

Міністерство освіти і науки України
Національний університет «Львівська політехніка»
кафедра інформаційних систем та мереж

Григорович Віктор

Об'єктно-орієнтоване програмування

Віртуальні функції. Поліморфізм

Навчальний посібник

2021

Григорович Віктор Геннадійович

Об'єктно-орієнтоване програмування. Віртуальні функції. Поліморфізм.

Навчальний посібник.

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «Віртуальні функції. Поліморфізм».

Розглядаються наступні роботи лабораторного практикуму:

Лабораторна робота № 4.1.

Віртуальні функції та абстрактні класи

Лабораторна робота № 4.2.

Інтерфейси (до 4.1)

Лабораторна робота № 4.3.

Віртуальні функції для класів з масивом

Лабораторна робота № 4.4.

Абстрактні класи

Лабораторна робота № 4.5.

Інтерфейси (до 4.4)

Відповідальний за випуск – Григорович В.Г.

Стислий зміст

Вступ.....	11
Тема 4. Віртуальні функції. Поліморфізм.....	12
Стисло та головне про віртуальні функції та поліморфізм.....	12
Віртуальні функції та поліморфізм	12
Поліморфні об'єкти.....	16
Теоретичні відомості.....	17
Віртуальні функції та поліморфізм	17
Абстрактні методи та абстрактні класи. Інтерфейси	54
Поліморфні об'єкти.....	63
Лабораторний практикум	67
Оформлення звіту про виконання лабораторних робіт	67
Лабораторна робота № 4.1. Віртуальні функції та абстрактні класи	69
Лабораторна робота № 4.2. Інтерфейси (до 4.1)	78
Лабораторна робота № 4.3. Віртуальні функції для класів з масивом.....	79
Лабораторна робота № 4.4. Абстрактні класи.....	96
Лабораторна робота № 4.5. Інтерфейси (до 4.4)	126
Питання та завдання для контролю знань	129
Предметний покажчик	134
Література	135

Зміст

Вступ.....	11
Тема 4. Віртуальні функції. Поліморфізм.....	12
Стисло та головне про віртуальні функції та поліморфізм.....	12
Віртуальні функції та поліморфізм	12
Раннє зв'язування та звичайні методи	12
Пізнє зв'язування та віртуальні методи	12
Необхідність віртуальних методів. Поліморфні методи	12
Таблиця віртуальних методів.....	13
Визначення віртуальних методів	13
Правила оголошення та використання віртуальних методів	13
Розмір класу з віртуальними методами.....	15
Віртуальні методи в конструкторах та деструкторах	15
Поліморфні об'єкти.....	16
Поняття поліморфних об'єктів	16
Визначення поліморфного об'єкту – оголошення та створення	16
Теоретичні відомості.....	17
Віртуальні функції та поліморфізм	17
Раннє зв'язування та звичайні методи	17
Недоліки раннього зв'язування – неможливість узагальнювати опис дій над об'єктами.....	23
Реалізація раннього зв'язування	24
Пізнє зв'язування та віртуальні методи	25
Необхідність віртуальних методів. Поліморфні методи	27
Таблиця віртуальних методів.....	27
Реалізація пізнього зв'язування	28
Визначення віртуальних методів	29
Правила оголошення та використання віртуальних методів	29
Перевизначення та перевантаження віртуальних методів	38
Перевантаження віртуальних методів в класі	38
Перевизначення віртуального метода в похідному класі.....	39
Невіртуальний виклик віртуального методу	48
Розмір класу з віртуальними методами.....	48
Віртуальні методи в конструкторах та деструкторах	49
Віртуальні деструктори	49

Чисті віртуальні деструктори.....	51
Віртуалізація зовнішніх функцій.....	52
Абстрактні методи та абстрактні класи. Інтерфейси.....	54
Абстрактні методи – чисті віртуальні функції.....	56
Абстрактні класи.....	57
Ключове слово abstract (MS C++).....	59
Інтерфейси.....	60
Поліморфні об'єкти.....	63
Поняття поліморфних об'єктів.....	63
Визначення поліморфного об'єкту – оголошення та створення.....	63
Поліморфні об'єкти – динамічні змінні.....	63
Поліморфні об'єкти – параметри-посилання.....	64
RTTI.....	64
Визначення справжнього типу поліморфного об'єкту.....	65
Клас type_info	65
Операція typeid()	65
Лабораторний практикум.....	67
Оформлення звіту про виконання лабораторних робіт.....	67
Вимоги до оформлення звіту про виконання лабораторних робіт №№ 4.1–4.5.....	67
Зразок оформлення звіту про виконання лабораторних робіт №№ 4.1–4.5.....	68
Лабораторна робота № 4.1. Віртуальні функції та абстрактні класи.....	69
Мета роботи.....	69
Питання, які необхідно вивчити та пояснити на захисті.....	69
Варіанти завдань.....	69
Варіант 1.....	69
Варіант 2.....	69
Варіант 3.....	70
Варіант 4.....	70
Варіант 5*.....	70
Варіант 6.....	70
Варіант 7.....	71
Варіант 8.....	71
Варіант 9.....	71
Варіант 10.....	71
Варіант 11.....	71

Варіант 12.....	71
Варіант 13.....	72
Варіант 14.....	72
Варіант 15.....	72
Варіант 16.....	72
Варіант 17*.....	72
Варіант 18.....	73
Варіант 19.....	73
Варіант 20*.....	73
Варіант 21.....	73
Варіант 22.....	73
Варіант 23.....	74
Варіант 24.....	74
Варіант 25.....	74
Варіант 26.....	74
Варіант 27.....	74
Варіант 28.....	74
Варіант 29.....	75
Варіант 30*.....	75
Варіант 31.....	75
Варіант 32.....	75
Варіант 33.....	76
Варіант 34.....	76
Варіант 35.....	76
Варіант 36.....	76
Варіант 37.....	76
Варіант 38.....	76
Варіант 39.....	77
Варіант 40.....	77
Лабораторна робота № 4.2. Інтерфейси (до 4.1)	78
Мета роботи	78
Питання, які необхідно вивчити та пояснити на захисті.....	78
Варіанти завдань	78
Лабораторна робота № 4.3. Віртуальні функції для класів з масивом.....	79
Мета роботи	79

Питання, які необхідно вивчити та пояснити на захисті.....	79
Варіанти завдань	79
Варіант 1.....	80
Варіант 2.....	80
Варіант 3.....	80
Варіант 4.....	81
Варіант 5.....	81
Варіант 6.....	81
Варіант 7.....	82
Варіант 8.....	82
Варіант 9.....	83
Варіант 10.....	83
Варіант 11.....	83
Варіант 12.....	84
Варіант 13.....	84
Варіант 14.....	84
Варіант 15.....	85
Варіант 16.....	85
Варіант 17.....	86
Варіант 18.....	86
Варіант 19.....	86
Варіант 20.....	87
Варіант 21.....	87
Варіант 22.....	87
Варіант 23.....	88
Варіант 24.....	88
Варіант 25.....	89
Варіант 26.....	89
Варіант 27.....	89
Варіант 28.....	90
Варіант 29.....	90
Варіант 30.....	90
Варіант 31.....	91
Варіант 32.....	91
Варіант 33.....	92

Варіант 34.....	92
Варіант 35.....	92
Варіант 36.....	93
Варіант 37.....	93
Варіант 38.....	94
Варіант 39.....	94
Варіант 40.....	94
Лабораторна робота № 4.4. Абстрактні класи.....	96
Мета роботи	96
Питання, які необхідно вивчити та пояснити на захисті.....	96
Приклад розв'язання завдання 1	96
Завдання	96
Розв'язок	96
Приклад розв'язання завдання 2.....	98
Завдання	98
Розв'язок за допомогою методу add()	99
Розв'язок за допомогою операції +	100
Варіанти завдань	101
Варіант 1.....	101
Варіант 2.*	101
Варіант 3.*	101
Варіант 4.....	102
Варіант 5.....	102
Варіант 6.*	102
Пояснення FuzzyNumber	103
Варіант 7.....	104
Варіант 8.....	105
Варіант 9.*	105
Варіант 10.*	106
Варіант 11.....	107
Варіант 12.*	107
Варіант 13.**	109
Варіант 14.*	109
Варіант 15.....	110
Варіант 16.....	110

Варіант 17.*	110
Пояснення FuzzyNumber	111
Варіант 18.**	112
Варіант 19.**	113
Варіант 20.**	113
Варіант 21.....	113
Варіант 22.*	113
Варіант 23.*	113
Варіант 24.....	114
Варіант 25.....	114
Варіант 26.*	115
Пояснення FuzzyNumber	115
Варіант 27.....	117
Варіант 28.....	117
Варіант 29.*	117
Варіант 30.*	118
Варіант 31.....	120
Варіант 32.*	120
Варіант 33.**	121
Варіант 34.*	121
Варіант 35.....	122
Варіант 36.....	122
Варіант 37.*	123
Пояснення FuzzyNumber	123
Варіант 38.**	125
Варіант 39.**	125
Варіант 40.**	125
Лабораторна робота № 4.5. Інтерфейси (до 4.4)	126
Мета роботи	126
Питання, які необхідно вивчити та пояснити на захисті.....	126
Варіанти завдань	126
Приклад розв'язання завдання.....	126
Завдання	126
Розв'язок	126
Питання та завдання для контролю знань	129

Предметний покажчик	134
Література	135

Вступ

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «Віртуальні функції. Поліморфізм».

Тема 4. Віртуальні функції. Поліморфізм

Стисло та головне про віртуальні функції та поліморфізм

Поліморфізм можна поділити на 2 види:

- *статичний поліморфізм*, який представлений перевантаженням функцій (на одному рівні можна описати багато однойменних функцій, – головне, щоб вони відрізнялися набором параметрів) та шаблонами функцій – ці теми розглядали раніше;
- *динамічний поліморфізм*, який власне реалізований за допомогою віртуальних функцій і який буде детально розглядатися в цьому розділі.

Віртуальні функції та поліморфізм

Раннє зв'язування та звичайні методи

Зв'язування – це співставлення виклику функції та її тіла.

Для звичайних методів виконується *раннє зв'язування (early binding)*. Це означає, що зв'язок між іменем методу в команді його виклику та кодом цього методу встановлюється на етапі компіляції. Тобто, вся необхідна інформація для того, щоб визначити, який саме метод буде викликаний, стає відомою на етапі компіляції – ще до запуску програми на виконання.

Пізнє зв'язування та віртуальні методи

Необхідність віртуальних методів. Поліморфні методи

Віртуальні методи необхідні для реалізації узагальнених дій – коли один і той же метод виконує різні дії залежно від того, який об'єкт викликав цей метод. За допомогою віртуальних методів вдалося розірвати жорсткий зв'язок між викликом методу і відповідним кодом!

Віртуальні методи реалізують *пізнє зв'язування (late binding)*. Це означає, що виклики віртуальних методів зв'язуються з відповідним кодом не на етапі компіляції, а при виконанні програми.

Метод, який містить виклики віртуальних методів і завдяки цьому реалізує узагальнену дію, називається *поліморфним методом*. Клас, який містить поліморфні методи, або – що те ж саме – віртуальні методи, називається *поліморфним класом*.

Поліморфний метод, визначений в базовому класі, звертається до методів, які визначені в похідному класі. Таке звертання «вниз» по ієрархії успадковування виглядає як диво. «Містика» полягає в тому, що коли ми визначаємо базовий клас, ми не вказуємо і система не знає, скільки буде похідних класів, як вони будуть називатися, скільки в них буде методів та які це будуть методи. Однак, такий клас вже може звертатися «вниз» по ієрархії успадковування. Звертання «вгору» по ієрархії успадковування нікого не дивує, адже класи-нащадки зберігають увесь контент своїх предків – це і є «фішка» успадковування. «Фішкою» динамічного поліморфізму є звертання поліморфних методів «вниз» по ієрархії класів.

Таблиця віртуальних методів

Якщо клас містить хоча би один віртуальний метод, то для класу створюється *таблиця віртуальних методів* VMT (одна на весь клас, не залежно від кількості об'єктів цього класу). Ця таблиця містить адреси віртуальних методів. Адреси методів містяться в VMT в порядку їх оголошення в класах.

Кожний об'єкт такого класу містить приховане поле – *посилання* на VMT, яке називається `vp`tr (**v**irtual **m**ethod **t**able **p**ointer). Це поле заповнюється конструктором при створенні об'єкта. Таким чином, конструктор встановлює зв'язок між об'єктом та VMT відповідного класу.

Для ієрархії успадковування – адреса кожного віртуального метода має одне і те ж зміщення для кожного класу в межах ієрархії. Це означає, що адреса віртуального методу у VMT похідного класу займе ту ж саму позицію, що і адреса однойменного віртуального метода (з тим самим набором параметрів) у VMT базового класу.

Визначення віртуальних методів

Для визначення *віртуального метода* перед його заголовком слід вказати ключове слово `virtual`. Наприклад:

```
virtual void Show();
```

Правила оголошення та використання віртуальних методів

Правило 1. Віртуальна функція може бути лише методом класу. Тому використовуємо термін «віртуальний метод».

Правило 2. Будь-яку операцію, яку можна перевантажувати, можна зробити віртуальною. Наприклад, операції присвоєння, інкременту, приведення типу, індексування тощо.

Правило 3. Віртуальний метод може бути константним.

Правило 4. Віртуальність методів успадковується (див. правила 5 та 6).

Правило 5. Якщо в базовому класі деякий метод визначено як віртуальний, то метод, визначений в похідному класі з *тим самим іменем і набором параметрів*, автоматично стає віртуальним. Тоді ключове слово `virtual` для такого метода можна не вказувати (хоча і рекомендується для більшої ясності коду). Однойменний метод з *іншим набором параметрів* в похідному класі буде звичайним (див. Правило 9).

Правило 6. Віртуальні методи *успадковуються*. Їх слід перевизначати в похідному класі лише тоді, коли необхідно задати інші дії порівняно з однойменними методами базового класу.

Правило 7. Конструктори не можуть бути віртуальними.

Правило 8. Деструктори можуть (а часто – мають) бути віртуальними. Це гарантує коректне звільнення пам'яті для поліморфних об'єктів (див. параграф Поліморфні об'єкти).

Правило 9. Якщо віртуальний метод перевизначений в похідному класі, то об'єкти цього класу можуть отримати доступ до методу базового класу за допомогою операції доступу ::

Таблиця віртуальних методів кожного класу містить адреси всіх віртуальних методів цього класу, – як власних, так і успадкованих від батьківських класів (тобто, таблиця віртуальних методів – це масив вказівників на відповідні функції).

Таблиць віртуальних методів буде стільки, скільки є класів, що містять віртуальні методи – по одній таблиці на клас. Кожний об'єкт містить саме вказівник на таблицю віртуальних методів, а не саму таблицю!

Правило 10. Віртуальний метод не можна оголошувати з модифікатором `static`. Тобто, статичні методи не можуть бути віртуальними, а віртуальні – не можуть бути статичними.

Правило 11. Дружня функція не може бути віртуальною (бо віртуальним може бути лише метод класу):

Правило 12. Віртуальний метод можна оголосити дружнім:

Правило 13. Якщо в класі є оголошення віртуального методу, потрібно або визначити його тіло, або задати його як *чистий віртуальний*.

Розмір класу з віртуальними методами

При наявності хоча б одного віртуального методу розмір класу (тобто, розмір кожного об'єкту цього класу) без полів дорівнює 4 байти (на платформі Intel) і не залежить від кількості віртуальних методів.

Якщо в класі є власні чи успадковані віртуальні методи, компілятор створює таблицю віртуальних методів VMT. В цій таблиці для кожного віртуального метода міститься вказівник на нього. Адреса цієї таблиці і записується в класі (фактично, в будь-якому об'єкті цього класу).

Таким чином, при використанні віртуальних методів витрачається додаткова пам'ять та додатковий час. По-перше, при створенні об'єкта поле вказівника на таблицю VMT слід ініціалізувати – це робить конструктор. По-друге, виклики віртуальних методів відбуваються опосередковано – через вказівник на таблицю VMT.

Проте динамічний поліморфізм – настільки потужна технологія, що розробники java зробили всі методи віртуальними за умовчанням. В C++ ми можемо керувати віртуальністю.

Віртуальні методи в конструкторах та деструкторах

Всередині конструкторів та деструкторів динамічне зв'язування не працює, хоча віртуальні методи можна викликати. Зазвичай віртуальні методи викликають за допомогою посилання чи вказівника на базовий клас, проте в конструкторах та деструкторах завжди викликається «рідний» метод.

В конструкторах це зумовлено тим, що конструктор нічого не знає про похідні класи. Під час виконання конструктора вже були створені об'єкти, проініціалізовані успадковані поля (конструкторами базових класів), та ініціалізуються власні поля. Про конструктори похідних класів та поля, які з'являються в похідних класах. – ще нічого не відомо. Цей конструктор міг сам бути викликаний із конструктора похідного класу, щоб створити і проініціалізувати об'єкт свого класу. Конструктор не знає, хто його викликав. Якби виклик міг бути віртуальним, то були б можливі ситуації використання неініціалізованих змінних похідних класів.

В деструкторах ситуація дещо інша – віртуальний виклик небезпечний тим, що тоді можливий виклик методу вже знищеного об'єкта. Тому навіть у віртуальному деструкторі завжди викликається «рідний» метод.

Поліморфні об'єкти

Поняття поліморфних об'єктів

Поліморфний об'єкт – це той об'єкт, справжній тип якого відрізняється від оголошеного.

Точніше, для того щоб об'єкт був поліморфним, необхідно дотримання наступних умов:

- 1) Клас поліморфного об'єкту має містити хоча б один віртуальний метод.
- 2) Поліморфний об'єкт необхідно оголошувати через вказівник чи посилання на базовий клас.
- 3) Цьому вказівнику чи посиланню слід присвоїти вказівник чи посилання на об'єкт похідного класу.

Визначення поліморфного об'єкту – оголошення та створення

Поліморфізм об'єктів буває двох видів:

- 1) поліморфні об'єкти – динамічні змінні;
- 2) поліморфні об'єкти – параметри-посилання.

Теоретичні відомості

Віртуальні функції та поліморфізм

Поліморфізм – це той третій «жит» (перші два – це інкапсуляція та успадковування), на якому тримається об'єктно-орієнтоване програмування.

Поліморфізм означає «багато форм», – характеризує «гнучкість» програмного коду.

Поліморфізм – це можливість об'єктів одного типу мати різну реалізацію; і найцікавіше – можливість об'єкта базового класу використовувати методи похідного класу, який ще не існує на момент створення базового.

Віртуальні функції (що реалізують поліморфізм) – одна з найбільш цікавих та потужних концепцій C++, яка (якщо не розуміти механізму її реалізації) зазвичай виглядає містикою.

Поліморфізм можна поділити на 2 види:

- *статичний поліморфізм*, який представлений перевантаженням функцій (на одному рівні можна описати багато однойменних функцій, – головне, щоб вони відрізнялися набором параметрів) та шаблонами функцій – ці теми розглядали раніше;
- *динамічний поліморфізм*, який власне реалізований за допомогою віртуальних функцій і який буде детально розглядатися в цьому розділі.

Раннє зв'язування та звичайні методи

Зв'язування – це співставлення виклику функції та її тіла.

Для звичайних методів виконується *раннє зв'язування (early binding)*. Це означає, що зв'язок між іменем методу в команді його виклику та кодом цього методу встановлюється на етапі компіляції. Тобто, вся необхідна інформація для того, щоб визначити, який саме метод буде викликаний, стає відомою на етапі компіляції – ще до запуску програми на виконання.

Розглянемо приклад – клас Точка:

```
// Приклад 1. «Точка»
#include <iostream>
using namespace std;

class Point
{
    int x, y;
    bool visible;

public:
    Point(int x=0, int y=0, bool visible=true) // конструктор за умовчанням
        : x(x), y(y), visible(visible)       // та ініціалізації
    {}                                         // ініціалізуємо поля
    int getX() { return x; }
```

```

int getY() { return y; }

void setX(int x) { this->x = x; }
void setY(int y) { this->y = y; }

void setVisible(bool visible)
{
    this->visible = visible;
}

void Show() // прорисовка точки
{
    setVisible(true);
    cout << "Point is visible." << endl;
}

void Hide()
{
    setVisible(false);
    cout << "Point is invisible." << endl;
}

void Move(int x, int y); // переміщення точки
};

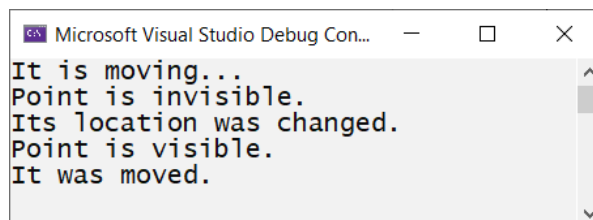
void Point::Move(int x, int y) // переміщення точки
{
    cout << "It is moving..." << endl;
    Hide(); // погасили точку
    setX(x); setY(y); // змінили координати
    cout << "Its location was changed." << endl;
    Show(); // прорисували точку
    cout << "It was moved." << endl << endl;
}

int main()
{
    Point p(100, 100);
    p.Move(200, 200);

    return 0;
}

```

Виконання приводить до очікуваного результату:



```

Microsoft Visual Studio Debug Con...
It is moving...
Point is invisible.
Its location was changed.
Point is visible.
It was moved.

```

Тепер утворимо похідний клас Коло (новий код виділено сірим фоном):

```

// Приклад 2. «Точка» та «Коло». Коло містить два екземпляри методу Move()
#include <iostream>
using namespace std;

class Point
{
    int x, y;
    bool visible;

```

```

public:
    Point(int x=0, int y=0, bool visible=true) // конструктор за умовчанням
        : x(x), y(y), visible(visible)       // та ініціалізації
    {}                                         // ініціалізуємо поля

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }

    void setVisible(bool visible)
    {
        this->visible = visible;
    }

    void Show() // прорисовка точки
    {
        setVisible(true);
        cout << "Point is visible." << endl;
    }

    void Hide()
    {
        setVisible(false);
        cout << "Point is invisible." << endl;
    }

    void Move(int x, int y); // переміщення точки
};

void Point::Move(int x, int y) // переміщення точки
{
    cout << "It is moving..." << endl;
    Hide(); // погасили точку
    setX(x); setY(y); // змінили координати
    cout << "Its location was changed." << endl;
    Show(); // прорисували точку
    cout << "It was moved." << endl << endl;
}

```

```

class Circle
    : public Point
{
    unsigned int R;
public:
    Circle(int x=0, int y=0, bool visible=true, // конструктор за умовчанням
            unsigned int R=0)                  // та ініціалізації
        : Point(x, y, visible), R(R)           // ініціалізуємо поля, спочатку
    {}                                           // викликали конструктор Point

    void Show() // свій алгоритм прорисовки
    {           // кола
        setVisible(true);
        cout << "Circle is visible." << endl;
    }
}

```

```

void Hide()
{
    setVisible(false);
    cout << "Circle is invisible." << endl;
}

void Move(int x, int y);           // переміщення кола
                                   // - власний метод

void Circle::Move(int x, int y)    // переміщення кола
{
    cout << "It is moving..." << endl;
    Hide();                       // погасили (коло)
    setX(x); setY(y);             // змінили координати
    cout << "Its location was changed." << endl;
    Show();                       // прорисували (коло)
    cout << "It was moved." << endl << endl;
}

int main()
{
    Circle c(100, 100);
    c.Move(200, 200);

    return 0;
}

```

Результат – знову очікуваний:

```

Microsoft Visual Studio Debug Con...
It is moving...
Circle is invisible.
Its location was changed.
Circle is visible.
It was moved.

```

Проаналізуємо наведений код. Методи `Circle::Show()` та `Circle::Hide()` мають бути перевизначені в класі `Circle`, бо алгоритм прорисовки кола відрізняється від алгоритму прорисовки точки. Порівняємо методи `Point::Move()` та `Circle::Move()`:

```

void Point::Move(int x, int y)     // переміщення точки
{
    cout << "It is moving..." << endl;
    Hide();                       // погасили точку
    setX(x); setY(y);             // змінили координати
    cout << "Its location was changed." << endl;
    Show();                       // прорисували точку
    cout << "It was moved." << endl << endl;
}

void Circle::Move(int x, int y)    // переміщення кола
{
    cout << "It is moving..." << endl;
    Hide();                       // погасили (коло)
    setX(x); setY(y);             // змінили координати
    cout << "Its location was changed." << endl;
    Show();                       // прорисували (коло)
    cout << "It was moved." << endl << endl;
}

```

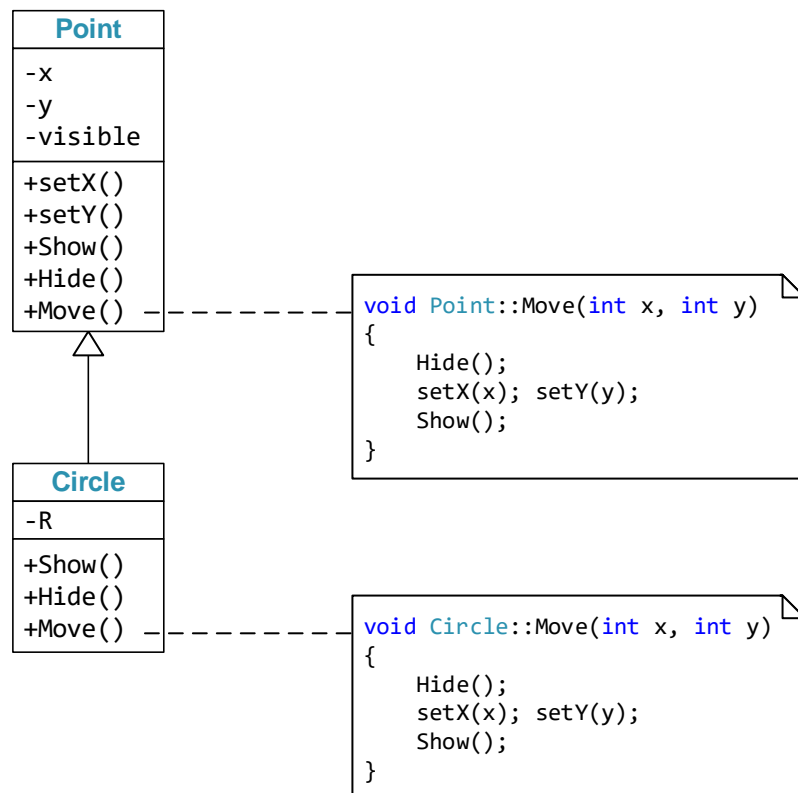
– видно, що ці методи нічим не відрізняються; тіло одного методу точно збігається з тілом іншого.

Основні команди метода `Move()`:

```
Hide();           // погасили
setX(x); setY(y); // змінили координати
Show();           // прорисували
```

І це не дивно, адже алгоритм переміщення кола – такий самий, як і алгоритм переміщення точки: погасити, змінити координати, прорисувати (в нових координатах).

Виникає закономірне питання: навіщо в класах, пов'язаних відношенням успадковування, – `Point` (предок) та `Circle` (нащадок) мати два методи, код яких повністю однаковий? Адже при успадковуванні клас-нащадок має всі поля і всі методи класу-предка. Це означає, що клас `Circle` має два однакових методи `Move()`: перший – успадкований від класу `Point`, і другий – власний:



Виглядає, що правильно буде метод `Move()` залишити лише в класі `Point`, з класу `Circle` вилучити власний метод `Move()` – буде використовуватися успадкований метод, такий підхід буде вірним з точки зору повторного використання коду:

```
// Приклад 3. «Точка» та «Коло». Коло не містить власного методу Move()
#include <iostream>
using namespace std;

class Point
{
    int x, y;
    bool visible;
```

```

public:
    Point(int x=0, int y=0, bool visible=true) // конструктор за умовчанням
        : x(x), y(y), visible(visible) // та ініціалізації
    {} // ініціалізуємо поля

    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }

    void setVisible(bool visible)
    {
        this->visible = visible;
    }

    void Show() // прорисовка точки
    {
        setVisible(true);
        cout << "Point is visible." << endl;
    }

    void Hide()
    {
        setVisible(false);
        cout << "Point is invisible." << endl;
    }

    void Move(int x, int y); // переміщення точки
};

void Point::Move(int x, int y) // переміщення точки
{
    cout << "It is moving..." << endl;
    Hide(); // погасили точку
    setX(x); setY(y); // змінили координати
    cout << "Its location was changed." << endl;
    Show(); // прорисували точку
    cout << "It was moved." << endl << endl;
}

class Circle
    : public Point
{
    unsigned int R;

public:
    Circle(int x=0, int y=0, bool visible=true, // конструктор за умовчанням
           unsigned int R=0) // та ініціалізації
        : Point(x, y, visible), R(R) // ініціалізуємо поля, спочатку
    {} // викликали конструктор Point

    void Show() // свій алгоритм прорисовки
    { // кола
        setVisible(true);
        cout << "Circle is visible." << endl;
    }

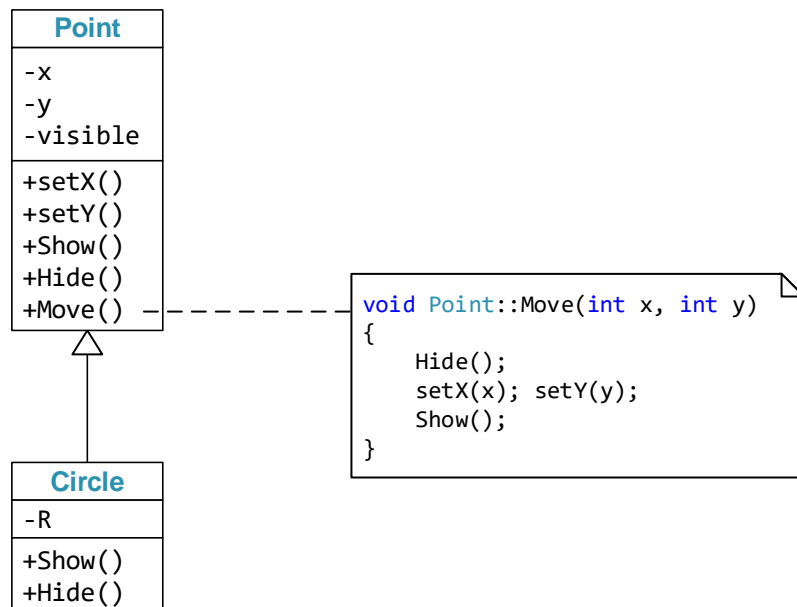
    void Hide()
    {
        setVisible(false);
        cout << "Circle is invisible." << endl;
    }

}; // використовуємо успадкований
    // метод Move()

```

```
int main()
{
    Circle c(100, 100);
    c.Move(200, 200);

    return 0;
}
```



Однак виконанні такої програми ми отримаємо несподіваний результат:

```

Microsoft Visual Studio Debug Con...
It is moving...
Point is invisible.
Its location was changed.
Point is visible.
It was moved.
  
```

— хоча ми давали команду перемістити коло, насправді переміщується точка!

Недоліки раннього зв'язування – неможливість узагальнювати опис дій над об'єктами

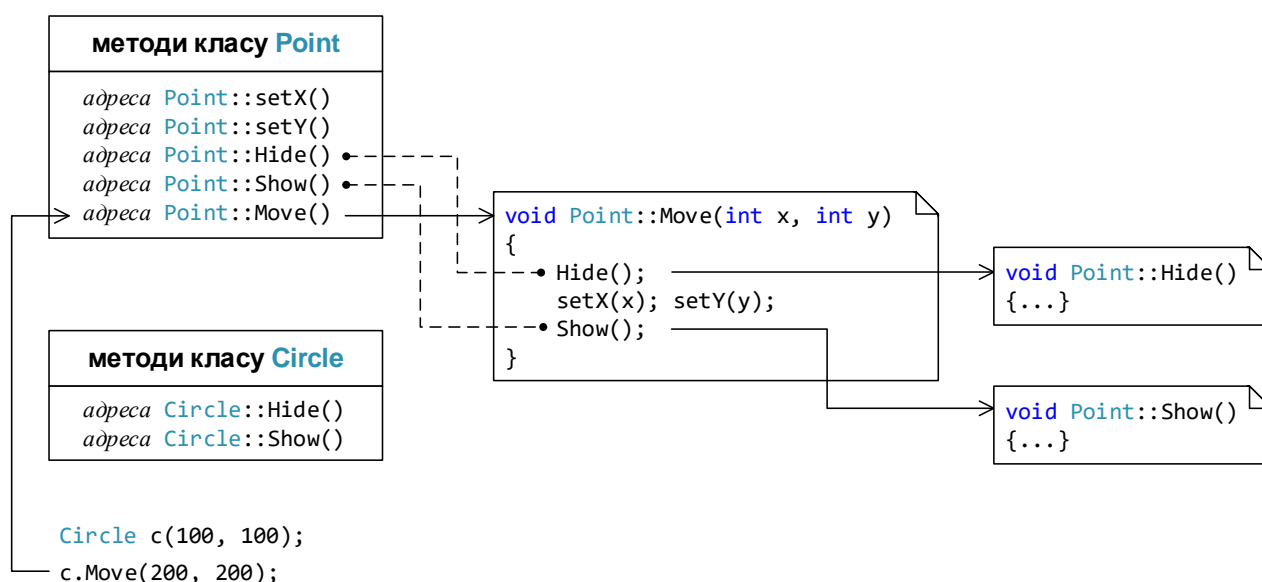
Отримали начебто парадоксальну ситуацію: дотримання правил об'єктно-орієнтованого програмування (спільні характеристики та дії переносимо у батьківський клас, а дочірній клас – використовує ці характеристики та дії, успадковані від батьківського класу) приводить до помилки!

Насправді ніякого парадоксу немає. Помилка полягає в тому, що ми пробуємо використати звичайні методи та раннє зв'язування для реалізації узагальнення опису дій над об'єктами. Причина помилки – раннє зв'язування робить неможливим таке узагальнення.

Реалізація раннього зв'язування

Розглянемо, як реалізовано раннє зв'язування та як відбуваються виклики методів з останнього прикладу.

Коли деякий об'єкт викликає певний метод, починається пошук адреси коду цього метода. Для цього переглядається таблиця методів відповідного класу. Якщо адреса цього метода є в цій таблиці – управління передається відповідному методу і виконується його код. Якщо ж адреси цього метода в таблиці немає – пошук продовжується в таблиці методів батьківського класу. Якщо адреса методу є в таблиці методів батьківського класу – передається управління цьому методу і виконується його код. Якщо адреси метода немає в таблиці методів батьківського класу – пошук продовжується в таблицях базових класів, вгору по ієрархії. Якщо на деякому рівні буде знайдено цей метод – йому буде передано управління. Якщо такого метода немає ніде в усій ієрархії класів – буде помилка.



В нашому прикладі об'єкт `c` викликає метод `Move()`. Пошук цього метода починається з таблиці методів класу `Circle` – бо об'єкт `c` належить до класу `Circle`. В класі `Circle` немає метода `Move()`, тому його пошук продовжується в таблиці методів класу `Point`. Клас `Point` містить метод `Move()`, тому виклик `c.Move(200, 200);` приводить до виконання методу `Point::Move()`. Далі – найцікавіше:

Виконання методу `Point::Move()` починається з команди виклику `Hide()`. Ця команда має знайти підходящий метод `Hide()` та передати йому управління. Раннє зв'язування означає, що при компіляції класу `Point` виклик метода `Hide()` в тілі метода `Point::Move()` зв'язується з методом `Point::Hide()`. Цей зв'язок «жорсткий»: він встановлюється на етапі компіляції і не може бути змінений при виконанні програми.

Аналогічно, виклик методу `Show()` в тілі метода `Point::Move()` приведе до виконання методу `Point::Show()`.

Це означає, що виклик метода `Move()` для об'єкта `с` класу `Circle` приводить до виконання методів `Point::Hide()` та `Point::Show()` – тобто, насправді буде переміщуватися не коло, а точка.

Отже, в наших проблемах винувате раннє зв'язування, яке встановлює на етапі компіляції жорсткий зв'язок між викликом методу і адресою відповідного коду. Нам потрібно розірвати цей жорсткий зв'язок – щоб на етапі виконання можна було зв'язувати виклик методу з потрібним кодом: щоб для об'єкта `с` класу `Circle` метод `Point::Move()` міг викликати методи `Circle::Hide()` та `Circle::Show()`.

Це можна реалізувати за допомогою віртуальних методів.

Пізнє зв'язування та віртуальні методи

Змінімо попередній приклад: оголосимо методи `Hide()` та `Show()` в класах `Point` та `Circle` віртуальними. Для цього перед заголовком метода напишемо ключове слово `virtual` (код, який залишився з попереднього прикладу без змін, виділено сірим фоном):

```
// Приклад 4. «Точка» та «Коло». Коло містить лише успадкований метод Move()
// Методи Hide() та Show() – віртуальні.

#include <iostream>
using namespace std;

class Point
{
    int x, y;
    bool visible;

public:
    Point(int x=0, int y=0, bool visible=true) // конструктор за умовчанням
        : x(x), y(y), visible(visible)       // та ініціалізації
    {}                                         // ініціалізуємо поля

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) { this->x = x; }
    void setY(int y) { this->y = y; }

    void setVisible(bool visible)
    {
        this->visible = visible;
    }

    virtual void Show() // прорисовка точки
    {
        setVisible(true);
        cout << "Point is visible." << endl;
    }

    virtual void Hide()
    {
        setVisible(false);
    }
}
```

```

        cout << "Point is invisible." << endl;
    }

    void Move(int x, int y); // переміщення точки
};

void Point::Move(int x, int y) // переміщення точки
{
    cout << "It is moving..." << endl;
    Hide(); // погасили точку
    setX(x); setY(y); // змінили координати
    cout << "Its location was changed." << endl;
    Show(); // прорисували точку
    cout << "It was moved." << endl << endl;
}

class Circle
: public Point
{
    unsigned int R;

public:
    Circle(int x=0, int y=0, bool visible=true, // конструктор за умовчанням
           unsigned int R=0) // та ініціалізації
        : Point(x, y, visible), R(R) // ініціалізуємо поля, спочатку
    {} // викликали конструктор Point

    virtual void Show() // свій алгоритм прорисовки
    { // кола
        setVisible(true);
        cout << "Circle is visible." << endl;
    }

    virtual void Hide()
    {
        setVisible(false);
        cout << "Circle is invisible." << endl;
    }
};

int main()
{
    Circle c(100, 100);
    c.Move(200, 200);

    return 0;
}

```

Після запуску програми на виконання отримаємо вірний результат:

```

Microsoft Visual Studio Debug Con...
It is moving...
Circle is invisible.
Its location was changed.
Circle is visible.
It was moved.

```

– при виконання методу **Move()** спочатку погасили коло, змінили координати, а потім
– прорисували коло в нових координатах. Тобто, метод **Move()** для об'єкта **c** класу **Circle** тепер приводить до переміщення кола!

Необхідність віртуальних методів. Поліморфні методи

Віртуальні методи необхідні для реалізації узагальнених дій – коли один і той же метод виконує різні дії залежно від того, який об'єкт викликав цей метод. В нашому прикладі метод `Move()` переміщує точку, якщо його викликає об'єкт класу `Point`. Якщо ж його викликає об'єкт класу `Circle`, то метод `Move()` переміщує коло. За допомогою віртуальних методів вдалося розірвати жорсткий зв'язок між викликом методу і відповідним кодом!

Віртуальні методи реалізують *пізнє зв'язування* (*late binding*). Це означає, що виклики віртуальних методів зв'язуються з відповідним кодом не на етапі компіляції, а при виконанні програми. Виклики методів `Hide()` та `Show()` в тілі метода `Move()` для об'єкта класу `Point` приведуть до виконання коду `Point::Hide()` та `Point::Show()`, а для об'єкта класу `Circle` – до виконання коду `Circle::Hide()` та `Circle::Show()`.

Таким чином, метод `Move()` реалізує узагальнену дію «переміщення» для об'єктів всіх класів, похідних від `Point`. Іншими словами, метод `Move()` – «гнучкий»: він має «багато форм» – коли він викликається для об'єкта-точки, то переміщує цю точку; коли ж його викликає об'єкт-коло, то він переміщує це коло. Тому метод, який містить виклики віртуальних методів і завдяки цьому реалізує узагальнену дію, називається *поліморфним методом*. Клас, який містить поліморфні методи, або – що те ж саме – віртуальні методи, називається *поліморфним класом*.

Зверніть увагу: тепер поліморфний метод `Move()`, визначений в базовому класі `Point`, звертається до методів `Show()` та `Hide()`, які визначені в похідному класі `Circle`. Таке звертання «вниз» по ієрархії успадковування виглядає як диво. «Містика» полягає в тому, що коли ми визначаємо базовий клас, ми не вказуємо і система не знає, скільки буде похідних класів, як вони будуть називатися, скільки в них буде методів та які це будуть методи. Однак, такий клас вже може звертатися «вниз» по ієрархії успадковування. Звертання «вгору» по ієрархії успадковування нікого не дивує, адже класи-нащадки зберігають увесь контент своїх предків – це і є «фішка» успадковування. «Фішкою» динамічного поліморфізму є звертання поліморфних методів «вниз» по ієрархії класів.

Розглянемо, як реалізоване пізнє зв'язування. Ключовим елементом для цього є *таблиця віртуальних методів* (Virtual Method Table, VMT або, як ще позначають: VTABLE).

Таблиця віртуальних методів

Якщо клас містить хоча би один віртуальний метод, то для класу створюється *таблиця віртуальних методів* VMT (одна на весь клас, не залежно від кількості об'єктів цього класу). Ця таблиця містить адреси віртуальних методів. Адреси методів містяться в

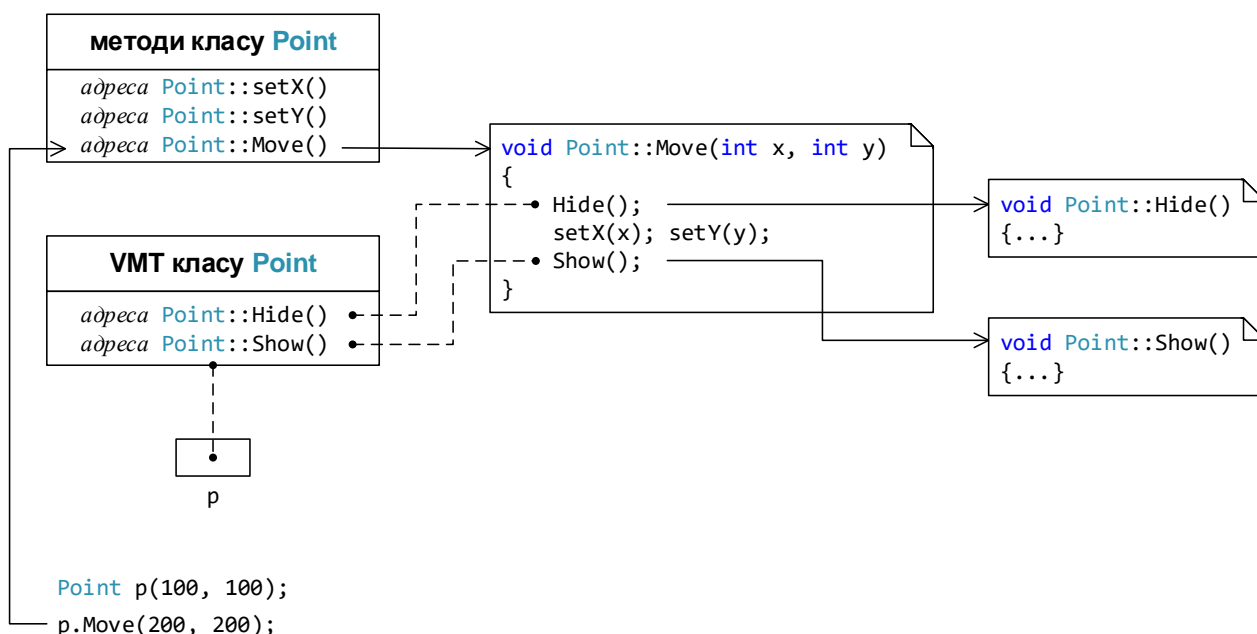
VMТ в порядку їх оголошення в класах.

Кожний об'єкт такого класу містить приховане поле – *посилання* на VMТ, яке називається *vpтр* (*virtual method table pointer*). Це поле заповнюється конструктором при створенні об'єкта. Таким чином, конструктор встановлює зв'язок між об'єктом та VMТ відповідного класу.

Для ієрархії успадковування – адреса кожного віртуального метода має одне і те ж зміщення для кожного класу в межах ієрархії. Це означає, що адреси віртуальних методів *Show()* та *Hide()* у VMТ класу *Circle* займуть ті ж самі позиції, що і адреси віртуальних методів *Show()* та *Hide()* у VMТ класу *Point*.

Реалізація пізнього зв'язування

Розглянемо, як працюють віртуальні методи. Коли ми переміщуємо точку, пошук та виклики методів виконуються в наступному порядку:

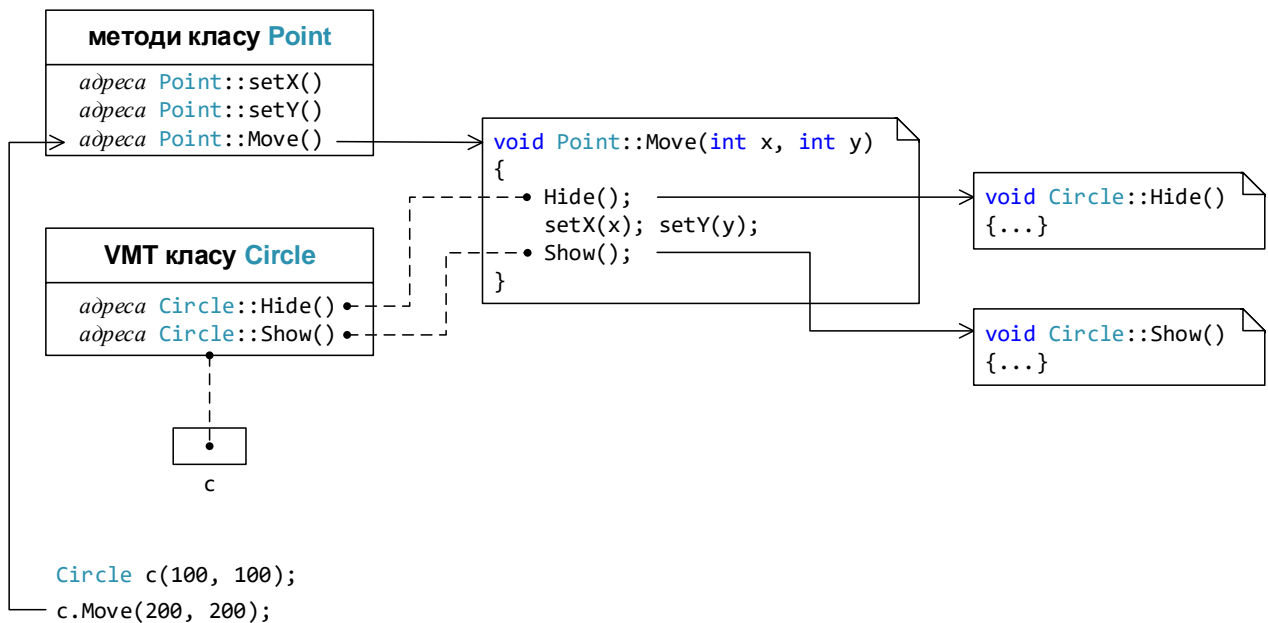


При виклику `p.Move(200, 200);` метода `Move()` об'єктом `p` цей метод шукається серед методів класу `Point` – класу об'єкта `p`. Цей метод є в класі `Point` і йому передається управління.

Далі потрібно викликати метод `Hide()`. Пошук цього метода починається серед звичайних методів класу `Point`. Там його немає. Пошук продовжується у таблиці віртуальних методів, зв'язаній з об'єктом `p`. Там є адреса метода `Point::Hide()`, і йому буде передано управління. Аналогічно виконується виклик метода `Show()` в тілі `Move()`. Таким чином, буде переміщена точка. Поки що нічого нового – те саме виконувалося і при використанні звичайних методів `Show()` та `Hide()`. Тепер виклики відбуваються навіть повільніше – бо після пошуку метода серед звичайних методів потрібно переходити до таблиці віртуальних

методів, адреса якої міститься в прихованому полі `vptr`.

Різниця буде відчутною при переміщенні кола:



Об'єкт `c` класу `Circle` викликає метод `Move()`. Пошук цього метода починається серед звичайних методів класу `Circle`. Там цього метода немає, і пошук продовжується серед успадкованих звичайних методів від батьківського класу `Point`. Метод `Move()` там є і йому передається управління.

Як і для точки, наступною виконується команда виклику метода `Hide()`. Пошук цього методу починається серед звичайних методів класу `Point`. Там цього методу немає. Пошук продовжується в таблиці віртуальних методів, зв'язаній з об'єктом `c`. Таблиця містить адресу метода `Circle::Hide()` і йому передається управління. Аналогічно, потім управління отримає метод `Circle::Show()`. Це і приведе до переміщення кола!

Визначення віртуальних методів

Для визначення *віртуального метода* перед його заголовком слід вказати ключове слово `virtual`. Наприклад:

```
virtual void Show();
```

Правила оголошення та використання віртуальних методів

Правило 1. Віртуальна функція може бути лише методом класу. Тому використовуємо термін «віртуальний метод».

Правило 2. Будь-яку операцію, яку можна перевантажувати, можна зробити віртуальною. Наприклад, операції присвоєння, інкременту, приведення типу, індексування тощо.

Правило 3. Віртуальний метод може бути константним.

Правило 4. Віртуальність методів успадковується (див. правила 5 та 6).

Правило 5. Якщо в базовому класі деякий метод визначено як віртуальний, то метод, визначений в похідному класі з *тим самим іменем і набором параметрів*, автоматично стає віртуальним. Тоді ключове слово **virtual** для такого метода можна не вказувати (хоча і рекомендується для більшої ясності коду). Однойменний метод з *іншим набором параметрів* в похідному класі буде звичайним (див. Правило 9).

```
class A
{
public:
    virtual void f(int);    // віртуальний метод
};

class B
    : public A
{
public:
    void f(int);            // теж віртуальний метод -
                           // хоча краще явно вказати virtual

    void f(int, double);    // звичайний метод
};
```

Правило 6. Віртуальні методи *успадковуються*. Їх слід перевизначати в похідному класі лише тоді, коли необхідно задати інші дії порівняно з однойменними методами базового класу.

В попередніх прикладах для класу **Circle** слід було визначити свої віртуальні методи **Show()** та **Hide()**, бо алгоритми прорисовки та гасіння кола відрізняються від алгоритмів прорисовки та гасіння точки, – і тому не можна було обійтися методами **Show()** та **Hide()** класу **Point**.

Правило 7. Конструктори не можуть бути віртуальними.

Правило 8. Деструктори можуть (а часто – мусять) бути віртуальними. Це гарантує коректне звільнення пам'яті для поліморфних об'єктів (див. параграф Поліморфні об'єкти).

Правило 9. Якщо віртуальний метод перевизначений в похідному класі, то об'єкти цього класу можуть отримати доступ до методу базового класу за допомогою операції доступу ::

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

class A
{
public:
    virtual void f(void)    // віртуальний метод
    {
        cout << "A::f()" << endl;
    }
};
```

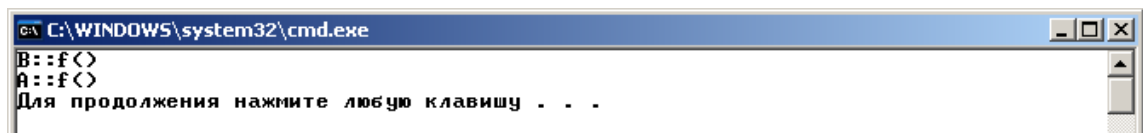
```

class B
    : public A
{
public:
    virtual void f(void)    // перевизначений віртуальний метод
    {
        cout << "B::f()" << endl;
    }
};

int main()
{
    B b;
    b.f();                // виводить: B::f()
    b.A::f();              // використовуємо операцію доступу A::f()
                           // виводить: A::f()

    return 0;
}

```



Тут в першій команді виклику метода $f()$ – $b.f()$; – відбувається звертання до VMT класу B , щоб отримати адресу метода $B::f()$. В другій команді $b.A::f()$; використовуємо пряме звертання до VMT класу A , і завдяки цьому отримуємо адресу та звертаємося до методу $A::f()$.

Розглянемо цей та інші приклади більш детально. Спочатку розберемося, що таке таблиця віртуальних методів і для чого вона потрібна.

Таблиця віртуальних методів кожного класу містить адреси всіх віртуальних методів цього класу, – як власних, так і успадкованих від батьківських класів (тобто, таблиця віртуальних методів – це масив вказівників на відповідні функції).

Таблиць віртуальних методів буде стільки, скільки є класів, що містять віртуальні методи – по одній таблиці на клас. Кожний об'єкт містить саме вказівник на таблицю віртуальних методів, а не саму таблицю!

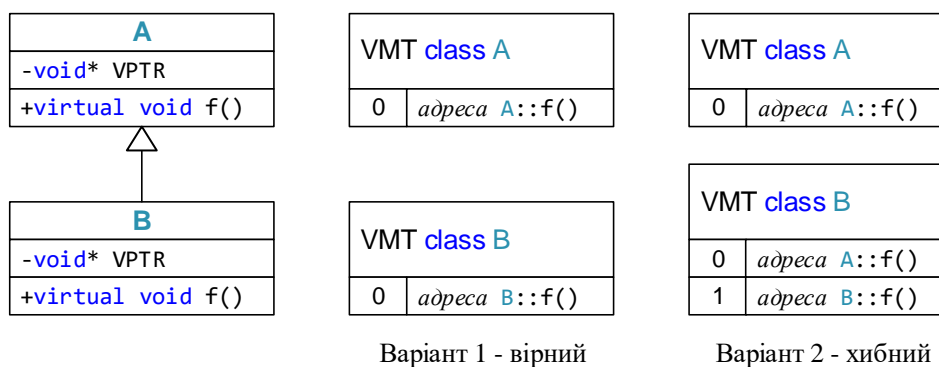
Питання на цю тему часто задають на екзаменах та на співбесідах. Приклади таких «каверзних» питань:

- 1) якщо клас містить таблицю віртуальних методів, то розмір об'єкта цього класу залежить від кількості віртуальних методів, які містяться в цій таблиці, чи не так? (ні, не так! Таблиця віртуальних методів VMT зберігається окремо від об'єктів. Об'єкт отримує додаткове поле – вказівник на VMT. Розмір вказівника не залежить від даних, на які він вказує, тому розмір об'єкта не залежить від розміру VMT.)
- 2) маємо масив вказівників на базовий клас, кожний вказівник вказує на об'єкт одного із похідних класів. Скільки в нас буде таблиць віртуальних методів? (кількість таблиць

віртуальних методів не залежить від кількості об'єктів. Створюється по одній таблиці віртуальних методів на кожний клас, який містить хоча б один віртуальний метод – власний чи успадкований).

Отже, для кожного класу буде створена таблиця віртуальних методів. Кожному віртуальному методу базового класу присвоюється індекс; віртуальні методи нумеруються починаючи від 0 в тому порядку, в якому вони оголошені в базовому класі. За цим індексом буде визначатися адреса віртуального метода в таблиці VMT. При успадковуванні похідний клас отримує свою таблицю віртуальних методів, яка містить як адреси успадкованих, так і адреси власних віртуальних методів.

Якщо деякий віртуальний метод перевизначається в похідному класі (перевизначення методів детально розглядається в наступному параграфі), то в таблиці віртуальних методів цього класу адреса відповідного метода базового класу буде замінена новою адресою – однойменного метода похідного класу:



Для попереднього прикладу саме варіант 1 буде вірним зображенням структури таблиць віртуальних методів.

Якщо в похідний клас добавлено нові віртуальні методи (з іншою сигнатурою, ніж у віртуальних методів базового класу), то VMT похідного класу розширюється. VMT базового класу, очевидно, змінена не буде і залишиться тою ж самою, як і раніше.

Тому через вказівник на базовий клас не можна віртуально викликати методи похідного класу, яких не було в базовому – адже базовий клас нічого про них не знає:

```
class A
{
public:
    virtual void f1() {}
};

class B
    : public A
{
public:
    virtual void f2() {}
};
```



```

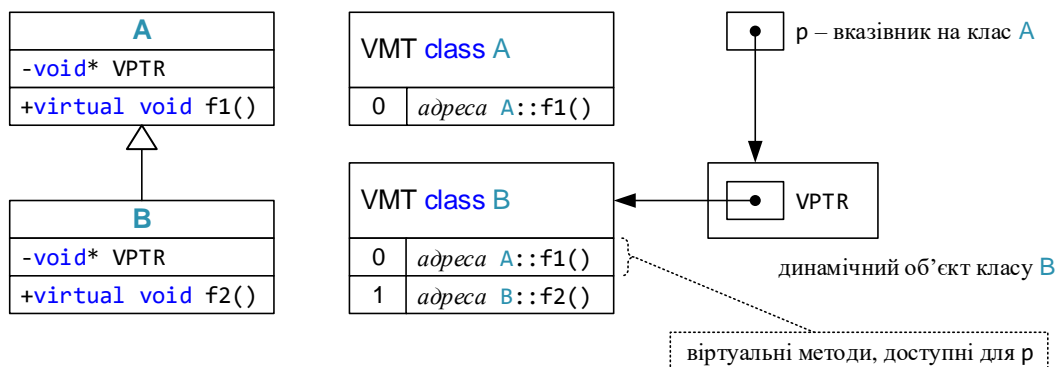
int main()
{
    A* p = new B();

    p->f1(); // виклик A::f1()
    p->f2(); // помилка: VMT класу A не містить адреси f2()
            // error C2039: 'f2' : is not a member of 'A'
            // - 'f2' не є елементом 'A'

    return 0;
}

```

– клас A нічого не знає про метод f2(), тому ми не можемо його викликати через вказівник p, бо вказівнику p доступні лише методи, відомі в класі A – тобто, оголошені в класі A:



Конструктор класу тепер має виконати додаткову дію: ініціалізувати вказівник VPTR адресою відповідної таблиці віртуальних методів. Коли ми створюємо об'єкт похідного класу, спочатку викликається конструктор базового класу, який ініціалізує VPTR адресою «своїх» таблиці віртуальних методів. Потім викликається конструктор похідного класу, який переписує (змінює) це значення.

При виклику віртуального метода через вказівник на базовий клас компілятор має за допомогою вказівника VPTR звернутися до відповідної таблиці віртуальних методів, від неї отримати адресу викликаного метода, а тоді вже здійснювати виклик – передавати управління цьому методу. Код звертання до таблиці віртуальних методів генерує компілятор, тому викликати можна буде лише ті методи, які відомі на етапі компіляції, – тобто, які оголошені в базовому класі.

Правильний виклик метода f2(), визначеного в похідному класі, потребує приведення типу:

```

((B*)p)->f2();
dynamic_cast<B*>(p)->f2();

```

Механізм пізнього зв'язування потребує додаткових затрат процесорного часу (ініціалізація VPTR конструктором, отримання адреси функції при виклику) та пам'яті (додаткове поле VPTR в кожному об'єкті) порівняно з раннім.

Розглянемо наступний приклад:

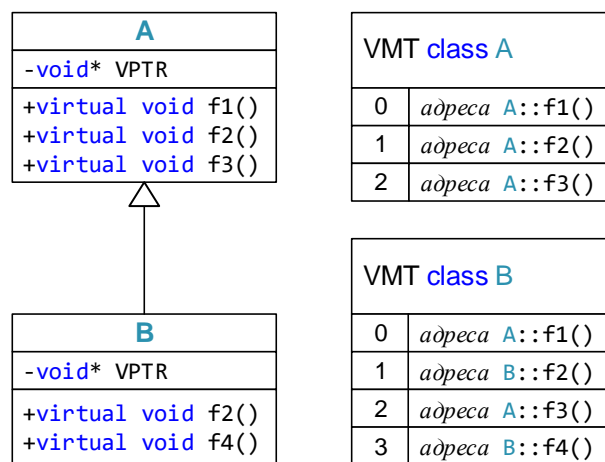
```

class A
{
public:
    virtual void f1() {}
    virtual void f2() {}
    virtual void f3() {}
};

class B
    : public A
{
public:
    virtual void f2() {}
    virtual void f4() {}
};

```

Структура VMT цих класів наступна:



Як бачимо, в таблиці похідного класу адреса другого метода A::f2() була замінена адресою перевизначеного метода B::f2().

Код для перевірки:

```

#include <iostream>
using namespace std;

class A
{
public:
    A() { cout << "A::A()" << endl; }
    virtual ~A() { cout << "A::~~A()" << endl; }
    virtual void f1() { cout << "A::f1()" << endl; }
    virtual void f2() { cout << "A::f2()" << endl; }
    virtual void f3() { cout << "A::f3()" << endl; }
};

class B
    : public A
{
public:
    B() { cout << "B::B()" << endl; }
    virtual ~B() { cout << "B::~~B()" << endl; }
    virtual void f2() { cout << "B::f2()" << endl; }
    virtual void f4() { cout << "B::f4()" << endl; }
};

```

```

int main()
{
    A* p = new B();

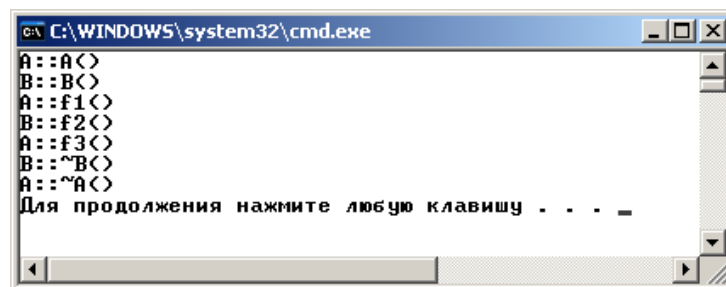
    p->f1(); // виклик A::f1()
    p->f2(); // виклик B::f2()
    p->f3(); // виклик A::f3()
    p->f4(); // помилка: A не містить f4()

    delete p;

    return 0;
}

```

Результат виконання (при закоментованій команді виклику `p->f4();`):

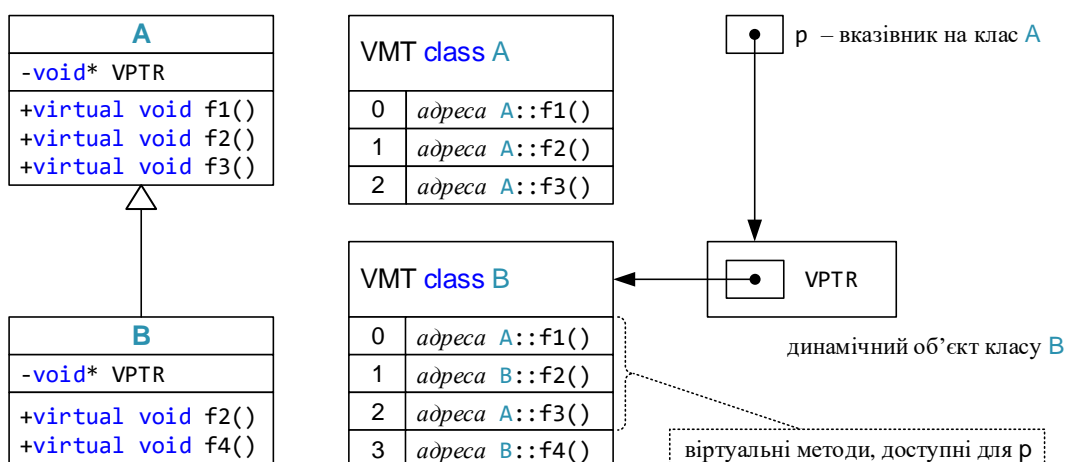


Розглянемо, що відбувається при виконанні цієї програми.

Спочатку оголошується вказівник на об'єкт типу `A`, якому присвоюється адреса динамічного об'єкта типу `B`. При цьому спочатку викликається конструктор `A`, який ініціалізує `VPTR` адресою `VMТ` класу `A`, а потім – конструктор `B`, який перезаписує значення `VPTR` адресою `VMТ` класу `B`.

При виклику `p->f1()`, `p->f2()` та `p->f3()` компілятор генерує код, який через `VPTR` видобуває фактичну адресу функції із таблиці віртуальних методів.

Розглянемо, як це відбувається. Діаграма таблиць віртуальних методів класів `A` та `B` (не вказані віртуальні деструктори):



Не залежно від конкретного типу об'єкта компілятор знає, що адреса метода `f1()` записана в позиції з індексом 0, адреса `f2()` – в позиції з індексом 1 і т.д. (в порядку

оголошення методів). Таким чином, для виклику метода `f3()` компілятор отримує його адресу у вигляді `VPTR+2` – зміщення від початку таблиці віртуальних методів, зберігає цю адресу і підставляє в команду виклику.

З цієї ж причини, виклик `p->f4()` приводить до помилки: хоча фактичний тип динамічного об'єкта – `B`, але коли ми присвоюємо його адресу вказівнику `p`, виконується приведення типу вгору по ієрархії – до типу `A`. VMT класу `A` не містить ніякого методу `f4()`, тому зміщення `VPTR+3` вказувало би на «чужу» пам'ять. Тому, на щастя, такий код навіть не компілюється.

Розглянемо ще один приклад.

Очевидно, що при таким чином реалізованій віртуальності методів в принципі неможливо зробити так, щоб метод був віртуальним лише на поточному рівні ієрархії (див. Правило 5):

```
#include <iostream>

using namespace std;

class A
{
public:
    virtual ~A() {}

    virtual void f1() const { cout << "A::f1()" << endl; }
    virtual void f2() const { cout << "A::f2()" << endl; }
    void f3() const { cout << "A::f3()" << endl; }
};

class B
    : public A
{
public:
    virtual void f1() const { cout << "B::f1()" << endl; }
    void f2() const { cout << "B::f2()" << endl; } // це - теж віртуальний метод
    void f3() const { cout << "B::f3()" << endl; }
};

class C
    : public B
{
public:
    virtual void f1() const { cout << "C::f1()" << endl; }
    virtual void f2() const { cout << "C::f2()" << endl; }
    void f3() const { cout << "V::f3()" << endl; }
};

int main()
{
    cout << "p is B:" << endl;
    A *p = new B();
    p->f1(); // виводить B::f1(), бо метод f1() - віртуальний в класі A
    p->f2(); // виводить B::f2(), бо метод f2() - віртуальний в класі A
    p->f3(); // виводить A::f3(), бо метод f3() - не віртуальний в класі A
    delete p;
}
```

```

cout << endl;
cout << "p is C:" << endl;
p = new C();
p->f1(); // виводить C::f1(), бо метод f1() - віртуальний в класі A
p->f2(); // виводить C::f2(), бо метод f2() - віртуальний в класі A
p->f3(); // виводить A::f3(), бо метод f3() - не віртуальний в класі A
delete p;

return 0;
}

```

Результат:

```

C:\WINDOWS\system32\cmd.exe
p is B:
B::f1()
B::f2()
A::f3()

p is C:
C::f1()
C::f2()
A::f3()
Для продолжения нажмите любую клавишу . . .

```

– віртуальний метод стає віртуальним до кінця ієрархії, а ключове слово `virtual` є «ключовим», коли зустрічається лише вперше, в усіх наступних класах ієрархії (нащадках) це слово несе лише інформативну функцію для зручності програмістів.

Тепер вже має бути зрозумілим, чому віртуальність методів спрацьовує лише при звертанні за адресою об'єкта (через вказівники або через посилання). В команді

```
A *p = new B();
```

виконується приведення типу вгору по ієрархії: `B*` приводиться до типу `A*`, але вказівник містить лише адресу «початку» об'єкта в пам'яті. Якщо ж приведення вгору робити безпосередньо для об'єкта, то він буде «обрізаний» до розміру об'єкта базового класу. Тому логічно, що для виклику методів «через об'єкт» (а не через вказівник чи через посилання) використовується раннє зв'язування – бо компілятор і так «знає» фактичний тип об'єкта.

Правило 10. Віртуальний метод не можна оголошувати з модифікатором `static`. Тобто, статичні методи не можуть бути віртуальними, а віртуальні – не можуть бути статичними.

Правило 11. Дружня функція не може бути віртуальною (бо віртуальним може бути лише метод класу):

```

class A
{
public:
    friend virtual void f(); // помилка!
};

void f() {}

```

Правило 12. Віртуальний метод можна оголосити дружнім:

```
class A
{
public:
    virtual void f() {}
};

class B
{
public:
    friend void A::f();
};
```

Правило 13. Якщо в класі є оголошення віртуального методу, потрібно або визначити його тіло, або задати його як *чистий віртуальний*. (див. параграф Абстрактні методи та абстрактні класи. Інтерфейси. Абстрактні методи – чисті віртуальні функції)

Перевизначення та перевантаження віртуальних методів

Тут розглянемо питання:

- 1) Чи можна перевантажувати віртуальні методи в класі?
- 2) До яких наслідків приведе перевизначення віртуального методу в похідному класі з іншим набором параметрів?
- 3) Чи можна при перевизначенні віртуального методу змінити лише тип результату (або лише константність методу)?
- 4) Чи можна віртуальний метод викликати невіртуально? Якщо можна, то коли і навіщо це потрібно?

Перевантаження віртуальних методів в класі

Перевантаження віртуальних методів, як і будь-яких інших методів, повністю допустиме:

```
class A
{
public:
    virtual void f(void); // віртуальний метод
    virtual void f(int);  // перевантажений віртуальний метод
};
```

Таблиця віртуальних методів класу A міститиме два входи – адреси віртуальних методів `f(void)` та `f(int)`:

VMT class A	
0	адреса <code>A::f(void)</code>
1	адреса <code>A::f(int)</code>

Перевизначення віртуального метода в похідному класі

Перевизначення віртуальних методів також частково розглядалося в попередньому параграфі – «Правила оголошення та використання віртуальних методів», див. пояснення до Правила 9.

Перевизначення з іншим набором параметрів

Припустимо, що ми маємо базовий клас з перевантаженими віртуальними методами:

```
class A
{
public:
    virtual int f() const                // віртуальний метод
    {
        cout << "A::f()" << endl;
        return 0;
    }

    virtual void f(const string &s) const // перевизначений віртуальний метод
    {
        cout << "A::f(string)" << endl;
    }
};
```

Ніяких проблем при трансляції такого класу не виникне. Визначимо похідний клас:

```
class B
    : public A
{
public:
    virtual int f(int i) const          // перевизначений віртуальний метод
    {                                  // з іншим набором параметрів
        cout << "B::f(int)" << endl;
        return 0;
    }
};
```

В класі **B** ми перевизначили віртуальний метод **f()** з іншим набором параметрів. Цей клас також компілюється без проблем. Проте при виклику методів виникають проблеми:

```
int main()                // виклик перевантажених віртуальних методів
{
    A a, *pa;              // об'єкти базового класу
    B b, *pb = &b;         // об'єкти похідного класу

    pa = &b;               // вказівнику pa присвоїли адресу b
                           // - тут потрібна віртуальність

    pa->f();                // виклик базового методу - виводить: A::f()
    pa->f("abc");          // виклик базового методу - виводить: A::f(string)
    pa->f(1);              // спроба викликати похідний метод - помилка!

    return 0;
}
```

В перших двох випадках викликаються методи базового класу. Третій виклик:

```
pa->f(1);                // спроба викликати похідний метод - помилка!
```

– тут начебто викликається похідний метод через вказівник базового класу, проте компілятор намагається викликати базовий метод

```
virtual void f(const string &s) const
```

після чого повідомляє про неможливість перетворення `int` до `string`.

Таким чином, похідний метод

```
virtual int f(int i) const
```

не можна викликати через вказівник базового класу. Похідні методи можна викликати лише через вказівник похідного класу `pb` або за допомогою явного приведення типу вказівника базового класу `pa`:

```
pb->f(1); // виклик похідного методу
          // через вказівник похідного класу

((B*)pa)->f(1); // виклик похідного методу
                // через приведення типу в стилі C
                // вказівника базового класу

dynamic_cast<B*>(pa)->f(1); // також виклик похідного методу
                             // через приведення типу в стилі C++
                             // вказівника базового класу
```

Причина цього – в тому, що хоча після виконання команди

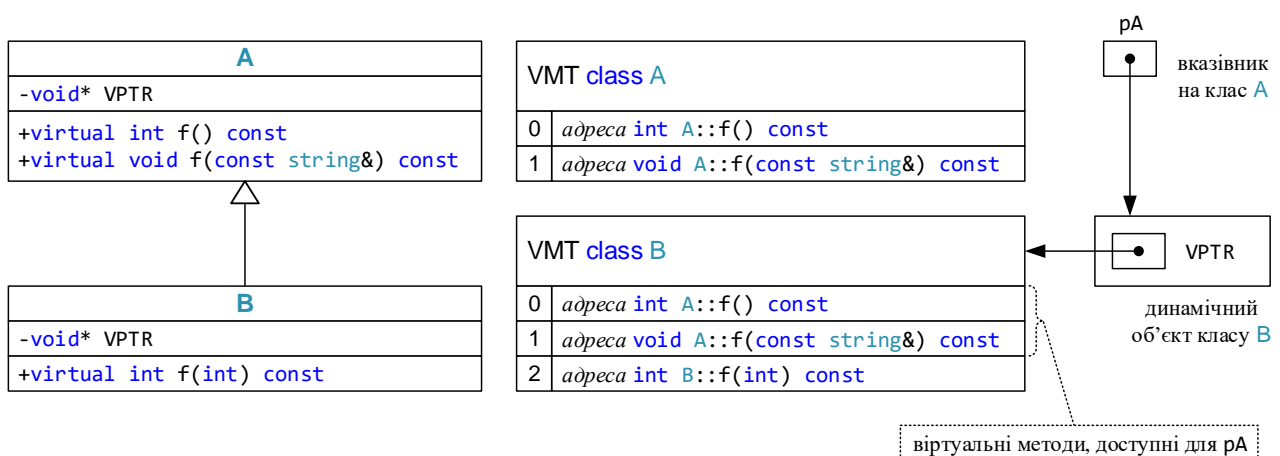
```
pa = &b; // вказівнику pa присвоїли адресу b
```

вказівник базового класу `pa` став налаштований на об'єкт похідного класу `b`, він оголошений як вказівник на базовий клас `A`, а при компіляції класу `A` нічого невідомо про похідний метод

```
virtual int f(int i) const
```

Через вказівник `pa` доступні лише (віртуальні) методи класу `A`, успадковані в класі `B`.

Структура таблиць віртуальних методів класів `A` та `B` – наступна:



Детальніше такі випадки розглядаються в параграфі Поліморфні об'єкти.

Аналогічно, виклики базових методів через вказівник похідного класу не компілюються – це виглядає, начебто похідний клас не успадкував базові методи:

```
pb->f();           // помилка - відсутність аргументу
pb->f("abc");       // помилка - спроба перетворити char[4] до int
b.f();            // помилка - відсутність аргументу
b.f("abc");       // помилка - спроба перетворити char[4] до int
a.f(1);           // помилка - спроба перетворити int до string
```

Таким чином, (як і для звичайних невіртуальних методів) віртуальний метод-нащадок з тим самим іменем, але іншим набором параметрів, просто приховує однойменні методи базового класу.

Причина такого приховування базових віртуальних методів для вказівників похідного класу полягає в тому, що похідний об'єкт зв'язаний з таблицею віртуальних методів похідного класу, і пошук метода в цій таблиці починається з адрес віртуальних методів саме похідного класу (пошук виконується «знизу вгору»).

Перевизначення із зміною константності

Все, сказане в попередньому параграфі, стосується і константності: константний метод вважається іншим порівняно з неконстантним методом з тим самим набором параметрів. Тобто, при перевизначенні віртуального методу можна змінити лише константність:

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

class A
{
public:
    virtual void f() const // віртуальний константний метод
    {
        cout << "const method" << endl;
    }

    virtual void f()      // перевизначений неконстантний віртуальний метод
    {
        cout << "non-const method" << endl;
    }
};

int main()
{
    A a;
    a.f();           // вивід: non-const method

    const A ca;
    ca.f();          // вивід: const method

    return 0;
}
```

Те саме справедливе і для перевизначення із зміною константності в похідному класі:

```

#include <iostream>
using namespace std;

class A
{
public:
    virtual void f() const // віртуальний константний метод
    {
        cout << "const method" << endl;
    }
};

class B
    : public A
{
public:
    virtual void f() // перевизначений неконстантний віртуальний метод
    {
        cout << "non-const method" << endl;
    }
};

int main()
{
    A a, *pa;
    B b, *pb = &b;

    pb->f(); // вивід: non-const method

    pa = &b;
    pa->f(); // вивід: const method

    return 0;
}

```

– через вказівник базового класу викликається метод базового класу.

Перевизначення з тим самим набором параметрів

Розглянемо попередній приклад, але тепер в похідному класі перевизначимо методи базового класу з такими ж прототипами:

```

#include <iostream>
using namespace std;

class A
{
public:
    virtual int f() const // віртуальний метод
    {
        cout << "A::f()" << endl;
        return 0;
    }

    virtual void f(const string &s) const // перевизначений віртуальний метод
    {
        cout << "A::f(string)" << endl;
    }
};

```

```

class B
: public A
{
public:
    virtual int f(int i) const           // перевизначений віртуальний метод
    {                                   // з іншим набором параметрів
        cout << "B::f(int)" << endl;
        return 0;
    }

    virtual int f() const                // перевизначений віртуальний метод
    {                                   // з тим самим набором параметрів
        cout << "B::f()" << endl;      // що і в базовому класі
        return 0;
    }

    virtual void f(const string &s) const // перевизначений віртуальний метод
    {                                   // з тим самим набором параметрів
        cout << "B::f(string)" << endl; // що і в базовому класі
    }
};

int main()                             // виклик перевантажених віртуальних методів
{
    A a, *pa;                           // об'єкти базового класу
    B b, *pb = &b;                       // об'єкти похідного класу

    pa = &b;                             // вказівнику pa присвоїли адресу b
                                         // - тут потрібна віртуальність

    pa->f();                             // виклик похідного методу - виводить: B::f()
    pa->f("abc");                       // виклик похідного методу - виводить: B::f(string)
    pa->f(1);                           // спроба викликати похідний метод - помилка!

    return 0;
}

```

В цьому випадку вірно працюють виклики

```

pa->f();           // виклик похідного методу - виводить: B::f()
pa->f("abc");      // виклик похідного методу - виводить: B::f(string)

```

– виклики успішні, бо ці методи є в базовому класі. При компіляції базового класу стає відомо, що в ньому є методи

```

virtual int f() const
virtual void f(const string &s) const

```

Компілятор знає про ці методи, тому формує команди їх виклику через вказівник на базовий клас `pa`. Оскільки вказівник `pa` пов'язаний з об'єктом похідного класу `b`, а з ним зв'язана VMT похідного класу `B`, то виклики віртуальних методів приводять до виконання методів похідного класу.

Проте, спроба викликати похідний метод, якого не було в базовому класі, приводить до помилки:

```

pa->f(1);          // спроба викликати похідний метод - помилка!

```

– бо метода

```
virtual int f(int i) const
```

немає в базовому класі і компілятор не може згенерувати команди його виклику для вказівника базового класу `pa`. Знову виводиться повідомлення про помилку: неможливо перетворити `int` до `string`.

Отже, через вказівник базового класу не можна викликати нові методи, визначені лише в похідному класі. Якщо ж все-таки це необхідно зробити, слід виконати явне приведення типу:

```
((B*)pa)->f(1);           // приведення типу в стилі C
                          // - виводить: B::f(int)
dynamic_cast<B*>(pa)->f(1); // приведення типу в стилі C++
                          // - виводить: B::f(int)
```

Як бачимо, перевизначення методів базового класу робить визначення похідного класу занадто громіздким. Якщо не хочемо цього робити, а базові методи все-таки потрібно викликати, то можна використати директиву `using`:

```
using A::f; // дозвіл використовувати приховані віртуальні базові методи
```

Тепер визначення похідного класу виглядає так:

```
class B
: public A
{
public:
    virtual int f(int i) const           // перевизначений віртуальний метод
    {                                   // з іншим набором параметрів
        cout << "B::f(int)" << endl;
        return 0;
    }

    using A::f;                         // дозвіл використовувати приховані
                                        // віртуальні базові методи
};
```

Для оголошень із наведеного прикладу результати викликів будуть наступними:

```
pa->f();           // виклик базового методу - виводить: A::f()
pa->f("abc");      // виклик базового методу - виводить: A::f(string)
```

Директива `using` діє на весь клас незалежно від місця її розміщення:

Можна викликати базовий метод із метода похідного класу:

```
virtual int f(int i) const
{
    f();
    return 0;
}
```

Хоча директива `using` записана після методу `f(int)`, ніякі повідомлення про помилки із-за відсутності визначення методу `f()` не виводяться.

Особливості перевизначення віртуальних методів

Без директиви `using` наступний код не працює:

```

#include <iostream>
#include <string>

using namespace std;

class A
{
public:
    virtual int f() const                // віртуальний метод
    {
        cout << "A::f()" << endl;
        return 0;
    }

    virtual void f(const string& s) const // перевизначений віртуальний метод
    {
        cout << "A::f(string)" << endl;
    }
};

class B
    : public A
{
public:
    virtual int f(int i) const           // перевизначений віртуальний метод
                                         // з іншим набором параметрів
    {
        cout << "B::f(int)" << endl;
        return 0;
    }

    // using A::f;                       // дозвіл використовувати приховані
                                         // віртуальні базові методи
};

int main()
{
    B b;
    int i = b.f();
    // E0165: too few arguments in function call
    // C2660: 'B::f': function does not take 0 arguments

    string s = "";
    b.f(s);
    // E0413: no suitable conversion function from "std::string" to "int" exists
    // C2664: 'int B::f(int) const': cannot convert argument 1 from 'std::string' to 'int'

    int k = b.f(1);

    system("pause");
}

```

Це – особливість реалізації перевизначення віртуальних методів у мові C++: клас **B** успадковує всі методи класу **A**, у т.ч. віртуальні. VMT класу **B** містить адреси всіх віртуальних методів класу **A**, проте при компіляції виводяться повідомлення про помилки, адже компілятор нічого не знає про пізні (динамічне) зв'язування. На етапі компіляції об'єкт **b** класу **B** не має доступу до віртуальних методів класу **A**.

З директивою `using` – все працює:

```
#include <iostream>
#include <string>

using namespace std;

class A
{
public:
    virtual int f() const                // віртуальний метод
    {
        cout << "A::f()" << endl;
        return 0;
    }

    virtual void f(const string& s) const // перевизначений віртуальний метод
    {
        cout << "A::f(string)" << endl;
    }
};

class B
    : public A
{
public:
    virtual int f(int i) const          // перевизначений віртуальний метод
                                        // з іншим набором параметрів
    {
        cout << "B::f(int)" << endl;
        return 0;
    }

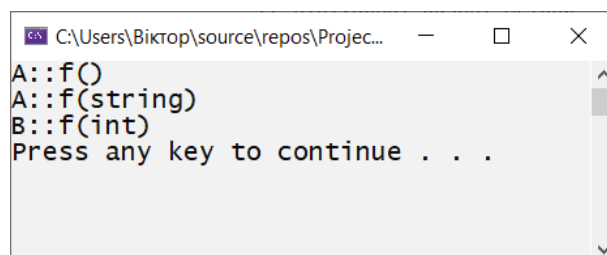
    using A::f;                        // дозвіл використовувати приховані
                                        // віртуальні базові методи
};

int main()
{
    B b;
    int i = b.f();                     // A::f()

    string s = "";
    b.f(s);                            // A::f(string)

    int k = b.f(1);                    // B::f(int)

    system("pause");
}
```



Директива `override`

Ключове слово `override` можна використовувати для того, щоб позначити віртуальні методи, які перевизначають віртуальні методи базового класу з такою ж самою сигнатурою.

`override` – контекстно-залежна директива: вона має спеціальне значення лише в тому випадку, якщо слово `override` використовується після оголошення віртуального метода, в іншому випадку це – не зарезервоване ключове слово.

Для запобігання випадковому створенню нових віртуальних методів у похідному класі слід використовувати директиву `override`. У наступному прикладі показано, що без використання `override`, поведінка методів похідного класу може бути не передбачуваною. Компілятор не видає помилки при використанні цього коду:

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass : public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const,
                        // so it does not override BaseClass::funcB() const
                        // and it is a new member function

    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a different
                                    // parameter type than BaseClass::funcC(int),
                                    // so DerivedClass::funcC(double)
                                    // is a new member function
};
```

При використанні `override` компілятор генерує помилки замість автоматичного створення нових віртуальних функцій:

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass : public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB() does not
                                    // override BaseClass::funcB() const

    virtual void funcC(double = 0.0) override; // compiler error:
                                                // DerivedClass::funcC(double) does not
                                                // override BaseClass::funcC(int)
};
```

```

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                           // override the non-virtual BaseClass::funcD()
};

```

Перевизначення із зміною лише типу результату

Розглянемо похідний клас, в якому робиться спроба змінити лише тип результату при перевизначенні успадкованого віртуального метода:

```

#include <iostream>

using namespace std;

class A
{
public:
    virtual int f() const                // віртуальний метод
    {
        cout << "A::f()" << endl;
        return 0;
    }
};

class B
    : public A
{
public:
    virtual void f() const              // інший тип результату
    {
        cout << "B::f()" << endl;
    }
};

```

В такому вигляді похідний клас не транлюється: перевизначати методи, які відрізняються лише типом результату – не можна: заборонено при перевантаженні методів змінювати лише тип результату.

Невіртуальний виклик віртуального методу

Віртуальний метод можна викликати невіртуально, якщо вказати кваліфікатор класу:

```

A *pa = new B();           // вказівник базового класу, налаштований
                           // на об'єкт похідного класу

pa->f();                    // виклик похідного методу
pa->A::f();                 // явний виклик базового методу

```

Такий виклик називається *статичним*. Він потрібний, коли в базовому класі реалізовані загальні дії, які мають виконуватися у всіх похідних класах.

Розмір класу з віртуальними методами

При наявності хоча б одного віртуального методу розмір класу (тобто, розмір кожного об'єкту цього класу) без полів дорівнює 4 байти (на платформі Intel) і не залежить від кількості віртуальних методів.

Якщо в класі є власні чи успадковані віртуальні методи, компілятор створює таблицю віртуальних методів VMT. В цій таблиці для кожного віртуального метода міститься вказівник на нього. Адреса цієї таблиці і записується в класі (фактично, в будь-якому об'єкті цього класу).

Таким чином, при використанні віртуальних методів витрачається додаткова пам'ять та додатковий час. По-перше, при створенні об'єкта поле вказівника на таблицю VMT слід ініціалізувати – це робить конструктор. По-друге, виклики віртуальних методів відбуваються опосередковано – через вказівник на таблицю VMT.

Проте динамічний поліморфізм – настільки потужна технологія, що розробники java зробили всі методи віртуальними за умовчанням. В C++ ми можемо керувати віртуальністю.

Віртуальні методи в конструкторах та деструкторах

Всередині конструкторів та деструкторів динамічне зв'язування не працює, хоча віртуальні методи можна викликати. Зазвичай віртуальні методи викликають за допомогою посилання чи вказівника на базовий клас, проте в конструкторах та деструкторах завжди викликається «рідний» метод.

В конструкторах це зумовлено тим, що конструктор нічого не знає про похідні класи. Під час виконання конструктора вже були створені об'єкти, проініціалізовані успадковані поля (конструкторами базових класів), та ініціалізуються власні поля. Про конструктори похідних класів та поля, які з'являються в похідних класах. – ще нічого не відомо. Цей конструктор міг сам бути викликаний із конструктора похідного класу, щоб створити і проініціалізувати об'єкт свого класу. Конструктор не знає, хто його викликав. Якби виклик міг бути віртуальним, то були б можливі ситуації використання неініціалізованих змінних похідних класів.

В деструкторах ситуація дещо інша – віртуальний виклик небезпечний тим, що тоді можливий виклик методу вже знищеного об'єкта. Тому навіть у віртуальному деструкторі завжди викликається «рідний» метод.

Віртуальні деструктори

Деструктор класу можна оголосити віртуальним. Коли деструктор базового класу – віртуальний, то і деструктори всіх нащадків – також віртуальні.

Деструктор необхідно робити віртуальним, коли доступ до динамічного об'єкту похідного класу виконується через вказівник базового класу (такі об'єкти називаються поліморфними). В цьому випадку при знищенні об'єкта через вказівник базового класу

спочатку викликається «рідний» деструктор похідного класу, а він вже викликає деструктор базового класу.

Якщо деструктор базового класу – не віртуальний, то при знищенні об'єкта похідного класу, який визначений через вказівник базового класу (об'єкта, який був би поліморфним, якщо клас мав би віртуальний метод), буде викликатися лише деструктор базового класу. Це приводить до некоректного звільнення пам'яті, бо базовий деструктор не знає про поля, визначені в похідному класі і не може їх звільнити.

Розглянемо ієрархії класів із звичайним та класів із віртуальним деструктором:

```
class Base
{
public:
    ~Base()                // не віртуальний деструктор
    {
        cout << "Base: not virtual destructor" << endl;
    }
};

class Derived
    : public Base
{
public:
    ~Derived()              // не віртуальний деструктор
    {
        cout << "Derived: not virtual destructor" << endl;
    }
};

class VBase
{
public:
    virtual ~VBase()        // віртуальний деструктор
    {
        cout << "VBase: virtual destructor" << endl;
    }
};

class VDerived
    : public VBase
{
public:
    virtual ~VDerived()      // віртуальний деструктор
                                // virtual можна не писати
    {
        cout << "VDerived: virtual destructor" << endl;
    }
};

int main()
{
    Base *b = new Derived();  // підстановка
    delete b;                // виклик лише базового деструктора

    VBase *vb = new VDerived(); // поліморфний об'єкт
    delete vb;                // виклик похідного, а потім
                                // - базового деструктора

    return 0;
}
```

Чисті віртуальні деструктори

Деструктор можна оголосити чистим віртуальним (тобто, абстрактним) методом:

```
virtual ~VBase() = 0;           // чистий віртуальний деструктор
```

Клас, в якому визначено чистий віртуальний деструктор, є абстрактним. Створювати об'єкти такого класу не можна. Проте клас-нащадок *не буде абстрактним*, оскільки деструктор не успадковується! Якщо в нащадку явно не визначено власного деструктора, система створить його автоматично.

При оголошенні чистого віртуального деструктора потрібно написати і його визначення – реалізацію метода (цим абстрактні деструктори відрізняються від інших абстрактних методів):

```
class ABase                               // абстрактний клас
{
public:
    virtual ~ABase() = 0;                 // чистий віртуальний деструктор
};

ABase::~~ABase()                          // реалізація чистого деструктора
{
    cout << "ABase: pure virtual destructor" << endl;
}

class VDerived
    : public ABase
{
public:
    virtual ~VDerived()                  // віртуальний деструктор
    {                                   // virtual можна не писати
        cout << "VDerived: virtual destructor" << endl;
    }
};

int main()
{
    ABase *vb = new VDerived();           // поліморфний об'єкт
    delete vb;                           // виклик похідного, а потім
                                         // - базового деструктора
    return 0;
}
```

Якщо не написати реалізації чистого віртуального деструктора, буде помилка. Вимога для чистого віртуального деструктора обов'язково мати реалізацію пояснюється тим, що деструктор нащадка викликає деструктор базового класу, тому чистий віртуальний деструктор має мати визначення.

В чистому віртуальному деструкторі зручно вказати код, який має виконуватися при знищенні об'єктів всіх класів-нащадків.

Віртуалізація зовнішніх функцій

Розглянемо цей механізм на прикладі реалізації універсальної операції виведення.

Будь-які зовнішні (у т.ч. дружні) функції не можуть бути віртуальними. Проте ми все-таки можемо забезпечити різну поведінку операції виведення `operator <<` для різних класів, що належать до однієї ієрархії.

Для цього використаємо механізм віртуальних методів наступним чином (цей спосіб вже став стандартним підходом):

В базовому класі оголошується чистий віртуальний метод, а в похідних класах реалізується його визначення. В зовнішню функцію (операцію виведення) передається параметр базового класу, для якого і буде викликатися базовий віртуальний метод.

Найпростішу схему такого способу демонструє наступний код:

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

class Control // абстрактний клас
{ // (точніше, інтерфейс)
public:
    virtual ostream& Print(ostream& out) const = 0;
};

class Button: public Control
{
public:
    virtual ostream& Print(ostream& out) const // Button знає, як себе виводити
    { // код виводу полів Button
        return (out << "Button" << endl);
    }
};

class Label: public Control
{
public:
    virtual ostream& Print(ostream& out) const // Label знає, як себе виводити
    { // код виводу полів Label
        return (out << "Label" << endl);
    }
};

ostream& operator << (ostream& out, const Control &r) // 2-й параметр – це інтерфейс:
{
    return r.Print(out); // r – поліморфний об'єкт
}

int main()
{
    Button b;
    cout << b; // вивід: Button

    Label l;
    cout << l; // вивід: Label

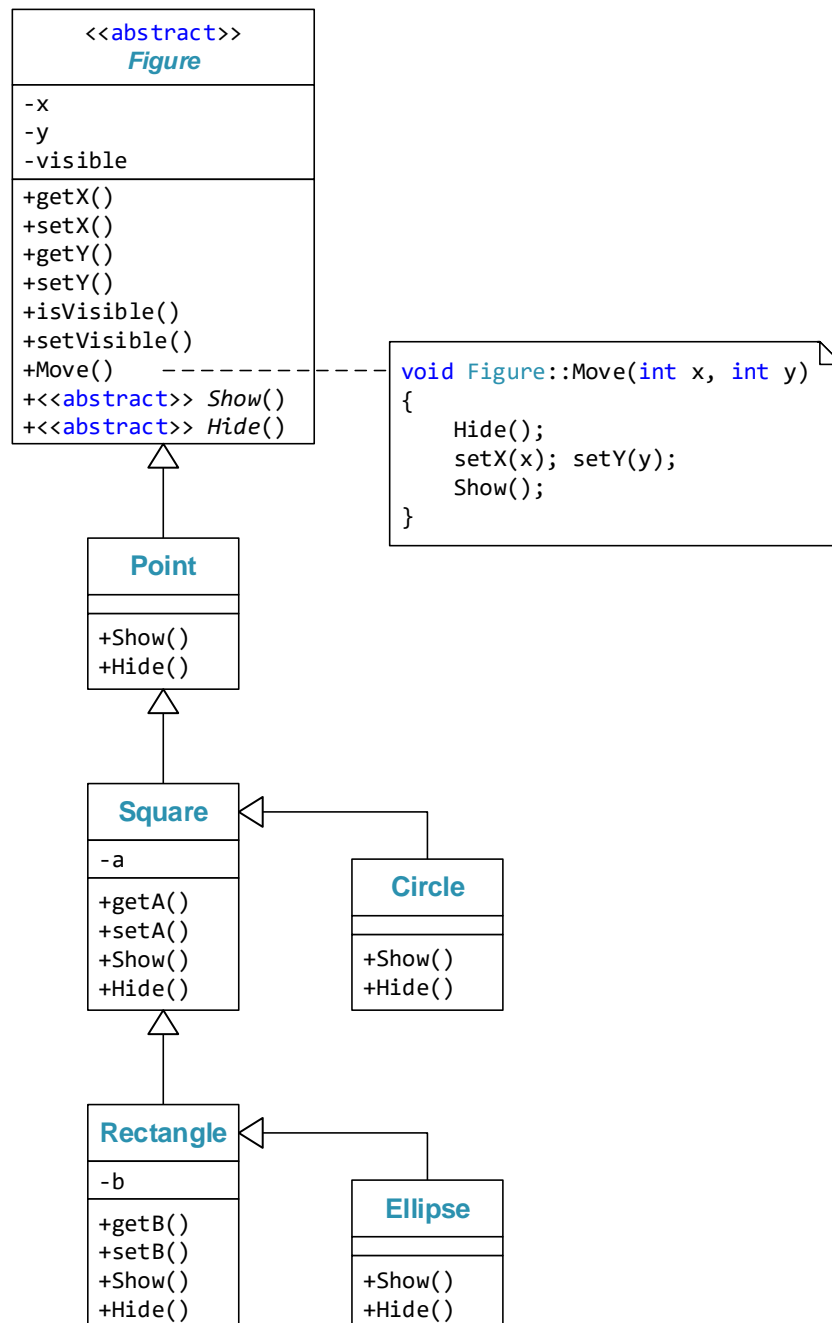
    return 0;
}
```

Зверніть увагу: єдина зовнішня функція `operator <<` навіть не є дружньою!

Тут знову використано принцип підстановки: при визначенні зовнішньої функції – операції виведення другий параметр – посилання на базовий тип, а викликаємо цю операцію з аргументами похідних типів.

Абстрактні методи та абстрактні класи. Інтерфейси

Коли ми створюємо ієрархію класів, вершиною ієрархії стає клас, який містить найбільш загальні властивості всіх нащадків. Проте в кожному нащадку ці властивості – різні. Наприклад, розглянемо сукупність геометричних фігур на екрані. Схема ієрархії класів таких фігур може бути наступною (що означає << abstract >> – пояснено далі):



Родоначальником ієрархії є клас **Figure**, який представляє найбільш загальні характеристики всіх фігур: координати `x`, `y` та признак видимості `visible`. В класі **Figure** також представлені найбільш загальні дії, які можна виконувати з усіма фігурами: отримати (`getX()`, `getY()`) та встановити (`setX()`, `setY()`) значення координат, отримати (`isVisible()`) та встановити (`setVisible()`) признак видимості, перемістити (`Move()`), погасити (`Hide()`)

та прорисувати (`Show()`).

Кожна фігура має певне положення, яке описується координатами `x` та `y`. Кожна фігура може бути видимою (`visible == true`) чи ні (`visible == false`). Ми можемо отримувати та встановлювати значення координат та признаку видимості. Кожну фігуру взагалі-то можна прорисувати та погасити (тобто, прорисувати фоновим кольором). Кожну фігуру можна перемістити – для цього її слід погасити, встановити нові значення координат та прорисувати.

Проте, ми не знаємо як саме слід прорисовувати фігуру – бо це занадто загальний клас. Синонімом для «занадто загальний» є термін *абстрактний*. Оскільки *фігура* – *абстрактне* поняття, ми не знаємо (і не можемо знати) алгоритму її прорисовки. Так само, ми не знаємо (і не можемо знати) рецепту приготування *абстрактної* їжі; не знаємо (і не можемо знати) про те, як слід використовувати *абстрактні меблі* і т.д. – таких прикладів ще можна навести багато.

Поліморфний метод `Move()` в класі `Figure` викликає методи `Hide()` та `Show()`, які будуть перевизначені в похідних класах. Зрозуміло, що методи `Hide()` та `Show()` мають бути віртуальними, – бо метод `Move()` має мати можливість викликати відповідні методи `Hide()` та `Show()` похідних класів, крім того, алгоритми їх реалізації для конкретних фігур дуже сильно відрізняються. Поліморфний клас `Figure` має мати методи `Hide()` та `Show()`, бо інакше компілятор не зможе згенерувати команд виклику цих методів в тілі метода `Move()`.

Виникає питання: як реалізувати методи `Hide()` та `Show()` в базовому класі `Figure`? Адже ці загальні методи не мають взагалі нічого робити в базовому класі – вся робота має виконуватися в похідних класах. Один з варіантів – визначити віртуальні методи з порожнім тілом, а в похідних класах перевизначити їх:

```
class Figure
{
    int x;
    int y;
    bool visible;

public:
    int getX() const { return x; }
    void setX(int value) { x = value; }

    int getY() const { return y; }
    void setY(int value) { y = value; }

    bool isVisible() const { return visible; }
    void setVisible(bool value) { visible = value; }

    void Move(int x, int y) // поліморфний метод
    {
        Hide();
        setX(x); setY(y);
        Show();
    }
}
```

```
virtual void Hide() {} // віртуальний метод з порожнім тілом
virtual void Show() {} // віртуальний метод з порожнім тілом
};
```

Так можна зробити, проте «порожнє тіло» насправді зовсім не означає відсутності команд. Зазвичай при виклику функції виконуються стандартні команди, згенеровані компілятором (*стандартний пролог*), зокрема: запам'ятати в стек адресу наступної команди, запам'ятати поточний стан регістрів в стек, занести в стек значення параметрів, перейти до виконання тіла функції, зчитати із стеку значення параметрів. При поверненні з функції теж виконуються команди, згенеровані компілятором (*стандартний епілог*), зокрема: зчитати із стеку значення регістрів, зчитати із стеку значення наступної після виклику функції команди і перейти до її виконання. Тобто, «порожнє тіло» – не те саме, що «зовсім без тіла». Тому навіть порожнє тіло при виконанні приведе до виконання якогось непотрібного для цього випадку коду.

В нашому прикладі добре б підійшли методи, які *насправді* не мають тіла. Відсутність тіла слід якимось чином позначити – адже ми не можемо просто написати прототипи:

```
virtual void Hide(); // прототип віртуального методу
virtual void Show(); // прототип віртуального методу
```

– бо тоді все-рівно потрібно буде визначати метод поза класом.

Абстрактні методи – чисті віртуальні функції

Розробник мови C++ Б. Страуструп позначив відсутність тіла метода нулем. Такий віртуальний метод називається «чистим» (англ. *pure*) і визначається наступним чином:

```
virtual тип ім'я(параметри) = 0; // абстрактний метод або
                                // чистий віртуальний метод
```

Синонімом для поняття *чистий віртуальний метод* є термін *абстрактний метод*. Багато мов програмування (наприклад, java та C#) для абстрактних методів вводять ключове слово **abstract**, але в C++ такі методи позначаються присвоєнням нуля.

Хоча чистий віртуальний метод не має тіла, для його класу все рівно створюється таблиця віртуальних методів. Б. Страуструп, позначаючи чистий віртуальний метод нулем, підкреслив, що в таблиці віртуальних методів адреса цього методу дорівнює нулю, тому його викликати не можна.

Абстрактні класи

Клас, який містить хоча б один абстрактний метод (тобто, чистий віртуальний метод) називається *абстрактний клас*.

На UML-діаграмах імена абстрактних класів записуються курсивом або із стереотипом << abstract >>, який вказується перед іменем класу.

Не можна створювати об'єктів абстрактного класу, навіть динамічно – за допомогою команди `new`. При передаванні параметрів у функцію за значенням не можна передати об'єкт абстрактного класу – бо це привело би до створення копії, що не дозволяється. Причина проста: коли створено об'єкт, то він має вміти виконувати всі дії, визначені для його класу. Припустимо, ми створили об'єкт `f` класу `Figure`. Тоді об'єкт `f` може викликати метод `Show()`:

```
f.Show(); // виклик абстрактного методу – помилка!
```

Стосовно прикладу з геометричною фігурою: алгоритму прорисовки абстрактної фігури – немає! Щоб усунути такі помилки, компілятор відслідковує спроби створення об'єктів абстрактних класів (чи як звичайних, чи динамічних змінних, чи при передаванні параметрів-значень) та забороняє їх.

Проте можна створювати посилання та вказівники на абстрактні класи. Також можна передавати параметри-посилання та параметри-вказівники на абстрактні класи. Це тому, що для вказівника розмір класу не має значення – для будь-якого вказівника виділяється 4 байти (на платформі Intel). Саме через посилання та вказівники реалізовано поліморфізм об'єктів.

При успадковуванні абстрактність зберігається: якщо похідний клас не реалізує (не перевизначає) успадкованого абстрактного методу, то він теж стає абстрактним:

```
class Figure // абстрактний клас
{
    int x;
    int y;
    bool visible;

public:
    int getX() const { return x; }
    void setX(int value) { x = value; }

    int getY() const { return y; }
    void setY(int value) { y = value; }

    bool isVisible() const { return visible; }
    void setVisible(bool value) { visible = value; }

    void Move(int x, int y) // поліморфний метод
    {
        Hide();
        setX(x); setY(y);
        Show();
    }
}
```

```

    virtual void Hide() = 0; // абстрактний метод
    virtual void Show() = 0; // абстрактний метод
};

class Point : public Figure // звичайний клас
{
public:
    virtual void Hide()      // перевизначили абстрактний метод
    {
        cout << "Point::Hide()" << endl;
    }

    virtual void Show()      // перевизначили абстрактний метод
    {
        cout << "Point::Show()" << endl;
    }
};

```

Якби ми не перевизначили методи `Hide()` та `Show()` в класі `Point`, то цей клас містив би однойменні абстрактні методи, успадковані від `Figure`. Тоді клас `Point` був би абстрактним і ми не могли б створювати об'єктів цього класу (як не можемо створювати об'єктів класу `Figure`). Для того, щоб зняти абстрактність з похідного класу, в ньому слід перевизначити абстрактний метод з тим самим набором параметрів.

Очевидно, що абстрактні методи мають бути доступними: вони описують деякі загальні дії, які в похідних класах будуть конкретизовані і стануть доступними для виконання (похідні класи мають перевизначати абстрактні методи).

Ключове слово `abstract` (MS C++)

Ключове слово `abstract` має два значення, – воно оголошує, що:

- Клас – абстрактний, тобто його можна використовувати в якості базового типу, але екземпляр (об’єкт) цього класу не може бути створений:

```
class A abstract
{
    // елементи класу
};
```

- Метод – абстрактний, тобто цей віртуальний метод має пере-визначатися в похідному класі:

```
class B
{
    virtual void foo() abstract;
};
```

Це – еквівалентно оголошенню такого метода як чистою віртуальною функцією:

```
class B
{
    virtual void foo() = 0;
};
```

Оголошення хоча б одного метода в класі абстрактним (тобто, чистою віртуальною функцією) призводить до того, що цей клас стає абстрактним.

Інтерфейси

Абстрактний клас – це узагальнення поняття *класу*. Клас стає настільки загальним, що містить лише перелік узагальнених дій, причому невідомо, як саме ці дії реалізувати. Звичайний клас може містити поля та методи, причому всі його методи мають мати реалізацію. Абстрактний клас має містити хоча би один абстрактний метод, який не має реалізації. Крім абстрактних методів, абстрактний клас може містити поля та звичайні методи.

Продовжимо процес узагальнення і доведемо поняття *абстрактного класу* до абсолюту.

Кожний предмет – об'єкт реального світу – виконує певну функцію (можливо, не одну). Ми цінуємо ці предмети саме за оті користі для нас функції, тобто – дії, які можуть виконувати такі предмети, або дії, які ми можемо виконувати за їх допомогою. Різні категорії користувачів використовують різні функції предметів, тобто – сприймають предмети по-різному. Наприклад, пасажир сприймає авто як крісло з дахом на колесах, яке може його перемістити в потрібне місце. Заправник – як каністру, яку слід заправити паливом певного сорту. Майстер СТО – як набір деталей, які потрібно скласти так, «щоб воно поїхало». Узагальнюючи класи таких предметів, ми прийдемо до наборів найбільш загальних функцій, які характерні для них. Тобто, їжу можна приготувати та спожити, меблі можна розмістити в кімнаті тощо.

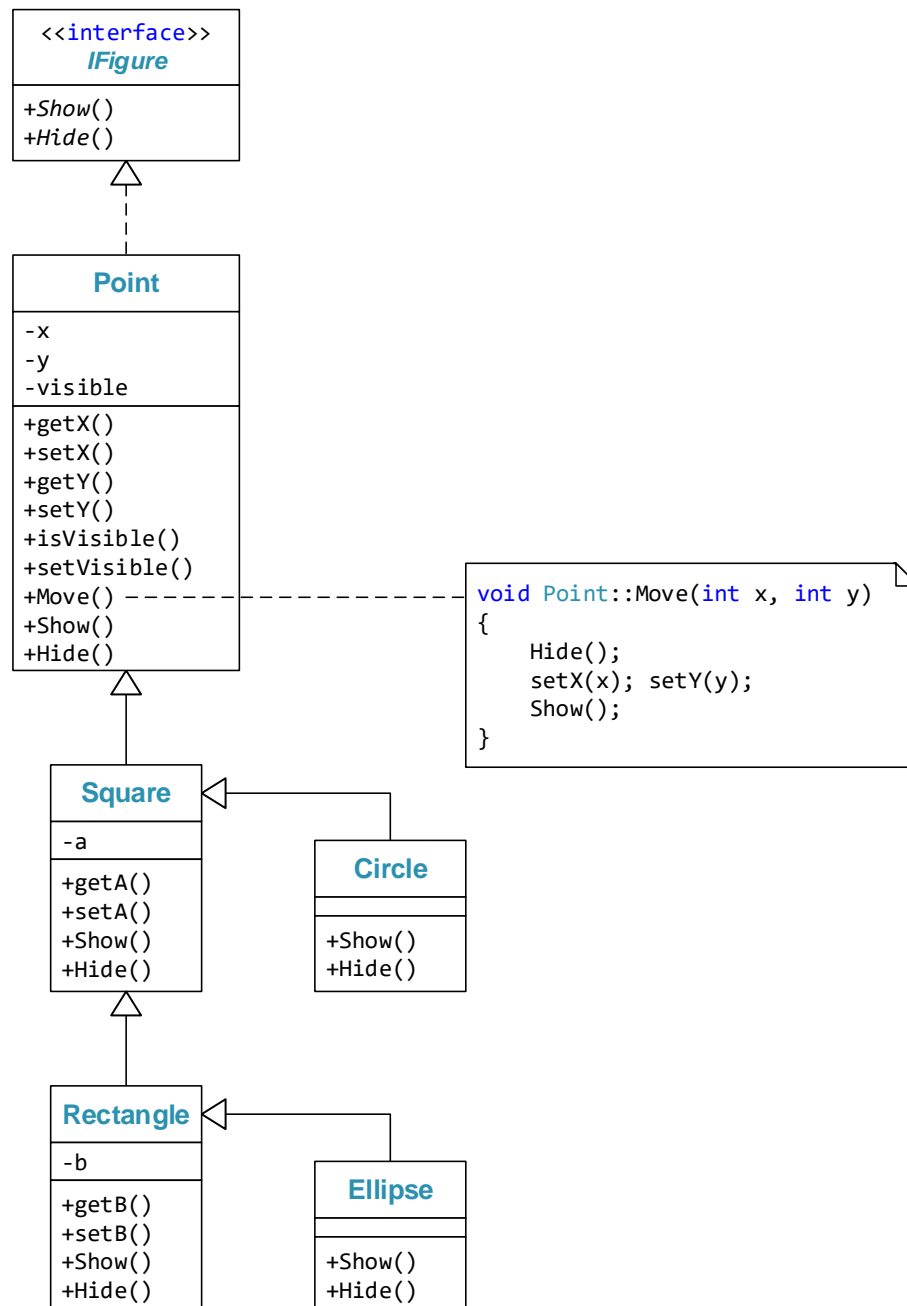
Таким чином ми приходимо до поняття *інтерфейсу*: «ідеального» абстрактного класу, який містить лише абстрактні методи.

Багато мов програмування (наприклад, java та C#) мають окрему синтаксичну конструкцію для визначення інтерфейсів. В C++ такого нема. Для позначення інтерфейсу будемо створювати клас, який містить лише чисті абстрактні методи.

На UML-діаграмах інтерфейси позначаються так само, як класи, лише із стереотипом `<< interface >>`, який вказується перед іменем інтерфейсу.

На заміну відношенню *успадковування* між класами приходить відношення *реалізації інтерфейсу*: коли клас є нащадком інтерфейсу, це називаємо «клас реалізує інтерфейс». При реалізації інтерфейсу клас має перевизначити всі абстрактні методи, що входять до складу інтерфейсу.

На UML діаграмах відношення реалізації позначається пунктирною трикутною незаповненою стрілкою від класу-нащадка до інтерфейсу-предка. UML-діаграму для сукупності геометричних фігур з попереднього прикладу можна зобразити за допомогою інтерфейсу наступним чином:



Відповідний код:

```

class IFigure // інтерфейс
{
public:
    virtual void Hide() = 0; // абстрактний метод
    virtual void Show() = 0; // абстрактний метод
};

class Point // звичайний клас
    : public IFigure // реалізує інтерфейс
{
    int x;
    int y;
    bool visible;

public:
    int getX() const { return x; }
    void setX(int value) { x = value; }

```

```

int getY() const { return y; }
void setY(int value) { y = value; }

bool isVisible() const { return visible; }
void setVisible(bool value) { visible = value; }

void Move(int x, int y)    // поліморфний метод
{
    Hide();
    setX(x); setY(y);
    Show();
}
virtual void Hide()        // перевизначили абстрактний метод
{
    cout << "Point::Hide()" << endl;
}
virtual void Show()        // перевизначили абстрактний метод
{
    cout << "Point::Show()" << endl;
}
};

```

Успадковування від абстрактного класу чи інтерфейсу (реалізація інтерфейсу) означають успадковування інтерфейсу – на відміну від успадковування реалізації при закритому успадковуванні.

Поліморфні об'єкти

Поняття поліморфних об'єктів

Поліморфний об'єкт – це той об'єкт, справжній тип якого відрізняється від оголошеного.

Точніше, для того щоб об'єкт був поліморфним, необхідно дотримання наступних умов:

- 1) Клас поліморфного об'єкту має містити хоча б один віртуальний метод.
- 2) Поліморфний об'єкт необхідно оголошувати через вказівник чи посилання на базовий клас.
- 3) Цьому вказівнику чи посиланню слід присвоїти вказівник чи посилання на об'єкт похідного класу.

Визначення поліморфного об'єкту – оголошення та створення

Поліморфізм об'єктів буває двох видів:

- 1) поліморфні об'єкти – динамічні змінні;
- 2) поліморфні об'єкти – параметри-посилання.

Поліморфні об'єкти – динамічні змінні

```
class A
{
public:
    virtual void f()
    {
        cout << "A::f()" << endl;
    }
};

class B
    : public A
{
public:
    virtual void f()
    {
        cout << "B::f()" << endl;
    }
};

int main()
{
    A *p = new B(); // поліморфний об'єкт - динамічна змінна
    p->f();          // вивід: B::f()

    return 0;
}
```

Поліморфні об'єкти – параметри-посилання

```
class A
{
public:
    virtual void f()
    {
        cout << "A::f()" << endl;
    }
};

class B
    : public A
{
public:
    virtual void f()
    {
        cout << "B::f()" << endl;
    }
};

void F(A& r)          // поліморфний об'єкт - параметр-посилання
{
    r.f();
}

int main()
{
    A a;               // передаємо об'єкт "рідного" базового класу
    F(a);              // вивід: A::f()

    B b;               // передаємо об'єкт похідного класу
    F(b);              // вивід: B::f()

    return 0;
}
```

RTTI

RTTI – run-time type identification – *динамічна ідентифікація типу* складається із трьох елементів:

- операції динамічного перетворення типу `dynamic_cast<>`;
- операції ідентифікації точного типу об'єкту `typeid()`;
- класу `type_info`.

Операція динамічного перетворення типу `dynamic_cast<>` розглядалася при вивченні Теми 4. Успадковування.

Для використання операції ідентифікації точного типу об'єкту `typeid()` та класу `type_info` слід підключити файл заголовку:

```
#include <typeinfo>
```


Визначення справжнього типу поліморфного об'єкту

Клас `type_info`

Після підключення файлу заголовку

```
#include <typeinfo>
```

стають доступними його ресурси: клас `type_info` та операція ідентифікації точного типу об'єкту `typeid()`, які дозволяють визначити справжній тип поліморфного об'єкту.

Інтерфейс класу `type_info` визначено так:

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
};
```

Не можна безпосередньо ні створювати об'єкти класу `type_info`, ні копіювати їх – тому що і єдиний конструктор копіювання, і операція присвоєння – закриті. Єдиний спосіб створити (тимчасовий) об'єкт класу `type_info` – використати операцію `typeid()`.

Операції `==` та `!=` використовуються для порівняння на рівність і нерівність з іншими об'єктами класу `type_info`.

Метод `before()` немає ніякого відношення до ієрархії успадковування.

Метод `name()` повертає літерний рядок типу `const char*` з нульових символом в кінці, що представляє ім'я типу у форматі, легкому для читання людиною.

Метод `raw_name()` повертає літерний рядок типу `const char*` з нульових символом в кінці, що представляє ім'я типу у закодованому форматі, зручному для порівняння, але не призначеному для читання людиною.

Операція `typeid()`

Операція `typeid()` має дві форми:

```
typeid( вираз )
typeid( ім'я_типу )
```

В якості виразів найчастіше використовуються вказівники. Таким чином, операція `typeid()` дозволяє визначити точний тип поліморфного об'єкта, на який посилається

вказівник. Якщо вказівник – нульовий, то генерується виняток `bad_typeid` (винятки розглядаються в Темі 5. Опрацювання виняткових ситуацій).

Оскільки операція `typeid()` повертає об'єкт класу `type_info`, то можна порівнювати ці об'єкти та викликати методи класу `type_info`:

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

using namespace std;

class Base {
public:
    virtual void func() {}           // щоб зробити клас поліморфним
};

class Derived : public Base {};

int main() {
    Derived* pd = new Derived;
    Base* pb = pd;                   // поліморфний об'єкт

    cout << typeid( pb ).name() << endl; // виводить "class Base *"
    cout << typeid( *pb ).name() << endl; // виводить "class Derived"
    cout << typeid( pd ).name() << endl;  // виводить "class Derived *"
    cout << typeid( *pd ).name() << endl; // виводить "class Derived"

    delete pd;

    return 0;
}
```

Лабораторний практикум

Оформлення звіту про виконання лабораторних робіт

Вимоги до оформлення звіту про виконання лабораторних робіт №№ 4.1–4.5

Звіт про виконання лабораторних робіт №№ 4.1–4.5 має містити наступні елементи:

- 1) заголовок;
- 2) мету роботи;
- 3) умову завдання;

Умова завдання має бути вставлена у звіт як фрагмент зображення (скрін) сторінки посібника.

- 4) UML-діаграму класів;
- 5) структурну схему програми;

Структурна схема програми зображує взаємозв'язки програми та всіх її програмних одиниць: схему вкладеності та охоплення підпрограм, програми та модулів; а також схему звертання одних програмних одиниць до інших.

- 6) текст програми;

Текст програми має бути правильно відформатований: відступами і порожніми рядками слід відображати логічну структуру програми; програма має містити необхідні коментарі – про призначення підпрограм, змінних та параметрів – якщо їх імена не значущі, та про призначення окремих змістовних фрагментів програми. Текст програми слід подавати моноширинним шрифтом (Courier New розміром 10 пт. або Consolas розміром 9,5 пт.) з одинарним міжрядковим інтервалом;

- 7) посилання на git-репозиторій з проектом (див. інструкції з Лабораторної роботи № 2.2 з предмету «Алгоритмізація та програмування»);
- 8) хоча б для одної функції, яка повертає результат (як результат функції чи як параметр-посилання) – результати unit-тесту: текст програми unit-тесту та скрін результатів її виконання (див. інструкції з Лабораторної роботи № 5.6 з предмету «Алгоритмізація та програмування»);
- 9) висновки.

Зразок оформлення звіту про виконання лабораторних робіт №№ 4.1–4.5

ЗВІТ

про виконання лабораторної роботи № < номер >

« назва теми лабораторної роботи »

з дисципліни

«Об'єктно-орієнтоване програмування»

студента(ки) групи КН-26

< Прізвище Ім'я По_батькові >

Мета роботи:

...

Умова завдання:

...

UML-діаграма класів:

...

Структурна схема програми:

...

Текст програми:

...

Посилання на git-репозиторій з проектом:

...

Результати unit-тесту:

...

Висновки:

...

Лабораторна робота № 4.1. Віртуальні функції та абстрактні класи

Мета роботи

Освоїти використання віртуальних та абстрактних функцій та абстрактних класів.

Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення поліморфізму.
- 2) Поняття та призначення віртуальних методів.
- 3) Таблиця віртуальних методів та її призначення.
- 4) Поняття поліморфного методу.
- 5) Поняття поліморфного класу.
- 6) Поняття та призначення чистих віртуальних методів (абстрактних методів).
- 7) Поняття та призначення абстрактних класів.
- 8) Поняття та призначення поліморфних об'єктів.
- 9) Визначення справжнього типу поліморфного об'єкта.

Варіанти завдань

Створити вказану у завданнях ієрархію класів. Створити поліморфні об'єкти. Під час виконання програми виводити справжній тип поліморфних об'єктів.

В реалізації за допомогою абстрактних класів всі спільні поля мають бути описані у абстрактному класі.

Варіант 1.

Створити абстрактний базовий клас з віртуальною функцією – площа(). Створити похідні класи: прямокутник, коло, прямокутний трикутник, трапеція зі своїми функціями обчислення площі. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів. Площа трапеції: $S = (a + b) \cdot h / 2$

Варіант 2.

Створити абстрактний клас з віртуальною функцією: норма(). Створити похідні класи: комплексні числа, вектор з 10 елементів, матриця (2x2). Визначити функцію норми: для комплексних чисел – модуль у квадраті, для вектора – корінь квадратний із суми квадратів елементів, для матриці – максимальне значення за модулем. Для перевірки

визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 3.

Створити абстрактний клас **криві** з віртуальною функцією: обчислення залежності y від x . Створити похідні класи: **пряма**, **еліпс**, **гіпербола** зі своїми функціями обчислення y в залежності від вхідного параметра x .

Рівняння прямої: $y = ax + b$, еліпса: $x^2/a^2 + y^2/b^2 = 1$, гіперболи: $x^2/a^2 - y^2/b^2 = 1$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 4.

Створити абстрактний базовий клас з віртуальною функцією – **сума_прогресії()**. Створити похідні класи: **арифметична прогресія** і **геометрична прогресія**. Кожен клас має два поля типу **double**. Перше – перший член прогресії, друге – постійна різниця для арифметичної і постійне відношення для геометричної прогресії. Визначити функції обчислення суми, де параметром є кількість елементів прогресії.

Арифметична прогресія $a_j = a_0 + jd, \quad j = 0, 1, 2, \dots$

Сума арифметичної прогресії: $s_n = (n + 1)(a_0 + a_n)/2$.

Геометрична прогресія: $a_j = a_0 r^j, \quad j = 0, 1, 2, \dots$

Сума геометричної прогресії: $s_n = (a_0 - a_n r)/(1 - r)$.

Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 5*.

Створити базовий клас – **список**. Реалізувати на базі списку **стек** і **чергу** з віртуальними функціями додавання та видалення елементів. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 6.

Створити базовий клас – **фігура**, і похідні класи – **коло**, **прямокутник**, **трапеція**. Визначити віртуальні функції визначення площі, периметра і виведення на екран. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 7.

Створити базовий клас – працівник і похідні класи – службовець з погодинною оплатою, службовець з окладом. Визначити віртуальну функцію нарахування зарплати. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 8.

Створити абстрактний базовий клас з віртуальною функцією – площа_поверхні(). Створити похідні класи: паралелепіпед, тетраедр, куля зі своїми функціями площі поверхні. Площа поверхні паралелепіпеда: $S = 2(xy + xz + yz)$. Площа поверхні кулі: $S = 4\pi r^2$. Площа поверхні тетраедра: $S = a^2 \sqrt{3}$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 9.

Створити абстрактний базовий клас з віртуальною функцією – об'єм(). Створити похідні класи: паралелепіпед, піраміда, тетраедр, куля зі своїми функціями об'єму. Об'єм паралелепіпеда: $V = xyz$ (x, y, z – сторони), піраміди з прямокутною основою: $V = xyh/3$ (x, y – сторони основи, h – висота), тетраедра: $V = a^3 \sqrt{2}/12$, кулі: $V = 4\pi r^3/3$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 10.

Створити абстрактний клас – ссавці. Визначити похідні класи – тварини і люди. У тварин визначити похідні класи собак і корів. Визначити віртуальні функції опису людини, собаки і корови. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 11.

Створити базовий клас – батько, у якого є ім'я. Визначити віртуальну функцію виведення імені на екран. Створити похідний клас дитина, у якого є поле по_батькові. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 12.

Створити абстрактний базовий клас з віртуальною функцією – корені_рівняння().

Створити похідні класи: клас лінійних рівнянь і клас квадратних рівнянь. Визначити функції обчислення коренів рівнянь. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 13.

Створити абстрактний базовий клас з віртуальною функцією – `периметр()`. Створити похідні класи: прямокутник, коло, прямокутний трикутник, трапеція зі своїми функціями обчислення периметру. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 14.

Створити абстрактний клас з віртуальною функцією: `додавання()`. Створити похідні класи: комплексні числа, вектор з 10 елементів, матриця (2x2). Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 15.

Створити абстрактний клас криві з віртуальною функцією: чи належить точка (x, y) заданій кривій. Створити похідні класи: пряма, еліпс, гіпербола зі своїми функціями визначення, чи належить точка (x, y) заданій кривій.

Рівняння прямої: $y = ax + b$, еліпса: $x^2/a^2 + y^2/b^2 = 1$, гіперболи: $x^2/a^2 - y^2/b^2 = 1$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 16.

Створити абстрактний базовий клас з віртуальною функцією – `елемент_прогресії()`. Створити похідні класи: арифметична прогресія і геометрична прогресія. Кожен клас має два поля типу `double`. Перше – перший член прогресії, друге – постійна різниця для арифметичної і постійне відношення для геометричної прогресії. Визначити функції обчислення значення елементу, де параметром є номер елементу прогресії.

Арифметична прогресія $a_j = a_0 + jd, \quad j = 0, 1, 2, \dots$

Геометрична прогресія: $a_j = a_0 r^j, \quad j = 0, 1, 2, \dots$

Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 17*.

Створити базовий клас – список. Реалізувати на базі списку лінійний_список і

кільцевий_список з віртуальними функціями включення і виключення елементів. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 18.

Створити базовий клас – фігура, і похідні класи – коло, еліпс, квадрат, прямокутник. Визначити віртуальні функції визначення площі та периметру. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 19.

Створити абстрактний базовий клас з віртуальною функцією – площа_поверхні(). Створити похідні класи: куб, паралелепіпед, тетраедр, куля зі своїми функціями обчислення площі поверхні. Площа поверхні паралелепіпеда: $S = 2(xy + xz + yz)$. Площа поверхні кулі: $S = 4\pi r^2$. Площа поверхні тетраедра: $S = a^2 \sqrt{3}$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 20*.

Створити базовий клас – список. Реалізувати на базі списку лінійний_список і кільцевий_список з віртуальною функцією обчислення кількості елементів. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 21.

Створити абстрактний клас – хребетні. Визначити похідні класи – ссавці і птахи. Для ссавців визначити похідні класи – тварини і люди. Для тварин визначити похідні класи собак і котів. Для птахів визначити похідний клас ворони. Визначити віртуальні функції опису ворони, людини, собаки і kota. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 22.

Створити абстрактний базовий клас з віртуальною функцією – об'єм(). Створити похідні класи: куб, паралелепіпед, піраміда, тетраедр, куля зі своїми функціями обчислення об'єму. Об'єм паралелепіпеда: $V = xyz$ (x, y, z – сторони), піраміди з прямокутною основою: $V = xyh/3$ (x, y – сторони основи, h – висота), тетраедра: $V = a^3 \sqrt{2}/12$, кулі: $V = 4\pi r^3/3$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 23.

Створити базовий клас – працівник з полем стаж роботи і похідні класи – робітник з погодинною оплатою, службовець з окладом. Визначити віртуальну функцію нарахування зарплати з врахуванням стажу. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 24.

Створити базовий клас – батько, у якого є поля ім'я та прізвище. Визначити віртуальну функцію виведення імені та прізвища на екран. Створити похідний клас дитина, у якого є поле по_батькові. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 25.

Створити абстрактний базовий клас з віртуальною функцією – існування_коренів_рівняння(). Створити похідні класи: клас лінійних рівнянь і клас квадратних рівнянь. Визначити функцію визначення, чи існують корені рівняння. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 26.

Створити абстрактний базовий клас з віртуальною функцією – площа(). Створити похідні класи: прямокутник, коло, прямокутний трикутник, трапеція зі своїми функціями обчислення площі. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів. Площа трапеції: $S = (a + b) \cdot h / 2$

Варіант 27.

Створити абстрактний клас з віртуальною функцією: норма(). Створити похідні класи: комплексні числа, вектор з 10 елементів, матриця (2x2). Визначити функцію норми: для комплексних чисел – модуль у квадраті, для вектора – корінь квадратний із суми квадратів елементів, для матриці – максимальне значення за модулем. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 28.

Створити абстрактний клас криві з віртуальною функцією: обчислення залежності у від x. Створити похідні класи: пряма, еліпс, гіпербола зі своїми функціями обчислення у в

залежності від вхідного параметра x .

Рівняння прямої: $y = ax + b$, еліпса: $x^2/a^2 + y^2/b^2 = 1$, гіперболи: $x^2/a^2 - y^2/b^2 = 1$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 29.

Створити абстрактний базовий клас з віртуальною функцією – `сума_прогресії()`. Створити похідні класи: арифметична прогресія і геометрична прогресія. Кожен клас має два поля типу `double`. Перше – перший член прогресії, друге – постійна різниця для арифметичної і постійне відношення для геометричної прогресії. Визначити функції обчислення суми, де параметром є кількість елементів прогресії.

Арифметична прогресія $a_j = a_0 + jd, \quad j = 0, 1, 2, \dots$

Сума арифметичної прогресії: $s_n = (n + 1)(a_0 + a_n)/2$.

Геометрична прогресія: $a_j = a_0 r^j, \quad j = 0, 1, 2, \dots$

Сума геометричної прогресії: $s_n = (a_0 - a_n r)/(1 - r)$.

Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 30*.

Створити базовий клас – `список`. Реалізувати на базі списку `стек` і `чергу` з віртуальними функціями `добавлення` та `вилучення елементів`. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 31.

Створити базовий клас – `фігура`, і похідні класи – `коло`, `прямокутник`, `трапеція`. Визначити віртуальні функції `визначення площі`, `периметра` і `виведення на екран`. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 32.

Створити базовий клас – `працівник` і похідні класи – `службовець з погодинною оплатою`, `службовець з окладом`. Визначити віртуальну функцію `нарахування зарплати`. Для перевірки визначити масив вказівників на базовий клас, яким присвоюються адреси різних об'єктів.

Варіант 33.

Створити абстрактний базовий клас з віртуальною функцією – `площа_поверхні()`. Створити похідні класи: `паралелепіпед`, `тетраедр`, `куля` зі своїми функціями площі поверхні. Площа поверхні паралелепіпеда: $S = 2(xy + xz + yz)$. Площа поверхні кулі: $S = 4\pi r^2$. Площа поверхні тетраедра: $S = a^2 \sqrt{3}$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 34.

Створити абстрактний базовий клас з віртуальною функцією – `об'єм()`. Створити похідні класи: `паралелепіпед`, `піраміда`, `тетраедр`, `куля` зі своїми функціями об'єму. Об'єм паралелепіпеда: $V = xyz$ (x, y, z – сторони), піраміди з прямокутною основою: $V = xyh/3$ (x, y – сторони основи, h – висота), тетраедра: $V = a^3 \sqrt{2}/12$, кулі: $V = 4\pi r^3/3$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 35.

Створити абстрактний клас – `савці`. Визначити похідні класи – `тварини` і `люди`. У тварин визначити похідні класи `собак` і `корів`. Визначити віртуальні функції опису людини, собаки і корови. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 36.

Створити базовий клас – `батько`, у якого є ім'я. Визначити віртуальну функцію виведення імені на екран. Створити похідний клас `дитина`, у якого є поле `по_батькові`. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 37.

Створити абстрактний базовий клас з віртуальною функцією – `корені_рівняння()`. Створити похідні класи: клас `лінійних рівнянь` і клас `квадратних рівнянь`. Визначити функції обчислення коренів рівнянь. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 38.

Створити абстрактний базовий клас з віртуальною функцією – `периметр()`. Створити

похідні класи: прямокутник, коло, прямокутний трикутник, трапеція зі своїми функціями обчислення периметру. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 39.

Створити абстрактний клас з віртуальною функцією: додавання(). Створити похідні класи: комплексні числа, вектор з 10 елементів, матриця (2x2). Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Варіант 40.

Створити абстрактний клас криві з віртуальною функцією: чи належить точка (x, y) заданій кривій. Створити похідні класи: пряма, еліпс, гіпербола зі своїми функціями визначення, чи належить точка (x, y) заданій кривій.

Рівняння прямої: $y = ax + b$, еліпса: $x^2/a^2 + y^2/b^2 = 1$, гіперболи: $x^2/a^2 - y^2/b^2 = 1$. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Лабораторна робота № 4.2. Інтерфейси (до 4.1)

Мета роботи

Освоїти використання віртуальних та абстрактних функцій та інтерфейсів.

Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення поліморфізму.
- 2) Поняття та призначення віртуальних методів.
- 3) Таблиця віртуальних методів та її призначення.
- 4) Поняття поліморфного методу.
- 5) Поняття поліморфного класу.
- 6) Поняття та призначення чистих віртуальних методів (абстрактних методів).
- 7) Поняття та призначення абстрактних класів.
- 8) Поняття та призначення поліморфних об'єктів.
- 9) Визначення справжнього типу поліморфного об'єкта.
- 10) Поняття та призначення інтерфейсів.

Варіанти завдань

Виконати завдання свого варіанту Лабораторної роботи № 4.1 «Віртуальні функції та абстрактні класи», використовуючи інтерфейси (класи, всі методи яких – абстрактні).

Створити поліморфні об'єкти. Під час виконання програми виводити справжній тип поліморфних об'єктів.

Лабораторна робота № 4.3. Віртуальні функції для класів з масивом

Мета роботи

Освоїти використання віртуальних та абстрактних функцій та абстрактних класів.

Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення поліморфізму.
- 2) Поняття та призначення віртуальних методів.
- 3) Таблиця віртуальних методів та її призначення.
- 4) Поняття поліморфного методу.
- 5) Поняття поліморфного класу.
- 6) Поняття та призначення чистих віртуальних методів (абстрактних методів).
- 7) Поняття та призначення абстрактних класів.
- 8) Поняття та призначення поліморфних об'єктів.
- 9) Визначення справжнього типу поліморфного об'єкта.

Варіанти завдань

Створити базовий клас `Array` з полями:

- масив типу `unsigned char i`
- поле для зберігання кількості елементів поточного об'єкту-масиву.
- Максимально можливий розмір масиву задається статичною константою.

Реалізувати:

- конструктор ініціалізації, який задає кількість елементів і початкове значення (за умовчанням 0).
- Реалізувати метод доступу до елементу, перевантаживши операцію індексування `[]`. При цьому має виконуватися перевірка індексу на допустимість – реалізувати метод `rangeCheck()` – перевірку заданого цілого числа на входження до діапазону.
- Реалізувати в класі `Array` віртуальну функцію по-елементного додавання масивів.

Реалізувати два нових класи, перевизначивши віртуальну функцію додавання.

Програма має продемонструвати всі варіанти виклику віртуальних функцій.

Створити поліморфні об'єкти. Під час виконання програми виводити справжній тип

поліморфних об'єктів.

Варіант 1.

Клас `Array` → клас `Fraction` і клас `BitString`.

На базі класу `Array` створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем `sign`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 2.

Клас `Array` → клас `BitString` і клас `Hex`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 3.

Клас `Array` → клас `Decimal` і клас `BitString`.

Створити клас `Decimal` для роботи з без-знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент

якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 4.

Клас `Array` → клас `Money` і клас `BitString`.

Створити клас `Money` для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 5.

Клас `Array` → клас `Decimal` і клас `BitString`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 6.

Клас `Array` → клас `Hex` і клас `BitString`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний

розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас **BitString** для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: **and**, **or**, **xor**, **not**. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 7.

Клас **Array** → клас **Octal** і клас **BitString**.

Створити клас **Octal** для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу **unsigned char**, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Створити клас **BitString** для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: **and**, **or**, **xor**, **not**. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 8.

Клас **Array** → клас **Hex** і клас **Octal**.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу **unsigned char**, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас **Octal** для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу **unsigned char**, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Варіант 9.

Клас `Array` → клас `Hex` і клас `Money`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас `Money` для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Варіант 10.

Клас `Array` → клас `Hex` і клас `Fraction`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

На базі класу `Array` створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем `sign`.

Варіант 11.

Клас `Array` → клас `Decimal` і клас `Hex`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний

розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 12.

Клас `Array` → клас `Octal` і клас `Hex`.

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 13.

Клас `Array` → клас `Decimal` і клас `Octal`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Варіант 14.

Клас `Array` → клас `Decimal` і клас `String`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими

десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятиковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 15.

Клас `Array` → клас `BitString` і клас `String`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 16.

Клас `Array` → клас `Fraction` і клас `String`.

На базі класу `Array` створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем `sign`.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті).

Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 17.

Клас `Array` → клас `Hex` і клас `String`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 18.

Клас `Array` → клас `Octal` і клас `String`.

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 19.

Клас `Array` → клас `Money` і клас `String`.

На базі класу `Array` створити клас `Money` для роботи з грошовими сумами. Сума має

бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

На базі класу `Array` створити клас `String` для роботи з літерними рядками, аналогічними рядкам `Turbo Pascal` (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 20.

Клас `Array` → клас `Money` і клас `Hex`.

На базі класу `Array` створити клас `Money` для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в `C++` операціям для цілих чисел та операції порівняння.

Варіант 21.

Клас `Array` → клас `Fraction` і клас `BitString`.

На базі класу `Array` створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем `sign`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 22.

Клас `Array` → клас `BitString` і клас `Hex`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100

біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 23.

Клас `Array` → клас `Decimal` і клас `BitString`.

Створити клас `Decimal` для роботи з без-знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 24.

Клас `Array` → клас `Money` і клас `BitString`.

Створити клас `Money` для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 25.

Клас `Array` → клас `Decimal` і клас `BitString`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 26.

Клас `Array` → клас `Hex` і клас `BitString`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас `BitString` для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 27.

Клас `Array` → клас `Octal` і клас `BitString`.

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції

порівняння.

Створити клас **BitString** для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу `unsigned char`, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: `and`, `or`, `xor`, `not`. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

Варіант 28.

Клас `Array` → клас `Hex` і клас `Octal`.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас **Octal** для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Варіант 29.

Клас `Array` → клас `Hex` і клас `Money`.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Створити клас **Money** для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Варіант 30.

Клас `Array` → клас `Hex` і клас `Fraction`.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами,

використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

На базі класу `Array` створити клас `Fraction` для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем `sign`.

Варіант 31.

Клас `Array` → клас `Decimal` і клас `Hex`.

На базі класу `Array` створити клас `Decimal` для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 32.

Клас `Array` → клас `Octal` і клас `Hex`.

Створити клас `Octal` для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Створити клас `Hex` для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні

операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Варіант 33.

Клас Array → клас Decimal і клас Octal.

На базі класу Array створити клас Decimal для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

Створити клас Octal для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

Варіант 34.

Клас Array → клас Decimal і клас String.

На базі класу Array створити клас Decimal для роботи з знаковими цілими десятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є десятковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Знак представити окремим полем `sign`.

На базі класу Array створити клас String для роботи з літерними рядками, аналогічними рядкам Turbo Pascal (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 35.

Клас Array → клас BitString і клас String.

Створити клас **BitString** для роботи з бітовими рядками довжиною не більше ніж 100 біт. Бітовий рядок має бути представлений масивом типу **unsigned char**, кожен елемент якого приймає значення 0 або 1. Реальний розмір масиву задається як аргумент конструктора ініціалізації. мають бути реалізовані всі традиційні операції для роботи з бітовими рядками: **and, or, xor, not**. Реалізувати зсув ліворуч та зсув праворуч на задану кількість бітів.

На базі класу **Array** створити клас **String** для роботи з літерними рядками, аналогічними рядкам **Turbo Pascal** (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 36.

Клас **Array** → клас **Fraction** і клас **String**.

На базі класу **Array** створити клас **Fraction** для роботи з дробовими десятковими числами. Кількість цифр в дробовій частині має задаватися в окремому полі і ініціалізуватися конструктором. Знак представити окремим полем **sign**.

На базі класу **Array** створити клас **String** для роботи з літерними рядками, аналогічними рядкам **Turbo Pascal** (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 37.

Клас **Array** → клас **Hex** і клас **String**.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу **unsigned char**, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

На базі класу **Array** створити клас **String** для роботи з літерними рядками, аналогічними рядкам **Turbo Pascal** (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані:

визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 38.

Клас Array → клас Octal і клас String.

Створити клас Octal для роботи з без-знаковими цілими вісімковими числами, використовуючи для представлення числа масив з 100 елементів типу unsigned char, кожен елемент якого є вісімковою цифрою. Молодша цифра має менший індекс (одиниці – в нульовому елементі масиву). Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ для цілих чисел, та операції порівняння.

На базі класу Array створити клас String для роботи з літерними рядками, аналогічними рядкам Turbo Pascal (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 39.

Клас Array → клас Money і клас String.

На базі класу Array створити клас Money для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає молодшій цифрі грошової суми. Молодші дві цифри – копійки.

На базі класу Array створити клас String для роботи з літерними рядками, аналогічними рядкам Turbo Pascal (рядок представляється як масив символів із 256 байт, елементи з 1 по 255 представляють значущі символи, фактична довжина – в нульовому байті). Максимальний розмір рядка має задаватися користувачем. Обов'язково мають бути реалізовані: визначення довжини рядка, пошук підрядка у рядку, видалення підрядка з рядка, вставка підрядка у рядок, зчеплення двох рядків.

Варіант 40.

Клас Array → клас Money і клас Hex.

На базі класу Array створити клас Money для роботи з грошовими сумами. Сума має бути представлена масивом, кожен елемент якого – десяткова цифра. Максимальна довжина масиву – 100 цифр, реальна довжина задається конструктором. Молодший індекс відповідає

молодшій цифрі грошової суми. Молодші дві цифри – копійки.

Створити клас **Hex** для роботи з без-знаковими цілими шістнадцятковими числами, використовуючи для представлення числа масив з 100 елементів типу `unsigned char`, кожний з яких є шістнадцятковою цифрою. Молодша цифра має менший індекс. Реальний розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції, аналогічні вбудованим в C++ операціям для цілих чисел та операції порівняння.

Лабораторна робота № 4.4. Абстрактні класи

Мета роботи

Освоїти використання віртуальних та абстрактних функцій та абстрактних класів.

Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення поліморфізму.
- 2) Поняття та призначення віртуальних методів.
- 3) Таблиця віртуальних методів та її призначення.
- 4) Поняття поліморфного методу.
- 5) Поняття поліморфного класу.
- 6) Поняття та призначення чистих віртуальних методів (абстрактних методів).
- 7) Поняття та призначення абстрактних класів.
- 8) Поняття та призначення поліморфних об'єктів.
- 9) Визначення справжнього типу поліморфного об'єкта.

Приклад розв'язання завдання 1

Завдання

Створити абстрактний базовий клас з віртуальною функцією – існування_коренів_рівняння(). Створити похідні класи: клас лінійних рівнянь і клас квадратних рівнянь. Визначити функцію визначення, чи існують корені рівняння. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Розв'язок

```
// Source.cpp
#include <iostream>
#include "Line.h"
#include "Square.h"
using namespace std;

int main()
{
    Line l(1,2);
    cout << l.ExistsRoot() << endl;

    Square s(1,4,3);
    cout << s.ExistsRoot() << endl;

    return 0;
}
```



```

// ABase.h

#pragma once

class ABase
{
    int a, b;

public:
    ABase(void);
    ABase(int a, int b);

    int getA(){ return a; }
    int getB(){ return b; }

    ~ABase(void);

    virtual bool ExistsRoot() = 0;
};

// ABase.cpp

#include "ABase.h"

ABase::ABase(void)
{}

ABase::ABase(int a, int b)
{
    this->a = a;
    this->b = b;
}

ABase::~~ABase(void)
{}

// Line.h

#pragma once
#include "abase.h"

class Line :
    public ABase
{
public:
    Line(void);
    Line(int a, int b);
    ~Line(void);

    virtual bool ExistsRoot();
};

// Line.cpp

#include "Line.h"

Line::Line(void)
{}

Line::Line(int a, int b)
: ABase(a, b)
{}

```

```

Line::~~Line(void)
{}

bool Line::ExistsRoot() //  $y = a*x + b$ 
{
    return getA() != 0;
}

// Square.h

#pragma once
#include "abase.h"

class Square :
    public ABase
{
    int c;

public:
    Square(void);
    Square(int a, int b, int c);
    ~Square(void);

    virtual bool ExistsRoot();
};

// Square.cpp

#include "Square.h"

Square::Square(void)
{}

Square::Square(int a, int b, int c)
: ABase(a, b)
{
    this->c = c;
}

Square::~~Square(void)
{}

bool Square::ExistsRoot() //  $y = a*x*x + b*x + c$ 
{
    return getB()*getB() - 4*getA()*c >= 0;
}

```

Приклад розв'язання завдання 2

Завдання

Створити абстрактний базовий клас **A** з полями **x** та **y** цілого типу та абстрактною арифметичною операцією **+**. Створити похідний клас **B** з полем **z** цілого типу, який реалізує віртуальні операції **+** та **=**.

Розв'язок за допомогою методу add()

```
#include <iostream>

using namespace std;

class A
{
public:
    int x;
    int y;

public:
    virtual A* add(A *r) =0;
};

class B: public A
{
public:
    int z;

public:
    virtual A* add(A *r)
    {
        B *t = new B();
        t->x = this->x + r->x;
        t->y = this->y + r->y;
        t->z = this->z + ((B*)r)->z;
        // а6о
        t->z = this->z + dynamic_cast<B*>(r)->z;
        return t;
    }

    virtual B* operator = (A *r)
    {
        this->x = r->x;
        this->y = r->y;
        this->z = ((B*)r)->z;
        // а6о
        this->z = dynamic_cast<B*>(r)->z;
        return this;
    }
};

int main()
{
    A *b1 = new B();
    A *b2 = new B();
    A *b3 = new B();
    b1->x = 1;
    b1->y = 2;
    ((B*)b1)->z = 3;
    b2->x = 4;
    b2->y = 5;
    ((B*)b2)->z = 6;
    b3 = b1->add(b2);
    cout << b3->x << " " << b3->y << " " << ((B*)b3)->z << endl;
}
```

Розв'язок за допомогою операції +

```
#include <iostream>

using namespace std;

class A
{
public:
    int x;
    int y;

public:
    virtual A* operator + (A *r) =0;
};

class B: public A
{
public:
    int z;

public:
    virtual A* operator + (A *r)
    {
        B *t = new B();
        t->x = this->x + r->x;
        t->y = this->y + r->y;
        t->z = this->z + ((B*)r)->z;
        // або
        t->z = this->z + dynamic_cast<B*>(r)->z;
        return t;
    }

    virtual B* operator = (A *r)
    {
        this->x = r->x;
        this->y = r->y;
        this->z = ((B*)r)->z;
        // або
        this->z = dynamic_cast<B*>(r)->z;
        return this;
    }
};

int main()
{
    A *b1 = new B();
    A *b2 = new B();
    A *b3 = new B();
    b1->x = 1;
    b1->y = 2;
    ((B*)b1)->z = 3;
    b2->x = 4;
    b2->y = 5;
    ((B*)b2)->z = 6;
    b3 = *b1 + b2; // щоб викликати операцію +,
                  // необхідно розіменувати вказівник b1
    cout << b3->x << " " << b3->y << " " << ((B*)b3)->z << endl;
}
```

Варіанти завдань

У наступних завданнях потрібно реалізувати абстрактний базовий клас, визначивши в ньому чисті віртуальні функції. Ці функції пере-визначаються в похідних класах. У базових класах мають бути оголошені чисті віртуальні функції вводу/виводу, які реалізуються в похідних класах.

Програма, яка викликається, має продемонструвати всі варіанти виклику віртуальних функцій за допомогою вказівників на базовий клас. Написати функцію виводу, яка отримує параметри базового класу за посиланням і демонструє віртуальний виклик.

В реалізації за допомогою абстрактних класів всі спільні поля мають бути описані у абстрактному класі.

Створити поліморфні об'єкти. Під час виконання програми виводити справжній тип поліморфних об'єктів.

Варіант 1.

Створити абстрактний базовий клас **Figure** з віртуальними методами обчислення площі і периметру. Створити похідні класи: **Rectangle** (прямокутник), **Circle** (круг), **Trapezium** (трапеція) з своїми функціями площі і периметру. Самостійно визначити, які поля необхідні, які з них можна задати в базовому класі, а які – в похідних.

Площа трапеції: $S = (a + b) \times h / 2$.

Варіант 2.*

Створити абстрактний базовий клас **Number** з віртуальними методами – арифметичними операціями. Створити похідні класи **Integer** (ціле) і **Real** (дійсне) – «обгортки» над стандартними типами **int** та **double** відповідно..

* – якщо абстрактний клас має поле, яке використовується нащадками.

Варіант 3.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **Money** і **Fraction**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Money** – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу **long** для гривень і
- типу **unsigned char** – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Клас **Fraction** – для роботи з дробовими числами. Число має бути представлене двома полями:

- ціла частина – довге ціле із знаком,
- дробова частина – без-знакове коротке ціле.

Реалізувати арифметичні операції:

- додавання,
- віднімання,
- множення,
- операції порівняння.

Варіант 4.

Створити абстрактний клас **Currency** (валюта) для роботи з грошовими сумами. Визначити віртуальні функції переведення в гривні і виводу на екран. Реалізувати похідні класи **Dollar** (долар) і **Euro** (євро) з своїми функціями переведення в гривні і виводу на екран.

Варіант 5.

Створити абстрактний базовий клас **Triangle** для представлення трикутника з віртуальними функціями обчислення площі і периметру. Поля даних мають включати дві сторони і кут між ними. Визначити класи-нащадки: прямокутний трикутник, рівнобедрений трикутник, рівносторонній трикутник з своїми функціями обчислення площі і периметра.

Варіант 6.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **FuzzyNumber** і **Complex**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **FuzzyNumber** – для роботи з нечіткими числами, які представляються трійками чисел $(x - l, x, x + r)$. Поля:

- x
- l
- r

Для чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ арифметичні операції виконуються за наступними формулами:

- додавання

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B);$$

- віднімання

$$A - B = (x_A - x_B - l_A - l_B, x_A - x_B, x_A - x_B + r_A + r_B);$$

- множення

$$A \times B = (x_A \times x_B - x_B \times l_A - x_A \times l_B - l_A \times l_B, x_A \times x_B, x_A \times x_B + x_B \times r_A + x_A \times r_B + r_A \times r_B);$$

- зворотне число

$$1 / A = (1/(x_A + r_A), 1 / x_A, 1/(x_A - l_A)), x_A > 0;$$

- ділення

$$A / B = ((x_A - l_A)/(x_B + r_B), x_A / x_B, (x_A + r_A)/(x_B - l_B)), x_B > 0;$$

Пояснення FuzzyNumber

Нечіткі числа подаються трійками $(x - l, x, x + r)$, де

x – координата центру,

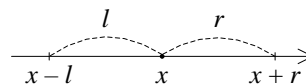
$x - l$ – координата лівої границі,

$x + r$ – координата правої границі.

Відповідно:

l – відстань від лівої границі до центру,

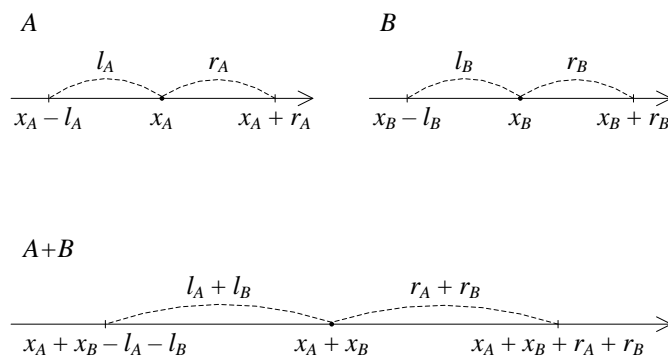
r – відстань від правої границі до центру:



Клас **FuzzyNumber** містить три поля: $\{x, l, r\}$.

Додавання двох нечітких чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ описується формулою

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B)$$



Тобто, для суми двох нечітких чисел $A+B$:

- $x_A + x_B$ — координата центру,
- $x_A + x_B - l_A - l_B$ — координата лівої границі,
- $x_A + x_B + r_A + r_B$ — координата правої границі.

Відповідно,

- $l_A + l_B$ — відстань від лівої границі до центру,
- $r_A + r_B$ — відстань від правої границі до центру.

Таким чином, якщо об'єкт-число A містить поля $\{x_A, l_A, r_A\}$, а об'єкт-число B — поля $\{x_B, l_B, r_B\}$, то об'єкт-сума містить поля $\{x_A + x_B, l_A + l_B, r_A + r_B\}$.

Комплексне число представляється парою дійсних чисел (a, b) , де поля

- a — дійсна частина,
- b — мніма частина.

Клас **Complex** — для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$;
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$;
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)$;
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2)$;
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2)$;
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b)$.

Варіант 7.

Створити абстрактний базовий клас **Root** (корінь) з віртуальними методами обчислення коренів рівняння і виведення результату на екран. Визначити похідні класи **Linear** (лінійне рівняння) і **Square** (квадратне рівняння) з власними методами обчислення коренів і виводу на екран.

Варіант 8.

Створити абстрактний базовий клас **Function** (функція) з віртуальними методами обчислення значення функції $y = f(x)$ в заданій точці x і виведення результату на екран. Визначити похідні класи **Ellipse** (еліпс), **Hyperbola** (гіпербола) з власними функціями обчислення y в залежності від вхідного параметра x . Рівняння еліпса $x^2/a^2 + y^2/b^2 = 1$; гіперболи: $x^2/a^2 - y^2/b^2 = 1$.

Варіант 9.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Реалізувати похідні класи **Complex** і **Rational**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Комплексне число представляється парою дійсних чисел (a, b) , де поля

- a – дійсна частина,
- b – мніма частина.

Клас **Complex** – для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$;
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$;
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)$;
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2)$;
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2)$;
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b)$.

Раціональний (нескоротний) дріб представляється парою цілих чисел (a, b) , де поля:

- a – чисельник,
- b – знаменник.

Клас **Rational** – для роботи з раціональними дробами. Обов'язково мають бути реалізовані наступні операції:

унарна (аргументом є поточний об'єкт):

- обчислення значення **value()**, a / b ;

```
double Rational::value()
{
    return 1.*a/b;
}

Rational z;
```

```
...
double x = z.value();
```

бінарні (перший аргумент – поточний об’єкт, другий аргумент – об’єкт-параметр):

- додавання $\text{add}()$, $(a_1, b_1) + (a_2, b_2) = (a_1 \cdot b_2 + a_2 \cdot b_1, b_1 \cdot b_2)$;

```
Rational Rational::add(Rational& v)
{
    Rational tmp;

    tmp.a = a * v.b + b * v.a;
    tmp.b = b * v.b;

    return tmp;
}

Rational z1, z2, z3;
...
z3 = z1.add(z2);
```

- віднімання $\text{sub}()$, $(a_1, b_1) - (a_2, b_2) = (a_1 \cdot b_2 - a_2 \cdot b_1, b_1 \cdot b_2)$;
- множення $\text{mul}()$, $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2, b_1 \cdot b_2)$;
- ділення $\text{div}()$, $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot b_2, a_2 \cdot b_1)$;
- порівняння «чи рівне» $\text{equal}()$;
- порівняння «чи більше» $\text{great}()$;
- порівняння «чи менше» $\text{less}()$.

Має бути реалізована приватна функція скорочення дробу $\text{Reduce}()$, яка обов’язково викликається при виконанні арифметичних операцій.

Варіант 10.*

Створити абстрактний базовий клас **Triad** з віртуальними методами збільшення на 1. Створити похідні класи **Date** і **Time**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Date** – для роботи з датами у форматі «рік.місяць.день» з трьома полями типу `unsigned int`:

- рік,
- місяць і
- номер дня.

Клас має включати не менше трьох функцій ініціалізації:

- числами,

- літерним рядком виду «рік.місяць.день» (наприклад, «2004.08.31») і
- датою.

Обов'язковими операціями є:

- * обчислення дати через задану кількість днів,
- * віднімання заданої кількості днів з дати,
- визначення, чи рік – високосний,
- присвоєння,
- отримання окремих частин (рік, місяць, день),
- порівняння дат (рівно, до, після),

Клас **Time** – для роботи з часом у форматі «година:хвилина:секунда» з трьома полями типу **unsigned int**:

- година,
- хвилина і
- секунда.

Клас має включати не менше чотирьох функцій ініціалізації:

- числами,
- літерним рядком (наприклад, «23:59:59»),
- секундами від початку доби і
- часом.

Обов'язковими операціями є:

- * обчислення різниці між двома моментами часу в секундах,
- * додавання часу і заданої кількості секунд,
- * віднімання з часу заданої кількості секунд,
- порівняння моментів часу,
- переведення в секунди,
- переведення в хвилини (з округленням до цілої хвилини).

Варіант 11.

Створити абстрактний базовий клас **Body** (тіло) з віртуальними функціями обчислення площі поверхні і об'єму. Створити похідні класи: **Parallelepiped** (паралелепіпед) і **Ball** (куля) з своїми функціями площі поверхні і об'єму.

Варіант 12.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями.

Реалізувати похідні класи Money1 і Money2.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас Money1 – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу long для гривень і
- типу unsigned char – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Номінали гривень можуть приймати значення 1, 2, 5, 10, 20, 50, 100, 200, 500. Копійки представити як 0.01 (1 копійка), 0.02 (2 копійки), 0.05 (5 копійок), 0.1 (10 копійок), 0.25 (25 копійок), 0.5 (50 копійок).

Клас Money2 – для роботи з грошовими сумами. Сума має бути представлена полями-номіналами, значеннями яких має бути кількість купюр відповідного номіналу. Поля:

- кількість банкнот по 500 грн.
- кількість банкнот по 200 грн.
- кількість банкнот по 100 грн.
- кількість банкнот по 50 грн.
- кількість банкнот по 20 грн.
- кількість банкнот по 10 грн.
- кількість банкнот по 5 грн.
- кількість банкнот по 2 грн.
- кількість банкнот по 1 грн.
- кількість монет по 50 коп.
- кількість монет по 25 коп.
- кількість монет по 10 коп.
- кількість монет по 5 коп.

- кількість монет по 2 коп.
- кількість монет по 1 коп.

Реалізувати:

- додавання сум,
- віднімання сум,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою.

Варіант 13.**

Створити абстрактний базовий клас **Integer** (ціле) з віртуальними арифметичними операціями і функцією виводу на екран. Визначити похідні класи **Decimal** (десятькове) і **Binary** (двійкове), що реалізують власні арифметичні операції і функцію виводу на екран. Число представляється масивом, кожен елемент якого – цифра.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 14.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **Money** і **Complex**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Money** – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу **long** для гривень і
- типу **unsigned char** – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,

- операції порівняння.

Комплексне число представляються парою дійсних чисел (a, b) , де поля

- a – дійсна частина,
- b – мніма частина.

Клас **Complex** – для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2);$
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2);$
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1);$
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2);$
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2);$
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b).$

Варіант 15.

Створити абстрактний базовий клас **Series** (прогресія) з віртуальними функціями обчислення j -го елемента прогресії і суми прогресії. Визначити похідні класи: **Linear** (арифметична) і **Exponential** (геометрична).

Арифметична прогресія: $a_j = a_0 + jd, \quad j = 0, 1, 2, \dots$

Сума арифметичної прогресії: $s_n = (n + 1)(a_0 + a_n)/2 .$

Геометрична прогресія: $a_j = a_0 r^j, \quad j = 0, 1, 2, \dots$

Сума геометричної прогресії: $s_n = (a_0 - a_n r)/(1 - r) .$

Варіант 16.

Створити абстрактний клас **Norm** з віртуальною функцією обчислення норми і модуля. Визначити похідні класи **Complex**, **Vector3D** з власними функціями обчислення норми і модуля.

Модуль для комплексного числа обчислюється як корінь з суми квадратів дійсної і уявної частин; норма для комплексних чисел обчислюється як модуль в квадраті. Модуль вектора обчислюється як корінь квадратний з суми квадратів координат; норма вектора обчислюється як максимальне з абсолютних значень координат.

Варіант 17.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **FuzzyNumber** і **Fraction**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **FuzzyNumber** – для роботи з нечіткими числами, які представляються трійками чисел $(x - l, x, x + r)$. Поля:

- x
- l
- r

Для чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ арифметичні операції виконуються за наступними формулами:

- додавання

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B);$$

- віднімання

$$A - B = (x_A - x_B - l_A - l_B, x_A - x_B, x_A - x_B + r_A + r_B);$$

- множення

$$A \times B = (x_A \times x_B - x_B \times l_A - x_A \times l_B - l_A \times l_B, x_A \times x_B, x_A \times x_B + x_B \times r_A + x_A \times r_B + r_A \times r_B);$$

- зворотне число

$$1 / A = (1/(x_A + r_A), 1 / x_A, 1/(x_A - l_A)), x_A > 0;$$

- ділення

$$A / B = ((x_A - l_A)/(x_B + r_B), x_A / x_B, (x_A + r_A)/(x_B - l_B)), x_B > 0;$$

Пояснення FuzzyNumber

Нечіткі числа подаються трійками $(x - l, x, x + r)$, де

x – координата центру,

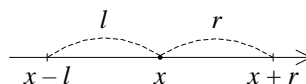
$x - l$ – координата лівої границі,

$x + r$ – координата правої границі.

Відповідно:

l – відстань від лівої границі до центру,

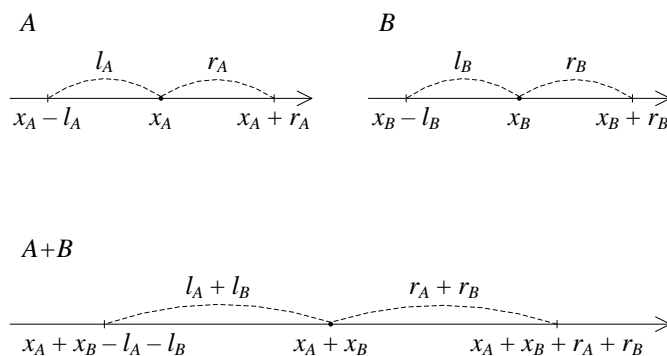
r – відстань від правої границі до центру:



Клас **FuzzyNumber** містить три поля: $\{x, l, r\}$.

Додавання двох нечітких чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ описується формулою

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B)$$



Тобто, для суми двох нечітких чисел $A+B$:

- $x_A + x_B$ — координата центру,
- $x_A + x_B - l_A - l_B$ — координата лівої границі,
- $x_A + x_B + r_A + r_B$ — координата правої границі.

Відповідно,

- $l_A + l_B$ — відстань від лівої границі до центру,
- $r_A + r_B$ — відстань від правої границі до центру.

Таким чином, якщо об'єкт-число A містить поля $\{x_A, l_A, r_A\}$, а об'єкт-число B — поля $\{x_B, l_B, r_B\}$, то об'єкт-сума містить поля $\{x_A + x_B, l_A + l_B, r_A + r_B\}$.

Клас **Fraction** — для роботи з дробовими числами. Число має бути представлене двома полями:

- ціла частина — довге ціле із знаком,
- дробова частина — без-знакове коротке ціле.

Реалізувати арифметичні операції:

- додавання,
- віднімання,
- множення,
- операції порівняння.

Варіант 18.**

Створити абстрактний базовий клас **Container**, який містить масив з віртуальними методами `sort()` і по-елементної обробки контейнера `foreach()`. Розробити похідні класи **Bubble** (бульбашка) і **Choice** (вибір). У першому класі сортування реалізується методом бульбашки, а по-елементна обробка полягає у обчисленні квадратного кореня. У другому класі сортування реалізується методом вибору, а по-елементна обробка — обчислення логарифма.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 19.**

Створити абстрактний базовий клас `Array` з віртуальними методами додавання і по-елементної обробки масиву `foreach()`. Розробити похідні класи `SortArray` і `XorArray`. У першому класі операція додавання реалізується як об'єднання множин, а по-елементна обробка – сортування. У другому класі операція додавання реалізується як виключне АБО, а по-елементна обробка – обчислення кореня.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 20.**

Створити абстрактний базовий клас `Array` з віртуальними методами додавання і по-елементної обробки масиву `foreach()`. Розробити похідні класи `AndArray` і `OrArray`. У першому класі операція додавання реалізується як перетин множин, а по-елементна обробка представляє собою обчислення квадратного кореня. У другому класі операція додавання реалізується як об'єднання, а по-елементна обробка – обчислення логарифма.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 21.

Створити абстрактний базовий клас `Figure` з віртуальними методами обчислення площі і периметру. Створити похідні класи: `Rectangle` (прямокутник), `Circle` (круг), `Trapezium` (трапеція) з своїми функціями площі і периметру. Самостійно визначити, які поля необхідні, які з них можна задати в базовому класі, а які – в похідних.

Площа трапеції: $S = (a + b) \times h/2$.

Варіант 22.*

Створити абстрактний базовий клас `Number` з віртуальними методами – арифметичними операціями. Створити похідні класи `Integer` (ціле) і `Real` (дійсне) – «обгортки» над стандартними типами `int` та `double` відповідно.

* – якщо абстрактний клас має поле, яке використовується нащадками.

Варіант 23.*

Створити абстрактний базовий клас `Pair` з віртуальними арифметичними операціями. Створити похідні класи `Money` і `Fraction`.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Money** – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу **long** для гривень і
- типу **unsigned char** – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Клас **Fraction** – для роботи з дробовими числами. Число має бути представлене двома полями:

- ціла частина – довге ціле із знаком,
- дробова частина – без-знакове коротке ціле.

Реалізувати арифметичні операції:

- додавання,
- віднімання,
- множення,
- операції порівняння.

Варіант 24.

Створити абстрактний клас **Currency** (валюта) для роботи з грошовими сумами. Визначити віртуальні функції переведення в гривні і виводу на екран. Реалізувати похідні класи **Dollar** (долар) і **Euro** (євро) з своїми функціями переведення в гривні і виводу на екран.

Варіант 25.

Створити абстрактний базовий клас **Triangle** для представлення трикутника з віртуальними функціями обчислення площі і периметру. Поля даних мають включати дві сторони і кут між ними. Визначити класи-нащадки: прямокутний трикутник, рівнобедрений трикутник, рівносторонній трикутник з своїми функціями обчислення площі і периметра.

Варіант 26.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями.
Створити похідні класи **FuzzyNumber** і **Complex**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **FuzzyNumber** – для роботи з нечіткими числами, які представляються трійками чисел $(x - l, x, x + r)$. Поля:

- x
- l
- r

Для чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ арифметичні операції виконуються за наступними формулами:

- додавання
 $A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B);$
- віднімання
 $A - B = (x_A - x_B - l_A - l_B, x_A - x_B, x_A - x_B + r_A + r_B);$
- множення
 $A \times B = (x_A \times x_B - x_B \times l_A - x_A \times l_B - l_A \times l_B, x_A \times x_B, x_A \times x_B + x_B \times r_A + x_A \times r_B + r_A \times r_B);$
- зворотне число
 $1 / A = (1 / (x_A + r_A), 1 / x_A, 1 / (x_A - l_A)), x_A > 0;$
- ділення
 $A / B = ((x_A - l_A) / (x_B + r_B), x_A / x_B, (x_A + r_A) / (x_B - l_B)), x_B > 0;$

Пояснення FuzzyNumber

Нечіткі числа подаються трійками $(x - l, x, x + r)$, де

x – координата центру,

$x - l$ – координата лівої границі,

$x + r$ – координата правої границі.

Відповідно:

l – відстань від лівої границі до центру,

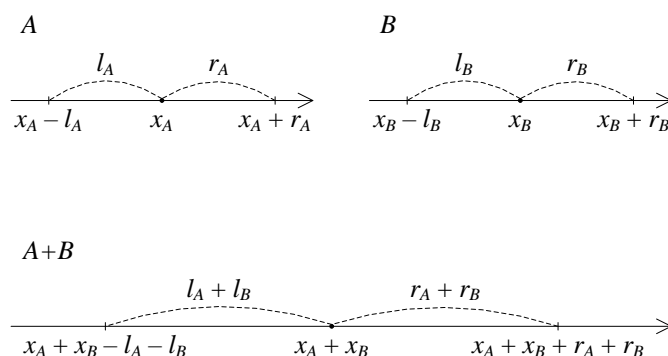
r – відстань від правої границі до центру:



Клас **FuzzyNumber** містить три поля: $\{x, l, r\}$.

Додавання двох нечітких чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ описується формулою

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B)$$



Тобто, для суми двох нечітких чисел $A+B$:

- $x_A + x_B$ — координата центру,
- $x_A + x_B - l_A - l_B$ — координата лівої границі,
- $x_A + x_B + r_A + r_B$ — координата правої границі.

Відповідно,

- $l_A + l_B$ — відстань від лівої границі до центру,
- $r_A + r_B$ — відстань від правої границі до центру.

Таким чином, якщо об'єкт-число A містить поля $\{x_A, l_A, r_A\}$, а об'єкт-число B — поля $\{x_B, l_B, r_B\}$, то об'єкт-сума містить поля $\{x_A + x_B, l_A + l_B, r_A + r_B\}$.

Комплексне число представляються парою дійсних чисел (a, b) , де поля

- a — дійсна частина,
- b — мніма частина.

Клас **Complex** — для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$;
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$;
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)$;
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2)$;
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2)$;
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b)$.

Варіант 27.

Створити абстрактний базовий клас **Root** (корінь) з віртуальними методами обчислення коренів рівняння і виведення результату на екран. Визначити похідні класи **Linear** (лінійне рівняння) і **Square** (квадратне рівняння) з власними методами обчислення коренів і виводу на екран.

Варіант 28.

Створити абстрактний базовий клас **Function** (функція) з віртуальними методами обчислення значення функції $y = f(x)$ в заданій точці x і виведення результату на екран. Визначити похідні класи **Ellipse** (еліпс), **Hyperbola** (гіпербола) з власними функціями обчислення y в залежності від вхідного параметра x . Рівняння еліпса $x^2/a^2 + y^2/b^2 = 1$; гіперболи: $x^2/a^2 - y^2/b^2 = 1$.

Варіант 29.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Реалізувати похідні класи **Complex** і **Rational**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Комплексне число представляється парою дійсних чисел (a, b) , де поля

- a – дійсна частина,
- b – мніма частина.

Клас **Complex** – для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$;
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$;
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)$;
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2)$;
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2)$;
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b)$.

Раціональний (нескоротний) дріб представляється парою цілих чисел (a, b) , де поля:

- a – чисельник,
- b – знаменник.

Клас **Rational** – для роботи з раціональними дробами. Обов'язково мають бути реалізовані наступні операції:

унарна (аргументом є поточний об'єкт):

- обчислення значення `value()`, a / b ;

```
double Rational::value()
{
    return 1.*a/b;
}

Rational z;
...
double x = z.value();
```

бінарні (перший аргумент – поточний об'єкт, другий аргумент – об'єкт-параметр):

- додавання `add()`, $(a_1, b_1) + (a_2, b_2) = (a_1 \cdot b_2 + a_2 \cdot b_1, b_1 \cdot b_2)$;

```
Rational Rational::add(Rational& v)
{
    Rational tmp;

    tmp.a = a * v.b + b * v.a;
    tmp.b = b * v.b;

    return tmp;
}

Rational z1, z2, z3;
...
z3 = z1.add(z2);
```

- віднімання `sub()`, $(a_1, b_1) - (a_2, b_2) = (a_1 \cdot b_2 - a_2 \cdot b_1, b_1 \cdot b_2)$;
- множення `mul()`, $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2, b_1 \cdot b_2)$;
- ділення `div()`, $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot b_2, a_2 \cdot b_1)$;
- порівняння «чи рівне» `equal()`;
- порівняння «чи більше» `great()`;
- порівняння «чи менше» `less()`.

Має бути реалізована приватна функція скорочення дробу `Reduce()`, яка обов'язково викликається при виконанні арифметичних операцій.

Варіант 30.*

Створити абстрактний базовий клас `Triad` з віртуальними методами збільшення на 1. Створити похідні класи `Date` і `Time`.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас `Date` – для роботи з датами у форматі «рік.місяць.день» з трьома полями типу `unsigned int`:

- рік,
- місяць і
- номер дня.

Клас має включати не менше трьох функцій ініціалізації:

- числами,
- літерним рядком виду «рік.місяць.день» (наприклад, «2004.08.31») і
- датою.

Обов'язковими операціями є:

- * обчислення дати через задану кількість днів,
- * віднімання заданої кількості днів з дати,
- визначення, чи рік – високосний,
- присвоєння,
- отримання окремих частин (рік, місяць, день),
- порівняння дат (рівно, до, після),

Клас **Time** – для роботи з часом у форматі «година:хвилина:секунда» з трьома полями типу **unsigned int**:

- година,
- хвилина і
- секунда.

Клас має включати не менше чотирьох функцій ініціалізації:

- числами,
- літерним рядком (наприклад, «23:59:59»),
- секундами від початку доби і
- часом.

Обов'язковими операціями є:

- * обчислення різниці між двома моментами часу в секундах,
- * додавання часу і заданої кількості секунд,
- * віднімання з часу заданої кількості секунд,
- порівняння моментів часу,
- переведення в секунди,
- переведення в хвилини (з округленням до цілої хвилини).

Варіант 31.

Створити абстрактний базовий клас **Body** (тіло) з віртуальними функціями обчислення площі поверхні і об'єму. Створити похідні класи: **Parallelepiped** (паралелепіпед) і **Ball** (куля) з своїми функціями площі поверхні і об'єму.

Варіант 32.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Реалізувати похідні класи **Money1** і **Money2**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Money1** – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу **long** для гривень і
- типу **unsigned char** – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Номінали гривень можуть приймати значення 1, 2, 5, 10, 20, 50, 100, 200, 500. Копійки представити як 0.01 (1 копійка), 0.02 (2 копійки), 0.05 (5 копійок), 0.1 (10 копійок), 0.25 (25 копійок), 0.5 (50 копійок).

Клас **Money2** – для роботи з грошовими сумами. Сума має бути представлена полями-номіналами, значеннями яких має бути кількість купюр відповідного номіналу. Поля:

- кількість банкнот по 500 грн.
- кількість банкнот по 200 грн.
- кількість банкнот по 100 грн.
- кількість банкнот по 50 грн.
- кількість банкнот по 20 грн.
- кількість банкнот по 10 грн.

- кількість банкнот по 5 грн.
- кількість банкнот по 2 грн.
- кількість банкнот по 1 грн.
- кількість монет по 50 коп.
- кількість монет по 25 коп.
- кількість монет по 10 коп.
- кількість монет по 5 коп.
- кількість монет по 2 коп.
- кількість монет по 1 коп.

Реалізувати:

- додавання сум,
- віднімання сум,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою.

Варіант 33.**

Створити абстрактний базовий клас **Integer** (ціле) з віртуальними арифметичними операціями і функцією виводу на екран. Визначити похідні класи **Decimal** (десятькове) і **Binary** (двійкове), що реалізують власні арифметичні операції і функцію виводу на екран. Число представляється масивом, кожен елемент якого – цифра.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 34.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **Money** і **Complex**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **Money** – для роботи з грошовими сумами. Число має бути представлене двома полями:

- типу **long** для гривень і
- типу **unsigned char** – для копійок.

Дробова частина (копійки) при виводі на екран має бути відокремлена від цілої частини комою. Реалізувати операції:

- додавання,
- віднімання,
- ділення сум,
- ділення суми на дробове число,
- множення на дробове число,
- операції порівняння.

Комплексне число представляються парою дійсних чисел (a, b) , де поля

- a – дійсна частина,
- b – мніма частина.

Клас **Complex** – для роботи з комплексними числами. Обов'язково мають бути реалізовані операції:

- додавання **add()** $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$;
- віднімання **sub()** $(a_1, b_1) - (a_2, b_2) = (a_1 - a_2, b_1 - b_2)$;
- множення **mul()** $(a_1, b_1) \times (a_2, b_2) = (a_1 \cdot a_2 - b_1 \cdot b_2, a_1 \cdot b_2 + a_2 \cdot b_1)$;
- ділення **div()** $(a_1, b_1) / (a_2, b_2) = (a_1 \cdot a_2 + b_1 \cdot b_2, a_2 \cdot b_1 - a_1 \cdot b_2) / (a_2^2 + b_2^2)$;
- порівняння **equ()** $(a_1, b_1) = (a_2, b_2)$, якщо $(a_1 = a_2)$ і $(b_1 = b_2)$;
- комплексно спряжене число **conj()** $\text{conj}(a, b) = (a, -b)$.

Варіант 35.

Створити абстрактний базовий клас **Series** (прогресія) з віртуальними функціями обчислення j -го елемента прогресії і суми прогресії. Визначити похідні класи: **Linear** (арифметична) і **Exponential** (геометрична).

Арифметична прогресія: $a_j = a_0 + jd, \quad j = 0, 1, 2, \dots$

Сума арифметичної прогресії: $s_n = (n + 1)(a_0 + a_n)/2$.

Геометрична прогресія: $a_j = a_0 r^j, \quad j = 0, 1, 2, \dots$

Сума геометричної прогресії: $s_n = (a_0 - a_n r)/(1 - r)$.

Варіант 36.

Створити абстрактний клас **Norm** з віртуальною функцією обчислення норми і модуля. Визначити похідні класи **Complex**, **Vector3D** з власними функціями обчислення норми і модуля.

Модуль для комплексного числа обчислюється як корінь з суми квадратів дійсної і

уявної частин; норма для комплексних чисел обчислюється як модуль в квадраті. Модуль вектора обчислюється як корінь квадратний з суми квадратів координат; норма вектора обчислюється як максимальне з абсолютних значень координат.

Варіант 37.*

Створити абстрактний базовий клас **Pair** з віртуальними арифметичними операціями. Створити похідні класи **FuzzyNumber** і **Fraction**.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Клас **FuzzyNumber** – для роботи з нечіткими числами, які представляються трійками чисел $(x - l, x, x + r)$. Поля:

- x
- l
- r

Для чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$ арифметичні операції виконуються за наступними формулами:

- додавання
$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B);$$
- віднімання
$$A - B = (x_A - x_B - l_A - l_B, x_A - x_B, x_A - x_B + r_A + r_B);$$
- множення
$$A \times B = (x_A \times x_B - x_B \times l_A - x_A \times l_B - l_A \times l_B, x_A \times x_B, x_A \times x_B + x_B \times r_A + x_A \times r_B + r_A \times r_B);$$
- зворотне число
$$1 / A = (1/(x_A + r_A), 1 / x_A, 1/(x_A - l_A)), x_A > 0;$$
- ділення
$$A / B = ((x_A - l_A)/(x_B + r_B), x_A / x_B, (x_A + r_A)/(x_B - l_B)), x_B > 0;$$

Пояснення FuzzyNumber

Нечіткі числа подаються трійками $(x - l, x, x + r)$, де

x – координата центру,

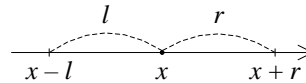
$x - l$ – координата лівої границі,

$x + r$ – координата правої границі.

Відповідно:

l – відстань від лівої границі до центру,

r – відстань від правої границі до центру:

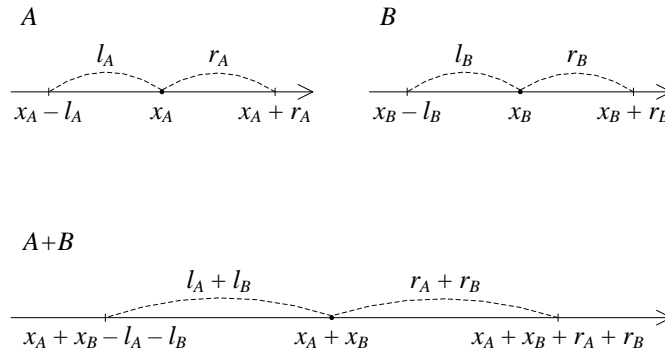


Клас **FuzzyNumber** містить три поля: $\{x, l, r\}$.

Додавання двох нечітких чисел $A = (x_A - l_A, x_A, x_A + r_A)$ та $B = (x_B - l_B, x_B, x_B + r_B)$

описується формулою

$$A + B = (x_A + x_B - l_A - l_B, x_A + x_B, x_A + x_B + r_A + r_B)$$



Тобто, для суми двох нечітких чисел $A+B$:

$x_A + x_B$ – координата центру,

$x_A + x_B - l_A - l_B$ – координата лівої границі,

$x_A + x_B + r_A + r_B$ – координата правої границі.

Відповідно,

$l_A + l_B$ – відстань від лівої границі до центру,

$r_A + r_B$ – відстань від правої границі до центру.

Таким чином, якщо об'єкт-число A містить поля $\{x_A, l_A, r_A\}$, а об'єкт-число B – поля $\{x_B, l_B, r_B\}$, то об'єкт-сума містить поля $\{x_A + x_B, l_A + l_B, r_A + r_B\}$.

Клас **Fraction** – для роботи з дробовими числами. Число має бути представлене двома полями:

- ціла частина – довге ціле із знаком,
- дробова частина – без-знакове коротке ціле.

Реалізувати арифметичні операції:

- додавання,
- віднімання,
- множення,
- операції порівняння.

Варіант 38.**

Створити абстрактний базовий клас `Container`, який містить масив з віртуальними методами `sort()` і по-елементної обробки контейнера `foreach()`. Розробити похідні класи `Bubble` (бульбашка) і `Choice` (вибір). У першому класі сортування реалізується методом бульбашки, а по-елементна обробка полягає у обчисленні квадратного кореня. У другому класі сортування реалізується методом вибору, а по-елементна обробка – обчислення логарифма.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 39.**

Створити абстрактний базовий клас `Array` з віртуальними методами додавання і по-елементної обробки масиву `foreach()`. Розробити похідні класи `SortArray` і `XorArray`. У першому класі операція додавання реалізується як об'єднання множин, а по-елементна обробка – сортування. У другому класі операція додавання реалізується як виключне АБО, а по-елементна обробка – обчислення кореня.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Варіант 40.**

Створити абстрактний базовий клас `Array` з віртуальними методами додавання і по-елементної обробки масиву `foreach()`. Розробити похідні класи `AndArray` і `OrArray`. У першому класі операція додавання реалізується як перетин множин, а по-елементна обробка представляє собою обчислення квадратного кореня. У другому класі операція додавання реалізується як об'єднання, а по-елементна обробка – обчислення логарифма.

* – якщо абстрактний клас має поля, які використовуються нащадками.

Лабораторна робота № 4.5. Інтерфейси (до 4.4)

Мета роботи

Освоїти використання віртуальних та абстрактних функцій та інтерфейсів.

Питання, які необхідно вивчити та пояснити на захисті

- 1) Поняття та призначення поліморфізму.
- 2) Поняття та призначення віртуальних методів.
- 3) Таблиця віртуальних методів та її призначення.
- 4) Поняття поліморфного методу.
- 5) Поняття поліморфного класу.
- 6) Поняття та призначення чистих віртуальних методів (абстрактних методів).
- 7) Поняття та призначення абстрактних класів.
- 8) Поняття та призначення поліморфних об'єктів.
- 9) Визначення справжнього типу поліморфного об'єкта.
- 10) Поняття та призначення інтерфейсів.

Варіанти завдань

Виконати завдання свого варіанту Лабораторної роботи № 4.4 «Абстрактні класи», використовуючи інтерфейси (класи, що містять лише абстрактні методи).

Створити поліморфні об'єкти. Під час виконання програми виводити справжній тип поліморфних об'єктів.

Приклад розв'язання завдання

Завдання

Створити інтерфейс з функцією – існування_коренів_рівняння(). Створити похідні класи: клас лінійних рівнянь і клас квадратних рівнянь. Визначити функцію визначення, чи існують корені рівняння. Для перевірки визначити масив вказівників на абстрактний клас, яким присвоюються адреси різних об'єктів.

Розв'язок

```
// Source.cpp  
  
#include <iostream>  
#include "Line.h"  
#include "Square.h"
```

```

using namespace std;

int main()
{
    Line l(1,2);
    cout << l.ExistsRoot() << endl;

    Square s(1,4,3);
    cout << s.ExistsRoot() << endl;

    return 0;
}

// IBase.h

#pragma once

class IBase
{
public:
    virtual bool ExistsRoot() = 0;
};

// IBase.cpp

#include "IBase.h"

// Line.h

#pragma once
#include "ibase.h"

class Line :
    public IBase
{
    int a, b;

public:
    Line(void);
    Line(int a, int b);
    ~Line(void);

    virtual bool ExistsRoot();
};

// Line.cpp

#include "Line.h"

Line::Line(void)
{}

Line::~Line(void)
{}

Line::Line(int a, int b)
{
    this->a = a;
    this->b = b;
}

```

```

bool Line::ExistsRoot() //  $y = a*x + b$ 
{
    return a != 0;
}

// Square.h

#pragma once
#include "ibase.h"

class Square :
    public IBase
{
    int a, b, c;

public:
    Square(void);
    Square(int a, int b, int c);
    ~Square(void);

    virtual bool ExistsRoot();
};

// Square.cpp

#include "Square.h"

Square::Square(void)
{}

Square::Square(int a, int b, int c)
{
    this->a = a;
    this->b = b;
    this->c = c;
}

Square::~~Square(void)
{}

bool Square::ExistsRoot() //  $y = a*x*x + b*x + c$ 
{
    return b*b - 4*a*c >= 0;
}

```


Питання та завдання для контролю знань

1. Поясніть, навіщо потрібні віртуальні функції.
2. Що таке зв'язування?
3. Чим раннє зв'язування відрізняється від пізнього?
4. Які два види поліморфізму реалізовано в C++?
5. Чи впливає наявність віртуальних функцій на розмір класу?
6. Дайте визначення поліморфного класу.
7. Чи може віртуальна функція бути дружньою функцією класу?
8. Чи успадковуються віртуальні функції?
9. Які особливості виклику віртуальних функцій в конструкторах і деструкторах?
10. Чи можна зробити віртуальною перевантажену операцію, наприклад додавання?
11. Чи можна віртуальну функцію викликати не віртуально?
12. Чи може конструктор бути віртуальним? А деструктор?
13. У яких випадках виклик віртуальної функції класу-нащадка через вказівник базового класу вимагає явного перетворення типу?
14. Як віртуальні функції впливають на розмір класу?
15. Як оголошується чиста віртуальна функція?
16. Дайте визначення абстрактного класу.
17. Чи успадковуються чисті віртуальні функції?
18. Чи можна оголосити деструктор чисто віртуальним?
19. Чим відрізняється чистий віртуальний деструктор від чистої віртуальної функції?
20. Навіщо потрібне визначення чистого віртуального деструктора?
21. Чи можна зробити операцію присвоєння віртуальною? А операцію індексування? А чистою віртуальною?
22. Поясніть, як можна реалізувати «віртуальність» незалежної функції.
23. Приведіть класифікацію цілей успадковування.
24. Поясніть різницю між успадковуванням інтерфейсу та успадковуванням реалізації.
25. Як зв'язані віртуальні функції і принцип підстановки?
26. Дано визначення класів:

```
class C1
{
    public:
        void aPolymorphMethod();           // поліморфний метод
};

class C2: public C1
{
    public:
        virtual void aVirtualMethod();    // віртуальний метод
};
```

```

void C1::aPolymorphMethod()
{
    aVirtualMethod();
}

void C2::aVirtualMethod() {}

```

Виправити помилку у визначенні цих класів, яка робить неможливим поліморфізм. Відповідь пояснити.

27. Дано визначення класів:

```

class C1
{
    public:
        void aPolymorphMethod();           // поліморфний метод
        virtual void aVirtualMethod() = 0; // абстрактний метод
};

class C2: public C1
{
    public:
        virtual void aVirtualMethod();     // віртуальний метод
};

void C1::aPolymorphMethod()
{
    aVirtualMethod();
}

void C2::aVirtualMethod() {}

```

Чому в класі C1 має бути метод aVirtualMethod ? Відповідь пояснити.

28. Дано визначення класів:

```

class C1
{
    public:
        void aPolymorphMethod();           // поліморфний метод
        virtual void aVirtualMethod() = 0; // абстрактний метод
};

class C2
{
    public:
        virtual void aVirtualMethod();     // віртуальний метод
};

void C1::aPolymorphMethod()
{
    aVirtualMethod();
}

void C2::aVirtualMethod() {}

```

Виправити помилку у визначенні цих класів, яка робить неможливим поліморфізм. Відповідь пояснити.

29. Дано визначення класів:

```
class C1
{
    public:
        void aPolymorphMethod(); // поліморфний метод
};

class C2: public C1 {
    public:
        void aMethod();
};

void C1::aPolymorphMethod()
{
    aMethod();
}

void C2::aMethod() {}
```

Виправити помилки у визначенні цих класів, які роблять неможливим поліморфізм.
Відповідь пояснити.

30. Дано визначення класів:

```
class C1
{
    public:
        void aPolymorphMethod(); // поліморфний метод
        virtual void aVirtualMethod() = 0; // абстрактний метод
};

class C2: public C1
{
    public:
        virtual void aVirtualMethod() = 0; // абстрактний метод
};

void C1::aPolymorphMethod()
{
    aVirtualMethod();
}
```

Чи є помилка у визначенні цих класів, яка робить неможливим поліморфізм?
Відповідь пояснити.

31. Дано визначення класів:

```
class C1
{
};

class C2: public C1
{
};
```

Створити поліморфний об'єкт, що має справжній тип C2. Відповідь пояснити.

32. Дано визначення класів і об'єктів:

```
class C1
{
    virtual void f() {}
};

class C2: public C1
{};

C1 *a = new C2;
```

Навести приклади, як в програмі можна визначити справжній тип об'єкта **a**.

33. Дано визначення класів і об'єктів:

```
class C1
{
    void f() {}
};

class C2: public C1
{};

C1 *a = new C2;
```

Яким буде справжній тип об'єкта **a** ?

34. Дано визначення класів і об'єктів:

```
class C1
{
    virtual void f() {}
};

class C2: public C1
{};

C1 *a = new C2;
```

Яким буде справжній тип об'єкта **a** ?

35. Дано визначення класів і об'єктів:

```
class C1
{
    public:
        int x;
        virtual void f() {}
};

class C2: public C1
{
    public:
        char x;
};

C1 *a = new C2;
```

- a) Написати звертання до того поля **x** об'єкту **a**, яке визначене в класі **C1**. Відповідь пояснити.
- b) Написати звертання до того поля **x** об'єкту **a**, яке визначене в класі **C2**. Відповідь пояснити.

36. Дано визначення класів і об'єктів:

```
class C1
{
    public:
        virtual void f() {}
        void DoIt() {}
};

class C2: public C1
{
    public:
        void DoIt() {}
};

C1 *a = new C2;
```

- a) Написати звертання до того методу **DoIt()** об'єкту **a**, який визначений в класі **C1**. Відповідь пояснити.
- b) Написати звертання до того методу **DoIt()** об'єкту **a**, який визначений в класі **C2**. Відповідь пояснити.

37. Дано визначення класів і об'єктів:

```
class C1
{
    public:
        virtual void DoIt() {}
};

class C2: public C1
{
    public:
        virtual void DoIt() {}
};

C1 *a = new C2;
```

- a) Написати звертання до того методу **DoIt()** об'єкту **a**, який визначений в класі **C1**. Відповідь пояснити.
- b) Написати звертання до того методу **DoIt()** об'єкту **a**, який визначений в класі **C2**. Відповідь пояснити.

Предметний покажчик

A

abstract, 59

R

RTTI, 64

A

Абстрактний клас, 57, 60
Абстрактні деструктори, 51
 необхідність реалізації, 51
Абстрактні методи, 14, 56

B

Віртуалізація зовнішніх функцій, 52
Віртуальні деструктори, 49
 звільнення пам'яті для поліморфних об'єктів, 50
Віртуальні методи, 12, 27
 в конструкторах та деструкторах, 49
 визначення, 13, 29
 невіртуальний виклик, 48
 перевантаження, 38
 перевизначення, 14, 30, 32, 39
 зміна константності, 41
 інший лише тип результату, 48
 інший набір параметрів, 39
 той самий набір параметрів, 42
 розмір класу, 48
 успадкування, 14, 29, 30

D

Динамічний поліморфізм, 12, 17

З

Зв'язування, 12, 17

пізнє зв'язування, 27
 механізм, 28
поняття, 17
раннє зв'язування, 17
 механізм, 24

I

Інтерфейс, 60
 поняття, 60

П

Пізнє зв'язування, 12, 27
Поліморфізм, 17
 поняття, 17
Поліморфні класи, 12, 27
Поліморфні методи, 12, 27, 55
Поліморфні об'єкти, 16, 63
 визначення справжнього типу, 65

P

Раннє зв'язування, 12, 17
Реалізація інтерфейсу
 позначення на UML-діаграмах класів, 60

C

Статичний поліморфізм, 12, 17

T

Таблиця віртуальних методів, 13, 27

Ч

Чисті віртуальні деструктори, 51
 необхідність реалізації, 51
Чисті віртуальні методи, 14, 56

Література

Основна

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня СПб.: Питер, 2007. – 461 с.
2. Павловская Т.А., Щупак Ю.А. С/С++. Объектно-ориентированное программирование: Практикум СПб.: Питер, 2005. – 265 с.
3. Дейтел Х.М., Дейтел П.Дж. Как программировать на С++ М.: Бином-Пресс, 2005. – 1248 с.
4. Уэллин С. Как не надо программировать на С++ СПб.: Питер, 2004. – 240 с.
5. Хортон А. Visual C++ 2005: Базовый курс М.: Вильямс, 2007. – 1152 с.
6. Солтер Н.А., Клеппер С.Дж. С++ для профессионалов. М.: Вильямс, 2006. – 912 с.
7. Лафоре Р. Объектно-ориентированное программирование в С++ СПб.: Питер, 2006. – 928 с.
8. Лаптев В.В. С++. Объектно-ориентированное программирование СПб.: Питер, 2008. – 464 с.
9. Лаптев В.В., Морозов А.В., Бокова А.В. С++. Объектно-ориентированное программирование. Задачи и упражнения СПб.: Питер, 2008. – 464 с.

Додаткова

10. Прата С. Язык программирования С++. Лекции и упражнения СПб.: ДиаСофт, 2003. – 1104 с.
11. Мейн М., Савитч У. Структуры данных и другие объекты в С++ М.: Вильямс, 2002. – 832 с.
12. Саттер Г. Решение сложных задач на С++ М.: Вильямс, 2003. – 400 с.
13. Чепмен Д. Освой самостоятельно Visual C++ .NET за 21 день М.: Вильямс, 2002. – 720 с.
14. Мартынов Н.Н. Программирование для Windows на С/С++ М.: Бином-Пресс, 2004. – 528 с.
15. Паппас К., Мюррей У. Эффективная работа: Visual C++ .NET СПб.: Питер, 2002. – 816 с. М.: Вильямс, 2001. – 832 с.
16. Грэхем И. Объектно-ориентированные методы. Принципы и практика М.: Вильямс, 2004. – 880 с.
17. Элиенс А. Принципы объектно-ориентированной разработки программ М.: Вильямс, 2002. – 496 с.

18. Ларман К. Применение UML и шаблонов проектирования М.: Вильямс, 2002. – 624 с.
19. Шилэт Г. Полный справочник по С. 4-е издание. М.-СПб.-К: Вильямс, 2002.
20. Прата С. Язык программирования С. Лекции и упражнения. М.: ДиаСофтЮП, 2002.
21. Александреску А. Современное проектирование на С++ М.: Вильямс, 2002.
22. Браунси К. Основные концепции структур данных и реализация в С++ М.: Вильямс, 2002.
23. Подбельский В.В. Язык СИ++. Учебное пособие. М.: Финансы и статистика, 2003.
24. Павловская Т.А., Щупак Ю.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2002.
25. Савитч У. Язык С++. Объектно-ориентированного программирования. М.-СПб.-К.: Вильямс, 2001.

<http://www.sql.ru/forum/879377/prosmotr-tablicy-virtualnyh-funkciy>

<https://habrahabr.ru/post/51229/>