Міністерство освіти і науки України
Рівненський державний гуманітарний університет
Кафедра інформаційних технологій
та моделювання

Степанія Михайлівна Бабич
Ігор Станіславович Войтович
Наталія Вікторівна Шевцова
Шліхта Ганна Олександрівна
Кирик Тетяна Анатоліївна

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ С++

Навчальний посібник

Рівне - 2025

УДК 004.43(075.8)
О-75

Рекомендовано до друку Вченою радою
Рівненського державного гуманітарного університету,
протокол № 7 від 26 грудня 2025 р.

Автори:

- *Бабич С. М.*, кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання РДГУ
- *Войтович І. С.*, доктор педагогічних наук, професор, проректор з навчально-виховної роботи РДГУ
- *Шевцова Н.В.*, кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання РДГУ
- *Шліхта Г.О.*, доктор педагогічних наук, професор кафедри інформаційних технологій та моделювання РДГУ
- *Кирик Т.А.,* старший викладач кафедри інформаційних технологій та моделювання РДГУ

Рецензенти:

- *Сяський В.А.*, кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету;
- *Турбал Ю.В.*, доктор технічних наук, професор, завідувач кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування.

С. М. Бабич, І. С. Войтович, Н. В. Шевцова, Г. О. Шліхта., Т.А. Кирик. Основи програмування мовою С++ : навч. посіб. [Електронний ресурс]/ автори. С.М. Бабич [та ін.]. — 2-ге вид., переробл. і доповн. - Рівне : РДГУ, 2025. — 291 с.

Призначається для студентів ІТ-спеціальностей та інших, а також для самоосвіти.

ЗМІСТ

ПЕРЕДМОВА

Мова С++ серед сучасних мов програмування є однією із найбільш поширених (входить в ТОП-10 мов програмування). Побудована на фундаменті та синтаксисі мови С (розробник – американський комп'ютерний спеціаліст Денніс Рітчі, 1972 р.). Мова С – універсальна, але найбільш ефективне її використання в задачах системного програмування розробки трансляторів, операційних систем, інструментальних засобів.

Мову С++ розробив данський програміст Б'ярн Страуструп на початку 80-х років. Спочатку вона називалася «Сі з класами», а в 1983 році переіменована на С++.

Мову С++ можна назвати розширеною та поліпшеною версією мови С, у якій реалізовано технологію об'єктно-орієнтованого, узагальненого та процедурного програмування. Вона також містить ряд інших удосконалень мови С, наприклад, розширений набір бібліотечних функцій.

Мова С++ добре зарекомендувала себе ефективністю, лаконічністю запису алгоритмів, логічною стрункістю програм. У багатьох випадках програми, написані на ній, можна порівняти по швидкості з програмами, написаними на мові

Асемблера, при цьому вони більш наглядні і прості у супроводі.

Основними перевагами мови С++ вважаться висока передача написаних на ній програм між комп'ютерами з різною архітектурою, між різними операційними середовищами. Транслятори мови С++ працюють практично на всіх персональних комп'ютерах, які використовуються в наш час.

Навчальний посібник «Інформатика: основи програмування в С++» висвітлює основні прийоми програмування мовою С++ і призначений головним чином для того, щоб допомогти майбутнім фахівцям вивчити мову програмування і застосовувати її при розробці власних проєктів. У посібнику дано ґрунтовний опис мови С++, багато прикладів і фрагментів програм. Основна увага приділена основам роботи у візуальному середовищі програмування Visual Studio С++.

Даний посібник може бути використаний при вивченні дисциплін «Інформатика», «Основи алгоритмізації та програмування» студентами освітніх і ІТ спеціальностей, а також для самоосвіти.

У якості середовища розробки використано Visual Studio 2019 Preview.

РОЗДІЛ 1. БАЗОВІ ЕЛЕМЕНТИ МОВИ

1.1. Алфавіт

Алфавіт мови визначає набір символів, які можуть використовуватись для формування лексичних елементів програм.

Всі символи алфавіту C++ поділяють на три групи. До першої групи входять символи ключових слів та ідентифікаторів, до яких належать:

- великі та малі літери латинської абетки: A..Z, a..z (всього 52 символи);
- цифри: 0..9 (всього 10 символів);
- знак підкреслення: _ .

До другої групи входять 28 символів, що використовуються як знаки операцій, символи пунктуації та роздільники:

```
    +   -
  *   /   %   =   <   >   &   |   !   ~   ^   ?   ,
.   ;   :   ‘   “
    (   )   [   ]   {   }   #   \ .
```

Третя група символів C++ – це т.зв. символи, кожен з яких має встановлений внутрішній код, як і символи першої та другої груп, але не має власного графічного позначення. До цієї групи належать символ пробілу та спеціальні керуючі символи, які ще називають ескейп-послідовностями: символи

табуляції, нового рядка, символи нової сторінки тощо. Пробільні неграфічні символи використовують у С-програмах для відокремлення лексем.

Спеціальні керуючі символи. Ці символи називають *керуючими послідовностями* або *ескейп-послідовностями (Esc-послідовностями)*. Вони позначають неграфічні символи, призначені для керування формою виведення даних і повідомлень: відтворення короткого звукового сигналу, повернення екранного курсора на одну позицію назад, переведення курсора на початок рядка тощо (табл. 1.1).

Ескейп-послідовності у С++-програмах можна записувати трьома способами, використовуючи символьну форму або вісімкове чи шістнадцяткове позначення. У всіх трьох варіантах запису першим символом послідовності є зворотна коса риска (її називають лівим слешем). У символьній формі ескейп-послідовності за слешем записується для даного символа літера, наприклад, \t позначає перехід до горизонтальної табуляційної позиції.

Вісімкове позначення ескейп-послідовності складається зі слеша та однієї, двох або трьох вісімкових цифр (цифри від 0 до 7), що задають код даного символа. Зокрема, ескейп-символ \t, ASCII-код

якого дорівнює 9, можна записати у формі \011 або \11. Шістнадцяткова форма ескейп-послідовності починається символами \x або \X, за якими вказуються одна або дві шістнадцяткові цифри (цифри 0..9 та літери a..f або A..F). Шістнадцяткове позначення \t може бути таким: \x09 або таким: \X9.

Таблиця 1.1. *Символи керуючих послідовностей*

Слеш-символ	**Призначення символа**	**ASCII-код**	**Вісімкове позначення**	**Шістнадцяткове позначення**
\a	звуковий сигнал	7	\007	\x07
\b	повернення на	8	\010	\x08
\t	горизонтальна	9	\011	\x09
\n	перехід на новий	10	\012	\x0a
\v	вертикальна	11	\013	\x0b
\f	перехід до нової	12	\014	\x0c
\r	перехід на початок рядка ("повернення каретки")	13	\015	\x0d

Як ескейп-послідовності записують також вказані нижче символи абетки, коли вони використовуються як окремі символьні константи або як елементи символьних рядків:

\’ – позначення апострофа;

\” – позначення лапок;

\\ – позначення лівого слеша;

\? – позначення знака запитання.

Слід запам'ятати ці символи, бо відсутність лівого слеша в їх записах може викликати неправильну інтерпретацію компілятором символьних рядків, тим самим спричинити помилковість результатів роботи програми.

Якщо лівий слеш записати перед будь-яким іншим символом, що не входить у наведений вище перелік керуючих або спеціальних символів, то такий слеш ігнорується. Тобто, двосимвольна послідовність \m інтерпретується як m, а послідовність \# – як звичайний символ # тощо.

Через вісімкову та шістнадцяткову форму можна записати довільний символ ASCII-таблиці. Наприклад, ескейп-послідовність \0 позначає символ з кодом 0 – перший символ ASCII-таблиці. Цей символ використовують у символьних рядках як ознаку кінця рядка. Ескейп-послідовність \X1B (або рівнозначна вісімкова форма \033) відповідає символу ASCII-таблиці з кодом 27, що позначає клавішу *Esc*, а ескейп-послідовність \XBA (або \272) задає символ псевдографіки з кодом 186 – подвійну вертикальну лінію ‖.

1.2. Лексеми

Найменшими змістовними елементами програм, що мають самостійне призначення, є *лексичні одиниці* (*лексеми*). Лексеми мови С++ поділяють на шість груп: ключові слова, ідентифікатори, константи, символьні рядки, знаки операцій та роздільники.

Ключові слова. Ключові слова (табл. 1.2), їх теж називають *службовими* або *зарезервованими* словами, – це набір визначених слів, що використовуються для встановлення типів даних, формування операторів тощо. Кожне ключове слово має своє призначення, Застосовувати ключові слова для іншої мети (зокрема як імена змінних чи функцій) заборонено.

Таблиця 1.2. *Ключові слова мови програмування С++*

asm	auto	bool	break
case	catch	char	class
const	const_class	continue	default
delete	do	double	dinamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline

int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Звернемо увагу на те, що в ключових словах великі та малі літери вважаються різними. Тому в програмах ці слова треба записувати так, як вони вказані вище, наприклад, int, а не Int чи INT.

Ідентифікатори. Імена змінних, макросів, міток, функцій та інших об’єктів програми називають *ідентифікаторами*. Ідентифікатори формують із символів першої групи, тобто з малих і великих латинських літер, цифр, а також знака підкреслення.

Приклади коректних і помилкових ідентифікаторів:

- ✓ Suma a1 mitka_2 ZnachFunc new_array – правильні;
- ✓ 2rezult – неправильно, починається з цифри;
- ✓ res’05 – неправильно, містить символ

апострофа;

- ✓ Suma Mas – неправильно, містить символ пропуску.

Для наочності та зрозумілості програми важливу роль відіграє змістовність імен об'єктів. Наприклад, ідентифікатор змінної file_name відразу вказує, що ця змінна пов'язана з іменем файлу, а ім'я функції OpenNewWindow асоціюється з процедурою відкриття нового вікна. Такої наочності не буде, якщо для цих об’єктів використати короткі ідентифікатори, наприклад, fn та win.

Як і в ключових словах, в ідентифікаторах розрізняються великі та малі літери. Зокрема, три наступних ідентифікатори: prod, Prod та PrOd – будуть розглядатись компілятором як різні імена.

Прийнято, що імена внутрішніх об'єктів системи програмування, які використовуються в реалізаціях компіляторів і бібліотечних функцій, починаються одним або двома знаками підкреслення: _heaplen чи __FILE__. Тому не рекомендується починати зі знака підкреслення імена змінних та інших об’єктів програми, щоб випадково не спричинити конфлікту імен.

Константи. *Константи* (у літературі можна зустріти також термін літерали) – це об'єкти програм,

значення яких змінювати не можна. У мові С++ константи належать до арифметичних даних. Стандарт мови поділяє константи на цілочислові, дійсні, символьні та перелічувані.

Цілочислові константи використовуються для позначення цілих чисел зі знаком або без знака. Можуть мати три форми зображення:

- ✓ *десяткові* константи: 5246 200 -13 +45
- ✓ *вісімкові* константи: 034 01002 0515 0777
- ✓ *шістнадцяткові* константи: 0x9243 0XDA07 0xaЗ 0X10045F

Вісімкові константи завжди починаються цифрою 0, за якою записуються вісімкові цифри значення константи (цифри від 0 до 7). Шістнадцяткові константи починаються префіксом 0x або 0X (нуль і маленька чи велика літера x), за яким вказуються шістнадцяткові цифри числа (цифри 0.. 9 та літери a.. f або A..F).

Дійсні константи застосовують для позначення чисел, які мають як цілу, так і дробову частину (тобто дійсних чисел), а також великих цілих чисел. Дійсні константи можна записувати у двох формах:

- ✓ константи *з фіксованою крапкою*: 7.123 0.060 -384.65
- ✓ константи *з плаваючою крапкою*: 276.8e-5 10.02e+8 5157E10

Константи з фіксованою крапкою містять цілу і дробову частини числа, які відокремлюються між собою десятковою крапкою. У константах з плаваючою крапкою додатково вказується десятковий порядок числа, який записується після літери e або E (від слова *експонента*). Наприклад, константа 276.8e-5 відповідає дійсному числу $276.8 \cdot 10^{-5}$. Це число можна записати інакше: 2.768e-3 або 0.2768e-2, а також через константу з фіксованою крапкою 0.002768.

Символьні константи (їх ще називають *літерними*) використовуються у програмах для звертання до окремих символів. Символьна константа позначається одним символом, записаним у апострофах, або ескейп-послідовністю, теж записаною в апострофах. У символьних константах можна використовувати всі символи активної кодової таблиці, зокрема, літери кирилиці, якщо вони наявні в цій таблиці. Приклади символьних констант:

'A' 'a' '*' '5' 'Я' 'щ' '|' '\n' '\a' '\'' '\XB2'

Символьні рядки. Ще одним видом лексем мови С є символьні рядки (у літературі їх також називають стрінговими літералами або рядковими (стрінговими) константами. Символьний рядок – це послідовність довільних символів кодової таблиці (серед них можуть бути ескейп-послідовності),

охоплена лапками:

"Press any key…"

"Рівненський державний гуманітарний університет \"РДГУ\""

"\t Розв\'язок: \n"

Зверніть увагу на ескейп-послідовності, якими в другому і третьому рядках позначено внутрішні лапки, апостроф і керуючі символи.

В оперативній пам'яті всі символи рядка розташовуються підряд. Кожен символ, включаючи ескейп-послідовності, займає один байт, у якому записується код символа (зазначимо, що деякі системи використовують двобайтове кодування символів).

Особливість символьних рядків мови С в тому, що компілятор автоматично долучає до них кінцевий нуль-символ '\0', який використовується у процесах опрацювання рядків як ознака їхнього кінця. Два наступні записи: 'C' і "C" є різними не тільки синтаксично, а й семантично. Запис 'C' позначає символьну константу, яка зберігається у пам'яті як ціле число, значення якого дорівнює коду літери С. У той час як запис "C" є символьним рядком, що складається з двох символів: літери С та нуль-символа ('C'+'\0').

Довжина символьного рядка не обмежується. У разі довгих стрінгових констант часто виникає

потреба запису їх у декількох рядках програми, Використовують два способи поділу символьних рядків. Перший полягає в тому, що в місці розриву рядка записують лівий слеш, за яким ставлять символ нового рядка (натискують клавішу Enter). Тоді записані в наступному рядку символи вважаються продовженням стрінгової константи:

```
"Приклад довгого символьного рядка \
        з перенесенням"
```

Записаний вище рядок є рівнозначним до наступного:

```
"Приклад довгого символьного рядка      з
        перенесенням"
```

Недолік цього способу в тому, що в рядок результату потрапляють всі символи пробілу, записані перед символом слеша та па початку нового рядка.

У другому способі перенесення стрінгів використовується та властивість, що компілятор об'єднує (конкатенує) у спільний рядок два записані підряд стрінги – між ними може бути довільна кількість символів роздільників: пробілів, символів нового рядка, табуляції тощо. Тому довгий символьний рядок можна просто поділити на частини:

```
"Ще один приклад перенесення"
        " довгого символьного рядка"
```

Відповідний повний рядок буде таким:

"Ще один приклад перенесення довгого символьного рядка"

Знаки операцій, роздільники, коментарі. Описані вище лексеми: ключові слова, ідентифікатори та константи – це т.зв. *лексеми-слова*. У програмі між двома лексемами-словами обов'язково має бути записаний *знак операції* або *роздільник*: знак пунктуації чи пробільний символ.

Знаки операцій можуть позначатись одним символом або дво- чи багатосимвольною комбінацією зі символів другої групи алфавіту мови С++. Приклади знаків операцій:

+ * & < . ^ = – односимвольні операції;

++ II >> -> /= <<= – багатосимвольні операції,

До роздільників належать символи, які називають знаками пунктуації:

( ) [ ] { } , ; : = * #

Зокрема, фігурними дужками {} охоплюють тіло функції, знаком ; завершують усі описи та оператори, через знак = ініціалізують змінні в оголошеннях, "зірочкою" * відзначають в описах вказівники тощо.

Один і той же символ у мові С++ може мати різне призначення залежно від контексту, у якому він

використовується. Наприклад, знаком * позначають дві операції: арифметичне множення і звертання до даних за їх адресами, а також застосовують його як знак пунктуації для оголошення даних вказівникового типу.

Роль роздільників лексем відіграють також символи, які називають *пробільними*: пробіл, символи горизонтальної і вертикальної табуляції, нового рядка, нової сторінки, переходу на початок рядка. Ці символи можна записувати в довільній кількості між будь-якими двома лексемами. Самі ж лексеми розривати не можна. Компілятор розглядає послідовність довільних пробільних символів як один розділовий символ, що відокремлює лексеми.

У цьому ж призначенні роздільником є і *коментар* – текст, що роз'яснює програму. Кожен коментар починається двосимвольною комбінацією /* і закінчується парою символів */. Приклад коментаря:

/* Сортування масиву методом бульбашки */

Коментарі можна записувати в довільному місці програми між лексемами.

Запитання для самоконтролю

1. Що таке ключові (службові) слова мови? Назвіть декілька ключових слів мови С++. Яке їх призначення?

2. Чи всі записані далі ідентифікатори є правильними:
   xyz, xy0, 1st, mult-2, part_3, rez#4?
3. Поділіть перелічені цілочислові константи на десяткові, вісімкові та шістнадцяткові: 1101, 01101, 792, 0, 0X1101, 0x7, 07, 7.
4. Які константи належать до дійсних? Наведіть приклади дійсних констант.
5. Чи однаковими є дві наступні константи: 'N' та "N"? До яких груп вони належать?
6. Чим можуть бути відокремлені лексеми-слова в тексті програми?
7. Які символи можна використовувати в символьних рядках і коментарях?
8. Як можна переносити на наступний рядок довгі стрінгові константи?

РОЗДІЛ 2. СТРУКТУРА ТА ВИКОНАННЯ С++-ПРОГРАМИ

2.1. Створення проєкту консольного застосунку у Visual Studio C ++

Звичайною відправною точкою для програміста на С++ є додаток "Hello World", що виконується в командному рядку.

Visual Studio використовує проєкти, щоб упорядкувати код для додатка, і рішення, щоб упорядкувати проєкти. Проєкт містить всі параметри, конфігурації і правила, використовувані для складання програми. Він керує зв'язком між усіма файлами проєкту і будь-якими зовнішніми файлами. Щоб створити додаток, спочатку потрібно створити проєкт і рішення.

1. У Visual Studio в меню Файл обираємо пункти *Створити⇨Проєкт*, щоб відкрити діалогове вікно *Створення проєкту*. Обираємо шаблон *Консольний додаток* з тегами С++, Windows і Консоль, а потім тиснемо кнопку *Далі*.

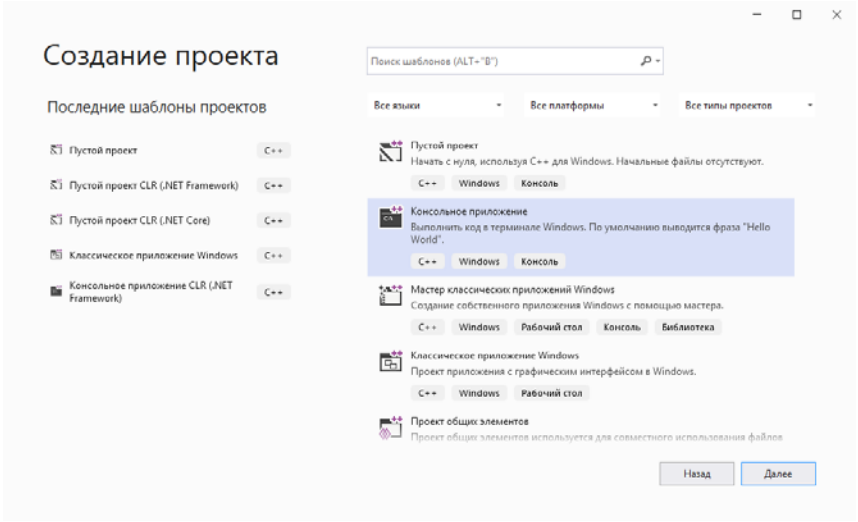


Рис. 2.1 Створення консольного додатку

2. У діалоговому вікні *Налаштувати новий проєкт* в поле *Ім'я проєкту* вводимо HelloWorld. Вибираємо *Створити*, щоб створити проєкт.

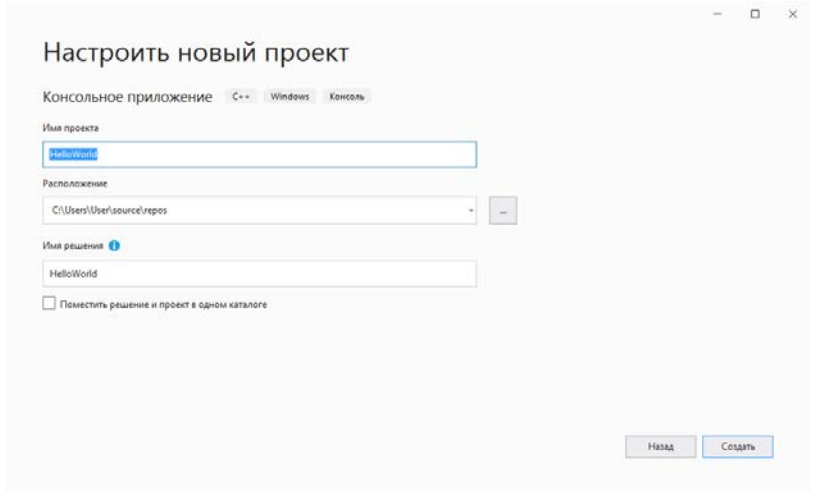


Рис. 2.2 Налаштування нового проєкту

3. Visual Studio створить проєкт. Можна приступати до додавання і зміни вихідного коду. За замовчуванням шаблон консольного застосунку додає вихідний код програми Hello World:

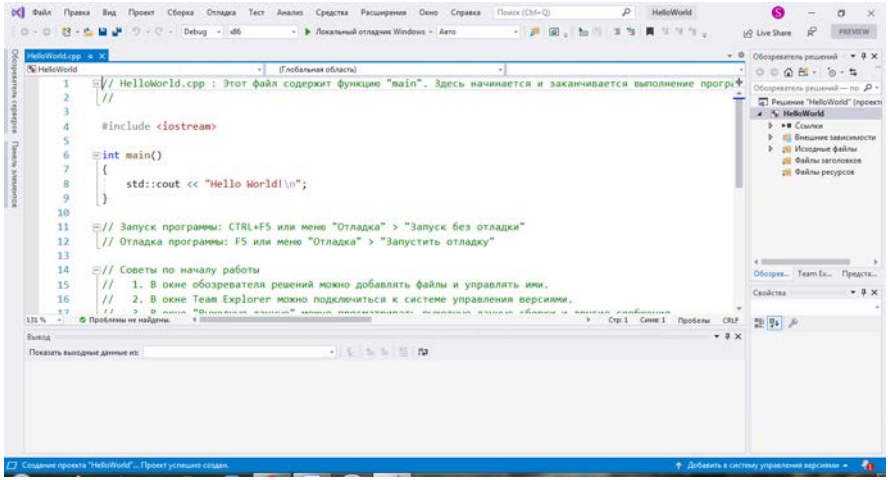


Рис. 2.3 Початковий код програми

Коли код в редакторі буде виглядати таким чином, можна перейти до наступного кроку і почати розробку програми.

2.2. Структура C++-програми

Основними частинами типової структури

програми на С++ є такі:

- директиви препроцесорної обробки;
- опис зовнішніх змінних (вихідних даних і результатів) та функцій;
- функції програми;
- головна функція програми **main()**, що має вигляд:

```
main()
{
опис змінних; виконавчі оператори;
}
```

У загальному випадку програма складається з декількох функцій, що не перетинаються (тобто «вкладення» однієї функції в іншу неприпустиме). Перед функціями та між ними можуть бути присутні оголошення об'єктів даних і оператори препроцесорної обробки. Функції користувача, які викликаються у головній функції main(), слід обов'язково описати до їх використання. Якщо жодної з функцій користувача не підключаємо у певну програму, то залишаємо лише відкриту та закриту дужки (згідно граматики мови щодо запису функцій).

Приклад запису простої програми. *Телефонні*

розмови з трьома мобільними операторами відповідно коштують c1, c2, c3 коп/хв. Розмови тривали t1, t2, t3 хв. відповідно. Яка сума нараховується до оплати за кожну розмову? За всі розмови разом?

```
#include <iostream>  // Директива препроцесора
#include <conio.h>
using namespace std;
int main()
{
	int c1=54, c2=75, c3=68; // Опис змінних
	int t1, t2, t3;
	setlocale(LC_ALL, ""); // Функція для підтримки
виведення кирилиці
	cout << "Введіть тривалості розмов в хвилинах t1, t2, t3
:\n"; // Виведення повідомлення на консоль
	cin >> t1 >> t2 >> t3; // Зчитування змінних з консолі
	cout << "Вартість першої розмови  : " << c1 * t1 / 100
<< " грн. " << c1 * t1 % 100 << " коп.\n";
	cout << "Вартість другої розмови  : " << c2 * t2 / 100
<< " грн. " << c2 * t2 % 100 << " коп.\n";
	cout << "Вартість третьої розмови : " << c3 * t3 / 100
<< " грн. " << c3 * t3 % 100 << " коп.\n";
	cout << "Сумарна оплата : " << (c1 * t1 + c2 * t2 + c3 *
t3) / 100 << " грн. " << (c1 * t1 + c2 * t2 + c3 * t3) % 100 <<
" коп.\n";
	_getch ();
	return 0;
}
```

Коментарі необхідні для пояснень призначення тих чи інших частин програми і їх текст завжди ігнорується компілятором. Мова С++ використовує два різновиди коментарів:

- *// текст* — **однорядковий коментар**, який

починається з двох символів «/» («коса риска») і закінчується символом переходу на новий рядок;

- */* текст */* — **багаторядковий коментар,** що розташовується між символами-дужками «/*» і «*/».

Багаторядкові коментарі не можуть бути вкладеними один в одний, а однорядкові коментарі можна вкладати в багаторядкові коментарі. Багаторядкові коментарі доцільно застосовувати для тимчасового виключення блоків при налагодженні програми.

Наведемо кілька порад стосовно раціонального складання коментарів:

- коментарі повинні бути добре складеними реченнями, мати правильну пунктуацію та містити тільки потрібну для супроводу інформацію;
- пропуск — один з найбільш ефективних коментарів, що значно поліпшує розуміння програми;
- штрихові лінії коментаря або порожні рядки застосовуються для поділу функцій та інших логічно завершених фрагментів програм.

Директива препроцесора **#include <iostream>**

забезпечує підключення до програми засобів зв'язку зі стандартними потоками введення-виведення даних. Ці засоби знаходяться у заголовному файлі **iostream**, де **i (*input*)** — введення, **o (*output*)** виведення, **stream** — потік, **h (*head*)** — заголовок. Ураховуючи те, що середовище **Visual Studio C++** не забезпечує затримку результатів на екрані, у текстах програм посібника з цією метою використано функцію **_getch()** зчитування символу з консолі без його виведення (бібліотека **conio.h**) або ж системну функцію **system("pause")** тощо.

При створенні програми враховують такі основні вимоги:

- усі використані константи, змінні, функції та нестандартні типи повинні бути оголошеними (описаними) до їхнього першого використання, і ці оголошення можна розміщати в будь-якому місці програми;
- кожний оператор мови закінчується символом **«;»;**
- фігурні дужки (**« { »** та **« } »**) виділяють складений оператор і все, що подано між такими дужками, синтаксично сприймається як один оператор;
- вкладені блоки повинні мати відступ у 3-4 символи, при цьому блоки одного рівня вкладеності слід вирівняти за вертикаллю.

Запитання для самоконтролю

1. З чого складається С++-програма?
2. Які види функцій використовуються в програмах?
3. У чому особливість функції main()?
4. Що таке заголовний файл, як він під’єднується до програми?
5. У яких місцях програми можна записувати коментарі?

РОЗДІЛ 3. ТИПИ ДАНИХ

У мові програмування С++ поняття типів даних відносять до великої системи оголошення змінних різних типів. Сама мова надає базові арифметичні типи, а також синтаксис для створення масивів.

Концепція типу даних

Основна мета будь-якої програми полягає в обробці даних. Дані різного типу зберігаються і обробляються по-різному. У будь-якій алгоритмічній мові кожна константа, змінна, результат обчислення виразу або функції повинні мати певний тип.

Тип даних визначає:

- внутрішнє представлення даних в пам'яті комп'ютера;
- множину значень, які можуть приймати величини цього типу;
- операції і функції, які можна застосовувати до величин цього типу.

Виходячи з цих характеристик, програміст вибирає тип кожної величини, використовуваної в програмі для подання реальних об'єктів. Обов'язковий опис типу дозволяє компілятору проводити перевірку допустимості різних конструкцій програми. Від типу величини залежать машинні команди, які будуть

використовуватися для обробки даних.

Всі типи мови С ++ можна розділити на *прості* та *складні*. У мові С ++ визначено шість основних типів даних для представлення цілих, дійсних, символьних і логічних величин. На основі цих типів програміст може вводити опис складових типів. До них відносяться масиви, перерахування, функції, структури, посилання, визначники, об'єднання і класи.

Основні типи даних часто називають *арифметичними*, тому що їх можна використовувати в арифметичних операціях. Для опису основних типів мови С++ використовують такі службові слова:

int (цілий);
char (символьний);
bool (логічний);
float (дійсний);
double (дійсний з подвійною точністю);
void (порожній, не має значення).

Базові типи даних

Кожна змінна має певний тип. І цей тип визначає, які значення може мати змінна, які операції з нею можна робити і скільки байт у пам'яті вона буде займати.

У мові С ++ визначені такі базові типи даних:

bool: логічний тип. Може приймати одну з двох

значень true (істина) і false (хиба). Кількість пам'яті для цього типу точно не визначена.
char: представляє один символ в кодуванні ASCII.
signed char і **unsigned char:** представляє один символ.
wchar_t: представляє розширений символ.
char16_t і **char32_t:** представляє один символ в кодуванні Unicode.
int: представляє ціле число.
short і **unsigned short:** також представляють ціле число.
unsigned int: представляє позитивне ціле число.
long, long long і **unsigned long long:** також представляють ціле число.
float, double і **long double:** представляють дійсне число ординарної точності з плаваючою точкою.
void: тип без значення.

Символьні типи

Для представлення символів додатково використовуються типи: **char**, **wchar_t**, **char16_t** і **char32_t**.

Цілочисельні типи

Цілочисельні типи представлені такими типами: **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **long long** і **unsigned long long**.

Типи чисел з плаваючою крапкою

Типи чисел з плаваючою крапкою та дробові числа представлені такими типами як **float**, **double** і **long double**.

Уточнення діапазону значень

Для уточнення внутрішнього подання та діапазону значень стандартних типів мова C++ використовує чотири специфікатори (модифікатори) типу:

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

Розміри типів даних

У таблиці для кожного типу вказано розмір, який він займає в пам'яті. Однак варто зазначити, що граничні розміри для типів розробники компіляторів можуть вибирати самостійно, виходячи з апаратних можливостей комп'ютера. Стандарт встановлює лише мінімальні значення, які повинні бути. Наприклад, для типів int і short мінімальне значення – 16 біт, для типу long – 32 біта. При цьому розмір типу long повинен бути не менше розміру типу int, а розмір типу int – не менший за розмір типу short, а розмір типу long double

повинен бути більший за double. Навіть в рамках однієї платформи різні компілятори можуть по різному підходити до розмірів деяких типів даних. Але в цілому використовуються ті розміри, які вказані в таблиці 3.1.

Таблиця 3.1. *Прості типи даних*

Тип	*Розмір, байт*	*Значення*
bool	1	true або false
unsigned short int	2	від 0 до 65 535
short int	2	від -32 768 до 32 767
unsigned long int	4	від 0 до 4 294 967 295
long int	4	від -2 147 483 648 до 2 147 483 647
int (16 розрядів)	2	від -32 768 до 32 767
int (32 розряди)	4	від -2 147 483 648 до 2 147 483 647
unsigned int (16 розрядів)	2	від 0 до 65 535
unsigned int (32 розряди)	4	від 0 до 4 294 967 295
char	1	від 0 до 255
float	4	від 1.2e-38 до 3.4e38
double	8	від 2.2e-308 до 1.8e308
long double	10	від 3.4e-4932 до 3.4e+4932

Специфікатор auto

Іноді буває важко визначити тип виразу. І згідно з останніми стандартами можна надати компілятору самому виводити тип об'єкта. І для цього застосовується специфікатор **auto**. При цьому, якщо ми визначаємо змінну зі специфікатором auto, ця змінна повинна бути обов'язково ініціалізована будь-яким значенням.

Перелічуваний тип даних

У мові С ++ існує особливий тип даних – *перелічуваний.* Це тип даних, що складається з множини іменованих значень, які називаються елементами, членами чи енумераторами типу. Він дозволяє створити тип, який містить кілька значень, заданих програмістом. Змінні цього типу не можуть прийняти значення, крім тих, що вказані в переліку.

Перелічуваний тип у програмі задається за допомогою ключового слова **enum** (скорочення від enumeration – перелічення):

enum назва_типу {перелік_значень} ;

При цьому вважається, що кожному значенню перелічуваного типу відповідає певне цілочислове значення. Значенням першої константи перелічуваного типу вважається нуль, а кожне наступне значення більше за попереднє на одиницю.

За замовчуванням для значень використовується основний тип int. Для довільного із значень, що перелічуються, можна явно задати власне ціле число. Це значення вводиться шляхом присвоєння константам певних значень, наприклад:

enum Animals {Cat = 5, Dog = 33, Hamster } ;

У подібних записах допускається повторення значень. Якщо для елемента не надано свого значення, то для нього надається значення на одиницю більше ніж у попередника. Так, елементу Hamster відповідатиме цілочислове значення 34.

Після того як перелічений тип буде оголошено, його можна використовувати для створення нових змінних. Змінні, що належать перелічуваному типу, визначаються так само, як і звичайні змінні. Наприклад:

Animals MyCat ;

Змінна MyCat має тип Animals і може приймати значення із заданого переліку (Cat, Dog, Hamster):

MyCat = Cat ;

MyCat = Hamster;

Зауважимо, що об’єкти перелічуваного типу можуть бути ініціалізовані або присвоєні лише у вигляді одного із перелічених значень, або іншим об’єктом того ж самого перелічуваного типу. Також їм не можна надавати цілочислові значення.

Контроль типів

Контроль типів повинен забезпечуватись розробником програми. Основною перевагою потоків C++ є автоматичний контроль типів.

Запитання для самоконтролю

1. Охарактеризуйте цілочисельні типи мови C++.
2. Охарактеризуйте дійсні типи мови C++.
3. Що таке модифікатор типу даних?
4. Як в C++ здійснюється приведення типів?
5. Для чого служить ключове слово enum?

РОЗДІЛ 4. ВИРАЗИ ТА ОПЕРАЦІЇ

Мова C++ надає розробнику багатий набір операторів, а також визначає їх призначення і використання до операндів вбудованих типів даних. Вона також визначає дію операторів, які використовуються в об'єктах класів. Така можливість називається перевантаженням операторів.

Вирази у C++ складаються з одного або декількох операндів (operand) які поєднуються за допомогою операторів (operator). Найпростішою формою виразу (expression) у C++ є одиничний літерал або змінна. Більш складніші вирази формуються із операторів та одного або декількох операндів.

Кожен вираз повертає певний результат (result). У виразах без операторів результатом буде сам операнд, наприклад літерал чи змінна величина. Результати виразів, що використовують оператори, залежать від використання кожного оператора до відповідного операнда. Окрім того, допоки не буде вказано тип кожного операнда, доти не буде відомо що означає даний вираз.

У C++ використовуються унарні оператори (unary operator) і парні оператори (binary operator). Унарні оператори, такі як звертання за адресою (&) та

звертання до значення (*) працюють з одним операндом. Парні оператори, такі як додавання (+) і віднімання (-), працюють із двома операндами. Існує також (лише один) трійний оператор (ternary operator), що працює з трьома операндами. Більш детально про тернарний оператор мова йтиме при вивченні операторів передачі керування.

Деякі символи мови, наприклад *, використовуються для позначень як унарних, так і парних операторів.

Для того, щоб знати порядок виконання виразу з декількома операторами, потрібно розглянути питання пріоритету оператора (precedence). Для кожного конкретного оператора існує порядок виконання, що дозволяє визначити послідовність опрацювання операндів.

Арифметичні оператори

Більшість програм на писаних на C++ виконують арифметичні обчислення. У таблиці 4.1 наведено арифметичні операції мови.

Таблиця 4.1. *Арифметичні оператори мови C++*

Дія в C++	***Арифметична операція***	***Алгебраїчний вираз***	***Вираз на C++***
Множення	*	ab	a*b

Ділення	/	x / y або x ÷ y	x /y
Остача	%	r mod s	r % s
Додавання	+	c + 7	c + 7
Віднімання	-	d - 4	d - 4

Призначення арифметичних операторів +, -, * та / є цілком очевидне. Результатом ділення цілих чисел є ціле число. Отримана при цьому дробова частина відкидається. Оператор ділення за модулем (остача) дозволяє вирахувати остачу від ділення лівого операнда на правий. Цей оператор застосовується тільки до цілочислових типів bool, char, short, int, long та їх беззнакових версій. Якщо при діленні або діленні за модулем обидва операнди додатні, то результат додатній або нуль. Якщо обидва операнди від'ємні, то результат ділення додатній (або нуль), а результат ділення за модулем від'ємний (або нуль). Якщо від'ємним є тільки один операнд, то результат виконання обох операторів залежатиме від конкретної машини.

```
21 % 6; // результат 3
21 % 7; // результат 0
-21 % -8; // результат -5
21 % -5; // результат 1 або -4
21 /6; // результат 3
21 / 7; // результат 3
```

-21 /-8; // результат 2
21 /-5; // результат -4 або -5

Оператори відношення та логічні оператори

Усі оператори відношення та логічні оператори повертають значення типу bool. Їм передають арифметичні операнди або операнди типу вказівника, а повертають вони логічні значення. У таблиці 4.2 наведено логічні оператори та оператори відношення.

Таблиця 4.2. *Оператори відношення та логічні оператори*

Оператор	***Дія***	***Використання***
= =	Рівне	вираз = = вираз
!=	Не рівне	вираз != вираз
<	Менше	вираз < вираз
<=	Менше рівне	вираз <= вираз
>	Більше	вираз > вираз
>=	Більше рівне	вираз >= вираз
!	Логічне NOT	! вираз
&&	Логічне AND	вираз && вираз
||	Логічне OR	вираз || вираз

Побітові оператори

Побітові оператори (bitwise operator) опрацьовують операнди як набір бітів, виконуючи перевірку і встановлення окремих бітів. Ці оператори працюють з операндами цілочислового типу. Їх можна застосовувати як до знакових, так і беззнакових цілих. У таблиці 4.3 наведено побітові оператори та вказано дію, яку вони виконують.

Таблиця 4.3. *Побітові оператори мови С++*

Оператор	***Дія***	***Використання***
~	Побітове *NOT*	~ вираз
<<	Зсув вліво	вираз1 << вираз2
>>	Зсув вправо	вираз1 >> вираз2
&	Побітове *AND*	вираз1 & вираз2
^	Побітове *XOR*	вираз1 ^ вираз2
|	Побітове *OR*	вираз1 | вираз2

Оператори інкремента і декремента

У мові програмування С++ є два оператори, яких немає в деяких інших мовах програмування. Це оператори інкремента (++) і декремента (--).

Оператор інкремента здійснює додавання до операнда число 1, а оператор декремента віднімає 1 від свого операнда. Це означає, що настанову

```
x = x + 1;
```
можна записати у вигляді префіксної форми
```
++x; // Префіксна форма оператора інкремента
```
або у вигляді постфіксної форми:
```
x++; // Постфіксна форма оператора інкремента
```
А настанова
```
x = x – 1;
```
аналогічна такій настанові:
```
--x; або x--;
```
У цих прикладах не має значення, у якій формі застосовано оператор інкремента: префіксній або постфіксній. Але, якщо оператор інкремента або декремента використовується як частина більшого виразу, то форма його застосування дуже важлива. Якщо такий оператор застосовується в префіксній формі, то мова програмування С++ спочатку виконає цю операцію, щоб операнд набув нового значення, яке потім буде використано іншою частиною виразу. Якщо ж оператор застосовується в постфіксній формі, то С++ використовує у виразі його старе значення, а потім виконає операцію, внаслідок якої операнд знайде нове значення. Для розуміння сказаного розглянемо такий фрагмент коду програми:
```
x = 10;
y = ++x;
```
У цьому випадку значення змінної у буде

дорівнювати 11. Але, якщо у цьому коді префіксну форму запису замінити постфіксною, то значення змінної у буде дорівнювати 10:

```
x = 10;
y = x++;
```

У обох випадках змінна x набуде значення 11. Різниця полягає тільки у тому, в який момент вона дорівнюватиме 11 (до або після присвоєння її значення змінній у).

Більшість C++-компіляторів для операцій інкремента і декремента створюють ефективніший код порівняно з кодом, що генерується під час використання звичайного оператора додавання і віднімання одиниці.

Запитання для самоконтролю

1. З чого формуються вирази в C++?
2. Які операції відносяться до унарних, бінарних, тернарних?
3. Які операції присвоєння існують в C++?
4. Поясніть різницю між постфіксною і префіксною формою інкремента та декремента.
5. Розставте операції в порядку зменшення пріоритетів: = % + * / ++.! &&

РОЗДІЛ 5. ВВЕДЕННЯ-ВИВЕДЕННЯ ДАНИХ

5.1. Форматоване введення-виведення

Форматоване введення-виведення величин здійснюється з використанням функцій scanf та printf, які успадковані з мови С. Щоб зв'язати програму користувача зі стандартною бібліотекою, де знаходяться ці функції, необхідно на початку програми включити заголовний файл stdio.h або cstdio.

Функція scanf, що забезпечує форматоване введення даних, має змінне число параметрів, при цьому перед відповідним параметром ставиться знак «&» — символ взяття адреси змінної. Наприклад, &x1 означає адресу змінної x1, а не значення, яке ця змінна має в даний момент. Рядок форматів функції scanf вказує, які дані очікуються на вході. Якщо функція зустрічає у форматному рядку знак «%», за яким розташований знак перетворення, то на вході будуть пропускатися символи, доки не з'явиться деякий непорожній символ.

Форма запису функції scanf має вигляд:

scanf ("*рядок форматних кодів*", *список імен змінних***);**

Рядок форматних кодів являє собою таку

структуру запису:

%[*][довжина][f|n][h|l] тип,

де

«**%**» — ознака початку форматного коду. Якщо за символом «%» йде символ, що не є символом керування форматом, то він розглядається як звичайна послідовність символів. При цьому наступні за ним символи (до наступного символу «%») також вважаються просто символами; якщо за символом «%» йде символ «*», то присвоювання наступного вхідного поля приглушується, поле читається, але не зберігається;

довжина — додатне десяткове ціле число, яке задає максимальне число символів, що може бути прочитане з вхідного потоку, доки не зустрінеться символ « » (пропуск) або символ, який не може бути перетворений відповідно до заданого формату;

f | n — дозволяють приглушити погодження за замовчуванням про використану модель пам'яті («далека», «близька» пам'ять);

h | l — предикати, що визначають відповідно аргументи типів short і long;

тип — задається одним із символів: d — десяткове ціле; i — десяткове, вісімкове чи шістнадцяткове ціле зі знаком; c — одиночний символ; u — беззнакове десяткове число; x, X —

беззнакове шістнадцяткове число; 0 — вісімкове число; s — сприймає символи без перетворення до символу «\n» або пропуску, доки не буде досягнута задана довжина (при виведенні видає до потоку всі символи до символу «\0» або до досягнення специфікованої точності); f, F — значення з плаваючою крапкою; e, E — значення у експоненціальній формі; G, g — значення зі знаком у формі f або e.

Аргументи у функції scanf мають бути записані у формі покажчиків, тобто у вигляді &x, &y, &mas[i] тощо. Для введення змінних k (типу int) і p (типу float) цю функцію можна записати так:

scanf(” %d %f \n ”, &k, &p);.

Таблиця 5.1. *Специфікатори формату функції* ***scanf()***

Код	**Значення**
%c	Зчитує один символ
%d	Зчитує десяткове ціле
%i	Зчитує ціле в будь-якому форматі (десяткове, вісімкове, шістнадцяткове)
%e	Зчитує дійсне число
%f	Зчитує дійсне число
%F	Аналогічно коду програми %f (тільки C99)
%g	Зчитує дійсне число
%o	Зчитує вісімкове число
%s	Зчитує рядок
%x	Зчитує шістнадцяткове число
%p	Зчитує покажчик

%n	Приймає ціле значення, дорівнює кількості символів, зчитаних дотепер
%u	Зчитує десяткове ціле без знаку
%[ ]	Проглядає набір символів
%%	Зчитує знак відсотка

Приклад. *Ввести два числа та обчислити їх суму.*

```
#include <iostream> // Директива препроцесора
#include <conio.h>
using namespace std;
int main()
{
	int a, b, c; // Опис змінних
	scanf_s(" %d  %d", &a, &b);//Ввід цілих десяткових чисел
	c = a + b; // Знаходження суми
	printf("Suma =%d \n", c); // Вивід суми на консоль
	_getch();
	return 0;
}
```

У результаті виконання програми буде виведено: Suma=13.

Форматний рядок наказує функції scanf ввести десяткове число, яке треба помістити в змінну **a**, а потім перейти до наступного непорожнього символу і з цього моменту почати введення нового десяткового числа, яке потім присвоюється змінній **b**. Якщо за рядком керування форматом аргументів більше, ніж специфікацій формату, зайві аргументи ігноруються. Коли для специфікацій формату недостатньо

аргументів, результат не визначений.

У наведеному фрагменті програми для форматованого виведення даних використовується функція printf.

Функція printf може використовуватися, наприклад, для виведення повідомлення на екран:

printf ("Введіть вхідні дані \n");.

Для звертання до функції використовуються параметри, які розташовані у круглих дужках. Найчастіше функція printf реалізується для виведення значень змінних. Першим аргументом у звертанні до функції ставиться рядок форматів (береться в лапки), а наступними, якщо вони є, — об'єкти, що виводяться.

Рядок форматів може включати звичайні символи, які копіюються при виведенні, і специфікації перетворення, що починаються із символу«% », за специфікаціями йде символ перетворення. Кожна специфікація перетворення відповідає одному з аргументів, що йдуть за форматним рядком, і між ними встановлюється взаємно однозначна відповідність, наприклад:

printf ("Значення a, b, c дорівнюють: %d %d. %d \n", a, b, c);.

Тут літера **d** у специфікації перетворення вказує, що значення аргументу має бути представлено як десяткове ціле число.

При виведенні функція printf використовує ті самі специфікації, що й функція scanf при введенні. Наприклад, у функції printf вигляду

printf (” % c=%d \n”, g, g);

значення змінної **g** виводиться як символ алфавіту, а після знаку «=» — як числове значення. Перед символом перетворення може стояти числовий коефіцієнт, що вказує кількість позицій у виведеному рядку, відведених для елемента виведення. Список форматних кодів має таку форму запису:

% [прапорець] [довжина] [точність] [f | n] [h | l] тип ,

де

прапорець — символ, що керує вирівнюванням виведення і виведенням пропусків, десяткової крапки, ознак чисел вісімкової і шістнадцяткової систем числення. Прапорець може задаватися одним із символів:

«-» — вирівнювання вліво усередині заданого поля;

«+» — виведення знака числа;

« » (**пропуск**) — приєднання пропуску до виведеного числа, якщо число є додатним і має тип зі знаком;

«#» — виводиться ідентифікатор системи числення для цілих: 0 — для вісімкових

чисел, 0x чи 0X — для шістнадцяткових чисел;

довжина — визначає мінімальну кількість виведених символів, якщо довжина більше виведеної кількості символів, то виведене значення доповнюється пропусками, у випадку, коли довжина менше кількості символів у виведеному значенні або вона не задана, виводяться всі символи значення (відповідно до поля точність, якщо воно є);

точність — задається цілим числом після крапки і визначає кількість виведених символів, кількість знаків після крапки; на відміну від поля довжини поле точність може привести до «зрізання» виведених даних.

Параметри **f**, **n**, **h**, **l** і **тип** списку форматних кодів за змістом аналогічні раніше описаним для функції scanf.

Виведення результатів з використанням форматних кодів функції printf може мати вигляд:

printf (” % 3.0 f % 6.1 f \ n ”, x, y);

Таблиця 5.2. *Специфікатори формату функції* ***printf()***

Код	Формат
%c	Символ
%d	Десяткове ціле із знаком
%i	Десяткове ціле із знаком
%e	Експоненціальне представлення (рядкова

	буква e)
%E	Експоненціальне представлення (прописна буква E)
%f	Значення з плаваючою крапкою
%g	Використовує коротший з двох форматів: %e або %f (якщо %e, використовує рядкову букву e)
%G	Використовує коротший з двох форматів: %E або %F (якщо %E використовує прописну букву e)
%o	Вісімкове ціле без знаку
%s	Рядок символів
%u	Десяткове ціле без знаку
%x	Шістнадцяткове ціле без знаку (рядкові букви)
%X	Шістнадцяткове ціле без знаку (прописні букви)
%p	Покажчик
%n	Відповідний аргумент повинен бути покажчиком на ціле. Даний специфікатор зберігає у цьому цілому число символів, виведених у вихідний потік до поточного моменту (до виявлення специфікатора %n)
%%	Виводить символ %

Приклад. *Обчислити значення функції* $y = ax^2 - \sin x$, *якщо* $a=10,5$; $x \in [-1; 2]$; $hx=0,5$.

```
#include <iostream>
#include<cmath>
//#include <conio.h>
using namespace std;
int main()
```

```
{
    float x, y, a (10.5);
    printf("\t Vyvid rezultatu\n");
    for (x = -1; x <= 2; x += 0.5)
    {
        y= a * pow(x, 2) - sin(x);     //y = a*x*x - sin(x);
        printf(" \t x = % 4.1f    y = % 6.3f \n", x, y);
    }
    //_getch ();      //задержка экрана
    system("pause"); //задержка экрана
    return 0;
       }
```

Результати обчислення:

Vyvid rezultatu

x = -1.0 y =11.341

x = -0.5 y = 3.104

x = 0.0 y = 0.000

x = 0.5 y = 2.146

x = 1.0 y = 9.659

x = 1.5 y = 22.628

x = 2.0 y = 41.091.

5.2. Потокове введення-виведення

У мові С++ дії, що пов’язані з операціями введення і виведення, виконуються за допомогою функцій бібліотек. Функції ведення та виведення бібліотек мови дозволяють читати дані з файлів і пристроїв і писати дані у файли та на пристрої.

Бібліотека мови С++ підтримує три рівні

введення-виведення даних:

- введення-виведення потоку;
- введення-виведення нижнього рівня;
- введення-виведення для консолі та порту.

При введенні-виведенні потоку всі дані розглядаються як потік окремих байтів. Для користувача потік — це файл на диску або фізичний пристрій, наприклад, дисплей чи клавіатура, або пристрій для друку, з якого чи на який направляється потік даних. Операції введення-виведення для потоку дозволяють обробляти дані різних розмірів і форматів від одиночного символу до великих структур даних. Програміст може використовувати функції бібліотеки, розробляти власні і включати їх у бібліотеку. Для доступу до бібліотеки цих класів треба включити в програму відповідні заголовні файли.

За замовчуванням стандартні введення і виведення повідомлень про помилки відносяться до консолі користувача (клавіатури та екрана). Це означає, що завжди, коли програма очікує введення зі стандартного потоку, дані повинні надходити з клавіатури, а якщо програма виводить дані — то на екран.

У мові C++ існує декілька бібліотек, які містять засоби введення-виведення, наприклад: cstdio (stdio.h), iostream. Найчастіше застосовують потокове введення-

виведення даних, операції якого включені до складу класів istream або iostream. Доступ до бібліотеки цих класів здійснюється за допомогою використання у програмі директиви компілятора #include <iostream>.

Для потокового введення даних вказується операція «>>» («читати з»). Це перевантажена операція, визначена для всіх простих типів і покажчика на char. Стандартним потоком введення є cin.

Формат запису операції введення має вигляд:

cin [>> values]; ,

де values — змінна.

Так, для введення значень змінних x і y можна записати:

cin >> x >> y;

Кожна операція «>>» передбачає введення одного значення. При такому введенні даних необхідно дотримуватись конкретних вимог:

- для послідовного введення декількох чисел їх слід розділяти символом пропуску або Enter (дані типу char розділяти пропуском необов'язково);
- якщо послідовно вводиться символ і число (або навпаки), пропуск треба записувати тільки в тому випадку, коли символ (типу char) є цифрою;

– потік введення ігнорує пропуски;

– для введення великої кількості даних одним оператором їх можна розташовувати в декількох рядках (використовуючи Enter);

– операція введення з потоку припиняє свою роботу тоді, коли всі включені до нього змінні одержують значення. Наприклад, для операції введення x і y, що вказана вище, можна ввести значення x та y таким чином:

2.345 789

або

2.345

789.

Оскільки в цьому прикладі пропуск є роздільником між значеннями, що вводяться, то при введенні рядків, котрі містять пропуски у своєму складі, цей оператор не використовується. У такому випадку треба застосовувати функції getline(), gets_s() тощо. У мові C++ бажано здійснювати потокові введення-виведення даних.

Для потокового виведення даних необхідна операція «<<» («записати в»), що використовується разом з ім'ям вихідного потоку **cout**. Наприклад, вираз

cout << x;

означає виведення значення змінної x (або запис у потік). Ця операція вибирає необхідну функцію

перетворення даних у потік байтів.

Формат запису операції виведення представляється як:

cout << data [<< data1];,

де data, data1 – це змінні, константи, вирази тощо.

Потокова операція виведення може мати вигляд:

cout << “y =” << x + a – sin(x) << “\n“;.

Застосовуючи логічні операції, вирази треба брати в дужки:

cout << “p =” << (a && b || c) << “\n“;.

Символ переведення на наступний рядок записується як рядкова константа, тобто “\n”, інакше він розглядається не як символ керуючої послідовності, а як число 10 (код символу). Таких помилок можна уникнути шляхом присвоювання значення керуючих символів змінним, тобто:

```
#define sp " "
#define ht "\t"
#define hl "\n"
```

Тепер операцію виведення можна здійснити так:

cout << "y = " << (x + a-sin(x)) << hl;

Слід пам’ятати, що при виведенні даних з використанням «cout <<» не виконується автоматичний перехід на наступний рядок, для

реалізації такого переходу застосовується керуюча послідовність **“\n”** або операція **endl**. Тобто, вивести рядкову константу можна, наприклад, так:

cout << “ Сенс успіху - в русі до нього. Крайньої точки не існує. \n“;

або

cout << “ Сенс успіху - в русі до нього. Крайньої точки не існує.” << endl;.

Приклад. *Написати програму, що здійснює виведення даних, пояснювальні повідомлення, а також символи переведення рядка.*

```
#include <iostream>
#include <clocale>
#include <conio.h>
using namespace std;
int main()
{
	setlocale(LC_CTYPE, "ukr");
	char first = 'W';
	char middle = 'P';
	char last = 'S';
	int wozrast = 20;
	int doplata = 2;
	float zarplata = 309.75;
	float prozent = 8.5;
	//---------------- Вивід результатів
	cout << "Перевірка вихідних даних\n";
	cout << first << middle << last << "\n\n";
	cout << "Вік доплата зарплата процент:\n";
	cout << wozrast << "     " << doplata << "     " <<
zarplata << "    " << prozent << endl;
	_getch();
	//system("pause");
	return 0;
```

```
}
```

Для додаткового керування даними, що виводяться, використовують символи табуляції («\t») чи маніпулятори **setw(w)** та **setprecision(d).** Маніпулятор **setw(w)** призначений для зазначення довжини поля, що виділяється для виведення даних (**w** — кількість позицій). Маніпулятор **setprecision(d)** визначає кількість позицій у дробовій частині дійсних чисел.

Маніпулятори змінюють вигляд деяких змінних в об'єкті cout, що у потоці розташовані за ними. Ці маніпулятори називають прапорцями стану. Коли об'єкт посилає дані на екран, він перевіряє прапорці, щоб довідатися, як виконати завдання, наприклад, запис:

cout << 456 << 789 << 123;

призводить до виведення значення у вигляді: 456789123, що ускладнює визначення групи значень.

Приклад. *Написати програму, використовуючи маніпулятор setw().*

```
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
int main()
{
	cout << 456 << 789 << 123 << endl;
```

```
      cout << setw(5) << 456 << setw(5) << 789 << setw(5) <<
123 << endl;
      cout << setw(7) << 456 << setw(7) << 789 << setw(7) <<
123 << endl;
      _getch();
      }
```

Результати виконання програми:

```
456789123
  456   789   123
     456     789     123
```

У цьому прикладі з'явився новий заголовний файл iomanip, що дозволяє застосовувати функції маніпуляторів. При використанні функції setw() число вирівнюється вправо в межах заданої ширини поля виведення. Якщо ширина недостатня, то вказане значення ігнорується.

Функція setprecision(2) повідомляє про те, що число з плаваючою крапкою виводиться з двома знаками після крапки з округленням дробової частини, наприклад, при виконанні операції

cout << setw(7) << setprecision(2) << 123.456789;

буде отримано такий результат: 123.46.

Функції cout.width(w) та cout.precision(d), які потребують підключення тільки заголовного файла iostream, виконують дії, подібні тим, що і функції setw(w) та setprecision(d).

Операція введення використовує ті ж самі маніпулятори, що й операція виведення. Список

змінних, у які будуть поміщені дані, визначений у values.

Приклад. *Написати програму обчислення податку на продаж.*

```
#include <iostream>
#include <iomanip>
#include <clocale>
#include <conio.h>
using namespace std;
int main()
{
	setlocale(LC_CTYPE, "ukr");
	float prod_sum; // prod_sum – сума продажу
	float nalog;
	//---------------  вивід підказки для користувача
	cout << "Введіть суму продажу ";
	cin >> prod_sum;
	//.............. обчислення податку на продаж
	nalog = prod_sum * 0.7;
	cout << " " << setprecision(2) << prod_sum;
	cout << " " << setprecision(2) << nalog << "\n";
	_getch();
	return 0;
}
```

Унаслідок того, що у першому операторі cout відсутня інструкція переведення рядка, відповідь користувача на підказку (тобто введене значення змінної prod_sum) з’явиться відразу праворуч за самою підказкою.

5.3. Файлове введення-виведення

Виведення у файловий потік

Як вже відомо, заголовок iostream визначає вихідний потік cout. Аналогічно, заголовок fstream визначає клас вихідного файлового потоку з ім'ям ofstream. Використовуючи об'єкти класу ofstream, програми можуть здійснювати вивід у файл. Для початку потрібно оголосити об'єкт типу ofstream, вказавши ім'я необхідного вихідного файлу як символьний рядок, що показано нижче:

ofstream file_object ("FILENAME.EXT");

Якщо вказати ім'я файлу при оголошенні об'єкту типу **ofstream**, C++ створить новий файл на вашому диску, використовуючи вказане ім'я, або перезапише файл з таким же ім'ям, якщо він вже існує на вашому диску.

Приклад. *Створити об'єкт типу ofstream і потім, використовуючи оператор вставки, вивести декількох рядків тексту у файл INFO.DAT.*

```
#include <fstream>
using namespace std;
void main()
{
	ofstream inf_file("INFO.DAT");
	inf_file << "Вчимося програмувати на мові C++, " <<
```

```
"другий семестр" << endl;
       inf_file << "RDGU, FMI" << endl;
       inf_file << "2020" << endl;
       }
```

У даному випадку програма відкриває файл INFO.DAT і потім записує три рядки у файл, використовуючи оператор вставки.

Читання з вхідного файлової потоку

Програми можуть виконати операції введення з файлу, використовуючи об'єкти типу ifstream. Знову ж таки, потрібно просто створити об'єкт, передаючи йому в якості параметра ім’я потрібного файлу:

ifstream input_file ("filename.EXT");

Приклад. *Написати програму для відкриття файлу INFO.DAT, створеного за допомогою попередньої програми, і читання, а потім відображення на екран перших три елементи файлу.*

```
#include <iostream>
#include <fstream>
#include <clocale>
using namespace std;
void main()
{
    setlocale(LC_CTYPE, "ukr");
    ifstream input_file("INFO.DAT");
    char one[64], two[64], three[64];
    input_file >> one;
    input_file >> two;
    input_file >> three;
```

```
    cout << one << endl;
    cout << two << endl;
    cout << three << endl;
}
```

Подібно cin, вхідні файлові потоки використовують порожні символи, щоб визначити, де закінчується одне значення і починається інше. У результаті при запуску програми на дисплеї з'явиться наступний результат:

вчимося
програмувати
на

Читання цілого рядка файлового вводу

Ви вже знаєте, що ваші програми можуть використовувати cin.getline для читання цілого рядка з клавіатури. Подібним чином об'єкти типу ifstream можуть використовувати getline для читання рядка файлового введення.

Приклад *програми використання функції getline для читання всіх трьох рядків файлу INFO.DAT:*

```
#include <iostream >
#include <fstream>
#include <clocale>
using namespace std;
void main()
{
    setlocale(LC_CTYPE, "ukr");
```

```
    ifstream input_file("INFO.DAT");
    char one[64], two[64], three[64];
    input_file.getline(one, sizeof(one));
    input_file.getline(two, sizeof(two));
    input_file.getline(three, sizeof(three));
    cout << one << endl;
    cout << two << endl;
    cout << three << endl;
}
```

У даному випадку програма успішно читає вміст файлу, тому що вона знає, що файл містить три рядки. Однак у багатьох випадках програма не знатиме, скільки рядків міститься у файлі. У таких випадках програми будуть просто продовжувати читання вмісту файлу поки не зустрінуть кінець файлу.

Визначення кінця файла

Звичайною файловою операцією в програмах є читання вмісту файлу, поки не зустрінеться його кінець. Щоб визначити кінець файлу, програми можуть використовувати функцію eof потокового об'єкта. Ця функція повертає значення 0, якщо кінець файлу ще не зустрівся, і 1 – у протилежному випадку. Використовуючи цикл while, програми можуть безперервно читати вміст файлу, поки не знайдуть кінець файлу, як показано нижче:

```
while (! input_file.eof ())
{
 // Оператори
```

}

У даному випадку програма буде продовжувати виконувати цикл, поки функція **eof** повертає FALSE (0).

Приклад *програми використання функції* ***eof*** *для читання вмісту файлу INFO.DAT, поки не досягнеться кінця файлу:*

```
#include <iostream>
#include <fstream>
#include <clocale>
using namespace std;
void main()
{
    setlocale(LC_CTYPE, "ukr");
    ifstream input_file("INFO.DAT");
    char line[64];
    while (!input_file.eof())
    {
        input_file.getline(line, sizeof(line));
        cout << line << endl;
    }
}
```

Приклад *програм, яка читає вміст файлу по одному слову за один раз, поки не зустрінеться кінець файлу:*

```
#include <iostream>
#include <fstream>
#include <clocale>
using namespace std;
void main()
```

```
{
    setlocale(LC_CTYPE, "ukr");
    ifstream input_file("INFO.DAT");
    char word[64];
    while (!input_file.eof())
    {
        input_file >> word;
        cout << word << endl;
    }
       }
```

І нарешті, ***приклад*** *програми, яка читає вміст файлу по одному символу за один раз, використовуючи функцію* ***get****, поки не зустріне кінець файлу:*

```
#include <iostream>
#include <fstream>
#include <clocale>
using namespace std;
void main()
{
    setlocale(LC_CTYPE, "ukr");
    ifstream input_file("INFO.DAT");
    char letter;
    while (!input_file.eof())
    {
        letter = input_file.get();
        cout << letter;
    }
}
```

Перевірка помилок при виконанні файлових операцій

Програми, представлені до теперішнього моменту, припускали, що під час файлових операцій В/В не відбуваються помилки. На жаль, це буває не

завжди. Наприклад, якщо відкрити файл для введення, програми повинні перевірити, що файл існує. Аналогічно, якщо програма пише дані у файл, то необхідно переконатися, що операція пройшла успішно (наприклад, відсутність місця на диску, швидше за все, завадить запису даних). Щоб допомогти програмам стежити за помилками, можна використовувати функцію fail файлового об'єкта. Якщо в процесі файлової операції помилок не було, функція поверне ФАЛЬШ (0). Однак, якщо зустрілася помилка, функція fail поверне ІСТИНУ. Наприклад, якщо програма відкриває файл, їй слід використовувати функцію fail, щоб визначити, чи відбулася помилка, як це показано нижче:

```
ifstream input_file ( "FILENAME.EXT");
if (input_file.fail ())
{
cerr << "Помилка відкриття FILENAME.EXT" << endl;
exit (1);
}
```

Таким чином, програми повинні переконатися, що операції читання і запису пройшли успішно.

Приклад *програми використання функції* ***fail*** *для перевірки різних помилкових ситуацій:*

```
#include <iostream>
#include <fstream>
#include <clocale>
using namespace std;
void main()
{
    setlocale(LC_CTYPE, "ukr");
    char line[256];
    ifstream input_file("INFO.DAT");
    if (input_file.fail()) cerr << "Помилка відкриття INFO.DAT"
<< endl;
    else
    {
        while ((!input_file.eof()) && (!input_file.fail()))
        {
            input_file.getline(line, sizeof(line));
            if (!input_file.fail()) cout << line << endl;
        }
    }
}
```

Закриття файла, якщо роботу з ним завершено

При завершенні програми операційна система закриє відкриті нею файли. Однак, як правило, якщо програмі файл більше не потрібен, вона повинна його закрити. Для цього використовується функція close, як показано нижче:

input_file.close ();

Коли закрити файл, всі дані, які ваша програма писала у цей файл, скидаються на диск, і оновлюється запис каталогу для цього файлу.

Управління відкриттям файлу

Якщо потрібно, щоб програма додавала інформацію в кінець існуючого файлу, то при його відкритті вказується другий параметр, як показано нижче:

ifstream output_file ("FILENAME.EXT", ios :: app);

У даному випадку параметр ios :: app вказує режим відкриття файлу для дозапису. Інші можливі значення другого параметру перераховані в таблиці 5.3.

Таблиця 5.3. *Значення режимів відкриття*

Режим відкриття	Призначення
ios :: app	Відкриває файл в режимі додавання, розташовуючи файловий покажчик у кінці файлу.
ios :: ate	Має у своєму розпорядженні файловий покажчик в кінці файлу.
ios :: in	Вказує відкрити файл для введення.
ios :: nocreate	Якщо вказаний файл не існує, не створювати файл і повернути помилку.
ios :: noreplace	Якщо файл існує, операція відкриття повинна бути перервана і повинна повернути помилку.
ios :: out	Вказує відкрити файл для виводу.
ios :: trunc	Скидає (перезаписує) утримуємо, з існуючого файлу.

Наступна операція відкриття файлу відкриває файл для виводу, використовуючи режим ios :: noreplace, щоб запобігти перезапису існуючого файлу:

ifstream output_file ("FIlename.EXT", ios :: out | ios :: noreplace);

Запитання для самоконтролю

1. Які способи передбачені в C++ для вводу (виводу) даних?
2. Які оператори використовуються для вводу(виводу) на потік?
3. Які види потоків є в C++?
4. Які бібліотеки необхідні для роботи з потоками вводу/виводу?
5. Які escape-послідовності використовують для форматування рядкових літералів?
6. Який синтаксис функції printf()?
7. Які специфікатори перетворення використовуються для виведення цілих (дійсних) чисел?
8. Який символ задає специфікатор перетворення?
9. Який синтаксис функції scanf()?

РОЗДІЛ 6. ОПЕРАТОРИ

6.1. Оператори-вирази

У мові C++ оператором вважається кожен допустимий вираз, що закінчується знаком «крапка з комою». Тобто всі наступні записи:

```
/*1*/   z = 2.15 * x;
/*2*/   (a + b) * c;
/*3*/   256;
/*4*/   _getch();
/*5*/   g > MAX ? ++k1 : ++k2;
```

синтаксично можна розглядати як *оператори-вирази*. Проте за означенням оператор повинен виконувати певну дію. Серед записаних операторів дії виконують: перший (присвоєння z значення виразу), четвертий (виклик функції зчитування символу з консолі) і п’ятий (збільшення змінної k1 або k2). Другий і третій оператори дій не виконують, тому як оператори вони є беззмістовними (хоча в другому операторі обчислюється значення виразу (a+b)*c, але воно ніде не використовується і втрачається після завершення оператора).

Серед операторів-виразів виділяють дві групи:

- оператори присвоєння;
- оператори звертання до функцій.

Оператори присвоєння. Синтаксично оператори

присвоєння є виразами, у яких останньою виконується операція присвоєння, а в кінці записано знак ;. Можна використовувати кожну з форм операцій присвоєння: звичайне присвоєння чи інкремент/декремент. Наведемо приклади таких операторів:

```
y = cos((x + 2.5) / 3 - x * x); // звичайне присвоєння
k = m = n = arr[1];             // послідовні присвоєння
num /= 10;                      // комбіноване присвоєння
x++;                            // інкремент
y--;                            // декремент
```

Оператори виклику функцій. У мові C++ звертання до функції може використовуватись як операнд виразу (за умови, що функція повертає відповідне значення) або виступати окремим оператором.

Якщо функція не повертає ніякого значення (тип її значення void), то така функція є аналогом процедури в інших мовах програмування, і її можна викликати тільки як окремий оператор. Нижче наведено два приклади звертання до функцій, які не повертають значення:

```
srand(time(0)); //ініціалізує генератор випадкових чисел
free(p);        //звільняє динамічну пам'ять
```

Як окремий оператор може виступати і звертання до функції, яка повертає певне значення. У цьому випадку виконуються дії, що передбачені у функції, а значення, яке вона повертає, не застосовується і

втрачається. Наприклад,

```
scanf_s("%d%d", k, m); //повертає кількість введених даних
puts("Кінець розрахунку"); //повертає ненульове значення
strcpy_s(s,buf); //повертає адресу скопійованого рядка
```

Порожній оператор. Найпростішим у записі є *порожній* оператор – він позначається тільки знаком ;. Порожній оператор не виконує ніяких дій, його застосовують у тих випадках, коли синтаксична конструкція вимагає запису оператора, але ніяких дій виконувати не треба. Найчастіше порожній оператор використовують в операторах циклу, рідше – в умовних операторах чи інших конструкціях.

6.2. Умовні оператори

Команди розгалуження if

Повний формат її запису є таким:

if(*вираз*) *команда*; **else** *команда*;

У цьому записі під елементом *команда* розуміємо одну команду мови програмування С++. Частина **else** необов'язкова. Замість елемента *команда* може бути використаний *блок команд*. У цьому випадку формат запису **if**-команди набуде такого вигляду:

if(*вираз*) { *блок команд* } **else** { *блок команд* }

Якщо елемент *вираз*, який є умовним виразом, під час обчислення дасть значення ІСТИНА, то буде виконана **if**-команда; інакше – **else**-команда (якщо така існує). Обидві команди ніколи одночасно не виконуються. Умовний вираз, який керує виконанням **if**-команди, може мати будь-який тип, що є дійсним для С++-виразів, але головне, щоб результат його обчислення можна було інтерпретувати як значення **true** або **false**.

Використання **if**-команди розглянемо на прикладі коду програми, яка є версією гри "Вгадай магічне число". Програма відображає випадкове число і пропонує його вгадати. Якщо Ви відгадуєте число, то програма виводить на екран повідомлення ** Правильно **. У цьому коді програми представлена ще одна бібліотечна функція **rand**(), яка повертає випадково вибране ціле число від нуля до **RAND_MAX**. Для використання цієї функції необхідно приєднати до програми заголовок <**cstdlib**>.

Приклад. *Написати програму "Вгадай магічне число".*

```
#include <iostream> // Потокове введення-виведення
#include <clocale>
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
```

```
{
       setlocale(LC_CTYPE, "ukr");
       int magic; // Магічне число
       int guess; // Варіант користувача
       magic = rand(); // Отримуємо випадкове число.
       cout << "Введіть свій варіант магічного числа: "; cin >>
guess;
       if (guess == magic) cout << "** Правильно **";
       _getch(); return 0;
     }
```

Приклад *удосконаленої програми "Вгадай магічне число":*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
       setlocale(LC_CTYPE, "ukr");
       int magic; // Магічне число
       int guess; // Варіант користувача
       magic = rand(); // Отримуємо випадкове число з діапазону
// від 0 до 32767
       cout << "Введіть свій варіант магічного числа: "; cin >>
guess;
       if (guess == magic) cout << "** Правильно **";
       else cout << "... Дуже шкода, але Ви помилилися.";
       _getch(); return 0;
     }
```

Як зазначалося раніше, нуль автоматично перетвориться в **false**, а всі ненульові значення – в **true**. Це означає, що будь-який вираз, який дає внаслідок обчислення нульове або ненульове значення, можна використовувати для керування **if-**

командою. Наприклад, наведена вище програма зчитує з клавіатури два цілі числа і відображає частку від ділення першого на друге. Щоб не допустити ділення на нуль, у програмі використано **if**-команду.

Приклад *програми обчислення частки двох цілих чисел:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int a, b;
	cout << "Введіть два числа: "; cin >> a >> b;
	if (b) cout << a / b << endl;
	else cout << "На нуль ділити не можна" << endl;
	_getch(); return 0;
}
```

Зверніть увагу на те, що значення змінної b (дільник) порівнюється з нулем за допомогою команди **if**(b), а не команди **if**(b!=0). Йдеться про те, що, коли значення b=0, умовний вираз, який керує командою c, оцінюється як ФАЛЬШ, то це призводить до виконання **else**-гілки. Інакше (коли b містить ненульове значення) умова оцінюється як ІСТИНА, тобто ділення легко здійснюється. Немає ніякої потреби використовувати наступну **if**-команду, яка до того ж не свідчить про хороший стиль програмування

мовою C++:

if(b != 0) **cout** << a/b << **endl**;

Ця форма **if**-команди вважається застарілою і потенційно неефективною.

Вкладені if-команди

*Вкладена **if**-команда — команда, яку використовують як елемент* команди *будь-якої іншої **if**- або **else**-команди.*

Слід пам'ятати, що **else**-команда завжди належить до найближчої **if**-команди, яка знаходиться усередині того ж програмного блоку, але ще не пов'язана ні з якою іншою **else**-командою. Наприклад:

```
if(c) {
if(d) statement1;
if(f) statement2; // Ця if-команда
else statement3; // пов'язана з цією else-командою.
}
else statement4; // Ця else-команда пов'язана з if(c).
```

Як стверджується в коментарях, остання **else**-команда не пов'язана з командою **if**(d), оскільки вони не знаходяться в одному блоці (незважаючи те, що ця **if**-команда – найближча, яка не має при собі "**else**-пари"). Внутрішня **else**-команда пов'язана з командою

if(f), оскільки вона найближча і знаходиться усередині того ж блоку.

Мова програмування C++ дає змогу 256 рівнів вкладення, але на практиці рідко доводиться вкладати **if**-команди на "таку глибину".

Приклад *програми "Вгадай магічне число" (друге удосконалення):*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int magic; // Магічне число
	int guess; // Варіант користувача
	magic = rand(); // Отримуємо випадкове число
	cout << "Введіть свій варіант магічного числа: "; cin >>
guess;
	if (guess == magic) {
		cout << "** Правильно **" << endl;
		cout << magic << " і є те саме магічне число" <<
endl;
	}
	else {
		cout << "... Дуже шкода, але Ви помилилися.";
		if (guess > magic)
			cout << "Ваш варіант перевищує магічне
число" << endl;
		else
			cout << "Ваш варіант є меншим від
магічного числа" << endl;
	}
	_getch(); return 0;
}
```

Конструкція "сходинок" if-else-if

Дуже поширеною у програмуванні конструкцією, в основі якої знаходиться вкладена **if**-команда, є "сходинки" **if-else-if**. Її можна представити в такому вигляді:

if(*умова*) команда;
else
if(умова) команда;
else
if(умова) команда;
else команда;

У цьому записі під елементом *умова* розуміють умовний вираз, який обчислюється зверху вниз. Як тільки у якій-небудь гілці виявиться істинний результат, то буде виконану команду, пов'язану з цією гілкою, а всі решта "сходинки" опускаються. Якщо виявиться, що жодна з умов не є істинною, то буде виконано останню **else**-команду (можна вважати, що вона здійснює роль умови, яка діє за замовчуванням). Якщо останню **else**-команду не задано, а всі інші виявилися помилковими, то взагалі ніяка дія не буде виконана.

Команда багатовибірного розгалуження switch

Команда **switch** – команда багатовибірного розгалуження, яка дає змогу вибрати одну з множини альтернатив. Хоча різнонаправлене тестування можна реалізувати за допомогою послідовності вкладених **if**-команд, однак у багатьох ситуаціях команда **switch** виявляється набагато ефективнішим і очевидним рішенням.

Команда багатовибірного розгалуження працює таким чином. Значення виразу послідовно порівнюється з константами із заданого переліку. Внаслідок виявлення збігу для однієї з умов порівняння здійснюється послідовність команд, пов'язана з цією умовою. Загальний формат запису команди **switch** є таким:

switch (*вираз*) {
case *константа1*:
послідовність команд
break;
case *константа2*:
послідовність команд
break;
case *константа3*:
послідовність команд

break;
...
default:
послідовність команд
}

Елемент *вираз* команди **switch** під час обчислення повинен давати цілочисельне або символьне значення. Вирази, що мають, наприклад, тип з плаваючою крапкою, тут не дозволені. Дуже часто як керівний **switch**-вираз використовується одна змінна.

*Команда **break** завершує виконання коду програми, що визначається командою **switch**.*

Послідовності команд **default**-гілки виконуються у тому випадку, якщо жодна із заданих **case**-констант не співпаде з результатом обчислення **switch**-виразу. Гілка **default** є необов'язковою. Якщо вона відсутня, то при не співпаданні результату розрахунку виразу ні з однією з **case**-констант ніякої дії виконано не буде. Якщо такий збіг все-таки виявиться, то виконуватимуться команди, відповідні цій **case**-гілці, доти, доки не трапиться команда **break** або не буде досягнуто кінець **switch**-команди (або у **default**-, або в останній **case**-гілці).

*Команди **default**-гілки виконуються у тому випадку, якщо жодна з **case**-констант не співпаде з*

*результатом обчислення **switch**-виразу.*

Отже, для застосування **switch**-команди необхідно знати таке:

- команда **switch** відрізняється від команди **if** тим, що **switch**-вираз можна тестувати тільки з використанням умови рівності (тобто на збіг **switch**-виразу із заданими **case**-константами), тоді як умовний **if**-вираз може бути будь-якого типу;
- ніякі дві **case**-константи в одній **switch**-команді не можуть мати однакових значень;
- команда **switch** зазвичай ефективніша, ніж вкладені **if**-команди;
- послідовність команд, пов'язана з кожною **case**-гілкою, не є блоком. Проте повна **switch**-команда визначає блок.

Згідно зі стандартом мови програмування C++, **switch**-конструкція може мати не більше ніж 16 384 **case**-команд. Але на практиці (виходячи з міркувань ефективності) зазвичай обмежуються набагато меншою їх кількістю.

Використання **switch**-команди продемонстровано у наведеному нижче коді програми. Вона створює просту "довідкову" систему, яка описує призначення **for**-, **if**- і **switch**-команд. Після відображення переліку пропонованих тем, згідно з якими можливе надання

довідки, програма переходить в режим очікування доти, доки користувач не зробить свій вибір. Введене користувачем значення використовується в команді **switch** для відображення інформації по вказаній темі.

Приклад *програми імітації роботи "довідкової" системи:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int vybir;
	cout << "Довідка на теми: \n" << endl;
	cout << "1. for" << endl;
	cout << "2. if" << endl;
	cout << "3. switch\n" << endl;
	cout << "Введіть номер теми (1-3): "; cin >> vybir;
	cout << endl;
	switch (vybir) {
	case 1: cout << "for – найуніверсальніший цикл в C++" <<
endl;
		break;
	case 2: cout << "if – команда умовного розгалуження" <<
endl;

	case 3: cout << "switch – багатовибірне розгалуження" <<
endl;
		break;
	default: cout << "програміст повинен ввести число від 1
до 3" << endl;
	}
	_getch(); return 0;
     }
```

Ось один з варіантів виконання цієї програми.

Довідка на теми:
1. **for**
2. **if**
3. **switch**
Введіть номер теми (1-3): 2
if – команда умовного розгалуження.

Формально команда **break** є необов'язковою, хоча здебільшого використання **switch**-конструкцій вона наявна. Команда **break**, що знаходиться в послідовності команд будь-якої **case**-гілки, призводить до виходу зі всієї **switch**-конструкції та передає керування команді, розташованій відразу після неї. Але, якщо команда **break** в **case**-гілці відсутня, то буде виконано всі команди, пов'язані з даною **case**-гілкою, а також всі подальші команди, що розташовані під нею, доти, доки все-таки не трапиться команда **break**, що належить до однієї з подальших **case**-гілок, або не буде досягнуто кінець **switch**-конструкції.

Оператор "знак запитання"

У мові C++ як заміну **if-else**-команд, що вживаються в такому загальному форматі:

if(умова)

змінна = вираз1;
else
змінна = вираз2;

можна використовувати оператор "?". У наведеному записі значення, що присвоюється змінній, залежить від результату обчислення елемента *умова*, що керує командою **if**.

Оператор "?" називається *тернарним,* оскільки він працює з трьома операндами. Ось його загальний формат запису:

***Вираз1 ? Вираз2 : Вираз3*;**

Всі елементи тут є виразами. Зверніть увагу на використання і розташування двокрапки.

Значення ?-виразу визначається таким чином. Обчислюється *Вираз1.*

Якщо він виявляється істинним, то обчислюється *Вираз2*, і результат його обчислення стає значенням всього ?-виразу. Якщо результат обчислення елемента *Вираз1* виявляється помилковим, то значення всього ?-виразу стає результатом обчислення елемента *Вираз3.* Розглянемо такий приклад:

while(something) {
x = *count* > 0 ? 0 : 1;

```
//...
}
```

У цьому записі змінній x присвоюватиметься значення 0 доти, доки значення змінної *count* не стане меншим або дорівнюватиме нулю. Аналогічний програмний код (але з використанням **if-else**-команді) виглядав би так:

```
while(something) {
if(count > 0) x = 0;
else x = 1;
//...
}
```

6.3. Оператори циклу

У більшості задач, що трапляються на практиці програмування, необхідно організувати багатократне виконання деякої дії. Така ділянка обчислювального процесу, що багато разів повторюється, називається *циклом*. Послідовність інструкцій, призначена для багаторазового виконання, називається *тілом циклу*. Одноразове виконання тіла циклу називається *ітерацією*. Вираз, що визначає чи буде вчергове виконуватися ітерація, чи цикл завершиться, називається *умовою виходу* або *умовою завершення циклу* (або умовою продовження в залежності від того,

як інтерпретується його істинність — як ознака необхідності завершення або продовження циклу. Змінна, в якій зберігається номер поточної ітерації, називається *лічильником ітерацій циклу* або *просто лічильником циклу*. Цикл не обов'язково містить лічильник, також лічильник не зобов'язаний бути одним — умова виходу із циклу може залежати від декількох змінюваних в циклі змінних, а може визначатися зовнішніми умовами (наприклад, настанням певного часу), в останньому випадку лічильник взагалі не знадобиться.

Якщо заздалегідь відома кількість необхідних повторень, то цикл називається *арифметичним*. Якщо ж кількість повторень заздалегідь невідома, то говорять про *ітераційний* цикл.

Циклічна команда for

Цикл **for** повторює вказану команду задану кількість разів.

Отже, загальний формат запису циклу **for** для багатократного виконання однієї команди має такий вигляд:

for(*ініціалізація*; *вираз*; *інкремент*) *команда*;

Якщо цикл **for** призначений для багатократного

виконання не однієї команди, а програмного блоку, то його загальний формат має такий вигляд:

for(*ініціалізація*; *вираз*; *інкремент*)
{ *послідовність команд* }

У цьому записі елемент *ініціалізація* є команда присвоєння, яка встановлює *керівній змінній циклу* початкове значення. Ця змінна діє як лічильник, який керує роботою циклу. Елемент *умова* є виразом, у якому тестується значення керівної змінної циклу. Результат цього тестування визначає, виконається цикл **for** ще раз чи ні. Елемент *інкремент* – вираз, який визначає, як змінюється значення керівної змінної циклу після кожної ітерації. Зверніть увагу на те, що всі ці елементи циклу **for** повинні відділятися крапкою з комою. Цикл **for** виконуватиметься доти, доки обчислення елемента *умова* дає істинний результат. Як тільки умова стане хибною, виконання програми продовжиться з команди, що знаходиться наступною за циклом **for**.

Приклад. *Написати програму для виведення на екран чисел від 1 до 100 за допомогою циклу* ***for***.

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
```

```
int main()
{
	int pm;
	for (pm = 1; pm <= 100; pm++) cout << pm << " ";
	_getch(); return 0;
}
```

Керівна змінна циклу **for** може змінюватися як з позитивним, так і з негативним приростом, причому величина цього приросту також може бути будь-якою.

Приклад *програми виведення чисел в діапазоні від 100 до -100 з декрементом, що дорівнює 5:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	for (int i = 100; i >= -100; i -= 5) cout << i << " ";
	_getch(); return 0;
}
```

Важливо розуміти, що умовний вираз завжди тестується на початку виконання циклу **for**. Це означає, що коли перша ж перевірка умови дасть значення ФАЛЬШ, програмний код тіла циклу не виконається жодного разу. Ось приклад:

for(pm=10; pm < 5; pm++)

cout << pm; // Ця команда не виконається.

Цей цикл ніколи не виконається, оскільки вже під час входу в нього значення його керівної змінної

pm більше п'яти. Це робить умовний вираз (pm <5) помилковим із самого початку. Тому навіть одна ітерація цього циклу не буде виконана.

Варіанти використання команди організації циклу for

Для керування циклом **for** можна використовувати декілька змінних. Для розуміння сказаного розглянемо такий фрагмент коду програми:

for(x=0, y=10; x<=10; ++x, --y) **cout** << x << " " << y << **endl**;

У цьому записі комами відокремлюються дві команди ініціалізації та два інкрементні вирази. Це робиться для того, щоби компілятор "розумів", що існує дві команди ініціалізації та дві команди інкремента (декремента).

У мові програмування C++ кома є оператором, який, по суті, означає "зроби це і те". Найчастіше він використовується в циклі **for**. Під час входу у цей цикл ініціалізувалися обидві змінні – x і y. Після виконання кожної ітерації циклу змінна x інкрементується, а змінна y декрементується. Використання декількох керівних змінних у циклі іноді дає змогу спростити алгоритми. У розділах ініціалізації та інкремента циклу **for** можна використовувати будь-яку кількість команд, але

зазвичай їх кількість не перевищує двох.

Умовним виразом, який керує циклом **for**, може бути будь-який допустимий С++-вираз. При цьому він не обов'язково повинен містити керівну змінну циклу. У наведеному нижче прикладі цикл виконуватиметься доти, доки користувач не натисне на клавішу клавіатури. У цьому коді програми представлена ще одна (дуже важлива) бібліотечна функція: **kbhit()**. Вона повертає значення ФАЛЬШ, якщо жодна клавіша не була натиснута на клавіатурі, і значення ІСТИНА – в іншому випадку. Функція чекає натиснення клавіші, даючи змогу тим самим циклу виконуватися доти, доки натискання не відбудеться. Функція **kbhit()** не визначається стандартом мови програмування С++, але включена в розширення мови програмування С++, яке підтримується більшістю компіляторів. Для її використання у програму необхідно внести заголовок **<conio>**.

Приклад *програми з використанням функції* ***kbhit()****:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main() // Виведення чисел на екран до натиснення будь-якої
клавіші.
{
	for (int i = 0; !_kbhit(); i++) cout << i << " ";
	_getch(); return 0;
```

```
}
```

На кожній ітерації циклу викликається функція **kbhit()**. Якщо після запуску програми натиснути на будь-яку клавішу, то ця функція поверне значення ІСТИНА, внаслідок чого вираз !**kbhit()** дасть значення ФАЛЬШ, і цикл зупиниться. Але, якщо не натискати на клавішу, то функція поверне значення ФАЛЬШ, а вираз !**kbhit()** дасть значення ІСТИНА, що дасть змогу циклу продовжувати "крутитися".

Відсутність елементів у визначенні циклу

У мові програмування С++ дозволено опустити будь-який елемент заголовка циклу (ініціалізація, умовний вираз, інкремент) або навіть все відразу.

Приклад *програми на створення циклу, який повинен виконуватися доти, доки з клавіатури не буде введене число 123:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	for (int x = 0; x != 123;)
	{
		cout << "Введіть число: "; cin >> x;
	}
	_getch(); return 0;
```

```
}
```

У цьому коді програми в заголовку циклу **for** відсутній вираз інкремента. Це означає, що під час кожного повторення циклу здійснюється тільки одна дія: значення змінної x порівнюється з числом 123. Але, якщо ввести з клавіатури число 123, умовний вираз, який перевіряється в циклі, стане помилковим, і цикл завершиться. Оскільки вираз інкремента в заголовку циклу **for** відсутній, то керівна змінна циклу не модифікується.

Наведемо ще один варіант організації циклу **for**, у заголовку якого, як це показує такий фрагмент коду програми, відсутній розділ ініціалізації:

```
cout << "Введіть номер позиції: ";
cin >> x;
for(; x < limit; x++) cout << " ";
```

У цьому записі команди організації циклу **for** є порожній розділ ініціалізації, а керівна змінна x ініціалізується значенням, що вводиться користувачем, з клавіатури до входу в цикл.

До розміщення виразу ініціалізації за межами циклу, як правило, вдаються тільки у тому випадку, коли початкове значення генерується складним процесом, який незручно помістити у визначення циклу. Окрім цього, розділ ініціалізації залишають порожнім і у разі, коли керування циклом

здійснюється за допомогою параметра певної функції, а як початкове значення керівної змінної циклу використовується значення, яке отримує параметр під час виклику функції.

Механізм реалізації нескінченного циклу

Нескінченний цикл – цикл, який ніколи не закінчується. Залишивши порожнім умовний вираз циклу **for**, можна створити нескінченний цикл (цикл, який ніколи не закінчується). Спосіб запису такого циклу показаний на прикладі такої конструкції циклу **for**:

```
for(;;) { //... }
```

Цей цикл працюватиме без кінця. Незважаючи на наявність деяких задач програмування (наприклад, командних процесорів операційних систем), які вимагають наявність нескінченного циклу, більшість "нескінченних циклів" – просто цикли із спеціальними вимогами до завершення (це можна здійснити за допомогою команди **break**).

Цикли часової затримки роботи програми

У програмах часто використовують так звані цикли часової затримки. Їх завдання – просто затримати час роботи програми. Для запису таких

циклів достатньо залишити порожнім тіло циклу, тобто опустити ті команди, які повторює цикл на кожній ітерації. Ось приклад:

```
for(int x=0; x<1000; x++);
```

Цей цикл тільки інкрементує значення змінної x і не робить нічого більше. Крапка з комою в кінці рядка необхідна внаслідок того, що цикл **for** чекає отримати команду, яка може бути порожньою (як у цьому випадку).

Ітераційні цикли

У ітераційних циклах здійснюється перевірка деякої умови і, залежно від результату цієї перевірки, відбувається або вихід з циклу, або повторення виконання тіла циклу. Якщо перевірка умови здійснюється перед виконанням блоку операторів, то такий ітераційний цикл називається *циклом з передумовою (цикл "доки")*, а якщо перевірка відбувається після виконання тіла циклу, то це цикл з *після умовою (цикл "до")*.

Особливість цих циклів полягає в тому, що тіло циклу з після умовою завжди виконується хоч би один раз, а тіло циклу з передумовою може жодного разу не виконатися. Залежно від вирішуваного завдання необхідно використовувати той або інший вигляд ітераційних циклів.

Ітераційна команда while

Загальна форма організації циклу **while** має такий вигляд:

while(*вираз*) *команда*;

У цьому записі під елементом *команда* розуміють або одиночну команду, або блок команд. Роботою циклу керує елемент *вираз*, який є будь-яким допустимим С++-виразом. Елемент *команда* здійснюється доти, доки умовний вираз повертає значення ІСТИНА. Як тільки цей вираз стає помилковим, то керування передається команді, яка знаходиться за цим циклом.

Використання циклу **while** можна продемонструвати на прикладі такої невеликої програми. Практично всі компілятори підтримують розширений набір символів, який не обмежується символами ASCII. У розширеному наборі часто містяться спеціальні символи і деякі букви з алфавітів іноземних мов. ASCII-символи використовують значення, що не перевищують число 127, а розширений набір символів – значення з діапазону 128-255.

Приклад. *Написати програму для виведення всіх символів, значення яких лежать в діапазоні 32-255 (32 – код пропуску).*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	unsigned char ch;
	ch = 32;
	while (ch) {
		cout << ch; ch++;
	}
	_getch(); return 0;
	}
```

Тут **while**-вираз складається всього тільки з однієї змінної ch. Оскільки змінна ch має тут тип **unsigned char**, то вона може містити значення від 0 до 255. Якщо її значення дорівнює 255, то після інкрементування воно "скидається" в нуль. Отже, факт рівності значення змінної ch нулю слугує зручним способом завершити **while**-цикл.

Подібно до циклу **for**, умовний вираз перевіряється під час входу в цикл **while**, а це означає, що тіло циклу (при помилковому результаті обчислення умовного виразу) може не виконатися жодного разу. Ця властивість циклу усуває необхідність окремого тестування до початку виконання циклу.

Приклад. *Написати програму, яка виводить рядок, що складається з крапок. Кількість виведених крапок дорівнює значенню, яке вводить користувач.*

```
#include <iostream> // Потокове введення-виведення
#include <clocale>
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int len;
	cout << "Введіть довжину рядка (від 1 до 79): "; cin >>
len;
	while (len > 0 && len < 80) {
		cout << '.'; len--;
	}
	_getch(); return 0;
	}
```

Ця програма не дає змоги виводити рядки, якщо їх довжина перевищує 80 символів. Перевірка на допустимість кількості крапок, що виводяться, здійснюється усередині умовного виразу циклу, а не зовні.

Тіло **while**-циклу може взагалі не містити жодної команди, наприклад:

while(**rand**()!= 100);

Цей цикл здійснюється доти, доки випадкове число, що генерується функцією **rand**(), не виявиться таким, що дорівнює числу 100.

Ітераційна команда do-while

На відміну від циклів **for** і **while**, у яких умова перевіряється під час входу, цикл **do-while** перевіряє умову при виході з циклу. Це означає, що цикл **do-while** завжди здійснюється хоч би один раз. Його загальний формат має такий вигляд:

do {
команди;
} **while**(*вираз*);

Хоча фігурні дужки є необов'язковими, якщо елемент команди складається тільки з однієї команди, то вони часто використовують для поліпшення читабельності конструкції **do-while**, не допускаючи тим самим плутанини з циклом **while**. Цикл **do-while** здійснюється доти, доки залишається істинним елемент вираз, який є умовним виразом.

Приклад *програми, у якій цикл* ***do-while*** *здійснюється доти, доки користувач не введе число 100:*

```
#include <iostream> // Потокове введення-виведення
#include <clocale>
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int n;
```

```
	do {
		cout << "Введіть число (100 – для виходу): "; cin
>> n;
	} while (n != 100);
	_getch(); return 0;
	}
```

Організація вкладених циклів

У програмах, коли це необхідно, один цикл можна вкласти в інший. У мові програмування С++ дозволено використовувати до 256 рівнів вкладення. Вкладені цикли використовуються для вирішення завдань найрізноманітнішого профілю.

Приклад *програми, у якій команда організації вкладеного циклу* ***for*** *дає змогу знайти прості числа в діапазоні від 2 до 1000:*

```
#include <iostream> // Потокове введення-виведення
#include <clocale>
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int j;
	for (int i = 2; i < 1000; i++) {
		for (j = 2; j <= (i / j); j++)
			if (!(i % j)) break; // Якщо число має
множник, значить воно не просте.
		if (j > (i / j)) cout << i << " – просте число"
<< endl;
	}
	_getch(); return 0;
     }
```

Ця програма визначає, чи є простим число, яке міститься в змінній *i*, шляхом послідовного його ділення на значення, розташоване між числом *2* і результатом обчислення виразу *i/j*. Якщо залишок від ділення *i/j* дорівнює нулю, то це означає, що число і не є простим. Але, якщо внутрішній цикл завершиться повністю (без дострокового завершення роботи з використанням команди **break**), то це означає, що поточне значення змінної *i* дійсно є простим числом.

6.4. Оператори переходу

Команда переходу continue

У мові програмування C++ існує засіб "дострокового" виходу з поточної ітерації циклу. Цим засобом є команда **continue**. Вона примусово здійснює перехід до наступної ітерації, опускаючи виконання коду програми, що залишився, в поточній.

Приклад *програми для "прискореного" пошуку парних чисел в діапазоні від 0 до 100 з використанням команди* ***continue:***

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
```

```
for (int x = 0; x <= 100; x++) {
       if (x % 2) continue;
       cout << x << " ";
}
_getch(); return 0;
}
```

У цьому коді програми виводяться тільки парні числа, оскільки внаслідок виявлення непарного числа відбувається передчасний перехід до наступної ітерації, і **cout**-команда опускається.

У циклах **while** і **do-while** команда continue передає керування безпосередньо команді, що перевіряє умовний вираз, після чого циклічний процес триває. А в циклі for після виконання команди continue спочатку обчислюється інкрементний вираз, а потім – умовний. І тільки після цього циклічний процес буде продовжено.

Команда break для виходу з циклу

За допомогою команди **break** можна організувати негайний вихід з циклу, знехтувавши виконанням коду програми, що залишився в його тілі, і перевірку умовного виразу. Завдяки виявленню усередині циклу команди **break** цикл завершується, а керування передається команді, що є наступною після циклу.

Приклад *програми з використанням команди*

break:

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	// Цикл працює для значень t від 0 до 9, а не до 100!
	for (int t = 0; t < 100; t++) {
		if (t == 10) break;
		cout << t << " ";
	}
	_getch(); return 0;
}
```

Ця програма виведе на екран числа від 0 до 9, а не до 100, оскільки настанова **break** при значенні параметра циклу t, що дорівнює 10, забезпечує негайний вихід з циклу.

Команда **break** зазвичай використовується в циклах, у яких при створенні особливих умов необхідно забезпечити негайне їх завершення. Такий фрагмент містить приклад ситуації, коли після натиснення клавіші виконання циклу зупиняється:

for(**int** i=0; i<1000; i++) {

// Виконання якихось дій.

if(kbhit()) **break**;

}

Команда **break** приводить до виходу з самого внутрішнього циклу.

Приклад *програми з використанням команди*

***break** і вкладених циклів:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	for (int t = 0; t < 100; t++) {
		int pm = 1;
		for (;;) {
			cout << pm << " ";
			pm++;
			if (pm == 10) break;
		}
		cout << endl;
	}
	_getch(); return 0;
}
```

Ця програма 100 разів виводить на екран числа від 0 до 9. Під час кожного виконання команди **break** керування передається назад в зовнішній цикл **for**.

Команда goto

Довгі роки команда безумовного переходу **goto** була непопулярною у програмістів, оскільки сприяла, із їхньої точки зору, створенню "спагетті-коду програми". Проте команда **goto**, як і раніше, використовується, а іноді й навіть дуже ефективно. Понад це, у будь-якій ситуації (у області програмування) можна обійтися без команди **goto**, оскільки вона не є елементом, що забезпечує повноту опису мови програмування. Водночас, у певних

ситуаціях її використання може бути дуже корисним.

Команда **goto** вимагає наявності у програмі мітки. *Мітка* – дійсний у мові програмування С++ ідентифікатор, за яким поставлено двокрапку. У процесі виконання команди **goto** керування програмою передається команді, вказаній за допомогою мітки. Мітка повинна знаходитися в одній функції з командою **goto**, яка посилається на цю мітку.

Наприклад, за допомогою команди **goto** і мітки можна організувати такий цикл на 100 ітерацій:

```
x = 1;
loop1: x++;
if(x < 100) goto loopl;
```

Іноді команду **goto** варто використовувати для виходу з глибоко вкладених команд циклу. Для розуміння сказаного розглянемо такий фрагмент коду програми:

```
for(...) {
for(...) {
while(...) {
if(...) goto stop;
...
}
}
}
stop:
```

cout << "Помилка у програмі" << **endl**;

Щоб замінити команду **goto**, довелося б виконати ряд додаткових перевірок. У цьому випадку команда **goto** істотно спрощує програмний код.

Простим застосуванням команди **break** тут не обійшлося, оскільки вона забезпечила б вихід тільки з самого внутрішнього циклу.

Запитання для самоконтролю

1. Які оператори та операції розгалуження існують в С++?
2. Запишіть скорочену форму оператора «якщо».
3. Поясніть механізм роботи оператора вибору.
4. Для чого призначене службове слово default в структурі оператора вибору?
5. Скільки вкладених операторів розгалуження допускається в одному *.cpp-файлі?
6. Для чого використовуються блоки { } в операторах розгалуження?
7. Які оператори порівняння існують в С++? Запишіть їх.
8. Які логічні оператори використовуються для задання умов в операторі розгалуження?
9. Які оператори повторення існують в С++?
10. Опишіть механізм роботи циклу «поки».

11. Як працює і для чого використовується цикл «do while»?
12. Опишіть особливості синтаксису та способів використання циклу «for».
13. Для чого використовується оператор break?
14. У яких випадках використовують оператор continue?
15. Чи можна вкладати в тіло циклу оператор іншого виду циклу?
16. Запишіть «вічний цикл» за допомогою різних операторів повторення.

РОЗДІЛ 7. ФУНКЦІЇ

7.1. Основні поняття про функції

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

Будь-яка С++-програма складається з "будівельних блоків", що називаються *функціями.*

Функція – підпрограма, яка містить одну або декілька С++-команд і здійснює одну або декілька задач. Хороший стиль програмування мовою С++ передбачає, що кожна функція виконує тільки одну задачу, наприклад, окремо введення та виведення даних.

Кожна функція має ім'я, яке використовують для її виклику. Своїм функціям програміст може давати будь-які імена за винятком імені **main**(), зарезервованого для функції, з якої починається виконання програми.

У мові програмування С++ жодна функція не може бути вбудована в іншу. На відміну від деяких мов програмування, які дають змогу використовувати вкладені функції, у мові програмування С++ всі функції розглядаються як окремі компоненти.

Як було зазначено вище, функція **main**() –

перша функція, яка виконується під час запуску програми. Її повинна містити кожна С++-програма. Взагалі, функції, які належить використовувати, бувають двох типів. До першого типу належать функції, написані програмістом (**main()** – приклад функції такого типу). Функції іншого типу знаходяться в *стандартній бібліотеці* С++-компілятора.

Наведена нижче програма містить дві функції: **main**() і FunC().

Приклад *програми, що містить дві функції:* ***main()*** *і FunC():*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
void FunC(); // Попереднє оголошення прототипу функції FunC()
int main()
{
	setlocale(LC_CTYPE, "ukr");
	cout << "У функції main().";
	FunC(); // Викликаємо функцію FunC().
	cout << "Знову у функції main().";
		_getch(); return 0;
}
void FunC() // Визначення функції FunC()
{
	cout << " У функції FunC(). ";
}
```

Програма працює таким чином. Спочатку викликається функція **main**() і здійснюється її перша

cout-команда. Потім з функції **main**() викликається функція FunC(). Зверніть увагу на те, як цей виклик реалізується у програмі: вказується ім'я функції FunC, за яким стоїть пара круглих дужок і крапка з комою. Виклик будь-якої функції є C++-командою і тому повинен завершуватися крапкою з комою. Потім функція FunC() здійснює свою єдину **cout**-команду і передає керування назад функції **main**(), причому тому рядку коду програми, який розташований безпосередньо за викликом функції. Нарешті, функція **main**() здійснює свою другу **cout**-команду, яка завершує всю програму. Отже, на екрані ми повинні побачити такі результати:

У функції **main**().

У функції FunC().

Знову у функції **main**().

У цьому коді програми необхідно розглянути таку команду:

void FunC(); // Попереднє оголошення прототипу функції FunC()

Як зазначено в коментарі, це – *прототип оголошення функції* FunC(). Прототип функції оголошує функцію до її визначення. Прототип дає змогу компіляторові дізнатися тип значення, що повертається цією функцією, а також кількість і тип параметрів, які вона може мати. Компіляторові

потрібно знати цю інформацію до першого виклику функції. Тому прототип розташовується ще до функції **main**(). Єдиною функцією, яка не вимагає прототипу, є **main**(), оскільки вона є вбудованою у мові програмування С++.

Загальний формат визначення С++-функцій

У попередніх прикладах було показано конкретний тип функції. Проте всі С++-функції мають такий загальний формат їх визначення:

тип_поверненого_значення ім'я_функції (*перелік_параметрів*)
{
. // тіло функції;
}

Розглянемо детально всі елементи, з яких складається функція. За допомогою елемента *тип_поверненого_значення* вказується тип значення, що повертається функцією. Як буде показано далі, це може бути практично будь-який тип, в т.ч. тип, що створюється безпосередньо програмістом.

Якщо функція не повертає ніякого значення, треба вказати тип **void**. Якщо функція дійсно повертає значення, воно повинно мати тип, сумісний з вказаним

у визначенні функції.

Кожна функція має ім'я. Воно, як неважко здогадатися, задається елементом *ім'я функції.* Після імені функції поміж круглих дужок вказують перелік параметрів, який є послідовністю пар (складаються з типу даних та імені), розділених між собою комами. Якщо функція не має параметрів, елемент *перелік_параметрів* відсутній, тобто круглі дужки залишаються порожніми.

У фігурні дужки поміщено тіло функції. Тіло функції становлять C++-команди, які визначають конкретні дії функції. Функція завершується (і керування передається процедурі, яка її викликає), досягши закритої фігурної дужки або команди **return.**

Передавання аргументів функції

Функції можна передати одне або декілька значень. Значення, що передається функції, називають *аргументом.* Хоча у програмах, які ми розглядали дотепер, жодна з функцій (ні **main**(), ні FunC()) не отримувала ніяких значень, функції у мові програмування C++ можуть приймати один або декілька аргументів. Верхня межа кількості аргументів, що приймаються, визначається конкретним компілятором. Згідно зі стандартом мови програмування C++, він дорівнює 256.

Аргумент *— значення, що передається функції під час її виклику.*

Параметр *— визначена функцією змінна, яка приймає аргумент, що передається функції.*

Під час розроблення функції, яка приймає один або декілька аргументів, іноді необхідно оголосити змінні, які зберігатимуть значення аргументів. Ці змінні називають *параметрами* функції.

Приклад *програми на використання користувацької функції funZ():*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Для консольного режиму роботи
using namespace std; // Використання стандартного простору імен
void funZ(int x, int y); // Попереднє оголошення прототипу
функції funZ()
int main()
{
	funZ(10, 20);
	funZ(5, 6);
	funZ(8, 9);
	_getch(); return 0;
}
void funZ(int x, int y) // Визначення функції funZ()
{
	cout << x * y << " ";
}
```

Ця програма виведе на екран числа 200, 30 і 72. Під час виклику функції funZ() C++-компілятор копіює значення кожного аргумента у відповідний параметр. У цьому випадку під час першого виклику

функції funZ() число 10 копіюється в змінну x, а число 20 – в змінну y. Під час другого виклику 5 копіюється в x, а 6 – в y. Під час третього виклику 8 копіюється в x, а 9 – в y.

Якщо C++-функції мають два або більше аргументів, то вони розділяються між собою комами. Для розглянутої вище функції funZ() перелік аргументів виражений у вигляді x, y.

Повернення функціями аргументів

У мові програмування C++ багато бібліотечних функцій повертають значення. Наприклад, функція **abs**() повертає абсолютне цілочисельне значення свого аргументу. Функції, які написано програмістом, також можуть повертати значення. У мові програмування C++ для повернення значення використовують команду **return**. Загальний формат цієї команди є таким:

return *значення*;

Неважко здогадатися, що тут елемент *значення* є значенням, що повертається функцією.

Щоб продемонструвати механізм повернення функціями значень, переробимо попередню програму так, як це показано далі. У цій версії функція funZ()

повертає добуток своїх аргументів.

Приклад *програми на реалізацію механізму повернення функціями значень:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Для консольного режиму роботи
using namespace std; // Використання стандартного простору імен
int funZ(int x, int y); // Попереднє оголошення прототипу
функції funZ()
int main()
{
	int rezult;
	rezult = funZ(10, 11); // Присвоєння значення, що
повертається функцією.
	cout << "x*y = " << rezult;
	_getch(); return 0;
}
// Ця функція повертає значення
int funZ(int x, int y) // Визначення функції funZ()
{
	return x * y; // Функція повертає добуток x і y
}
```

У наведеному прикладі функція funZ() повертає результат обчислення виразу x * y за допомогою команди **return**. Потім значення цього результату присвоюється змінній rezult. Таким чином, значення, що повертається командою **return**, стає значенням функції funZ() у програмі, яка її викликає, а присвоюється змінній rezult.

У попередніх версіях мови програмування C++ для типів значень, що повертаються функціями, існувала домовленість, що діє за замовчуванням.

Якщо тип значення, що повертається функцією, не вказано, то передбачалося, що ця функція повертає цілочисельне значення. Наприклад,

funZ(**int** x, **int** y) // За замовчуванням тип значення,

// що повертається функцією, використовується тип **int**.

{

return x * y; // Функція повертає добуток x і y.

}

Досягши команди **return,** функція негайно завершується, а увесь решта програмний код ігнорується. Функція може містити декілька команд **return**.

Повернення з функції можна забезпечити за допомогою команди **return** без вказання значення, що повертається, але таку її форму допустимо застосовувати тільки для функцій, які не повертають ніяких значень і оголошені з використанням ключового слова **void**.

Спеціальна функція main()

На відміну від деяких інших мов програмування, у яких виконання завжди починається "зверху", тобто з першого рядка коду програми, кожна C++-програма

завжди починається з виклику основної функції **main**() незалежно від її розташування у програмі.

У програмі може бути тільки одна функція **main**(). Оскільки функція **main**() вбудована у мову програмування С++, то вона не вимагає прототипу.

7.2. Правила дії областей видимості функцій

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

Існує три види змінних: локальні змінні, формальні параметри і глобальні змінні. Тут ми розглянемо правила дії областей видимості з огляду на використання функцій.

Локальні змінні

Як зазначалося вище, змінні, які оголошено всередині функції, називаються *локальними.* Але у мові програмування С++ змінні можуть бути внесені в блоки. Це означає, що змінну можна оголосити усередині будь-якого блоку коду програми, після чого вона стане локальною змінною стосовно цього блоку.

Найпоширенішим програмним блоком є функція. У мові програмування С++ кожна функція визначає блок коду програми. Тіло функції надійно приховане

від решти частини програми і, якщо у функції не використовуються глобальні змінні, то вона не може зробити ніякого впливу на інші частини програми, однаково, як і ті на неї.

Таким чином, вміст однієї функції зовсім незалежний від вмісту іншої. Змінні, які оголошено в одній функції, не роблять ніякого впливу на змінні, які оголошено в іншій, причому навіть у тому випадку, якщо ці змінні мають однакові імена. Розглянемо, наприклад, таку програму:

Приклад *програми на реалізацію механізму використання області видимості локальних змінних:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
void Fun();
int main()
{
	setlocale(LC_CTYPE, "ukr");
	char str[] = "Це масив str у функції main().";
	cout << str << endl;
	Fun();
	cout << str << endl;
	_getch(); return 0;
}
void Fun()
{
	char str[80];
	cout << "Введіть будь-який рядок: "; cin >> str;
	cout << str << endl;
}
```

Наприклад, неможливо використовувати команду **goto** для переходу в середину коду іншої функції.

Символьний масив str оголошується тут двічі: перший раз у функції **main**() і ще раз у функції Fun(). При цьому масив str, оголошений у функції **main**(), не має жодного стосунку до однойменного масиву з функції Fun(). Як пояснювалося вище, кожен масив (у цьому випадку str) відомий тільки блоку коду програми, у якому він оголошений. Щоб переконатися у цьому, достатньо виконати наведену вище програму. Як бачите, хоча масив str отримує рядок, що вводиться користувачем у процесі виконання функції Fun(), вміст масиву str у функції **main**() залишається незмінним.

Мова C++ містить ключове слово **auto**, яке можна використовувати для оголошення локальних змінних. Але оскільки всі не глобальні змінні є за замовчуванням **auto**-змінними, то до цього ключового слова практично ніколи не вдаються. Розміщувати його потрібно безпосередньо перед типом змінної:

auto char ch;

Якщо ім'я змінної, оголошеної у внутрішньому блоці, збігається з іменем змінної, оголошеної в зовнішньому блоці, то "внутрішня" змінна перевизначає "зовнішню" у межах області видимості внутрішнього блоку. Розглянемо такий приклад.

Приклад *програми на перевизначення "зовнішньої" змінної на "внутрішню":*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int c = 10, d = 100;
	if (d > 0) {
		int c; // Ця змінна с відокремлена від зовнішньої
змінної с.
		c = d / 2;
		cout << "Внутрішня змінна с: " << c << endl;
	}
	cout << "Зовнішня змінна с: " << c << endl;
	_getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми.

Внутрішня змінна с: 50

Зовнішня змінна с: 10

Тут змінна с, оголошена усередині **if**-блоку, перевизначає або приховує зовнішню змінну с. Зміни, яким піддалася внутрішня змінна с, не роблять ніякого впливу на зовнішню змінну с. Понад це, поза **if**-блоком внутрішня змінна с більше не існує, і тому зовнішня змінна с знову стає видимою.

Локальні змінні не зберігають своїх значень між викликами функцій.

Локальні змінні не зберігають своїх значень між активізаціями.

Формальні параметри

Як уже зазначалося вище, якщо функція використовує аргументи, то вона повинна оголосити змінні, які прийматимуть значення цих аргументів. Ці змінні називаються *формальними параметрами* функції. Якщо не рахувати отримання значень аргументів під час виклику функції, то поведінка формальних параметрів нічим не відрізняється від поведінки будь-яких інших локальних змінних усередині функції. Область видимості параметра обмежується рамками його функції. Програміст повинен гарантувати, що тип оголошуваних ним формальних параметрів збігається з типом аргументів, що передаються функції.

Глобальні змінні

Глобальні змінні у багатьох аспектах протилежні до локальних. Вони відомі впродовж всієї програми, їх можна використовувати в будь-якому її місці, і вони зберігають свої значення у процесі виконання всього коду програми. Отже, їх область видимості розширюється до обсягу всієї програми.

Глобальна змінна створюється шляхом її

оголошення поза будь-якою функцією. Завдяки їх глобальності доступ до цих змінних можна отримати з будь-якого виразу, незалежно від функції, у якій цей вираз знаходиться.

Якщо глобальна і локальна змінні мають однакові імена, то перевага знаходиться на стороні локальної змінної. Іншими словами, ***локальна змінна приховає глобальну з таким самим іменем***. Таким чином, незважаючи на те, що до глобальної змінної теоретично можна отримати доступ з будь-якого коду програми, практично це можливо тільки у випадку, якщо однойменна локальна змінна не перевизначить глобальну.

Приклад *програми-тренажера з виконання операції додавання:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
void drill(); // Попереднє оголошення функції
int k; // Змінні k і num_right – глобальні
int num_right;
int main()
{
	setlocale(LC_CTYPE, "ukr");
	cout << "Скільки практичних вправ: "; cin >> k;
	num_right = 0;
	do {
		drill();
		k--;
	} while (k);
```

```
	cout << "Ви дали " << num_right << " правильних
відповідей" << endl;
	_getch(); return 0;
}
void drill() // Визначення функції
{
	int k; /* Ця змінна локальна і ніяк не пов'язана з
однойменною глобальною. */
	int a, b, ans;
	// Генеруємо два числа між 0 і 99.
	a = rand() % 100;
	b = rand() % 100;
	// Користувач отримує три спроби дати правильну
// відповідь.
	for (k = 0; k < 3; k++) {
		cout << "Скільки буде " << a << " + " << b << "?
"; cin >> ans;
		if (ans == a + b) {
			cout << "Правильно" << endl;
			num_right++;
			return;
		}
	}
	cout << "Ви використовували всі свої спроби" << endl;
	cout << "Відповідь дорівнює " << a + b << endl;
     }
```

Проте без особливої потреби необхідно уникати використання глобальних змінних, і на це є три причини:

– вони займають пам'ять протягом всієї тривалості виконання програми, а не тільки тоді, коли дійсно вони необхідні;
– використання глобальної змінної в "ролі", з якою легко б "справилася" локальна змінна, робить таку функцію менш універсальною, оскільки вона покладається на потребу визначення даних поза

цією функцією;

– використання великої кількості глобальних змінних може призвести до появи помилок в роботі програми.

Основна проблема, характерна для розроблення великих С++-програм, – випадкове модифікування значення змінної у якомусь іншому місці програми. Чим більше глобальних змінних у програмі, тим більшою є ймовірність помилки.

7.3. Механізм використання команди return у функціях

Команда **return** здійснює дві важливі операції. По-перше, вона забезпечує негайне повернення керування до ініціатора виклику функції. По-друге, її можна використовувати для передачі значення, що повертається функцією.

Завершення роботи функції

Як зазначалося вище, керування від функції передається ініціатору її виклику в двох ситуаціях: або внаслідок виявлення фігурної дужки, що закривається, або у процесі виконання команди **return**. Команду **return** можна використовувати з певним заданим значенням або без нього. Але, якщо в оголошенні

функції вказано тип значення (тобто не тип **void**), що повертається, то функція повинна повертати значення цього типу. Тільки **void**-функції можуть використовувати команду **return** без будь-якого значення.

Для **void**-функцій команда **return** в основному використовується як елемент програмного керування. Наприклад, у наведеній нижче функції виводиться результат зведення числа в позитивний цілочисельний степінь. Якщо ж показник степеня виявиться від'ємним, команда **return** забезпечить вихід з функції, перш ніж буде зроблена спроба обчислити такий вираз. У цьому випадку команда **return** діє як керівний елемент, тобто запобігає небажаному виконанню певної частини функції.

```
void power(int base, int exp)
{
if(exp<0) return; /* Щоб не допустити зведення числа у від'ємний степінь, тут здійснюється повернення у функцію, яка викликає, та ігнорується решта частини функції. */
int c = 1;
for(; exp; exp--) c = base * c;
cout << "Результат дорівнює: " << c;
}
```

Функція може містити декілька команд **return**.

Функція буде завершена у процесі виконання хоч би одного з них. Наприклад, такий фрагмент коду програми є абсолютно правомірним.

```
void Fun()
{
//...
switch(c) {
case 'a': return;
case 'b': //...
case 'c': return;
}
if(count<100) return; //...
}
```

Проте необхідно мати на увазі, що дуже велика кількість команд **return** може погіршити ясність алгоритму і ввести в оману тих, хто буде у ньому розбиратися. Декілька команд **return** варто використовувати тільки у тому випадку, якщо вони сприяють ясності функції.

Повернення значень з функції

Кожна функція, окрім типу **void**, повертає яке-небудь значення. Це значення безпосередньо задається за допомогою команди **return**. Іншими словами, будь-яку не **void**-функцію можна використовувати як операнд у виразі.

Отже, кожний з наступних виразів допустимо у мові програмування С++:

x = power(y);

if(max(x, y)) > 100) **cout** << "більше";

switch(fabs(x)) {

Незважаючи на те, що всі не **void**-функції повертають значення, вони необов'язково мають бути використані у програмі. Якщо значення, що повертається функцією, не бере участі в операції присвоєння, воно просто відкидається (втрачається).

Розглянемо наведену нижче програму, у якій використовується стандартна бібліотечна функція **abs**().

Приклад *програми на використання стандартної бібліотечної функції* ***abs()****:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
	int c = abs(-10); // рядок 1
	cout << abs(-23); // рядок 2
	abs(100); // рядок 3
	_getch(); return 0;
}
```

Якщо функція, тип якої є відмінним від типу **void**, завершується внаслідок виявлення фігурної дужки, що закривається, то значення, яке вона повертає, не

визначене (тобто невідоме). Через особливості формального синтаксису C++ не **void**-функція не зобов'язана виконувати команду **return**. Це може відбутися у тому випадку, якщо кінець функції буде досягнуто до виявлення команди **return**. Але, оскільки функція оголошена як така, що повертає значення, значення буде таки повернено, навіть якщо це просто "сміття".

У загальному випадку не кожна зі створюваних **void**-функцій повинна повертати значення за допомогою безпосередньо виконуваної команди **return**.

Вище згадувалося, що **void**-функція може мати декілька команд **return**.

Те саме стосується і функцій, які повертають значення.

Прототипи функцій

У мові програмування C++ всі функції мають бути оголошені до їх використання. Переважно це реалізується за допомогою прототипу функції.

Прототипи містять три види інформації про функцію: тип значення, що повертається нею; тип її параметрів; кількість параметрів.

Прототипи дають змогу компіляторові виконати такі три операції:

- вони повідомляють компілятор, програмний код якого типу необхідно генерувати під час виклику функції. Відмінності в типах параметрів і значенні, що повертається функцією, забезпечують різне оброблення компілятором;
- вони дають змогу C++ виявити неприпустимі перетворення типів аргументів, що використовуються під час виклику функції, в тип, який було вказано в оголошенні її параметрів, і повідомити про них;
- вони дають змогу компіляторові виявити відмінності між кількістю аргументів, що використовуються під час виклику функції, і кількістю параметрів, заданих у визначенні функції.

Загальна форма прототипу функції аналогічна її визначенню за винятком того, що в прототипі не представлено тіла функції.

type func_name (*type parm_name1, type parm_name2, ..., type parm_nameN*);

Використання імен параметрів у прототипі необов'язкове, але дає змогу компіляторові ідентифікувати будь-який збіг типів під час виникнення помилки, тому краще імена параметрів все ж таки помістити в прототип функції.

Способи передачі аргументів функціям

Щоб зрозуміти походження посилання, необхідно знати теорію процесу передачі аргументів функціям. У загальному випадку в мовах програмування, як правило, передбачається два способи, які дають змогу передавати аргументи в підпрограми (функції, методи, процедури). Перший називається *викликом за значенням* (call-by-value). У цьому випадку значення аргументу копіюється у формальний параметр підпрограми. Значить зміни, внесені в параметри підпрограми, не впливають на аргументи, що використовуються під час її виклику.

Під час виклику функції за значенням передається значення аргументу.

Другий спосіб передачі аргументу підпрограмі називається *викликом за посиланням* (call-by-reference). У цьому випадку в параметр копіюється адреса аргументу (а не його значення). У межах підпрограми, що викликається, цю адресу використовують для доступу до реального аргументу, що задається під час її виклику. Це означає, що зміни, внесені в параметр, нададуть дію на аргумент, що використовується під час виклику підпрограми.

Під час виклику функції з використанням механізму посилання зразу ж передається адреса аргументу.

7.4. Організація рекурсивних функцій

Рекурсія, яку іноді називають циклічним визначенням, є процес визначення чого-небудь на власній основі. В області програмування під рекурсією розуміють процес виклику функцією самої себе. Функцію, яка викликає саму себе, називають рекурсивною.

Класичним прикладом рекурсії є обчислення факторіалу від числа за допомогою користувацької функції factr(). Факторіал числа *N* є добуток всіх цілих чисел від 1 до *N*. Наприклад, факторіал числа 3 дорівнює 1·2·3, або 6. Рекурсивний спосіб обчислення факторіалу від числа продемонстровано у наведеному нижче коді програми. Для порівняння сюди ж включений і його нерекурсивний (ітеративний) еквівалент.

Приклад *програми обчислення факторіалу рекурсивним способом:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int factr(int n), fact(int n);
int main()
{
```

```
	setlocale(LC_CTYPE, "ukr");
	// Використання рекурсивної версії.
	cout << "Факторіал числа 5 дорівнює " << factr(5) <<
endl;
	// Використання ітеративної версії.
	cout << "Факторіал числа 4 дорівнює " << fact(4) <<
endl;
	_getch(); return 0;
}
int factr(int n) // Рекурсивна версія функції.
{
	int rezult;
	if (n == 1) return (1);
	rezult = factr(n - 1) * n;
	return (rezult);
}
int fact(int n) // Ітеративна версія функції.
{
	int rezult = 1;
	for (int t = 1; t <= n; t++) rezult = rezult * (t);
	return (rezult);
     }
```

Нерекурсивна версія функції fact() достатньо проста і не вимагає розширених пояснень. У ній використовується цикл, у якому організовано множення послідовних чисел, починаючи з 1 і закінчуючи числом, заданим як параметр: на кожній ітерації циклу поточне значення керованої змінної циклу множиться на поточне значення добутку, отримане внаслідок виконання попередньої ітерації циклу.

Рекурсивна функція factr() є дещо складнішою. Якщо вона викликається з аргументом, що дорівнює 1, то відразу повертає значення 1. В іншому випадку

вона повертає добуток factr(n-1)*n. Для обчислення цього виразу викликається метод factr() з аргументом n-1. Цей процес повторюється доти, доки аргумент не стане таким, що дорівнює 1, після чого викликані раніше методи почнуть повертати значення. Наприклад, під час обчислення факторіалу від числа 2 перше звернення до методу factr() приведе до другого звернення до того ж методу, але з аргументом, що дорівнює 1. Другий виклик методу factr() поверне значення 1, яке буде помножене на 2 (початкове значення параметра n).

Основна перевага рекурсії полягає у тому, що деякі типи алгоритмів рекурсивно реалізуються простіше, ніж їх ітеративні еквіваленти. Наприклад, алгоритм сортування Quicksort достатньо важко реалізувати ітеративним способом. Окрім цього, деякі завдання (особливо ті, які пов'язані з штучним інтелектом) просто створені для рекурсивних вирішень. Нарешті, у деяких програмістів процес мислення організований так, що їм простіше думати рекурсивно, ніж ітеративно.

Під час написання рекурсивної функції необхідно включити в неї команду перевірки умови (наприклад, **if**-команду), яка б забезпечувала вихід з функції без виконання рекурсивного виклику. Якщо цього не зробити, то, викликавши одного разу таку функцію, з

неї вже не можна буде повернутися. Під час роботи з рекурсією це найпоширеніший тип помилки. Тому у процесі розроблення програм з рекурсивними функціями не варто ігнорувати команду **cout**, щоб бути в курсі того, що відбувається в конкретній функції, і мати можливість перервати її роботу у разі виявлення помилки.

7.5. Перевизначення функцій

Одна з найдивовижніших можливостей мови програмування С++ – перевизначення функцій. У мові С++ декілька функцій можуть мати однакові імена, але за умови, що їх параметри будуть різними. Таку особливість мови програмування С++ називають *перевизначенням функцій* (function overloading), а імена функцій, які в ній задіяно, перевизначеними (overloaded) функціями. Перевизначення функцій – один із способів реалізації поліморфізму у мові програмування С++.

Перевизначення функцій *– механізм програмування, який дає змогу двом спорідненим функціям мати однакові імена.*

Приклад *програми на "триразове" перевизначення функції Fun():*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
void Fun(int c); // Один цілочисельний параметр
void Fun(int c, int d); // Два цілочисельні параметри
void Fun(double f); // Один параметр типу double
int main()
{
	setlocale(LC_CTYPE, "ukr");
	Fun(10); // Виклик функції Fun(int)
	Fun(10, 20); // Виклик функції Fun(int, int)
	Fun(12.23); // Виклик функції Fun(double)
	_getch(); return 0;
}
void Fun(int c)
{
	cout << "У функції Fun(int) індекс c дорівнює " << c <<
endl;
}
void Fun(int c, int d)
{
	cout << "У функції Fun(int, int) індекс c дорівнює " <<
c;
	cout << ", d дорівнює " << d << endl;
}
void Fun(double f)
{
	cout << "У функції Fun(double) індекс f дорівнює " << f
<< endl;
}
```

Внаслідок виконання ця програма відображає такі результати:

У функції Fun(**int**) індекс c дорівнює 10

У функції Fun(**int**, **int**) індекс c дорівнює 10, d дорівнює 20

У функції Fun(**double**) індекс f дорівнює 12.23

Як бачимо, функція Fun() перевантажується три рази. Перша версія приймає один цілочисельний параметр, друга – два цілочисельні параметри, а третя – один **double**-параметр. Оскільки переліки параметрів для всіх трьох версій функцій є різними, то компілятор володіє достатньою інформацією, щоб викликати правильну версію кожної функції. У загальному випадку для створення перевизначення певної функції достатньо оголосити різні її версії.

7.6. Передача аргументів функції за замовчуванням

У мові програмування C++ ми можемо надати параметру функції певне значення, яке буде автоматично використане, якщо під час виклику функції не задається аргумент, що відповідає цьому параметру. Аргументи, що передаються функції за замовчуванням, можна використовувати для спрощення звернення до складних функцій, а також як "скорочену форму" перевизначення функцій.

Завдання аргументів, що передаються функції за замовчуванням, синтаксично є аналогічним ініціалізації змінних. Розглянемо наведений нижче приклад, у якому оголошується функція myFunc(), що приймає один аргумент типу **double** з діючим за замовчуванням значенням 0.0 і один символьний

аргумент з діючим за замовчуванням значенням 'X'.

void myFunc(**double** num = 0.0, **char** ch = 'X')

{. . .}

Після такого оголошення функцію myFunc() можна викликати одним з трьох таких способів:

myFunc(198.234, 'A'); // Передаємо безпосередньо задані значення.

myFunc(10.1); // Передаємо для параметра num значення 10.1,

// а для параметра ch дозволяємо застосувати

// аргумент, що задається за замовчуванням ('X').

myFunc(); // Для обох параметрів num і ch дозволяємо застосувати

// аргументи, що задаються за замовчуванням.

Під час першого виклику параметру num передається значення 198.234, а параметру ch – символ 'A'. Під час другого виклику параметру num передається значення 10.1, а параметру ch за замовчуванням встановлюється значення, що дорівнює символу 'X'. Нарешті, внаслідок третього виклику як параметр num, так і параметр ch за замовчуванням встановлюються такими значення, що задаються в оголошенні функції.

Аргумент, що передається функції за замовчуванням, є значенням, яке буде автоматично передано параметру функції у випадку, якщо

аргумент, що відповідає цьому параметру, безпосередньо не задано.

Важливо розуміти, що всі параметри, які приймають значення за замовчуванням, мають бути розташовані праворуч від інших. Наприклад, у наведеному нижче прототипі функції міститься помилка:

void Fun(**int** a = 1, **int** b); // Неправильно!

Якщо Ви почали визначати параметри, які приймають значення за замовчуванням, то не можна після них вказувати параметри, що задаються під час виклику функції тільки безпосередньо. Тому наведене нижче оголошення є також неправильним і тому воно не буде скомпільовано:

int myFunc(**float** f, **char** *str, **int** c=10, **int** d); // Неправильно!

Оскільки для параметра c визначено значення за замовчуванням, то для параметра d також потрібно задати значення за замовчуванням.

Приклад *програми на виникнення неоднозначності при передачі аргументів функції за замовчуванням:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int myFunc(int c) { return c; }
int myFunc(int c, int d = 1) { return c * d; }
```

```
int main()
{
	cout << myFunc(4, 5) << " "; // Неоднозначності немає
	cout << myFunc(10); // Виникнення неоднозначності
	_getch(); return 0;
}
```

У цьому коді програми в першому зверненні до функції myFunc() задається два аргументи, тому у компілятора немає ніяких сумнівів у виборі потрібної функції, а саме myFunc(**int** c, **int** d), тобто ніякої неоднозначності у цьому випадку не виникає. Але під час другого звернення до функції myFunc() ми отримуємо неоднозначність, оскільки компілятор "не знає", чи то йому викликати версію функції myFunc(), яка приймає один аргумент, чи то використовувати можливість передачі аргументу функції за замовчуванням до версії, яка приймає два аргументи.

Запитання для самоконтролю

1. У якому місці програми можна оголошувати функції?
2. Що таке прототип функції?
3. Яким чином використовують аргументи за замовчуванням? Як їх задати?
4. Скільки параметрів може приймати функція?
5. Як забезпечити повернення результату функції?

6. Як описати функцію, що не повертає результату?
7. Як працює механізм перевантаження функцій?
8. Які змінні називають глобальними, які локальними?
9. Який час життя локальних та глобальних змінних?
10. Якими значеннями локальні та глобальні змінні ініціалізуються системою?
11. Які класи пам’яті вам відомі?

РОЗДІЛ 8. ПОКАЖЧИКИ

Покажчики (вказівники), поза сумнівом, – один з найважливіших і складних аспектів мови програмування С++. Значною мірою потужність багатьох засобів мови програмування С++ визначається використанням покажчиків. Наприклад, завдяки їм забезпечується підтримка зв'язних списків і динамічного виділення області пам'яті, саме вони дають змогу функціям змінювати значення своїх аргументів.

8.1. Основні поняття про покажчики

Покажчики – змінні, які зберігають адреси іншої змінної. Найчастіше ці адреси позначають місцезнаходження в пам'яті інших змінних. Наприклад, якщо змінна x містить адресу змінної y, то про змінну x говорять, що вона "вказує" на y.

Змінні-покажчики (або *змінні типу покажчика*) мають бути відповідно оголошені. Формат оголошення змінної-покажчика є таким:

тип *******ім'я_змінної*;

У цьому записі елемент *тип* означає базовий

тип покажчика, причому він повинен бути допустимим C++-типом. Елемент *ім'я_змінної* є іменем змінної-покажчика.

Наприклад. Щоб оголосити змінну p покажчиком на **int**-значення, використовується така команда:

int *p;

Для оголошення покажчика на **float**-значення потрібно використати таку команду:

float *p;

У загальному випадку використання символу "зірочка" (*) перед іменем змінної в команді оголошення перетворює цю змінну на покажчик.

Тип даних, на які посилатиметься покажчик, визначається його базовим типом. Розглянемо ще один приклад:

int *ip; // Покажчик на цілочисельне значення

double *dp; // Покажчик на значення типу **double.**

Як зазначено в коментарях, змінна ip – покажчик на **int**-значення, оскільки його базовим типом є тип **int**, а змінна dp – покажчик на **double**-значення, оскільки його базовим типом є тип **double**. Отже, в попередніх прикладах змінну ip можна використовувати для вказівки на **int**-значення, а змінну dp – на **double**-значення. Проте слід пам'ятати:

не існує реального засобу, який би зміг перешкодити покажчику посилатися на "казна-що". Ось через це покажчики потенційно небезпечні.

Приклад *програми на ввикористання покажчиків:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
int main()
{
    char* pc, c='t';
    double* pf, f=9.2, f2 ;
    pc = &c; pf = &f;
    f2 = *pf;
    cout << f2 << endl;
    f = 3.6;
    *pf = *pf*2;
    cout << *pc << " " << *pf << endl;
    _getch();
    return 0;
}
```

Запитання для самоконтролю

1. Що таке вказівник?
2. Які з оголошених змінних є вказівниками?
   int sum, x1, *y2, k;
   float *a, b, *m1;
3. Які значення можна присвоювати вказівнику?
4. Що позначає константа NULL? До яких вказівників її можна застосовувати?
5. Яким буде значення змінної x після виконання наступного фрагмента програми?

```
int x, *p;
p=&x;  x=5;
*p=x*5; x+=*p;
```

РОЗДІЛ 9. МАСИВИ

Масив (array) – перелік змінних однакового типу, звернення до яких відбувається із застосуванням імені, загального для всіх його елементів. У мові програмування C++ масиви можуть бути одно-, дво- та багатовимірними, хоча в основному використовуються одновимірні масиви. Масиви є зручними засобами для групування взаємопов'язаних між собою змінних.

9.1. Одновимірні масиви

Одновимірний масив – перелік взаємопов'язаних між собою змінних. Для оголошення одновимірного масиву використовують така форма запису:

тип ім'я_масиву[розмір];

У цьому записі за допомогою елемента запису *тип* оголошується базовий тип масиву. *Базовий тип* визначає тип даних кожного елемента, з яких складається масив. Кількість елементів, які зберігатимуться в масиві, визначається елементом *розмір.* Наприклад, у процесі виконання наведеної нижче команди оголошується **int**-масив (що

складається з 10 елементів) з іменем Array:

int Array[10];

Індекс у прямокутних дужках після імені масиву вказує на конкретний елемент масиву. Доступ до окремого елемента масиву здійснюється за допомогою індексу, який описує позицію елемента усередині масиву. У мові програмування С++ перший елемент масиву має нульовий індекс. Оскільки масив Array містить 10 елементів, то його індекси змінюються від 0 до 9. Щоб отримати доступ до елемента масиву за індексом, достатньо вказати потрібний номер елемента в квадратних дужках. Так, наприклад, першим елементом массиву Array є Array[0], а останнім – Array[9].

int Array[10]; // Ця команда резервує область пам'яті для 10 елементів типу **int**.

// Поміщаємо в масив значення.

for(**int** t=0; t<10; ++t) Array[t] = (t+1)*(t+1);

// Відображається масив.

for(**int** t=0; t<10; ++t) **cout** << Array[t] << " ";

У мові програмування С++ всі масиви займають суміжні елементи пам'яті. Іншими словами, елементи масиву в пам'яті розташовані послідовно один за одним. Клітина з найменшою адресою належить до першого елемента масиву, а з найбільшою – до останнього.

Для одновимірних масивів загальний розмір масиву в байтах обчислюється так:

всього байтів = розмір типу елемента в байтах ×кількість елементів.

Масиви часто використовують під час програмування, оскільки дають змогу легко обробляти велику кількість взаємопов'язаних між собою змінних.

У мові програмування С++ не можна присвоїти один масив іншому. Наприклад:

```
int aMas[10], bMas[10];
//...
aMas = bMas; // Помилка!!!
```

Щоб помістити вміст одного масиву в інший, необхідно окремо виконати присвоєння кожного значення:

```
int aMas[10], bMas[10], i;
//...
for(i=1; i<10; i++) aMas[i] = bMas[i]; // Правильно!
//...
```

Організація контролю меж масивів

У мові програмування С++ не здійснюється ніякої перевірки порушення контролю меж масивів, тобто нічого не може перешкодити програмісту звернутися до масиву за його межами. Якщо це

відбувається у процесі виконання команди присвоєння, то можуть бути змінені значення в елементах пам'яті, виділених деяким іншим змінним або навіть Вашій програмі. Іншими словами, звернення до масиву (розміром у N елементів) за межею N-го елемента може призвести до руйнування програми за відсутності яких-небудь зауважень з боку компілятора і без видачі повідомлень про помилки під час роботи програми. Це означає, що вся відповідальність за дотримання "кордонів" масивів покладається тільки на програмістів, які повинні гарантувати коректну роботу з масивами. Іншими словами, програміст зобов'язаний використовувати масиви достатньо великого розміру, щоб в них можна було без ускладнень поміщати дані. Однак найкраще у програмі передбачити перевірку перетину "кордонів" масивів.

Наприклад, C++-компілятор "мовчки" скомпілює і дасть змогу запустити таку програму на виконання, хоча у ній відбувається вихід за межі масиву myArray.

```
int main()
{
int myArray[10];
for(int i=0; i<100; i++) myArray[i] = i;
return 1;
```

}

У цьому випадку цикл **for** виконає 100 ітерацій, хоча масив myArray призначений для зберігання тільки десяти елементів. У процесі виконання цієї програми можливий перезапис важливої інформації, що може призвести до аварійної зупинки програми.

Приклад *створення масиву з десяти елементів, кожному елементу присвоюється випадкове число, а потім на екрані відображаються мінімальне та максимальне його значення:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int min, max, Array[10];
	for (int i = 0; i < 10; i++) Array[i] = rand();
	// Знаходимо мінімальне значення.
	min = Array[0];
	for (int i = 1; i < 10; i++)
		if (min > Array[i]) min = Array[i];
	cout << "Мінімальне значення: " << min << endl;
	// Знаходимо максимальне значення.
	max = Array[0];
	for (int i = 1; i < 10; i++)
		if (max < Array[i]) max = Array[i];
	cout << "Максимальне значення: " << max << endl;
	_getch(); return 0;
}
```

Побудова символьних рядків

Найчастіше одновимірні масиви використовуються для побудови символьних рядків. У мові програмування C++ рядок визначається як масив символів, який завершується нульовим символом ('\0'). Під час визначення довжини символьного масиву необхідно враховувати ознаку його завершення, тобто задавати його довжину на одиницю більше довжини найбільшого рядка, які передбачають зберігати у цьому масиві.

Оголошуючи масив str, призначений для зберігання 10-символьного рядка, потрібно використовувати таку команду:

char str[11];

Заданий тут розмір (11) дає змогу зарезервувати місце для нульового символу в кінці рядка.

Оголошення рядкового літерала

Мова програмування C++ дає змогу визначати рядкові літерали. Пригадаємо, що *рядковий літерал* – перелік символів, поміщений в подвійні лапки. Ось декілька прикладів:

"Привіт"

"Мені подобається мова програмування C++"

"#$%@@#$"

""

Рядок, наведений останнім (""), називається *нульовим.* Він складається тільки з одного нульового символу (ознаки завершення рядка). Нульові рядки використовуються для представлення порожніх рядків.

Програмісту не потрібно вручну добавляти в кінець рядкових констант нульові символи. С++-компілятор робить це автоматично.

Зчитування рядків з клавіатури та виведення на екран

Найпростіше зчитувати рядок з клавіатури, створивши масив, який прийме цей рядок за допомогою команди cin. Зчитування рядка, введеного користувачем з клавіатури, відображено у наведеному нижче коді програми.

```
char str[80];
cout << "Введіть рядок: "; cin >> str; // Зчитуємо рядок з клавіатури «Це перевірка»
cout << "Ось Ваш рядок: ";
cout << str;
```

Результат її виконання такий:

Введіть рядок: Це перевірка

Ось Ваш рядок: Це

Під час виведення рядка, введеного з клавіатури, програма відображає тільки слово "Це", а не весь

рядок. Йдеться про те, що оператор ">>" припиняє зчитування рядка, як тільки трапляється символ пропуску, табуляції або нового рядка (пропускні символи). Для вирішення цього питання можна використовувати ще одну бібліотечну функцію **gets_s**(). Загальний формат її виклику є таким:

gets_s(*ім'я_масиву*);

Якщо у програмі необхідно зчитувати рядок з клавіатури, то краще викликати функцію **gets_s**(), а як аргумент передати ім'я масиву, не вказуючи індексу. Після виконання цієї функції заданий масив міститиме текст, введений з клавіатури. Функція **gets_s**() зчитує символи, які вводяться користувачем, доти, доки він не натисне на клавішу <Enter>. Для виклику функції **gets_s**() у програму необхідно включити заголовок <**cstdio**> або <**stdio**.h>. Наприклад:

```
char str[80];
cout << "Введіть рядок: ";
gets_s(str); // Зчитуємо рядок з клавіатури.
cout << "Ось Ваш рядок: ";
cout << str;
```

9.2. Дво- та багатовимірні масиви

Масив може бути двовимірним (матрицею), та багатовимірним, тобто таким, де індексом є не одне число, а кортеж (сукупність) з декількох чисел, кількість яких збігається з розмірністю масиву.

Організація двовимірних масивів

У мові програмування C++ можна використовувати двовимірні масиви. Двовимірний масив, по суті, є списком одновимірних масивів. Щоб оголосити двовимірний масив цілочисельних значень розміром 10×20 з іменем num, достатньо записати таке:

int num[10][20];

На відміну від багатьох інших мов програмування, у яких під час оголошення масиву значення розмірностей відокремлюються комами, у мові програмування C++ кожна розмірність полягає у власну пару квадратних дужок.

Щоб отримати доступ до елемента масиву num з координатами 3×5, необхідно використовувати запис num[3][5].

У двовимірному масиві позиція будь-якого елемента визначається двома індексами. Якщо представити двовимірний масив у вигляді таблиці даних, то один індекс означає рядок, а другий – стовпець (рис. 9.1.). З цього виходить, що якщо доступ

до елементів масиву надати в порядку, у якому вони реально зберігаються в пам'яті, то правий індекс змінюватиметься швидше, ніж лівий.

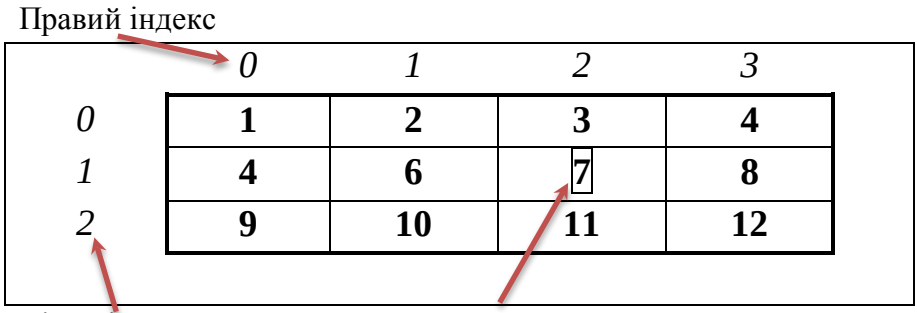


Рис. 9.1. *Схематичне представлення масиву пит*

Для визначення кількості байтів пам'яті, займаної двовимірним масивом, використовується така формула:

к-сть байтів = к-сть рядків × к-сть стовпців ×розмір типу в байтах.

Отже, двовимірний цілочисельний масив розмірністю 10×5 займає в пам'яті 10×5×2, тобто 100 байтів (якщо цілочисельний тип має розмір 2 байт).

Організація багатовимірних масивів

У мові програмування С++, окрім двовимірних, можна визначати масиви трьох і більш вимірів. Ось як оголошується багатовимірний масив:

тип *ім'я*[*розмір1*][*розмір2*]...[*розмірN*];

Наприклад, за допомогою такого оголошення створюється тривимірний цілочисельний масив розміром 4×10×3:

int multidim[4][10][**3**];

Як було зазначено вище, пам'ять, виділена для зберігання всіх елементів масиву, використовується протягом всього часу наявності масиву. Масиви з числом вимірювань, що перевищує три, використовуються нечасто, хоча б тому, що для їх зберігання потрібен великий об'єм пам'яті. Наприклад, зберігання елементів чотиривимірного символьного масиву розміром 10×6×9×4 займе 2 160 байтів. А якщо кожну розмірність збільшити в 10 разів, то займана масивом пам'ять зросте до 21 600 000 байтів. Як бачимо, великі багатовимірні масиви здатні "з'їсти" великий об'єм пам'яті, а програма, яка їх використовує, може дуже швидко наштовхнутися на проблему відсутності пам'яті.

9.3. Ініціалізація елементів масивів

Ознакою масиву при описі є наявність парних дужок []. Константа або константний вираз в квадратних дужках задає число елементів масиву. При описі масиву може бути виконана ініціалізація його

елементів. Існує два методи ініціалізації елементів масивів: розмірних і безрозмірних масивів.

Ініціалізація елементів "розмірних" масивів

У мові програмування С++ передбачено можливість ініціалізації елементів масиву. Формат ініціалізації елементів масиву подібний до формату ініціалізації інших змінних:

тип ім'я_масиву[*розмір*] = {*перелік_значень*};

У цьому записі елемент *перелік_значень* є перелік значень ініціалізації елементів масиву, розділених між собою комами. Тип кожного значення ініціалізації повинен бути сумісний з базовим типом масиву (елементом *тип*).

Перше значення ініціалізації буде збережено в першій позиції масиву, друге значення – в другій і т.д. Слід звернути увагу на те, що крапка з комою ставиться після закритої фігурної дужки (}). Наприклад, в прикладі 10-елементний цілочисельний масив ініціалізувався числами від 1 до 10:

int Array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Після виконання цієї команди елемент Array[0] набуде значення 1, а

елемент Array[9] –значення 10.

Для символьних масивів, призначених для зберігання рядків, передбачено скорочений варіант ініціалізації, який має таку форму:

char *ім'я_масиву*[*розмір*] = "*рядок*";

Наприклад, такий фрагмент коду програми ініціалізує масив str фразою "привіт".

char str[7] = "привіт";

Це рівнозначно по-елементній ініціалізації:

char str[7] = {'п', 'р', 'и', 'в', 'і', 'т', '\0'};

Оскільки у мові програмування С++ рядки повинні завершуватися нульовим символом, то під час оголошення масиву його розмір вказано з врахуванням ознаки кінця. Саме тому в попередньому прикладі масив str оголошений як 7-елементний, хоча у слові "привіт" тільки шість букв. Під час використання рядкового літерала компілятор добавляє нульову ознаку кінця рядка автоматично.

Багатовимірні масиви ініціалізуються за аналогією з одновимірними.

Наприклад, такий фрагмент коду програми масив Array ініціалізувався числами від 1 до 10 і квадратами цих чисел.

```
int Array[10][2] = {
1, 1,
2, 4,
3, 9,
```

```
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100
};
```

Під час ініціалізації багатовимірного масиву перелік ініціалізацій кожної розмірності (підгрупу ініціалізацій) можна помістити у фігурні дужки. Ось, наприклад, як виглядає ще один варіант запису попереднього оголошення:

```
int Array[10][2] = {
{1, 1},
{2, 4},
{3, 9},
{4, 16},
{5, 25},
{6, 36},
{7, 49},
{8, 64},
{9, 81},
{10, 100}
};
```

Під час використання підгруп ініціалізацій

відсутні члени підгрупи ініціалізуються нульовими значеннями автоматично.

"Безрозмірна" ініціалізація елементів масивів

У мові програмування C++ передбачено можливість автоматичного визначення довжини масивів шляхом використання їх "безрозмірного" формату. Якщо в команді ініціалізації масиву не вказано його розмір, то мова програмування C++ автоматично створить масив, розмір якого буде достатнім для зберігання всіх значень ініціалізацій.

```
char e1[] = "Ділення на 0\n";
char e2[] = "Кінець файлу\n";
char e3[] = "У доступі відмовлено \n";
```

Крім зручності в первинному визначенні масивів, метод "безрозмірної" ініціалізації дає змогу змінити будь-яке повідомлення без зазначення його довжини. Тим самим усувається можливість внесення помилок, викликаних випадковим прорахунком.

"Безрозмірна" ініціалізація елементів масивів не обмежується одновимірними масивами. Під час ініціалізації багатовимірних масивів треба вказати усі дані, за винятком крайньої зліва розмірності, щоб C++-компілятор міг належним чином індексувати масив. Використовуючи "безрозмірну" ініціалізацію

масивів, можна створювати таблиці різної довжини, даючи змогу компіляторові автоматично виділяти область пам'яті, достатню для їх зберігання.

У такому прикладі масив Array[][] оголошується як "безрозмірний":

```
int Array[][2] = {
1, 1,
2, 4,
3, 9,
4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 81,
10, 100
};
```

Перевага такої форми оголошення перед "габаритною" (з точним указанням усіх розмірностей) полягає у тому, що програміст може подовжувати або укорочувати таблицю значень ініціалізації, не змінюючи розмірності масиву.

Приклад. *Демонстрація механізму пошуку потрібного елемента в ініціалізованому двовимірному*

масиві:

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int Array[10][2] = {
{1, 1},
{2, 4},
{3, 9},
{4, 16},
{5, 25},
{6, 36},
{7, 49},
{8, 64},
{9, 81},
{10, 100}
};
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int i, j;
	for (;;) {
		cout << "Введіть число від 1 до 10: "; cin >> i;
		if (i >= 1 && i <= 10) break;
	}
	// Пошук значення i.
	for (j = 0; j < 10; j++)
		if (Array[j][0] == i) break;
	cout << "Квадрат числа " << i << " дорівнює " <<
Array[j][1];
	_getch(); return 0;
	}
```

9.4. Двовимірні масиви рядків

Існує спеціальна форма двовимірного символьного масиву, яка є масивом рядків. У його використанні немає нічого незвичайного. Наприклад,

при програмуванні баз даних для з'ясування коректності команд, що вводяться користувачем, вхідні дані порівнюються з вмістом масиву рядків, у якому записано допустимі у цьому додатку команди.

Для побудови масиву рядків використовується двовимірний символьний масив, у якому розмір лівого індексу визначає кількість рядків, а розмір правого – максимальну довжину кожного рядка. Наприклад, у процесі виконання такої команди оголошується масив, призначений для зберігання 30 рядків завдовжки 80 символів:

```
char strArray[30][80];
```

Отримати доступ до окремого рядка достатньо просто: достатньо вказати тільки лівий індекс. Наприклад, така команда викликає функцію gets() для запису третього рядка масиву:

```
gets(strArray[2]);
```

Приклад *програми ведення бази даних службовців, у якій зберігається ім'я, номер телефону, кількість годин, відпрацьованих службовцями за звітний період, і розмір погодинного окладу для кожного службовця:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
#define n 10 // Оголошується стала n=10
using namespace std; // Використання стандартного простору імен
```

```
char name[10][80]; // Масив імен службовців.
char phone[10][20]; // Масив телефонних номерів службовців.
float hours[10]; // Масив годин, відпрацьованих за тиждень.
float oklad[10]; // Масив погодинних окладів.
int MenuSelect();
void Enter(), report();
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int vybir;
	do {
		// Отримуємо команду вибрану користувачем.
		vybir = MenuSelect();
		switch (vybir) {
		case 0: break;
		case 1: Enter();
			break;
		case 2: report();
			break;
		default: cout << "Спробуйте ще раз.\n" << endl;
		}
	} while (vybir != 0);
	_getch(); return 0;
}
// Функція повертає команду, вибрану користувачем
int MenuSelect()
{
	int vybir;
	cout << "0. Вихід з програми" << endl;
	cout << "1. Введення інформації" << endl;
	cout << "2. Генерування звіту" << endl;
	cout << endl << " Виберіть команду: "; cin >> vybir;
	return vybir;
}
// Функція введення інформації в базу даних
void Enter()
{
	for (int i = 0; i < n; i++) {
		cout << "Введіть прізвище службовця: "; cin >>
name[i];
		cout << "Введіть номер телефону службовця: "; cin
>> phone[i];
		cout << "Введіть кількість відпрацьованих годин:
"; cin >> hours[i];
```

```
            cout << "Введіть оклад службовця: "; cin >>
oklad[i];
            cout << endl;
      }
}
// Відображення звіту
void report()
{
      for (int i = 0; i < n; i++) {
            cout << name[i] << " " << phone[i] << endl;
            cout << "Заробітна плата за тиждень: " <<
oklad[i] * hours[i];
            cout << endl;
      }
      cout << endl;
}
```

9.5. Покажчики та масиви

У мові програмування C++ покажчики і масиви тісно пов'язані між собою, причому настільки, що часто поняття "покажчик" і "масив" взаємозамінні. Для початку розглянемо такий фрагмент програми:

char str[80];

char *p1;

p1 = str;

У цьому записі str є іменем масиву, що містить 80 символів, а p1 – покажчик на тип **char**. Особливий інтерес представляє третій рядок, у процесі виконання якого покажчику p1 присвоюється адреса першого елемента масиву str.

Іншими словами, після цього присвоєння p1 вказуватиме на елемент str[0].

Йдеться про те, що у мові програмування C++ використання імені масиву без індексу генерує покажчик на перший елемент цього масиву. Таким чином, у процесі виконання операції присвоєння p1 = str адреса str[0] присвоюється покажчику p1. Це і є ключовим моментом, який необхідно чітко розуміти: неіндексоване ім'я масиву, використане у виразі, означає покажчик на початок цього масиву.

Оскільки після розглянутого вище присвоєння покажчик p1 вказуватиме на початок масиву str, то p1 можна використовувати для доступу до елементів цього масиву. Наприклад, якщо потрібно отримати доступ до п'ятого елемента масиву str, використовується один з таких виразів: str[4] або *(p1+4).

У обох випадках буде виконане звернення до п'ятого елемента. Необхідно пам'ятати, що індексування елементів масиву починається з нуля, тому при індексі, що дорівнює чотирьом, забезпечується доступ до п'ятого елемента. Таке саме враження справляє підсумовування значення початкового покажчика (p1) з числом 4, оскільки p1 вказує на перший елемент масиву str.

Необхідність використання круглих дужок, у які поміщено вираз p1+4, пов'язана з тим, що оператор "*" має вищий пріоритет, ніж оператор додавання "+". Без

цих круглих дужок вираз звівся би до отримання значення, яке адресується покажчиком p1, тобто значення першого елемента масиву, яке потім було б збільшено на 4.

Проте в загальному випадку покажчики і масиви не є взаємозамінними. Розглянемо, наприклад, такий фрагмент коду програми:

```
int num[10];
for(int i=0; i<10; i++) {
*num = i; // Тут все гаразд.
num++; // Помилка – змінну num модифікувати не можна.
}
```

Тут використовується масив цілочисельних значень з іменем num. Як зазначено в коментарі, незважаючи на те, що абсолютно прийнятно застосувати до імені num оператор "*" (який зазвичай застосовується до покажчиків), проте абсолютно неприпустимо модифікувати значення num. Йдеться про те, що num – константа, яка вказує на початок масиву. І її, як наслідок, інкрементувати ніяк не можна. Іншими словами, хоча ім'я масиву (без індексу) дійсно генерує покажчик на початок масиву, однак його значення зміні не підлягає.

Хоча ім'я масиву генерує константу-покажчик, його, проте, (подібно до покажчиків) можна помістити

у вирази, якщо, звичайно, воно при цьому не модифікується. Наприклад, наступна команда, у процесі виконання якої елементу num[3] присвоюється значення 100, є цілком допустимою:

*(num+3) = 100; // Тут все гаразд оскільки num не змінюється.

9.6. Масиви покажчиків

Покажчики, подібно до інших типів даних, можуть зберігатися в масивах. Ось, наприклад, як виглядає оголошення 10-елементного масиву покажчиків на **int**-значення.

int *Array[10];

У цьому записі кожен елемент масиву Array містить покажчик на цілочисельне значення.

Щоб присвоїти адресу **int**-змінній з іменем var третьому елементу цього масиву покажчиків, записується таке:

Array[2] = &var;

Необхідно пам'ятати, що тут Array – масив покажчиків на цілочисельні значення. Елементи цього масиву можуть містити тільки значення, які є адресами змінних цілого типу. Ось тому змінна var передує оператору "&".

Щоб присвоїти значення змінної var

цілочисельній змінній x за допомогою масиву Array, використовують такий синтаксис:

x = *Array[2];

Оскільки адреса змінної var зберігається в елементі Array[2], то застосування оператора "*" до цієї індексованої змінної дасть змогу набути значення змінній var.

Подібно до інших масивів, масиви покажчиків можна ініціалізувати. Як правило, масиви ініціалізованих покажчиків використовують для зберігання покажчиків на рядки. Наприклад, щоб створити функцію, яка виводить щасливі передбачення, можна таким чином визначити масив **fortunes**:

```
const char *fortunes[] = {
"Незабаром гроші потечуть до Вас рікою.\n" ,
"Ваше життя осяє нове кохання.\n" ,
"Ви житимете довго і щасливо.\n" ,
"Гроші, вкладені зараз в справу, незабаром принесуть дохід.\n" ,
"Близький друг шукатиме Вашої підтримки.\n"
};
```

Не можна забувати, що мова програмування C++ забезпечує зберігання всіх рядкових літералів у таблиці рядків, пов'язаній з конкретною програмою, тому масив потрібен тільки для зберігання покажчиків

на ці рядки. Таким чином, для виведення другого повідомлення достатньо використовувати команду, подібну до такої:

cout << fortunes[1];

Наведений нижче код програми «передбачень» є цілком коректним. Для отримання випадкових чисел у цій програмі використовується функція **rand**(), а для отримання випадкових чисел в діапазоні від 0 до 4 – оператор ділення за модулем, оскільки саме такі числа можуть слугувати для доступу до елементів масиву за індексом.

Слід звернути увагу на цикл **while**, який викликає функцію **rand**() доти, доки не буде натиснуто на яку-небудь клавішу. Оскільки функція **rand**() завжди генерує одну і ту саму послідовність випадкових чисел, важливо мати можливість програмно використовувати цю послідовність з певної довільної позиції. Ефект випадковості досягається за рахунок повторних звернень до функції **rand**(). Коли користувач натисне на клавішу, цикл зупиниться на деякій випадковій позиції послідовності чисел, що генеруються, і ця позиція визначить номер повідомлення, яке буде виведено на екран.

Приклад *програми «передбачень» із використанням масиву покажчиків:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
const char* fortunes[] = {
"Незабаром гроші потечуть до Вас рікою.\n" ,
"Ваше життя осяє нове кохання.\n" ,
"Ви житимете довго і щасливо.\n" ,
"Гроші, вкладені зараз в справу, незабаром принесуть дохід.\n"
,
"Близький друг шукатиме Вашої підтримки.\n"
};
int main()
{
	setlocale(LC_CTYPE, "ukr");
	int chance;
	cout << "Щоб дізнатися про свою долю, натисніть будь-яку
клавішу: ";
	// Рандомізуємо генератор випадкових чисел.
	while (!_kbhit()) rand();
	cout << endl;
	chance = rand() % 5;
	cout << fortunes[chance];
	_getch(); return 0;
}
```

9.7. Динамічні масиви

За способом розміщення масиви поділяються на статичні та динамічні. Описані вище масиви є статичними. Як вже відомо, розмір статичного масиву можна задавати константою або константним виразом. Оскільки ділянка у оперативній пам’яті під масив виділяється на етапі компіляції і її розмір визначається типом елементів масиву та їх кількістю, розмірність

масиву повинна бути визначена у тексті програми, а не під час її виконання. Для визначення масиву змінного розміру використовується механізм динамічного виділення пам'яті. Наприклад, динамічне виділення пам'яті під 10 цілочисельних елементів:

int*m=**new** int [10];

Враховуючи те, що ім'я масиву є вказівником, зрозумілим стає зміст останньої операції: вказівникові на int m присвоюється початкова адреса ділянки пам'яті, виділеної у динамічній області під 10 цілочисельних елементів.

Перевагою динамічних масивів є те, що їх розмірність може бути змінною, тобто у програмі можна працювати з масивами довільного розміру, не вносячи змін до тексту програми. Проте, динамічні масиви не можна ініціалізувати при визначенні і вони за замовчуванням не заповнюються нулями.

Пам'ять, зарезервовану під динамічний масив за допомогою **new[ ]**, потрібно звільняти оператором **delete** [] <ім'я масиву>.

Отже, якщо розмірність масиву необхідно задавати в процесі виконання програми (до введення його елементів), то доцільно створювати динамічний масив.

Для створення динамічного двовимірного масиву необхідно вказати в операції new всі його

розмірності, причому ліва розмірність (кількість рядків) може бути змінною:

```
int nriad=5; //к-сть рядків
int **m=(int**) new int[nriad][10];
```

Більш ефективний і безпечний спосіб виділення пам'яті під двовимірний масив, коли обидві його розмірності вказуються на етапі виконання програми, тобто є змінними. Наприклад:

```
int nriad, nstp;  //к-сть рядків, к-сть стовпців
cout <<"Введіть кількість рядків та стовпців:";
cin >> nriad >> nstp;
int**a = new int *[nriad];  // 1
for (int i=0; i < nriad; i++)  // 2
a[i] = new int[nstp];  // 3
```

У цьому прикладі в операторі 1 оголошується змінна типу "вказівник на вказівник на int" і виділяється пам'ять під масив вказівників на рядки масиву (nriad – кількість рядків). В операторі 2 організовано цикл для виділення пам'яті під кожен рядок масиву. В операторі 3 кожному елементу масиву вказівників на рядки присвоюється адреса початку ділянки пам'яті, виділеної під рядок двовимірного масиву з кількістю елементів типу int у ній, рівною nstp.

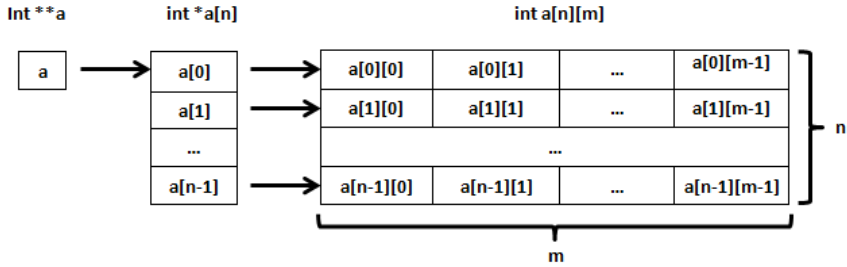


Рис. 6.10. Виділення пам'яті для двовимірного масиву

9.8. Виклик функцій з масивами

Якщо масив є аргументом функції, то необхідно розуміти, що під час виклику такої функції їй передається тільки адреса першого елемента масиву, а не повна його копія. Це означає, що оголошення параметра повинно мати тип, сумісний з типом аргумента. Взагалі існує три способи оголосити параметр, який приймає покажчик на масив. По-перше, параметр можна оголосити як масив, тип і розмір якого збігається з типом і розміром масиву, використовуваного під час виклику функції.

Приклад *програми із реалізацією передачі у функцію масиву:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
using namespace std; // Використання стандартного простору імен
void Display(int num[10]); // Попереднє оголошення функції
int main()
```

```
{
	int t[10];
	for (int i = 0; i < 10; ++i) t[i] = i;
	Display(t); // Передаємо функції масив t.
	_getch(); return 0;
}
// Функція виводить усі елементи масиву.
void Display(int num[10])
{
	for (int i = 0; i < 10; i++) cout << num[i] << endl;
	}
```

Другий спосіб оголошення параметра-масиву полягає в його представленні у вигляді безрозмірного масиву, як це показано далі:

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

У цьому записі параметр num оголошується як цілочисельний масив невідомого розміру. Оскільки мова C++ не забезпечує перевірки порушення меж масиву, то реальний розмір масиву не релевантний чинник для подібного параметра (але, безумовно, не для програми в цілому). Цілочисельний масив при такому способі оголошення також автоматично перетвориться C++-компілятором у покажчик на цілочисельне значення.

Нарешті, розглянемо третій спосіб оголошення параметра-масиву. При передачі масиву функції її параметр можна оголосити як покажчик. Якраз цей

варіант найчастіше використовується програмістами. Ось приклад:

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

Можливість такого оголошення параметра (у цьому випадку num) пояснюється тим, що будь-який покажчик (подібно до масиву) можна індексувати за допомогою символів квадратних дужок "[]". Таким чином, всі три способи оголошення параметра-масиву приводяться до однакового результату, який можна виразити одним словом: покажчик.

Проте окремий *елемент* масиву, що використовується як аргумент, обробляється подібно до звичайної змінної.

Запитання для самоконтролю

1. Яку конструкцію називають масивом?
2. Опишіть синтаксис оголошення одновимірного масиву.
3. Якими способами можна ініціалізувати масив?
4. Як можна здійснити виведення масиву?
5. Яких значень можуть набувати індекси елементів масиву?

6. За допомогою яких функцій генерують випадкові числа?
7. Опишіть синтаксис оголошення двовимірного масиву?
8. Яким чином можна ініціалізувати двовимірний масив?
9. Які функції використовують для управління пам’яттю при роботі з динамічними масивами?

РОЗДІЛ 10. РЯДКИ

10.1. Застосування бібліотечних функцій для обробки рядків

Мова програмування C++ підтримує багато функцій для оброблення рядків. Найпоширенішими з них є такі: **strcpy(), strcat(), strlen(), strcmp()**.

Для виклику всіх цих функцій у програму необхідно включити заголовок <**cstring**>. Тепер познайомимося з кожною функцією окремо.

Механізм використання функції strcpy()

Функція **strcpy()** копіює вміст рядка from в рядок to. Загальний формат її виклику є таким:

strcpy(*to, from*);

Необхідно пам'ятати, масив, який використовують для зберігання рядка *to*, повинен бути достатньо великим, щоб в нього можна було помістити рядок з масиву from. Інакше масив to переповниться, тобто відбудеться вихід за його межі, що може призвести до руйнування програми.

Механізм використання функції strcat()

Функція **strcat**() приєднує рядок *s2* до кінця рядка *s1*; при цьому рядок *s2* не змінюється. Обидва

рядки повинні завершуватися нульовим символом. Звернення до цієї функції має такий формат:

strcat(*s1*, *s2*);

Результат виклику цієї функції, тобто остаточний рядок *s1* також завершуватиметься нульовим символом.

Механізм використання функції strcmp()

Функція **strcmp**() порівнює рядок *s2* з рядком *s1* і повертає значення 0, якщо вони однакові. Якщо рядок *s1* лексикографічно (тобто відповідно до алфавітного порядку) більший від рядка *s2*, то повертається позитивне число.

Якщо рядок *s1* лексикографічно менший від рядка *s2*, то повертається негативне число.

Звернення до функції **strcmp**() має такий формат:

strcmp(*s1*, s2);

Використання цієї функції продемонстровано у наведеному нижче коді програми, яка слугує для перевірки правильності пароля, введеного користувачем (для введення пароля з клавіатури і його верифікації слугує функція користувача **password**()).

Приклад *програми на використання функції* ***strcmp():***

```
#include <iostream> // Потокове введення-виведення
#include <cstring> // Робота з рядковими типами даних
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
bool password();
int main()
{
	setlocale(LC_CTYPE, "ukr");
	if (password()) cout << "Вхід дозволено" << endl;
	else cout << "У доступі відмовлено" << endl;
	_getch(); return 0;
}
// Функція password() повертає значення true, якщо пароль
прийнятий, і значення false інакше.
bool password()
{
	char str[80];
	cout << "Введіть пароль: "; gets_s(str);
	if (strcmp(str, "password")) { // Рядки різні.
		cout << "Пароль недійсний" << endl;
		return false;
	}
	// Порівнювані рядки збігаються
	return true;
}
```

Під час застосування функції **strcmp**() важливо пам'ятати, що вона повертає число 0 (тобто значення **false**), якщо порівнювані рядки однакові. Отже, якщо Вам необхідно виконати певні дії за умови збігу рядків, то програміст повинен використовувати оператор НЕ (!).

Механізм використання функції strlen()

Загальний формат виклику функції **strlen**() є таким:

strlen(*s*);

При підрахунку символів, з яких складається заданий рядок, ознака завершення рядка (нульовий символ) не враховується.

Покажемо застосування цієї функції для відображення рядка на екрані в зворотному порядку. Наприклад, під час введення слова "привіт" програма відобразить слово тівирп. Дя цього використаємо властивість, що рядки є символьні масиви, які дають змогу посилатися на кожен елемент (символ) окремо.

Приклад *програми на відображення рядка в зворотному порядку:*

```
#include <iostream> // Потокове введення-виведення
#include <cstring> // Робота з рядковими типами даних
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	char str[80];
	cout << "Введіть рядок: "; gets_s(str);
	for (int i = strlen(str) - 1; i >= 0; i--) cout <<
str[i];
	_getch(); return 0;
	}
```

Механізм використання ознаки завершення рядка

Факт завершення нульовими символами всіх

С++-рядків можна використовувати для спрощення різних операцій над ними. Наведемо приклад програмного коду для заміни всіх символів рядка їх прописними еквівалентами. У ньому використовується бібліотечна функція **toupper**(), яка повертає прописний еквівалент свого символьного аргументу. Для виклику функції **toupper**() необхідно приєднати до програми заголовок <**cctype**>.

Зверніть увагу на те, що як умову завершення циклу **for** використано масив str, індексований змінній i, що керує (str[i]). Такий спосіб керування циклом цілком прийнятний, оскільки за дійсне значення у мові програмування С++ приймається будь-яке ненульове значення. Згадаймо, що всі друкарські символи представляються значеннями, які не дорівнюють нулю, і тільки символ, що завершує рядок, дорівнює нулю. Отже, цей цикл працює доти, доки індекс не вкаже на нульову ознаку кінця рядка, тобто доки значення str[i] не стане нульовим. Оскільки нульовий символ відзначає кінець рядка, цикл зупиняється точно там, де потрібно.

Приклад *програми на індексування покажчика подібно до індексування масиву:*

```
#include <iostream> // Потокове введення-виведення
#include <cctype> // Робота з символьними аргументами
#include <conio.h> // Консольний режим роботи
```

```
using namespace std; // Використання стандартного простору імен
int main()
{
	char str[20] = "I love informatics";
	char* p;
	p = str; // Індексуємо покажчик.
	for (int i = 0; p[i]; i++) p[i] = toupper(p[i]); //
Повертає прописні символи
	cout << p; // Відображаємо рядок.
	_getch(); return 0;
	}
```

У процесі виконання програма відобразить на екрані таке:

I LIKE INFORMATICS

Ось як працює ця програма. Спочатку в масив str вводиться рядок " I love informatics". Потім адреса початку цього рядка присвоюється покажчику p.

Після цього кожен символ рядка str за допомогою функції **toupper**() перетвориться в його прописний еквівалент за допомогою індексування покажчика p. Пам'ятайте, що вираз p[i] за своєю дією однаковий виразу *(p+i).

10.2. Бібліотечні функції для перетворення символьних рядків у числовий формат і навпаки

Функція **atoi ()** перетворює рядок, на який вказує параметр str, у величину типу int. Рядок повинен містити коректний запис цілого числа. В іншому випадку повертається 0.

Формат запису функції **atoi ()** (прототип: stdlib.h):

int atoi (const char * str);

Число може завершуватися будь-яким символом, який не може входити до складу рядкового подання цілого числа. Сюди відносяться пропуски, знаки пунктуації та інші знаки, які не є цифрами. Таким чином, виклик функції atoi () для числа 123.23 поверне ціле значення, а частина 0.23 буде опущена.

Формат запису функції **atol()** (прототип: stdlib.h):

long atol (const char * str).

Функція atol () перетворює рядок, на який вказує параметр str, у ціле число типу long int. Рядок повинен містити коректний запис довгого цілого числа. В іншому випадку повертається 0.

Число може закінчуватися будь-яким символом, який не може входити до складу рядкового подання цілого числа. Сюди відносяться пропуски, знаки пунктуації та інші символи, які не є цифрами. Таким чином, якщо викликати функцію **atol** () і передати їй як аргумент число 123.23, то вона поверне значення

123, а частина 0.23 буде опущена.

Формат запису функції **atof ()** (прототип: math.h, stdlib.h):

double atof (const char * str);

Функція atof () перетворює рядок str у величину типу double. Рядок повинен містити коректне число з плаваючою крапкою. У разі помилки повертається 0, а змінна errno встановлюється рівною ERANGE.

Число може закінчуватися будь-яким символом, який не може бути частиною числа з плаваючою крапкою. Наприклад, цим символом може бути пробіл, знак пунктуації, відмінний від точки, буква, відмінна від «E» або «e». Це означає, що виклик atof () для рядка «100.000HELLO» поверне 100.00.

Формат запису функції **atold ()** (прототип: math.h, stdlib.h):

long double atold (const char * str);

Функція atold () є версією функції atof (), що повертає значення типу long double.

***Приклад**. Зчитати два числа з плаваючою крапкою і вивести їх суму.*

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	char num1[80], num2[80];
	printf("Enter first number:");
	gets_s(num1);
	printf("Enter second number:");
	gets_s(num2);
	printf("The sum is:% f", atof(num1) + atof(num2));
	return 0;
}
```

Формат запису функції **itoa ()** (прототип: stdlib.h):

char * itoa (int num, char * str, int radix)

Функція itoa не визначена стандартом ANSI C. Вона перетворює ціле число num в рядковий еквівалент і поміщає результат у рядок, на який вказує параметр str. Основу системи числення для запису вихідного рядка визначено параметром radix, який може приймати значення в інтервалі від 2 до 36.

Функція itoa () повертає покажчик на str. Функція не має значення, що повертається, відповідного помилку. Необхідно дбати про те, щоб рядок для виведення даних був досить довгим, щоб вмістити в себе результат. Максимальна необхідна довжина становить 17 байт.

Щоб використовувати ці функції без

попередження про старіння, потрібно визначити _CRT_SECURE_NO_WARNINGS макрос препроцесора перед включенням всіх заголовків CRT.

Приклад *програми, що виводить число 1423 в шістнадцятковому форматі (58F):*

```
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
	char p[17];
	_itoa(1423, p, 16);
	printf(p);
	return 0;
}
```

Запитання для самоконтролю

1. Як реалізовано рядки в C++?
2. Що є ознакою кінця рядка в C++?
3. Назвіть способи ініціалізації рядків в C++.
4. Яким чином можна здійснювати ввід/вивід рядків в C++?
5. Як називається бібліотека, що містить функції обробки рядків?
6. Які функції обробки рядків вам відомі?

РОЗДІЛ 11. СТРУКТУРИ ТА ОБ'ЄДНАННЯ ДАНИХ

11.1. Механізм використання структур

У мові програмування С++ *структура* представляє колекцію змінних, об'єднаних загальним іменем, яка забезпечує зручний засіб зберігання споріднених даних в одному місці. Структура – сукупність різних типів даних, оскільки вони складаються з декількох різних, але логічно взаємопов'язаних між собою змінних. З цих самих причин структури іноді називають складеними або конгломератними типами даних.

Структура — група взаємопов'язаних між собою змінних.

Перед визначенням структурних змінних, необхідно визначити формат структури. Це робиться за допомогою оголошення структури. Оголошення структури дає змогу компілятору зрозуміти, змінні якого типу вона містить.

Змінні, які належать до структури, називаються її *членами*. Члени структури також називають *полями*. Член структури — змінна, яка є частиною структури. У загальному випадку всі члени структури мають бути логічно пов'язані одна з одною! Наприклад, структури

зазвичай використовують для зберігання такої інформації, як поштові адреси, банківські реквізити, елементи книжкової бібліографії і т.ін. Безумовно, відносини між членами структури абсолютно суб'єктивні і визначаються програмістом. Компілятор "нічого про них не знає" (або "не хоче знати").

Ключове слово struct означає початок оголошення структури.

Загальний формат оголошення структури має такий вигляд:

struct *ім'я_типу_структури* {
тип ім'я_члена_1;
тип ім'я_члена_2;
тип ім'я_члена_3;
. . . . . .
тип ім'я_члена_n;
} *структурна_змінна_1, структурна_змінна_2, ..., структурна_змінна_m;*

Розглянемо деякі приклади оголошення структур. Визначимо структуру, яка може містити інформацію про продукцію, що зберігаються на складі. Один запис інвентарної відомості зазвичай складається з декількох даних, наприклад: назви продукції, вартості та наявної кількості. Тому для

керування такою інформацією зручно використовувати саме таку структуру. У наведеному нижче коді програми оголошується структура, яка визначає такі елементи: назву продукції, її вартість, роздрібну ціну, наявну кількість, кількість днів до поновлення запасів. Цих даних часто цілком достатньо для керування складом. Про початок оголошення структури компіляторові повідомляє ключове слово struct:

struct invStruct { // *Попереднє оголошення типу структури*

char namеProd[40]; // *Назва продукції*

double vartProd; // *Вартість продукції*

double rozdrCina; // *Роздрібна ціна*

int nayavKilk; // *Наявна кількість*

int kilkDniv; // *Кількість днів до поновлення запасів_*

};

Слід звернути увагу на те, що оголошення структури завершується крапкою з комою, тобто вона може бути настановою. Ім'ям типу структури тут є invStruct. Іншими словами, ім'я invStruct ідентифікує конкретну структуру даних і є її специфікатором типу.

У нашому оголошенні структури насправді не було створено жодної структурної змінної, а визначено тільки формат типу даних. Щоб за

допомогою цієї структури визначити реальну структурну змінну (тобто фізичний об'єкт), потрібно записати таку команду:

invStruct InvVidom;

Ось тепер визначається структурна змінна типу invStruct з іменем InvVidom.

Під час визначення структурної змінної компілятор мови програмування C++ автоматично виділить об'єм пам'яті, достатній для зберігання всіх членів структри.

Одночасно з оголошенням імені типу структури можна визначити одну або декілька структурних змінних:

struct invStruct { // Попереднє оголошення типу структури

char namеProd[40]; // Назва продукції

double vartProd; // Вартість продукції

double rozdrCina; // Роздрібна ціна

int nayavKilk; // Наявна кількість

int kilkDniv; // Кількість днів до поновлення запасів

} InvVidomA, InvVidomB, InvVidomC; // Визначення структурної змінної

Цей код програми оголошує структурний тип invStruct і визначає структурні змінні InvVidomA, InvVidomB і InvVidomC цього типу. Важливо

розуміти, що кожна структурна змінна містить власні копії членів структури. Наприклад, поле vartProd структури InvVidomA ізольовано від поля vartProd структури InvVidomB. Отже, зміни, що вносяться в певне поле однієї структурної змінної, ніяк не впливають на вміст такого самого поля іншої структурної змінної.

Якщо для коду програми достатньо тільки однієї структурної змінної, то оголошення структури необов'язково міститиме ім'я структурного типу. Для розуміння сказаного розглянемо такий приклад:

```
struct { // Попереднє оголошення типу
структури
    char nameProd[40]; // Назва продукції
    double vartProd; // Вартість продукції
    double rozdrCina; // Роздрібна ціна
    int nayavKilk; // Наявна кількість
    int kilkDniv; // Кількість днів до поновлення
запасів
    } vidom; // Визначення структурної змінної
```

Цей код програми визначає одну структурну змінну vidom відповідно до оголошення структури, яка їй передує.

Доступ до членів структури

До окремих членів структури доступ

здійснюється за допомогою оператора "крапка". Наприклад, у процесі виконання такого коду програми значення 10.39 буде присвоєно полю vartProd структурної змінної InvVidom, яка була оголошена вище:

InvVidom.vartProd = 10.39;

Щоб звернутися до члена структури, потрібно перед його іменем поставити ім'я структурної змінної та оператор "крапка". Так здійснюється доступ до всіх членів структури. Загальний формат доступу до члена структури записується так:

ім'я_структурної_змінної.ім'я_члена_структури;

Щоб вивести значення поля vartProd структурної змінної InvVidom на екран, необхідно записати таку команду:

cout << InvVidom.vartProd;

Аналогічним способом можна використовувати символьний масив InvVidom.nameProd у виклику функції **gets**():

gets(InvVidom.nameProd);

У цьому записі функції **gets**() буде передано символьний покажчик на початок області пам'яті, відведеної члену nameProd структури InvVidom.

Якщо виникає потреба отримати доступ до

окремих елементів массиву структур під назвою, наприклад, InvVidom.nameProd, необхідно використати індексацію масиву. Наприклад, за допомогою цього коду програми можна посимвольно вивести на екран монітора вміст поля масиву структур InvVidom.nameProd:

```
for(int t=0; InvVidom.nameProd[t]; t++)
cout << InvVidom.nameProd[t];
cout << endl;
```

Поняття про масиви структур

Структури можуть бути елементами масивів, тобто, масиви структур використовуються достатньо часто. Щоб визначити масив структур, необхідно спочатку оголосити структуру, а потім визначити масив елементів цього структурного типу. Наприклад, щоб визначити 100-елементний масив структур типу invStruct (який було оголошено вище), достатньо записати таку настанову:

```
invStruct prodArray[100];
```

Щоб отримати доступ до конкретної структури в масиві структур, необхідно індексувати ім'я структури. Щоб відобразити на екрані вміст члена nayavKilk третьої структури, достатньо використати таку настанову:

```
cout << prodArray[2].nayavKilk;
```

Механізм присвоєння структур

Вміст однієї структури можна присвоїти інший, якщо обидві ці структури мають однаковий тип.

У мові програмування C++ кожне нове оголошення структури визначає новий тип. Отже, навіть якщо дві структури фізично однакові, але мають різні імена типів, компілятор вважатиме їх різними і не дасть змоги присвоїти значення однієї з них іншій.

Передача структури функції як аргументу

При передачі структури функції як аргумент використовується механізм передачі параметрів за значенням. Це означає, що будь-які зміни, внесені у вміст структури в тілі функції, якій вона передана, не впливають на структуру, що використовується як аргумент цією функцією. Проте необхідно мати на увазі, що передача великих структур вимагає значних витрат системних ресурсів. Як правило, чим більше даних передається функції, тим більше витрачається системних ресурсів.

Використовуючи структуру як параметр, необхідно також пам'ятати, що тип аргументу повинен відповідати типу параметра.

Повернення функцією структури як значення

Для повернення функцією структури як значення використовується механізм повернення параметрів за значенням. Це означає, що будь-які зміни, внесені у вміст структури, яка визначена в тілі функції, впливають на структуру, якій присвоюється значення, що повертається цією функцією. Проте необхідно мати на увазі, що повернення великих структур вимагає значних витрат системних ресурсів. Як правило, чим більше даних повертається функцією, тим більше витрачається системних ресурсів.

Використовуючи структуру як параметр, необхідно також пам'ятати, що тип структури повинен відповідати типу повернутого значення.

Механізм використання покажчиків на структури й оператора "стрілка"

У мові програмування C++ покажчики на структури можна використовувати так само, як і покажчики на змінні будь-якого іншого типу. Проте використання покажчиків на структури має ряд особливостей, які необхідно враховувати.

Покажчик на структуру оголошується так само, як покажчик на будь-яку іншу змінну, тобто за допомогою символу "*", поставленого перед іменем структурної змінної. Наприклад, використовуючи

визначену вище структуру invStruct, можна записати таку настанову, яка оголошує змінну invPointer покажчиком на дані типу invStruct:

invStruct *invPointer;

Щоб знайти адресу структурної змінної, необхідно перед її іменем розмістити оператор "&". Наприклад, припустимо, що за допомогою наведенного нижче коду програми ми оголошуємо структуру, визначаємо структурну змінну і покажчик на структуру оголошеного типу:

struct balStruct { // Попереднє оголошення типу структури

float balance;

char name[80];

} person; // Визначення структурної змінної

balStruct *p; // Визначаємо покажчик на структуру.

Тоді у процесі виконання настанови

p = &person;

у покажчик p буде поміщено адресу структурної змінної person.

До членів структури можна отримати доступ за допомогою покажчика на цю структуру. Але в цьому випадку використовується не оператор "крапка", а оператор "->". Наприклад, у процесі виконання такої настанови ми отримуємо доступ до поля balance через

покажчик p:

p->balance

Оператор "->" називається *"оператором стрілка"*. Він утворюється з використанням знаків "мінус" і "більше".

Покажчик на структуру можна використовувати як параметр функції.

Важливо пам'ятати про такий спосіб передачі параметрів, оскільки він працює набагато швидше, ніж у випадку, коли функції "власною персоною" передається об'ємна структура.

Посилання на структури

Для доступу до структури можна використовувати посилання. Посилання на структуру часто використовують як параметр функції або значення, що повертається функцією. Під час отримання доступу до членів структури за допомогою посилання використовують оператор "крапка".

Механізм використання як членів структур масивів і структур

Кожен член структури може мати будь-який допустимий тип даних, у тому числі і такі складені типи, як масиви й інші структури. Масив, який використовується як член структури, обробляється

цілком звичайним способом. Розглянемо таку структуру:

struct demoStruct { // Попереднє оголошення типу структури

int Array[10][10]; // Цілочисельний масив розміром 10x10.

float b;

} var; // Визначення структурної змінної

Щоб присвоїти певне значення елементу масиву Array з "координатами" (3, 7) у структурі var типу demoStruct, необхідно записати таку настанову:

var.Array[3][7] = 2.37;

Як показано в цьому прикладі, якщо масив є членом структури, то для доступу до елементів цього масиву індексується ім'я масиву, а не ім'я структури.

Якщо певна структура є членом іншої структури, то вона називається *вкладеною структурою.* У наведеному нижче прикладі структура addrStruct вкладена у структуру empStruct:

struct addrStruct { // Попереднє оголошення адреси структури

char name[40]; // Прізвище службовця

char street[40]; // Вулиця

char city[40]; // Місто

char zip[10]; // Поштовий індекс

};

struct empStruct { // Попереднє оголошення реквізитів структури

addrStruct address; // Адреса службовця

float wage; // Оклад службовця

} worker;

Тут структура empStruct має два члени. Першим членом є структура типу addrStruct, яка міститиме адресу службовця. Другим членом є змінна wage, яка зберігає його оклад. У процесі виконання наведеної нижче настанови полю zip структури address, яка є членом структури worker, буде присвоєно поштовий індекс 76285:

worker.address.zip = 76285;

Як бачимо, члени структур вказуються зліва направо, – від крайньої зовнішньої до найдальшої внутрішньої. Структура також може містити як свій член покажчик на цю ж структуру. Тобто для структури цілком допустимо містити член, який є покажчиком на неї саму. Наприклад, в такій структурі змінна strP оголошується як покажчик на структуру типу myStruct, тобто на оголошувану тут структуру:

struct myStruct { // Попереднє оголошення типу структури

int a;

char str[80];

myStruct *strP; // Покажчик на структуру типу

myStruct
};

Приклад *створення і обробки бази даних студентів.*

```
#include <iostream>
#include <conio.h>
#include <iomanip>
#include <clocale>
#define n 2
using namespace std; // Використання стандартного простору імен

struct pib_inf {
    char pr[20];
    char name[10];
};

struct student
{
    pib_inf pib;
    char ch;
    int kurs;
    float bal;
};

void print(student a[]);

int main(int argc, char* argv[])
{
    setlocale(LC_CTYPE, "ukr");
    student st[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Прізвище    : "; cin >> st[i].pib.pr;
        cout << "Ім'я        : "; cin >> st[i].pib.name;
        cout << "Стать (m/w) : "; cin >> st[i].ch;
        cout << "Курс        : "; cin >> st[i].kurs;
        cout << "Сер. бал    : "; cin >> st[i].bal;
        cout << endl;
    }
```

```
    print(st);
    cout << endl << endl;
    cout << "         На відрахування" << endl;
    cout << "---------------------------------" << endl;
    cout << "        ПІБ      Курс  Бал" << endl;
    cout << "---------------------------------" << endl;
    for (int i = 0; i < n; i++)
        if (st[i].bal < 60)
            cout << setw(10) << st[i].pib.pr << setw(2) <<
st[i].pib.name[0] << "." << setw(6) << st[i].kurs << setw(6) <<
st[i].bal << endl;

    _getch();
    return 0;
}

void print(student a[])
{
    cout << "          Інформація про студентів" << endl;
    cout << "-----------------------------------------------" <<
endl;
    cout << "    Прізвище    Ім'я  Стать  Курс  Бал" << endl;
    cout << "-----------------------------------------------" <<
endl;
    for (int i = 0; i < n; i++)
        cout << setw(10) << a[i].pib.pr << setw(10) <<
a[i].pib.name << setw(5) << a[i].ch << setw(6) << a[i].kurs <<
setw(6) << a[i].bal << endl;
}
```

Структури, що містять покажчики на самих себе, часто використовують при створенні таких структур даних, як зв'язні списки.

11.2. Механізм використання об'єднань

Об'єднання (англ. union) – тип даних користувача, який дуже схожий на структуру, проте тут всі дані займають одну і ту ж область пам'яті. Тому розмір об'єднання дорівнює розміру його найбільшого члена. У будь-який момент часу об'єднання зберігає значення тільки одного з членів. Тобто на якомусь етапі Вам потрібно один тип даних, на іншому – інший. Тому, загалом, об'єднання економить пам'ять комп'ютера від непотрібних на данному етапі змінних.

Оскільки об'єднання зберігає і використовує завжди одне поле їх множини на вибір, то виникає питання про виділення пам'яті під це поле. Тут принцип зрозумілий – вибирається найбільший з типів даних. У мові програмування C++ до об'єднання звертаються так само, як і до структури: через символ "->" при використанні покажчика, або "." при використанні звичайної змінної.

Оголошення об'єднання

Об'єднання складається з декількох змінних, які розділяють між собою одну і ту саму область пам'яті. Це означає, що об'єднання забезпечує можливість

інтерпретації однієї і тієї ж самої конфігурації бітів двома (або більше) різними способами. Оголошення об'єднання, як неважко переконатися на наведеному нижче прикладі, є подібним до оголошення структури:

```
union myUnion { // Попереднє оголошення типу об'єднання
short int c;
char ch;
};
```

У наведеному прикладі оголошується об'єднання, у якому значення типу **short int** і значення типу **char** розділяють одну і ту саму область пам'яті. Необхідно відразу ж з'ясувати один момент: неможливо зробити так, щоб це об'єднання зберігало і цілочисельне значення, і символ одночасно, оскільки змінні c та ch накладаються (у пам'яті) одне на друге. Але програма у будь-який момент може обробляти інформацію, що міститься у цьому об'єднанні, як цілочисельне значення або як символ. Отже, об'єднання забезпечує два (або більше) способи представлення однієї і тієї ж самої множини даних. Як видно з цього прикладу, об'єднання оголошується за допомогою ключового слова **union**.

Як і під час оголошення структур, так і під час оголошення об'єднання не визначається жодна змінна. Змінну можна визначити, розмістивши її ім'я в кінці

оголошення або скориставшись окремою настановою визначення. Щоб визначити змінну об'єднання іменем uVar типу myUnion, достатньо записати:

myUnion uVar;

У змінній об'єднання uVar як змінна с типу **short int**, так і символьна змінна ch розділяють одну і ту саму область пам'яті.

Під час оголошення об'єднання компілятор автоматично виділяє область пам'яті, достатню для зберігання в об'єднанні змінних найбільшого за об'ємом типу.

Щоб отримати доступ до елемента об'єднання, використовують такий самий синтаксис, який застосовується і для структур: оператори "крапка" і "стрілка". При безпосередньому зверненні до об'єднання (або за допомогою посилання) використовують оператор "крапка". Якщо ж доступ до змінної об'єднання здійснюється через покажчик, використовують оператор "стрілка". Наприклад, щоб присвоїти букву "A" елемента ch об'єднання uVar, достатньо використати таку настанову:

uVar.ch = "A";

У наведеному нижче прикладі функції Fun1() передається посилання на об'єднання uVar. У тілі цієї функції за допомогою покажчика змінної с присвоюється значення 10:

```
Fun1(&uVar); // Передаємо функції Fun1() покажчик
// на об'єднання uVar.
//...
}
void Fun1(myUnion *un)
{
un->c =10; // Присвоюємо число 10 члену об'єднання uVar через покажчик.
}
```

Оскільки об'єднання дають змогу складеній програмі інтерпретувати одні і ті ж самі дані по-різному, то вони часто використовують у випадках, коли потрібне незвичайне перетворення типів.

Поняття про анонімні об'єднання

У мові програмування C++ передбачено спеціальний тип об'єднання, який називається *анонімним.* Анонімне об'єднання не має назви типу, і тому об'єкт такого об'єднання визначити неможливо. Але анонімне об'єднання повідомляє компіляторові про те, що його члени розділяють одну і ту саму область пам'яті. При цьому звернення до самих змінних об'єднання відбувається безпосередньо, без використання оператора "крапка".

Приклад *програми на використання анонімного об'єднання:*

```
#include <iostream> // Потокове введення-виведення
#include <conio.h> // Консольний режим роботи
#include <clocale>
using namespace std; // Використання стандартного простору імен
int main()
{
	setlocale(LC_CTYPE, "ukr");
	union { // Попереднє оголошення анонімного об'єднання.
		short int count;
		char ch[2];
	};
	// Ось як відбувається безпосереднє звернення до членів
анонімного об'єднання.
	ch[0] = 'x'; ch[1] = 'y';
	cout << "Об'єднання як символи: " << ch[0] << ch[1] <<
endl;
	cout << "Об'єднання як цілі значення: " << count <<
endl;
	_getch(); return 0;
}
```

Ця програма відображає наступний результат.

Об'єднання у вигляді символів: xy

Об'єднання у вигляді цілого значення: 31096

Число 31096 отримане внаслідок занесення символів x і y в молодший і старший байти змінної *count* відповідно. Як видно, до обох змінних, що входять до складу об'єднання, як *count*, так і ch, можна отримати доступ так само, як до звичайних змінних, а не як до складових об'єднання. Незважаючи на те, що вони оголошені як частина анонімного об'єднання, їх імена знаходяться на тому ж самому рівні області

видимості, що і інші локальні змінні, які оголошено на рівні об'єднання. Таким чином, член анонімного об'єднання не може мати імені, що збігається з іменем будь-якої іншої змінної, оголошеної в тій самій області видимості.

Анонімне об'єднання є засобом, за допомогою якого програміст може повідомити компілятор про свій намір, щоб дві (або більше) змінні розділяли одну і ту саму область пам'яті. За винятком цього моменту, члени анонімного об'єднання поводяться подібно до будь-яких інших змінних.

Механізм використання оператора sizeof для гарантії переносності коду програми

Як було показано вище, структури та об'єднання створюють об'єкти різних розмірів, які залежать від розмірів і кількості їх членів. Окрім цього, розміри таких вбудованих типів як **int** можуть змінюватися при переході від одного комп'ютера до іншого. Іноді компілятор заповнює структуру або об'єднання так, щоб вирівняти їх по границі парного слова або абзацу. Тому, якщо у програмі потрібно визначити розмір (у байтах) структури або об'єднання, використовують оператор **sizeof**. Не намагайтеся вручну виконувати додавання окремих членів. Через заповнення або інші апаратно-залежні чинники розмір структури або

об'єднання може виявитися більшим від суми розмірів окремих їх членів.

Розглянемо такий короткий приклад:

union xUnion { // Попереднє оголошення типу об'єднання

char ch;

int c;

double f;

} uVar; // Визначення змінної об'єднання

Тут у процесі виконання оператора **sizeof** для обчислення розміру об'єднання uVar отримаємо результат 8 (за умови, що **double**-значення займає 8 байтів). У процесі виконання програми немає значення, що реально зберігатиметься в змінній uVar; тут важливим є розмір найбільшої змінної, що входить до складу об'єднання, оскільки об'єднання повинно мати розмір найбільшого його елемента.

1 Абзац містить 16 байтів.

Запитання для самоконтролю

1. Чи можливо у одному масиві зберігати елементи різних типів? А у структурі?
2. Як оголосити структуру в мові С++?
3. Як оголосити змінні типу структури?
4. Як звернутися до значення елемента структури?
5. Як передати структуру у функцію?

6. Як описати масив структур?
7. Для чого використовуються об’єднання?
8. Що таке довжина об’єднання? Як вона обчислюється?
9. Як здійснюється доступ до полів об’єднання?

РОЗДІЛ 12. ЕЛЕМЕНТИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

12.1. Модульне й об'єктно-орієнтоване програмування

Об'єктно-орієнтований підхід до програмування є пріоритетним при створюванні переважної більшості програмних проєктів.

В остаточному вигляді кожна програма є набором інструкцій для процесора. І чим вище є рівень мови, тим простіше записуються одні й ті ж самі дії. З нарощуванням обсягу програм стає потрібним структурувати інформацію, виокремлювати в ній головне та відкидати несуттєве. Цей процес називається *підвищенням ступеня абстракції програми*.

Першим кроком до підвищення абстракції є *використання функцій*, що дозволяє після написання та налагодження функції дистанціюватись від деталей її реалізації, оскільки для виклику функції треба знати лише її інтерфейс.

Наступний крок – *оголошення власних типів даних*, які дозволяють структурувати та групувати інформацію, подаючи її в природному вигляді. Оскільки для роботи з власними типами даних потрібні спеціальні функції, тому вважається за

природне згруповувати їх разом з оголошенням цих типів в одному місці програми і у певний спосіб відокремити від решти її частин.

Отже, *об'єднання в модулі* оголошень типів даних та функцій, призначених для роботи з цими типами, разом із приховуванням від користувача модуля несуттєвих деталей, є подальшим розвитком структуризації програми.

Всі три згаданих вище методи підвищення абстракції ставлять за мету спростити структуру програми, тобто подати її у вигляді невеликої кількості більших блоків та мінімізувати зв'язки між ними. Це дозволяє керувати великим обсягом інформації, а отже, успішно налагоджувати великі програми.

Введення поняття *класу* є розвитком ідей модульності. У класі поєднуються структури даних і функції їхнього опрацювання. Класи дуже схожі на структури. Ідея класів є підґрунтям об'єктно-орієнтованого програмування (ООП). Програми на C++ широко використовують класи.

Клас є типом даних, який визначає користувач. У класі задаються властивості та характер певного предмета чи процесу у вигляді полів даних (аналогічно до структури) і функцій для роботи з ними. Наприклад, клас телефон може мати поля даних:

номер, тип телефону (стаціонарний чи мобільний) і функції роботи з телефоном: дзвінок, набирання номера, з'єднання з абонентом тощо. Групування даних про об'єкт та кодування їх однією змінною спрощує процес програмування та збільшує можливість повторного використання коду.

Суттєвою властивістю класу є те, що деталі його реалізації приховано від користувачів класу за інтерфейсом. Це захищає їх від випадкових змін.

Інтерфейсом класу є заголовки його методів.

Ідея класів відображає будову об'єктів реального світу. Адже кожен об'єкт чи процес має набір певних характеристик чи відмінностей, інакше кажучи, певні властивості й поведінку. Так характеристиками автомобіля є марка, модель, колір, максимальна швидкість, потужність двигуна тощо; характеристиками підсилювача є частотний діапазон, потужність, коефіцієнт нелінійних перекручувань тощо. Функціональні можливості об'єктів теж відрізняються: автомобіль може їздити, посилювач – підвищувати рівень сигналів. А користуватись об'єктами можна, не знаючи їхньої внутрішньої будови. Наприклад, керувати автомобілем можна, не маючи жодного уявлення про будову його двигуна чи будь-яких інших його частин.

Кожен клас займає певне місце в ієрархії класів,

наприклад, усі автомобілі належать до класу наземний транспорт (більш високого в ієрархії), а клас автомобілі містить багато різновидів автомобілів: вантажні, легкові, позашляховики тощо. Отже, будь-який клас визначає певну категорію об'єктів, а кожен об'єкт є екземпляром деякого класу. ООП – це методика, яка концентрує основну увагу програміста на зв'язках поміж об'єктами, а не на деталях їхньої реалізації.

Основними поняттями, на яких базується ООП, є:

- інкапсуляція;
- успадкування;
- поліморфізм.

Інкапсуляцією (encapsulation) називається поєднання даних з функціями їхнього опрацювання разом із приховуванням зайвої для користування цими даними інформації. Інкапсуляція підвищує рівень абстракції програми, дозволяє змінювати реалізацію класу без модифікації основної частини програми та використовувати клас в іншому оточенні.

Успадкування (inheritance) – це можливість створювання ієрархії класів, коли класи-нащадки успадковують властивості своїх базових класів (предків), можуть змінювати ці властивості й набувати нових. Властивості базових класів (предків) при

успадкуванні не описуються, що скорочує обсяг програми.

Поліморфізм (polymorphism) – це можливість використовувати у різних класах ієрархії одне ім'я для позначення близьких за змістом дій та гнучко обирати відповідні дії у перебігу виконування програми.

Отже, об'єктно-орієнтоване програмування у жодний спосіб не пов'язане з процесом виконування програми, а є лише новим способом її організації, тобто новою *парадигмою* програмування (парадигма – набір теорій та методів, які визначають спосіб організації знань).

Існує три групи мов програмування, які пов'язані з поняттям клас: об'єктно-орієнтовані, об'єктні, об'єктно-базовані. *Об'єктно-орієнтовані мови* у повному обсязі підтримують парадигму ООП: інкапсуляцію, успадкування та поліморфізм. Типовими представниками таких мов є C++, Java, C#. До *об'єктних мов* відносять мови програмування, які підтримують тільки інкапсуляцію та дозволяють створювати об'єкти; це мови Visual Basic (до шостої версії включно) та Ada. До *об'єктно-базованих* мов програмування відносять мови, які можуть використовувати існуючі об'єкти, але не мають механізму створення об'єктів користувача. Мова JavaScript є об'єктно-базованою мовою

програмування. Так, за допомогою цієї мови можна використовувати численні об’єкти об’єктної моделі документа (DOM – Document Object Model), за допомогою яких можна задавати зміст веб-сторінки.

Існує помилкова думка, що об’єктно-орієнтоване програмування є щось складне та незрозуміле. Але об’єктна декомпозиція є не менш природною та інтуїтивно зрозумілою, ніж алгоритмічна, яка панувала до появи ООП. До програмування основні поняття ООП перейшли з інших галузей знань, таких як філософія, логіка й математика, причому, без особливих змін, принаймні того, що стосується суттєвості цих понять. Об’єктний спосіб декомпозиції (уявлення) є природним, і використовується протягом багатьох століть. Тому й не дивно, що в процесі еволюції технології програмування цей спосіб посідає гідне місце та підтримується так чи інакше практично всіма сучасними алгоритмічними мовами.

12.2. Визначення класу

Клас є абстрактним типом даних, визначуваним користувачем, і зображує модель реального світу у вигляді даних та функцій їхнього опрацювання. Дані класу називають *полями* чи *даними-членами* (data members), а функції класу – *методами* чи *функціями-*

членами (methods, member functions). Поля та методи називаються елементами класу.

Здебільшого специфікація класу складається з двох частин:

- *оголошення* класу – у ньому прописано всі поля й методи (елементи даних) класу на рівні інтерфейсу в термінах функцій-елементів;
- *визначення* методів класу, тобто реалізації конкретних функцій-членів даного класу.

Оголошення класу має такий формат:

class <*ім'я класу*>
{ [private:]
< *Оголошення прихованих елементів класу* >
protected:
< *Оголошення елементів, доступних тільки нащадкам* >
public:
< *Оголошення загальнодоступних елементів* >
}; // *Оголошення завершується крапкою з комою*

Специфікатори доступу **private**, **protected** та **public** керують видимістю елементів класу. Елементи, визначені після ключового слова **private**, є доступними лише у цьому класі. Цей різновид доступу прийнято у класі за замовчуванням. Специфікатор

доступу **protected** містить поля і методи, які приховані від усіх, окрім нащадків класу. Поля і методи, визначені ключовим словом public, є доступними поза класом, тобто до них можна звертатися безпосередньо з програми, використовуючи об'єкти класу. Вміст загальнодоступного розділу public становить абстрактну частину конструкції, тобто загальнодоступний інтерфейс, який містить прототипи функцій-елементів чи повне визначення (для невеликих функцій).

Розглянемо приклад оголошення класу:

```
class myclass
{ private:     //Прихованні елементи
int a;
public:              //Загальнодоступні елементи
void set_a(int num){a = num;} //Метод, який змінює значення поля
int get_a(){return a;}       //Метод, який повертає значення поля
};
```

У цьому прикладі клас myclass містить лише одне поле даних a, яке має тип int, причому це поле є доступним лише у класі, оскільки воно оголошено з ключовим словом private. Звертання до цього поля з програми неможливе і спричинить помилку. Однак доступ до даних можна організувати за допомогою

методів, які ще називають методами доступу. Такими у класі myclass є два методи: set_a(), який присвоює полю значення, та get_a(), який повертає значення поля. Ці методи є доступними за межами класу, оскільки їх оголошено ключовим словом public. Тіла обох методів (коди функцій) містяться безпосередньо в оголошенні класу. Тут реалізація функцій не означає, що код функції розміщується у пам'яті. Це відбудеться лише при створенні об'єкта класу. Методи класу, визначені у подібний спосіб, за замовчуванням є *вбудованими функціями*.

При збільшенні методів за кількістю та обсягом, визначення функцій класу як вбудованих може спричинити безладдя в оголошенні класу. Щоб уникнути цього, функції всередині класу найчастіше не визначаються, а лише оголошуються (тобто розміщуються їхні прототипи), а саме визначення функцій відбувається в іншому місці. *Функція, визначена поза класом*, за замовчуванням вже не є вбудованою.

Для визначення функції-члена класу поза класом необхідно поєднати ім'я класу з ім'ям функції за допомогою *операції визначення області видимості* "**::**" за форматом:

<тип функції> <ім'я класу>::*< ім'я функції-члена>*

Наприклад, оголосимо клас myclass *без вбудованих функцій-членів класу*:

```
class myclass
{ private:          // Прихованні елементи
int a;
public:            //Загальнодоступні елементи (методи)
void set_a(int num);
int get_a();
};
```

Функції класу тепер можна визначити в іншому місці (чи то після оголошення класу, чи навіть в іншому файлі). Наприклад, реалізація функцій класу myclass поза класом матиме вигляд:

```
void myclass::set_a(int num)
{ a = num; }
int myclass::get_a()
{ return a; }
```

Приклад. *Побудувати клас worker (робітник) без нащадків, який містить оголошення полів даних (name – прізвище, worker_id – код, salary – розмір зарплатні робітника) і визначення методу класу – функцію поза класом (show_worker() – виведення інформації про робітника у консольному режимі):*

```
//…
```

```
class worker
{
private:      // Прихованні поля класу
       char name[64]; long worker_id; float salary;
public:       // Загальнодоступний метод класу
       void show_worker(void);
};
// Реалізація методу класу поза класом
void worker::show_worker(void)
{
       cout << "Прізвище: " << name << endl;
       cout << "Код робітника: " << worker_id << endl;
       cout << "Зарплатня: " << salary << endl;
        };
```

12.3. Створення об'єктів класу

Тепер, коли певний клас визначено, можна створювати конкретні змінні цього класу, які називають *екземпляри класу* чи *об'єкти*. Час їхнього життя та видимість об'єктів залежить від форми та місця їхнього оголошення і підпорядковуються загальним правилам C++. Насправді об'єкт в ООП – це змінна, тип для якої оголошує програміст. Кожен елемент даних такого типу є складеною змінною.

Наприклад, створимо три об'єкти класу myclass:

myclass ob1, ob2, *ob3; // *Об'єкти типу myclass*

Після того, як об'єкти класу створено, можна звертатися до відкритих елементів класу. Використовуючи операцію доступу до члена класу (**.**), звернемося до методів класів об'єктів ob1 та ob2:

ob1.set_a(10); // *Звертання до методу set_a() для об'єкта ob1*

ob2.set_a(99); // *Звертання до методу set_a() для об'єкта ob2*

Тут доступ до елементів об'єкта (у даному разі до методів) є аналогічний доступу до полів структури. Для цього після імені об'єкта (ob1) ставиться крапка (**.**), а після неї зазначається ім'я методу (set_a()). У такий спосіб задається метод та об'єкт застосування цього методу, тобто крапка пов'язує метод з ім'ям об'єкта. Доступ до такого (статичного) члена класу здійснюється за допомогою операції-крапки (**.**). Тобто оператор ob1.set_a(10) викликає метод set_a(int num) об'єкта ob1, який присвоює полю a об'єкта ob1 значення 10. Оператор ob2.set_a(99) присвоює полю a об'єкта ob2 значення 99.

Оператори:

int x1 = ob1.get_a();

int x2 = ob2.get_a();

викликають метод get_a() для об'єктів ob1 та ob2 і присвоюють змінним x1 та x2 значення полів відповідних об'єктів, повернутих цим методом. Як бачимо, дужки після імені методу є обов'язковими, навіть якщо немає параметрів усередині. Дужки "говорять" про те, що здійснюється виклик функції, а не використовується значення змінної.

Для *динамічної змінної* *ob3 необхідно попередньо виділити пам'ять, а вже потім використовувати операцію. Доступ до динамічного члена класу здійснюється за допомогою операції стрілка (->), наприклад:

myclass *ob3 = **new** myclass; // *Виділення пам'яті для об'єкта*

ob3->set_a(46); // *Полю* а *об'єкта* *ob3 *надано значення 46*

Отже, результат виконання розглянутих вище операторів матиме вигляд:

В об'єкті ob1 поле a=10

В об'єкті ob2 поле a=99

В об'єкті ob3 поле a=46

Примітки. Звернутися за допомогою операції крапки (**.**) чи операції (->) можна лише до елементів зі специфікатором public. Здобути доступ чи змінити значення елементів зі специфікаторами private чи protected можна лише через звертання до відповідних методів.

Операція вибору члена (.) працює так само, як операція (->), за винятком того, що імені об'єкта передує неявно згенерований компілятором оператор адреси (&). Отже, інструкцію

ob1.set_a(10);

компілятор трактує як

(&ob1)->set_a(10);

Якщо до закритого методу елемента класу звернутися безпосередньо через об’єкт, а не через методи класу, то така дія призведе до помилки. Наприклад, інструкція ob1.a=10; спричинить помилку (поле а оголошено у розділі private), а якщо елемент класу int a, оголосити у розділі доступу public, то тоді до змінної а можна буде звертатися з будь-якої частини програми, приміром так: ob1.a=10;

12.4. Використання загальнодоступних та приватних елементів класу

Наведемо програму, яка оголошує клас з трьома полями даних (name, worker_id, salary) та функцією-членом (show_worker()), яку реалізовано поза класом. Усі елементи класу оголосимо як загальнодоступні.

У програмі створимо два об’єкти для класу worker, надамо значення елементам даних і за допомогою функції show_worker() виведемо інформацію про робітників у консольному режимі.

Приклад *програми з використанням класу worker:*

```
#include <iostream>
```

```
#include <string.h>
#include <conio.h>
#include <clocale>
using namespace std; // Використання стандартного простору імен

class worker
{
public: // Загальнодоступні елементи класу
	char name[64]; long worker_id; float salary;
	void show_worker(void);
};

// Реалізація функції-члена класу поза класом
void worker::show_worker(void)
{
	cout << "Прізвище: " << name << endl;
	cout << "Код робітника: " << worker_id << endl;
	cout << "Зарплатня: " << salary << endl;
};

// Головна програма – створення та робота з об’єктами класу
int main(void)
{
	setlocale(LC_CTYPE, "ukr");
	worker engineer, boss; // створення об’єктів типу worker
	strcpy_s(engineer.name, "Павленко");
	engineer.worker_id = 345;
	engineer.salary = 5000.00;
	strcpy_s(boss.name, "Марченко");
	boss.worker_id = 101;
	boss.salary = 10500.00;
	engineer.show_worker();
	boss.show_worker();
	_getch();
	return 0;
	}
```

Як бачимо, у головній програмі **main**() оголошено два об’єкти типу worker: engineer та boss, для яких використано операцію крапки при присвоюванні значень елементам об’єкта та при

звертанні до функції show_worker() для виведення результатів.

Результати виконання програми:

```
Прізвище: Павленко
Код робітника: 345
Зарплатня: 5000
Прізвище: Марченко
Код робітника: 101
Зарплатня: 10500
```

Зверніть ще раз увагу на те, що в цьому прикладі всі елементи класу є загальнодоступними, що надало змогу звертатися до всіх елементів класу безпосередньо через об’єкти engineer та boss (engineer.name, boss.show_worker(), boss.salary).

При створенні класу можливо мати елементи, значення яких використовується тільки в класі, але звертатися до яких у самій програмі немає потреби. Такі елементи є приватними (private), їх слід приховувати від програми. Якщо спеціально не використовувати специфікатор public, то за замовчуванням C++ вважає, що всі елементи класу є приватними. Зазвичай, треба захищати елементи класу від прямого доступу до них, оголошуючи для них приватний доступ. Тільки при такому доступі можна

гарантувати користувачеві програмного продукту, що елементам класу буде надано припустимі значення. Приміром, у наведеному прикладі програми об’єкти engineer та boss використовують змінну на ім’я salary, яка може набувати значення тільки в діапазоні від 1000 до 15000. Якщо елемент salary є загальнодоступний як у вищенаведеному прикладі, то програма може безпосередньо звертатися до елемента, змінюючи його значення без обмежень:

engineer.salary = 101; boss.salary = 20000;

Якщо оголосити змінну salary приватною, то для присвоювання значень цій змінній треба створити та використати додатковий метод класу. Наприклад, функція assign_salary() може перевіряти, чи є значення поля salary припустимим.

Приклад *класу з використанням приватної змінної:*

```
class worker
{
private:
	float salary;
public:
	int assign_salary(int value);
};

int worker::assign_salary(int value)
{
	if ((value >= 1000) && (value <= 15000))
	{
		salary = value; return(0);
```

```
}      // Успішне присвоєння
else return(-1);              // Неприпустиме значення
}
```

Методи класу, які керують доступом до елементів даних, – це інтерфейсні функції. При створенні класів бажано використовувати ці інтерфейсні функції для захисту даних своїх класів.

Вищерозглянуті приклади демонструють способи використання методів класу для ініціалізації полів об’єкта класу. Зручнішою є ініціалізація поля об’єкта на момент його створювання, а не явний виклик у програмі відповідного методу. Такий спосіб можна реалізувати за допомогою спеціального методу класу, який називається *конструктором*.

12.5. Конструктори

Властивості конструкторів

При створюванні об’єктів одною з найважливіших операцій, яку виконують в усіх програмах, є ініціалізація елементів даних об’єкта. Для того, щоби звернутися до приватних елементів класу у попередніх прикладах спеціально було створено функції (assign_salary(), set_a(), get_a()). Для спрощення процесу ініціалізації елементів класу, алгоритмічна мова C++ має спеціальну функцію-член,

яку називають конструктор.

Конструктор (constructor) – це спеціальна функція (метод) класу, яка автоматично виконується при створюванні об'єкта та присвоює значення елементам класу для об'єкта, що створюється. Він може не лише ініціалізовувати змінні класу, а й виконувати будь-які інші завдання ініціалізації, необхідні для підготовки об'єкта до використання (наприклад, перевірку припустимості значень). Конструктор пов'язує ім'я класу з його закритими для доступу полями за форматом:

<ім'я класу>(*<змінна поля1>*, *<змінна поля2>*, . . .)

Приклад *оголошення класу worker з конструктором:*

```
class worker // Оголошення імені класу
{
private:      // Прихованні поля класу
       char name[64];
       long worker_id;
       float salary;
public:       //Загальнодоступні методи класу
       worker(char* iname, long iworker_id, float isalary);//
Конструктор
       void show_worker(void); // Метод для виведення
інформації
};
```

У цьому конструкторі будуть ініціалізуватись

змінні (поля) з розділу доступу **private**: name, worker_id, salary. Щоби відрізняти відповідну змінну від подібного імені поля класу в оголошенні конструктора до імені полів класу було долучено літеру “i” (**i**name, **i**worker_id, **i**salary)

Для *створення екземпляра класу* необхідно, щоби конструктор було оголошено як загальнодоступний (**public**). При оголошенні об’єкта класу значення його полів передаються конструкторові за форматом:

<ім’я класу> <ім’я об’єкта>(*<значення поля1>*,*< значення поля2>*, . . .);

Наприклад, використовуючи оголошення конструктора класу worker, створимо об’єкт engineer:

worker engineer ("Павленко", 345, 5000.00);

Використання конструктора у наведеному прикладі зменшило кількість операторів ініціалізації полів об’єкта класу, причому поля даних є захищені від загального доступу.

Розглянемо *основні властивості* конструктора.

- Конструктор *виконується автоматично* в момент створення об’єкта, чим спрощує задачу ініціалізації даних об’єкта.
- Ім’я конструктора абсолютно *збігається з*

іменем класу. Отже, компілятор відрізняє конструктор від інших методів класу.

- Конструктор *не повертає жодного значення*, навіть типу void. Не можна здобути і вказівник на конструктор. Це пояснюється тим, що конструктор автоматично викликається системою, а отже, не існує програми чи функції, яка його викликає, і якій конструктор міг би повернути значення. Отже, задавати значення, яке повертається, для конструктора немає сенсу.
- Клас може мати *кілька конструкторів з різними параметрами* для різних видів ініціалізації, при цьому використовується механізм перевантаження.
- Конструктор може прийняти *яку завгодно кількість аргументів* (включаючи нульову).
- *Параметри конструктора можуть бути якого завгодно типу*, окрім цього класу. Можна задавати значення параметрів за замовчуванням. Їх може містити лише один із конструкторів.
- Конструктори *не успадковуються*.
- Конструктори *не можна оголошувати з модифікаторами* const, virtual та static.
- Знищення об’єкта з пам’яті комп‟ютера

виконує спеціальна функція – *деструктор.*

Розглянемо різні способи визначення конструкторів.

Конструктор з параметрами

Конструктор може ініціалізувати поля класу, використовуючи оператор присвоювання за форматом:

<ім'я класу>(*<змінна поля1>*,*< змінна поля2>*, …)

{*<ім'я поля1>* = *< змінна поля1>*; *<ім'я поля2>* = *< змінна поля2>*; . . . }

Наприклад, визначимо конструктор з операторами присвоєння для класу worker (як вбудовану функцію):

```
worker (char *iname, long iworker_id, float isalary)
  { name = iname; worker_id = iworker_id;salary =
                        isalary; }
```

У цьому разі для створення об'єкта викликається конструктор, до якого передаються відповідні значення параметрів:

```
worker boss ("Марченко",101,10500.00);
```

За цього способу визначення конструктора легко реалізувати умови на припустимість значень. Наприклад, при створенні об'єкта завжди будемо

перевіряти: чи належить значення поля salary діапазону від 1000 до 15000:

```
worker (char *iname, long iworker_id, float isalary)
{ name = iname;
worker_id = iworker_id;
if ((isalary>=1000) && (isalary<=15000)) salary = isalary;
        else salary = 0;   // Якщо значення є неприпустиме, присвоюємо 0
}
```

Окрім того, за такого способу *створення екземпляра класу з вказівником*, значення параметрів можна передавати до конструктора, використовуючи оператор new. Наприклад, створимо динамічний об’єкт *menedger:

```
worker *menedger = new worker("Зименко", 223, 7500.00);
```

Конструктор зі списком ініціалізації

Вищеописаний конструктор не може бути використаним при ініціалізації полів-констант чи полів-посилань, оскільки їм не можуть бути присвоєні значення. Для цього передбачено спеціальну властивість конструктора, називану *списком ініціалізації*, який дозволяє ініціалізувати одну чи більше змінних і не надавати їм значення. Список

ініціалізації розташовується поміж прототипом методу та його тілом і після двокрапки, при цьому ініціалізуюче значення розміщується у дужках після імені поля, значення у списку розділяються комами за форматом:

<ім'я класу>():*<ім'я поля1>*(*<значення поля1>*),
<ім'я поля2>(*<значення поля2>*), . . . { . . . }

Наприклад, конструктор зі списком ініціалізації усіх полів класу worker матиме вигляд:

worker():name("Марченко"),worker_id(101),salary (10500.00){}

Такий конструктор називають *конструктором без параметрів*. Для ство- рення об'єктів з використанням такого конструктора достатньо записати ім'я класу та імена об'єктів через кому, наприклад:

worker w1, w2, w3;

Тут усі створювані об'єкти (w1, w2, w3) матимуть такі само початкові значен- ня, як у об'єкта boss.

При створенні об'єктів поля можна також ініціалізувати за допомогою списку змінних поля, в якому значення можуть бути виразами, наприклад:

worker (char *iname, long iworker_id, float isalary):
name(iname), worker_id(iworker_id),

salary(isalary) {}

При створюванні цього об’єкта буде викликано конструктор з відповідними, зазначеними у дужках параметрами, приміром:

worker w4("Ткаченко", 454, 6000.00);

Можливо також використовувати комбінацію способів визначення конструкторів зі списками ініціалізації та змінних поля, наприклад:

worker(char *iname, long worker_id):
name(*iname), worker_id(worker_id),
salary(1000.00) {}

Цей конструктор має тільки дві змінні поля та одне константне значення (salary = 1000.00). Створення об’єкта (наприклад, w5) за таким конструктором матиме вигляд:

worker w5("Соловейко",255);

Конструктор за замовчуванням

Конструктор без параметрів називають *конструктором за замовчуванням*. Такий конструктор зазвичай ініціалізує поля класу константними значеннями, як для об’єктів w1, w2, w3, які розглянуто вище. Якщо конструктор для будь-якого класу не визначено, то компілятор сам генерує конструктор за замовчуванням.

Якщо програмістові треба однозначно

ініціалізувати поля, треба визначити власний конструктор (ним може бути і конструктор за замовчуванням). Тоді для класу, який має конструктор за замовчуванням, можна визначити об’єкт класу без передавання параметрів, наприклад:

worker obj1, obj2;

У класі може бути визначено не один конструктор з одним і тим самим іменем. Тоді говорять, що *конструктор є перевантаженим*. Який з них буде виконуватись при створюванні об’єктів, залежить від того, скільки аргументів використовується у виклику. *Якщо у класі визначено будь-який конструктор, то компілятор не створить конструктора за замовчуванням.* І якщо у класі не буде конструктора за замовчуванням, у певних ситуаціях можуть виникати помилки. У такому разі слід визначити свій власний конструктор за замовчуванням, наприклад:

worker ():name (""), worker_id (0), salary (0) {}

Примітка. Конструктори за замовчуванням не присвоюють початкових значень полям класу.

Конструктор копіювання

Вище розглянуто два види конструкторів – конструктор без аргументів, який ініціалізує поля об’єкта константними значеннями, та конструктор,

який має хоча б один аргумент, який ініціалізує поля значеннями, переданими йому в якості аргументів.

Тепер розглянемо ще один спосіб ініціалізації об’єкта, який використовує значення полів вже існуючого об’єкта. Такий конструктор не треба самим створювати, він надається компілятором для кожного створюваного класу і називається *конструктором копіювання за замовчуванням*. Він може мати лише один аргумент, який є об’єктом того ж самого класу.

Наприклад, створимо три об’єкти work1, work2, work3 у різні способи:

worker work1("Серченко", 300, 5000.00); // *Створення об’єкта* work1

worker work2(work1); // *Об’єкт* work2– *копія об’єкта* work1

work3 = work1; // *Об’єкт* work3 *дорівнює об’єктові* work1

Тут ініціалізовано об’єкт work1 значеннями ("Серченко", 300, 5000.00) за допомогою конструктора з трьома параметрами. Потім визначено ще два об’єкти класу з іменами work2 та work3, які ініціалізуються значеннями об’єкта work1. Для копіювання значень полів об’єкта work1 у відповідні поля об’єктів work2 та work3 двічі викликається конструктор копіювання. У результаті всі три об’єкти матимуть однакові значення, ідентичні значенням

об’єкта work1.

Примітка. Якщо клас містить *вказівники чи посилання*, виклик конструктора копіювання призведе до того, що й копія, й оригінал вказуватимуть на одну й ту саму ділянку пам’яті. У такому разі конструктор копіювання має бути створений програмістом у такому вигляді:

<*ім’я класу*> :: <*ім’я класу*> (const <*ім’я класу*> &) {...}

12.6. Деструктори

Деструктор – це особлива форма методу класу, який застосовується для звільнення пам’яті, зайнятої об’єктом. Деструктор за суттю є антиподом конструктора. Якщо конструктор – це функція, яка допомагає будувати (конструювати) об’єкт, то деструктор являє собою функцію, яка допомагає знищувати об’єкт, тобто звільняти від нього пам’ять. Деструктор викликається автоматично, коли об’єкт виходить за межі області видимості.

Деструктор при визначенні класу має такий формат:

~ <*ім’я класу*> () { <*оператори деструктора*> }

Розглянемо *основні властивості* деструктора.

- Ім’я деструктора розпочинається з тільди (~), безпосередньо за якою йде ім’я класу.
- Деструктор – це метод, який *виконується автоматично.*
- Конструктор *не повертає жодного значення*, навіть типу void.
- Деструктор *не має аргументів*.
- Вказівник на деструктор визначити не можна.
- Деструктор *не успадковується*.
- Якщо деструктор явно не визначено, *компілятор автоматично створює порожній деструктор.*
- *По завершенні програми об’єкти усіх класів знищуються* з пам’яті комп’ютера навіть тоді, коли деструктор явно не визначено.

У попередніх програмах об’єкти створювалися після їхнього оголошення. По завершенні програми C++ автоматично викликався деструктор для кожного об’єкта, хоча його не було явно визначено у програмі.

Деструктор, окрім деініціалізації об’єктів, може виконувати деякі дії, наприклад, виведення остаточних значень елементів даних класу, що буває зручно при налагодженні програми. Як приклад, визначимо клас worker з деструктором та конструктором:

```
class worker
{ private:                          // Приховані поля класу
char name[64] ;
long worker_id;
float salary;
        public:                     //
    Загальнодоступні методи класу
worker(char *iname, long iworker_id, float isalary); //конструктор
void ~worker(void);                 // Деструктор
void show_worker(void); // Функція виведення інформації
};
//Реалізація деструктора для класу worker (поза класом):
void worker::~worker(void)
{ cout << "Знищення об’єкта для " << name << endl; }
```

Цей деструктор у консольному режимі виведе на екран значення елемента класу name та повідомлення про те, що C++ знищує цей об’єкт. Наприклад, створимо об’єкт для визначеного вище класу:

```
void main(void)
{ worker worker("Василенко", 777, 10101.0);
                worker.show_worker();
```

}

У результаті виконання цієї програми на екрані побачимо:

Прізвище: Василенко

Код робітника: 777

Зарплатня: 10101.00

Знищення об’єкта для Василенко

Явно розміщувати деструктор у класі треба у разі, якщо об’єкт містить вказівники на пам’ять, виділену динамічно, – інакше при знищуванні об’єкта пам’ять, на яку посилались його поля-вказівники, не буде позначено як звільнену.

Приклад *класу з використанням конструктора та деструктора:*

```
class CMyClass
{ public:
CMyClass();   //Конструктор класу CMyClass
~CMyClass();  //Деструктор класу CMyClass
private:
int MyInt;    //Змінна типу int (ціле число)
int* point;   //Змінна типу вказівник на int (ціле число)
};

CMyClass::CMyClass()//Конструктор
{
       MyInt = 10;   //На етапі ініціалізації об’єкта класу
CMyClass
 //присвоюється змінній цього об’єкта MyInt значення 10
       point = new int;    //Виділення пам’яті під ціле число
для вказівника
       *point = 20;  //Записування у виділену пам’ять числа 20
}
```

```
CMyClass::~CMyClass()//Деструктор
{
	MyInt = 0;	// Об’єкт класу CMyClass фактично припинив
своє існування,
// але надамо змінній класу MyInt значення 0
	delete point; // Використаємо вказівник на число для
того,
	// щоби звільнити пам’ять, яку виділено під це число
	// (Автоматично деструктор не виконує цього!)
	}
```

12.7. Успадкування

Успадкування (inheritance) – це механізм, за допомогою якого один об’єкт може набувати властивості іншого. Точніше, об’єкт може успадковувати основні властивості іншого об’єкта та набувати нових рис, які характерні лише для нього. Успадкування є дуже важливим, оскільки воно дозволяє підтримувати принцип *ієрархії класів* (hierarchical classification). Наприклад, клас мобільних телефонів є підкласом класу “Телефон”, який, у свою чергу, входить до ще більшого класу “Електрозв’язок”. Разом з тим, клас “Електрозв’язок” є підкласом класу “Способи зв’язку”, до складу якого, крім електрозв’язку, входять супутниковий зв’язок, радіозв’язок, поштовий зв’язок тощо (рис. 12.1). Застосування ієрархії класів дозволяє керувати великими потоками інформації. Прикладом подібної ієрархії є системи класифікації в ботаніці, зоології

тощо.

Рис. 12.1. *Ієрархія класів "Способи зв'язку"*

В об'єктно-орієнтованому програмуванні *успадкування* – це процес створювання нових класів, які називають *похідними класами* (*нащадками*) на базі вже існуючих *батьківських* (*базових*) *класів*. Похідний клас успадковує всі можливості батьківського (базового) класу, але може бути удосконаленим за рахунок *змінювання* існуючих методів і *долучення* нових власних полів та методів. Батьківський клас (чи клас вищого рівня) при цьому залишається незмінним. Похідний клас (клас нижчого рівня), у свою чергу, сам може слугувати за батьківський.

Найважливішою позитивною якістю успадкування в ООП є те, що воно дає можливість

уникати повторювань програмного коду для кожного об'єкта, адже спільний для множини подібних класів код може бути винесено до методів їхнього спільного батьківського класу. Це є доволі зручний спосіб, оскільки програміст може використовувати класи, створені будь-ким іншим, без модифікації коду, лише створюючи похідні класи.

Успадкування буває простим і множинним. *Простим* називається успадкування, за якого похідний клас має один батьківський клас. *Множинне* успадкування означає, що клас має кілька батьківських класів, і застосовується для того, щоб забезпечити похідний клас їхніми властивостями. При застосовуванні множинного успадкування треба ретельно стежити за тим, щоби похідний клас не успадкував поля чи методи з однаковими іменами, але різні за змістом.

Для *створювання похідного класу* використовують ключове слово **class**, після якого записують ім'я нового класу, двокрапку (**:**), *ключ доступу класу* (**public**, **private**, **protected**), а потім зазначають ім'я батьківського класу:

class
<*ім'я_похідного_класу*>:[<*ключ_доступу*>]<*ім'я_батьківського_класу*>

{ <*тіло_класу*> };

Вище розглядались лише специфікатори доступу **private** (закритий) та **public** (відкритий), які застосовуються до полів класу. Однак члени базового класу, оголошені як private, у похідному класі є недоступні незалежно від ключа доступу. Звертання до них може відбуватися лише через методи базового класу. Для базових класів можливе використання ще одного специфікатора – **protected** (захищений), який для поодиноких класів, що не входять до ієрархії, означає те ж саме, що й private.

Ключ доступу **public** означає, що всі відкриті й захищені члени базового класу стають такими для похідного класу. Ключ доступу private означає, що всі відкриті й захищені члени базового класу стають закритими членами для похідного класу. Ключ доступу protected означає, що всі відкриті й захищені члени базового класу стають захищеними членами похідного класу.

12.8. Поліморфізм

При успадкуванні деякі методи класу мають можливість бути замінені на інші. Так, батьківський клас способів зв’язку (див. рис. 12.1) матиме

узагальнений метод – спосіб передавання інформації. У похідних класах цей метод буде уточнено: радіозв'язок – це передавання радіосигналів від радіостанції до радіоприймачів, поштовий зв'язок – це перевезення транспортом поштових відправлень (посилок, бандеролей, листів тощо), супутниковий зв'язок – це передавання сигналів на супутники та прийом цих сигналів супутниковими антенами. Отже, одне ім'я методу використовується для розв'язання декількох схожих, але технічно різних задач. Таке змінювання змісту методу називається поліморфізмом. Взагалі *поліморфізм* (polymorphism) (від грецької – polymorphos) – це здатність об'єкта змінювати форму (poly означає багато, а morphism має відношення до змінювання форми). Отже поліморфний об'єкт, являє собою об'єкт, який може набувати різноманітних форм.

Мета поліморфізму в об'єктно-орієнтованому програмуванні – використання одного імені для схожих дій (методів) класу. Виконання кожної конкретної дії буде визначено типом даних. У різних мовах програмування поліморфізм реалізовано різноманітними засобами. Наприклад, у Pascal та C++ його реалізовано за допомогою механізму віртуальних функцій. Разом з тим, у мові C++ поліморфізм підтримується недостатньо, наприклад, обчислення

абсолютного значення змінної можна виконати трьома функціями: abs(), labs() та fabs(). Ці функції обчислюють та повертають абсолютне значення змінних цілого, довгого цілого та дійсного типів відповідно. У Pascal така задача виконується однією функцією abs(). У мові C++ вибір конкретної функції для цієї задачі здійснює програміст відповідно до типу даних.

Поліморфізм може також застосовуватись до операторів. Фактично в усіх мовах програмування обмежено використовується поліморфізм в арифметичних операціях. Так, у мові C++, символ плюс (+) використовується для додавання цілих, довгих цілих, дійсних чисел, а також для символьних змінних та рядків. У такому разі компілятор автоматично визначає, який тип арифметики слід застосувати.

При роботі з типом даних class у C++ є можливість застосовувати механізм поліморфізму, тобто одне ім’я методу класу використовувати для множини різноманітних дій. Перевагою поліморфізму є те, що він допомагає зменшити складність програм. Вибір конкретної дії, відповідно до ситуації, покладено на компілятор, і програмістові не треба вибирати самому. Необхідно лише пам’ятати та використовувати загальний інтерфейс. Приклад з

функцією обчислення абсолютного значення змінної, показує, як за наявності трьох імен у мові С++ замість одного, будь-яка задача стає більш складною, аніж це дійсно потрібно.

Щоб використовувати поліморфізм, треба забезпечити виконання певних умов. По-перше, всі похідні класи мають бути нащадками одного й того самого базового класу. По-друге, методи, які забезпечують поліморфізм (назвемо їх поліморфними методами), мають бути оголошені в батьківському класі як віртуальні. Віртуальний (virtual) – означає видимий, але неіснуючий у реальності. *Віртуальна функція* – це функція базового класу, перед прототипом якої стоїть ключове слово virtual за форматом:

virtual <*тип*> <*ім'я_функції*>
(<*список_параметрів*>);

Віртуальні функції не визначаються у батьківському класі, до оголошення цього класу записують лише прототипи цих функцій із ключовим словом virtual перед ними. Похідний клас перевизначає ці функції, пристосовуючи їх для власних потреб. І, якщо функції було оголошено як віртуальні, вони залишатимуться віртуальними й в

усіх похідних класах. Проте, зазвичай, надають перевагу зберіганню цього специфікатора і в класах-нащадках для більшої зрозумілості суті цих класів.

Базовий клас здебільшого буває *абстрактним класом*, об'єкти якого ніколи не буде реалізовано. Такий клас існує з єдиною метою – бути батьківським відносно похідних класів, об'єкти яких буде реалізовано, для створювання ієрархічної структури. Наприклад, клас "Способи зв'язку" має метод "Засоби передавання інформації", який не може бути реалізовано для об'єктів цього класу, а в похідних класах цей метод має конкретне значення: для класу "Радіозв'язок" – це радіосигнал, для класу "Поштовий зв'язок" – це транспорт для перевезення поштових відправлень тощо. Але, якщо об'єкти батьківського (абстрактного) класу не призначені для реалізації, то в який спосіб захистити базовий клас від використання не за призначенням? – Захистити його треба програмно. Для цього достатньо ввести у клас хоча б одну *суто віртуальну функцію* (pure virtual function).

Для суто віртуальної функції використовується формат:

virtual <*тип*> <*ім'я_функції*>
(<*список_параметрі в* >) **= 0**;

Ключовою частиною цього оголошення є присвоєння суто віртуальній функції значення нуль. Це повідомляє компіляторові, що в батьківському класі немає тіла функції. Якщо функцію задано як суто віртуальну, то це означає, що вона обов'язково повинна бути заміненою в кожному похідному класі, інакше при компіляції виникне помилка. Отже, створення суто віртуальних функцій – це гарантія того, що похідні класи забезпечать їхнє перевизначення. Якщо клас містить хоча б одну суто віртуальну функцію, то його називають *абстрактним класом*. Оскільки в абстрактному класі є хоча б одна функція, в якої відсутнє тіло функції, технічно такий клас не є повністю визначений, і для нього неможливо створити жодного об'єкта. Тому абстрактні класи можуть бути лише похідними.

Основні концепції поліморфізму в мові програмування С++:

- поліморфізм – це властивість об'єкта змінювати форму під час виконання програми;
- для створення методів, які є поліморфними, у програмі необхідно використовувати віртуальні (virtual) функції;
- якщо об'єкти батьківського (абстрактного) класу не призначені для реалізації, необхідно ввести у клас хоча б одну суто віртуальну

функцію;

- будь-який клас, похідний від батьківського, має можливість використовувати чи перевантажувати віртуальні функції;
- для створення поліморфного об’єкта в С++ слід використовувати вказівник на об’єкт батьківського класу;
- поліморфізм спрощує програмування та створення поліморфних об’єктів і зменшує складність програм.

Розглянемо механізм поліморфізму на прикладах.

Приклад. *Створити поліморфний клас – телефон, який відображує (імітує) телефонні операції: набирання номера, дзвінок, роз’єднання та індикацію зайнятості лінії зв’язку. Передбачити у програмі можливість імітувати, за бажанням, роботу дискового, кнопкового чи мобільного телефону. Тобто об’єкти класу мають бути поліморфними – від одного дзвінка до іншого об’єкт-телефон має змінювати форму (своє призначення).*

Розв’язок. Створимо батьківський клас phone – дисковий телефон з полем number – номером телефону і методами: dial – набирання номера, ring – дзвінок, answer – очікування відповіді, hangup – роз’єднання.

Далі створимо два похідні класи: touch_tone – кнопковий телефон та mobile_phone – мобільний телефон, причому в цих класах будуть визначені свої власні методи dial.

Код програми з виведенням результатів у консольному режимі має такий вигляд:

```
#include <iostream>
#include <cstring>
#include <conio.h>
#include <clocale>
using namespace std; // Використання стандартного простору імен
class phone  // Оголошення батьківського класу – дисковий
телефон
{
protected:
	char number[13];
public:
	virtual void dial(char* number)   // Віртуальна функція
	{
		cout << "Набирання номера " << number << endl;
	}
	void answer(void)
	{
		cout << "Очікування відповіді" << endl;
	}
	void hangup(void)
	{
		cout << "Дзвінок виконано – покладіть слухавку "
<< endl;
	}
	void ring(void)
	{
		cout << " Дзвінок, дзвінок, дзвінок " << endl;
	}
	phone(char* number)
	{
		strcpy_s(phone::number, number);
	};	// Конструктор
};
```

```
// Оголошення похідного класу – кнопковий телефон
class touch_tone : phone
{
public:
	void dial(char* number)
	{
		cout << "Пік, пік. Набирання номера " << number
<< endl;
	}
	touch_tone(char* number) :phone(number) { } //
Конструктор
};
// Оголошення похідного класу – мобільний телефон
class mobile_phone : phone
{
private:
	int amount;
public:
	void dial(char* number)
	{
		cout << "Будь ласка, сплатіть " << amount << "
копійок" << endl;
		cout << "Набирання номера " << number << endl;
	}
	mobile_phone(char* number, int amount) : // Конструктор
		phone(number) {
		mobile_phone::amount = amount;
	}
};

// Головна програма – створення та робота з об’єктами класу
int main()
{
	setlocale(LC_CTYPE, "ukr");
	char  s[] = "201-555-1212",
		s1[] = "602-555-1212",
		s2[] = "555-1212",
		s3[] = "818-555-1212",
		s4[] = "303-555-1212",
		s5[] = "212-555-1212",
		s6[] = "702-555-1212";
	strcpy_s(s, "201-555-1212");
  // Створення об’єкта rotary – дисковий телефон
	phone rotary(s); rotary.dial(s1);
```

```
  // Створення об’єкта home_phone – кнопковий телефон
       touch_tone home_phone(s2);
       home_phone.dial(s2);
       // Створення об’єкта city_phone – платний телефон
       mobile_phone city_phone(s6, 25); city_phone.dial(s5);
       cout << "     Поліморфні об’єкти: " << endl;
       // Створення об’єкта-вказівника на батьківський клас –
дисковий телефон
       phone* poly_phone = &rotary;
       poly_phone->dial(s3);
       // Змінення форми об’єкта на кнопковий телефон
       poly_phone = (phone *) &home_phone; poly_phone-
>dial(s4);
       // Змінювання форми об’єкта на платний телефон
       poly_phone = (phone *) &city_phone; poly_phone-
>dial(s5);
       _getch();
       return 0;
}
```

Результати виконання програми:

Набирання номера 602-555-1212

Пік, пік. Набирання номера 212-555-1212

Будь ласка, сплатіть 25 копійок

Набирання номера 212-555-1212

Поліморфні об’єкти: Набирання номера 818-555-1212

Пік, пік. Набирання номера 303-555-1212

Будь ласка, сплатіть 25 копійок

Набирання номера 212-555-1212

Наведені класи телефону мають однакову назву, але різну за виконанням функцію dial(). Цю функцію було визначено як віртуальну (з ключовим словом

virtual) у батьківському класі phone.

Для створення поліморфного об’єкта у програмі використано вказівник на об’єкт батьківського класу (phone *poly_phone). Для змінювання форми об’єкта цьому вказівникові надано адресу об’єкта похідного класу:

poly_phone = (phone *) &home_phone;

Вираз у круглих дужках (phone *), записаний за оператором присвоєння, є операцією зведення типів, яка сповіщає компілятору C++, що вказівнику на змінну одного типу (phone) треба надати адресу змінної іншого типу (home_phone). Оскільки програма присвоює вказівнику об’єкта poly_phone адреси різних об’єктів, то цей об’єкт може змінювати свою форму, а, отже, він є поліморфним. Згідно з програмою, об’єкт poly_phone змінює форму з дискового телефону на кнопковий, а після цього – на платний.

Запитання для самоконтролю

1. Що таке клас в C++?
2. Який синтаксис опису класу?
3. Що таке поля, методи класу?
4. Які є специфікатори доступу класу, їхнє призначення?

5. Як можна забезпечити доступ до елементів класу?
6. Що таке конструктор? Які правила створення та роботи конструктора ?
7. Що таке конструктор позамовчунні, конструктор копіювання, конструктор перетворення?
8. Що таке деструктор? Які правила створення та роботи деструктора ?
9. Які існують ініціалізації елементів у конструкторах?
10. Який порядок виклику конструкторів та деструкторів?

РОЗДІЛ 13. ЕЛЕМЕНТИ ВІЗУАЛЬНОГО ПРОГРАМУВАННЯ

13.1. Вступ у візуальне програмування Visual Studio C++

Основні поняття. Технологія роботи у середовищі Visual Studio C++ базується на ідеях об'єктно-орієнтованого та візуального програмування. Ідея об'єктно-орієнтованого програмування полягає в інкапсуляції (об'єднанні) даних і засобів їх опрацювання (методів) у тип, який називається класом. Конкретною змінною певного класу і є об'єкт. Прикладами об'єктів можуть бути елементи керування у вікні: кнопки, списки, текстові поля тощо.

Середовище візуального програмування Visual Studio C++ – це графічна автоматизована оболонка над об'єктно-орієнтованою мовою програмування C++ . Якщо у мові Сі структурними одиницями є дані та команди, то тут такою структурною одиницею є візуальний об'єкт, який називається ***компонентом***. Автоматизація програмування досягається завдяки можливості переносити компонент на форму (у програму) з панелі елементів і змінювати його властивості, не вносячи вручну змін до програмного коду.

Формою називають компонент, який володіє

властивостями вікна Windows і призначений для розташування інших компонентів. Компоненти на формі можуть бути видимими (візуальні) та невидимими (невізуальні). Видимі призначені для організації діалогу з користувачем. Це різні кнопки, списки, текстові поля, зображення тощо. Вони відображаються на екрані під час виконання програми. Невидимі компоненти з'являються на формі під час проєктування як піктограми. Вони не видні під час виконання, але мають певне функціональне навантаження (призначені для доступу до системних ресурсів комп'ютера).

Проєкт – це сукупність файлів, з яких складається С++-програма.

Інструменти середовища Visual Studio C++. Вікно середовища містить головне меню, панелі інструментів, а також:

- панель елементів (палітра компонентів);
- вікно властивостей об'єктів;
- вікно форми;
- редактор коду програми.

Ці інструменти стають доступними після запуску Visual Studio C++: два знаходяться у головному вікні (верхня частина екрана, рис. 13.1), а решта – в окремих вікнах. Ці вікна також можна

відкрити командами головного меню *Вид* (або комбінацією клавіш **Ctrl+Alt+X**); вікно *Свойства* – меню *Вид* → *Другие окна* → *Окно свойств* (**Alt+ENTER**)

Головне меню та панель інструментів. Головне меню складається з таких елементів: *Файл*, *Правка*, *Вид*, *Проєкт*, *Сборка*, *Отладка*, *Тест*, *Анализ*, *Средства*, *Расширения*, *Окно*, *Справка*, *Поиск* (див. рис. 13.1).

Меню *Файл* містить стандартні команди для роботи з файлами проєкту. За допомогою цих команд можна створити проєкт, новий файл, відкрити чи закрити файл чи проєкт, зберегти файл або все відразу.

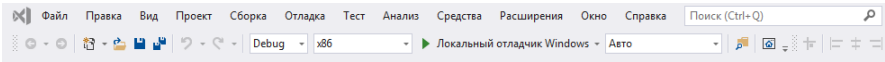


Рис. 13.1. *Головне вікно Visual Studio C++*

Характеристика основних компонентів середовища

Кожен компонент має три різновиди характеристик: властивості, події і методи.

Властивості є атрибутами компоненту, що визначають його зовнішній вигляд і поведінку. Багато властивостей компоненту мають значення, що

встановлюється за замовчуванням (наприклад, висота кнопок). Властивості компоненту відображаються на сторінці властивостей. Можна визначити властивості під час проєктування або написати код для їх видозміни під час виконання додатку (видимий чи невидимий компонент в залежності від виконання умови).

Сторінка **подій** показує список подій, розпізнаваних компонентом (програмування для операційних систем з графічним інтерфейсом користувача). Кожен компонент має свій власний набір обробників подій. Створюючи обробник події, ви доручаєте програмі виконати написану функцію, якщо ця подія відбудеться.

Метод є функцією, яка пов'язана з компонентом і яка оголошується як частина об'єкту.

У бібліотеці візуальних компонентів існує багато компонентів, що дозволяють відображати, вводити та редагувати текстову інформацію.

Вікно форми. Форма – це вікно Windows, яке утворюється в одному з можливих для вікон стилів. Увесь внутрішній простір є робочою областю. Для додавання компоненту у форму можна вибрати мишею компонент в палітрі (рис. 13.2) та клацнути лівою клавішею миші в потрібному місці проєктованої форми. Компонент з'явиться на формі, і далі його

можна переміщати та встановлювати характеристики.

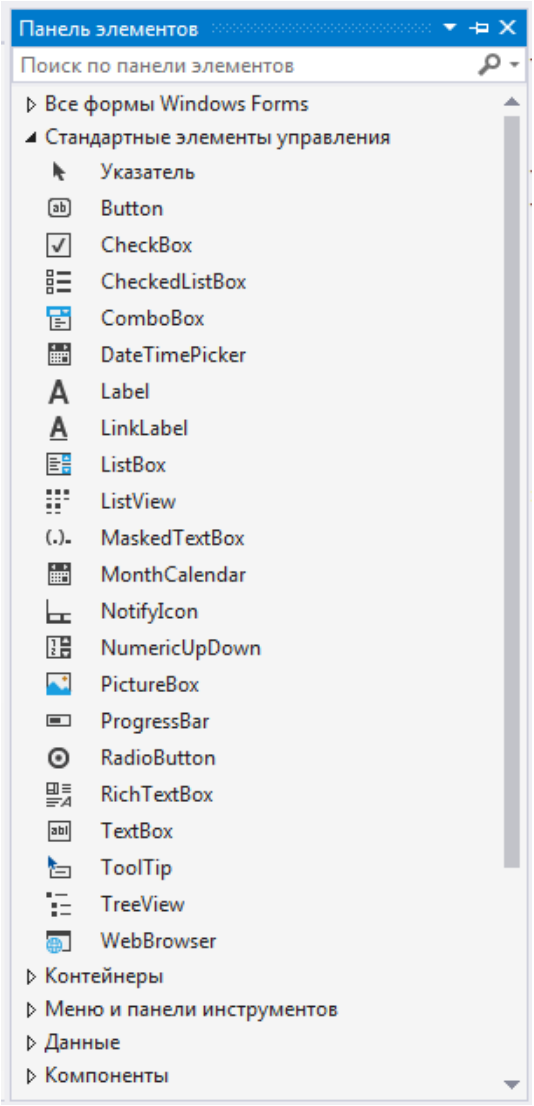


***Рис. 13.2** Панель елементів*

Для виконання групових операцій декілька компонентів можна об'єднувати. Для цього необхідно натиснути на ліву клавішу миші і переміщенням вказівника охопити всі потрібні компоненти. У групу долучаються компоненти, які хоча б частково потрапляють в охоплену область. Можна також долучити/вилучити окремий елемент. Для цього необхідно натиснути на клавішу Shift та, не відпускаючи її, вибрати мишею потрібний компонент на формі. Вилучення виокремлених компонентів чи групи виконується клавішею Delete. Переміщення виокремленого компонента в межах форми здійснюється як мишею так і відповідними клавішами клавіатури. Над компонентами та їхніми групами можна виконувати операції вирізання, копіювання в буфер обміну та вставляння з буфера.

Властивості компонентів. Вікно властивостей об'єктів (рис. 13.3) складається з двох стовпців: лівий містить назви властивостей компонентів, а правий – їхні значення. Властивості можуть бути простими або комплексними. Комплексні властивості складаються з набору інших властивостей. Такі властивості в інспекторі об'єктів позначені символом "+", наприклад +Font.

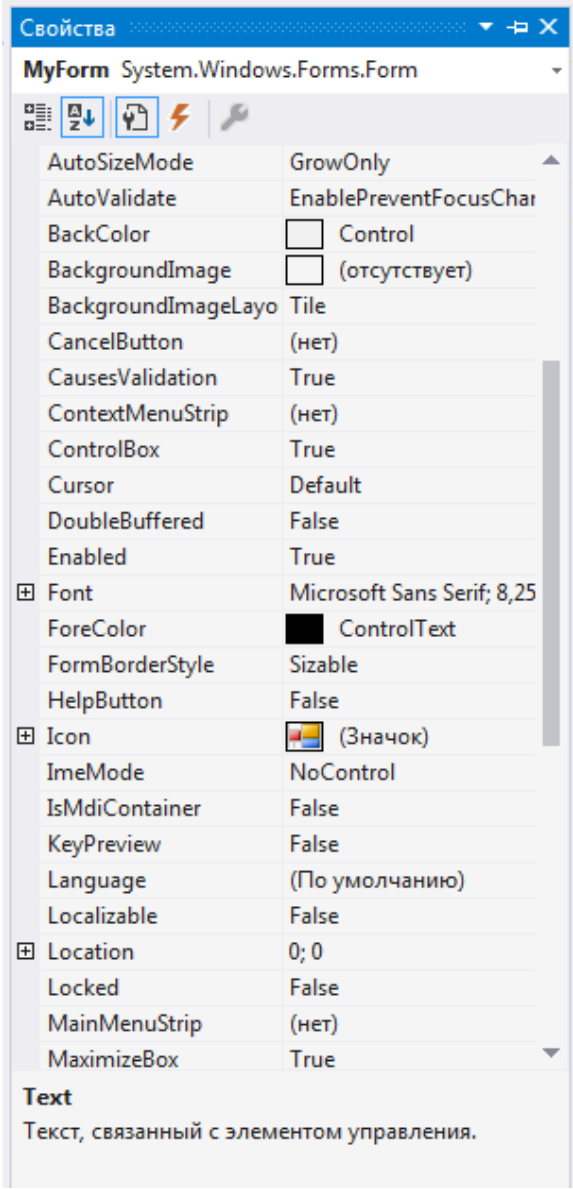


Рис. 13.3. *Вікно властивостей*

Закладка подій також має два стовпці. У лівому відображаються імена стандартних подій, на які об'єкт може реагувати, а в правому – імена методів-обробників (функцій), які реалізовуватимуть реакцію на подію. Кожній стандартній події відповідає назва методу, яка з'являється після подвійного клацання мишею у правому стовпці. У цей момент у вікно тексту програми додається шаблон базового коду (функції) для відповідного методу, який треба заповнити.

Для введення значень властивостей числового і текстового типу (Width, (Name) тощо) використовується стандартне поле введення.

Значення властивостей перерахованого типу (TextAlign, Cursor тощо) задаються комбінованим списком, звідки вибирають потрібне. Деякі комплексні властивості (Font, Image) використовують діалогові вікна, набір керуючих елементів яких залежить від певної властивості.

Структура проєкту. Проєктом називають сукупність файлів, з яких Visual Studio C++ створює готову для виконання програму.

До складу кожного проєкту обов'язково входять такі файли:

- файл проєкту *.sln. Це невеликий файл, який містить посилання на всі файли проєкту й ініціалізує програму;
- файли опису всіх форм, які входять у проєкт: файл тексту програми *.cpp і . файл форми *.h;
- файл ресурсів програми *.resx. У ньому описані ресурси, які не входять у форму, наприклад, піктограма програми;

Усі інші файли створюються після компіляції проєкту.

Редактор коду. Застосування методу до певного об'єкта веде до появи заготовки базового коду відповідної функції у вікні редактора. Заготовка (шаблон) складається із заголовка функції та операторних дужок { }. Заготовку заповнює користувач.

Файл MyForm.cpp має такий вигляд:

```
#include "MyForm.h"
using namespace System;
using namespace System::Windows::Forms;
[STAThread]
void main(array<String^>^ args) {
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	Project1::MyForm form;
	Application::Run(% form);
}
```

Файл MyForm.h має такий вигляд:

```
#pragma once

namespace Project1 {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

	/// <summary>
	/// Конструктор MyForm
	/// </summary>
	public ref class MyForm : public
System::Windows::Forms::Form
	{
	public:
		MyForm(void)
		{
			InitializeComponent();
			//
			//TODO: додайте код конструктора
			//
		}

	protected:
		/// <summary>
		/// Звільнити всі ресурси, що використовувалися.
		/// </summary>
		~MyForm()
		{
			if (components)
			{
				delete components;
			}
		}

	private:
		/// <summary>
		///Обов’язкова змінна конструктора.
		/// </summary>
```

```
            System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code

            void InitializeComponent(void)
            {
                  this->components = gcnew
System::ComponentModel::Container();
                  this->Size =
System::Drawing::Size(300,300);
                  this->Text = L"MyForm";
                  this->Padding =
System::Windows::Forms::Padding(0);
                  this->AutoScaleMode =
System::Windows::Forms::AutoScaleMode::Font;
            }
#pragma endregion
      };
}
```

13.2. Об'єкти: форма, текстове поле, зображення, кнопка та інші

Особливості роботи з формами

Форми є основою додатків. Створення інтерфейсу користувача додатку полягає в додаванні у вікно форми об'єктів C++ компонентами.

Для додавання у проєкт нової форми необхідно обрати команду «Проєкт ⇨ Добавить новий елемент ⇨ Среда CLR». У діалоговому вікні обрати «Форма Windows Forms» та задати ім’я форми (рис. 13.4). У результаті з’явиться вкладка MyForm1.h [Конструктор] поряд з вкладкою конструктора

головної форми MyForm.h [Конструктор].

Форма, що з'являється у проєкті при його створенні завантажується першою за замовчуванням, тобто завжди вважається стартовою (головною).

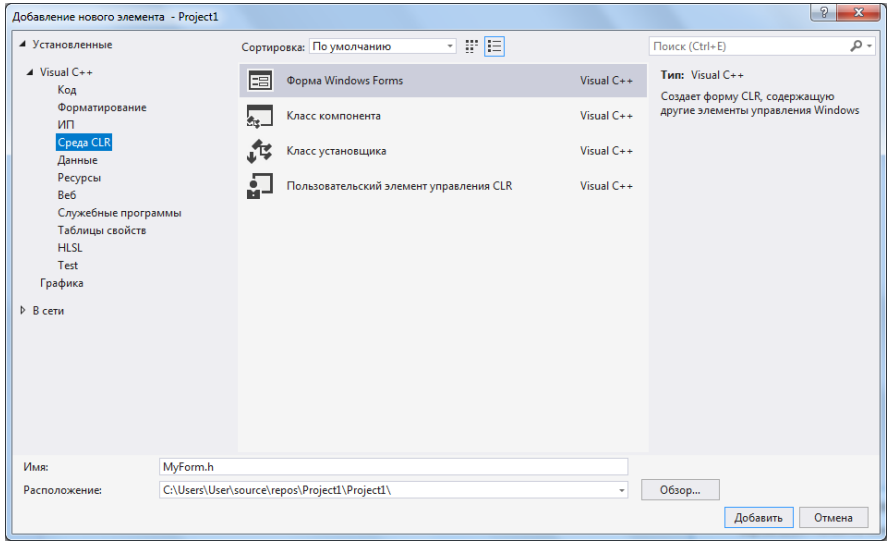


Рис. 13.4. *Додавання нової форми*

Якщо у проєкт додано форму її необхідно викликати відповідним методом.

Форма викликається на виконання у двох режимах: модальному та немодальному. У модальному режимі – метод форми ShowDialog(), у звичайному – методом форми Show().

Розглянемо деякі властивості форми.

Таблиця 13.1. *Характеристики компоненту* ***Form***

Властивість	***Опис властивості***	***Приклади значень***
AutoScroll	AutoScroll Наявність у формі True, False	AutoScroll Наявність у формі True, False
FormBorderStyle	Можливість змінювати розміри вікна	Sizeable (вікно з довільними розмірами) FixedDialog, None (вікно з фіксо- ваними розмірами)
BackColor	Колір фону форми	Maroon (перелічений тип) або 255; 192; 128 (числове значення)
Size (Width, Height)	Ширина і висота вікна у пікселях	357, 422 (числове значення)
Font	Шрифт	Комплексна властивість, задається у діалоговому вікні
Cursor	Вигляд вказівника миші під час виконання проєкту	Default, No, Arrow (перелічений тип)
Enabled	Доступність для дій об’єктів у формі під час виконання	True, False
Icon	Задання піктограми, яка буде в заголовку форми підчас	(Значок) – стандартна піктограма для Visual Studio C++,

	виконання програми	або завантажена з певного файлу *.ico.
Text	Заголовок форми	Довільний рядок символів
Методи форми		
Close()	Закрити форму. Для головної форми, це означає закриття додатку	
Hide()	приховати форму (зробити невидимою)	
Show()	виведення форми на екран	

Компонент Button

Компонент **Button** (рис. 13.5) є командною кнопкою.

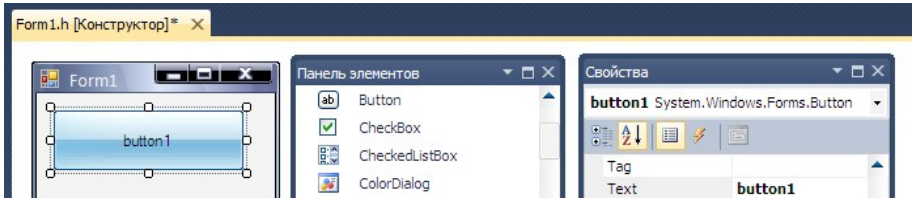


Рис.13.5. *Компонент* ***Button***

Кожен компонент має визначений набір властивостей методів та подій.

Таблиця 13.2. *Характеристики компоненту* ***Button***

Name	Ім'я компоненту. Використовується в програмі для доступу до властивостей компоненту
Text	Текст (напис) на кнопці
TextAlign	Положення тексту на кнопці. Напис може розташовуватися в центрі (**MiddleCenter**), бути притиснута до лівої (**MiddleLeft**) або правої (**MiddleRight**) межі.
Location	Положення кнопки на поверхні форми.
Size	Розмір кнопки
Font	Шрифт тексту
ForeColor	Колір тексту
Enabled	Ознака доступності кнопки. Кнопка доступна, якщо значення властивості рівне **True**, і недоступна (*наприклад, подія **Click** на кнопці не виникає*), якщо значення властивості рівне **False**
Visible	Дозволяє приховати кнопку (**False**) або зробити її видимою (**True**)
Image	Картинка на поверхні форми. Рекомендується використовувати gif-файл, в якому визначений прозорий колір фону
ImageAlign	Положення картинки на кнопці
Anchor	Закріплення позиції компоненту
DialogResult	Забезпечує роботу з модальними формами
TabStop	Відключення отримання фокуса за допомогою клавіші Tab (=False)

Події компоненту Button	
Click	Виникає, коли на кнопці клікають мишею
Enter	Виникає, коли кнопка отримує фокус, тобто
Методи Button	
Hide()	Приховує кнопку
Focus()	Робить кнопку активною
Show()	Відображає кнопку

Компонент Label

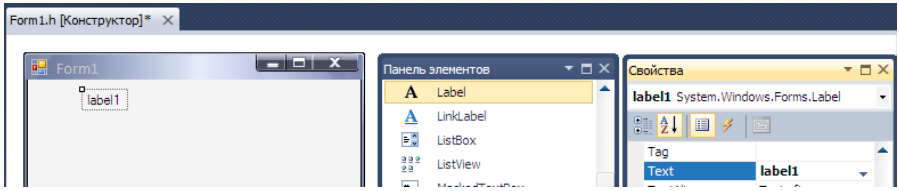


Рис.13.6. *Компонент* ***Label***

Компонент **Label** (рис. 13.6) призначений лише для відображення текстової інформації (редагування тексту не можливо). Задати текст, що відображається в полі компоненту, можна як під час розробки форми, так і під час роботи програми, присвоївши значення властивості **Text**.

Компонент може бути використаний також для виведення зображень та ідентифікації об’єктів.

Таблиця 13.3. *Властивості компоненту* ***Label***

Name	Ім'я компоненту. Використовуєтьсяпрограмі для доступу до властивостей компоненту
Text	Текст (напис) у полі
TextAlign	Положення тексту у полі компоненту
Location	Положення компоненту на поверхні форми
Size	Розмір компоненту
Font	Шрифт тексту
ForeColor	Колір тексту
BackColor	Колір фону
BorderStyle	Вид рамки (межі) компоненту

Щоб в полі компоненту **Label** вивести числове значення, це значення необхідно перетворити в рядок. Зробити це можна за допомогою методу **ToString**.

Convert::ToString(s) – *перетворення змінної до рядкового типу*

Колір тексту «**ForeColor»** і фону «**BackColor**» можна задати, вказавши назву кольори (**Color::Red, Color::Blue, Color::Green** і т. д.), або елемент колірної схеми операційної системи. У другому випадку колір буде "прив'язаний" до поточної колірної

схеми операційної системи і автоматично змінюватиметься при кожній її зміні. За замовчанням для елементів управління використовується другий спосіб кодування кольору. Колір фону може бути "прозорим" (**Color::Transparent**).

Компонент TextBox

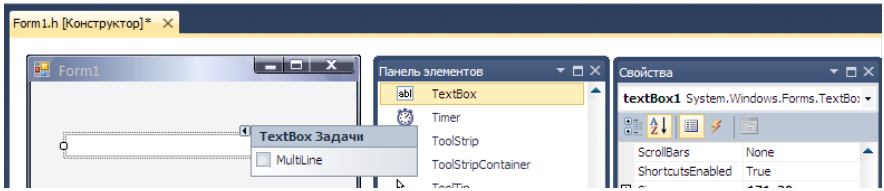


Рис.13.7. *Компонент* ***TextBox***

Компонент **TextBox** (рис. 13.7) призначений для введення даних з клавіатури. Залежно від налаштування компоненту, в полі редагування можна вводити один або декілька (**Multiline = True**) рядків тексту.

Всі дані ведені у текстове поле сприймаються програмою, як набір сиволів. Для перетворення символів до дійсного типу використовують методи **ToDouble** або **ToInt32** (*цілочисловий тип*).

Convert::ToDouble – *перетворення змінної до дійсного типу*

Таблиця 13.4. *Характеристики компоненту* ***TextBox***

Name	Ім'я компоненту. Використовується в програмі для доступу до компоненту і його властивостей, зокрема до тексту, який знаходиться в полі редагування
Text	Текст, який знаходиться в полі редагування
Location	Положення компоненту на поверхні форми
Size	Розмір компоненту
Font	Шрифт, використаний для відображення тексту в полі компоненту
ForeColor	Колір тексту
BackColor	Колір фону поля компоненту
BorderStyle	Вид рамки (межі) компоненту. Межа компоненту може бути звичайною (**Fixed3D**), тонкою (**FixedSingle**). Межа навколо компоненту може бути відсутньою (в цьому випадку значення властивості рівне **None**)
TextAlign	Спосіб вирівнювання тексту в полі
MaxLength	Максимальна кількість символів, яку можна ввести в поле компоненту
PasswordChar	Символ, який використовується для відображення символів, що вводяться користувачем (*введений користувачем рядок знаходиться у властивості* **Text**)

Multiline	Вирішує (**True**) або забороняє (**False**) введення декількох рядків тексту
Lines	Масив рядків, елементи якого містять текст, що знаходиться в полі редагування, якщо компонент знаходиться в режимі **MultiLine**. Доступ до рядка здійснюється по номеру. Рядки нумеруються з нуля
SkroolBars	Задає смуги прокрутки, що відображаються: **Horizontal** - горизонтальна; **Vertical** - вертикальна;
Методи компоненту TextBox	
AppendText ()	Додає текст до поточного тексту у вікні
Clear ( )	Очищення поля
Copy ( )	Копіювання обраного рядка у буфер обміну
Cut ( )	Вирізання виділеного блоку тексту у буфер
Focus ( )	Встановлення фокуса
Hide ( )	Приховування компоненту
Paste ( )	Заміняє поточну вибірку в полі вмістом
Select ( )	Вибирає текст у компоненті
Show ( )	Робить компонент видимим

Об’єкт PictureBox використовують для вставки графічних об’єктів із файлів типу *.bmp, *.gif, *.jpg, *.jpeg, *.png, *.ico, *.emf, *.wmf у форму. Крім відомих властивостей (Name), Size, Enabled, Visible, використовуються ще й такі (табл. 13.5):

Таблиця 13.5. *Властивості компоненту* ***PictureBox***

Властивість	***Опис властивості***	***Приклади значень***
BorderStyle	Визначає тип границі для PictureBox	None, FixedSingle
SizeMode	Визначає, як буде оброблятися розміщення рисунка в даній області рисунка	AutoSize, StretchImage
Modifiers	Вказує рівень видимості об’єкта	Public, Private, Protected

13.3. Програмування кнопок. «Задача про анкету»

Приклад Створити форму "Анкета студента" з даними про себе й двома фотографіями (портретною та художньою), які перекривають одна одну і мають з'являтися в результаті натискання на кнопки (рис. 2).

1) Завантажуємо середовище візуального програмування Visual Studio.

Запуск системи візуального програмування Visual Studio виконують клацанням на піктограмі  або за

допомогою каскадного меню **Пуск ⇨ Visual Studio 2012**.

2) Створюємо новий проєкт

Создать проєкт → Пустой проєкт CLR.

Називаємо проєкт, наприклад, **Project1.** Тоді виконуємо наступні дії:

Обозреватель решений → Добавить → Создать елемент.

Visual C++ → UI → Форма Windows Forms.

Вказуємо ім'я форми, наприклад **Form1.**

Відкриваємо файл вихідного коду **Form1.cpp** і вставляємо код:

```
#include "Form1.h"  //Цей рядок має бути вже наявний
using namespace Project1;
[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
 Application::EnableVisualStyles();
 Application::SetCompatibleTextRenderingDefault(false);
 Application::Run(gcnew Form1());
 return 0;
}
```

Далі виконуємо такі дії:

- *Проєкт → Свойства: Project1 … → Свойства конфигурации → Компоновщик → Система → Подсистема → Windows (/SUBSYSTEM:WINDOWS);*

- *Компоновщик* → *Дополнительно* → *Точка входа.* Напишіть ***main;***
- *Застосувати* → *ОК.*

Відкрити *Панель элементов* у разі її відсутності можна через меню *Вид* (або комбінацією клавіш **Ctrl+Alt+X**); вікно *Свойства* – меню *Вид* → *Другие окна* → *Окно свойств* (**Alt+ENTER**).

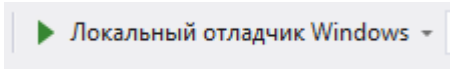
Перейти до редактора коду із вікна форми можна за допомогою комбінації клавіш **<Ctrl>+<Alt>+<0**>, або виконавши команду *Код* головного меню *Вид.*

Із режиму написання коду в режим дизайнера форми можна переключитися комбінацією клавіш **<Shift>+<F7>.**

3) Запускаємо програму Project1 на виконання і розглядаємо вікно порожньої поки-що форми. Експериментуємо з вікном форми.

Запустити програму можна декількома способами:

• виконати команду *Отладка* ⇨ *Начать отладку* головного меню;

• клацнути на кнопці 

панелі інструментів;

• натиснути функціональну клавішу.

Виконаємо такі дії: максимізуємо вікно,

відновимо його попередній розмір, мінімізуємо та знову розгорнемо вікно, пересунемо на робочому столі та змінимо його розміри, викличемо системне меню (натиснути піктограму або **Alt + Пробіл**). Виконаємо ті самі дії за допомогою команд *Перемістити*, *Розмір* та інших і клавіатури.

Висновок: вікно форми володіє усіма властивостями стандартного вікна операційної системи Windows.

4) Зберігаємо створену програму.

Для цього виберіть команду головного меню *File* → *Сохранить все* (**Ctrl+Shift+S**) або натисніть на кнопку  панелі інструментів.

5) Візуально ознайомтеся з властивістю форми Size (Width, Height).

Змініть розміри форми. Переконайтесь, що тепер змінюються властивості Width (ширина) та Height (висота) форми у вікні властивостей об'єкта.

Спробуйте безпосередньо ввести відповідне значення ширини і висоти форми у пікселях і натиснути на клавішу **Enter**.

6) Змінюємо колір фону форми.

Для цього у вікні властивостей форми *Свойства*

у рядку Appearance (BackColor) потрібно вибрати значення кольору фону одним із двох способів:

• подвійним клацанням мишею на поточному значенні властивості BackColor.

• за допомогою випадаючого меню.

7) Досліджуємо властивість форми FormBorderStyle. Встановлюємо її значення (на свій смак).

8) Змінюємо властивість Text, наприклад, на «Анкета» чи «Форма А».

9) Виконуємо програму ще раз (див. п. 3).

10) Вставляємо у форму текстове поле (об'єкт типу Label) з текстом "Анкета студента".

Двічі клацаємо мишею на піктограмі 

на закладці *Стандартные элементы управления* панелі елементів або перетягуємо його у форму. Розташовуємо вставлений об'єкт, наприклад, так, як показано на рис. 13.8, переміщуючи його мишею. Активізовуємо елемент Label1. Змінюємо значення властивості Text з Label1 на текст "АНКЕТА СТУДЕНТА" без лапок. Змінюємо значення властивості Font (шрифт) цього текстового поля на

такі:

Font: Candara;
Bold: True;
Size: 14;
ForeColor: Maroon (вкладка Web).

У вікні *Свойства* відображається список властивостей лише активного у певний момент об'єкта.

11) Аналогічно вставляємо у форму ще декілька текстових полів з вашими біографічними даними (група, факультет, університет, рік народження, адреса, …).

Один із варіантів розміщення текстових полів показаний на рис. 13.8, 13.9.

12) Вставляємо у форму об'єкт типу PictureBox (зображення).

Для цього двічі клацаємо лівою клавішею миші на піктограмі PictureBox закладки *Стандартные элементы управления* панелі елементів або перетягуємо у форму. На формі з’явиться прямокутна область для майбутнього зображення (фотографії). Розташуємо її, наприклад, у нижньому правому куті форми. Якщо потрібно, змінюємо розмір форми чи

вставленого об'єкта та досягаємо якнайкращого розташування на ній створених раніше об'єктів. Змінювати розміри об'єкта можна методом їх "розтягування" за маркери (білі габаритні квадратики). Запам'ятовуємо назву, яку Visual Studio присвоїть цьому об'єкту (значення властивості (Name)) або замінюємо її на свій розсуд. За замовчуванням цей об'єкт матиме стандартну назву pictureBox1.

13) Вставляємо свою портретну фотографію за допомогою властивості Picture (ілюстрація) об'єкта pictureBox1.

Для цього виокремлюємо об'єкт pictureBox1 та активізовуємо рядок Image у вікні *Свойства*. Клацнувши на потрібній кнопці, викликаємо діалогове вікно вибору малюнка. Або натискаємо на кнопку меню  області зображення. З'явиться меню PictureBox Tasks, натискаємо Choose Image. Завантажуємо попередньо підготовлену фотографію чи картинку. Задаємо властивість SizeMode.

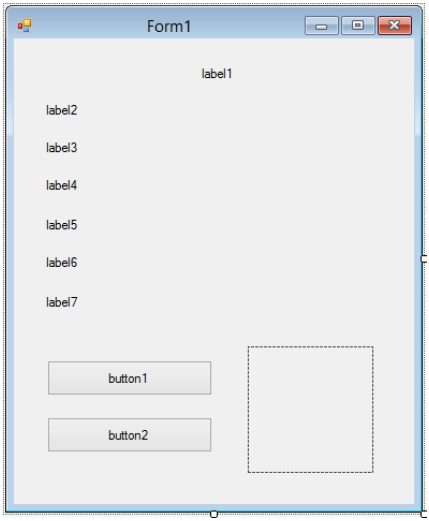


Рис. 13.8 Макет форми «Анкета студента»

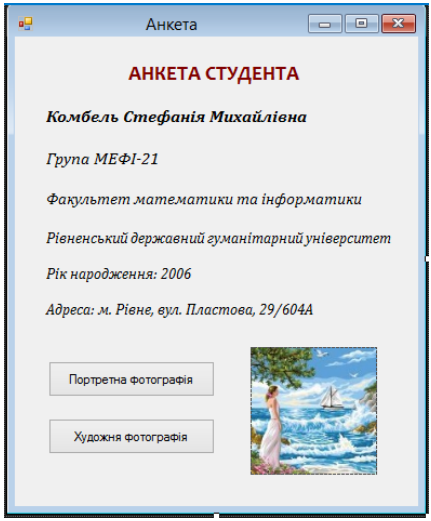


Рис. 13.9 Готова форма

14) Вставляємо свою художню фотографію у форму поверх наявної, скориставшись ще одним об'єктом типу PictureBox.

Один із варіантів розташування фотографії показаний на рис. 13.8. Вважатимемо, що цей об'єкт має назву pictureBox2.

Під час накладання об'єктів може виникнути потреба використати команди *На передний план* чи *На задний план*, які є в їх контекстних меню.

15) Експериментуємо з властивістю Visible (видимість) обох зображень, кожного разу виконуючи програму (див. пункт 3).

Після цього встановлюємо значення властивості Visible у False для обох зображень.

16) Вставляємо у форму кнопки для засвічування фотографій — два об'єкти типу Button з назвами button1 і button2.

Піктограма об'єкта типу Button (кнопка) знаходиться у вкладці *Стандартные элементы управления* панелі елементів. Міняємо підписи на кнопках (змінюємо властивість Text) на "Портретна фотографія" та "Художня фотографія" відповідно. Обираємо найкращий, на свій смак, кирилізований шрифт для підписів. Якщо використано інші картинки, обираємо для кнопок цікаві підписи. Один із варіантів розміщення кнопок показаний на рис. 13.8-13.9.

17) Запрограмовуємо кнопку "Портретна фотографія" так, щоб після її натискання у формі з'являлась портретна фотографія.

Для програмування кнопки button1 необхідно двічі клацнути на ній лівою клавішею миші. У результаті активізується вікно тексту програми із заготовкою функції button1_Click, яка опрацьовуватиме подію клацання на кнопці button1:

```
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e) {
```

```
}
```

У заготовку необхідно вставити текст програми реакції на цю подію. Процедура матиме такий вигляд:

```
private: System::Void button1_Click(System::Object^ sender,
System::EventArgs^ e) {
                        pictureBox1->Visible=true;    // Портретна
                        фотографія стає видимою
                        pictureBox2->Visible=false;    // Художня
                        фотографія стає невидимою
}
```

За допомогою функції властивість видимості для об'єкта pictureBox1 задаємо, і цю ж властивість для об'єкта pictureBox2 забираємо. Для кнопки "Художня фотографія" дії будуть протилежні.

18) Запрограмовуємо кнопку "Художня фотографія" відповідно до її призначення (див. п. 17).

Текст функції для цієї кнопки матиме такий вигляд:

```
private: System::Void button2_Click(System::Object^ sender,
System::EventArgs^ e) {
                        pictureBox1->Visible=false;   // Портретна
фотографія стає невидимою
                        pictureBox2->Visible=true;    // Художня
фотографія стає видимою
```

}

Щоб створити таку функцію швидко, можна скопіювати дві команди присвоєння з попередньої функції у нову і поміняти вирази праворуч.

19) Виконуємо програму і переконуємося, що кнопки виконують свої функції. Закриваємо вікно програми "Анкета студента".

20) Зберігаємо створену програму.

21) Закриваємо Visual Studio C++, виконуємо створену програму і експериментуємо зі створеними кнопками.

Запускаємо exe-файл з іменем проєкту (Project1/Debug/Project1.exe).

ВПРАВИ

1. Вставте у форму третю фотографію (фото вашого будинку, машини, чи університету) і ще одну кнопку з відповідним підписом, яка її висвітлюватиме.
2. Додайте у форму ще одну кнопку "Забрати фотографію". Запрограмуйте її відповідно до нового призначення. Виконайте програму і переконайтесь у правильності її роботи.

Підказка. У тексті функцій, що описують роботу кнопок, можна скористатися такими командами:

```
if (pictureBox1->Visible==true) … // Якщо видимість = true
```

3. Поміняйте сценарій роботи програми з п. 2 на такий:
   - відразу після запуску програми фотографій не видно, тільки є дві кнопки "Портретна фотографія" і "Забрати фотографію". Доступною є лише перша кнопка;
   - після клацання на кнопці "Портретна фотографія" у формі з'являється портретне фото, підпис на першій кнопці змінюється на "Художня фотографія", стає доступною кнопка "Забрати фотографію";
   - після клацання на кнопці "Художня фотографія" фотографія у формі змінюється на художню, а підпис на цій кнопці змінюється на "Третя фотографія";
   - після клацання на кнопці "Третя фотографія" фотографія у формі змінюється на третю, а підпис на цій кнопці змінюється на "Портретна фотографія";
   - після клацання на кнопці "Забрати фотографію" фотографія зникає і ця кнопка

стає невидимою (або недоступною);
- виконайте програму і переконайтесь у правильності її роботи.

Підказка. У тексті функцій, які описують роботу кнопок, можна скористатися командами, що змінюють властивості кнопок Text, Visible, Enabled.

4. Доповніть програмний код розв'язування задачі з п. 3 так, щоб після вимкнення фотографій напис на першій кнопці завжди відповідав фотографії, яка повинна з'явитися після її натискання.
5. Виходячи з умови задачі з п. 4, добийтеся того, щоб послідовність перемикання фотографій не порушувалася внаслідок їх вимкнення, а також додайте текстовий підпис з назвою фотографії, видимою у поточний момент.
6. Скопіюйте програмний код у звіт.

Зауваження 4. При бажанні форму можна доповнити додатковими текстовими полями та зображеннями (рис. 13.10).

Рис. 13.10 Доповнена форма (тут використано 14 текстових полів, 4 кнопки та 4 зображення)

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Шевченко О. В. Програмування мовою C++ : навч. посіб. / О. В. Шевченко. — Київ : КПІ ім. Ігоря Сікорського, 2022. — 248 с.
2. Ковальчук М. І., Петренко Д. С. Об’єктно-орієнтоване програмування мовою C++ : навч. посіб. / М. І. Ковальчук, Д. С. Петренко. — Львів : Вид-во Львів. політехніки, 2023. — 312 с.
3. Іванюк О. С., Гнатюк В. О. Основи програмування мовою C++ : навч. посіб. / О. С. Іванюк, В. О. Гнатюк. — Тернопіль : ТНТУ, 2022. — 220 с.
4. Мельник Р. П. Алгоритмізація та програмування мовою C++ : навч. посіб. / Р. П. Мельник. — Харків : ХНУРЕ, 2023. — 264 с.
5. Савченко І. Л., Бондар М. В. Сучасне програмування мовою C++ : практ. курс : навч. посіб. / І. Л. Савченко, М. В. Бондар. — Київ : Освіта України, 2024. — 286 с.
6. Нац. техн. ун-т України «КПІ ім. Ігоря Сікорського». Методичні рекомендації з програмування мовою C++ [Електронний ресурс]. — Київ, 2024. — Режим доступу: https://ela.kpi.ua.
7. Stroustrup B. A Tour of C++ / B. Stroustrup. — 3rd ed. — Boston : Addison-Wesley, 2022. — 320 p.
8. Josuttis N. M. C++20: The Complete Guide / N. M. Josuttis. — Bonn : Leanpub, 2022. — 800 p.
9. ISO/IEC 14882:2023 Information technology — Programming languages — C++. — Geneva : ISO, 2023.
10. Microsoft Corporation. C++ documentation [Electronic resource]. — 2024. — Available at: https://learn.microsoft.com/cpp.
11. https://uk.wikipedia.org/wiki/Типи_даних_в_C%2B%2B