* const cpc=&ci;   //  вказівник-константа    на цілу константу.
```

Вказівники *можна переадресовувати.* Зокрема, у прикладі 3 вказівником *rost* спочатку присвоєно адресу змінної *stud1*, а потім – змінної *stud3*. Це правило не діє, якщо вказівник є константою. Постійний вказівник вказує на одну і ту ж адресу і оголошують його так: **const int ***god*;

Вказівники на вказівники

Вказівники самі можуть посилатися на інші вказівники. При цьому в комірках пам'яті, на які посилається вказівник, міститься не значення, а адреса якого-небудь об'єкта. Сам об'єкт також може бути вказівником і т. д.



Тут за Адресою+1 зберігається значення деякої змінної, на яку вказує звичайний вказівник, що знаходиться за Адресою+3 (зберігає значення Адреса+1), на яку в свою чергу посилається вказівник, що знаходиться за

Адресою+5 (містить в якості значення Адресу+3).

Синтаксис вказівника на вказівник виглядає так:

```
int **pPtrInt;
```

Оголошення вказівника на вказівник, який сам є вказівником:

```
char ***ppSymbol;
```

При оголошенні вказівник на вказівник може ініціалізуватися адресом об’єкта:

```
char *pSymb = &myChar;
char **pPtr = &pSymb;
char ***ppPtr = &pPtr;
```

Число символів * при оголошенні говорить про «порядок» вказівника. Щоб отримати доступ до значення, такий вказівник повинен бути розіменованний відповідну кількість разів (по числу символів *).

Посилання.

Посилання – особливий тип даних, що є прихованою формою вказівника, який при використанні автоматично розіменовується. Іншими словами, він може використовуватися просто як інше ім'я, або псевдонім (синонім) об'єкта. При оголошенні посилання перед її ім'ям ставиться знак амперсанда (&), а сама вона повинна бути тут же проініціалізована ім'ям того об'єкта, на який посилається:

тип &ім'я посилання = ім'я;

Тип об'єкта, на який вказується посилання, може бути будь-яким. Оголошення не ініціалізованого посилання викликає повідомлення компілятора про помилку.

```
тип &ім'я посилання = ім'я;
```

```
int x;
int &sx = x;
```

Будь-яка зміна значення посилання веде за собою зміну того об'єкта, на який дане посилання вказує.

Приклад 9.

```
int x;
int &sx = x;
int sx+=10; //те ж, що й x+=10;
```

Використання посилань не пов'язано з додатковими витратами пам'яті.

Слід зазначити, що посилання не можна перепризначувати – ініціалізувавши посилання якимсь адресом деякої змінної, будь-яка дія з посиланням позначається на самому об'єкті. Спроба змінити наявне посилання якоїсь іншої змінної призведе до присвоєння оригіналу об'єкта значення другої

змінної.

Крім того, слід врахувати, що посилатися можна тільки на сам об'єкт. Не можна оголосити посилання на тип об'єкту.

Ще одне обмеження, що накладається на посилання, полягає в тому, що вони не можуть вказувати на нульовий об'єкт (приймає значення NULL). Таким чином, якщо є ймовірність того, що об'єкт в результаті роботи програми стане нульовим, від посилання слід відмовитися на користь застосування вказівника.

Масиви

Масив – це впорядкований скінчений набір даних одного типу, які зберігаються в *послідовно розташованих* комірках оперативної пам'яті і мають спільну назву. Назву масиву надає користувач.

Масив складається з *елементів*. Кожний елемент має індекси, якими його можна знайти в масиві. Кількість індексів визначає розмірність масиву. Розрізняють одно - та багатовимірні масиви. Наприклад, двовимірний масив даних – це таблиця, яка складається з декількох рядків і стовпців. У математиці поняття масив відповідають поняття вектору і матриці.

Загальний вид конструкції описання одновимірного масиву такий:

```
<тип> <ім'я масиву>[<розмір>]
```

Розмір – *кількість* елементів масиву. Розмір масиву необхідно знати і задавати відразу, оскільки компілятор повинен зарезервувати для нього необхідний обсяг пам'яті. Розміром може бути тільки постійна величина (не змінна).

Ім'я масиву в програмі змінювати не можна – це постійна величина, яка містить адресу першого елемента. Отже, назва масиву є вказівником на перший елемент.

Наприклад, команда **int** stud оголошує масив з ім'ям *stud*, який складається з п'яти цілих чисел; команда **float** *rost* оголошує масив *rost*, який містить десять чисел дійсного типу; **char** *alphavit* – оголошення масиву з 6 символів.

Звернутися до елементів масиву можна двома способами: *за допомогою імені масиву* або *використовуючи вказівники*.

Нумерація елементів масиву завжди починається з *нуля*. Щоб звернутися до деякого елементу, необхідно записати ім'я масиву, а в квадратних дужках – його номер. Наприклад, змінна *stud[2]* є третім елементом масиву *stud*, а *stud[4]* – п'ятим, оскільки масив *stud* має елементи *stud[0], stud[1], stud[2], stud[3] і stud[4]*.

Компілятор мови С++ не контролює чи належить індекс заданому діапазону. Відповідальність за це несе програміст. Наприклад, якщо в програмі оголосити масив *mas* з п'ятьма речовими числами і написати команду mas[54]=2, то повідомлення про помилку не буде, проте невідомо, в яку ділянку пам'яті потрапить число 2 і що станеться.

Назва масиву *stud* є вказівником на його перший елемент. Змінна **stud*

містить значення першого елемента масиву (елемента *stud[0]).* Оскільки всі елементи масиву розміщені в послідовних комірках оперативної пам'яті комп'ютера, то покажчик (*stud+1*) буде вказувати на другий елемент масиву (зміщення відносно вказівки *stud* на одну одиницю пам'яті), а покажчик (*stud+4*) – на п'ятий (зсув на чотири одиниці).

Ініціалізувати масив (задати значення елементів масиву) можна одним із способів:

- використовуючи принцип за замовчуванням;
- безпосередньо під час його оголошення;
- застосовуючи команду присвоєння;
- під час введення даних з клавіатури.

За замовчуванням всім елементам масиву присвоюється значення 0. Масив можна ініціалізувати повністю або частково відразу під час його оголошення, записуючи значення змінних через кому в фігурних дужках.

Наприклад

```
int Stud[] = {2, 10, 5, 7, 3};
float rost[10]= {163.4, 154.6, 170, 172.8};
char alphavit[6] = "Азбука" чи char alphavit[6] = {А', 'б', 'е', 'т', 'к', 'а'}.
```

Перші чотири елемента масиву *rost* були проініціалізовані, а інші – ні.

Якщо масив повністю буде ініціалізований під час оголошення, то його розмір вказувати не обов'язково. У цьому випадку компілятор сам визначає, скільки пам'яті необхідно зарезервувати. У наведеному прикладі масив *stud* буде складатися з п'яти цілих чисел.

Присвоїти значення інших елементів масиву *rost* або змінити значення вже проініціалізованих елементів можна командою присвоювання, наприклад, так: *rost[3]*=175.4, *rost[9*]=184.1 або так:

**(rost+2)*=164.5, **(rost+7)*=148.0 тощо. Елементи масиву також можна вводити з клавіатури під час виконання програми, як ми це робимо для змінних простих стандартних типів.

Масиви-константи (значення яких змінювати в програмі можна) оголошують так: **const int** *flag[]* = {1, 2}.

Постійні масиви потрібно ініціалізувати під час оголошення, інакше елементам масиву автоматично будуть присвоєні нульові значення, а змінювати їх не можна.

Для обробки елементів масиву найчастіше використовують команду циклу *for*, хоча можна застосувати і *while* або *do-while*.

Приклад 1. Створити масив з перших ста цілих чисел і обчислити суму всіх його значень можна одним із способів:

```
    int n[100];   // 1-й спосіб
int S=0;
for (k=0; k<100;)
{
*(n+k)=++k;
S+=*(n+k);
}
```

```
    int n[100];   // 2-й спосіб
int S=0;
for (k=0; k<100; k++)
{
n[k]=k+1;
S+=n[k];
}
```

Задачі на знаходження в масиві конкретних даних вирішують методом сканування (перебору, перегляду) *всіх елементів* масиву за допомогою циклу і умовної команди, де зазначають умови пошуку потрібних даних.

ЗМІСТОВИЙ МОДУЛЬ 3
РОЗШИРЕНІ ОПЕРАЦІЇ З ДАНИМИ У МОВІ С++

Тема 3.1. Дослідження функцій, їх створення та використання. Робота з посиланнями

Оголошення функцій користувача.

Прототипи стандартних функцій розміщені в папці **INCLUDE**. Кожну функцію користувача перед першим викликом, перш за все, необхідно оголосити (задекларувати, створити прототип, сигнатуру). Зазвичай оголошення функцій розміщують в заголовних файлах, які потім підключаються до початкового тексту програми за допомогою директиви **#include.** Але синтаксис мови надає можливість розмістити прототип і в основній програмі.

Функцію користувача оголошують так:

```
<Тип функції> <ім'я функції> (<список формальних
                    параметрів>);
```

де, тип функції – це тип даного, яке функція повертає в основну програму. Тип самої функції можна не вказувати.

За замовчуванням функція повертає в програму дане цілого типу **int**. Функцію, яка ніякі результати в програму не повертає, оголошують з типом **void**. Для функції, яка не залежить ні від яких параметрів, в круглих дужках записують службове слово **void**.

Ім'я функції дає користувач за правилами створення ідентифікаторів. У списку формальних параметрів через кому записують змінні, вказуючи їх типи. Тип необхідно вказувати для кожної змінної окремо. Імена змінних можна пропускати (не вказувати). Якщо функція не приймає ніяких значень, то список формальних параметрів може бути відсутнім.

Круглі дужки опускати не можна.

Приклад.

```
float Summa (int kol, float cena); kod (int k1, int k2);
void drobi (float, float);
double loto (void);
```

Під час оголошення можна відразу форматувати формальні параметри функції, тобто присвоювати їм певні значення. Такі значення називають значеннями за замовчуванням. Їх записують в кінці списку. Значення таких параметрів в програмі можна змінювати.

Приклад.

```
float Summa (int kol, float cena=2.5);
```

```
void drobi (float v=1.2, float n=3);
kod (int k1, int k2=5);
```

Опис функції користувача.

Опис функції складається з заголовка без крапки з комою і тіла функції, записаного в фігурних дужках і несе смислове навантаження:

```
<тип функції> <ім'я функції> (<список формальних параметрів>)
{
<тіло функції>;
return (<ім'я змінной>);
}
```

При описі функції в списку формальних параметрів обов'язково повинні бути імена змінних, навіть якщо в прототипі вони опускалися. У тілі функції записують команди, які задають дію функції. Результат виконання повертається в основну програму (в точку виклику) за допомогою змінної командою **return**. Тип змінної повинен збігатися з типом функції. У тілі функцій з типом **void** команду **return** не пишуть. У команді **return** круглі дужки не обов'язкові**.**

Приклад:

```
float Summa (int kol, float cena)
{
float s=kol*cena;
return (s);
}
void drobi (float v, float n)
{
cout<<"\n drobi="<<v/n;
}
```

Якщо потрібно проініціалізувати значення змінних, які входять в функцію, то це можна зробити в сигнатурі функції. Тоді в описі у заголовка функції значення за замовчуванням не задають.

Приклад.

сигнатура

```
float perimetr (int k=4, float r=2.5);
```

опис функції

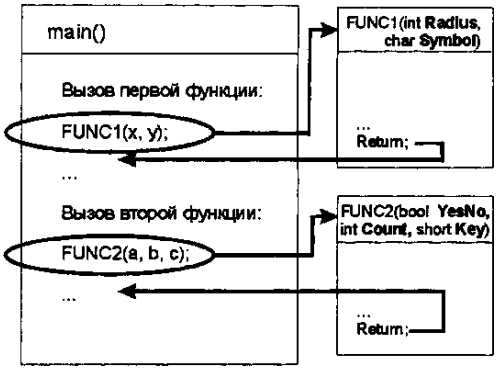
```
float perimetr (int k, float r)
{
float p; p=k*r; return (p);
```

```
}
```

Виклик функції користувача.

До функції користувача звертаються з розділу команд основної програми (функції **main** ()) або з іншої функції. Виклик функції можна записати двома способами: або просто командою виклику або викликати з виразів таким чином: <Ім'я функції> (<список фактичних параметрів>) Список фактичних параметрів може містити константи, змінні, посилання, покажчики, вирази. Списки формальних і фактичних параметрів повинні бути узгоджені за типами та кількістю елементів. Якщо в списку формальних параметрів є проініціалізувати змінні, то в списку фактичних параметрів ці змінні можуть бути відсутніми, їм будуть присвоєні значення за замовчуванням.

Будь-яка програма на С ++ обов'язково включає в себе головну функцію main (). Саме з цієї функції починається виконання програми. На малюнку показаний схематичний порядок виклику функцій.



Приклад.

Сигнатура

```
float perimetr (int k = 4, float r = 2.5);
```

до цієї функції можна звернутися одним із таких способів:

```
perimetr (7,2.8);
perimetr (7);
perimetr ();
```

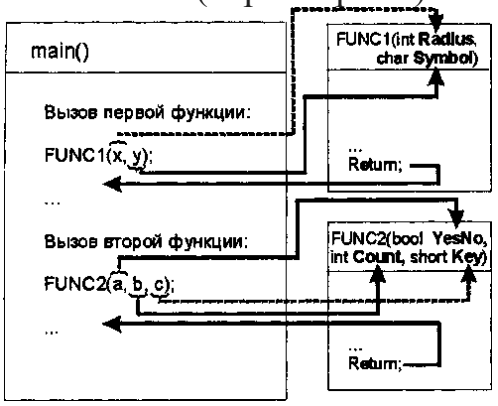
У списку фактичних параметрів не можна пропускати змінні з середини списку. Тобто не можна написати *perimetr* (1.65); так як тут пропущена ініціалізація цілої змінної k. Розглянемо найпростіший приклад створення своїх функцій. Напишемо свою функцію, яку назвемо *Say*, і яка виводить напис *«Hello, world!»* на екран.

```
void Say()
{
cout<<"Hello, world!";
}
```

У визначенні функції *Say ()* список параметрів порожній, тобто оголошень параметрів немає. Тіло функції складається з єдиною інструкції cout, яка виводить на екран вітання *"Hello, world!"*. У прикладі в заголовку функції *Say* () тип значення функції void. Цей тип застосовується тільки для функцій і для покажчиків, які ми будемо розглядати трохи пізніше; змінні типу void самостійно оголошувати не можна, компілятор відразу ж виведе помилку.

Якщо тип значення вказано як void (тобто «порожньо») це означає, що функція не повертатиме ніякого значення. У попередньому прикладі функція тільки виводила напис на екран і все, тобто ніяких значень вона повертати не повинна, тому її тип вказаний як void. В подальшому при створенні власних функцій необхідно уважно розібрати що функція буде робити і для чого вона призначена, визначити, чи буде функція повертати якесь значення і, якщо буде, то яке, і після цього тільки вказується тип значення функції. Наприклад, якщо раптом доведеться написати функцію користувача, яка обчислює косинус числа, то можна визначити, що функція косинуса повинна прийняти число з плаваючою точкою в списку параметрів, для цього числа знайти значення косинуса, яке теж буде числом з плаваючою точкою, і повернути це значення туди, звідки функція викликалася. Тобто для даного прикладу тип функції повинен бути **float** або **double,** так як необхідно буде повернути число з плаваючою крапкою.

Синтаксично, параметри - це ідентифікатори (імена змінних) і вони можуть використовуватися всередині тіла функції. У списку формальних параметрів параметри вказуються через кому. Параметри в оголошенні і описі функції називають формальними параметрами. Тим самим підкреслюється їх сутність: формальні параметри - це те, замість чого будуть підставлені фактичні значення, передані функції в момент її виклику. Після виклику функції значення аргументу, відповідне формальному параметру і передане при виконанні функції, використовується в тілі виконуваної функції. У С ++ такі параметри є викликаються за значенням (call-by-value). Коли застосовується виклик за значенням, змінні передаються функції як аргументи, їх значення копіюються у відповідні параметри функції, а самі змінні не змінюються в зухвалій оточенні. По суті, викликані за значенням, параметри є локальними в своїй функції. Їм можуть передаватися вираження, значення яких присвоюються цим локальним змінним (параметрами).



Програма починає виконуватися з функції **main()** до виклику функції *FUNC1(x, y).* З цього моменту управління програмою передається у функцію *FUNC1(x, y),* причому в якості значення змінної *Radius* ця функція використовує величину змінної x, а в якості змінної *Symbol* передається значення y (на малюнку проілюстрована передача параметрів у функції). Далі до оператора **return** виконується тіло функції FUNC1(x, y), після чого керування повертається в тіло функції **main(),** а саме, наступного за викликом *FUNC1(x, y)* оператору – управління повертається в те місце, звідки функція була викликана – викликає в оточення (*calling environment*).

Після цього продовжується виконання функції **main()** до виклику функції *FUNC2(a, b, з).* При виклику цієї функції мінлива a передає значення логічної змінної *YesNo*, змінна b – цілочисельний змінної **Count**, а змінна – коротким цілого *Key*. Розглянемо невеликий приклад створення функції, яка приймає деякі параметри. Напишемо функцію, яка обчислює квадрат якого-небудь цілочисельного значення і повідомляє результат.

```
int square (int a)
{
return a*a;
}
```

Як видно, функція *square()* приймає один цілочисельний параметр a, який буде передаватися значення з викличної функції, яке необхідно звести в квадрат. Тип повертаного значення для функції *square()* – **int**, тобто функція повинна буде повернути цілочисельне значення викликає функцію, що і робить інструкція **return** з параметром a, в якому зберігається результат роботи функції (квадрат значення параметра a).

Рядок **int main()** – це оголошення функції з ім'ям **main**, яка нічого не повертає і нічого не приймає в списку параметрів. А всі програми, які ми досі писали між фігурними дужками після рядка **int main(),** є не що інше, як тіло функції **main.**

Код програми повинен обов'язково перебувати в тілі якої-небудь функції (між фігурними дужками після заголовка функції), і не може «*висіти*» де-небудь поза функцій.

Ще одне суттєве зауваження: ***не можна визначати*** яку-небудь функцію в тілі іншої функції.

Програма на C++ складається з однієї або більше функцій, одна з яких – **main().**

Зазвичай функція, яка викликає якусь іншу функцію, називається *викликаною функцію*, а функція, яку *викликають на виконання*, називається *спричинюваною функцією*.

Для виклику функції необхідно вказати ім'я функції і список параметрів, які необхідно передати їй. Список переданих параметрів повинен бути укладений у круглі дужки.

Деякі функції повертають значення, як, наприклад, функція, обчислює

квадратний корінь від числа, повертає значення цього кореня. Якщо функція повертає значення, то потрібно прийняти це значення в якусь змінну (вивести на екран або використовувати в складеному вираженні). Наприклад, в бібліотеці *math.h* визначена функція мають квадратний корінь, яка приймає як параметр значення, з якого слід витягти корінь, і повертає значення цього кореня. Як її можна використовувати

int *i*=2;

double *kor*1=***sqrt***(*i*); або **cout**<<***sqrt***(5);

Виклик функції є виразом, тому його можна використовувати як складову частину більш складного вираження.

Наприклад:

cout<<2***sqrt***(3)+1; або **float** *res*=***sqrt***(***sqrt***(7)/2.0)+10;

Викликати функцію можна тільки після оголошення функції (тобто нижче оголошення).

Оголошення функції стандартної бібліотеки виконується за допомогою підключення бібліотеки директиви препроцесора **#include**.

Існує кілька способів повернення управління до точки, з якої була викликана функція:

- Якщо функція не повинна повертати результат, керування повертається або просто при досягненні правої фігурної дужки, завершальній функцію, або при виконанні оператора **return.**
- Якщо функція повинна повертати результат, то оператор **return** вираз; повертає значення виразу в точку звернення до функції. Таким чином, оператор повернення має дві форми:
- **return;** – В цьому випадку, коли виконується оператор повернення, управління програмою негайно передається назад. Використовується коли функція не повертає значення (можна і без нього).
- **return** вираз; – В цьому випадку, в викликане середовище, повертається значення виразу, яке слід за ключовим словом **return**. Це значення повинно бути конвертованим до повертання типу з заголовка визначення функції.

  *Приклад***:**

  return; не повертає значення;

  return 3; обчислене значення=3;

  return (a+b); // обчислене значення=значення виразу (a+b); дужки необо'язкові, використовуються для поліпшення читаності коду.

Приклад 1

/* Програма для введення з клавіатури трьох чисел пропонує виконати наступні дії: перевірити на парність, обчислити добуток, обчислити середнє арифметичне непарних чисел */

```
#include < iostream.h >
// прототип функції
bool IsEven(int);  /* функція перевіряє число на парність*/
int abcMultiple(int, int, int); /* функція вичисляє суму
трьох чисел */
float SrArith(int, int, int);    /* функція обчислює середнє
арифметичне непарних чисел*/
// описання функції
void OutputMenu()  /* функція виводить на екран меню
програми*/
{
cout<<"1. Ведення чисел \n"
"2. Перевірка чисел на парність\n"
"3. Обчислити добуток \n"
"4.Обчислити середнє арифметичне непарних чисел \n";
cout<<"\n \n Виберіть пункт меню \n";
}
int InputNumber()  /* функція введення числа в заданому
діапазоні */
{
int number=0;
  do
{
cout<<"\n Введіть ціле число в діапазоні от 0 до 500: ";
cin>>number;
}
while(number<0 || number>500);
return number;
}
int main()
{
int a=0, b=0, c=0, /* оголошення і ініціалізація змінних
цілого типу для збереження чисел, введених користувачем з
клавіатури*/
menuVal=0;    /*оголошення и ініціализація змінной цілого
типу для збереження значення вибраного пункта меню */
  do
```

```
{
OutputMenu(); /*виклик функції OutputMenu*/
cin>>menuVal;
switch(menuVal)
{
case 1:  /* Мітка відповідає задачі першого пункта меню */
a=InputNumber(); cout<<" \n a="<<a; b=InputNumber(); cout<<"
\n b="<<b; c=InputNumber(); cout<<" \n c="<<c; break;
case 2:  /* Мітка відповідає задачі другого пункту меню */
if(IsEven(a)) cout<<"\n a="<<a<<" - парне"; if(IsEven(b))
cout<<"\n b="<<b<<" - парне"; if(IsEven(c)) cout<<"\n
c="<<c<<" - парне"; break;
case 3:  /* Мітка відповідає задачі третього пункту меню */
cout<<"\n Добуток a*b*c="
<<abcMultiple(a, b, c);
break;
case 4:  /* Мітка відповідає задачі четвертого пункту меню */
cout<<"\n Середнє арифметичне не парних чисел="
<< SrArith (a, b, c);
break;
default:
/* У випадку, коли користувач ввів значення за межами
діапазону 1-4 */
cout<<"\n Будьте уважними. Є всього 4 пункту меню :)";
}
cout<<"\n\n Далі? " "\n 0 - Нет, 1 - Так "; cin>>menuVal;
}
while(menuVal);
}
// описання функції, прототип функції оголошені вище
  bool IsEven(int val)
{
bool valIsEven=val%2? false : true;
return valIsEven;
}

int abcMultiple (int a, int b, int c)
```

```
{
return a*b*c;
}
float SrArith(int val1, int val2, int val3)
{
int Sum=0, count=0;     /* оголошуємо допоміжні змінні для
обчислення середнього арифметичного Sum/count */
if(!IsEven(val1))
{
count++; Sum+=val1;
}
if(!IsEven(val2))
{
count++; Sum+=val2;
}
if(!IsEven(val3))
{
count++; Sum+=val3;
}
if(count) return float(Sum)/count; else return 0;
}
```

Приклад 2.

//Складення трьох цілих чисел – ілюстрація прототипів функції

```
#include <iostream.h>
int add3(int, int, int);
double sred(int);
void main()
{
int n1, n2, n3, sum; cout<<"\nEnter three marks: ";
cin>>n1>>n2>>n3; sum=add3(n1, n2, n3); cout<<"\nSum= "<<sum;
cout<<"\n sred= "<<sred(sum); sum=add3(1.5*n1, n2, 0.5*n3);
cout<<"\nWeight sum= "<<sum<<".";
cout<<"\nWeight sred= "<<sred(sum)<<"."<<"\n";
}
```

```
int add3(int a, int b, int c)
{
return (a+b+c);
}
double sred(int s)
{
return (s/3.0);
}
```

Розбір програми

```
int add3(int, int, int);
double sred(int);
```

Ці оголошення є прототипами функцій. Вони інформують компілятор про тип і кількість аргументів, передбачуваних для кожної оголошеної таким чином і визначеною в іншому місці функції.

sum=add3(1.5**n*1, *n*2, 0.5**n*3); – Виклик функції підсумовування трьох цілих параметрів add3, параметри в дужках будуть неявно перетворені до цілого типу.

```
int add3(int a, int b, int c)
{
return (a+b+c);
}
```

Це власне визначення функції. Воно відповідає оголошення прототипу функції перед **main().** Так як список аргументів у прототипі функції може включати імена змінних, **int add3(int a, int b, int**); теж допустимо.

Область видимості. Локальні і глобальні змінні

Змінні можуть бути оголошені як усередині тіла функції, так і за межами будь-якої з них. Змінні, оголошені усередині тіла функції, називаються локальними. Такі змінні розміщаються в стеку програми і діють лише всередині тієї функції, в якій оголошені. Як тільки управління повертається викликає функції, пам'ять, відведена під локальні змінні, звільняється. Кожна змінна характеризується

- ✓ областю дій,
- ✓ областю видимості
- ✓ часом життя.

Під областю дії змінної розуміють область програми, в якій змінна доступна для використання. З цим поняттям тісно пов'язані поняття області *видимості змінної.*

Якщо змінна виходить з області дії, вона стає невидимою. З іншого боку,

змінна може перебувати в області дії, але бути невидимою.

Змінна знаходиться в області видимості, якщо до неї можна отримати доступ (за допомогою операції дозволу видимості, в тому випадку, якщо вона безпосередньо не видно).

Часом життя змінної називається інтервал виконання програми, протягом якого вона існує.

Локальні змінні мають своєю областю видимості функцію або блок, в яких вони оголошені. У той же час область дії локальної змінної може виключати внутрішній блок, якщо в ньому оголошена змінна з тим же ім'ям. Час життя локальної змінної визначається часом виконання блоку або функції, в якій вона оголошена.

Це означає, наприклад, що в різних функціях можуть використовуватися змінні з однаковими іменами абсолютно незалежно один від одного. У розглянутому прикладі змінні з ім'ям x визначені відразу в двох функціях - в *main()* і в *Sum(),* що, однак, не заважає компілятору розрізняти їх між собою:

```
#include <iostream.h>
int Sum(int a, int b);
int main()
{
// Локальні змінні:
int x = 2;
int y = 4;
cout<<Sum(x, y) ;
}
int Sum(int a, int b)
{
// Локальна змінна x видна тільки в тілі функції Sum()
int x=a+b;
return x;
}
```

функцій і діють протягом виконання всієї програми.

Такі змінні доступні в будь-якої з функцій програми, яка описана після оголошення глобальної змінної.

Звідси випливає висновок, що імена локальних і глобальних змінних не повинні збігатися. Якщо глобальна змінна не проініціалізована явним чином, вона ініціалізується значенням 0.

Область дії глобальної змінної збігається з областю видимості і простягається від точки її опису до кінця файлу, в якому вона оголошена. Час життя глобальної змінної – *постійне*, тобто збігається з часом виконання програми.

Взагалі кажучи, на практиці програмісти намагаються уникати

використання глобальних змінних і їх застосовують лише в разі крайньої необхідності, так як вміст таких змінних може бути змінено всередині тіла будь-якої функції, що загрожує серйозними помилками у роботі програми. Розглянемо приклад, що пояснює вищесказане:

Приклад 3.

```
#include <iostream.h>
// Оголошуємо глобальну змінну Test:
int Test=200;
void PrintTest(void);
int main()
{
// Оголошуємо локальну змінну Test:
int Test=10;
// Виклик функції друку глобальної змінної:
PrintTest();
cout<<"Локальна: "<<Test<<'\n';
}
void PrintTest(void)
{
cout<<"Глобальна: "<<Test<<' \n';
}
```

Спочатку оголошується глобальна змінна *Test*, якій присвоюється значення 200. Далі оголошується локальна змінна з тим же ім'ям *Test*, але зі значенням 10. Виклик функції *PrintTest() main()* фактично здійснює тимчасовий вихід з тіла головної функції. При цьому всі локальні змінні стають недоступні і *PrintTest()* виводить на друк глобальну змінну *Test*.

Після цього управління програмою повертається у функцію main(), де конструкцією *cout* виводиться на друк локальна змінна *Test*. Результат роботи програми виглядає наступним чином:

Глобальна: 200

Локальна: 10

В C++ допускається оголошувати локальну змінну не тільки на початку функції, а взагалі в будь-якому місці програми. Якщо оголошення відбувається всередині якого-небудь блоку, змінна з таким же ім'ям, оголошена поза тіла блоку, "ховається". Видозмінимо попередній приклад з тим, щоб продемонструвати процес приховування локальної змінної:

```
#include <iostream.h>
// Викликаємо глобальну змінну Test:
int Test = 200;
```

```
void PrintTest(void);
int main()
{
// Викликаємо локальну змінну Test:
int Test=10;
// Виклик функції печаті глобальної змінної:
PrintTest();
cout<<"Локальная: "<<Test<<'\n';
// Добавляем новий блок з ще одной локальной змінноїTest:
{
int Test = 5;
cout<<" Локальна: "<<Test<<' \n';
}
// Повертаємося до локальної Test поза блоку:
cout<<" Локальна: "<<Test<<'\n';
}
void PrintTest(void)
{
cout<<"Глобальна: "<<Test<<'\n';
}
```

Результат модифікованої програми буде виглядати наступним чином:

```
Глобальна: 200
Локальна: 10
Локальна: 5
Локальна: 10
```

Операція :

Як було показано вище, оголошення локальної змінної приховує глобальну змінну з таким же ім'ям. Таким чином, всі звернення до імені глобальної змінної в межах області дії локального оголошення викликають звернення до локальної змінної. Однак C++ дозволяє звертатися до глобальної змінної з будь-якого місця програми з допомогою використання операції дозволу області видимості. Для цього перед ім'ям змінної ставиться префікс у вигляді подвійного двокрапки (::):

```
#include <iostream.h>
// Оголошення глобальної змінної
int Turn=5;
int main ()
```

```
{
 // Виклик функції глобальної змінної:
int Turn=70;
// Вивід локального значення:
cout<<Turn<<'\n';
// Вивід глобального значення:
cout<<::Turn<<'\n';
}
```

В результаті у два рядки буде виведено два значення: 5 і 70. З розглянутого прикладу видно, що були оголошені глобальна і локальна змінні з ім'ям *Turn*, які згодом були виведені на друк.

Правила дій областей видимості.

Правила дії областей видимості визначають можливість отримання доступу до об'єкта і час його існування.

Областю видимості ідентифікатора називається область програми, в якій на даний ідентифікатор можна посилатися.

На деякі ідентифікатори можна посилатися в будь-якому місці програми, тоді як інші - лише в певних її частинах.

Існує чотири області видимості ідентифікатора - область видимості функції, область видимості файл, область видимості блок і область видимості прототип функції.

Ідентифікатор, оголошений поза будь-якої функції (на зовнішньому рівні), має **область видимості файл**. Такий ідентифікатор "відомий" всіх функцій від точки його оголошення до кінця файлу. Змінні, оголошення функцій і прототипи функцій, що знаходяться **поза** функції - всі мають областю видимості файл.

Змінні, викликані поза функції, називаються глобальними змінними. Мітки (ідентифікатори з подальшим двокрапкою, наприклад, *start*: - єдині ідентифікатори, які мають областю видимості функцію. Мітки можна використовувати всюди в функції, в якій вони з'явилися, але на них не можна посилатися поза тіла функції. Мітки використовуються в структурах *switch* (як мітки *case)* і в операторів *goto*. Мітки ставляться до тих деталей реалізації, які функції "ховають" один від одного. Це приховування − один з найбільш фундаментальних принципів розробки хорошого програмного забезпечення.

Ідентифікатори, оголошені всередині блоку (на внутрішньому рівні) мають областю видимості блок. (Блок починається відкривається фігурною дужкою і завершується закриває). Область видимості блок починається оголошенням ідентифікатора і закінчується кінцевою правою фігурною дужкою блоку.

Змінні, які мають область видимості блок, називаються локальними змінними.

Змінні, оголошені в описах функцій, мають областю видимості блок так само, як і параметри функції, і є локальними змінними. Будь блок може містити оголошення змінних. Якщо блоки вкладені і ідентифікатор у зовнішньому блоці має таке ж ім'я, як ідентифікатор у внутрішньому блоці, ідентифікатор зовнішнього блоку "невидимий" (приховано) до моменту завершення роботи внутрішнього блоку. Це означає, що поки виконується внутрішній блок, він бачить значення своїх власних локальних ідентифікаторів, а не значення ідентифікаторів з ідентичними іменами охоплює блоці.

Розглянемо це на прикладі.

```
{                                   //зовнішній блок
int a=2,                            //виклик і ініціализація змінної a
cout<<a<<'\n';                      // виводить на екран 2
{                                   // вхід в внутрішній блок
int a=7,                            // змінна a із внутрішнього блоку
int s=a;                            //виклик і ініціализація змінної s
cout<<"s="<<s;                      //виводить на екран s=7
cout<<"a="<<a<<'\n';                //виводить на екран a=7
}                                   //вихід із внутрішній блок
cout<<++a<<'\n';                    // виводить на екран 3
cout<<s<<'\n';                      //помилка компіляції: змінна s не
}                                   // викликана, вихід із внутрішнього блока
```

Внутрішні блоки можуть бути вкладені на довільну глибину, визначену обмеженнями системи. Єдиними ідентифікаторами з областю видимості прототип функції є ті, які використовуються в списку параметрів прототипу функції. Прототипи функцій не вимагають імен у списку параметрів − потрібні тільки типи. Якщо в списку параметрів прототипу функції використовується ім'я, компілятор це ім'я ігнорує.

Ідентифікатори, що використовуються в прототипі функції, можна повторно використовувати де завгодно в програмі, не побоюючись двозначності. Змінну можна оголосити у розділі ініціалізації циклу for або умовному вираженні інструкцій *if, switch* або *while*.

Приклад

```
int main ()
{
for (int i=0; i<10; i++)
{cout<<i<<" ";
cout<<" kvadrat ="<<i*i<<'\n';
}
i=10 // помилка! i - невідома
}
```

Приклад 2.

```
int main()
{
int choice;
cout<<"(1) скласти  числа"; cout<<"(2) конкатенувати рдки";
cin >> choice;
if (choice==1)
{
int a,b;
cout<<"Введіть два числа:";
cin >> a >> b;
cout<<"Сума рівна"<<a+b<<'\n';
}
else
{char s1[80], s2[80];
cout<<"Введіть два рядки";
cin>>s1; cin>>s2; strcat(s1,s2);
cout<<"Конкатенація рівна"<<s1<<'\n'
}
}
```

Новий стиль заголовків

Історично так склалося, що в мові C++ при підключенні заголовних файлів використовувався той же синтаксис, що і в мові C для сумісності з розробленим на той момент програмним забезпеченням. Однак при стандартизації мови цей стиль був змінений і тепер замість заголовних файлів (як це було З) вказуються деякі стандартні ідентифікатори, за яким компілятор

сам знаходить необхідні файли. Системні ідентифікатори являють собою ім'я заголовка в кутових дужках без вказівки розширення (.h). Нижче наводиться приклад включення заголовків у стилі C++:

```
#include <iostream>
#include <stdlib>
#include <new>
```

Крім цього, для включення в програму бібліотек функцій мови у відповідність з новим стандартом заголовок перетворюється наступним чином: відкидається розширення **.h** і до імені заголовка додається с. Таким чином, наприклад, заголовок <*string.h*> замінюється заголовком <*cstring*>. Якщо ж використовується компілятор не підтримує оголошення заголовків у новому стилі, можна використовувати заголовки в стилі мови, хоча це і не рекомендується стандартом C++.

Простір імен

Визначення функцій і змінних в заголовних файлах нерозривно пов'язані з поняттям простору імен. Це поняття з'явилося порівняно недавно. До введення поняття простору всі оголошення ідентифікаторів і констант, зроблені в заголовному файлі, містилися компілятором в глобальний простір імен. Таке становище призводило до виникнення маси конфліктів, пов'язаних з використанням різними об'єктами однакових імен. Найчастіше непорозуміння виникали, коли в одній програмі використовувалися бібліотеки, розроблені різними виробниками. Введення поняття простору імен дозволило значно знизити кількість подібних конфліктів імен. Коли в програму включається заголовок нового стилю, його вміст не поміщається в глобальний простір імен, а в простір імен *std* Якщо в програмі потрібно визначити деякі ідентифікатори, які можуть замінити вже наявні, просто треба завести свій власний, новий простір імен. Це досягається шляхом використання ключового слова *namespace:*

```
namespace ім'я_простіру_імен
{
// оголошення
}
```

Таким чином, оголошення в межах нового простору імен будуть знаходитися тільки в межах видимості *певного имені_простіру_імен,* запобігаючи тим самим виникнення конфліктів. В якості прикладу створимо наступне простір імен:

```
namespace NewNameSpace
{
int x, y, z;
void SomeFunction(char smb);
}
```

Для того, щоб вказати компілятору, що слід використовувати імена з конкретного іменного простору (в даному випадку з *NewNameSpace*), можна скористатися операцією дозволу видимості:

```
NewNameSpace::x=5;
```

Однак, якщо в програмі звернення до власного простору імен проводяться досить часто, такий синтаксис викликає певні незручності. В якості альтернативи можна скористатися інструкцією using, синтаксис якої має дві форми:

```
using namespace им'я_простіру_імен; або
using им'я_простіру_імен::ідентифікатор;
```

При використанні першої форми компілятору повідомляється, що в подальшому необхідно використовувати ідентифікатори із зазначеного іменного простору аж до того моменту, поки не зустрінеться наступна інструкція using. Наприклад, вказавши в тілі програми

```
using namespace NewNameSpace ;
```

можна безпосередньо працювати з відповідними ідентифікаторами:

```
x=0; y=z=4; SomePunction('A');
```

На практиці часто після включення в програму заголовків явно вказується використання ідентифікаторів стандартного простору імен:

```
using namespace std;
```

Друга форма запису наказує компілятору використовувати зазначене простір імен лише для конкретного ідентифікатора. Таким чином, визначивши

```
using namespace std;
using NewNameSpace::z;
```

можна використовувати ідентифікатори стандартної бібліотеки C++ і цілочисельну змінну *z* з простору імен *NewNameSpace* без використання операції дозволу видимості: *z=12*;

Слід розуміти, що вказівка нового простору імен інструкцією using namespace скасовує видимість стандартного простору *std*, тому для отримання доступу до відповідних ідентифікаторів з std потрібно кожен раз використовувати операцію дозволу видимості std::. Простору імен не можуть бути оголошені усередині тіла функції, однак можуть оголошуватися всередині інших просторів. При цьому для доступу до ідентифікатора внутрішнього

простору необхідно вказати імена всіх вищих іменованих просторів. Наприклад, оголошено наступне простір імен:

```
namespace Highest
{
namespace Middle
{
namespace Lowest
{
int nAttr;
}
}
}
```

Використання оголошеної змінної *nAttr* буде виглядати:

```
Highest::Middle::Lowest::nAttr=0;
```

Математичні функції

Прототипи стандартних математичних функцій визначені у заголовному файлі *math.h*. Розглянемо деякі з них, найбільш часто вживані в повсякденній роботі. Наприклад, функція *pow()*, що дозволяє зводити число до степеня. Синтаксис цієї функції виглядає наступним чином: *double pow (double x, double);* Таким чином, компілятор повідомляється, що необхідно число подвійної точності x звести до степеня числа подвійної точності. До цієї категорії також належать логарифмічні функції та функція добування кореня числа:

```
double log(double);     // натуральні логарифми
float logf(float);
long double logl(long double);
double log10(double);   // десяткові логарифми
float log10f(float);
long double log10l(long double);
double sqrt(double);    // корінь числа
float sqrtf(float);
long double sqrtl(long double);
Друга більша група - функції отримання абсолютної величини
числа,
int abs(int); // цілі
```

```
double fabs(double);    // двійної точності
long labs(long);   // довгі
float fabsf(float);     // з плаваючою точкою
long double fabsl(long double);  // довгі двійної точності
```

сприймають у якості параметра аргумент деякого типу (свій для кожної з функцій) і повертають його беззнакову форму.

Для обчислення залишку від ділення числа x на y використовується функція *fmod()*, яка має наступний синтаксис: *double fmod(double x, double);* Стандартна бібліотека має широким набором тригонометричних функцій і їх модифікацій для різних типів аргументів:

```
double acos(double);    // Арккосинус
float acosf(float);
double asin(double);    // Арксинус
float asinf(float);
double atan(double);    // Арктангенс
float atanf(float);
double atan2(double x, double y);     // Арктангенс
відношення y/x
float atan2f(float, float);
double cos(double);     // Косинус
float cosf(float);
double cosh(double);    // Гіперболічний косинус
float coshf(float);
double sin(double);     // Синус
float sinf(float);
double sinh(double);    // Гіперболічний синус
float sinhf(float);
double tan(double);     // Тангенс
float tanf(float);
double tanh(double) ;   // Гіперболічний тангенс
float tanhf(float);
```

Слід відзначити, що кути тригонометричних функцій зазначаються в радіанах. Нижче наводиться приклад, що здійснює переклад градусів в радіани і виведення значення синуса для введеного в градусах числа.

```
#include <iostream.h>
#include <math.h>
int main()
{
double Angle;
double PI=3.14159; cout<<"Введіть угол в градусах: ";
cin>>Angle;
cout<<"Значення синуса: ";
cout<<sin(Angle*PI/180)<<'\n';
}
```

На жаль, в С++ немає готової реалізації функції зведення аргументу в квадрат. Як її реалізувати ми розглядали в прикладі вище.

Функції округлення

Часто потрібно скористатися округленим значенням тієї чи іншої змінної. С++ пропонує набір функцій для рішення цієї задачі. Залежно від конкретної ситуації може знадобитися функція, округляючи значення аргументу в більшу або меншу сторону. Розглянемо найбільш часто використовувані варіанти викликів.

Округлення числа в меншу сторону використовується функція *floor()* та її різновиди для різних типів аргументів і повертаються параметрів. Ця функція має наступний синтаксис:

```
double floor(double x);
long double floorl(long double x);
```

Округлення у більшу сторону проводиться за допомогою функції *ceil():*

```
double ceil(double x);
long double ceil(long double x);
```

Проте в реальності проблема вибору в яку ж сторону проводити округлення, покладається на розроблювану програму.

Посилання.

До даних можна звернутися за допомогою імен або посилань. Посилання служить для присвоєння ще одного імені (псевдоніма, синонімів, псевдонімів) даним. Посилання створюють так:

```
<тип даного> &<ім'я посилання>=<ім'я змінної>;
```

Приклад. *float &cena=summa*; – у цьому випадку посилання *cena* і мінлива *summa* будуть вказувати на один і той же адресу в пам'яті комп'ютера. Для посилань не резервується додаткова оперативна пам'ять. В С++ можна створювати посилання на дані, але не на їх типи. Значення посилання

ініціалізують відразу під час її оголошення, тобто на етапі компіляції. У наведеному прикладі посилання cena проініціалізована змінної *summa*. Таким чином, якщо *summa*=11, то і значення посилання *cena* теж буде 11. Під час зміни значення посилання змінюється значення змінної, на яку це посилання. Таким чином, якщо в програмі записати команду *cena*=15.7, то змінної summa теж буде присвоєно значення 15.7.

Змінювати (переадресування) посилання в програмі не можна. Посилання завжди вказує на один і той же адресу в оперативній пам'яті. Це використовують під час створення і виклику функцій. Під час виклику функції копії всіх її фактичних параметрів заносяться в спеціально організовану область пам'яті. Потім виконуються відповідні команди функції і результат повертається в програму командою **return.** Оскільки всі дії виконуються з копіями параметрів, а не з самими параметрами (копії і власне параметри розміщені в різних ділянках пам'яті), то значення фактичних параметрів в основній програмі не змінюються. Як параметри функції можна використовувати посилання або покажчики. Тоді значення фактичних параметрів в основній програмі будуть змінюватися, так як функція буде повертати значення в основну програму не тільки через змінну з команди **return,** а також через відповідні посилання і покажчики, так як вони вказують на той же самий ділянку пам'яті, де розміщені фактичні параметри. Щоб передати посилання або покажчики в функцію і не змінити значення фактичних параметрів, потрібно в оголошенні функції до кожного параметру дописати ключове слово **const.**

Приклад:

```
int sort (const int *p, const int *g);
```

В С++ посиланням може бути не тільки змінна або константа, а й функція

<тип> &<ім'я функції>(<список формальних параметрів>)

Така функція повертає синонім імені комірки, в яку занесено результат (посилання на змінну певного типу). Функція-посилання має двояке призначення. По-перше, як і звичайна функція, вона може повертати значення в основну програму. По-друге, їй самій можна присвоювати значення, що є унікальним випадком в програмуванні.

Приклад.

```
float *prt, u;  // Оголошуємо змінну і покажчик на дійсний тип
float &Item(float *a, int i)        // Оголошення функції-покажчика
{
return *(a+i);
}
prt=new float[10];  // Виділяємо ділянка пам'яті для зберігання
// значень десяти дійсних чисел
u=Item(prt, 3);       // Викликати цю функцію можна зазвичай// Змінної u буде присвоєно значення
// четвертого елемента.
```

//Введемо значення п’ятого числа так

Item(*prt*, 4)=10; // У цьому випадку функції Item() присвоюємо значення,// тобто ділянка пам'яті буде внесено число 10.

Передача параметрів в функцію.

У багатьох мовах програмування є два способи передачі параметрів у функцію за значенням та за посиланням. Коли параметр передається за значенням, створюється його копія і вона передається викликається функції. Зміни копії не впливають на значення оригіналу.

При передачі аргументів за значенням компілятор створює тимчасову копію об'єкта, який повинен бути переданий, і розміщує її в області стекової пам'яті, призначеної для зберігання локальних об'єктів. Викликається функція оперує саме з цією копією, не надаючи впливу на оригінал об'єкта. Прототипи функцій, що приймають аргументи за значенням, передбачають в якості параметрів вказівка типу об'єкта, а не його адреси. Наприклад, функція int GetMax(int, int); приймає два цілочисельних аргументів за значенням.

Якщо ж необхідно, щоб функція модифікувала оригінал об'єкта, використовується передача параметрів за посиланням. При цьому в функцію передається не сам об'єкт, а лише його адреса. Таким чином, всі модифікації в тілі функції переданих їй за посиланням аргументів впливають на об'єкт. Беручи до уваги той факт, що функція може повертати лише єдине значення, використання передачі адреси об'єкта виявляється дуже ефективним способом роботи з великою кількістю даних. Крім того, так як передається адреса, а не сам об'єкт, істотно економиться стекова пам'ять.

В C++ передача за посиланням може здійснюватися двома способами:

- ✓ використовуються безпосередньо посилання;
- ✓ за допомогою покажчиків.

Синтаксис передачі з використанням посилань передбачає застосування в якості аргументу посилання на тип об'єкту.

Наприклад:

```
double Glue(float& var1, int& var2);
```

Функція отримує два посилання на змінні типу float і int. При передачі у функцію параметра-посилання компілятор автоматично передає в функцію адресу змінної, зазначеної в якості аргументу. Ставити знак амперсанду перед аргументом у викликах функції не потрібно. Наприклад, для попередньої функції виклик з передачею параметрів за посиланням виглядає наступним чином: Glue(var1, var2);

Прототип функції при передачі параметрів через покажчик:

```
void SetNumber(int*, float *);
```

Крім того, функції можуть повертати не тільки значення деякої змінної, але і покажчик або посилання на нього. Наприклад, функції, прототип яких:

```
*int Count(int);
&int Increase();
```

повертають покажчик та посилання відповідно на цілочисельну змінну типу int. Слід мати на увазі, що повернення посилання або вказівника зі функції може призвести до проблем, якщо змінна, на яку робиться посилання, вийшла з області видимості.

Приклад.

```
int Inc(int s) // передача параметра за значенням
{
s=s+1; // значення змінної s=1 return s;
}
void main()
{
int a=0;
cout<<a; // значення змінної a до виклику функції Inc, a=0
int b=Inc(a); // значення змінної b=1
cout <<a; // значення змінної a після виклику функції Inc,a=0
}
```

У цьому прикладі у функції *main* оголошується змінна a, яка потім передається у функцію *Inc* за значенням, тобто передається копія. Зміни, яким піддається змінна s у функції Inc не впливають на значення змінної a у функції *main*. Після повернення з функції *Inc* у функцію *main*, мінлива a як і до виклику функції *Inc* буде мати значення 0.

Один з недоліків передачі параметрів за значенням полягає в тому, що якщо передається великий елемент даних, створення копії цих даних може призвести до значних втрат часу виконання.

Інший спосіб передачі параметрів у функцію – передача параметрів за по силанням. У разі передачі параметрів за посиланням викликається функція отримує можливість прямого доступу до передаваних даних, а значить можливість зміни цих даних. Копії цих даних не створюються.

Приклад.

```
void swap(int &x, int &y);
int main ()
{
int i, j; i=10; j=20;
cout<<"i="<<i<<' '<<"j="<<j<<'\n';
swap (i, j);
```

```
cout<<"i="<<i<<' '<<"j="<<j<<'\n';
}
void swap(int &x, int &y)
{
int temp; temp=x; x=y; y=temp;
}
```

Виклик функцій з масивами.

Якщо масив є аргументом функції, то при виклику такої функції їй передається тільки адресу першого елемента масиву, а не повна його копія. Це означає, що оголошення формального параметра повинне мати тип, сумісний з типом аргументу.

Існує три способи оголосити формальний параметр, який приймає покажчик на масив:

1) Параметр можна оголосити як масив, тип і розмір якого збігається з типом і розміром масиву, що використовується при виклику функції.

Приклад.

```
void display(int num[10]);
int main ()
{
int t[10];
int i;
for (i=0; i<10; i++) t[i]=i;
display (t);
}
void display (int num [10])
{
int i;
for (i=0; i<10; i++) cout<<num [i]<<' ';
}
```

2) Параметр-масив можна оголосити як безрозмірний масив:

```
void display (int num [])
{
int i;
for (i=0; i<10; i++) cout<<num [i]<<' ';
}
```

3) При передачі масиву функції її параметр можна оголосити як дороговказ:

```
void display (int *num)
{
int i;
for (i=0; i<10; i++) cout<<num [i]<<' ';
}
```

Окремий елемент масиву, що використовується в якості аргументу, обробляється подібно до звичайної змінної.

Якщо масив використовується в якості аргументу функції, то функції передається адреса цього масиву. Це означає, що код функції може змінити реальний вміст масиву.

Приклад:

```
void cube (int *n, int kol);
int main()
{
int nums[10];
int i;
for (i=0; i<10; i++) nums[i]=i+1;
for (i=0; i<10; i++) cout<<nums[i]<<' ';
cout<<'\n';
cube (nums, 10);
for (i=0; i<10; i++) cout<<nums[i]<<' ';
}
void cube (int *n, int kol)
{
while (kol)
{
*n=*n**n**n;
 kol--;
 n++;
}
}
```

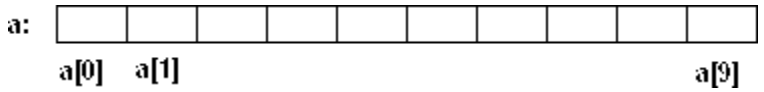
Тема 3.2. Вивчення роботи з багатовимірними масивами та методи динамічного виділення пам'яті для масивів

Вказівники та масиви.

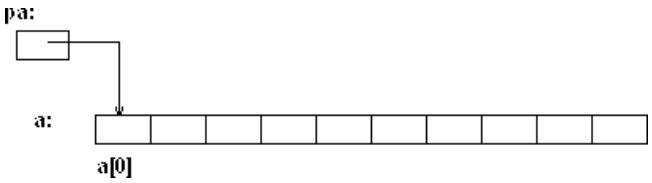
В С ++ існує тісний зв'язок між вказівниками і масивами. Будь-який доступ до елементу масиву, здійснюваний операцією індексування, може бути виконаний за допомогою вказівника.

Наприклад, коли оголошується масив у вигляді **int** a [25], то при цьому не тільки виділяється пам'ять для 25 елементів масиву, але і формується вказівник з ім'ям a, значення якого дорівнює адресі першого за рахунком (нульового) елемента масиву. Доступ до елементів масиву може здійснюватися через вказівник з ім'ям a. З точки зору синтаксису мови вказівник a є константою, значення якої можна використовувати у виразах, але змінити це значення не можна.

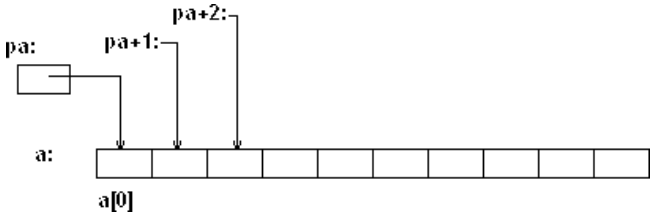
Оголошення **int** a [10]; визначає масив a розміру 10, тобто блок з 10 послідовних об'єктів з іменами a [0], a [1], ..., a [9].



Запис a [i] відсилає нас до i-му елементу масиву. Якщо pa є вказівник на **int**, тобто визначений як int * pa; то в результаті присвоювання pa = & a [0]; pa буде вказувати на нульовий елемент a; інакше кажучи, pa буде містити адресу елемента a [0].



Тепер присвоювання x = * pa; буде копіювати зміст a [0] в x. Якщо pa вказує на певний елемент масиву, то pa + 1 по визначенню вказує на наступний елемент, pa + i - на i-й елемент після pa, а pa-i - на i-й елемент перед pa. Таким чином, якщо pa вказує на a [0], то * (pa + 1) є зміст a [1], pa + i - адреса a [i], а * (pa + i) - зміст a [i].



Зроблені зауваження вірні до типу і розміру елементів масиву a. Сенс слів "додати 1 до вказівника", як і сенс будь-якої арифметики з вказівниками, в тому, щоб pa + 1 вказував на наступний об'єкт, а pa + i - на i-й після pa.

Між індексуванням і арифметикою з вказівниками існує дуже тісний

зв'язок. За визначенням ім'я масиву- це адреса його нульового елемента. Після присвоювання ра = & а [0]; ра і а мають одне і те ж значення. Оскільки ім'я масиву є не що інше, як адреса його початкового елемента, присвоювання ра = & а [0]; можна також записати в наступному вигляді: ра = а;

Крім того, а [і] можна записати як * (а + і). Зустрічаючи запис а [і], компілятор відразу перетворює її в * (а + і); зазначені дві форми запису еквівалентні. З цього випливає, що, отримані в результаті застосування оператора & записи & а [і] і а + і, також будуть еквівалентні, тобто і в тому, і в іншому випадку це адреса і-го елемента після а. З іншого боку, якщо ра - вказівник, то у виразах його можна використовувати з індексом, тобто запис ра [і] еквівалентна запису * (ра + і). Елемент масиву однаково дозволяється зображати і у вигляді вказівника зі зміщенням, і у вигляді імені масиву з індексом.

Між ім'ям масиву і вказівником, виступаючим в ролі імені масиву, існує одна відмінність. Вказівник - це змінна, тому можна написати ра = а чи ра ++. Але ім'я масиву є константою, і записи типу а = ра або а ++ не допускаються.

Відзначимо різницю в виразах: * ра + 1 і * (ра + 1).

Операція роз іменування має більш високий пріоритет, ніж операція додавання. Тому перший вираз спочатку роз іменує змінну ра і отримує перший елемент масиву, а потім додає до нього 1. Другий вираз доставляє значення другого елементу.

Масиви не самодостатні в тому сенсі, що не гарантовано зберігання інформації про кількість елементів разом з самим масивом. У більшості реалізацій С ++ відсутня перевірка діапазону індексів для масивів. Такий традиційний низькорівневий підхід до масивів. Більш повне уявлення масиву можна реалізувати за допомогою класів.

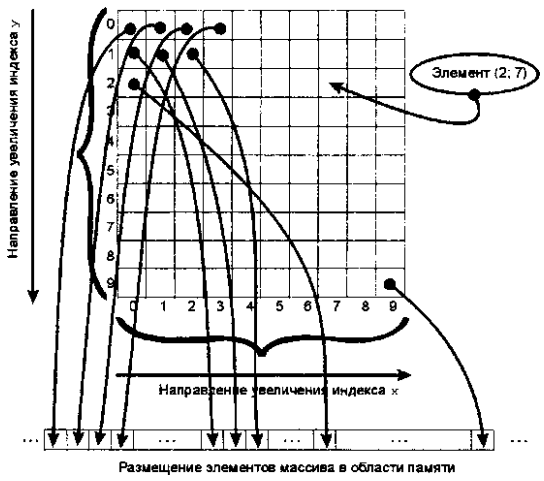
В С ++ масиви тісно пов'язані з вказівниками. Ім'я масиву можна використовувати в якості вказівника на його перший елемент. Гарантується осмисленість значення вказівника на елемент, що настає за останнім елементом масиву. Це важливо для багатьох алгоритмів. Але з огляду на те, що такий вказівник насправді не вказує ні на який елемент масиву, його не можна використовувати ні для читання, ні для запису. Результат отримання адреси елемента масиву, що передує першому, не визначений, і такої операції слід уникати.

Неявне перетворення імені масиву в вказівник на його перший елемент широко використовується у викликах функцій.

Багатовимірні масиви

Багатовимірні масиви – це масиви з більш ніж одним індексом. Частіше всього використовуються двовимірні масиви.

При описі багатовимірного масиву треба вказати С++, що масив має більш ніж один вимір.



Багатовимірний масив розмірності N можна представити як одномірний масив з масивів розмірності (N–1). Таким чином, наприклад трьохвимірний масив – це масив, кожний елемент якого представляє собою двохвимірну матрицю.

Приклади об'явлення багатовимірних масивів:

// Двохвимірний масив 6×9 елементів:

char Matrix2D[6][9];

// Тривимірний:

unsigned long Arr3D[4][2][8];

// Масив 7-ї степені вимірності:

my_type Heaven[22][16][7][47][345][91][3];

Вираз Array [idx] [idy], що представляє двовимірний масив, перекладається компілятором в еквівалентний вираз: * (* (Array + idx) + idy).

Тобто, багатовимірні масиви в мові С - це масиви масивів, тобто масиви, елементами яких, в свою чергу, є масиви. При оголошенні таких масивів в пам'яті комп'ютера створюється кілька різних об'єктів. Наприклад, при виконанні оголошення двовимірного масиву int a2 [4] [3] в програмі створюється покажчик a2, який визначає в пам'яті розташування першого елемента масиву і, крім того, є вказівником на масив з чотирьох вказівників. Кожен з цих чотирьох вказівників містить адресу одновимірного масиву, що представляє собою рядок двовимірного масиву і складається з трьох елементів типу int, і дозволяє звернутися до відповідної рядку масиву.

Таким чином, оголошення a2 [4] [3] породжує в програмі три різних об'єкта: вказівник з ідентифікатором a2, безіменний масив з чотирьох вказівників і безіменний масив з дванадцяти чисел типу int. Для доступу до безіменним масивів використовуються адресні вирази з покажчиком a2. Доступ до елементів масиву вказівників здійснюється із зазначенням одного індексного виразу в формі a2 [2] або * (a2 + 2). Для доступу до елементів двовимірного масиву чисел типу int повинні бути використані два індексних вирази в формі a2 [1] [2] або еквівалентних їй * (* (a2 + 1) +2) і (* (a2 + 1)) [2] . Слід враховувати, що з точки зору синтаксису мови С покажчик a2 і покажчики a2 [0], a2 [1], a2 [2], a2 [3] є константами, і їх значення не можна змінювати під час

виконання програми.

Розміщення тривимірного масиву відбувається аналогічно. Так, наприклад, оголошення float a3 [3] [4] [5] породжує в програмі, крім самого тривимірного масиву з 60 чисел типу float, масив з чотирьох покажчиків на тип float, масив з трьох покажчиків на масив покажчиків на float і покажчик на масив масивів покажчиків на float.

При розміщенні елементів багатовимірних масивів вони розташовуються в пам'яті поспіль по рядках, тобто швидше за все змінюється останній індекс, а повільніше - перший. Такий порядок дає можливість звертатися до будь-якого елементу багатовимірного масиву, використовуючи адресу його початкового елемента і тільки одне індексний вираз.

Наприклад, звернення до елементу a2[1][2] можна здійснити за допомогою покажчика ptr2, оголошеного в формі int * ptr2 = a2[0], як звернення ptr2[1 × 3 + 2] (тут 1 і 2 - це індекси використовуваного елемента, а 3 - число елементів в рядку) або як ptr2 [5]. Зауважимо, що зовні схоже звернення a2 [6] виконати неможливо, так як покажчика з індексом 6 не існує.

Для звернення до елемента a3 [2] [3] [4] з тривимірного масиву теж можна використовувати покажчик, описаний як float * ptr3 = a3 [0] [0], з одним індексним виразом у формі ptr3 [2 × 20 + 3 × 5 + 4] або ptr3 [59].

Багатовимірні масиви, як і одномірні, можуть бути ініційовані на етапі оголошення:

```
int ia [4][3] = {
{0, 1, 2},
{3, 4, 5},
{6, 7, 8},
{9, 10, 11}
};
```

Внутрішні фігурні дужки, що розбивають список значень на рядки, необов'язкові і використовуються, як правило, для зручності читання коду. Наведена далі ініціалізація в точності відповідає попередньому прикладу, хоча менш зрозуміла:

```
int ia [4][3] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Наступне визначення ініціалізує тільки перші елементи кожного рядка. Решта елементи будуть дорівнюють нулю:

```
int ia [4] [3] = {{0}, {3}, {6}, {9}};
```

Якщо ж опустити внутрішні фігурні дужки, результат виявиться зовсім іншим. Всі три елементи першого рядка і перший елемент другої отримають вказане значення, а інші будуть неявно ініціалізовані 0.

```
int ia [4] [3] = {0, 3, 6, 9};
```

Доступ до елементів багатовимірного масиву через вказівники здійснюється трохи складніше. Оскільки, наприклад, двовимірний масив Matrix [x] [y] може бути представлений як одновимірний (Matrix [x]), кожен елемент якого також є одновимірним масивом (Matrix [y]), покажчик на двовимірний масив pMtrx, посилаючись на елемент масиву Matrix [x] [y], по суті, вказує на масив Matrix [y] в масиві Matrix [x]. Таким чином, для доступу до вмісту осередку покажчик pMtrx доведеться розіменовувати двічі.

Важливо знати, що в багатовимірних масивах потрібен якийсь час на обчислення кожного індексу. Це означає, що доступ до елементу в багатовимірних масивах відбувається повільніше, ніж доступ в одновимірних масивах. З цієї та інших причин, якщо виникає необхідність в багатовимірних масивах, для них найчастіше пам'ять виділяється динамічне з використанням функції динамічного виділення пам'яті.

Динамічне виділення масивів

У програмі кожна змінна може розміщуватися в одному з трьох місць: в області даних програми, в стеці або у вільній пам'яті (так звана купа).

Кожній змінній в програмі пам'ять може відводитися або статично, тобто в момент завантаження програми, або динамічно - в процесі виконання програми. До сих пір всі обумовлені масиви оголошувалися статично, і, отже, зберігали значення всіх своїх елементів в стековій пам'яті або області даних програми. Якщо кількість елементів масиву невелика, таке розміщення виправдано. Однак досить часто виникають випадки, коли в стековій пам'яті, що містить локальні змінні і допоміжну інформацію (наприклад, точки повернення з вкладених функцій), недостатньо місця для розміщення всіх елементів великого масиву. Ситуація ще більш ускладняється, якщо масивів великого розміру має бути багато. Тут на допомогу приходить можливість використання для зберігання даних динамічної пам'яті.

Оператори вільної пам'яті new і delete

Всі змінні, які ми розглядали дотепер, статичні. Під час їх оголошення система автоматично надає для зберігання їх значень певний обсяг оперативної пам'яті, і цей розподіл пам'яті залишається незмінним протягом виконання програми. Пам'ять, надана цим змінним, резервується (фіксується) всередині ***exe-файлу*** скомпільована. Навіть якщо не всі змінні будуть використані в програмі, пам'ять буде зарезервована, тобто буде використана неефективно. Пам'ять, надана таким змінним, вивільняється лише після виконання програми. Тому в оперативній пам'яті можна розмістити лише обмежена кількість даних.

Однак є завдання, де заздалегідь невідомо, скільки змінних потрібно для їх вирішення, а, отже, який обсяг пам'яті потрібно зарезервувати, або завдання, в яких, навпаки, відразу відомо, що змінних буде багато, наприклад, завдання обробки масивів великих розмірів. У таких випадках застосовують динамічну організацію пам'яті.

Принцип динамічної організації пам'яті полягає в тому, що для змінних пам'ять виділяється в разі потреби (за вказівкою програміста). Далі ці змінні обробляють і в потрібний момент пам'ять вивільняють (знову за вказівкою програміста). Такі змінні називаються ***динамічними.***

Унарні оператори **new** і **delete** служать для керування динамічною (вільною) пам'яттю. Вільна пам'ять - це надана системою область пам'яті для об'єктів, час життя яких безпосередньо управляється програмістом. Програміст створює об'єкт за допомогою ключового слова **new** (перекладається з англійської як " новий "), а знищує його, використовуючи delete (перекладається з англійської як " видалити ").

Для роботи з динамічними змінними використовують покажчики. Для виділення динамічної пам'яті застосовується команда **new**. В С ++ оператор new приймає такі форми:

```
new і'мя_типу;
new ім'я_типу (ініціалізатор);
new ім'я_типу [вираз];
```

У кожному разі відбувається, щонайменше, два ефекти. По-перше, виділяється належний обсяг вільної пам'яті для зберігання зазначеного типу. По-друге, повертається базовий адреса об'єкта (як значення оператора **new**). Коли пам'ять недоступна, оператор new повертає значення 0 (**NULL**). Отже, можна контролювати процес успішного виділення пам'яті оператором **new**.

Дії команди ***new.*** Для відповідного типу змінній автоматично надається необхідна безперервна ділянка пам'яті. Команда **new** повертає обсяг цієї ділянки, а вказівник вказує на її початок. Наприклад, щоб зарезервувати в пам'яті комп'ютера область для зберігання значення цілого типу, застосовують таку команду:

```
int *prt=new int;
```

Надати ділянку пам'яті і відразу занести в неї значення можна так:

```
float *prt2=new float(3.14);.
```

На адресу, на який вказує *prt*2, буде занесено число 3,14.

З динамічної змінної можна виконати операції, визначені для даних відповідного базового типу.

Розглянемо наступний приклад використання оператору **new**:

```
int *p, *q;
p=new int (5);//виділили пам'ять і ініціалізували
q=new int [10];    //отримуємо масив від q[0] до q[9]
```

У цьому фрагменті вказівнику на ціле *p* присвоюється адреса комірки пам'яті, отримана при розміщенні цілого об'єкта. Місце в пам'яті, на яке вказує *p*, ініціалізується значенням 5. Такий спосіб зазвичай не використовується для простих типів на кшталт *int*, так як набагато зручніше і природніше оголосити

змінну звичним для нас чином. А ось використання прикладу з покажчиком *q* на масив зустрічається значно частіше. Це, так звані, динамічні масиви.

Після обробки динамічних змінних пам'ять необхідно вивільнити, а відповідну вказівку обнулити. Якщо цього не зробити, то пам'ять можна вичерпати. Вивільняють пам'ять за допомогою команди

```
delete <назва покажчика>
```

Оператор *delete* знищує об'єкт, створений за допомогою *new*, віддаючи тим самим розподілену пам'ять для повторного використання. Оператор *delete* може набувати таких форм:

```
delete вираз;
delete [] вираз;
```

Перша форма використовується, якщо відповідний вираз **new** розміщував не масив. У другій формі присутні порожні квадратні дужки, що показують, що спочатку розміщувався масив об'єктів. Оператор **delete** не повертає значення, тому можна сказати, що повертається їм тип - **void**.

Щоб вказівник не вказував ні на одну ділянку пам'яті, його необхідно обнулити такою командою:

```
<назва вказівника> = NULL;
```

Значенням (адресом) такого вказівника буде нульовий адрес 0x00000000. Тут не може бути розміщено значення жодного даного.

Приклад. Розглянемо, як проходить виділення пам'яті.

Виділимо пам'ять для двох Змінних цілого типу і присвоїмо їм деякі значення. Потім виділимо пам'ять.

```
#include <iostream.h>
void main()
{
int *c1=new int;   // Готуєм пам'ять для цілого числа
*c1=5;    // Змінна *c1 отримує значення 5
int *c2=new int(7);     // Виділяється пам'ять для c2 и *c2
отримує значеня 7
cout<<*c1<<'\t'<<*c2<<'\n'; // Виводимо 5 і 7
c1=c2;    // Переадресація - c1 буде вказувати на ту ж частину
пам'яті, що і c2
cout<<*c1<<'\t'<<*c2<<'\n'; // Виводимо 7 і 7
delete(c2);   //Пам'ять, представлена для c2, вивільняємо
c2= NULL;// Анулювання вказівника
cout<<*c1<<"\n";   // Виводимо 0
}
```

Справка. Замість значення NULL можна записати <назва вказівника>=0.

Багатовимірні динамічні масиви

При вирішенні на комп'ютері серйозних завдань, наприклад, при розробці додатків, що інтенсивно використовують ресурси графіки, завжди потрібно мати під рукою достатню кількість ресурсів, які лімітовані системою. Тому ефективні алгоритми і способи управління динамічною пам'яттю часто набувають вирішального значення. Принцип організації динамічного двовимірного масиву, який часто використовується в подібних випадках, найпростіше усвідомити за допомогою такої схеми:

Адреса масиву	Адреси масивів адрес	Серія окремих одномірних масивів		
A	*a*[0]	*a*[0][0]	*a*[0][1]	*a*[0][*n*-1]
	a[1]	*a*[1][0]	*a*[1][1]	*a*[1][*n*-1]
	...	...	...	...
	a[*n*-1]	*a*[*n*-1][0]	*a*[*n*-1][1]	*a*[*n*-1][*n*-1]

Відмінність описаної схеми від схеми статичного двомірного масиву полягає в тому, що тепер для адрес *a, a [0], a [1], ... a [n-1]* має бути відведено реальний фізичний простір пам'яті. У той час як для статичного двомірного масиву вираз *виду a, a [0], a [1], ... a [n-1]* були всього лише можливими конструкціями для посилань на реально існуючі елементи масиву, але самі ці вказівники не існували як об'єкти в пам'яті комп'ютера.

Алгоритм виділення пам'яті для двовимірного динамічного масиву такий:

1. Визначаємо змінну **a** як адресу масива адрес:

```
float **a;
```

2. Виділяємо область пам'яті для масиву з *n* покажчиків на тип *float* і присвоюємо адресу початку цієї пам'яті вказівником *a*. Оператор, який виконує ці дії виглядає так:

```
a=new float* [n];
```

3. В циклі пробігаємо по масиву адрес *a*[], присвоюючи кожному вказівнику *a*[*i*] адреса знову виділеної пам'яті під масив з *n* чисел типу **float**.

При роботі з масивами що задаються динамічно часто забувають звільняти пам'ять, захоплену для масиву. Пам'ять слід знову повертати в

розпорядження операційної системи, тобто звільняти за допомогою операції **delete**. Правда, при завершенні роботи функції **main** автоматично знищуються всі змінні, створені в програмі, і покажчики сегментів пам'яті отримують свої вихідні значення. Однак при розробці складних багатомодульних комплексів програм слід пам'ятати про те, що виділена пам'ять «повисає», стає недоступною операційній системі при виході з області дії покажчика, який посилається на її початок. Це може викликати відмову у виділенні нової пам'яті в якомусь іншому програмному модулі, якщо весь обсяг вільної області пам'яті буде вичерпаний. Операція **delete** спільно з операцією **new** дозволяє контролювати процес послідовного виділення і вивільнення динамічної пам'яті. Щоб звільнити пам'ять, виділену для однієї змінної d, наприклад, за допомогою оператора **double * d = new double** ;, досить в кінці функції або блоку, де використовувалася змінна d, записати **delete d** ;. Якщо був розміщений масив змінних, наприклад float

*** p = new float [200]**, то в сучасних версіях компіляторів слід звільняти пам'ять оператором **delete [] p ;.**

Тут квадратні дужки вказують компілятору на те, що звільняти слід ту кількість осередків, яке було захоплено в останній операції new в застосуванні до покажчика p. Явно вказувати це число не потрібно.

Приклад.

```
// Динамічне виділення та звільнення пам'яті
#include <iostream.h>
double *a;    // Одна змінна
double *d;    // Одномірний масив
double **dd;  // Двомірний масив

void GetMem()
{
int i;
       // Захват пам'яті
a=new double; // Одна змінна
```

```
d=new double[4];   // Одномірний масив
// Двомірний масив dd=new double*[3]; for(int i=0; i<3; i++)
dd[i]=new double[2];
// Присвоєння
*a=1.0;  // Одна змінна cout<<"a="<<*a<<endl; cout<<"Address
of a="<<a<<endl;
// Одномірний масив
cout<<"Massiv nachinaetsya po adresu"<<d<<" i soder*it\n";
for(i=0; i<4; i++)
{
d[i]=double(i);
cout<<"d["<<i<<"]="<<d[i]<<endl;
}
cout<<"2D-massiv nachinaetsya po adresu "<<dd<<" i soder*it
\n";
for(i=0; i<3; i++, cout<<endl)   // Двомірний массив
for(int j=0; j<2; j++)
{
dd[i][j]=(double)(i + j);
cout<<"dd["<<i<<"]["<<j<<"]="<<dd[i][j]<<endl;
}
}
void FreeMem()//Звільнення памяті
{
delete a;// Одна змінна
delete [] d;  // Одномірий масив
// Двомірний масив
for(int i=0; i < 3; i++)
delete [] dd[i];
delete [] dd;
}
int main()
{
GetMem(); FreeMem();
}
```

Після звільнення пам'яті покажчики *a, d, і dd* продовжують, проте, вказувати на ті ж адреси, що і до звільнення (проте ця пам'ять вже не наша). У цьому легко можна переконатися, вставивши до і після операцій **delete** висновок:

```
cout<<"a="<<a<<", d="<<d<<", dd="<<dd<<", dd[0]="<<dd[0];
```

Отже, необхідно акуратно працювати з вказівниками, які адресують звільнену пам'ять і стежити за зверненням до пам'яті, раніше займаної об'єктом.

Масиви в якості параметрів функцій

У тіло функцій в якості аргументів можна передавати значення, що зберігаються в масивах. При виконанні функції, параметр типу масиву перетворюється компілятором в покажчик на тип масиву. Наприклад, якщо аргумент-масив має тип unsigned long, при виклику він буде перетворений в *unsigned long* *. Таким чином, зміна в функції значення будь-якого елемента масиву, що є аргументом, обов'язково вплине і на оригінал. Масиви відрізняються від інших типів тим, що їх не можна передавати за значенням - всередину тіла функції потрапляє тільки адреса масиву.

Синтаксис виклику функції при цьому може бути наступним:

```
FunctionName(ArrayName);
```

Тоді прототип функції включає вказівку як параметр типу переданого масиву і наступних за ним прямокутних дужок. наприклад:

```
FunctionName(char[]);
```

Другий варіант синтаксису передачі масива в функцію – коли прототип функції містить символ операції взяття адреси після вказівки типу аргумента:

```
char FunctionName(char&);
```

При цьому синтаксис виклику функції приймає наступний вигляд:

```
FunctionName(*ArrayName);
```

Нижче приводиться приклад, ілюструючий обидва варіанти передачі масива в якості параметра функції.

```
#include <iostream.h>
// Прототипи функцій:
void Out1(int[]);
 void Out2(int&);
int main()
{
int Array[]={10,8,6,4,2,0};
// Виклик першої функції:
Out1(Array);
```

```
cout<<'\n';
// Виклик другої функції:
Out2 (*Array);
cout<<'\n';
}
// Реалізація обох функцій:
void Out1(int arr[])
{
for (int i=0; i<sizeof(arr); i++)
cout<<arr[i]<<' ';
}
void Out2(int& arr)
{
for(int i=0; i<sizeof(arr); i++)
cout<<*(&(arr)+i)<<' ';
}
```

У розглянутому прикладі оголошується масив *Array []*, який містить шість цілочисельних значень, і здійснюється його передача в функції *Out1 () і Out2 ()*. Обидві функції виконують одну і ту ж дію - виводять вміст масиву-аргументу на екран і відрізняються тільки інтерфейсом.

Неявне перетворення масиву у вказівник при виконанні функції призводить до втрати інформації про розмір масиву. Викликаюча, повинна якимось чином визначити цей розмір, щоб виконувати осмислені дії.

При оголошенні багатовимірного масиву як параметра функції можна опустити тільки першу розмірність.

int g (..., int x [] [10], ...) {...} // Друга і наступні розмірності обов'язкові

Використання в якості параметра функції багатовимірного масиву ускладнено, тому на практиці частіше за все здійснюється передача масиву вказівників, що значно спрощує синтаксис.

Приклад.

```
/*Програма обчислює суми елементів рядків двомірного
динамічного масиву*/
#include <iostream.h>
/*функцуя, обчислююча суму елементів строки */
int sum_str(int*, int);
int main()
{
int **data;   //змінна - вказівник на двомірний масив
int size_str; //тут будемо збнрігати розмір масиву
```

```
int size_stb;
int i;
cout<<"\nVvedite kolichestvo strok massiva:";
cin>>size_str;
cout<<"\n Vvedite kolichestvo stolbcov massiva:";
cin>>size_stb;
data=new int*[size_str]; for(int i=0; i<size_str; i++)
data[i]=new int[size_stb];
/* тут data використовується в якості базової адреси
динамічно розміщуваного двомірного масиву*/
//організовуємо ввід елементів масиву
for(i=0; i<size_str; i++)
{
for(int j=0; j < size_stb; j++)
{
cout<<"Vvedite A["<<i<<"]["<<j<<"]=";
cin>>data[i][j];
}
cout<<endl
}
//виводимо результат
for(i=0; i<size_str; i++)
cout<<" Symma=" <<sum_str(data[i],size_stb)<<"\n";
// звільнюємо пам'ять
delete[] data;}
/* функція виконує підрахунок суми елементів рядку*/
int sum_str(int *a, int size)
{
int sum=0;
for (int i=0; i<size; i++)
sum+=a[i];
return (sum);
}
```

Розглянемо приклад, який використовує динамічний розподіл пам'яті.

Приклад.

```
/* Програма обчислює середнє арифметичне елементів
динамічного масиву. Розмір масиву задається користувачем! */
 #include <iostream.h>   /*функція, що обчислює середнє
арифметичне елементів масиву */
double avg_arr(const int[], int);
 int main()
{
int *data;    //змінна - вказівник на ціле
int size;     //тут будемо зберігати розмір масиву
/ * Рядок виводить на екран напис - Введіть розмір масиву * /
cout<<"\Vvedite razmer massiva:";
/*запросили у користувача інформацію відносно розміру масиву
*/
cin>>size;
data=new int[size];
/* тут data використовується в якості базової адреси
динамічно розміщуваного масиву з кількістю елементів
задається значенням size */
//організуємо ввід елементів масиву
for (int j=0; j<size; j++)
{
cout<<"Vvedite element A["<<j<<"]=";
cin>>data[j];
}
//виводимо результат
cout<<"AVG massiva"<<avg_arr(data,size)<<"\n";
// звільняємо пам'ять
delete[] data;
}
/* функція виконує підрахунок середнього арифметичного
елементів масиву */
double avg_arr(const int a[], int size)
{
int sum=0;
for (int i=0; i<size; i++) sum+=a[i];
/* перетворимо sum до double, інакше отримали б тількм цілу
```

```
частину від ділення */
return double (sum)/size;
}
```

Тема 3.3. Дослідження операцій та роботи з рядками в мові програмування С++

Робота з рядками в С ++.

Дуже часто, на практиці, доводиться стикатися з завданнями, які зводяться до роботи над рядками. Але мова С ++ не підтримує окремий строковий тип даних

Рядок в С ++ - це масив символів, що закінчується нульовим символом ('\0').

Таким чином, можна визначити рядки двома способами: як масив символів або як покажчик на перший символ рядка, наприклад:

char str1 [] = "string1"; // оголошення рядка за допомогою масиву символів.

Отже, тепер докладніше. Масив - це набір однорідних значень. Так ось рядок є не що інше, як набір символів, і, відповідно, для зберігання рядків можна використовувати символьні масиви. Наприклад, рядок "QWERTY" має тип **char**, а порожній рядок "" має тип char. Чому **char**? Саме тому, що будь-який рядок завершується так званим нульовим символом, тобто символом, код якого в ASCII-таблиці дорівнює 0 (цей символ також є escape-символом і його символьний еквівалент представляється як '\ 0'). Завдяки цій властивості завжди можна визначити кінець рядка, якщо рядок займає меншу кількість символів, ніж та кількість, що було зазначено в квадратних дужках при оголошенні масиву, тобто визначити фактичну довжину рядка, що зберігається в масиві.

Одна з чудових особливостей при роботі з рядками - це можливість спрощеної початкової ініціалізації. Наприклад, оголошення **char** *str*[] = "ABCDE"; привласнює змінній-рядку початкове значення "ABCDE". А точніше, створює масив з 6 символів: 'A', 'B', 'C', 'D', 'E' і символу '\ 0'.

Початкова ініціалізація символьного масиву дійсно відрізняється від ініціалізації будь-якого іншого масиву - можна просто привласнити необхідний рядок імені масиву з порожніми квадратними дужками. С ++ сам підрахує довжину рядка і виділить відповідний обсяг пам'яті під масив для розміщення в ньому необхідного рядка.

Відразу необхідно зазначити, що С ++ сам автоматично зробить останній елемент масиву нульовим символом, тобто, хоча в даному випадку, масиву str присвоюється рядок "ABCDE", довжина якого становить 5 символів, С ++ виділяє пам'ять під 6 символів, записує туди рядок і потім в останній символ (п'ятий за рахунку від 0) записує нульовий символ.

Слід також зазначити, що при початковій ініціалізації символьного

масиву (як і будь-якого іншого) можна вказувати в квадратних дужках його розмір з метою подальшого використання масиву ще для будь-яких цілей (наприклад, для зберігання будь-якої іншої рядка). Оголошення **char** *str*[10]="ABCDE"; створює масив з 10 символів і перші п'ять елементів цього масиву приймають значення 'A', 'B', 'C', 'D' і 'E' відповідно, інші символи будуть нуль-символи. В даному випадку, в перші 5 елементів масиву записується рядок "ABCDE", а всім іншим елементам присвоюються нулі. Для початкової ініціалізації символьного масиву можна використовувати правила ініціалізації довільного масиву, тобто, використовуючи фігурні дужки, тільки в даному випадку доведеться явно вказувати нульовий символ, яким закінчується рядок. Давайте розглянемо попередній приклад з використанням загальних правил початкової ініціалізації масиву.

```
char str[]={'A','B','C','D','E','\0'};
```

Очевидно, спрощений варіант початкової ініціалізації строкового масиву значно простіший, але ще раз зазначимо, що його можна використовувати тільки для символьних масивів.

Типова помилка програмування.

1. Не виділяється достатньо місця в масиві символів для зберігання нульового символу, що завершує рядок.
2. Створення або використання "рядки", яка не містить завершального нульового символу.
3. Плутанина в символьні і рядкові константи.

Символьна константа - це один символ, взятий в апострофи, наприклад: 'A' або '\ n'. *Строкова константа* - це послідовність символів, взята в подвійні лапки. У числі символів рядка можуть перебувати будь-які символьні константи, наприклад, "Visual C ++ \ n" складається з наступних символів: 'V', 'i', 's', 'u', 'a', 'l', '', 'C', '+', '+', '\ n', '\ 0'. Таким чином, "A" - це строкова константа і складається з двох символів: 'A' і '\ 0'. Сусідні рядкові константи транслятором "склеюються", наприклад: "АБВ" "ДЕ" означає те ж, що "АБВГД".

Приклад 1.

```
// Задано рядок, скопіювати його в символьний масив
#include<iostream.h>
void main()
{
/ * Оголошуємо символьний масив str1 і ініціалізуємо його * /
   char str1[]="1234567890", str2[11];  // оголошуємо
символьний масив без ініціалізації
/ * В циклі, поки не зустрінеться кінець рядка присвоюємо
поточного елементу масиву str2 символ з масиву str1 */
for(int i=0; str1[i]!='\0'; i++) str2[i]=str1[i];
str2[i] = '\0';    // копіюємо нуль-символ в str2.
```

```
 cout<<str2<<'\n';  // вивід рядка на екран
}
```

Зверніть увагу, вихід з циклу відбувається, коли str1 [i] дорівнює нуль-символу, тобто нуль-символ не буде копіюватися в str2, отже, це потрібно зробити за циклом.

Ще одна особливість роботи з символьними масивами - якщо елементи довільного масиву можна вводити з клавіатури і виводити на екран тільки по одному елементу в циклі, то в символьний масив можна ввести відразу весь рядок, використовуючи оператор введення

```
cin>>Ім'я_масиву;
```

і, аналогічним чином, вивести відразу весь рядок на екран, використовуючи оператор виведення

```
cout<< Ім'я_масиву;
```

Слід відразу зазначити, що при введенні з клавіатури рядка оператор cin автоматично додає в кінець рядка нульовий символ, так що необхідно враховувати цей факт при вказівці кількості елементів при оголошенні масиву.

Приклад 2.

```
#include <iostream.h>
void main()
{
char str[31]; // оголошення символьного масиву
cout<<"Vvedit raydok (max 30 symvoliv):";
cin>>str; // введення рядка
cout<<"\nVu vveli raydok:"<<str; // виведення рядка
}
```

Як видно, в даному прикладі виділяється пам'ять під 31 символ, але користувачеві в запрошенні вказується, що він може ввести рядок з розміром максимум 30 символів, з огляду на той факт, що оператор cin додають ще один нульовий символ в кінець рядка автоматично, і під нього також необхідно передбачити виділення пам'яті. Далі після запрошення вводимо відразу весь рядок з клавіатури в масив і потім з відповідним повідомленням виводимо весь рядок на екран монітора.

Другий спосіб визначення рядка - це використання вказівника на символ. Оголошення **char** * b; задає змінну b, яка може містити адресу деякого об'єкту. Однак в даному випадку компілятор резервує місце для зберігання символів і не ініціалізує змінну b конкретним значенням. Зробити це можна, наприклад, присвоївши b покажчик на вже існуючий символьний масив, або динамічно виділити пам'ять під новий масив.

Приклад 3.

```
#include<iostream.h>
void main()
{
char str[]="Привіт, світ!"; // оголошуємо символьний масив
char *b; // оголошуємо вказівник на символ
b=&str[8];    // тепер b вказує на 8-ий сивол str
*b='M';  // присвоюємо першому елементу b символ 'C'
cout<<b; // виводимо рядок b на екран (Світ!)
}
```

Отже, підіб'ємо підсумки.

- Рядок можна визначити як масив символів або як вказівник на символ.
- Будь-яка рядок закінчується нульовим символом. (Завдяки цій властивості завжди можна визначити кінець рядка, якщо рядок займає меншу кількість символів, ніж та кількість, що було зазначено в квадратних дужках при оголошенні масиву).
- Для рядків можлива спрощена початкова ініціалізація (*в порівнянні з не символьними масивами*).
- У символьний масив можна ввести відразу весь рядок, використовуючи оператор введення **cin** >> *Ім'я_масиву* ;, і аналогічним чином вивести відразу весь рядок на екран, використовуючи оператор виведення **cout** << *Ім'я_масиву* ;.

Функції роботи з рядками з бібліотеки обробки рядків

Розглянемо деякі типові функції стандартної бібліотеки *string.h*. Це бібліотека обробки рядків, яка забезпечує багато корисних функцій для роботи із рядковими даними, наприклад, порівняння рядків, пошук в рядках потрібних символів і інших підрядків, розмітку рядків (поділ рядків на логічні шматки) і визначення довжини рядка.

1. Функція

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

Визначає довжину рядка s. Повертає кількість символів, що передують завершального нульового символу. Зверніть увагу, завершальний нуль-символ в довжину не включається. Наприклад,

```
cout<<strlen("Hello!"); // на екрані буде 6
char *str="one";
cout<<strlen(str); // на екрані буде 3
```

2. Функція

```
char *strcpy (char *s1, const char *s2);
```

Копіює рядок s2 в масив символів s1. Повертає значення s1. Масив символів s1 повинен бути досить великим, щоб зберігати рядок і її завершальний нульовий символ, який також копіюється.

Наприклад:

```
char str[25]; // оголошуємо символьний масив з 25 елементів
char *ps = new char [25]; /*Оголошуємо вказівник на символ і динамічно виділяємо пам'ять під 25 символів */
strcpy(str, "ABCDE");  // копіюємо в str строкову константу "ABCDE"
cout<<str; // виводимо str на екран. На екрані буде ABCDE
strcpy(ps, "QWERTY"); // копіюємо в ps строкову константу "QWERTY"
cout<<ps; // виводимо ps на екран. На екрані буде QWERTY
delete[] ps; // звільняємо пам'ять
```

Зверніть увагу , якщо треба, щоб один рядок містив інший, потрібно **скопіювати** його вміст, **а не присвоїти!** Так, наприклад, в даному випадку інструкція *ps* = "QWERTY" була б помилковою. Компілятор, зустрічаючи таку інструкцію, створюють рядок "QWERTY", за якою слідує нульовий символ і привласнює значення початкового адреси цього рядка (адреси символу Q) змінної ps. Таким чином, втрачається початкове значення *ps*, а значить неможливо коректно звільнити пам'ять під *ps*.

3. Функція

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

Порівнює рядки s1 і s2 (по ASCII-кодами). Функція повертає значення 0, якщо рядки s1 і s2 рівні, значення менше нуля, якщо рядок s1 менше s2, і значення більше нуля, якщо s1 більше s2. Зверніть увагу, рядки порівнюються не по довжині, а посимвольно, по ASCII-кодам (тобто "g" більше "ff"). Наприклад,

```
cout<<strcmp("compare", "string");  /* на екрані буде -1, оскільки "compare" менше "string" */
cout<<strcmp("abcde", "abc"); /* на екрані буде 1, оскільки "abcde" більше "abc" */
cout<<strcmp("one", "one"); /* на екрані буде 0, оскільки*/
```

4. Функція рядки рівні

```
char *strcat(char *s1, const char *s2);
```

Додає рядок s2 до рядка s1. Перший символ рядка s2 записується поверх нуль-символу рядка s1. Повертає s1. Під s1 має бути виділено пам'яті не менше ніж (strlen (s1) + strlen (s2) +1). *Наприклад*,

```
char st1[25] = "День";
cout<<strcat(st1, " добрий!");  // на екрані буде День
добрий!
```

5. Функція

```
char *strncpy(char *s1, const char *s2, int n);
```

Копіює не більше n символів рядка s2 в масив символів s1. Повертає s1.

6. Функція

```
int *strncmp(char *s1, const char *s2, int n);
```

Порівнює до n символів рядка s1 з рядком s2. Повертає 0, менше, ніж 0 або більше, ніж 0, якщо s1 відповідно дорівнює, менше або більше s2.

7. Функція

```
char *strncat(char *s1, const char *s2, int n);
```

Приєднує перші n символів рядка s2 до рядка s1. Повертає s1.

8. Функція

```
char *strchr(const char *s, char c);
```

Перевіряє рядок s на вміст символу, який зберігається в c. Результатом функції є адреса першого входження символу c в рядок s, тобто повертається рядок, який починається від першого входження заданого символу до кінця рядка s. Якщо символ не знайдено, повертається NULL. Застосовується в виразах. *Наприклад*,

```
char str[20]="ABCDEXYZ";
cout<<strchr(str, 'X'); // на екрані буде XYZ
або
char str[20]="ABCDEXYZ";
if (strchr(str, 'q') == NULL) cout<<"Нема такого символу!";
```

9. Функція

```
char *strstr(const char *s1, const char *s2);
```

Перевіряє рядок s1 на утримання підрядка s2. Результатом функції є адреса першого входження підрядка s2 в рядок s1. Якщо підрядок не знайдено, повертається NULL. *Наприклад*,

```
char str[20]="ABCDEXYZ";
```

```
char *ps=strstr(str, "DEX");
if (ps!=NULL)
cout<<ps;  // На екрані буде DEXYZ
else cout<<"Нема такого підрядка !";
```

10. Функція

```
char *strlwr(char *s);
```

Конвертує рядок до нижнього регістру (тобто переводить рядок в рядкові символи). *Наприклад*,

```
char str[30] = "ABCDE_123_ijk_XYZ"; cout<<strlwr(str);  // на екрані буде abcde_123_ijk_xyz
```

11. Функція

```
char *strupr(char *s);
```

Конвертує рядок до верхнього регістру (тобто переводить рядок в прописні символи).

12. Функція

```
char *strset(char *s, char ch);
```

Замінює ВСІ символи в рядку s на символ ch. *Наприклад*,

```
char str[30] = "ABCDE";
cout<<strset(str, 'x');  // на екрані буде xxxxx
```

13. Функція

```
char *strnset(char *s, char ch, int n);
```

Замінює перші n символів в рядку s на символ ch.

14. Функція

```
char *strrev(char *s);
```

Змінює порядок проходження символів в рядку на протилежний (змінює перший символ з останнім, другий символ з передостаннім і т.д.). Наприклад,

```
char str[30] = "12345";
cout<<strrev(str);   // на екрані буде 54321
```

Типова помилка програмування.

Необхідно включити заголовки *string.h* при використанні функцій з бібліотеки обробки рядків.

Також ознайомимося з двома функціями, які можуть допомогти програмісту при читанні символів з клавіатури:

Функція

int *getch*(void); Повертає ASCII-код натиснутої клавіші.

Функція

int *getche*(void);

Повертає ASCII-код натиснутої клавіші і виводить символ на екран.

Прототипи останніх двох функцій описані в файлі *conio.h*, який входить в стандартну бібліотеку мови С ++.

Робота з рядками в С ++. Приклади.

Приклад.

Задача. Дано рядок символів, підрахувати скільки разів серед символів рядка зустрічається буква x.

```
#include <iostream.h>
void main( void )
{
char str[100];// оголошення рядка символів cout<<"\nVvedite
stroky: ";
// просимо користувача ввести рядок символів
cin >> str;   // зчитуємо рядок, введену користувачем int
count = 0;    /* Оголошення змінної-лічильника, в якій
зберігатимемо кількість входжень x в рядок */
int i = 0; while(str[i]!='\0')
{
if (str[i]=='x') count++; i++;
}
cout<<"\n Simvol x vxodit v stroky -"<<count;
// виводимо результат на екран
}
```

Приклад.

Задача. Написати програму, яка отримує від користувача набір символів, виключаючи пробіл, і видаляє з цього набору все входження символів S і s.

Найбільший інтерес представляє аналіз рядка. Для реалізації цього аналізу потрібно поелементно рухатися від нульового індексу масиву до останнього і робити перевірку кожного елемента. Якщо буде зустрінутий такий елемент з індексом i, то нам потрібно змістити всі елементи з індексами, великими ніж i, на один індекс менше. Іншими словами:

Нехай дано наступний рядок: A b s D e f

0 1 2 3 4 5 - індекси

Перевіряючи індексно кожен елемент масиву, бачимо, що 2 елемент масиву і є шуканий символ. Тоді, потрібно змістити кожен елемент масиву на 1 індекс менше, тобто отримати наступний результат:

A b D e f

0 1 2 3 4

```
#include <iostream.h>

void main()
{
const int CharCount=10;
// задамо розмірність масиву через константу
char arr[CharCount]; // оголошення символьного масиву
// Попередимо користувача, що введення обмежено розмірністю
масиву  cout<<"\nVvedite stroky, no ne bolee, chem
"<<CharCount-1<<"simvolov\n";
cin >> arr; // введення рядка
int i=0;
while (arr[i]!='\0')
// Цикл працює поки не зустрінеться ознака кінця рядка
if (arr[i]=='S'||arr[i]=='s')
// Перевірка на наявність шуканого символу
{ /* Якщо це шуканий символ, то перенесемо решту рядка на
один елемент вліво. */
for (int j=i;arr[j]!='\0';j++) arr[j]=arr[j+1];
}
else i++;
// а якщо це не шуканий символ, то будемо рухатися по рядку
далі cout<<endl<<arr<<endl; // вивести результат
}
```

Приклад.

Задача. Написати програму порівняння двох рядків.

Перед тим як перейти безпосередньо до програми, зробимо примітку. У деяких випадках бажано вводити в масив повний рядок тексту. З цією метою C ++ забезпечений функцією **cin.getline (s).** Функція **cin.getline**

(s) вимагає три аргументи - масив символів, в якому повинна зберігатися рядок тексту, довжина і символ обмежувач. *Наприклад*, фрагмент програми

```
char sentence[80];
cin.getline(sentence, 80, '\n');
```

оголошує масив sentence з 80 символів, потім зчитує рядок тексту з клавіатури в цей масив. Функція припиняє зчитування символів у випадках, якщо зустрічається символ-обмежувач '\ n', якщо вводиться вказівник кінця файлу або

якщо кількість лічених символів виявляється на один менше, ніж зазначено в другому аргументі (останній символ в масиві резервується для завершального нульового символу). Якщо зустрічається символ обмежувач, він зчитується і відкидається. Третій аргумент **cin.getline (s)** має '\ n' в якості значення за замовчуванням, так що попередній виклик функції міг бути написаний в наступному вигляді:

```
cin.getline(sentence, 80);
#include<iostream.h> #include<string.h> void main()
{
int len; // довжина рядка, який вводиться
char s[81]; // місце зберігання рядка, який вводиться char
*s1,*s2;
cout<<"Vvedite pervyyu stroky: "; cin.getline(s, 80); //
введення першого рядка len = strlen(s);   // визначення
довжини рядка
s1 = new char[ len + 1]; // динамічне виділення пам'яті під
рядок s1 strcpy(s1, s); // копіювання введеного рядка в рядок
s1
cout<<" Vvedite vtoryyu stroky: "; cin.getline(s, 80); //
введення другого рядка
len = strlen(s);
s2 = new char[len + 1]; // динамічне виділення пам'яті під
рядок s2 strcpy(s2, s);
// який з введених рядків більше? if(strcmp(s1, s2) > 0)
cout<<"Stroka s1:\t"<<s1<<"\n\t > \n"
<< "Stroka s2:\t"<<s2<<endl;
 else if (strcmp(s1, s2) == 0)
cout<<"String s1:\t"<<s1<<"\n\t=\n"
<<"String s2:\t"<<s2<<endl; else
cout<<"String s1:\t"<<s1<<"\n\t < \n"
<<"String s2:\t"
<<s2<<endl;
delete []s1;  // видалення рядків з  пам'яті delete []s2;
}
```

Приклад.

Розглянемо таку задачу: необхідно реалізувати наступні функції для роботи з масивами.

1) Функція введення елементів масиву;
2) Функція роздруківки масиву;

3) Функція сортування масиву;
4) Функція додавання нового елемента в кінець масиву;
5) Функція видалення заданого елемента.

Для даної задачі були використані матеріали поточного уроку, а саме: передача масивів у функцію, передача параметрів по посиланню (щоб змінювалися реальні дані), повернення покажчика з функції.

Всі ці функції часто застосовні в реальному житті, наприклад, при створенні всіляких довідників.

```
#include <iostream.h>
int* Add(int*, int&); // Функція додавання елемента в кінець масиву
 int* Del(int*, int&); // Функція видалення елемента в заданій позиції
 void Input(int*, int);// Функція введення елементів масиву
void Sort(int*, int); // Функція сортування елементів масиву
 void Print(int*, int);// Функція роздруківки елементів масиву
 void main()  // Приклад реалізації
{
int n, *a;
cout<<"Vvedite razmer massiva:\t";
cin >> n;// Кількість елементів масиву
a = new int[n];// Виділення пам'яті для масиву з n елементів
   Input(a, n);    // Введення елементів масиву
cout<<"\nArray:\n";
Print(a, n);  // Виведення масиву на екран
Sort(a, n);   // Сортування масиву
cout<<"Otsortirovannuj massiv:\n";
Print(a, n);  // Виведення відсортованого масиву на екран
a = Add(a, n); // Зміна масиву - додавання нового елемента
cout<<"Novuj massiv:\n";
Print(a, n);  // Виведення зміненого масиву
a = Del(a, n);// Зміна масиву - видалення зазначеного елемента
cout<<" Novuj massiv:\n";
Print(a, n);  // Виведення зміненого масиву
delete [] a;  // Звільнення пам'яті, відведеної під масив
}
```

```
void Input(int *a, int n)
{
for(int i = 0; i < n; i++)
{
cout<<"Element #"<<i + 1<<"\t";
cin >> a[i];  // Введення елементів масиву
}
}
void Print(int *a, int n)
{
for(int i = 0; i < n; i++) cout<<a[i]<<"  ";
cout<<endl;   // Виведення елементів масиву
}
void Sort(int *a, int n)
{
int temp;// Тимчасова змінна для обміну значень
bool flag = true;  // Прапор закінчення сортування
for(int j = 1; ; j++)
{
for(int i = 0; i < n - j; i++)
if(a[i] > a[i+1])
{
temp = a[i];
a[i] = a[i+1];
a[i+1] = temp;
 flag = false;
}
if(flag == true) break;
flag = true;
}
}
int* Add(int *a, int &n)
{
int i, m;
int *p = new int[++n];
// Створення тимчасового масиву більшого розміру
```

```
cout<<"Enter the element:\t";
cin >> m;// Додається елемент
 for(i = 0; i < n - 1; i++)
p[i] = a[i];  // Збереження елементів
delete [] a;  // Видалення старого масиву
p[n - 1] = m; // Новий елемент
return p;// Повернення адреси нового масиву
}
int* Del(int* a, int &n)
{
int i, m, j = 0;
n--; // Зменшення розмірності масиву
int *p = new int[n];    // Створення тимчасового масиву
меншого розміру нуля)
cout<<"Element s kakim indeksom ydalim:\t";
cin >> m;// Введення індексу видаляється елемента (індекс з
for(i = 0; i < n; i++)
{
if(i == m) j = 1;

p[i] = a[i+j];          // Збереження елементів, не враховуючи
зазначений
}
delete [] a;  // Видалення масиву
return p;// Повернення адреси нового масиву
```

Приклад на багатовимірні динамічні масиви

Розглянемо наступну задачу: необхідно створити масив, що дозволяє створювати рядки і працювати з ними (аналог довідника). Для роботи з цим масивом рядків визначено такий набір операцій:

1) додавання рядка в кінець масиву;

2) вставка рядка в масив по заданому індексу;

3) видалення рядка з масиву по заданому індексу;

4) очистка масиву (видалення всіх рядків);

5) висновок на екран вмісту масиву.

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

// Набір констант, що представляють різні пункти меню
enum {ChoiceAddEnd=1, ChoiceInsert, ChoiceDelete, ChoiceDeleteAll,
ChoicePrint, ChoiceQuit};

int Menu();    // Виведення меню
char** AddLine(char**, int&);      // Додавання рядка в кінець
масиву char** InsLine(char**, int&);    // Вставка рядка в масив
char** DelLine(char**, int&); // Видалення зазначеного рядка з
масиву void DelAllLines(char**, int&); // Видалення всіх рядків
масиву
void Print(char**, int); // Роздруківка рядків масиву
bool IsArrayEmpty(int&); // Перевірка на наявність рядків в
масиві
void main()
{
char **c; // Масив рядків
int m = 0;// Початкова кількість рядків масиву int choice =
ChoiceAddEnd;

while (choice != ChoiceQuit)  // Поки не обраний пункт ВИХІД
{
choice = Menu();          // Виведення меню cin.ignore(1);   //
Очищення потоку введення

switch (choice)// Вибір пункту меню
{
case ChoiceAddEnd:
c = AddLine(c, m);  // Додавання рядка в кінець масиву break;

case ChoiceInsert:
c = InsLine(c, m);  // Вставка рядка в масив break;

case ChoiceDelete:
if (!IsArrayEmpty(m))    // Якщо масив не порожній c = DelLine(c,
m); // Видалення рядка
break;
```

```
case ChoiceDeleteAll:
if (!IsArrayEmpty(m))    // Якщо масив не порожній DelAllLines(c,
m); // Видалення всіх рядків масиву
break;
case ChoicePrint:
if (!IsArrayEmpty(m))    // Якщо масив не порожній Print(c, m);
     // Роздруківка масиву
break;
case ChoiceQuit: break;

default:  // В інших випадках cout<<"Error in choice!\n";
break;
}
}
}

char** AddLine(char **c, int &m)
// Додавання рядка в кінець масиву
{
char str[256];      // Масив для введення нового рядка int n;
     // Довжина введеного рядка
int i;
cout<<"Input string: ";
cin.getline(str, 256);   // Введення рядка
n = strlen(str);    // Обчислення довжини нового рядка
if (m == 0)    // Якщо масив рядків порожній
{
m++;
c = new char*[m];
c[0] = new char[n + 1];  // Створюємо новий рядок в масиві рядків
strcpy(c[0], str);
}
else
{
return c;
m++;
char** t = new char*[m];// Новий масив рядків for(i = 0; i < m -
1; i++)
{
t[i] = c[i];
```

```
}
t[m - 1] = new char[n + 1];
strcpy(t[m - 1], str);   // Копіювання нового рядка
delete [] c;
return t; // Повернення нової адреси масиву рядків }
}
}
char** InsLine(char **c, int &m)
// Вставка рядка в масив
{
char str[256];      // Масив для введення нового рядка int n;
     // Довжина введеного рядка
int k;    // Позиція нового рядка в масиві int i, j = 0;
cout<<"Input string: "; cin.getline(str, 256);
cout<<"Input position # (0-"<<m<<"): "; cin >> k;
while(k < 0 || k > m)    // Перевірка на хибність введення
{
cout<<"Error !!!\nInput position # (0-"<<m<<"): "; cin >> k;
}
n = strlen(str);    // Довжина нового рядка
if (m == 0)    // Якщо масив рядків порожній
{
m++;
c = new char*[m];
c[0] = new char[n + 1];  // Створюємо новий рядок в масиві рядків
strcpy(c[0], str);
return c; // Повернення нового адреси масиву рядків
}
else
{
m++;
char** t = new char*[m];// Новий масив рядків for(i = 0; i < m;
i++)
{
if (i == k)
{
t[i] = new char[n + 1];
strcpy(t[i], str);
j = 1;
}
```

```
else
{
t[i] = c[i - j];
}
}
delete [] c;   // Видалення масиву рядків
return t; // Повернення нового адреси масиву рядків
}
}
char** DelLine(char **c, int &m)
// Видалення зазначеного рядка з масиву
{
int k;    // Індекс видаляється рядка int i, j = 0;
cout<<"Input position # (0-"<<m - 1<<"): "; cin >> k;
while(k < 0 || k >= m)   // Перевірка на хибність введення
{
cout<<"Error !!!\nInput deleting position #: "; cin >> k;
}
m--;
char** t = new char*[m]; // Створення нового масиву рядків
for(i = 0; i < m; i++)
{
if (i == k) j = 1;
t[i] = c[i + j];
}

delete [] c;   // Видалення масиву рядків

return t; // Повернення нової адреси масиву рядків
}
void DelAllLines(char **c, int &m)
// Видалення всіх рядків масиву
{
for(int i = 0; i < m; i++) delete [] c[i];
delete [] c;   // Видалення всіх рядків масиву

m = 0;
}
bool IsArrayEmpty(int &m)
// Повертає істину, якщо масив рядків порожній; в зворотному
```

```
випадку - брехня
{
if (m == 0)
{
cout<<"Your line array is empty.\n"; return true;
}
else
return false;
}
int Menu()
// Виведення меню
{
int choice;
cout<<"\n******* Menu *******\n";
cout<<"1-Add 2-Insert 3-Delete 4-Delete All 5-Print 6-Quit\n"; cin
>> choice;// Вибір пункту меню
if(choice < 0 || choice > 6)  // Перевірка вибору
choice = 0;
return choice; // Повернення значення установок
}
void Print(char **c, int m)
// Роздруківка масиву рядків
{
for(int i = 0; i < m; i++) cout<<i<<": "<<c[i]<<endl;
}
```

Вказівники на функції

Перш ніж вводити вказівник на функцію, нагадаємо, що кожна Функція характеризується типом значення, що повертається, ім'ям, кількістю, порядком проходження і типами параметрів.

При використанні імені функції без подальших дужок і параметрів ім'я функції виступає в якості вказівник а на цю функцію, і його значенням служить адреса розміщення функції в пам'яті. Це значення адреси може бути присвоєно іншому вказівнику, і потім вже цей новий вказівник мож- на застосовувати для виклику функції. Однак у визначенні нового вказівник а повинен бути той же тип, що і повертається функцією значення, то ж кількість, порядок проходження і типи параметрів. Вказівник на функцію визначається наступним чином:

```
тип функції (* імя_вказівника) (специфікація_параметров);
```

Наприклад: int (* func1ptr) (char); - визначення вказівника func1ptr на функцію з параметром типу char, що повертає значення типу int.

Якщо наведену синтаксичну конструкцію записати без перших круглих дужок, тобто у вигляді int * fun (char); то компілятор сприйме її як прототип якоїсь функції з ім'ям fun і параметром типу char, що повертає значення покажчика типу int *.

Другий Приклад: char * (* func2Ptr) (char *, int); - визначення покажчика func2Ptr на функцію з параметрами типу покажчик на char і типу int, що повертає значення типу покажчик на char.

У визначенні вказівника на функцію тип значення і сигнатура (типи, кількість і порядок опцій) повинні збігатися з відповідними типами та сигнатурами тих функцій, адреси яких передбачається привласнювати вводиться вказівником при ініціалізації або за допомогою оператора присвоювання. В якості найпростішої ілюстрації - невеликий Приклад:

Приклад

```
/* Приклад визначення і використання вказівників на функції*/
#include<iostream.h>
// опис функції f1 void f1(void)
{
cout<<"\n Виконується f1 () ";
}
// опис  функції f2 void f2(void)
{
cout<<"\n Виконується f2 () ";
}
void main()
{
// оголошення вказівника на функцію,
void (*ptr)(void); // ptr - вказівник на функцію
 ptr = f2; // ptr ініціалізується адресом f2() (*ptr)(); //
виклик f2 () на її адресу
ptr = f1; // присвоюється адреса f1 ()
(*ptr)();
 // виклик f1 () на її адресу ptr();
// виклик еквівалентний (*ptr)();
}
```

Результат виконання програми:

```
Виконується f2() Виконується f1() Виконується f1()
```

У програмі описаний вказівник ptr на функцію, і йому послідовно присвоюються адреси функції f2 і fl. Зверніть увагу на форму виклику функції за допомогою вказівника на функцію:

```
(*ім'я вказівник)(список фактичних параметрів);
```

Значним імені вказівника служить адреса функції, а за допомогою операції розіменування * забезпечується звернення за адресою до цієї функції. Однак буде помилкою записати виклик функції без дужок у вигляді * ptr (); Справа в тому, що операція () має більш високий пріоритет, ніж операція звернення за адресою *. Отже, відповідно до синтаксисом буде спочатку зроблена спроба звернутися до функції ptr (). І вже до результату буде віднесена операція розіменування, що буде прийнято як синтаксична помилка.

При визначенні вказівник на функцію може бути ініціалізований. Як не започатковано значення має використовуватися адреса функції, тип і сигнатура (типи, кількість і порядок опцій) якої відповідають визначається вказівником.

При присвоєнні вказівників на функції також необхідно дотримуватися відповідність типів повертаються значенні функції і сигнатур для покажчиків правої і лівої частин оператора присвоювання. Те ж справедливо і при наступному виклику функцій за допомогою вказівників, тобто типи і кількість фактичних параметрів, використовуваних при зверненні до функції за адресом, повинні відповідати формальним параметрам, що викликається.

Наступна програма ілюструє гнучкість механізму викликів функцій за допомогою вказівників.

Приклад.

```
/* Виклик функцій за адресами через покажчик  */
 #include<iostream.h>
// опис функцій
// функції одного типу з однаковими сигнатурами
int add (int n, int m) { return n + m;}
int div (int n, int m) { return n % m;}
int mult (int n, int m) { return n * m;}
int subt (int n, int m) { return n - m;}
void main()
{
int (*par)( int); // вказівник на функцію
int a = 6, b = 2; /* задаються початкові значення для змінних
a i b */
char c = '+'; while(c != ' ')
```

```
{
cout<<"\nАргументи: a = "<<a
<< ", b = "<<b;
cout<<"\n Результат для c = \'"<<c<<"\'"
<< " дорівнює ";
switch(c)
{
case '+':
case '-':
par = add; // par отримує адрес функції add c = '%';
 break;
par = subt; // par отримує адрес функції subt c = ' '; break;
case '*':
par = mult; // par отримує адрес функції mult c = '-'; break;
case '%':
par = div; // par отримує адрес функції div c = '*'; break;
}
cout<<(*par)(a,b); /* виклик функції за адресою, результат,
який повертає функція, виводится на екран */
}
cout<<endl;
}
```

Результат виконання програми

```
Аргументи: a = 6, b = 2. Результат для c = '+' дорівнює 8
Аргументи: a = 6, b = 2. Результат для c = '%' дорівнює 0
Аргументи: a = 6, b = 2. Результат для c = '*' дорівнює 12
Аргументи: a = 6, b = 2. Результат для c = '-' дорівнює 4
```

Цикл триває, поки значенням змінної з не стане пробіл. У кожній ітерації вказівник par отримує адресу однієї з функцій, і змінюється значення c. За результатами програми легко простежити порядок виконання її операторів.

можуть бути об'єднані в масиви. Наприклад, float (* ptrArray) (char) [4]; - опис масиву з ім'ям ptrArray з чотирьох вказівників на функції, кожна з яких має параметр типу char і повертає значення типу float. Щоб звернутися, наприклад, до третьої з цих функцій, потрібно такий оператор:

```
float a = (*ptrArray[2]) ('f');
```

Як завжди, індексація масиву починається з 0, і тому третій елемент масиву має індекс 2.

Масиви вказівників на функції зручно використовувати при розробці всіляких меню, точніше програм, управління якими виконується за допомогою меню. Для цієї дії, пропоновані на вибір майбутнього користувачеві програми, оформляються у вигляді функцій, адреси яких містяться в масив вказівників на функції. Користувачеві пропонується вибрати з меню потрібний йому пункт і за номером пункту, як за індексом, з масиву вибирається відповідний адресу функції. Звернення функції до цього адресу забезпечує виконання необхідних дій. Найбільшу загальну схему реалізації такого підходу ілюструє наступна програма.

Приклад.

```
/* Приклад з масивом вказівників на функції */
 #include<iostream.h>
// оголошення функцій
void func1(int);
void func2(int); void func3(int);
void main()
{
/* оголошення масиву, в якому зберігаються вказівники на
функції func1, func2 и func3*/
void (*f[3])( int) = {func1, func2, func3}; int choice;

cout<<"Введіть число між 1 і 3," " друге число - закінчення:
";
cin >> choice;
while(choice >= 1 && choice < 4)
{
(*f[choice-1])(choice); // виклик функції f[choice] cout<<"
Введіть число між 1 і 3,"
" друге число - закінчення: ";
cin >> choice;
}
cout<<"Ви ввели "<<choice<<" для закінчення\n";
}
// опис функцій void func1(int a)
{
cout<<"\nВи ввели "<<a
<< ", тому була викликана func1 \n";
}
void func2(int b)
```

```
{
cout<<"\nВи ввели "<<b
<< ", тому була викликана func2\n";
}
void func3(int a)
{
cout<<"\nВи ввели "<<a
<< ", тому була викликана func3\n";
}
```

Результат виконання програми

```
Введіть число між 1 і 3, інше число - закінчення: 1 Ви ввели
1, тому була викликана func1
Введіть число між 1 і 3, інше число - закінчення: 2
Ви ввели 2, тому була викликана func2
Введіть число між 1 і 3, інше число - закінчення: 3 Ви ввели
3, тому була викликана func3
Введіть число між 1 і 3, інше число - закінчення: 10 Ви ввели
10 для закінчення
```

У програмі визначено три функції - func1, func2, func3 - кожна з яких приймає цілий аргумент і нічого не повертає. Вказівник на ці три функції зберігаються в масиві f, який оголошений таким чином:

```
void (* f [3]) (int) = {func1, func2, func3};
```

Масив отримує в якості початкових значень імена 3-х функцій. Коли користувач вводить значення 1, 2 або 3, це значення (мінус 1) використовується в якості індексу в масиві вказівників на функції.

Виклик функції виконується наступним чином:

```
(*f[choice-1])(choice);
```

У цьому виклику (* f [choice-1]) визначає вказівник, розташований в елементі масиву з індексом choice-1. Вказівник разіменовує, щоб викликати функцію, і choice передається функції як аргумент. Кожна Функція друкує ім'я функції, щоб показати, що виклик вірний.

ЗМІСТОВИЙ МОДУЛЬ 4
ВСТУП ДО ООП ТА БАЗОВІ КОНСТРУКЦІЇ КЛАСІВ

Тема 4.1. Вступ до об'єктно-орієнтованого програмування. Поняття класу та об'єкту

Необхідність використання об'єктно-орієнтованого підходу зумовлена бар'єром можливостей технологій процедурного програмування, що виник у 60-х роках XX століття. Причиною цієї кризи стало значне зростання складності та розміру програмних систем. При використанні процедурного підходу ускладняється структура програми та можливість її модифікації. Це пов'язано із зростанням числа глобальних змінних, функцій та зв'язків між ними. Крім того, процедурний підхід розділяє дані та методи їх опрацювання , що не відповідає картині реального світу, що складається із об'єктів, які одночасно характеризуються властивостями (даними) та поведінкою (діями). Логічне об'єднання даних та операцій над ними є головною ідеєю об'єктно-орієнтованої методології, що забезпечила подолання цих труднощів.

Об'єктно-орієнтоване проектування полягає у формуванні класів, об'єктів, та відношень між ними на основі характеристик (атрибутів) та дій (операцій).

Визначальною ідеєю об'єктно-орієнтованого підходу до розробки сучасних програмних продуктів є поєднання даних і дій, що виконуються над ними, в єдине ціле, яке називають об'єктом.

Функції об'єкта, які у мові програмування C++ називають методами або функціями-членами класу, зазвичай призначені для доступу до даних об'єкта та виконання певних дій над ними. Наприклад, якщо необхідно зчитувати будь-які значення даних об'єкта, то потрібно викликати відповідну функцію, яка їх зчитає та поверне об'єкту. Зазвичай прямий доступ до даних є неможливим, тому вони приховані від зовнішніх дій, що захищає їх від випадкових змін. Вважають, що дані та методи класу між собою інкапсульовані. Терміни приховування та інкапсуляція даних є ключовими в описі об'єктно-орієнтованих мов програмування.

Зв'язок між даними та методами їх обробки в об'єктно-орієнтованому програмуванні можна зобразити за допомогою наступної схеми:

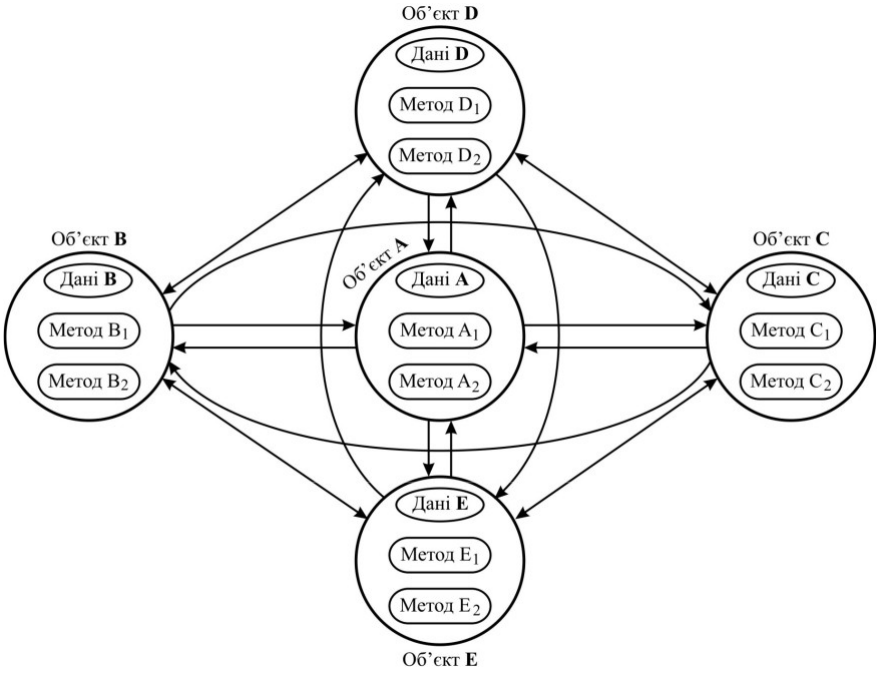


Рисунок 1 - Зв’язок між даними та методами їх обробки в об'єктно орієнтованому програмуванні

Об’єктно-орієнтоване програмування ґрунтується на трьох основних принципах:

- інкапсуляція даних та функцій (методів)їх опрацювання в єдиній інформаційній структурі, що називають класом:
- імпорт властивостей базових класів у похідні класи - успадкування класів;
- поліморфізм, який полягає у використанні однакових інтерфейсів для роботи з різними за функціональністю об’єктами.

Об’єктно-орієнтоване програмування ґрунтується на пріоритеті даних над кодом, на передачі повідомлень об’єктам за допомогою викликів відповідних функцій. На відміну від процедурного програмування, в об’єктно-орієнтованому підході не код керує даними, а дані керують кодом програми.

Об’єктно-орієнтований підхід до розробки програмних продуктів базується на такому понятті як клас. Клас визначає новий тип даних, який задає формат об’єкта. Клас містить як дані, так і коди програм, призначені для виконання дій над ними. Загалом, клас пов’язує дані з кодами програми. У мові програмування C++ специфікацію класу використовують для побудови об’єктів. Об’єкти – це примірники класового типу. Загалом, клас є набором планів і дій, які вказують на властивості об’єкта та визначають його поведінку. Важливо розуміти, що клас – це логічна абстракція, яка реально не існує доти, доки не буде створено об’єкт цього класу, тобто це те, що стане фізичним представленням цього класу в пам’яті комп’ютера.

Визначаючи клас, оголошують ті глобальні дані, які міститимуть об’єкти, і програмні коди, які виконуватимуться над цими даними. Хоча інколи прості класи можуть містити тільки програмні коди або тільки дані, проте більшість реальних класів містять обидва компоненти. У класі дані оголошуються у

вигляді змінних, а програмні коди оформляють у вигляді функцій. Змінні та функції називаються членами класу. Змінну, оголошену в класі, називають членом даних, а оголошену в класі функцію – функцією-членом класу. Іноді замість терміну член даних класу використовують термін змінна класу, а замість терміну функція-член класу використовують термін метод класу.

Тобто клас (class) – це структурований тип даних, що інкапсулює (приховує, об'єднує, обмежує область досяжності) оголошення полів даних (атрибутів) та функцій для їх опрацювання. Функції класу називають методами. Дані, які мають класовий тип, називають об'єктами або екземплярами класу.

Клас використовують для означення множини об'єктів, які мають однакову структуру, поведінку та відношення між об'єктами класів.

Структура (зміст) класу називається його протоколом. Дані та методи класу захищені оголошенням класу.

```
class ім'я_класу {
private:
дані та методи класу
public:
дані та методи класу
protected:
дані та методи класу
}
```

C++ надає даним та методам 3 рівні захисту:

- закриті (використовуються за замовчуванням) **private** можуть використовуватись тільки у межах класу;
- відкриті **public** досяжні у класі та поза класом у межах дії встановленого простору імен;
- захищені – **protected** – доступні у самому класі та похідних від нього.

Дозволяється довільний порядок розміщення розділів private, public та protected у протоколі класу.

Оголошення класу синтаксично є подібним до оголошення структури.

Загальні формати оголошення класу мають такий вигляд:

Варіант А

```
class ім'я_класу {
private:
закриті дані та функції класу
public:
відкриті дані та функції класу
```

```
} перелік_об'єктів_класу;
```

Варіант В

```
class ім'я_класу {
private:
закриті дані та функції класу
public:
відкриті дані та функції класу
};
```

```
ім'я_класу перелік_об'єктів_класу;
```

У цих оголошеннях елемент *ім'я_класу* означає ім'я "класового" типу. Воно стає іменем нового типу, яке можна використовувати для побудови об'єктів цього класу. Об'єкти класу інколи створюють шляхом вказання їх імен безпосередньо за закритою фігурною дужкою оголошеного класу як елемент *перелік_об'єктів_класу* (варіант А). Проте найчастіше об'єкти створюють за потребою після оголошення класу (варіант В).

Наприклад, наведений нижче клас під іменем ClassPoint визначає тип майбутнього об'єкта, призначеного для зберігання координат точки на площині:

```
class ClassPoint { // оголошення класового типу
double x; double y;
public:
void Init(); // ініціалізація даних класу ClassPoint
void Set(double, double); // зміна значень даних об'єкта
класу
void Get(); // виведення з об'єкта значення
};
```

Розглянемо детально механізм визначення цього класу. Усі члени класу ClassPoint оголошені в тілі настанови class. Членом даних класу ClassPoint є змінні x та y. Окрім цього, тут визначено три функції-члени класу: Init() – ініціалізація об'єкта, Set() – зміна даних об'єкта класу, Get() – виведення з об'єкта значення.

Кожен клас може містити як закриті, так і відкриті члени. За замовчуванням усі члени, визначені в класі, є закритими (private-членами). Наприклад, змінна a є закритим членом даних класу. Це означає, що до неї можуть отримати доступ тільки функції-члени класу ClassPoint; ніякі інші частини програми цього зробити не можуть. У цьому полягає один з проявів інкапсуляції: розробник повною мірою може керувати доступом до певних

елементів даних. Закритими можна оголосити функції (у наведеному вище прикладі таких немає), внаслідок чого їх зможуть викликати тільки інші члени цього класу.

Щоб зробити члени класу відкритими (тобто доступними для інших частин програми), необхідно визначити їх після ключового слова public. Усі змінні або функції, визначені після специфікатора доступу public, є доступними для всіх інших функцій програми, в тому числі і функції main(). Отже, в класі myClass функції Init(), Set() і Get() є відкритими. Зазвичай у програмі організовується доступ до закритих членів класу через його відкриті функції. Зверніть увагу на те, що після ключового слова public знаходиться двокрапка.

Визначивши клас, можна створити об'єкт цього "класового" типу, якщо використати ім'я класу. Отож, ім'я класу стає специфікатором нового типу. Наприклад, у процесі виконання наведеної нижче команди створюється два об'єкти PointA та PointB класу ClassPoint.

Після створення об'єктів класу кожен з них набуває власні копії членів-даних, які утворює клас. Це означає, що кожний з об'єктів класу ClassPoint матиме власні копії змінних x та y. Отже, дані, пов'язані з об'єктом PointA, відокремлені (ізольовані) від даних, які пов'язані з об'єктом PointB.

Щоб отримати доступ до відкритого члена класу через об'єкт цього класу, використовують крапкову нотацію (оператор "крапка"). Наприклад, щоб вивести на екран значення змінної x, яка належить об'єкту PointA, використовують таку команду:

```
cout << " PointA.x= " << PointA.x;
```

Закриті члени класу доступні тільки функціям, які є членами цього класу. Наприклад, таку вказівку PointA.x = 1.5; не можна помістити у функцію main(). Закриті дані класу можна змінити лише методами самого класу.

Для зміни даних класу використовують метод Set(). Даний метод не є обов'язковим, але вважається "правилом хорошого тону" використовувати метод саме з таким іменем для змін даних у об'єкті. Наведемо приклад його реалізації

```
class ClassPoint { // оголошення класового типу
double x; double y;
public:
void Set(double tmpX, double tmpY){ // зміна значень даних
об'єкта класу
x = tmpX; y = tmpY;
}
void Get(); // виведення з об'єкта значення
};
```

Для отримання даних класу використовують метод Get(). Для описаного нами класу ClassPoint взагалі доцільно оголошувати два Get()-методи: GetX()

та GetY() із такою реалізацією:

```
class ClassPoint { // оголошення класового типу
double x; double y;
public:
void Set(double tmpX, double tmpY); // зміна значень даних
об'єкта класу
double GetX(){// виведення з об'єкта значення
return x;
}
double GetY(){// виведення з об'єкта значення
return y;
}
};
```

Такий підхід спростить отримання доступу до даних та полегшить його підтримку.

Розглянемо приклад звернення до методів та даних класу ClassPoint. А також додамо до протоколу класу метод, який обчислюватиме відстань від точки, що викликає метод, до точки заданої як параметр методу.

```
class ClassPoint { // оголошення класового типу
double x;
double y;
public:
void Set(double tmpX, double tmpY){ // зміна значень даних
об'єкта класу
x = tmpX; y = tmpY;
}
double GetX(){// виведення з об'єкта значення
return x;
}
double GetY(){// виведення з об'єкта значення
return y;
}
double Distn(ClassPoint Point){ //метод обчислення відстані
між точками
return sqrt(pow((x-Point.x), 2)+ pow((y-Point.y), 2));
};
```

```
int main (){
ClassPoint PointA, PointB;
PointA.Set(12, -5); // демонстрація викликів методів у
наступних 3 рядках
PointB.Set(4, -3);
cout<<PointA.GetX()<<endl;
cout<<PointB.GetY()<<endl;
cout<<PointA.Distn(PointB); //   обчислення    відстані між
     точками PointA та PointB
return 0;
}
```

Результат роботи цієї програми наступний:

```
12
-3
10
```

Ще одним важливим поняттям у класах є покажчик this. Під час кожного виклику функції-члена класу їй автоматично передається покажчик на об’єкт, який іменується ключовим словом this, для якого викликається ця функція. Покажчик this – це *неявний* параметр, який приймається всіма функціями-членами класу. Отже, в будь-якій функції-члені класу покажчик this використовується для посилання на об’єкт, що викликається. За допомогою покажчика this можна отримати доступ до всіх елементів класу (даних та методів).

Щоби зрозуміти, як працює покажчик this, розглянемо модифікацію методів для класу ClassPoint.

```
class ClassPoint { // Оголошення класового типу
double x; double y;
public:
void Set(double tmpX, double tmpY){}; // Зміна значень
даних об'єкта класу
double GetX(){// Виведення з об'єкта значення
return this->x;
}
double GetY(){// Виведення з об'єкта значення
return this->y;
}
};
```

```
int main (){
ClassPoint PointA, PointB;
PointA.Set(12, -5); // демонстрація викликів методів у
наступних 3 рядках
PointB.Set(4, -3);
cout<<PointA.GetX()<<endl;
cout<<PointB.GetY()<<endl;
return 0;
}
```

Результат роботи цієї програми наступний:

```
 12
-3
```

Цей приклад тривіальний, але у ньому показано, як можна використовувати покажчик this. Практика у написанні програм мовою C++ надасть можливість впевнитись і зручності та важливості використання покажчика this.

Тема 4.2. Конструктори та деструктор класу.

Як правило, певну частину об'єкта, перш ніж його можна буде використовувати, необхідно ініціалізувати. Але, оскільки вимога ініціалізації членів-даних класу є достатньо поширеною, то у мові програмування C++ передбачено реалізацію цієї потреби при створенні об'єктів класу. Така автоматична ініціалізація членів-даних класу здійснюється завдяки використанню конструктора.

Конструктор – це спеціальна функція-член класу, яка викликається при створенні об'єкта, а її ім'я обов'язково збігається з іменем класу.

Особливості конструктора:

- призначений для виділення динамічної пом'яті та ініціалізації даних класу;
- якщо конструктор не оголошений, то компілятор згенерує конструктор за замовчуванням (без параметрів);
- для конструктора явно не вказується результуючий тип, конструктор не повертає значення;
- визначається в середині класу або поза протоколом класу;
- може перевантажуватися;
- може мати параметри за замовчуванням;
- не успадковується;
- може мати список ініціалізацій, який задається у заголовку після символу «:». (Так ініціалізуються константні поля (const), посилання на довільний

тип, поля з типом іншого класу, конструктори базових класів в разі успадкування.)

- може бути конструктором копіювання, якщо має один параметр з типом T& або const T&;
- не може бути викликаний явно з використанням об’єкта класу (або вказівника чи посилання на клас);
- при успадкуванні раніше викликаються конструктори базових класів, а потім – похідних.
- може бути розміщений у private, public, protected-частинах класу.
- автоматично викликається при створенні об’єкта у сегменті даних, стеку, типів, коли значення певного типу перетворюється у в об’єкт класу у виразі, передачі аргументів у функцію, поверненні результату із функції;
- не викликається при оголошенні вказівника або посилання на клас;
- при успадкуванні раніше викликаються конструктори базових класів, а потім – похідних.

Розглянемо приклад застосування конструктора для ініціалізації членів-даних класу:

```
class TestClass{ // оголошення класового типу
int first;
     public:
TestClass (int parametr){ // конструктор класу
first = parametr; cout<<"Object inicialized";
}
void setTestClass(int a) {first = a}; //set-метод
int getTestClass() {return first}; //get-метод
}
```

Зверніть увагу на те, що в оголошенні конструктора TestClass відсутній тип

значення, що повертається. У мові програмування C++ конструктори не повертають значень і, як наслідок, немає сенсу вказувати їх тип. У описаному вище коді у процесі виконання конструктора ним виводиться повідомлення ”Object inicialized” – " Об’єкт ініціалізовано", яке слугує виключно ілюстративній меті. На практиці ж здебільшого конструктори не виводять ніяких повідомлень.

Конструктор об’єкта викликається при створенні об’єкта. Це означає, що він викликається у процесі виконання вказівки створення об’єкта. Конструктори глобальних об’єктів викликаються на самому початку виконання програми, тобто ще до звернення до функції main(). Що стосується локальних об’єктів, то їх конструктори викликаються кожного разу, коли виникає потреба створення такого об’єкта.

Існує декілька типів конструкторів. Класифікуються вони за своїм призначенням:

– конструктори ініціалізації (звичайні);
– копіювання;
– перетворення типів.
– Конструктор ініціалізації викликається:
– при створенні нового об’єкта у сегменті даних, стеку, динамічній пам’яті та ініціалізації його полів (всіх або деяких з них);
– для створення локального об’єкта у виразі тип_класу (список параметрів конструктора).

У прикладі виклику конструктора для класу TestClass нами був реалізований саме конструктор ініціалізації.

Реалізація конструктора копіювання для класу TestClass матиме наступний вигляд:

```
class TestClass{ // оголошення класового типу
int first;
public:
TestClass (int parametr){ // конструктор ініціалізації
first = parametr;
}
TestClass (const TestClass &Object){// конструктор копіювання
first = Object.first;
}
void setTestClass(int a) {first = a}; //set-метод
int getTestClass() {return first}; //get-метод
}
```

Властивості конструктор копіювання:
– має один параметр з посиланням на тип класу. Може мати більше одно параметра, які задаються за замовчуванням;
– не допускає перевантаження — у класі можу бути лише один конструктор копіювання;
– якщо не визначений у класі, то конструктор копіювання генерується автоматично і виконує порозрядне копіювання об’єктів;
– виконує копіювання усіх видів полів даних;
– якщо поля класу мають тип вказівника або посилання на будь-який тип, то конструктор копіювання повинен бути визначений користувачем;
– якщо визначення конструктора копіювання визначено у закритій або захищеній частинах класу, то неможливо створити копію об’єктів поза межами класу (крім друзів класу).

Конструктор копіювання викликається:
– при створення нового об’єкта та його ініціалізації вже існуючим об’єктом цього ж класу;
– при передачі об’єктів через список параметрів функцій «за значенням»

(не через вказівники або посилання);

- при поверненні функціями локальних об’єктів класу «за значенням» (не через вказівники або посилання).

Конструктор перетворення типів призначений для перетворення значення заданого типу до типу класу. Має один параметр з типом, який вимагає перетворення, або декілька параметрів, значення яких задається за замовчуванням.

Реалізація конструктора перетворення типів для класу TestClass матиме наступний вигляд:

```
class TestClass{ // оголошення класового типу
int first;
public:
TestClass (float prm){ // конструктор перетворення типів
first = int(prm);
}
void setTestClass(int a) {first = a}; //set-метод
int getTestClass() {return first}; //get-метод
}
```

Викликається конструктор перетворення типіву таких випадках:

- для ініціалізації об’єкта класу значенням, тип якого є відмінним від цього класу;
- у виразах виду «класовий тип • інший тип»;
- для явного перетворення будь-якого типу до типу класу;
- в операції присвоєння «класовий тип = інший тип»;
- для передавання параметрів у функцію, коли формальний параметр має класовий тип, а фактичний – інший тип;
- для повернення значення функцією, коли функція має класовий тип, а вираз оператора return має інший тип.

Доповненням до конструктора слугує деструктор – це функція, яка викликається під час руйнування об’єкта. У багатьох випадках під час руйнування об’єкта необхідно виконати певну дію або навіть певні послідовності дій. Локальні об’єкти створюються під час входу в блок, у якому вони визначені, і руйнуються при виході з нього. Глобальні об’єкти руйнуються внаслідок завершення програми. Існує багато чинників, які заставляють використовувати деструктори. Наприклад, об’єкт повинен звільнити раніше виділену для нього пам’ять. У мові програмування C++ саме деструкторам доручається оброблення процесу де активізації об’єкта.

Ім’я деструктора має збігатися з іменем конструктора, але йому передує сим-вол "~" (тильда). Подібно до конструкторів, деструктори не повертають значень, а отже, в їх оголошеннях відсутній тип значення, що повертається.

Розглянемо вже знайомий нам клас TestClass, але тепер він має містити конструктор і деструктор:

```
class TestClass{ // оголошення класового типу
int first;
public:
TestClass (int parametr){ // конструктор класу
first = parametr; cout<<"Object inicialized";
}
~TestClass (){ // деструктор класу
cout<<"Object deleted";
}
void setTestClass(int a) {first = a}; //set-метод
int getTestClass() {return first}; //get-метод
}
```

Властивості:

- не має параметрів;
- не повертає значення;
- не перевантажується;
- не успадковується;
- існує за замовчуванням, але повинен бути оголошений, якщо для полів класу було виділено динамічну пам’ять.

ЗМІСТОВИЙ МОДУЛЬ 5
РОБОТА З ОБ'ЄКТАМИ, МАСИВАМИ ТА КОМПОЗИЦІЯМИ КЛАСІВ

Тема 5.1. Робота із масивами об’єктів

Масиви об’єктів можна організувати так само, як і масиви значень стандартних типів. Масиви об’єктів можуть оголошуватись одним із таких способів:

```
Тип_класу ідентифікатор_масиву [кількість об'єктів];
Тип_класу *вказівник = new Тип_класу [кількість об'єктів];
Тип_класу&    посилання = *new Тип_класу  [кількість
об'єктів];
```

В цілому робота із масивами об’єктів нічим не відрізняється від роботи із масивами елементів простих типів або структур (елементи типу struct): звертання відбувається за ідентифікатором масиву та номером елемента. Звернення до даних та методів класу здійснюється через крапкову нотацію. При оголошенні посилання на динамічний масив слід зважати на те, що ім’я посилання еквівалентно лише елементу з індексом 0. Для доступу до елемента слід враховувати зміщення відносно адреси нульового елемента.

Розглянемо наступний приклад. У наведеному нижче коді створюється клас displayClass, який містить значення розширення для різних режимів роботи монітора. У функції main() створюється масив для зберігання трьох об’єктів типу displayClass, а доступ до них здійснюється за допомогою звичайної процедури індексування елементів масиву.

```
enum resolution {low, medium, high}; class displayClass {
int width; int height;
resolution res; public:
void Set(int w, int h) {width = w; height = h; } void Get(int
&w, int &h) {w = width; h = height; } void Show(resolution r)
{res = r; }
resolution getRes() {return res; }
};
char names[3][9] = { "Low", "Medium", "Height"};
int main (){ displayClass Monitor[3]; Monitor[0].Show(low);
Monitor[0].Set(640, 480); Monitor[1].Show(medium);
Monitor[1].Set(800, 600); Monitor[2].Show(high);
Monitor[2].Set(1600, 1200); cout << "Regimes: " << endl; int
w, h;
```

```
for (int i=0; i<3; i++) {
cout << names[Monitor[i].getRes()] << ": "; Monitor[i].Get(w,
h);
cout << w << " x " << h << endl; }
return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Regimes:
Low: 640 x 480
Medium: 800 x 600
Height: 1600 x 1200
```

Варто звернути увагу на використання двовимірного символьного масиву names для перетворення перерахованого значення (enum) в еквівалентний символьний рядок. В усіх перерахунках (enum), які не містять безпосередньо заданої ініціалізації, перша константа має значення 0, друга – значення 1 і т.д.

Отже, значення, що повертається функцією getRes(), можна використовувати для індексації елементів масиву names, що дає змогу вивести на екран відповідну назву режиму відображення. Багатовимірні масиви об’єктів індексуються так само, як і багатовимірні масиви значень інших типів.

Розглянемо, як відбувається ініціалізація масивів об’єктів. Якщо клас містить параметризований конструктор, то масив об’єктів такого класу можна ініціалізувати. Наприклад, у наведеному нижче коді програми використовується клас myClass і параметризований масив об’єктів типу array цього класу, що ініціалізується конкретними значеннями.

```
class myClass { // оголошення класового типу
int a;
public:
myClass(int b) { a = b; }
double Put() { return a; } };

int main() {
myClass array[4] = { -1, -2, -3, -4 };
for (int i=0; i<4; i++)
cout << "array[" << i << "]= " << array[i].Put() << endl;
return 0;
}
```

Результати виконання цієї програми наступні:

```
array[0]= -1
array[1]= -2
array[2]= -3
array[3]= -4
```

Вони підтверджують, що конструктору myClass дійсно були передані значення від -1 до -4. Насправді синтаксис ініціалізації масиву, виражений рядком myClass array[4] = { -1, -2, -3, -4 };, є скороченим варіантом такого (довшого) формату: myClass array[4] = { myClass(-1), myClass(-2), myClass(-3), myClass(-4)};

Формат ініціалізації, представлений у наведеній вище програмі, використовується частіше, ніж його довший еквівалент, проте необхідно пам'ятати, що він працює для масивів таких об'єктів, конструктор яких приймає тільки один аргумент. При ініціалізації масиву об'єктів, конструктор яких приймає декілька аргументів, необхідно використовувати довший формат ініціалізації. Розглянемо такий приклад.

```
class myClass { // оголошення класового типу
int a, b;
public:
myClass(int c, int d) { a = c; b = d; }
int PutA() { return a; }
int PutB() { return b; }
};

int main() {
myClass array[4][2] = { myClass(1, 2), myClass(3, 4),
myClass(5, 6), myClass(7, 8),
myClass(9, 10), myClass(11, 12),
myClass(13, 14), myClass(15, 16) };
for (int i=0; i<4; i++)
for (int j=0; j<2; j++) {
{cout << "array[" << i << "," << j << "] ==> a= ";
cout << array[i][j].PutA() << "; b= ";
cout << array[i][j].PutB() << endl;
}
return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
array[0,0] ==> a= 1; b=2
array[0,1] ==> a= 3; b=4 array[1,0] ==> a= 5; b=6
array[1,1] ==> a= 7; b=8 array[2,0] ==> a= 9; b=10 array[2,1]
==> a= 11; b=12 array[3,0] ==> a= 13; b=14 array[3,1] ==> a=
15; b=16
```

У наведеному вище прикладі параметризований конструктор класу myClass приймає два аргументи. В основній функції main() оголошується та ініціалізується масив array об’єктів шляхом безпосередніх викликів конструктора myClass(). Для ініціалізації масиву можна завжди використовувати довгий формат ініціалізації, навіть якщо об’єкт приймає тільки один аргумент (коротка форма просто зручніша для застосування).

Крім одновимірних, можна використовувати і двовимірні масиви об’єктів.

Оголошуються такі масиви аналогічно до одновимірних:

```
Тип_класу ідентифікатор_масиву [кількість рядків] [кількість
стовпців];
```

Динамічна пам’ять для двовимірного масиву виділяється так:

```
Тип_класу (*вказівник) [кількість стовпців] = new Тип_класу
[кількість рядків] [кількість стовпців];
```

Кількість рядків та стовпців у цих оголошеннях повинні бути константами. Звільнення пам’яті, що виділяється під об’єкти динамічного масиву, відбувається так:

```
delete [] вказівник;
delete [] & посилання;
```

Тема 5.2. Види класів. Вкладені класи

Об’єктно-орієнтована програма здебільшого ґрунтується на основі декількох класів. Тому важливо визначити правила взаємовідношення між класами та їх розміщення у програмі.

Отже, класи можуть бути:

- глобальні, оголошення яких здійснюється поза функціями;
- локальні, які розміщуються всередині функції;
- дружні, яким надають права доступу до усіх частин класу;
- не вкладені, коли у протоколі класу немає оголошення інших класів;
- вкладені, коли у протоколі класу є оголошення інших класів;
- контейнери та колекції, які містять об’єкти інших класів;
- ітератори, призначені для забезпечення доступу до елементів інших

класів;
- успадковані, коли похідний клас використовує оголошення даних та методів базового класу;
- шаблонні, які є узагальненим визначенням множини класів;
- абстрактні, що містять нереалізовані віртуальні методи;
- інтерфейсні, призначені для однакового доступу до різних версій об’єктів (компонентів) відкомпільованого класу, незалежно від їх внутрішньої реалізації.

До цього часу ми розглянули лише один вид класів – глобальні невкладені. Глобальними називаються класи, оголошення яких розміщено поза функціями. Допускається використання імен вище розміщених класів для оголошення імен у нижче розміщених класах.

Одним із важливих видів класів є контейнерні. Контейнерним називається клас, який містить дані з типом іншого класу. Такий вид класів забезпечує можливість повторного використання коду програми.

Контейнерні класи будуються на основі аналізу внутрішньої структури складного об’єкта або системи та виявленні їх складових елементів або підсистем. Під час аналізу виділяються функціонально повні або самодостатні елементи (підсистеми), які дають змогу побудувати класи із можливістю їх подальшого повторного використання для побудови інших програмних продуктів. Між контейнерними класами існує відношення “містить”: контейнерний клас містить об’єкти-представники інших класів. Контейнерний клас називають зовнішнім, а інкапсульовані об’єкти – внутрішніми.

Наприклад, розглянемо клас ClassPoint, з яким ми знайомились у першій темі даного посібника. До протоколу цього класу додаємо метод printPoint виведення координат точки на екран.

```
class ClassPoint { // оголошення класового типу
double x; double y; public:
void Set(double tmpX, double tmpY){};  // зміна значень даних об'єкта класу
double GetX(){ // виведення з об'єкта значення
return this->x;
}
double GetY(){// виведення з об'єкта значення
return this->y;
}
void printPoint(){ // виведення координат точки на екран
cout<<"("<<x<<"; "<<y<<")"<<endl;
}
};
```

Побудуємо також клас ClassCircle, який містить дані для побудови кола

на площині. Такий клас буде контейнером для об’єкта класу ClassPoint, що використовується для задання координат центру кола на площині. До того ж клас ClassCircle містить поле radii для визначення радіусу кола. У відкритій частині класу ClassCircle розміщено конструктор ініціалізації даних та метод їх виведення на екран.

```
class ClassCircle {  // оголошення контейнерного класу
ClassPoint center; // інкапсульований об'єкт з координатами центру кола
double radii; // радіус кола
public:
ClassCircle (ClassPoint &p, double r): center(p) // конструктор ініціалізації
{ radii = r; }
void printCircle (){ // виведення координат точки та радіусу кола на екран
center.printPoint;
cout<<"radii = "<<radii<<endl;
}
}
```

Якщо у класі внутрішнього об’єкта перекрито конструктор за замовчуванням, то ініціалізація такого об’єкта повинна здійснюватися у списку ініціалізації конструктора контейнерного класу (одразу після символу “:”). За інших обставин ініціалізацію інкапсульованого об’єкта можна виконати у тілі контейнерного класу (у фігурних дужках “{ }”).

Розглянемо функцію main() для демонстрації роботи описаних нами вище класів. Головна функція оголошує об’єкти класів ClassCircle та ClassPoint, потім викликає метод printCircle() для того, аби вивести на екран інформацію про коло.

```
int main(){
ClassPoint point(-4, 7); // об'єкт класу
ClassPoint ClassCircle circle (point, 10); // об'єкт класу ClassCircle
circle.printCircle (); // виклик методу контейнерного класу ClassCircle
return 0;
}
```

Контейнери восновному будуються на основі шаблонних класів. Вони (контейнери) не допускають явного задання числа елементів і не підтримують розгалуженої структури. Найпростіші види контейнерів вбудовані

безпосередньо у мову C++.

Розглянемо приклад, коли до складу класу-контейнера входить не один інкапсульований об'єкт, а масив об'єктів. Нехай необхідно розробити клас File, який містить дані про назву, розмір, дату зміни та розширення файлу. Клас Catalog містить масив елементів класу File. Визначити методи роботи з файлами та каталогами.

Реалізація для такої задачі буде виглядати наступним чином.

```
class File{
private:
string name, date, attributes; float size;
public:
File (){ // конструктор без параметрів
name = "New file"; date = "00.00.00";
attributes = " - ";
 size = 0;
}
// конструктор з параметрами
void Set ( string name, string date, float size, string
attributes){
this->name = name;
this->date = date; this->size = size;
this->attributes = attributes;
}
void Set (){
cout << "Enter a file name - "; cin >> name;
cout << "Enter the date the file was created - ";
cin >> date ;
cout << "Enter the file size - ";
cin >> size;
cout << "Enter the file attributes - ";
cin >> attributes;
cout << endl;
}
void Get (){
cout << name << " " << date<< " " << time << " " << size
<< " kb\t\t" << attributes << " ";
}
~File (){}
```

```
};
class Catalog { // контейнерний клас
private:
File arr[3]; public:
void SetCatalog(File f1, File f2, File f3){ arr[0] = f1;
arr[1] = f2;
arr[2] = f3;
}
void GetCatalog(){
arr[0].Get();
arr[1].Get();
arr[2].Get();
}
~Catalog (){}
};

int main(){
File f1, f2, f3;
Catalog k;
cout << "За замовчування коструктор створює об'єкт із такими
значеннями полів";
f3.Get();
cout << "Відкоригуйте дані"; f1.Set("foto", "17.06.14", 1371,
"A");
f2.Set("System", "31.03.20", 4567, "DA");
f3.Set(); k.SetCatalog(f1, f2, f3);
cout << "Ваш файл потрапив до каталогу"; k.GetCatalog();
return 0;
}
```

ЗМІСТОВИЙ МОДУЛЬ 6
ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ ТА ОДИНАРНЕ УСПАДКУВАННЯ

Тема 6.1. Перевантаження операторів

У математиці зміст операції залежить від її операндів. Якщо a і b є цілими числами, їх сума a+b обчислюється по одним правилах, якщо матрицями — по іншим. У людини, що знає природу операндів, не виникає складнощів при інтерпретації змісту операції +.

Як відомо, кожен вбудований тип даних пов'язаний з визначеним набором операцій, які можна до нього застосовувати. Ці операції позначаються відповідними символами, зміст яких відомий компілятору заздалегідь. Однак цілком природно спробувати розширити застосування операторів на класи, що створюються програмістом. Наприклад, матричні обчислення стають набагато наочнішими, якщо сума двох матриць записується як a+b, а не як summa(a,b). Для цього в мові C++ існує механізм перевантаження операторів.

Операції у C++ є контекстно залежними і можуть бути перевантажені безпосередньо у класі.

Перевантаження допускається до таких операцій:

1. Первинних

[]	()	->

2. Бінарних

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,

Існують також оператори, заборонені до перевантаження. Зміна їх змісту зруйнувала би логіку програми. До таких операторів належать :: (оператор дозволу області видимості), . (“ точка” — оператор доступу до члена класу), ?: (тернарний оператор), .* (доступ до розіменованого вказівника-члена класу), sizeof, typeid, static_cast, dynamic_cast, const_cast і reinterpret_cast. Крім того, не рекомендується перевантажувати логічні оператори && і ||, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів. (Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються.)

Перевантаження операторів відбувається за допомогою операторних функцій.

Синтаксис операторних функцій виглядає наступним чином.

```
тип_значення_що_повертається operator
символ_операції(параметри)
{
...
}
```

Наприклад, операторна функція, що перевантажує операцію +, має вигляд operator+().

Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями.

Необхідно пам’ятати про обмеження, що супроводжують застосування перевантажених операторів.

- Перевантажені функції не можуть змінити пріоритет операторів.
- Кількість операндів фіксована: жодного, один чи два.
- Значення операндів не можна задавати за замовчуванням.

Розглянемо, як відбувається перевантаження унарних операторів за допомогою функцій-членів.

Оператори, як відомо, можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний – два. Нагадаємо, що до унарних операторів, що перевантажуються, належать такі оператори, як +, -, ++, --, &, ~ і !. До бінарних операторів, що перевантажуються, належать всі інші оператори, перераховані в таблиці вище.

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника this. Отже, така функція-член класу не має явних параметрів.

Розглянемо приклад перевантаження унарного мінуса.

```
class Tcomplex // оголошення класу для роботи із комплексними
числами
{
double Re; double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ } //конструктор з
параметром
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;} //конструктор
копіювання
~TComplex(){}
void printTComplex(){ cout<<x << " +i"<<y;
}
TComplex operator-() {Re = -Re; Im = -Im; return *this;}
```

```
//перевантаження оператора "-"
};

int main()
{
TComplex z(1,1),u(0,0); z.print();
u=-z; u.printTComplex(); return 0;
}
```

Помітимо, що операторна функція може повертати об’єкт класу (точніше, розіменований вказівник this) чи посилання на об’єкт, а може і нічого не повертати. Вибираючи тип значення, що повертається, необхідно керуватися здоровим глуздом. Якщо унарний оператор повинен використовуватися всередині виразів, то операторна функція повинна повертати об’єкт або посилання. Якщо ж оператор використовується ізольовано, функція може нічого не повертати.

Рядок програми u = - z; можна переписати в еквівалентному вигляді:

```
u = z.operator-();
```

Цей рядок демонструє, що виклик операторної функції operator-() здійснюється об’єктом z. Варто мати на увазі, що хоча зміст оператора перевантажувати можна, його природу змінювати заборонено. Це значить, що унарний оператор не можна перевантажити як бінарний і навпаки.

Розглянемо найбільш важливі приклади унарних операторів.

У мові C++ передбачено дві форми операторів інкремента і декремента: префіксна і постфіксна. Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій – вводиться фіктивний цілочисловий параметр.

Наприклад, для класу TComplex прототипи відповідних операторних функцій можуть виглядати в такий спосіб.

```
TComplex operator++();      // префіксна форма TComplex
operator++(int x); // постфіксна форма TComplex operator--();
// префіксна форма
TComplex operator--(int x); // постфіксна форма

Спробуємо розібратися на конкретному прикладі
перевантаження постфіксної та префіксної форм інкременції.
class TComplex
{
double Re; double Im;
```

```
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~Tcomplex(){}
void printTComplex(){ cout<<x << "+i "<<y;
}
TComplex& operator++()
{
++Re;
++Im;
cout<<"Префіксна форма ++"<<endl; return *this;
}
const TComplex operator++(int i)
{
++Re;
++Im;
cout<<"Постфіксна форма ++"<<endl; return *this;
}
TComplex& operator--()
{
--Re;
--Im;
cout<<"Префіксна форма --"<<endl; return *this;
}
const TComplex operator--(int i)
{
--Re;
--Im;
cout<<"Постфіксна форма --"<<endl; return *this;
}
};

int main()
{
TComplex z(1,1);
++z;
z.printTComplex(); z++;
```

```
z.printTComplex();
--z;
z.printTComplex(); z--;
z.printTComplex(); return 0;
}
```

У функції main() до об’єкта z послідовно застосовуються префіксна і постфіксна форми операторів ++ і --.

Результат роботи описаної вище програми наступний:

```
Префіксна форма ++ 2.000000 + i*2.000000
Постфіксна форма ++ 0 3.000000 + i*3.000000
Префіксна форма -- 000000 + i*2.000000
Постфіксна форма -- 0 1.000000 + i*1.000000
```

Якщо символ операції ++ стоїть перед операндою, викликається операторна функція operator++(), якщо після — операторна функція operator++(int i). Змінна і відіграє роль прапорця, що повідомляє компілятору, що дана функція перевантажує постфіксну форму оператора інкремента і декремента.

Незважаючи на те що розробник вільний самостійно трактувати перевантажений оператор, прагнучи зберегти аналогію з його убудованими значеннями, необхідно дотримуватись визначених правил. Наприклад, як відомо, оператор інкрементації цілих чисел повертає посилання на неконстантний об’єкт. Ця властивість повинна зберігатися і при перевантаженні.

Оператори заперечення (!), взяття адреси (&) і побітового заперечення (~) допускають перевантаження, але не мають універсальних альтернатив, хоча варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора ! можна позначати операцію звертання матриці, а за допомогою символу ~ — її транспонування. Щоправда, застосування тильди закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанину. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.

Оператор посилання на член об’єкта -> є унарним. Операторна функція, що перевантажує його, виглядає таким чином.

```
об'єкт -> елемент
```

Цей запис еквівалентний наступному виразу.

```
об'єкт.operator->(елемент);
```

Функція operator ->() повинна бути нестатичним членом класу. Як параметр вона одержує об’єкт чи класу посилання на нього, повертаючи

вказівник this на об’єкт, де виконується виклик, або посилання на об’єкт будь-якого іншого класу, у якому визначений оператор ->. її зручно використовувати в контейнерних класах, що містять усередині себе вказівник на інший клас. Основний зміст перевантаження оператора -> полягає в додатковій функціональності, що розширює можливості звичайних вказівників.

Наприклад, у приведеній нижче програмі функція operator->() веде підрахунок посилань на кожен об’єкт класу.

```
class TClass
{
int n;
int counter; public:
TClass(int x):n(x),counter(0) { } TClass* operator->();
int get(void) { return n;}
int ref(void) { return counter; }
};
TClass* TClass::operator ->()
{
counter++; return this;
}
int main()
{
TClass a(1), b(2);
cout<<"n = "<<a->get())<<endl; cout<<"n = "<<b->get())<<endl;
cout<<"n = "<<a->get())<<endl; cout<<"counter = "<<a-
>ref())<<endl; cout<<"counter = "<<b->ref())<<endl; return 0;
}
```

Врезультаті роботи програми на екран виводяться наступні рядки.

```
n = 1
n = 2
n = 1
counter = 3
counter = 2
```

Розглянемо, як відбувається перевантаження бінарних операторів за допомогою функцій-членів. Бінарний оператор має два операнди. Його виклик виконується об’єктом, розташованим у лівій частині оператора. Отже, бінарний оператор

```
a+b
```

еквівалентний такому оператору.

```
a.operator+(b)
```

Таким чином, бінарна операторна функція-член повинна має тільки один параметр, що задає другий операнд. Вказівник this на перший операнд вона одержує неявно.

Для того щоб бінарну операторну функцію можна було застосовувати всередині виразів, необхідно, щоб вона повертала об’єкт свого класу. Розглянемо приклад перевантаження бінарного оператора + на прикладі класу TComplex.

```
class TComplex{
double Re; double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void printTComplex(){ cout<<x << " +i"<<y;
}
TComplex operator+(TComplex z){
TComplex w(0,0); w.Re = Re+z.Re; w.Im = Im+z.Im; return w;
}
};

int main()
{
TComplex u(1,1),v(2,2),z(0,0);
z=u+v;
z.printTComplex();
 return 0;
}
```

У цій програмі сума комплексних чисел u і v присвоюється об’єкту z.

Виклик операторної функції operator+() здійснюється з об’єкта u. Оскільки сума двох об’єктів класу TComplex є об’єктом цього ж класу, не обов’язково створювати новий об’єкт і записувати в нього результат підсумовування. Можна скористатися наступною синтаксичною конструкцією.

```
(u+v).printTComplex();
```

У цьому випадку функція printTComple() буде викликатися тимчасовим об’єктом, створеним операторної функцією operator+(). Цікаво, що цей механізм допускає вживання “безглуздих виразів”. Наприклад, компілятор не заперечує проти такого оператора.

```
(u+v)=z;
```

Для того щоб запобігти цьому, оператор + повинний повертати константний об’єкт. Крім того, бажано, щоб він не змінював другий операнд. Отже, параметр повинний бути константним. І останнє: тому що об’єкт може бути досить громіздким, його варто передавати за значенням. Отже, одержуємо наступну реалізацію.

```
const TComplex operator+(const TComplex& z){
TComplex w(0,0); w.Re =
Re+z.Re; w.Im= Im+z.Im; return w;
}
```

Продемонструємо, як здійснюються обчислень над комплексними числами із використанням перевантажених операторів.

```
class TComplex
{
double Re; double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void printTComplex();
TComplex operator+(const TComplex& z) // додавання
{
TComplex w(0,0); w.Re = Re+z.Re; w.Im = Im+z.Im;
cout<<"Operator + "<<endl; return w;
}
TComplex operator-(const TComplex& z) // віднімання
{
TComplex w(0,0); w.Re = Re-z.Re;
```

```
w.Im = Im-z.Im; cout<<"Operator - "<<endl;
return w;
}
TComplex operator*(const TComplex& z)// множення
{
TComplex w(0,0);
w.Re = Re*z.Re-Im*z.Im; w.Im = Re*z.Im+Im*z.Re;
pcout<<"Operator * "<<endl; return w;
}
TComplex operator-()    // унарний мінус
{
Re = -Re;
Im = -Im;
cout<<"Unary operator - "<<endl; return *this;
}
TComplex operator~()    // спряжене комплексне число
{
TComplex w(0,0); w.Re = Re;
w.Im = -Im;
cout<<"Unary operator ~ "<<endl; return w;
}
TComplex operator/(TComplex& z)  // ділення
{
TComplex w(0,0); w=(*this)*(~z);
w.Re = w.Re/modul2(z); w.Im = w.Im/modul2(z); cout<<"Operator
/ "<<endl;
return w;
}
double modul2(const TComplex& z)
{
return z.Re*z.Re+z.Im*z.Im;
}
};

int main()
{
TComplex u(1,1),v(2,2),z(0,0); z=(u+v)*u/v;
```

```
z.printTComplex();
return 0;
}
```

У класі TComplex операція розподілу двох комплексних чисел реалізована через обчислення спряженого числа (операція ~) і розподіл дійсної і уявної частин на квадрат модуля дільника (функція modul2()).

От у якому порядку виконувалися зазначені оператори.

```
operator +
operator * Unary
operator ~
operator *
operator /
```

У підсумку, як і слід було очікувати, одержуємо наступне число.

```
1.500000 + i*1.500000
```

Реалізуючи ланцюжок обчислень, необхідно обережно використовувати ключове слово const. Безперечно, захист параметра від необережної модифікації необхідний. У той же час повернення константних об'єктів не завжди виправданий.

У програмах, розглянутих вище, ми вільно маніпулювали об'єктами, привласнюючи їх один одному. Це відбувалося завдяки вбудованому оператору присвоювання, що виконує побітове копіювання. Як і у випадку з конструктором копіювання, це не краще рішення. Якщо об'єкт містить вказівник на деяку область пам'яті, його копія також буде посилатися на неї. Для того, щоб цього не відбулося, перевантажений оператор присвоювання повинен виконувати глибоке копіювання, уникаючи подвійної адресації.

```
class TArray{
int *p;
int size;
public:
TArray(long n, int x); TArray(TArray&);
TArray& operator=(TArray& X); void view();
};

int main()
{
TArray x(10,1),y(10,0);
x.view();
y = x;
```

```
y.view();
return 0;
}

TArray::TArray(long n, int x)
{
size = n;
p = new int[size];
for (long i=0; i<size; i++)p[i] = x; cout<<"Адреса =
"<<p<<endl;
}


TArray::TArray(TArray& X)
{
size=X.size;
p = new int[size]; // глибоке копіювання
for (long i=0; i<X.size; i++) p[i] = X.p[i];
cout<<"Адреса = "<<p<<endl;
}

void TArray::view()
{
for(long i=0; i<size; i++) cout<<" "<<p[i]);
}

TArray& TArray::operator=(TArray& X)
{
if(this == &X) return *this;// перевірка
самоприсвоювання.
if(size==X.size)   // глибоке копіювання
// (вказівник не копіюється!)
for (long i=0; i<X.size; i++) p[i] = X.p[i];
else cout<<" Size !"<<endl; cout<<endl;
return *this;
}
```

На екрані будуть виведені наступні рядки.

```
Адреса = 008803A0
Адреса = 00880340
1111111111
1111111111
```

Зверніть увагу на два моменти. По-перше, у перевантаженому операторі присвоювання виконується не побітове, а глибоке копіювання. Адреса об'єкта у залишається незмінною, змінюється лише зміст. По-друге, у функції operator=() передбачена перевірка саме присвоювання. Це дозволяє коректно обробляти вирази виду x = x;

Незважаючи на зовнішню схожість перевантаженого оператора присвоювання і конструктора копіювання, між ними існує принципова різниця. При виклику конструктора копіювання створюється новий об'єкт, що ініціалізується раніше існуючим об'єктом. При присвоєнні обидва об'єкти вже існують.

В арифметичних обчисленнях дуже корисні скорочені оператори присвоювання. Розумно спробувати перевантажити їх для знову створюваних класів. При цьому варто врахувати обмеження, що накладаються на оператор присвоювання й арифметичних операторів. Причому, як правило, перевантажені арифметичні оператори реалізуються за допомогою скорочених операторів присвоювання.

```
class TComplex{ double Re; double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void printTComplex();
const TComplex operator+=(const TComplex& z) // додавання
{
Re = Re+z.Re;
Im = Im+z.Im;
cout<<"Operator += ";
return *this;
}
const TComplex operator+(const TComplex& z)    // додавання
{
TComplex w=*this; w+=z; cout<<"Operator + ";
return w;
```

```
}
};

int main()
{
TComplex u(1,1),v(2,2),z(3,3);
u+=v;
u.print(); z=u+v;
z.printTComplex();
return 0;
}
void TComplex::print()
{
cout<<Re<<" +i"<<Im);
}
```

На початку цієї теми ми виділили оператор [] як особливий клас операторів для перевантаження — первинний. Розглянемо, як здійснюється зміна його змісту. Бінарний оператор доступу до члена масиву перевантажується за допомогою наступної операторної функції, що повинна бути нестатичним членом класу.

```
тип_ що повертається_значення&
ім'я_класу::operator[](інтегральний_тип і)
{
// ...
}
```

Відзначимо два моменти. По-перше, оскільки операція доступу застосовується до індексованих масивів, параметр операторної функції повинний бути цілочисельним. По-друге, елементи масиву можуть стояти як у лівої, так і в правій частині оператора присвоювання. Отже, функція operator[]() повинна повертати посилання або вказівник.

Хоча зовні функція operator[]() виглядає унарною, насправді вона є бінарною. Її перший параметр явно задає індекс елемента, а другий параметр, що представляє собою вказівник this на об’єкт, де виконується виклик, передається неявно.

От типовий приклад використання цього оператора. Для демонстрації прикладу використаємо клас, що містить в собі розмірність двовимірного масиву дійсних чисел (кількість стовпців та кількість рядків) та вказівник на сам масив.

```
class TMatrix{
```

```
int n;
public:
TMatrix(int x,int y); //конструктор з параметром
TMatrix(); //конструктор без параметрів

~TMatrix() { if (p!=NULL) delete p; p=NULL; } //деструктор
double* operator[](int i) { return p+i*m; }
//перевантажений оператор []
void output();//виведення матриці на екран};

TMatrix::TMatrix(int x,int y):n(x),m(y)
{
p=new double[n*m];
for (int i=0; i<n; i++) for (int j=0; j<m; j++)
if(i==j)p[i*m+j]=1; else p[i*m+j]=0;
}

void TMatrix::output()
{
for (int i=0; i<n; i++)
{
for (int j=0; j<m; j++)
{
cout<<" ", (*this)[i][j];
}
cout<<endl;
}
cout<<endl;
}

int main()
{
TMatrix c(3,3); c[1][2]=2.0;
c[2][1]=2.0;
c.output();
return 0;
}
```

Як відомо, у мові C++ за замовчуванням не передбачена перевірка виходу індексу масиву за припустимі межі. З цієї причини визначення класу TMatrix варто було б доповнити генеруванням відповідних виняткових ситуацій.

Розглянемо також особливості перевантаження оператора (). Об'єкти, що містять операторну функцію operator()(), називаються функціями-об'єктами, чи функторами. Ця функція може одержувати довільну кількість параметрів і повертати значення будь-яких типів. Такі об'єкти бувають корисними при виконанні операцій, зв'язаних з декількома індексами. Як відзначалася вище, операторна функція operator()() повинна бути нестатичним членом класу.

Проілюструємо створення функторів програмою, у якій функція operator()() повертає заданий рядок матриці.

```
class TMatrix;
class Str
{
int m; double *q;
public:
Str(int x):m(x) {q = new double[m];}
double& operator[](int i) { return
*(q+i); } void output();
friend class TMatrix;
};

class TMatrix
{
int n; int m;
double *p;
public:
TMatrix(int x,int y);

~TMatrix() { if (p!=NULL) delete p; p=NULL; }
Str operator()(int);

double* operator[](int i) { return p+i*m; }
void output();
};
TMatrix::TMatrix(int x,int y):n(x),m(y)
{
p=new double[n*m];
for (int i=0;
```

```
i<n; i++) for (int j=0;
j<m; j++)
if(i==j) p[i*m+j]=1;
else p[i*m+j]=0;
}
Str TMatrix::operator()(int i)
{
Str s(3);
for (int j=0;
j < m;
j++) s[j]=(*this) [i][j];
 return s;
}
void TMatrix::output()
{
for (int i=0; i<n; i++)
{
for (int j=0; j<m; j++)
{
cout<<" ", (*this)[i][j];
}
cout<<endl;
}
cout<<endl;
}

void Str::output()
{
for(int j = 0; j < m; j++) cout<<" ",q[j];
}

int main()
{
TMatrix c(3,3); Str s(3); c[1][2]=2.0;
c[2][1]=2.0;
cout<<"Матриця: "<<endl; c.output(); cout<<"Рядок: "<<endl;
s=c(1);
```

```
s.output();
return 0;
}
```

Зверніть увагу на те, що в класі Str операторна функція operator[]() повертає посилання на число типу double. Це дозволяє використовувати вираження s[j] у лівій частині оператора присвоювання.

Тема 6.2. Успадкування. Одинарне успадкування

Успадкування – один з трьох фундаментальних механізмів об'єктно-орієнтованого програмування, оскільки саме завдяки йому уможливлюється створення ієрархічних класифікацій. Використовуючи механізми успадкування, можна розробити загальний клас, який визначає характеристики, що є властиві множині взаємопов'язаним між собою елементам. Цей клас потім може успадковуватися іншими, вузько спеціалізованими класами з додаванням у кожен з них своїх, властивих тільки їм унікальних особливостей. У стандартній термінології мови програмування C++ початковий клас називається базовим. Клас, який успадковує базовий клас, називається похідним. Похідний клас можна використовувати як базовий для іншого похідного класу. За таким механізмом якраз і будується багаторівнева ієрархія класів.

В основі механізму, що дозволяє створювати ієрархії класів, лежить принцип успадкування. Клас, що лежить в основі ієрархії, називається базовим. Класи, що успадковують властивості базового класу, називаються похідними. Похідні класи, у свою чергу, можуть бути базовими стосовно своїх спадкоємців, що в результаті призводить до ланцюжка успадкування. Процес утворення похідного класу на основі базового називається виводом класу. З одного базового класу можна вивести декілька похідних. Крім того, похідний клас може бути спадкоємцем декількох базових класів. Успадкування буває одинарним і множинним. При одинарному успадкуванні в кожного похідного класу є лише один базовий клас, а при множинному — декілька. Розглянемо механізм успадкування ближче.

Мова програмування C++ підтримує механізм успадкування, який дає змогу в оголошенні класу вбудовувати інший клас. Для цього базовий клас задається під час оголошення похідного класу. Щоб зрозуміти сказане, почнемо з конкретного прикладу. Розглянемо клас vehicle, який загалом визначає дорожній транспортний засіб. Його члени даних дають змогу зберігати наявну кількість коліс і можливу кількість пасажирів, яких може перевозити транспортний засіб:

```
class vehicle {
int wheel; // кількість коліс
int passenger; // кількість пасажирів
public:
void setWheel(int f) { wheel = f; }
int getWheel() { return wheel; }
void setPassenger(int t) { passenger = t; }
int getPassenger() { return passenger; }
};
```

Таке загальне визначення дорожнього транспортного засобу є частиною визначення будь-якого конкретного типу автотранспорту. Наприклад, у наведеному нижче оголошенні класу шляхом успадкування класу vehicle створюється клас truck – вантажних автомобілів:

```
class truck : public vehicle{
float loadCapacity; // вантажомісткість у кубічних метрах
public:
void setCapacity(int h) { mistkist = h; }
int getCapacity() { return mistkist; }
}
```

Той факт, що клас truck успадковує клас vehicle, означає, що клас truck успадковує весь вміст класу vehicle. До вмісту класу vehicle клас truck додає свого члена даних loadCapacity, а також функції-члени, необхідні для його підтримки. Зверніть увагу на те, як успадковується клас vehicle. Загальний формат для забезпечення механізму успадкування має такий вигляд:

```
class ім'я_похідного_класу: доступ ім'я_базового_класу{
тіло нового класу
}
```

У такому оголошенні похідного класу елемент доступ є необов’язковим. У разі потреби він може бути виражений одним із специфікаторів доступу: public, private або protected. Поки у визначеннях усіх успадкованих класів будемо використовувати специфікатор доступу public. Це означає, що всі public-члени базового класу також будуть public-членами похідного класу. Отже, у наведеному вище прикладі члени класу truck мають доступ до відкритих функцій-членів класу vehicle, неначе вони (ці функції) були оголошені в тілі класу truck. Проте клас truck не має доступу до private-членів класу vehicle. Наприклад, для класу truck закритий доступ до членів даних wheel і passenger. Розглянемо код програми, яка демонструє механізм успадкування двох підкласів класу vehicle: truch і car.

```
// Оголошення базового класу транспортних засобів
class vehicle {
int wheel; // кількість коліс
int passenger; // кількість пасажирів
public:
void setWheel(int f) { wheel = f; }
int getWheel() { return wheel; }
void setPassenger(int t) { passenger = t; } int
getPassenger() { return passenger; }
};
// Оголошення похідного класу вантажівок.
class truck : public vehicle{
float loadCapacity; // вантажомісткість у кубічних метрах
public:
void setCapacity(int h) { mistkist = h; }
int getCapacity() { return mistkist; } void Show() ;
}
enum type {car, van, wagon}; // перерахунковий тип даних
// оголошення похідного класу автомобілів.
class car : public vehicle {
enum type carType;
public:
void setType(type t) { carType = t; }
enum type getType() { return carType; }
void Show();
};
void truck::Show() {    // метод виведення інформації про
об'єкт класу truck на екран
cout << "Транспортний засіб: " << endl;
cout << "коліс: " << getWheel() << " шт" << endl;
cout << "пасажирів: " << getPassenger() << " осіб" << endl;
cout << "вантажомісткість: " << loadCapacity << " мкуб" <<
endl; } void car::Show() { // метод виведення інформації про
об'єкт класу car на екран
cout << "Транспортний засіб: " << endl;
cout << "коліс: " << getWheel() << " шт" << endl;
cout << "пасажирів: " << getPassenger() << " осіб" << endl;
```

```
cout << "тип: ";
switch(getType()) {
case van: cout << "van" << endl; break; case car: cout <<
"car" << endl; break; case wagon: cout << "wagon" << endl; }
}


int main()
{ truck ObjT, ObjF; car ObjG;
// ініціалізація об'єкта типу truck ObjT.setWheel(18);
ObjT.setPassenger(2); ObjT.setCapacity(160);
// Ініціалізація об'єкта типу truck ObjT.setWheel(6);
ObjT.setPassenger(3); ObjT.setCapacity(80);
// Виведення інформації про об'єкт типу truck
ObjT.Show();
ObjF.Show();
// ініціалізація об'єкта типу car ObjG.setWheel4);
ObjG.setPassenger(6); ObjG.setType(van);
// Виведення інформації про об'єкт типу car
ObjG.Show(; return 0;
}
```

Розглянемо, як відбувається управління механізмом доступу до членів базового класу. Якщо один клас успадковує інший, то члени базового класу стають членами похідного. Статус доступу до членів базового класу у похідному класі визначається специфікатором доступу, який використовують для успадкування базового класу. Специфікатор доступу до членів базового класу виражається одним з ключових слів: public, private або protected. Якщо специфікатор доступу не вказано, то за замовчуванням використовується специфікатор private, коли йдеться про успадкування типу class. Якщо базовий клас успадковується, як public-клас, то всі його public-члени стають public-членами похідного класу.

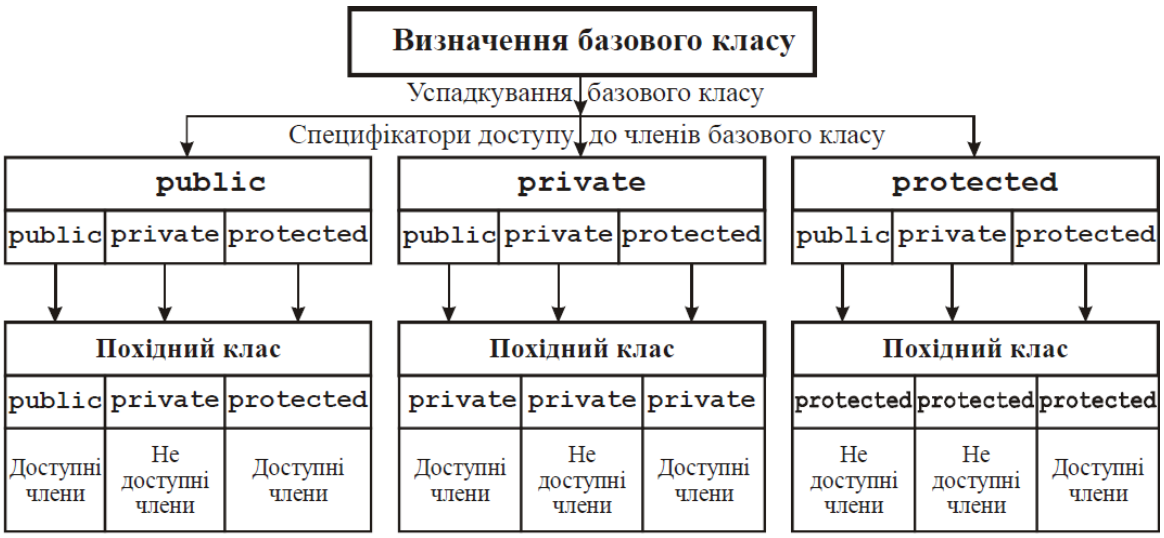


Рисунок 2 – Успадкування специфікаторів доступу

В усіх випадках private-члени базового класу залишаються закритими у межах цього класу і недоступні для членів похідного. Наприклад, у наведеному нижче коді програми public-члени класу baseClass стають public-членами класу derived. Отже, вони будуть доступними і для інших частин програми.

Продемонструємо розмежування доступу на такому навчальному прикладі.

```
class baseClass { // оголошення базового класу
int c, d; public:
void setB(int a, int b) { c = a; d = b; }
void showB() { cout << "c= " << c << "; d= " << d << endl;
}
};
class derived : public baseClass { // оголошення похідного
класу
int f;
public: derived(int x) { f = x; }
void showF() {
showB();
cout << "f= " << f << endl; }
};
int main() {
derived ObjD(3); // доступ до членів класу baseClass
ObjD.setB(1, 2); // доступ до членів класу baseClass
ObjD.showB();
cout << endl;
```

```
// доступ до члена класу derived
ObjD.showF(); return 0;
}
```

Результатом роботи такоє програми є наступна інформація виведена на еран:

```
c = 1; d = 2 // виведено дані базового класу
c = 1; d = 2 // виведено дані успадкованого класу
f = 3
```

ЗМІСТОВИЙ МОДУЛЬ 7
МНОЖИННЕ УСПАДКУВАННЯ ТА МОДУЛЬНА ОРГАНІЗАЦІЯ КОДУ

Тема 7.1. Множинне успадкування. Механізми успадкування декількох базових класів

Як уже було сказано вище, успадкування буває одинарним і множинним. При одинарному успадкуванні в кожного похідного класу є лише один базовий клас, а при множинному – декілька. Розглянемо механізм множинного успадкування ближче. Похідний клас може успадковувати два або більше базових класів. Наприклад, у навчальному прикладі клас derivedClass успадковує обидва класи base1 і base2.

```
class base1 { // оголошення базового класу
protected:
int x;
public:
void showX() { cout << x << endl; }
};
class base2 { // оголошення базового класу
protected:
int y;
public:
void showY() { cout << y << endl; }
};
// Успадкування двох базових класів.
// Оголошення похідного класу
class derivedClass : public base1, public base1 { public:
void setXY(int c, int d) {x = c; y = d; }
};
int main() {
derived ObjD; // створення об'єкта класу
ObjD.setXY(10, 20);     // член класу
derived ObjD.showX();   // функція з класу base1
 ObjD.showY();// функція з класу base2 return 0;
}
```

Як видно з цього коду програми, щоб забезпечити успадкування декількох базових класів, необхідно через кому перерахувати їх імена у вигляді списку. При цьому потрібно вказати специфікатор доступу для кожного

успадкованого базового класу.

З’ясуємо, у чому полягають собливості використання конструкторів і деструкторів при реалізації механізму успадкування. Базовий і/або похідний клас може містити конструктор і/або деструктор. Важливо розуміти послідовність, у якій виконуються ці функції при створенні об’єкта похідного класу і його (об’єкта) руйнування. Для розуміння сказаного розглянемо такий приклад.

```
class basedClass { // Оголошення базового класу
public:
basedClass() { cout << "Creating basedClass-object" <<
endl; }
};
~basedClass() { cout << "Destructing basedClass-object" <<
// Оголошення похідного класу
class derivedClass : public basedClass {
public:
derivedClass() {
cout << "Creating derivedClass-object" <<
endl; }
};

~derived() { cout << "Destructing derivedClass-object" <<
endl; }
int main() {
derivedClass ObjD; // створення об'єкта класу
// Ніяких дій, окрім створення і руйнування об'єкта ObjD.
return 0;
}
```

Як зазначено у коментарях для функції main(), ця програма тільки створює і відразу руйнує об’єктObjD, який має тип derived. Внаслідок виконання ця програма відображає на екрані такі результати:

```
Creating basedClass-object
Creating derivedClass-object
Destructing derivedClass-object
Destructing basedClass-object
```

Аналізуючи отримані результати, бачимо, що спочатку виконується конструктор класу basedClass, а за ним – конструктор класу derivedClass. Потім (через руйнування об’єкта ObjD у цьому коді програми) викликається

деструктор класу derivedClass, а за ним – деструктор класу basedClass. Результати описаного вище експерименту можна узагальнити. При створенні об’єкта похідного класу спочатку викликається конструктор базового класу, а за ним – конструктор похідного класу. Під час руйнування об’єкта похідного класу спочатку викликається його "рідний" конструктор, а за ним – конструктор базовогокласу.

Конструктори викликаються у порядку походження класів, а деструктори - у зворотньому.

Цілком логічно, що функції конструкторів виконуються у порядку походження їх класів. Оскільки базовий клас "нічого не знає" ні про який похідний клас, операції з ініціалізації, які йому потрібно виконати, не залежать від операцій ініціалізації, що виконуються похідним класом, але, можливо, створюють попередні умови для подальшої роботи. Тому конструктор базового класу повинен виконуватися першим. Така ж логіка простежується і у тому, що деструктори виконуються у порядку, зворотному порядку походження класів. Оскільки базовий клас знаходиться в основі похідного класу, то руйнування першого передбачає руйнування другого. Отже, деструктор похідного класу є сенс викликати до того, як об’єкт буде повністю зруйнований. При розширеній ієрархії класів (тобто за ситуації, коли похідний клас стає базовим класом для ще одного похідного) застосовується таке загальне правило: конструктори викликаються у порядку походження класів, а деструктори – у зворотному порядку. Для розуміння сказаного розглянемо такий приклад.

```
class basedClass { // оголошення базового класу
public:
baseClass() { cout << "Creating basedClass-object" <<
endl; }
};

~baseClass() { cout << "Destructing baseddClass-object"
<<endl; }
// Оголошення похідного класу
class derived1 : public baseClass { public:
derived1() { cout << "Creating derived1-object" << endl; }
~derived1() { cout << "Destructing derived1-object" <<
endl; }
};
// Оголошення похідного класу
class derived2 : public derived1 { public:
derived2() { cout << "Creating derived2-object" << endl; }
~derived2() { cout << "Destructing derived2-object" <<
```

```
endl; }
};
int main() {
derived2 Obj; // створення об'єкта класу
// створення і руйнуванняоб'єкта Obj
return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Creating basedClass-object
Creating derived1-object
Creating derived2-object
Destructing derived2-object
Destructing derived1-object
Destructing basedClass-object.
```

Те саме загальне правило застосовується і у ситуаціях, коли похідний клас успадковує декілька базових класів.

Продемонструємо послідовності виконання конструкторів і деструкторів під час успадкування декількох базових класів

```
class basedClass1 { // оголошення базового класу1
public:
baseA() { cout << "Creating basedClass1-object" << endl; }
~baseA() { cout << "Destructing basedClass1-object" <<
endl; }
};
class basedClass2 { // оголошення базового класу2
public:
baseA() { cout << "Creating basedClass2-object" << endl; }
~baseA() { cout << "Destructing basedClass2-object" <<
endl; }
};
// оголошення похідного класу
class derivedClass : public basedClass1, public basedClass2 {
public:
baseA() { cout << "Creating derivedClass-object" <<endl; }
~baseA() { cout << "Destructing derivedClass-object" <<endl;
}};
```

```
int main() {
derivedClass Obj; // створення об'єкта похідного класу
// створення і руйнування об'єкта Obj
return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Creating basedClass1-object
Creating basedClass2-object
Creating derivedClass-object
Destructing derivedClass-object
Destructing basedClass2-object
Destructing basedClass1-object
```

Як бачите, конструктори викликаються у порядку походження їх класів, зліва на право, у порядку їх задавання у переліку успадкування для класу derivedClass. Деструктори викликаються у зворотному порядку, зправа на ліво. Це означає, що якби клас base2 знаходився передкласом base1 у переліку класу derivedClass, тобто відповідно до такої настанови: class derivedClass : public base2, public base1, то результати виконання попереднього коду програми були б такими:

```
Creating basedClass2-object
Creating basedClass1-object
Creating derivedClass-object
Destructing derivedClass-object
Destructing basedClass1object
Destructing basedClass2-object
```

До тепер жоден з попередніх прикладів не містив конструкторів, для яких потрібно було б передавати аргументи. У випадках, коли конструктор тільки похідного класу вимагає передачі одного або декількох аргументів, достатньо використовувати стандартний синтаксис параметризованого конструктора. Але як передати аргументи конструктору базового класу? У цьому випадку необхідно використовувати розширену форму оголошення конструктора похідного класу, у якому передбачено можливість передачі аргументів одному або декільком конструкторам базового класу.

Загальний формат розширеного оголошення конструктора похідного класу має такий вигляд:

```
конструктор_похідного_класу(перелік_аргументів):
base1(перелік_аргументів),
base2(перелік_аргументів),.........
```

```
    baseN(перелік_аргументів); {
тіло конструктора похідного класу}
```

Тут елементи base1, ..., baseN означають імена базових класів, що успадковуються похідним класом. Зверніть увагу на те, що оголошення конструктора похідного класу відділяється від переліку базових класів двокрапкою, а імена базових класів розділяються між собою комами (у разі успадкування декількох базових класів). Для демонстрації опсаного принципу розгялнемо наступну програму.

```
class baseClass {  // оголошення базового класу
protected:
int c;
public:
baseClass(int x) { c = x; cout << "Creating basedClass-
object" << endl; }
~baseClass() { cout << "Destructing basedClass-object" <<
endl; }
};
// Оголошення похідного класу
class derivedClass : public baseClass { int d;
public: // Клас derivedClass використовує параметр x, а
параметр y передається конструктору класу baseClass.
derivedClass(int x, int y) : baseClass(y) { d = x;
cout << "Creating derivedClass-object" << endl;
endl;
}
~derived() { cout << "Destructing derivedClass-object" <<

}
void show(string s) {
cout << s << "c= " << c << "; d= " << d << endl;
}
};
int main() { derivedClass Obj(12, -1);
Obj.show("Base Class: "); // відображає числа 12 -1
return 0;
}
```

У цьому коді програми конструктор класу derived оголошується з двома

параметрами x і y. Проте конструктор derivedClass() використовує тільки параметр x, а параметр y передається конструктору baseClass(). У загальному випадку конструктор похідного класу повинен оголошувати параметри, які приймає його клас, а також ті, які потрібні базовому класу. Як це показано у наведеному вище прикладі, будь-які параметри, що потрібні базовому класу, передаються йому у переліку аргументів базового класу, які вказуються після двокрапки. Розглянемо приклад програмного коду, де продемонстровано механізм передачі параметрів конструкторам декількох базових класів.

```
class baseClass1 { // оголошення базового класу
protected:
int c;
public:
baseClass1(int x) { c = x;
cout << "Creating baseClass1-object" << endl;
}
~baseClass1() {
cout << "Destructing baseClass1-object" << endl;
}
};
class baseClass2 { // оголошення базового класу
protected:
int c;
public:
baseClass2(int x) { c = x;
cout << "Creating baseClass2-object" << endl;
}
~baseClass1() {
cout << "Destructing baseClass2-object" << endl;
}
};
// Оголошення похідного класу
class derivedClass : public baseClass1, public baseClass2 {
int d;
public:
derivedClass (int x, int y, int z): baseA(y), baseB(z) { d =
x;
cout << "Creating derivedClass-object" << endl;
}
```

```
~derived() {
cout << "Destrucring derivedClass-object" << endl;
endl;
};
}
void show(string s) {
cout << s << "c= " << c << "; d= " << d << "; f= " << f <<

}
int main() {
derived Obj(8, 9, 10);
Obj.show("Based class"); // Відображає числа c= 8; d= 9; f=
10
return 0;
}
```

Важливо розуміти, що аргументи для конструктора базового класу передаються через аргументи, які приймаються конструктором похідного класу. Навіть якщо конструктор похідного класу не використовує ніяких аргументів, то він повинен оголосити один або декілька аргументів, якщо базовий клас приймає один або декілька аргументів. У цій ситуації аргументи, що передаються похідному класу, "транзитом" передаються базовому. Наприклад, у наведеному нижче коді конструктори baseClass1() і baseClass2(), на відміну від конструктора класу derivedClass, приймають аргументи.

```
class baseClass1 { // оголошення базового класу
protected:
int c; public:
baseClass1(int x) { c = x;
cout << "Creating baseClass1-object" << endl;
}
~baseClass1() {
cout << "Destructing baseClass1-object" << endl;
}
};
class baseClass2 { // оголошення базового класу
protected:
int c; public:
baseClass2(int x) { c = x;
```

```
cout << "Creating baseClass2-object" << endl;
}
~baseClass1() {
cout << "Destructing baseClass2-object" << endl;
}
};
// Оголошення похідного класу
class derivedClass : public baseClass1, public baseClass2 {
public:  /* конструктор класу derived не використовує
параметрів, але повинен оголосити їх, щоб передати
конструкторам базових класів. */
derivedClass( int x, int y) : baseClass1(x), baseClass2(y) {
cout << "Creating derivedClass-object" << endl;
}
endl;
~derived() {
cout << "Destructing derivedClass-object " <<
}
void show(string s) { cout << s << "c =" << c << "; f= " << f
<< endl; }
};
int main() {
derived Obj (5, -17);
ObjD.showB("Base class: "); // відображає числа c= 5; f= -17
return 0;
}
```

Конструктор похідного класу може використовувати будь-які (або всі) параметри, які ним оголошені для прийняття, незалежно від того, чи передаються вони (один або декілька) базовому класу. Іншими словами, той факт, що деякий аргумент передається базовому класу, не заважає його використанню і самим похідним класом.

Наприклад, наведений нижче фрагмент коду програми є абсолютно допустимим:

```
class derivedClass : public baseClass {
int d;
public: /* Клас derived використовує обидва параметри x і y,
а також передає їх класу baseClass */
derivedClass(int x, int y) : baseClass(x, y) { d = x*y;
```

```
cout << "Creating derivedClass-object" << endl; }
//. . .
```

При передачі аргументів конструкторам базового класу необхідно мати на увазі, що аргумент, який передається, може містити будь-який (дійсний на момент передачі) вираз, що містить виклики функцій і змінних. Це можливо завдяки тому, що мова C++ дає змогу виконувати динамічну ініціалізацію даних.

Тема 7.2. Оголошення класів у заголовочних файлах

Усі класи, які ми використовували до цього часу, були досить простими, тому ми описували методи безпосередньо всередині тіла класів. Наприклад:

```
class Date{   // клас, що описує дату
private:
int m_day;    // день
int m_month;  // місяць
int m_year;   // рік
public:
Date(int day, int month, int year){  // конструктор з
параметром
m_day = day; m_month = month; m_year = year;
}
void setDate(int day, int month, int year){ // set()-метод
m_day = day; m_month = month; m_year = year;
}
int getDay() { return m_day; }   // get()-методи
int getMonth() { return m_month; } int getYear(){ return
m_year; }
};
```

Однак, як тільки класи стають більшими і складнішими, наявність всіх методів всередині тіла класу може утруднити його управління і роботу з ним. Використання вже написаного класу вимагає розуміння тільки його відкритого інтерфейсу, а не того, як він реалізований. На допомогу у цьому випадку приходить можливість відокремлення оголошення від реалізації. C ++ надає спосіб відокремити «оголошення» від «реалізації». Це робиться шляхом визначення методів поза тілом самого класу. Для цього просто визначають методи класу, ніби це звичайні функції, але в якості префіксу додають до імені функції ім’я класу з оператором дозволу області видимості :: (того, що використовується з просторами імен). Ось наш клас Date з конструктором

Date() і методом setDate (), оголошеними поза тілом класу. Зверніть увагу, прототипи цих функцій все ще знаходяться всередині тіла класу, але їх фактична реалізація перебуває за його межами:

```
class Date{
private:
int m_day;
int m_month;
int m_year;
public:
Date(int day, int month, int year);
void SetDate(int day, int month, int year);
int getDay() { return m_day; }
int getMonth() { return m_month; }
int getYear() { return m_year; }
};
// Конструктор класу Date
Date::Date(int day, int month, int year)
{
SetDate(day, month, year);
}
// Метод класу Date
void Date::SetDate(int day, int month, int year)
{
m_day = day; m_month = month;
m_year = year;
}
```

Просто, чи не так? Оскільки в багатьох випадках функції доступу можуть складатися всього з одного рядка, то їх зазвичай залишають в тілі класу, хоча перемістити їх за межі класу можна завжди. Ось ще один приклад класу з конструктором, оголошеним ззовні, та списком параметрів ініціалізації.

```
class Mathem{
private:
int m_value = 0; public:
Mathem(int value=0): m_value(value) {}
Mathem& add(int value) { m_value += value; return *this; }
Mathem& sub(int value) { m_value -= value; return *this; }
 Mathem& divide(int value) { m_value /= value; return *this;
```

```
}
int getValue() { return m_value ; }
};
```
Конвертуємо у наступний код:
```
class Mathem{
private:
int m_value = 0; public:
Mathem(int value=0); Mathem& add(int value); Mathem& sub(int
value); Mathem& divide(int value);
int getValue() { return m_value; }
};
Mathem::Mathem(int value): m_value(value){
}
Mathem& Mathem::add(int value)
{
m_value += value; return *this;
}
Mathem& Mathem::sub(int value)
{
m_value -= value; return *this;
}
Mathem& Mathem::divide(int value){ m_value /= value;
return *this;}
```

Оголошення функцій можна помістити у заголовочні файли, щоб потім мати можливість використовувати ці функції в декількох файлах або навіть в декількох проєктах. Класи в цьому плані нічим не відрізняються від функцій. Визначення класів можуть бути поміщені в заголовки для полегшення їх повторного використання в декількох файлах або проектах. Зазвичай, визначення класу поміщається в заголовки з тим же ім'ям, що й у класу, а методи, оголошені поза тілом класу, поміщаються в файл .cpp з тим же ім'ям, що й у класу. Ось наш Date, але вже розбитий на файли .cpp і .h:

Date.h:

```
#ifndef DATE_H
#define DATE_H

class Date {
private:
```

```
    int m_day;
    int m_month;
    int m_year;

public:
    Date(int day, int month, int year);

    void SetDate(int day, int month, int year);

    int getDay() const { return m_day; }
    int getMonth() const { return m_month; }
    int getYear() const { return m_year; }
};

#endif
```

Date.cpp:

```
#include "Date.h"

Date::Date(int day, int month, int year) {
    SetDate(day, month, year);
}

void Date::SetDate(int day, int month, int year) {
    m_day = day;
    m_month = month;
    m_year = year;
}
```

Тепер у будь-який інший файл .h або .cpp, який захоче використовувати клас Date, можна просто додати #include "Date.h". Зверніть увагу, Date.cpp також необхідно додати до компіляції в проєкт, який використовує Date.h, щоб лінкер зміг розібратися з реалізацією класу Date.

При використанні такого типу оголошень класу можуть виникнути наступні питання:

Питання №1: Хіба визначення класу в заголовному файлі не порушує правило одного визначення?

Ні. Класи - це призначені для користувача типи даних, які звільняються від визначення тільки в одному місці. Тому клас, оголошений у заголовочному файлі, можна вільно підключати до інших файлів.

Питання №2: Хіба визначення методів класу в заголовному файлі не порушує правило одного визначення?

Методи, означені всередині тіла класу, вважаються неявно вбудованими. Вбудовані функції звільняються від правила одного визначення. А це означає, що проблем з визначенням простих методів (таких як функції доступу) всередині самого класу виникати неповинно. Методи,оголошені поза тілом класу, розглядаються як звичайні функції і підпорядковуються правилу одного визначення. Тому ці функції повинні бути визначені в файлі .cpp, а не всередині .h. Параметри за замовчуванням для методів повинні бути оголошені в тілі класу (в заголовках), де вони будуть видні всім, хто вказав #include для заголовка з класом.

Поділ оголошення класу і його реалізації дуже подібне до використання бібліотек, які використовуються для розширення можливостей вашої програми. Ви також підключали заголовки зі стандартної бібліотеки, такі як iostream, string, vector, array і інші. Зверніть увагу, ви не додавали iostream.cpp, string.cpp, vector.cpp або array.cpp в ваші проекти. Ваша програма потребує тільки в оголошеннях з заголовних файлів, щоб компілятор міг перевірити коректність вашого коду відповідно до правил синтаксису. Однак, реалізації класів, які перебувають в стандартній бібліотеці, містяться в попередньо скомпільованому файлі, який додається на етапі лінкінгу. Ви ніколи не маєте доступу до цього коду. Поза програмою з відкритим вихідним кодом (де надаються обидва файли: .h і .cpp), більшість сторонніх бібліотек надають тільки заголовки разом з попередньо скомпільований файлом бібліотеки. На це є кілька причин:

- На етапі лінкінгу швидше буде підключити попередньо компільована бібліотеку, ніж виконувати перекомпіляцію кожного разу, коли вона потрібна.
- Захист інтелектуальної власності (розробники не хочуть, аби хтось брав їхній код).
- Наявність власних файлів, розділених на оголошення (файли .h) та реалізацію (файли .cpp) є не тільки хорошою формою змісту коду, але і спрощує створення власних користувацьких бібліотек.
- Можливо, у вас виникне бажання помістити всі визначення методів класу в заголовки всередині тіла класу. Хоча це компілюється, але тут є кілька нюансів:
- По-перше, як згадувалося вище, це призведе до захаращення визначення
- вашого класу.
- По-друге, функції, певні всередині класу, є неявно вбудованими. Великі функції, які викликаються з багатьох файлів, можуть сприяти, таким чином, до «роздування» вашого коду.
- По-третє, якщо ви зміните щось у заголовковому файлі, то слід перекомпілювати кожен файл, який містить цей заголовок. Це може мати "ефект метелика", коли одна незначна зміна призводить до перекомпілювання всього проекту, що може бути достатньо довго та повільно. Якщо був змінений тільки файл .cpp, то слід перекомпілювати

лише його!

Тому рекомендації до використання заголовочних файлів наступні:

- Класи, призначені для повторного використання, оголошуються у окремому файлі з тим же іменем, що й ім’я класу.
- Тривіальні методи (які будуть повторно використані), оголошуються в класі, у файлі .h.
- Нетривіальні методи оголошуються у файлі .cpp

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Основна література

1. Гришанович Т. О., Глинчук Л. Я. Основи об’єктно-орієнтованого програмування: навчальний посібник. Луцьк: ВНУ імені Лесі Українки, 2022. 120 с.
2. Васильєв О. М. Програмування мовою Java. Тернопіль : Богдан НК, 2022. 696 с.
3. Висоцька В. А., Оборська О. В. Python: алгоритмізація та програмування : навчальний посібник. Львів : Новий світ-2000, 2021. 514 с.
4. Григорович В. Г. Алгоритмізація та програмування. Частина 1 : навчальний посібник. Львів : Магнолія 2006, 2023. 357 с.
5. Злобін Г. Г. Основи алгоритмізації та програмування мовою Сі : підручник. Київ : Каравела, 2022. 168 с.
6. Гришанович Т. О., Глинчук Л. Я. Основи об’єктно-орієнтованого програмування : навч. посібник. Луцьк : ВНУ імені Лесі Українки, 2022. 120 с. URL: https://evnuir.vnu.edu.ua/handle/123456789/20320
7. Муляр В. П. Об’єктно-орієнтоване програмування: лабораторний практикум. Луцьк : Вежа-Друк, 2022. 112 с. URL: https://evnuir.vnu.edu.ua/handle/123456789/21289
8. Порєв В. М. Об'єктно-орієнтоване програмування: конспект лекцій : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2022. 271 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/854443cb-3375-4e76-961a-1f9a74a15b07/content
9. Проектування інформаційних систем: Загальні питання теорії проектування ІС : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2020. 192 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/c136860d-44cb-4f05-adaf-dcdd20830483/content
10. Решевська К. С., Лісняк А. О., Борю С. Ю. Об’єктно-орієнтоване програмування : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Комп’ютерні науки» освітньо-професійної програми «Комп’ютерні науки». Запоріжжя : ЗНУ, 2020. 94 с. URL: https://dspace.znu.edu.ua/jspui/handle/12345/3139
11. Григорович В. Г. Об'єктно-орієнтоване програмування. Частина 1 : навчальний посібник. Львів : Магнолія-2006, 2023. 284 с.
12. Григорович В. Г. Об’єктно-орієнтоване програмування. Частина 2: Винятки. Контейнери та шаблони. STL – стандартна бібліотека шаблонів. S.O.L.I.D. : навч. посібник. Львів : Магнолія 2006, 2024. 343 с.

Додаткова література

1. Махровська Н. А., Погромська Г. С., Булгакова О. С., Зосімов В. В. Програмування: мова програмування С++ : навч.-метод. посіб. Миколаїв, 2017. 273 с.
2. Замуруєва О. В., Кримусь А. С., Ольхова Н. В. Об’єктно-орієнтоване

програмування в Python : курс лекцій. Луцьк : Вежа-Друк, 2018. 64 с. URL: https://evnuir.vnu.edu.ua/bitstream/123456789/14344/1/Python.pdf

3. Об’єктно-орієнтоване програмування. Лабораторний практикум : навчальний посібник / уклад. : Б. І. Бойко, Л. Л. Омельчук, Н. Г. Русіна. Київ : КНУ, 2016. 90 с. URL: http://csc.knu.ua/media/filer_public/4a/35/4a3533cd-4ec7-45f3-85d2-4edaafdf1b82/oop_2016.pdf

4. Настенко Д. В., Нестерко А. Б. Об’єктно-орієнтоване програмування. Частина 1. Основи об’єктно-орієнтованого програмування на мові C# : навчальний посібник. Київ : НТУУ «КПІ», 2016. 76 с. URL: https://ela.kpi.ua/handle/123456789/16671

Навчальне видання

Об'єктно-орієнтоване програмування

Конспект лекцій

Укладачі: **Ємельянов** Святослав Ігорович
Тищенко Світлана Іванівна
Пархоменко Олександр Юрійович
Жебко Олександр Олегович
Богатєнкова Олександра Євгенівна

Формат 60x84 1/16. Ум. друк. арк. 4,0.
Тираж 20 прим. Зам. № ____

Надруковано у видавничому відділі
Миколаївського національного аграрного університету
54008, м. Миколаїв, вул. Георгія Гонгадзе, 9

Свідоцтво суб'єкта видавничої справи ДК № 4490 від 20.02.2013 р.

200