Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Д. Д. Татарчук
Ю. В. Діденко

Інформатика

Навчальний посібник

Затверджено Вченою радою НТУУ «КПІ»
як навчальний посібник для студентів, які навчаються
за спеціальністю «Мікро- та наносистемна техніка»

Київ
НТУУ «КПІ»
2016

УДК 004.4(075.8)
ББК 32.973.26-018я7
Т23

Затверджено Вченою радою НТУУ «КПІ»
(Протокол № 5 від 11.04.2016 р.)

Рецензенти: *С. С. Забара*, д-р техн. наук, проф.,
Інститут комп'ютерних технологій
Відкритого міжнародного університету розвитку людини «Україна»

А. П. Кудін, д-р фіз.-мат. наук, проф.,
Національний педагогічний університет ім. М.П. Драгоманова

Відповідальний редактор *О.В. Борисов*, канд. техн. наук, проф.,
Національний технічний університет України
«Київський політехнічний інститут»

Татарчук Д. Д.

Т23 Інформатика : навч. посіб. / Д.Д. Татарчук, Ю.В. Діденко – Київ : НТУУ «КПІ» Вид-во «Політехніка», 2016. – 216 с. – Бібліогр. : с. 214–215. – 300 пр.

ISBN 978-966-622-759-4

Викладено основи програмування мовами C та C++. Описано основні конструкції та типи даних алгоритмічних мов C та C++, методи налагодження програм. Особливу увагу приділено основам об'єктно-орієнтованого програмування. Розглянуто типові структури даних, які широко застосовують у програмуванні (статичні та динамічні масиви, списки, стеки, деки, черги, бінарні дерева й графи), а також алгоритми їх оброблення. Містить велику кількість прикладів, які ілюструють використання мов програмування C та C++ для вирішення практичних задач.

Для студентів вищих навчальних закладів, які навчаються за програмою підготовки бакалаврів за спеціальностями «Електроніка» та «Мікро- та наносистемна техніка». Також може бути корисним магістрам і аспірантам електронних спеціальностей.

УДК 004.4(075.8)

ББК 32.973.26-018я7

ISBN 978-966-622-759-4

Вступ

C – проста й вишукана мова програмування, що була розроблена співробітником фірми *Bell Labs* Денісом Рітчі спільно з Кеном Томпсоном у 1972 році [1].

Спочатку її використовували як інструментальну мову операційної системи *UNIX*. Однак вона виявилась настільки вдалою, що досить швидко стала однією з найпопулярніших мов системного програмування. Швидке зростання популярності привело до розробки в 1983 році стандарту мови C після чого її стали використовувати повсюдно.

Мова C має ряд достоїнств, які роблять її універсальним інструментом програміста, а саме: простота; відносно висока швидкодія та відносно малий розмір програм, написаних на C; високий рівень мобільності програм, написаних на C; суміщення можливостей мови низького рівня зі зручністю мови програмування високого рівня.

Все наведене вище робить мову C зручним засобом для розробки програм різного призначення.

Однак програмування – це галузь знань, що досить швидко розвивається, виникають нові концепції та методи програмування. Нові методи вимагають нових можливостей, нових засобів розробки. Так розвиток концепції об’єктно-орієнтованого програмування привів до модифікації мови програмування C та розробки на її основі мови «C з класами», яка в подальшому розвинулась у мову програмування C++ [1]. Мова C++ вдало поєднала в собі достоїнства мови C з перевагами об’єктно-орієнтованого програмування, завдяки чому на сьогоднішній день вона стала одним з основних засобів розробки сучасного програмного забезпечення.

1. Елементи мови C

Під елементами мови програмування розуміють її базові конструкції, які використовують при розробці програм [1], а саме:

- алфавіт;
- константи;
- ідентифікатори;
- ключові слова;
- коментарі.

З елементів мови формуються лексеми. Лексема – це одиниця тексту програми, яка має самостійний зміст для компілятора мови C. Лексема не може містити інших лексем.

1.1. Алфавіт мови

Алфавіт – це сукупність дозволених в мові символів чи груп символів, що розглядаються як єдине ціле. В програмах на C використовують дві множини символів: множину символів мови C та множину відображуваних символів [1]. Множина символів мови C містить літери, цифри та знаки пунктуації, які мають певний сенс для компілятора мови C.

Множина символів мови C є підмножиною відображуваних символів [1]. Множина відображуваних символів складається із всіх літер, цифр і символів, що можуть бути представлені як окремий символ на клавіатурі персонального комп'ютера.

Оператори, ідентифікатори, службові слова та інші елементи програми на мові C можуть містити тільки символи із множини символів мови C, однак символьні змінні, символьні константи й коментарі можуть містити будь-який символ із множини відображуваних символів [1].

Розглянемо докладніше множини символів і правила їх використання.

1.1.1. Букви й цифри

Множина символів мови C містить великі (A-Z) та малі (a-z) літери латинського алфавіту й арабські цифри (0-9).

Літери й цифри використовують при формуванні констант, ідентифікаторів і ключових слів.

Компілятор мови C розрізняє великі й малі літери, вважаючи їх різними символами.

1.1.2. Пробільні символи

Символи «пробіл», «табуляція», «перехід на нову строку», «повернення каретки», «нова сторінка», «вертикальна табуляція», «горизонтальна табуляція» називають пробільними, оскільки вони виконують ті ж функції, що й пробіли між словами в тексті. Розглядається як пробільний і символ кінця файлу.

Компілятор мови C ігнорує ці символи, якщо їх використовують не в складі символьних змінних і констант.

Коментарі компілятор мови C також розглядає як пробільні символи.

1.1.3. Розділові символи

Розділові символи з множини символів мови C використовують для різноманітних призначень. Розділові символи наведено у таблиці 1.1 [1].

Таблиця 1.1.

Символ	Назва	Символ	Назва
,	Кома	.	Крапка
;	Крапка з комою	:	Двокрапка
?	Знак питання	'	Одинарні лапки (апостроф)
!	Знак оклику	|	Вертикальна лінія
/	Слеш, знак ділення	\	Обернений слеш
~	Тильда	_	Підкреслювання
(	Ліва кругла скобка	)	Права кругла скобка
{	Ліва фігурна скобка	}	Права фігурна скобка
[	Ліва квадратна скобка	]	Права квадратна скобка
<	Знак «менше»	>	Знак «більше»
%	Процент	&	Амперсанд
#	Ґратка	^	Стрілка вгору
-	Знак мінус	=	Знак рівності
+	Знак плюс	*	Знак множення

Ці символи мають спеціальний зміст для компілятора мови C. Правила їх використання будуть розглянуті пізніше.

1.1.4. Спеціальні символи

Спеціальні символи призначені для представлення пробільних символів, невідображуваних символів і символів, що мають спеціальне призначення. Спеціальні символи наведено у таблиці 1.2 [1].

Таблиця 1.2.

Символ	Назва	Символ	Назва
\n	Перехід на нову строку	\r	Повернення каретки
\v	Вертикальна табуляція	\t	Горизонтальна табуляція
\b	Знищення символу	\f	Перехід на нову сторінку
\a	Звуковий сигнал	\'	Одинарні лапки
\\	Обернений слеш	\''	Подвійні лапки
\ddd	Вісімкове значення байту	\xdd	Шістнадцяткове значення байту

1.1.5. Операції, операнди й вирази

Операції – це дії, що виконуються над даними. Операції позначають спеціальними символами чи групами символів, що визначають дії з обробки даних. Дані, над якими виконуються операції, називають операндами. Компілятор інтерпретує кожну з таких комбінацій як самостійну лексему. Наприклад, у виразі « x + y » змінні *x*, *y* – це операнди, а «+» – це символ, що означає операцію додавання.

Якщо операцію застосовують одночасно до двох операндів, то її називають бінарною (наприклад, додавання чи віднімання), а якщо тільки до одного операнда, то – унарною (наприклад, символ «-» у позначенні від'ємного числа). Кожна операція може застосовуватись лише до операндів певних типів. Операції мови програмування C наведено у таблиці 1.3 [1]. Операції повинні використовуватись в програмі саме так, як вони представлені у таблиці, без пробільних символів в тих операціях, що зображуються кількома символами.

Таблиця 1.3.

Операція	**Назва**	**Операція**	**Назва**
!	Логічне НІ	?:	Умовна операція
~	Побітове НІ	++	Інкремент
-	Віднімання, унарний мінус	--	Декремент
*	Множення, опосередкована адресація	/	Ділення
=	Присвоювання	+=	Додавання з присвоюванням
-=	Віднімання з присвоюванням	%	Залишок від ділення
*=	Множення з присвоюванням	<<	Зсув вліво
>>	Зсув вправо	<	Менше
>	Більше	==	Перевірка на рівність
<=	Менше чи дорівнює	>=	Більше чи дорівнює
!=	Не дорівнює	&	Побітове І, адресація
|	Побітове АБО	/=	Ділення з присвоюванням
%=	Присвоювання залишку ділення	^	Побітове виключне АБО
|=	Побітове АБО з присвоюванням	^=	Побітове виключне АБО з присвоюванням
&&	Логічне І	||	Логічне АБО
,	Послідовне виконання, кома	&=	Побітове І з присвоюванням
<<=	Зсув вліво з присвоюванням	>>=	Зсув вправо з присвоюванням

Операнди й операції формують вирази. Кожен вираз в C має своє значення (результат виконання операцій над операндами). У найпростішому

випадку вираз може складатися лише з одного операнда. Операції у виразі виконуються у порядку їхнього пріоритету. Першими виконуються операції з вищим рівнем пріоритету. Пріоритет операцій наведено у таблиці 1.4 [1].

Таблиця 1.4.

Операції	**Категорія операцій**
`() [] . ->`	Унарні операції та операція взяття адреси
`- ~ ! * &`	
`++ --`	
Приведення типу	
`* / %`	Мультиплікативні операції (множення ділення і т.і.)
`+ -`	Адитивні операції (додавання, віднімання і т.і.)
`<< >>`	Операції зсуву
`<> < > <= >=`	Операції відношення
`== !=`	
`&`	Побітові операції
`^`	
`|`	
`&&`	Логічні операції
`||`	
`?:`	Умовна операція
`= *= /= %=`	Операції присвоювання
`+= -= <<= >>=`	
`&= |= ^=`	
`,`	Операція послідовного обчислення

Пріоритет операцій в таблиці зменшується в напрямку зверху вниз. Операції, що знаходяться в одному рядку таблиці, мають однаковий пріоритет. Для зміни порядку виконання операцій використовують дужки (операції, що розміщені в дужках, виконуються в першу чергу).

1.2. Константи

Константа – це число, символ або строка символів. Константи використовують для визначення постійних величин. В C розрізняють чотири типи констант: цілі, з плаваючою крапкою, символьні константи й символьні рядки.

1.2.1. Цілі константи

Цілі константи – це набір цифр, які являють собою ціле число. Цифри записують підряд без будь-яких інших символів між ними. Цілі константи можуть бути десятковими, вісімковими або шістнадцятковими.

Десяткові константи складаються з арабських цифр «0»–«9». Десяткова константа починається з ненульової цифри за винятком випадку, коли константа дорівнює 0.

Вісімкові константи складаються з арабських цифр «0»–«7». Вісімкова константа завжди починається з цифри 0, що є ознакою вісімкової константи.

Шістнадцяткові константи складаються з арабських цифр «0»–«9» та літер «A»–«E» або «a»–«e». Літери заміняють цифри із значеннями «10»–«15» відповідно. Шістнадцяткова константа завжди починається з 0x або 0X, що є ознакою шістнадцяткової константи.

Цілі константи можуть бути додатними або від'ємними. Перед від'ємними константами ставиться знак «-», перед додатними – «+». Константа без знаку вважається додатною.

Наведемо кілька прикладів цілих констант:

123 – десяткове ціле число 123;

–12 – десяткове ціле від'ємне число, модуль якого дорівнює 12;

012 – вісімкове ціле число. В десятковій системі його значення – 10;

0x12 – шістнадцяткове ціле число. В десятковій системі його значення – 18.

1.2.2. Константи з плаваючою крапкою

Константи з плаваючою крапкою – це дійсні десяткові числа. Вони складаються із цілої частини, дробової частини, а також показника експоненти.

Ціла частина складається з десяткових цифр, перед якими може ставитись знак «+» або знак «-». Дробова частина складається з десяткових цифр і відділяється від цілої крапкою. Показник експоненти складається з десяткових цифр, перед якими може ставитись знак «+» або знак «-». Показник експоненти відділяється від інших складових константи з плаваючою крапкою за допомогою символу «e» або «E». Будь-які інші символи, включаючи пробільні, являються неприпустимими. Будь-яка з частин константи з плаваючою крапкою може бути відсутня, але не можуть бути відсутніми ціла й дробова частина одночасно.

Наведемо кілька прикладів:

`1.32e2` – дійсне десяткове число із значенням 132;

`-1.7e-1` – дійсне десяткове число із значенням –0,17;

`7e2` – дійсне десяткове число із значенням 700;

`.7` – дійсне десяткове число із значенням 0,7;

`-12E-3` – дійсне десяткове число із значенням –0,012.

1.2.3. Символьні й строкові константи

Символьна константа – це будь-який символ із дозволених в мові програмування `C` символів. Значенням символьної константи є код символу (ціле число). В програмі на мові програмування `C` символьна константа може бути представлена як ціле невід'ємне число або як символ, поміщений в лапки, якщо даний символ має екранне відображення.

Наприклад символ «0» у програмі може бути представлений як 0 або як ціле десяткове число 48, де 48 – це код символу «0» із таблиці символів *ASCII*.

Строкові константи – це набір символів, розміщених між подвійними лапками. Наприклад, «Турбо С \n».

1.3. Ідентифікатори

Ідентифікатори – це імена змінних, функцій і міток, які використовують в програмі. Ідентифікатор вводять при визначенні змінної або функції, або як мітка оператора. Ідентифікатор може складатися з літер, цифр, символів підкреслення, але починатися він може лише з літери або символу підкреслення. Проте, хоч формально ідентифікатор може починатися із символу підкреслення, бажано їх не використовувати. Це пов'язано з тим, що такі ідентифікатори використовуються як системні.

Регістр має значення. Так, наприклад, змінні Dog та dog з точки зору компілятора С є різними змінними. Довжина ідентифікатора може бути довільна, проте значущими є перші 32 символи. Якщо перші 32 символи двох ідентифікаторів співпадають, то такі ідентифікатори вважаються однаковими.

1.4. Ключові слова

Ключові слова – це слова, які в мові програмування С мають спеціальне значення.

Список ключових слів:

auto	do	for	return	switch
break	double	goto	shot	typedef
case	else	if	signed	union
char	enum	int	sizeof	unsigned
continue	extern	long	static	void
default	float	register	struct	while

Ключові слова не можна використовувати як ідентифікатори, вони можуть бути використані лише так, як це передбачено мовою програмування C.

У вищенаведеному списку подано лише ті ключові слова, які описані в стандарті мови C. Кожна конкретна реалізація компілятора може мати свої власні додаткові ключові слова, які не увійшли до наведеного вище списку. Повну інформацію про ключові слова для конкретного компілятора можна отримати із супровідної документації на даний конкретний компілятор.

1.5. Коментарі

Коментарі – це будь-яка послідовність символів, що міститься між парами символів «/*» та «*/» [1,2,3]. Коментарі вводять для документування програми. Компілятором вони сприймаються як окремий пробільний символ та ігноруються при компіляції. Коментарі можуть займати кілька рядків тексту.

Наприклад:

```
/* Коментар */

/* Багаторядковий
коментар */
```

Недопустимими є вкладені коментарі:

```
/* Коментар /* вкладений коментар не допускається, це – помилка */
продовження коментарю */
```

1.6. Оператори

Оператори – це основні елементи програми. Власне програма складається з послідовності операторів. Оператори керують процесом виконання програми. Оператор в мові C – це вираз, що закінчується

символом « ; ». Мова програмування C містить повний набір операторів структурного програмування [1,2,3,45]:

- оператори присвоювання;
- умовний оператор (if else);
- порожній оператор (;);
- складений оператор ({ } блочний оператор);
- оператор покрокового циклу (for);
- оператор циклу з передумовою (while);
- оператор циклу з післяумовою (do while);
- оператор продовження циклу (continue);
- оператор розриву (break);
- оператор вибору (switch);
- оператор безумовного переходу (goto);
- оператор виклику функції;
- оператор повернення із функції (return).

Розглянемо наведені оператори.

1.6.1. Оператори присвоювання

Мова програмування C, як і більшість інших мов програмування, підтримує простий оператор присвоювання. При виконанні оператора присвоювання змінна отримує значення (тобто у вказану комірку пам'яті записується значення).

Наприклад:

```
/* змінна x отримує значення 1*/
    x=1;
/* змінна y отримує значення, що є результатом виконання операції x+2*/
    y=x+2;
```

Крім простого присвоювання мова C підтримує велику кількість складних операторів присвоювання, що відповідають складним операціям з присвоюванням (див. табл. 1.3).

Наприклад:

```
/* змінна x отримує значення, що є результатом виконання операції x+5*/
    x+=5;
/* змінна y отримує значення, що є результатом виконання операції y*x  */
    y*=x;
```

1.6.2. Умовний оператор

Умовний оператор використовують тоді, коли, в залежності від якоїсь умови, необхідно виконати один з двох операторів. Умова – це вираз, що складається з допустимих в C операцій. Умовний оператор має наступний синтаксис:

```
if(умова)
    оператор1;
 else
    оператор2;
```

Якщо умова виконується (є істинною), тобто результат виразу не дорівнює 0, то виконується оператор з розділу «`if`». Якщо ж умова не виконується (результат виразу дорівнює 0), то виконується оператор з розділу «`else`».

Допускається також скорочена форма умовного оператора:

```
if(умова) оператор;
```

Таку форму використовують тоді, коли якась дія повинна бути виконана лише при істинності умови, а в протилежному випадку ніякі дії не повинні виконуватись.

Розглянемо кілька прикладів:

```
/*Якщо i>0, то y=x/i. В протилежному випадку оператор y=x/i – ігнорувати*/
    if(i>0)
        y=x/i;

/*Якщо i не дорівнює 0, то x=x/i. В протилежному випадку – x=i */
    if(i!=0)
        x=x/i;
    else
        x=i;
```

1.6.3. Порожній оператор

Порожній оператор – це оператор, який складається лише з символу «крапка з комою» – « ; ». Цей оператор не виконує ніяких дій. Його використовують там, де за синтаксисом повинен бути оператор, але при цьому потрібно, щоб не виконувалось ніяких дій. Наприклад, замість скороченої форми умовного оператора можна використати повну форму з порожнім оператором у розділі «`else`».

```
/*Якщо i>0, то y=x/i. В протилежному випадку оператор y=x/i – ігнорувати */
    if(i!=0)
        y=x/i;
    else
        ;
```

1.6.4. Складений оператор

Складений оператор в мові C – це кілька операторів, розміщених між символами фігурних дужок « { » та « } ». Складений оператор використовують в тих випадках, коли за синтаксисом C у даному місці може знаходитись лише один оператор, а за алгоритмом програми необхідно виконати кілька операторів:

/*Якщо i>0, то виконати блок операторів. В протилежному випадку – ігнорувати */

```
if(i>0)
{
    y=x/i;
    x=i;
}
```

1.6.5. Оператор покрокового циклу

В мові C оператор покрокового циклу має синтаксис:

```
for(початковий вираз; умовний вираз; вираз приросту)
    оператор;
```

Оператор покрокового циклу виконується наступним чином:

1. Розраховується початковий вираз.
2. Перевіряється умовний вираз. Якщо результат умовного виразу дорівнює 0 (умова не виконується), то закінчуємо виконання циклу, якщо ж результат умовного виразу не дорівнює 0 (умова виконується), то виконуємо оператор.
3. Виконується вираз приросту і здійснюється перехід до п. 2.

Якщо умовний вираз з самого початку дорівнює нулю, то цикл не виконується жодного разу.

Як правило початковий вираз – це початкове значення змінної, що використовується в циклі, умовний вираз – кінцеве значення змінної, а вираз приросту – це вираз, за яким обчислюється наступне значення змінної.

Наприклад:

/*Розрахунок суми $\sum_{i=1}^{10}\frac{1}{i^2}$ */

```
s=0;
for(i=1; i<11; i++)
    s=s+1/(i*i);
```

1.6.6. Оператор циклу з передумовою

Оператор циклу з передумовою в C має синтаксис:

```
while(умова)
    оператор;
```

Алгоритм виконання оператора циклу з передумовою наступний:

1. Перевіряється умова.
2. Якщо умова не виконується, то цикл завершується. Якщо ж умова виконується (результат операції не дорівнює 0), то виконується оператор і відбувається перехід до п.1.

Якщо умова не виконується з самого початку, то цикл не виконується жодного разу.

Наприклад:

/*Розрахунок суми $\sum_{i=1}^{10}\frac{1}{i^2}$ */

```
i=1; s=0;
while(i<=10)
{   s=s+1/(i*i);
    i=i+1;
}
```

1.6.7. Оператор циклу з післяумовою

Оператор циклу з післяумовою в C має синтаксис:

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

Алгоритм виконання оператора циклу з післяумовою наступний:

1. Виконується оператор.
2. Перевіряється умова. Якщо умова не виконується, то цикл завершується. Якщо ж умова виконується, то відбувається перехід до п.1.

На відміну від покрокового циклу та циклу з передумовою даний цикл завжди виконується хоча б один раз.

Наведемо приклад використання циклу з післяумовою для розрахунку суми $\sum_{i=1}^{10}\frac{1}{i^2}$:

```
/*Розрахунок суми ∑(i=1..10) 1/i^2 */
   i=1;
   s=0;
do
   {
   s=s+1/(i*i);
   i=i+1;
   }
while (i<=10);
```

1.6.8. Оператор продовження циклу

Оператор продовження циклу передає керування на початок наступної ітерації циклу, ігноруючи всі оператори циклу, що знаходяться після нього. Даний оператор може використовуватись лише в циклах.

Синтаксис оператора продовження циклу має вигляд:

```
continue;
```

Приклад:

```
/* розрахунок факторіалу числа n */
    n=5;
    f=1;
    for (i=0; i<=n; i++)
    {
        if(i==0)
            continue;
        f=f*i;
    }
```

1.6.9. Оператор розриву

Оператор розриву використовується для миттєвого припинення процесу виконання циклу або оператора вибору (див. наступний пункт).

Синтаксис оператора розриву має вигляд:

```
    break;
```

Для прикладу реалізуємо за допомогою даного оператора розрахунок факторіалу числа:

```
/* розрахунок факторіалу числа n, використовуючи нескінченний цикл з оператором break */
    f=1;
```

```
while(5) /*нескінченний цикл, бо в дужках не нульове значення*/
{
    if(n!=0)
        {
            f=f*n;
            n=n-1;
        }
    else break;
}
```

1.6.10. Оператор вибору

Оператор вибору використовують тоді, коли треба зробити вибір більш ніж з двох альтернативних варіантів.

Оператор вибору має синтаксис:

```
switch(змінна цілого типу)
{
    case альтернатива_1:
    оператори;
    break;
    case альтернатива_2:
    оператори;
    break;
…
    case альтернатива_n:
    оператори;
    break;
default:
    оператори;
}
```

Алгоритм виконання оператора вибору наступний: значення змінної цілого типу, яка подана в дужках після ключового слова `switch`, порівнюється з альтернативними значеннями, записаними після ключових слів `case`. Якщо знайдено значення, що співпадає, то виконуються оператори із відповідного розділу `case`. Якщо таких значень не виявлено, то виконуються оператори з розділу `default`. Розділ `default` може бути відсутній. Тоді, при відсутності значень, які співпадають, керування передається на оператор, що знаходиться безпосередньо після оператора вибору.

Розглянемо приклад:

```
/*
після виконання даного фрагменту програми змінна x отримає значення 1
*/
    i=3;
    switch(i)
    {
        case 1:
        x=3;
        break;
        case 2:
        x=5;
        break;
        case 3:
        x=1;
        break;
    }
```

1.6.11. Оператор безумовного переходу

Оператор безумовного переходу використовується для переходу до вказаного місця програми.

Даний оператор має синтаксис:

```
goto мітка;
…
мітка:
```

Мітка – це особливий вид ідентифікатора, який вказує адресу місця переходу. Мітка завжди закінчується символом двокрапка – « : ». В усьому іншому правила формування мітки такі ж, як і для будь-яких інших ідентифікаторів.

Розглянемо приклад. Використаємо оператор безумовного переходу на прикладі обчислення модуля числа.

```
/* Якщо значення змінної x додатне, то пропустимо виконання оператора
x=-x */
    if(x>=0) goto m1
    x=-x;
    m1:
```

На цьому поки що завершимо розгляд операторів. Два останніх оператора, а саме оператор виклику функції та оператор повернення із функції, розглянемо трохи пізніше, після розгляду поняття функції.

1.7. Типи даних мови програмування C

Одним з ключових питань при програмуванні практичних задач, є правильна організація даних, оскільки від цього залежить ефективність використання пам’яті та процесорного часу. Для спрощення цієї задачі в С реалізовано набір базових типів даних і механізм розробки користувачем нестандартних типів даних на основі базових. Базові типи даних умовно можна розділити на прості та структуровані.

1.7.1. Прості типи даних

До простих типів даних відносяться цілі типи й типи з плаваючою крапкою. Перелік простих типів та їх характеристики наведено у таблиці 1.5.

Таблиця 1.5.

Тип	**Розмір, біт**	**Діапазон значень**	
`signed char, char`	8	–128..127	
`signed int, int, signed`	16	залежить від реалізації	
`signed short int, signed short, short`	16	–32768..32767	
`signed long int, signed long, long`	32	–2147483648.. 2147483647	
`unsigned char`	8	0..255	
`unsigned int, unsigned`	16	залежить від реалізації	
`unsigned short int, unsigned short`	16	0..65535	
`unsigned long int, unsigned long`	32	0..4294967295	
		діапазон модуля	точність (число десяткових цифр)
`float`	32	$3.4*10^{-38}..3.4*10^{38}$	7
`double, long float`	64	$1.7*10^{-308}..1.7*10^{308}$	15
`long double`	80	$3.4*10^{-4932}..3.4*10^{4932}$	19

Для опису в програмі змінних базових типів достатньо вказати тип і після нього перерахувати через кому ідентифікатори змінних, які повинні мати даний тип.

Наприклад:

```
/* змінні символьного типу */
    char ch1, ch2, a;
    unsigned char c, d;
/* змінні з плаваючою крапкою */
    float velocity;
    double force, mass;
```

На основі простих (базових) типів можна формувати структуровані типи даних – масиви, структури, об'єднання та перераховні типи.

1.7.2. Масиви

Масив – це набір однотипних елементів. Кожен елемент має свій номер (індекс). Всі елементи масиву впорядковані за своїм індексом.

При описі масиву вказується тип його елементів, ім'я масиву й кількість елементів масиву – `n`. Кількість елементів вказується у квадратних дужках одразу після імені масиву:

```
/* одновимірний масив з 5 елементів символьного типу */
    char symbol[5];
```

Конкретний елемент масиву визначається його індексом:

```
/*  змінна a отримала значення елементу масиву symbol з індексом 3 */
    a=symbol[3];
```

Нумерація елементів завжди починається з 0, тому індекс конкретного елемента завжди лежить в межах від 0 до `n-1`.

Масив може бути одновимірним чи багатовимірним. Кількість пар квадратних дужок після імені масиву вказує на вимірність масиву. Якщо масив багатовимірний, то кількість елементів визначається як добуток кількостей елементів по всім вимірам.

Наведемо приклади опису масивів:

```
/* двовимірний масив елементів символьного типу з 3x5=15 елементів */
    char symbol_matrix[3][5];
/* тривимірний масив з 3x3x3=27 елементів з плаваючою крапкою */
    float velocity[3][3][3];
```

Конкретний елемент багатовимірного масиву визначається його координатами. Координати – це набір індексів елемента по кожному з вимірів.

Наприклад:

```
/* змінна ch отримала значення елементу двовимірного масиву
symbol_matrix з координатами (2,1) */
    ch=symbol_matrix[2][1];
/* змінна ch1 отримала значення елементу двовимірного масиву
symbol_matrix з координатами (1,0) */
    ch1=symbol_matrix[1][0];
/* змінна v отримала значення елементу тривимірного масиву velocity з
координатами (1,2,2) */
    v=velocity[1][2][2];
```

1.7.3. Структури

Структура – це складний тип даних, що об’єднує кілька змінних (однакових або різних типів) в єдине ціле для організації зручного доступу до них. Елементи структури називають полями. Назви полів в межах однієї

структури повинні бути унікальними, в той же час в різних структурах можна використовувати поля з однаковими назвами.

Опис структури здійснюється за допомогою ключового слова `struct`.

Наприклад:

```
/*  структура, що імітує комплексне число. Дана структура складається з двох полів типу float. Поле real призначено для збереження дійсної частини комплексного числа, а поле imag – уявної */
    struct complex_number
        {
            float real, imag;
        };
```

Після того як структура описана, можна описувати змінні такого типу:

```
/* описано дві змінні a1 та a2 типу struct complex_number */
    struct complex_number a1,a2;
```

Можна об’єднати опис структури з описом змінних:

```
/* одночасно з описом структури complex_number описано дві змінні a1 та a2 типу struct complex_number */
    struct complex_number
        {
            float real, imag;
        } a1,a2;
```

Доступ до конкретного поля структури здійснюється за допомогою складеного імені, що складається з імені змінної та назви поля, розділених крапкою у випадку безпосереднього доступу або символом « `->` » у випадку використання покажчиків (див. пункт 1.7.7.).

Наприклад:

```
/* поле real змінної a1 типу struct complex_number із попереднього прикладу отримало значення 3 */
    a1.real=3;
/* поле imag змінної a2 типу struct complex_number із попереднього прикладу отримало значення поля real змінної a1 */
    a2.imag = a1.real;
```

Особливим видом структур є бітові поля. В таких структурах, розмір полів задається в бітах після символу двокрапки, що знаходиться одразу після назви поля. Розмір поля в бітових полях можна вибрати виходячи з максимального значення даних, які будуть зберігатись у цьому полі. Це дає змогу більш раціонально використовувати пам’ять.

Наприклад:

```
/* Структура для збереження атрибутів екранного символу у текстовому режимі. Дозволяє реалізувати 16 кольорів. Займає в пам’яті всього 2 байти. Якби не використовували бітові поля, то дана структура займала б у пам’яті мінімум 4 байти */
    struct
    {
        unsigned symbol : 8;
        unsigned color : 4;
        unsigned underline : 1;
        unsigned blink : 1;
    } screen_symbol;
```

1.7.4. Об’єднання

Об’єднання – це засіб, який дозволяє звертатись до одних і тих же даних (у різні моменти часу) по-різному, як до кількох різних змінних різних

типів. Синтаксис об’єднань аналогічний синтаксису структур за винятком того, що замість ключового слова `struct` використовується ключове слово `union`.

Розглянемо приклад:

```
/* Визначено об’єднання, що дозволяє звернутись до даної області пам’яті трьома різними способами */
    union example
        {
            int digit;
            double big_float;
            char letter;
        }
/* Оголошується змінна з іменем var, яка є об’єднанням типу example */
    union example var;
/* Здійснюється звернення до змінної var трьома різними способами: */
/* значення записується у змінну var, використовуючи 2 байти */
    var.digit=55;
/* значення записується у змінну var, використовуючи 8 байтів; значення 55 втрачається */
    var.big_float=-2.52e-4;
/* значення записується у змінну var, використовуючи 1 байт; попереднє значення (–2.52e–4) втрачається */
    var.letter=’a’;
```

Зрозуміло, що розмір змінної, оголошеної як об’єднання, дорівнює розміру найбільшого з типів оголошених в об’єднанні. В об’єднанні можна оголошувати не тільки прості змінні, а й складні – масиви, структури тощо.

1.7.5. Перераховний тип

Перерахування (enumeration) – це синонім слова список. В програмах на C перераховний тип – це іменований список цілих констант [1, 5]. Кожна з констант має власне ім'я. Імена констант повинні бути унікальними. Для створення такого списку використовують ключове слово enum:

```
/* створюється список з ім'ям boolean з двох констант – «no» та «yes» */
enum boolean {no, yes};
```

Зазвичай значення в списку упорядковані. Перша в списку константа має значення 0, друга 1 і т.д. Якщо є необхідність, то значення констант можуть починатися не з 0, але це треба вказати явним чином при визначенні списку.

```
/* значення констант почнуться з одиниці – 1, 2 …12 */
enum months {jan=1, feb, mar, apr, may, jun, jul, aug,
sep, oct, nov, dec};
```

При необхідності можна задати невпорядкований список:

```
/* список констант, що мають значення кодів спеціальних символів */
enum {bell = '\a', backspace = '\b', tab = '\t',
newline = '\n', return = '\r'};
```

До значень перераховного типу можна застосовувати операції відношення (« > », « < », « != » і т.і.).

1.7.6. Покажчики

Покажчик – це змінна, яка містить адресу іншої змінної (вказує на цю змінну) [1, 2, 5]. Покажчики є ефективним засобом доступу до значень змінних.

В C покажчики використовують досить інтенсивно. Достатньо сказати, що ім'я масиву є не що інше, як покажчик на перший елемент масиву. Задається покажчик за допомогою символу « * » :

```
/* покажчик на змінну типу int, ім'я покажчика ptr */
    int* ptr;
/* покажчик на змінну типу float, ім'я покажчика ptr1 */
    float* ptr1;
```

Покажчик може вказувати не лише на просту змінну, але й на структуру, на об'єднання, на інший покажчик і т.і.

В мові C над покажчиками можна виконувати наступні операції:

1. Присвоювання значення покажчику.

```
...
int i;
int* ptr;
/* покажчику ptr за допомогою операції & присвоюється значення адреси, за якою розміщена змінна i */
ptr=&i;
```

2. Визначення значення, яке знаходиться за вказаною адресою.

```
...
int i,j;
int* ptr;
ptr=&i;
/* змінній j присвоюється значення змінної i, адреса якої міститься у покажчику ptr */
j=*ptr;
```

3. Отримання адреси самого покажчика.

```
int* ptr1;
int** ptr2;  /*  ptr2 – змінна-покажчик, що містить адресу покажчика, тобто є покажчиком на покажчик */
int i=9;
...
ptr1=&i;
ptr2=&ptr1;
```

4. Збільшення (зменшення) покажчика. Ця операція непридатна по відношенню до імені масиву, оскільки це означало б переміщення масиву в іншу область пам’яті.

5. Визначення різниці покажчиків. Зазвичай це робиться для покажчиків, які посилаються на елементи одного масиву, щоб визначити відстань між ними.

1.7.7. Декларація `typedef`

Мова `C` має в своєму складі засіб `typedef`, який дозволяє давати нові імена типам даних [5]. Наприклад декларація

```
typedef int length;
```

визначає ім’я `length`, як синонім типу `int`. З цього моменту тип `length` можна використовувати в деклараціях, в операціях приведення типу і т.д. таким же чином, як тип `int`:

```
length max, min;
```

За допомогою декларації `typedef` можна давати нові імена не тільки базовим типам, але й структурам, масивам і т.д. Наприклад:

```
/* визначимо структуру типу struct complex, що складається з двох полів типу float */
typedef struct complex
            {
             float re, im;
            };
/* тепер можемо використовувати слово complex як базовий тип при описі змінної a */
    complex a;
/* визначимо c_ptr, як покажчик на тип complex */
typedef complex* c_ptr;
/* тепер можемо використовувати слово c_ptr як базовий тип при описі змінної d */
c_ptr d;
      a.re=1;
      a.im=0;
      d=&a;
```

Для використання typedef існують дві важливі причини. Перша – це параметризація програми, пов'язана з проблемою сумісності програм і апаратного забезпечення. Якщо за допомогою typedef визначити типи, які залежать від апаратної платформи, то при переносі програми на іншу платформу достатньо буде внести зміни лише до визначень typedef. Друга причина – це бажання зробити програму більш зрозумілою.

1.7.8. Класи пам'яті

Одним з головних достоїнств мови C є те, що вона дає змогу керувати ресурсами програми. До таких ресурсів відноситься і пам'ять програм.

Кожна змінна в програмі належить до певного типу, який визначає скільки пам'яті необхідно виділити для її збереження. Крім того, кожна змінна належить до певного класу пам'яті. Клас пам'яті визначає час життя змінної та область її доступності. В C визначено чотири класи пам'яті:

`auto` – автоматична змінна; створюється в пам'яті, при виклику блоку, в якому вона визначена, і знищується, коли блок завершує свою роботу. Область дії – блок, в якому змінна створена. При повторному зверненні до блоку змінна створюється знову. Старе значення змінної при цьому втрачається.

`register` – регістрова змінна; аналогічна автоматичній, але зберігається в регістрах центрального процесора, а не в пам'яті, тому доступ до неї набагато швидший. Проте, якщо змінна навіть описана як регістрова, а вільних регістрів немає, то створюється автоматична змінна.

`static` – статична змінна; область дії – файл, в якому вона визначена. Час життя змінної – до кінця роботи програми. При виході з блоку статична змінна зберігає своє значення.

`extern` – зовнішня змінна; може бути визначена в будь-якому місці програми (навіть в іншому модулі). Область дії – вся програма. Час життя з моменту створення і до кінця роботи програми.

Якщо тип змінної явно не вказаний, то змінна має тип `auto`.

1.8. Функції

Програмування мовою C базується на понятті функції. Що ж таке функція?

1.8.1. Поняття функції

Функція – це самостійна одиниця програми, спроектована для реалізації конкретної задачі [1,2,5]. Використання функцій дозволяє більш ефективно використовувати результати попередніх робіт при розробці нової

програми й створювати бібліотеки готових рішень для задач, які часто зустрічаються в програмуванні. До складу кожної програми на C входить, як мінімум, одна функція з назвою main, з якої розпочинається виконання програми.

Визначення функції має вигляд:

```
тип_результату ім'я_функції (декларація аргументів)
{
    тіло функції, декларації та інструкції;
}
```

Окремі частини визначення можуть бути відсутні. Наприклад, функція може не мати аргументів. Може також бути опущений тип результату, тоді, за замовчуванням, тип результату int.

Визначення функції може міститись у будь-якому з файлів проекту. Якщо функцію необхідно виконати в іншому файлі, то в ньому повинен знаходитись опис функції:

```
тип_результату ім'я_функції (декларація аргументів);
```

Порядок і типи аргументів та тип результату у визначенні й описі функції повинні співпадати.

1.8.2. Оператор виклику функції

Для того, щоб запустити функцію на виконання, необхідно виконати оператор виклику функції. Оператор виклику функції має вигляд:

```
ім'я_функції (параметри виклику);
```

Типи параметрів виклику та їх порядок повинні бути узгоджені з типами й порядком аргументів у визначенні функції.

1.8.3. Оператор повернення із функції

Для завершення виконання функції використовують оператор `return`:

```
return результат;
```

де `результат` – це число або вираз. Тип результату повинен узгоджуватись з типом результату, який задекларовано при визначенні функції.

Таких операторів може бути кілька. Як тільки управління передається на оператор `return`, виконання функції завершується, незалежно від того досягнуто кінець функції чи ні. Управління передається оператору, який знаходиться безпосередньо за оператором виклику функції.

Оператор `return` може бути відсутній. В такому випадку функція виконується до кінця, після чого управління передається оператору, який знаходиться безпосередньо за оператором виклику функції. При цьому значення результату виконання функції невизначено і його не можна використовувати.

Приклади визначення функції, опису функції та виклику функції наведемо після розгляду структури програми на `C`.

1.8.4. Покажчики на функцію

У мові програмування `C` сама функція не є змінною, але є можливість визначити покажчик на функцію і працювати з ним, як зі звичайною змінною: присвоювати, розміщувати в масиві, передавати як параметр функції, повертати як результат виконання функції і т.і.

Фактично покажчик на функцію містить адресу першої команди функції, на яку він вказує. Це може бути зручно в тому випадку, якщо наперед невідомо яку функцію необхідно викликати в даному місці програми і це залежить від результатів деяких попередніх подій, визначених в програмі.

Розглянемо приклад:

```
...
float a[100], b[100]; /* створюємо два масиви */
...
/* описуємо дві функції, що повертають значення типу float і приймають по два аргументи типу float */
float equ_fun(float w,float bdp);
float bequ_fun(float w, float bdp);
...
typedef float (*funpt)(float,float); /* визначаємо покажчик на функцію */
funpt fun_ptrs[2]; /* визначаємо масив з двох покажчиків на функцію */
...
fun_ptrs[0]=equ_fun;
fun_ptrs[1]=bequ_fun;
...
int main()
{
...
/* для парних і викликаємо фукцію equ_fun, а для непарних – bequ_fun */
for(i=0;i<100;i++)
    {
        fun_ptrs[i mod 2](a[i],b[i]);
    }
...
     return 0;
}
```

```
float equ_fun(float w,float bdp)
{
...
    return n;
}
float bequ_fun(float w, float bdp)
{
...
    return p;
}
```

У даному прикладі створюємо масив з двох елементів, кожен елемент якого є покажчиком на функцію, а потім у циклі проводиться виклик цих функцій. Для парних `i` викликається функція `equ_fun`, а для непарних – `bequ_fun`.

Використання покажчиків на функції полегшує реалізацію багатьох завдань. Наприклад, з їх допомогою значно легше реалізується меню програми.

1.9. Директиви препроцесора

Однією з важливих особливостей мови програмування C є наявність препроцесора. Він являє собою макропроцесор, який використовується для обробки тексту програми перед компіляцією [1, 4]. Компілятор мови C сам викликає препроцесор, однак препроцесор може бути викликаний і незалежно від компілятора. Обробка програми препроцесором керується директивами препроцесора.

Директиви препроцесора – це інструкції, записані в програмі на мові C. Їх зазвичай використовують для того, щоб полегшити модифікацію програм при переході на іншу платформу, при розробці кількох варіантів програми

(наприклад повної та демонстраційної версій). Директиви препроцесора дозволяють замінити лексеми програми деякими значеннями, додати до файлу з програмою інший файл, виконати умовну компіляцію (відключення деяких частин програми від проекту при виконанні певних умов) і т.і. Директиви препроцесора мови C завжди починаються із символу «#».

Препроцесор C розрізняє наступні директиви:

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

В даному переліку наведено не всі директиви препроцесора. За більш детальною інформацією рекомендуємо звернутись до керівництва користувача конкретної системи програмування.

1.9.1. Директиви `#define` та `#undef`

Директиву `#define` використовують в мові C для заміни однієї текстової послідовності на іншу. Це дає можливість вводити в програму імена для числових констант або невеликі макроси, які виконують нескладні дії. Розглянемо приклади:

`/*` Введемо ім'я `pi` для числової константи 3.141592654. Після цієї директиви в програмі можна замість числа 3.141592654 писати `pi`. Перед компіляцією препроцессор відшукає символьну послідовність `pi` і підставить замість неї 3.141592654 `*/`

```
#define pi 3.141592654
```

`/*` Введемо макрос, що обчислює квадрат числа `x`, назвемо його `sqr(x)`. Препроцесор перед компіляцією замінить символьну послідовність `sqr(x)` на `((x)*(x)) */`

```
#define sqr(x) ((x)*(x))
```

Директива #define діє від того рядку програми, де вона введена, до кінця файлу, у якому вона знаходиться. Якщо проект складається з кількох файлів, то директиву #define треба ввести в кожний файл проекту, де буде використовуватись ім'я даної константи (pi) або макросу (sqr(x)) до першого їх використання.

Відмінити дію директиви #define можна за допомогою директиви #undef:

```
/* Після цієї директиви препроцесор буде ігнорувати символьну послідовність pi. Заміна не буде проводитись */
#undef pi
```

При використанні макросів визначених за допомогою директиви #define треба бути дуже обережним, оскільки при цьому у програмі можуть з'являтись непередбачувані помилки. Наприклад, якщо у визначенні макросу sqr(x) опустити скобки (записати x*x), то операція sqr(x+3) буде виконуватись неправильно, оскільки замість ((x+3)*(x+3)) препроцесор підставить вираз x+3*x+3.

Взагалі макроси замість функцій слід використовувати лише тоді, коли є необхідність підвищити швидкодію програми, оскільки вони виконуються значно швидше ніж виклик функції, але при цьому слід пам'ятати, що вони значно збільшують розмір програми.

1.9.2. Директива #include

Директиву #include використовують для того, щоб у вказане місце програми був вставлений текст із іншого файлу. Формат директиви такий:

```
#include < ім'я файлу >
```

або

```
#include " ім'я файлу "
```

Перший варіант використовується тоді, коли файл знаходиться у системних каталогах даної системи програмування, а другий – в іншому випадку. Наприклад:

```
/* Включити файл stdio.h, який знаходиться у системному каталозі даного компілятора. */
#include <stdio.h>
/* Включити файл st.1, який знаходиться у тому ж каталозі, що й програма */
#include “st.1”
```

1.9.3. Директива #pragma

Директиву #pragma застосовують тоді, коли необхідно використати системно залежні директиви, які можуть не підтримуватись іншими компіляторами мови C. Це дозволяє полегшити процес переведення програм на інші платформи, якщо виникає така потреба. Директива #pragma має вигляд:

```
#pragma аргументи
```

Діє директива наступним чином, якщо система програмування розпізнає директиви, записані в аргументах, то вони виконуються, якщо ні то вони просто ігноруються.

За більш повною інформацією необхідно звернутись до керівництва користувача конкретної системи програмування.

1.9.4. Директиви умовної компіляції

Умовна компіляція – це зручний засіб для створення кількох версій однієї програми. Наприклад, при розробці робочої, навчальної та демонстраційної версій програми замість того, щоб розробляти три різних

проекти, можна задати три різних сценарії компіляції одного проекту. Зробити це можна за допомогою директив умовної компіляції – `#if`, `#elif` та `#endif`.

Директиви умовної компіляції дозволяють вказати блоки коду, які будуть включені в кінцеву програму в залежності від версії програми.

Наприклад:

```
/* Константа, що вказує на необхідність компіляції демонстраційної версії програми */
#define DEMO 0
/* Константа, що вказує на необхідність компіляції навчальної версії програми */
#define EDUCATIONAL 1
/* Константа, що вказує на необхідність компіляції робочої версії програми */
#define WORK 2
/* Константа, від значення якої залежить вибір варіанту компіляції. В даному випадку вибрано демонстраційний варіант програми */
#define VERSION DEMO
/* Тут здійснюється перевірка, яку саме версію необхідно компілювати */
#if VERSION == DEMO
/* Тут розміщують оператори, які включають лише в демонстраційну версію програми */
#elif VERSION == EDUCATIONAL
/* Тут розміщують оператори, які включають лише в навчальну версію програми */
#elif VERSION == WORK
/* Тут розміщують оператори, які включають лише в робочу версію програми */
#endif
/* Тут розміщують оператори, які включають в будь-яку версію програми*/
```

Як бачимо робота директив умовної компіляції аналогічна роботі оператора `if`, а саме: директиву `#if` використовують для перевірки умови, директиву `#elif` використовують, якщо треба перевірити більш ніж одну умову, а директива `#endif` вказує на кінець дії директив `#if` та `#elif`. Такі конструкції розміщують у всіх місцях програми, які повинні компілюватися по-різному в залежності від версії програми. Разом з директивою `#if` можна використовувати директиву `#else`, яка виконує таку ж функцію, як `else` в операторі `if`.

Замість директиви `#if` можуть бути використані директиви `#ifdef` або `#ifndef`.

Вони мають наступний синтаксис:

```
#ifdef ім'я_константи
#ifndef ім'я_константи
```

Директива `#ifdef` дозволяє виконання операторів, що слідують за нею, якщо в програмі визначена константа, ім'я якої вказане після директиви `#ifdef`. Константа повинна бути визначена до використання даної директиви. Директива `#ifndef` дозволяє виконання операторів, що слідують за нею, якщо в програмі не визначена константа, ім'я якої вказане після директиви `#ifndef`. При цьому неважливо, яке саме значення має вказана константа. Важливим є лише сам факт існування чи не існування такої константи в програмі. В усьому іншому дані директиви нічим не відрізняються від директиви `#if`.

1.10. Структура програми на C

Програма на мові програмування `C`, являє собою сукупність розглянутих вище елементів: директив препроцесора, описів змінних, функцій, операторів. Програма може складатися з довільної кількості перелічених елементів. Порядок їх розміщення має значення. Так будь-яка

змінна повинна бути описана до першого її використання. Порядок розміщення директив препроцесора може суттєво впливати на кінцеву програму.

Будь-яка програма на C повинна містити функцію з назвою `main`. Саме з цієї функції починається виконання програми. Вона визначає дії, що виконуються програмою, та здійснює, за необхідності, виклик інших функцій.

Наприклад:

```
/* Директива препроцесора */
#define pi 3.14
/* Головна функція програми */
int main()
{
/* Визначення змінних */
    float r,s;
/* Інші оператори програми */
    r=4;
    s=pi*r*r;
return 0;
}
```

Програма на C може міститись в одному або в кількох файлах. Всі файли, необхідні для компіляції програми, повинні бути підключені до проекту.

1.11. Стандартна бібліотека C

Сама по собі бібліотека мови C не є частиною мови, однак закладений в ній набір функцій, типів і макросів складає системне середовище, що підтримує стандарт мови C. Ми розглянемо лише основні бібліотечні засоби

стандарту ANSI [5]. Для більш детального знайомства з бібліотеками різних версій мови C пропонуємо звернутись до технічної документації.

Функції, типи й макроси декларуються в наступних файлах:

assert.h	float.h	math.h	stdarg.h	stdlib.h
ctype.h	limits.h	setjmp.h	stddef.h	string.h
errno.h	locale.h	signal.h	stdio.h	time.h

1.11.1 Бібліотека математичних функцій: math.h

Математичні функції мови програмування C декларуються у файлі math.h. Це наступні функції:

- double sin(double x) – обчислює синус від x (x задається в радіанах);
- double asin(double x) – обчислює арксинус від $x \in [-1,1]$, результат обчислюється в радіанах в діапазоні $[-\pi/2,\pi/2]$;
- double sinh(double x) – обчислює гіперболічний синус від x (x задається в радіанах);
- double cos(double x) – обчислює косинус від x (x задається в радіанах);
- double acos(double x) – обчислює арккосинус від $x \in [-1,1]$, результат обчислюється в радіанах в діапазоні $[0,\pi]$;
- double cosh(double x) – обчислює гіперболічний косинус від x (x задається в радіанах);
- double tan(double x) – обчислює тангенс від x (x задається в радіанах);
- double tanh(double x) – обчислює гіперболічний тангенс від x (x задається в радіанах);;
- double atan(double x) – обчислює арктангенс від x, результат обчислюється в радіанах в діапазоні $[-\pi/2,\pi/2]$;

`double atan2(double y, double x)` – обчислює арктангенс від `y/x`, результат обчислюється в радіанах в діапазоні $[-\pi,\pi]$;

`double exp(double x)` – обчислює e^x;

`double log(double x)` – обчислює натуральний логарифм від $x > 0$;

`double log10(double x)` – обчислює десятковий логарифм від $x > 0$;

`double pow(double x, double y)` – обчислює x^y;

`double sqrt(double x)` – обчислює корінь квадратний від $x \geq 0$;

`int abs(int n)` – обчислює модуль від цілого числа `n`;

`double fabs(double x)` – обчислює модуль від `x`;

`double ceil(double x)` – обчислює найменше ціле $n \geq x$ (округлює до більшого значення);

`double floor(double x)` – обчислює найбільше ціле $n \leq x$ (округлює до меншого значення);

`double ldexp(double x, int n)` – обчислює $x \cdot 2^n$;

`double frexp(double x, int* exp)` – розбиває `x` на два множники, перший з яких – нормалізований дріб у діапазоні [1/2,1], а другий – степінь числа 2. Показник степеня запам'ятовується в комірці на яку вказує покажчик `exp`. Якщо $x = 0$, то обидві частини результату дорівнюють 0;

`double modf(double x, double* ip)` – `x` розбивається на цілу й дробову частини, обидві частини мають той же знак, що й `x`. Ціла частина запам'ятовується в комірці пам'яті, на яку вказує покажчик `ip`, а дробова повертається, як результат функції;

`double fmod(double x, double y)` – залишок від ділення `x` на `y`. Результат – число з плаваючою крапкою, знак якого співпадає зі знаком змінної `x`.

1.11.2 Функції загального призначення: `stdlib.h`

У файлі `stdlib.h` декларуються функції призначені для перетворення чисел, виділення пам'яті та інших задач. Розглянемо найбільш уживані з цих функцій.

`double atof(const char* s)` – перетворює символьну послідовність у число типу `double`, якщо це можливо;

`int atoi(const char* s)` – перетворює символьну послідовність у число типу `int`, якщо це можливо;

`long atol(const char* s)` – перетворює символьну послідовність у число типу `long`, якщо це можливо;

`int rand(void)` – генерує псевдовипадкове число;

`void srand(unsigned int seed)` – ініціалізує нову послідовність генератора псевдовипадкових чисел, використовуючи для ініціалізації параметр `seed`;

`void* calloc(size_t nobj, size_t size)` – повертає покажчик на область пам'яті, виділену для розміщення масиву `nobj` об'єктів, кожен з яких має розмір `size` або `NULL`, якщо виділити вказаний об'єм пам'яті неможливо;

`void* malloc(size_t size)` – виділяє область пам'яті розміром `size`, якщо це можливо. Повертає покажчик на виділену область. Якщо виділити область вказаного розміру неможливо – повертає `NULL`;

`void* realloc(void* p, size_t size)` – змінює розмір області пам'яті, на яку вказує покажчик `p`. Новий розмір вказується в змінній `size`;

`void free(void* p)` – вивільняє область пам'яті на яку вказує покажчик `p`. Цю операцію можна застосовувати лише до областей пам'яті, які раніше були виділені за допомогою однієї з функцій динамічного розподілу пам'яті – `calloc`, `malloc` або `realloc`;

`void exit(int status)` – викликає нормальне завершення програми;

`int atexit(void (*fcn)(void))` – реєструє функцію, на яку вказує покажчик `fcn`, як функцію, що буде виконуватись при нормальному завершенні програми;

`int system(const char* s)` – передає операційній системі команду, що міститься у символьній стрічці `s`;

`int abs(int n)` – обчислює модуль від цілого числа `n`.

1.11.3 Функції перевірки та обробки літер: `ctype.h`

У файлі `ctype.h` знаходяться функції, призначені для роботи з літерами. Розглянемо основні з цих функцій.

`int isalnum(int c)` – повертає ненульове значення, якщо `c` – цифра чи літера, 0 – в протилежному випадку;

`int isalpha(int c)` – повертає ненульове значення, якщо `c` – літера, 0 – в протилежному випадку;

`int iscntrl(int c)` – повертає ненульове значення, якщо `c` – керуючий символ, 0 – в протилежному випадку;

`int isdigit(int c)` – повертає ненульове значення, якщо `c` – десяткова цифра, 0 – в протилежному випадку;

`int isgraph(int c)` – повертає ненульове значення, якщо `c` – будь-який друкований символ крім пробілу, 0 – в протилежному випадку;

`int islower(int c)` – повертає ненульове значення, якщо `c` – літера нижнього регістру, 0 – в протилежному випадку;

`int isprint(int c)` – повертає ненульове значення, якщо `c` – будь-який друкований символ, включаючи пробіл, 0 – в протилежному випадку;

`int ispunct(int c)` – повертає ненульове значення, якщо `c` – знак пунктуації;

`int isspace(int c)` – повертає ненульове значення, якщо `c` – символ пробілу, символ нової строки, символ повернення каретки, символ переходу на іншу сторінку або символ табуляції;

`int isupper (int c)` – повертає ненульове значення, якщо `c` – літера верхнього регістру, 0 – в протилежному випадку;

`int isxdigit(int c)` – повертає ненульове значення, якщо `c` – будь-який друкований символ крім пробілу, літери чи цифри, 0 – в протилежному випадку;

`int tolower(int c)` – переводить `c` до нижнього регістру;

`int toupper(int c)` – переводить `c` до верхнього регістру.

1.11.4. Функції для роботи з рядковими даними: `string.h`

Рядковий тип даних в мові програмування C реалізовано як лінійний масив символів. Для полегшення роботи з рядковими даними в C реалізовано ряд функцій.

`char* strcat(char* s, char* t)` – додає рядок `t` в кінець рядка `s`, повертає рядок `s`;

`char* strncat(char* s, char* t, int n)` – додає `n` перших символів рядка `t` в кінець рядка `s`;

`char strcmp(char* s, char* t)` – порівнює рядки `s` і `t`, повертає значення < 0, якщо `s` < `t`, 0 при `s` = `t` та значення > 0 при `s` > `t`;

`char strncmp(char* s, char* t, int n)` – порівнює `n` перших символів рядків `s` і `t`, повертає значення аналогічні до значень функції `strcmp(s,t)`;

`char* strcpy(char* s, char* t)` – копіює рядок `t` у рядок `s`, повертає рядок `s`. Рядок `s` повинен мати розмір достатній для розміщення всіх символів рядка `t`;

`char* (char* s, char* t, int n)` – аналогічна функції `strcpy(s,t)`, але копіює лише `n` перших символів рядка `t` до рядка `s`;

`int strlen(char* s)` – повертає кількість символів, що містяться в `s`. Символ кінця рядка не враховується;

`char* strchr(char* s, int c)` – повертає покажчик на місце першого входження символу `c` до рядка `s` або `NULL`, якщо такого символу у `s` немає;

`char* strrchr(char* s, int c)` – повертає покажчик на місце останнього входження символу `c` до рядка `s` або `NULL`, якщо такого символу у `s` немає.

1.11.5. Функції введення-виведення, робота з файлами: `stdio.h`

Визначені у файлі `stdio.h` функції введення-виведення, а також типи й макроси, складають приблизно одну третину бібліотеки. В основі бібліотеки введення-виведення лежить поняття потоку [4].

Потік – це джерело або одержувач даних; його можна пов’язати з диском або з якимсь іншим зовнішнім пристроєм. Бібліотека підтримує два види потоків: текстовий і бінарний, хоча на деяких системах, зокрема в *UNIX*, вони не розрізняються. Текстовий потік – це послідовність рядків; кожен рядок має нуль або більше символів і закінчується символом «`\n`».

Бінарний потік – це послідовність неперетворених байтів, що є деякими проміжними даними, які характеризуються тим, що якщо їх записати, а потім прочитати тією ж системою введення-виведення, то одержимо інформацію, що співпадає з початковою.

Потік з’єднується з файлом або пристроєм за допомогою його відкриття, вказаний зв’язок розривається шляхом закриття потоку. Відкриття файлу повертає покажчик на об’єкт типу `FILE`, який містить всю інформацію, необхідну для управління цим потоком. Якщо не виникає

двозначності, ми користуватимемося термінами «файловий покажчик» і «потік» як рівнозначними.

Коли програма починає роботу, вже відкриті три стандартних потоки: `stdin` (стандартний потік введення, зазвичай, це клавіатура), `stdout` (стандартний потік виведення, зазвичай, це дисплей) і `stderr` (стандартний потік помилок, зазвичай, це дисплей).

Тип `FILE` визначений в заголовному файлі `stdio.h`. Фактично тип `FILE` – це структура, що містить поля, в які записана інформація про даний файл.

У більшості компіляторів мови `C` файловий тип визначений `stdio.h` наступним чином:

```
struct _iobuf
    {
        char* _ptr; /* поточний покажчик буферу */
        int _cnt;   /* поточний лічильник байтів */
        char* _base /* базова адреса буферу введення-виведення */
        char _flag  /* ознака стану (прапор стану) */
        char _file  /* номер файлу */
    };
    #define FILE struct _iobuf;
```

У деяких системах замість директиви `#define` використовують визначення типу за допомогою ключового слова `typedef`.

Для полегшення роботи з типом `FILE` в файлі `stdio.h` визначено ряд функцій:

`FILE* fopen(const char* filename, const char* mode)` – відкриває файл із заданим ім’ям і повертає покажчик на потік або `NULL`,

якщо спроба відкриття виявилася невдалою. Режим mode допускає наступні значення:

«r» – звичайний текстовий файл відкривається для читання (від read (англ.) – читати);

«w» – звичайний текстовий файл створюється для запису; старий вміст (якщо він був) викидається (від write (англ.) – писати);

«a» – звичайний текстовий файл відкривається або створюється для запису в кінець файлу (від append (англ.) – додавати);

«r+» – звичайний текстовий файл відкривається для виправлення (тобто для читання і для запису);

«w+» – звичайний текстовий файл створюється для виправлення; старий вміст (якщо він був) викидається;

«a+» – звичайний текстовий файл відкривається або створюється для виправлення вже існуючої інформації і додавання нової в кінець файлу.

Режим «виправлення» дозволяє читати й писати в один і той же файл. При переходах від операцій читання до операцій запису й назад повинні здійснюватися звернення до fflush або до функції позиціонування файлу. Якщо покажчик режиму доповнити буквою b (наприклад «rb» або «w+b»), то це означатиме, що файл бінарний. Обмеження на довжину імені файлу задається константою FILENAME_MAX. Константа FOPEN_MAX обмежує число одночасно відкритих файлів.

int fclose(FILE* stream) – виконує дозапис ще незаписаних буферизованих даних, вивільняє всі зв'язані з потоком буфери, після чого закриває потік. Повертає EOF у разі помилки або нуль у іншому випадку;

FILE* freopen(const char* filename, const char* mode, FILE* stream) – відкриває файл з вказаним режимом і пов'язує його з потоком stream, повертає stream або, у разі помилки, NULL. Зазвичай, freopen використовують для заміни файлів, пов'язаних з stdin, stdout або stderr, іншими файлами;

`int fflush(FILE* stream)` – виконує дозапис всіх даних, що залишилися в буфері. Для потоку введення ця функція не визначена. Повертає `EOF` у разі виникнення при записі помилки або нуль у іншому випадку;

`int remove(const char* filename)` – видаляє файл з вказаним ім'ям; подальша спроба відкрити файл з цим ім'ям викличе помилку. Повертає ненульове значення у разі невдалої спроби;

`int rename(const char* oldname, const char* newname)` – змінює ім'я файлу. Повертає ненульове значення у випадку, якщо спроба змінити ім'я виявилася невдалою. Перший параметр задає старе ім'я, другий – нове.

`FILE* tmpfile(void)` – створює тимчасовий файл з режимом доступу «`wb+`», який автоматично видаляється при його закритті або звичайному завершенні програмою своєї роботи. Ця функція повертає потік або `NULL`, якщо не змогла створити файл;

`int fprintf(FILE* stream, const char* format, ...)` – перетворює та записує дані в потік `stream` під управлінням `format`. Повертає значення – число записаних символів або, у разі помилки, негативне значення.

Форматний рядок містить два види об'єктів: звичайні символи, що копіюються у потік виведення, і специфікації перетворення, які викликають перетворення і друк решти аргументів в тому порядку, в якому вони перераховані. Кожна специфікація перетворення починається з «%» і закінчується символом-специфікатором перетворення. Між % і символом-специфікатором в порядку, в якому вони тут перераховані, можуть бути розташовані наступні елементи інформації:

- «-» – знак мінус означає, що даний аргумент повинен вирівнюватись по лівому краю;
- «+» – вказує на те, що перед аргументом завжди виводиться знак;

- « » – символ пробілу вказує на необхідність виведення пробілу перед аргументом при відсутності знаку;

- «0» – символ нуль вказує на необхідність доповнення аргумента нулями зліва до повної ширини поля;

- «#» – даний символ вказує на одну з наступних форм виведення: для 0 – першою цифрою повинен бути 0; для `x` або `X` перед ненульовим результатом повинно виводитись 0`x` чи 0`X`; для `e`, `E`, `f`, `g` та `G` результат повинен містити десяткову крапку; для `g` та `G` кінцеві нулі не відкидаються;

- число, що специфікує мінімальну ширину введення-виведення;

- точка, що відділяє покажчик ширини поля від покажчика точності;

- модифікатори `h`, `l` та `L`. Модифікатор `h` – вказує на те, що число виводиться у форматі `short`, `L` або `l` – у форматі `long`.

Найбільш уживані формати введення-виведення наведено нижче:

- `d`, `i` – знакове десяткове число;

- `o` – вісімкове число без знаку;

- `x`, `X` – шістнадцяткове число без знаку;

- `u` – десяткове ціле число без знаку;

- `c` – один символ;

- `s` – строка;

- `f` – число з плаваючою крапкою;

- `e`, `E` – інженерна форма запису числа з плаваючою крапкою.

`int fscanf(FILE* stream, const char* format, ...)` – здійснює форматоване введення даних із вхідного файлу;

`int fgetc(FILE* stream)` – читає з файлу символ;

`char* fgets(char* s, int n, FILE* stream)` – читає з файлу рядок символів, але не більше за `n` символів;

`int fputc(int c, FILE* stream)` – записує у файл символ;

`int fputs(const char* s, FILE* stream)` – записує у файл рядок символів;

`int fseek(FILE* stream, long offset, int origin)` – встановлює позицію для `stream`; подальше читання або запис проводитиметься з цієї позиції. У разі бінарного файлу позиція встановлюється із зсувом `offset` відносно початку, якщо `origin` дорівнює `SEEK_SET`, відносно поточної позиції, якщо `origin` дорівнює `SEEK_CUR` і відносно кінця файлу, якщо `origin` дорівнює `SEEK_END`. Для текстового файлу `offset` повинен бути нулем або значенням, одержаним за допомогою виклику функції `ftell`. При роботі з текстовим файлом `origin` завжди повинен бути рівний `SEEK_SET`.

`long ftell(FILE* stream)` – повертає поточну позицію потоку `stream` або «`-1L`» у разі помилки;

`int fgetpos(FILE* stream, fpos_t* ptr)` – записує поточну позицію потоку `stream` в покажчик `ptr` для подальшого використання її в `fsetpos`. У разі помилки `fgetpos` повертає ненульове значення;

`int fsetpos(FILE* stream, const fpos_t* ptr)` – встановлює позицію в `stream`, читаючи її з покажчика `ptr`, куди вона була записана раніше за допомогою `fgetpos`. У разі помилки `fsetpos` повертає ненульове значення.

Щоб показати елементарні приклади використання файлів, розглянемо просту програму.

```
#include <stdio.h>
int main()
{
    FILE* in; /* «FILE» обов’язково великими літерами */
    int ch;
```

```
    /* Відкриваємо файл для читання, якщо не кінець файлу, то читаємо з
    файлу символ і відправляємо його в стандартний потік виведення (на
    екран) */
    if((in=fopen("test","r"))!=NULL); /* test – ім'я
                                        файлу, r – режим доступу */
    {
        while(ch=fgetc(in)!=EOF)
            putc(ch, stdout);
        fclose(in); /* закриваємо файл */
    }
    return 0;
}
```

Для роботи із стандартними потоками введення-виведення в файлі `stdio.h` вводяться функції консольного введення-виведення, що є спрощеними варіантами функцій для роботи з файлами. Від функцій для роботи з файлами вони відрізняються тим, що немає необхідності вказувати потік, оскільки кожна з функцій консольного введення-виведення працює лише з конкретним стандартним потоком (`stdin`, `stdout` або `stderr`). Розглянемо ці функції:

`int printf(const char* format, ...)` – відображає на екрані інформацію у відповідності до заданого форматного рядка;

`int scanf (const char* format, ...)` – зчитує інформацію з клавіатури у відповідності до заданого форматного рядка;

`int getchar(void)` – зчитує символ з клавіатури;

`char* gets(char* s)` – зчитує рядок з клавіатури;

`int putchar(int c)` – відображає символ на екрані;

`int puts(const char* s)` – відображає рядок на екрані.

Більш детальну інформацію можна отримати з технічної документації або з [1, 4].

Контрольні запитання

1. З яких елементів складається мова програмування C?
2. Які символи можна використовувати в іменах змінних?
3. Перелічіть розділові символи мови C.
4. Яке призначення спеціальних символів? Перелічіть спеціальні символи, що використовують в мові C.
5. Які операції є допустимими в мові C?
6. Наведіть категорії операцій мови C за пріоритетом.
7. Які операції мають найвищий пріоритет?
8. Що таке константа? Які типи констант є в мові C?
9. Що таке ідентифікатор?
10. Які правила формування ідентифікатора?
11. Що таке ключове слово? Які ключові слова використовують в мові програмування C?
12. Що таке коментар?
13. Які типи операторів є в мові C?
14. Що таке умовний оператор? Для чого його використовують?
15. Що таке порожній оператор? Для чого його використовують?
16. Що таке складений оператор? Для чого його використовують?
17. Які оператори циклів є в мові C? Чим вони відрізняються?
18. Для чого використовують оператор `continue`?
19. Для чого використовують оператор `break`?
20. Що таке оператор вибору? В яких випадках його використовують?
21. Що таке мітка? Яке її призначення?
22. Для чого використовують оператор безумовного переходу?
23. Які типи даних є в мові програмування C?

24. Перелічіть прості типи даних, які використовують в мові C.
25. Що таке масив? Як описати масив у програмі?
26. Що таке структура? Наведіть синтаксис опису структури в програмі.
27. Що таке об’єднання? У чому полягає відмінність між структурою та об’єднанням?
28. Для чого використовують перераховний тип даних у мові C?
29. Дайте визначення покажчика. Які операції над покажчиками є допустимими в мові C?
30. Для чого використовують декларацію `typedef`?
31. Перелічіть класи пам’яті в мові C. Чим вони відрізняються?
32. Що таке функція? Для чого використовують функції?
33. Що таке директиви препроцесора? Яке їх призначення?
34. З яких елементів складається програма на C?
35. Дайте визначення потоку. Які стандартні потоки є в мові C?

2. Особливості мови програмування C++

Мову C++, було розроблено як об'єктно-орієнтоване розширення мови C співробітником фірми *AT&T Laboratories* Бьярном Страуструпом. Завдяки своїй сумісності з мовою C та наявності ефективної реалізації для більшості типів комп'ютерів і операційних систем вона стала однією з основних мов, що втілили ідеї об'єктно-орієнтованого програмування (ООП) в процес практичної розробки як системних, так і прикладних програм [6].

Сьогодні вже не викликає сумніву той факт, що використання принципів ООП допомагає значно прискорити процес розробки програм, що зайвий раз вказує на необхідність освоєння C++ тими, хто хоче займатися розробкою програмного забезпечення. Проте, не дивлячись на це, багатьох відлякує складність C++. Серед програмістів навіть існує думка, що чим краще знаєш C, тим важче освоїти C++.

Насправді – це чистої води помилка. Програмісту, добре знайомому із C, не так уже й важко освоїти C++. Переходити із C на C++ можна скільки завгодно плавно й поступово, користуючись майже повною сумісністю C++ із C «знизу-вгору». І, оскільки C++ підтримує програмування із застосуванням традиційних методів, немає необхідності починати відразу з об'єктів і класів. У C++ є ряд засобів, не пов'язаних безпосередньо ООП, з яких добре починати вивчення мови. Ці засоби роблять значно зручнішим створення навіть звичайних, не об'єктно-орієнтованих програм. І ми почнемо розгляд мови C++ саме з розгляду засобів не пов'язаних безпосередньо з ООП.

2.1. Засоби C++ не пов'язані безпосередньо з ООП

2.1.1. Коментарі в C++

У C++ можна використовувати два види коментарів: звичайні, оформлені за правилами C, та однорядкові, що починаються з символів «//» і продовжуються до кінця рядка [4, 6, 7].

Приклад:

```
/* це звичайний коментар в стилі C */
// а це однорядковий коментар в стилі C++
```

2.1.2. Опис змінних всередині блоку

Мова C вимагає, щоб всі описи локальних змінних усередині блоку поміщалися перед першим виконуваним оператором. У C++ можна описувати змінні в будь-якій точці блоку до першого їх використання. Часто це буває дуже зручно.

Змінні, визначені таким чином, є локальними. Область доступності таких змінних від місця їх визначення і до кінця блоку, в якому вони визначені. Час життя змінної визначається часом виконання даного блоку від місця визначення змінної [4, 6, 7].

Розглянемо приклад.

```
#include <stdio.h>
int main()
    {
        int i; // стандартне визначення в стилі C
        ...
        for(i=0; i<5; i++)
        {
            ...
```

```
        for(int j=0; j<5; j++) // а ось змінна j
                                // визначена в стилі C++
        printf("j=%d",j); // відобразимо її на екрані
        ...
    // змінна j існує в програмі від місця визначення до цієї
    // дужки
    }
    ...
    // а ось змінна i існує до кінця програми
    return 0;
}
```

При використанні таких визначень слід пам’ятати про особливості різних типів C. Наприклад, визначення `int j=0;` еквівалентне операторам `int j; j=0;`, проте у разі використання статичних змінних такої еквівалентності не спостерігається. Це викликано тим, що статичні змінні ініціалізуються тільки один раз, у момент створення, і при подальших входженнях в даний блок мають значення, яке було при попередньому виході з блоку, на відміну від звичайних змінних, які ініціалізуються при кожному входженні в блок, де вони визначені. Ця обставина може призводити до помилок і вимагає від програміста особливої уважності.

2.1.3. Прототипи функцій

У «класичному» C наявність прототипів функцій не є обов’язковою. Це часто призводить до появи помилок, які важко виявити, оскільки компілятор не може перевірити, чи відповідають визначенню даної функції типи аргументів і тип значення, що є результатом дії функції.

C++ вимагає, щоб в кожному модулі, де відбувається звернення до функції, було присутнє або її оголошення (прототип), або її визначення, де

вказано типи аргументів і тип результату [1, 6]. Якщо в модулі знаходиться прототип функції, то її визначення може знаходиться в іншому модулі.

Розглянемо приклад:

```
// вміст файлу header.h
void func1(int, double*); /* void означає, що функція не повертає значень */

double func2(); /* Увага! На відміну від C в C++ відсутність аргументів в описі означає не змінне число аргументів будь-якого типу, а їх відсутність. */

void func3(int, ...); /* у описі func3 вказано, що перший аргумент має тип int, а число і тип інших аргументів може змінюватися */

// вміст файлу mod1.cpp
#include "header.h"
...
int main()
{
    int i;
    double x[20], y;
    ...
    func1(i,x);
    y=func2();
    ...
    func3(i,x,func2);
    ...
    return 0;
}
```

```
// вміст файлу mod2.cpp
void func1(int i, double* x)
{
     ... // тіло функції
}

double func2()
{
     ... // тіло функції
}

void func3 (int i, ...)
{
     ... // тіло функції
}
```

2.1.4. Аргументи за замовчуванням

C++ дозволяє використання аргументів за замовчуванням. Їх застосовують, якщо при виклику функції передано недостатню кількість аргументів. Наведемо приклад: є функція, яка відображує на екрані коло заданого радіусу з центром у заданій точці, і описана наступним чином:

```
//  x – координата по осі OX, y – координата по осі OY, r – радіус
// задані значення – це значення за замовчуванням
void DrawCircle(int x=100, int y=100, int r=10);
```

Тоді виклики цієї функції будуть проінтерпретовані таким чином:

```
// намалює коло з центром в точці (100;100) й радіусом 10.
DrawCircle();
```

```
// намалює коло з центром в точці (200,150) й радіусом 10.
DrawCircle(200,150);

// намалює коло з центром в точці (200,150) й радіусом 20.
DrawCircle(200,150,20);
```

Задавати аргументи за замовчуванням можна для всіх або для декількох аргументів функції, але починати треба завжди з крайнього правого аргументу. Якщо передостанній аргумент має значення за замовчуванням, то і останній також обов’язково повинен мати значення за замовчуванням.

Приклад:

```
// недопустимий опис, бо передостанній аргумент має
// значення за замовчуванням, а останній – ні.
void DrawCircle(int x, int y=200, int r);

// допустимий опис; два крайніх правих аргументи мають
// значення за замовчуванням.
void DrawCircle(int x, int y=200, int r=20);
```

2.1.5. Доступ до глобальних змінних прихованих локальними змінними з тим же ім'ям

У класичному C локальна змінна перекриває глобальну змінну з тим же ім’ям, і нема засобів для доступу до такої глобальної змінної в цій ситуації. У C++ існує оператор дозволу області доступності « `::` », який дає можливість звернутися до такої глобальної змінної.

Приклад:

```
int=0; // глобальна змінна
...
int f()
{
    int i=1; // локальна змінна
    i++; // збільшуємо локальну змінну
    ::i++; // збільшуємо глобальну змінну
    return 0;
}
```

2.1.6. Модифікатори `const` та `volatile`

Модифікатор `const`, як і в звичайному `C`, забороняє зміну значень змінних. Така константа повинна отримати значення при описі, оскільки надалі їй нічого не можна присвоїти.

У `C++` константи, визначені за допомогою модифікатора `const`, мають таку ж область доступності як і статичні змінні. Вони не доступні в інших модулях програми, навіть якщо в цих модулях описати їх як `extern`. Тому, якщо якусь константу необхідно використовувати в різних модулях проекту, то вона повинна бути визначена в кожному модулі, де потрібно її використовувати.

В більшості випадків компілятор трактує константу, описану за допомогою модифікатора `const`, так само, як і іменовану константу, створену директивою `#define`, тобто просто підставляє у відповідних місцях величину, що відповідає значенню даної константи. Проте модифікатор `const` має перевагу перед `#define`, оскільки забезпечує можливість контролю типів на етапі компіляції, що дозволяє уникнути багатьох помилок, які важко визначити.

Модифікатор `volatile`, навпаки, повідомляє компілятор, що значення даної змінної може бути змінено фоновим процесом, наприклад, при обробці переривань. З погляду компілятора це означає, що при обчисленні виразів, що містять таку змінну, потрібно працювати безпосередньо з тією ділянкою пам'яті, де вона записана, а не з її копією, що знаходиться в регістрі процесора.

2.1.7. Передача параметрів за посиланням

У більшості мов програмування параметри передаються в підпрограми або за посиланням, або по значенню. У першому випадку підпрограми працюють із самою змінною і можуть змінювати її значення, а в другому випадку підпрограми працюють з копією значення змінної і саму змінну модифікувати не можуть.

У класичному C параметри передаються тільки по значенню. Щоб мати можливість модифікувати в підпрограмі змінну, передану як параметр, необхідно передати в підпрограму її адресу, використовуючи покажчики. У C++ не потрібно вдаватися до таких хитрощів, бо в ньому реалізований повноцінний механізм передачі параметрів за посиланням.

Приклад:

```
swap(int x, int y); // так передаються параметри по значенню
swap1(int* x, int* y); // так за посиланням в мові C
swap2(int& x, int& y); // а ось так це робиться в C++
```

2.1.8. Модифікатор `inline` в C++

У звичайному C для збільшення швидкодії програми невеликі функції, що часто викликаються, замінюють макросами з параметрами. Це дає необхідний ефект, проте такий підхід сильно заплутує програму й служить невичерпним джерелом помилок, здатних розлютити навіть загартованого програміста, що досконало володіє технікою програмування на C.

C++ пропонує, натомість, використовувати функції, описані як inline. Для того, щоб розібратися, що це дає, розглянемо приклади.

Припустімо, потрібно в C програмі визначити макрос для піднесення числа до квадрату. Здавалося б, це можна зробити таким чином:

```
#define sqr(x) (x*x)
```

Після цього препроцесор буде вирази типу sqr(y) замінювати в програмі на (y*y), і все буде чудово, поки не потрібно буде виконати щось на зразок (y+1)*(y+1).

Запис типу sqr(y+1) буде замінено препроцесором на (y+1*y+1), що зовсім не відповідає бажаному результату.

Для отримання правильного результату слід було б визначити початковий макрос таким чином:

```
#define sqr(x) ((x)*(x))
```

Проте у багатьох випадках ці нюанси не такі очевидні, і часто буває важко розібратися звідки виникає помилка.

Для того, щоб зробити теж саме на C++, достатньо визначити функцію за допомогою модифікатора inline.

```
inline float sqr(float x)
{
    return x*x;
}
```

При цьому ми одержуємо всі ті ж переваги, які дає підстановка макросу, і одночасно отримуємо можливості контролю типів, а також позбавляємося від побічних ефектів, подібних розглянутому вище.

2.1.9. Оператори динамічного розподілу пам’яті

Операція динамічного розподілу пам’яті – це одна з найбільш важливих і часто використовуваних операцій у будь-якій більш-менш серйозній програмі, тому дуже важливо мати зручні засоби для її виконання.

У класичному C для цієї мети існують функції `calloc`, `malloc`, `realloc` та `free`. За допомогою цих операторів можна ефективно керувати розподілом пам’яті під різні динамічні об’єкти. Проте при їх використанні програміст повинен дуже чітко орієнтуватися в роботі з покажчиками й чудово уявляти структуру об’єктів під які виділяється пам’ять, тому динамічний розподіл пам’яті в класичному C викликає священний трепет навіть у бувалих програмістів.

У C++ для цієї мети введені два інтелектуальні оператори: `new` і `delete`, для застосування яких досить лише мати загальне уявлення про покажчики.

Вказані оператори мають наступний синтаксис:

```
new (type[size]);

delete name;
```

Приклад:

```
int* data;
int size,i,j;
scanf("%d", &size};
data = new int[size]; // виділення пам’яті під масив символів
// розміром size. Працювати з таким масивом можна так само, як із
// звичайним, потрібно лише стежити за тим, щоб не вийти за його межі
for (i=0;i<size;i++)
    data[i]= i;
```

```
    ...
// а тепер видалимо його, вивільнену пам'ять можна використати
// повторно для інших цілей
delete data;
...
// виділимо пам'ять під двовимірний масив
// для цього необхідно визначити покажчик на покажчик
float** matrix;
...
// виділення пам'яті під двовимірний масив дещо складніше
matrix=new float* [size];
for(i=0; i<size; i++)
    matrix[i]= new float[size];
// проте результат того вартий, адже у нас є двовимірний масив, з яким
// можна працювати звичайним чином, треба лише відстежувати його
// межі
...
for(i=0; i<size; i++)
    for(j=0; j<size; j++)
        matrix[j][i]=i*j;
...
// вивільнення пам'яті також дещо складніше, але все ж значно простіше,
// ніж в класичному C
for(i=0;i<size;i++)
    delete matrix[i];
delete matrix;
```

2.1.10. Перевантаження функцій

Припустімо, нам по ходу програми часто необхідно виводити на екран значення типу `int`, `double` та `char*`. Для зручності роботи добре б оформити ці операції у вигляді функцій. У класичному `C` для цього потрібно буде визначити три функції:

```
void print_int(int i)
{
    printf("%d",i); return;
}
void print_double(double x)
{
    printf("%f",x); return;
}
void print_string(char* s)
{
    printf("%s",s); return;
}

...
int main ( )
{
    int j=5;
    print_int(j);
    print_double(3.14);
    print_string("Hi, there!");
...
}
```

У C++ можна написати функцію print, що існує в трьох іпостасях:

```
#include <stdio.h>
void print(int i)
{
    printf("%d",i);
}
void print(double x)
{
    printf("%f",x);
}
void print(char* s)
{
    printf ("%s", s);
}
int main ( )
{
    int j=5;
    double e=2.7183;
    float pi=3.1415926;
    print(j);
    print(e);
    print(pi);
    print("Hi, there!");
    return 0;
}
```

У стандартному C цим функціям довелося б давати різні імена, а C++ дозволяє дати їм одне ім’я і при використанні не замислюватися над тим, яку з них викликати у кожному випадку. Компілятор сам визначить для якого типу даних яку функцію викликати. Такі функції називаються перевантаженими. Критерієм вибору служить тип і кількість аргументів.

При цьому істотними є три моменти:

1. Перевантажені функції не можуть розрізнятися тільки типом результату.

```
void f(int, int);
int f(int, int); // помилка !!!
```

2. Перевантаження функцій не повинне призводити до конфлікту з аргументами заданими за замовчуванням.

```
void f(int i=0)
{…}
void f()
{…}
...
f( ) // невідомо, яку функцію викликати
```

3. Не можна перевантажувати функції, оголошені як «`extern "C"`».

2.1.11. Шаблони функцій

При написанні програм на C++ часто доводиться створювати багато майже однакових функцій для обробки даних різних типів. Використовуючи ключове слово `template`, можна задати компілятору зразок, по якому він сам згенерує код, необхідний для конкретного типу.

Для прикладу розглянемо маленьку функцію `swap`, яка обмінює значення двох змінних.

```
#include <string.h>
#include <stdio.h>
template <class T> void swap ( T &a, T &b)
{
    T c;
    c=a; a=b; b=c;
};
```

```
int main()
{
    int i=0, j=1;
    float l=0, k=1;
    char* s1="first string";
    char* s2="second string";
    ...
    swap(i,j);
    swap(l,k);
    swap(s1,s2);
    ...
}
```

2.1.12. Перевантаження операторів

При визначенні в програмі нестандартних типів часто виникає проблема виконання стандартних операцій над ними. Хочеться, щоб ці операції виконувалися над нестандартними типами так само просто, як і над стандартними. Для вирішення цієї проблеми в C++ існує перевантаження операторів.

Для прикладу створімо власний тип – «комплексне число» і визначимо для нього операцію додавання:

```
#include <stdio.h>
// визначаємо власний тип COMPLEX
typedef struct COMPLEX
{
    float re;
    float im;
};
```

```
// перевантажуємо оператор «+» для визначення операції додавання
// комплексних чисел
COMPLEX operator+( COMPLEX a, COMPLEX b)
{
    COMPLEX c;
    c.re=a.re+b.re;
    c.im=a.im+b.im;
    return c;
}

int main ( )
{
    COMPLEX m,n,d;
    m.re=n.re=1;
    m.im=0;
    n.im=1;
    // тепер можна додавати комплексні числа так же просто, як числа
    // будь-якого із стандартних типів
    d=m+n;
    return 0;
}
```

Як видно з прикладу – все дуже просто. Необхідно тільки визначити потрібні дії, пов’язати їх з потрібним оператором і визначити, для якого типу ці дії повинні виконуватися.

При цьому лише необхідно запам’ятати, що перевантажувати можна не всі оператори. Список операторів, які не можна перевантажувати можна знайти в технічній документації.

На цьому опис особливостей C++, що не мають відношення до ООП, можна закінчити. Настав час переходити до найцікавішого – до основ ООП.

2.2. Об’єктно-орієнтоване програмування (ООП)

Архітектура сучасних операційних систем стає все більш об’єктно-орієнтованою. Працювати в таких системах неможливо без володіння основами ООП, тому розглянемо основні поняття та принципи ООП.

2.2.1. Поняття класу. Основні принципи ООП

Основна ідея об’єктно-орієнтованого програмування полягає в об’єднанні даних і алгоритмів для їх обробки в єдине ціле, в деякий абстрактний тип даних. У C++ цей тип, створений програмістом, прийнято називати класом. Клас специфікує члени-дані, які називають полями, і операції для роботи з цими даними, які називають методами [4].

Як вже наголошувалося, клас – це тип даних. Конкретний екземпляр даного типу називається об’єктом даного класу. У принципі, будь-який з раніше розглянутих типів даних можна, в деякому розумінні, вважати класом, а конкретну змінну цього типа – об’єктом [4].

У основі ООП лежать три основні поняття – інкапсуляція, успадкування і поліморфізм.

Інкапсуляція – це об’єднання в єдине ціле даних (полів) і алгоритмів для їх обробки (методів) [1, 4, 6].

Успадкування – це властивість класів, яка дозволяє розробляти нові (породжені) класи на основі вже існуючих (базових). При цьому породжені класи можуть успадковувати властивості базових [4, 6].

Поліморфізм – це властивість споріднених класів, породжених від загального базового класу, вирішувати схожі за змістом задачі різними способами [4, 6].

Фактично, об’єктно-орієнтоване програмування – це модульне програмування нового рівня, коли основний акцент переноситься на смисловий зв’язок між даними й процедурами їх обробки.

Приклад:

```
// float – це клас (тип), а j – це об’єкт класу (типу) float
float j;

// AnyClass – це клас (тип)
class AnyClass
{
    // в розділі private описуються закриті поля і методи
    // до них мають доступ тільки функції-члени даного класу, а також
    // функції-друзі класу
    private:
    int x;
    double x,y;
    ...
    void f1();
    int f2();
    // в розділі public описуються загальнодоступні поля і методи
    // вони призначені для забезпечення інтерфейсу об’єкту з
    // програмою
    public:
    char c1, c2;
    int f3(int, int);
}
    ...
// a – це об’єкт класу (типу) AnyClass
AnyClass a;
```

Мітки «private:» і «public:» визначають режим доступу до членів класу. Поля і методи, описані в розділі «private:» є закритими,

доступ до них мають тільки методи даного класу або функції-друзі класу (про них мова йтиме далі). Поля і методи з розділу «public:» є загальнодоступними. Вони призначені, як правило, для забезпечення інтерфейсу об'єкту з програмою.

Кількість розділів «private:» і «public:», а також порядок їх розташування в описі класу – довільні.

Для визначення класу допустимо використовувати ключові слова struct та union. При цьому є деякі відмінності.

При використанні ключового слова «class» всі поля і методи, не включені явно в розділи «private:» і «public:», є за замовчуванням закритими, а при використанні ключового слова «struct» – відкритими. При використанні ключового слова union члени класу можуть бути тільки загальнодоступними.

В більшості випадків визначення класу не локалізоване в блоці, і областю доступності класу є весь файл, в якому він визначений.

Визначення функцій-членів класу можуть знаходитися безпосередньо усередині визначення класу (якщо вони дуже короткі), тоді вони автоматично вважаються inline-функціями. Можна визначати члени класу поза визначенням об'єкту. Тоді для завдання приналежності функції до даного класу використовується оператор « :: ».

Приклад:

```
int AnyClass::f3(int x, int y)
    {
    ...
    }
```

Таке визначення потрібно розмістити до першого використання даної функції. При бажанні такі функції можуть бути об'явлені inline-функціями за допомогою ключового слова inline.

Слід зазначити, що визначення класу не створює об’єктів даного класу. Об’єкти створюються тільки при описі змінних:

```
AnyClass a,b;
```

І ще одне важливе зауваження: при роботі з багатомодульними проектами визначення класу повинно бути присутнім у всіх модулях, де використовуються об’єкти даного класу або визначаються його функції-члени, тому визначення класу та всіх його функцій краще помістити в заголовний файл і підключати його при необхідності директивою `#include`.

2.2.2. Доступ до полів і методів класу

Доступ до відкритих полів класу здійснюється так само, як в структурах і об’єднаннях: з використанням операторів прямого «`.`» та опосередкованого «`->`» вибору.

Приклад:

```
class MyClass
{
    // за замовчуванням i є закритим членом
    int i;
    public:
    // state – відкритий член
    int state;
    int get_i();
    void set_i(int);
    int get_state();
};
int MyClass::get_i() {return i;}
void MyClass::set_i(int x) {i=x; return;}
int MyClass::get_state() {return state;}
    ...
```

```
int main()
{
    int m,n,s;
    MyClass obj;
    MyClass* obj_ptr=&obj;
    ...
    //  пряме звернення до відкритого члена
    obj.state=1;
    ...
    s=obj.state;
    // опосередковане звернення при роботі з покажчиком на об'єкт
    n=obj_ptr->state;
    ...
    n=2;
    obj_ptr->state=n;
    // а ось так не можна, оскільки i – закритий член
    // obj.i=1;
    // m=obj_ptr->i;
    // треба так
    obj.set_i(1);
    m=obj.get_i();
    n=obj_ptr->get_i();
}
```

2.2.3. Статичні члени класу

Кожен створюваний об'єкт класу має свою базову адресу. Таким чином, однойменні члени різних об'єктів даного класу ніяк не пов'язані між собою. А як же бути якщо необхідно, щоб всі об'єкти даного класу мали деякий загальний член? Немає нічого простішого. Достатньо описати його як `static`.

Приклад:

```
#include <stdio.h>
class prob_stat
{
    private:
    static int counter;
    public:
    void set_stat(){++counter;}
    int get_stat(){return counter;}
};

int prob_stat::counter=0;

int main()
{
    prob_stat a,b;
    printf("before all set_stat()\n");
    printf("a=%d b=%d\n",a.get_stat(),b.get_stat());
    a.set_stat();
    printf("after a.set_stat()\n");
    printf("a=%d b=%d\n",a.get_stat(),b.get_stat());
    b.set_stat();
    printf("after b.set_stat()\n");
    printf("a=%d b=%d\n",a.get_stat(),b.get_stat());
    return 0;
}
```

У даному прикладі створено два об’єкти класу `prob_stat` – `a` та `b`. Клас `prob_stat` містить статичну змінну `counter`, значення якої є спільним для об’єктів `a` і `b`. Кожен з цих об’єктів має доступ на читання і

редагування значення змінної counter, при цьому будь-яка зміна значення counter відображається у всіх об’єктах класу prob_stat.

2.2.4. Друзі класу

Іноді виникає необхідність, щоб якась функція, що не є членом класу, мала доступ до закритих полів об’єктів даного класу. У C++ є така можливість. Для цього необхідно оголосити дану функцію другом цього класу. Розглянемо це на прикладі:

```
#include <stdio.h>
class prob
{
    private:
    int status;
    public:
    int get_status(){return status;}
    void set_status(int x){status=x; return;}
    void friend prob_frd(prob &obj);
};

int main()
{
    prob a;
    a.set_status(1);
    printf("before friend's help a.status=%d\n",
                                        a.get_status());
    prob_frd(a);
    printf("after friend's help a.status=%d\n",
                                        a.get_status());
    return 0;
}
```

```
void prob_frd(prob &obj)
{
    obj.status++;
    return;
}
```

У даному прикладі функція `prob_frd` описана, як друг класу `prob`. Це дозволяє їй отримати доступ до закритого поля «`status`» даного класу.

Оскільки в програмі може існувати одночасно декілька об’єктів даного класу, такій функції необхідно передавати як параметр покажчик на об’єкт, з яким вона повинна працювати в даний момент.

Аналогічним чином можна цілий клас описати, як друг даного класу. Наприклад, визначити клас `a`, як друг класу `b`. Тоді методи об’єктів класу `a` дістануть доступ до закритих полів об’єктів класу `b`.

Приклад:

```
#include <stdio.h>

class MyClass1;

class MyClass2
{
    private:
    int status;
    public:
    int get_status(){return status;}
    void set_status(int x){status=x; return;}
    friend MyClass1;
};
```

```
class MyClass1
{
    public:
    void set_status(MyClass2 &obj, int x)
                                {obj.status=x;return;}
};

int main()
{
    MyClass2 a;
    MyClass1 b;
    a.set_status(1);
    printf("before friend's help a.status=%d\n",
                                        a.get_status());
    b.set_status(a,2);
    printf("after friend's help a.status=%d\n",
                                        a.get_status());
    return 0;
}
```

У даному прикладі клас `MyClass1` оголошується другом класу `MyClass2`. Це дозволяє методу `set_status` класу `MyClass1` звертатися до закритого поля `status` класу `MyClass2`.

Для того, щоб мати можливість зробити це оголошення довелося використовувати випереджуючий неповний опис класу `MyClass1`. Без такого опису компілятор видав би помилку. Крім того методу `set_status` класу `MyClass1` для реалізації доступу до конкретного об’єкту необхідно знати адресу цього об’єкту. Тому, як один з аргументів, йому передається покажчик на об’єкт класу `MyClass2`.

Наявність механізму установлення дружніх відносин дозволяє моделювати досить складні відносини між класами, що, у свою чергу, значно полегшує створення програм для вирішення складних практичних завдань.

2.2.5. Перевантаження операторів для класів

Перевантаження операторів для типів, введених користувачем, вже розглядалося раніше. Проте тоді його розглядали поза концепцією ООП. Тепер же ми маємо достатньо інформації, щоб узагальнити поняття перевантаження операторів на класи.

Як ви вже, напевно, здогадалися, перевантажувати оператори для класів можна або описуючи оператор як метод класу, або використовуючи дружню функцію. Розглянемо обидві можливості.

Спочатку використаємо дружню функцію для перевантаження оператора додавання. Перевантажимо цей оператор для складання двох об’єктів класу `complex`, що моделює комплексне число.

Приклад:

```
#include <stdio.h>

class complex
{
    private:
    float re,im;
    public:
    void set_value(float x, float y)
                                    {re=x;im=y;return;}
    float get_re(){return re;}
    float get_im(){return im;}
    friend complex operator+(complex&,complex&);
};
```

```
complex operator+(complex &a, complex &b)
{
    complex c;
    c.re=a.re+b.re;
    c.im=a.im+b.im;
    return c;
}

int main()
{
    complex a,b,c;
    a.set_value(1,-1);
    printf("a=(%f, %f)\n", a.get_re(), a.get_im());
    b.set_value(2,0);
    printf("b=(%f, %f)\n", b.get_re(), b.get_im());
    c=a+b;
    printf("c=(%f, %f)\n", c.get_re(), c.get_im());
    return 0;
}
```

А зараз для класу `vector` перевантажимо оператор «++». Нехай для нашого класу `vector` ця операція означає додавання одиничного вектора до початкового вектора. Крім того, перевантажений оператор визначимо локально усередині класу.

Приклад:

```
#include <stdio.h>
class vector
{
    private:
    float x,y,z;
```

```
    public:
    void set_value(float xv, float yv, float zv)
                          {x=xv; y=yv; z=zv; return;}
    float get_x(){return x;}
    float get_y(){return y;}
    float get_z(){return z;}
    void operator++(){x++; y++; z++; return;}
};
int main()
{
    vector a;
    a.set_value(1,0,-1);
    printf("a=(%f, %f, %f)\n", a.get_x(), a.get_y(),
                                          a.get_z());
    ++a;
    printf("a=(%f, %f, %f)\n", a.get_x(), a.get_y(),
                                          a.get_z());
    return 0;
}
```

У даному прикладі унарний оператор ++ перевантажений в префіксній формі. Проте, починаючи із стандарту *AT&T* 2.1, C++ надає можливість перевантажувати як префіксну, так і постфіксну форми унарних операторів (за умови, що такі мають місце).

2.2.6. Ініціалізація та знищення об’єктів. Конструктори й деструктори

При роботі з об’єктом програма виконує ряд дій, а саме:

- виділення пам’яті під об’єкт;
- ініціалізація об’єкту;
- знищення об’єкту.

Для виконання цих операцій в компіляторі передбачений стандартний набір дій. Проте часто цей набір не задовольняє розробників. Це буває, наприклад, якщо потрібно задати значення за замовчуванням для тих чи інших полів об’єкту, або якщо механізм ініціалізації і знищення об’єктів даного класу залежить від ситуації і т.д.

В C++ передбачена можливість заміни стандартних процедур шляхом використання конструкторів і деструкторів.

Конструктор – це функція-метод класу з таким самим ім’ям, як у класу. Цей метод призначений для створення та ініціалізації об’єктів класу, виділення пам’яті під об’єкти й присвоювання початкових значень полям об’єкту.

Деструктор – це функція-метод класу, яка відповідає за коректне вивільнення пам’яті при знищенні об’єкту. Ім’я функції-деструктора завжди таке саме, як і у класу, до якого додається символ «~» на початку.

Конструктори й деструктори не можуть повертати значень, і тому у їх описі відсутній тип результату.

Конструктор може не мати вхідних параметрів. В такому випадку його називають конструктором за замовчуванням.

Клас може мати кілька конструкторів і лише один деструктор. При використанні кількох конструкторів діють ті самі правила, що і для перевантажених функцій.

Приклад:

```
#include <stdio.h>
class complex
{
    private:
    float re, im;

    public:
    // конструктор за замовчуванням
```

```
    complex(){re=0;im=0;return;}
    // конструктор з вхідними параметрами
    complex(float x, float y){re=x; im=y; return;}
    void set_value(float x, float y) {re=x;im=y;return;}
    float get_re(){return re;}
    float get_im(){return im;}
};

int main()
{
    // при визначенні об'єкту використовується конструктор за
    //замовчуванням. Об'єкт a ініціалізується нулями за
    // замовчуванням
    complex a;
    // при визначенні об'єкту використовується конструктор з вхідними
    // параметрами. Об'єкт b ініціалізується значеннями 2 і 3
    complex b(2,3);
    printf("a=(%f, %f)\n", a.get_re(), a.get_im());
    printf("b=(%f, %f)\n", b.get_re(), b.get_im());
    return 0;
}
```

У даному прикладі клас `complex` має два конструктори. Конструктор за замовчуванням, який створює об'єкт даного класу ініціалізований нулями, і інший конструктор, що дозволяє при створенні об'єкту ініціалізувати його будь-якими необхідними значеннями. Знищення об'єкту проводиться стандартним способом, оскільки деструктор класу не визначено.

Оскільки даний клас має два конструктори, то визначати об'єкти даного класу можна у два способи, а саме за замовчуванням та із заданим значенням ініціалізації.

2.2.7. Шаблони класів

Шаблони класів називають «генераторами класів», або «родовими класами», або «узагальненими класами». Вони дозволяють визначити структуру сімейства класів, по якій компілятор надалі створює класи ґрунтуючись, як і в разі шаблонів функцій, на параметрах, що задаються. Найбільш типовий приклад їх використання – це створення контейнерних класів, наприклад, векторів для розміщення об’єктів довільних типів.

Приклад:

```
#include <iostream>

using namespace std;

template <class T> class Vector
{
    private:
    T* elements;
    int size;
    public:
    Vector(int razm=0);
    ~Vector(){delete elements;}
    T& operator[](int i) {return elements[i];}
    void print_contents();
};

template <class T>
Vector<T>::Vector(int razm)
{
    elements=new T[razm];
    for(int i=0; i< razm; i++) {elements[i]=(T) 0;}
```

```
    size=razm;
};
template <class T>
void Vector<T>::print_contents()
{
    cout << "elements num-"<<size<<"\n";
    for(int i=0; i<size; i++)
    cout << "el[" << i << "]=" << elements[i]<< "\n";
}

int main()
{
    int razmer=10;
    Vector <int> i(razmer);
    Vector <float> x(razmer);
    Vector <char> z(razmer);
    for(int count=0;count<razmer;count++)
    {
        i[count]=count;
        x[count]=count/10;
        z[count]='a'+count;
    }
    i.print_contents();
    x.print_contents();
    z.print_contents();
    return 0;
}
```

2.2.8. Успадкування. Поліморфізм. Доступ до базових класів

Як уже було сказано, успадкування та поліморфізм – головні механізми ООП. З їх допомогою можна розробляти дуже складні класи, просуваючись від загального до приватного, а також «нарощувати» вже створені, одержуючи з них нові класи з необхідними властивостями.

Приступаючи до проектування складного класу необхідно з'ясувати, якими найбільш загальними властивостями характеризується даний клас, і чи немає вже готового класу, який би можна було «доробити». Знайшовши такий клас потрібно успадкувати необхідні властивості й додати нові (або змінити деякі з тих, що вже існують). При цьому потрібно пам'ятати, що в C++ можливе множинне успадкування. Це означає, що необхідний клас можна побудувати на основі використання декількох базових класів одночасно.

Часто, при програмуванні практичних завдань, виникає необхідність створення декількох, майже однакових класів, що відрізняються один від одного реалізацією одного або декількох членів класу. Часто такі класи виконують однакові дії над даними, що відрізняються по типу. В цьому випадку доцільно створити клас, який став би базовим для кожного з них, і будувати ці подібні класи на його основі. Тоді ці споріднені класи будуть як би різними формами одного і того самого класу, призначеними для роботи з різними типами. Таку здатність об'єкта реагувати на деякий запит згідно свого типу в ООП називають поліморфізмом.

Спільне використання успадкування і поліморфізму дозволяє моделювати в програмах складні процеси реального світу. Для ілюстрації вищесказаного наведемо приклад створення нового об'єкту на основі двох базових об'єктів з використанням успадкування їх членів і заміною двох функцій-членів, а саме конструктора й функції для виведення вмісту об'єкту на екран.

Настійно рекомендуємо звернути увагу на визначення конструктора результуючого класу й на звернення до конструкторів базових класів, а також на використання оператора « :: » при зверненні до однойменних членів-функцій get_info() базових класів з функції get_info() новоствореного класу. Порівняйте відмінності при роботі з конструкторами й з іншими функціями-членами.

Приклад:

```
#include <iostream>
#include <string.h>
using namespace std;
class human
{
    private:
    char* name;
    char* surname;
    public:
    human(char* h_name="anybody",
                        char* h_surname="anybody");
    ~human(){delete name; delete surname; return;}
    void get_info(){cout << "\n name - " << name <<
            "\n surname - " << surname <<"\n"; return;}
};
human::human(char* h_name, char* h_surname)
{
    int len;
    len=strlen(h_name);
    name = new char[len+1];
    strcpy(name,h_name);
    len=strlen(h_surname);
    surname = new char[len+1];
```

```
    strcpy(surname,h_surname);
    return;
};

class qualification
{
    private:
    char* trade;
    int time;
    public:
    qualification(char* q_trade="NO", int q_time=0);
    ~qualification(){delete trade; return;}
    void get_info(){cout << "\n my trade is -" <<
                trade << "\n work in my trade - " <<
                time << "-year(s)\n";
                return;}
};

qualification::qualification(char* q_trade,
                             int q_time)
{
    int len;
    len=strlen(q_trade);
    trade=new char[len+1];
    strcpy(trade,q_trade);
    time=q_time;
    return;
}
```

```
class employee: human, qualification
{
    public:
    employee(char* h_name, char* h_surname,
                          char* q_trade, int q_time);
    void get_info(){human::get_info();
                qualification::get_info(); return;}
};

employee::employee(char* h_name, char* h_surname,
                          char* q_trade, int q_time):
human(h_name,h_surname), qualification(q_trade,q_time)
{
    return;
}

int main()
{
    human a("Dmitry", "Tatarchuk");
    qualification b("doctor",20);
    employee c("Yuriy","Didenko","lecturer",10);
    a.get_info();
    b.get_info();
    c.get_info();
    return 0;
}
```

Навіть під час розгляду такого простого прикладу виникає питання: а як же здійснювати доступ до членів базових класів? Для відповіді на це питання слід запам’ятати наступні положення:

- доступ до відкритого члена базового класу здійснюється таким же чином, як і до відкритого члена звичайного класу;

- якщо об’єкт побудовано на основі декількох базових класів і в цих класах або в похідному класі є відкриті члени з однаковими іменами, то для доступу до цих членів необхідно використовувати оператор « :: »;

- для доступу до закритих членів базових класів необхідно використовувати відкриті члени базових класів і при необхідності використовувати оператор « :: »;

- у класах можна визначати не тільки розділи «private:» та «public:», але і ще один розділ – «protected:». Члени цього розділу є закритими, але мають одну особливість. Вони доступні функціям-членам похідних класів, а також функціям-друзям цих похідних класів.

Більш докладну інформацію з цього питання можна одержати в документації по компілятору, а також в літературі по ООП із застосуванням C++ [4, 6, 7].

2.2.9. Стандартні потоки введення-виведення

У мові C++, як і в C, відсутні вбудовані оператори введення-виведення. У програмах на C++ операції введення-виведення зазвичай здійснюються за допомогою функцій, що містяться в стандартних бібліотеках [2, 4, 6].

У C++ при організації введення-виведення ключовим поняттям є потік. Відповідно до концепції C++ потік визначається, як деякий абстрактний тип даних, що представляє деяку послідовність елементів даних, спрямовану від джерела до приймача. Кількість елементів потоку називають його довжиною length, а порядковий номер деякого доступного в даний момент елементу називають поточною позицією. Кожен потік має один з трьох можливих режимів доступу:

- тільки для читання;

- тільки для записування;

- для читання і записування.

На основі поняття потоку в C++ для введення-виведення створена стандартна бібліотека потокового введення-виведення `iostream`. У цій бібліотеці визначені стандартні потоки введення-виведення:

- `cin` – надає послідовний доступ на читання стандартного вхідного потоку. Зазвичай він пов'язаний з клавіатурою, але може бути перевизначений;
- `cout` – надає послідовний доступ на записування в стандартний вихідний потік. Зазвичай він пов'язаний з дисплеєм, але може бути перевизначений;
- `cerr` – надає доступ на записування в стандартний потік помилок. Зазвичай він пов'язаний з дисплеєм. Від потоку `cout` відрізняється тим, що коли стандартний вихідний потік перевизначений у файл, то цей потік все одно пов'язаний з дисплеєм;
- `clog` – повністю аналогічний потоку `cerr`, але використовує буферизацію.

Крім того в цій бібліотеці визначені перевантажений оператор форматованого введення даних з потоку « `>>` », перевантажений оператор форматованого виведення даних в потік « `<<` », маніпулятори й прапори, що визначають формати даних, а також ряд функцій неформатованого введення-виведення.

Самі потоки в C++ реалізовані як класи. У основі ієрархії класів лежить клас `ios`, на основі якого породжені класи `ostream` (описує вихідний потік) та `istream` (описує вхідний потік).

2.2.10. Маніпулятори

Маніпулятори введення-виведення – це визначені об’єкти, які впливають на інформацію, змінюючи її формат або додаючи до неї керуючі символи [1]. В C++ визначено наступні маніпулятори:

- dec – зумовлює десятковий формат виведення;
- endl – додає символ нового рядка;
- ends – додає нуль-символ;
- flush – примусово скидає інформацію з буфера в потік;
- hex – зумовлює шістнадцятковий формат виведення;
- oct – зумовлює вісімковий формат виведення.

Приклад:

```
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    ...
    // прочитати з вхідного потоку (клавіатури) значення змінних a, b
    cin >> a >> b;
    ...
    // вивести ці змінні у вихідний потік (на екран) у шістнадцятковому
    // форматі й перейти на іншу строку
    cout << hex << a << " " << b << endl;
    ...
}
```

2.2.11. Потокові класи

Як уже говорилося, в основі ієрархії класів, що описують потоки введення-виведення, лежить клас `ios`, на основі якого породжені класи `ostream` (описує вихідний потік) та `istream` (описує вхідний потік). Кожний з цих класів має свої функції-члени, призначені для виконання певних операцій з потоками. Розглянемо ці класи.

Клас* `ios`. *Функції-члени класу:

`int bad()` – повертає ненульове значення, якщо була помилка;

`void clear(int=0)` – встановлює стан потоку в нульове положення;

`int eof()` – повертає ненульове значення, якщо має місце умова кінця файлу (`EOF`);

`int fail()` – повертає ненульове значення, якщо попередня операція закінчилась невдало;

`char fill()` – повертає поточне значення символу заповнення потоку;

`char fill(char)` – встановлює нове значення символу заповнення, повертає попереднє значення;

`long flags()` – повертає поточне значення прапорів форматування;

`long flags(long)` – встановлює прапори форматування у вказане значення, повертає попереднє значення;

`int good()` – повертає ненульове значення, якщо не встановлений жоден прапор стану (помилок немає);

`int precision()` – повертає поточне значення точності дійсних чисел;

`int precision(int)` – встановлює вказане параметром значення точності дійсних чисел, повертає попереднє значення;

`streambuf* rdbuf()` – повертає покажчик на буфер (об’єкт класу `streambuf`) пов’язаний з потоком;

`int rdstate()` – повертає поточний стан потоку;

`long setf(long)` – встановлює прапори введення-виведення у вказане значення, повертає попереднє значення;

`long setf(long _setbits, long _field)` – скидає біти стану відмічені у змінній `_field`, у стан вказаний у змінній `_setbits`;

`ostream* tie()` – повертає покажчик на паралельний потік;

`ostream* tie(ostream)` – організує потік паралельний вказаному, повертає покажчик на попередній паралельний потік;

`long unsetf(long)` – очищає біти стану потоку відмічені переданим параметром, повертає попереднє значення стану потоку;

`int width()` – повертає поточне значення ширини поля введення-виведення;

`int width(int)` – встановлює ширину поля введення-виведення, повертає попереднє її значення.

Клас* `ios`*. Прапори форматування:

`dec` – використовує десяткову систему числення для цілих чисел;

`fixed` – показує дійсні числа в стандартній формі з десятковою крапкою (наприклад, 5.35);

`hex` – використовує шістнадцяткову систему числення для цілих чисел;

`internal` – заповнює символами простір між знаком або основою системи й даними, якщо ширина поля більша, ніж розмір поля виведення;

`left` – вирівнює символи по лівому краю;

`oct` – використовує вісімкову систему числення для цілих чисел;

`right` – вирівнює символи по правому краю;

`scientific` – показує числа з плаваючою крапкою в інженерному форматі (наприклад, 5.35e2);

`showbase` – виводить 0X або 0 перед шістнадцятковим або вісімковим значеннями відповідно;

`showpoint` – відображає всі дійсні числа з обов'язковою десятковою крапкою і кінцевими нулями;

`showpos` – показує знак «+» для додатного числа;

`skipws` – пропускає початкові символи пробілу при введенні числа;

`stdio` – записує потоки `stdout` та `stderr` на диск після кожної операції введення-виведення;

`unitbuf` – примусово записує вміст буфера на диск після кожної операції виведення;

`uppercase` – виводить у верхньому регістрі шістнадцяткові цифри, символ експоненти (`E`) та шістнадцятковий префікс (`0X`).

Наприклад:

```
#include <iostream>
using namespace std;
int main()
    {
//встановити виведення чисел з плаваючою крапкою в інженерному форматі
        cout.setf(ios::scientific);
// показувати знак «+» перед додатними числами
        cout.setf(ios::showpos);
// вивести на екран число 503.5, використовуючи встановлені вище
// правила
        cout << 503.5 << endl;
        return 0;
    }
```

Клас* `ios`*. Прапори режиму відкриття потоку:

`in` – відкриття потоку для читання;

`out` – відкриття потоку для записування;

`ate` – перехід до кінця потоку перед його відкриттям;

`app` – відкриття потоку в режимі доповнення;

`trunc` – відкриття потоку з усіканням до нульової довжини, увесь вміст потоку втрачається;

`nocreate` – відкриття вже існуючого потоку, за відсутності потоку повертає помилку;

`noreplace` – створення потоку, якщо файл вже існує, то повертає помилку;

`binary` – відкриття потоку в бінарному (двійковому) режимі.

Класс `ostream`

Клас `ostream` породжений від класу `ios` і успадковує всі його властивості. Він описує вихідний потік і доповнений функціями-членами, що дозволяють виконувати додаткові операції над вихідними потоками:

`flush()` – примусово пересилає дані з буфера потоку в сам потік;

`seekp(long)` – встановлює покажчик потоку на абсолютну позицію, задану параметром;

`seekp(long, seek_dir)` – переміщує покажчик поточної позиції на число позицій, вказане в першому параметрі щодо точки вказаної в другому параметрі (0 – від початку, 2 – від кінця, 1 – від поточної позиції у напрямі до кінця);

`put(char)` – поміщає символ у потік виведення;

`tellp()` – повертає поточну позицію потоку виведення;

`write(const signed char*, int n)` – поміщає в потік виведення `n` символів з масиву, на який вказує перший параметр;

`write(const unsigned char*, int n)` – те саме, що і попередня функція.

Класс `istream`

Вхідні потоки описуються класом `istream`, який також породжений від класу `ios` і успадковує всі його властивості. Для управління вхідним потоком в ньому додатково визначені функції:

`gcount()` – повертає число символів, зчитаних з потоку;

`get()` – одержує з потоку введення одиночний символ;

`get(signed char*, int len char='\n')` – отримує з потоку введення в буфер `len`-1 символ, може витягнути менше символів, якщо зустріне `EOF` або символ-роздільник, заданий третім параметром;

`get(unsigned char*, int len char='\n')` – те саме, що і попередня функція;

`get(unsigned char&)` – отримує з потоку введення символ і поміщає його у вказаний байт;

`get(signed char&)` – те саме, що і попередня функція;

`get(streambuf&, char='\n')` – отримує символи з потоку у вказаний буфер, поки не зустріне символ-роздільник;

`getline(signed char*, int, char='\n')` – те саме, що і `get` з трьома параметрами, але символ-роздільник розташовується у рядку;

`getline(unsigned char*, int, char='\n')` – те саме, що і що попередня функція;

`ignore(int n=1, int delim=EOF)` – пропускає `n` символів в потоці, може пропустити менше, якщо зустріне `EOF`;

`peek()` – отримує з потоку наступний символ;

`putback(char)` – повертає символ назад в потік;

`read(signed char*, int)` – отримує з потоку в буфер задане число символів;

`read(unsigned char*, int)` – те саме, що і попередня функція;

`seekg(long)` – встановлює покажчик на задану позицію;

`seekg(long, seek_dir)` – повністю аналогічна функції `seekp` члену класу `ostream`;

`tellg()` – повертає поточну позицію.

Приклад:

```
#include <iostream>
using namespace std;
int main()
{
    char* buff;
    int max_len=21;
    buff=new char[max_len];
    // читати з потоку введення max_len-1 символів у буфер buff
    cin.read(buff, max_len-1);
    // записати в потік виведеня max_len-1 символ з буфера buff
    cout.write(buff, max_len-1);
    return 0;
}
```

2.2.12. Файлове введення-виведення

Файлове введення-виведення є похідним від потокового введення-виведення. В основі файлового введення-виведення лежать три потокові класи:

`fstream` – похідний від `iostream` клас, який пов’язує файл з прикладною програмою для виконання як операцій введення, так і операцій виведення інформації;

`ifstream` – похідний від `istream` клас, що пов’язує файл з прикладною програмою для виконання тільки операцій введення інформації;

ofstream – похідний від ostream клас, що пов’язує файл з прикладною програмою для виконання тільки операцій виведення інформації.

Описи цих класів містяться в файлі fstream.h. При його підключенні автоматично підключається також файл iostream.h.

Оскільки файлові класи є нащадками потокових класів, то вони успадковують операції << і >>, а також функції-члени й поля потокових класів, розглянутих вище.

Приклад:

```
#include <fstream>
#include <iostream>
#include <string.h>
using namespace std;
int main()
{
    char* buffer;
    buffer=new char[256];
    cout << "This program write data in file" <<endl;
// зчитування з клавіатури імені файлу
    cout << "Input filename" << endl;
    cin >> buffer;
//  декларація об’єктів файлового типу fout для записування інформації у файл
    ofstream fout(buffer, ios::out);
// fin для зчитування інформації з файлу
    ifstream fin(buffer, ios::in);
// перевірка на виникнення помилки при відкритті файлу
    if(!buffer)
    {
        cerr << "error!" << endl;
        return 0;
    }
```

```
// якщо помилки не було, продовжуємо виконання програми
    cout << "input text in file" << endl;
// записуємо до файлу строки "Good method" та "Hello, All"
    fout<< "Good method" <<endl<<"Hello, All"<< endl;
    cout << "Now reading data from file" << endl;
// зчитуємо з файлу записане по строках і виводимо на екран
    fin.getline(buffer,256);
    cout << buffer << endl;
    fin.getline(buffer,256);
    cout << buffer << endl;
// закриваємо доступ на зчитування з файлу
    fin.close();
// закриваємо доступ на записування до файлу
    fout.close();
    return 0;
}
```

Більш докладну інформацію можна отримати із літератури, наведеної в списку використаної та рекомендованої літератури, а також у документації по компілятору.

2.3. Вимоги до оформлення програм

При написанні програм необхідно використовувати правильний стиль оформлення файлів з програмним кодом. Це необхідно для того, щоб легше було розуміти програму, а також вносити до неї зміни. Крім того використання правильного стилю полегшує розробку складних програм, коли над проектом працює група програмістів.

Поняття правильного стилю оформлення файлів з програмним кодом не є стандартом. Воно включає набір поширених серед програмістів інтуїтивно зрозумілих правил, а саме:

- програма має бути складена з невеликих зрозумілих процедур (функцій), кожна з яких виконує певні дії, що використовуються в програмі багаторазово. Імена процедур (функцій), а також всіх змінних повинні бути сформовані таким чином, щоб було зрозуміло їх функціональне призначення;
- програма має бути добре прокоментована. При описі кожної нової змінної необхідно вказувати її призначення, а при описі процедури (функції) – призначення, вхідні параметри та результат дії;
- в кожному рядку тексту програми має знаходитись лише один оператор;
- оператори у файлах з програмним кодом мають бути розміщені так, щоб легко було виділяти логічні (функціональні) блоки коду. Зазвичай, це роблять за допомогою відступів, таким чином, щоб оператори певного логічного блоку програми мали однакові відступи. Крім того кожен логічний блок треба добре коментувати.

2.4. Вимоги до програм

Для того, щоб програмою було зручно користуватися, вона має задовольняти ряду вимог:

- відповідати технічному завданню;
- бути ефективною;
- раціонально використовувати ресурси ЕОМ;
- мати дружній інтерфейс з користувачем, який виключає можливість тяжких наслідків для роботи й для обчислювальної системи внаслідок неправильних дій користувача.

2.5. Методи розробки програм

Найбільш сучасним методом розробки програм є метод низхідного програмування. Відповідно до цього методу створення програми починається «зверху», тобто з розробки «скелету» програми. Для цього алгоритм програми розбивається на логічні блоки, кожен з яких виконує певну закінчену дію. Кожен з таких блоків реалізується окремою процедурою або функцією. Часто на цьому етапі ще не відомі деталі реалізації цих функцій, тому спочатку вони реалізуються з пустим тілом. На цьому етапі визначається кількість вхідних параметрів та їх типи, а також тип результату (для функцій). Такі процедури і функції називають заглушками. При цьому добиваються, щоб програма з цими заглушками компілювалась і виконувала мінімальний набір дій – коректне запускання програми та її коректне завершення. Коли цей етап завершено, починається реалізація алгоритмів роботи заглушок. Якщо алгоритм заглушки складний, то його розбивають на набір більш простих процедур і функцій, які викликаються програмою при виконанні даної процедури (функції). Ці процедури спочатку також реалізують як заглушки, поступово наповнюючи їх конкретним змістом. Цей процес продовжується «вниз» доки не буде отримано повністю роботоздатної програми, яка реалізує всі необхідні дії.

2.6. Методи налагодження програм

Після того, як програму розроблено, починається етап її налагодження. Лише у дуже простих випадках вдається одразу отримати повністю роботоздатну програму, яка виконує потрібні дії без помилок і точно відповідає поставленому завданню. В більшості випадків процес налагодження програм є достатньо тривалим.

Процес налагодження програми складається з кількох етапів:

- тестування програми з метою виявлення дефектів;
- виявлення дефекту та відтворення умов, за яких він проявляється;

- аналіз дефекту (пошук причин виникнення);

- аналіз можливих шляхів виправлення дефекту;

- внесення змін до програми;

- перевірка та аналіз результатів виправлення.

Якщо результат перевірки позитивний, то на цьому процес виправлення дефекту закінчується, якщо ж ні, то повертаються до аналізу дефекту. Ці дії повторюють для всіх виявлених дефектів.

Найбільш складними та довготривалими є процеси виявлення дефекту та умов, за яких він проявляється, а також аналізу дефекту.

Для спрощення цих процесів використовують такі методи налагодження програм:

1. Запускання програми у налагоджувальному режимі (який реалізується за допомогою спеціального програмного або програмно-апаратного забезпечення) і аналіз ходу виконання програми (аналіз стану змінних, стеку, регістрів і логіки роботи програми).
2. Журналювання (логування) коду – виведення під час виконання критичних блоків коду у файл або на екран значень вхідних аргументів функцій та результатів їх виконання, проміжних станів змінних, стеку, регістрів та іншої інформації з метою аналізу даних.
3. Аналіз коду без виконання програми – пошук причин виникнення дефекту шляхом аналізу проблемного коду.
4. Аналіз поведінки системи або її частини, ізолювання проблеми шляхом спрощення сценарію виконання. Ізольоване тестування окремих блоків програми (в тому числі окремих функцій, бібліотек тощо) для виявлення проблемного коду.
5. Аналіз дампів пам’яті.
6. Аналіз супровідної документації.
7. Налагодження трансляцією коду – коли програма або її фрагмент переписується іншою мовою програмування, налагоджується, а потім налагоджений варіант переписується знову на цільову мову.

Контрольні запитання

1. Які види коментарів використовують в C++?
2. Яка різниця між локальними змінними в C та C++?
3. Які особливості опису прототипів функцій в C++?
4. Для чого використовують аргументи за замовчуванням?
5. Як здійснюється доступ до глобальних змінних прихованих локальними змінними з тим же ім’ям?
6. В яких випадках використовують модифікатори `const` і `volatile`?
7. В чому полягає різниця між передачею змінних за посиланням і за значенням?
8. Для чого використовують модифікатор `inline`?
9. За допомогою яких операторів здійснюють динамічний розподіл пам’яті в C++?
10. Що означає перевантаження функції? Для чого його використовують?
11. Які є обмеження на використання перевантажених функцій?
12. Яке призначення шаблону функції?
13. В яких випадках використовують перевантаження операторів?
14. Дайте визначення класу та об’єкту класу.
15. Перелічіть основні принципи об’єктно-орієнтованого програмування.
16. Що таке інкапсуляція?
17. Що таке успадкування?
18. Що таке поліморфізм?
19. Які є режими доступу до членів класу?
20. Яким чином здійснюється доступ до полів і методів класу?
21. Наведіть особливості статичних членів класу.
22. Для чого використовують «друзів класу»?
23. Які особливості перевантаження операторів класів?

24. Що таке конструктор? Яке його призначення?
25. Що таке деструктор?
26. Для чого використовують шаблони класів?
27. Дайте визначення потоку відповідно до концепції C++.
28. Які стандартні потоки введення-виведення використовують в C++?
29. Що таке маніпулятор?
30. Перелічіть маніпулятори, які визначені в мові C++.
31. Які потокові класи є в мові C++?

3. Алгоритми і структури даних

3.1. Поняття алгоритму. Властивості алгоритму

Алгоритм – набір інструкцій, що описує порядок дій, які необхідно виконати для переходу об’єкту із одного визначеного стану в інший визначений стан.

Алгоритм має задовольняти наступним вимогам:

1. Зрозумілість для виконавця: бути записаним способом зрозумілим для виконавця алгоритму.
2. Дискретність: описувати розв’язок задачі, як послідовність простих, або заздалегідь обумовлених кроків.
3. Визначеність: кожна інструкція алгоритму має бути чіткою й однозначною.
4. Результативність: за скінченну кількість кроків приводити до розв’язку задачі або до закінчення виконання алгоритму у зв’язку із неможливістю отримати розв’язок.
5. Масовість: бути придатним до розв’язку будь-якої задачі даного типу.

Виконавцем алгоритму може бути людина або технічний пристрій. Від формального виконавця алгоритму вимагається не розуміння суті алгоритму, а чітке виконання заданої послідовності дій.

3.2. Способи записування алгоритмів

При розробці алгоритму необхідно задати його ім’я, позначити початок і кінець, описати вхідні та вихідні дані, вказати інструкції (команди), які потрібно виконати.

Це можна зробити кількома способами:

1. За допомогою словарного опису. Наприклад алгоритм обчислення площі кругу можна записати наступним чином: радіус круга піднести до квадрату й помножити на число π, отриманий добуток помножити на число 2.

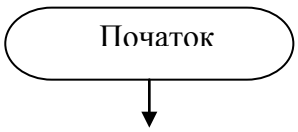
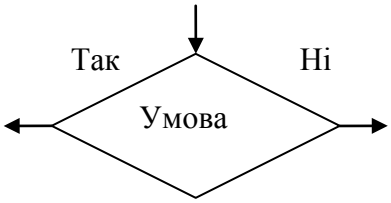
2. За допомогою алгоритмічної мови програмування. Наприклад написати процедуру мовою С.

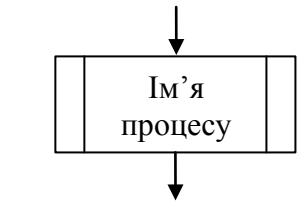
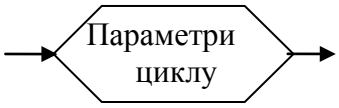
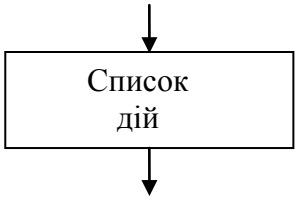
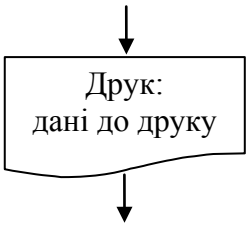
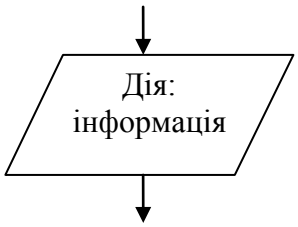
3. За допомогою псевдокоду. Псевдокод – це дещо середнє між звичайною мовою та алгоритмічною мовою програмування. Псевдокод складається з набору ключових слів, які мають певне значення і не можуть бути використані в інший спосіб. Прикладом може бути мова запису алгоритмів, яку використовують у шкільному курсі інформатики:

```
алгоритм максимум(X1,X2)
початок
    якщо X1 > X2      результат = X1
    інакше            результат = X2
кінець.
```

4. За допомогою операторних символів. Прикладом може бути запис формул у математиці.

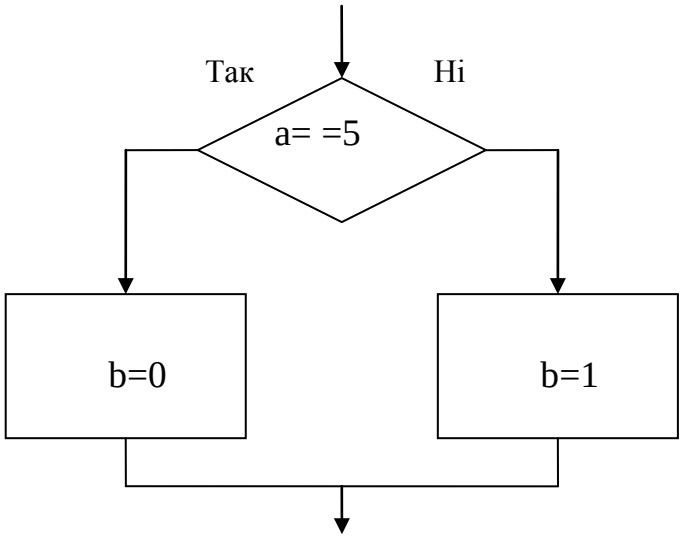
5. Графічним способом (у вигляді блок-схем). При цьому кожен етап обчислень відображується за допомогою певної геометричної фігури – блоку. Для цього розроблено спеціальну систему позначень. Розглянемо найбільш уживані з них:

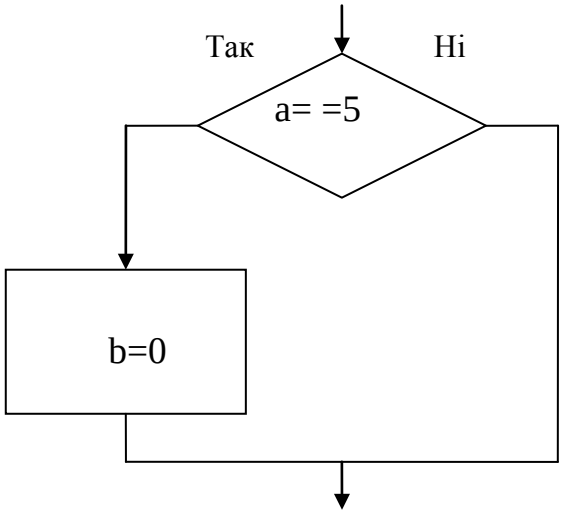
Блок	**Назва блоку**	**Призначення блоку**
	Початок	Даний блок позначає початок алгоритму. Під словом початок можна вказати назву алгоритму.
	Кінець	Цей блок позначає кінець алгоритму. Під словом кінець можна вказати назву алгоритму.
	Вибір	Блок позначає перевірку умови. Якщо умова істинна, то управління передається на гілку «Так», у протилежному випадку – на гілку «Ні».

Блок	Назва блоку	Призначення блоку
	Визначений процес	Даний блок позначає місце виклику бібліотечної або розробленої користувачем процедури (функції). У блоці вказують ім’я процедури або функції, яку викликають.
	Модифікація	Цей блок позначає місце початку циклу з параметром (`for`). У блоці вказують початкове й кінцеве значення параметру та умову закінчення циклу.
	Процес	Даний блок позначає виконання обчислень. Всередині цього блоку вказують дії, які необхідно виконати.
	Документ	Наведений блок позначає місце алгоритму в якому відбувається друк інформації на принтер. У блоці вказують яку саме інформацію треба відправити на друк.
	Відображення або зчитування інформації	Блок вказує місце алгоритму в якому відбувається зчитування або відображення інформації без указування конкретного пристрою. Зазвичай це консоль. Замість слова «Дія» вказують, яку саме дію (зчитування чи відображення інформації) потрібно виконати. Нижче вказують над якою інформацією виконується дія (імена змінних тощо).

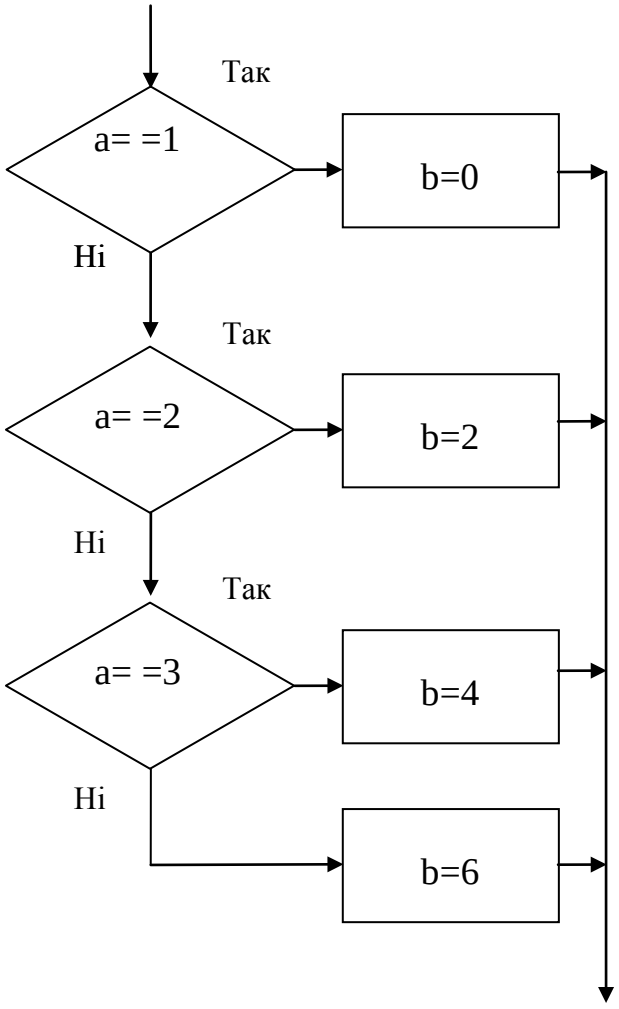
Наведемо приклади їх використання:

Фрагмент блок-схеми	**Аналог мовою C++**
Умовний оператор	

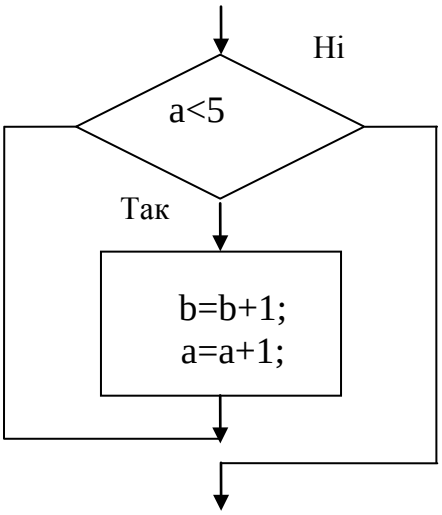
	`if (a==5)`
	`  b=0;`
	`else`
	`  b=1;`

Скорочена форма умовного оператора	
	`if (a==5)` `  b=0;`

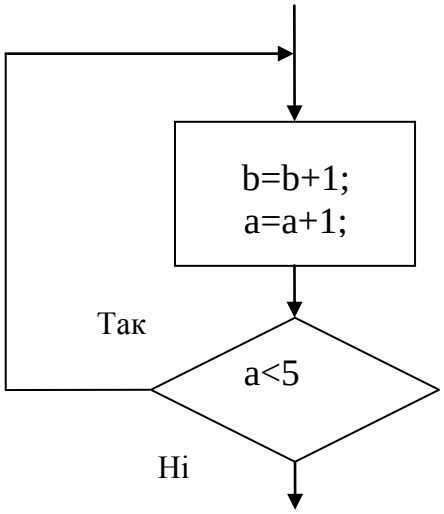
Фрагмент блок-схеми	Аналог мовою C++
Вибір	

	`if (a==1)`
	`  b=0;`
	`else if (a==2)`
	`  b=2;`
	`else if (a==3)`
	`  b=4;`
	`else`
	`  b=6;`
	
	або
	
	`  switch(a)`
	`  {`
	`  case 1:`
	`  b=0;`
	`  break;`
	`  case 2:`
	`  b=2;`
	`  break;`
	`  case 3:`
	`  b=4;`
	`  break;`
	`  default:`
	`  b=6;`
	`}`

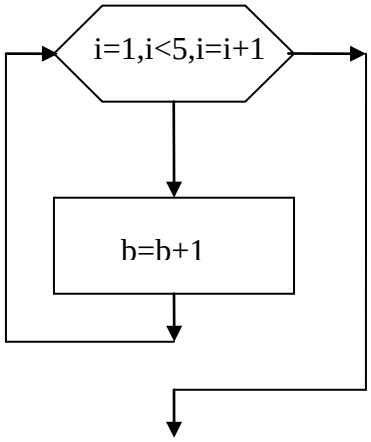
Фрагмент блок-схеми	**Аналог мовою C++**
Цикл з передумовою	

	`while (a<5)`
	`{`
	`  b=b+1;`
	`  a=a+1;`
	`}`

Цикл з післяумовою	

	`do`
	`{`
	`  b=b+1;`
	`  a=a+1;`
	`} while (a<5)`

Фрагмент блок-схеми	Аналог мовою C++
Цикл з параметром	

	`for(i=1;i<5;i++)`
	`{`
	`  b=b+1;`
	`}`

Будь-який найскладніший алгоритм можна представити як сукупність простих, що дозволяє розв'язувати складні задачі шляхом розбиття їх на кілька простих.

3.3. Статичні та динамічні масиви

Основною особливістю масивів є те, що на етапі виділення пам'яті для збереження його елементів відома їх точна кількість. Основними операціями при роботі з масивами є:

- виділення пам'яті для збереження елементів масиву;
- вивільнення пам'яті;
- сортування елементів масиву;
- пошук елемента масиву, який відповідає заданому критерію пошуку.

Виділення пам'яті під масив може бути ***статичним***, коли кількість елементів масиву відома на етапі розробки програми, або ***динамічним***, коли кількість елементів масиву стає відомою лише під час виконання програми. У разі статичного виділення пам'яті її вивільнення відбувається автоматично при закінченні роботи програми, і програміст не може контролювати цей

процес. У разі динамічного виділенні пам’яті її вивільнення також відбувається автоматично при закінченні роботи програми, але програміст може за необхідності вивільнити пам’ять у процесі виконання програми. Це робить програми більш гнучкими, тому в сучасному програмуванні все частіше використовують динамічне виділення пам’яті при роботі з масивами.

3.4. Методи сортування масивів

При роботі з масивами найчастіше доводиться виконувати операції сортування та пошуку даних. Від ефективності реалізації цих операцій часто залежить ефективність всієї програми, тому розглянемо їх докладніше.

Сортування – це процес перегрупування даних у заданому порядку. Основна мета сортування – полегшити пошук потрібної інформації у заданій послідовності даних [8, 9].

Методи сортування поділяють на внутрішні (дані розміщуються в оперативній пам'яті) та зовнішні (дані розміщуються в файлах). Для внутрішнього сортування використовують наступні методи: за допомогою включення, за допомогою вибору, за допомогою обміну тощо.

Розглянемо вказані внутрішні методи. Для визначеності будемо вважати, що необхідно виконати сортування у порядку зростання елементів масиву, і всі приклади, які будуть розглянуті, відповідатимуть саме такому порядку сортування. Читачеві ж рекомендуємо для тренування модифікувати наведені програми для сортування у зворотному напрямку.

3.4.1. Сортування за допомогою включення

При сортуванні за допомогою ***включення*** елементи умовно поділяють на вже відсортовану послідовність $a_1 \ldots a_{i-1}$ і вихідну (початкову) послідовність $a_i \ldots a_n$. На кожному кроці, починаючи з $i = 2$ та збільшуючи кожен раз i на одиницю, із вихідної послідовності видобувають i-тий елемент і перекладають його в потрібне місце уже відсортованої послідовності. У процесі пошуку потрібного місця чергуються операція порівняння даного

елемента з елементами послідовності та операція переміщення по послідовності, тобто вибраний елемент a_i порівнюється з черговим елементом a_j, а тоді він або вставляється на місце елемента a_j (якщо виконано критерій сортування), тоді відповідно a_j зміщується на одну позицію вправо, або порівнюється з наступним елементом a_{j-1} (якщо критерій сортування не виконано), при цьому a_j знову ж таки зміщується на одну позицію вправо. Нижче наведено приклад сортування методом включення масиву цілих чисел, у якому, для демонстрації, на екран виводяться вихідний масив та результуючий масив:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;
//Шаблон функції сортування елементів масиву методом
//включення
template <class T> void vstavka(T* massive,int size);

int main()
{
//Оголошення змінних, необхідних для реалізації
//алгоритму
int* mas;
int n,i;
//Очищення екрану
system("cls");
//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
```

```
cin >> n;
cout << "creating massive of random numbers:"<<endl;

//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];

//Підключення генератору випадкових чисел
srand(time(NULL));

//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
   //Формування елементу масиву випадковим чином
   mas[i]=rand()%100-50;
   //Відображення на екрані сформованого елементу
   cout << mas[i]<<"   ";
}

//Переведення курсору на наступну строку екрану
cout << endl;

//Сортування елементів масиву методом включення
vstavka(mas, n);

//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
   cout << mas[i] << "   ";
}
```

```
//Очищення пам’яті, виділеної під масив
delete mas;
return 0;
}

//Реалізація шаблону функції сортування
template <class T> void vstavka (T* massive,int size)
{
 T tmp;
 int i,j;
for(i=1; i<size; i++)
{
 tmp = massive[i];
 j=i;
 while(tmp < massive[j-1])
 {
  massive[j] = massive[j-1];
  j--;
  if(j==0) break;
 }
 massive[j]=tmp;
}
return;
}
```

3.4.2. Сортування за допомогою прямого вибору

Сортування за допомогою прямого вибору ґрунтується на нижченаведених операціях:

1. Обирають елемент з найменшим значенням.

2. Обмінюють його місцями з першим елементом.

3. Операції 1 і 2 повторюють з елементами від 2-го до *n*-го, потім від 3-го до *n*-го і так далі, поки не буде відсортовано весь масив.

Наведемо приклад сортування методом прямого вибору масиву цілих чисел. Як і у попередньому прикладі для демонстрації на екран виводяться вихідний масив та результуючий масив.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

//Шаблон функції сортування масиву методом прямого
//вибору
template <class T> void vibir(T* massive,int size);
int main()
{
//Оголошення змінних, необхідних для реалізації
алгоритму
int* mas;
int n,i;

//Очищення екрану
system ("cls");

//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
cin >> n;
cout << "creating massive of random numbers:"<<endl;
```

```
//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];

//Підключення генератору випадкових чисел
srand(time(NULL));

//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
   //Формування елементу масиву випадковим чином
    mas[i]=rand()%100-50;
   //Відображення на екрані сформованого елементу
   cout << mas[i]<<"   ";
}

//Переведення курсору на наступну строку екрану
cout << endl;

//Сортування елементів масиву методом прямого вибору
vibir(mas,n);

//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
   cout << mas[i] << "   ";
}

//Очищення пам’яті, виділеної під масив
delete mas;
```

```
return 0;
}

//Реалізація шаблону функції сортування масиву методом
//прямого вибору
template <class T> void vibir(T* massive,int size)
{
   T tmp;
   int i,j,e;
   tmp=massive[0];
   for(i=0;i<size;i++)
   {
      e=i;
      tmp=massive[i];
      for(j=i;j<size;j++)
      {
          if(massive[j]<tmp)
          {
             tmp=massive[j];
             e=j;
          }
      }
      if(e!=i)
      {
         massive[e]=massive[i];
         massive[i]=tmp;
      }
   }
return;
}
```

3.4.3. Сортування за допомогою обміну

Сортування за допомогою обміну ґрунтується на процесі порівняння і, за необхідності, обміну місцями двох сусідніх елементів масиву. Ці операції повторюються доки не буде упорядковано весь масив. Треба зазначити, що після першого проходу по всьому масиву максимальний елемент переміщується у крайнє праве положення, і на наступному етапі немає сенсу перевіряти весь масив. Тому на практиці при першому проході перевіряють елементи з номерами від 1-го до *n*-го (останнього), на другому від 1-го до (*n* – 1)-го і т.д.

Наведемо приклад сортування методом обміну масиву цілих чисел. Як і в попередньому прикладі для демонстрації на екран виводяться вихідний масив, та результуючий масив.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

//Шаблон функції сортування елементів масиву методом
//обміну
template <class T> void obm_sort(T* massive,int size);

int main()
{
//Оголошення змінних необхідних для реалізації
//алгоритму
int* mas;
int n,i;
```

```
//Очищення екрану
system ("cls");

//Запит на введення розміру масиву
cout << "Input n" << endl;
cout << "n=";
cin >> n;
cout << "creating massive of random numbers:"<<endl;

//Динамічне виділення пам’яті під елементи масиву
mas=new int[n];

//Підключення генератору випадкових чисел
srand(time(NULL));

//Формування масиву випадкових чисел
for (i=0;i<n;i++)
{
   //Формування елементу масиву випадковим чином
   mas[i]=rand()%100-50;
   //Відображення на екрані сформованого елементу
   cout << mas[i]<<"   ";
}

//Переведення курсору на наступну строку екрану
cout << endl;

//Сортування елементів масиву методом обміну
obm_sort(mas,n);
```

```
//Відображення на екрані відсортованого масиву
cout << "massiv after sorting:" << endl;
for(i=0;i<n;i++)
{
   cout << mas[i] << "   ";
}

//Очищення пам’яті, виділеної під масив
delete mas;
return 0;
}
//Реалізація шаблону функції сортування елементів
//масиву методом обміну
template <class T> void obm_sort(T* massive,int size)
{
int i,j,d;
T tmp;
d=size;
for(i=0;i<size-1;i++)
   {
      for(j=0;j<d-1;j++)
      {
         if(massive[j]>massive[j+1])
         {
            tmp=massive[j];
            massive[j]=massive[j+1];
            massive[j+1]=tmp;
         }
      }
   d--;
```

```
   }
return;
}
```

Продемонстровані методи можуть бути модифіковані для збільшення їх ефективності, але викладення цього матеріалу виходить за рамки даного курсу. Студентам же пропонується самостійно вивчити метод Шелла (модифікований метод сортування за допомогою включення) і метод швидкого сортування [9] та написати параметризовані функції сортування для цих методів, а також для всіх методів сортування розглянутих вище.

3.5. Методи пошуку у масивах

Пошук – це процес знаходження серед елементів даного типу елемента із заданими властивостями. Задане значення критерію пошуку називається ключем пошуку. Ним може бути умова рівності елемента певному значенню або інша умова. При подальшому розгляді методів пошуку будемо вважати, що кількість елементів у масиві, в якому провадиться пошук, відома.

3.5.1. Прямий лінійний пошук

Найпростішим, але не оптимальним, методом пошуку є прямий лінійний пошук. Цей метод використовують коли немає додаткової інформації про групу елементів серед яких провадиться пошук.

Метод полягає у послідовній перевірці всіх елементів на відповідність ключу пошуку. Умовою закінчення пошуку може бути або факт знаходження даного елемента, або той факт, що дану сукупність елементів перевірено повністю і не знайдено елементів, які відповідають критерію пошуку. Розглянемо приклад:

```
#include <iostream>
#include <stdlib.h>
```

```
#include <time.h>
using namespace std;

//Опис параметризованої функції
template <class T> int search (T* mas, T search_key,
int size);

int main ()
{
//Опис змінних
int n, i, key;
int* massive;
float* massive1;
float key1;

//Очищення екрану
system ("cls");

//Запит на введення розміру масивів
cout << "Input n=" ;
cin >> n;

//Динамічне виділення пам’яті під масив цілих чисел
massive= new int[n];

//Динамічне виділення пам’яті під масив дійсних чисел
massive1= new float[n];

//Активація генератора випадкових чисел
srand(time(NULL));
```

```
//Заповнення масивів випадковими числами
for(i=0;i<n;i++)
{
 massive[i]=rand()%50-25;
 massive1[i]=massive[i]/2.0;
}

//Відображення на екрані масиву цілих чисел
cout << "Massive of integer numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout.width(7);
 cout << massive[i];
}
cout <<endl;

//Запит на введення ключа для пошуку у масиві цілих
//чисел
cout << "Input integer key=";
cin >> key;

//Пошук у масиві цілих чисел та виведення результату на
//екран
search(massive, key, n);

//Відображення на екрані масиву дійсних чисел
cout << "Massive of float numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout << massive1[i] << "   ";
```

```
}
cout <<endl;

//Запит на введення ключа для пошуку у масиві дійсних
//чисел
cout << "Input float key1=";
cin >> key1;

//Пошук у масиві дійсних чисел та виведення результату
//на екран
search(massive1, key1, n);

//Очищення пам’яті, виділеної під масиви
delete massive;
delete massive1;
return 0;
}

//Параметризована функція прямого пошуку
template <class T> int search(T* mas, T search_key,
                                              int size)
{
 int index=0;
 for (index=0;index<size;index++)
 {
  if(search_key==mas[index])
  {
   cout << "Found element with number n= " << index+1
<< endl;
   return index;
```

```
  }
 }
  cout << "Element not found"<<endl;
 return -1;
}
```

3.5.2. Бінарний пошук

Прямий пошук вимагає великих затрат машинного часу. Без додаткової інформації про задану сукупність елементів пошук прискорити неможливо. Проте його можна зробити значно ефективнішим, якщо відомо, що задана послідовність елементів є впорядкованою за критерієм пошуку. Прикладом такої впорядкованої послідовності може бути телефонний довідник, всі записи якого впорядковано відповідно до алфавіту.

Основою пошуку у такій послідовності є вибір деякого випадкового елемента і порівняння його з критерієм пошуку. При цьому можливі три випадки:

— значення обраного елемента відповідає критерію пошуку. Тоді шуканий елемент знайдено і пошук можна завершити;

— значення обраного елемента більше за значення ключа пошуку. Тоді треба продовжити пошук у тій частині сукупності де значення менші за значення обраного елемента;

— значення обраного елемента менше за значення ключа пошуку. Тоді треба продовжити пошук у тій частині сукупності де значення більші за значення обраного елемента.

За такої організації пошуку критерієм зупинки може бути або факт знаходження даного елемента, або той факт, що дану сукупність елементів перевірено повністю і не знайдено елементів, які відповідають критерію пошуку. Найпростішим способом реалізації такого алгоритму є метод бінарного пошуку (метод поділу навпіл). За такого методу розглядувану послідовність ділять навпіл і порівнюють критерій пошуку з центральним

елементом послідовності. Якщо критерій співпадає, то елемент знайдено. Якщо значення елементу менше за заданий критерій, то ділять навпіл ту частину послідовності, де значення елементів більші за значення обраного елемента. Якщо ж воно більше, то ділять ту половину, де значення елементів менші за значення обраного елемента. Ці дії виконують доки не буде знайдено потрібний елемент або у досліджуваній частині сукупності не залишиться лише один елемент. Оскільки при цьому кожен раз кількість досліджуваних елементів зменшується вдвічі, то швидкість пошуку значно зростає порівняно з лінійним пошуком.

Розглянемо приклад:

```
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T> int b_search(T* mas,T search_key,int
size);
template <class T> void sort(T* mas, int size);

int main ()
{
//Опис змінних
int n,i,key;
int* massive;
float* massive1;
float key1;

//Очищення екрану
system ("cls");
```

```
//Запит на введення розміру масивів
cout << "Input n=" ;
cin >> n;

//Виділення пам’яті під масиви чисел
massive= new int[n];
massive1= new float[n];

//Активація генератора випадкових чисел
srand(time(NULL));

//Формування масивів випадкових чисел
for(i=0;i<n;i++)
{
 massive[i]=rand()%50-25;
 massive1[i]=massive[i]/2.0;
}

//сортування масиву цілих чисел
sort(massive,n);

//Відображення відсортованого масиву на екрані
cout << "Massive of integer numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout.width(7);
 cout << massive[i];
}
```

```
//Запит на введення ключа для пошуку у масиві цілих
//чисел
cout <<endl;
cout << "Input integer key=";
cin >> key;

//Пошук у масиві цілих чисел та відображення на екрані
//результату пошуку
b_search(massive, key, n);

//Сортування масиву дійсних чисел
sort(massive1,n);

//Відображення відсортованого масиву на екрані
cout << "Massive of float numbers:"<<endl;
for(i=0;i<n;i++)
{
 cout << massive1[i] << "   ";
}
cout << endl;

//Запит на введення ключа для пошуку у масиві дійсних
//чисел
cout << "Input float key1=";
cin >> key1;

//Бінарний пошук та відображення на екрані результату
//пошуку
b_search(massive1, key1, n);
```

```
//Вивільнення пам’яті виділеної під масиви
delete massive;
delete massive1;
return 0;
}

//Параметризована функція бінарного пошуку
template <class T> int b_search(T* mas, T search_key,
                                                int size)
{
 int low, high, mid;
 low=0;
 high=size-1;
 while(low<=high)
 {
  mid=(low+high)/2;
  if(search_key<mas[mid])
  {
   high=mid-1;
  }
  else
  {
   if(search_key>mas[mid])
   {
    low=mid+1;
   }
   else
   {
    cout << "Found element with number n= " << mid+1 <<
endl;
```

```
    return mid;
   }
  }
 }
  cout << "Element not found" << endl;
 return -1;
}

//Параметризована функція сортування методом
//перестановок
template <class T> void sort(T* mas, int size)
{
 int i,j,d,tmp;
 d=size;
for(i=0;i<size-1;i++)
{
 for(j=0;j<d-1;j++)
 {
  if(mas[j]>mas[j+1])
  {
   tmp=mas[j];
   mas[j]=mas[j+1];
   mas[j+1]=tmp;
  }
 }
 d--;
}
return;
}
```

3.6. Списки

Список – це сукупність елементів, кількість яких може змінюватись під час виконання програми. При цьому пам’ять для розміщення нового елемента виділяється і вивільняється динамічно за необхідністю. Для реалізації такого типу даних необхідно використовувати покажчики.

Списки можуть реалізовуватись за допомогою одинарних або подвійних зв’язків [8,9]. У списку з одинарними зв’язками кожен елемент містить покажчик на наступний елемент списку. У списку із подвійними зв’язками кожен елемент містить покажчики на попередній і наступний елементи списку.

Натепер найчастіше використовують списки із подвійними зв’язками. Це пов’язано з тим, що список із подвійними зв’язками можна читати в обох напрямках, такий список легше відновлювати при пошкодженні, крім того при використанні таких списків операції оброблення інформації реалізуються простіше. Виходячи із сказаного, розглянемо лише списки із подвійними зв’язками.

Основними операціями при роботі зі списками є:

— додавання елементу до списку;

— вилучення елементу із списку;

— сортування елементів списку;

— пошук елемента списку, який відповідає заданому критерію пошуку.

Розглянемо приклад:

```
#include <stdio.h>
#include <iostream>
#include <conio.h>
#include <stdlib.h>
using namespace std;
```

```
//Шаблон класу список
template <class T> class my_list
{
 private:
//Покажчик на наступний елемент списку
  my_list *next;
//Покажчик на попередній елемент списку
  my_list *prev;
//Поле даних елемента списку
  T info;
//Поточна кількість елементів, занесених до списку
  int list_size;
 public:
//Функція визначення адреси наступного елемента списку
  my_list<T> * get_next() {return next;}
//Функція визначення адреси попереднього елемента
//списку
  my_list<T> * get_prev() {return prev;}
//Функція запису адреси наступного елемента списку
  my_list<T> * set_next(my_list<T> *a) {return next=a;}
//Функція запису адреси попереднього елемента списку
  my_list<T> * set_prev(my_list<T> *a) {return prev=a;}
//Функція додавання елемента до списку
  void add_elem(my_list<T> *first,T inf);
//Функція вилучення елемента із списку
  void del_elem(my_list<T> *first);
//Функція сортування елементів списку за зростанням
  void sort_elements(my_list<T> *first);
```

```
//Функція зчитування даних, збережених в елементі
//списку
  T get_info() {return info;}
//Функція запису даних до елемента списку
  void set_info(T c) {info=c;return;}
//Функція запису розміру списку
  void set_size(int c) {list_size=c;return;}
//Функція зчитування розміру списку
  int get_size() {return list_size;}
//Функція відображення інформаційних полів списку на
//екрані
  void print_list(my_list<T> *first);
//Функція пошуку елемента із заданим значенням у списку
  int b_search(my_list<T> *first,T key);
};

//Реалізація функції видалення елемента із списку
template <class T> void my_list<T>::del_elem(my_list<T>
*first)
        {
         my_list<T> *tmp;
         tmp=first->next;
         first->list_size--;
         first->next=first->next->next;
         delete tmp;
         return;
        }
//Реалізація функції відображення списку на екрані
template <class T> void my_list<T>::print_list
                                (my_list<T> *first)
```

```
        {
          int i;
          my_list<T> *tmp;
          tmp=first;
          for(i=0;i< first->get_size();i++)
           {
            cout << tmp->get_info() << endl;
            tmp=tmp->get_next();
           }
        }

//Реалізація функції сортування елементів списку
template <class T> void
my_list<T>::sort_elements(my_list<T> *first)
        {
         my_list<T> *tmp_ptr;
         T tmp_elem_value;
         int i,j,d,size;
         tmp_ptr=first;
         size=first->list_size;
         d=size;
         for(i=0;i<size-1;i++)
          {
           for(j=0;j<d-1;j++)
           {
            tmp_elem_value=tmp_ptr->info;
            if(tmp_ptr->info>tmp_ptr->next->info)
             {
              tmp_elem_value=tmp_ptr->info;
              tmp_ptr->info=tmp_ptr->next->info;
```

```
            tmp_ptr->next->info=tmp_elem_value;
           }
          tmp_ptr=tmp_ptr->next;
         }
         d--;
         tmp_ptr=first;
        }
       return;
      }

//Реалізація функції додавання елементу до списку
template <class T> void my_list<T>::add_elem(my_list<T>
*first,T inf)
      {
      //тимчасовий покажчик для елемента
         my_list<T> *tmp;
      //виділення пам’яті під елемент
         tmp=new my_list<T>;
      //запис інформації до елемента
         tmp->set_info(inf);
      //додавання елемента до початку списку
         tmp->next=first->next;
       tmp->prev=first;
       first->prev=NULL;
       first->next=tmp;
       first->list_size++;
       return;
      }
```

```
//Реалізація функції пошуку значення у списку (бінарний
//пошук)
 template <class T> int my_list<T>::b_search(my_list<T>
*first,T key)
        {
         my_list<T> *tmp,*tmp_beg;
         int beg,mid,end,i,num;
       //перевірка першого елемента
         if(first->info==key) return 1;
         tmp_beg=tmp=first;
       //налаштування початкових параметрів списку
         num=1;
         beg=1;
         end=first->list_size;
       //цикл пошуку
         while(beg<end)
         {
       //знаходження середини списку
          mid=(end+beg)/2;
          for(i=beg;i<mid;i++)
          {
       //перехід до центрального елемента
           tmp=tmp->next;
          }
       //закінчити пошук, якщо шуканий елемент знайдено
          if(tmp->info==key)
          {
           num=mid;
           return num;
          }
```

```
      //вибір потрібної для пошуку частини
          if(tmp->info>key)
          {
           end=mid;
           tmp=tmp_beg;
           num=beg;
          }
          else
          {
           tmp_beg=tmp;
           num=mid;
           beg=mid;
          }
          if (mid == (beg+end)/2) break;
         }
         tmp=first->next;
     for(i=2;i<first->list_size;i++)
         {
          tmp=tmp->next;
         }
      //повернути значення
         if(tmp->info==key) return first->list_size;
      //якщо не знайдено, повернути код помилки
         return -1;
        }

//Головна функція
int main(void)
{
 int i;
```

```
 my_list <int> *first,*last,*tmp;

//Очищення екрану
 system("cls");

//Створення списку
 first=new my_list<int>;
 last=new my_list<int>;
 tmp=new my_list<int>;
 first->set_next(NULL);
 first->set_prev(NULL);
 first->set_info(0);
 first->set_size(1);
 last->set_next(NULL);
 last->set_prev(NULL);
 tmp=first;

//Додавання чотирьох елементів до списку
 for(i=1;i<5;i++)
 {
  tmp->add_elem(first,i);
 }

//Відображення списку
 first->print_list(first);

//Сортування списку
 first->sort_elements(first);
```

```
//Відображення списку
 first->print_list(first);

//Пошук елементу із значенням «3»
 cout << endl << first->b_search(first,3)<<endl<<endl;

//Видалення всіх елементів списку, які знаходяться між
//першим і останнім
 for(i=0;i<3;i++)
 {
  first->del_elem(first);
 }

//Відображення списку на екрані
 first->print_list(first);
return 0;
}
```

З прикладу видно, що навіть елементарні операції із списками потребують досить високої кваліфікації програміста, але вони дають змогу оптимально використовувати пам’ять.

У даному прикладі продемонстровано сортування списку методом обміну. Для простоти в прикладі було опущено ряд необхідних дій, таких як контролювання виходу за межі списку, оброблення помилок тощо.

Для кращого засвоєння матеріалу автори рекомендують самостійно реалізувати сортування списку методом включення та методом прямого вибору.

Для пошуку елементів у списку, які відповідають заданому критерію пошуку, використовують ті ж методи, що й для масивів. У наведеному прикладі розглянуто бінарний пошук.

3.7. Стеки та деки

3.7.1. Стеки

Стек – одна з найбільш корисних і найчастіше використовуваних структур даних [8,9]. Його використовують для зберігання змінних при викликанні процедур або при виконанні переривань, для збереження даних про стан екрану при розробці інтерфейсу з користувачем на основі багаторівневого меню тощо. Доступ до елементів стеку здійснюється за принципом LIFO («Last In, First Out» – «останнім зайшов, першим вийшов»).

Концептуально цей тип даних дозволяє виконувати записування та зчитування даних лише в одному елементі – вершині стеку. Елементи зчитуються у зворотному порядку – першим зчитується елемент, який був записаний до стеку останнім. Довільний доступ до елементів стеку неможливий.

Стек можна реалізувати на основі масиву або списку. Реалізація на основі масиву більш проста, але має ряд недоліків: обмежену кількість елементів, неефективне використання пам’яті тощо. Реалізація на основі списку більш складна, але позбавлена вказаних недоліків, тому в практичному програмуванні частіше реалізують стеки на основі списків. Саме тому розглянемо лише стеки на основі списків.

При роботі зі стеком використовують такі операції:

— додавання елементу даних до вершини стеку (метод `push`);

— зчитування та вилучення елементу даних із вершини стеку (метод `pop`);

— визначення наявності даних у стеку.

Розглянемо ці операції на прикладі простого стеку на основі списку із подвійними зв’язками.

Приклад:

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

```
#include <iostream>
#include <conio.h>
#include <time.h>
using namespace std;

//Шаблон класу стек
template <class T> class user_stack
{
 private:
//Розмір стеку
    int stack_size;
//Елемент даних стеку
    T data;
//Покажчик на попередній елемент стеку
    user_stack *prev;
//Покажчик на наступний елемент стеку
    user_stack *next;

public:
//Конструктор стеку
 user_stack();
//Функція, яка перевіряє наявність даних в стеку
 int empty() {if (stack_size>0) return 0;
              else return -1;};
//Функція, яка зчитує елемент даних стеку (потрібна
//лише для налагоджування програми)
 T get_data(){return data;};
//Функція, яка визначає адресу попереднього елементу
//стеку
 user_stack<T> * get_prev(){return prev;};
```

```
//Функція, яка додає елемент до стеку (метод push)
 user_stack<T> * push(T var);
//Функція, яка видаляє елемент стеку (метод pop)
 T pop();
};

//Реалізація конструктора
template <class T> user_stack<T>::user_stack()
 {
  stack_size=0;
  data=NULL;
  prev=NULL;
  next=NULL;
 }

//Реалізація методу push
template <class T> user_stack<T> *
                              user_stack<T>::push(T var)
 {
  this->next=new user_stack<T>;
  this->next->prev=this;
  this->next->data=var;
  this->next->stack_size=this->stack_size+1;
  return this->next;
 }

//Реалізація методу pop
template <class T> T user_stack<T>::pop()
 {T dat;
  dat=this->next->data;
```

```
  delete this->next;
  this->next=NULL;
  return dat; }

int main(void)
{   user_stack<int> *my_stack;
     int my_data;
//Створення стеку
     my_stack=new user_stack<int>;
//Очищення екрану
     system("cls");
//Активація генератора випадкових чисел
     srand(time(NULL));
//Присвоювання змінній випадкового значення
     my_data=rand()%10+5;
//Відображення значення змінної на екрані
     cout << "data="<<my_data << endl;
//Записування значення змінної в стек
     my_stack=my_stack->push(my_data);
//Відображення на екрані даних, записаних у стек
     cout << "In stack stored - " << my_stack->
                                    get_data() << endl;
//Змінювання значення змінної
     my_data+=20;
//Відображення нового значення змінної
     cout << "Changing data. Now  data =" << my_data <<
                                                  endl;
//Відновлення старого значення змінної із стеку
     my_stack=my_stack->get_prev();
     my_data=my_stack->pop();
```

```
//Відображення відновленого значення змінної на екрані
     cout << "After popping data="<<my_data << endl;
  return 0;
}
```

3.7.2. Деки

Дек (deque) – це впорядкований набір елементів, у якому додавання нових елементів та вилучення наявних дозволено з будь-якого кінця набору.

Дек можна реалізувати на основі масиву або списку із подвійними зв’язками.

При роботі із деком використовують такі операції:

— додавання елементу даних до вершини деку;

— зчитування та вилучення елементу даних із вершини деку;

— додавання елементу даних до кінця деку;

— зчитування та вилучення елементу даних із кінця деку;

— визначення наявності даних в деку.

Дані операції реалізують аналогічно до операцій стеку, з урахуванням можливості доступу до елементів даних з обох кінців набору. Автори пропонують самостійно написати функції для цих операцій на основі наведених вище функцій для стеку.

3.8. Черги

Черга – це лінійний список, доступ до елементів якого здійснюється за принципом FIFO («First In, First Out» – «першим зайшов, першим вийшов»). Першим із черги видаляється елемент, який був першим записаний до неї, потім – елемент, який був записаний другим, і так до кінця черги. Довільний доступ до елементів черги неможливий.

Черги широко використовують на практиці, наприклад, при моделюванні процесів реального часу, для диспетчеризації завдань операційної системи, для буферизації операцій введення-виведення тощо.

При роботі із чергою використовують такі операції:

— додавання елементу даних у кінець черги;

— зчитування та вилучення елементу даних з початку черги.

Розглянемо приклад, який демонструє роботу з чергами:

```
#include <stdlib.h>
#include <iostream>
using namespace std;

//Шаблон класу черга
template <class T> class queue
{
 private:
//Покажчик на динамічний масив з даними
    T *q;
//Номер позиції, з якої будуть зчитуватись дані
    int read_loc;
//Номер позиції, у яку будуть записуватись дані
    int write_loc;
//Розмір черги - максимальна кількість елементів,
//які можуть бути розміщені у черзі
    int length;

 public:
//Функція-конструктор, яка створює чергу заданого
//розміру
    queue(int size);
```

```
//Функція-деструктор, яка знищує чергу
    ~queue() {delete [] q;};
//Функція, яка записує елемент даних до черги
    void write_data(T data);
//Функція, яка зчитує елемент даних з черги
    T read_data();
};

//Реалізація функції-конструктора
template <class T> queue<T>::queue (int size)
{
//Виділення пам’яті для черги
 q=new T[size];

//Якщо вільної пам’яті немає, то вивести на екран
//повідомлення про помилку
 if(!q)
 {
  cout << "Error! Can't create turn." << endl;
  exit(1);
 }

//Встановлення розміру черги
 length=size;

//Переведення індексів зчитування та запису
//у початок черги
 read_loc=write_loc=0;
}
```

```
//Реалізація функції, яка записує елемент до черги
template <class T> void queue<T>::write_data(T data)
{
//Перевірка наявності вільного місця у черзі та друк
//повідомлення про помилку за його відсутності
 if (write_loc==length)
 {
  cout << "Error! Turn is full." << endl;
  return;
 }

//Запис елементу у чергу за наявності вільного місця
 q[write_loc++]=data;
}

//Реалізація функції, яка зчитує елемент даних з черги
template <class T> T queue<T>::read_data()
{
//Перевірка наявності незчитаних елементів у черзі та
//друк повідомлення про помилку у разі їх відсутності
 if (write_loc==read_loc)
 {
  if(read_loc==length){ read_loc=write_loc=0;}
  cout << "Error! Turn is empty." << endl;
  return NULL;
 }

//Повернення зчитаних даних як результату функції
 return q[read_loc++];
}
```

```
//Головна функція
int main()
{
//Створення черги цілих чисел величиною 5 елементів
 queue<int> a(5);
 int turn_data;

//Очищуємо екран
 system("cls");

//Записуємо 5 елементів даних до черги
 a.write_data(1);
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);

//Записуємо 6-ий елемент даних до заповненої черги.
//На екрані з’явиться повідомлення про помилку
 a.write_data(6);

//Зчитуємо елементи даних з черги, поки не вичерпаємо
//їх. Після того, як черга стане пустою, при наступному
//зчитуванні з’явиться повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
 return 0;
}
```

Як видно з прикладу, черга просто реалізується на основі динамічного масиву. Така черга називається простою (лінійною) чергою. Вона має деякі недоліки, а саме: фіксований розмір, який не можна змінити в процесі роботи, можливість втрати даних у випадку переповнення черги, оскільки не можна продовжувати запис даних до заповненої черги, поки не будуть зчитані всі елементи. Лише після того, як буде зчитаний останній елемент черги, до неї можна знов записувати дані, починаючи з першого елементу. Цього можна уникнути, створюючи, у разі необхідності, нові черги в динамічному режимі. Але при такому підході неефективно використовується пам’ять. Можна розробити безрозмірну чергу на основі списку. Однак, це складне завдання, розгляд якого виходить за рамки даного посібника. Компромісним варіантом, що оптимізує використання пам’яті і, при цьому, просто реалізується є використання циклічних черг.

Циклічні (кільцеві) черги мають таку ж структуру, як і прості, але з однією відмінністю, яка полягає в тому, що після заповнення черги записування знов починається з першого елементу черги за умови, що цей елемент уже зчитано. Далі записування продовжується, як і у звичайній черзі за умови, що відповідні елементи черги уже прочитані. При правильному виборі розміру черги, з урахуванням швидкості записування й зчитування даних, можна уникнути проблем з її переповненням.

Розглянемо приклад практичної реалізації циклічної черги на основі динамічного масиву:

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
```

```
//Шаблон класу циклічна черга
template <class T> class queue
{
 private:
//Покажчик на адресу, за якою починається черга
    T *q;
//Змінна, що вказує на елемент, який буде зчитано при
//наступному зчитуванні
    int read_loc;
//Змінна, яка вказує куди буде записано елемент при
//наступному записуванні
    int write_loc;
//Розмір черги - максимальна кількість елементів, що
//можуть розміститись в черзі одночасно
    int length;
//Кількість елементів, розміщених в черзі
    int elem_nums;

 public:
//Конструктор черги
    queue(int size);
//Деструктор черги
    ~queue() {delete [] q;};
//Функція, яка записує елемент до черги
    void write_data(T data);
//Функція, яка зчитує елемент із черги
    T read_data();
};
```

```
//Реалізація конструктора
template <class T> queue<T>::queue (int size)
{
//Виділення пам’яті для черги
 q=new T[size];

//Повідомлення про помилку в разі невдачі
 if(!q)
 {
  cout << "Error! Can't create turn." << endl;
  exit(1);
 }

//Активація нулем лічильника, тобто змінної, що містить
//кількість записаних елементів черги
 elem_nums=0;

//Встановлення розміру черги
 length=size;

//Встановлення індексів зчитування та запису
//у початкову позицію
 read_loc=write_loc=0;
}

//Реалізація функції, яка записує елемент до черги
template <class T> void queue<T>::write_data(T data)
{
```

```
//Перевірка наявності вільного місця у черзі і
//відображення повідомлення про помилку за відсутності
//вільного місця
 if (elem_nums==length)
 {
  cout << "Error! Turn is full." << endl;
  return;
 }

//Записування даних до черги за наявності вільного
//місця та збільшення індексу запису
 q[write_loc++]=data;

//Якщо досягнуто кінця черги, то перевести індекс
//запису на початок
if (write_loc>=length) {write_loc=0;}

//Збільшення значення лічильника на одиницю
 elem_nums++;
}

//Реалізація функції зчитування елементу даних із черги
template <class T> T queue<T>::read_data()
{
//Перевірка на наявність незчитаних елементів та друк
//повідомлення про помилку за їх відсутності
 if (elem_nums==0)
 {
  if(read_loc==length){ read_loc=write_loc=0;}
  cout << "Error! Turn is empty." << endl;
```

```
  return NULL;
 }

//Зменшення лічильника записаних елементів на одиницю
 elem_nums--;

//Зчитування елементу даних у тимчасову змінну та
//збільшення індексу зчитування на одиницю
int tmp=q[read_loc++];

//Якщо досягнуто кінця черги, то перевести індекс
//зчитування у початок черги
 if (read_loc>=length) { read_loc=0;}

//Повернути зчитане значення, як результат роботи функції
 return tmp;
}

//Головна функція програми
int main()
{
//Створення черги з п’яти елементів
 queue<int> a(5);

//Оголошення змінної для тимчасового зберігання
//елементу даних, зчитаного з черги
 int turn_data;

//Очищення екрану
 system("cls");
```

```
//Заповнення черги до кінця елементами даних
 a.write_data(1);
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);

//Спроба записати дані в заповнену чергу.
//На екрані відобразиться повідомлення про помилку
 a.write_data(6);

//Зчитування елементу черги і вивільнення місця для
//одного елементу даних
 turn_data=a.read_data();

//Відображення зчитаних даних на екрані
 cout << turn_data << endl;

//Записування нового елементу даних до черги на
//звільнене місце
 a.write_data(6);

//Зчитування всіх елементів черги. При досягненні кінця
//черги друк повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
```

```
//Повторне заповнення черги даними
 a.write_data(1);
 a.write_data(2);
 a.write_data(3);
 a.write_data(4);
 a.write_data(5);

//Зчитування першого елементу черги
 turn_data=a.read_data();

//Запис ще одного елементу даних на звільнене місце
 a.write_data(6);

//Зчитування всіх елементів даних черги. При досягненні
//кінця черги друк повідомлення про помилку
 while((turn_data=a.read_data())!=NULL)
 {
  cout << turn_data << endl;
 }
 return 0;
}
```

3.9. Дерева

3.9.1. Бінарні дерева

Дерево – це різновид динамічного списку. Існує багато типів дерев, але ми розглянемо лише бінарні дерева, оскільки вони значно простіші й зручніші у використанні, порівняно з іншими типами дерев [9].

Бінарне дерево – це структура даних, кожен елемент якої, окрім самих даних, містить покажчики на два наступних елементи структури. Один з цих наступних елементів умовно називається лівим, а інший – правим.

Кожен елемент дерева називається вузлом. Перший вузол дерева (з якого воно починається) називається коренем.

Фрагмент дерева разом із вузлом, від якого він починається, називається піддеревом або гілкою.

Множина всіх вузлів, рівновіддалених від кореня, називається рівнем.

Вузол, з якого не починається жодна гілка, називається кінцевим (термінальним) вузлом або листом дерева.

Оскільки дерево бінарне, кожен вузол може породжувати два вузли наступного рівня. Породжені вузли є дочірніми по відношенню до вузла, що їх породив. Породжуючий вузол є батьківським по відношенню до своїх дочірніх вузлів. Батьківський вузол разом із своїми дочірніми складає ланку.

Предками даного вузла називаються вузли, із яких існує шлях до даного вузла. Потомками даного вузла називаються вузли, до яких існує шлях із даного вузла.

Внутрішнім називається вузол, що має потомків.

Сумарна кількість рівнів дерева називається висотою дерева. Вузол має висоту (0). Висота порожньої гілки – (мінус 1).

Строго бінарним деревом називається дерево, у якого кожен внутрішній вузол має не пусті праву й ліву гілки.

Повним бінарним деревом називається строго бінарне дерево, у якого всі рівні заповнені.

Основними операціями при роботі з деревами є: додавання елемента до дерева; пошук елемента дерева, який відповідає заданому критерію; сортування елементів дерева; вилучення елемента дерева.

Процес доступу до елементів дерева називається проходженням дерева. Існує три способи проходження дерев: послідовний (симетричний), низхідний (прямий) та висхідний (кінцьовий).

При послідовному методі доступу до елементів дерева порядок проходження наступний: ліве піддерево → корінь піддерева → праве піддерево. Порядок сортування за зростанням для цього способу

проходження показано на рис. 3.1. При низхідному проходженні дерева порядок обходу елементів наступний: корінь піддерева → ліве піддерево → праве піддерево. При висхідному проходженні дерева порядок обходу елементів наступний: ліве піддерево → праве піддерево → корінь піддерева.

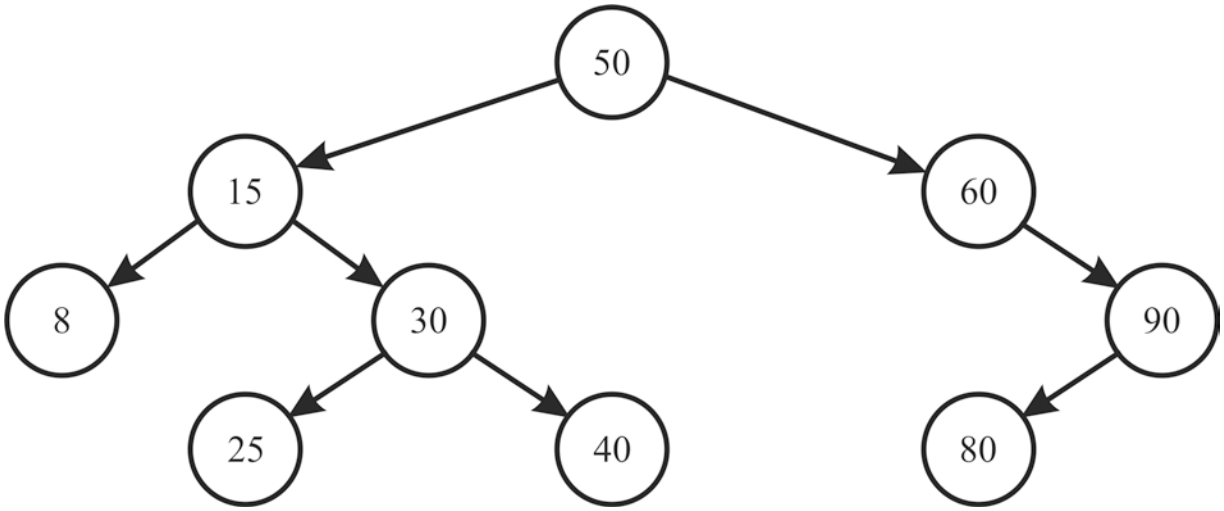
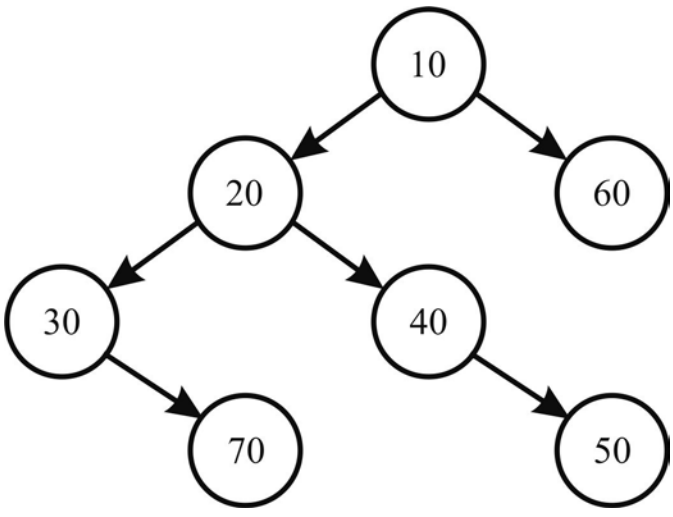


Рис. 3.1. Бінарне дерево; кількість вузлів – 9, глибина – 3

Порядок доступу до елементів бінарного дерева за різних способів проходження показано на рис. 3.2.



Спосіб проходження	Порядок доступу
Послідовний	30 70 20 40 50 10 60
Низхідний	10 20 30 70 40 50 60
Висхідний	70 30 50 40 20 60 10

Рис. 3.2. Способи проходження бінарного дерева

Розглянемо приклад програмної реалізації бінарного дерева, у якому будемо використовувати послідовний метод проходження. Для простоти вважатимемо, що дерево упорядковано (відсортовано) за зростанням відповідно до прямого порядку проходження. За такого упорядкування ліва

гілка, породжена вузлом, містить значення менші за значення цього (батьківського) вузла, а права гілка, породжена даним вузлом, містить значення більші за значення цього вузла. Сам процес упорядкування виконуватимемо методом вставки під час формування дерева. Операцію видалення елемента тут не розглядатимемо. Інформацію з цього питання можна знайти у літературі [8, 9, 10, 11].

```
// автор програми Герберт Шилдт
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;

//клас бінарне дерево
template <class T> class tree
{
//поле, що містить інформацію
  T info;

//покажчик на лівий елемент ланки
  tree * left;

//покажчик на правий елемент ланки
  tree * right;

public:
//покажчик на корінь дерева
  tree * root;

//конструктор
  tree(){root=NULL;};
```

```
//функція, яка будує дерево
  void stree(tree<T> *r, tree<T> *previous, T info);

//функція, яка відображає дерево на екрані
  void print_tree(tree *r, int l);

//функція пошуку елемента дерева
  tree<T>  *search_tree(tree<T> *r, T key);
};

//реалізація функції, яка будує дерево
template <class T> void tree<T>::stree(tree<T> *r,
                                tree<T> *previous, T info)
{
  if (!r)
  {
  //виділення пам'яті під елемент дерева
    r=new tree<T>;
  //якщо не вдалося виділити пам'ять під елемент,
  //повідомляємо про це і виходимо з процедури
    if (!r)
    {
      cout << "Out of memory." << endl;
      exit(1);
    }
  //ініціалізуємо елемент
    r->left=NULL;
    r->right=NULL;
    r->info=info;
```

```
  //якщо треба, робимо його коренем дерева
    if(!root)
    {
      root=r;
    }
  //у протилежному випадку, робимо його звичайним вузлом
    else
    {
    //якщо елемент менший за попередній, то переходимо
    //до лівої гілки
      if(info < previous->info)
      {
        previous->left=r;
      }
      else
      {
      //інакше - до правої
        previous->right=r;
      }
    }
    return;
  }
//коли дійшли до потрібного термінального листа, то
//розміщуємо елемент
//Якщо елемент менший за попередній, то розміщуємо його
//в лівому листі
  if (info < r->info)
  {
    stree(r->left,r,info);
  }
```

```
  else
  {
  //інакше - в правому листі
    stree(r->right,r,info);
  }
return;
};

//реалізація процедури відображення дерева, відповідно
//до порядку послідовного проходження
template <class T> void tree<T>::print_tree(tree<T> *r,
                                            int l)
{
  int i;
  if(!r)
  {
   //якщо нема батьківського вузла, то це корінь і
   //процедуру закінчено
     return;
  }
//друкуємо елементи правої гілки
  print_tree(r->right,l+1);
  for(i=0;i<l;++i)
  {
    cout << " ";
  }
  cout << r->info << endl;
//друкуємо елементи лівої гілки
  print_tree(r->left,l+1);
}
```

```
//реалізація процедури пошуку елемента
template <class T> tree<T> *tree<T>::search_tree
                                    (tree<T> *r, T key)
{
 if(!r)
 {
//якщо нема батьківського вузла, то це корінь і
//процедуру закінчено
  return r;
 }
//перевіряємо елементи дерева, доки не знайдемо шукане
//або не досягнемо кінця
 while(r->info!=key)
 {
  if(key<r->info)
  {
 //перевіряємо елементи лівої гілки
   r=r->left;
  }
  else
  {
 //перевіряємо елементи правої гілки
   r=r->right;
   if(r==NULL){break;}
  }
 }
 return r;
}
```

```
//основна програма
int main()
{
 char s[80], key;

//створюємо об’єкт класу бінарне дерево
 tree<char> chTree;

//очищуємо екран
 system("cls");

//вводимо елементи дерева
 do
 {
   cout << "Input character (. - for stopping):";
   cin >> s;
   if(*s!='.')
   {
     chTree.stree(chTree.root,NULL,*s);
   }
 }
//для закінчення введення натискаємо крапку
 while(*s!='.');

//друкуємо дерево на екрані
chTree.print_tree(chTree.root,0);

//затримуємо зображення, щоб побачити результат
 getch();
```

```
cout << "input character for searching-";
//отримуємо значення для пошуку елемента
cin >> key;

//виконуємо пошук
chTree.print_tree(chTree.search_tree(chTree.root,key),0);

getch();
return 0;
}
```

Як бачимо, бінарне дерево – досить складна структура даних. Фактично – це ускладнений варіант списку. Деякі операції реалізуються значно складніше, ніж у списку. Однак операції пошуку у відсортованому бінарному дереві виконуються значно ефективніше, ніж у списку, тому бінарні дерева використовують на практиці часто.

Зверніть увагу на те, що майже всі методи класу бінарне дерево є рекурсивними, оскільки така реалізація при роботі з бінарним деревом більш ефективна.

3.9.2. АВЛ-дерева

Дерева можуть бути збалансованими або незбалансованими. Збалансоване за висотою дерево – це дерево, у якому висоти лівої і правої гілок будь-якого рівня розрізняються не більше, ніж на задану константу ***k***.

У програмуванні використовують кілька видів збалансованих дерев, але у даному посібнику розглянуто лише АВЛ-дерева.

АВЛ-дерево – це збалансоване за висотою дерево, у якому висоти лівої і правої гілок будь-якого рівня розрізняються не більше, ніж на одиницю (рис. 3.1) [12].

У процесі додавання чи вилучення елементів дерева збалансованість може порушуватись. Це може призводити до виродження дерева у список і, відповідно, до суттєвого зниження ефективності операції пошуку інформації. Тому, основною ідеєю роботи з АВЛ-деревами є перевірка збалансованості після кожної операції додавання чи вилучення елементу дерева і проведення балансування у разі необхідності.

Для контролю збалансованості вводиться коефіцієнт збалансованості вузла (K) – різниця висот його лівої (H_L) і правої (H_R) гілок ($K = H_L - H_R$). Для АВЛ-дерева цей коефіцієнт може приймати лише значення із множини $\{-1, 0, 1\}$. Висота вузла – це довжина найбільшої з породжених ним гілок. На рис. 3.3 висоту кожного вузла показано числом, розміщеним справа від нього, а число зліва від вузла вказує коефіцієнт його збалансованості.

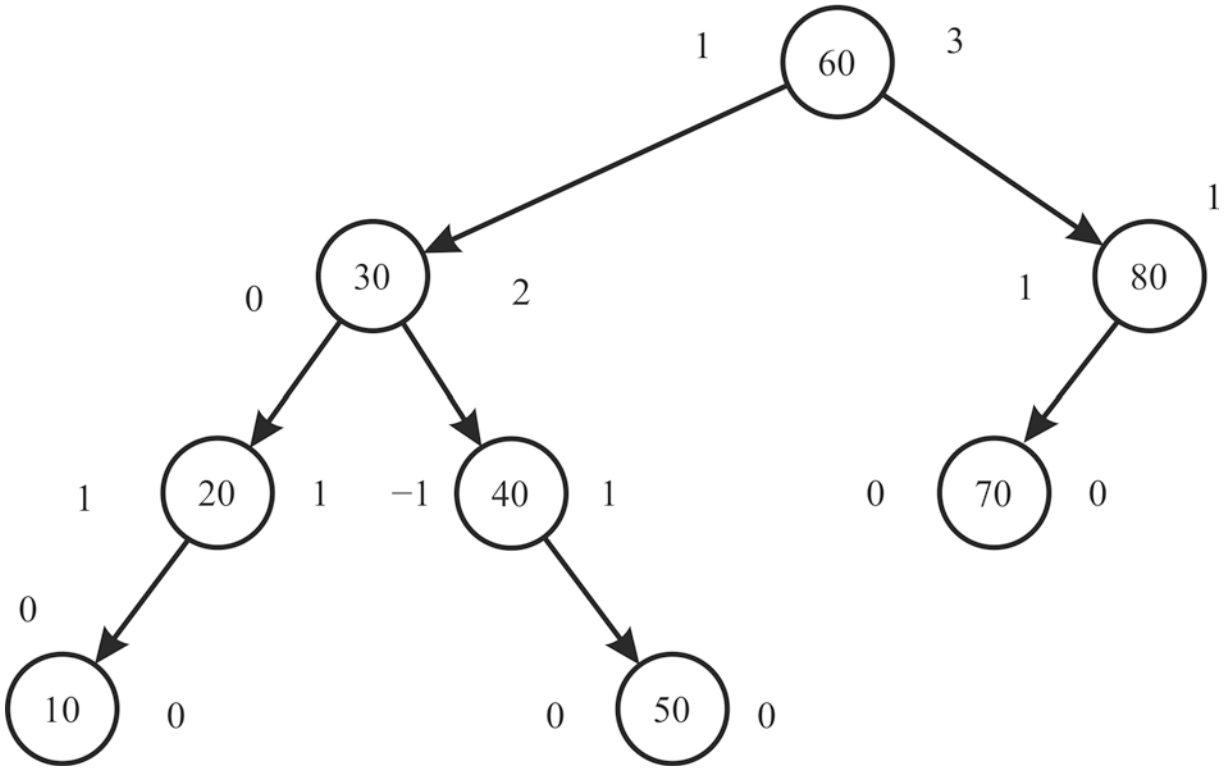


Рис. 3.3. Незбалансоване дерево

Якщо у процесі додавання елемента хоч один з вузлів отримав значення коефіцієнту збалансованості (2) або (–2), то необхідно виконати балансування гілки шляхом обертання.

Обертання бувають наступних типів:

— одиночне праве;

— одиночне ліве;

— подвійне ліво-праве;

— подвійне право-ліве.

Розглянемо приклад. Нехай після додавання елементу отримали ліву гілку зображену на рис. 3.4, *а* (система позначень така ж, як на рис. 3.3). Верхній вузол має коефіцієнт збалансованості $K = 2$. Необхідно виконати балансування шляхом одиночного правого обертання (рис. 3.4, *б*).

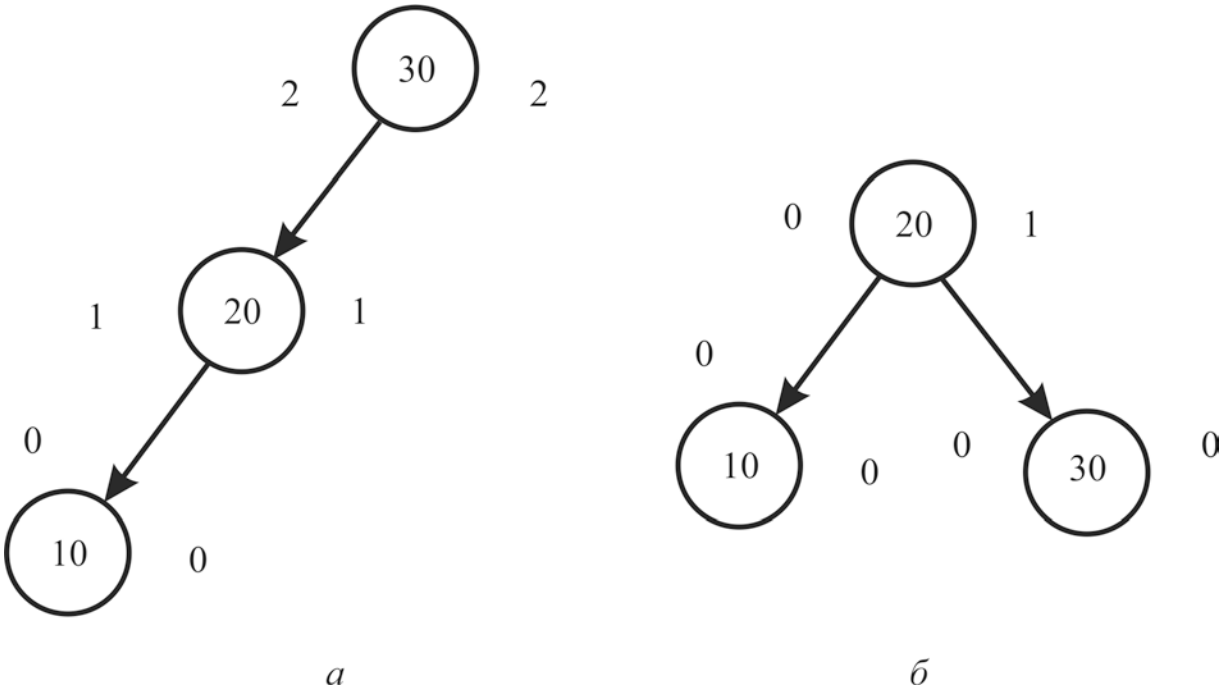


Рис. 3.4. Одиночне праве обертання

Аналогічно у випадку розбалансування правої гілки виконуємо одиночне ліве обертання (рис. 3.5).

У більш загальному випадку одиночне праве обертання показано на рис. 3.6. Додавання елемента Е призводить до порушення збалансованості дерева. Права гілка вузла П (3) коротша від лівої гілки (Л–1–Е) на 2. Для виправлення цієї ситуації треба виконати одиночне праве обертання. При цьому коренем замість вузла П стає вузол Л, а права гілка вузла Л (2) стає

лівою гілкою вузла П. Різниця висот гілок, що виходять з будь-якого вузла не перевищує один елемент. Отже, дерево збалансоване.

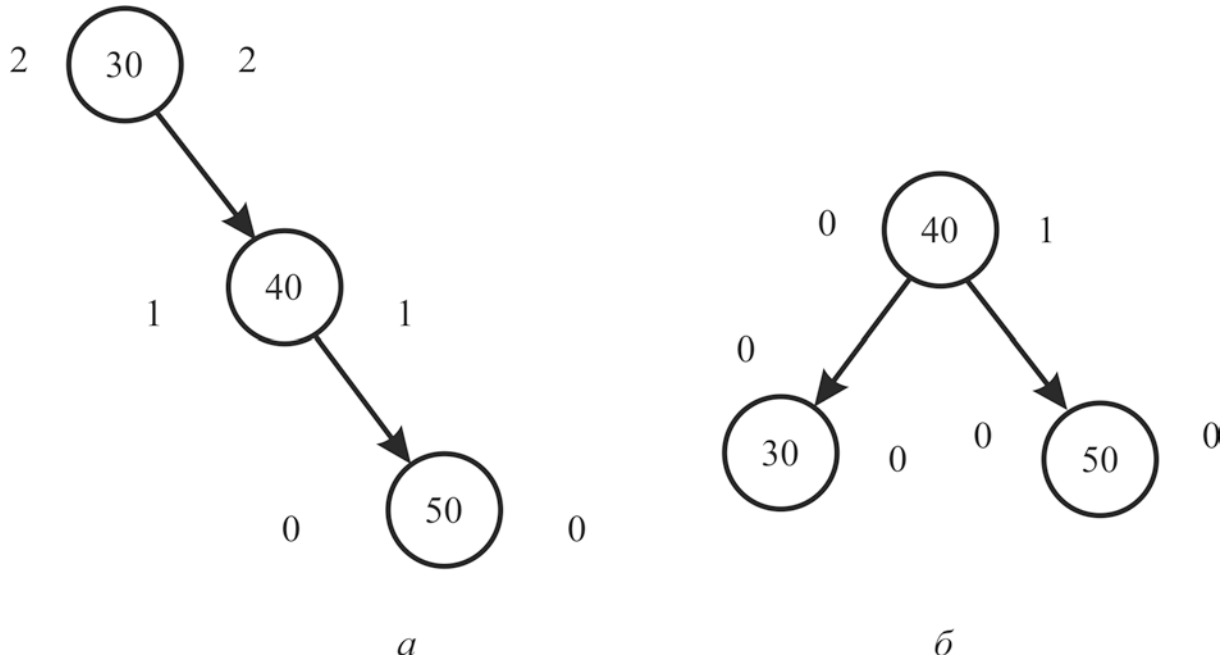


Рис. 3.5. Одиночне ліве обертання

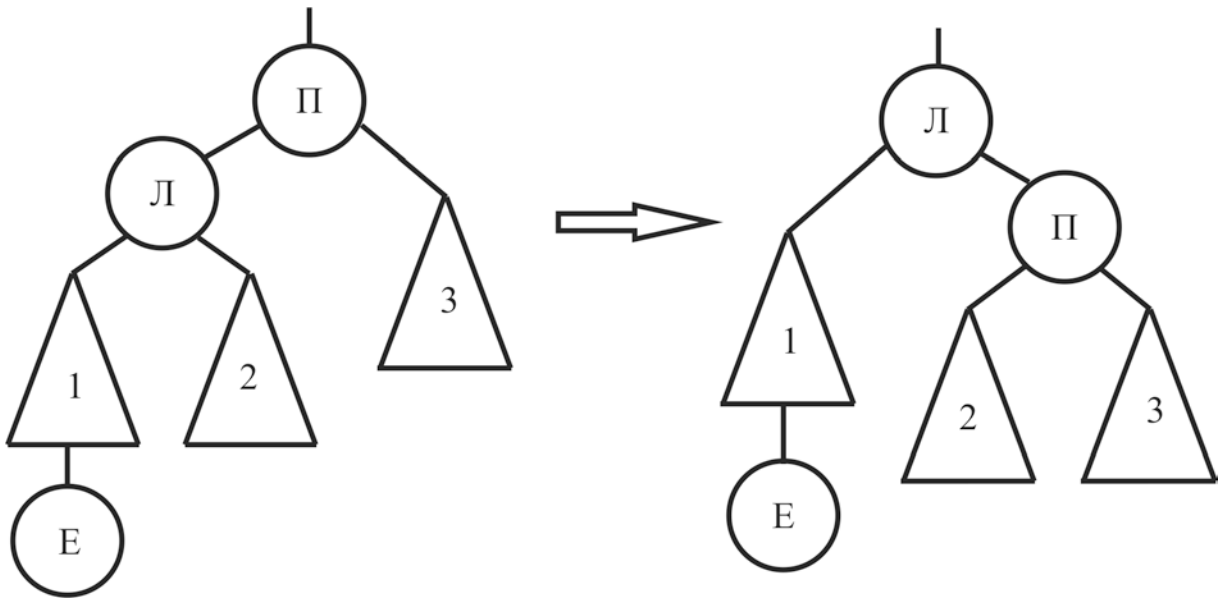


Рис. 3.6. Одиночне праве обертання у загальному випадку

У загальному випадку ліве одиночне обертання (за необхідності) виконують аналогічно з урахуванням зміни напряму. Автори пропонують самостійно зобразити схему для лівого одиночного обертання у загальному випадку.

Подвійне ліво-праве (ЛП) обертання виконують при порушенні балансу дерева після додавання елемента у праву гілку лівого дочірнього вузла (рис. 3.7). Спочатку виконують одинарне ліве обертання. При цьому вузол 2 займає місце вузла 1, а вузол 1 включається в ліву гілку вузла 2; після цього проводять одинарне праве обертання.

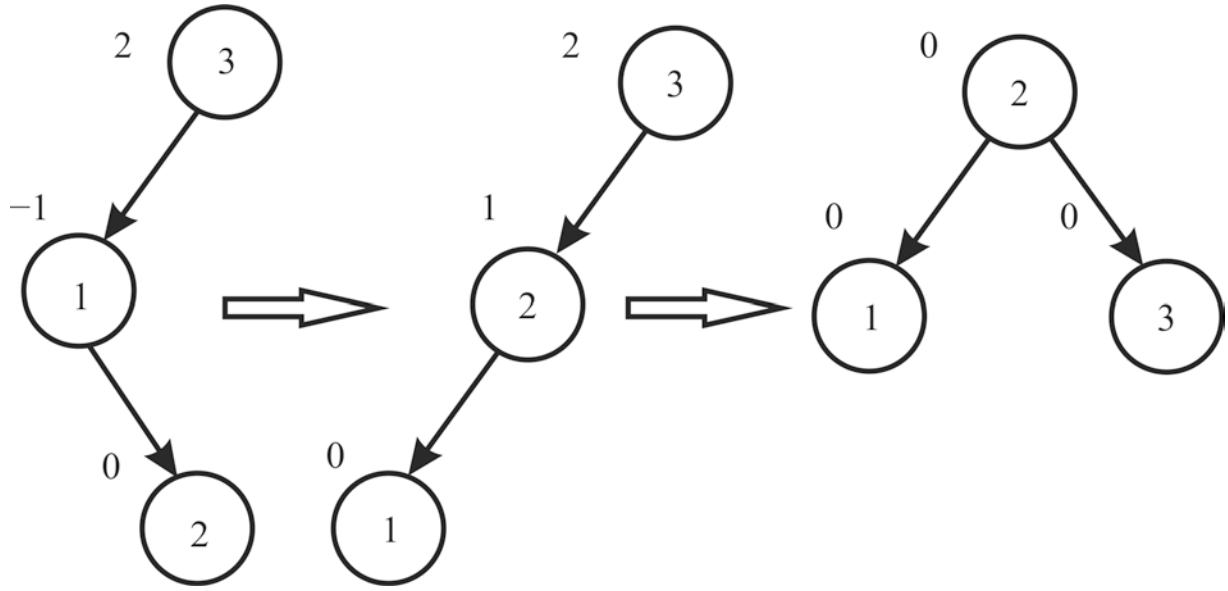


Рис. 3.7. Подвійне ліво-праве обертання

Аналогічно виконують подвійне право-ліве (ПЛ) обертання, якщо баланс дерева порушився після додавання елементу в ліву гілку правого дочірнього вузла (рис. 3.8).

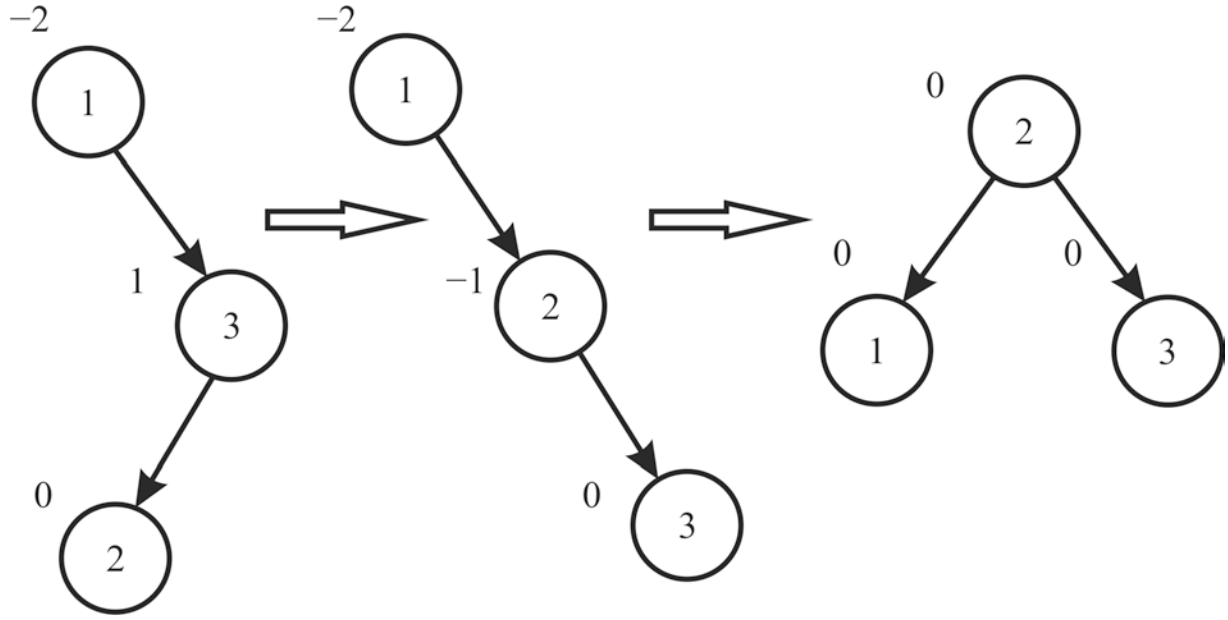


Рис. 3.8. Подвійне право-ліве обертання

Розглянемо приклад програмної реалізації АВЛ дерева:

```
//джерело - http://www.cplusplus.com/forum/beginner/54835/
#include <iostream>
#include <ctype.h>
#include <stdlib.h>
using namespace std;

//вузол дерева
struct node
{
    int element;
    node *left;
    node *right;
    int height;
};

typedef struct node *nodeptr;
class bstree
{
    public:
        void insert(int,nodeptr &);
        void del(int, nodeptr &);
        int deletemin(nodeptr &);
        void find(int,nodeptr &);
        nodeptr findmin(nodeptr);
        nodeptr findmax(nodeptr);
        void makeempty(nodeptr &);
        void copy(nodeptr &,nodeptr &);
        nodeptr nodecopy(nodeptr &);
        void preorder(nodeptr);
```

```
        void inorder(nodeptr);
        void postorder(nodeptr);
        int bsheight(nodeptr);
        nodeptr srl(nodeptr &);
        nodeptr drl(nodeptr &);
        nodeptr srr(nodeptr &);
        nodeptr drr(nodeptr &);
        int max(int,int);
        int nonodes(nodeptr);
};

//додавання вузла
void bstree::insert(int x,nodeptr &p)
{
    if (p == NULL)
    {
        p = new node;
        p->element = x;
        p->left=NULL;
        p->right = NULL;
        p->height=0;
        if (p==NULL)
        {
            cout<<"Out of Space\n"<<endl;
        }
    }
    else
    {
```

```
if (x<p->element)
{
    insert(x,p->left);
    if ((bsheight(p->left) –
                        bsheight(p->right))==2)
    {
        if (x < p->left->element)
        {
            p=srl(p);
        }
        else
        {
            p = drl(p);
        }
    }
}
else if (x>p->element)
{
    insert(x,p->right);
    if ((bsheight(p->right) –
                         bsheight(p->left))==2)
    {
        if (x > p->right->element)
        {
            p=srr(p);
        }
        else
        {
            p = drr(p);
        }
```

```
            }
        }
        else
        {
            cout<<"Элемет существует\n"<<endl;
        }
}
int m,n,d;
m=bsheight(p->left);
n=bsheight(p->right);
d=max(m,n);
p->height = d + 1;
}

//пошук найменшого значення вузла
nodeptr bstree::findmin(nodeptr p)
{
    if (p==NULL)
    {
        cout<<"У дереві немає елементів"<<endl;
        return p;
    }
    else
    {
        while(p->left !=NULL)
        {
            p=p->left;
            //return p;
        }
        return p;
```

```
    }
}

// пошук найбільшого значення вузла
nodeptr bstree::findmax(nodeptr p)
{
    if (p==NULL)
    {
        cout<<"У дереві немає елементів"<<endl;
        return p;
    }
    else
    {
        while(p->right !=NULL)
        {
            p=p->right;
            //return p;
        }
        return p;
    }
}

//пошук елементу
void bstree::find(int x,nodeptr &p)
{
    if (p==NULL)
    {
        cout<<"Шуканий елемент відсутній"<<endl;
    }
    else
```

```
    {
        if (x < p->element)
        {
            find(x,p->left);
        }
        else
        {
            if (x>p->element)
            {
                find(x,p->right);
            }
            else
            {
                cout<<"Шуканий елемент знайдено"<<endl;
            }
        }
    }
}

//копіювання дерева
void bstree::copy(nodeptr &p,nodeptr &p1)
{
    makeempty(p1);
    p1 = nodecopy(p);
}

//очищення дерева
void bstree::makeempty(nodeptr &p)
{
    nodeptr d;
```

```
    if (p != NULL)
    {
        makeempty(p->left);
        makeempty(p->right);
        d=p;
        free(d);
        p=NULL;
    }
}

//копіювання вузлів
nodeptr bstree::nodecopy(nodeptr &p)
{
    nodeptr temp;
    if (p==NULL)
    {
        return p;
    }
    else
    {
        temp = new node;
        temp->element = p->element;
        temp->left = nodecopy(p->left);
        temp->right = nodecopy(p->right);
        return temp;
    }
}
```

```
//видалення вузла
void bstree::del(int x,nodeptr &p)
{
    nodeptr d;
    if (p==NULL)
    {
        cout<<"Елемент не знайдено"<<endl;
    }
    else if ( x < p->element)
    {
        del(x,p->left);
    }
    else if (x > p->element)
    {
        del(x,p->right);
    }
    else if ((p->left == NULL) && (p->right == NULL))
    {
        d=p;
        free(d);
        p=NULL;
        cout<<"Елемент видалено"<<endl;
    }
    else if (p->left == NULL)
    {
        d=p;
        free(d);
        p=p->right;
        cout<<"Елемент видалено"<<endl;
    }
```

```
    else if (p->right == NULL)
    {
        d=p;
        p=p->left;
        free(d);
        cout<<"Елемент видалено"<<endl;
    }
    else
    {
        p->element = deletemin(p->right);
    }
}

int bstree::deletemin(nodeptr &p)
{
    int c;
    cout<<"Обрано видалення найменшого значення"<<endl;
    if (p->left == NULL)
    {
        c=p->element;
        p=p->right;
        return c;
    }
    else
    {
        c=deletemin(p->left);
        return c;
    }
}
```

```
void bstree::preorder(nodeptr p)
{
    if (p!=NULL)
    {
        cout<<p->element<<"\t";
        preorder(p->left);
        preorder(p->right);
    }
}

//відображення елементів дерева у прямому порядку
void bstree::inorder(nodeptr p)
{
    if (p!=NULL)
    {
        inorder(p->left);
        cout<<p->element<<"\t";
        inorder(p->right);
    }
}

//відображення елементів дерева у зворотньому порядку
void bstree::postorder(nodeptr p)
{
    if (p!=NULL)
    {
        postorder(p->left);
        postorder(p->right);
        cout<<p->element<<"\t";
    }
```

```
}

int bstree::max(int value1, int value2)
{
    return ((value1 > value2) ? value1 : value2);
}
int bstree::bsheight(nodeptr p)
{
    int t;
    if (p == NULL)
    {
        return -1;
    }
    else
    {
        t = p->height;
        return t;
    }
}

nodeptr bstree:: srl(nodeptr &p1)
{
    nodeptr p2;
    p2 = p1->left;
    p1->left = p2->right;
    p2->right = p1;
    p1->height   =   max(bsheight(p1->left),bsheight(p1-
>right)) + 1;
    p2->height  =  max(bsheight(p2->left),p1->height)  +
1;
```

```
    return p2;
}
nodeptr bstree:: srr(nodeptr &p1)
{
    nodeptr p2;
    p2 = p1->right;
    p1->right = p2->left;
    p2->left = p1;
    p1->height = max(bsheight(p1->left),
                                bsheight(p1->right)) + 1;
    p2->height = max(p1->height,bsheight(p2->right)) + 1;
    return p2;
}
nodeptr bstree:: drl(nodeptr &p1)
{
    p1->left=srr(p1->left);
    return srl(p1);
}
nodeptr bstree::drr(nodeptr &p1)
{
    p1->right = srl(p1->right);
    return srr(p1);
}

int bstree::nonodes(nodeptr p)
{
    int count=0;
    if (p!=NULL)
    {
        nonodes(p->left);
```

```
        nonodes(p->right);
        count++;
    }
    return count;
}

int main()
{
    nodeptr root,/*root1,*/min,max;//,flag;
    int a,choice,findele,delele;
    bstree bst;

    root = NULL;
    cout<<"\n\t\t\t\tАВЛ Дерево"<<endl;
    cout<<"\t\t\t\t::::::::::::::::::\n"<<endl;

    do
    {
        cout<<"1 - Додати новий вузол"<<endl;
        cout<<"2 - Знайти мінімальний елемент"<<endl;
        cout<<"3 - Знайти максимальний елемент"<<endl;
        cout<<"4 - Пошук за значенням"<<endl;
        cout<<"5 - Видалити елемент"<<endl;
        cout<<"6 - Варіант проходження #1"<<endl;
        cout<<"7 - Варіант проходження #2"<<endl;
        cout<<"8 - Варіант проходження #3"<<endl;
        cout<<"9 - Відобразити висоту дерева"<<endl;
        cout<<"0 - Вихід з програми"<<endl;
        cout<<endl<<endl<<"Оберіть потрібну дію, і
                                    натисніть Enter: ";
```

```
cin>>choice;

switch(choice)
{
    case 1:
        cout<<"\nДодавання вузла"<<endl;
        cout<<"Введіть елемент: ";
        cin>>a;
        bst.insert(a,root);
        cout<<"\nНовий елемент додано"<<endl;
        break;
    case 2:
        if (root !=NULL)
        {
            min=bst.findmin(root);
            cout<<"\nМінімальний елемент у
                 дереві: "<<min->element<<endl;
        }
        break;
    case 3:
        if (root !=NULL)
        {
            max=bst.findmax(root);
            cout<<"\nМаксимальний елемент у
                 дереві: "<<max->element<<endl;
        }
        break;
    case 4:
        cout<<"\nВведіть ключ пошуку: ";
        cin>>findele;
```

```
            if (root != NULL)
            {
                bst.find(findele,root);
            }
            break;
        case 5:
            cout<<"\nЯкий вузол видалити? : ";
            cin>>delele;
            bst.del(delele,root);
            bst.inorder(root);
            cout<<endl;
            break;
        case 6:
            cout<<"\nВаріант проходження #1"<<endl;
            bst.preorder(root);
            cout<<endl;
            break;
        case 7:
            cout<<"\nВаріант проходження #2"<<endl;
            bst.inorder(root);
            cout<<endl;
            break;
        case 8:
            cout<<"\nВаріант проходження #3"<<endl;
            bst.postorder(root);
            cout<<endl;
            break;
        case 9:
            cout<<"Висота дерева: "<<
                        bst.bsheight(root)<<endl;
```

```
                break;
            case 0:
                break;
            default:
                cout<<"Неправильний вибір"<<endl;
                break;
        }
    }while(choice != 0);
    return 0;
}
```

3.10. Графи

3.10.1. Основні визначення

Граф – це сукупність вузлів (вершин) та дуг (ребер), що їх об'єднують. Дуги графу можуть бути напрямлені чи не напрямлені. Якщо дуги графу напрямлені, то такий граф являється ***напрямленим або орієнтованим*** (рис. 3.9).

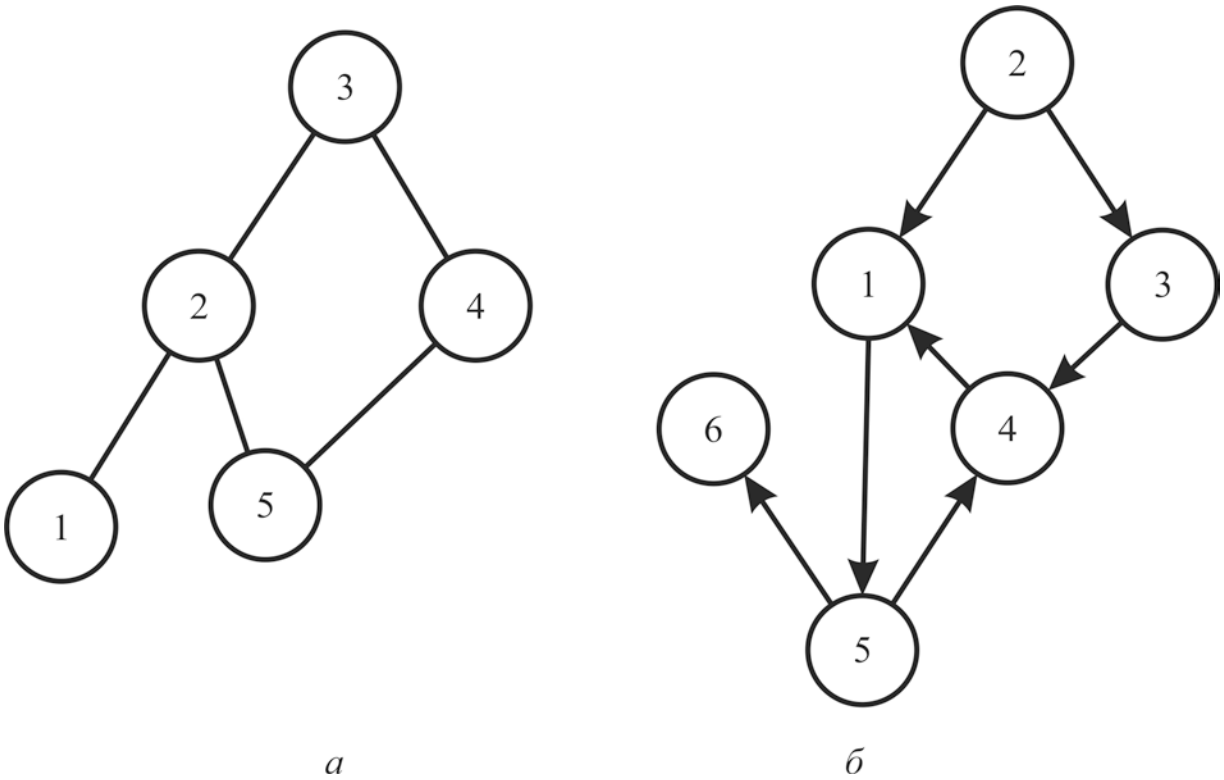


Рис. 3.9. Ненапрямлений (а) та напрямлений (б) графи.

Послідовність ребер, що з'єднують дві (можливо не сусідні) вершини називається ***ланцюгом*** для неорієнтованого графа або ***шляхом*** для орієнтованого.

Граф називається ***зв'язним***, якщо існує ланцюг (шлях) між кожною парою його вершин. Якщо граф не зв'язний, то його можна розбити на k зв'язних компонент, в такому випадку він називається ***k-зв'язним***.

Взважений граф – це граф, ребра якого характеризуються довжиною (вагою). Такий граф ще називають ***мережею***.

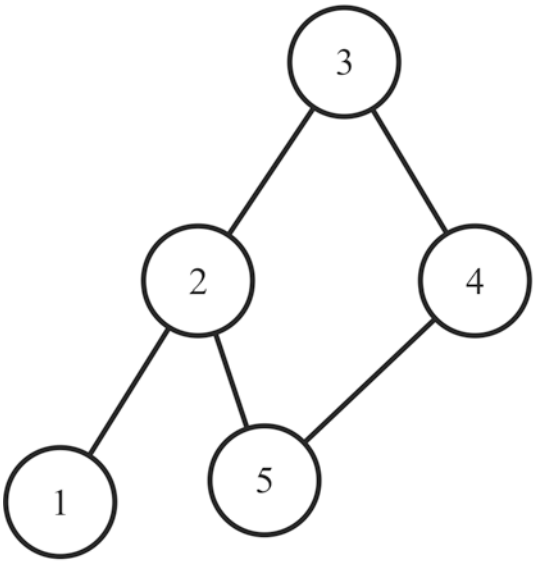
Циклом називається замкнений ланцюг, який закінчується в тій самій вершині з якої почався.

Повним називається граф в якому проведені всі можливі ребра. (для n - вершин – $n*(n-1)/2$ ребер)

3.10.2. Способи опису графів.

Для опису незважених графів користуються матрицею суміжності, а для зважеих графів – ваговою матрицею.

Матриця суміжності для графа з кількістю вершин n – це квадратна матриця розміром $n \times n$, де кожен єлемент з індексами (i, j) вказує на явність дуги із вершини i у вершину j. Якщо значення елементу з індексами (i, j) дорівнює 1 (true), то між вузлами з номерами i та j є дуга. Якщо ж це значення дорівнює 0 (false), то дуга – відсутня (рис. 3.10).

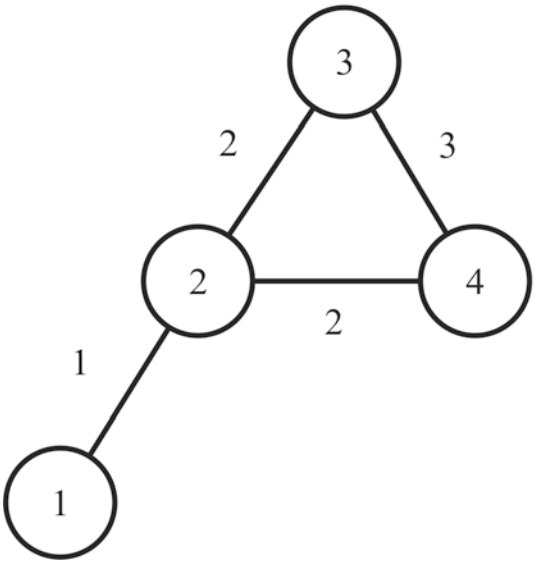


0	1	0	0	0
1	0	1	0	1
0	1	0	1	0
0	0	1	0	1
0	1	0	1	0

Рис. 3.10. Ненапрямлений граф і його матриця суміжності.

Зверніть увагу на те, що діагональні елементи завжди нульові. Крім того для не напрямленого графа матриця суміжності завжди симетрична відносно головної діагоналі. Це дає змогу для економії пам’яті зберігати лише ті елементи матриці, що лежать вище головної діагоналі.

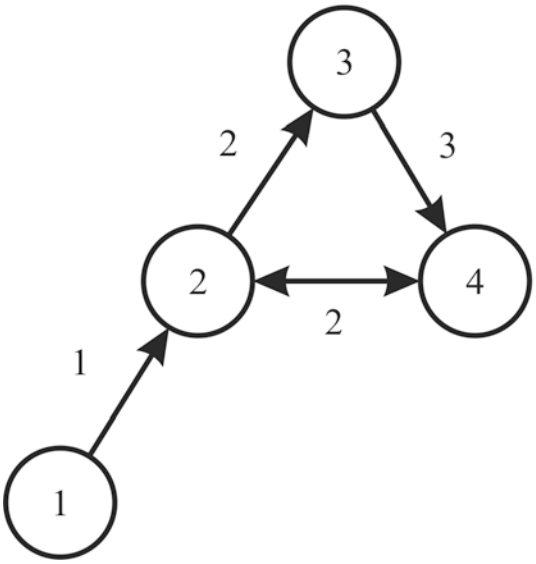
Вагова матриця для графа з кількістю вершин n – це квадратна матриця розміром $n \times n$, де кожен єлемент з індексами (i, j) вказує довжину дуги із вершини i у вершину j. Якщо значення елементу з індексами (i, j) значення дорівнює 0, то дуга – відсутня (рис. 3.11). Вагова матриця для не напрямленого графа також симетрична і має ульові діагональні елементи.



0	1	0	0
1	0	2	2
0	2	0	3
0	2	3	0

Рис. 3.11. Ненапрямлений граф і його вагова матриця.

Для напрямлених графів матриці суміжності і вагові матриці можуть бути несиметричними, оскільки кожна дуга має свій напрям (рис. 3.12).



0	1	0	0
0	0	2	2
0	0	0	3
0	2	0	0

Рис. 3.12. Напрямлений граф і його вагова матриця.

3.10.3. Використання графів.

Графи використовуються в багатьох задачах. Розглянемо деякі з них.

1. Пошук наймених шляхів від заданої вершини графа до всіх інших. Така задача виникає, наприклад, при прокладенні шляхів між населеними пунктами або ліній зв'язку.

Для розв'язку цієї задачі можна скористатися алгоритмом Дейкстри.

Для програмної реалізації алгоритму потрібні два масиви: один з елементами логічного типу visited – для збереження інформації про вершини, які були відвідані (оброблені) та числовий масив distance, в якому зберігатимуться найкоротші шляхи. Маємо деякий граф $G=(V,E)$, де V – множина вершин графа, а E – множина ребер (дуг) графа. Кожна з вершин, що входить в множину V спочатку помічається як не відвідана, тобто всім елементам масиву visited присвоєно значення false. Оскільки найвигідніші шляхи ще невідомі, то в кожний елемент вектора distance записується максимально можливе для обраного типу даних число, що імітує нескінченність. В якості вихідного пункту обирається вершина s, якій приписується нульовий шлях, оскільки відсутнє ребро із вершини в неї ж саму – distance[s]=0. Далі знаходять всі сусідні вершини, тобто які з'єднані з вершиною s ребром або дугою (нехай, наприклад це будуть вершини t та u) і по черзі досліджуються, а саме почергово обчисдюється довжина маршруту в кожну з них:

distance[t]=distance[s]+довжина ребра із s в t;

distance[u]=distance[s]+довжина ребра із s в u.

Можливо, що між двома сусудніми вершинами існує кілька ребер, тоді обирається ребро меншої довжини. Після того, як оброблені всі вершини сусідні з вершиною s, вона позначається як відвідана visited[s]=true, а активною стає сусідня вершина, шлях до якої найменший. Нехай, наприклад, distance[u] менше за distance[t], тоді активною стає вершина u і досліджуються всі сусідні з нею вершини, за винятком вершини s. Далі u помічається як відвідана і активною стає вершина t. Алгоритм Дейкстри виконується доти доки не будуть досліджені всі вершини доступні з вершини s.

Розглянемо алгоритм на конкретному графі (рис. 3.13).

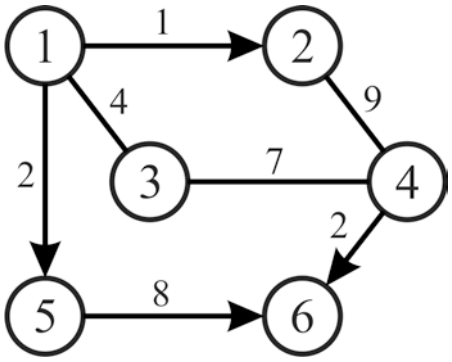


Рис. 3.13.

Розмір (кількість ребер) даного графа 7 ($|E| = 7$), а порядок (кількість вершин) – 6 ($|V| = 6$). Це взважений граф, тому довжина маршруту залежить не тільки від кількості ребер (дуг) між вершинами, але і від їх довжин.

Із всіх вершин оберемо одну – базову, ту від якої потрібно знайти найкоротші шляхи до всіх інших. Нехай це буде вершина 1 (намальована жирною лінією). Позначимо довжини шляхів до кожної з вершин графу як нескінченність, а до неї – як 0 (рис. 3.14).

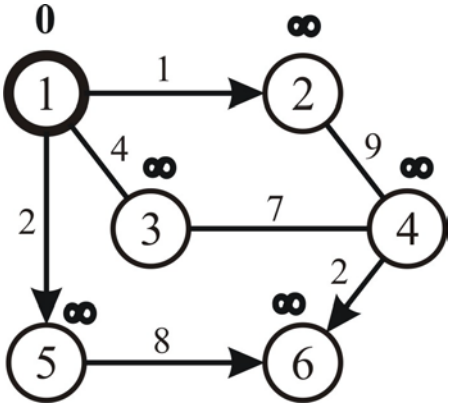


Рис. 3.14.

У вершини 1 є 3 сусіди (вершини 2, 3, 5) і щоб обчислити довжину шляхів до них необхідно просумувати довжини дуг, що лежать між

вершинами 1 і 2, 1 і 3, 1 і 5 із довжиною шляху із 1 в 1 (з нулем): 1 і 2←1+0=1, 1 і 3←4+0=4, 1 і 5←2+0=2.

Як уже було сказано, отримані значення привласнюються вершинам лише в тому випадку, якщо вони менші за ті, що актуальні на поточний момент. На поточний момент всі шляхи крім із 1 в 1 позначені як нескінченні, тому вони отримують нові обчислені значення (рис. 3.15).

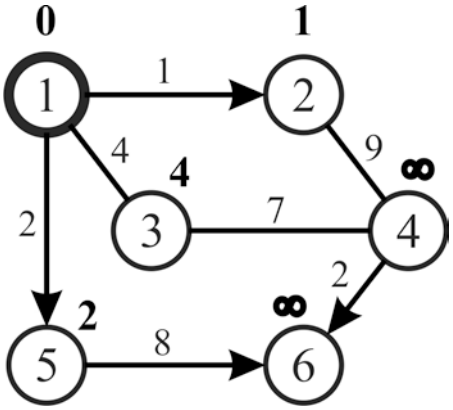


Рис. 3.15.

Далі активна вершина позначається як відвідана (закреслюється на рисунку). Активною стає одна з сусідок, а саме вершина 2, оскільки вона найближча до вершини 1 (рис. 3.16).

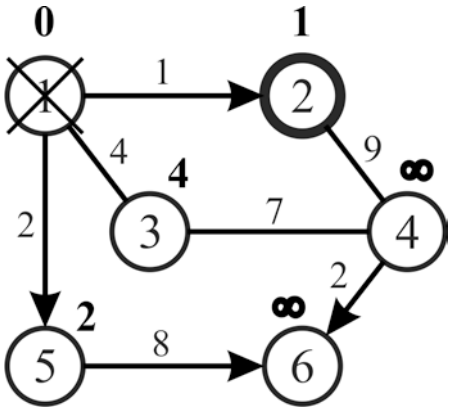


Рис. 3.16.

У вершини 2 лише один не розглянутий сусід (4), відстань до якого 9. обчислимо довжину шляху із 1 в 4:1 і 4←1+9=10. Це значення менше за нескінченність, тому дане значення привласнюють вершині 4 (рис. 3.17).

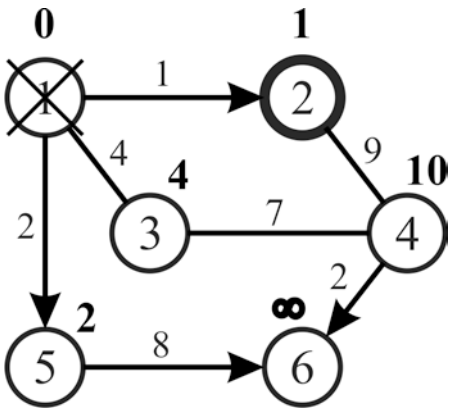


Рис. 3.17.

Вершина 2 перестає бути активною і отримує статус відвіданої.

Тепер в той же спосіб досліджуються сусіди вершини 5 (рис. 3.18).

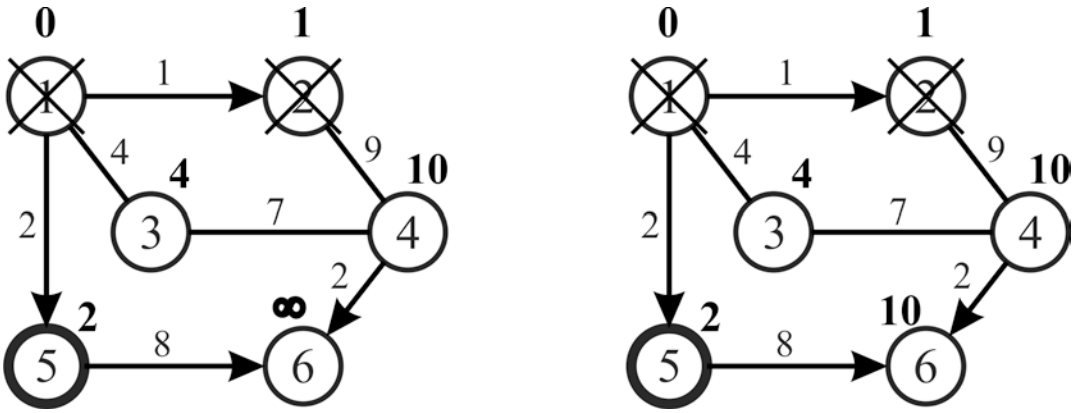


Рис. 3.18

Далі досліджуємо сусідів вершини 3. Це вершина 4. Шлях до вершини 4 через вершину 3 має довжину: 4+7=11. Але вершина 4 вже має обчислений шлях із 1 в 4 через вершину 2. Довжина цього шляху – 10, що менше за 11. Тому шлях із 1в 4 залишаємо рівний 10 (рис. 3.19).

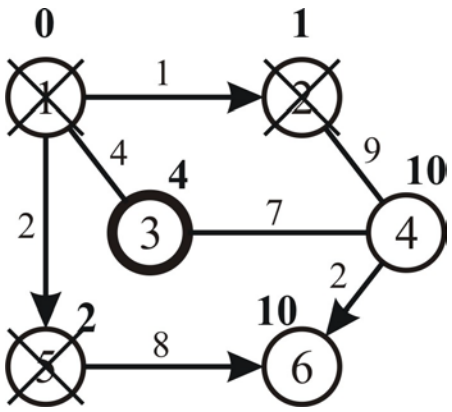


Рис. 3.19

Аналогічна ситуація з вершиною 6. Найкоротший шлях із 1 в 6 лежить через вершину 5 і дорівнює 10 (рис. 3.20).

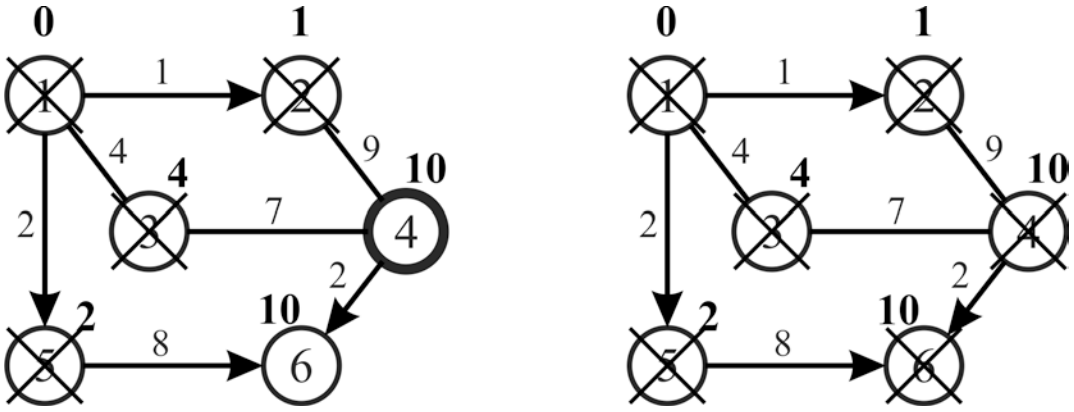


Рис. 3.20

Таким чином задачу розв’язано.

Програмна реалізація розв’язку даної задачі наведена нижче:

```
#include <iostream>
#include <stdlib.h>
#include <limits.h>

using namespace std;
const int V=6;
```

```
//алгоритм Дейкстри
void Dijkstra(int GR[V][V], int st)
{
    //unsigned int  tatar;
    int distance[V], count, index, i, u, m=st+1;
    bool visited[V];
    //cout <<"INT_MAX="<< INT_MAX<<endl;
    for (i=0; i<V; i++)
    {
        distance[i]=INT_MAX; visited[i]=false;
    }
    distance[st]=0;
    for (count=0; count<V-1; count++)
    {
        int min=INT_MAX;
        for (i=0; i<V; i++)
            if (!visited[i] && distance[i]<=min)
            {
                min=distance[i]; index=i;
            }
        u=index;
        visited[u]=true;
        for (i=0; i<V; i++)
            if    (!visited[i]    &&    GR[u][i]    &&
                      distance[u]!=INT_MAX &&
                distance[u]+GR[u][i]<distance[i])
            distance[i]=distance[u]+GR[u][i];
    }
    cout<<"Шлях з початкової вершини до інших:\t\n";
```

```
    for (i=0; i<V; i++) if (distance[i]!=INT_MAX)
        cout<<m<<" > "<<i+1<<" = "<<distance[i]<<endl;
    else
        cout<<m<<" > "<<i+1<<" = "
            <<"маршрут недоступний"<<endl;
}
//головна функція
int main()
{
    setlocale(LC_ALL, "Rus");
    int start, GR[V][V]={
        {0, 1, 4, 0, 2, 0},
        {0, 0, 0, 9, 0, 0},
        {4, 0, 0, 7, 0, 0},
        {0, 9, 7, 0, 0, 2},
        {0, 0, 0, 0, 0, 8},
        {0, 0, 0, 0, 0, 0}};
    cout<<"Початкова вершина >> "; cin>>start;
    Dijkstra(GR, start-1);
    system("pause>>void");
    return 0;
}
```

2. Пошук найменшого шляху, при якому обходяться всі задані вершини (задача побудови мінімального кістякового дерева). Така задача виникає, наприклад, коли потрібно з’єднати телефонним зв’язком кілька міст.

Для розв’язку цієї задачі можна скористатися алгоритмом Прима. Побудова починається з дерева, що включає в себе одну (довільну) вершину. Протягом роботи алгоритму дерево розростається, поки не охопить всі вершини вихідного графа. На кожному кроці алгоритму до поточного дереву

приєднується найменше з ребер, що з'єднують вершину з побудованого дерева і вершину, що не належить дереву.

1. Спочатку ребра сортують за зростанням ваги.
2. Додають найменше ребро в дерево.
3. Зі списку ребер із найменшою довжиною вибирають таке нове ребро, щоб одна з його вершин належала дереву, а інша — ні.
4. Це ребро додають у дерево і знову переходять до кроку 3.
5. Робота закінчується, коли всі вершини будуть у дереві.

Приклад:

1. Вихідний зважений граф. Числа біля ребер показують їх ваги, які можна розглядати як відстані між вершинами.

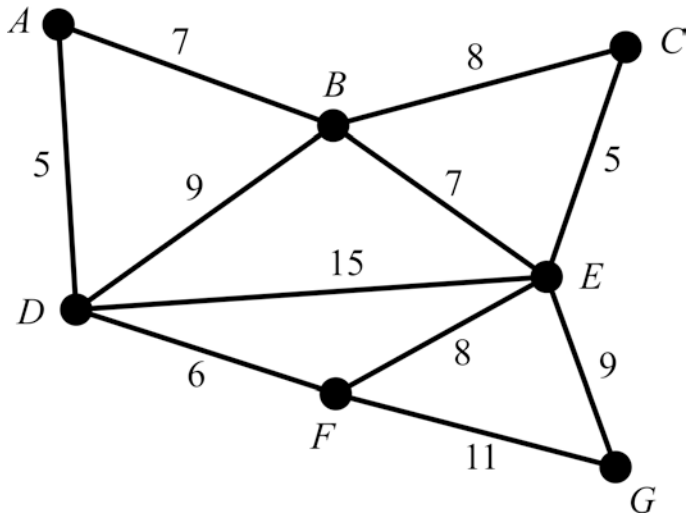


Рис. 3.21

2. В якості початкової довільно вибирається вершина D. Кожна з вершин A, B, E і F з'єднана з D єдиним ребром. Вершина A - найближча до D, і вибирається як друга вершина разом з ребром AD.

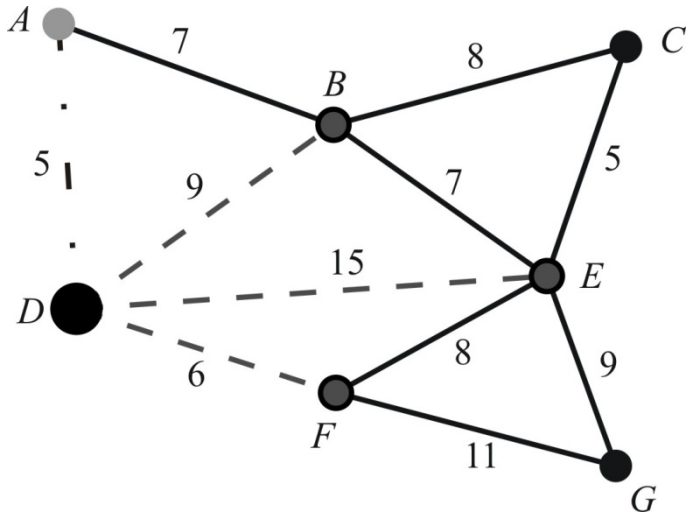


Рис. 3.22

3. Наступна вершина - найближча до будь-якої з обраних вершин D або A. B віддалена від D на 9 і від A - на 7. Відстань до E дорівнює 15, а до F - 6. F є найближчою вершиною, тому вона включається в дерево F разом з ребром DF.

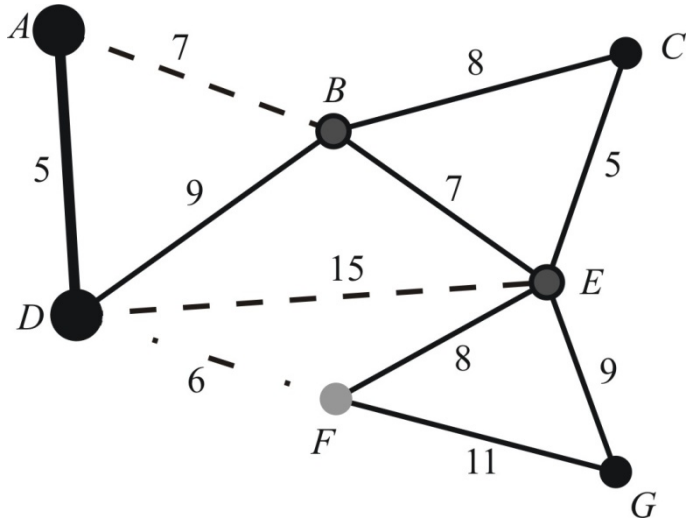


Рис. 3.23

4. Аналогічним чином виконуємо наступні кроки. У цьому випадку є можливість вибрати або C, або E, або G. C віддалена від B на 8, E віддалена від B на 7, а G віддалена від F на 11. E - найближча вершина, тому вибирається E і ребро BE.

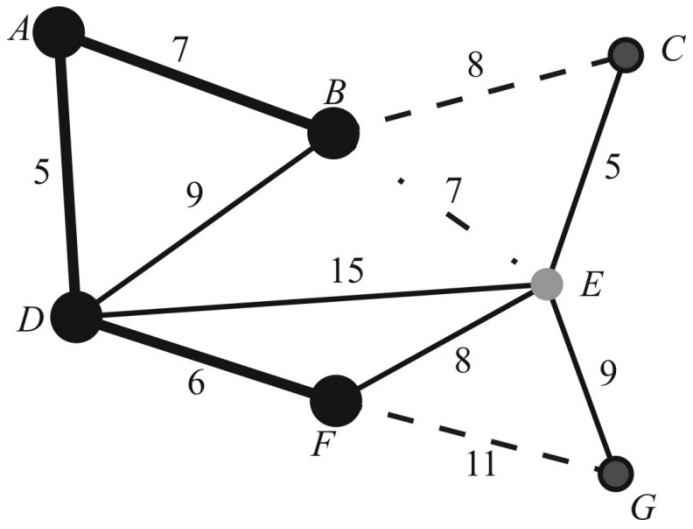


Рис. 3.24

5.Єдина вершина, що залишилася - G. Відстань від F до неї 11, від E - 9. E ближче, тому обирається вершина G і ребро EG.

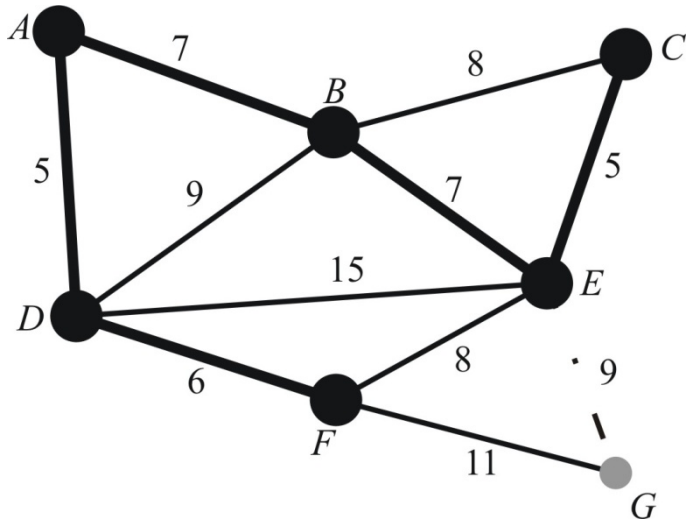


Рис. 3.25

Обрані всі вершини, мінімальне кістякове дерево побудовано

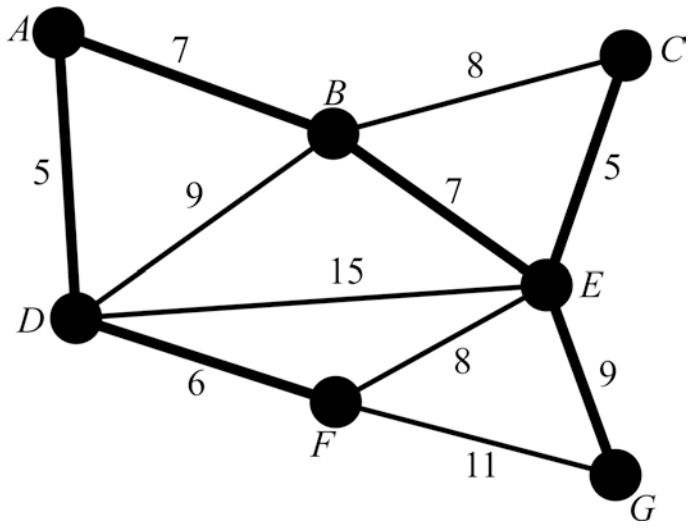


Рис. 3.26

Програмна реалізація алгоритму прима наведена нижче:

```
#include<iostream>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},m_min,mincost=0,cost[10][10];
using namespace std;
int main()
{
    int path[100]={0};    //В цей масив будуть
    //записуватися вершини, по яким складається шлях
    int path_index=0;
    cout<<"Введи количество вершин  "; cin>>n;
    cout<<"Введи матрицу смежности\n";
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
            cin>>cost[i][j];
            if(cost[i][j]==0)
            cost[i][j]=9999; //9999 - нескінченність
        }
        visited[1]=1;
```

```
cout<<"\n";
while(ne < n)
{
    m_min=999;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(cost[i][j]< m_min)
                if(visited[i]!=0)
                {
                    m_min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            if(visited[u]==0 || visited[v]==0)
            {
                path[path_index]=b;
                path_index++;
                cout<<"\n "<<ne++<<"   "<<a<<"
                         "<<b<<min;
                mincost+=m_min;
                visited[b]=1;
            }
            cost[a][b]=cost[b][a]=999;
}
cout<<"\n";
cout<<1<<" --> ";
for (int i=0;i<n-1;i++)
{
    cout<<path[i];
    if (i<n-2) cout<<" --> ";
```

```
        }
        cout<<"\n Мінімальна вартість  "<<mincost;
        cin.get();
        cin.get();
        return 0;
}
```

Контрольні запитання

1. Що таке масив?
2. Які бувають масиви?
3. Чим статичний масив відрізняється від динамічного?
4. Які основні операції виконують над масивами?
5. Що таке сортування?
6. Наведіть класифікацію методів сортування.
7. Які методи сортування називають внутрішніми?
8. Які методи сортування називають зовнішніми?
9. Опишіть алгоритм сортування методом обміну.
10. Опишіть алгоритм сортування методом вибору.
11. Опишіть алгоритм сортування методом включення.
12. Опишіть алгоритм методу прямого пошуку.
13. Опишіть алгоритм методу бінарного пошуку.
14. Що таке список?
15. Наведіть класифікацію списків.
16. Перелічіть основні операції, які виконують над списками.
17. Перелічіть методи сортування, які використовують для списків.
18. Перелічіть методи пошуку, які використовують для списків.
19. Що таке стек?
20. Для чого використовують стеки?
21. Які операції потрібно реалізувати для стеку?

22. Які можливі варіанти реалізації стеку?
23. Чи допускається довільний доступ до елементів стеку?
24. Що таке дек?
25. Які операції потрібно реалізувати для деку?
26. Яка основна відмінність деку від стеку?
27. Які можливі варіанти реалізації деку?
28. Чи допускається довільний доступ до елементів деку?
29. Що таке черга?
30. Для чого використовують черги?
31. Наведіть класифікацію черг?
32. У чому полягає відмінність між простими та циклічними чергами?
33. Що таке бінарне дерево?
34. Що таке вузол дерева?
35. Що таке корінь дерева?
36. Що таке лист дерева?
37. Що таке гілка дерева (піддерево)?
38. Що таке рівень дерева?
39. Який вузол називається кінцевим (термінальним)?
40. Які вузли називають дочірніми?
41. Які вузли називають батьківськими?
42. Які вузли називають предками даного вузла?
43. Які вузли називають потомками даного вузла?
44. Які вузли називають внутрішніми?
45. Що таке висота дерева?
46. Що таке проходження дерева?
47. Які є методи проходження дерева?
48. Який порядок проходження дерева послідовним методом?
49. Що таке збалансоване дерево?
50. Що таке АВЛ-дерево?
51. Що таке коефіцієнт збалансованості?

52. Які значення може приймати коефіцієнт збалансованості для АВЛ-дерева?
53. Що таке висота вузла?
54. Яким чином може здійснюватись балансування АВЛ-дерева?
55. Які види обертань використовують для балансування дерев?
56. Що таке граф?
57. Який граф називається напрямленим?
58. Що таке шлях? Чим він відрізняється від ланцюга?
59. Який граф називається зв’язним?
60. Який граф називається k-зв’язним?
61. Що таке зважений граф?
62. Що таке мережа?
63. Що таке цикл?
64. Який граф називається повним?
65. Що таке матриця суміжності?
66. Що таке вагова матриця?
67. Які характерні риси матриці суміжності або вагової матриці для не напрямленого графа?

Список використаної та рекомендованої літератури

1. Уэйт М. Язык Си. Руководство для начинающих / Уэйт М., Прата С., Мартин Д.; пер. с англ. — М.: Мир, 1988. — 512 с. — ISBN 5-03-001309-1.
2. Карпов Б. C++: специальный справочник / Карпов Б., Баранова Т. — СПб.: Питер, 2000. — 480 с. — ISBN 5-272-00076-5.
3. Бочков С.О. Язык программирования Си для персонального компьютера / Бочков С.О., Субботин Д.М. — М.: Радио и связь, 1990. — 384 с. — ISBN 5-256-00974-5.
4. Пол Ирэ. Объектно-ориентированное программирование с использованием C++ / Пол Ирэ; пер. с англ. — К.: НИПФ «ДиаСофт Лтд», 1995. — 480 с. — ISBN 5-7707-7219-0.
5. Кернигам Б. Язык программирования Си / Кернигам Б., Ритчи Д.; пер. с англ. — М.: Финансы и статистика, 1992. — 272 с. — ISBN 5-279-00473-1.
6. Рассохин Д. От Си к Си++ / Рассохин Д. — М.: Эдэль, 1993. — 128 с. ISBN 5-85308-006-7.
7. Двоеглазов И.М. Язык программирования C++. Справочное руководство / Двоеглазов И.М. — К.: Евроиндекс, 1993. — 128 с. — ISBN 5-7707-3836-7.
8. Вирт, Н. Алгоритмы и структуры данных [Текст] / Н. Вирт – 2-е изд., испр. – С-Пб.: Невский Диалект, 2001.-352 с.: ил.; – Библиогр.: с. 337. – Предм. указ.: с. 346 – 348. – Перевод с английского Д. Б. Подшивалова. – ISBN 5-7940-0065-1 (рус.).
9. Шилдт, Г. Теория и практика C++ [Текст] / Г. Шилдт. – С-Пб.: “BHV – Санкт-Петербург”, 1996. – 416 с.: ил.; – Предм. указ.: с. 409 – 412. – Перевод с английского Ольга Кокарева. – 10000 экз. – ISBN 5-7791-0029-2 (рус.).
10. Фридман А. C/C++. Архив программ [Текст] / А. Фридман, Л. Кландер, М. Михаэлис, Х. Шильдт; под общ. ред. В. Тимофеева – М.: ЗАО „Издательство БИНОМ”, 2001 г. – 640 с.: ил.; –Перевод с английского. – 4000 экз. – ISBN 5-7989-0205-6 (рус.).

11. Седжвик, Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах [Текст] / Роберт Седжвик. – СПб: ООО „ДиаСофтЮП”, 2002. – 496 с. : ил.; – Предм. указ.: с. 479 – 484. – Перевод с английского. –3000 экз. – ISBN 5-93772-054-7.
12. Левитин А.В. Алгоритмы: Введение в разработку и анализ. – М.: Вильямс, 2006. – 576 с.
13. http://www.mkurnosov.net
14. http://kpolyakov.narod.ru
15. http://www.rsdn.ru/article/alg/bintree/avl.xml