Тарас РУДИЙ
Ярослав ПАРАНЧУК
Володимир СЕНИК

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Частина 2
Модульне програмування

Навчальний посібник

Львів
2024

УДК 81:004.93
А45

Рекомендовано до друку та розміщення в електронних сервісах ЛьвДУВС
Вченою радою Львівського державного університету внутрішніх справ
(протокол від 31 січня 2024 року № 8)
Рекомендовано до опублікування Науково-методичною радою
Національного університету «Львівська політехніка»
(протокол від 22 лютого 2024 року № 76)

Р е ц е н з е н т и:
Соколовський Я. І., доктор технічних наук, професор
(Національний університет ″Львівська політехніка″)
Крошний І. М., кандидат технічних наук, доцент,
(Національний лісотехнічний університет України)
Зачек О. І., кандидат технічних наук, доцент
(Львівський державний університет внутрішніх справ)

Рудий Т. В., Паранчук Я. С., Сеник В. В.
А45 Алгоритмізація та програмування. Частина 2. Модульне програмування : навчальний посібник. Львів : Львівський державний університет внутрішніх справ, 2024. 176 с.

ISBN 978-617-511-392-9

Видання підготовлено для того, щоб допомогти здобувачам вищої освіти опанувати технології модульного програмування мовою С++. Наголошено на видатних можливостях вказівників, подано переваги та особливості організації динамічної пам’яті. Значну увагу надано типам даних користувача: структурам, об’єднанням, переліченням. Викладення теоретичного матеріалу супроводжується прикладами програмних кодів для аналізу самими користувачами.

Для здобувачів першого (бакалаврського) рівня вищої освіти спеціальностей: 126 ″Інформаційні системи та технології″, 141 ″Електроенергетика, електротехніка та електромеханіка″, 152 ″Метрологія та інформаційно-вимірювальна техніка″, а також усіх бажаючих освоїти мову програмування С++.

The publication has been prepared in order to support of this discipline, and the main goal of the manual is to help students of higher education master the technologies of modular programming in the C++ language. The irreplaceable role of pointers is emphasized, the advantages and features of dynamic memory organization are presented. Considerable attention is paid to user data types: structures, associations, enumerations. Presentation of the theoretical material is accompanied by examples of program codes for analysis by the users themselves.

For students of the first (bachelor) level of higher education, majors: 126 "Information systems and technologies", 141 "Electroenergetics, electrical engineering and electromechanics", 152 "Metrology and information and measurement technology", as well as all those who wish to learn the C++ programming language.

© Рудий Т. В., Паранчук Я. С.,
Сеник В. В., 2024
© Львівський державний університет
внутрішніх справ, 2024

ISBN 978-617-511-392-9

ЗМІСТ

ВСТУП 5

Розділ 1. ВКАЗІВНИКИ 9
1.1. Основні поняття про вказівники 10
1.2. Оператор отримання адреси & 11
1.3. Оператор розіменування ″*″ 12
1.4. Присвоєння значень вказівнику 16
1.5. Арифметика вказівників 18
Контрольні запитання 24

Розділ 2. ВКАЗІВНИКИ І МАСИВИ 25
2.1. Використання вказівників при роботі з масивами 25
2.2. Схожість між вказівниками і масивами 29
2.3. Відмінності між вказівниками і масивами 31
2.4. Вказівники на багатовимірні масиви 38
Контрольні запитання 41

Розділ 3. ДИНАМІЧНІ МАСИВИ 42
3.1. Відведення та вивільнення динамічної пам'яті 44
3.2. Вказівник на вказівник 46
3.3. Динамічні одновимірні та двовимірні масиви 47
Контрольні запитання 56

Розділ 4. СИМВОЛИ І РЯДКИ 57
4.1. Символьний тип даних у C++ 57
4.2. Рядки символів 59
4.3. Функції для роботи з рядками 62
Контрольні запитання 69

Розділ 5. РЯДКИ ТИПУ string 70
5.1. Використання рядків типу string 70
5.2. Приклади розв'язування задач
з використанням рядків символів 82
Контрольні запитання 95

Розділ 6. МАСИВИ ЯК ПАРАМЕТРИ ФУНКЦІЙ 96
6.1. Одновимірні масиви як параметри функцій 98
6.2. Виклик функцій з аргументом у вигляді масиву 99
6.3. Двовимірні масиви як параметри функцій 102
6.4. Приклади використання масивів
як параметрів функцій 105
Контрольні запитання 115

Розділ 7. ФАЙЛИ 116
7.1. Робота з текстовими файлами у C-stile 117
Контрольні запитання 142

Розділ 8. СТРУКТУРИ 143
8.1. Перейменування типів 143
8.2. Поняття структури 146
8.2.1. Ініціалізування елементів стуктури 148
8.2.2. Доступ до полів структури 149
8.2.3. Масиви структур 153
8.2.4. Вказівники на структури 157
8.2.5. Структури як параметри функцій 157
Контрольні запитання 165

Розділ 9. ОБ’ЄДНАННЯ, ПЕРЕЛІЧУВАННЯ 166
9.1. Поняття об'єднання 166
9.2. Перелічуваний тип даних 169
Контрольні запитання 173

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 174

ВСТУП

Базовий курс ″Алгоритмізація та програмування″ присвячений теоретичним та практичним аспектам розроблення алгоритмів і проєктів програмних кодів мовою С++. У навчальному посібнику синхронно викладено теоретичні компоненти та засоби мови програмування С++ з практичним реалізуванням класичних алгоритмів засобами цієї мови програмування.

Цей курс дає студентам глибоке розуміння структури і можливостей мови програмування С++ для подальшого застосування її потенціалу під час виконання курсових та дипломних проєктів, вивчення дисциплін, що пов’язані з розв’язанням практичних задач.

Основною метою посібника, зокрема частини 2 ″Модульне програмування″, є допомога здобувачам вищої освіти навчитися розробляти проєкти програмних кодів мовою С++ з використанням технології модульного програмування як складової парадигми імперативного програмування.

Додатковим призначенням цього посібника є формування у студентів алгоритмічного мислення, виховання високого рівня абстрагування у практичному реалізуванні базових алгоритмічних конструкцій для виконання математичних обчислень, оброблення і перетворення даних та інформаційних структур.

Технологія модульного програмування є найбільш очевидним підходом у технології програмування назагал. До речі, довільна технологія виробництва складних промислових виробів ґрунтується на компонуванні наборів сумісних і взаємозамінних стандартних деталей.

У імперській політиці такий підхід реалізовує принцип *″Divide et impera″* (розділяй і володарюй).

У чому ж полягає різниця між модулями (функціями) у програмі та модулями іншої технічної системи, наприклад, автомобіля? В обидвох випадках йдеться про завершені вироби, які мають стандартні інтерфейси під’єднання модулів (шланг подавання палива і провідники під’єднання акумулятора у силовому агрегаті автомобіля). Проте, у конкретній технічній системі модулі під’єднуються раз і назавжди, а у інтерфейсах відбуваються безперервні процеси: шлангом подається

паливо, а від акумулятора – напруга. Усі модулі працюють безперервно та паралельно.

У програмних модулях у кожний момент виконується одна функція (F). Якщо у тілі функції F трапляється звернення (виклик) – ім’я іншої функції (G), тоді між ними встановлюється тимчасовий зв’язок: виконання першої функції припиняється доти, доки не виконається друга. Цей принцип отримав назву вкладеності викликів функцій і може бути повторений багаторазово.

З часом, коли принцип модульності став підтримуватись різними мовами програмування, на перший план висунулась вимога логічної та програмної незалежності модулів. Необхідно зауважити, що повної незалежності між модулями бути не може. Залежність між модулями існує тоді, коли:

– використовуються спільні переліки параметрів;

– вони користуються спільними (глобальними) змінними;

– модулі програми залежать від структури даних цієї програми;

– модулі програми залежать від логіки функціонування програми (від того чинника, що модуль може викликатися іншими модулями).

Перше, у чому не можна помилитися: функції синтаксично записуються як незалежні модулі, зв’язки між ними встановлюються через вкладені виклики у процесі виконання, тобто є динамічними.

Далі необхідно встановити різницю між формальними та фактичними параметрами, що насамперед є двома різними поглядами на програмний інтерфейс функції. Формальні параметри – це опис інтерфейсу зсередини, який подається у вигляді оголошення змінних, тобто описуванні властивостей об’єкта, який може бути пересланим на вхід. Ім’я формального параметра – це узагальнене (абстрактне) позначення деякої змінної, яка є ″видимою″ у процесі виконання функції зсередини.

Під час звернення до функції у переліку наявні фактичні параметри, які синтаксично подаються виразами, тобто уже визначеними змінними або проміжними результатами, які у цьому зверненні увідповідрюються формальним параметрам. У такому вигляді вони є поглядом на той же інтерфейс, але уже з боку викликаючої функції.

Отже, формальні та фактичні параметри мають принципово різне синтаксичне подання: оголошення змінних (визначення) і використання (вирази). Зв’язок між ними встановлюється у момент виклику динамічно.

Функція проєктується для оброблення даних взагалі, це узагальнене описування алгоритму для деяких довільних даних, імен, які є їхніми ″майбутніми позначеннями″ під час роботи функції.

Виклик функції навпаки є частковим випадком виконання алгоритму для конкретних даних.

Головна і необхідна умова модульного програмування – навчитися абстрагуватися від конкретних оброблюваних даних і виносити їх ″за межі″ проєктованого алгоритму. Щодо результату виконання функції можна сформулювати ті ж принципи. Результат – це узагальнене значення, яке повертається після звернення до функції у конкретний вираз, з якого здійснюється звернення.

Отже, модулі мають бути незалежні у межах інтерфейсу програми і структури даних. Практика програмування свідчить, що чим вищий ступінь незалежності модулів, то простіше розібратись в окремих модулях і у самій програмі загалом, то менша ймовірність появи нових помилок при усуненні старих, або внесенні змін у програму, тобто менша ймовірність так званого хвильового ефекту. Із сказаного зрозуміло, що не варто без крайньої на те потреби використовувати в модулях глобальні змінні. Всі зв'язки між модулями мають підтримуватися через списки параметрів.

Викладений у посібнику теоретичний матеріал поданий так, що для освоєння матеріалу достатньо знань, здобутих під час вивчення першої частини посібника ″Алгоритмізація та програмування″ (Структурне програмування).

Подані у посібнику приклади ретельно аналізуються стосовно кожного з етапів вивчення окремих розділів.

Посібник дає можливість ознайомити здобувачів вищої освіти та курсантів з технологіями створення проєктів програмних кодів мовою C++, спрямувати останніх на індивідуальну роботу шляхом детального аналізу запропонованих прикладів. Автори подали достатню кількість прикладів, необхідних для успішного засвоєння матеріалу. Такий підхід дає змогу зосередитись на самостійному вивченні та відпрацюванні основних прийомів програмування.

Реалізуванню основної мети підпорядковано і структуру другої частини посібника, яка, на наш погляд, є зручною і мотивованою, тому що чітко відповідає та доповнює прийнятий у закладах вищої освіти принцип: викладення теоретичного матеріалу на лекціях чергується з подальшим засвоєнням його на практичних і лабораторних заняттях. Виклад навчального матеріалу реалізується за принципом повторюваності з попереднім поясненням дій для виконання тієї чи іншої операції. Детальні пояснення подаються, якщо операція виконується вперше. Проте, викладаючи теоретичний матеріал автори не забули і про засадничий принцип у навчанні – щоб навчити студентів основам алгоритмізації та програмування, насамперед необхідне прагнення

студента досягти успіху. Навчити програмувати можна *тільки студента, який сам хоче навчитись*.

Викладення подальшого навчального матеріалу передбачає, що користувач вже володіє попереднім теоретичним матеріалом і ретельно виконав аналіз поданих прикладів. Тому, якщо деяке формулювання, приклад, фрагмент програмного коду здаються незрозумілими, потрібно повернутися назад, до попередніх теоретичних пояснень або відповідних прикладів.

Автори наполегливо рекомендують виконувати усі подані приклади проєктів програмних кодів. Виконуючи це, ви здобудете такий необхідний для вас досвід відлагодження програмних кодів та уміння аналізувати отримані результати на рівні числа.

Звичайно, цей навчальний посібник не претендує на повноту викладення теоретичного матеріалу, розглянуті тільки основні аспекти, а приклади програмних кодів не можуть охопити всю повноту можливостей C++, бо для цього існують довідники, документація і контекстна допомога.

Так, вилучено з розгляду такі аспекти мови C++, як бітові поля, розширене подання даних у вигляді списків та дерев, багатопотоковість та міжпроцесорна взаємодія. Їх вивчення не є предметом цього посібника.

Проте, набутих навичок буде достатньо для вільного володіння C++ у межах початківця, використання цих засобів для розв’язання типових задач та для подальшої самостійної роботи.

Навчальний посібник рекомендований, насамперед, для здобувачів першого (бакалаврського) рівня вищої освіти спеціальностей: 126 ″Інформаційні системи та технології″, 141 ″Електроенргетика, електротехніка та електромеханіка″, 152 ″Метрологія та інформаційно-вимірювальна техніка″, а також усіх охочих освоїти мову програмування C++.

Автори ще раз висловлюють свою *вдячність і повагу* студентам та курсантам другого і третього курсів (на момент виходу з друку другої частини навчального посібника) спеціальності 126 ″Інформаційні системи та технології″ факультету № 2 Львівського державного університету внутрішніх справ, які з неймовірним терпцем, без зайвих нарікань сумлінно виконували лабораторний практикум, чим дуже допомогли нам апробувати теоретичний і практичний матеріал, викладений у другій частині посібника.

Також висловлюємо щиру подяку рецензентам та доктору технічних наук, професорові Юрію Івановичу Грицюку за співпрацю у роботі над посібником і такі необхідні та важливі практичні поради.

Розділ 1 ВКАЗІВНИКИ

Вказівник – один з найпотужніших засобів мови С ++. Вказівник допомагає оперувати числовими значеннями змінних через їхні адреси.

У комп'ютерних науках термін ″вказівник″ тлумачать як різновид посилання, але окремі мови програмування можуть вводити власне інтерпретування цього поняття.

Вказівни́к, пока́жчик або пока́зник, іноді також посилання (*pointer, reference*) – тип даних у мовах програмування, об'єкт програми, який містить у пам'яті комп'ютера адресу іншого об'єкта.

Вказівники разом з оператором їх розіменування (*dereference operator*) винайшла Катерина Логвинівна Ющенко у 1955 році. За термінологією К. Л. Ющенко розіменування вказівника називалося ″штрих-операцією″. Закордонні вчені тривалий час вважали, що вказівники винайшов у 1964 році Гарольд Лоусон.

Узагальнено, що вказівник – це різновид посилання і вказівник посилається на адреси даних, які зберігаються у пам'яті, а для отримання цих даних виконується розіменування вказівника. Основною відмінністю вказівника від посилання є те, що значення вказівника інтерпретується як адреса в пам'яті, що є концепцією низького рівня у програмуванні.

Значення вказівника визначає яка фізична адреса пам'яті (тобто, які дані) використовуються при обчисленнях. Вказівники часто вважаються фундаментальним типом даних у мовах програмування. У статично (або сильно) типізованих мовах програмування (це стосується С++), тип вказівника визначає тип даних, на які посилається вказівник.

Для виокремлення вказівника з-поміж ідентифікаторів змінних перед іменем змінної записують префікс "p". Цей прийом використовується для того, щоб показати, що змінна є вказівником і є доволі розповсюдженим серед програмістів.

1.1. Основні поняття про вказівники

Пам'ять комп'ютера є сукупністю комірок для зберігання інформації, кожна з яких має власний номер. Ці номери називаються адресами. Розмір однієї комірки пам'яті становить 1 байт. При оголошенні змінної у пам'яті виокремлюється ділянка загальним обсягом, який відповідає оголошеному типу змінної, наприклад, 4 байти для типу `int`. Ця ділянка пам'яті пов'язується з іменем змінної. Адреса змінної – це *адреса першої комірки* цієї ділянки.

Як і у інших мовах програмування, у С++ існують спеціальні змінні – вказівники, в яких можна зберігати адреси комірок пам'яті, тобто адреси змінних. Вказівники дають змогу працювати з адресами комірок оперативної пам'яті реалізовуючи непрямий доступ до їхнього вмісту (значення).

Мови програмування, у яких передбачено тип вказівник, зазвичай містять дві основні операції над ними: *присвоювання* і *розіменування*.

Вказівник – це змінна, значенням якої є адреса комірки в оперативній пам'яті. Вказівник, значення якого дорівнює нулю, має назву нульового вказівника. Усі вказівники, коли вони створюються, одразу або пізніше повинні бути ініціалізовані. Якщо певне значення поки не можна визначити, вказівнику доцільно присвоїти нуль. Неініціалізований вказівник називається диким вказівником (*wild pointer*). Дикі вказівники дуже небезпечні.

Вказівники оголошуються так само, як і звичайні змінні, тільки із зірочкою між типом даних та ідентифікатором.

Формат синтаксичної конструкції для оголошення вказівника:

```
<тип> * <ідентифікатор>;
```

де:
`тип` – це базовий тип вказівника, котрим може бути довільний тип;
ідентифікатор – ідентифікатор змінної-вказівника.

Варто зауважити, що тип – це не тип вказівника, а тип даних, фізичну адресу яких буде присвоєно вказівнику.

Базовий тип вказівника визначає тип даних, на які він посилатиметься.

Наприклад:

```
int *iptr;       //вказівник на значення типу int
double *dptr;    //вказівник на значення типу double
```

Змінна `iptr` – це *фізична адреса* першого байту ділянки пам'яті, в якій розташоване ціле число, `dptr` – фізична *адреса* першого байту ділянки пам'яті, в якій розташоване дійсне число.

Синтаксично мова С++ приймає оголошення вказівника, коли зірочка розташована поруч з типом даних, з ідентифікатором або навіть посередині між ними. Зауважте, ця зірочка **НЕ** є оператором розіменування. Це лише частина синтаксичної конструкції для оголошення вказівника.

Однак, при оголошенні кількох вказівників зірочка повинна знаходитися біля *кожного* ідентифікатора. Це легко забути, якщо Ви звикли вказувати зірочку поруч з типом даних, а не поруч з ідентифікатором змінної.

Наприклад:

```
int *iptr1, iptr2;
```

`*iptr1` – це вказівник на значення типу `int`, а `iptr2` – звичайна змінна типу `int`.

З цієї причини, при оголошенні вказівника, рекомендується вказувати зірочку поруч з ідентифікатором змінної.

Вказівники не ініціалізуються при оголошенні. Вмістом неініціалізованого вказівника є "звичайне сміття".

Фактично вказівник довільного типу може посилатися на якесь довільне місце в пам'яті. Однак операції, які можуть виконуватися з вказівником, суттєво залежать від його типу. Вказівникові можна присвоїти лише адресу змінної відповідного типу, оскільки С++ не виконує автоматичного перетворення типів вказівників. З огляду на сказане, такі настанови присвоєння є *помилковими*:

```
int *iptr;
double *dptr;
iptr = dptr; /* не можна переприсвоювати один одному
вказівники на різні типи значень*/
iptr = 10;  /* не можна присвоювати числа вказівни-
кам*/
```

1.2. Оператор отримання адреси &

Під час оголошення змінної їй автоматично присвоюється вільна адреса в оперативній пам'яті та відповідне числове значення, яке ми присвоюємо змінній, зберігається за цією адресою в пам'яті. Наприклад:

```
int a;
```

Під час виконання цієї настанови процесором, відводиться частина оперативної пам'яті. Наприклад, припустімо, що змінній `a` відводиться комірка пам'яті під номером 150. Кожного разу, коли компіля-

тор зустрічає змінну a у виразі або у настанові, він розуміє, що для того, щоб отримати значення, йому потрібно зазирнути до комірки пам’яті під номером 150.

Хороша новина – нам не потрібно турбуватися про те, які конкретно адреси в пам’яті визначені для певних змінних. Ми просто посилаємося на змінну через присвоєний їй ідентифікатор (символічне ім’я), а компілятор конвертує цей ідентифікатор у відповідну адресу в пам’яті.

Оператор *отримання адреси* & визначає, яку адресу в оперативній пам’яті присвоєно певній змінній. Наприклад:

```
#include <iostream>
using namespace std;
int main()
{
	int a = 7;
	cout << "\nZnachennia zminnoji a = " << a;
	//вивід значення змінної a
	cout << "\nAdresa zminnoji a: " << &a;
	//вивід з пам’яті адреси значення змінної a
	system("pause");
	return 0;
}
```

Результат виконання програмного коду на моєму комп’ютері:

```
Znachennia zminnoji a = 7
Adresa zminnoji a: 004FFC10
```

Оголосимо цілу змінну a, якій присвоїмо значення 7 і вказівник ptra. Отже, щоб адресу змінної a присвоїти вказівнику ptra, треба записати такий фрагмент програмного коду:

```
int a = 7;
int *ptra;
// значенням змінної ptra є адреса змінної a
ptra = &a;
```

1.3. Оператор розіменування "*"

У C++ важливе значення має оператор розіменування вказівника "*" (*dereferencing*). Оператор "*" надає можливість звертатися до числового значення змінної через вказівник, у якому міститься адреса цієї змінної.

Отже, оператор розіменування "*" дозволяє отримати значення за вказаною адресою.

Скористаємося аналізом програмного коду, який подано у такому прикладі:

```
#include <iostream>
using namespace std;
int main()
{
      int a = 7;
      //вивід значення змінної a
      cout << "\nZnachennia zminnoji a= " << a;
      //вивід з пам'яті адреси значення змінної a
      cout << "\nAdresa zminnoji a: " << &a;
      //вивід з комірки пам'яті значення змінної a
      cout << "\nRozimenovane znachenni zminnoji a"
<< "\nza vkazanoju adresoju a = " << *&a;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду на моєму комп'ютері:

```
Znachennia zminnoji a= 7
Adresa zminnoji a: 008FF908
Rozimenovane znachenni zminnoji a
za vkazanoju adresoju a = 7
```

Тепер розглянемо випадок, коли до значення, яке зберігається у змінній `a`, можна звернутися не лише за ім'ям `a`, але і через вказівник `ptra`. Для цього треба застосувати оператор розіменування вказівника (*), наприклад: *`ptra`. Це означає: *"отримати значення, яке зберігається за адресою, записаною у змінній* `ptra`*"*.

Оголосимо цілі змінні `b` і `a`, якій присвоїмо значення 7 та вказівник `ptra`. Вказівникові `ptra` присвоїмо результат виконання оператора отримання адреси змінної `a`, після чого змінній `b` – результат розіменування вказівника `ptra`. Оскільки вказівники містять тільки адреси, тому при присвоюванні значення вказівнику – це значення повинно бути адресою. Для отримання адреси змінної `a` використовується оператор отримання адреси:

Проілюструємо цей процес проєктом програмного коду:

```
#include <iostream>
using namespace std;
int main()
```

```
{
	int a = 7, b;
	int *ptra;
	cout << "\nZnachennia zminnoji a = " << a;
	cout << "\nAdresa zminnoji a: " << &a;
	ptra = &a;
	cout << "\nZnachennia zminnoji ptra: " <<
ptra;
	b = *ptra; \
	cout << "\nZnachennia zminnoji b = " << b;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду на моєму комп'ютері:

```
Znachennia zminnoji a = 7
Adresa zminnoji a: 00F7FBE8
Znachennia zminnoji ptra: 00F7FBE8
Znachennia zminnoji b = 7
```

Хоча оператор розіменування виглядає так само, як і оператор множення, відрізнити їх можна за тією ознакою, що оператор розіменування – унарний і має асоціативність зліва праворуч, а оператор множення – бінарний.

Оператор розіменування слугує доповненням до *оператора отримання* адреси.

Оператор розіменування є унарним оператором і він звертається до значення змінної, розташованої за адресою, заданою його операндом. Інакше кажучи, він посилається на значення змінної, яка адресується заданим вказівником.

Пам'ятайте, що оператори "*" і "&" мають вищий пріоритет, ніж довільний з арифметичних операторів, за винятком унарного мінуса, пріоритет якого такий же, як і у операторів, що вживаються для роботи з вказівниками.

Операції, які виконуються за допомогою вказівників, часто називають *операціями непрямого доступу*, оскільки ми опосередковано отримуємо доступ до змінної за допомогою деякої іншої змінної. *Операція непрямого доступу – механізм використання вказівника для доступу до деякого об'єкта*.

Використання оператора отримання адреси та вказівника для доступу до значення змінної подано у таблиці (рис. 1.1) і покрокові операції з цієї таблиці також реалізуються проєктом програмного коду.

```
#include <iostream>
using namespace std;
int main()
{
// оголошення вказівника з іменем pi на тип int
	int *pi;
	int i;
	i = 8;
	cout << "i=" << i << endl;
// pi вказує на адресу змінної i
	pi = &i;
	cout << "Adresa pi=" << pi << "\tznachennia
pi=" << *pi << endl;
	*pi = 10;
	cout << "Adresa pi=" << pi << "\tznachennia
pi=" << *pi << endl;
	cout << "i=" << i << endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду на моєму комп’ютері:

```
i=8
Adresa pi=00AFF914      znachennia pi=8
Adresa pi=00AFF914      znachennia pi=10
i=10
```

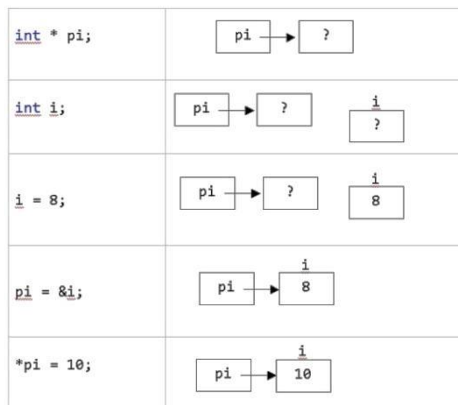

Рис. 1.1. Використання оператора отримання адреси та вказівника для доступу до значення змінної

Потрібно пам'ятати, що у програмному коді оголошенням змінної визначається тільки її символічне ім'я (ідентифікатор), а її числове значення є не визначеним. Це стосується і вказівників.

1.4. Присвоєння значень вказівнику

Тепер, коли ми вже дещо знаємо про оператори отримання адреси і розіменування, можемо поговорити про присвоєння значень вказівнику.

Отже, нагадуємо. Щоб присвоїти значення або *ініціалізувати вказівник*, перед ім'ям змінної, адреса значення якої присвоюється вказівнику, розташовують *оператор отримання адреси* (&). Наприклад:

```
int i = 5;
int *ptr;
ptr = &i;
```

Цим фрагментом програмного коду ми насамперед оголошуємо змінну `i` цілого типу, а пізніше оголошуємо вказівник `*ptr` теж цілого типу і присвоюємо йому адресу змінної `i` (ініціалізуємо вказівник).

Тепер, щоб отримати доступ до числового значення, яке зберігається за адресою `&i`, використовуємо розіменування (*dereferencing*) вказівника `ptr`. Для розіменування вказівника оператор розіменування (*) встановлюється перед ім'ям вказівника.

З урахуванням результату попереднього прикладу отримаємо:

```
int j = *ptr + i; // j = 5 + 5 = 10
```

Фактично `*ptr` є синонімом `i`. З `*ptr` можна працювати як зі звичайною змінною цілого типу, наприклад:

```
*ptr = 6; // i = 6
```

Тип вказівника повинен відповідати типу змінної, на яку він вказує. Приклад фрагменту програмного коду:

```
int ivalue = 7;
double dvalue = 9.0;
int *iptr = &ivalue;    // Ok
double *dptr = &dvalue; // Ok
iptr = &dvalue; /*неправильно: вказівник типу int
                не може вказувати на адресу змінної
                типу double*/
dptr = &ivalue; /*неправильно: вказівник типу
                double не може вказувати на адресу
                змінної типу int*/
```

Наступна настанова є неприпустимою:

```
int ptr = 7;
```

Це пов’язано з тим, що вказівники можуть містити тільки адреси, а цілочисельне значеня 7 не має адреси в пам’яті. Якщо ви все ж зробите це, тоді компілятор повідомить вам, що він не може перетворити цілочисельне значення у цілочисельний вказівник.

C++ також не дозволить вам напряму присвоювати адреси в пам’яті вказівнику:

```
double *dptr = 0098F9C4;
```

Такий запис кваліфікується як присвоєння цілочисельного літералу.

Під час присвоєння значення області пам’яті, яка адресується вказівником, його (вказівник) можна використовувати у лівій частині настанови присвоєння використовуючи оператор розіменування. Наприклад, у процесі виконання такої настанови (якщо `ptr` – вказівник на цілочисельний тип)

```
*ptr = 102;
```

число 102 присвоюється області пам’яті, в `ptr`, яка адресується вказівником, Отож, цю настанову можна прочитати так: "за адресою `ptr` поміщаємо значення 102".

Збільшити на 2 значення оголошеної у попередніх прикладах змінної `a` можемо так:

```
a = a + 2;
```

або так:

```
*ptra = (*ptra) + 2;
```

З ділянки пам’яті за адресою, записаною у змінній `ptra`, буде взяте значення, яке зберігається у ній, збільшене на 2 і записане до тієї ж ділянки.

Щоб використати оператор інкременту або декременту до значення, розташованого у області пам’яті, яка адресується вказівником, можна використовувати настанову, схожу до такої:

```
(*ptr)++;
```

Круглі дужки тут є обов’язковими, оскільки оператор "*" має нижчий пріоритет, ніж оператор "++".

Присвоєння значень з використанням вказівників продемонстровано поданим далі проєктом програмного коду.

```
#include <iostream>
using namespace std;
```

```
int main()
{
int *ptr, n;
ptr = &n;
*ptr = 100;
cout << "\nn = " << n;
(*ptr)++;
cout << "\nn = " << n;
(*ptr)--;
cout << "\nn = " << n;
system("pause");
return 0;
}
```

Результат виконання програмного коду.

```
n = 100
n = 101
n = 100
```

1.5. Арифметика вказівників

Вказівники можна використовувати у більшості виразів, які записані відповідно до синтаксичних вимог мови програмування С++. Але водночас потрібно застосовувати спеціальні правила і не забувати, що деякі частини таких виразів необхідно брати в круглі дужки, щоб гарантовано отримати бажаний результат.

До вказівників можна використовувати тільки чотири арифметичних оператори: ++; --; + і -. Щоб краще зрозуміти, що відбувається у процесі виконання арифметичних дій з вказівниками, почнемо з конкретного прикладу.

У нашому випадку `ptr1` – вказівник на змінну типу `int` з поточним значенням 2 000 (тобто значенням вказівника `ptr1` є адреса 2 000). Після виконання (у 32-розрядному середовищі) виразу `ptr1++;` **вміст змінної-вказівника** `ptr1` **дорівнюватиме 2 004, а не 2 001!** Йдеться про те, що під час виконання кожного оператора інкременту вказівник `ptr1` вказуватиме на *таке* `int`-значення (збільшуватиметься значення адреси на 4 байти, необхідних для розміщення `int`-значення).

Для оператора декременту справедливе зворотне твердження, тобто під час виконання кожного оператора декременту значення `ptr1` зменшуватиметься на 4. Наприклад, після виконання оператора

ptr1--; вказівник ptr1 матиме значення 1 996, якщо до цього воно дорівнювало 2 000.

Отже, кожного разу, коли до вказівника використано оператор інкременту, він вказуватиме на область пам'яті, яка містить такий елемент базового типу цього вказівника, а за виконання оператора декременту він вказуватиме на область пам'яті, яка містить попередній елемент базового типу цього вказівника.

Для вказівників на значення символьного типу результат виконання операторів інкременту і декременту буде таким самим, як за "звичайної" арифметики, оскільки значення символьного типу займають тільки один байт. Але під час використання довільного іншого типу вказівника у процесі виконання опратора інкременту/декременту значення змінної-вказівника збільшуватиметься/зменшуватиметься на величину, яка дорівнює розміру його базового типу.

Арифметичні операції над вказівниками не обмежуються використанням операторів інкременту і декременту. Вказівники можуть бути операндами у операторах додавання і віднімання, використовуючи як другий операнд цілочисельні значення.

Настанова

```
ptr1 = ptr1 + 9;
```

примушує ptr1 змістити посилання на дев'ять елементів базового типу вказівника ptr1 щодо елемента, на який ptr1 посилався до виконання цієї настанови.

Хоча віднімати вказівники ризиковано, проте один вказівник все ж можна відняти від іншого (якщо вони обидва мають один і той самий базовий тип). Різниця покаже кількість елементів базового типу, які розділяють ці два вказівники.

Крім додавання вказівника з цілочисельним значенням і віднімання його від вказівника, а також обчислення різниці двох вказівників, над вказівниками жодні інші арифметичні операції *не виконуються*. Наприклад, до вказівників не можна додавати float- або double-значення.

Приміром, з вказівником ptr, який вказує на тип int, виконано оператор компаундного присвоєння

```
ptr += 3;
```

Якщо початкове значення ptr дорівнює 0012FE60, то після додавання цілого числа 3 отримаємо адресу 0012FE6C. Якщо змінна-вказівник ptr має значення 0012FE6C, тоді оператор віднімання

```
ptr -= 2;
```

поверне значення 0012FE64 (рис. 1.2).

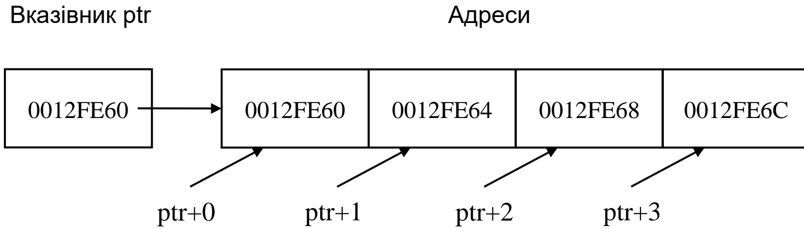

Рис. 1.2. Арифметичні операції над вказівниками

Щоб зрозуміти як формується результат виконання арифметичних операторів з вказівниками, виконаємо такий програмний код. У результаті будуть виводитися реальні фізичні адреси, які містять вказівник на `int`-значення (`c`) і вказівник на `float`-значення (`f`). Візьміть до уваги кожну зміну адреси (залежну від базового типу вказівника), яка відбувається під час кожної ітерації циклу.

Також зауважте на те, що під час використання вказівника в `cout`-настанові його адреса автоматично відтворюється у форматі адресування, що вживається для поточного процесора і середовища виконання.

```
#include <iostream>
using namespace std;
int main()
{
      int *c, a = 0;
      double *f, b = 0.0;
      c = &a;
      f = &b;
      for (int i = 0; i < 10; i++)
            cout << c + i << " " << f + i << endl;
      system("pause");
      return 0;
}
```

Так виглядає варіант виконання проєкту програмного коду на моєму комп'ютері:

```
008FFCD8 008FFCBC
008FFCDC 008FFCC4
008FFCE0 008FFCCC
008FFCE4 008FFCD4
008FFCE8 008FFCDC
008FFCEC 008FFCE4
```

```
008FFCF0 008FFCEC
008FFCF4 008FFCF4
008FFCF8 008FFCFC
008FFCFC 008FFD04
```

Для більшості 32-розрядних компіляторів значення c збільшуватиметься на 4, а значення f – на 8. Відповідно, стільки байтів відводиться для зберігання значень цілого та дійсного типів.

Наступними прикладами проілюструємо деякі особливості виконання операторів з вказівниками.

Працюючи з вказівниками, необхідно відрізняти оператори з самими вказівниками та оператори зі значеннями за адресою, на яку вказує вказівник, наприклад (схожий приклад ми уже розглядали раніше, але у попередньому випадку розіменований вказівник не використовувався як операнд, а також не було оператора з самим вказівником) проєкт програмного коду:

```
#include <iostream>
using namespace std;
int main()
{
	int a = 10;
	int *pa = &a;
	cout << "pa: Adresa = " << pa <<
"\tznachennia pa = " << *pa << endl;
	/* операндом є значення, на яке
	вказує вказівник */
	int b = *pa + 20;
	cout << "b = " << b << endl;
	// операндом є вказівник pa
	pa += 3;
	cout << "pa: Adresa = " << pa << endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду на моєму комп’ютері виглядає так:

```
pa: Adresa = 00EFFBA0   znachennia pa = 10
b = 30
pa: Adresa = 00EFFBAC
```

У цьому разі оператором розіменування вказівника `*pa` отримуємо його значення (це число 10) і виконуємо настанову присвоєння,

де оператором додавання збільшуємо розіменоване значення вказівника (`*pa`) на число 20 і присвоюємо результат змінній `b`:

```
int b = *pa + 20;
```

Це звиклий оператор додавання між двома операндами з огляду на те, що `*pa` є числом.

Настановою

```
pa += 3;
```

ми виконуємо зсув поточної адреси ра на три елементи базового типу вказівника.

Проте, є особливості у використанні операторів інкременту та декременту стосовно вказівників. Річ у тім, що оператор ″*″ (розіменування вказівника) має нижчий пріоритет, ніж оператори інкременту та декременту тому, у поданому прикладі виконаються зліва праворуч.

```
#include <iostream>
using namespace std;
int main()
{
	int a = 10;
	int *pa = &a;
	cout << "pa: Adresa = " << pa <<
"\tznachennia = " << *pa << endl;
// інкремент адреси вказівника
	int b = *pa++;
	cout << "b: znachennia = " << b << endl;
	cout << "pa: adresa = " << pa << "\n*pa:
znachennia = " << *pa << endl;
	system("pause");
	return 0;
}
```

Спочатку настановою

```
b = *pa++;
```

до вказівника додається одиниця (до адреси додається 4 байти, бо вказівник вказує на тип `int`). Тобто, значенням вказівника `pa` уже не є адреса змінної `a`, а адреса зсунута відносно адреси змінної `a` на 4 байти, а за цією адресою розташоване випадкове значення.

Далі, зважаючи на постфіксну форму оператора інкременту, оператором розіменування вказівника отримуємо значення, яке було до виконання оператора інкременту, тобто число 10. Це число присвоюється змінній `b`.

На моєму комп'ютері результат виконання поданого програмного коду буде таким:

```
pa: Adresa = 0077FD74   znachennia = 10
b: znachennia = 10
pa: adresa = 0077FD78
*pa: znachennia = -858993460
```

Змінимо вираз у настанові:

```
b = (*pa)++;
```

Дужки поміняють порядок виконання операторів. Спочатку виконається оператор розіменування вказівника і отримання значення, а потім отримане значення збільшиться на одиницю. Тепер за адресою у вказівнику є число 11. Але оператор інкременту у постфіксній формі, тому змінна b отримує значення, яке було до збільшення, тобто знову число 10. Отже, на відміну від попереднього випадку усі оператори виконуються над значенням за адресою, яке зберігає вказівник, але не над самим вказівником. Зміниться і результат роботи, який на моєму комп'ютері виглядає так:

```
pa: Adresa = 00FCFEC8   znachennia = 10
b: znachennia = 10
pa: adresa = 00FCFEC8
*pa: znachennia = 11
```

Щось схоже буде і з використанням префіксної форми опретора інкременту:

```
b = ++*pa;
```

У цьому разі, спочатку використавши оператор розіменування вказівника *pa, одержуємо значення за адресою з вказівника (це число 10) і до цього значення додається одиниця. Тепер є значення за адресою, яке дорівнює 11 і присвоюється змінній b. Результат виконання програмного коду на моєму комп'ютері виглядає так:

```
pa: Adresa = 00CFFAD0   znachennia = 10
b: znachennia = 11
pa: adresa = 00CFFAD0
*pa: znachennia = 11
```

Знову ж таки змінимо вираз у настанові:

```
b = *++pa;
```

Тепер, спочатку, змінюється адреса у вказівнику, пізніше отримуємо за цією адресою значення і присвоюємо його змінній b. Отримане значення є невизначеним (відбувся зсув адреси змінної a, яка

присвоєна вказівникові `pa` на 4 байти, а за цією адресою розташоване невизначене числове значення). Результат виконання програмного коду на моєму комп'ютері виглядає так:

```
pa: Adresa = 010FFB68   znachennia = 10
b: znachennia = -858993460
pa: adresa = 010FFB6C
*pa: znachennia = -858993460
```

Якщо ви не зрозуміли, в чому полягає суть вказівників, рекомендуємо ще раз уважно опрацювати викладений матеріал. Вказівники – це доволі складне поняття і може знадобитись певний час, зусилля, терпець та отримання практичних навичок, щоб опанувати його.

Контрольні запитання

1. Що таке вказівник?
2. Чому операції, які можуть виконуватися з вказівником, суттєво залежать від його типу?
3. Що визначає оператор отримання адреси?
4. Як отримати значення за вказаною адресою?
5. Що є результатом розіменування вказівника?
6. Які оператори можна використовувати до вказівників?
7. Оголошення та ініціалізування вказівників.

Розділ
2

ВКАЗІВНИКИ І МАСИВИ

Вказівники є надзвичайною і характерною особливістю С++. Сподіваємося, що ви виконали наші рекомендації у кінці першого розділу, тому, освоївши поняття вказівника, у цьому і подальших розділах навчимося використовувати їх для:

– доступу до елементів масиву;

– передавання аргументів у функцію, від якої вимагається можливість змінювати ці аргументи

– передавання у функцію масивів та рядкових змінних;

– відведення динамічної пам'яті;

– створення складних структур.

2.1. Використання вказівників при роботі з масивами

Масиви та вказівники у С++ пов'язані та можуть використовуватись майже еквівалентно. Ім'я масиву можна сприймати як константний вказівник на адресу першого байту початкового елемента масиву. *Відмінність константного вказівника від звичайного вказівника полягає у тому, що його не можна модифікувати*.

У С++ ім'я масиву є постійним вказівником на адресу першого елемента масиву. Отож в оголошенні

```
int mas[5];
```

`mas` – це вказівник на `&mas[0]`, тобто ім'я масиву є вказівником на адресу початкового (нульового) елементу масиву.

Здійснимо оголошення масиву `mas` з п'яти цілих чисел з ініціалізуванням значень елементів і вказівника `ptr` цілого типу:

```
int mas[5] = {10, -2, 0, 40, 3}, *ptr;
```

За такого оголошення масиву пам'ять відводиться не лише для п'яти елементів масиву, але і для вказівника з ім'ям `mas`, значення якого дорівнюватиме адресі першого елемента масиву `mas[0]`, тобто

доступ до елементів масиву здійснюватиметься через вказівник з ім'ям `mas` (рис. 2.1).

Для того, щоб звернутися до 3-го елемента цього масиву, можна записати `mas[2]` або `*(mas+2)`. У процесі реалізування у програмному коді перший з цих способів зводиться до другого, тобто індексний вираз перетвориться до адресного. Оскільки операції над вказівниками виконуються швидше і якщо елементи масиву обробляються почергово, тоді доцільніше використовувати другий спосіб. Якщо ж вибір елементів є випадковим, тоді, аби уникнути помилок, прийнятнішим є перший спосіб (індексний). Крім того, перший спосіб є більш наочним і звичним для сприйняття, що сприяє кращому візуальному сприйняттю програмного коду.

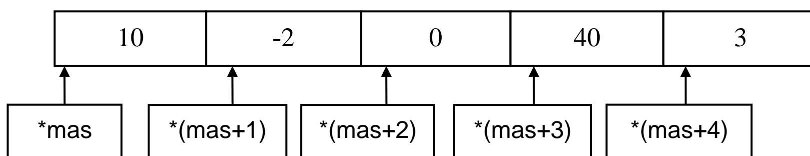

Рис. 2.1. Доступ до елементів масиву
через вказівник з ім'ям `mas`

Оскільки ім'я масиву є вказівником на перший елемент масиву, можна надати вказівнику адресу першого елемента масиву за допомогою настанови:

```
ptr = mas;
```

Цей запис є еквівалентним присвоюванню адреси першого елемента масиву (тобто елемента з нульовим індексом) у вигляді настанови:

```
ptr = &mas[0];
```

Графічно доступ до адрес, де зберігаються значення елементів масиву `mas` через вказівник `ptr` це можна пояснити так (рис. 2.2):

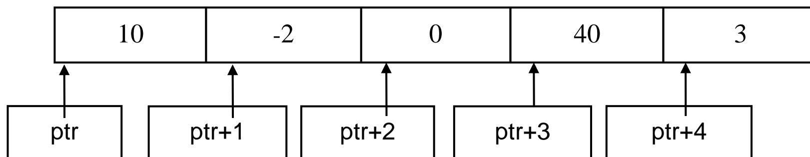

Рис. 2.2. Доступ до адрес, де зберігаються значення
елементів масиву `mas` *через вказівник* `ptr`

Після цього звернутися до першого елемента масиву і надати йому значення 2 можна довільною з шести настанов:

```
*mas = 2;
mas[0] = 2;
*(mas+0) = 2;
*ptr = 2;
ptr[0] = 2;
*(ptr+0) = 2;
```

Усі ці настанови за дією є тотожними, але найшвидше виконуватимуться присвоювання `*mas = 2` та `*ptr = 2`, оскільки у них не потрібно виконувати додавання.

Вказівники можна індексувати так само, як і масиви. Наприклад, `ptr[3]` посилається до четвертого елемента масиву `mas[3]`.

Зауважимо, що не варто плутати такі оголошення:

```
int *p1[10];    // приклад 1
int (*p2)[10]; // приклад 2
```

У першому прикладі оголошено масив вказівників з ім'ям `p1`. Масив складається з 10-ти елементів, кожний з яких є вказівником на змінну типу `int`. У другому прикладі оголошено змінну-вказівник з ім'ям `p2`, яка вказує на масив з 10-ти цілих чисел типу `int`.

Можна використовувати імена масивів як константні вказівники і навпаки, наприклад:

```
int a[5] = {1, 2, 4, 8, 16};
int *ptr = a;
cout << *a << endl;         // 1
cout << ptr[2] << endl;   // 4
```

Як ми уже знаємо, до вказівників можна застосувати оператори додавання, віднімання і порівняння. Під час виконання арифметичних операторів з вказівниками передбачається, що вказівники вказують на масив об'єктів. Додавши ціле значення до вказівника, ми переміщуємо його на відповідне число об'єктів у масиві. Якщо, наприклад, тип має розмір 10 байт, а потім ми додали ціле число 5, вказівник переміщується на 50 байт у пам'яті.

Можна застосувати оператори інкременту і декременту, а також оператори компаундного присвоювання для неконстантних вказівників.

Подамо приклад проєкту програмного коду, проаналізувавши який, ви зрозумієте викладений вище матеріал.

Проєкт програмного коду:

```
#include <iostream>
using namespace std;
int main()
{
int a[5] = { 1, 2, 4, 8, 16 };
int *ptr = a;
// ptr вказує на адресу a[0]
cout << "Vyvid znachennia a[3] " << *(a + 3) <<
endl;
// 8 (виведення значення a[3])
ptr++;
// ptr вказує на адресу a[1]
cout << "Vyvid adresy a[1] (ptr) " << ptr << endl;
cout << "Vyvid adresy a[1] (&a[1]) " << &a[1] <<
endl;
/*виведення адреси a[1] через оператор отримання
адреси*/
cout << "Vyvid znachennia a[1] (*ptr) " << *ptr <<
endl;
// 2 (виведення значення a[1])
ptr += 3;
// ptr вказує на адресу a[4]
cout << "Vyvid adresy a[4] (ptr) " << ptr << endl;
// виведення адреси a[4]
cout << "Vyvid znachennia a[4] (*ptr) " << *ptr <<
endl;
// 16 (виведення значення a[4])
//a++;
// Помилка! a є константним вказівником
system("pause");
return 0;
}
```

Отриманий результат на моєму комп’ютері виглядає так:

```
Vyvid znachennia a[3] 8
Vyvid adresy a[1] (ptr) 00CFFE80
Vyvid adresy a[1] (&a[1]) 00CFFE80
Vyvid znachennia a[1] (*ptr) 2
Vyvid adresy a[4] (ptr) 00CFFE8C
Vyvid znachennia a[4] (*ptr) 16
```

Різниця між двома вказівниками на різні елементи масиву повертає кількість елементів, які розташовані між цими вказівниками

(включаючи перший і не включаючи останній). Наприклад, фрагмент програмного коду:

```
int a[5] = { 1, 2, 4, 8, 16 };
int *ptr1 = a;
// ptr1 вказує на a[0] (*ptr1 = &a[0])
int *ptr2 = a + 3;        // ptr2 вказує на a[3]
cout << ptr2 - ptr1;      // 3
```

Нагадуємо, різниця між двома вказівниками має сенс тільки тоді, коли обидва вказівники вказують на елементи одного масиву.

Перевірка виходу за межі масиву не здійснюється. У поданому далі прикладі фрагменту програмного коду `ptr3` вказує на неіснуючий елемент:

```
int a[5] = {1, 2, 4, 8, 16};
int *ptr3 = a + 5;      // Такого елемента не існує
```

2.2. Схожість між вказівниками і масивами

Схожість між вказівниками і масивами продемонструємо на прикладах з детальним аналізом.

Фіксований масив оголошується та ініціалізується таким способом:

```
int array[5] = {5, 8, 6, 4, 16}; // масив містить 5
                                 // цілих чисел
```

Для нас це масив з 5 цілих чисел, але для компілятора `array` є змінною типу `int[5]`. Ми знаємо, що

```
array[0] = 5;
array[1] = 8;
array[2] = 6;
array[3] = 4;
array[4] = 16;
```

Але яке значення має сам `array`? Розглянемо приклад програмного коду.

```
#include <iostream>
using namespace std;
int main()
{
      int array[5] = { 5, 8, 6, 4, 16 };
      /* Виводимо на екран значення імені масиву
(змінну array)*/
```

```
      cout << "array maje znachennia : " << array
<< '\n';
      /* Виводимо на екран адресу першого елемента
масиву*/
      cout << "Pershyj element array maje adresu :
" << &array[0] << '\n';
      system("pause");
      return 0;
}
```

Результат на моєму комп’ютері:

```
array maje znachennia : 00EFF704
Pershyj element array maje adresu : 00EFF704
```

Зауважте, адреса, яка зберігається у змінній `array`, є адресою першого елемента масиву.

Поширена помилка думати, що змінна `array` і вказівник на `array` є одним і тим же об’єктом. Це не так. Хоча обидва вказують на перший елемент масиву, *інформація про тип даних у них різна*. У поданому вище прикладі типом змінної `array` є `int[5]`, тоді як типом вказівника на масив є `int *`.

Плутанина зумовлена тим, що в багатьох випадках, при обчисленні, фіксований масив розпадається (неявно перетворюється) у вказівник на перший елемент масиву.

Доступ до елементів як і раніше здійснюється через вказівник, але інформація, отримана з типу масиву (наприклад, його розмір), не може бути доступна з типу вказівника.

Однак і це не є настільки вагомим аргументом, щоб розглядати фіксовані масиви і вказівники як різні значення. Наприклад, ми можемо розіменувати масив, щоб отримати значення першого елемента:

```
#include <iostream>
using namespace std;
int main()
{
     int array[5] = { 5, 8, 6, 4, 16 };
/* Розіменування масиву (змінної array) призведе до
виведення значення першого елемента масиву (елемен-
та з індексом 0)*/
     cout << "Rezultat rozimenuvannia masyvu
(zminnoji array): = " << *array;
     // виведеться 5!
     system("pause");
```

```
    return 0;
}
```

Результат виконання програмного коду

```
Rezultat rozimenuvannia masyvu (zminnoji array): = 5
```

Зосередьтесь на тому, що ми не розіменовуємо фактичний масив. Масив (типу `int[5]`) неявно конвертується у вказівник (типу `int *`), і ми розіменовуємо вказівник, який вказує на значення першого елемента масиву.

Також ми можемо створити вказівник і присвоїти йому `array`, наприклад:

```
#include <iostream>
using namespace std;
int main()
{
      int array[5] = { 5, 8, 6, 4, 16 };
      cout << "\nRezultat rozimenuvannia masyvu
*array: " << *array;
// виведеться 5
      int *ptr = array;
      cout << "\nRezultat rozimenuvannia vkazivnyka
*ptr: " << *ptr << endl;
// виведеться значення першого елементу масиву, 5
      system("pause");
      return 0;
}
```

Результат виконання програмного коду

```
Rezultat rozimenuvannia masyvu *array: 5
Rezultat rozimenuvannia vkazivnyka *ptr: 5
```

Це працює через те, що змінна `array` розпадається у вказівник типу `int *`, а тип нашого вказівника такий же (`int *`).

2.3. Відмінності між вказівниками і масивами

Якщо, наприклад, `par` – вказівник, тоді у виразах його можна використовувати з індексом, тобто `par[i]`. Але між ім’ям масиву і вказівником, який має значення адреси масиву, є істотне розходження.

Вказівник – це змінна, тому можна записати такі настанови:

```
int array[5];
int *par = array;
par++;
```

А ім’я масиву – це константний вказівник, тобто використання настанов

```
int array[5], mas[5];
int *par = array;\
```

є коректним, а настанов

```
mas = par;      // Помилка!
mas++;          // Помилка!
```

помилковим.

Потрібно також розрізняти вирази `*(array + 2)` і `*array + 2`: перший повертає значення третього елемента масиву `array`, а другий – це додавання числа 2 до значення першого елемента масиву.

Однак є випадки, коли різниця між фіксованими масивами і вказівниками має значення. Основна відмінність виникає при використанні оператора `sizeof`. При використанні в фіксованому масиві, оператор `sizeof` повертає розмір всього масиву (`довжина_масиву * розмір_елементу`). При використанні з вказівником, оператор `sizeof` повертає розмір адреси в пам’яті (в байтах). Наприклад:

```
#include <iostream>
using namespace std;
int main()
{
      int array[5] = { 5, 8, 6, 4, 16 };
      cout << "\nRozmir masyvu array (bajty): " <<
sizeof(array);
// виведеться sizeof(int) * довжина array *
      int *ptr = array;
      cout << "\nRozmir adresy vkazivnyka (bajty):
" << sizeof(ptr) << '\n';
// виведеться розмір вказівника
system("pause");
return 0;
}
```

Результат виконання програмного коду на моєму комп’ютері:

```
Rozmir masyvu array (bajty): 20
Rozmir adresy vkazivnyka (bajty): 4
```

Фіксований масив знає свою довжину, а вказівник на масив – ні.

Інша відмінність виникає при використанні оператора отримання адреси `&`. Використовуючи адресу вказівника, ми отримуємо адресу пам'яті змінної вказівника. Використовуючи адресу масиву, повертається вказівник на весь масив. Цей вказівник також вказує на перший елемент масиву, але інформація про тип відрізняється.

Розглянемо кілька програмних кодів для розв'язання задачі обчислення середнього значення додатних елементів одновимірного масиву `array`, який складається не більше як з 10 елементів дійсного типу.

```
// Example program 1
#include <iostream>
using namespace std;
int main()
{
	int const n = 8;
	float array[n], Seredne, sd = .0;
	int kd = 0;
	for (int i = 0; i < n; i++)
	{
		cout << "Vvesty " << i + 1 << " element
array" << endl;
		cin >> *(array + i);
	}
	for (int i = 0; i < n; i++)
		if (array[i] > 0)
		{
			sd += array[i];
	//накопичення суми
			kd++;
	//накопичення кількості додатних елементів
		}
	Seredne = sd / kd;
	cout << "Suma dod = " << sd << endl;
	cout << "Kil'kist' dod = " << kd << endl;
	cout << "Seredne = " << Seredne << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду на моєму комп'ютері:

```
Suma dod = 10
Kil'kist' dod = 4
Seredne = 2.5
```

Використовуючи ім’я масиву як вказівник на початок масиву (перший елемент), можна подати другий варіант проєкту програмного коду:

```
// Example program_2
#include <iostream>
using namespace std;
int main()
{
    int const n = 8;
    float array[n], Seredne, sd = .0;
    int i, kd = 0;
    for (i = 0; i < n; i++)
    {
        cout << "Vvesty " << i + 1 << " element
array" << endl;
        cin >> *(array + i);
        if (*(array + i) > 0)
        {
            sd += *(array + i);
            kd++;
        }
    }
    Seredne = sd / kd;
    cout << "Suma dod = " << sd << endl;
    cout << "Kil'kist' dod = " << kd << endl;
    cout << "Seredne = " << Seredne << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Suma dod = 10
Kil'kist' dod = 4
Seredne = 2.5
```

Якщо описати вказівник і зв’язати його з масивом (адресувати на початок масиву), тоді з використанням арифметики вказівників можна написати третій варіант проєкту програмного коду:

```
// Example program_3
#include <iostream>
using namespace std;
int main()
{
```

```
    int const n = 8;
    float array[n], sd = 0., Seredne;
    float* par = &array[0];
    int i, kd = 0;
    for (i = 0; i < n; i++)
    {
        cout << "Vvesty " << i + 1 << " element
array" << endl;
        cin >> *par++;
        if (array[i] > 0)
        {
            sd += array[i];
            kd++;
        }
    }
    Seredne = sd / kd;
    cout << "Suma dod = " << sd << endl;
    cout << "Kil'kist' dod = " << kd << endl;
    cout << "Seredne = " << Seredne << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Suma dod = 10
Kil'kist' dod = 4
Seredne = 2.5
```

У проєкті поданого програмного коду для введення масиву застосований вказівник `*par`, а для роботи з масивом – ім'я масиву з індексом `array[i]`.

В останньому випадку використання вказівника `*par` *призвело б до помилкового результату, оскільки цей вказівник збільшував би свою адресу* (`par++`) *після введення поточного елемента масиву і надалі вказував на ще не введений елемент.*

Подамо четвертий варіант проєкту програмного реалізування задачі:

```
// Example program 4
#include <iostream>
using namespace std;
int main()
{
     int const n = 8;
```

```
	float array[n], sd = 0, Seredne;
	float *par = &array[0];
	int i, kd = 0;
	for (i = 0; i < n; i++)
	{
		cout << "Vvesty " << i + 1 << " element
array" << endl;
		cin >> *par;
		if (*par > 0)
		{
			sd += *par;
			kd++;
		}
		par++;
	}
	Seredne = sd / kd;
	cout << "Suma dod = " << sd << endl;
	cout << "Kil'kist' dod = " << kd << endl;
	cout << "Seredne = " << Seredne << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Suma dod = 10
Kil'kist' dod = 4
Seredne = 2.5
```

Під час введення значення поточного елемента масиву у настанові умовного переходу і компаундного присвоєння користуємося розіменованим значенням, а оператором `par++`, який є останнім у тілі циклу, змінюємо адресу поточного елемента на адресу наступного.

Ми свідомо подали результати виконання обчислень. Пропонуємо вам виконати такі ж обчислення з однаковими початковими даними. Результати будуть однаковими.

З пункту бачення користувача ви не зауважите відмінностей у функціонуванні програмного коду, хоча з пункту бачення програміста різниця є суттєвою. Ці приклади у подальшому можна використати як стандартні підходи до впорядкування елементів одновимірних масивів.

З метою засвоєння викладеного матеріалу розглянемо ще два способи обчислення суми елементів масиву `array` цілого типу з використанням вказівника `ptr` та індексуванням для доступу до елементів

масиву (зміщенням адреси до і-го елемента масиву). Для отримання значення елемента масиву через вказівник `ptr` необхідно використати оператор розіменування.

```
#include <iostream>
using namespace std;
int main()
{
	int i;
	int const n = 8;
	int array[n];
	for (i = 0; i < n; i++)
	{
		cout << "Vvesty " << i + 1 << " element 
array" << endl;
		cin >> *(array + i);
	}
	cout << "Vvedenyj masyv array" << endl;
	for (i = 0; i < n; i++)
		cout << array[i] << "   ";
	int *ptr = array;
	// або ptr = &array[0]
	// 1 варіант: через вказівник ptr
	int sum1 = 0;
	for (ptr = array; ptr < &array[n]; ++ptr)
		sum1 += *ptr;
	// 2 варіант: з використанням індексів
	int sum2 = 0;
	for (i = 0; i < n; ++i)
		sum2 += array[i];
	// аналог sum2 += *(array + i);
	cout << "\nSuma elementiv sum1 = " << sum1 << 
endl;
	cout << "Suma elementiv sum2 = " << sum2 << 
endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Vvedenyj masyv array
1   -2   3   -5   -6   1   2   3
Suma elementiv sum1 = -3
Suma elementiv sum2 = -3
```

Пояснення стосовно ходу виконання програмного коду подано у коментарях.

2.4. Вказівники на багатовимірні масиви

Вказівники на багатовимірні масиви у мові С++ – це масиви масивів, тобто такі масиви, елементами яких є масиви. При оголошенні таких масивів у пам'яті комп'ютера створюється декілька різних об'єктів.

Приміром, при виконанні оголошення двовимірного масиву

```
int arr[4][3];
```

в оперативній пам'яті відводиться ділянка для зберігання значення змінної `arr`, яка є вказівником на масив з чотирьох вказівників. Для цього масиву з чотирьох вказівників теж відводиться пам'ять. Кожний з цих чотирьох вказівників містить адресу масиву з трьох елементів типу `int`, і, отже, у пам'яті комп'ютера відводиться чотири ділянки для зберігання чотирьох масивів чисел типу `int`, кожна з яких складається з трьох елементів. Такий розподіл пам'яті показано на схемі:

```
    arr↓
 arr[0]    →    arr[0][0]   arr[0][1]    arr[0][2]
 arr[1]    →    arr[1][0]   arr[1][1]    arr[1][2]
 arr[2]    →    arr[2][0]   arr[2][1]    arr[2][2]
 arr[3]    →    arr[3][0]   arr[3][1]    arr[3][2]
```

Отже, оголошення `arr[4][3]` створює у програмі три різних об'єкти: вказівник з ідентифікатором `arr`, безіменний масив з чотирьох вказівників і безіменний масив з дванадцяти чисел типу `int`. Для доступу до безіменних масивів використовують адресні вирази з вказівником `arr`. Доступ до елементів масиву вказівників здійснюється із зазначенням одного індексного виразу у формі `arr[2]` або `*(arr+2)`.

Для доступу до елементів двовимірного масиву чисел типу `int` має бути використано два індексні вирази у формі `arr[1][2]` або еквівалентних їй `*(*(arr+1)+2)` та `(*(arr+1))[2]`. Нагадаємо, що елементи двовимірних масивів розміщуються у пам'яті поряд і між його рядками немає ніяких проміжків. Такий порядок дає можливість звертатис/я до довільного елемента багатовимірного масиву, викорис-

товуючи адресу його початкового елемента та лише один індексний вираз.

Для двовимірних масивів ім'я є вказівником-константою на масив вказівників-констант. Елементами масиву вказівників є вказівники-константи на початок кожного з рядків масиву: тому, при використанні вказівників "точкою відліку" може бути як найперший елемент масиву, так і перший елемент кожного з рядків, тобто можуть використовуватися як вказівник-константа, що задається ім'ям масиву, так і вказівники на рядки масиву.

Отже, для двовимірного масиву `arr` звертання `*(arr)` є посиланням на елемент `arr[0][0]`, звертання `*(arr+2)` – на елемент `arr[0][2]`, а звертання `*(arr+3)` – на елемент `arr[1][0]` тощо. Тобто, застосовуючи настанови циклу, можна організувати поелементне опрацювання елементів матриці, наприклад, введення усієї матриці де "точкою відліку" є перший елемент матриці:

```
for(int i=0; i<4; i++)
for(int j=0; j<3; j++)
cin >> *(arr+i*4+j);
```

Зауважимо, що вираз `*(arr+i*4+j)` є аналогічним до `arr[i][j]`.

Виведення у консольний додаток значення вказівника на елемент масиву з i-го рядка та j-го стовпця, що демонструє поданий проєкт програмного коду:

```
#include <iostream>
using namespace std;
int main()
{
      int const n = 3, m = 4; // розмір масиву
      /* оголошення й ініціалізування двовимірного
масиву*/
      int matr[n][m] = { {1,2,3,4}, {5,6,7,8},
{9,10,11,12} };
      for (int i = 0; i < n; i++)
      {
      // вивід на екран номер рядка
            cout << "\ni = " << i << "    ";
      // для всіх номерів стовпців j від 0 до m
            for (int j = 0; j < m; j++)
      /* вивід на екран значення вказівника на еле-
мент масиву з i-го рядка та j-го стовпця з i-го
рядка та j-го стовпця */
```

```
            cout << *(matr + i) + j << "\t";
                        cout << "   "; ;
      }
      system("pause");
      return 0;
}
```

Результат виконання програмного коду на моєму комп’ютері:

```
i = 0   007BFB54   007BFB58   007BFB5C   007BFB60
i = 1   007BFB64   007BFB68   007BFB6C   007BFB70
i = 2   007BFB74   007BFB78   007BFB7C   007BFB80
```

Розіменування вказівників на елементи двовимірного масиву дає змогу одержати значення самих елементів, наприклад, проєкт програмного коду:

```
#include <iostream>
using namespace std;
int main()
{
      int const n = 3, m = 4; // розмір масиву
     // оголошення й ініціалізація двовимірного
масиву
     int matr[n][m] = { {1,2,3,4}, {5,6,7,8},
{9,10,11,12} };
     for (int i = 0; i < n; i++)
     {// виведення на екран номер рядка
     cout << "\ni = " << i << "   ";
     // для всіх номерів стовпців j від 0 до m
     for (int j = 0; j < m; j++)
     /* виведення на екран значення вказівника на
елемент масиву
     з i рядка й j стовпця */
     cout << *(*(matr + i) + j) << "   ";
     cout << "\t"; ;
     }
     system("pause");
     return 0;
}
```

Результат виконання програмного коду:

```
i = 0   1   2   3   4
i = 1   5   6    7   8
i = 2   9   10   11   12
```

Виведення елементів матриці можна здійснити з використанням вказівника. Вказівник на кожний окремий елемент масиву розіменовується, а також виконується контроль порядкового номера елемента у рядку. Якщо порядковий номер елемента (через значення вказівника) кратний m, тоді здійснюється перехід на новий рядок виводу.

```
#include <iostream>
using namespace std;
int main()
{
	int const n = 3, m = 4; // розмір масиву
	/* оголошення та ініціалізування двовимірного масиву*/
int matr[n][m] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};	int* ptr = &matr[0][0];
	/* виведення на екран розіменованого значення вказівника на елемент масиву */
	cout << "Vyvid elementiv masyvu\n";
	for (int i = 0; i < n * m; i++)
	{
		cout << *ptr++ << '\t';
		if ((i + 1) % m == 0)
			cout << "\n";
	}
	system("pause");
	return 0;
}
```

Результат виконання програмного коду на моєму комп'ютері:

```
Vyvid elementiv masyvu
1	2	3	4
5	6	7	8
9	10	11	12
```

Контрольні запитання

1. У чому схожість між вказівниками і масивами?
2. У чому полягають відмінності між вказівниками і масивами?
3. Вказівники на багатовимірні масиви?
4. Розіменування вказівників на елементи двовимірного масиву.
5. Як реалізувати виведення елементів двовимірного масиву з використанням вказівника?

Розділ 3 ДИНАМІЧНІ МАСИВИ

Вказівники використовуються в С++ дуже інтенсивно і не тільки для роботи з масивами. Одне з основних застосувань вказівників полягає у роботі з динамічно створеними об'єктами, зокрема з динамічними масивами.

Властивості вказівників дають змогу однаковим чином оперувати і динамічними масивами, і масивами фіксованої розмірності.

Терміни статичний/динамічний характеризують зміну властивостей об'єкта у процесі виконання програмного коду. Якщо ці властивості не змінюються (жорстко задаються у процесі транслювання), тоді об'єкти статичні, якщо змінюються – динамічні. Це стосується й існування самих об'єктів.

Статичний об'єкт створюється у процесі транслювання програмного коду, а *динамічний* – у процесі виконання програмного коду.

Щодо змінних це виглядатиме так: якщо змінна створюється у процесі транслювання (створення змінної насамперед пов'язано з розподілом пам'яті для неї), тоді її можна називати статичною, якщо ж створюється у процесі виконання програми – тоді динамічною. З цього пункту бачення усі іменовані змінні є статичними.

Головним недоліком статичних змінних є їхня фіксована розмірність, яка визначається при транслюванні програмного коду.

На рівні бібліотек у С++ наявний механізм створення і видалення змінних виконуваним програмним кодом. Такі змінні називаються динамічними, а область пам'яті, у якій вони створюються, – динамічною.

Найважливішою властивістю динамічних змінних є їх ″безіменність″ (доступ до них здійснюється за вказівником), що визначає можливість змінювати кількість таких змінних у програмних кодах.

У програмі кожна змінна може розміщуватися в одному із трьох місць: в області даних програми, у стеку або у вільній пам'яті (*heap* – так звана купа). Кожній змінній в програмі пам'ять може відводитися або статично, тобто в момент завантаження програми, або динамічно – у процесі виконання програми. Досі обумовлені масиви оголошувалися

статично, і, отже, зберігали значення всіх своїх елементів у стековій пам'яті та/або області даних програми. Якщо кількість елементів масиву невелика, то таке розміщення виправдане.

Однак доволі часто виникають випадки, коли у стековій пам'яті, яка містить локальні змінні та допоміжну інформацію (наприклад, точки повернення із вкладених функцій), недостатньо місця для розміщення всіх елементів великого масиву. Ситуація ще погіршується, якщо масивів великого розміру повинно бути багато. Тут на допомогу приходить можливість використання динамічної пам'яті для зберігання даних.

В основній пам'яті дані можуть зберігатися двома способами. Пам'ять відводиться у сегменті стека, де автоматично розміщуються змінні за їхнього оголошення і залишається закріпленою до завершення виконання конкретної функції – *статичне відведення пам'яті*. Пам'ять відводиться у сегменті даних на весь час виконання програми. Пам'ять відводиться у міру потреби – *динамічне відведення пам'яті*, в якій змінні можуть розміщуватися динамічно. Це означає, що пам'ять відводиться під час виконання програми, і лише тоді, коли у програмі трапиться спеціальна настанова. Основна потреба у динамічному виділенні пам'яті виникає, коли розмір або кількість даних заздалегідь є невідомі, а визначаються в процесі виконання програмного коду (рис. 3.1).

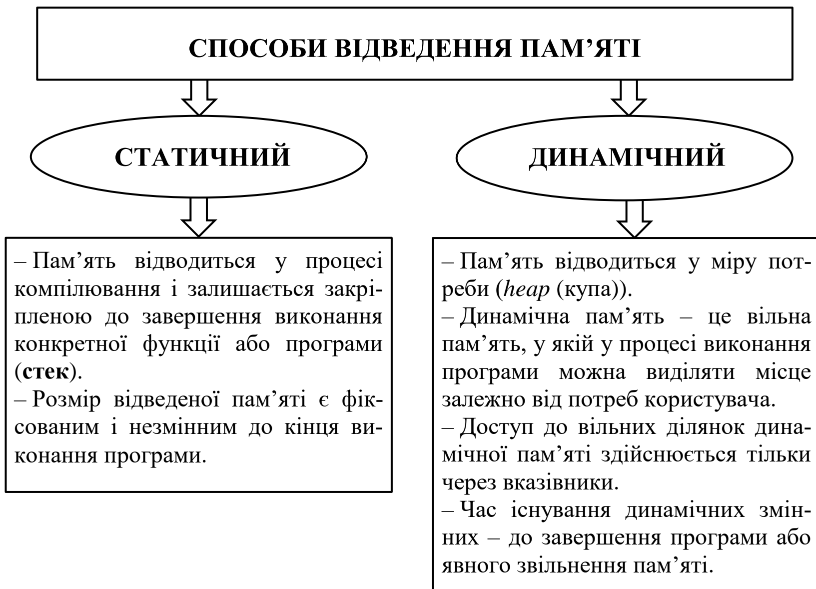

Рис. 3.1. Способи відведення пам'яті

Динамічна пам’ять – це вільна пам’ять, у якій під час виконання програми можна виокремити місце залежно від потреб користувача. Доступ до відведених ділянок динамічної пам’яті, які називаються динамічними змінними, здійснюється тільки через вказівники. Час існування динамічних змінних – від початку створення до кінця програми або до явного вивільнення пам’яті.

Зауважимо, що всі приклади, розглянуті раніше, демонструють роботу з даними, які зберігаються першим способом.

Як уже йшлося, робота з динамічними змінними пов’язана з використанням вказівників. Вказівник містить адресу поля в динамічній пам’яті, де зберігається змінна певного типу. Сам вказівник розташовується у статичній пам’яті. Адреса змінної – це адреса першого байта поля пам’яті, в якому розташовується змінна. Розмір поля пам’яті визначається типом.

Прикладом статичного відведення пам’яті може слугувати уже відоме вам оголошення масиву із 10 цілих чисел:

```
int array[10]; /* пам’ять для масиву відводиться
один раз, розмір пам’яті фіксований */
```

Підсумовуючи викладене, можемо вважати, що у процесі запуску програмного коду на виконання операційна система налаштовує різні області пам’яті:

– глобальні змінні знаходяться у *глобальному просторі імен*, для якого відводиться спеціальний сегмент пам’яті;

– регістри утворюють особливу область пам’яті, вмонтовану в центральний процесор;

– стек (s*tack*) – це спеціальна область пам’яті, відведена для зберігання даних окремих функцій (ознакою стеків як структур даних є принцип *last-in, first-out* (LIFO, першим зайшов, останнім вийшов);

– решта пам’яті, розподіленої для програми, – це *динамічна пам’ять*.

Програмісти можуть використовувати динамічну пам’ять для контрольованого відведення пам’яті та вивільнення змінних з пам’яті.

3.1. Відведення та вивільнення динамічної пам’яті

У С++ для відведення та вивільнення динамічної пам’яті використовуються оператори **new** та **delete**. Ці оператори призначені для керування вільною пам’яттю.

Формат синтаксичної конструкції для реалізування оператора **new** записується як:

<тип> *<ім'я_вказівника> = new <тип>;

Оператор **new** виконує дві дії. Оголошується змінна-вказівник визначеного типу і далі вказівнику надається адреса відведеної області пам'яті відповідно до заданого типу об'єкта.

Інакше кажучи, оператор **new** створює об'єкт заданого типу, відводить йому пам'ять і повертає вказівник відповідного типу на цю ділянку пам'яті.

Потрібно пам'ятати, що динамічні масиви при створенні не можна ні ініціалізовувати, ні обнулювати.

Розглянемо кілька прикладів.

Приклад 1.

```
int *p = new int;
```

Відводиться місце в пам'яті під число цілого типу і адреса цієї ділянки пам'яті записується у змінну-вказівник `p`. Звернення через вказівник:

```
*p = 2;
```

Приклад 2.

```
int *p = new int (5);
```

Відводиться місце у пам'яті під число цілого типу і записується до цієї ділянки пам'яті значення 5. Адреса першої комірки відведеної ділянки пам'яті присвоюється змінній-вказівнику `p` . Звернення через вказівник:

```
(*p) += 2;
```

Приклад 3.

```
int *a = new int [10];
```

Створення *динамічного масиву* з 10-ти елементів. Відводиться пам'ять для 10-ти цілих чисел. Звернутися до кожного з цих чисел можна за його номером: `a[0]`, `a[1]`, … , або через вказівник: `*a` – теж саме, що і `a[0]`; `*(a+1)` – те ж саме, що і `a[1]`.

За допомогою оператора **delete** відбувається вивільнення пам'яті, на яку вказує змінна-вказівник. Загальний формат синтаксичної конструкції реалізування оператора **delete**:

delete []<ім'я_вказівника>;

Квадратні дужки повинні бути порожніми, операційна система контролює обсяг відведеної пам'яті і при вивільненні їй відома необхідна кількість байтів.

Наприклад, вивільнення динамічної пам'яті, відведеної оператором **new**, реалізовано оператором **delete**:

```
delete p;
delete []a;
```

Перевагами статичного способу відведення пам'яті є:

– статичне (фіксоване) відведення пам'яті краще використовувати, коли розмір масиву інформації наперед відомий і є незмінним протягом виконання усієї програми;

– статичне відведення пам'яті не потребує додаткових операцій вивільнення з допомогою оператора **delete**. Звідси випливає зменшення помилок програмування. Кожнму операторові **new** має відповідати оператор **delete**;

– природність (натуральність) подання програмного коду, що оперує статичними масивами.

Динамічне відведення пам'яті, порівняно зі статичним відведенням пам'яті, дає такі переваги:

– пам'ять відводиться в міру необхідності програмним шляхом;

– немає зайвих витрат невикористаної пам'яті. Відводиться стільки пам'яті, скільки потрібно і коли потрібно;

– можна відводити пам'ять для масивів інформації, розмір яких наперед невідомий. Визначення розміру масиву формується у процесі виконання програми;

– зручно здійснювати перерозподіл пам'яті. Або, інакше кажучи, зручно відводити новий фрагмент для того самого масиву, якщо потрібно відвести додаткову пам'ять або вивільнити непотрібну;

– при статичному способі відведення пам'яті важко перерозподіляти пам'ять для змінної-масиву, оскільки вона вже відведена фіксовано. У випадку динамічного відведення, це здійснюється легко і зручно.

3.2. Вказівник на вказівник

Вказівник на вказівник типу **int** оголошується з використанням двох зірочок:

```
int **pptr;
```

Вказівник на вказівник працює як звичайний вказівник. Ви можете його розіменувати для отримання значення, на яке він вказує, і, оскільки цим значенням є інший вказівник, для отримання початкового значення вам потрібно буде виконати розіменування ще раз. Їх потрібно виконувати послідовно.

Наприклад, розглянемо проєкт програмного коду використання вказівника на вказівник:

```
#include <iostream>
using namespace std;
int main()
{
      int *ptr;
      int **pptr;
      int var = 10;
      ptr = &var;
      cout << "Znachennia ptr = " << *ptr << endl;
      pptr = &ptr;
      cout << "Znachennia pptr = " << **pptr <<
endl;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
Znachennia ptr = 10
Znachennia pptr = 10
```

Наголошуємо, *заборонено* ініціалізувати вказівник на вказівник безпосередньо значенням:

```
int var = 10;
int **pptr = &&var; // заборонено!
```

Це пов'язано з тим, що оператор отримання адреси (&) вимагає `l-value`, а &`value` – це `r-value`.

Поширеним застосуванням вказівників на вказівники є динамічне відведення багатовимірних масивів і про це детальна розмова ітиме далі.

3.3. Динамічні одновимірні та двовимірні масиви

Відмінності динамічного масиву від звичайного:

– пам'ять під динамічний масив відводиться динамічно (за допомогою оператора **`new`**);

– кількість елементів динамічного масиву може бути задано змінною (у програмному коді її неодмінно має бути визначено до відведення пам'яті під масив).

Приклад оголошення динамічного масиву дійсного типу зі змінною кількістю елементів:

```
int n;
cin >> n;
float *a = new float[n];
```

Тут кількість елементів масиву є змінною і вводиться з клавіатури перед оголошенням масиву. Вивільнення пам’яті від цього масиву матиме вигляд:

```
delete []a;
```

Приклад 3.1. Згенерувати масив цілих чисел заданого розміру і створити на основі нього масив ненульових елементів.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
      int n, j = 0, k = 0;
      /*змінна k для кількості ненульових елементів
масиву */
      cout << "Vvedit' kilkist' elementiv masyvu a:
n = ";
      cin >> n;
      /*відведення пам'яті під динамічний масив */
      int *a = new int[n];
            cout << "Pochatkovyj masyv a:\n";
      for (int i = 0; i < n; i++)
      {
            a[i] = rand() % 5;
            cout << a[i] << "  ";
            if (a[i] != 0) k++;
      }
      cout << endl;
      int *b = new int[k];
      cout << "Masyv nenuliovyh elementiv b:\n";
      for (int i = 0; i < n; i++)
      //записування ненульових значень до масиву
            if (a[i] != 0)
            {
                  b[j] = a[i];
                  cout << b[j] << "  ";
                  j++;
```

```
		}
	cout << endl;
	//вивільнення пам'яті від масивів
	delete []a;
	delete []b;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду

```
Vvedit' kilkist' elementiv masyvu a: n = 6
Pochatkovyj masyv a:
1  2  4  0  4  4
Masyv nenuliovyh elementiv b:
1  2  4  4  4
```

Очевидно, що у поданому прикладі розмірність масиву b заздалегідь невідома, це залежить від структури початкового масиву a, тобто наявності у ньому ненульових елементів та їх кількості, яка і буде розмірністю динамічного масиву b.

Приклад 3.2. Ввести масив з 10-ти цілих чисел і створити з нього два нових масиви: перший масив з непарних елементів, а другий – з парних.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
	int const n = 10;
	int a[n], k = 0, j1 = 0, j2 = 0;
	cout << "Vvedit’ pochatkovyj masyv:" << endl;
	for (int i = 0; i < n; i++)
	{
		cin >> a[i];
		if (a[i] % 2 == 0) k++;
	}
	cout << "Pochatkovyj masyv a:\n";
	for (int i = 0; i < n; i++)
		cout << a[i] << " ";
	cout << endl;
	int *b1 = new int[k];
	int *b2 = new int[n - k];
	for (int i = 0; i < n; i++)
```

```
	{
		if (a[i] % 2 == 0)
			b1[j1++] = a[i];
		else
			b2[j2++] = a[i];
	}
	cout << "Masyv parnyh elementiv b1:\n";
	for (int i = 0; i < k; i++)
		cout << b1[i] << " ";
	cout << endl;
	cout << "Masyv neparnyh elementiv b2:\n";
	for (int i = 0; i < n - k; i++)
		cout << b2[i] << " ";
	cout << endl;
	delete []b1;
	delete []b2;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду

```
Vvedit' pochatkovyj masyv:
1 2 3 4 5 6 7 8 9 10
Pochatkovyj masyv a:
1 2 3 4 5 6 7 8 9 10
Masyv parnyh elementiv b1:
2 4 6 8 10
Masyv neparnyh elementiv b2:
1 3 5 7 9
```

Як і у попередньому прикладі розмірності масивів `b1` та `b2` попередньо невідомі і обчислюються у ході виконання програмного коду, властиво тому масиви `b1` та `b2` оголошені як динамічні. Після завершення обчислень ці масиви видаляються з пам'яті.

Для створення двовимірного масиву використаємо оператор вказівник на вказівник.

Відведення пам'яті для динамічного двовимірного масиву трохи відрізняється від відведення пам'яті для фіксованого масиву.

Створити двовимірний масив `a` цілих чисел розмірністю `n x m`. Спочатку ми створюємо масив вказівників `**a`, а потім перебираємо кожен елемент масиву вказівників і створюємо динамічний масив для кожного елемента цього масиву `a[i]` (рис. 3.2.). У підсумку наш створений динамічний двовимірний масив – це *динамічний одновимірний масив динамічних одновимірних масивів*!

```
int **a = new int*[n];
for (int i = 0; i < n; i++)
  a[i] = new int[m];
```

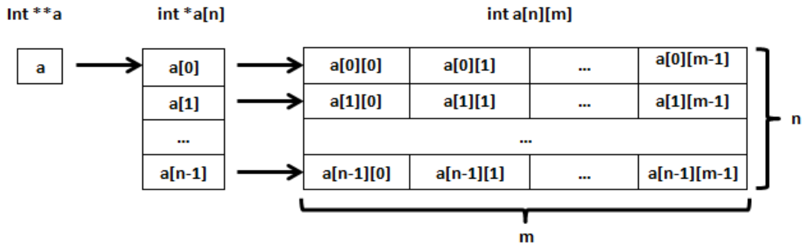

Рис. 3.2. Структура масиву a[n][m] як динамічного одновимірного масиву динамічних одновимірних масивів

Буде відведено пам'ять під кожен рядок масиву окремо, тобто буде утворено `n` різних одновимірних масивів. Адреси нульових елементів цих масивів зберігатимуться в допоміжному масиві (пам'ять під нього потрібно відвести заздалегідь). Елементами цього масиву будуть адреси цілих чисел, тому вони матимуть тип "*вказівник на ціле число*", тобто `int *`.

Приклад 3.3. Ввести кількість рядків і стовпців матриці a[n][m] та елементи самої матриці. Обчислити добуток її додатних елементів.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
      int n, m;
      float dd = 1;
      cout << "\nVvedit' k-st' riadkiv n = ";
      cin >> n;
      cout << "\nVvedit' k-st' stovpciv m = ";
      cin >> m;
      float **a = new float* [n];
      /* відведення пам'яті під допоміжний масив */
      /* відведення пам'яті під кожний рядок матри-
ці окремо*/
      for (int i = 0; i < n; i++)
            a[i] = new float[m];
      cout << "\nInput matrix n*m:" << endl;
```

```
	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++)
		{
			cin >> a[i][j];
			if (a[i][j] > 0)
				dd *= a[i][j];
	// обчислення добутку додатних елементів
		}
	cout << "\nDobutok dodatnyh elementiv dd = "
<< dd << endl;
	for (int i = 0; i < n; i++)
	// вивільнення пам'яті від рядків матриці
		delete []a[i];
	// вивільнення пам'яті від допоміжного масиву
	delete []a;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду

```
Vvedit' k-st' riadkiv n = 2
Vvedit' k-st' stovpciv m = 3
Input matrix n*m:
1.1
-2.2
-3.
4.
5.
6.

Dobutok dodatnyh elementiv dd = 132

Dobutok dodatnyh elementiv p = 132
```

Розглянемо цей приклад більш детально. Спочатку ми створюємо подвійний вказівник `float **a`: вказівник на масив вказівників. Під цей масив вказівників ми відводимо ділянку пам'яті за допомогою оператора **new** (див. рис. 12.2)

```
a = new float*[n];
```

Потім для кожного такого вказівника (їх кількість становить `n`) створюється окремий динамічний одновимірний масив розмірності `m`:

```
for(int i = 0; i < n; i++)
a[i] = new float[m];
```

Отже, ми отримаємо матрицю a розміром n*m.

Інший спосіб створення багатовимірного масиву a[n][m] полягає у використанні одновимірного. Розглянемо приклад формування матриці на основі одновимірного динамічного масиву.

```
#include <iostream>
using namespace std;
int main()
{
	int n, m;
	cout << "Vvedit' k-st' riadkiv n = ";
	cin >> n;
	cout << "\nVvedit' k-st' stovpciv m = ";
	cin >> m;
	// оголошення вказівника
	int *a;
	/*відведення пам’яті для масиву розмірності
n*m ... */
	a = new int[n * m];
	cout << "Vvedit' pochatkovyj masyv a:" <<
endl;
	for (int i = 0; i < n * m; i++)
		cin >> a[i];
	cout << "Pochatkovyj masyv a:\n";
	for (int i = 0; i < n*m; i++)
		cout << a[i] << " ";
	cout << endl;
	//виведення елементів масиву a матрицею
	cout << "Vyvedennia masyvu a matryceju:\n";
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < m; j++)
		{
			cout << a[i * m + j] << "    ";
		}
		cout << endl;
	}
	//вивільнення пам’яті від масиву a
	delete []a;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду;

```
Vvedit' k-st' riadkiv n = 2
Vvedit' k-st' stovpciv m = 3
Vvedit' pochatkovyj masyv:
1 2 3 4 5 6
Pochatkovyj masyv a:
1 2 3 4 5 6
Vyvedennia masyvu a matryceju:
1   2   3
4   5   6
```

Поданий програмний код є ілюстративним і жодних перетворень початкового масиву не виконується. Завдання полягало у тому, щоб показати як можливо на програмному рівні подати двовимірний масив одновимірним та виконати виведення елементів одновимірного масиву у вигляді матриці.

Тому, при вивільненні пам'яті від елементів масиву використана настанова

```
delete[] a;
```

Приклад 3.4. У двовимірному масиві замінити усі елементи у стовпці з максимальним елементом на цей максимальний елемент. Кількість рядків та стовпців масиву задаються користувачем.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
      int n, m;
      cout << "\nVvedit' k-st' riadkiv n = ";
      cin >> n;
      cout << "\nVvedit' k-st' stovpciv m = ";
      cin >> m;
      /* відведення пам'яті під допоміжний масив */
      int **matr = new int *[n];
      /* відведення пам'яті під кожний рядок матри-
ці окремо*/
      for (int i = 0; i < n; i++)
            matr[i] = new int[m];
      cout << "\nPochaykovyj masyv " << endl;
      for (int i = 0; i < n; i++)
```

```
	{
		for (int j = 0; j < m; j++)
		{
			matr[i][j] = rand() % 20-10;
			cout << matr[i][j] << "\t";
		}
		cout << "\n";
	}
	int ns = 0;
	int matrmax = matr[0][0];
	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++)
			if (matr[i][j] > matrmax)
			{
				matrmax = matr[i][j];
				ns = j;
			}
	/*Заміна значень усіх елементів у стовпці
	  з найбільшим елементом
	  на цей елемент*/
	for (int i = 0; i < n; i++)
		matr[i][ns] = matrmax;
	cout << "\nPeretvorenyj masyv ";
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < m; j++)
			cout << matr[i][j] << "\t";
		cout << "\n";
	}
	//вивільнення пам'яті від рядків матриці
	for (int i = 0; i < n; i++)
	delete []matr[i];
	//вивільнення пам'яті від допоміжного масиву
	delete []matr;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду;

```
Vvedit' k-st' riadkiv n = 5
Vvedit' k-st' stovpciv m = 7
Pochaykovyj masyv
```

```
0         6         -7        -1        18        -7        7
7         11        3         -6        -6        -10       16
-10       0         14        -9        16        -5        10
13        -9        -8        11        11        10        15
-3        -6        6         -5        0         7         -2
Peretvorenyj masyv
0         6         -7        -1        18        -7        7
7         11        3         -6        18        -10       16
-10       0         14        -9        18        -5        10
13        -9        -8        11        18        10        15
-3        -6        6         -5        18        7         -2
```

Беручи до уваги окреслене, можемо сформувати такі зауваження та підсумки:

– варто дотримуватись розглянутих правил при визначенні вказівників та масивів вказівників (наприклад, `int *a[10]` – масив з 10 вказівників);

– використовуючи явні механізми для відведення пам`яті, не забувати про необхідність вивільнення відведеної пам’яті (враховувати, що під час вивільнення відповідний вказівник не змінює свого значення);

– відводячи пам’ять, враховувати можливість відсутності потрібного обсягу вільної пам’яті;

– не порушувати правил ″адресної арифметики″ та порівнянь вказівників;

– не допускати виходу вказівника за межі пам’яті об’єкта.

Контрольні запитання

1. Як терміни статичний/динамічний характеризують зміну властивостей об’єкта у процесі виконання програмного коду?
2. Що таке динамічна пам’ять?
3. Які способи відведення пам’яті ви знаєте?
4. Як робота з динамічними змінними пов’язана з використанням вказівників?
5. Які оператори використовуються у С++ для відведення та вивільнення динамічної пам'яті?
6. У чому полягають відмінності динамічного масиву від звичайного?
7. Як використати оператор вказівник на вказівник для створення двовимірного масиву?

Розділ 4

СИМВОЛИ І РЯДКИ

Ефективність мови C++ багато в чому визначається наявністю у ній розвинутих засобів для оброблення символьної інформації. У Стандартній бібліотеці C++ передбачено багато функцій, які виконують широкий спектр перетворень із символьними даними.

У наукових джерелах, які стосуються вивчення мови програмування C++, застосовують термін ″рядок у стилі C″ (C-*style*). Спробуємо з'ясувати чому у C++ вживають цей термін.

Рядки/символи, зазвичай притаманні мові програмування C і становлять ″спадок″ від C до C++, у якій підтримуються та інтенсивно використовуються. Рядок символів у C інтерпретується типом контейнера/даних, який містить символи (включно з символом пробіл) у вигляді масиву, який є одновимірним і завершується нуль-термінатором – '\0'. Символ – значення типу `char` для збереження цілочисельного коду символу. Отже, такі рядки називають C-рядками або рядками у C-*style*.

C++ допускає використання бібліотеки функцій, які виконують перетворення з рядками і підтримуються C++ за обов'язкового їх включення у заголовний файл.

Сучасний C++ підтримує такі два різні типи рядків:

– **`string`** (як частина Стандартної бібліотеки мови C++);

– рядки C-*style* (успадковані від мови C).

Тепер перейдемо до викладення основного матеріалу розділу.

4.1. Символьний тип даних у C++

Символами вважаються: великі й малі латинські літери, великі й малі літери кирилиці, цифри, пробіл, розділові знаки ('**.**', '**,**', '**;**', '**:**', '!', '?', '–'), знаки арифметичних дій ('+', '–', '*', '/', '='), службові символи, що відповідають клавішам [Enter], [Esc], [Tab] тощо. У C++ зна-

чення символьних констант записуються в одинарних лапках: '3', 'f', '+', '%'.

Існує єдиний міжнародний стандарт – так звана таблиця ASCII-кодів (*American Standard Code for Information Interchange* – американський стандартний код для обміну інформацією). Символи ASCII мають коди від 0 до 127, тобто значення першої половини можливих значень байта, хоча часто кодами ASCII називають всю таблицю з 256 символів. Перші 128 ASCII-кодів є єдині для всіх країн, а коди від 128 до 255 називають розширеною частиною таблиці ASCII, де залежно від країни розміщується національний алфавіт і символи псевдографіки. У різних версіях С++ коди символів національного алфавіту можуть мати різні значення.

Символьний тип у С++ називаєтьсяться **char**. Наприклад, за оголошення змінних

```
char c, s, g;
```

в оперативній пам'яті для кожної з цих трьох змінних буде відведено один байт. Коли символьні змінні набудуть певних значень, тоді комп'ютер запише в пам'ять не самі символи, а їхні коди (у вигляді цілих чисел). Наприклад, замість літери 'A' зберігатиметься її код 65. Тому, якщо присвоїти символьній змінній певне ціле число – С++ буде сприймати його як код символу з таблиці ASCII-кодів.

Крім типу **char**, існує його беззнакове модифікування **unsigned char**. Дані типу **unsigned char** мають значення у діапазоні 0...255.

Символи можна порівнювати. Більшим вважається той символ, в якого значення коду є більшим, тобто символ, розміщений у таблиці ASCII-кодів пізніше. Наприклад: 'a' < 'h', 'A' < 'a'.

Специфіка мови С++ при опрацюванні символьних даних полягає в тому, що за замовчуванням тип є символьним, тобто зберігає значення кодів символів від –128 до 127. Водночас від'ємні значення мають коди розширеної частини таблиці ASCII.

Наприклад, визначити чи є символ c цифрою можна двома способами: перевірити його належність до символьного проміжку від '0' до '9' за допомогою настанови:

```
if (c >= '0' && c <= '9') …
```

або застосувати функцію **isdigit()** для перевірки належності символу до множини цифр:

```
if (isdigit(c)) …
```

4.2. Рядки символів

Символьний масив – це зазвичай рядок у стилі С, який підтримує С ++. У С++ рядки розглядають як масиви, елементи яких мають тип **char**.

Рядок – це послідовність символів, яка закінчується нульовим символом (нуль-термінатором) – '\0'.

Нуль-термінатор – це спеціальний символ (ASCII-код якого дорівнює нулю ″ 0 ″), який застосовується для позначення завершення рядка.

Рядок може містити символи літер, цифр і спеціальних символів, які записують у подвійних лапках. Ім'я рядка є константним вказівником на адресу першого символа.

Фактично за розташуванням '\0' символу визначається довжина рядка. Отже, кількість елементів символьного масиву, який поданий рядком, є кількість символів + 1.

Особливою рисою символьного масиву є те, що в ньому насправді може бути менше символів, ніж задекларовано при оголошенні. Крім того, з цими масивами можна виконувати певні специфічні дії, які не можна здійснювати з числовими масивами (наприклад перевіряти наявність у масиві літери або послідовності літер, порівнювати масиви за алфавітом, дописувати один масив наприкінці іншого тощо).

Формат синтаксичної конструкції оголошення рядка має вигляд:

```
char <ім'я_рядка> [<розмір>];
```

Наприклад,

```
char s[10];
// Рядок s може містити до 9-ти символів
```

Нумерація символів у рядку починається з нуля (0). Останній символ рядка – службовий символ, завершальний нуль-символ '\0' (нуль-термінатор), який є ознакою кінця рядка. При оголошенні рядка потрібно вказати максимальну кількість символів, які будуть зберігатися у рядку, водночас слід враховувати наявність наприкінці рядка нуль-термінатора і відводити додатковий байт для нього.

Ініціалізування символьних послідовностей відбувається за тими ж правилами, що і ініціалізування звичайного масиву. Наприклад, якщо ми хочемо записати у масив якусь заздалегідь відому послідовність символів, можемо зробити це так само, як зі звичайним масивом, де кожен елемент буде мати тип `char`, наприклад:

```
char masyv[20];
```

У оголошеному масиві можна зберігати до 20 символів. Проте, можна зберігати і коротші послідовності. У змінній `masyv` можна зберігати послідовності різної довжини, наприклад рядок "`Hello`" або "`Wikipedia world`", оскільки довжина цих послідовностей є меншою, ніж 20 символів. Довжину цих послідовностей визначимо нуль-символом '\0'.

Отже, оголошений нами масив, у якому можуть зберігатися запропоновані рядки, буде виглядати як на рисунку 4.1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
H	**e**	**l**	**l**	**o**	**\0**														

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
W	**i**	**k**	**i**	**p**	**e**	**d**	**i**	**a**		**w**	**o**	**r**	**l**	**d**	**\0**				

Рис. 4.1. Варіанти розміщення рядків символів у масиві masyv

Елементи масиву, які розташовані праворуч від символу '\0', не є частиною рядків, але оскільки масив має фіксовану довжину, вони залишаються зарезервованими у пам'яті.

Ініціалізування елементів символьних послідовностей здійснюється за тими ж правилами, що і звичайних масивів, наприклад:

```
char masyv[20] = {'H', 'e', 'l', 'l', 'o', '!',
'\0'};
```

Для символьних масивів можна використовувати скорочений варіант ініціалізування використанням такої синтаксичної конструкції:

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

Рядки у лапках завжди неявно містять нуль-символ, тому при ініціалізуванні записувати його немає потреби.

Якщо рядок застосовується для ініціалізування масиву типу **`char`**, адреса його першого елемента стає синонімом імені масиву. Наприклад, ідентичними є такі описи масиву:

```
char st[ ] = "Студент";
char st[8] = "Студент";
char st[8] = {'С' 'т' 'у' 'д' 'е' 'н' 'т' '\0'};
```

Рядок при оголошенні можна ініціалізувати початковим значенням, наприклад так:

```
char s[10] = "abcdef";
char Strings[5][60] =
{
```

```
    "Text-1",
    "Text-2",
    "Text-3",
    "Text-4",
    "Text-5"
};
```

Перші шість символів рядка `s` набудуть значень кодів вказаних символів, а решта чотири символи – значення ‘\0’.

У цьому випадку кожному елементу масиву присвоюється один символ рядка. Значення `розмір` задає розмір ділянки пам’яті, яка відводиться для масиву. Значення `розмір` повинно бути не меншим від довжини рядка, в іншому разі компілятор надасть повідомлення про помилку.

Оголошений нами масив `Strings[5][60]`, у якому можуть зберігатися запропоновані рядки, буде виглядати як на рисунку 4.2.

Strings

		0	1	2	3	4	5	6	7	...	59
Strings[0]=	0	'T'	'e'	'x'	't'	'-'	'1'	'\0'	'\0'	...	'\0'
Strings[1]=	1	'T'	'e'	'x'	't'	'-'	'2'	'\0'	'\0'	...	'\0'
Strings[2]=	2	'T'	'e'	'x'	't'	'-'	'3'	'\0'	'\0'	...	'\0'
Strings[3]=	3	'T'	'e'	'x'	't'	'-'	'4'	'\0'	'\0'	...	'\0'
Strings[4]=	4	'T'	'e'	'x'	't'	'-'	'5'	'\0'	'\0'	...	'\0'

Рис. 4.2. Структура розміщення рядків символів у масиві `Strings`

Як і в масивах з класичним інтерпретуванням (індексуванням) доступу до символа є запис:

<ім’я_рядка>[<номер>]

Для роботи з масивом рядків можна використовувати вказівники. Вказівник оголошується так:

char *<ім’я_рядка>;

Наприклад:

```
char masyv[20] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘!’, ‘\0’};
char *S = masyv;
```

оголошений вказівник S містить адресу першого символу (H), а *S – містить перший символ масиву masyv – ‘H’. Як і в адресній арифметиці *(S + 2) – ‘l’.

Необхідно зауважити, що при оголошенні символьного масиву його ім'я – не змінна, а вказівник-константа на перший символ рядка, тому її не можна використовувати у деяких операторах адресної арифметики.

Можна працювати з кожним окремим символом, використовуючи його індекс (порядковий номер елемента у масиві), наприклад, фрагмент коду:

```
for (int i = 0; i < n; i++)
Str[i] = Str1[i];
```

Для роботи з двовимірним символьним масивом використовується 2 індекси. Наприклад,

```
char str[3][9] = {"student", "kursant",
"asystent"};
```

Кожне слово – це окремий рядок. Тоді, str[2][0] – це символ 'a'.

До речі, якщо рядок при оголошенні ініціалізується, тоді зовсім не обов'язково вказувати його розмір у квадратних дужках:

```
char s[] = "abcdef";
```

Водночас компілятор визначає довжину рядка самостійно, а у кінець рядка записується значення '\0'. Отож, якщо ви зміните рядок пізніше, вам не доведеться вручну змінювати значення довжини масиву.

4.3. Функції для роботи з рядками

У мові C++ розроблено потужний набір інструментів для створення різноманітних програм – *бібліотеку стандартних шаблонів* (*Standard Template Library*, STL). Ця бібліотека, фактично, є невід'ємною складовою мови C++, що є великим досягненням її розробників. Бібліотека STL включена до стандарту мови C++ і містить універсальні шаблонні класи та функції, які реалізовують широкий спектр алгоритмів та структур даних. Ці засоби програмування можна застосовувати практично до довільних типів даних.

Кожен заголовок із Стандартної бібліотеки C включено в Стандартну бібліотеку C++ під іншим ім'ям, з огляду на ту обставину, що вилучено розширення `.h`, і додано '`c`' на початку; наприклад, '`time.h`' перетворився на '`ctime`'. Єдиною різницею між файлами заголовків C++ і тими, що були в Стандартній бібліотеці C є те, що там, де це було можливо, функції мають знаходитись у просторі імен

`std`. У стандарті ISO C, функції в Стандартній бібліотеці могли реалізовуватись за допомогою макросів, що не дозволено в ISO C++.

У C++ існують спеціальні функції для роботи з символьними даними (див. табл. 4.1).

Таблиця 4.1.

Функції для роботи з символьними даними

Функція	Призначення
`tolower()`	повертає символ у нижньому регістрі
`toupper()`	повертає символ у верхньому регістрі
Подальші функції перевіряють належність символу до множини:	
`isalnum()`	латинських літер і цифр (‘A’ – ‘Z’, ‘a’ – ‘z’, ‘0’ – ‘9’)
`isalpha()`	латинських літер (‘A’ – ‘Z’, ‘a’ – ‘z’)
`iscntrl()`	керувальних символів (з кодами 0..31 й 127)
`isdigit()`	цифр (‘0’ – ‘9’)
`isgraph()`	видимих символів, тобто не є відповідними клавішам [Esc], [Tab] тощо
`islower()`	латинських літер нижнього регістру (‘a’ – ‘z’)
`isprint()`	друкованих символів (isgraph() + пробіл)
`isupper()`	літер верхнього регістру (‘A’ – ‘Z’)
`ispunct()`	знаків пунктуації
`isspace()`	символів-розділювачів

Примітка. Функції перевірки й перетворення регістру працюють лише з латинськими літерами. Спроба використати ці функції для символів кирилиці спричинить виведення повідомлення про помилку. Тому для роботи з літерами кирилиці, потрібно використовувати перевірку належності символу до відповідного символьного проміжку, а для перетворення регістру – зменшення або збільшення коду символу на різницю кодів між великими та малими літерами (для більшості літер вона становить 32).

Потокове введення/виведення рядків у консольному додатку, крім вже знайомих вам потокових команд C++ **`cin`** >> та **`cout`** << (з бібліотеки **`iostream`**), використовують C-функції **`gets`**/**`puts`** та **`scanf`**/**`printf`** (з бібліотеки **`stdio.h`**). Ці функції ми не будемо розглядати, а розглянемо новіші версії, які використовуються у стандарті C++.

Специфіка потокової команди введення **cin** >> полягає у тому, що вона вводить послідовність символів до першого пробілу або іншого розділювача, тобто цю команду *недоречно використовувати для введення послідовностей з пробілами*.

C++ має велику колекцію функцій опрацювання рядків із завершальним нульовим символом. Якщо у рядку відсутній нульовий символ (нуль-термінатор), опрацювання рядка може тривати безконечно,

доки в пам’яті не трапиться ‘\0’. Як аргументи до функцій зазвичай передаються вказівники на рядки.

Якщо при виконанні функції здійснюється перенесення символів рядка з місця-джерела до місця-призначення, для рядка-призначення потрібно завчасно зарезервувати місце в пам’яті.

Копіювання рядків з використанням просто вказівника, а не адреси початку завчасно оголошеного масиву – одна з найпоширеніших помилок програмування. Під час виділення місця для рядка-призначення слід відводити місце і для нуль-термінатора. Більшість функцій потребує, щоб рядок був оголошений як вказівник типу **char** *. Для використання цих функцій треба під’єднати бібліотеку <**cstring**>.

Для введення рядка символів використовується функція **gets_s**(), оскільки вона дає змогу контролювати кількість зчитуваних символів, наприклад:

```
char s[50];
gets_s(s);
або
gets_s(s,50);
```

Функція **`getline`** призначена для введення послідовності типу **`char`** з потоку до деякого розділювача, який не записується в отриманий масив даних. Виклик функції виглядає так:

```
cin.getline(s,n,d);
```

де: `s` – зчитуваний рядок типу **`char`**; `n` – найбільша кількість символів, яку можна записати в рядок; `d` розділювач, який вказує на кінець зчитуваного рядка. Останнім параметром функції можна знехтувати. У цьому разі буде використано розділювач ‘`\n`’, який генерується натисканням клавіші *Enter*.

Подамо приклад застосування цієї функції для зчитування рядка до символу-розділювача – знаку оклику `'!'`.

Проєкт програмного коду:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
      char s[99];
      cout << "Vvedit' riadok: Ja dobryj student!"
<< endl;
      //Розділювачем буде !
      cin.getline(s, 99, '!');
```

```
      cout << "\nOtrymanyj rezul’tat: " << s <<
endl;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду

```
Vvedit' riadok: Ja dobryj student!
Ja dobryj student!
Otrymanyj rezul’tat: Ja dobryj student
```

Функція **getline** має ще один варіант застосування – до рядкової змінної необмеженої довжини типу **string**, опис якої подано далі:

getline(f, s, d);

де: f – потік даних, s – змінна для запису рядка, d – розділювач, який вказує на кінець рядка. Останній параметр функції можна не вказувати. У цьому разі буде використано розділювач '\n'. Подамо приклад застосування цієї функції.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s;
      cout << "Vvedit' riadok: Ja dobryj student!"
<< endl;
      //Розділювачем буде !
      getline(cin, s, '!');
      cout << "\nOtrymanyj rezul’tat: " << s <<
endl;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду

```
Vvedit' riadok: Ja dobryj student!
Ja dobryj student!
Otrymanyj rezul’tat: Ja dobryj student!
```

Розглянемо найбільш уживані функції для роботи з символами і рядками типу **char** *.

Бібліотека cstring:

– Функція **strlen**(s) – обчислює кількість символів у рядку s без врахування нуль-символу.

– Функція **strnlen_s**(s, n) – обчислює кількість символів у рядку s розміром n без врахування останнього символу '\0'.

– Функція **strcpy**(s,t) – копіювання символів з рядка t у рядок s.

– Функція **strncpy**(s,t,n) – копіювання n символів з рядка t у рядок s, повертає s.

– Функція **strcat**(s,t) – зберігає в s результат об'єднання рядків s і t.

– Функція **strncat**(s,t,n) – зберігає в s результат об'єднання рядка s і n символів рядка t.

– Функція **strcmp**(s,t) – порівнює з урахуванням регістру рядки s і t, повертає ціле значення типу int: 0, якщо рядки збігаються; від'ємне, якщо s < t; додатне, якщо s > t.

– Функція **strncmp**(s,t,n) – так як і strcmp порівнює n символів рядка s з рядком t.

– Функція **strtok**(string, delim) – виконує пошук лексем у рядку string. Послідовність викликів цієї функції розбиває рядок string на лексеми, які є послідовністю символів, відокремлених символами-разділювачами.

На перший виклик, функція приймає рядок string як аргумент, чий перший символ використовується як початкова точка для пошуку лексем. У наступні виклики, функція очікує нульового вказівника та використовує позицію відразу після закінчення останньої лексеми як нову початкову точку для сканування.

Для визначення початку лексеми функція спочатку визначає символи, які у є рядку delim, тобто є символами-розділювачами, а потім символьно перевіряє решту рядка до першого символу-розділювача, який сигналізує про кінець лексеми. Цей кінцевий маркер автоматично замінюється нульовим символом і функція повертає лексему. Після цього наступні "дзвінки" функції **strtok** починаються з цього нульового символу.

Параметри:

- string – рядок для пошуку у ньому лексем. Вміст цього рядка буде змінено, він поділяється на дрібніші рядки (лексеми). Цей параметр може містити нульовий вказівник, у цьому разі функція продовжує сканування з місця, де було зупинено попередній успішний виклик функції.

– `delim` – рядок, який містить набір символів-розділювачів. Вони можуть змінюватись від одного виклику до іншого виклику функції.

Наприклад, використовуючи задані символи-розділювачі вивести у консольний додаток початковий рядок окремими словами у стовпець.

Приклад програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
//Початковий рядок
    char str[] = "Straustrup later renamed the
language to C++ in 1983.";
    cout << "Podil riadka "" << str << "" na
leksemy:\n";
    char *pch = strtok(str, " ,.-");
/* другим параметром задано масив символів-
розділювачів (пробіл, кома, крапка, тире)*/
// Цикл доки наявні лексеми
    while (pch != NULL)
    {
        cout << pch << "\n";
        pch = strtok(NULL, " ,.-");
    }
      system ("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
Podil riadka  << str <<  na leksemy:
Straustrup
later
renamed
the
language
to
C++
in
1983
```

– Функція **`isalnum`**`(c)` – повертає ненульове ціле значення, якщо `c` – літера або цифра, інакше – `false`.

– Функція **isalpha**(c) – повертає ненульове ціле значення, якщо c – літера, інакше – false.

– Функція **isdigit**(c) – повертає ненульове ціле значення, якщо c – цифра, інакше – false.

– Функція **isblank**(c) – повертає ненульове ціле значення, якщо c – пробіл, інакше – false.

– Функція **iscntrl**(c) – повертає ненульове ціле значення, якщо c – символ керування, інакше – false.

– Функція **isgraph**(c) – повертає ненульове ціле значення, якщо c – друкований символ, інакше – false.

– Функція **islower**(c) – повертає ненульове ціле значення, якщо c – літера нижнього регістру, інакше – false.

– Функція **isprint**(c) – повертає ненульове ціле значення, якщо c – друкований символ, інакше – false.

– Функція **ispunct**(c) – повертає ненульове ціле значення, якщо c – знак пунктуації, інакше – false.

– Функція **isspace**(c) – повертає ненульове ціле значення, якщо c – пробіл, інакше – false.

– Функція **isupper**(c) – повертає ненульове ціле значення, якщо c – літера верхнього регістру, інакше – false.

– Функція **isxdigit**(c) –повертає ненульове ціле значення, якщо c –16-кова цифра, інакше – false.

– Функція **strchr**(s,c) – пошук першого входження символу c у рядку s. У разі вдалого пошуку повертає вказівник на місце першого входження символу c.

Настанова:

```
cout << strchr(s,c);
```

виводить рядок s, починаючи з цього першого входження. Якщо символ не знайдено, тоді буде повернуто NULL.

– Функція **strcspn**(s,t) – повертає довжину початку s, який не містить символів t.

– Функція **strspn**(s,t) – повертає довжину початку s, який містить лише символи t.

– Функція **strprbk**(s,t) – повертає вказівник першого входження довільного символу рядка t у рядок s.

Настанова:

```
cout << strpbrk(s,t);
```

виводить рядок s, починаючи з цього першого входження. Якщо символ не знайдено, тоді буде повернуто NULL.

– Функція **toupper**(c) – якщо символ c – маленька літера *латинського алфавіту*, тоді функція повертає відповідну велику літеру, інакше – символ без змін.

Приклад програмного коду:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
      char s[] = "little letters 1234f56";
      cout << "\nPochatkovyj riadok: " << s <<
endl;
      cout << "\nPeretvorenyj riadok: " << endl;
      for (int j = 0; j < strlen(s); j++)
            putchar(toupper(s[j]));
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
Pochatkovyj riadok: little letters 1234f56
Peretvorenyj riadok:
LITTLE LETTERS 1234F56
```

Бібліотека cstdlib

– Функція **atof**(s) – перетворює рядок s у тип double.

– Функція **atoi**(s) – перетворить рядок s у тип int.

– Функція **atol**(s) – перетворить рядок s у тип long.

Контрольні запитання

1. Які рядки називають С-рядками або рядками у С-*style*?
2. Які два різних типи рядків підтримує сучасна С++?
3. Символьний тип даних у С++.
4. Ініціалізування елементів символьних послідовностей.
5. Перелічіть відомі вам функції для введення рядка символів.

Розділ 5

РЯДКИ ТИПУ `string`

5.1. Використання рядків типу `string`

Раніше розглядалися задачі оброблення символьних даних засобами мови С++, зокрема і символьних рядків. Однак в новітніх версіях мови С++ введена стандартна бібліотека шаблонів - *Standard Template Library* (STL), яка містить клас **`string`** з більш потужними засобами оброблення рядків.

Для під’єднання цього класу до програмного коду потрібно записати директиву:

```
#include <cstring>
using namespace std;
```

і під’єднати простір імен бібліотеки шаблонів.

Після цього можна оголошувати змінні типу **`string`**:

```
string <ім’я_змінної>;
```

Для масиву рядків пам’ять можна відводити:

– статично (у цьому випадку вказується фіксоване константне значення масиву на етапі компілювання);

– динамічно (з допомогою оператора **`new`**. У цьому випадку розмір масиву створюється динамічно і може задаватись у процесі виконання програми).

Рядки можна об’єднувати у масиви, які оголошуються звичайним засобом, тобто

```
string <ім’я_змінної> [<розмір>];
```

Доступ до символів рядка здійснюється шляхом запису порядкового номера символа-індексу, який починається з нуля.

Наприклад, якщо записати

```
string str = ″Мій рядок″;,
```

тоді

`str[2]` - це буде літера `'й'`.

Для масивів рядків потрібний символ визначається шляхом запису двох індексів: індексу елемента масиву та індексу символа в цьому елементі, тобто у вигляді `Strarray[i][j]`.

Ініціалізування рядків при оголошенні виконується одним із способів: з вказанням розміру масиву і автоматичним визначенням розміру масиву, наприклад.

```
string st1 = ”Це рядок класу string”;
string st2 (”Це інший рядок класу string”);.
```

Приклад проєкту програмного коду ініціалізування елементів масиву рядків типу `string`:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
	// Масиви рядків типу string
	/* 1. Ініціалізуваня масиву рядків з вказанням розміру масиву*/
	int const N_DAYS = 7;
	string daysOfWeek[N_DAYS] = { "Sunday", 
"Monday", "Tuesday", "Wednesday", "Thirsday", 
"Friday", "Saturday" };
	// Виведення масиву рядків на екран
	cout << "Array of days:\n";
	for (int i = 0; i < N_DAYS; i++)
		cout << "Day " << i + 1 << " = " << 
daysOfWeek[i] << endl;
	/* 2. Ініціалізуваня без вказання розміру 
масиву*/
	string Numbers[]{ "One", "Two", "Three" };
	// Виведення масиву у консольний додаток
	cout << "\nArray of Numbers:" << endl;
	for (int i = 0; i < 3; i++)
		cout << Numbers[i] << " ";
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Array of days:
Day 1 = Sunday
Day 2 = Monday
```

```
Day 3 = Tuesday
Day 4 = Wednesday
Day 5 = Thirsday
Day 6 = Friday
Day 7 = Saturday

Array of Numbers:
One Two Three
```

Оголошення вказівника на рядок здійснюється так:

string *<ім'я_вказівника>;.

Пам'ять для вказівника може бути відведена з довільним початковим значенням за допомогою оператора **new**, наприклад:

`string *pstr1 = new string;`

оголошується порожній рядок,

`string *pstr2 = new string ("Новий рядок");`

вказівник вказує на рядок `"Новий рядок"`.

Раніше оголошеному вказівнику `*pstr1`, який ні на що не вказує, можна присвоїти значення у вигляді

`pstr1 = new string ("Це перший рядок");.`

Значення рядка **string** містить довільний набір символів, записаний у лапках.

Для рядків типу **string** визначено такі оператори:

– конкатенації (приєднання), котрі позначаються символом " + ";

– відношення (" == ", " != ", " > ", " >= ", " < ", " <= ").

Наприклад:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
      // Масиви рядків типу string
      string st1 = "And one will say - Glory to
Ukraine!";
      string st2 = "And millions will respond to -
Heroes Glory!";
      string st3 = st1 + ' ' + st2;
            // Виведення масивів рядків
      cout << "\n st1\n" << st1;
      cout << "\n st2\n" << st2;
```

```
       cout << "\n st3\n" << st3;
       system("pause");
       return 0;
}
```

Результат виконання програмного коду, який виведе у консольний додаток вміст об’єднаних рядків `st1` та `st2`:

```
 st1
And one will say Glory to Ukraine!
 st2
And millions will respond to Heroes Glory!
 st3
And one will say Glory to Ukraine! And millions
will respond to Heroes Glory!
```

Для введення рядків **`string`**, крім настанов присвоювання, застосовують функції введення даних:

```
cin >> st;
getline(cin, st, ‘\n’);
```

Функції визначення довжини рядка:

```
str.size();
str.length();
str.max_size();
```

Наприклад:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
       string str, st = " To be or not to be - that
is the question!";
       str = "question ";
       cout << "String length str = " << str.size()
       << "\nString length st = " << st.length() <<
endl;
       system("pause");
       return 0;
}
```

Результат виконання програмного коду:

```
String length str = 9
String length st = 43
```

Виведення рядків у консольному додатку здійснюється шляхом використання звичайних настанов виведення даних.

Розглянемо найбільш уживані функції для роботи з рядками типу **string.**

Функція **assign** має кілька реалізувань:

– **assign**(s) – надання значення рядка s;

– **assign**(s,j,n) – копіювання n символів з рядка s, починаючи з символа з номером j;

– **assign**(p,n); – копіювання перших n символів з рядка p типу char *.

Наступна частина коду подає приклади застосування цих реалізувань у порядку переліку:

```
string s0 = "abcdefgh", s1,s2,s3;
char *p  = "abcdefgh";
s1.assign(s0);        // s1 = "abcdefgh"
s2.assign(s0, 5, 2); // s2 = "fg"
s3.assign(p, 5);     // s3 = ""abcde"
```

Функція **append** – дописування у кінець рядка – має такі варіанти реалізації:

– функція отримує посилання на рядок s, який додають (дописують) до об'єкта, який її викликає;

– функція отримує вказівник на рядок типу **const char** *, яка завершується символом '\ 0' – див. наступний проєкт програмного коду:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s0 = "abc", s, t = "0123456";
    const char *p = "0123456";
    s = s0;  s.append(t);
    cout << s << endl; // abc0123456
    s = s0;  s.append(t, 2, 3);
    cout << s << endl; // abc234
    s = s0;  s.append(p);
    cout << s << endl; // abc0123456
    s = s0;  s.append(p, 2);
    cout << s << endl; // abc01
    s = s0;  s.append(p, 2, 3);
    cout << s << endl; // abc234
```

```
      system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
bc0123456
abc234
abc0123456
abc01
abc234
```

Функція **`insert`** – надає можливість вставити у задану позицію рядка інший рядок повністю або лише частину (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s = "abcdef", t = "123456";
      s.insert(4, t);
      cout << s << endl; // abcd123456ef
      s.insert(2, t, 2, 4);
      cout << s << endl; // ab3456cd123456ef
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
abcd123456ef
ab3456cd123456ef
```

Функція **`replace`** виконує заміну символів одного рядка на інший або лише його частину. Перші два аргументи задають діапазон замінюваної частини – з якого індексу і якої довжини. Третій аргумент – рядок, на який або частину якого буде здійснено заміну. Четвертий і п’ятий (якщо їх записано) – діапазон частини третього аргумента, на який буде здійснено заміну (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
```

```
	string  t = "abcdef",
	      s, s0 = "0123456";
	s = s0;
	s.replace(2, 3, t);
	cout << s << endl; // 01abcdef56
	s = s0;
	s.replace(2, 6, t);
	cout << s << endl; // 01abcdef
	s = s0;
	s.replace(7, 4, t);
	cout << s << endl; // 0123456abcdef
	s = s0;
	s.replace(2, 7, t);
	cout << s << endl; // 01abcdef
	s = s0;
	s.replace(2, 3, t, 3, 2);
	cout << s << endl; // 01de56
	s = s0;
	s.replace(2, 3, t, 3, 7);
	cout << s << endl; // 01def56
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
01abcdef56
01abcdef
0123456abcdef
01abcdef
01de56
01def56
```

Функція **`clear`** видаляє всі символи з рядка.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string s = "0123456789";
s.clear();
cout << ">" << s << "<" << endl; // ><
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
><    (порожний рядок)
```

Функція **`erase`** видаляє символи з рядка. Вона має два аргументи: індекс (позиція), починаючи з якої потрібно видалити, і кількість символів, які видаляють. Якщо ці аргументи не вказано, то буде видалено усі символи (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s = "0123456789";
      s.erase(3, 5);
      cout << s << endl; // 01289
      s.erase();
      cout << ">" << s << "<" << endl; // ><
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
01289
><  (порожний рядок)
```

Функції **`find`** і **`rfind`** повертають індекс позиції першого входження рядка-аргумента у цей рядок, починаючи з цього індекса-аргумента. Функції відрізняються напрямом пошуку:

– від початку до кінця – функцією **`find`**;

– від кінця до початку – функцією **`rfind`**.

У разі відсутності відповідного входження функції буде повернуто число за межами діапазону значень індексів елементів рядка, у якому здійснюють пошук. Пишуть, що буде повернуто –1, але може бути інакше (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s = "01234567890123456789";
      string t = "345";
      string u = "abcd";
      cout << s.find(t) << endl;    // 3
```

```
	cout << s.find(t, 5) << endl; // 13
	cout << s.find(t, 15) << endl;
//18446744073709551615
	cout << s.find(u) << endl;
//18446744073709551615
	cout << s.rfind(t) << endl;   // 13
	cout << s.rfind(t, 9) << endl;// 3
	cout << s.rfind(t, 2) << endl;
//18446744073709551615
	cout << s.rfind(u) << endl;
//18446744073709551615
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
3
13
4294967295
4294967295
13
3
4294967295
4294967295
```

Функція **find_first_of**(t,j) повертає індекс першої появи будь-якого з символів рядка t, починаючи позиції j. Якщо значення j не вказано, тоді вважають, що воно дорівнює 0, тобто пошук здійснюють з початку рядка.

Функція **find_last_of**(t,j) повертає індекс останньої появи будь-якого з символів рядка t, починаючи позиції j. Якщо значення j не вказано, тоді вважають, що воно дорівнює 0, тобто пошук здійснюють з початку рядка.

Функція **find_first_not_of**(t,j) повертає індекс першої появи символу, відмінного від символів рядка t, починаючи позиції j. Якщо значення j не вказано, тоді вважають, що воно дорівнює 0, тобто пошук здійснюють з початку рядка.

Функція **find_last_not_of**(t,j) повертає індекс останньої появи символу, відмінного від символів рядка t, починаючи позиції j. Якщо значення j не вказано, тоді вважають, що воно дорівнює 0, тобто пошук здійснюють з початку рядка (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s = "01234567890123456789";
      string t = "abc654";
cout << s.find_first_of(t) << endl;          //  4
cout << s.find_first_of(t, 11) << endl;      // 14
cout << s.find_last_of(t) << endl;           // 16
cout << s.find_last_of(t, 11) << endl;       //  6
cout << s.find_first_not_of(t) << endl;      //  0
cout << s.find_first_not_of(t, 11) << endl;// 11
cout << s.find_last_not_of(t) << endl;       // 19
cout << s.find_last_not_of(t, 11) << endl; // 11
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
4
14
16
6
0
11
19
11
```

Функція **compare** надає можливість порівнювати рядок або його частину з іншим рядком-параметром і має такі аргументи (перші два або вказують, або не вказують одночасно):

– індекс, з якого починають перегляд символів рядка для порівнювання;

– кількість символів у підпослідовності, утвореній для порівнювання;

– рядок, який порівнюють з рядком, до якого застосовано функцію.

Функція **compare** працює так: порівнюють коди перших символів, якими різняться рядки; якщо початок одного рядка збігається з іншим, тоді коротший рядок вважають меншим; якщо функцію викликає менший рядок, тоді вона повертає від'ємне ціле значення; якщо функцію викликає більший рядок, тоді вона повертає додатне

ціле значення; якщо функцію викликає такий самий рядок, тоді вона повертає ціле значення нуль – див. наступний проєкт програмного коду:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
	string s = "01234";
	string t = "43210";
	string u = "0123432100";
	cout << s.compare(t) << endl; // -1
	cout << t.compare(s) << endl; // 1
	cout << s.compare(s) << endl; // 0
	cout << s.compare(u) << endl; // -5
	cout << u.compare(s) << endl; // 5
	cout << u.compare(0, 5, s) << endl; // 0
	cout << s.compare(0, 5, u) << endl; // -5
	cout << u.compare(4, 4, t) << endl; // -1
	cout << u.compare(4, 5, t) << endl; // 0
	cout << u.compare(4, 6, t) << endl; // 1
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
-1
1
0
-1
1
0
-1
-1
0
1
```

Функція **c_str** повертає рядок типу **char** * і модифікатором **const** з тією самою послідовністю символів (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
       string s = "abcd";
       const char* p;
       p = s.c_str();
       cout << p;   // abcd
       system("pause");
       return 0;
}
```

Результат виконання програмного коду:

```
abcd
```

Функція **push_back**(c) додає до рядка символ c типу **char** (див. наступний проєкт програмного коду):

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
       string s = "Hello World";
       s.push_back('!');
       cout << s; // Hello!
       system("pause");
       return 0;
}
```

Результат виконання програмного коду:

```
Hello World!
```

Функція **resize**(n) змінює довжину рядка на n. Якщо після n записати символ, тоді при збільшенні довжини рядка вільні місця буде заповнено цим символом:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
       string s = "Hello";
       s.resize(8, '!');
       cout << s;   // Hello!!!
       system("pause");
       return 0;
}
```

Результат виконання програмного коду:

```
Hello!!!
```

Функція **substr**(j) повертає підрядок цього рядка, починаючи з символу з індексом j, до кінця рядка. Якщо після j вказати необов'язковий параметр k, то буде повернуто підрядок довжини k цього рядка, починаючи з символу з індексом j – див. наступний проєкт програмного коду:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string s = "0123456789";
      cout << s.substr(5) << endl;    // 56789
      cout << s.substr(2, 5) << endl; // 23456
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
56789
23456
```

5.2. Приклади розв'язування задач з використанням рядків символів

Приклад 5.1. У введеному текстовому рядку обчислити, скільки разів повторюється визначений символ.

У цьому прикладі для визначення довжини рядка St[100] використовується функція **strnlen_s**(St, 100) Ця функція обчислює кількість символів у рядку без врахування останнього символу '\0'.

Щоб використовувати цю функцію у Visual Studio C++ потрібно на початку файлу перед задаванням простору імен під'єднати бібліотеку **<cstring>**.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
int main()
{
// riadok symvoliv
char St[100];
cout << "vvedit' riadok St:" << endl;
gets_s(St, 100);
cout << "Vvedenyj riadok St:" << endl;
cout << St << endl;
// заданий символ
char a;
cout << "Vvedit' zadanyj symvol: " << endl;
cin >> a;
cout << "Zadanyj symvol: " << a << endl;
int i;
/* результат - кількість входжень символу a в рядку
S*/
int k;
// на початку обнулити лічильник k
k = 0;
for (i = 0; i < strnlen_s(St, 100); i++)
if (St[i] == a)
// збільшити лічильник на 1
k++;
cout << "K-st' vhodzhen' symvolu " << a << " u
riadku St : " << k << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok St:
Ja najkrashchyj student na 2 fakul'teti
Vvedenyj riadok St:
Ja najkrashchyj student na 2 fakul'teti
Vvedit' zadanyj symvol:
a
Zadanyj symvol: a
K-st' vhodzhen' symvolu a u riadku St : 5
```

Приклад 5.2. У введеному текстовому рядку `str[50]` обчислити кількість символів ‘+’ та ‘-’.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
```

```
#include <cstring>
using namespace std;
int main()
{
// обчислення кількості входжень символу в рядку
	char str[50]; // рядок символів
	int i;
// результат - кількість символів '+'
	int kp;
// результат - кількість символів '-'
	int km;
	cout << "Vvedit' riadok str:" << endl;
	gets_s(str, 50);
	cout << "Vvedenyj riadok str:" << endl;
	cout << str << endl;
	kp = 0;
	km = 0;
	for (i = 0; i < strnlen_s(str, 50); i++)
	{
		if (str[i] == '+') kp++;
		if (str[i] == '-') km++;
	}
	cout << " kil'kist' symvoliv '+': " << kp <<
endl;
	cout << " kil'kist' symvoliv '-': " << km <<
endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok str:
Dva + dva = 4, a try - dva = 1.
Vvedenyj riadok str:
Dva + dva = 4, a try - dva = 1.
 kil'kist' symvoliv '+': 1
 kil'kist' symvoliv '-': 1
```

Приклад 5.3. У введеному текстовому рядку `str[50]` замінити всі символи ‘ + ‘ на ‘ - ‘.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
int main()
{
// рядок символів
	char str[50];
	int i;
	cout << "Vvedit' riadok str:" << endl;
	gets_s(str, 50);
	cout << "Vvedenyj riadok str:" << endl;
	cout << str << endl;
	for (i = 0; i < strnlen_s(str, 50); i++)
		if (str[i] == '+') str[i] = '-';
	cout << "Peretvorenyj riadok str:" << endl;
	cout << "\t" << str << endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok str:
Sport + navchannia - zaporuka uspihu
Vvedenyj riadok str:
Sport + navchannia - zaporuka uspihu
Peretvorenyj riadok str:
        Sport - navchannia - zaporuka uspihu
```

Приклад 5.4. Задано декілька рядків тексту (двовимірний масив символів). Замінити всі символи ‘a’ на ‘b’.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
// заданий масив, який містить 5 рядків
	char str[5][50];
	int i, j;
	cout << "\n Vvedit' masyv str[5][50] " <<
endl;
	for (i = 0; i < 5; i++)
		gets_s(str[i], 50);
	cout << "\n Vvedenyj masyv " << endl;
	for (i = 0; i < 5; i++)
	{
```

```
	for (j = 0; j < strnlen_s(str[i], 50); j++)
		cout << str[i][j];
		cout << endl;
	}
	for (i = 0; i < 5; i++)
	for (j = 0; j < strnlen_s(str[i], 50); j++)
		if (str[i][j] == 'a') str[i][j] = 'b';
	cout << "\n Peretvorenyj masyv " << endl;
	for (i = 0; i < 5; i++)
	{
	for (j = 0; j < strnlen_s(str[i], 50); j++)
		cout << str[i][j];
		cout << endl;
	}
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' masyv str[5][50]
Doba maje 24 hodyny
Dobra sprava
Velyka zahroza
Prohramuvannia na C++
Dynamichnyj masyv

 Vvedenyj masyv
Doba maje 24 hodyny
Dobra sprava
Velyka zahroza
Prohramuvannia na C++
Dynamichnyj masyv

 Peretvorenyj masyv
Doba mbje 24 hodyny
Dobrb sprbvb
Velykb zbhrozb
Prohrbmuvbnnib nb C++
Dynbmichnyj mbsyv
```

Приклад 5.5. У введеному довільному тексті відокремити всі слова, вивести їх на екран та визначити найдовше слово

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
int main()
{
    string txt, sl, slmax;
    // slmax — найдовше слово
    int k = 0, n = 0, max = 0;
    // введення тексту
    cout << "***** Enter text\n";
    getline(cin, txt, '\n');
    for (int i = 0; i <= txt.size() - 1; i++)
    {
        if ((txt[i] == ' ') || (txt[i] == ',') ||
(txt[i] == '.'))
        {
// n — лічильник пропусків і розділових знаків
            n++;
            if (n > 1) continue;
//k — лічильник слів
            k++;
            cout << k << " slovo - " << sl << " = "
<< sl.size() << " symbols\n";
            // визначення найдовшого слова
            if (sl.size() > max)
            {
                max = sl.size();
                slmax = sl;
            }
//вивільнення sl для формування слова
            sl = "";
        }
        else {
            n = 0; sl = sl + txt[i];
//змінна s1 формує слово
        }
    }
    cout << "\nslmax — " << slmax << " = " <<
slmax.size() << " symbols\n";
    system("pause");
    return 0;
```

```
}
```

Результат виконання програмного коду:

```
***** Enter text
Kursanty LvDUVS duzhe vidpovidal'ni u navchanni
1 slovo - Kursanty = 8 symbols
2 slovo - LvDUVS = 6 symbols
3 slovo - duzhe = 5 symbols
4 slovo - vidpovidal'ni = 13 symbols
5 slovo - u = 1 symbols
slmax - vidpovidal'ni = 13 symbols
```

У програмі текст вводиться за допомогою функції **`getline()`.** Візьміть до уваги її вигляд, слова виокремлюються не з використанням функцій, а шляхом конкатенації символів, якщо вони не є символами-розділювачами.

Приклад 5.6. Ввести перелік прізвищ студентів групи і відсортувати його за алфавітом.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
int main()
{
      const int n = 5;
      string list[n];
      int i, k;
      //введення переліку прізвищ
      for (i = 0; i < n; i++)
      {
            cout << "*** Enter " << (i + 1) << "
name\n";
            getline(cin, list[i], '\n');
      }
      //сортування переліку прізвищ
      for (k = 1; k < n; k++)
            for (i = 0; i < n - k; i++)
                  if (list[i] > list[i+1])
list[i].swap(list[i + 1]);
      //виведення сортованого переліку
      cout << "\n***** Rezult list\n";
      for (i = 0; i < n; i++)
```

```
            cout << (i + 1) << ' ' << list[i] <<
endl;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
*** Enter 1 name
Ivaniv
*** Enter 2 name
Petriv
*** Enter 3 name
Fedak
*** Enter 4 name
Bondarenko
*** Enter 5 name
Andrusiv

***** Rezult list
1 Andrusiv
2 Bondarenko
3 Fedak
4 Ivaniv
5 Petriv
```

Приклад 5.7. Ввести рядок символів і обчислити кількість розділових знаків з використанням розробленої функції.

```
#include <iostream>
#include <cstring>
using namespace std;
int KRozdznaky(char* s)
{
      int k = 0;
      for (int i = 0; s[i] != '\0'; i++)
            if (s[i] == '.' || s[i] == ',' || s[i]
== ';' || s[i] == '!' || s[i] == '?')
                  k++;
      return k;
}
int main()
{
      setlocale(LC_ALL, ".1251");
      char St[100]; // riadok symvoliv
```

```
        int res;
        cout << "vvedit riadok S:" << endl;
        gets_s(St, 100);
        cout << "Vvedenyj riadok S:" << endl;
        cout << St << endl;
        res = KRozdznaky(St);
        cout << "Kilkist rozdilovyh znakiv: " << res
<< endl;
        system("pause");
        return 0;
}
```

Результат виконання програмного коду:

```
vvedit riadok S:
Hochete staty kryminalistom? Vy, napevno, vstupyte
do LvDUVS.
Vvedenyj riadok S:
Hochete staty kryminalistom? Vy, napevno, vstupyte
do LvDUVS.
Kilkist rozdilovyh znakiv: 4
```

Приклад 5.8. Ввести рядок символів і обчислити кількість слів з використанням розробленої функції.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstring>
using namespace std;
int Ksliv(char* s)
{
    int i;
    int k = 0;
    bool f = true;
    for (i = 0; i < strlen(s); i++)
        if ((s[i] == ' ') || (s[i] == ',') || (s[i]
== '.'))
        {
            f = true; // kinets slova
        }
        else
        {
            if (f)
            {
                // pochatok novoho slova
```

```
                k++;
                f = false;
            }
        }

    return k;
}
int main()
{
    char St[100]; // riadok symvoliv
    int res;
    cout << "vvedit riadok S:" << endl;
    gets_s(St, 100);
    cout << "Vvedenyj riadok S:" << endl;
    cout << St << endl;
    res = Ksliv(St);
    cout << "Kilkist sliv u riadku=" << res <<
endl;
    system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
vvedit riadok S:
LvDUVS hotuje fahivciv dlia pidrozdiliv National
Police of Ukraine.
Vvedenyj riadok S:
LvDUVS hotuje fahivciv dlia pidrozdiliv National
Police of Ukraine.
Kilkist sliv u riadku=9
```

Приклад 5.9. Ввести рядок символів і замінити послідовність символів ‘**...**’ на ‘**.**’.

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
int main()
{
// заміна послідовності символів у рядку
// рядок символів
    char str[50];
    int i, j;
```

```
// допоміжний масив
    char str2[50];
// довжина початкового рядка str
    int d;
    cout << "\nVvedit' riadok\n";
    gets_s(str, 50);
// поточна позиція в масиві str2
    j = 0;
    d = strnlen_s(str, 50);
    for (i = 0; i < d; i++)
        if (i < d - 2)
        {
            if ((str[i] == '.') && (str[i + 1] ==
'.') && (str[i + 2] == '.'))
            {
                str2[j++] = '.';
                i += 2;
            }
            else
                str2[j++] = str[i];
        }
// додати символ '\0' кінця рядка
    str2[j] = '\0';
cout << "\n Pochatkovyj riadok: " << str << endl;
//скопіювати допоміжний масив у початковий
    for (i = 0; i < 50; i++)
        str[i] = str2[i];
    cout << "\n Peretvorenyj riadok: " << str2 <<
endl;
    system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok
End... End. Finish.
Pochatkovyj riadok: End... End. Finish.
Peretvorenyj riadok: End. End. Finis
```

Подамо деякі пояснення до ходу виконання програмного коду. Отже, вводиться допоміжний масив `str2` такої ж розмірності, як і початковий `str`. У початковому масиві відбувається перевірка на наявність послідовності символів **‘...’**

```
if ((str[i]=='.')&&(str[i+1]=='.')&&(str[i+2]=='.'))
```

Якщо така послідовність знайдена, тоді в допоміжний масив записується символ ' . ', а поточне значення лічильника символів початкового масиву зсувається на дві позиції

```
str2[j++] = '.';
i+=2;
```

В іншому випадку, символ з початкового масиву копіюється у допоміжний.

```
str2[j++] = str[i];
```

Приклад 5.10. Задано деякий рядок символів. Створити новий рядок, який утворено із цього читанням з кінця до початку.

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
int main()
{
    // заміна послідовності символів у рядку
    // рядок символів
    char str[50];
    // допоміжний масив
    char str1[50];
        cout << "\nVvedit' riadok\n";
    gets_s(str, 50);
    cout << "\n Pochatkovyj riadok: " << str <<
endl;
    //довжина початкового рядка str
    int d;
    d = strnlen_s(str, 50);
    for (int i = 0; i < d; i++)
    // інвертуємо рядок str
        str1[i] = str[d - i - 1];
        str1[d] = '\0';
        cout << "\n Peretvorenyj riadok: \n" <<
str1 << endl;
    system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok
```

```
Vitannia studentam LvDUVS

 Pochatkovyj riadok: Vitannia studentam LvDUVS

 Peretvorenyj riadok:
SVUDvL matneduts ainnatiV
```

Приклад 5.11. Задано рядок символів. У заданому тексті визначити позицію першої крапки ‘ **.** ‘. Вважати, що перший символ у рядку має позицію 1.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
// пошук першого входження символу в тексті
	char str[50]; // заданий текст
	int i;
// шукане значення - позиція
	int poz;
	cout << "\nVvedit' riadok\n";
// ввід масиву str
	gets_s(str, 50);
	cout << "\n Pochatkovyj riadok: " << str << 
endl;
	poz = -1;
	for (i = 0; i < strnlen_s(str, 50); i++)
		if (str[i] == '.')
		{
		poz = i + 1; // запам'ятати позицію
		break;
		}
	if (poz == -1)
		cout << "\nSymvol krapka vidsutnyj." << 
endl;
	else
		cout << "\nPozycija pershoji krapky: " 
<< poz << endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Vvedit' riadok
LvDUVS - prestyzhno. NU LP - kruto.

 Pochatkovyj riadok: LvDUVS - prestyzhno. NU LP -
kruto.
 Pozycija pershoji krapky: 20
```

Виконавши, у процесі навчання, запропоновані вашій увазі приклади, ви будете спроможні самостійно дати відповідь на запитання, чому початкове значення `poz` дорівнює -1?

Отже, вважаємо, що подали достатню кількість типових прикладів впорядкування елементів масивів символів і це дасть можливість опанувати викладений матеріал на достатньому рівні.

Контрольні запитання

1. Для чого потрібне під’єднання директиви

```
#include <cstring>
```

та

```
using namespace std;
```

до програмного коду?

2. Який формат оголошення змінних типу **`string`**?
3. Як можна відводити пам’ять для масиву рядків?
4. Як здійснюється доступ до символів рядка?
5. Яким символом завершується рядок у С++?
6. Способи ініціалізування рядків при оголошенні.

Розділ
6

МАСИВИ ЯК ПАРАМЕТРИ ФУНКЦІЙ

Перед тим, як викласти основний матеріал розділу, хочемо нагадати читачам деякі аспекти формування і використання функцій.

Із поняттям функції у мові С++ пов'язано три головні її компоненти:

– прототип функції;
– означення функції;
– виклик функції.

Оголошення функцій здійснюється прототипами – короткими деклараціями, які вказують лише тип результату та перелік типів формальних параметрів без оголошень і настанов тіла функцій.

Прототип функції виконує два завдання. Перше завдання полягає в ідентифікуванні типу значення, яке поверне функція, так, щоб компілятор міг згенерувати коректний код для типу даних, які повертаються. Друге– у визначенні типу та кількості параметрів, які використовуються функцією. Формат прототипу такий:

```
<тип> <ім'я_функції>
(<перелік формальних параметрів>);
```

Прототипи здебільшого розташовують перед головною програмою або у заголовному файлі.

При передаванні параметрів у функцію додатково виконується приведення типів фактичних параметрів до типу, який вказано у прототипі функції. Прототипи є потужним доповненням до С++. Вони дають змогу компілятору виявляти помилки, яких можна припуститися, використовуючи функції. Якщо їх не виявити вчасно, вони перетворяться у проблеми, які можуть виявитися надто важкими для відстеження.

Чи зобов'язаний розробник програмного коду застосовувати прототипи? Ні – існує спосіб уникнути прототипу, одночасно зберігши переваги прототипіювання. Оголошення прототипу є повідомленням

компілятору про те, як повинна використовуватися функція до першого фактичного звернення до неї.

Того ж результату можна досягти, оголосивши повне означення функції до першого звернення до неї. У цьому разі означення діє як власний прототип. Найчастіше це робиться з короткими функціями.

Одна із сильних сторін C++ пов'язана з тим, що вона дає змогу керувати тонкими аспектами програми. Система керування пам'яттю в C++ слугує ілюстрацією такого керування, даючи змогу визначати, яким функціям відомі ті чи інші змінні та час існування змінних у програмі.

У кожному прикладі програмних кодів, які тут подаються, дані зберігаються в пам'яті. Для цього існує апаратний аспект – довільне збережене значення знаходиться у фізичній пам'яті. У C++ для описування такої ділянки пам'яті застосовується термін об'єкт. Об'єкт може зберігати одне або декілька значень. У певний момент об'єкт може поки не містити збереженого значення, але він буде мати правильний розмір для відповідного значення.

Є також і програмний аспект – програмі потрібен спосіб доступу до об'єкта. Цього можна досягти, наприклад, шляхом оголошення змінної.

Оголошення ініціює створення ідентифікатора. Ідентифікатор є ім'ям, яке можна застосувати для позначення вмісту окремого об'єкта. Ім'я змінної – не єдиний метод позначення об'єкта. Наприклад, до нього можна звернутися за адресою через вказівник.

Ідентифікатор, який використовується для доступу до об'єкта, може бути описаний за допомогою його області видимості, що вказує, у якій частині програми цей ідентифікатор дозволено застосовувати. Область видимості описує ділянку або ділянки програми, де можна звертатися до ідентифікатора.

Змінна в C++ має одну з таких областей видимості: у межах блоку, у межах функції, у межах прототипу функції і в межах файлу. Блок – це частина коду, яка розташована між відкритою фігурною дужкою і відповідною їй закритою дужкою. Наприклад, блоком є все тіло функції. Змінна, визначена у блоці, має область видимості в межах блоку, і її "видно" від місця, де вона визначена, до кінця блоку, який містить визначення. Крім того, формальні параметри функції, хоча вони записуються до відкритої фігурної дужки функції, мають область видимості в межах блоку і належать блоку, який містить тіло функції.

Область видимості в межах функції застосовується тільки до позначок, які використовуються з настановами `goto`. Це означає, що якщо позначка вперше з'являється у внутрішньому блоці функції, її область видимості поширюється на всю функцію.

Область видимості в межах прототипу функції застосовується до імен змінних, які використовуються у прототипах функцій. Область видимості в межах прототипу функції поширюється від місця визначення змінної до кінця оголошення прототипу. Це означає, що під час оброблення аргументу прототипу функції компілятор цікавить тільки тип аргументу. Якщо вказані імена, тоді зазвичай вони не мають ніякого значення і не обов’язково повинні збігатися з іменами, які застосовуються у означенні функції.

Змінна, визначена у межах довільної функції, має область видимості в межах файлу. Змінну, яка має область видимості в межах файлу, ″видно″ від місця її визначення і до кінця файлу, який містить її визначення.

6.1. Одновимірні масиви як параметри функцій

Масиви можна використовувати як параметри функцій. Нагадаємо, що передавання аргументів у функцію здійснюється:

– за значенням;
– за посиланням;
– за вказівником.

Про виклик функції з передаванням параметрів за значенням йшлося у дев’ятому розділі першої частини посібника. Цей метод виклику функції полягає у тому, що значення аргументу копіюється як формальний параметр функції. Зміни значення цього параметра всередині функції не впливають на значення змінних, які використовувалися для виклику.

Під час передавання параметрів за значенням у функцію передаються їх копії, тому функція не може змінити значення цих аргументів.

Проте передаванням параметрів через посилання або вказівник можна змінювати значення аргументу всередині функції. Вказівники передаються так само, як і інші типи.

У C++ масиви завжди передаються у функцію через вказівники. Це пов’язано з тим, що ім’я масиву є вказівником на адресу його першого елемента.

Разом із вказівником на масив потрібно передавати кількість елементів масиву, тому що вказівник на масив визначає адресу його першого елемента, а обсяг пам’яті, який займає масив, визначатиметься кількістю елементів. Використання адрес елементів масиву забезпечує можливість безпосередньо змінювати їх вміст.

6.2. Виклик функцій з аргументом у вигляді масиву

Коли масив використовується як аргумент функції, передається лише адреса масиву, а не копія всього масиву. Коли викликається функція з аргументом у вигляді масиву, передається лише адреса першого елемента масиву. Оскільки ім'я масиву – це адреса його першого елемента, фактичний параметр у вигляді імені масиву вимагає, щоб відповідний формальний параметр був вказівником.

У цьому і тільки в цьому контексті C++ інтерпретує **int** mas[] як **int** *mas, тобто mas є типом вказівника на **int**. Отже, чотири подані прототипи є еквівалентними:

```
int sum (int *mas, int n);
int sum (int *, int);
int sum (int mas[], int n);
int sum (int [], int);
```

Для того, щоб викликати функцію з параметром масивом необхідно вказати як параметр функції тільки назву масиву без дужок:

```
ZminaMass(mas, n);
```

У означенні функції параметр **int** mas[] визначає вказівник на масив, параметр **int** n – кількість елементів масиву, які будуть оброблятися. Розмір масиву в квадратних дужках писати не потрібно, компілятор його буде ігнорувати.

Можна використовувати масиви як фіксованого розміру, так і невизначеного (масиви змінної довжини). При застосуванні масивів фіксованої довжини у прототипі функції указується тип масиву і його розмір, наприклад:

```
void sort(int mas[30]);
```

Якщо описується функція з масивом змінної довжини, тоді у прототипі функції вказується тип масиву невизначеного розміру і обов'язково ще один параметр, за допомогою якого задається розмірність масиву.

```
void sort(int mas[],int n);
```

Наприклад, утворити масив з елементів цілого типу, які генеруються функцією **rand**() і використавши розроблену функцію збільшити кожен елемент масиву на 2. Перетворений масив вивести у консольний додаток.

Приклад проєкту програмного коду:

```
#include <iostream>
using namespace std;
int const n = 7;
```

```
//оголошення функції
//(прототип)
void ZminaMass(int mas[], int n);
int main()
{
//оголошення масиву
	int mas[n];
//формування масиву
	for (int i = 0; i < n; i++)
		mas[i] = rand() % 20;
	cout << "\nVvedenyj masyv" << endl;
	for (int i = 0; i < n; i++)
		cout << mas[i] << " ";
	cout << endl;
//виклик функції
	ZminaMass(mas, n);
	cout << "\nPeretvorenyj masyv: " << endl;
	for (int i = 0; i < n; i++)
		cout << mas[i] << " ";
	cout << endl;
	return 0;
}
// означення функції:
void ZminaMass(int mas[], int n)
{
	for (int i = 0; i < n; i++)
		mas[i] += 2;
}
```

Результат виконання програмного коду:

```
Vvedenyj masyv
1 7 14 0 9 4 18

Peretvorenyj masyv:
3 9 16 2 11 6 20
```

Усі масиви у функції передаються за адресою (як вказівники), тому у разі модифікування значень елементів масивів у функції ці зміни зберігаються при поверненні у викликаючу функцію.

Передавання масиву як **int** *A розглянемо таким прикладом.

Отримати масив A цілого типу розмірності n з випадкових чисел і обчислити суму елементів утвореного масиву.

Тепер передавання масиву **int** *A у функцію здійснимо двома способами і проілюструємо таким проєктом програмного коду:

```
#include <iostream>
```

```
using namespace std;
int const n = 5;
//оголошення функції
//(прототип)
int SumA(int, int *X);

int main()
{
    int SA;
//оголошення масиву
    int A[n];
//формування масиву
    for (int i = 0; i < n; i++)
        A[i] = rand() % 9;
    cout << "\nVvedenyj masyv" << endl;
    for (int i = 0; i < n; i++)
        cout << A[i] << " ";
    cout << endl;
    SA = SumA(n, A);
    cout << "\nSuma eltmtntiv (1 variant) SA = " <<
SA;
    SA = SumA(n, &A[0]);
    cout << "\nSuma eltmtntiv (2 variant) SA = " <<
SA;
    cout << "\nOK! Rezul'tat odnakovyj "  << endl;
    system("pause");
    return 0;
}
// означення функції:
// масив передаєтсья як *X
int SumA(int n, int *X)
{
    int i;
// сума
    int suma = 0;
    for (i = 0; i < n; i++)
        suma += X[i];
    return suma;
}
```

Результат виконання програмного коду:

```
Vvedenyj masyv
5 8 7 4 8
Suma eltmtntiv (1 variant) SA = 32
Suma eltmtntiv (2 variant) SA = 32
OK! Rezul'tat odnakovyj
```

Обчислення суми елементів масиву реалізовується зверненням до функції `SumA` другим параметром якої, у першому випадку, є вказівник (ідентифікатор масиву), а у другому випадку – параметром є адреса першого елемента масиву, яка отримана використанням оператора отримання адреси. У обидвох випадках це буде адреса першого `A[0]` елемента масиву.

6.3. Двовимірні масиви як параметри функцій

Параметрами функцій можуть бути не тільки одновимірні, але й багатовимірні. У цьому разі використовують масиви і фіксованої розмірності, і невизначеної довжини.

У прототипі функції під час роботи з багатовимірним масивом фіксованого розміру, наприклад матриці matr(7, 10), вказуються розмірності масиву, тобто:

```
void fun1(int matr[7][10]);
```

Якщо застосовується багатовимірний масив невизначеної довжини, тоді невизначеним може бути тільки один вимір розмірності, наприклад:

```
void fun2(int matr[][10], int rows);
```

Формуючи функцію, яка приймає як аргумент двовимірний масив, необхідно пам'ятати, що ім'я масиву трактується як адреса його початкового елемента, тому відповідний формальний параметр є вказівником – так само, як і у випадку з одновимірним масивом. Складність полягає в тому, щоб правильно оголосити вказівник. Припустімо, що ви починаєте з такого коду:

```
int array[3][4] =
{
{1,2,3,4}, {9,8,7,6}, {2,4,6,8}
};
int total = sum(array, 3);
```

Як повинен виглядати прототип `sum()`? І чому у функцію передається кількість рядків (3), але не передається кількість стовпців (4)?

Отже, `array` – ім'я масиву з трьох елементів. Перший елемент сам собі є масивом з чотирьох значень типу `int`. Тобто тип `array` – це вказівник на масив з чотирьох **`int`**-значень, тому відповідний прототип повинен бути таким:

```
int sum(int (*array)[4], int size);
```

Дужки необхідні тому, що вказане далі оголошення визначило б масив з чотирьох вказівників на **int** замість одного вказівника на масив з чотирьох **int**, а параметр функції не може бути масивом:

```
int *array[4]
```

Альтернативний формат:

```
int sum(int array[][4], int kr);
```

І той, і інший прототип встановлює, що `array` – це вказівник, а не масив. Також зауважте, що тип вказівника свідчить про те, що він вказує на масив з чотирьох **int**. Отже, тип вказівника задає кількість стовпців – ось чому кількість стовпців не передається окремим аргументом функції.

Оскільки тип вказівника задає кількість стовпців, функція `sum()` працює тільки з масивами з чотирьох стовпців. Однак кількість рядків задається змінною `kr`, тому `sum()` може працювати з довільною кількістю рядків:

```
int a[100][4];
int b[6][4];
…
int total1 = sum(a, 100); // сума всіх елементів a
int total2 = sum(b, 6);   // сума всіх елементів b
int total3 = sum(a, 10);
// сума перших 10 рядків a
int total4 = sum(a + 10, 20);
// сума наступних 20 рядків a
```

Можливий варіант означення функції `sum()`:

```
int sum(int array[][4], int kr)
{
int total = 0;
for (int i = 0; i < kr; i++)
for (int j = 0; j < 4; j++)
total += array[i][j];
return total;
}
```

Наприклад, обчислити суму елементів матриці.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
const int m = 4;
int sum(int array[][m], int kr);
int main()
{
```

```
	int a[3][m] =
	{
	{ 1, 2, 3, 4 },
	{ 9, 8, 7, 6 },
	{ 2, 4, 6, 8 }
	};
	int b[5][m] =
	{
	{ 1, 2, 3, 4 },
	{ 9, 8, 7, 6 },
	{ 2, 4, 6, 8 },
	{ 1, 2, 0, 7 },
	{ 1, 2, 0, 7 }
	};
	cout << "\nMasyv a\n";
	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < m; j++)
		cout << a[i][j] << " ";
				cout << endl;
	}
	cout << "\nMasyv b\n";
	for (int i = 0; i < 5; i++)
	{
		for (int j = 0; j < m; j++)
			cout << b[i][j] << " ";
		cout << endl;
	}
// сума всіх елементів  масиву a
	int totala = sum(a, 3);
// сума всіх елементів масиву b
	int totalb = sum(b, 5);
	cout << "\nSuma elementiv masyvu a = " <<
totala;
	cout << "\nSuma elementiv masyvu b = " <<
totalb << endl;
	system("pause");
	return 0;
}
int sum(int array[][m], int kr)
{
	int total = 0;
	for (int i = 0; i < kr; i++)
		for (int j = 0; j < m; j++)
			total += array[i][j];
```

```
	return total;
}
```

Результат виконання програмного коду:

```
Masyv a
1 2 3 4
9 8 7 6
2 4 6 8

Masyv b
1 2 3 4
9 8 7 6
2 4 6 8
1 2 0 7
1 2 0 7

Suma elementiv masyvu a = 60
Suma elementiv masyvu b = 80
```

Кількість стовпців масиву `array[][m]` визначається константою `m`, а кількість рядків `kr` передається у функцію як параметр значення, отже, не визначається на рівні введення початкових даних.

6.4. Приклади використання масивів як параметрів функцій

Приклад 6.1. Обчислити найменший елемент матриці (параметри матриці задати самостійно) і його індекси.

Для отримання числових значень індексів мінімального елемента `min_i` та `min_j` використаємо параметри за посиланням, а властиво значення мінімального елементу `min` передемо у викликаючу функцію настановою **`return`**.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
double MIN(double a[4][3], int &imin, int &jmin)
{
	double min = a[0][0];
	imin = 0;
	jmin = 0;
	for (int i = 0; i < 4; i++)
		for (int j = 0; j < 3; j++)
			if (a[i][j] < min)
```

```
                        {
                              min = a[i][j];
                              imin = i;
                              jmin = j;
                        }
      imin++;
      jmin++;
      return min;
}
int main()
{
      double a[4][3], min;
      int i, j, min_i, min_j;
      cout << " Utvoryty matryciu z 4-x riadkiv i 
3-x stovpciv:" << endl;
      for (int i = 0; i < 4; i++)
      {
            for (int j = 0; j < 3; j++)
            {
                  a[i][j] = rand() % 11 - 6;
                  cout << a[i][j] << " ";
            }
            cout << endl;
      }
      min = MIN(a, min_i, min_j);
      cout << "\nminimal'nyj element " << min << 
"\nrozmishcheno u "
            << min_i << "-mu ridku i " << min_j << 
"-mu stovpci" << endl;
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
Utvoryty matryciu z 4-x riadkiv i 3-x stovpciv:
2 3 3
-5 1 -1
-1 4 -5
-6 1 1

minimal'nyj element -6
rozmishcheno u 4-mu ridku i 1-mu stovpci
```

Масив `a[4][3]` передається у функцію фіксованої розмірності, тобто кількість рядків і кількість стовпців визначені на рівні числових значень.

Функція повинна бути побудована у такий спосіб, щоб вона обчислювала і повертала три значення. У цьому разі треба передати до функції два додаткових параметри `min_i` та `min_j` оголошені у головній функції `main()`, в які буде записано значення обидвох індексів мінімального елемента у звичній формі, коли нумерація індексів починається не з нуля, а з одиниці. Для цього в означенні функції `MIN` використано настанови

```
imin++;
jmin++;
```

Ці параметри мають передаватися за посиланням (або за вказівником), щоб у тілі функції можна було їх змінити і повернути ці зміни у точку виклику.

Тобто у функцію передаються адреси початкових байтів `min_i` та `min_j`, які використовуються для доступу до пам'яті, коли відбувається модифікування значень цих змінних всередині функції. Головна функція і розроблена функція доступаються до тих самих адрес, де зберігаються значення `min_i` та `min_j`.

Значення найменшого елемента матриці повертається у викликаючу функцію настановою

```
return min;
```

Приклад 6.2. Задано матриці A(5, 5), B(5, 5). Розробити проєкт програмного коду обчислення добутку C = A * B, використавши розроблену функцію множення матриць.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int const m = 5;
void DobMatr(int ar1[][m], int ar2[][m], int
ar3[][m]);
int main()
{
      int A[m][m], B[m][m], C[m][m];
      cout << "****************" << endl;
      for (int i = 0; i < m; i++)
      {
            for (int j = 0; j < m; j++)
            {
                  A[i][j] = rand() % 5;
                  cout << A[i][j] << " ";
            }
```

```
		cout << endl;
	}
	cout << "****************" << endl;
	for (int i = 0; i < m; i++)
	{
		for (int j = 0; j < m; j++)
		{
			B[i][j] = rand() % 6;
			cout << B[i][j] << " ";
		}
		cout << endl;
	}
	DobMatr(A, B, C);
	cout << "\nRezultujucha matrycia" << endl;
	for (int i = 0; i < m; i++)
	{
		for (int j = 0; j < m; j++)
			cout << C[i][j] << "\t";
		cout << endl;
	}
	system("pause");
	return 0;
}
void DobMatr(int ar1[][m], int ar2[][m], int
ar3[][m])
{
	for (int i = 0; i < m; i++)
	for (int j = 0; j < m; j++)
		{
		ar3[i][j] = 0;
		for (int k = 0; k < m; k++)
		ar3[i][j] += ar1[i][k] * ar2[k][j];
		}
}
```

Результат виконання програмного коду:

```
****************
1 2 4 0 4
4 3 3 2 4
0 0 1 2 1
1 0 2 2 1
1 4 2 3 2
****************
4 3 2 2 5
5 0 5 0 3
```

```
4 5 1 1 0
5 3 2 3 3
2 3 1 5 4

Rezultujucha matrycia
38      35      20      26      27
61      45      34      37      51
16      14      6       12      10
24      22      9       15      15
51      28      32      23      34
```

Приклад 6.3. Завдання те ж, що і у попередньому прикладі, але матриці оголошено як динамічні масиви.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;

int **DobMatrix(int **X, int **Y, int m)
{
    int **result = new int *[m];
    for (int i = 0; i < m; i++)
        result[i] = new int[m];
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
        {
            result[i][j] = 0;
            for (int k = 0; k < m; k++)
                result[i][j] += X[i][k] * Y[k][j];
        }
    return result;
}
int main()
{
    int m;
    cout << "\nVvesty poriadok marix" << endl;
    cin >> m;
    int **A = new int *[m];
    for (int i = 0; i < m; i++)
        A[i] = new int[m];
    cout << "Matrix A : " << endl;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
```

```
            A[i][j] = rand() % 5;
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
    cout << "*****************" << endl;
    int **B = new int *[m];
    for (int i = 0; i < m; i++)
        B[i] = new int[m];
    cout << "Matrix B : " << endl;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
        {
            B[i][j] = rand() % 5;
            cout << B[i][j] << " ";
        }
        cout << endl;
    }
    cout << "*****************" << endl;
    int **C = new int *[m];
    for (int i = 0; i < m; i++)
        C[i] = new int[m];
    C = DobMatrix(A, B, m);
    cout << "Matrix C : " << endl;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < m; j++)
            cout << C[i][j] << "\t";
        cout << endl;
    }
    for (int i = 0; i < m; i++)
    {
// вивільнення пам'яті від рядків матриці
        delete[] A[i];
        delete[] B[i];
        delete[] C[i];
    }
// вивільнення пам'яті від допоміжного масиву
    delete[] A;
    delete[] B;
    delete[] C;
    return 0;
}
```

Результат виконання програмного коду:

```
Vvesty poriadok marix
5
Matrix A :
1 2 4 0 4
4 3 3 2 4
0 0 1 2 1
1 0 2 2 1
1 4 2 3 2
****************
Matrix B :
2 1 1 3 0
2 1 1 3 4
2 2 4 0 4
3 1 2 3 3
4 1 1 3 3
****************
Matrix C :
30      15      23      21      36
42      19      27      39      42
12      5       9       9       13
16      8       14      12      17
31      14      21      30      39
```

У поданому прикладі матриця `result` оголошена динамічно (як вказівник), тому результат виконання функції `DobMatrix` передається у точку виклику (у головну функцію) настановою

```
return result;
```

Тобто, усі необхідні перетворення з елементами матриці `result` відбуваються у тілі функції `DobMatrix`, а результат перетворень передається у точку виклику ваказівником, значення якого є скалярним значенням, адресою першого байту елемента `result[0][0]`.

Також зауважимо, що вказівник на функцію – це адреса, де зберігається скомпільований код цієї функції, тобто адреса, за якою передається управління, коли ця функція викликається. Так само, як ім’я масиву є константним вказівником на перший елемент масиву, ім’я функції можна розглядати як константний вказівник на функцію. У нашому прикладі це

```
int **DobMatrix(int **X, int **Y, int m)
```

Приклад 6.4. Обчислити матричний вираз $S=2AB + 3(AB^T - EA)$. A, B – цілочисельні матриці п’ятого порядку. Для формування матриць A та B використати стандартну функцію **`rand`**`()`. E – одинична мат-

риця п'ятого порядку. Обчислення суми, різниці, добутку, транспонування матриць оформити у вигляді відповідних функцій.

Виконання розрахунків передбачає формування проміжних матриць, яким присвоєно такі символічні імена:

A2 – 2*A;
AB2 - 2*A*B;
BT – B^T;
ABT – A*B^T;
EA – E*A;
ABTEA - (AB^T – EA);
ABTEA3 - 3(AB^T - EA);

Приклад програмного коду реалізування задачі

```
#include<iostream>
using namespace std;
int const m = 5;
void DobMatr(int X[][m], int Y[][m], int Z[][m]);
void kMatr(int k, int X[][m], int Y[][m]);
void TranspMatr(int X[][m], int XT[][m]);
void SumaMatr(int X[][m], int Y[][m], int Z[][m]);
void RiznMatr(int X[][m], int Y[][m], int Z[][m]);
int main()
{
      int A[m][m], B[m][m], E[m][m] = {};
      int A2[m][m], BT[m][m], EA[m][m],
      ABT[m][m], AB2[m][m], ABTEA[m][m],
      S[m][m], ABTEA3[m][m];
      cout << "****************" << endl;
      cout << "Matrix A : " << endl;
      for (int i = 0; i < m; i++)
      {
            for (int j = 0; j < m; j++)
            {
                  A[i][j] = rand() % 5 - 2;
                  cout << A[i][j] << " ";
            }
            cout << endl;
      }
      cout << "****************" << endl;
      cout << "Matrix B : " << endl;
      for (int i = 0; i < m; i++)
      {
```

```
		for (int j = 0; j < m; j++)
		{
			B[i][j] = rand() % 6 - 3;
			cout << B[i][j] << " ";
		}
		cout << endl;
	}
	for (int i = 0; i < m; i++)
		E[i][i] = 1;
	cout << "*****************" << endl;
	cout << "Matrix E : " << endl;
	for (int i = 0; i < m; i++)
	{
		for (int j = 0; j < m; j++)
			cout << E[i][j] << "\t";
		cout << endl;
	}
	kMatr(2, A, A2);
	DobMatr(A2, B, AB2);
	TranspMatr(B, BT);
	DobMatr(A, BT, ABT);
	DobMatr(A, E, EA);
	RiznMatr(ABT, EA, ABTEA);
	kMatr(3, ABTEA, ABTEA3);
	SumaMatr(AB2, ABTEA3, S);
	cout << "*****************" << endl;
	cout << "\nMatrix S:" << endl;
	for (int i = 0; i < m; i++)
	{
		for (int j = 0; j < m; j++)
			cout << S[i][j] << "\t";
		cout << endl;
	}
	system("pause");
	return 0;
}
void DobMatr(int X[][m], int Y[][m], int Z[][m])
{
	for (int i = 0; i < m; i++)
		for (int j = 0; j < m; j++)
		{
		Z[i][j] = 0;
```

```
            for (int k = 0; k < m; k++)
                  Z[i][j] += Y[i][k] * X[k][j];
            }
}
void kMatr(int k, int X[][m], int Y[][m])
{
      for (int i = 0; i < m; i++)
            for (int j = 0; j < m; j++)
                  Y[i][j] = k * X[i][j];
}
void SumaMatr(int X[][m], int Y[][m], int Z[][m])
{
      for (int i = 0; i < m; i++)
            for (int j = 0; j < m; j++)
                  Z[i][j] = X[i][j] + Y[i][j];
}
void RiznMatr(int X[][m], int Y[][m], int Z[][m])
{
      for (int i = 0; i < m; i++)
            for (int j = 0; j < m; j++)
                  Z[i][j] = X[i][j] - Y[i][j];
}
void TranspMatr(int X[][m], int XT[][m])
{
      for (int i = 0; i < m; i++)
            for (int j = 0; j < m; j++)
                  XT[i][j] = X[j][i];
}
```

Результат виконання програмного коду:

```
*****************
Matrix A :
-1 0 2 -2 2
2 1 1 0 2
-2 -2 -1 0 -1
-1 -2 0 0 -1
-1 2 0 1 0
*****************
Matrix B :
1 0 -1 -1 2
2 -3 2 -3 0
1 2 -2 -2 -3
```

```
2 0 -1 0 0
-1 0 -2 2 1
****************
Matrix E :
1       0       0       0       0
0       1       0       0       0
0       0       1       0       0
0       0       0       1       0
0       0       0       0       1
****************

Matrix S:
3       -2      9       -3      11
-54     -26     -20     -8      -32
66      26      21      -10     38
-6      25      1       4       -5
16      22      21      -6      17
```

Ми свідомо не подали результати проміжних розрахунків, оскільки це потребує багато місця і робить громіздким програмний код. Хоча, розв’язуючи реальні задачі, рекомендуємо на етапі відлагодження і тестування програмного коду використовувати проміжне виведення для контролю достовірності реальності результатів обчислень.

Спробуйте самостійно пояснити як реалізовується обмін даними між викликаючою функцією і розробленими функціями, якщо тип останніх оголошено як `void`.

Контрольні запитання

1. Які три головні компоненти пов’язані з поняттям функції у C++?
2. Що таке прототип функції і які завдання він виконує?
3. Як у C++ передаються масиви у функцію через вказівники?
4. У чому полягає особливість виклику функцій з аргументом у вигляді масиву?
5. Як реалізовується обмін даними між викликаючою функцією і розробленими функціями, якщо тип останніх оголошено як `void`?

Розділ 7

ФАЙЛИ

Файлами є іменовані області пам'яті на зовнішньому носії, призначені для довготривалого зберігання інформації. Файли мають **імена** та організовані в ієрархічну деревоподібну структуру з **каталогів** (тек) і простих файлів.

Для розуміння процесу роботи з файлами використаємо таку метафору: якщо уявляти собі файли як книжки (лише зчитування) і блокноти (зчитування й записування), що розміщені на полиці, то відкриття файлу – це вибір книжки або блокнота за заголовком на його обкладинці та відкриття обкладинки (на першій сторінці). Після відкриття можна читати, дописувати, викреслювати і правити записи, перегортати книжку. Сторінки можна порівняти з блоками файлу, а полицю з книжками – з каталогом.

Для доступу до даних файлу з програми у ній потрібно створити функцію відкриття або створення цього файлу, чим встановити зв'язок між ім'ям файлу і певною файловою змінною у програмі.

Доступ до даних файлу відбувається з так званої позиції зчитування/записування, яка автоматично пересувається при операціях зчитування/записування, тобто файл є видимим послідовно. Існують, щоправда, функції для довільного змінювання цієї позиції.

C++ надає засоби опрацювання двох типів файлів: **текстових** і **бінарних**. Текстові файли призначено для зберігання текстів, тобто сукупності символьних рядків змінної довжини. Кожен рядок завершується керуючою послідовністю `'\n'`, а розділювачами слів та чисел у рядку є пробіли й символи табуляції. Оскільки вся інформація текстового файлу є символьною, програмне опрацювання такого файлу полягає у читанні рядків, виокремленні з рядка слів і, за потреби, перетворюванні цифрових символьних послідовностей на числа відповідними функціями перетворювання.

Створювати і редагувати текстові файли можна не лише в програмному коді, а й у довільному текстовому редакторі, наприклад, Word, WordPad або Блокнот.

Бінарні (двійкові) файли зберігають дані у тому самому форматі, в якому вони були оголошені, а їхній вигляд є такий самий, як і в пам'яті комп'ютера, тому зникає необхідність у використанні розділювачів: пробілів, керуючих послідовностей, а отже, розмір використовуваної пам'яті порівняно з текстовими файлами з аналогічною інформацією є значно меншим.

Окрім того, немає потреби у застосуванні функцій перетворювання числових даних. Але кожне опрацювання даних бінарних файлів можливе лише за наявності програми, якій має бути ″відомо″, що саме і в якій послідовності зберігається у цьому файлі.

7.1. Робота з текстовими файлами у C-stile

Попередньо ми обговорювали, що будемо розуміти під рядками у C-*stile* чому, власне, виникла потреба у таких поясненнях? За початковим визначенням розробника C++ – це C з класами. Класи належать до понять об'єктно орієнтованого програмування і не є предметом дисципліни ″Алгоритмізація та програмування″.

Ми використовували клас `iostream` у C ++, який визначає стандартну функціональність потоків введення/виведення, включаючи **`cin`** і **`cout`**. Цей клас обмежений стандартними пристроями введення/виведення, такими як клавіатура та монітор, відповідно.

Що стосується файлових операцій, C ++ має інший набір класів, які можна використовувати.

Тому, щоб обійти використання класів з необхідними поясненнями, використано роботу з текстовими файлами у C-*stile*.

Файлове введення/виведення є частиною загальної системи введення/виведення мови C++. Для того, щоб у програмі можна було використовувати файлове введення/виведення попередньо має бути під'єднаний (явно встановлений програмістом) заголовний файл (модуль) `fstream`

```
#include <fstream>
```

У цьому модулі реалізовані основні класи для роботи з файловою системою:

– `ifstream` – використовується для організації введення з файлового потоку;

– `ofstream` – використовується для організації виведення у файл (файловий потік);

– `fstream` – використовується для забезпечення введення/виведення.

Прив'язування об'єкта одного з класів потоку до файлу або для читання, або для запису, або обох поспіль називається відкриттям файлу. Відкритий файл подається у програмному коді за допомогою цього потокового об'єкта. Отож довільна операція читання/запису, виконана з цим об'єктом потоку, буде застосована і до фізичного файлу.

У мові С файл розглядається як потік послідовності байтів. Вся робота з файлом виконується через файлову змінну-вказівник на структуру типу **FILE**.

Оголошення файлової змінної-вказівника потоку може мати вигляд:

```
FILE *f;
```

Функція **fopen**() для відкривання файлу має такий формат:

```
FILE *fopen(const char *filename, const char *mode);
```

Перший параметр **filename** визначає ім'я фізичного файлу, який відкривається і пов'язується з вказівником `f`. Другий параметр **mode** задає режим відкривання файлу, тобто визначає, які дії будуть доступні для виконання з даними у відкритому файлі (табл. 7.1).

До зазначених у таблиці 7.1 специфікаторів наприкінці або перед знаком "+" може дописуватись символ `"t"` – для текстових файлів, або `"b"` – для бінарних (двійкових) файлів.

Таблиця 7.1

Специфікатори режиму відкривання файлів

Параметр	Опис
r	відкрити файл лише для зчитування даних
r+	відкрити файл для зчитування та записування даних
a	відкрити або створити файл для записування даних у кінець файлу
a+	відкрити або створити файл для зчитування та записування даних у кінець файлу
w	створити файл для записування даних
w+	створити файл для зчитування та записування даних

Функція **fopen**() повертає вказівник на об'єкт, який керує потоком, тобто адресу пам'яті початку потоку. Наприклад, створити файл з ім'ям `Pryklad.txt` можна так:

```
f = fopen("Pryklad.txt", "w+");
```

Якщо файл не вдалося відкрити, функція **fopen()** повертає нульовий вказівник NULL. Для уникнення помилок після відкриття файлу варто перевірити, чи насправді файл відкрився:

```
if (f == NULL)
{
cout << "Файл не вдалось відкрити";
return;
}
```

Припинити роботу з файлом можна за допомогою функції

fclose(**FILE** *f);

Ця функція закриває файл, на який посилається параметр функції, зберігаючи зміни.

Подамо приклад фрагменту програмного коду відкривання текстового файлу для зчитування.

```
FILE *f;
/*Перевіряємо, чи повертає ця функція
нульовий вказівник*/
if ((f = fopen("t.txt","r+"))==NULL)
{
cout << "Неможливо відкрити файл";
return;
}
// Команди зчитування з файлу
. . .
// Закриття файлу
fclose(f);
```

З текстового файлу можна читати цілі рядки і окремі символи. Варто зауважити, що тип **FILE** * перейшов у С++ зі стандарту С, і такий файл ″розуміє″ лише C-*style* рядки, тобто рядки типу **char** *.

Зчитування рядка з текстового файлу можна виконати за використання функції **fgets**(), яка має формат:

char *fgets(char *s, int m, FILE *stream);

де:
s – рядок типу **char***;
m – кількість зчитуваних символів (байтів);
stream – вказівник на потік даних файлу.

Перевірка кінця файлу здійснюється функцією **feof**():

```
int feof(FILE *stream);
```

Розглянемо докладніше використання цих функцій:

```
char s[50];
```

```
do
{
fgets(s,50,f);
cout << s;
}
while (!feof(f));
```

Цей приклад ілюструє зчитування даних з файлу порядко́во за допомогою функції **`fgets`**`()` і виведення рядків на екран. Перший параметр функції **`fgets`**`()` – це C-*style* рядок, який читає поточний рядок текстового файлу. Зчитування рядка відбувається до появи символу кінця рядка `'\n'` або ж припиняється, коли прочитано `m-1` символ. У нашому прикладі `m = 50`.

Отже, якщо файл містить рядок, який має понад 50 символів, тоді буде прочитано лише перші 49 символів.

Водночас поточна позиція файлу залишиться у тому самому рядку і за подальшого використання функції **fgets**() читатиметься залишок рядка. Третій параметр відзначає потік, з якого здійснюється зчитування.

У цьому прикладі при зчитуванні всіх даних з файлу f використовується функція **feof**(f), яка перевіряє, чи не прочитано символ кінця файлу. Після зчитування цього символу функція **feof**(f) поверне ненульове значення – і цикл перерветься.

До речі, інколи використання функції **feof**() призводить до появи дубліката останнього рядка файлу, тому рекомендовано використовувати в умові циклу функцію зчитування даних, наприклад **fgets**(), або контролювати добігання кінця файлу, тобто перевіряти, чи прочитано всі записані у файлі символи (нагадаємо, що 1 символ – 1 байт). Поданий приклад зчитування рядків з файлу можна тепер записати у такий спосіб:

```
while (fgets(s,50,f))
{
if (s[strlen(s)-1]=='\n') s[strlen(s)-1] = '\0';
puts(s);
}
```

Зчитування форматованих даних можна також здійснювати за допомогою функції **`fscanf`**`()`:

```
int fscanf(FILE *stream, const char *format[,
address.]);
```

Подамо приклад використання цієї функції:

```
float r;
while (fscanf(f,"%e",&r)>0 )
cout << r;
```

У цьому прикладі здійснюється зчитування даних з файлу, який містить дійсні числа. За розділювачі поміж числами вважаються пробіли. Перший параметр `f` функції **`fscanf`**`()` визначає файл, з якого відбувається зчитування. Другий параметр задає формат рядка аргументів, заданих їхніми адресами.

Функція **`fscanf`**`()` є цілого типу і повертає, як своє значення, кількість прочитаних елементів. При форматованому читанні часто можуть виникати помилки через невідповідність форматів чи кінець файлу. Для уникання таких помилок доцільно організовувати перевірки кількості прочитаних функцією **`fscanf`**`()` елементів. У поданому прикладі формат `"%e"` визначає дійсне число з плаваючою крапкою. Прочитане дійсне число з файлу зберігається за адресою змінної `r` типу **`float`**.

Рядок форматування параметра `format` складається з послідовності символів-специфікаторів типів читаних даних, найпоширеніші з яких подано у таблиці 7.2.

Записувати дані у текстовий файл можна за допомогою функції:

```
int fputs(const char *s, FILE *stream);
```

де `s` – рядок типу **`char`**;
`stream` – файловий потік.

Таблиця 7.2

Специфікатори параметра `format`

Символ	Значення, що вводиться	Тип аргументу
`i, l`	десяткове, вісімкове чи шістнадцяткове ціле число	`int, long`
`d, D`	десяткове ціле число	`int, long`
`e, E`	дійсне число з плаваючою крапкою	`float`
`f, F`	дійсне число з фіксованою крапкою	`float`
`s`	рядок символів	`char s[]`
`c`	символ	`char`

Для записування даних до файлу треба відкрити файл у форматі записування даних у кінець файлу, за потреби перетворити рядок на тип **`char`** і записати дані до файлу. Послідовність процесу показано у фрагменті програмного коду:

```
FILE *f; // Оголошення файлової змінної
f = fopen("a.txt","a+");
if (f==0)
{
cout<<"Не вдається створити файл!";
return 0;
}
```

```
char s[40];
// Введення рядка s
gets(s);
/* Долучення до рядка символу [Enter], інакше у
файлі все буде записано одним рядком*/
// Записування рядка s у файл
strcat(s,"\n");
fputs(s,f);
fclose(f);
```

Записування у текстовий файл можна здійснити також за допомогою функції **fprintf**():

```
int fprintf (FILE *stream, const char *format [,
                    argument]);
```

Ця функція є схожою до функції **`fscanf`**`()`, але має ширші можливості побудови рядка форматування, наприклад:

```
char s[20];
strcpy(s, "Петрівський");
int year=1985;
fprintf(F, "ХАРАКТЕРИСТИКА\nпрацівник %s, %i
р.н.\n", &s, year);
```

Унаслідок виконання цих команд до файлу буде записано таке:

```
ХАРАКТЕРИСТИКА
працівник Петрівський, 1985 р.н.
```

Як бінарні, так і текстові файли дають змогу переміщувати поточну позицію зчитування/записування. Для визначення поточної позиції файлу, яка автоматично зміщується на кількість опрацьованих байтів, використовується функція **`ftell`**`()`:

```
long int ftell(FILE *stream);
```

А змінити поточну позицію файлу можна за допомогою функції **fseek**():

```
int fseek(FILE *stream, long offset, int whence);
```

Ця функція задає зсув на кількість байтів offset щодо точки відліку, яка задається параметром whence. Параметр whence може набувати таких значень:

Константа	whence	Точка відліку
SEEK_SET	0	початок файлу
SEEK_CUR	1	поточна позиція
SEEK_END	2	кінець файлу

Наприклад, для переміщення поточної позиції на початок файлу можна скористатися функцією

```
fseek(f, 0, SEEK_SET);
```

або

```
fseek(f, 0, 0);
```

За допомогою функцій **ftell**() та **fseek**() можна визначити сумарний обсяг пам'яті у байтах, який займає файл. Для цього достатньо переміститися у кінець файлу:

```
fseek(f, 0, SEEK_END);
int d = ftell(f);
```

Отже, *послідовність роботи з файлом під час записування* даних є така:

1. Відкрити або створити файл fname.txt для записування (або для зчитування й записування) у кінець файлу:

```
f = fopen("fname.txt", "a+");
```

2. Записати дані у файл f однією з команд:

```
// Записування С-рядка s
fputs(s, f);
//Форматоване записування С-рядка
//nazva, дійсного числа price і цілого kilkist
fprintf(f, "%s %6.2f %i\n", nazva, price, kilkist);
```

3. Закрити файл:

```
fclose(f);
```

Для зчитування даних з файлу треба виконати таку послідовність дій:

1. Відкрити файл для зчитування

```
f = fopen("fname.txt","r+");
```

2. Здійснити зчитування даних з файлу однією з команд:

```
/*Зчитування С-рядка s
довжиною у 80 символів  */
fgets(s, 80, f);
```

3. Для зчитування послідовно всіх даних з файлу треба організувати цикл з умовою, яка перевіряє досягнення кінця файлу, використовуючи функцію **feof**(f), яка є ідентична до true при досяганні кінця файлу, або функцію зчитування даних, яка дає нульовий результат за досягнення кінця файлу. Можна застосувати інформацію про сумарний розмір файлу і контролювати досягнення кінця файлу функцією **ftell**(f).

4. Закрити файл:

```
fclose(f);
```

Недоліком функцій С порівняно з потоками С++ є те, що ці функції є небезпечними з погляду типів. Це може призвести до низки помилок під час виконання, пов'язаних з невідповідністю типів змінних. Компілятор не може перевірити відповідність під час транслювання програмного коду.

Небезпечною є функція **scanf**(). Некоректне використання цієї функції може призвести до зависання програми. Тому в сучасних версіях Visual Studio спроба скомпілювати програму, яка містить виклики **scanf**(), спричиняє виникнення помилки компіляції "'scanf': This function or variable may be unsafe". Для того, щоб скомпілювати таку програму, потрібно заборонити відповідний контроль, додавши першим рядком програми директиву препроцесору

#define _CRT_SECURE_NO_WARNINGS

Багато старих функцій CRT мають новіші, безпечніші версії. Якщо безпечна функція існує, тоді старіша, менш безпечна, версія позначається як застаріла. Нова версія має додавання до імені символів _s ("безпечний").

У цьому контексті "застаріле" означає, що використання функції не рекомендовано. Це не означає, що функція буде видалена з CRT.

Функції безпеки не запобігають і не виправляють помилки безпеки. Натомість вони виявляють помилки, коли вони виникають. Вони проводять додаткові перевірки на наявність помилок. Якщо є помилка, то вони викликають обробник помилок.

CRT – це бібліотека C Run Time. _CRT_SECURE_NO_WARNINGS означає, що ви не хочете, щоб компілятор пропонував безпечні версії бібліотечних функцій, приміром, **scanf_s**, коли ви використовуєте **scanf**.

У прикладах, які подано далі, усі ці можливості послідовно використано у проєктах програмних кодів.

Приклад 7.1. Створити у кореневій теці диска (умовно С) за допомогою програми Блокнот текстовий файл з ім'ям Myfile.txt. Створити проєкт програмного коду, який виводить вміст цього файлу у консольний додаток.

Приклад проєкту програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
int main()
```

```
{
const char *name =
"C:\\Users\\taras\\source\\Myfile.txt";
FILE* f;
char s[100];
/* Відкрити файл для читання як текстовий*/
f = fopen(name, "r+");
if (f == NULL)
{
cout << "Cannot file open\n";
system("pause");
return;
}
cout << "\nPerehliad failu" << endl;
// Зчитувати рядки із файлу,
// допоки вони не закінчаться (до кінця файлу)
/* Вилучити останній символ (інакше після кожного
рядка буде виведено порожній) */
while (fgets(s, 100, f))
{
s[strlen(s) - 1] = '\0';
// Вивести рядок у консольний додаток
puts(s);
}
fclose(f);
system("pause");
return 0;
}
```

Результати виконання програмного коду:

```
Perehliad failu
There are a large number of functions
to handle file I/O (Input Output) in C++
When dealing with files, there are two types of
files:
1. Text files
2. Binary files
```

Трішки ускладнимо задачу.

Приклад 7.2. Створити у кореневій теці диска (умовно С) за допомогою програми Блокнот текстовий файл з ім'ям `Myfile.txt` і ввести до нього кілька рядків. Вивести вміст цього файлу у консольний додаток. Також відкрити файл `Myfile1.txt` для запису, записа-

ти у нього кілька рядків і вивести його вміст у консольний додаток. Створити проєкт програмного коду для розв'язання задачі. Формування вмісту файлу `Myfile1.txt` виконати довільним текстом.

Приклад проєкту програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
int main()
{
	const char *name =
"C:\\Users\\taras\\source\\Myfile.txt";
	const char* name1 =
"C:\\Users\\taras\\source\\Myfile1.txt";
	FILE* f;
	char s[100];
	FILE* f1;
	char s1[100];
	/* Відкрити файл для читання як текстовий*/
	f = fopen(name, "r+");
	if (f == NULL)
	{
		cout << "Cannot file open\n";
		return;
	}
	cout << "\nperehliad failu Myfile.txt" <<
endl;
// Зчитувати рядки із файлу,
// допоки вони не закінчаться (до кінця файлу)
	while (fgets(s, 100, f))
	{
	/* Вилучити останній символ (інакше після
кожного рядка буде виведено порожній) */
		s[strlen(s) - 1] = '\0';
	// Вивести рядок на екран
		puts(s);
	}
	fclose(f);
	// Відкрити файл для створення як текстовий
	f1 = fopen(name1, "w+");
	// Перевірити, чи відкрився файл
```

```
	if (f1 == NULL)
	{
		cout << "Cannot file create\n";
		return;
	}
	cout << "Vvesty riadky fajlu Myfile1.txt: "
<< endl;
	// Цикл, поки не введено порожній рядок
	do
	{
	// ввести рядок із клавіатури
		gets_s(s1, 100);
	// записати його до файлу
		fputs(s1, f1);
	// і перейти на новий рядок
		fputs("\n", f1);
	} while (strcmp(s1, ""));
	// Закрити і зберегти файл
	fclose(f1);
	// Перегляд файлу Myfile1.txt
	/* Відкрити файл для читання як текстовий*/
	f1 = fopen(name1, "r+");
	if (f1 == NULL)
	{
		cout << "Cannot file open\n";
		return;
	}
	cout << "\nperehliad failu Myfile1.txt" <<
endl;
// Зчитувати рядки із файлу,
// допоки вони не закінчаться (до кінця файлу)
	while (fgets(s1, 100, f1))
	{
	//Вилучити останній символ (інакше після
	//кожного рядка буде виведено порожній)
		s[strlen(s1) - 1] = '\0';
	//Вивести рядок на екран
		puts(s1);
	}
	// Закрити і зберегти файл
	fclose(f1);
	return 0;
}
```

Результати виконання програмного коду:

```
perehliad failu Myfile.txt
There are a large number of functions
to handle file I/O (Input Output) in C++
When dealing with files, there are two types of
files:
1. Text files
2. Binary files
Vvesty riadky fajlu Myfile1.txt:
Nashi kursanty duzhe vidpovidal'ni u navchanni.
Studenty takozh.
Kafedra KIAZDPO retel'no hotuje metodychne
zabezpechennia.


perehliad failu Myfile1.txt
Nashi kursanty duzhe vidpovidal'ni u navchanni.

Studenty takozh.

Kafedra KIAZDPO retel'no hotuje metodychne
zabezpechennia.
```

Приклад 7.3. Створити в кореневій теці диска (умовно C) за допомогою програми Блокнот текстовий файл з ім’ям `Myfile.txt` і ввести до нього кілька рядків. Розробити проєкт програмного коду, який виводить вміст цього файлу на екран і обчислює кількість рядків у файлі та їхню середню довжину.

Приклад проєкту програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
int main()
{
      const char *name =
"C:\\Users\\taras\\source\\newfile.txt";
      FILE* f;
      char s[150];
      int  kilk = 0;
      double sum = 0;
```

```
	f = fopen(name, "w+");
// Перевірити, чи відкрився файл
	if (f == NULL)
	{
		cout << "Cannot newfile create\n";
		return 0;
	}
	cout << "Vvesty riadky fajlu newfile.txt: "
<< endl;
// Цикл, поки не введено порожній рядок
	do
	{
	kilk++;
// ввести рядок із клавіатури
		gets_s(s, 150);
// записати його до файлу
cout << "\nRiadok nomer " << kilk << ":  ";
puts(s);
	fputs(s, f);
	sum += strlen(s);
// і перейти на новий рядок
	fputs("\n", f);
	} while (strcmp(s, ""));
// Закрити і зберегти файл
	fclose(f);
	cout << "\nKilkist riadkiv u fajli = " <<
kilk-1 << endl;
	cout << "\nZahal'na dovzhyna riadkiv u fajli
= " << sum << endl;
/* Якщо у файлі є рядки, обчислити та вивести сере-
дню довжину рядків */
	if (kilk) cout << "\nSerednia dovzhyna
riadkiv = " << sum / (kilk-1) << endl;
	system("pause");
	return 0;
}
```

Результати виконання програмного коду:

```
Vvesty riadky fajlu newfile.txt:
There are a large number of functions
Riadok nomer 1:  There are a large number of
functions
```

```
to handle file I/O (Input Output) in C++

Riadok nomer 2:  to handle file I/O (Input Output)
in C++
When dealing with files, there are two types of
files:

Riadok nomer 3:  When dealing with files, there are
two types of files:
1. Text files

Riadok nomer 4:  1. Text files
2. Binary files

Riadok nomer 5:  2. Binary files


Riadok nomer 6:

Kilkist riadkiv u fajli = 5

Zahal'na dovzhyna riadkiv u fajli = 159

Serednia dovzhyna riadkiv = 31.8
```

Аналізуючи проєкт програмного коду, деякі запитання викликає настанова

```
cout << "\nKilkist riadkiv u fajli = " << kilk-1 <<
endl;
```

Беручи до уваги, що ознакою завершення формування файлу `newfile.txt` є введення порожнього рядка, їх у файлі виявиться на одиницю більше від кількості реальних рядків. Для розуміння цих пояснень нами залишено `Riadok nomer 6:`

Змінну `sum` (загальна кількість символів у файлі) оголошено типу `double` з тих міркувань, що частка від ділення (`sum/(kilk-1)`), у загальному випадку є середньою кількістю символів у рядках і буде дійсним числом.

Приклад 7.4. Створити текстовий файл (кінець введення – порожній рядок). Розробити проєкт програмного коду, який спочатку виводить увесь вміст файлу на екран, а тоді рядки довжиною понад 50 символів.

Для створення та переглядання файлу скористаймося функціями create_file() та view_file().

Приклад проєкту програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
// Функція створення файлу
void create_file(const char *name)
{
      FILE *f;
      char s[100];
/* Відкрити файл для створення як текстовий*/
      f = fopen(name, "w+");
// Перевірити, чи відкрився файл
      if (f == NULL)
      {
            cout << "Cannot file open w+\n";
            return;
      }
      cout << "Vvesty riadky" << endl;
// Цикл, поки не введено порожній рядок
      do
      {
// ввести рядок із клавіатури
            gets_s(s, 100);
// записати його до файлу
            fputs(s, f);
// перейти на новий рядок
            fputs("\n", f);
      } while (strcmp(s, ""));
// Закрити і зберегти файл
      fclose(f);
}
// Функція переглядання файлу
void view_file(const char *name)
{
      FILE* f;
      char s[100];
      /* Відкрити файл для читання
      як текстовий if(f==NULL)*/
```

```
	f = fopen(name, "r+");
	if (f == NULL)
	{
		cout << "Cannot file openn r+\n";
		return;
	}
	cout << "\nPerehliad fajlu" << endl;
	while (fgets(s, 100, f))
	{
		s[strlen(s) - 1] = '\0';
		puts(s);
	}
	fclose(f);
}
/* Функція виведення рядків
довжиною понад 50 символів*/
void Symbols50(const char *name)
{
	FILE* f;
	char s[100];
	f = fopen(name, "r+");
	if (f == NULL) {
		cout << "Cannot open file\n";
		return;
	}
	cout << "\nRiadky dovzhynoju ponad 50
symvoliv:" << endl;
	while (fgets(s, 100, f))
	{
		s[strlen(s) - 1] = '\0';
		if (strlen(s) > 50) puts(s);
	}
	fclose(f);
}
int main()
{
// Ім'я фізичного файлу в теці проєкту
	const char *name =
"C:\\Users\\Taras\\source\\newfile.txt";
	create_file(name);
	view_file(name);
	Symbols50(name);
```

```
	system("pause");
	return 0;
}
```

Результати виконання програмного коду:

```
Vvesty riadky
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014


Perehliad fajlu
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014


Riadky dovzhynoju ponad 50 symvoliv:
Straustrup later renamed the language to C++ in 1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
```

Приклад 7.5. Створити файл з рядків (кінець введення – порожній рядок). Розробити проєкт програмного коду для виведення слів, які починаються і закінчуються однаковим символом.

Для створення та переглядання файлу скористаймося функціями `create_file()` та `view_file()` з попереднього прикладу.

Проєкт програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
```

```
using namespace std;
// Функція створення файлу
void create_file(const char *name)
{
	FILE *f;
	char s[100];
/* Відкрити файл для створення як текстовий*/
	f = fopen(name, "w+");
// Перевірити, чи відкрився файл
	if (f == NULL)
	{
		cout << "Cannot file open w+\n";
		return;
	}
	cout << "Vvesty riadky" << endl;
// Цикл, поки не введено порожній рядок
	do
	{
// ввести рядок із клавіатури
		gets_s(s, 100);
// записати його до файлу
		fputs(s, f);
// перейти на новий рядок
		fputs("\n", f);
	} while (strcmp(s, ""));
// Закрити і зберегти файл
	fclose(f);
}
// Функція переглядання файлу
void view_file(const char *name)
{
	FILE *f;
	char s[100];
/* Відкрити файл для читання як текстовий */
	f = fopen(name, "r+");
	if (f == NULL)
	{
		cout << "Cannot file openn r+\n";
		return;
	}
	cout << "\nPerehliad fajlu" << endl;
	while (fgets(s, 100, f))
	{
```

```
            s[strlen(s) - 1] = '\0';
            puts(s);
      }
      fclose(f);
}
/* Функція виведення слів, які починаються і закін-
чуються однаковими символами */
void Same_symbols(const char *name)
{
      FILE *f;
      char s[100], *t;
      f = fopen(name, "r+");
      if (f == NULL)
      {
            cout << "Cannot open file\n"; return;
      }
      cout << "\nSlova, jaki pochynajut'sia ta
zakinchujut'sia odnakovym symvolom:"
            << endl;
      while (fgets(s, 100, f) > 0)
      {
            s[strlen(s) - 1] = '\0';
            t = strtok(s, " .,;?!-");
            while (t != NULL)
            {
            if (t[0] == t[strlen(t) - 1])
                        puts(t);
                  t = strtok(NULL, " .,;?!-");
            }
      }
      fclose(f);
}
int main()
{
      const char *name =
"C:\\Users\\Taras\\source\\myfile.txt";
      create_file(name);
      view_file(name);
      Same_symbols(name);
      system("pause");
      return 0;
}
```

Результати виконання програмного коду:

```
Vvesty riadky
slon sos pip
george king words arka
lemon sosna klerk

Perehliad fajlu
slon sos pip
george king words arka
lemon sosna klerk

Slova, jaki pochynajut'sia ta zakinchujut'sia
odnakovym symvolom:
sos
pip
arka
klerk
```

Відзначимо, що функцію `strtok()` ми уже розглядалали у розділі 4.

Приклад 7.6. Створити файл з рядків (кінець введення – порожній рядок).

Розробити проєкт програмного коду для визначення кількості рядків, які містять цифри. Для створення та переглядання файлу скористаймося функціями `create_file()` та `view_file()` з попередніх прикладів.

Проєкт програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
// Функція створення файлу
void create_file(const char *name)
{
      FILE *f;
      char s[100];
/* Відкрити файл для створення як текстовий*/
/* Перевірити, чи відкрився файл    */
f = fopen(name, "w+");
      if (f == NULL)
      {
```

```
            cout << "Cannot file open w+\n";
            return;
      }
      cout << "Vvesty riadky" << endl;
// Цикл, поки не введено порожній рядок
      do
      {
// ввести рядок із клавіатури
            gets_s(s, 100);
// записати його до файлу
            fputs(s, f);
// перейти на новий рядок
            fputs("\n", f);
      } while (strcmp(s, ""));
// Закрити і зберегти файл
      fclose(f);
}
// Функція переглядання файлу
void view_file(const char *name)
{
      FILE *f;
      char s[100];
/* Відкрити файл для читання як текстовий
if(f==NULL)*/
      f = fopen(name, "r+");
      if (f == NULL)
      {
            cout << "Cannot file openn r+\n";
            return;
      }
      cout << "\nPerehliad fajlu" << endl;
      while (fgets(s, 100, f))
      {
            s[strlen(s) - 1] = '\0';
            puts(s);
      }
      fclose(f);
}
/* Функція виведення рядків з цифрами та обчислення
їх кількості. */
int digits(const char *name)
{
```

```
      FILE *f;
      char s[100];
      int k = 0;
      f = fopen(name, "r+");
      if (f == NULL)
{
cout << "Cannot open file\n";
return 0;
}
      cout << "\nRiadky, jaki mistiat' cyfry:" <<
endl;
      while (fgets(s, 100, f))
      {
            s[strlen(s) - 1] = '\0';
            for (int i = 0; i < strlen(s); i++)
                  if (s[i] >= '0' && s[i] <= '9')
                  {
                        k++;
                        puts(s);
                        break;
                  }
      }
cout << "\nKil'kist' riadkiv z cyframy: " << k <<
endl;
      fclose(f);
      }
int main()
{
      const char *name =
"C:\\Users\\Taras\\source\\myfile.txt";
      create_file(name);
      view_file(name);
      digits(name);
      system("pause");
      return 0;
}
```

Результати виконання програмного коду:

```
Vvesty riadky
C++ is a high-level programming language.
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
```

```
Straustrup later renamed the language to C++ in
1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014


Perehliad fajlu
C++ is a high-level programming language.
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014


Riadky, jaki mistiat' cyfry:
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014

Kil'kist' riadkiv z cyframy: 3
```

Приклад 7.7. Створити файл з кількох рядків (кінець введення – порожній рядок). Розробити проєкт програмного коду для копіювання вмісту цього файлу у інший файл без змін.

Проєкт програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
void Copy_file(const char *name1, const char
*name2)
{
      FILE *f1;
      FILE *f2;
```

```
      char s1[100];
      char s2[100];
/* Відкрити файл name1 для читання*/
      f1 = fopen(name1, "r+");
// Перевірити, чи відкрився файл
      if (f1 == NULL)
      {
      cout << "Cannot file f1 open r+\n";
      return;
      }
// Відкрити файл для створення як текстовий
      f2 = fopen(name2, "w+");
// Перевірити, чи відкрився файл
      if (f2 == NULL)
      {
      cout << "Cannot file f2 create w+\n";
      return;
      }
      cout << "\nPerehliad fajlu name 1" << endl;
      while (fgets(s1, 100, f1))
      {
      s1[strlen(s1) - 1] = '\0';
//формування рядка s2
      strcpy(s2, s1);
      puts(s1);
//запис рядка s2 до файлу name 2
      fputs(s2, f2);
      fputs("\n", f2);
      }
// Закрити і зберегти файли
fclose(f1);
      fclose(f2);
// Перегляд файлу name2
/* Відкрити файл name2 для читання як текстовий*/
      f2 = fopen(name2, "r+");
      if (f2 == NULL)
      {
      cout << "Cannot file f2 open r+\n";
      return;
      }
      cout << "\nPerehliad fajlu mane2" << endl;
// Зчитувати рядки із файлу,
```

```
// допоки вони не закінчаться (до кінця файлу)
      while (fgets(s2, 100, f2))
      {
//Вилучити останній символ (інакше після
//кожного рядка буде виведено порожній)
      s2[strlen(s2) - 1] = '\0';
//Вивести рядок на екран
      puts(s2);
      }
// Закрити і зберегти файл
      fclose(f2);
}
int main()
{
// Імена фізичних файлів у теці проєкту
      const char *name1 =
"C:\\Users\\Taras\\source\\Myfile.txt";
      const char *name2 =
"C:\\Users\\Taras\\source\\Myfile1.txt";
      Copy_file(name1, name2);
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
Perehliad fajlu name 1
C++ is a high-level programming language.
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014

Perehliad fajlu mane2
C++ is a high-level programming language.
Developed by Bjarn Straustrup
1979 and was originally named <C with classes>.
Straustrup later renamed the language to C++ in
1983.
```

```
It is based on the C language. First described by
the standard
ISO/IEC 14882:1998, the current standard is ISO/IEC
14882:2014
```

Ми подали і розглянули достатньо прикладів, щоб осягнути технологію роботи з файлами. Якщо ви самостійно виконали ці приклади і з’ясували з викладачем усі запитання, які виникли у процесі роботи, тоді можемо розпочати викладення наступного матеріалу.

Контрольні запитання

1. Сформулюйте поняття файлу.
2. У якому типі файлів інформація зберігається у символьному вигляді?
3. За допомогою якої функції відкривається файл?
4. Який специфікатор використовується для відкриття наявного текстового файлу при зчитуванні/записуванні?
5. За допомогою якої функції відбувається переміщення файлом?
6. Як здійснити доступ до даних файлу?
7. Як реалізовується прив'язування об'єкта одного з класів потоку до файлу або для читання, або для записування?

Розділ
8

СТРУКТУРИ

Якщо ви розпочали опрацювання цього розділу, це означає, що ви осягнули матеріал попередніх розділів і доволі багато знаєте про типи даних у С++. А тепер уявімо ситуацію, що ви самостійно можете моделювати свої типи даних, які вам потрібні і з якими зручно буде працювати. Це порівняно нескладно і варто розпочати вивчення. Набирайтеся мужності, терпцю та до успіхів.

8.1. Перейменування типів

До складу мови С++ входить спеціальний оператор typedef, який дає програмісту змогу надавати власні імена типам даних у програмі. Крім наочності та простоти іменування типів оператор typedef дає програмістові можливість створювати машино незалежні типи даних. Насамперед це стосується тих типів, параметри яких залежать від апаратних особливостей комп'ютера. Якщо такому типові надати певне typedef-ім'я і використовувати це ім'я у тексті програмного коду, то в новому середовищі достатньо внести зміни у рядок оголошення typedef, щоб встановити нові параметри для перейменованого типу.

На перший погляд, це зовсім зайве. Якщо тип уже має одне ім'я, навіщо йому інше? Здебільшого це роблять, аби полегшити подальше модифікування програмного коду. Розглянемо перейменування типів детальніше.

Отже, С++ надає можливість моделювання нових типів даних на базі вже наявних типів. Для створювання типів користувача на базі створених типів використовується оператор `typedef` (від англійської *"type definition"* – визначення типу).

У загальному вигляді для перейменування типу використовують таку синтаксичну конструкцію:

typedef <тип> <нове_ім'я_типу> [<розмірність>];

Тут квадратні дужки є елементом синтаксису. Розмірність може бути відсутньою.

Перейменування типу ″беззнаковий `char`″ на `byte`

```
typedef unsigned char byte;
```

Перейменування типу ″рядок з 50-ти символів″ на МС

```
typedef char MC[50];
```

Оголошення без перейменування:

```
char s[20]; // Слово s – рядок з 20-ти символів
char mas[10][20]; // mas – масив з 10 слів
```

Перейменування типу ″рядок з 20-ти символів″ на тип ″слово″:

```
typedef char slovo[20];
slovo s; // Оголошення змінної s типу slovo
slovo mas[10]; // і масиву з 10-ти слів
```

Оператор `typedef` є засобом спрощення запису настанов оголошення змінних.

Наприклад, деякий програмний код реалізовує обчислення з числами типу `float`. У якійсь ситуації можуть знадобитися більш точні обчислення. Якщо не використовувати перейменування типу, доведеться ″пройтися″ усім програмним кодом і замінити `float` на `double`.

А от якщо у програмному коді тип `float` був перейменований, то його псевдонім, допустимо `real`, стає деяким параметром, який можна знову перевизначити. Для того, щоб перейти до обчислень з підвищеною точністю, досить змінити лише одну настанову: замість

```
typedef float real;
```

варто записати

```
typedef double real;
```

Тоді програмний код буде виглядати так:

```
#include <iostream>
using namespace std;
int main()
{
      typedef double real;
      real a = 1.0e300, b = 1.0e200, c = a * b;
      cout << c << endl;
      system("pause");
      return 0;
}
```

Цей код має деякі особливості. Значення змінних a і b є у допустимому діапазоні значень для типу double (+/- 1.7E-308 до 1.7E+308), але їхній добуток виходить за його межі. Ви отримаєте повідомлення про помилку.

Очевидно, що необхідно розширити діапазон значень, задавши тип long double. У поданому прикладі для цього достатньо замінити рядок

```
typedef double real;
```

рядком

```
typedef long double real;
```

Тепер здобутий результат буде достовірним: 1.0e+500.

Звичайно, у такому короткому прикладі програмного коду застосування оператора typedef не надто помітне, однак у великому програмному коді, до якого входить багато файлів, він дає змогу уникнути втомлюючої перевірки і заміни оголошень змінних типу double на тип змінних long double.

Варто пам’ятати, що оператор typedef не створює новий тип, він просто створює псевдонім навяного типу. До речі, це перейменування діє у межах цілого програмного коду. Інакше кажучи, один тип можна перейменувати тільки один раз.

Розглянемо оголошення

```
double A[5][7];
```

Оголошується змінна A як матриця (двовимірний масив) дійсних чисел розмірністю 5 рядків і 7 стовпців. Якщо у програмі оголошується кілька таких масивів, або якщо у межах програмного коду використовуються функції, параметрами яких є матриця 5×7, зручніше створити окремий тип "матриця" для оголошення змінної A, щоб уникнути помилок і покращити читання програмного коду.

Отже, оголошення описаного типу matrix може мати вигляд

```
typedef double matrix[5][7];
```

Цей тип є синонімом двовимірного масиву розмірністю 5×7 дійсного типу, і його можна використовувати як звичайний тип для оголошення змінної A:

```
matrix A;
```

Змінна A має тип matrix, тобто є двовимірним масивом дійсного типу. У перейменуванні типу можна використовувати новий тип matrix для оголошення всіх дійсних двовимірних масивів 5×7, наприклад:

```
matrix B, C;      // B і C – матриці 5x7
// Заголовок функції, параметром якої є матриця 5x7
double sumr(matrix C);
/* Масив з чотирьох матриць (еквівалентне до double
M[4][5][7];) */
matrix M[4];
```

8.2. Поняття структури

Ви уже знаєте, що мова програмування C++ дає змогу створювати власні типи, використання яких може бути не менш зручним і більш виразним, ніж використання наявних стандартних типів. Як правило, у практиці програмування для створення користувацьких типів використовують класи, проте, іноді, також можна використовувати інші конструкції C++ – структури, об'днання та перелічування (переліки).

З структурами ми ознайомимося у цьому розділі, а двома іншими – об'єднаннями та перелічуваннями у наступному. Знайомство з ще одним типом користувача – класом – ми відкладемо до вивчення дисципліни "Об'єктно-орієнтоване програмування". І хоча структури та об'єднання призначено для задоволення різних програмних потреб, обидва вони є зручними засобами керування групами взаємопов'язаних змінних.

Водночас важливо розуміти, що створення структури (або об'єднання, перелічення) означає створення нового *визначеного програмістом* типу даних. Сама можливість створення власних типів даних є ознакою потужності мови C++.

У мові програмування C++ користувацькі типи мають як об'єктно-орієнтовані, так і не об'єктно-орієнтовані атрибути. У цьому і наступному розділах детально проаналізуємо тільки останні. Об'єктно-орієнтовані атрибути та їх властивості будемо розглядати після введення таких понять як класи і об'єкти.

У програмуванні *структури даних* – способи організації даних у програмних кодах. Зі структурою даних пов'язується і специфічний перелік операцій, які можуть бути виконаними над даними, організованими у таку структуру. Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх оброблення. Добре побудовані структури даних дають змогу раціонально використовувати машинний час та пам'ять комп'ютера для виконання найбільш критичних операцій.

У мові програмування C++ *структура* є колекцією змінних, об'єднаних загальним іменем, яка забезпечує зручний засіб зберігання

споріднених даних в одному місці. *Структура* – це сукупність різних типів даних, оскільки вони складаються з декількох різних, але логічно взаємопов'язаних змінних і з такою групою можна працювати як з одним цілим. З означених причин структури іноді називають *складеними* або *конгломератними типами даних.*

Перед визначенням структурних змінних, необхідно визначити шаблон структури. Це робиться за допомогою оголошення структури. Оголошення структури дає змогу компілятору зрозуміти, змінні якого типу вона містить. Змінні, які належать до структури, називаються її *елементами.* Елементи структури також називають *полями*.

У загальному випадку всі елементи структури мають бути логічно пов'язані між собою. Наприклад, структури зазвичай використовують для зберігання такої інформації, як поштові адреси, банківські реквізити, елементи книжкової бібліографії тощо. Безумовно, відносини між елементами структури абсолютно суб'єктивні і визначаються програмістом. Компілятор ″нічого про них не знає″ (або ″не хоче знати″).

Прикладом структури може слугувати довільний об'єкт, який визначає набір своїх характеристик. Наприклад, крапку на площині визначається абсцисою і ординатою, дата – днем, місяцем і роком, для книги такими характеристиками можуть бути: прізвище автора, назва, рік видання, кількість сторінок тощо.

Структура – це тип даних, який може поєднувати різнотипові елементи. Поля структури можуть мати довільний тип, крім типу тієї самої структури, але можуть бути вказівником на нього.

Формат синтаксичної конструкції для оголошення структури має вигляд:

```
struct <ім'я_структури>
        {
   <тип1> ім'я поля1;
   <тип2> ім'я поля2;
         . . .
   <типN> ім'я поляN;
        }
    strc1,strc2,    ;
```

`де:`
`struct` – ключове слово, означає початок оголошення структури;
`<ім'я структури>` – ім'я структури, інакше кажучи – ім'я шаблону структури, який створює новий тип даних (ім'я структури ідентифікує конкретну структуру даних і є її специфікатором типу);
`<тип 1>, <тип 2>` … – імена стандартних або визначених типів;
`ім'я поля1, ім'я поля2,…` – імена полів структури;

strc1, strc2 . . .; – імена змінних типу структура (структурні змінні).

Структурні змінні strc1 і strc2 можна оголосити окремо, наприклад:

```
Stud strc1 strc2;
```

Структури, як і інші визначені користувачем типи даних, зазвичай визначаються у глобальній області видимості. Оголошуючи структуру, оголошуємо новий тип даних, але він реалізовується тільки по оголошенні структурної змінної того типу, який уже реально оголошений.

Щоб оголосити структурну змінну, потрібно спочатку задати шаблон структури. Шаблон структури ще називається форматом структури. При задаванні шаблону структури **пам'ять не відводиться.** Шаблон – це тільки оголошення (інформація), змінні яких типів мають входити до структури.

Визначення структурної змінної дає команду компілятору С++ відвести необхідний обсяг пам'яті для зберігання усіх полів структури.

Наприклад, для обчислення середнього балу, отриманого студентами в період сесії з дисциплін ″Математика″, ″Фізика″ та ″Програмування″, визначимо таку структуру:

```
struct Stud
{
char prizv [50];            // прізвище та ініціали
int math, phys, prog;       // предмети
float sb;                   // середний бал
};
```

8.2.1. Ініціалізування елементів стуктури

Наприклад, уже оголошена структура Stud має п'ять полів: прізвище та ініціали студента (поле prizv), предмети: математика (поле math), фізика (поле phys), програмування (поле prog), середній бал (поле sb).

Таким оголошенням не відводиться пам'ять, а лише створюється новий тип даних Stud, ім'я якого може використовуватись при оголошуванні змінних.

Наприклад, оголошення структурних змінних strc1, strc2, масиву структур Mas та вказівника на структуру *ptrs:

```
Stud strc1, strc2, Mas[3], *ptrs;
```

При оголошуванні структурної змінної відводиться пам'ять під усі поля структури послідовно для кожного поля. У поданому прикладі

структури `Stud` під змінну `strc1` послідовно буде виділено 50, 4, 4, 4, 4 байтів. Однак, розмір структури не завжди дорівнює сумі розмірів її полів, оскільки внаслідок вирівнювання об'єктів різної довжини у структурі можуть з'являтися безіменні ″дірки″.

Сумарну пам'ять, яка відводиться під структуру, може бути збільшено компілятором на вимогу процесора для так званого вирівнювання адрес. Реальний обсяг пам'яті, яку займає структура, можна визначити за допомогою оператора `sizeof()`.

Ініціалізування полів структури потрібно здійснювати або при її оголошенні, або у тілі функції `main()`. Під час оголошення структури ініціалізування полів виглядає, наприклад, так:

```
struct Stud
{
char prizv[50];
int math, phys, prog;
float sb;
}
strc1 = {″Тарас Петрів″, 4, 5, 5};
strc2 = {″Андрій Заблоцький″, 3, 4, 5};
```

Якщо ініціалізування виконується в тілі головної функції, то для звернення до імені поля треба спочатку записати ім'я структурної змінної, а потім ім'я поля. Ці обидва записи відокремлюються крапкою і є складеним ім'ям.

Отже, у разі оголошення структурної змінної `strc2` у програмному коді для її ініціалізування можна записати

```
Stud strc2 = {″Андрій Заблоцький″, 3, 4, 5};
```

або ініціалізування виконується за допомогою складених полів

```
strc1.math = 4;
strc1.phys = 5;
strc1.prog = 5;
```

8.2.2. Доступ до полів структури

Доступ до полів структури здійснюється за допомогою операторів вибору: оператора ″ **.** ″ (крапка) при звертанні до полів через ім'я структури та оператора ″ -> ″ (стрілка, послідовно введені з клавіатури символи ″ - ″ мінус та ″ > ″ більше) при звертанні за вказівником, наприклад, для вище оголошених змінних:

```
strc1.prizv = {″Тарас Петрів″, 3, 4, 5};
```

```
strc1.math = 4;
strc1.phys = 5;
strc1.prog = 5;
ptrs -> math = 3;
gets_s(strc2.prizv, 50);
```

До речі, для вказівника попередньо потрібно задати об'єкт посилання настановою

```
ptrs = &strc1;
```

тобто записати адресу конкретної структурної змінної.

Щоб вивести значення поля `prog` структурної змінної `strc1` у консольний додаток, необхідно записати таку настанову:

```
cout << strc1.prog;
```

Якщо виникає потреба отримати доступ до окремих елементів поля структури під назвою, наприклад, `strc2.prizv`, необхідно використати індексування масиву. Наприклад, за допомогою цього фрагмента програмного коду можна вивести у консольному додатку вміст поля структури `strc2.prizv` окремими символами:

```
for(int i = 0; i < strlen(strc2.prizv); i++)
cout << strc2.prizv[i];
cout << endl;
```

Як приклад розглянемо таку задачу. Маємо інформацію про двох студентів (прізвища відомі) та результати складання іспитів з трьох предметів (математика, фізика, програмування). Обчислити середній бал за результатами екзаменаційної сесії.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
struct Stud     // оголошення структури
{
char prizv[50];
int math, phys, prog;
float sb;
}
st1, st2;
float s = 0;
cout << "Enter name and last name: " << endl;
gets_s(st1.prizv, 50);
st1.math = 4;
st1.phys = 5;
```

```
st1.prog = 5;
s += st1.phys + st1.math + st1.prog;
st1.sb = s / 3.;
cout << "Name and last name: " << st1.prizv <<
endl;
cout << "Math = " << st1.math << "  Physics = " <<
st1.phys << "  Progr = " << st1.prog <<
"  Serednyj bal = " << st1.sb << endl;
st2 = st1;
cout << "Enter name and last name:" << endl;
gets_s(st2.prizv, 50);
cout << "Name and last name: " << st2.prizv <<
endl;
cout << "Math = " << st2.math << "  Physics = " <<
st2.phys << "  Progr = " << st2.prog <<
"  Serednyj bal = " << st2.sb << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Enter name and last name:
Sofia Fedyniak
Name and last name: Sofia Fedyniak
Math = 4  Physics = 5  Progr = 5  Serednyj bal =
4.66667
Enter name and last name:
Jarema Bondarenko
Name and last name: Jarema Bondarenko
Math = 4  Physics = 5  Progr = 5  Serednyj bal =
4.66667
```

У поданому програмному коді організовується присвоювання усім полям структури `Stud` відповідних значень. Зауважимо, що поле `st1.prizv` одержує значення шляхом використання уже відомої функції

```
gets_s(st1.prizv, 50);
```

Структурна змінна `st2` того ж типу, що і `st1`, тому справедливим є запис настанови

```
st2 = st1;.
```

Попередження: Одна з найпоширеніших помилок у С++ – забути записати крапку з комою наприкінці оголошення структури. Це генерує помилки компілювання у наступному рядку коду. Сучасні

компілятори, такі як Visual Studio версії 2017, а також новіші версії, інформують вас, що ви забули крапку з комою в кінці оголошення структури, але старіші версії компіляторів можуть цього і не зробити, через що таку помилку буде важко знайти.

Розглянемо ще один приклад, у якому структуру використано для відстеження книг у бібліотеці за такими атрибутами:

– назва книги;
– автор;
– галузь знань;
– ідентифікатор книги.

Проілюструємо це таким проєктом програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

struct Books
{
	char title[50];
	char author[50];
	char subject[150];
	int book_id;
};
int main()
{
// Declare Book1 of type Book
	Books Book1;
// Declare Book2 of type Book
	Books Book2;
	// book 1 specification
	strcpy(Book1.title, "Algorithmization and 
programming");
	strcpy(Book1.author, "Jurij Hrytsiuk");
	strcpy(Book1.subject, "C++ Programming");
	Book1.book_id = 6495407;
	// book 2 specification
	strcpy(Book2.title, "Algorithms and data 
structures");
	strcpy(Book2.author, "Jarema Kuleshnyk");
	strcpy(Book2.subject, "C++ Programming");
	Book2.book_id = 6495700;
	// Book1 info
```

```
    cout << "Book 1 specification : " << endl;
    cout << "Title : " << Book1.title << endl;
    cout << "Author : " << Book1.author << endl;
    cout << "Subject : " << Book1.subject <<
endl;
    cout << "ID : " << Book1.book_id << endl;
    // Book2 info
    cout << "Book 2 specification : " << endl;
    cout << "Title : " << Book2.title << endl;
    cout << "Author : " << Book2.author << endl;
    cout << "Subject : " << Book2.subject << endl;
    cout << "ID : " << Book2.book_id << endl;
    system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
Book 1 specification :
Title : Algorithmization and programming
Author : Jurij Hrytsiuk
Subject : C++ Programming
ID : 6495407
Book 2 specification :
Title : Algorithms and data structures
Author : Jarema Kuleshnyk
Subject : C++ Programming
ID : 6495700
```

Ініціалізування полів структури здійснено використанням відомої вам з попередніх розділів функції `strcpy()`.

Тепер найвищий час трохи ускладнити уже подані приклади.

8.2.3. Масиви структур

С++ надає нам можливість створювати не тільки масиви довільних типів даних, але і масиви структур.

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

```
Stud strArray[100];
```

Щоб отримати доступ до конкретної структури у масиві структур, необхідно індексувати ім’я структури. Щоб відтворити у консольному додатку вміст поля `phys` третьої структури, достатньо використати таку настанову:

```
cout << strArray[2].phys;
```

Як і у разі впорядкування елементів звичайного фіксованого масиву у масиві структур нумерування індексів починається з нуля.

Приклад 8.1. Створити програмний код для опрацювання даних телефонної книги (не менше п’яти абонентів): прізвище, ім’я, номер телефону, дата народження. Передбачити можливість виведення всіх даних телефонної книги та відбір даних про абонентів, номери яких розпочинаються з 067 або 068 (абоненти Київстар).

У поданому далі проєкті програмного коду дані телефонної книги заповнюються у процесі ініціалізування полів масиву структур.

Приклад проєкту програмного коду:

```
#include <iostream>
using namespace std;
int main()
{
struct Abonent
{
char surname[25], name[15], tel[15], d[11];
}
z[] = { "Stadnycka","Ivanna","066-99-55-
444","11.05.85",
"Lozynskyj","Myron","093-10-12-500","23.12.87",
"Kernytska","Olena","067-55-10-123","30.12.89",
"Petranchuk","Vasyl","067-55-10-123","03.02.56",
"Holodenko","Halyna","068-33-11-850","15.08.79" };
int k = sizeof(z) / sizeof(Abonent);
cout << " Telefonna knyha z " << k
<< " abonentamy: \nPrizvyshche Imja Nomer telefonu
Data narodzhennia" << endl;
for (int i = 0; i < k; i++)
cout << z[i].surname << "\t" << z[i].name << "\t"
<< z[i].tel << "\t" << z[i].d << endl;
cout << endl << " Abonenty, z nomeramy 067 or 068:"
<< endl;
int n = 0;
for (int i = 0; i < k; i++)
```

```
if (!strncmp(z[i].tel, "067", 3) ||
!strncmp(z[i].tel, "068", 3))
{
n++;
cout << z[i].surname << "\t" << z[i].name << "\t"
<< z[i].tel << "\t" << z[i].d << endl;
}
cout << endl << "Kilkist abonentiv Kyjivstar - " <<
n << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Telefonna knyha z 5 abonentamy:
Prizvyshche Imja Nomer telefonu Data narodzhennia
Stadnycka       Ivanna  066-99-55-444   11.05.85
Lozynskyj       Myron   093-10-12-500   23.12.87
Kernytska       Olena   067-55-10-123   30.12.89
Petranchuk      Vasyl   067-55-10-123   03.02.56
Holodenko       Halyna  068-33-11-850  15.08.79

 Abonenty, z nomeramy 067 or 068:
Kernytska       Olena   067-55-10-123   30.12.89
Petranchuk      Vasyl   067-55-10-123   03.02.56
Holodenko       Halyna  068-33-11-850  15.08.79

Kilkist abonentiv Kyjivstar - 3
```

У поданому прикладі розмірність масиву структур не вказано явно, тому розмірність масиву визначається компілятором за кількістю ініціалізованих структур.

Приклад 8.2. Розробити структуру, яка містить дані про студентів за результатами екзаменаційної сесії, визначити середній бал за предметами та вивести перелік відмінників.

Приклад проєкту програмного коду:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

struct Student
{
```

```
    char prizv[30];
    int math, fiz, pro;
};
int main()
{
    const int n = 3;
    Student st[n];
    float s1 = 0, s2 = 0, s3 = 0;
    cout << "Vvesty informaciju pro " << n << "
studentiv (Prizv. math. fiz. progr.):" << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> st[i].priz >> st[i].mat >> st[i].fiz
>> st[i].pro;
        //сумарні бали
        s1 += st[i].math; s2 += st[i].fiz; s3 +=
st[i].pro;
    }
    s1 /= n; s2 /= n; s3 /= n;
    cout << endl;
    cout << "Average grade in math: " << s1 <<
endl;
    cout << "Average grade in fiz: " << s2 << endl;
    cout << "Average grade in programming: " << s3
<< endl;
    cout << endl;
    cout << "Vidminnyky:" << endl;
    for (int i = 0; i < n; i++)
    if (st[i].math == 5 && st[i].fiz == 5 &&
st[i].pro == 5)
    cout << st[i].prizv << endl;
    system("pause");
    return 0;
}
```

Результат виконання програмного коду:

```
Vvesty informaciju pro 3 studentiv (Prizv. math.
fiz. progr.):
Kolodnytskyj 3 4 3
Stefanchuk 4 3 3
Tarnavskyj 5 5 5
```

```
Average grade in math: 4
Average grade in fiz: 4
Average grade in programming: 3.66667

Vidminnyky:
Tarnavskyj
```

8.2.4. Вказівники на структури

C++ дає змогу оголосити не об’єкт структури, а вказівник на об’єкт. Наприклад, уже відома структура `Stud` про наших студентів:

```
Stud *p;
```

Водночас створюється вказівник на об’єкт, але самого об’єкта ще не існує, бо для нього не відведено пам’ять. Відвести пам’ять можна за допомогою оператора `new`:

```
p = new Stud;
```

Доступ до полів об’єкта в цьому випадку здійснюється з допомогою оператора -> :

```
strcpy(p->prizv, "Тарас Петрів");
p->math = 5;
p->phys = 4;
p->prog = 5;
```

Вказівнику можна присвоїти адресу вже наявного об’єкта структури, наприклад:

```
Stud *p;
Stud st1;
p = &st1;
```

8.2.5. Структури як параметри функцій

Як і довільний інший об’єкт C++ структури можуть використовуватися як параметри функцій, а також можуть бути об’єктом, який повертає функція. Передавати цілу структуру у функцію, яка працює з її полями, є великою перевагою над передавання окремих змінних.

Існує два способи передавання структури як параметрів функції:

– передавання структури за значенням (за такого передавання робиться копія структурної змінної у пам’яті та її адреса через стек передається у функцію. Якщо структура має великий розмір, тоді цей спосіб є неефективним. Перевагою такого способу є те, що усі перет-

ворення з копією цієї структури у тілі функції не впливають на початкові значення полів структурної змінної);

– передавання вказівника на структуру (у такому разі через стек у функцію передається тільки вказівник на структуру, тобто її адреса у пам'яті. Якщо під структуру відведено великий обсяг пам'яті, то такий спосіб забезпечує швидке передавання значень полів структурної змінної у функцію. Це пояснюється тим, що не створюється у пам'яті копії структурної змінної).

Проілюструємо використання структур як параметрів функцій такими прикладами. Наші студенти знову не будуть мати спокою.

Приклад 8.3. Створити проєкт програмного коду для опрацювання інформації про результати сесії студентів: прізвище студента, курс, група і результати трьох екзаменів. Передбачити можливість введення і виведення даних та переведення студентів, які успішно склали сесію, на наступний курс, змінивши відповідно з цим номер групи.

Приклад проєкту програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstring>
using namespace std;
struct Student
      // Оголошення типу структури з полями:
{
      char prizv[30];    // прізвище,
      int kurs;          // курс,
      char gr[10];           // група,
      int ekz[3];      /* масив екзаменаційних оці-
нок від 0 до 100. */
};
void Transf_course(Student &S)
// Функція переведення на наступний курс
{
      int k = 0;
      for (int j = 0; j < 3; j++)
            if (S.ekz[j] > 50 && k == 0) k = 0;
            else k = 1;
      if (k == 1) cout << "\nRepeat course  ";
      else
      {
            cout << "\nTransferred to the next
course  ";
            S.kurs++;
```

```
	}
}
void View_stud(Student S)
// Функція виведення на екран даних структури
{
	cout << "Last name: " << S.prizv << endl;
	cout << "Kurs: " << S.kurs << endl;
	cout << "Grupa: " << S.gr << endl;
	cout << "Exameny: ";
	for (int j = 0; j < 3; j++)
		cout << S.ekz[j] << " ";
	cout << endl;
}
int main()
{
	const int N = 3;
	// Оголошення масиву студентів
	Student s[N];
	cout << "Vvesty informaciju pro studentiv: "
<< endl;
	for (int i = 0; i < N; i++)
	// Введення даних до масиву студентів
	{
		cout << "Last name: ";
		cin >> s[i].prizv;
		cout << "Kurs: ";
		cin >> s[i].kurs;
		cout << "Grupa: ";
		cin >> s[i].gr;
		cout << "Exameny: " << endl;
		for (int j = 0; j < 3; j++)
		cin >> s[i].ekz[j];
	}
	for (int i = 0; i < N; i++)
	/* Виклик функції Transf_course() для кожного
студента*/
	{
		Transf_course(s[i]);
		cout << s[i].prizv;
	}
	cout << "\nMasyv pislia analizu: " << endl;
	for (int i = 0; i < N; i++)
```

```
      // Виклик функції View_stud() для кожного 
студента
            View_stud(s[i]);
      system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
Vvesty informaciju pro studentiv:
Last name: Kolodnytskyj
Kurs: 1
Grupa: IT-
Exameny:
67 73 81
Last name: Stefanchuk
Kurs: 1
Grupa: IT-
Exameny:
49 51 65
Last name: Tarnavskyj
Kurs: 1
Grupa: IT-
Exameny:
91 90 82

Transferred to the next course  Kolodnytskyj
Repeat course Stefanchuk
Transferred to the next course  Tarnavskyj
Masyv pislia analizu:
Last name: Kolodnytskyj
Kurs: 2
Grupa: IT-2
Exameny: 67 73 81
Last name: Stefanchuk
Kurs: 1
Grupa: IT-1
Exameny: 49 51 65
Last name: Tarnavskyj
Kurs: 2
Grupa: IT-2
Exameny: 91 90 82
```

Як і у попередньому прикладі, початкові дані про студентів та результати складених іспитів зберігаються у масиві структур `s` розмірністю `N`.

У запропонованому прикладі програмного коду введення початкових даних типу `char` здійснюється з використанням настанов

```
cin >> s[i].prizv;
cin >> s[i].gr;
```

Це має свою специфіку (введення символів здійснюється до першого пропуску), тому вводяться тільки прізвища студентів без їхнього імені.

Для виведення у консольний додаток значень полів структури розроблено функцію `void View_stud(Student S)`. Використано передавання структури у функцію за значенням, оскільки завдання цієї функції не полягає у перетворенні значень полів структури, а тільки виведення цих значень у консольний додаток.

Інша справа з функцією `void Transf_course(Student &S)`, яка призначена для аналізу результаів екзаменаційної сесії і переведення студентів на наступний курс. Структурна змінна передається у функцію за посиланням, тобто передається адреса першого байта ділянки, яку займає в оперативній пам'яті структура. Звертання до розроблених функцій виконується у циклі, тобто для кожного студента.

Приклад 8.4. Повернемося до уже відомої нам студентської бібліотеки. Для зберігання інформації про кожну окрему книгу використаємо динамічний масив структур, оскільки кількість книг у бібліотечному фонді може змінюватися. Тепер у нас буде `Nk` кількість книг, кожна з яких ідентифікується за відомими нам критеріями. З метою урахування поповнення бібліотечного фонду використаємо динамічний масив структур. Введення початкових даних та їх виведення у консольний додаток організуємо відповідними функціями `Get_Data()` та `Show_Data()`. Передавання масиву структур у функції виконаємо вказівником.

Проєкт програмного коду розв'язання задачі:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
struct Library
{
	char title[50];
	char author[50];
```

```
	char subject[150];
	int publication_year;
	char book_id[20];
};
void Show_Data(Library *Book, int Nk)
{
	for (int i = 0; i < Nk; i++)
	{
		cout << Book[i].title << endl;
		cout << Book[i].author << endl;
		cout << Book[i].subject << endl;
		cout << Book[i].publication_year << 
endl;
		cout << Book[i].book_id << endl;
	}
}
void Get_Data(Library *Book, int Nk)
{
	cin.ignore();
	for (int i = 0; i < Nk; i++)
	{
		cout << "\nBook # " << i+1 << endl;
		cout << "Title book: ";
		cin.getline(Book[i].title, 50);
		cout << "Author book: ";
		cin.getline(Book[i].author, 50);
		cout << "Subject book: ";
		cin.getline(Book[i].subject, 150);
		cout << "Publication_year book: ";
		cin >> Book[i].publication_year;
		cin.ignore();
		cout << "Book ID: ";
		cin.getline(Book[i].book_id, 20);
	}
}
int main()
{
	int Nk;
	cout << "Number of books  Nk: ";
	cin >> Nk;
	Library *Book = new Library[Nk];
	// Введення даних у масив структур
```

```
	cout << "\nCreate Library ";
	Get_Data(Book, Nk);
	/* Виведення масиву структур у консольний
додаток*/
	cout << "\nShow Library " << endl;
	Show_Data(Book, Nk);
	delete []Book;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Number of books  Nk: 3

Create Library
Book # 1
Title book: Algorytmozation and programming
Author book: Jurij Hrytsiuk
Subject book: Programming
Publication_year book: 2013
Book ID: 47859655

Book # 2
Title book: Algorithms and data structures
Author book: Ihor Kroshnyj
Subject book: Programming
Publication_year book: 2021
Book ID: 47882197

Book # 3
Title book: Databases
Author book: Myhajlo Dendiuk
Subject book: Programming
Publication_year book: 2018
Book ID: 47546123

Show Library
Algorytmozation and programming
Jurij Hrytsiuk
Programming
2013
47859655
Algorithms and data structures
```

```
Ihor Kroshnyj
Programming
2021
47882197
Databases
Myhajlo Dendiuk
Programming
2018
47546123
```

Усе чудово. Програмний код виконується, результат достовірний, не забули вилучити динамічний масив структур. Проте, ми побачили нову для нас функцію `cin.ignore()`. Вона використана у функції формування вмісту масиву структур `Library Book` і працює з буферами введення/виведення стандартних потоків.

Отже, у головній функції до стандартного потоку введення долучається кількість книг у бібліотеці настановою

```
cin >> Nk;
```

Після введення з клавіатури числового значення змінної `Nk` натискаємо клавішу Enter. Потік введення **`cin`** ″захоплює″ разом з числовим значенням `Nk` і символ нового рядка ″`\n`″. Пізніше числове значення присвоюється змінній `Nk`, а символ ″`\n`″ (символ нового рядка) залишився у стандартному потоці введення. Потім, коли настановою

```
cin.getline(Book[i].title, 50);
```

хочемо отримати значення поля `Book[i].title` функція `cin.getline` ″бачить″ у потоці ″`\n`″ і сприймає це за *введення порожнього рядка*! Тобто, замість введення назви книги вводиться порожній рядок, а потім успішно виконується настанова

```
cin.getline(Book[i].author, 50);
```

а це, безумовно, не те, що ми хотіли зробити.

Добрим тоном у практиці програмування вважається вилучати з стандартного потоку введення даних символ нового рядка. Отже, з урахуванням викладеного, візьмемо за *правило,* при введенні числових значень не забувайте вилучати символ нового рядка з потоку введення за допомогою функції **`cin.ignore()`**.

Структури дуже важливі у мові C++, оскільки їх розуміння – це перший великий крок до об’єктно-орієнтованого програмування! Трішки пізніше ви ознайомитеся з іншим користувацьким типом даних – клас (який є продовженням теми структур). Структури, як правило, використовуються для групування даних, а не замість класів.

Контрольні запитання

1. Як здійснюється доступ до елементів структури?
2. Що таке структури даних у програмуванні?
3. Запишіть формат оголошення структури.
4. Що таке шаблон структури?
5. Як здійснюється ініціалізування елементів структури?
6. Як визначити масив структур?
7. Як отримати доступ до конкретної структури у масиві структур?
8. Які способи передавання структури у якості параметрів функції?

Розділ 9

ОБ’ЄДНАННЯ, ПЕРЕЛІЧУВАННЯ

9.1. Поняття об'єднання

Об’єднання – це такий формат даних, який є схожим до структури і призначений зберігати у межах однієї зарезервованої області пам’яті різні типи даних. Проте, у кожний певний момент часу в об’єднанні зберігається тільки значення одного з цих типів даних і допустимо використовувати тільки значення цього елементу.

Властиво сенс **union** полягає у тому, що це є вказівкою компілятору зарезервувати місце для типу даних з максимальним обсягом пам’яті, а користувач уже самостійно вирішить, які значення туди ввести.

Отже, об’єднання слугує для розміщення в одній і тій же області пам’яті за тією ж адресою даних різних типів.

Формат синтаксичної конструкції для оголошення об’єднання має вигляд:

```
union <ім’я>
      {
<тип> <змінна_1>;
<тип> <змінна_2>;
      . . .
<тип> <змінна_n>;
      } <змінна>;
```

Об’єднання можна розглядати як структуру, елементи якої не мають зміщення (0-зміщення) у пам’яті. Звертання до елементів об’єднання таке ж, як і до полів структур. Поточне значення елемента об’єднання втрачається після того, як іншому елементові буде присвоєно значення.

Наприклад:

```
union Digit
{
int x;
```

```
double y;
char letter;
} Sun;
Sun.x = 12;
Sun.y = 2.56;
Sun.letter = 't';
```

Відведення пам'яті для значень елементів об'єднання відповіднодо їх типів подано у таблиці 9.1.

У таблиці 9.1 сірим кольором позначено області спільної пам'яті для елементів об'єднання, які залишаються вільними при зверненні.

Таблиця 9.1.

Відведення пам'яті для значень елементів об'єднання відповідно до їх типів

БАЙТИ							
1	2	3	4	5	6	7	8
x							
y							
letter							

Зрозуміло, що всі ці значення записуються з одного і того ж місця в пам'яті (з початку відведеної області пам'яті), тому об'єднання має своїм значенням останнє звернення елементу об'єднання, до якого здійснювалося звернення.

Компілятор ***НЕ*** *контролює, якого типу було останнє значення*. Це повинен контролювати програміст.

Об'єднання можуть входити у масиви структур і навпаки. Наприклад:

```
struct St
{
char name;
int var;
union Ugc
{
int x;
double y;
char letter;
};
} Str[N];
```

`Str[N]` – це масив структур, кожна з яких складається із трьох елементів. Третім елементом є об'єднання. Тому можна записати:

```
Str[i].Ugc.y = 3.141526;
```

Ініціалізувати об'єднання можна лише значенням першого елементу, тобто у попередньому випадку – цілим значенням. Тому, крім того, що значення елементів об'єднання записуються на те саме місце в пам'яті, усі інші їхні властивості такі ж, як і у структур.

Приклад проєкту програмного коду ініціалізування елементів об'єднання:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

union Data
{
	int x;
	float y;
	char s[10];
};
int main()
{
// Оголошення об'єднання
	Data Mydata;
	Mydata.x = 10;
cout << "\nx= " << Mydata.x;
mydata.x++;
cout << "\nx++= " << Mydata.x;
	Mydata.y = 25.793;
	cout << "\ny= " << Mydata.y;
	strcpy(Mydata.s, "student");
	cout << "\ns= " << Mydata.s << endl;
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
x= 10
x++= 11
y= 25.793
s= student
```

Застосування об'єднань зумовлено необхідністю економії пам'яті, коли потрібно зберігати і використовувати дані різних типів, але звертатися до них можна не одночасно.

9.2. Перелічуваний тип даних

Перелічуваний тип даних або перерахування (перелічування) – це тип даних, де довільне значення (енумератор) визначається як символьна константа.

Розв'язуючи певні задачі, виникає потреба у визначенні наперед відомої кількості іменованих констант, яким присвоюються різні значення (водночас конкретні значення можуть бути неважливими). Для цього зручно користуватися типом перерахування **enum**, всі можливі значення якого задаються переліком цілочисельних констант і оголошується такою синтаксичною конструкцією:

enum [<ім'я_типу>] {<перелік_констант>};

Ім'я типу задається тоді, коли у програмному коді потрібно визначати змінні цього типу.

Оголошення перелічування не супроводжується відведенням області пам'яті. У разі, коли змінна перелічуваного типу визначена, тоді відводиться область пам'яті для цієї змінної.

Імена констант мають бути унікальними і, як правило, починаються великою літерою.

Змінним перелічуваного типу можна присвоювати кожне значення з переліку констант, визначених при оголошенні типу. Енумератори подаються і опрацьовуються як цілі числа та ініціалізуються у звичайний спосіб.

Кожному енумератору автоматично присвоюється цілочисельне значення залежно від його позиції у переліку перелічування. За замовчуванням або за відсутності явного ініціалізування, першому енумератору присвоюється ціле число ″0″, а кожному наступному – на одиницю більше (застосовується оператор інкременту), ніж попередньому. Крім того, енумераторам можна присвоїти однакові значення, а також від'ємні.

Наприклад, розглянемо такий фрагмент програмного коду:

```
enum Week
{
sat = 0,
sun = 0,
mon,
tue,
wed,
thu,
fri
}
budnyj_den;
```

Тут описано перелічування Week з відповідною множиною значень і оголошено змінну budnyj_den типу Week . Енумераторам sat та sun присвоєно значення 0, mon – значення 1, tue – 2, wed – 3, thu – 4, fri – 5.

Оскільки значеннями енумераторів є цілі числа, то їх можна присвоювати цілочисельним змінним, а також виводити у консольний додаток (як змінні типу int), наприклад, подамо для аналізу такі проєкти програмних кодів:

```
//example 1
#include <iostream>
using namespace std;
// Визначаємо шаблон перелічуваного типу Animals
enum Animals
{
      animal_pig = -4,
      animal_lion,
      animal_cat,
      animal_zebra = 5,
      animal_horse = 5
};
int main()
{
cout << "\nRezult:" << animal_horse << endl;
system ("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Rezult: 5
```

Хочемо зауважити, що у поданому прикладі animal_zebra та animal_horse отримали однакові значення. У цьому випадку енумератори стають нерозрізнюваними – по суті animal_zebra і animal_horse є взаємозамінними. Хоча C++ це дозволяє, загалом потрібно уникати просвоювання однакових значень кільком енумераторам у одному перелічуванні.

Наша порада – *уникайте* просвоювання однакових значень кільком енумераторам у одному перелічуванні, якщо ви не маєте для цього вагомих підстав.

```
//example 2
#include <iostream>
using namespace std;
// Визначаємо шаблон перелічуваного типу Year
enum Year {
```

```
    Jan,
    Feb,
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec
};
int main()
{
    int i;
    // Виводимо значення енумераторів
    for (i = Jan; i <= Dec; i++)
        cout << i << " ";
    cout << endl;
    system("pause");
      return 0;
}
```

Результат виконання програмного коду:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Компілятор не буде неявно конвертувати цілочисельне значення у значення енумератора. Наступна настанова з першого прикладу викличе помилку компілювання:

```
Animals tvaryna = -3;
```

Кожен перелічуваний тип вважається окремим типом, тому спроба присвоїти енумератор з одного перелічування енумератору з іншого – викличе помилку компілювання.

До перелічуваних змінних можна застосовувати арифметичні операції і операції відношення, наприклад:

```
#include <iostream>
using namespace std;
// Визначаємо шаблон перелічуваного типу Days
enum Days
{
      sun, mon, tue, wed, thu, fri, sat
};
```

```
int main()
{
      Days day1 = mon, day2 = thu;
      int dif_day = day2 - day1;
      cout << "Riznytsia u dniah: " << dif_day <<
endl;
      if (day1 < day2)
            cout << "day1 nastane skorshe za day2
\n";
system ("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Riznytsia u dniah: 3
day1 nastane skorshe za day2
```

Істотним недоліком перелічуваного типу є те, що змінні цього типу не розпізнаються засобами введення/виведення C++. При виведенні такої змінної виводиться не її формальне значення, а її внутрішнє подання, тобто ціле число.

Подамо приклад програмного коду, яким можна скористатись за потреби виведення значень таких змінних у звичому для сприйняття вигляді.

Ввести порядковий номер дня тижня і вивести його назву. Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstring>
using namespace std;
enum Day
{
      mon = 1, tue, wed, thu, fri, sat, sun
};
string dayToString(const Day &one)
{
      string arr[] = { "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday",
"Sunday" };
      return arr[one - 1];
}
Day Int_to_day(int one)
{
      return static_cast<Day>(one);
}
```

```
int main()
{
	int numb;
	cout << "Enter number of day (1,2,3,4,5,6 or 
7): ";
	cin >> numb;
	Day one = Int_to_day(numb);
	cout << dayToString(one) << '\n';
	system("pause");
	return 0;
}
```

Результат виконання програмного коду:

```
Enter number of day (1,2,3,4,5,6 or 7): 5
Friday
```

Використаний у функції `Int_to_day()` оператор `static_cast` має такий формат:

`static_cast<тип>(вираз)`

і перетворює без жодних перевірок відповідності під час виконання програмного коду вираз до вказаного типу.

У нашому випадку відбуватиметься перетворення введеної цілої змінної `one` до перелічуваного типу `Day`.

Перелічування, яке формується з фіксованого набору констант або, можна сказати, набору інтегральних констант рекомендується використовувати у практиці програмування, це заощаджує багато часу та пам'яті, що опосередковано робить код ефективнішим і швидшим з точки зору продуктивності. Тому, коли ви визначаєте шаблон **`enum`**, буде створено лише проєкт для змінної перелічуваного типу і для цього не відводиться пам'ять.

Контрольні запитання

1. У чому полягає зміст поняття об'єднання?
2. Яке призначення об'єднання?
3. Запишіть формат синтаксичної конструкції для оголошення об'єднання.
4. Як здійснюється звертання до елементів об'єднання?
5. Як здійснюється ініціалізування об'єднання?
6. Як подаються і опрацьовуються енумератори?
7. Які арифметичні операції можна застосовувати до перелічуваних змінних?
8. Який найважливіший недолік перелічуваного типу?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Алгоритмізація та програмування: спеціальність 122 "Комп'ютерні науки" / Ю. С. Процик, Т. С. Самотій, М. В. Левкович. Львів: НЛТУ України, 2017. URL: http://vee.nltu.edu.ua/course/ view.php?id=3.

2. Бандоріна Л. М., Климкович Т. О., Удачина К. О. Основи алгоритмізації та програмування : навч. посібник. УДУНТ, 2022. 158 с.

3. Бєлов Ю. А. Вступ до програмування мовою С++. Організація обчислень: навчальний посібник / Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. К. : Видавничо-поліграфічний центр "Київський університет", 2012. 175 с.

4. Бублик В. В. Об'єктно-орієнтоване програмування : підручник. К.: ІТкнига, 2015. 624 с.

5. Грицюк Ю. І., Рак Т. Є. Програмування мовою С++: навчальний посібник. Львів : Вид-во Львівського ДУ БЖД, 2011. 292 с.

6. Ковалюк Т. В. К 56 Алгоритмізація та програмування: Підручник. – Львів: «Магнолія 2006», 2013. 400 с.

7. Ментинський С. М., Пелех Я. М. Основи програмування на С++: навчальний посібник. Львів : Галицька Видавнича Спілка, 2021. 256 с.

8. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 1: навчальний посібник. Львів: Видавництво "Новий Світ-200", 2021. 337 с.

9. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 2: навчальний посібник. Львів: Видавництво "Новий Світ-200", 2021. 315 с.

10. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування. Частина 1. Структурне програмування: навчальний посібник. Львів: Львівський державний університет внутрішніх справ, 2023. 240 с.

11. Трофименко О. Г., Прокоп Ю. В., Задерейко О. В. Алгоритмізація та програмування : навчально-методичний посібник. Одеса: Фенікс, 2020. 310 с. URL: http://dspace.onua.edu.ua/handle/11300/12345.

12. Фратавчан В. Г., Фратавчан Т. М., Лазорик В. В. Алгоритмізація та програмування, навчальний посібник для закладів вищої освіти. ЧНУ, 2022, 286 с.

13. Яворський Н. Б., Марікуца У. Б., Андрійчук М. І., Фармага І. В. Лабораторний практикум з дисципліни "Алгоритмізація та програмування": навчальний посібник. Львів: Видавництво Львівської політехніки, 2018. 191 с.

14. Ярошко С. А. Методи розробки алгоритмів. Програмування мовою С++: навчальний посібник. Львів : ЛНУ імені Івана Франка, 2022. 248 с.

Інформаційні ресурси

1. http://cpp.dp.ua/lecture/
2. https://www.enseignement.polytechnique.fr/informatique/INF478/docs/Cpp/en/cpp/language.1.html
3. https://www.bestprog.net/uk/sitemap_ua/c/
4. https://ela.kpi.ua/bitstream/123456789/28216/1/Alhorytmizatsiya-ta-prohramuvannia-Praktykum_2019Kublii.pdf
5. https://nmetau.edu.ua/file/099.pdf
6. http://om.univ.kiev.ua/users_upload/15/upload/file/prog_lecture_06.pdf
7. http://cpp.dp.ua/vykorystannya-fajliv/
8. https://www.geeksforgeeks.org/enumeration-in-cpp/
9. https://unetway.com/tutorial/c-struktury-dannyh
10. https://www.bestprog.net/uk/2019/09/11/examples-of-using-c-tools-for-working-with-files-ua/
11. https://www.learncpp.com/cpp-tutorial/stdarray-and-enumerations/

НАВЧАЛЬНЕ ВИДАННЯ

Тарас РУДИЙ
Ярослав ПАРАНЧУК
Володимир СЕНИК

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Частина 2

Модульне програмування

Навчальний посібник

Редагування *Ірина Краївська*

Макетування *Андріана Кузьмич-Походенко*

Друк *Андрій Радченко*

Підписано до друку 29.02.2024.
Формат 60×84/16.
Папір офсетний. Умовн.-друк. арк. 10,56
Тираж 80 прим. Зам. № 08-24.

Львівський державний університет внутрішніх справ
вул. Городоцька, 26, Львів, 79007, Україна

Свідоцтво про внесення суб’єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції
ДК № 2541 від 26 червня 2006 р.