ТЕХНОЛОГІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

ЧАСТИНА II

САМОСТІЙНА РОБОТА ТА ВИКОНАННЯ СЕМЕСТРОВИХ ЗАВДАНЬ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ТЕХНОЛОГІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

ЧАСТИНА II
САМОСТІЙНА РОБОТА ТА ВИКОНАННЯ СЕМЕСТРОВИХ ЗАВДАНЬ

*Рекомендовано Методичною радою НТУУ «КПІ ім. Ігоря Сікорського»
як навчальний посібник для студентів,
які навчаються за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології»*

Київ
КПІ ім. Ігоря Сікорського
2018

ТЕХНОЛОГІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ: ЧАСТИНА ІІ. САМОСТІЙНА РОБОТА ТА ВИКОНАННЯ СЕМЕСТРОВИХ ЗАВДАНЬ [Електронний ресурс]: навч. посіб. для студ. спеціальності 151 – «Автоматизація та комп'ютерно-інтегровані технології» / КПІ ім. Ігоря Сікорського; уклад.: В. І. Бендюг, Б. М. Комариста. – Електронні текстові данні (1 файл: 2,14 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2018. – 131 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 10 від 21.06.2018 р.)
за поданням Вченої ради інституту/факультету (протокол № 6 від 30.05.2018 р.)

Електронне мережне навчальне видання

ТЕХНОЛОГІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ
ЧАСТИНА ІІ
САМОСТІЙНА РОБОТА ТА ВИКОНАННЯ СЕМЕСТРОВИХ ЗАВДАНЬ

Укладачі: Бендюг Владислав Іванович, канд. техн. наук, доцент
Комариста Богдана Миколаївна, канд. техн. наук

Відповідальний редактор: А. М. Шахновський, канд. техн. наук, доцент

Рецензенти: О.І. Букет, канд. техн. наук, доцент

Зміст

Вступ

Згідно робочого навчального плану кредитний модуль «*Прикладне програмне забезпечення - 2. Технології об'єктно-орієнтованого програмування*» дисципліни «*Прикладне програмне забезпечення*» викладається студентам четвертого року підготовки ОКР «бакалавр» спеціальності *151 - Автоматизація та комп'ютерно-інтегровані технології* у сьомому навчальному семестрі. Матеріал кредитного модуля базується на дисциплінах «Комп'ютерні технології та програмування - 1. Основи алгоритмізації», «Комп'ютерні технології та програмування - 2. Програмування типових задач», «Комп'ютерна техніка та організація обчислювальних робіт», «Комп'ютерні технології та програмування - 3. Розробка інтерфейсу користувача», «Інформаційні системи та комплекси». Знання уміння та навички, одержані під час вивчення даного модуля, у подальшому використовуються в усіх дисциплінах, які потребують програмної реалізації розрахунків на комп'ютері, і в першу чергу – в курсах «Ідентифікація та моделювання об'єктів автоматизації», «Основи проектування систем автоматизації і систем керування експериментом», «Методи штучного інтелекту та їх застосування в хімічній технології», «Прикладне програмне забезпечення - 3. Проектування програмних додатків», в курсових і дипломних роботах і проектах.

Метою навчальної дисципліни є формування у студентів здібностей:

КСП-24 – вміння застосовувати комп'ютерну техніку для вирішення технічних задач;

КСП-25 – вміння використовувати комп'ютерно-інтегровані технологічні та інформаційні системи;

КСП-26 – вміння розробляти, тестувати та застосовувати програмне забезпечення для вирішення прикладних задач.

Згідно з вимогами освітньо-професійної програми студенти після засвоєння навчальної дисципліни мають продемонструвати такі результати навчання:

знання:

- стандартів сучасних об'єктно-орієнтованих мов програмування;
- інтерфейсу середовища програмування Microsoft Visual Studio;
- сучасних методів конструювання програм у тому числі засобами об'єктно-орієнтованої мови програмування C++;

- ефективних методів збереження і обробки інформації;

уміння:

- самостійно розробляти ефективні алгоритми і програми з використанням сучасної алгоритмічної мови для вирішення поставленої задачі;
- налагоджувати програми на ПК до надійної працездатності в ОС Windows;

досвід:

- виконувати налаштування середовища програмування для ефективної роботи;
- оформляти розроблені програми згідно вимогам ЄСПД.

1. Структура кредитного модуля

Таблиця 1.1 – Структура кредитного модуля дисципліни

Галузь знань, напрям підготовки, освітньо-кваліфікаційний рівень	**Загальні показники**	**Характеристика кредитного модуля**
Галузь знань **0502 – Автоматика і управління** **(шифр і назва)**	Назва дисципліни, до якої належить кредитний модуль Прикладне програмне забезпечення	Форма навчання денна (денна / заочна)
Спеціальність **151 Автоматизація та комп'ютерно-інтегровані технології** **(шифр і назва)**	Кількість кредитів ECTS 4	Статус кредитного модуля за вибором ВНЗ (нормативний або за вибором ВНЗ/студентів)
Спеціалізація - **(назва)**	Кількість розділів 2	Цикл до якого належить кредитний модуль II.2. Дисципліни вільного вибору студентів
	Індивідуальне завдання: ДКР (вид)	Рік підготовки 4
		Семестр *сьомий*
Освітньо-кваліфікаційний рівень **бакалавр**	Загальна кількість годин 120	Лекції 18 год.
		Практичні (семінарські) -
		Комп'ютерний практикум 54 год.
	Тижневих годин: аудиторних – 3 СРС – 2,7	Самостійна робота 48 год. в тому числі на виконання індивідуального завдання 7 год.
		Вид та форма семестрового контролю залік (екзамен / залік / диф. залік; усний / письмовий / тестування тощо)

2. Рейтингова система оцінювання результатів навчання

Рейтинг студента з дисципліни складається з балів, що він отримує за:

1) Виконання та захист комп'ютерних практикумів;
2) Написання модульної контрольної роботи;
3) Виконання домашньої контрольної роботи;

Шкала балів за відповідні рівні оцінювання з кожного виду контролю.

а) Комп'ютерні практикуми:
«відмінно» –7 балів;
«добре» – 5-6 балів;
«задовільно» – 4 бали;
«незадовільно» – 0 балів.

б) ДКР:
«відмінно» – 18-20 балів;
«добре» – 15-17 балів;
«задовільно» – 13-14 балів;
«незадовільно» – 0 балів.

в) МКР:
«відмінно» – 15 – 17 балів;
«добре» – 13 – 14 балів;
«задовільно» – 11 - 12 балів;
«незадовільно» – 0 балів.

Контрольна перевірка: студент, який отримав мінімальні позитивні бали за всіма контролями, матиме у підсумку не менше 60 балів.

$$4_{КП} \times 9 + 13_{ДКР} + 11_{МКР} = 60 \text{ балів.}$$

Система рейтингових (вагових) балів та критерії оцінювання

1. Комп'ютерні практикуми.

Ваговий бал – 7.

Рейтингові бали комп'ютерного практикуму складаються з балів за підготовку та виконання практикуму (від 1 до 2), балів за оформлення протоколу практикуму (від 1 до 2 балів) і балів за захист практикуму (від 2 до 3). Таким чином за результатами практикуму студент може отримати від 4 до 7 балів.

За ***виконання роботи*** бали виставляються наступним чином:

- ∑ робота повністю і вірно виконана у відведений час – 2 бали;
- ∑ робота виконана у відведений час, але не повністю висвітлює деякі аспекти завдання – 1 бал;
- ∑ робота виконана невчасно або має суттєві недоліки – 0 балів.

За ***оформлення протоколу практикуму*** бали виставляються наступним чином:

- протокол відповідає вимогам, оформлений охайно, без виправлень і помарок (допускається не більше 1 виправлення на 1 сторінці протоколу) – 2 бали;
- протокол відповідає вимогам, проте є певні недоліки – 1 бал;
- протокол не відповідає основним вимогам до оформлення – 0 балів.

За ***захист практикуму*** бали виставляються наступним чином:

- студент вірно і повністю відповів на всі поставлені йому запитання (виконав надані для захисту роботи завдання) – 3 бали;
- студент відповів на запитання в цілому вірно, проте при відповідях припускався несуттєвих помилок – 2 бали;
- студент припустився помилок при відповідях на більшість питань або взагалі не відповів на більшу частину запитань – (0 балів);
- при отриманні 0 балів за захист, студент має повторно захищати лабораторну роботу на наступному занятті.

2.Модульний контроль

Ваговий бал – 17.

Модульна контрольна робота являє собою практичне завдання – створення консольного додатку для вирішення поставленої задачі згідно отриманого варіанту протягом відведеного часу. Модульна контрольна робота проводиться після вичитування основної частини лекцій курсу. За результатами виконання роботи для позитивної оцінки студент може отримати від 11 до 17 балів. Оцінювання такої роботи проводиться за наступною шкалою:

- студент повністю вірно і у повному обсязі виконав завдання (15 - 17 балів);
- студент вірно і у повному обсязі виконав завдання проте припустився несуттєвих помилок (13 - 14 балів);
- студент виконав основну частину завдання проте припустився помилок та виконав завдання не в повному обсязі (11 - 12 балів);
- студент не виконав основної частини завдання, або припустився суттєвих помилок (0 балів);
- студент не з'явився без поважних причин на модульний контроль (– 2 бали).

У разі, якщо студент не закінчив виконання роботи вчасно, оцінюється та частина, яка фактично виконана.

3.Домашня контрольна робота

Ваговий бал – 20.

Домашня контрольна робота являє собою практичне завдання – створення програмного модулю на мові програмування С++ згідно отриманого варіанту індивідуального завдання. Оцінювання такої роботі проводиться за наступною шкалою:

- студент повністю вірно і у повному обсязі виконав поставлене завдання та відповів на додаткові питання (18-20 балів);
- студент повністю вірно і у повному обсязі виконав поставлене завдання але не відповів на додаткові питання (від 15 до 17 балів);
- студент в цілому вірно та повністю виконав поставлене завдання але припустився несуттєвих помилок (від 13 до 14 балів);
- студент виконав завдання неповністю та (або) припустився при виконанні суттєвих помилок (0 балів);
- студент не здав домашню контрольну роботу у відведений час без поважної причини (– 1 бал за кожен тиждень затримки).

Штрафні та заохочувальні бали за *:

- Вдосконалення дидактичного матеріалу до комп'ютерного практикуму +1 бал;
- Вдосконалення дидактичного матеріалу до домашньої контрольної роботи +5 балів;
- Недопущення до комп'ютерного практикуму у зв'язку з незадовільним вхідним контролем (відсутність протоколу, недостатня підготовка до практикуму) –1 бал;
- Відсутність на комп'ютерному практикуму без поважної причини –1 бал.

Умови допуску до заліку

Умовами допуску до заліку є:

- Захист 9-ти комп'ютерних практикумів на позитивну оцінку (4 і більше балів);
- Написання модульної контрольної роботи на позитивну оцінку (11 або більше балів);
- Виконання домашньої контрольної роботи на позитивну оцінку (13 або більше балів);
- Наявність конспекту лекцій.

Умови отримання семестрової атестації

Календарна атестація студентів (на 8 та 14 тижнях семестрів) з дисциплін проводиться викладачами за значенням поточного рейтингу студента на час атестації. Якщо значення цього рейтингу не менше 50 % від максимально можливого на час атестації, студент вважається задовільно атестованим. В іншому випадку в атестаційній відомості виставляється «незадовільно».

1-ша атестація

$$r_C = (4*7)*0,5=14$$

Якщо $r_C \geq 14$ студент вважається задовільно атестованим

2-га атестація

$$r_C = (7*7)*0,5=25$$

Якщо $r_C \geq 25$ студент вважається задовільно атестованим

Розрахунок шкали рейтингу:

Сума вагових балів контрольних заходів протягом семестру складає:

$$R_C = 7*9 + 17 + 20 = 100 \text{ балів.}$$

Таким чином, рейтингова шкала з дисципліни складає **R = 100 балів**.

Студенти, які набрали протягом семестру рейтинг менше **60 балів** зобов'язані виконувати залікову контрольну роботу.

Студенти, які набрали протягом семестру необхідну кількість балів (**60 балів і більше**), мають можливості отримати залікову оцінку (залік) відповідно до набраного рейтингу (табл. 1);

4. Залікова контрольна робота

Студенти, які наприкінці семестру мають рейтинг менше 60 балів, а також ті, хто хоче підвищити оцінку в системі ECTS, виконують ***залікову контрольну роботу***. При цьому до балів за МКР ($r_{МКР}$) та ДКР ($r_{ДКР}$) додаються бали за контрольну роботу і ця рейтингова оцінка є остаточною.

Завдання контрольної роботи складається з *практичного завдання зі створення програмного комплексу мовою С++. Додаткове питання* з тем лекційних занять отримують студенти, які були відсутні на певній лекції. Незадовільна відповідь на додаткова питання знижує загальну оцінку на 2 бали.

Оцінка за залікову контрольну роботу складається з 3-х частин: вірність і оптимальність розрахункового алгоритму програми – 23 балів; обробка та введення вихідних даних і виведення результатів розрахунку – 20 балів; оформлення інтерфейсу користувача, його зручність та оптимальність – 20 балів.

Оцінювання такої роботи проводиться за наступною шкалою:

- програмний комплекс відповідає всім вимогам та виконує вірно всі розрахунки – 57-63 бали;
- програмний комплекс працює вірно, проте, не відповідає певним вимогам до зручності програмного інтерфейсу – 47-56 балів;
- програмний комплекс працює в цілому вірно, проте, не відповідає більшості вимог до створення програмного інтерфейсу, або обробки вхідних даних та виведення результатів розрахунку – 38-46 балів;
- програмний комплекс працює не вірно, інтерфейс не відповідає сучасним вимогам, обробка вхідних даних та результатів не на належному рівні – 0 балів.

Якщо в програмному комплексі присутні помилки, або інтерфейс повною мірою не відповідає вимогам, то бали знімаються наступним чином:

- ∑ програмний комплекс працює, але не вірно виконує розрахунки – знімається 5 балів;
- ∑ не запрограмований обробник помилок або не всі стандартні помилки при роботі програми перехоплюються програмою чи не виводяться відповідні конкретні повідомлення по певним типам помилок – знімається 3 бали;
- ∑ відсутні підказки для користувача при роботі з програмою – знімається 3 бали;
- ∑ не передбачена можливість завантаження даних із файлу чи збереження результатів роботи у файл за допомогою відповідних засобів мови С++ – знімається 3 бали;
- ∑ не оптимально виконана компоновка програмного коду, не виділені блоки введення даних і виведення результатів окремими логічними блоками - знімається 2 бали;
- ∑ допущено дрібні помилки при створенні чи роботі програмного комплексу – знімається 1 бал за кожну помилку.

У разі, якщо студент не закінчив виконання роботи вчасно, оцінюється та частина, яка фактично виконана.

Загальна сума балів за залікову контрольну роботу розподіляється наступним чином:

- ***«відмінно»,*** завдання виконане повністю без помилок (не менше 90% потрібної інформації) – ***57-63 бали***;
- ***«добре»,*** завдання виконане в достатньому обсязі з дрібними похибками (не менше 75% обсягу завдання або незначні похибки) – ***47-56 балів***;
- ***«задовільно»,*** виконана більша частина завдання, або наявні деякі помилки (не менше 60% виконаного завдання та деякі помилки) – ***38-46 балів***;
- ***«незадовільно»,*** незадовільне виконання завдання – ***0 балів***.

$$R = r_{МКР} + r_{ДКР} + r_{ЗКР}$$

Сума балів за виконання залікової контрольної роботи та МКР і ДКР переводиться до залікової оцінки згідно з таблицею 2.1.

Таблиця 2.1 – Системи переведення балів до залікової оцінки

Бали, R	***ECTS оцінка***	***Залікова оцінка***
95-100	*A*	***Зараховано***
85-94	*B*	
75-84	*C*	

65-74	*D*	
60-64	*E*	
Менше 60	*Fx*	**Незараховано**
МКР не зараховано	**F**	**Не допущено**

3. Підготовка до лекцій

Загальні рекомендації

Лекція, як вид навчального заняття, не передбачає окремих процедур контролю підготовленості студентів. Проте це не означає, що студент має прийти на лекцію непідготовленим. Готуючись до кожної лекції з дисципліни «Технології об'єктно-орієнтованого програмування» студент повинен:

- ознайомитися з темою лекції та переліком питань, які будуть розглядатися на лекції;
- повторити попередньо освоєний матеріал, який потрібен для успішного засвоєння лекції;
- підготувати запитання з теми лекції, відповіді на які студент хотів би отримати;
- самостійно освоїти питання, які винесені на самостійне опрацювання за темою лекції.

Теми лекцій і перелік питань, що на них будуть розглядатися, наведені у цьому розділі. Для кожного питання наведені посилання на літературні джерела. Також наведений перелік матеріалів, знання яких є передумовою продуктивної роботи студента на лекції.

Запитання до теми лекції студент може готувати як письмово, так і в усній формі. Ймовірно, що в процесі читання лекції викладач дасть відповіді на питання, що зацікавили студента і частину питань буде, таким чином, знято. Ті запитання, що не знайшли відповіді у самій лекції, а також ті, відповіді на які не зрозумілі студенту, наполегливо рекомендується задати викладачеві у кінці лекції або на консультаціях.

Після прочитання кожної лекції студентам пропонується декілька питань на самостійне опрацювання за темою. Такі питання та перелік літературних джерел для їх освоєння наведені у цьому розділі.

Лекція 1. Створення простої програми на C++

Питання, які розглядаються на лекції:

- Створення консольного додатку Win32.
- Компіляція та запуск.
- Стандартні оператори введення-виведення.
- Введення даних із файлу.
- Виведення інформації у файл.
- Використання коментарів.

*Література: [*1, с. 25-36*].*

Питання, які виносяться на самостійне опрацювання:

Об’єкти, типи і значення:

- введення і тип;
- операції і оператори;
- присвоєння і ініціалізація;
- імена; типи і об’єкти;
- безпечні перетворення;
- небезпечні перетворення.

Література: [2, с. 93-120].

Лекція 2. Засоби керування ходом виконання програми

Питання, які розглядаються на лекції:

Засоби керування ходом виконання програми.

- Оператор while.
- Оператор for.
- Введення невідомої кількості даних.
- Оператор if.

Введення в класи

- Поняття класу.
- Перший погляд на функції-члени.

*Література: [*1, с. 37-51*].*

Питання, які виносяться на самостійне опрацювання:

Початкові відомості про C++:

- функція main();
- коментарі в мові C++;
- препроцесор C++ і файл iostream;
- форматування вихідного коду C++;
- стиль форматування вихідного коду програм на C++.

*Література: [*3, с. 44-50, 56-57*].*

Лекція 3. Змінні та базові типи

Питання, які розглядаються на лекції:

Прості вбудовані типи:

- арифметичні типи;
- перетворення типів;
- літерали.

Змінні:

- визначення змінних;
- об’явлення та визначення змінних;
- ідентифікатори;
- область видимості імен.

Складені типи:

- посилання;
- покажчики;
- поняття опису складених типів.

Специфікатор const:

- посилання на константу;
- покажчики та специфікатор const;
- специфікатор const верхнього рівня;
- змінні constexpr і константні вирази.

Робота з типами:

- псевдоніми типів;
- специфікатор типу auto;
- специфікатор типу decltype.

Визначення власних структур даних:

- визначення типу;
- використання визначеного типу;
- створення власних файлів заголовку.

*Література: [*1, с. 61-118*].*

Питання, які виносяться на самостійне опрацювання:

Початкові відомості про C++:

- імена заголовочних файлів;
- простори імен.

Тип даних char:

- символи і малі цілі числа.

Числа з плаваючою комою:

- запис чисел із плаваючою комою;
- типи даних із плаваючою комою;
- константи із плаваючою комою;
- переваги та недоліки типів із плаваючою комою.

*Література: [*3, с. 51-52, 93-101, 103-109*]*

Лекція 4. Типи string, vector та масиви

Питання, які розглядаються на лекції:

Побудова імен і визначення using.

Бібліотечний тип string:

- визначення та ініціалізація рядків;
- операції з рядками;
- робота з символами рядка.

Бібліотечний тип vector:

- визначення та ініціалізація векторів;
- додавання елементів у вектор;
- інші оператори з векторами.

Знайомство з ітераторами:

- використання ітераторів;
- арифметичні дії з ітераторами.

Масиви:

- визначення та ініціалізація вбудованих масивів;
- доступ до елементів масиву;
- покажчики та масиви;
- символьні рядки у стилі С;
- взаємодія із застарілим кодом;
- багатомірні масиви.

*Література: [*1, с. 123-182*].*

Питання, які виносяться на самостійне опрацювання:

Вектор:

- збільшення вектору;
- числовий приклад;
- текстовий приклад.

Переліки:

- встановлення значень переліків;
- діапазони значень для переліків.

Покажчики і вільна пам’ять:

- об’явлення і ініціалізація покажчиків;
- покажчики і числа;
- виділення пам’яті за допомогою оператора new;
- вивільнення пам’яті за допомогою оператору delete;
- використання оператора new для створення динамічних масивів.

Покажчики, масиви та арифметика покажчиків:

- покажчики та рядки;

- використання оператора new для створення динамічних структур;
- автоматичне, статичне та динамічне виділення пам'яті.

*Література: [*2, с. 148-152; 3, с. 148-180*]*

Лекція 5. Вирази

Питання, які розглядаються на лекції:

Основи роботи з виразами:

- фундаментальні концепції;
- пріоритет і порядок;
- порядок обчислення.

Арифметичні оператори.

Логічні оператори та оператори відношення.

Оператори присвоєння.

Оператори інкременту та декременту.

Оператори доступу до членів.

Умовний оператор.

Побітові оператори.

Оператор sizeof.

Оператор кома.

Перетворення типів:

- арифметичні перетворення;
- інші неявні перетворення;
- явні перетворення.

Таблиця пріоритетів операторів.

*Література: [*1, с. 187-228*].*

Питання, які виносяться на самостійне опрацювання:

Арифметичні операції в мові C++:

- пріоритет операцій і асоціативність;

- різновиди операцій ділення;
- операції ділення за модулем;
- перетворення типів даних;
- перетворення даних у виразах.

*Література: [*3, с. 110-120*]*

Лекція 6. Оператори

Питання, які розглядаються на лекції:

Прості оператори.

Операторна область видимості.

Умовні оператори:

- оператор if;
- оператор switch.

Ітераційні оператори:

- оператор while;
- традиційний оператор for;
- серійний оператор for;
- оператор do while.

Оператори переходу:

- оператор break;
- оператор continue;
- оператор goto.

Блок try та обробка виключень:

- оператор throw;
- блок try;
- стандартні виключення.

*Література: [*1, с. 233-263*].*

Питання, які виносяться на самостійне опрацювання:

Помилки:

- джерела помилок;
- помилки під час компіляції;
- помилки під час редагування зв'язків;
- помилки під час виконання програми;
- виключення;
- логічні помилки;
- оцінка;
- відладка;
- тестування.

*Література: [*2, с. 161-200*]*

Лекція 7. Функції

Питання, які розглядаються на лекції:

Основи функцій:

- локальні об'єкти;
- об'явлення функцій;
- роздільна компіляція.

Передача аргументів:

- передача аргументу по значенню;
- передача аргументу за посиланням;
- константні параметри та аргументи;
- параметри у вигляді масиву;
- функції зі змінною кількістю параметрів.

Типи значень, які повертає функція та оператор return:

- функції, які не повертають значення;
- функції, які повертають значення;
- повернення покажчика на масив.

Перевантажувані функції.

Спеціальні засоби:

- аргументи за замовчуванням;
- вбудовані функції та функції constexpr.

Перетворення типів аргументів.

Покажчики на функції.

*Література: [*1, с. 267-325*].*

Питання, які виносяться на самостійне опрацювання:

Функції мови С++:

- функції і рядки у стилі С;
- функції і структури;
- рекурсія.

Шаблони функцій:

- перевантажувані шаблони;
- явні спеціалізації;
- спеціалізації третього покоління;
- створення екземплярів і спеціалізація;
- вибір функцій;
- використання правил часткового впорядкування.

Класи і функції:

- проектування і об’явлення.

*Література: [*3, с. 305-319, 358-374; 4, с. 79-118*].*

Лекція 8. Класи

Питання, які розглядаються на лекції:

Визначення абстрактних типів даних:

- розробка власного класу;
- визначення класу;
- визначення функцій, які не є членами класу але поєднані з ним;
- конструктори;
- копіювання, присвоєння та видалення.

Керування доступом та інкапсуляція.

Додаткові засоби класу:

- члени класу;
- функції, що повертають покажчик *this;
- типи класів;
- дружні відношення.

*Література: [*1, с. 331-391*].*

Питання, які виносяться на самостійне опрацювання:

Робота з класами:

- перевантаження операцій;
- використання дружніх структур;
- клас vector;
- автоматичне перетворення і приведення типів для класів.

*Література: [*3, с. 482-535; 4, с. 119-144*].*

Лекція 9. Класи

Питання, які розглядаються на лекції:

Область видимості класу.

Застосування конструкторів:

- список ініціалізації конструктору;
- делегуючий конструктор;
- роль стандартного конструктора;
- неявне перетворення типів класу;
- агрегатні класи;
- літеральні класи;
- статичні члени класу.

*Література: [*1, с. 331-391*].*

Питання, які виносяться на самостійне опрацювання:

Класи і функції:

- реалізація.

*Література: [*3, с. 482-535; 4, с. 119-144*].*

4. Підготовка до комп'ютерних практикумів

Загальні вимоги та рекомендації

Комп'ютерні практикуми проводяться у комп'ютерних класах із окремими групами студентів. У випадку великої кількості студентів у групі, вона може поділятися на дві підгрупи і заняття проводяться у двох класах одночасно. Метою комп'ютерних практикумів є вироблення та закріплення умінь та навичок програмування мовою C++.

Студент зобов'язаний ретельно готуватися до кожного з комп'ютерних практикумів. Самостійна робота студента при підготовці до комп'ютерного практикуму з дисципліни «Технології об'єктно-орієнтованого програмування» передбачає:

– освоєння теоретичного матеріалу з теми практикуму на основі лекцій та літературних джерел;
– відповіді на питання контролю підготовленості до практикуму;
– підготовку звіту з практикуму.

Перелік теоретичного матеріалу, який необхідно освоїти та питання для контролю підготовленості до кожного з практикумів наведені у цьому розділі. Звіт із комп'ютерного практикуму оформлюється на аркушах формату А4 і має містити:

- *титульний аркуш* із зазначенням номеру практикуму, його теми, групи і прізвища студента, який виконав роботу та прізвища викладача, який її перевірив (зразок титульного аркуша наведено в додатку А);
- *мету* та *завдання* практикуму;
- *короткі теоретичні відомості* для висвітлення питань, яким присвячена робота;
- *порядок виконання* завдань практикуму, з зазначенням виконуваних дій та операцій;
- *результати виконання* завдань у роздрукованому вигляді;
- *код програмного модуля*.

Перші три пункти цього переліку готуються студентом заздалегідь і пред'являються викладачу на початку практикуму. Це є необхідною умовою його успішного виконання. Результати практикуму друкуються, очевидно, після виконання завдань і перевірки їх викладачем. Результати роботи створеної

програми, а також код програмного модулю слід друкувати як частину протоколу на аркушах формату А4.

Виконання кожного з комп'ютерних практикумів проводиться за методикою, викладеною у відповідних методичних вказівках [5]. Порядок оцінювання комп'ютерних практикумів наведено в розділі 2.

Комп'ютерний практикум 1. Створення консольного додатку

Мета: ознайомитись із навичками роботи в командному режимі та принципами створення *консольного додатку* C++. Засвоїти методи *компіляції*, *запуску* та *відлагоджування* програми. Вивчити оператори *потоків введення-виведення* в консольному додатку.

Завдання: створити консольний додаток для відпрацювання основних навичок створення та компіляції додатку, а також реалізувати поставлену задачу згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Що таке функція `main()`, з чого вона складається?
2) Який тип даних має повертати функція `main()`?
3) Що таке вбудований тип даних?
4) Що таке тіло функції, з чого воно складається?
5) Що таке оператор `return`, для чого він використовується?
6) Що таке потік введення чи виведення?
7) Що таке стандартне введення та стандартне виведення?
8) Що таке об'єкт `cerr` та `clog` і для чого вони використовуються?
9) Що таке директива `#include` та для чого вона використовується?
10) Що таке оператор виведення та для чого він використовується?
11) Що таке оператор введення та для чого він використовується?
12) Що таке маніпулятор та для чого він використовується?
13) Що таке простір імен?
14) Що таке оператор області видимості та для чого він використовується?
15) Що таке ініціалізація?

Комп'ютерний практикум 2. Оператор if та логічні оператори

Мета: ознайомитись із використанням *математичних функцій* у C++, вивчити роботу *умовного оператора if*, розглянути можливості застосування *логічних операторів* та *операторів відношення*.

Завдання: створити консольний додаток для реалізації обчислень поставленої задачі згідно отриманого варіанту завдання з використанням умовного оператора *if* та математичних функцій.

Питання для контролю підготовленості до практикуму:

1) Опишіть структуру та принцип роботи оператора `if`?
2) Унарні та парні (бінарні) оператори. У чому їх відмінність?
3) Перерахуйте арифметичні оператори згідно з їх пріоритетом.
4) Перерахуйте логічні оператори та оператори відношення.
5) Перерахуйте складені оператори присвоєння та поясніть принцип їх дії.
6) Що таке оператор прирощення та оператор зменшення? Поясніть їх використання на прикладі.
7) Що таке префіксний оператор та постфіксний оператор? Поясніть відмінність між ними.
8) Що таке об'явлення `using` та для чого воно використовується?
9) Що потрібно для використання в програмі математичних функцій? Перерахуйте основні математичні функції.

Комп'ютерний практикум 3. Засоби керування процесом виконання програми

Мета: ознайомитись із операторами для зміни послідовності виконання програми, вивчити роботу *оператора ітераційного циклу while*, вивчити роботу *оператора арифметичного циклу for*, вивчити роботу *генератора випадкових чисел*.

Завдання: створити консольний додаток для реалізації обчислень поставленої задачі згідно двох частин отриманого варіанту завдання з використанням операторів циклу *while* та *for*, а також умовного оператора *if*.

Питання для контролю підготовленості до практикуму:

1) Опишіть структуру та принцип роботи оператора `while`?
2) Що таке блок? Наведіть приклад.

3) Опишіть структуру та принцип роботи оператора `for`?
4) З чого складається заголовок оператора `for`?
5) У чому відмінність між операторами циклу `for` та `while`?
6) Для чого використовуються об'єкти класу `default_random_engine`?
7) Як згенерувати випадкові цілі числа в заданому діапазоні?
8) Як згенерувати випадкові дійсні числа в заданому діапазоні?
9) Як забезпечити генерування різних послідовностей випадкових чисел при кожному запуску програми?

Комп'ютерний практикум 4. Змінні та базові типи

Мета: ознайомитись із символьними та чисельними обчисленнями засобами C++; вивчити існуючі типи даних, структуровані типи даних (*вказівники, посилання, створення власної структури даних*); навчитись створювати *файли заголовку*; відпрацювати засоби візуалізації розрахунків у C++. Навчитись використовувати кирилицю в консолі: коректне розпізнавання введених символів кирилицею; коректне виведення в консолі символів кирилицею.

Завдання: створити консольний додаток для реалізації обчислень поставленої задачі згідно двох частин отриманого варіанту завдання з використанням створеної структури даних та покажчиків і посилань.

Питання для контролю підготовленості до практикуму:

1) Перерахуйте арифметичні типи мови C++ та вкажіть, які значення вони можуть зберігати.
2) Що таке знакові та беззнакові цілочисельні типи? Перерахуйте беззнакові цілочисельні типи.
3) Розкажіть що таке автоматичне перетворення типів та за якими правилами воно виконується.
4) Що таке змінна? З чого складається визначення змінної?
5) Що таке ініціалізація та для чого вона потрібна?
6) Що таке ідентифікатори? Які існують правила до створення ідентифікатора?
7) Що таке область видимості? Яка область видимості у об'єктів класу?
8) Що таке складений тип? Які складені типи ви знаєте?
9) Що таке посилання? Які правила використання та обмеження існують для посилань?

10) Що таке покажчик? Які правила використання існують для покажчиків? Як звернутись до значення покажчика?
11) Що таке специфікатор типу `auto`? Коли і як його слід використовувати?
12) Що таке специфікатор типу `decltype`? Коли і як його слід використовувати?
13) Що таке власна структура даних або клас? Із чого складається визначення класу?
14) Що таке змінні-члени? Що таке внутрішньокласовий ініціалізатор?
15) Для чого потрібні файли заголовку? Як підключити заголовок до програми? Як організувати захист заголовку?
16) Що таке змінні препроцесора? Які правила створення імен змінних препроцесору та файлів заголовку загальноприйняті? Як працює директива `#include`?
17) Які таблиці кодування використовуються у `Windows`? У чому їх відмінності?
18) Що потрібно для коректного відображення кирилиці у консольному додатку? Для чого використовують файл заголовку `windows.h`? Для чого потрібні функції `SetConsoleCP()` і `SetConsoleOutputCP()`?

Комп'ютерний практикум 5. Типи string та масиви

Мета: ознайомитись із бібліотечним типом *string*: операції з рядками; робота з символами рядка. Засвоїти використання *оператору індексування* для роботи з *структурами даних*. Ознайомитись із використанням *ітераторів* для типу *string*. Вивчити роботу з *масивами*: *визначення* та *ініціалізація*; доступ до елементів; застосування *покажчиків*; *динамічні масиви*.

Завдання: створити консольний додаток для роботи з рядковим текстом, а також проведення розрахунків за допомогою одномірного масиву для вирішення поставленої задачі згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Для чого потрібний тип `string`, як створити об'єкт даного типу у програмі?
2) Як можна ініціалізувати об'єкти типу `string`? Наведіть приклади способів ініціалізації.
3) Що таке пряма ініціалізація та ініціалізація копією?

4) Перерахуйте основні операції визначені для класу `string`.
5) Як можна організувати циклічне введення невизначеної кількості даних?
6) Для чого використовується функція `getline()`, в чому її відмінність від оператора >>?
7) Для чого потрібен допоміжний тип даних `size_type` у класі `string`? Як створити об'єкт даного типу?
8) Перерахуйте функції для виконання операцій із символами рядка та їх призначення.
9) Що таке оператор індексування? Для чого він потрібен?
10) Що таке ітератори? Для чого вони використовуються?
11) Для чого потрібні функції-члени `begin()` та `end()` у контейнерах?
12) Який тип мають ітератори? Як визначити об'єкт даного типу?
13) Перерахуйте та поясніть стандартні операції з ітераторами.
14) Як за допомогою ітератора звернутись до значення контейнеру? Чому в циклі `for` при перевірці умови досягнення останнього елементу контейнера використовують оператор `!=` замість оператора <?
15) Як організувати переміщення по елементам контейнеру за допомогою ітераторів?
16) Що таке масив? Як об'явити масив?
17) Що таке багатомірний масив? Як можна ініціалізувати масив? Як отримати доступ до елементів масиву за допомогою оператора індексації?
18) Що таке динамічний масив? Як об'явити динамічний масив? Як організувати доступ до елементів масиву за допомогою покажчиків?
19) Для чого потрібен оператор `delete`? Яка форма його запису?

Комп'ютерний практикум 6. Тип vector

Мета: вивчити бібліотечний тип *vector*: *визначення* і *ініціалізація*; додавання елементів; інші операції. Засвоїти використання *оператору індексування* для типу *vector*. Ознайомитись із використання *ітераторів* для типу *vector*. Опрацювати використання двомірних *векторів*.

Завдання: створити консольний додаток для організації роботи з одномірним та двомірним масивом із використанням бібліотечного типу *vector* для реалізації поставленої задачі згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Що таке вектор? Що потрібно для використання векторів у програмі?
2) Що таке шаблон класу? Що таке створення екземпляру шаблону? Наведіть приклади.
3) Наведіть способи ініціалізації об'єктів класу `vector`.
4) Що таке ініціалізація переліком та ініціалізація значення для об'єкту типу `vector`? У чому відмінність використання круглих та фігурних дужок під час ініціалізації об'єкту типу `vector`, наведіть приклади?
5) Перерахуйте основні операції з векторами.
6) Поясніть як можна отримати доступ до елементів вектора за допомогою індексування та ітераторів? Наведіть приклади.
7) Що таке псевдонім типу та які методи створення псевдонімів ви знаєте? Наведіть приклади об'явлення псевдонімів.
8) Що таке оператор звернення до значення та оператор звернення до члену? Наведіть приклади.
9) Для чого потрібен оператор стрілки? Наведіть приклади.
10) Перерахуйте основні відмінності між масивами та векторами. Який із цих типів більш зручний у використанні?

Комп'ютерний практикум 7. Оператори

Мета: вивчити *умовні оператори*: оператор *if*, оператор *switch*; *ітераційні оператори*: цикл *do while*, *серійний оператор циклу for*; *оператори переходу*: *break*, *continue*. Засвоїти принципи застосування *обробки виключень*: робота з оператором *throw* та блоком *try*. Відпрацювати роботу зі *структурами даних:* використання оператора *struct*; створення та підключення файлів заголовку.

Завдання: створити консольний додаток для реалізації обчислень поставленої задачі згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Що таке оператор? Що таке оператор керування потоком виконання? Що таке оператор виразу?
2) Що таке пустий оператор? Які особливості його використання?
3) Що таке складений оператор? Що таке область видимості блоку, область видимості керуючої структури?
4) Що таке умовний оператор (`?:`), наведіть приклад його використання.

5) Що таке умовний оператор `switch`? Що таке блок `case` та блок `default`, для чого вони використовуються?
6) Що таке оператор `break`? Які особливості використання міток `case`? Наведіть приклад використання оператора `switch`?
7) Що таке ітераційні оператори? Що таке цикли з передумовою та післяумовою?
8) Що таке серійний оператор `for`? Опишіть принцип його роботи. В чому відмінність серійного оператора `for` від традиційного?
9) Що таке оператор `do while`? В чому його відмінність від оператора `while`? Наведіть приклад організації програми з можливістю повторної організації розрахунків по запиту користувача.
10) Опишіть особливості використання операторів `break` та `continue`, у чому їх відмінність?
11) Що таке виключення? Що таке обробка виключень?
12) Що таке оператор `throw`? Що таке блок `try`? Що таке розділ `catch`?
13) Які бібліотечні класи виключень ви знаєте, в яких заголовках вони визначені? Перерахуйте стандартні класи виключень заголовку `stdexcept`?
14) Для чого потрібна функція `what()`? Наведіть приклад її використання.
15) Як організувати власну структуру даних? Як створити файл заголовку для об'явлення класу?

Комп'ютерний практикум 8. Функції

Мета: вивчити принципи створення *функцій користувача* та роботи з ними: *визначення функції*; *тіло функції*; тип значення, яке повертає функція із використанням оператору *return*; *об'явлення функцій* у *файлі заголовку*; виклик функції; передача аргументів; створення програм за допомогою функцій.

Завдання: створити консольний додаток для обчислення значень функціональної залежності згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Що таке функція?
2) Що таке визначення функції, тип повернутого значення, параметри, тіло функції?
3) Що таке оператор виклику функції, з чого він складається?

4) Для чого потрібен оператор `return`?
5) Що таке аргументи функції? Для чого використовуються аргументи?
6) Що таке перелік параметрів функції? Як об'являються параметри функції? Чи може бути функція без параметрів? Чи може функція не повертати значення? Наведіть приклади.
7) Що таке область видимості імен та тривалість існування об'єкту?
8) Що таке локальні змінні? Чим вони відрізняються від глобальних?
9) Що таке автоматичний об'єкт? Що таке локальний статичний об'єкт? Наведіть приклади?
10) Що таке об'явлення функції? Що таке інтерфейс функції? Наведіть приклад.
11) Як створити зовнішню функцію? Як підключити зовнішню функцію користувача для використання у файлі вихідного коду?
12) Що таке роздільна компіляція та для чого вона потрібна?
13) Чим відрізняється передача аргументу за посиланням від передачі аргументу за значенням?
14) Скільки значень може повертати функція? Чи може функція повернути кілька значень, якщо так, то як це реалізувати? Наведіть приклади.
15) Як потрібно визначати параметр функції, який не повинен змінюватись? Наведіть приклади.
16) Що таке перевантажені функції? Які особливості їх використання? Наведіть приклади.

Комп'ютерний практикум 9. Класи

Мета: вивчити принципи створення власних *класів* та роботи з ними: *визначення класу*; створення *конструктору класу*; створення *інтерфейсу класу*; створення *реалізації класу*; *об'явлення функцій-членів класу*; виклик *функції-членів* класу; створення змінної типу власного класу; передача аргументів *функціям-членам* класу; створення програм із обчислювальною частиною, яка реалізована у власному класі та його *функціях-членах*.

Завдання: створити консольний додаток для розрахунку інтегралу методом Сімпсона згідно отриманого варіанту завдання.

Питання для контролю підготовленості до практикуму:

1) Що таке абстракція даних? Що таке абстрактний тип даних?

2) Що таке інтерфейс та реалізація класу?
3) Що таке інкапсуляція? Що таке специфікатори доступу?
4) У чому різниця між використанням ключових слів `struct` та `class`?
5) Що таке функції-члени? Де та як вони об'являються та визначаються?
6) Що таке область видимості класу? Наведіть приклад визначення функції-члену поза тілом класу.
7) Для чого використовується параметр `this`? Як звернутися до члену об'єкту класу? Наведіть приклади.
8) Як підключити допоміжні функції для використання класом без їх включення у клас?
9) Що таке конструктор, для чого він використовується?
10) Що таке стандартний конструктор? Що таке синтезований стандартний конструктор? Як об'явити стандартний конструктор?
11) Що таке перелік ініціалізації конструктора? Для чого він потрібен?
12) Де потрібно визначати конструктор? Із чого складається конструктор? Наведіть приклади об'явлення та визначення конструкторів?
13) Що таке дружні відносини? Як об'явити дружню функцію? Як зробити дружню функцію доступною для користувачів класу?

5. Самостійна робота

Перелік тем для самостійного вивчення

Потоки введення і виведення:

- модель потоку введення і виведення;
- файли;
- відкриття файлу;
- читання і запис файлу;
- обробка помилок введення-виведення;
- зчитування окремого значення;
- оператори введення, які визначені користувачем;
- оператори виведення, які визначені користувачем;
- стандартний тип введення;
- читання структурованого файлу.

*Література: [*2, с. 361-398*].*

Виведення на екран, використання графічних примітивів:

- використання бібліотеки графічного інтерфейсу користувача;
- координати;
- клас shape;
- використання графічних примітивів.

Графічні класи.

Проектування графічних класів.

*Література: [*2, с. 431-530*].*

Графічні функції і дані:

- побудова простих графіків;
- клас function;
- вісі;
- апроксимація;
- графічні дані.

Графічний інтерфейс користувача:

- альтернативи інтерфейсу користувача;
- просте вікно;
- додавання меню;
- відлагоджування інтерфейсу користувача.

*Література: [*2, с. 531-590*].*

6. Домашня контрольна робота

Вимоги до виконання та оформлення роботи

Домашня контрольна робота виконується студентом самостійно в час, вільний від занять. Із усіх питань, які виникають у процесі виконання домашньої контрольної роботи, студент може звернутися до викладача у час, відведений для проведення консультацій.

Завдання на домашню контрольну роботу з дисципліни «Технології об'єктно-орієнтованого програмування» видається студентам після прослуховування переважної частини лекційного курсу, зазвичай на 8-9 тижні семестру. Видача завдань та закріплення варіантів домашньої контрольної роботи обов’язково фіксується у листі реєстрації індивідуальних завдань.

Час виконання роботи складає один місяць, рахуючи з дати видачі завдання. Протягом всього цього терміну студенти можуть звертатися до викладача за консультаціями щодо неї. Після завершення терміну виконання, консультації з питань домашньої контрольної роботи завершуються і студентам надається додатково 3-4 дні для здачі роботи. У разі, якщо студент не дотримав термінів здачі роботи, він отримує до свого семестрового рейтингу штрафні рейтингові бали (див. розділ 2).

Виконана робота здається викладачу в роздрукованому вигляді з додаванням електронної версії програмного модуля свого варіанту завдання. Роздрукований примірник домашньої контрольної роботи має складатись з:

1) *Титульного аркушу*;
2) *Завдання* (загальної частини та індивідуального завдання);
3) *Теоретичних відомостей* (коротко описуються застосовані при виконанні теоретичні аспекти);
4) *Ходу виконання* (описується створення програмного модулю, описується алгоритм роботи з програмним модулем, наводиться приклад застосування програмного модулю та результати розрахунку);
5) *Коду програмного модулю* (наводиться повний код програмного модуля з усіма окремими файлами модулів *.cpp та файлами заголовку *.h).

До оголошення оцінок за контрольну роботу студент повинен зберігати електронні варіанти проекту з вихідними файлами виконаного завдання.

Методичні рекомендації до виконання роботи

З метою закріплення отриманих знань та навичок, у рамках кредитного модулю передбачене виконання домашньої контрольної роботи. Домашня контрольна робота виконується за темою: «Створення та використання класів користувача». Для виконання роботи студентам видаються додому індивідуальні завдання. За результатами домашньої контрольної роботи робиться висновок про достатнє (недостатнє) володіння студентом матеріалом дисципліни.

Мета: вивчити принципи створення власних *класів* та роботи з ними: *визначення класу*; створення *конструктору класу*; створення *інтерфейсу класу*; створення *реалізації класу*; *об'явлення функцій-членів класу*; виклик *функції-члену* класу; створення змінної типу власного класу; передача аргументів *функціям-членам* класу; створення програм з обчислювальною частиною, яка реалізована у власному класі та його *функціях-членах*.

Завдання: створити консольний додаток який реалізує розрахунок з використанням математичного методу згідно отриманого варіанту завдання у вигляді класу.

Загальні вимоги.

1) Створити консольний додаток з ім'ям виду *PrizvischeDKPR*.
2) Програмний код модуля моє бути чітко структурований.
3) Імена об'єктів мають нести сенсові навантаження.
4) Програмний код має супроводжуватись коментарями в тексті програми.

Вимоги до виконання.

1) Створити за допомогою *файлу заголовку* власний *клас даних*, який реалізуватиме розрахунок відповідним математичним методом (згідно варіанту).
2) Введення вхідних даних реалізувати в основному тілі програми.
3) Всі етапи обчислення реалізувати у власному класі.
4) У класі передбачити три конструктори: конструктор за замовчуванням; конструктор для ініціалізації класу всіма необхідними даними (коефіцієнти рівняння, межі інтегрування, точність інтегрування); для ініціалізації класу лише мінімально необхідними даними (наприклад, у реалізації класу передбачити ініціалізацію коефіцієнтів рівняння, меж інтегрування та точності інтегрування значеннями за замовчуванням);

5) У класі створити інтерфейс класу (відкриту для користувача частину) та реалізацію класу (інкапсульовану всередині класу його частину). Інтерфейсна частина класу та частина його реалізації повинні мати не менше однієї функції-члену кожна.
6) У програмі передбачити можливість вибору користувача: вводити всі вхідні дані (наприклад, коефіцієнти рівняння, межі інтегрування, точність інтегрування) для ініціалізації об'єкту класу за допомогою розширеного конструктора; вводити лише мінімально необхідні дані (наприклад, інтервал інтегрування) для виклику конструктора класу, який передає ініціалізацію інших вхідних даних значеннями за замовчуванням у блоці реалізації класу.
7) Передбачити виведення результатів розрахунку на кожній ітерації, а також знайденого результату з заданою точністю на екран та у зовнішній файл.

Теоретичні відомості

Тема 6.1. Операції введення-виведення та області видимості

6.1.1 Типи istream та ostream

В самій мові C++ немає операторів для *введення та виведення* (Input/Output - IO). Їх функції забезпечує *стандартна бібліотека* (standard library). Основу *бібліотеки* iostream складають два типи, istream та ostream, які представляють потоки введення та виведення відповідно. *Потік* (stream) – це послідовність символів, яка записується чи зчитується з пристрою введення-виведення деяким способом. Термін «потік» передбачає, що символи надходять та передаються послідовно протягом певного часу.

В бібліотеці визначені чотири об'єкти введення-виведення. Для здійснення введення використовують об'єкт cin (вимовляється «сі-ін») типу istream. Цей об'єкт називають також *стандартним введенням* (standard input). Для виведення використовується об'єкт cout (вимовляється «сі-аут») типу ostream. Його часто згадують як *стандартне виведення* (standard output).

В бібліотеці визначені ще два об'єкти типу ostream – це cerr та clog (вимовляється як «сі-ерр» та «сі-лог» відповідно). Об'єкт cerr називають також *стандартною помилкою* (standard error) та, як правило, використовують в програмах для створення попереджень і повідомлень про помилки, а об'єкт clog – для створення інформаційних повідомлень.

6.1.2 Використання директиви препроцесору

Для включення компілятором в програму бібліотеки `iostream` використовується *директива препроцесору* (`preprocessor directive`)

```
#include <iostream>
```

Ім'я в кутових дужках – це *заголовок* (`header`). Кожна програма, яка використовує засоби, що зберігаються в бібліотеці, має підключити відповідний заголовок. *Директива* `#include` має бути написана в одному рядку та знаходитись поза тілом функції. Зазвичай всі директиви `#include` програми розташовані на початку файлу вихідного коду.

6.1.3 Оператори виведення та виведення

Перший оператор в тілі функції `main()` виконує *вираз* (`expression`). В мові C++ вираз складається з одного чи кількох *операндів* (`operand`) і, як правило, *оператора* (`operator`). Щоб відобразити підказку на стандартному пристрої виведення, використовується *оператор виведення* (`output operator`), або *оператор* `<<`.

```
std::cout << "Input t1" << std::endl;
```

В даному рядку об'єднані два оператори виведення. Перший оператор `std::cout << "Input t1"` виводить повідомлення для користувача. Це повідомлення є *рядковим літералом* (`string literal`) – послідовність символів, які заключені у подвійні лапки. Оператор складається з двох операндів, лівий повинен бути об'єктом класу `ostream`, а правий операнд – це значення для відображення. Оператор `<<` заносить передане значення в об'єкт `cout` класу `ostream`. Таким чином, результатом буде об'єкт класу `ostream`, в який записане передане значення. Текст у подвійних лапках виводиться на стандартний пристрій виведення.

Оператор введення `>>` (`input operator`) поводить себе аналогічно оператору виведення. Його лівим операндом (відносно оператору `>>`) є об'єкт класу `istream`, а правим операндом – об'єкт, який запам'ятовує отримані дані. Подібно оператору виведення, оператор введення повертає як результат свій лівий операнд (об'єкт класу `istream`), тому можна об'єднувати кілька операторів введення

```
std::cin >> v1 >> v2;
```

що рівноцінно

```
std::cin >> v1;
std::cin >> v2;
```

Другий оператор в наведеному вище прикладі `std::cout << std::endl` виводить `endl` (читається як «енд-ел») – спеціальне значення, що зветься *маніпулятором* (`manipulator`). При його запису у потік виведення здійснюється перехід на новий рядок та скидання *буферу* (`buffer`), який пов'язаний з даним пристроєм. Скидання буферу гарантує, що все виведення, яке програма сформувала на даний момент, буде одразу записане в потік виведення, а не буде очікувати запису, знаходячись у пам'яті.

Оскільки перша частина оператору повертає об'єкт класу `ostream`, то два оператори

```
std::cout << "Input v1 and v2";
std::cout << std::endl;
```

можна об'єднати, що і було зроблено.

6.1.4 Оператор області видимості ::

В наведених операторах використана форма запису `std::cout` та `std::endl` замість `cout` та `endl`. Префікс `std::` означає, що імена `cout` та `endl` визначені у *просторі імен* (`namespace`) на ім'я `std`. Простори імен дозволяють уникнути імовірних конфліктів, причинами яких може стати співпадіння імен, визначених в різних бібліотеках. Всі імена, які визначені в стандартній бібліотеці, знаходяться у просторі імен `std`. В запису `std::cout` застосовується *оператор області видимості* `::` (`scope operator`), який вказує що тут використовується ім'я `cout`, яке визначене в просторі імен `std`.

6.1.5 Внутрішня область видимості

Внутрішня область видимості (inner scope) - це область видимості, вкладена в іншу область видимості. В наступному прикладі проілюстроване оголошення змінної `std::string область` у внутрішній області видимості.

```
std::string область("Глобальна область видимості функції main");
if (true)
{
	std::string область("Внутрішня область видимості блоку if");
	std::cout << область << std::endl;
}
```

За межами своєї внутрішньої області змінна `std::string область` буде невідома.

6.1.6 Глобальна область видимості

Глобальна область видимості (global scope) - це область видимості, зовнішня для всіх інших областей видимості. В програмному коді нижче оголошена змінна `std::string область` в глобальній області видимості. Вона буде відома у всіх вкладених областях видимості. Проте у блоці оператора `if` оголошена нова змінна з ім'ям `область`. У блоці оператора `if` локальна змінна `область` матиме пріоритет над глобальною змінної з тим самим ім'ям. При зміні значення змінної `область` у блоці `if` мінятиметься лише локальна змінна, а значення глобальної змінної залишиться тим самим.

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include <locale>;
#include "Windows.h"

std::string область("Глобальна область видимості");

int main()
{
	//Встановлення таблиці кодування для укр. мови
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	std::cout << область << std::endl;
	std::string область("Глобальна область видимості функції main");
	if (true)
	{
		std::string область("Внутрішня область видимості блоку if");
		std::cout << область << std::endl;
	}
	std::cout << область << std::endl;
	return 0;
}
```

Результат роботи прикладу:

```
Глобальна область видимості
Внутрішня область видимості блоку if
Глобальна область видимості функції main
```

6.1.7 Блок

Блок (block) - це послідовність будь-якої кількості операторів, укладена в фігурні дужки. Блок операторів може бути використаний скрізь, де очікується

один оператор. Блоки можуть мати функції та інші оператори. Блоки також можуть бути вкладеними. Також блок визначає свою область видимості.

```
#include "stdafx.h"
#include <iostream>

int main()
{ // Початок блоку функції main()
      std::cout << "Enter an integer: ";
      int значення;
      std::cin >> значення;
      if (значення >= 0)
      { // Початок вкладеного блоку
            std::cout << значення << " is a positive integer (or zero)"
                  << std::endl;
            std::cout << "Double this number is " << значення * 2 <<
std::endl;
      } // Кінець вкладеного блоку
      else
      { // Початок іншого вкладеного блоку
            std::cout << значення << " is a negative integer" <<
std::endl;
            std::cout << "The positive of this number is " << -значення
                  << std::endl;
      } // Кінець іншого вкладеного блоку
      return 0;
}// Кінець блоку функції main()
```

Певна змінна може знаходитись *в області видимості (in scope)*. Її ім'я буде видиме в межах поточної області видимості.

```
int main()
{// Блок 1, область видимості функції main()
      int x = 32, y = 3;
      if (y>0)
      {// Блок 2, область видимості оператора if
            // Змінна x переоголошена у області видимості блоку 2, x=-15
int x = -15;
            std::cout << "Block 2: x = " << x << std::endl;
      }// Кінець блоку 2
      //У області видимості блоку 1 змінна x зі значенням -15 невідома,
      //тут існує інша змінна x зі значенням 32
      std::cout << "Block 1: x = " << x << std::endl;
      return 0;
}// Кінець блоку 1
```

Результат роботи прикладу:

```
Block 2: x = -15
Block 1: x = 32
```

6.1.8 Ініціалізація змінних

Після запиту на введення даних, необхідно організувати читання введених користувачем даних. Але перед цим потрібно визначити *змінні* (variable), в які будуть запам'ятовуватись введені значення.

```
int v1 = 0, v2=0;
```

Ці змінні визначені як такі, що відносяться до типу int. Тип int є вбудованим типом даних для цілочисельних значень. Окрім того, ми також *ініціалізуємо* (initialize) змінні, тобто присвоюємо вказане значення на момент створення.

Тема 6.2. Форматоване введення-виведення в C++

6.2.1 Управління форматом введення-виведення

Можливість управляти форматом введенням-виведенням в C++, забезпечують функції-члени, прапори і маніпулятори. Прапори, функції та маніпулятори виконують одну й ту ж саму задачу - задають певний формат введення/виведення інформації в потоках. Введення/виведння на екран / з екрану в C++ здійснюється за допомогою операторів cin і cout відповідно, а значить маніпулятори форматування використовуються спільно з цими операторами введення/виведення. Різниця між функціями, прапорами і маніпуляторами форматування полягає в способі їх застосування. Тепер розглянемо способи застосування об'єктів форматування.

6.2.2 Форматування введення-виведення за допомогою функцій-членів

В наступному прикладі наведені основні функції форматування введення/виведення.

```
// Основні функції-члени форматування:
cout.fill('/* symbol */'); // Встановлює символ заповнювач,
// де symbol - символ заповнювач, символ передається в одинарних лапках
cout.width(/* width_field */); // Задає ширину поля,
// де width_field - кількість позицій (одна позиція вміщує один символ)
cout.precision(/* number */); // Задає максимальну кількість цифр для
// відображенн, де number - кількість цифр
```

Доступ до *функцій* здійснюється через оператор *точка*, а в круглих дужках передається аргумент. Аргумент *функції* `fill()` може передаватися у вигляді символу, обрамленого одинарними лапками, або у вигляді числа (код символу).

Функція `width()` задає ширину поля, тобто наступне виведення буде мати вказану у параметрі функції `width()` загальну довжину разом з усіма виведеними символами. Якщо кількість символів менша ніж вказано у функції `width()`, то поле доповниться потрібною кількістю пробілів перед виведеним значенням. При використанні функції width() у парі з функціює fill() можна задати інші символи замість пробілів, якими буде доповнюватись поле виведення.

```
double j = 123.45;
std::cout.fill('*'); // Встановлює символ заповнювач
std::cout.width(10); // Задає ширину поля
std::cout << j;
```

Результатом роботи даного коду буде виведене наступне значення

```
****123.45
```

Тобто наше дійсне число `123.45`, яке складається з шести символів разом з крапкою, доповнене чотирма вказаними у функції `fill()` символами `*` до загальної кількості символів – десять.

Функція `precision()` задає кількість цифр, які будуть відображатися у числа з плаваючою точкою в наступному операторі виведення. При цьому рахуються всі символи як до десяткової точки, так і після неї. Для прикладу розглянемо наступний код:

```
double i = 12.345678;
std::cout.precision(3);
std::cout << i << std::endl;
std::cout.precision(5);
std::cout << i << std::endl;
std::cout.precision(10);
std::cout << i << std::endl;
```

Результатом буде наступне виведення:

```
12.3
12.346
12.345678
```

В першому випадку `precision(3)` встановлена кількість символів 3 і відповідно виводиться значення `12.3`, тобто значення округлене до трьох символів, інші відкидаються. В другому випадку `precision(5)` встановлена

кількість символів 5 і відповідно виводиться значення 12.346, тобто значення округлюється до п'яти символів, інші відкидаються. В третьому випадку precision(10) встановлена кількість символів 10 більша ніж кількість символів у значенні з плаваючою точкою і відповідно виводиться значення 12.345678, тобто значення з меншою кількістю символів ніж задано у функції precision().

Результат роботи функції precision() залежить від заданого формату виведення числових значень. Формат виведення числових значень можна задати за допомогою функції setf().

Недоліком функцій форматування введення/виведення є те, що кожна функція зазвичай *спрацьовує лише один єдиний раз на наступному операторі виведення*, після чого формат виведення повертається до форматування за замовчуванням.

Одних функцій мало для форматування потоків введення/виведення, тому в C++ передбачено ще один спосіб форматування - прапори.

Функція setf() встановлює прапори форматування чи прапори форматування та маску формату:

```
setf(прапор)
setf(прапор, маска)
```

При цьому комбінація прапорів та масок може бути наступною (табл. 6.2.1).

Таблиця 6.2.1 –Прапори та маски функції setf()

Маска	Прапори
adjustfield	left, right або internal
basefield	dec, oct або hex
floatfield	scientific або fixed

Призначення прапорів наведене в табл. 6.2.2.

Наступний приклад дозволяє доповнювати нулями кількість символів дійсного числа до значення вказаного у функції precision() за допомогою встановленого формату у функції setf().

```
double i = 12.345678;
std::cout.setf(std::ios::fixed, std::ios::floatfield);
std::cout.precision(10);
std::cout << i << std::endl;
```

Результатом роботи даного коду буде наступне виведення:

```
12.3456780000
```

Тобто при встановленій масці `floatfield` та формату чисел з плаваючою точкою `fixed` функція `precision()` тепер вказує не загальну кількість символів в числі з плаваючою точкою, а кількість символів після коми. При цьому число вже буде доповнюватись нулями, якщо кількість символів після коми менша ніж вказано у функції `precision()`.

Функція `unsetf()` відміняє певний режим форматування.

6.2.3 Форматування введення-виведення за допомогою прапорів

Прапори форматування дозволяють включити або виключити один з параметрів введення/виведення. Щоб встановити прапор введення/виведення, необхідно викликати функцію `setf()`, якщо необхідно відключити прапор виведення, то використовується функція `unsetf()`. Далі показані конструкції установки і зняття прапорів виведення.

```
// Установка прапора виведення
cout.setf(ios :: /* name flag */);
// Де name_flag - це ім'я прапора
```

Доступ до функцій оператора виведення виконується через оператор *крапка*. Метод (функція) `setf()` приймає один аргумент - ім'я прапора. Прапори виведення оголошені в класі `ios`, тому, перед тим, як звернутися до прапора, необхідно написати ім'я класу - `ios`, після якого, за допомогою *оператора дозволу області дії* (`::`), викликати потрібний прапор.

```
// Зняття прапора виведення
cout.unsetf(ios :: /* name flag */);
// Де name_flag - це ім'я прапора
```

Якщо при введенні/виведенні необхідно встановити (зняти) кілька прапорів, то можна скористатися *порозрядною логічною операцією АБО* `|`. У цьому випадку конструкція мови C++ буде такою:

```
// Установка декількох прапорів
cout.setf(ios :: /* name_flag 1 */ | ios :: /* name_flag 2 */ | ios ::
/* name_flag_n */);
// Зняття декількох прапорів
cout.unsetf(ios :: /* name_flag1 */ | ios :: /* name_flag2 */ | ios ::
/* name_flag_n */);
```

У таблиці 6.2.2 детально описані основні прапори форматування, а також показані приклади використання прапорів.

Таблиця 6.2.2 –Прапори форматування в С++

Прапор	Призначення	Приклад	Результат

boolalpha	Виведення логічних величин в текстовому вигляді (true, false)	`cout.setf(ios::boolalpha);`
		`bool log_false = 0, log_true = 1;`
		`cout << log_false << endl << log_true << endl;`
			false
			true

oct	Введення/виведення величин в вісімковій системі числення (спочатку знімаємо прапор dec, потім встановлюємо прапор oct)	`cout.unsetf(ios::dec);`
		`cout.setf(ios::oct);`
		`int value;`
		`cin >> value;`
		`cout << value << endl;`
			введення:**99_{10}**
			виведення:**143_8**

dec	Введення/виведення величин в десятковій системі числення (прапор встановлений за замовчуванням)	`cout.setf(ios::dec);`
		`int value = 148;`
		`cout << value << endl;`
			148

hex	Введення/виведення величин в шістнадцятковій системі числення (спочатку знімаємо прапор dec, потім встановлюємо прапор hex)	`cout.unsetf(ios::dec);`
		`cout.setf(ios::hex);`
		`int value;`
		`cin >> value;`
		`cout << value << endl;`
			введення:**99_{10}**
			виведення:**63_{16}**

showbase	Виводити індикатор основи системи числення	`cout.unsetf(ios::dec);`
		`cout.setf(ios::oct | ios::showbase);`
		`int value;`
		`cin >> value;`
		`cout << value << endl;`
			введення:**99_{10}**
			виведення:**0143_8**

uppercase	У шістнадцятковій системі числення використовувати літери верхнього регістру (за замовчуванням встановлено літери нижнього регістра)	`cout.unsetf(ios::dec);`
		`cout.setf(ios::hex | ios::uppercase);`
		`int value;`
		`cin >> value;`
		`cout << value << endl;`
			введення:**255_{10}**
			виведення:**FF_{16}**

showpos	Виведення знака плюс + для позитивних чисел	`cout.setf(ios::showpos);`
		`int value = 15;`
		`cout << value << endl;`
			+15

Прапор	Призначення	Приклад	Результат

scientific	Виведення чисел з плаваючою точкою в експоненційної формі	`cout.setf(ios::scientific);`
		`double value = 1024.165;`
		`cout << value << endl;`
			1.024165e+003

fixed	Виведення чисел з плаваючою точкою у фіксованій формі (за замовчуванням)	`double value = 1024.165;` `cout << value << endl;`	**1024.165**

right	Вирівнювання праворуч (за замовчуванням) та додає на початку символи заповнення. Спочатку необхідно встановити ширину поля (ширина поля повинна бути свідомо більшою ніж, довжина рядка, що виводиться).	`cout.setf(ios:: right);`
		`cout.width(40);`
		`cout << "cppstudio.com" << endl;`
			__cppstudio.com

left	Вирівнювання ліворуч та додає в кінець символи заповнення. Спочатку необхідно встановити ширину поля (ширина поля повинна бути свідомо більшою ніж, довжина рядка, що виводиться).	`cout.setf(ios::left);`
		`cout.width(40);`
		`cout << "cppstudio.com" << endl;`
			cppstudio.com__

6.2.4 Форматування введення-виведення за допомогою маніпуляторів

Ще один спосіб форматування - форматування за допомогою маніпуляторів. *Маніпулятор* - об'єкт особливого типу, який управляє потоками введення/виведення, для форматування переданої в потоки інформації. Почасти маніпулятори доповнюють функціонал, для форматування введення/виведення. Але більшість маніпуляторів виконують точно, те ж саме, що і функції з прапорами форматування. Є випадки, коли простіше користуватися прапорами або функціями форматування, а іноді зручніше використовувати маніпулятори форматування. Саме для цього в C++ передбачено кілька засобів форматування введення/виведення. У таблиці 6.2.3 показані основні маніпулятори форматування C++.

Таблиця 6.2.3 –Маніпулятори форматування в C++

Маніпулятор	Призначення	Приклад	Результат
endl	Перехід на новий рядок при виведенні	`cout << "website: " << endl << "cppstudio.com";`	**website: cppstudio.com**
boolalpha	Виведення логічних величин в текстовому вигляді (true, false)	`bool log_true = 1; cout << boolalpha << log_true << endl;`	**true**
noboolalpha	Виведення логічних величин в числовому вигляді (true, false)	`bool log_true = true; cout << noboolalpha << log_true << endl;`	**1**
oct	Виведення величин в восьмеричній системі числення	`int value = 64; cout << oct << value << endl;`	**100_8**
dec	Виведення величин в десятковій системі числення (за замовчуванням)	`int value = 64; cout << dec << value << endl;`	**64_{10}**
hex	Виведення величин в шістнадцятиричній системі числення	`int value = 64; cout << hex << value << endl;`	**40_8**
showbase	Виводити індикатор основи системи числення	`int value = 64; cout << showbase << hex << value << endl;`	**0x40**
noshowbase	Не виводити індикатор основи системи числення (за замовчуванням)	`int value = 64; cout << noshowbase << hex << value << endl;`	**40**
uppercase	У шістнадцятиричній системі числення використовувати літери верхнього регістру (за замовчуванням встановлено літери нижнього регістра)	`int value = 255; cout << uppercase << hex << value << endl;`	**FF_{16}**
nouppercase	У шістнадцятиричній системі числення використовувати літери нижнього регістра (за замовчуванням)	`int value = 255; cout << nouppercase << hex << value << endl;`	**ff_{16}**
showpos	Виведення знака плюс + для позитивних чисел	`int value = 255; cout << showpos<< value << endl;`	**+255**

Маніпулятор	Призначення	Приклад	Результат
noshowpos	Не виводити знак плюс + для позитивних чисел (за замовчуванням)	`int value = 255;` `cout << showpos << value << endl;`	**255**
scientific	Виведення чисел з плаваючою точкою в експоненційної формі	`double value = 1024.165;` `cout << scientific << value << endl;`	**1.024165e+003**
fixed	Виведення чисел з плаваючою точкою у фіксованій формі (за замовчуванням)	`double value = 1024.165;` `cout << fixed << value << endl;`	**1024.165**
setw(int number)	Встановити ширину поля, де number - кількість позицій, символів (вирівнювання за замовчуванням праворуч). Маніпулятор з параметром	`cout << setw(40) << "cppstudio.com" << endl;`	**__cppstudio.com**
right	Вирівнювання за правою межею (за замовчуванням). Спочатку необхідно встановити ширину поля (ширина поля повинна бути свідомо більшою ніж, довжина рядка, що виводиться)	`cout << setw(40) << right << "cppstudio.com" << endl;`	**__cppstudio.com**
left	Вирівнювання ліворуч. Спочатку необхідно встановити ширину поля (ширина поля повинна бути свідомо більшою ніж, довжина рядка, що виводиться)	`cout << setw(40) << left << "cppstudio.com" << endl;`	**cppstudio.com__**
setprecision(int count)	Задає кількість знаків після коми, де cout - кількість знаків після десяткової точки	`cout << fixed << setprecision(3) << (13.5 / 2) << endl;`	**6.750**
setfill(int symbol)	Встановити символ заповнювач. Якщо ширина поля більше, ніж величина яка виводиться, то вільні	`cout << setfill('0') << setw(4) << 15 << ends << endl;`	**0015**

Маніпулятор	Призначення	Приклад	Результат
	місця поля будуть заповнюватися символом symbol - символ заповнювач		

Форматоване введення/виведення в C++ - це одна з найпростіших тем в програмуванні. Як використовувати ті чи інші засоби форматування показано в таблицях, тому труднощів з даної теми виникнути не повинно.

6.2.5 Функція printf

Функція printf форматує та друкує ряд символів та значень у стандартний вихідний потік, stdout. Якщо аргументи слідують за стрічкою форматування, рядок формату повинен містити специфікатори, які визначають формат виводу аргументів.

```
printf("Line one\n\t\tLine two\n");
```

Результат роботи даної стрічки коду:

```
Line one
                Line two
```

В даному прикладі застосовані оператори переходу на новий рядок \n та два підряд оператори табуляції (відступу) \t.

Специфікатори формату завжди починаються зі знаку відсотка (%) і читаються зліва направо. Коли printf знаходить перший специфікатор формату (якщо такий є), вона перетворює значення першого аргументу після форматування та відповідно виводить його. Другий специфікатор формату зумовлює перетворення та виведення другого аргументу і так далі. Якщо є більше аргументів, ніж є специфікаторів формату, додаткові аргументи ігноруються. Результати не визначені, якщо аргументів для всіх специфікаторів формату недостатньо.

```
int intgr = -9234;
double fp = 28.061977;
// Відображення цілого числа у різних форматах
printf("Integer formats:\n"
      "\tDecimal: %d  Justified: %.6d  "
      "Unsigned: %u\n",
      intgr, intgr, intgr, intgr);
// Відображення десяткового числа у різних форматах
printf("Decimal formats %d as:\n\tHex: %Xh  "
```

```
      "C hex: 0x%x  Octal: %o\n",
      intgr, intgr, intgr, intgr);
// Відображення дійсного числа у різних форматах
printf("Real numbers:\n\t%f %.2f %e %E\n",
      fp, fp, fp, fp);
```

Результатом роботи даного прикладу буде наступне форматоване виведення числових значень:

```
Integer formats:
        Decimal: -9234  Justified: -009234  Unsigned: 4294958062
Decimal formats -9234 as:
        Hex: FFFFDBEEh  C hex: 0xffffdbee  Octal: 37777755756
Real numbers:
        28.061977 28.06 2.806198e+01 2.806198E+01
```

6.2.6 Використання кирилиці в консольних додатках

При введенні чи виведенні в консольному додатку *символів кирилиці* виникне помилка або вони будуть некоректно відображені на екрані. При програмуванні під Windows використовуються наступні *таблиці кодування*: cp866, cp1251 и utf-8 (стандарт Unicode). Хоча вже давно розроблений єдиний стандарт кодування символів - Unicode, в Windows досі використовуються кілька таблиць кодування, а саме - cp866 та cp1251. Використання декількох таблиць кодування символів і є причиною появи некоректних символів, замість повідомлення українською мовою.

Unicode - єдиний стандарт кодування символів, що дозволяє представити знаки всіх письмових мов. Таким чином *стандарт Unicode* привласнює кожному символу унікальний код, незалежно від мови. Зараз Unicode вважається кращим стандартом кодування символів.

Так уже повелося, що в командному рядку Windows кодування символів відповідає стандарту cp866. Тобто всі символи в командному рядку Windows закодовані по кодувальній таблиці cp866. Причому поміняти кодування в командному рядку Windows неможна.

У всіх україномовних ОС Windows кодування cp1251 є стандартним 8 - бітним кодуванням. І при створенні проекту в Microsoft Visual Studio цей стандарт кодування символів успадковується проектом, тобто програмою. Хоча кодування для проекту в Visual Studio можна легко поміняти, це не вирішує проблеми, оскільки консоль розуміє тільки одне кодування cp866, якого в MVS немає. У результаті, виходить, що програма передає коди символів повідомлення за стандартом cp1251. Командний рядок приймає ці коди і переводить їх у символи, але вже за стандартом cp866, оскільки іншого

стандарту не знає. У підсумку повідомлення було надіслане консолі, але символи інтерпретовані неправильно, ось так і з'являються некоректні символи.

Для того, щоб запити в консольному додатку відображалися українською мовою необхідно підключити до проекту файл заголовку `windows.h`. У файлі містяться прототипи функцій `SetConsoleCP()` і `SetConsoleOutputCP()`, вони нам і потрібні. Аргументом цих функцій є ідентифікатор кодової сторінки, потрібна нам сторінка `win-cp 1251`. Функція `SetConsoleCP()` встановлює потрібну кодову таблицю на потік введення, тоді як функція `SetConsoleOutputCP()` встановлює потрібну кодову таблицю на потік виводу.

У результаті до програми потрібно додати три додаткові рядки:

```
#include "windows.h"
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
```

Приклад програми з коректним використанням кирилиці буде наступним:

```
#include "stdafx.h"
#include <iostream>
#include "windows.h"  //Необхідно для використання функцій SetConsoleCP()
                      //та SetConsoleOutputCP()
using std::cout;
using std::cin;
using std::endl;
int main()
{
      SetConsoleCP(1251);   //Необхідно для введення в консолі
                            //української мови
      SetConsoleOutputCP(1251);  //Необхідно для виведення в консолі
                                 //української мови
      char ch = 'т'; //ініціалізація змінної ch початковим значенням 'т'
      while (ch != 'н') //умова виходу з циклу
      {
            cout << "Введіть два значення:\n";
            int val1 = 0, val2 = 0;
            cin >> val1 >> val2;  //значення вводяться через пробіл
                                  //чи натисканням клавіші Enter
            cout << "Сума " << val1 << " та " << val2 << " = "
                 << val1 + val2 << "\n"
                 << "Продовжити введення? \nВведіть 'т' чи 'н': ";
            cin >> ch;
      }
      return 0;
}
```

Приклад роботи програми наведений нижче.

```
Введіть два значення:
25
36
Сума 25 та 36 = 61
Продовжити введення?
Введіть 'т' чи 'н': т
Введіть два значення:
125
69
Сума 125 та 69 = 194
Продовжити введення?
Введіть 'т' чи 'н': н
Press any key to continue . . .
```

Даний приклад сумує будь-яку кількість пари чисел за допомогою циклу `while`. Умова циклу `while` перевіряє, який символ увів користувач у відповідь на запит "`Продовжити введення? Введіть 'т' чи 'н': `". Якщо користувач введе будь-який символ окрім ‘н’, цикл продовжить своє виконання, тобто програма запросить ввести наступні два числа для розрахунку їх суми. При введенні користувачем у відповідь на запит "`Продовжити введення? Введіть 'т' чи 'н': `" символу ‘н’, виконання циклу завершиться.

Тема 6.3. Вирази та оператори

6.3.1 Унарні та парні оператори

Існують *унарні оператори* (`unary operator`) та *парні оператори* (`binary operator`). Унарні оператори, такі як *оператор звернення до адреси &* та *оператор звернення до значення* *, діють на один операнд. Парні чи бінарні оператори, такі як рівність (==) та множення (*), діють на два операнди. Деякі символи, наприклад *, використовуються для позначення як унарних (звернення до значення), так і парних (множення) операторів. Тип оператору визначає контекст, в якому він використовується. У використанні цих операторів немає ніякого зв’язку, тому їх слід вважати двома різними символами.

6.3.2 Арифметичні оператори

У мові C/C++ підтримуються наступні арифметичні операції:

- + – додавання;
- – - віднімання;
- * - множення;
- / – ділення;
- % – остача від ділення.

Усі ці операції є *бінарними (парними)*. Це означає, що для отримання результату, потрібно 2 операнди. Загальний вигляд арифметичної операції:

```
операнд1 операція операнд2
```

де операція – одна з операцій +, –, *, %, /.

Операції додавання + та віднімання – можуть бути як бінарними, так і унарними. Бінарні операції + та – використовуються у виразах при проведенні обчислень. Унарні операції + та – використовуються для позначення знаку числа (додатне число або від‘ємне число).

```
int a, b;
a = -8; // унарна операція '-', позначає знак числа
b = +9; // унарна операція '+', b = 9
a = b - 5; // бінарна операція '-', використовується у виразі для обчислення
```

6.3.3 Особливості використання операції % та /

Остачею від ділення націло натурального числа m на натуральне число n є таке ціле число p < n, для якого справджується рівність

$$m = kn + p,$$

де k — певне натуральне число, яке називається часткою.

Наприклад, число 1 є остачею від ділення числа 7 на 2, оскільки

7 = 3 × 2 + 1.

Якщо число m < n, тоді остачею від ділення націло натурального числа m на натуральне число n буде саме число m, оскільки

$$m = 0 \cdot n + p.$$

Наприклад, число 2 є остачею від ділення числа 2 на 7, оскільки

2 = 0 × 7 + 2.

Якщо остача від ділення числа m на число n дорівнює нулю, то говорять, що число m ділиться на n без остачі, або, що число m кратне числу n.

```
// Операція % - взяття остачі від ділення
int a, b;
int c;
a = 3;
b = 5;
c = a % b; // c = 3
a = 8;
```

```
b = 4;
c = a % b; // c = 0
c = 12 % 35; // c = 12
c = 35 % 12; // c = 11
c = 16 % 42; // c = 16
c = 78 % 154; // c = 78
c = 7 % 4; // c = 3
c = -5 % -3; // c = -2
```

Операція ділення / має свої особливості, які полягають в наступному:

- якщо два операнди мають цілочисельний тип, то результат повертається цілого типу. У цьому випадку відбувається ділення націло. Остача від ділення відкидається;
- якщо один з операндів має тип з плаваючою комою, тоді результат має також тип з плаваючою комою.

```
// Операція ділення
int a, b;
int c;
float x;
a = 8;
b = 3;
c = a / b; // c = 2
x = a / b; // x = 2.0
x = a / (float)b; // x = 2.666667
x = 17.0 / 3; // x = 5.666667
x = 17 / 3; // x = 5.0
```

6.3.4 Пріоритет арифметичних операторів

В табл. 6.3.1 оператори згруповані за пріоритетом. Унарні арифметичні оператори мають більш високий пріоритет, ніж оператори множення та ділення, які в свою чергу мають більш високий пріоритет, ніж парні оператори віднімання та додавання.

Таблиця 6.3.1 – Арифметичні оператори

Оператор	Дія	Застосування
+	унарний плюс	+ вираз
-	унарний мінус	- вираз
*	множення	вираз * вираз
/	ділення	вираз / вираз
%	залишок	вираз % вираз
+	додавання	вираз + вираз
-	віднімання	вираз - вираз

6.3.5 Логічні оператори та оператори відношення

Загальним результатом *оператора логічного AND* (&&) буде true, якщо обидва його операнди розглядаються як true. *Оператор логічного OR* (||) повертає значення true, якщо будь-який з його операндів розглядається як true.

Оператор логічного NOT (!) повертає обернене значення свого операнда (!true повертає false).

Перелік логічних операторів та операторів відношення, які використовуються в C++, наведені в табл. 6.3.2.

Таблиця 6.3.2 – Логічні оператори та оператори відношення

Оператор	Дія	Застосування
!	логічне NOT	! вираз
<	менше	вираз < вираз
<=	менше чи дорівнює	вираз <= вираз
>	більше	вираз > вираз
>=	більше чи дорівнює	вираз >= вираз
==	рівність	вираз == вираз
!=	не дорівнює	вираз != вираз
&&	логічне AND	вираз && вираз
||	логічне OR	вираз || вираз

У якості *оператора присвоєння* (assignment operator) в C++ використовується оператор = (наприклад, a=5) на відміну від логічного *оператора рівності* (equality operator) == (перевірка рівності лівої та правої частини виразу в умові), який використовується в логічних виразах (наприклад, if (a==5) ++i;).

6.3.6 Складені оператори присвоєння

В C++ також використовують *складені оператори присвоєння* (compound assignment), які існують для кожного з арифметичних операторів (табл. 6.3.3). Ці оператори є зручними, коли в програмі використовуються довгі імена змінних. У цьому випадку відпадає необхідність зайвий раз вводити довге ім’я змінної.

Таблиця 6.3.3 – Складені оператори присвоєння

Оператор	Застосування	Приклад
*=	вираз *= вираз	a *= b рівноцінно a = a * b
/=	вираз /= вираз	a /= b рівноцінно a = a / b
%=	вираз %= вираз	a %= b рівноцінно a = a % b
+=	вираз += вираз	a += b рівноцінно a = a + b
-=	вираз -= вираз	a -= b рівноцінно a = a - b

6.3.7 Оператори інкременту та декременту

В C++ існують *оператор прирощення ++* (increment) та *оператор зменшення --* (decrement), які дозволяють в короткій та зручній формі додавати чи віднімати одиницю з об'єкту. Ця форма запису часто використовується при роботі з *ітераторами*, оскільки більшість ітераторів не підтримують арифметичні дії.

Ці оператори існують в двох формах: префіксній (prefix) та постфіксній (postfix). *Префіксний оператор прирощення* (*зменшення*) здійснює збільшення (зменшення) свого операнду на одиницю і повертає *змінений* об'єкт як результат. *Постфіксний оператор прирощення* (*зменшення*) повертає копію вихідного операнду *незмінною*, а потім здійснює збільшення (зменшення) свого операнду на одиницю.

```
int i = 0, j;
j = ++i;
std::cout << "j=" << j << ", i=" << i << std::endl;
// j=1, i=1 префікс повертає збільшене значення
int k = 0, l;
l = k++;
std::cout << "l=" << l << ", k=" << k << std::endl;
// l=0, k=1 постфікс повертає вихідне значення
```

6.3.8 Оголошення using

До сих пір імена зі стандартної бібліотеки згадувались у програмах явно, тобто перед кожним з них було вказано ім'я простору імен std. Наприклад, при читанні зі стандартного пристрою введення застосовувалась форма запису std::cin. Тут застосований оператор області видимості ::. При частому використанні бібліотечних імен така форма запису може бути надто громіздкою. Щоб уникнути цього застосовують оголошення using (using declaretion). *Оголошення using* дозволяє використовувати імена з іншого простору імен без вказування префіксу *ім'я_простору_імен*::. Оголошення using має наступний формат:

```
using простірімен::ім'я;
```

Після того як оголошення using було зроблено один раз, до вказаного в ньому імені можна звертатися без вказування простору імен.

```
#include <iostream>
using std::cout;
using std::endl;
int main()
```

```
{
	int i = 0, j;
	j = ++i;
	cout << "j=" << j << ", i=" << i << endl;
	return 0;
}
```

Для кожного імені необхідно індивідуальне оголошення `using`. Оголошення `using` *не можна застосовувати у файлах заголовку* для запобігання конфлікту імен.

6.3.9 Використання математичних функцій

Щоб використовувати у програмі математичні функції необхідно приєднати до програми заголовковий файл `math.h` за допомогою директиви препроцесора `#include`.

```
#include <math.h>
```

Таблиця 6.3.4 – Основні математичні функції

Функція	Пояснення
abs(x)	модуль цілого числа
fabs(x)	модуль дробового числа
sin(x)	синус
cos(x)	косинус
tan(x)	тангенс
asin(x)	арксинус
acos(x)	арккосинус
atan(x)	арктангенс
log(x)	натуральний логарифм
log10(x)	десятковий логарифм
exp(x)	піднесення e до ступеню x
pow(x,y)	піднесення x до ступеню y
pow10(x)	піднесення 10 до ступеню x
sqrt(x)	квадратний корінь
ceil(x)	округлення вгору
floor(x)	округлення вниз
fmod(x,y)	остача від ділення x на y

Тема 6.4. Змінні та базові типи

6.4.1 Арифметичні типи

У мові C++ визначений набір базових типів, включно з *арифметичними типами* (`arithmetic type`) та спеціальним типом `void`. *Арифметичні типи* представляють символи, цілі числа, логічні значення та числа з плаваючою комою. З типом `void` не пов'язано значень і застосовується він тільки за деяких обставин, частіше за все як тип значення, що повертає функція, котра не повертає нічого.

Існує два різновиди арифметичних типів: *цілочисельні типи* (включно з символьними і логічними) та *типи з плаваючою комою*. Розмір (тобто кількість бітів) арифметичних типів залежить від конкретного комп'ютера. Стандарт гарантує мінімальні розміри, наведені в табл. 6.4.1.

Таблиця 6.4.1 – Арифметичні типи мови C++

Тип даних	Значення	Мінімальний розмір
`bool`	Логічний тип	Не визначений
`char`	Символ	8 бітів
`wchar_t`	Широкий символ	16 бітів
`char16_t`	Символ Unicode	16 бітів
`char32_t`	Символ Unicode	32 біти
`short`	Коротке ціле число	16 бітів
`int`	Ціле число	16 бітів
`long`	Довге ціле число	32 бітів
`long long`	Довге ціле число	64 бітів
`float`	Число з плаваючою комою одинарної точності	6 значущих цифр
`double`	Число з плаваючою комою подвійної точності	10 значущих цифр
`long double`	Число з плаваючою комою підвищеної точності	10 значущих цифр

Тип `bool` приймає лише значення true (істина) та false (фальш).

Існує кілька *символьних типів*, більшість із яких призначена для підтримки національних наборів символів. *Типи з плаваючою комою* надають значення з одинарною, подвійною та розширеною точністю. Стандарт визначає мінімальну кількість значущих цифр.

За виключенням типу `bool` і розширених символьних типів *цілочисельні типи* можуть бути *знаковими* (`signed`) або *беззнаковими* (`unsigned`). *Знаковий*

тип може представляти від'ємні та додатні числа (включно з нулем), а *беззнаковий тип* – тільки додатні числа та нуль.

Типи `int`, `short`, `long` та `long long` є знаковими. Відповідні беззнакові типи отримують приставку `unsigned` до назви типу, наприклад `unsigned long`. Тип `unsigned int` може бути скорочений до `unsigned`.

Перелік беззнакових цілочисельних типів наведений у таблиці 6.4.2.

Таблиця 6.4.2 – Беззнакові типи в C++

Тип даних	Значення	Мінімальний розмір
unsigned int	4	Від 0 до 4 294 967 295
unsigned __int8	1	Від 0 до 255
unsigned __int16	2	Від 0 до 65 535
unsigned __int32	4	Від 0 до 4 294 967 295
unsigned __int64	8	Від 0 до 18 446 744 073 709 551 615
unsigned char	1	Від 0 до 255
unsigned short	2	Від 0 до 65 535
unsigned long	4	Від 0 до 4 294 967 295
unsigned long long	8	Від 0 до 18 446 744 073 709 551 615

6.4.2 Складені типи

6.4.2.1 Покажчик

Покажчик (`pointer`) – це складений тип, змінна якого вказує на об'єкт іншого типу. Подібно посиланням, покажчики використовують для опосередкованого доступу до інших об'єктів. Загальні правила щодо використання покажчиків:

- на відміну від посилання, покажчик – це справжній *об'єкт*;
- покажчики можуть бути присвоєні та скопійовані;
- один покажчик за час свого існування може вказувати на кілька різних об'єктів;
- на відміну від посилання, покажчик можна не ініціалізувати у момент визначення;
- типи покажчика та об'єкту, яким він ініціалізується мають співпадати;
- оскільки посилання не об'єкти, в них немає адрес, а відповідно, неможливо визначити покажчик на посилання.

Тип покажчика визначається оператором у формі *d, де d – ім'я покажчика, що створюється. Символ * слід повторювати для кожної змінної покажчика.

```
int *p1, *p2, c; //c не є покажчиком
```

Покажчик містить адресу іншого об'єкту. Для отримання адреси об'єкту використовують *оператор звернення до адреси* (address-of operator), або оператор &.

```
int i=77;
int *p=&i; //p містить адресу змінної i, p - покажчик на змінну i
```

У прикладі p визначене як покажчик на тип int та ініціалізоване адресою об'єкту i типу int.

Коли покажчик вказує на об'єкт, для доступу до цього об'єкту можна використати *оператор звернення до значення* (dereference operator), або *оператор* *.

```
int i=28;
int *p=&i; //p містить адресу змінної i, p - покажчик на змінну i
cout << *p; //* повертає об'єкт, на який вказує p, виводиться 28
```

Звернення до значення покажчика (*) повертає об'єкт, на який вказує покажчик.

Деякі символи в C++ можуть використовуватись в різному сенсі. Так символи & і * використовуються і як оператор у виразах, і як частина оголошення. Контекст, в якому використовується символ, визначає його значення.

```
int i = 40;
//& стоїть після типу в частині оголошення і оголошує посилання
int &посил = i;
//* стоїть після типу в частині оголошення і оголошує покажчик
int *покажч;
//& використовується у виразі як оператор звернення до адреси
покажч = &i;
//* використовується у виразі як оператор звернення до значення
*покажч = i;
//& є частиною оголошення типу, * - оператор звернення до значення
int &посил2 = *покажч;
//Виводимо значення об'єкту, на який вказує посилання
std::cout << посил << std::endl; // 40
//Виводимо значення об'єкту, на який вказує покажчик
std::cout << *покажч << std::endl; // 40
//Виводимо адресу об'єкту в пам'яті, на який вказує покажчик
std::cout << покажч << std::endl; // 00E5F934
```

В оголошеннях символи & і * використовуються для формування складений типів. У виразах ці ж символи використовуються для позначення оператора. Оскільки ті ж самі символи використовуються в різному контексті з різним сенсом, їх слід сприймати, як абсолютно різні символи.

Нульовий покажчик (null pointer) не вказує на жоден об'єкт. Код може перевіряти, чи не є покажчик нульовим, перед тим як його використовувати. Існує декілька способів отримати нульовий покажчик.

```
int *покажч1 = nullptr; // еквівалентно int *покажч1 = 0;
int *покажч2 = 0; // безпосередньо ініціалізує покажч2
                  //літеральною константою 0
int *покажч3 = NULL; // еквівалентно int *покажч3 = 0;
```

Краще за все ініціалізувати нульовий покажчик літералом `nullptr`. В якості альтернативи можна ініціалізувати покажчик літералом `0`. Деякі програмісти можуть використовувати *змінну препроцесора* (`preprocessor variable`) NULL, яку заголовок `cstdlib` визначає як `0`, але таке визначення нульового покажчика є небажаним.

І покажчика, і посилання надають опосередкований доступ до інших об'єктів. Однак є важливі відмінності у способі, яким вони це роблять. Найважливіше то, що посилання не є об'єктом. Після визначення посилання немає ніякого способу примусити його посилатись на інший об'єкт. При використанні посилання завжди використовується об'єкт, з яким воно було пов'язане спочатку.

Між покажчиком і адресою, яку він містить, немає такого зв'язку. Подібно будь-якій змінній, при присвоєнні покажчика для нього встановлюється нове значення. Присвоєння примушує покажчик вказувати на інший об'єкт.

6.4.2.2 Арифметичні дії з покажчиками

Арифметичні операції, які допустимі для покажчиків. Покажчики на масиви підтримують ті ж операції, що і арифметичні дії з ітераторами.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
const int кількість = 3;

int main() {
	int  масив[кількість] = { 28, 36, 97 };
	int  *покажчик;

	// Встановлюємо покажчик на перший елемент масиву
	покажчик = масив;
	// Знаходимо індекс останнього елементу масиву
	//за допомогою виразу sizeof(масив)/sizeof(*масив)-1
	for (int i = 0; i < sizeof(масив)/sizeof(*масив); i++) {
		cout << "Address of array[" << i << "] = ";
		// Друкуємо адресу елементу масиву на який вказує покажчик
```

```
            cout << покажчик << endl;
            cout << "Value of array[" << i << "] = ";
            // Друкуємо значення елементу масиву на який вказує покажчик
            cout << *покажчик << endl;
            // Збільшуємо покажчик на одиницю
            покажчик++;
      }
      cout << endl;
      // Встановлюємо покажчик на адресу останнього елементу масиву
      покажчик = &масив[кількість - 1];
      for (int i = кількість -1; i >= 0; i--) {
            cout << "Address of array[" << i << "] = ";
            // Друкуємо адресу елементу масиву на який вказує покажчик
            cout << покажчик << endl;
            cout << "Value of array[" << i << "] = ";
            // Друкуємо значення елементу масиву на який вказує покажчик
            cout << *покажчик << endl;
            // Зменшуємо покажчик на одиницю
            покажчик--;
      }
      return 0;
}
```

Результат роботи прикладу:

```
Address of array[0] = 00AFFE80
Value of array[0] = 28
Address of array[1] = 00AFFE84
Value of array[1] = 36
Address of array[2] = 00AFFE88
Value of array[2] = 97

Address of array[2] = 00AFFE88
Value of array[2] = 97
Address of array[1] = 00AFFE84
Value of array[1] = 36
Address of array[0] = 00AFFE80
Value of array[0] = 28
```

6.4.2.3 Посилання

Посилання (reference) є альтернативним іменем об’єкту. Тип посилання «посилається на» інший тип. У визначенні типу посилання використовується *оператор оголошення* у формі &d, де d – ім’я, що об’являється. Символ & слід повторювати для кожного імені покажчика.

```
int &a, &b, c; //c не є посиланням
int i;
```

```
int &j = i;
```

Зазвичай при ініціалізації змінної значення ініціалізатора копіюється в створюваний об'єкт. При визначенні посилання замість копіювання значення ініціалізатора відбувається *зв'язування* (bind) посилання з його ініціалізатором. Після ініціалізації посилання залишається зв'язаним із вихідним об'єктом. Неможливо змінити прив'язку посилання і зв'язати його з іншим об'єктом, тому *посилання обов'язково слід ініціалізувати*.

Посилання – це не об'єкт, а тільки *інше ім'я (псевдонім) вже існуючого об'єкту*. Загальні правила щодо використання посилань:

- після того, як посилання визначене, *всі* операції з ним фактично здійснюються з об'єктом, із яким пов'язане посилання;
- при присвоєнні посилання присвоюється об'єкт, із яким воно пов'язане;
- при доступі до значення посилання фактично відбувається звернення до значення об'єкту, з яким пов'язане посилання;
- коли посилання використовується як ініціалізатор, у дійсності для цього використовується об'єкт, із яким пов'язане посилання;
- оскільки посилання не об'єкти, то неможливо визначити посилання на посилання;
- типи посилання та об'єкту, на який воно посилається мають співпадати точно;
- посилання може бути пов'язане лише з об'єктом, але не з літералом чи результатом більш загального виразу.

```
int i;
float &k = i; //Помилка! Неспівпадіння типів посилання та об'єкту
int &j = 10; //Помилка! Посилання на літерал
```

6.4.3 Перетворення типів

Тип об'єкту визначає дані, які він може містити, та операції, які з ним можна виконувати. Серед операцій, які підтримуються множиною типів, є можливість *перетворювати* (convert) об'єкти даного типу в інший, пов'язаний тип.

Перетворення типів відбувається автоматично, коли об'єкт одного типу використовується там, де очікується об'єкт іншого типу. Коли значення одного арифметичного типу присвоюється іншому, результат залежить від діапазону значень, які підтримує тип.

- Коли значення одного з не логічних арифметичних типів присвоюється типу bool, результатом буде false, якщо значення є 0, а в іншому разі – true.

- Коли значення типу `bool` присвоюється одному з інших арифметичних типів, буде отримане значення `1`, якщо логічним значенням було `true`, та `0`, якщо значення було `false`.
- Якщо значення з плаваючою комою присвоюється об'єкту цілочисельного типу, воно урізується до частини перед десятковою крапкою.
- Коли цілочисельне (інтегральне) значення присвоюється об'єкту типу з плаваючою комою, дробова частина дорівнює нулю. Якщо у цілого числа більше бітів, ніж може вмістити об'єкт із плаваючою комою, то точність може бути *втрачена*.
- Якщо об'єкту беззнакового типу присвоюється значення не з його діапазону, результатом буде залишок від ділення за модулем значення, яке може містити тип призначення. Тобто присвоєне значення буде зовсім *іншим*.
- Якщо об'єкту знакового типу присвоюється значення не з його діапазону, результат буде *невизначеним*.

Нижче наведений приклад автоматичного перетворення між типами `char` та `int`:

```
//Перетворення між типами char та int
char c;
int d;
c = 'A';
d = c; // d = 65
d = 67;
c = d; // c = 'C'
```

Приклад автоматичного перетворення між типами `int` та `float`:

```
//Перетворення між типами int та float:
int d = 28;
float x;
x = d;        // x = 28.0 - тип float
d = 5.5 + 7; // d = 12 - тип int
```

Автоматичне перетворення між типами `float` і `double` наведене у прикладі нижче:

```
/Перетворення між типами float і double
float f;
double d;
int size;
f = 3.14f;
d = f; // d = 3.14 - результат типу double
d = 7.7f + 8.5; // d = 16.2 - результат типу double
```

Окрім автоматичного перетворення типів, в C++ існує *оператор явного приведення типів*. Загальний вигляд операції приведення типу:

```
(тип) вираз
```

де тип – тип, до якого потрібно привести результат обчислення виразу.

Використання операцій явного приведення типів наведене у наступному прикладі.

```
//Приклади використання операції приведення типів.
int a;
float x;
a = 5;
x = a / 2; // x = 2.0
x = (float)(a / 2); // x = 2.0
x = (float)a / 2; // x = 2.5 - типу float
x = a / 2.0; // x = 2.5 - типу float
x = (int)(8 / 3.0);   // x = 2
x = (float)(8 / 3.0); // x = 2.666667
```

6.4.4 L-значення та R-значення

L-значення (I-value) – це вираз, що повертає об'єкт або функцію. Неконстантне l-значення позначає об'єкт, який може бути лівим операндом оператора присвоєння. Під значенням l-value розуміється об'єкт, існуючий за межами одного виразу. Значення l-value можна уявити як об'єкт з ім'ям. Всі змінні, включаючи незмінні змінні (const), є значеннями l-value.

```
int x = 3 + 4; //x - це l-значення оскільки воно продовжує
               //існувати за межами виразу
```

R-значення (r-value) – це вираз, що повертає значення, але не асоційовану з ним область, якщо таке значення взагалі є. Може бути лише правим операндом оператора присвоєння. R-value - це тимчасове значення, яке не зберігається за межами виразу, в якому воно використовується.

```
int i, j, *p;
i = 7; // i є l-значенням

// Помилка: лівий операнд має бути l-значенням
7 = i; // Помилка!
j * 4 = 7; // Помилка!

*p = i; // Покажчик p є l-значенням

const int ci = 7;
```

```
// Помилка: змінна ci є незмінюваним l-значенням
ci = 9; // Помилка!

((i < 3) ? i : j) = 7; // Умовний оператор повертає l-значення
```

6.4.5 Змінні

Змінна (variable) –це іменоване сховище, яким можуть маніпулювати програми. У кожної змінної в мові C++ є тип. Тип визначає розмір і розташування змінної в пам'яті, діапазон значень, які можуть зберігатись у ній, та набір операцій, які можна застосувати до змінної. Програмісти C++ використовують терміни «змінна» та «об'єкт» як синоніми.

6.4.5.1 Визначення змінних

Просте визначення змінної складається зі *специфікатора типу* (type specifier), який супроводжується переліком із одного чи декількох імен змінних, що відділені комами, та завершується крапкою з комою.

```
unsigned char c;
int i, t=0, k;
bool b;
float f, f1;
```

Ініціалізація (initialization) присвоює об'єкту певне значення у момент створення. Значення, які використовуються для ініціалізації змінних, можуть бути як завгодно складними виразами.

```
//змінна price визначена і ініціалізована раніше, ніж вона використана для
//ініціалізації змінної discount
double price = 99.99, discount = price*0.15;
```

При визначенні змінної без ініціалізатора відбувається її *ініціалізація за замовчуванням* (default initialization). Таким змінним присвоюється *значення за замовчуванням* (default value). Це значення залежить від типу змінної і може також залежати від того, де визначається змінна.

Значення об'єкту вбудованого типу, яке не ініціалізовано явно, залежить від того, де саме воно визначається. Змінні, які визначені поза межами функції, ініціалізуються значенням 0. Значення *неініціалізованих* (uninitialized) об'єктів вбудованого типу, які визначені у тілі функції, невизначені.

Рекомендується ініціалізувати кожен об'єкт вбудованого типу. Це не завжди необхідно, але простіше, ніж з'ясовувати, чи можна в даному конкретному випадку обійтись без ініціалізатора.

Ідентифікатори (identifier) або імена в мові C++ можуть складатись із символів, цифр та символів підкреслення. Мова не накладає обмежень на довжину імен. Ідентифікатори мають починатися з літери чи символу підкреслення. Символи у верхньому та нижньому регістрі відрізняються, тобто ідентифікатори мови чутливі до регістру.

Існує певний набір ключових слів, зарезервованих в мові C++, наприклад auto, int, class, enum, struct, new та деякі інші. Дані слова не можуть використовуватись у якості ідентифікаторів. Також імена змінних не повинні починатись із *двох символів підкреслення* підряд, наприклад __indx не можна використовувати в якості ідентифікатора. Крім того ім'я змінної не може починатися *з символу підкреслення за яким іде прописна літера*, наприклад _Indx заборонено використовувати в якості імені об'єкту.

Існують певні загальноприйняті правила для іменування змінних. Дотримування цих правил покращує зручність читання коду.

- Ідентифікатор має бути змістовним.
- Імена змінних мають містити лише рядкові символи. Наприклад index, а не Index чи INDEX.
- Імена *класів* зазвичай починаються з прописної літери, наприклад Birthday_struct.
- Декілька слів в ідентифікаторі розділяють символами підкреслення або прописними першими літерами кожного слова окрім першого символу ідентифікатора, наприклад student_group або studentGroup, але не studentgroup.

6.4.5.2 Оголошення і визначення змінних

Для забезпечення можливості розділити програму на декілька логічних частин мова C++ надає технологію *роздільної компіляції* (separate compilation). Роздільна компіляція дозволяє складати програму з декількох файлів, кожен з яких може бути відкомпільований незалежно.

При поділі програми на кілька файлів потрібен спосіб спільного використання коду цих файлів. Наприклад, код, який визначений в одному файлі, можливо має використовувати змінну, що визначена в іншому файлі. Наприклад, класи об'єктів std::cout та std::cin визначені десь у стандартній бібліотеці, але наші програми можуть їх використовувати.

Для підтримки роздільної компіляції мова C++ розрізняє оголошення та визначення.

Оголошення (declaration) робить ім'я об'єкту відомим програмі. Файл, котрий має використовувати ім'я, що визначене в іншому місці, включає оголошення для цього імені.

Визначення (definition) створює відповідну сутність. Оголошення змінної визначає її тип та ім'я. Визначення змінної – це її оголошення. Окрім вказування імені та типу змінної, визначення також резервує місце для її збереження і може надати змінній вихідне значення.

Для отримання оголошення, що не є визначенням, додається ключове слово extern (зовнішня) і можна не надавати явний ініціалізатор.

```
extern int i;    //оголошення без визначення змінної i
int j;           //оголошення та визначення змінної j
```

Будь-яке оголошення, яке включає явний ініціалізатор, є визначенням. Оголошені змінні можуть бути багато разів, проте визначені лише раз. Використання змінної в декількох файлах потребує оголошень, окремо від визначення. Для використання змінної в кількох файлах, її слід визначити в одному, і лише в одному файлі. В інших файлах, де використовується та ж сама змінна, її потрібно оголосити, проте не визначати.

6.4.5.3 Ідентифікатори

Ідентифікатори (identifier) або імена в мові C++ можуть складатись з символів, цифр та символів підкреслення. Мова не накладає обмеження на довжину імені. Ідентифікатори мають починатись з літер або символу підкреслення. Символи у верхньому та нижньому регістрі відрізняються, тобто ідентифікатори чуттєві до регістру.

В якості ідентифікатора не можна використовувати ключові слова мови C++. Окрім того ідентифікатори не можуть містити два послідовних символи підкреслення, а також починатись з символу підкреслення за яким іде прописна літера.

Існують деякі загальноприйняті домовленості для найменування змінних. Застосування подібних домовленостей полегшує розуміння та сприйняття коду.

- Ідентифікатор повинен мати сенс.
- Імена змінних зазвичай складаються з малих літер. Наприклад, index, а не Index чи INDEX.
- Імена класів зазвичай починаються з великої літери, наприклад My_class.
- Кілька слів у ідентифікаторі розділяють символом підкреслення або першою великою літерою в кожному новому слові. Наприклад, student_group чи studentGroup, але не studentgroup.

6.4.5.4 Область видимості імен

В любому місці програми кожне ім'я відноситься до певної сутності – змінної, функції, типу і т. ін. Проте ім'я може бути використане багаторазово для звернення до різних сутностей в різних місцях програми.

Область видимості (scoup) – це частина програми, у якій в імені є конкретне значення. Як правило, області видимості в мові C++ розмежовуються фігурними дужками. Імена видимі від моменту їх оголошення і до кінця області видимості, в якій вони оголошені.

Розглянемо наступний приклад програмного коду:

```
#include <iostream>
int main()
{
	int кількість;
	кількість = 7;
	float сума = 0;
	for (size_t i = 0; i <= кількість; i++)
	{
		сума += i;
	}
	std::cout << сума << std::endl;
	return 0;
}
```

В цій програмі визначаються три імені – `main`, `кількість`, `сума`, `i`, а також використовується ім'я простору імен `std`, разом з двома іменами з цього простору імен – `cout` та `endl`.

Ім'я `main` визначене поза фігурними дужками. Воно, як і більшість імен, які визначені поза функціями, має *глобальну область видимості* (global scope). Після оголошення, імена в глобальній області видимості доступні в усій програмі.

Імена змінних `кількість` та `сума` оголошені в межах блоку, яким є тіло функції `main()`. Ці імена доступні від моменту оголошення і до кінця функції `main()`, але за межами функції вони невідомі. Змінні `кількість` та `сума` мають *область видимості блоку* (block scope).

Змінна `i` визначена в межах оператора `for`. Її ім'я відоме лише в середині оператора `for` і за межами цього оператора у функції `main()` воно невідоме.

Об'єкти потрібно визначати ближче до місця їх першого використання. Це полегшує розуміння коду програми і спрощує присвоєння належного вихідного значення.

Області видимості можуть включати інші області видимості. Вкладена область видимості зветься *внутрішньою областю видимості* (inner scope), а

область видимості, які її включає в себе, називається *зовнішньою областю видимості* (outer scope).

Після оголошення імені в області видимості, воно стає доступним у вкладених в неї областях видимості. При цьому імена, які оголошені у зовнішній області видимості, можуть бути перевизначені у внутрішній області видимості:

```
#include <iostream>
#include <windows.h> //Необхідно для виведення в консолі української мови
int глобальна = 28; //Змінна глобальна має глобальну область видимості
int main()
{
	//Необхідно для виведення в консолі української мови
	SetConsoleOutputCP(1251);
	int блочна = 17;
	std::cout << "Результат №1:\tЗначення глобальної змінної = "
		<< глобальна << std::endl;
	std::cout << "\t\tЗначення локальної змінної = " << блочна
		<< std::endl;
	//Нова локальна змінна з ім'ям глобальна приховує глобальну змінну
	int глобальна = блочна;
	std::cout << "Результат №2:\tЗначення перевизначеної глобальної
		змінної = " << глобальна << std::endl;
	std::cout << "\t\tЗначення локальної змінної = " << блочна
		<< std::endl;
	if (глобальна == блочна)
	{
		//Нова локальна змінна з ім'ям блочна приховує локальну змінну
		int блочна = ::глобальна;
		std::cout << "Результат №3:\tЗначення перевизначеної
			глобальної змінної = " << глобальна << std::endl;
		std::cout << "\t\tЗначення перевизначеної локальної змінної
			= " << блочна << std::endl;
	}
	std::cout << "Результат №4:\tЯвне звернення до глобальної змінної
		= " << ::глобальна << std::endl;
	std::cout << "\t\tЗначення локальної змінної = " << блочна
		<< std::endl;
	return 0;
}
```

Результатом роботи даної програми буде наступне консольне виведення:

```
Результат №1:	Значення глобальної змінної = 28
		Значення локальної змінної = 17
Результат №2:	Значення перевизначеної глобальної змінної = 17
		Значення локальної змінної = 17
Результат №3:	Значення перевизначеної глобальної змінної = 17
		Значення перевизначеної локальної змінної = 28
```

```
Результат №4:   Явне звернення до глобальної змінної = 28
                Значення локальної змінної = 17
```

Результат №1 виводить значення глобальної змінної з ім'ям `глобальна` та локальної змінної з ім'ям `блочна`, яка визначена в області видимості блоку:

- `глобальна = 28,`
- `блочна = 17.`

Результат №2 виводить значення перевизначеної в межах функції `main()` змінної з ім'ям `глобальна`, яка приховує глобальну змінну, та локальної змінної з ім'ям `блочна`, яка визначена в області видимості блоку:

- `глобальна = 17,`
- `блочна = 17.`

Результат №3 виводить значення перевизначеної в межах функції `main()` змінної з ім'ям `глобальна`, яка приховує глобальну змінну, та перевизначене в межах оператора `if` значення змінної з ім'ям `блочна`, яка визначена у внутрішній області видимості:

- `глобальна = 17,`
- `блочна = 28.`

Результат №4 виводить глобальне значення змінної `глобальна`, шляхом явного звернення до її значення за допомогою оператора області видимості `::` - `::глобальна`. Також тут виводиться значення змінної `блочна`, що оголошена у зовнішній області видимості по відношенню до оператора `if`:

- `глобальна = 28,`
- `блочна = 17.`

В даному прикладі для форматування виведених результатів використовується символ табуляції "`\t`". Для можливості відображення в консолі символів кирилиці підключена бібліотека `windows.h`, в якій оголошена функція `SetConsoleOutputCP()`. Ця функція дозволяє змінювати кодову таблицю символів при виведенні на консоль.

Тема 6.5. Класи в C++

6.5.1 Базові поняття ООП

Класи в мові C++ використовуються для визначення власних типів даних. Фундаментальними поняттями концепції класів є абстракція даних та інкапсуляція.

Абстракція даних (data abstraction) – програмний підхід, що заснований на розділенні інтерфейсу та реалізації. *Інтерфейс* (interface) класу складається з операцій, які користувач класу може виконати з його об'єктом. *Реалізація* (implementation) включає змінні-члени класу, тіла функцій, що складають інтерфейс, а також інші функції, котрі потрібні для визначення класу, проте не призначені для загального використання.

Інкапсуляція (encapsulation) забезпечує розділення інтерфейсу та реалізації класу. Інкапсульований клас приховує свою реалізацію від користувачів, які можуть використовувати інтерфейс, проте не мають доступу до реалізації класу.

Під *користувачами* класу в C++ маються на увазі розробники програмного коду, які використовують вже створений раніше іншими розробниками клас.

Клас, який використовує абстракцію даних та інкапсуляцію, називають *абстрактним типом даних* (abstract data type). Програмісти, які працюють із класом, не мають знати як внутрішньо працює цей тип. Вони можуть розглядати його як абстракцію.

Для забезпечення інкапсуляції в C++ використовують *специфікатори доступу* (access specifier).

- Члени класу, які визначені після *специфікатора public*, доступні для всіх частин програми. *Відкриті члени* (public member) визначають *інтерфейс класу*.
- Члени, які визначені після *специфікатору private*, є *закритими членами* (private member), вони доступні для функцій-членів класу, але не доступні для коду, який цей клас використовує. Розділи private інкапсулюють (приховують) реалізацію.

6.5.2 Ключове слово struct

На найпростішому рівні *структура даних* (data structure) – це спосіб групування взаємопов'язаних даних та стратегії їх використання.

Визначення класу починається із *ключового слова struct*, яке супроводжується ім'ям класу та (можливо пустим) тілом класу. *Тіло класу* обмежується фігурними дужками і формує нову *область видимості*. Визначені

у класі імена мають бути унікальними в межах класу, але поза класом вони можуть повторюватись.

```
struct MyStruct //Створення власної структури даних (класу)
{
	std::string name;
	unsigned index;
	float sum;
};
```

За фігурною дужкою, що закриває тіло класу, моє бути крапка з комою. Крапка з комою необхідна, оскільки після тіла класу можливо визначити змінні.

У тілі класу визначені члени (member) класу. В нашому прикладі у класу є лише *змінні-члени* (`data member`). Змінні-члени класу визначають вміст об'єктів цього класу. Кожен об’єкт класу має власний екземпляр змінних-членів класу.

У змінних-членів класу можна визначити *внутрішньокласовий ініціалізатор* (`in-class initializer`). Він використовується для ініціалізації змінних-членів при створенні об’єктів. Члени без ініціалізаторів ініціалізуються за замовчуванням. Внутрішньокласові ініціалізатори мають бути укладені у фігурні дужки або йти за знаком =. Неможна визначити внутрішньокласовий ініціалізатор у круглих дужках.

```
struct MyStruct //Створення власної структури даних (класу)
{
	std::string name; // ініціалізатор за замовчуванням
	unsigned index = 1; // внутрішньокласовий ініціалізатор
	float sum = 1.0; // внутрішньокласовий ініціалізатор
	vector<int> vect{ 1,2,3,4,5 };  //внутрішньокласовий ініціалізатор
                                    // із використанням фігурних дужок
};
```

6.5.3 Ключове слово class

Мова C++ використовує для визначення власних структур окрім ключового слова `struct` також ключове слово `class`.

Для того, щоб скористатися створеною структурою даних, необхідно визначити об’єкт типу створеної структури. *Визначення об’єкту* типу класу в найпростішому випадку може бути аналогічне визначенню об’єктів інших типів.

```
string word; //визначення змінної вбудованого типу string
MyStruct my_str; //визначення змінної власного типу MyStruct
```

Після визначення змінної типу класу стає можливим звернення до членів даного класу. Зверення до членів класу використовується для отримання значень змінних-членів, зміни (присвоєння нового значення) значень змінних-членів, виконання певних операцій, які передбачені даним класом. Звернення до членів класу виконується за допомогою *оператору звернення до члену* (`.`).

```
int i = my_str.index; //значення i=1
my_str.sum = 5.5;
vector<int> vec1;
vec1 = my_str.vect; //vec1={ 1,2,3,4,5 }
```

Визначення класу у загальному вигляді з використанням ключового слова `class` виглядає наступним чином:

```
class MyClass //Ім'я класу
{
public:
     //Інтерфейс класу
private:
     //Реалізація класу
};
```

При визначені класу можна використовувати ключові слова `struct` або `class`. Відмінність у визначенні класу за допомогою цих ключових слів полягає у заданому за замовчуванням *рівні доступу*. Якщо використовується *ключове слово `struct`*, то члени, які визначені до першого специфікатора доступу, будуть *відкритими* (`public`), тобто входитимуть до інтерфейсу класу. Якщо використовується *ключове слово `class`*, то члени, які визначені до першого специфікатора доступу, будуть *закритими* (`private`), тобто входитимуть до реалізації класу.

Інкапсуляція надає дві важливі переваги.

- Код користувача не може за необачністю пошкодити інкапсульований об'єкт.
- Реалізація інкапсульованого класу може з часом змінитися і це не вимагатиме змін у коді на рівні користувача.

6.5.4 Функції-члени та дружні функції

Змінні, які входять у визначення класу, називають *змінними-членами* (data member).

Члени класу, які є функціями, називають *функціями-членами* (member function).

Функції-члени визначають та об'являють як звичайні функції. Функції-члени *мають бути* об'явлені в класі, проте визначені вони *можуть* бути безпосередньо в класі або поза тілом класу. Функції, які не є членами класу, але є складовими інтерфейсу, об'являються та визначаються поза класом, але в тому ж самому файлі що і клас.

```
class MyClass
{
public: //Інтерфейс класу
      void Method(); //Об'явлення функції-члену класу без параметрів
private: //Реалізація класу
      float k1 = 1; //Ініціалізація даних-членів класу
      float k2 = 1;
      float k3 = 1;
      float a=-10;
      float b=10;
      double MyFunc(const double&); //Об'явлення функції-члену класу
                                    //з параметром - константним посиланням
      MyClass& MyFunc2(const MyClass&) //Об'явлення функції-члену класу
                  //з параметром - константним посиланням на тип класу
}; //Завершення тіла класу

void MyClass::Method() //Визначення функції-члену класу без параметрів
{                              //поза тілом класу
      double za, zb;
      za = MyFunc(a);
      zb = MyFunc(b);
}

double MyClass::MyFunc(const double &x) //Визначення функції-члену класу
{                  //з параметром – константним посиланням поза тілом класу
      double z;
      z = k1*exp(k2*x) + k3*x;
      return z;
}
```

Ім'я функції-члена, яка об'явлена в тілі класу, але визначена поза тілом класу, повинне включати ім'я класу, якому вона належить:

```
void MyClass::Method() //Визначення функції-члену поза тілом класу
```

Ім'я функції `MyClass::Method()` використовує оператор області видимості, щоб вказати, що дана функція об'явлена в межах класу `MyClass`.

Кожен клас визначає власну область видимості. Поза *областю видимості класу* (class scope) до звичайних даних і функцій його члени можуть звертатись лише через об'єкт, посилання або покажчик, використовуючи

оператор доступу до члену (.). Для доступу до членів типу з класу використовується *оператор області видимості* (::). У будь-якому випадку наступне за оператором ім'я має бути членом відповідного класу.

У функції-члені можна безпосередньо звернутися до членів об'єкту, з якого вона була викликана. Для цього використовується покажчик `this`. *Параметр* `this` визначається неявно та автоматично і його можна використовувати в тілі функції-члену.

```
MyClass& MyClass::MyFunc2(const MyClass &rhs) //Визначення функції-члену
{          //класу з параметром – константним посиланням поза тілом класу
      a += rhs.a;
      return *this; //повертає об'єкт, для якого була викликана функція
}
```

Автори класів інколи визначають *допоміжні функції*, які використовуються як частина інтерфейсу класу, але при цьому членами класу вони не є. Такі функції об'являються (але не визначаються) в тому ж заголовку, що і сам клас після визначення класу. Таким чином, щоб використовувати будь-яку частину інтерфейсу класу, користувачу достатньо підключити лише один файл заголовку.

6.5.5 Вбудована функція

Вбудована функція (inline function) - функція, тіло якої вбудовується за місцем звернення, якщо це можливо. Вбудовувані функції дозволяють уникнути звичайних додаткових витрат, оскільки їх виклик замінює код тіла функції.

```
// Загальний вигляд вбудованої функції
inline тип_поверненого_значення ім'я_функції(аргументи)
{
      //Код функції
      return результат;
}

// Приклад використання вбудованих функцій
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
//Необхідно для введення/виведення української мови в консолі
#include <locale>;
#include "windows.h";

class МійКлас {// Оголошення нового класу
public:
      // Оголошення вбудованої функції
      inline float множення(float x, float y) {
```

```
            return (x * y);
        }
        // Оголошення вбудованої функції
        inline float куб(float x) {
            return (x * x * x);
        }
};

int main() {
        //Встановлення таблиці кодування для введення укр. мови
        SetConsoleCP(1251);
        //Встановлення таблиці кодування для виведення укр. мови
        SetConsoleOutputCP(1251);
        МійКлас обкт;// Створюємо об'єкт класу МійКлас
        float числ1, числ2;
        cout << "Введіть два значення:";
        cin >> числ1 >> числ2;
        cout << "\nРезультат множення чисел " << числ1 << " та " << числ2
            << " = " << обкт.множення(числ1, числ2);
        cout << "\nЧисло " << числ1 << " у кубі = " << обкт.куб(числ1);
        cout << "\nЧисло " << числ2 << " у кубі = " << обкт.куб(числ2);
        _getch();
}
```

Результат роботи прикладу:

```
Введіть два значення:14 28

Результат множення чисел 14 та 28 = 392
Число 14 у кубі = 2744
Число 28 у кубі = 21952
```

6.5.6 Конструктор класу

Кожен клас визначає, як можуть бути ініціалізовані об’єкти його типу. Клас контролює ініціалізацію об’єкту за рахунок визначення однієї чи декількох спеціальних функцій-членів, відомих як *конструктори* (constructor). Задача конструктора – ініціалізувати змінні-члени об’єкту класу. Конструктор виконується кожен раз, коли створюється об’єкт класу.

Ім’я конструктора *співпадає* з іменем класу. На відміну від інших функцій, у конструкторів *немає типу* повернутого значення. Як і інші функції, конструктори мають список параметрів (можливо пустий) і тіло (можливо пусте). У класу може бути *декілька* конструкторів. Подібно будь-якій іншій перевантаженій функції, конструктори мають *відрізнятися* один від одного *кількістю* або *типами* своїх параметрів. Конструктори, на відміну від інших функцій, не можуть бути об’явлені константами.

Якщо в класі не визначено жодного конструктору, класи самі контролюють ініціалізацію відкритих змінних-членів значеннями за замовчуванням, визначаючи спеціальний конструктор, що зветься *стандартним конструктором* (default constructor). Стандартним вважається конструктор, який не отримує ніяких аргументів. У такому випадку змінні-члени ініціалюзуються внутрішньокласовими ініціалізаторами чи значеннями за замовчуванням.

Якщо клас не визначає конструктори явно, компілятор сам визначить стандартний конструктор неявно. Створений компілятором конструктор зветься *синтезованим стандартним конструктором* (synthesized default constructor).

Класи, члени яких мають *вбудований* чи *складений тип* (відповідно не мають ініціалізатора за замовчуванням), можуть розраховувати на синтезований стандартний конструктор, *тільки якщо у всіх* таких членів є внутрішньокласові ініціалізатори.

```
struct MyStruct
{
      int *pi = 0;     //Внутрішньокласовий ініціалізатор змінної
                       //складеного типу
      float y = 1;     //Внутрішньокласовий ініціалізатор змінної
                       //вбудованого типу
      double a;        //Визначення змінної-члену вбудованого типу
                       //без ініціалізатора
}; //Завершення тіла класу
```

Якщо у класу визначений хоча б один конструктор, то компілятор не буде створювати автоматично стандартний конструктор. У такому разі стандартний конструктор потрібно обов’язково визначити в класі самостійно.

Якщо в класі є інші конструктори, то можна попросити компілятор створити *стандартний конструктор* автоматично. Для цього після списку параметрів вказується частина = default.

```
MyClass() = default;
```

Оскільки цей конструктор не має параметрів – він є стандартним конструктором. Застосування цього конструктора можливе лише у випадку, коли для змінних-членів вбудованих та складених типів є *внутрішньокласові ініціалізатори*.

При створенні конструктору класу після переліку параметрів ставиться двокрапка, за якою розміщується *перелік ініціалізації конструктора* (constructor initializer list), який визначає вихідні значення для однієї чи декількох змінних-членів створюваного об’єкту. Завершується конструктор

фігурними дужками, які містять тіло конструктора. *Ініціалізатор конструктору* – це перелік імен змінних-членів класу, кожне з яких супроводжується вихідним значенням у круглих (або фігурних) дужках. Якщо ініціалізацій кілька, вони відділяються комами.

```
MyClass() = default; //Стандартний конструктор
MyClass(const float &a1, const float &b1) : a(a1), b(b1) {};
//Конструктор класу
MyClass(const float &kof1, const float &kof2, const int &n, const float
&a1, const float &b1) : k1(kof1), k2(kof2), k3(kof2*n), a(a1), b(b1) {};
//Конструктор класу
```

У даному прикладі першим іде стандартний конструктор, який для ініціалізації змінних членів при створенні об’єкту класу використовує внутрішньокласові ініціалізатори (якщо вони є), в іншому разі використовує ініціалізацію змінних-членів значеннями за замовчуванням (окрім вбудованих та складених типів).

Другим іде конструктор класу, який ініціалізує змінні-члени a та b параметрами a1 та b1 відповідно. Змінні члени k1, k2 та k3 будуть ініціалізовані внутрішньокласовими ініціалізаторами.

Третій конструктор ініціалізує всі змінні-члени створюваного об’єкту класу переліком ініціалізації конструктора, який розміщений між двокрапкою та фігурними дужками. При цьому змінна-член k3 ініціалізується добутком параметрів kof2 та n.

Зазвичай для конструктора краще використовувати внутрішньокласовий ініціалізатор, якщо він є і присвоює члену класу вірні значення. Якщо компілятор не підтримує внутрішньокласову ініціалізацію (введена в стандарті C++11),кожен конструктор повинен явно ініціалізувати кожен член вбудованого типу.

У наведених вище конструкторів тіла пусті. Єдине їх призначення – присвоїти значення змінним-членам. Якщо нічого іншого робити не потрібно, то тіло функції залишається пустим.

Внутрішньокласовой ініціалізатор (in-class initializer) - це ініціалізатор, наданий як частина оголошення змінної-члена класу. Він використовується для ініціалізації змінних-членів при створенні об’єктів. Члени без ініціалізатора ініціалізуються за замовчуванням. За внутрішньокласовим ініціалізатором стоїть символ =, або він вкладається в фігурні дужки. Неможливо визначити внутрішньокласовий ініціалізатор у круглих дужках.

У наступному прикладі проілюстроване використання внутрішньокласового ініціалізатора.

```
#include "stdafx.h"
#include <string>
```

```
#include <iostream>
#include <locale>;
#include "Windows.h"

class МійКлас
{
public:
	std::string речення{"Внутрішньокласовий ініціалізатор"};
	//Помилка!!! Не можна ініціалізувати змінну у круглих дужках
	//std::string речення("Внутрішньокласовий ініціалізатор");
	int ціле = 28;
	double дійсне{23.58};

};

int main()
{
	//Встановлення таблиці кодування для укр. мови
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	МійКлас обкт;
	std::string речення("Пряма ініціалізація у функції");
	std::cout << обкт.речення << std::endl;
	std::cout << обкт.ціле << std::endl;
	std::cout << обкт.дійсне << std::endl;
	std::cout << речення << std::endl;
	return 0;
}
```

Результат роботи прикладу:

```
Внутрішньокласовий ініціалізатор
28
23.58
Пряма ініціалізація у функції
```

Як і інші функції-члени, конструктори можуть визначатися як у тілі класу, так і за його межами. У тілі класу визначають функції, які складаються не більше як із одного-двох операторів, щоб не ускладнювати розуміння класу. Функції-члени, в тому числі конструктори, які мають більший обсяг коду у своєму тілі, об’являються в тілі класу, а визначаються одразу після нього в тому ж самому файлі заголовку. Перед іменем конструктора, який визначений за межами класу, як і перед іменем інших функцій-членів, вказується ім’я класу та оператор області видимості.

```
MyClass::MyClass()
```

```
{
	//Тіло конструктору
}
```

Ім’я класу перед іменем функції вказується для того, щоб зазначити що дана функція відноситься до вказаного класу. Оскільки ім’я функції співпадає з іменем класу, значить ця функція є конструктором зазначеного класу.

Клас може дозволити іншому класу чи функції отримати доступ до своїх закритих членів, встановивши для них *дружні відносини* (`friend`). Клас об’являє функцію дружньою, включивши її об’явлення з ключовим словом `friend`.

```
class MyClass
{
friend float add(const float&);
friend std::ostream &print(std::ostream&, const MyClass&);
public: //Інтерфейс класу
	MyClass() = default; //Стандартний конструктор
	MyClass(const float &a1, const float &b1) : a(a1), b(b1) {};
	//Конструктор класу
	MyClass(const float &kof1, const float &kof2, const int &n, const
float &a1, const float &b1) : k1(kof1), k2(kof2), k3(kof2*n), a(a1),
b(b1) {}; //Конструктор класу
	void Method(); //Об'явлення функції-члену класу без параметрів
private: //Реалізація класу
	float k1 = 1; //Ініціалізація даних-членів класу
	float k2 = 1;
	float k3 = 1;
	float a=-10;
	float b=10;
	double MyFunc(const double&); //Об'явлення функції-члену класу
	                              //з параметром - константним посиланням
	MyClass& MyFunc2(const MyClass&) //Об'явлення функції-члену класу
	                 //з параметром - константним посиланням на тип класу
}; //Завершення тіла класу
//Об’явлення частин, що не є складовими інтерфейсу класу
float add(const float&);
std::ostream &print(std::ostream&, const MyClass&);
```

Об’явлення друзів може розташовуватись лише *у визначенні* класу, використовуватись у класі вони можуть будь-де. Друзі не є членами класу і не підкоряються специфікатору доступу розділу, в якому вони об’явлені. Об’явлення друзів бажано групувати на початку чи в кінці визначення класу.

Об’явлення дружніх відносин встановлює лише права доступу. Це не об’явлення функції. *Для можливості виклику* дружньої функції користувачами класу її слід також *об’явити*. Для доступу користувачів до дружніх функцій їх зазвичай об’являють поза класом, у тому ж заголовку, що і сам клас.

6.5.7 Внутрішньокласовой ініціалізатор

Внутрішньокласовой ініціалізатор (in-class initializer) - ініціалізатор, наданий як частина оголошення змінної-члена класу. Він використовується для ініціалізації змінних-членів при створенні об'єктів. Члени без ініціалізатора ініціалізуються за замовчуванням. За внутрішньокласовим ініціалізатором стоїть символ =, або він вкладається в фігурні дужки. Неможливо визначити внутрішньокласовий ініціалізатор у круглих дужках.

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include <locale>;
#include "Windows.h"

class МійКлас
{
public:
	std::string речення{"Внутрішньокласовий ініціалізатор"};
	//Помилка!!! Не можна ініціалізувати змінну у круглих дужках
	//std::string речення("Внутрішньокласовий ініціалізатор");
	int ціле = 28;
	double дійсне{23.58};

};

int main()
{
	//Встановлення таблиці кодування для укр. мови
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	МійКлас обкт;
	std::string речення("Пряма ініціалізація у функції");
	std::cout << обкт.речення << std::endl;
	std::cout << обкт.ціле << std::endl;
	std::cout << обкт.дійсне << std::endl;
	std::cout << речення << std::endl;
	return 0;
}
```

Результат роботи прикладу:

```
Внутрішньокласовий ініціалізатор
28
23.58
Пряма ініціалізація у функції
```

6.5.8 Оператор = default

Оператор = `default` – це синтаксис, який використовується після списку параметрів оголошення стандартного конструктора класу, щоб повідомити компілятору про необхідність створити конструктор, навіть якщо у класу є інші конструктори. Приклад використання оператора `default` наведений в програмному коді нижче.

```
class МійКлас
{
public:
      МійКлас() {} //Конструктор без параметрів
      МійКлас(int x) {/* ... */ }
};

class МійКлас2
{
public:
      МійКлас2() = default;  //Конструктор без параметрів
      МійКлас2(int x) {/* ... */ }
};
```

6.5.9 Визначення класу у файлі заголовку

Класи зазвичай визначають у *файлах заголовку* (`header`). Файли заголовку мають розширення `*.h`. Як правило, класи зберігаються в заголовках, ім'я яких співпадає з іменем класу. Наприклад, бібліотечний тип `string` визначений у заголовку `string`. За аналогією наш клас `MyStruct` має бути визначений у заголовку `MyStruct.h`.

Заголовки підключаються до програмного файлу за допомогою директиви препроцесора `#include`. Коли препроцесор зустрічає директиву `#include`, він заміняє її вмістом файлу заголовку, який підключений за допомогою цієї директиви. З огляду на це, заголовки також не повинні містити об'явлення `using`, щоб запобігти включенню в програму не передбачених бібліотечних імен.

Заголовки містять сутності, які можуть бути визначені у будь-якому файлі лише раз. Для запобігання кількаразового включення вмісту одного й того самого заголовку, програми C++ використовують препроцесор для *захисту заголовку* (`header guard`). Захист заголовку опирається на змінні препроцесору. Змінні препроцесору можуть знаходитись у двох станах: визначена чи не визначена. Директива `#define` отримує ім'я і визначає його як змінну препроцесору. Існують дві директиви, що дозволяють перевірити чи була визначена змінна препроцесора. Директива `#ifdef` істинна, якщо змінна була

визначена, а директива #ifndef істинна, якщо змінна *не була визначена*. При істинності перевірки виконується все, що розташоване після директиви #ifdef або #ifndef і до наступної директиви #endif.

Ці засоби можна використовувати для боротьби з багаторазовим включенням вмісту файлів заголовку наступним чином

```
#ifndef MY_STRUCT_H
#define MY_STRUCT_H
#include <string>
struct MyStruct //Створення власної структури даних (класу)
{
	std::string name;
	unsigned index = 1;
	float sum = 0.0;
};
#endif
```

При першому включенні заголовку MyStruct.h директива #ifndef істинна, і препроцесор обробляє рядки після неї до директиви #endif. У результаті змінна MY_STRUCT_H буде визначена, а вміст заголовку MyStruct.h скопійовано до програми. Якщо надалі включити заголовок MyStruct.h в той же самий файл, то директива #ifndef виявиться брехнею і рядки між нею та директивою #endif будуть проігноровані.

Змінні препроцесора мають бути унікальними в усій програмі. Для цього в ім’я змінної препроцесора включається ім’я класу та ім’я змінної препроцесора задається лише великими літерами. Наприклад змінна препроцесора MY_STRUCT_H для включення визначення класу MyStruct із заголовку MyStruct.h.

Кожен заголовок повинен містити захист від повторного включення до програми його вмісту.

6.5.10 Абстрактний тип даних

Абстрактний тип даних (abstract data type). -це структура даних, яка інкапсулює (приховує) свою реалізацію. Мова C++ дозволяє створювати типи даних, які ведуть себе аналогічно базовим типам мови Сі. Такі типи зазвичай називають абстрактними типами даних (АТД).

```
struct Дата
{
private: //Змінні-члени класу закриті
	int місяць, день, рік;
public:
```

```
    void set(int, int, int);
    void print();

};
```

6.5.11 Агрегатний клас

Агрегатний клас (aggregate class) - клас лише з відкритими змінними-членами, без внутрішньокласових ініціалізаторів або конструкторів. Члени агрегатного класу можуть бути ініційовані укладеним у фігурні дужки списком ініціалізаторів.

```
// Класс з відкритими членами без конструкторів
//та внутрішньокласових ініціалізаторів
struct МояСструктура
{
    std::string рядок;
    int змінна;
};
// Ініціалізація об'єкту класу переліком ініціалізаторів
//у фігурних дужках
МояСструктура a = { "Початковий рядок", 0 };
```

Приклад програмної реалізації

Приклад роботи з класами

```
// Програма пошуку кореня рівняння методом половинного ділення
//

#include "stdafx.h"
#include "MyMethod.h"
#include "windows.h" //Необхідно для виведення в консолі української мови
using std::cout;
using std::cin;
using std::endl;

int main()
{
	SetConsoleCP(1251); //Необхідно для виведення в консолі
                            //української мови
	SetConsoleOutputCP(1251); //Необхідно для виведення в консолі
                                //української мови
	float k1, k2, k3, a, b, e;
	cout << "Ввведіть коефіцієнти рівняння" << endl;
	cout << "Введіть коефіцієнт k1" << endl;
	cin >> k1;
	cout << "Введіть коефіцієнт k2" << endl;
	cin >> k2;
	cout << "Введіть коефіцієнт k3" << endl;
	cin >> k3;
	cout << "Задайте ліву межу інтервалу пошуку кореня рівняння a"
		<< endl;
	cin >> a;
	cout << "Задайте праву межу інтервалу пошуку кореня рівняння b"
		<< endl;
	cin >> b;
	cout << "Задайте точність пошуку кореня рівняння e" << endl;
	cin >> e;
	MyClass MyCalc(k1, k2, k3, a, b, e);
	MyCalc.Method();
	system("pause"); // Команда затримки екрану
	return 0;
}
```

Приклад створення класу користувача у файлі заголовку

```
// MyMethod.h створений клас для пошуку кореня рівняння методом
половинного ділення

#include "math.h"
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

class MyClass
{
public: //Інтерфейс класу
      //Конструктор класу за замовчуванням
      MyClass() = default;
      //Конструктор класу
      MyClass(const float &k1, const float &k2, const float &k3,
            const float &a1, const float &b1, const float &e1) :
            A(a1), B(b1), E(e1), K1(k1), K2(k2), K3(k3) {};
      //Конструктор класу
      MyClass(const float &a1, const float &b1) : A(a1), B(b1) {};
      void Method(); //Об'явлення функції-члену класу без параметрів

private: //Реалізація класу
      float K1 = 1; //Ініціалізація даних-членів класу
      float K2 = 1;
      float K3 = 1;
      float A=-100;
      float B=100;
      float E=0.001;
      double MyFunc(const double&); //Об'явлення функції-члену класу
                                    //з параметром - константним посиланням
};

void MyClass::Method() //Визначення функції-члену класу без параметрів
{
      double za, zb, zx, x;
      double a(A), b(B); //Ініціалізація змінних a та b
                         //значеннями  змінних A та B відповідно
      unsigned i=0;
      za = MyFunc(A);
      zb = MyFunc(B);
      cout << "Знаходження коренів рівняння методом половинного ділення"
            << endl;
      if (za*zb<0)
      {
            do
            {
```

```
                ++i;
                x = (a + b) / 2;
                zx = MyFunc(x);
                cout << "Ітерація " << i << ", x=" << x << ", z="
                     << zx << endl;
                za * zx < 0 ? b = x : a = x, za = zx;
            } while (abs(b-a)>E);
            cout << "Корінь рівняння x = " << x << endl;
        }
        else
        {
            cout << "Корінь на інтервалі відсутній. Перевірте
                 відокремлення коренів!" << endl;
        }
}

double MyClass::MyFunc(const double &x) //Визначення функції-члену
                                        //класу з параметром –
                                        //константним посиланням
{
        double z;
        z = K1*exp(K2*x) + K3*x;
        return z;
}
```

Контрольні питання

1) Що таке власна структура даних або клас? Із чого складається визначення класу?
2) Що таке змінні-члени? Що таке внутрішньокласовий ініціалізатор?
3) Що таке абстракція даних? Що таке абстрактний тип даних?
4) Що таке інтерфейс та реалізація класу?
5) Що таке інкапсуляція? Що таке специфікатори доступу?
6) У чому різниця між використанням ключових слів `struct` та `class`?
7) Що таке функції-члени? Де та як вони об'являються та визначаються?
8) Що таке область видимості класу? Наведіть приклад визначення функції-члену поза тілом класу.
9) Для чого використовується параметр `this`? Як звернутися до члену об'єкту класу? Наведіть приклади.
10) Як підключити допоміжні функції для використання класом без їх включення у клас?
11) Що таке конструктор, для чого він використовується?
12) Що таке стандартний конструктор? Що таке синтезований стандартний конструктор? Як об'явити стандартний конструктор?
13) Що таке перелік ініціалізації конструктора? Для чого він потрібен?
14) Де потрібно визначати конструктор? Із чого складається конструктор? Наведіть приклади об'явлення та визначення конструкторів?
15) Що таке дружні відносини? Як об'явити дружню функцію? Як зробити дружню функцію доступною для користувачів класу?
16) Для чого потрібні файли заголовку? Як підключити заголовок до програми? Як організувати захист заголовку?
17) Що таке змінні препроцесора? Які правила створення імен змінних препроцесору та файлів заголовку загальноприйняті? Як працює директива `#include`?

7. Модульна контрольна робота

Для перевірки засвоєння студентами знань, отриманих при прослуховуванні лекцій, виконанні комп'ютерних практикумів та при самостійній роботі у відповідності до учбового плану проводиться модульна контрольна робота. Завдання модульної контрольної роботи носять як теоретичний, так і практичний характер. Модульна контрольна робота проводиться за всіма темами кредитного модуля.

Мета: ознайомитись із символьними та чисельними обчисленнями засобами C++; вивчити існуючі типи даних, структуровані типи даних (*вказівники, посилання, створення власної структури даних*); вивчити прийоми роботи з контейнером *vector*; навчитись створювати *файли заголовку*; відпрацювати засоби візуалізації розрахунків у C++; вивчити принципи створення *функцій користувача* та роботи з ними: *визначення функції*; *тіло функції*; тип значення, яке повертає функція з використанням оператору *return*; *об'явлення функцій* у *файлі заголовку*; виклик функції; передача аргументів; створення програм за допомогою функцій.

Завдання: створити консольний додаток для реалізації обчислень поставленої задачі згідно отриманого варіанту завдання з використанням створеної структури даних та покажчиків і посилань та зовнішніх функцій.

Загальні вимоги.

1) Створити консольний додаток з ім'ям виду *PrizvischeMKR*.
2) Програмний код модуля моє бути чітко структурований.
3) Імена об'єктів мають нести сенсові навантаження.
4) Програмний код має супроводжуватись коментарями в тексті програми.

Вимоги до виконання.

1) Створити власну *структуру даних* з ім'ям виду *my_struct*, яка включатиме в якості полів даних змінні для збереження значень *x, y(x), z(x)* та константні змінні *a*, *b* та *c*.
2) Визначити власну *структуру даних* у *файлі заголовку* з ім'ям виду *PrizvischeMKR.h.*
3) Створити відповідні типи *посилань* та *посилань на константу* для кожного поля створеної *структури даних*.

4) Запрограмувати вирішення поставленої задачі з використанням лише *посилань* на поля *структури*.
5) Введення вхідних даних організувати в режимі діалогу з користувачем.
6) Задати початкове, кінцеве значення змінної *x* та крок її зміни.
7) Створити зовнішню функцію для розрахунку *y(x)*.
8) Створити зовнішню функцію для розрахунку *z(x)*.
9) Розрахунок значень функцій *y(x)* та *z(x)* для різних значень змінної *x* організувати у циклі шляхом виклику відповідних зовнішніх функцій на кожній ітерації.
10) Створити вектор типу власної структури даних *my_struct*, та записати в нього значення *x, y(x), z(x)* на кожній ітерації.
11) Завдання для обчислень, вхідні дані та результати розрахунків вивести у зовнішній текстовий файл з ім'ям виду *ПрізвищеМКР_OUT*.
12) При виведенні вхідних даних та результатів розрахунків використовувати безпосередньо звернення до полів даних створеної *структури* замість *посилань* на них.
13) Виведення даних у вихідних файл організувати в режимі додавання в кінець файлу.

Теоретичні відомості

Тема 7.1. Тип vector

7.1.1 Визначення і ініціалізація векторів

Вектор (`vector`) – це колекція об'єктів однакового типу, кожному з яких просвоєний цілочисельний індекс, який надає доступ до цього об'єкту. Вектор – це *контейнер* (`container`), оскільки він «містить» інші об'єкти.

Щоб використовувати вектор, необхідно включити відповідний заголовок. Потрібно також включити об'явлення using

```
#include <vector>;
using std::vector;
```

Тип vector – це *шаблон класу* (`class template`). Мова C++ підтримує шаблони і класів, і функцій. Шаблони самі по собі не є ні функціями, ні класами. Їх можна вважати інструкцією для компілятора по створенню класів чи функцій. Процес створення компілятором класів чи функцій за шаблоном зветься *створенням екземпляру* (`instantiation`) шаблону. При створенні шаблону необхідно вказати, екземпляр якого класу чи функції має створити компілятор.

Для створення екземпляру шаблону класа слід вказати додаткову інформацію, характер якої залежить від шаблону. Ця інформація завжди задається однаково: в кутових дужках після імені шаблону.

У випадку вектора додатковою інформацією бути тип об'єктів, які він повинен містити:

```
vector<int> vecInt;          //вектор містить об'єкти типу int
vector<string> vecStr;       //вектор містить об'єкти типу string
vector<MyStruct> vecMyStr; //вектор містить об'єкти типу MyStruct,
                             //створеного користувачем
```

Можна визначити вектори для зберігання об'єктів практично будь-якого типу. Оскільки посилання не є об'єктами, не може бути вектору посилань. Також може бути вектор, елементами якого є інший вектор.

```
//Вектор, елементами якого є вектори, що містять текстові рядки
vector<vector<string>> vecVecStr;
```

Подібно будь-якому типу класа, шаблон vector контролює спосіб визначення і ініціалізації векторів. Найбільш розповсюджені способи визначення векторів наведені в табл. 7.1.1.

Таблиця 7.1.1 – Способи ініціалізації об'єкту класу `vector`

Ініціалізація	Роз'яснення
`vector<T> v1`	Вектор, що містить об'єкти типу T. Стандартний конструктор, v1 пустий
`vector<T> v2(v1)`	Вектор v2 - копія всіх елементів вектору v1
`vector<T> v2 = v1`	Еквівалент `v2(v1)`, вектор v2 - копія елементів вектору v1
`vector<T> v3(n, val)`	Вектор v3 містить n елементів зі значенням `val`
`vector<T> v4(n)`	Вектор v4 містить n екземплярів об'єкту типу T, ініціалізованих значенням за замовчуванням
`vector<T> v5{a,b,c …}`	Вектор v5 містить стільки елементів, скільки вказано у фігурних дужках. Елементи вектору ініціалізуються вказаними у фігурних дужках значеннями (ініціалізація переліком)
`vector<T> v5 = {a,b,c …}`	Еквівалент `v5{a,b,c …}`. Пряма ініціалізація

Ініціалізація вектор за замовчуванням дозволяє створити пустий вектор певного типу.

```
//Ініціалізація за замовчувнннням, у вектора sVec немає елементів
vector<string> sVec;
```

Вектор використовується в мові C++ замість масивів. Шаблон `vector` створений на основі типу масив із додаванням у нього необхідних властивостей. На відміну від масивів, у вектор *можна додавати елементи*. Тому створення пустого вектору певного типу є звичайною операцією з метою подальшого наповнення вектору елементами вказаного типу.

При визначенні вектору для його елементів можна також вказати початкові значення. Наприклад, можна скопіювати елементи з іншого вектора (на відміну від масивів, які копіювати не можна). При копіюванні векторів кожен елемент нового вектору буде копією відповідного елементу вихідного вектору. Обидва вектори повинні мати однаковий тип:

```
vector<int> vecInt(10, 1); //вектор містить десять одиниць
vector<int> iVec1 = vecInt; //вектор iVec1 містить десять одиниць
vector<int> iVec2(iVec1); //вектор iVec2 містить десять одиниць
```

Вектори підтримують *ініціалізацію переліком*, коли значення задаються у фігурних дужках після імені вектору:

```
//Вектор articles міститиме три рядкових елементи a, an та the
vector<string> articles{"a","an","the"};
```

При ініціалізації вектора значення можна пропустити, а вказати тільки кількість елементів. У такому випадку відбудеться *ініціалізація значення* (`value initialization`), тобто бібліотека створить ініціалізатор елементу сама. Кількість елементів можна вказувати тільки за допомогою прямої ініціалізації – у круглих дужках після імені вектору:

```
vector<int> iVec(10); //вектор із 10 нулів
vector<string> iVec(10); //вектор із 10 пустих рядків
```

Використання круглих та фігурних дужок дає різні результати:

```
vector<int> vec1(10); //вектор містить 10 елементів зі значенням 0
vector<int> vec2{ 10 }; //вектор містить один елемент зі значенням 10
vector<int> vec3(10,1); //вектор містить 10 елементів зі значенням 1
vector<int> vec4{ 10, 1 }; //вектор містить містить два елементи
                           //зі значенням 10 та 1
```

7.1.2 Операції з векторами

Зазвичай при створенні вектора невідома кількість його елементів та їх значення. Для додавання елементів у вектор використовують функцію `push_back()`, яка додає новий елемент у кінець вектора.

```
vector<int> v1; //пустий вектор
for (size_t i = 0; i != 10; i++)
{
      v1.push_back(i + 1); //додавання елементу в кінець вектора зі
                           //значенням i + 1
      cout << v1[i] << endl; //виведення значення вектора на екран
}
```

Результатом роботи даного приклада буде виведення на екран у стовпчик значень створеного вектора, якими будуть числа від 1 до 10.

Окрім функції push_back(), шаблон vector надає ще декілька операцій, більшість із яких подібна операціям класу string. Найбільш важливі з них наведені в табл. 7.1.2.

Таблиця 7.1.2 – Операції з векторами

Операція	Роз'яснення
v.push_back(t)	Додає елемент зі значенням t в кінець вектора v
v.empty()	Повертає значення true, якщо вектор v пустий. Інакше повертає значення false
v.size()	Повертає кількість елементів вектору v
v[n]	Повертає посилання на елемент у позиції n вектора v; позиції відраховуються від 0
v1 = {a,b,c …}	Замінює елементи вектору v1 копією елементів із розділеного комами переліку
v1 = v2	Замінює елементи вектору v1 копією елементів вектору v2
v1 == v2	Вектори v1 та v2 рівні, якщо кількість елементів рівна і вони містять однакові елементи на тих самих позиціях
v1 != v2	Вектори v1 та v2 не рівні, якщо хоча б один елемент відрізняється чи різна кількість елементів у векторах
<, <=, >, >=	Мають звичне значення і покладаються на алфавітний порядок символів

Доступ до елементів вектору здійснюється так само, як і до символів у рядку: за їх позицією у векторі.

Функції-члени empty() та size() працюють аналогічно, як і в класі string. Функція-член size() повертає значення типу size_type, яке визначене відповідним типом шаблону vector, наприклад vector<string>::size_type.

7.1.3 Оператор індексування

Оператор індексування ([]), як і оператор виклику функції, вважається бінарним оператором. Оператор індексування повинен бути нестатичною функцією-членом, яка приймає один аргумент. Цей аргумент може бути будь-якого типу і визначає необхідний індекс масиву.

Оператор індексування (оператор `[]`) для типу `string` отримує значення типу `string::size_type`, яке позначає позицію символу, до якого потрібен доступ. Оператор повертає посилання на символ у вказаній позиції.

Індексування рядків починається з нуля, першим символом буде `s[0]`, другим `s[1]`, а останнім `s[s.size() -1]`.

Значення оператора індексування називається індексом (index). Індекс може бути любим виразом, який повертає цілочисельне значення.

За допомогою *оператора індексування* можна вибрати вказаний елемент. Подібно рядкам, індексування вектора починається з `0`; індекс має тип `size_type` відповідного типу; якщо вектор не константний, то у повернутий оператором індексування елемент можна здійснити запис; можна розрахувати індекс і безпосередньо звернутись до елементу в даній позиції. За допомогою оператора індексування *неможливо додати елементи* у вектор, для цього використовується лише функція `push_back()`. Оператор індексування дозволяє звертатися (в тому числі і змінювати) лише до вже *існуючих елементів* вектору.

7.1.4 Використання ітераторів

Для доступу до символів рядка також можна використовувати *ітератори* (`iterator`). Окрім рядків і векторів у всіх бібліотечних контейнерів є ітератори, але тільки деякі з них підтримують оператор індексування. Тому ітератори є більш загальним та універсальним інструментом.

Як і вказівники, ітератори забезпечують опосередкований доступ до об’єкту. У випадку ітератора цим об’єктом є елемент в контейнері чи символ у рядку. Ітератор дозволяє вибрати елемент, а також підтримує операції переміщення з одного елементу на інший. Подібно покажчикам, ітератор може бути припустимим та неприпустимим. Припустимий ітератор вказує або на елемент, або на позицію за останнім елементом в контейнері. Всі інші значення ітератора неприпустимі.

Тип `vector` є контейнером і підтримує роботу з *ітераторами* аналогічно типу `string`. Вектор використовує функції-члени `begin()` та `end()` для звернення до першого та наступного за останнім елементу вектора.

```
vector<float> v{ 0.1, 0.2, 0.3, 0.4, 0.5 }
//Використання ітератора для звернення до елементів вектору
```

```
for (vector<float>::iterator iter = v.begin(); iter!= v.end(); iter ++)
{
      //Виведення на екран елементів вектору в рядок
      cout << *iter << " ";
}
```

Оскільки вектор може містити інші вектори, то, наприклад, для збереження елементів матриці використовують вектор векторів. Так для збереження матриці з цілих чисел необхідно створити наступний вектор:

```
vector<vector<int>> vec1;
```

7.1.4.1 Перелік функцій для роботи з ітераторами

На відміну від покажчиків, для отримання ітератора не потрібно використовувати оператор звернення до адреси (&). Для цього у типів, які мають ітератори, є члени, котрі повертають ці ітератори. В таблиці 7.1.3 наведений перелік основних функцій-членів для роботи з ітераторами.

Таблиця 7.1.3 – Функції для роботи з ітераторами типу `string`

Функція	Операція
begin	Повертає ітератор до початку
end	Повертає ітератор до кінця
rbegin	Повертає ітератор на початок при переборі у зворотному порядку
rend	Повертає ітератор у кінець при переборі у зворотному порядку
cbegin	Повертає const_iterator на початок
cend	Повертає const_iterator у кінець
crbegin	Повертає const_reverse_iterator на початок при переборі у зворотному порядку
crend	Повертає const_reverse_iterator у кінець при переборі у зворотному порядку

7.1.4.2 Функції begin() та end()

Типи, які підтримують роботу з ітераторами, мають функції-члени `begin()` та `end()`. Функція-член `begin()` повертає ітератор, який позначає перший елемент (або перший символ), якщо він є. Функція-член `end()` повертає ітератор, який вказує на наступну позицію за кінцем контейнеру (або рядку). Цей ітератор позначає неіснуючий елемент за кінцем контейнеру. Він використовується як індикатор того, що оброблені всі елементи контейнеру. Ітератор, який повертає функція `end()`, називають *ітератором після кінця* (`off-the-end iterator`), або скорочено *ітератором end*. Якщо контейнер пустий, функція `begin()` повертає той самий ітератор, що і функція `end()`.

Тип значень ітератора визначається в контейнері, для якого використовується ітератор. Наприклад, тип ітератора для рядкового типу буде `string::iterator`. Для визначення типу ітератора, як і багатьох інших типів, коли запис типу досить складний або невідомий до отримання значення виразу, зручно використовувати специфікатор `auto`.

```
string s;
//a позначає перший символ рядку s, b – елемент після останнього символу
auto a = s.begin(), b = s.end();
```

7.1.4.3 Використання зворотного ітератора і функцій rbegin() та rend()

Окрім перебору елементів контейнера по порядку від першого до останнього, в мові C++ є спеціальний тип ітератора, що дозволяє перебирати елементи у зворотному порядку. Для об'єктів типу `string` це тип *`string::reverse_iterator`*. В наступному прикладі використовуються функції `rbegin()` та `rend()` для перебору елементів контейнера у зворотному порядку.

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include <Windows.h>
#include <locale>
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	string речення("Робота з ітераторами");
	// Використання зворотного ітератора
	for (string::reverse_iterator ітератор = речення.rbegin();
		ітератор != речення.rend(); ітератор++)
	{
		cout << *ітератор;
	}
	cout << endl;
	return 0;
}
```

Даний програмний код виводить рядкову змінну типу `string` посимвольно у зворотному порядку за допомогою зворотного ітератора `string::reverse_iterator` та функцій, які з ним використовуються - `rbegin()` та `rend()`. Результатом роботи цієї програми буде наступний рядок:

```
имаротареті з атобоР
```

При використанні специфікатора auto попередній код у циклі for можна дещо спростити:

```
for (auto ітератор = речення.rbegin(); ітератор != речення.rend();
      ітератор++)
{
      cout << *ітератор;
}
```

Специфікатор auto автоматично визначає необхідний тип для ітератора у циклі for.

З об'єктами типу string також можна використовувати серійний цикл for, якщо необхідно перебрати змінну посимвольно. В цьому випадку форма запису циклу значно скорочується. В наступному прикладі за допомогою серійного циклу for підраховується кількість великих та малих літер «р» у реченні.

```
int кількість(0);
string речення("Робота з ітераторами");
// Підрахунок літер 'р' та 'Р' у реченні
for (auto літера: речення)
{
      if (літера == 'р' || літера == 'Р')
      {
            кількість++;
      }
      cout << літера;
}
cout << endl;
cout << "Кількість літер 'р' та 'Р' у реченні = " << кількість << endl;
```

Результат роботи даного фрагменту коду наведений нижче:

```
Робота з ітераторами
Кількість літер 'р' та 'Р' у реченні = 3
```

7.1.4.4 Використання константного ітератора і функцій cbegin() та cend()

В C++ також використовується тип *const_iterator* - це ітератор, який вказує на константний вміст. Даний ітератор може бути збільшений і зменшений (якщо він сам не є константним), точно так само, як і звичайний ітератор, який працює з функціями begin() та end(), але не може бути використаний для зміни вмісту, на який він вказує, навіть якщо строковий об'єкт не є сам константним.

Приклад використання константного ітератора з об’єктом типу string наведений нижче.

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <windows.h>
#include <locale>

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	std::string рядок("Робота з константними ітераторами");
	// Приклад роботи з константним ітератором
	for (std::string::const_iterator ітератор = рядок.cbegin();
		ітератор != рядок.cend(); ++ітератор)
		// Виводимо всі символи окрім пробілів
		if (*ітератор != ' ')
			std::cout << *ітератор;
	std::cout << '\n';
	// З використанням специфікатора auto друкуємо всі символи
	for (auto ітератор = рядок.cbegin(); ітератор != рядок.cend();
		++ітератор)
		std::cout << *ітератор;
	std::cout << '\n';
	return 0;
}
```

Результат роботи даної програми буде наступним:

```
Роботазконстантнимиітераторами
Робота з константними ітераторами
```

7.1.4.5 Використання константного зворотного ітератора і функцій crbegin() та crend()

Робота з типом *const_reverse_iterator* аналогічна роботі зі звичайним зворотним ітератором та відрізняється тим, що за допомогою даного ітератора не можна змінювати значень, на які він вказує.

Ілюстрація використання константного зворотного ітератора наведена нижче.

```
#include "stdafx.h"
#include <iostream>
#include <string>
#include <windows.h>
```

```
#include <locale>

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	std::string рядок("Робота з зворотними константними ітераторами");
	// Приклад роботи з константним ітератором
	for (std::string::const_reverse_iterator ітератор =
			рядок.crbegin(); ітератор != рядок.crend(); ++ітератор)
		// Виводимо всі символи окрім пробілів у зворотному порядку
		if (*ітератор != ' ')
			std::cout << *ітератор;
	std::cout << '\n';
	// З використанням специфікатора auto друкуємо всі символи
	// у зворотному порядку
	for (auto ітератор = рядок.crbegin(); ітератор != рядок.crend();
			++ітератор)
		std::cout << *ітератор;
	std::cout << '\n';
	return 0;
}
```

Результат роботи програмного коду наступний:

```
имаротаретіиминтнатснокиминторовззатобоР
имаротареті иминтнатснок иминторовз з атобоР
```

7.1.4.6 Операції з ітераторами

Ітератори підтримують лише кілька операцій, які перераховані в табл. 7.1.4.

Таблиця 7.1.4 – Стандартні операції з ітераторами контейнеру

Операція	Пояснення
`*iter`	Повертає посилання на елемент, позначений ітератором `iter`
`iter->mem`	Звернення до значення ітератора `iter` і обробка члена mem основного елементу. Еквівалентне `(*iter).mem`
`++iter`	Інкремент ітератора `iter` для звернення до наступного елементу контейнеру
`--iter`	Декремент ітератора `iter` для звернення до попереднього елементу контейнеру
`iter1 == iter2`	Порівняння двох ітераторів. Два ітератори рівні, якщо вказують на один і той самий елемент контейнеру або обидва вказують на елемент після останнього того самого контейнеру

Операція	Пояснення
`iter1 != iter2`	Перевірка на нерівність двох ітераторів. Два ітератори нерівні, якщо вказують на різні елементи контейнеру або елементи різних контейнерів

Подібно покажчикам, до значення ітератора можна звернутися (за допомогою оператора звернення до значення *), щоб отримати елемент, на який він посилається. Можна звертатись до значень лише припустимого ітератора.

Наступний код перетворює рядкові символи в прописні з використанням ітератора

```
string s("some string"); //Пряма ініціалізація рядкової змінної
//Заголовок оператора for, де задається початкове та кінцеве значення
//змінної циклу (ітератора) та її прирощення
for (string::iterator it = s.begin(); it != s.end(); it++)
{
	//Звернення до значення ітератора (символу рядка) та переведення
	//його у верхній регістр
	*it = toupper(*it);
}
cout << s << endl; //В результаті рядок матиме вигляд SOME STRING
```

В умові оператора `for` записаний логічний оператор `!=` замість оператора <. Це пов'язане з тим, що всі контейнери в мові C++ мають ітератори і для них визначені оператори == та !=. Але лише типи `string` та `vector` мають оператори індексування для яких також визначені оператори < та >. Щоб не думати про те, для якого контейнеру можна використовувати оператор <, програмісти C++ завжди обирають оператор `!=`.

Ітератори використовують оператор інкременту (оператор ++) для переміщення з одного елементу на наступний. Операція прирощення логічно подібна прирощенню цілого числа. У випадку цілих чисел результатом буде цілочисельне значення збільшене на 1. У випадку ітераторів результатом буде переміщення ітератора на одну позицію вперед.

Ітератори рядків та векторів підтримують додаткові операції, що дозволяють переміщувати ітератори на декілька позицій за раз (наприклад, `iter +n`, `iter - n`, iter += n, `iter1 - iter2`). Вони також підтримують всі оператори порівняння. Ці оператори часто називають *арифметичними діями з ітераторами* (`iterator arithmetic`).

Операції з ітераторами векторів і рядків. Додавання і віднімання цілого числа з ітератора призводить до зміни позиції ітератора на відповідну кількість елементів вперед або назад від початкового. Віднімання двох ітераторів дозволяє обчислити дистанцію між ними. Арифметичні дії припустимі лише для ітераторів, що відносяться до елементів того ж контейнера.

```
int i = 1;
vector<int> вект(10);
for (vector<int>::iterator ітер = вект.begin(); ітер != вект.end();
ітер++)
{
	*ітер = i++;
	std::cout << *ітер << "\t";
}
std::cout << "\n";
// Використання реверсивного ітератора для виведення вектора в
зворотньому напрямку
for (vector<int>::const_reverse_iterator ітер = вект.rbegin();
ітер != вект.rend(); ітер++)
{
	std::cout << *ітер << "\t";
}
std::cout << "\n";
vector<int>::iterator ітер1 = вект.end(), ітер2 = вект.begin();
// Приклад віднімання та додавання цілих чисел до ітератора
std::cout << *(ітер1 - 2) << "\t" << *(ітер2 + 3) << "\t" << std::endl;
```

Результат роботи прикладу:

```
1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1
9	4
```

7.1.4.7 Використання псевдонимів типів при роботі з ітераторами

Для спрощення формату запису складних типів використовують псевдоніми типів. *Псевдонім типу* (type alias) – це ім’я, яке є синонімом імені типу. Псевдонім типу дозволяє спростити складні визначення типів, полегшуючи їх використання. Псевдоніми типу дозволяють також підкреслити мету використання типу. Визначити псевдонім типу можна одним із двох способів. Традиційно для цього використовують ключове слово typedef:

```
typedef double wages; // wages – синонім для double
typedef vector<vector<float>> matrix; // matrix – синонім для двомірного
                                      //вектора (вектора векторів) дійсних чисел
wages w1; //еквівалент об’явлення double w1;
matrix m1; //еквівалент об’явлення vector<vector<float>> m1;
```

Іншим способом створення псевдоніму типу є *об'явлення псевдоніму* (alias declaration) за допомогою ключового слова using та знаку =.

```
using wages = double;
wages w1;
```

7.1.4.8 Точковий оператор та оператор стрілки

Щоб звернутись до значення вектору можна скористатись ітератором з *оператором звернення до значення* `*`, наприклад `*iter`. Якщо ми створили вектор векторів, то звернення до значення вектора векторів поверне об'єкт, яким буде вектор.

При зверненні до значення ітератора отримуємо об'єкт, на який вказує ітератор. Якщо цей об'єкт має тип класу, то може знадобитися доступ до члену отриманого об'єкту. Наприклад, якщо є вектор рядків, то може знадобитися дізнатись, чи не пустий певний елемент. Із урахуванням того, що `it` – це ітератор даного вектору, можна наступним чином перевірити чи не пустий рядок, на який вказує ітератор:

```
(*it).empty()
```

Круглі дужкі у виразі необхідні, бо вони вимагають застосувати *оператор звернення до значення* (`*`) до ітератора `it`, а до повернутого ітератором значення застосувати *точковий оператор* (`.`) (оператор звернення до члену класу). Якщо б не було круглих дужок, точковий оператор відносився б до ітератору `it`, а не до повернутого ним об'єкту.

Для спрощення таких виразів, мова пропонує *оператор стрілки* (`->`) (`arrow operator`). Оператор стрілки об'єднує оператори звернення до значення і доступ до члену. Таким чином, вираз `(*it).empty()` можна переписати у спрощеному вигляді:

```
it->empty()
```

Використання векторів разом із ітераторами надає більш гнучкий інструмент для роботи з наборами однотипних елементів, ніж масиви. Всі вектори, на відміну від масивів, є динамічними, дозволяють додавати значення, копіювати та присвоювати вектори, не вимагають програмування звільнення пам'яті.

Тема 7.2. Функції в C++

7.2.1 Визначення функції

Функція (`function`) – це іменований блок коду. Запуск цього коду на виконання здійснюється при виклику функції. Функція може отримувати будь-яку кількість аргументів і (зазвичай) повертає результат. Функція може бути перевантажена, відповідно, те ж ім'я може відноситись до кількох різних функцій.

Визначення функції (function definition) зазвичай складається з *типу повернутого значення* (return type), імені, переліку *параметрів* (parameter) і *тіла функції*. Параметри визначаються в розділеному комами переліку, укладеному в круглі дужки. Дії, які виконує функція, визначаються у блоці операторів, що зветься *тілом функції* (function body).

Для запуску коду функції використовується *оператор виклику* (call operator), що являє собою пару круглих дужок. Оператор виклику отримує вираз, який є функцією чи покажчиком на функцію. В круглих дужках через кому розміщується перелік *аргументів* (argument). Аргументи використовуються для ініціалізації параметрів функції. Тип викликаного виразу – це тип значення, яке повертає функція.

Для прикладу створимо функцію обчислення факторіалу заданого числа. Факторіал числа n є добутком чисел від 1 до n. Дану функцію можна визначити наступним чином:

```
int fact(int val)
{
	int ret = 1; //локальна змінна для збереження результатів у ході
		//їх обчислення
	while (val>1)
	{
		ret *= val--; // ret = ret*val, val= val-1

	}
	return ret; //повернення результату роботи функції
}
```

Функції присвоєне ім’я fact. Вона отримує один параметр типу int і повертає значення типу int. У циклі while обчислюється факторіал із використанням складеного оператора присвоєння та постфіксного оператору декременту. Оператор return виконується в кінці функції і повертає значення змінної ret.

Для виклику функції fact(), потрібно надати їй значення типу int. Результатом виклику також буде значення типу int.

```
int main()
{
	int j = fact(5); //j = 120, тобто результату fact(5)
	cout << "Факторіал п'яти 5! = " << j << endl;
	return 0;
}
```

Виклик функції виконує дві дії: ініціалізує параметри функції відповідними аргументами і передає керування коду функції. При цьому виконання функції,

яка викликає (calling) призупиняється і починається виконання *викликаної* (called) функції.

Виконання функції завершується оператором return. Як і виклик функції, оператор return виконує дві дії: повертає значення (якщо воно є) і передає керування назад функції, *яка викликає*.

7.2.2 Аргументи та перелік параметрів функції

Аргументи – це ініціалізатори для параметрів функції. Перший аргумент ініціалізує перший параметр, другий аргумент ініціалізує другий параметр і т.д. Тип кожного аргументу має співпадати з типом відповідного параметру, як і тип ініціалізатора має співпадати з типом об'єкту, який він ініціалізує. Потрібно передати точно таку ж кількість аргументів, скільки у функції параметрів.

Перелік параметрів функції може бути пустим, проте не може бути відсутнім. При визначенні функції без параметрів зазвичай використовують пустий список параметрів. Список параметрів, як правило, складається з розділеного комами переліку параметрів, кожен із яких виглядає як одиночне об'явлення. Навіть коли типи двох параметрів однакові, об'явлення слід повторити:

```
int f1(int v1, int v2) {/* ... */}
int f2(int v1, v2) {/* ... */ } //Помилка!!!
```

Параметри не повинні мати однакові імена.

У якості типу повернутого значення функції застосовується більшість типів. Зокрема, типом повернутого значення може бути void, це означає, що функція *не повертає значення*. Типом повернутого значення *не може бути* масив чи функція. Проте функція може повертати *покажчик* на масив чи функцію.

У мові C++ ім'я має область видимості, а об'єкт – тривалість існування (object lifetime).

- *Область видимості імен* – це *частина тексту програми*, в якій ім'я відомо.
- *Тривалість існування об'єкту* – це *час при виконанні програми*, коли об'єкт існує.

Тіло функції, як блок операторів, формує *нову область видимості*, в якій можна визначати змінні. Параметри та змінні, які визначені в тілі функції, називають *локальними змінними* (local variable). Вони є локальними для даної функції і *приховують* (hide) об'явлення того ж імені у зовнішній області видимості. Локальні змінні мають пріоритет над глобальними, об'явленими поза межами блоку з тим же ім'ям.

Об’єкти, які визначені поза межами будь-якої з функцій, існують протягом виконання програми. Такі об’єкти створюються при запуску програми і не видаляються до її завершення. Тривалість існування локальної змінної залежить від того, як вона визначена.

Об’єкти, які відповідають звичайним локальним змінним, створюються у місці визначення змінної у функції (блоці) і видаляються, коли процес виконання досягає кінця блоку. Об’єкти, що існують тільки під час виконання блоку, в якому вони були визначені, називають *автоматичними об’єктами* (automatic object).

По завершенні виконання блоку значення автоматичних об’єктів, які були створені в цьому блоці, невизначені.

Параметри – це автоматичні об’єкти. Значення неініціалізованих локальних змінних вбудованого типу невизначені.

Для створення локальної змінної, тривалість існування якої не переривається між викликами функції, при визначенні локальної змінної використовують ключове слово static. Кожен *локальний статичний об’єкт* (local static object) ініціалізується раніше, ніж хід виконання досягне визначення об’єкту. Локальна статична змінна не видаляється по завершенню роботи функції; вона видаляється по завершенню роботи програми.

```
static size_t i = 0; //значення змінної зберігається між викликами функції
```

Як і будь-яке інше ім’я, ім’я функції має бути об’явлене раніше, ніж його можна буде використовувати. Подібно до змінних, функція може бути *визначена* тільки один раз, проте *об’явлена* може бути багаторазово.

7.2.3 Аргумент за замовчуванням

Аргумент за замовчуванням (default argument) - це значення, яке визначене для використання, коли аргумент пропущено при виконанні функції.

```
void обл() { // Початок нової області видимості
void функ(int, int);  // Внутрішнє оголошення
//без ініціалізаторів за замовчуванням
     функ(4); // Помилка: недостатньо аргументів у виклику функції
функ(int, int)
     void функ(int, int = 6);   // Переоголошення функції
//з другим ініціалізатором за замовчуванням
     функ(4); // OK: виклик функції з аргументами функ(4,6)
     void функ(int, int = 7);   // Помилка: перевизначення аргументу
//за замовчуванням в тій же області видимості
}
```

7.2.4 Оголошення функції

Об'явлення функції подібно до її визначення, але у об'явлення *немає тіла функції*. В об'явленні тіло функції замінюється крапкою з комою. Оскільки в об'явленні немає тіла функції, *відпадає необхідність* в іменах параметрів. Проте імена параметрів у об'явленні функції часто використовують для полегшення розуміння користувачем призначення функції.

```
void print(vector<int>::const_iterator beg, vector<int>::const_iterator end);
```

Три елементи об'явлення функції: тип повернутого значення, ім'я функції та тип параметрів – описують *інтерфейс* (`interface`) функції. Вони задають всю інформацію, що необхідна для виклику функції. Об'явлення функції називають також *прототипом функції* (`function prototype`).

Об'явлення функцій розміщають у файлах *заголовку*, а *визначення* – у *файлах вихідного коду*. Файл вихідного коду, в якому функція *визначена*, повинен підключати *заголовок*, в якому функція *об'явлена*. Так компілятор зможе перевірити відповідність визначення та об'явлення.

У *файлі вихідного коду*, де передбачається *використання зовнішньої функції*, також необхідно *підключити заголовок*, в якому ця функція об'явлена. Через файл заголовку з об'явленням функції здійснюється зв'язок між файлом вихідного коду, в якому функція визначена, та файлом вихідного коду, де функція використовується. Відповідно, заголовок з об'явленням функції включається в обидва файли вихідного коду: файл із визначенням функції та файл, в якому вона застосовується. Створені файли заголовку та файли вихідного коду потрібно *підключати* до проекту щоб компілятор їх включив у програму.

Мова C++ дозволяє розділяти програми на логічні частини, надаючі засіб, відомий як *роздільна компіляція* (`separate compilation`). Роздільна компіляція дозволяє поділити програму на кілька файлів, кожен із яких може бути відкомпільований окремо.

При кожному виклику функції її параметри створюються заново. Параметри ініціалізуються так само, як і звичайні змінні. Якщо параметр – посилання, то параметр прив'язується до свого аргументу. В іншому випадку, значення аргументу копіюється.

Коли параметр – посилання, говорять що аргумент *передається за посиланням* (`pass by reference`) або що функція *викликається за посиланням* (`call by reference`). Подібно будь-якому іншому посиланню, параметр посилання – це лише *псевдонім* об'єкту, до якого він прив'язаний, тобто параметр посилання – це псевдонім свого аргументу.

```
void reset(int i) //i – копія аргументу
```

```
{
      i = 0; //змінює значення в середині функції, значення переданого
            //аргументу за межами функції не міняється
}
```

Коли значення аргументу копіюється, параметр і аргумент – незалежні об’єкти. Кажуть, що такі аргументи *передаються за значенням* (`pass by value`) або що функція *викликається за значенням* (`call by value`).

```
void reset(int &i) //i - лише інше ім'я об'єкту
{
      i = 0; //змінює значення об'єкту, на який посилається i
            //у функції змінюється значення зовнішнього об’єкту
}
```

7.2.5 Передача параметрів у функції

Передача за значенням (pass by value) параметрів функції - спосіб передачі аргументів параметрами, що не мають тип посилання. Параметр значення - це копія значення відповідного аргументу.

Нижче наведений приклад програми, в якій визначена функція `swap(int x, int y)`. З визначення та оголошення функції видно, що параметри до неї передаються за значенням.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
//Необхідно для введення/виведення української мови в консолі
#include <locale>;
#include "windows.h";

// Оголошення функції
void swap(int x, int y);

int main() {
      //Встановлення таблиці кодування для введення укр. мови
      SetConsoleCP(1251);
      //Встановлення таблиці кодування для виведення укр. мови
      SetConsoleOutputCP(1251);
      // Оголошення локальних змінних:
      int a = 100;
      int b = 250;
      cout << "Перед заміною, значення a = " << a << endl;
      cout << "Перед заміною, значення b = " << b << endl;
      // Викликаємо функцію та передаємо в неї аргументи
      //за значенням
```

```
        // При цьому у функції створюються копії значень
        //переданих аргументів
        swap(a, b);
        cout << "Після заміни значення a = " << a << endl;
        cout << "Після заміни значення b = " << b << endl;
        // У функції main() значення a та b не помінялись!!!
        _getch();// Очікування натискання клавіші
        //перед закриттям вікна
        return 0;
}

// Визначення функції для заміни місцями двох чисел
void swap(int x, int y) {
        cout << "Перед заміною у функції swap() значення x = " << x
             << endl;
        cout << "Перед заміною у функції swap() значення y = " << y
             << endl;
        int temp;
        temp = x; // Зберігаємо значення змінної x
        x = y;    // Записуємо значення змінної y до змінної x
        y = temp; // Записуємо збережене значення до змінної y
        cout << "Після заміни у функції swap() значення x = " << x
             << endl;
        cout << "Після заміни у функції swap() значення y = " << y
             << endl;
        // Всередині функції swap() значення помінялись,
        //а за її межами - ні
        return;
}
```

Результат роботи прикладу:

```
Перед заміною, значення a = 100
Перед заміною, значення b = 250
Перед заміною у функції swap() значення x = 100
Перед заміною у функції swap() значення y = 250
Після заміни у функції swap() значення x = 250
Після заміни у функції swap() значення y = 100
Після заміни значення a = 100
Після заміни значення b = 250
```

Коли ми передаємо до функції `swap(int x, int y)` параметри a та b, у функції створюються копії цих змінних, з якими і працює функція. При цьому значення змінних за межами області видимості функції змінюватись не будуть.

Функція може повертати лише *одне значення*. У випадках, коли функція має повертати більше ніж одне значення, використовують додаткові *параметри посилання*. Так як посилання – лише інше ім’я об’єкту, то змінюючи такі додаткові

параметри всередині функції, ми міняємо значення об’єктів, на які вони посилаються, і за її межами. Тобто повертаємо змінені значення переданих за посиланням аргументів в тіло програми по завершенню виконання функції.

Передача за посиланням (pass by reference) параметрів функції - спосіб передачі аргументів параметрам посилального типу. Параметри посилання працюють так само, як і будь-яке інше посилання; параметр пов'язаний зі своїм аргументом.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
//Необхідно для введення/виведення української мови в консолі
#include <locale>;
#include "windows.h";

// Оголошення функції з параметрами посиланнями
void swap(int &x, int &y);

int main() {
	//Встановлення таблиці кодування для введення укр. мови
	SetConsoleCP(1251);
	//Встановлення таблиці кодування для виведення укр. мови
	SetConsoleOutputCP(1251);
	// Оголошення локальних змінних:
	int a = 100;
	int b = 250;
	cout << "Перед заміною, значення a = " << a << endl;
	cout << "Перед заміною, значення b = " << b << endl;
	// Викликаємо функцію та передаємо в неї аргументи
	//за посиланням
	// При цьому у функції безпосередньо використовуються змінні,
	//передані як аргументи
	swap(a, b);
	cout << "Після заміни значення a = " << a << endl;
	cout << "Після заміни значення b = " << b << endl;
	_getch();// Очікування натискання клавіші
	//перед закриттям вікна
	return 0;
}

// Визначення функції для заміни місцями двох чисел
void swap(int &x, int &y) {
	int temp;
	temp = x; // Зберігаємо значення змінної x
	x = y;    // Записуємо значення змінної y до змінної x
	y = temp; // Записуємо збережене значення до змінної y
	return;
```

```
}
```

Результат роботи прикладу:

```
Перед заміною, значення a = 100
Перед заміною, значення b = 250
Після заміни значення a = 250
Після заміни значення b = 100
```

Якщо параметр, який передається у функцію не повинен мінятися, потрібно визначати його як *константне посилання* за допомогою ключового слова const.

```
void print(const double &y)      //Визначення функції для друку значення
                                 //аргументу
{
      static int i = 0;     //Змінна, яка зберігає своє значення між
                            //викликами функції
      cout << "Ітерація " << ++i << endl;
      cout << "y = " << y << endl;
}
```

Використання неконстантних посилань обмежує можливість використання функції. Через константний параметр посилання до функції можна передати як константний, так і неконстантний аргумент. Через неконстантний параметр посилання можна передати лише неконстантний аргумент і не можна передавати константні аргументи, в тому числі рядкові чи чисельні літерали.

7.2.6 Первантажені функції

Функції, які розташовані в одній області видимості, називають *перевантаженими* (overload), якщо вони мають однакові імена, але різні списки параметрів. При виклику такої функції компілятор приймає рішення про використання певної версії на основі типу переданого аргументу. У перевантажених функцій має однозначно відрізнятися тип параметрів, або має бути різна кількість параметрів.

```
void print(const double &y);
void print(const int *beg, const int *end);
```

Перевантажена функція (overloaded function) - це функція, яка має ту ж назву, що і принаймні одна інша функція. Перевантажені функції повинні відрізнятися за кількістю або типом їх параметрів. Приклад використання перевантажених функцій наведений у прикладі нижче.

```
// Оголошення перевантажених функцій:
string додавання(string&, string&);
int додавання(int, int);
int додавання(char&, char&, char&);

// Перевантажена функція №1
string додавання(string &ряд1, string &ряд2)
{
	return ряд1 + " " + ряд2;
}

// Перевантажена функція №2
int додавання(int число1, int число2)
{
	return число1 + число2;
}

// Перевантажена функція №3
int додавання(char &символ1, char &символ2, char &символ3)
{
	return (символ1 + символ2 + символ3);
}
```

В цьому прикладі використовується три різні функції з однаковими іменами, які виконують схожі дії. Для того, щоб можна було використовувати дані функції, кількість параметрів у функціях №1 та №2 відрізняється від кількості параметрів у функції №3. У функціях №1 та №2 кількість параметрів однакова, проте ці параметри відрізняються за типом, що дозволяє однозначно виконати підбір функції компілятором при її виклику.

Підбір функції (function matching) – це процес, в ході якого компілятор асоціює виклик функції з певною версією з набору перевантажених функцій. При підборі функції використовуються вказані у зверненні аргументи, які порівнюються зі списком параметрів кожної версії перевантаженої функції.

Надалі попередній приклад програмного коду з визначеннями перевантажених функцій доповнений функцією `main()`, в якій виконується виклик цих перевантажених функцій.

```
/ Оголошення перевантажених функцій:
string додавання(string&, string&);
int додавання(int, int);
int додавання(char&, char&, char&);

int main() {
	int a = 87;
	int b = 189;
	string s1("Результат роботи");
```

```
    string s2("перевантаженої функції");
    char ch1('a'), ch2('b'), ch3('c');
    // Виклику відповідає перевантажена функція string
    //додавання(string&, string&)
    cout << додавання(s1, s2) << endl;
    // Виклику відповідає перевантажена функція int
    //додавання(int, int)
    cout << додавання(a, b) << endl;
    // Виклику відповідає перевантажена функція int
    //додавання(char, char, char)
    cout << додавання(ch1, ch2, ch3) << endl;
    // Помилка! Відсутня відповідність функції даному виклику
    cout << додавання(s1, ch3) << endl;
    return 0;
}

// Перевантажена функція №1
string додавання(string &ряд1, string &ряд2)
{
    return ряд1 + " " + ряд2;
}

// Перевантажена функція №2
int додавання(int число1, int число2)
{
    return число1 + число2;
}

// Перевантажена функція №3
int додавання(char &символ1, char &символ2, char &символ3)
{
    return (символ1 + символ2 + символ3);
}
```

У виклику функції `додавання(s1, ch3)` виникає помилка через те, що немає перевантаженої функції з набором параметрів такого типу.

Приклад програмної реалізації

Приклад роботи з функціями користувача

```
//Програма обчислення значень функції F(x,y) у заданих точках

#include "stdafx.h"
#include <iostream>
#include "windows.h"
#include "HeaderFxy.h"          //Підключення власного файлу заголовку,
                                //де об'явлена структура даних
#include "fanctFxy.h"           //Підключення власного файлу заголовку,
                                //де об'явлена функція f
#include <vector>
using std::vector;
using std::cin;
using std::cout;
using std::endl;


int main()
{
      SetConsoleCP(1251);               //Необхідно для введення в консолі
                                        //української мови
      SetConsoleOutputCP(1251);         //Необхідно для виведення в консолі
                                        //української мови
      while (true)
      {
            const double sgX = 1.9;           //Об’явлення та ініціалізація
                                              //константної змінної
            const double sgY = 0.2;
            fxy fi;                     //Об’явлення змінної типу створеної
                                        //структури даних fxy
            vector<fxy> vectF;          //Об’явлення вектору для збереження
                                        //змінної типу fxy
            size_t n;
            cout << "Задайте початкове значення x(0)" << endl;
            cin >> fi.x;
            cout << "Задайте початкове значення y(0)" << endl;
            cin >> fi.y;
            cout << "Задайте кількість розрахунків значень функції F(x,y)"
                  << endl;
            cin >> n;
            for (size_t i = 0; i < n; i++)
            {
                  functXY(fi);          //Виклик зовнішньої функції
                                        //розрахунку залежності
                  vectF.push_back(fi);  //Запис змінної типу fxy у вектор
                  printFXY(vectF[i]);   //Виклик зовнішньої функції
```

```
                                        //для друку елементу вектору
                fi.x += sgX;
                fi.y += sgY;
            }
            cout << "Запустити розрахунок ще раз? [y] [n]" << endl
                << endl;
            char y;
            cin >> y;
            if (y == 'n')
            {
                break;                  //Переривання роботи програми
                                        //шляхом виходу з циклу while
            }
        }
        return 0;
}
```

Приклад визначення функцій користувача

```
//FunctXY.cpp

#include "stdafx.h"
#include <math.h>
#include <iostream>
#include "HeaderFxy.h"          //Підключення власного файлу заголовку,
                                //де об'явлена структура даних
#include "fanctFxy.h"           //Підключення власного файлу заголовку,
                                //де об'явлені функції
using std::cin;
using std::cout;
using std::endl;

void functXY(fxy &f)        //Визначення функції для розрахунку значення
                            //на окремій ітерації
{
     f.f = (log(sqrt(f.x)) - log(sqrt(f.y)))*pow((f.x - f.y), 1 / 3)
          / (1 / tan(f.x));
}

void printFXY(const fxy &f)     //Визначення функції для друку елементів
                                //вектору
{
     static int i = 0;      //Змінна, яка зберігає своє значення між
                            //викликами функції
     cout << "Ітерація " << ++i << endl;
     cout << "F(" << f.x << "," << f.y << ")=" << f.f << endl;
}
```

Приклад об'явлення функцій користувача у файлі заголовку

```
//functFxy.h


#pragma once
#ifndef FANCTFXY_H          //Попередження повторного визначення класу
#define FANCTFXY_H

void functXY(fxy &f);           //Об'явлення функції functXY із
                                //параметром посиланням,
                                //яка не повертає значення

void printFXY(const fxy &f);    //Об'явлення функції printFXY із
                                //параметром константне посиланням,
                                //яка не повертає значення

#endif FANCTFXY_H
```

Приклад об'явлення структури даних користувача у файлі заголовку

```
//HeaderFxy.h

#pragma once
#ifndef HEADERFXY_H          //Попередження повторного визначення класу
#define HEADERFXY_H

struct fxy                   //Створення власної структури даних (класу)
{
     double x;
     double y;
     double f;
};

#endif HEADERFXY_H
```

Контрольні питання

1) Що таке вектор? Що потрібно для використання векторів у програмі?
2) Що таке шаблон класу? Що таке створення екземпляру шаблону? Наведіть приклади.
3) Наведіть способи ініціалізації об'єктів класу `vector`.
4) Що таке ініціалізація переліком та ініціалізація значення для об'єкту типу `vector`? У чому відмінність використання круглих та фігурних дужок під час ініціалізації об'єкту типу `vector`, наведіть приклади?
5) Перерахуйте основні операції з векторами.
6) Поясніть як можна отримати доступ до елементів вектора за допомогою індексування та ітераторів? Наведіть приклади.
7) Що таке псевдонім типу та які методи створення псевдонімів ви знаєте? Наведіть приклади об'явлення псевдонімів.
8) Що таке оператор звернення до значення та оператор звернення до члену? Наведіть приклади.
9) Для чого потрібен оператор стрілки? Наведіть приклади.
10) Перерахуйте основні відмінності між масивами та векторами. Який із цих типів більш зручний у використанні?
11) Що таке функція?
12) Що таке визначення функції, тип повернутого значення, параметри, тіло функції?
13) Що таке оператор виклику функції, з чого він складається?
14) Для чого потрібен оператор `return`?
15) Що таке аргументи функції? Для чого використовуються аргументи?
16) Що таке перелік параметрів функції? Як об'являються параметри функції? Чи може бути функція без параметрів? Чи може функція не повертати значення? Наведіть приклади.
17) Що таке область видимості імен та тривалість існування об'єкту?
18) Що таке локальні змінні? Чим вони відрізняються від глобальних?
19) Що таке автоматичний об'єкт? Що таке локальний статичний об'єкт? Наведіть приклади?
20) Що таке об'явлення функції? Що таке інтерфейс функції? Наведіть приклад.

21) Як створити зовнішню функцію? Як підключити зовнішню функцію користувача для використання у файлі вихідного коду?
22) Що таке роздільна компіляція та для чого вона потрібна?
23) Чим відрізняється передача аргументу за посиланням від передачі аргументу за значенням?
24) Скільки значень може повертати функція? Чи може функція повернути кілька значень, якщо так, то як це реалізувати? Наведіть приклади.
25) Як потрібно визначати параметр функції, який не повинен змінюватись? Наведіть приклади.
26) Що таке перевантажені функції? Які особливості їх використання? Наведіть приклади.

Рекомендована література

Базова

1) Стенли Б. Липпман, Жози Лажойе, Барбара Э. Му Язык программирования C++. Базовый курс. М.: Вильямс, 2014. – 1120 с.
2) Бьярне Страуструп Программирование: принципы и практика использования C++. - М.: ООО «И.Д. Вильямс», 2011. – 1248 с.
3) Прата С. Язык программирования C++. Лекции и упражнения. Учебник. - СПб.: ООО «ДиаСофтЮП», 2005. 1104 с.
4) Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. - М.: Питер-ДМК, 2006. – 240 с.
5) Прикладне програмне забезпечення - 2. Технології об'єктно-орієнтованого програмування: методичні рекомендації до виконання комп'ютерних практичних робіт для студентів напряму підготовки 6.050202 – «Автоматизація та комп'ютерно-інтегровані технології» [Електронний ресурс] / [уклад. Бендюг В. І., Комариста Б. М.]. – К: 2016. – 153 с.

Допоміжна

6) Дейтел Х.М., Дейтел П.Дж. Как программировать на C++: 5-ое изд. Пер. с англ. – М.: Бином-Пресс, 2008. - 1456 с.
7) Аверкин В.П., Бобровский А.И. и др. под ред. Хомоненко А.Д. Программирование на C++. Учебное пособие. Корона-Принт. 1999. - 252 с.
8) Александреску А. Современное проектирование на C++. Серия C++ In-Depth, т. 3.: Пер. с англ. – М.: Вильямс, 2002. – 336 с.
9) Астахова И.Ф., Власов С.В. Язык C++. Учебное пособие / И.Ф. Астахова, С.В. Власов, В.В. Фертиков, А.В. Ларин. – Мн.: Новое знание, 2003. – 203 с.
10) Глушаков С.В., Коваль А.В., Смирнов С.В.. Язык программирования C++, учебный курс. – Харьков: Фолио, 2001. – 500 с.
11) Девис С.Р. C++ для чайников, 4-е изд.: Пер. с англ. – М.: Вильямс, 2003. – 336 с.

Інформаційні ресурси

12) Язык программирования C++ [Електрон. ресурс] // Основы программирования на языках Си и C++ для начинающих - Режим доступу: http://cppstudio.com/cat/274/
13) Бендюг Владислав Іванович [Електрон. ресурс] // Офіційний сайт – Режим доступу: http://бендюг.укр/
14) Єдине інформаційне середовище - Національний технічний університет України «КПІ» [Електрон. ресурс] // Електронний КАМПУС НТУУ «КПІ» – Режим доступу: http://login.kpi.ua/

Додаток А

Титульний аркуш

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Хіміко-технологічний факультет

Кафедра кібернетики хіміко-технологічних процесів

Комп’ютерний практикуму №1

з дисципліни

«ТЕХНОЛОГІЇ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ»

тема: Створення консольного додатку

Виконав: ст. групи ХА-41 Іваненко І.І.
Перевірив: доц. каф. КХТП Бендюг В.І.

Оцінка, балів:
Виконання _____ ;
оформлення звіту _____ ;
захист результатів _____.

Загальна оцінка _____.

_____
(підпис викладача)

Київ – 2018

Індивідуальні завдання до домашньої контрольної роботи

1. Обчислення кореня заданого нелінійного рівняння $f(x) = 0$ на попередньо визначеному інтервалі $[a, b]$[1] **методом хорд** із заданою точністю ε.
2. Обчислення кореня заданого нелінійного рівняння $f(x) = 0$ на попередньо визначеному інтервалі $[a, b]$[1] **методом дотичних** із заданою точністю ε.
3. Обчислення кореня заданого нелінійного рівняння $f(x) = 0$ на попередньо визначеному інтервалі $[a, b]$[1] **комбінованим методом хорд-дотичних** із заданою точністю ε.
4. Розв'язання системи лінійних алгебраїчних рівнянь **методом ітерацій** із заданою точністю ε.
5. Розв'язання системи лінійних алгебраїчних рівнянь **методом Гауса-Зейделя** із заданою точністю ε.
6. Розв'язання системи двох нелінійних рівнянь **методом ітерацій** із заданою точністю ε.
7. Розв'язання системи двох нелінійних рівнянь **методом Ньютона** із заданою точністю ε.
8. Розв'язання задачі інтерполяції (як прямої, так і оберненої) **методом Лагранжа**.
9. Розв'язання прямої задачі інтерполяції (екстраполяції) **методом Ньютона**.
10. Розв'язання оберненої задачі інтерполяції (екстраполяції) **методом Ньютона**.
11. Розв'язання звичайного диференціального рівняння першого порядку **методом Ейлера**.
12. Розв'язання звичайного диференціального рівняння першого порядку **удосконаленим методом Ейлера**.
13. Розв'язання звичайного диференціального рівняння першого порядку **методом Ейлера-Коші**.
14. Розв'язання звичайного диференціального рівняння першого порядку **методом Ейлера-Коші з наступною ітераційною обробкою**.
15. Розв'язання звичайного диференціального рівняння першого порядку **методом Рунге-Кутта**.
16. Обчислення визначеного інтегралу **методом трапецій**.
17. Обчислення визначеного інтегралу **методом Сімпсона**.
18. Розв'язання системи лінійних алгебраїчних рівнянь **методом ітерацій** із заданою точністю ε.

19. Розв’язання системи лінійних алгебраїчних рівнянь **методом Гауса-Зейделя** із заданою точністю ε.
20. Розв’язання системи лінійних алгебраїчних рівнянь 2-го порядку **за правилом Крамера**.
21. Розв’язання системи лінійних алгебраїчних рівнянь 3-го порядку **за правилом Крамера**.
22. Розв’язання задачі екстраполяції (як прямої, так і оберненої) **методом Лагранжа**.

Індивідуальні завдання до модульної контрольної роботи

1. $y(x) = 2 - x - \ln x,$ $\quad z(x) = \sqrt[3]{3ax^2 + bx} + \tan(cx - 8).$

2. $y(x) = 0{,}3e^{-0{,}6x} - x,$ $\quad z(x) = cx^3 - a\cos x^2 + b.$

3. $y(x) = 2^x + 4x - 3,$ $\quad z(x) = \frac{x^3+a}{bx-1} + \sqrt[3]{3x^2} + cx.$

4. $y(x) = e^{-2x} - 3x + 1,$ $\quad z(x) = \frac{ax^3-3x^2}{bx-2} + 3c\sin x.$

5. $y(x) = \ln(4{,}6x) - 5{,}2x + 1{,}4,$ $\quad z(x) = ax^3\sin(b + x) - c\,x.$

6. $y(x) = 0{,}4\,e^{-0{,}7x} - x - 6,$ $\quad z(x) = \mathrm{a}\tan x^2 - (bx^2 + c\,x)/(x + 4).$

7. $y(x) = \ln 3x - 3{,}4x + 2{,}6,$ $\quad z(x) = \frac{ax^3-3x^2}{bx-5} + \cos cx.$

8. $y(x) = 0{,}81\,e^{-0{,}6x} - 2x,$ $\quad z(x) = \mathrm{a}\log x^3 - (bx^2 + cx)/4b.$

9. $y(x) = \ln(1{,}5x) - 1{,}7x + 3,$ $\quad z(x) = x^3 - ax^2 + \tan(bx - c).$

10. $y(x) = 0{,}6\,e^{-0{,}5x} - 0{,}8x,$ $\quad z(x) = \sqrt[3]{x^3 - ax^2} + \sin(bx - c).$

11. $y(x) = 3^x + 2x - \ln x,$ $\quad z(x) = x^3 - \sqrt{ax^2 + 2bx} - \cos cx.$

12. $y(x) = 2e^x - 3x + 1,$ $\quad z(x) = \log_a x^3 + b\sqrt[3]{cx^2 - 8}.$

13. $y(x) = \ln 4x - 4{,}5x + 2,$ $\quad z(x) = (x^2 + ax)^{3/2} - \frac{2-bx}{3x^2+cx} + 3b.$

14. $y(x) = 0{,}73e^{-0{,}6x} - x,$ $\quad z(x) = \frac{1}{x^3-ax^2} + \sqrt{3bx - c}.$

15. $y(x) = \ln 6x - 7{,}1x + 1,$ $\quad z(x) = 3a\cos^2(-3bx^2) + \frac{bx-2}{cx}.$

16. $y(x) = 0{,}9e^{-0{,}8x} - 0{,}5x,$ $\quad z(x) = x^{3a}\sin\left(\frac{ax}{b+3x} - c\right) - bx^2.$

17. $y(x) = \lg(7{,}6x) - 8{,}6x + 0{,}5,$ $\quad z(x) = \frac{ax^3-5x^2}{b+\tan(cx^2)} + bx - c.$

18. $y(x) = 2e^{x} - 2x + 3,$ $z(x) = \sqrt[3]{3ax^2 + bx} + \cos(cx - 8b).$

19. $y(x) = \lg(8{,}5x) - 9{,}6x + 2,$ $z(x) = \frac{1}{x^3} - 5ax^{2-bx} + \sin(cx - 5b).$

20. $y(x) = e^{-3x} - 3x + 1,$ $z(x) = \mathrm{a}\tan^2 3x^2 - \frac{bx^2+cx}{ax-5b} - c.$

21. $y(x) = e^{-3x} - 3x + 1,$ $z(x) = \sqrt[3]{ax^3 + 3bx^2} + \frac{\sin cx}{2bx+1} + 3a.$

22. $y(x) = \lg 2{,}3x - 2{,}5x + 2{,}7,$ $z(x) = (x^3 - ax^2)^{bx} + \cos(cx + 2) - b.$

23. $y(x) = 0{,}54e^{-0{,}56x} - 0{,}98x,$ $z(x) = (ax^3 + x^2)^{1/bx} + c\cos x + bx.$

24. $y(x) = \lg 3{,}8x - 4{,}3x + 2,$ $z(x) = \frac{2ax+\sin(bx-3)}{x^3-cx^2} + bx.$

25. $y(x) = 0{,}65e^{-0{,}6x} - 0{,}89x,$ $z(x) = a\cos 2x + x^3 - bx^2 + c\sin(bx - 5).$

26. $y(x) = \lg 0{,}9x - 1{,}4x + 2{,}8,$ $z(x) = \tan^2 x^3 - \frac{ax^2+bx}{cx+3} + b.$

27. $y(x) = 0{,}47e^{-0{,}7x} - 0{,}93x,$ $z(x) = \sqrt[3]{\frac{x^3-ax^2}{bx^3}} + \sin(cx - b).$

28. $y(x) = 5^{x} + 4x^{2-x} - 3,$ $z(x) = \frac{\cos(ax+b)}{\sqrt{cx^3}+2x}x^3 + bx.$

29. $y(x) = \lg 3{,}8x - 4{,}6x + 2{,}1,$ $z(x) = 3a\tan x^3 - \sqrt{3x^2 + bx} - c.$

30. $y(x) = 0{,}17\, e^{-0{,}86x} - 1{,}1x,$ $z(x) = \frac{x^3-3ax^2}{\sin 2x^3} + \sqrt{bx - c}.$