О.М. Березький
Ю.М. Батько
М.М. Касянчук
І.З. Якименко
Г.М. Мельник
О.Й. Піцун

“Програмування”

Тернопіль – 2018

Колектив авторів: О.М. Березький, Ю.М. Батько, М.М. Касянчук, І.З. Якименко, Г.М. Мельник, О.Й. Піцун

Рецензенти:

Антощук Світлана Григорівна – д.т.н., професор інституту комп'ютерних систем Одеського національного політехнічного університету.

Теслюк Василь Миколайович – д.т.н., професор, професор кафедри систем автоматизованого проектування Національного університету "Львівська політехніка".

Яровий Андрій Анатолійович - д.т.н., професор, завідувач кафедри комп'ютерних наук, Вінницького національного технічного університету.

Рекомендовано Вченою Радою Тернопільського національного економічного університету, протокол № 3 від 15 грудня 2017 року.

Березький О.М. Програмування: навч. посіб. / О.М. Березький, Ю.М. Батько, М.М. Касянчук, та інші/– Тернопіль: ТНЕУ, 2018. – 260 с.

Посібник адаптований до навчальної дисципліни "Програмування" ОКР бакалавр напрямку підготовки "Комп'ютерна інженерія". В ньому представлено технології проектування програмних продуктів, розглянуто алгоритми вирішення типових задач з використанням структурного та об'єктно-орієнтованого підходів та наведено велику кількість програмних кодів для спрощення сприйняття теоретичного матеріалу.

ПЕРЕДМОВА

Ця книга є не тільки підручником з мови C ++, але й підручник з програмування. В ній розглянуті питання не лише створення програмного коду, але і супутні задачі: розробка алгоритмів, оптимізація коду тощо. Оскільки теорія без практики абсолютно безглузда, то автори намагались максимально проілюструвати теоретичний матеріал за допомогою програмних кодів. У книзі в першу чергу описаний широке коло понять і прийомів програмування, необхідних для того, щоб стати професійним програмістом, і в набагато меншому ступені - можливості мови програмування C ++.

Дана книга в першу чергу призначена для тих, кому хотілося б швидко навчитися писати справжні програми на мові C ++. Найчастіше програмісти-початківців C++ намагаються освоїти мову чисто механічно, навіть не спробувавши дізнатися, як можна ефективно застосувати її можливості для вирішення як повсякденних так і нетривіальних задач.

Автори сподіваються, що даний посібник допоможе опанувати мистецтво створення програмних додатків для програмістів-початківців, а також систематизувати та поглибити свої знання досвідченим спеціалістам.

Автори з вдячність врахують критичні зауваження та побажання фахівців.

ВСТУП

Перед вивченням самих принципів програмування необхідно зрозуміти хто ж такий програміст і його основні вміння. Чим займається Програміст? Програмістами узагальнено називають категорію людей, що займаються розробкою алгоритмів та програм на основі математичних моделей. Програмістів умовно можна розділити на три категорії:

- Прикладні програмісти. Такі фахівці займаються розробкою конкретних програм, необхідних для роботи організації. Наприклад, сюди можна віднести програмістів 1С.
- Системні програмісти програмують операційні системи, інтерфейси до розподіленим баз даних, що працюють з мережами.
- Web-програмісти мають справу з мережами, але, як правило, з глобальними, такими, як Internet. Вони пишуть web-інтерфейси до баз даних, створюють динамічні web-сторінки тощо.

Специфіка професії програміста:

Переваги професії:

- Постійне професійне самовдосконалення,
- Високий попит на ринку,
- Висока заробітна плата,
- Працювати можна не маючи диплома,
- Переважно творча професія.

Мінуси професії:

- Те, що зрозуміло програмісту, не завжди зрозуміло користувачеві доводиться багато пояснювати,
- Трапляється працювати в авральному режимі,
- Робота за комп'ютером погано позначається на здоров'ї,
- І тут знаходиться місце рутині,
- Професія накладає відбиток на характер.

Де працює Програміст:

- Науково-дослідні центри.
- IT-компанії.
- Організації, які в свій структурі увазі відділи програмістів (або штатну одиницю).

Ким працює Програміст:

Успішний починаючий програміст в майбутньому може претендувати на посади:

- керівник групи програмістів;
- IT-директор;
- менеджер проекту;
- можна переміщатися в межах спеціальності, вдосконалюючись професійно.

Особисті якості:

- Перш за все, програміст повинен володіти терпінням і витримкою. Це абсолютно незамінні якості в його роботі.
- Потрібно вміти швидко адаптуватися і постійно вивчати щось нове. Інакше через кілька років ваша цінність як фахівця може помітно знизитися.
- Вміння об'єктивно оцінювати можливості технологій та їх використання в кожному конкретному випадку.

РОЗДІЛ I Основні поняття програмування

Під програмуванням розуміється написання інструкцій на конкретній мові програмування. Відповідно, люди, які цим займаються, називаються програмістами, а ті, хто розробляє алгоритми - алгоритмістами, фахівцями предметної області, математиками.

У більш широкому значенні під програмуванням розуміють весь спектр діяльності, пов'язаний зі створенням і підтримкою в робочому стані програм - програмного забезпечення ЕОМ. Сюди входять аналіз і постановка задачі, проектування програми, побудова алгоритмів, розробка структур даних, написання текстів програм, налагодження і тестування програми (випробування програми), документування, настройка (конфігурація), доробка та супровід. Програмування для ЕОМ ґрунтується на використанні мов програмування, на яких записується програма. Щоб програма могла бути зрозуміла і виконана ЕОМ, потрібен спеціальний інструмент - транслятор.

В даний час активно використовуються інтегровані середовища розробки, що включають в свій склад також редактор для введення і редагування текстів програм, відладчики для пошуку і усунення помилок, транслятори з різних мов програмування, компонувальники для збірки програми з декількох модулів і інші службові модулі.

Текстовий редактор середовища програмування може мати специфічну функціональність, таку як індексація імен, відображення документації, засоби візуального створення користувальницького інтерфейсу. За допомогою текстового редактора програміст проводить набір і редагування тексту створюваної програми, який називають вихідним кодом. Мова програмування визначає синтаксис і початкову семантику вихідного коду. Компілятор перетворює текст програми в машинний код, безпосередньо виконуваний електронними компонентами комп'ютера. Інтерпретатор створює віртуальну машину для виконання програми, яка повністю або частково бере на себе функції виконання програм.

Програмування в широкому сенсі можна розбити на кілька стадій:

- аналіз;
- проектування - розробка комплексу алгоритмів;
- кодування і компіляцію - написання вихідного тексту програми і перетворення його в виконані код за допомогою компілятора;
- тестування і відладку - виявлення та усунення помилок в програмах;
- випробування і здачу програм;
- супровід.

Парадигми програмування.

Усе програмування прийнято поділяти на два основних види:

- Декларативне.

·- Імперативне.

Декларативне програмування - термін з двома різними значеннями. Згідно першому визначенню, програма «декларативна», якщо вона описує щось, а не як його створити. Наприклад, веб-сторінки на HTML декларативні, оскільки вони описують що повинна містити сторінка, а не як відображати сторінку на екрані. Цей підхід відрізняється від мов імперативного програмування, що вимагають від програміста вказувати алгоритм для виконання.

Згідно другому визначенню, програма «декларативна», якщо вона написана на виключно функціональній, логічній або константній мові програмування.

Імператвне програмування — парадигма програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми. Подібно до того, як з допомогою наказового способу в мовознавстві перелічується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп'ютеру для виконання.

Парадигма програмування — це спосіб мислення розробника програми. Мова програмування може підтримувати або не підтримувати ту чи іншу парадигму. В першому випадку застосування парадигми стає зручним, тобто простим, безпечним і ефективним. Ми розглянемо три основних наказових парадигми — процедурне, об’єктне (модульне) і об’єктно-орієнтовне (ієрархічне) програмування.

Перелічимо їх з короткими поясненнями:

імперативне	програма = послідовність дій, пов'язаних умовними і безумовними переходами
процедурне	програма = послідовність процедур, кожна з яких є послідовність елементарних дій і викликів процедур, структурованих за допомогою структурних операторів if , for iwhile
об'єктно-орієнтоване	програма = кілька взаємодіючих об'єктів, функціональність (дії) і дані розподіляються між цими об'єктами
функціональне	прогамма = система визначень функцій, опис того, що потрібно обчислити, а як це зробити - вирішує транслятор; послідовність дій не простежується
продукційне (логічне)	програма = система визначень і правил виду "умова => новий факт"
сентенціальний	програма = система правил виду "шаблон => трансформує дія"
подієве	програма = система правил виду "подія => нові події" + диспетчер подій
автоматне	програма = кінцевий автомат або автомат спеціального типу

Приклади програм з різними парадигмами. Тут розглянемо докладніше різні парадигми програмування, і постараємося привести найбільш яскраві демонструє їх приклади програм.

Імперативне програмування - програмування від "дієслів". Програми являють собою послідовність дій з уловного і безумовними переходами. Програміст мислить у термінах дій і вибудовує послідовності дій в складніші макро-дії (процедури).

функція Вскіп'ятіть_чайник

початок

Запалити плиту;

Взяти чайник;

Налити в чайник води;

Поставити на плиту;

Почекати 5 хвилин;

кінець

Програміст, коли пише такі програми, тримає в голові наявний функціонал - безліч дій, які на поточний момент реалізовано в програмі і бібліотеках.

Крім того, він знає вхідні умови (prerequisites) для кожної дії і намагається їх задовольнити. Перевірку цих умов корисно виробляти всередині самої процедури в самому її початку.

Крім того для самоконтролю корисно перед поверненням здійснювати перевірку всіх вихідних умов (postrequisites), щоб переконається, що необхідне макро-дію було вироблено та вироблено без помилок.

Це дозволяє відловити багато помилок на етапі тестування програми.

Але в індустрії програмування дуже активна діяльність йде в напрямку розробки інструментів відлову помилок на етапі компіляції програми, а не в момент її запуску.

Чим імперативне програмування відрізняється від процедурного?

Нічим. Коли говорять про процедурному програмуванні, хочуть підкреслити метасистемний перехід від елементарних дій до більш високорівневих діям, представлених процедурами і функціями.

Термін структурне програмування ввів Е.Дейкстра в 1975 році: Можна розглядати структурне програмування як стиль написання програм, що виключає оператор goto.

Сьогодні багато помилково структурне програмування інтерпретують як процедурне або модульне програмування. Частково це так. Саме ж головне в структурному програмуванні - це правильне складання правильної логічної схеми програми, реалізація якої мовними засобами - справа вторинна. Програма повинна являти собою безліч вкладених блоків (або по-іншому - ієрархії блоків), кожен з яких має один вхід і один вихід. При цьому передача управління між блоками і операторами на кожному рівні дерева виконується послідовно.

По-великому рахунку подієва модель програм організована точно також, просто роль сполучної блоку верхнього рівня виконує ядро системи.

Процедурне програмування.

Процедурне програмування базується на такому принципі.

Програміст повинен визначити потрібні процедури і використати найкращі алгоритми.

Головний акцент в процедурному програмуванні робиться на обробці, тобто на алгоритмі. Основним механізмом процедурного програмування є функція. Процес розв'язання задачі в рамках процедурної парадигми називається функціональною абстракцією. Від дозволяє розробляти функції відносно ізольовано одну від одної, налагоджуючи зв'язки між ними за допомогою механізму передачі параметрів і повертання результатів.

Процедурне програмування подає програму у вигляді набору алгоритмів, для оформлення яких можуть застосовуватися іменовані програмні блоки — процедури і функції. В останньому випадку

передбачається наявність механізмів передачі параметрів і поверненні результату.

Спочатку процедурне програмування користувалося довільними засобами керування, в тому числі, переходом за міткою — одним з найбільш вживаних операторів керування в Фортрані.

До мов процедурного програмування відносяться Fortran, Cobol, Pascal, Basic, та інші.

В 1968 році голландський вчений Е. Дейкстра вперше звернув увагу на проблеми, що виникають у програмах з неконтрольованими переходами, в 1970 році проголосив новий напрямок, який він назвав структур(ова)ним програмуванням.

Структурне програмування — це варіант процедурного, що вживає три типи структур керування: послідовне виконання дій, розгалуження і цикл. Не дивно, що Фортран не підтримував цю парадигму — в наборі його засобів не було циклів за умовами. Починаючи з Алголу, а особливо в Паскалі, цикли стають основним засобом організації обчислень в програмі.

Автор Паскалю, професор Н. Вірт, відібрав до створюваної ним мови програмування лише прості в поясненні і легкі в реалізації конструкції. Завдяки сильній типізації програми в Паскалі відзначаються високою надійністю, вони мобільні завдяки закладеній в них концепції Паскаль-машини, їх легко читати і розуміти завдяки дисципліні програмування, продиктованої вжитою парадигмою.

Але разом з цим застосування Паскалю гальмувалося саме складністю виходу за межі віртуальної машини, потребою ефективного використання наявної апаратури. Головним критерієм, вжитим Б.Керніганом і Д.Річі до створеної ними мови С, стала саме гнучкість використання особливостей конкретної апаратури і ефективність виконання програм.

Припустимо, що розробляється програма, яка сортує список. Ми подаємо список у вигляді масиву і пишемо функцію, яка сортує його методом швидкого сортування. Дотримуючись принципу процедурного

програмування, визначимо потрібні процедури: ввід — швидке сортування — вивід. Все, що нам потрібно, — визначити спосіб вводу, параметри функції сортування і вигляд виводу. Якщо нам знадобиться в подальшому змінити програму, наприклад, застосувати кращий метод сортування, її доведеться перекомпілювати заново.

Модульне, або об'єктне програмування більше уваги приділяє не проектуванню процедур, а організації структур даних. Це пов'язано із збільшенням розмірів програм. Принцип модульного програмування формулюється так: програміст повинен вирішити, які потрібні модулі, а потім розбити їх так, щоб приховати дані в цих модулях.

Цей принцип також називається принципом приховання даних, або інкапсуляцією. Модулем називається набір зв'язаних процедур разом з даними, що піддаються обробці. Основним механізмом модульного програмування є простори імен і абстракція даних.

Абстракція даних зосереджує увагу на призначенні операцій, а не на деталях їх виконання. Інші модулі програми "знають", що робить та чи інша операція, але не "знають", як саме вона це робить. Абстракція даних використовує два поняття: абстрактний тип, тобто сукупність даних і операцій над ними, і структуру даних, тобто конструкцію, що визначена в мові програмування для зберігання набору даних.

Повертаючись до попереднього прикладу, уявімо, що наша програма — маленька частина великого проекту. Природно зробити так, щоб її модифікація мінімально впливала на решту модулів. Для цього потрібна її максимальна ізоляція і захист. В такому випадку, ми можемо приховати алгоритм в модулі, а сам модуль розмістити в бібліотеці, де його легко замінити. Основним механізмом модульного програмування і абстракції даних є типи, визначені користувачем — класи.

Об'єктно-орієнтований підхід в програмуванні.

Процедурна парадигма віддала належне алгоритмічній компоненті програмування. Але з ростом обсягу програм і складності даних з'явилася нова проблема структурної організації даних, найбільш ємко висловлена Віртовською формулою “алгоритми + структури даних = програми”.

Поняття модуля як абстракції даних було вперше запропоноване Парнасом у 1972 році, правда на той час уже існувала мова програмування Симула 67, в якій використовувалася парадигма об'єктів. У найбільш повному виді поняття абстракції даних було реалізоване в мові програмування Модула-2.

Головна ідея полягає в забезпеченні доступу до даних, не залежному від їх конкретного представлення. Самі дані і програми їх обробки вбудовуються (інкапсулюються) в окремій одиниці програми.

Імперативне і об’єктне програмування не відрізняються підвищеною гнучкістю. Чим більшими стають програми, тим складнішими стають способи узгодження взаємозв’язку між їхніми компонентами. Для організації розробки великих програм була запропонована об’єктно-орієнтована парадигма, яка доповнює інкапсуляцію ще двома принципами — успадкуванням і поліморфізмом.

Об'єктно-орієнтована парадигма наділила класи ієрархією. Об'єктно-орієнтоване програмування за метафорою Б.Страуструпа, автора С++ — однієї з найпопулярніших мов об'єктно-орієнтованого програмування, — це високоінтелектуальний синонім доброго програмування. Дійсно, нові парадигми програмування з'являються не так часто, не частіше однієї в десятиліття. Той факт, що об'єктно-орієнтована парадигма успішно використовується протягом 20 років, сам по собі служить вагомим підтвердженням її життєздатності.

Алгоритми, реалізовані в процедурному програмуванні, надто конкретні. Будь-яка модифікація — це вже новий алгоритм і таким чином кількість процедур і функцій, що знаходяться у використанні, надмірно

зростає. Модульне програмування групує алгоритми в модулі, одночасно інкапсулюючи структури даних. Тепер залишається зробити наступний крок — побудувати ієрархію модулів або класів.

Таких ієрархій може бути дві. Перша з них — бути частиною чогось. Наприклад, грань є частиною многогранника, ребро — частиною грані, вершина — частиною ребра. Інша ієрархія — бути узагальненням або конкретизацією. Наприклад, овал і многокутник служать конкретизацією плоскої фігури, коло — конкретизацією овалу, чотирикутник — конкретизацією многокутника, подальшими конкретизаціями чотирикутника можуть служити паралелограм, прямокутник, ромб, квадрат. Той факт, що квадрат, ромб, прямокутник є повноцінними паралелограмами дозволяє їм користуватися усіма програмними засобами, створеними для паралелограма, паралелограм в свою чергу є повноцінним чотирикутником і так далі. Цей принцип, відомий під назвою reusable — знову вживаний — став одним з найважливіших досягнень об'єктно-орієнтованої парадигми. Знову вживаючи вже існуюче програмне забезпечення в більш конкретизованих умовах, ми дописуємо лише ту його частину, яка стосується особливостей наявної конкретизації. Цей принцип дістав назву programming by difference або дописування програм.

І, нарешті, об'єктно-орієнтована парадигма доводить до логічної завершеності принцип моделювання реального світу, а точніше тієї його частини, абстракцією якої служить програма. При цьому підході програма складається з об'єктів, що відповідають реальним поняттям або предметам. Виконання програми зводиться до взаємодії об'єктів, яке служить абстракцією реальної взаємодії їх прототипів. Все це разом забезпечило об'єктно-орієнтованому підходу беззаперечне лідерство в галузі розробки програм.

Інкапсуляція полягає в тому, що об’єкти об’єднують дані і операції в одне ціле.

Успадкування дозволяє класам набувати властивості інших класів.

Поліморфізм дозволяє об’єктам обирати відповідні операції під час виконання програми.

Об'єктно-орієнтований підхід має такі переваги, як:

- зменшення складності програмного забезпечення;
- підвищення надійності програмного забезпечення;
- забезпечення можливості модифікації окремих компонентів програмного забезпечення без зміни інших його компонентів;
- забезпечення можливості повторного використання окремих компонентів програмного забезпечення.

Більш детально переваги і недоліки об'єктно-орієнтованого програмування будуть розглянуті в кінці лекції, так як для їх розуміння необхідне знання основних понять і положень ООП.

Систематичне застосування об'єктно-орієнтованого підходу дозволяє розробляти добре структуровані, надійні в експлуатації, досить просто модифікуються програмні системи. Цим пояснюється інтерес програмістів до об'єктно-орієнтованого підходу й об'єктно-орієнтованим мовам програмування. ООП є одним з напрямків теоретичного і прикладного програмування, що найбільш інтенсивно розвивається.

Сьогодні в сімействі мов об'єктно-орієнтованого програмування три найбільш відомих представника: C++, Java і C # (читається Сі шарп). C++ і сьогодні залишається визнаним лідерів в розробці великих і складних програмних систем. Java і C # виросли з C++. Вони мають свою сферу застосування в розподіленому програмуванні і будуть вивчатися нами пізніше.

Основи алгоритмізації та програмування

Поняття алгоритму

Одним з фундаментальних понять в інформатиці є поняття алгоритму. Походження самого терміна "алгоритм" пов'язане з математикою. Це слово походить від Algorithmi - латинського написання імені Мухаммеда аль-хорезми (787 - 850) видатного математика середньовічного сходу. Визначення алгоритму не можна вважати строгим - не цілком ясно, що таке "точне приписання" або "послідовність дій, що забезпечує одержання необхідного результату". Тому звичайно формулюють кілька загальних властивостей алгоритмів, що дозволяють відрізняти алгоритми від інших інструкцій.

Такими властивостями є:

- Зрозумілість. Щоб виконавець міг досягти поставленої перед ним мети, використовуючи даний алгоритм, він повинен уміти виконувати кожну його вказівку, тобто розуміти кожну з команд, що входять до алгоритму.

- Дискретність (перервність, роздільність) - алгоритм повинен представляти процес рішення завдання як послідовне виконання простих (або раніше певних) кроків. Кожна дія, передбачена алгоритмом, виконується тільки після того, як закінчилося виконання попередні.

- Визначеність - кожне правило алгоритму повинне бути чітким, однозначним і не залишати місця для сваволі. Завдяки цій властивості виконання алгоритму носить механічний характер і не вимагає ніяких додаткових вказівок або відомостей про розв'язуване завдання.

- Результативність (кінцівка) - алгоритм повинен приводити до рішення завдання за кінцеве число кроків.

- Масовість - алгоритм рішення завдання розробляється в загальному виді, тобто, він повинен бути застосуємо для деякого класу завдань, що розрізняються тільки вихідними даними. При цьому вихідні дані можуть

вибиратися з деякої області, що називається областю застосовності алгоритму.

На підставі цих властивостей іноді дається визначення алгоритму, наприклад:

"Алгоритм - це послідовність математичних, логічних або разом узятих операцій, що відрізняються детермінованістю, масовістю, спрямованістю й, що приводить до рішення всіх завдань даного класу за кінцеве число кроків".

Проте і таке трактування поняття "алгоритм" є неповним й неточним. По-перше, невірно зв'язувати алгоритм із рішенням якого-небудь завдання. Алгоритм взагалі може не вирішувати ніякого завдання. По-друге, поняття "масовість" ставиться не до алгоритмів як до таким, а до математичних методів у цілому. Рішення поставлених практикою завдань математичними методами засновано на абстрагуванні – ми виділяємо ряд істотних ознак, характерних для деякого кола явищ, і будуємо на підставі цих ознак математичну модель, відкидаючи несуттєві ознаки кожного конкретного явища. У цьому змісті будь-яка математична модель має властивість масовості. Якщо в рамках побудованої моделі ми вирішуємо завдання й рішення представляємо у вигляді алгоритму, то рішення буде "масовим" завдяки природі математичних методів, а не завдяки "масовості" алгоритму.

З іншої сторони вислів "властивості алгоритму" некоректне. Властивостями володіють об'єктивно існуючі реальності. Можна говорити, наприклад, про властивості якої-небудь речовини. Алгоритм - штучна конструкція, що ми споруджуємо для досягнення своїх цілей. Щоб алгоритм виконав своє призначення, його необхідно будувати за певними правилами. Тому потрібно говорити не про властивості алгоритму, а про правила побудови алгоритму, або про вимоги, пропонованих до алгоритму.

Перше правило - при побудові алгоритму насамперед необхідно задати множину об'єктів, з якими буде працювати алгоритм. Формалізоване (закодоване) подання цих об'єктів зветься даних. Алгоритм приступає до

роботи з деяким набором даних, які називаються вхідними, і в результаті своєї роботи видає дані, які називаються вихідними. Таким чином, алгоритм перетворить вхідні дані у вихідні.

Це правило дозволяє відразу відокремити алгоритми від "методів" й "способів". Поки ми не маємо формалізованих вхідних даних, ми не можемо побудувати алгоритм.

Друге правило - для роботи алгоритму потрібна пам'ять. У пам'яті розміщаються вхідні дані, з якими алгоритм починає працювати, проміжні дані й вихідні дані, які є результатом роботи алгоритму. Пам'ять є дискретної, тобто складається з окремих осередків. Пойменована комірка пам'яті зветься змінної. У теорії алгоритмів розміри пам'яті не обмежуються, тобто вважається, що ми можемо надати алгоритму будь-який необхідний для роботи обсяг пам'яті.

Третє правило - дискретність. Алгоритм будується з окремих кроків (дій, операцій, команд). Безліч кроків, з яких складений алгоритм, звичайно.

Четверте правило - детермінованість. Після кожного кроку необхідно вказувати, який крок виконується наступної, або давати команду зупинки.

П'яте правило - збіжність (результативність). Алгоритм повинен завершувати роботу після кінцевого числа кроків. При цьому необхідно вказати, що вважати результатом роботи алгоритму.

Отже, алгоритм - невизначуване поняття теорії алгоритмів. Алгоритм кожному певному набору вхідних даних ставить у відповідність деякий набір вихідних даних, тобто обчислює (реалізує) функцію. При розгляді конкретних питань у теорії алгоритмів завжди мається на увазі якась конкретна модель алгоритму.

При рішенні будь-якого математичного завдання ми становимо алгоритм рішення. Але колись ми самі й виконували цей алгоритм, тобто доводили рішення до відповіді. Тепер же ми будемо тільки писати, що потрібно зробити, але обчислення проводити не будемо. Обчислювати буде

комп'ютер. Наш алгоритм буде являти собою набір вказівок (команд) комп'ютеру.

Коли ми обчислюємо яку-небудь величину, ми записуємо результат на папері. Комп'ютер записує результат своєї роботи на згадку у вигляді змінної. Тому кожна команда алгоритму повинна включати вказівку, у яку змінну записується результат.

Трактування роботи алгоритму як перетворення вхідних даних у вихідні природно підводить нас до розгляду поняття "постановка завдання". Для того щоб скласти алгоритм рішення завдання, необхідно з умови виділити ті величини, які будуть вхідними даними й чітко сформулювати, які саме величини потрібно знайти. Інакше кажучи, умова завдання потрібно сформулювати у вигляді "Дано ... Потрібно" - це і є постановка завдання.

Алгоритм стосовно до обчислювальної машини - точне приписання, тобто набір операцій і правил їхнього чергування, за допомогою якого, починаючи з деяких вихідних даних, можна вирішити будь-яке завдання фіксованого типу.

Класифікація алгоритмів

Види алгоритмів як логіко-математичних засобів відбивають зазначені компоненти людської діяльності й тенденції, а самі алгоритми залежно від мети, початкових умов завдання, шляхів її рішення, визначення дій виконавця підрозділяються в такий спосіб:

Механічні алгоритми, або інакше детерміновані, тверді (наприклад, алгоритм роботи машини, двигуна тощо). Механічний алгоритм задає певні дії, позначаючи їх у єдиній і достовірній послідовності, забезпечуючи тим самим однозначний необхідний або шуканий результат, якщо виконуються ті умови процесу, завдання, для яких розроблений алгоритм.

Гнучкі алгоритми, наприклад стохастичні, тобто імовірнісні й евристичні.

Імовірнісний (стохастичний) алгоритм дає програму рішення завдання декількома шляхами або способами, що приводять до ймовірного досягнення результату.

Евристичний алгоритм (від грецького слова "еврика") - це такий алгоритм, у якому досягнення кінцевого результату програми дій однозначно не визначено, так само як не позначена вся послідовність дій, не виявлені всі дії виконавця. До евристичних алгоритмів відносять, наприклад, інструкції й приписання. У цих алгоритмах використаються універсальні логічні процедури й способи прийняття рішень, засновані на аналогіях, асоціаціях і минулому досвіді рішення схожих завдань.

Лінійний алгоритм - набір команд (вказівок), виконуваних послідовно в часі один за одним.

Алгоритм розгалуження - алгоритм, що містить хоча б одна умова, у результаті перевірки якого ЕОМ забезпечує перехід на один із двох можливих кроків.

Циклічний алгоритм - алгоритм, що передбачає багаторазове повторення того самого дії (тих самих операцій) над новими вихідними даними. До циклічних алгоритмів зводиться більшість методів обчислень, перебору варіантів.

Цикл програми - послідовність команд (серія, тіло циклу), що може виконуватися багаторазово (для нових вихідних даних) до задоволення деякої умови.

Допоміжний (підлеглий) алгоритм (процедура) - алгоритм, раніше розроблений і цілком використовуваний при алгоритмізації конкретного завдання. У деяких випадках при наявності однакових послідовностей вказівок (команд) для різних даних з метою скорочення запису також виділяють допоміжний алгоритм.

При рішенні завдань на комп'ютері необхідно не стільки вміння становити алгоритми, скільки знання методів рішення завдань (як і взагалі в математику). Тому вивчати потрібно не програмування як таке (і не

алгоритмізацію), а методи рішення математичних завдань на комп'ютері. Завдання варто класифікувати не по типах даних, як це звичайно робиться (завдання на масиви, на символьні змінні тощо), а по розділі "Потрібно".

В інформатиці процес рішення завдання розподіляється між двома суб'єктами: програмістом і комп'ютером. Програміст становить алгоритм (програму), комп'ютер його виконує. У традиційній математиці такого поділу ні, завдання вирішує одна людина, що становить алгоритм рішення завдання й сам виконує його. Сутність алгоритмізації не в тім, що рішення завдання представляється у вигляді набору елементарних операцій, а в тім, що процес рішення завдання з на два етапи: творчий (програмування) і не творчий (виконання програми). І виконують ці етапи різні суб'єкти - програміст і виконавець.

Способи задання алгоритмів

Першій спосіб – це словесний опис алгоритму. Сьогодні розібрано вже кілька алгоритмів, і всі вони подавалися виконавцю за допомогою словесного опису.

Другий спосіб - це подача алгоритму у вигляді таблиць, формул, схем, малюнків тощо. Наприклад, всіх вас вчили правилам поведінки на дорозі. І найкраще сприймають алгоритм, що поданий у вигляді схематичних малюнків. Дивлячись на них, людина, відпрацьовує ту лінію поведінки, що їй пропонується. Аналогічно можна навести приклади алгоритмів, що записані у вигляді умовних позначок на купленому товарі, щодо його користування (заварювання чаю, прання білизни тощо). В математиці наявність формул дозволяє розв'язати задачу, навіть "не використовуючи слів".

Третій спосіб - запис алгоритмів за допомогою блок-схеми.

Блок-схемою алгоритму називається геометричне подання операторної схеми алгоритму, в якому оператори відображаються у вигляді геометричних фігур, а послідовність їх виконання вказується стрілками.

При побудові блок-схеми алгоритму прийнята стандартна система геометричних фігур, що використовуються для різних видів операторів. В таблиці 1.1 приведені основні оператори і їх геометричне подання, що відповідає прийнятому стандарту.

Виконавець алгоритмів - це пристрій, що виконує алгоритми. Одним з найбільш гнучких й універсальних виконавців алгоритмів, поряд з людиною, є комп'ютер. Коли програміст розробляє алгоритм, то при цьому він є першим його виконавцем.

Самі основні частини комп'ютера з погляду проектування алгоритмів - це процесор й оперативна пам'ять.

Процесор є активною частиною комп'ютера й виконує дії з даними.

Оперативна пам'ять необхідна для зберігання даних й алгоритмів, з якими комп'ютер працює в цей момент.

Для успішного створення програм потрібно використати технологію програмування. Технологія складається з набору правил, які необхідно строго дотримувати.

Одним із правил є проектування програми, що полягає в попередній побудові її алгоритму. Алгоритм будується й записується на папері, досвідчені програмісти можуть тримати прості алгоритми в розумі.

Наочним зображенням алгоритмів є блок-схема. Блок-схема, або діаграма, кодує алгоритм наочними графічними засобами. Вона складається з наступних графічних елементів:

1) овалів, що кодують початок і кінець, вхід і вихід з алгоритму;

2) паралелограмів, що описують уведення й висновок даних;

3) прямокутників, у яких описується дія з даними;

4) ромбів, що визначають перевіряють алгоритмом умови.

Ці фігури з'єднуються лініями зі стрілками, що вказують послідовність дій.

Таблиця 1.1 – Графічні позначення на блок-схемах

Найменування	Позначення	Функція
1	2	3
Термінатор		Елемент відображає вхід із зовнішнього середовища або вихід з неї (найчастіше застосування - початок і кінець програми). Всередині фігури записується відповідна дія.
Процес		Виконання однієї або кількох операцій, обробка даних будь-якого виду (зміна значення даних, форми подання, розташування). Всередині фігури записують безпосередньо самі операції.
Дані		Перетворення у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Цей символ не визначає носія даних (для вказівки типу носія даних використовуються специфічні символи).
Межа циклу		Символ складається з двох частин - відповідно, початок і кінець циклу - операції, що виконуються всередині циклу, розміщуються між ними. Умови циклу і збільшення записуються всередині символу початку або кінця циклу - в залежності від типу організації циклу. Часто для зображення на блок-схемі циклу замість цього символу використовують символ рішення, вказуючи в ньому умову, а одну з ліній виходу замикають вище в блок-схемі (перед операціями циклу).

Продовження таблиці 1.1

1	2	3
Зумовлений процес		Символ відображає виконання процесу, що складається з однієї або кількох операцій, що визначені в іншому місці програми (у підпрограмі, модулі). Всередині символу записується назва процесу і передані в нього дані.
Рішення		Показує рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення умов, визначених всередині цього елементу. Вхід в елемент позначається лінією, що входить зазвичай у верхню вершину елементу. Якщо виходів два чи три то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижній).
З'єднувач		Символ відображає вихід в частину схеми і вхід з іншої частини цієї схеми. Використовується для обриву лінії та продовження її в іншому місці (приклад: поділ блок-схеми, що не поміщається на листі). Відповідні сполучні символи повинні мати одне (при тому унікальне) позначення.
Коментар		Використовується для детальнішої інформації про кроки, процесу або групи процесів. Опис поміщається з боку квадратної дужки і охоплюється нею по всій висоті. Пунктирна лінія йде до описуваного елементу, або групи елементів (при цьому група виділяється замкнутою пунктирною лінією).

Просте слідування.

Слідування означає, що дії повинні виконуватись послідовно одна за одною.

Приклад, алгоритм знаходження суми S трьох чисел a,b,c (рисунок 1.1).

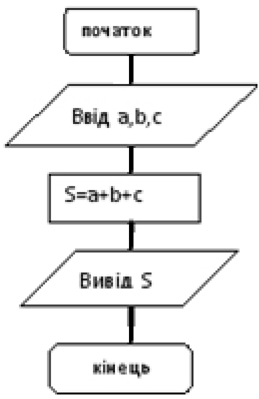

Рисунок 1.1 – Приклад блок-схеми лінійного алгоритму

Розгалуження

Розгалуження – це така форма організації дій, які містять умови і в залежності від того чи вона виконується чи ні здійснюється або одна або друга послідовність дій (рисунок 1.2).

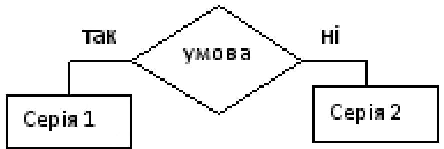

Рисунок 1.2 – Приклад алгоритму розгалуження

Умова - це будь-яке твердження, яке або виконується або не виконується, тобто можна дістати одну з двох відповідей: «так» або «ні». Блок-схемою це можна зобразити так:

Якщо умова виконується, то виконується серія команд 1 (гілка так), якщо умова не виконується, то виконується команд 2 (гілка ні). Після

виконання серії команд виконавець переходить до наступної команди після команди розгалуження.

В серію може входити також команда розгалуження. В цьому випадку кажуть, що команди розгалуження вкладені одна в одну.

Можливий випадок, що у випадку невиконання умови не потрібно виконувати ніяких дій. Тоді використовується скорочена форма розгалуження («Якщо-то») (рисунок 1.3).

Рисунок 1.3 – Приклад блок-схеми алгоритму розгалуження з одною умовою

Повторення(цикл).

Часто зустрічаються такі задачі при виконанні яких потрібно виконувати одні і ті самі дії декілька разів. Тоді кажуть, що така структура команд називається циклічною, або утворена структура «повторення».

Цикл – це форма організації дій, за якою одна і та сама послідовність дій виконується кілька разів доти, поки виконується деяка умова. Серія команд, що виконується декілька разів без змін при кожному проході циклу, називається тілом циклу.

В загальному випадку є два типи повторень: з передумовою та післяумовою. У першому випадку спочатку перевіряється умова і, якщо вона істинна, то вказана дія виконується черговий раз, якщо ж ні – то виконання дії припиняється.

На практиці алгоритми розв’язування складних задач містять у собі всі три типи базових структур алгоритмів. Розглянуті принципи конструювання алгоритмів називають принципами структурного програмування.

Блок-схеми, як й алгоритми, також мають ієрархічну структуру. У блок-схемі, що описує алгоритм рішення якого-небудь завдання, на місце прямокутників підставляються блок-схеми алгоритмів, що описують рішення подзадач цього завдання.

При виконанні алгоритму по його схемі рухається виконавець, що перебуває в кожен момент у якімсь місці алгоритму, називаному активною крапкою. Для перевірки правильності простого алгоритму досить його протестувати, тобто пройти алгоритм, моделюючи виконавця.

При тестуванні алгоритму беруть прості вхідні дані й по них розраховують результат. Потім вручну проходять алгоритм й одержують його вихідні дані. Якщо отримані вихідні дані збіглися з розрахованим результатом, то алгоритм, може бути, побудований правильно.

Графічне зображення алгоритму широко використається перед програмуванням завдання внаслідок його наочності, тому що зорове сприйняття звичайно полегшує процес написання програми, її коректування при можливих помилках, осмислювання процесу обробки інформації.

Контрольні запитання:

1) Дайте визначення алгоритму.
2) Назвіть види алгоритмів.
3) Назвіть основні види циклічних алгоритмів.
4) Перерархуйте властивості алгоритмів.
5) Назвіть способи задання алгоритмів.
6) Перерахуйте основні графічні позначення при запису алгоритмів за допомогою блок-схем.

Поняття мови програмування

З чого починають вчити мову програмування. З того ж, з чого починають вивчати довільну розмовну мову – з алфавіту та перших обов'язкових слів.

Програмування це процес створення програми, або послідовності інструкцій, а мова програмування буде тим засобом, який допоможе нам у спілкуванні з ЕОМ.

Рівні мов програмування

Різні типи процесорів мають різні набори команд. Якщо мова програмування орієнтована на конкретний тип процесора і враховує його особливості, то вона називається мовою програмування низького рівня. У даному випадку "низький рівень" не значить "поганий". Мається на увазі, що оператори мови близькі до машинного коду й орієнтовані на конкретні команди процесора.

Мовою самого низького рівня є мова Асемблера, яка представляє кожну команду машинного коду за допомогою символьних позначень, яке називається мнемонікою. Однозначне перетворення однієї машинної інструкції в одну команду Асемблера називається трансляцією.

На мові низького рівня створюються дуже ефективні і компактні програми, так як розробник отримує доступ до всіх можливостей процесора. З іншого боку потрібно дуже добре розуміти будову комп'ютера. Такі мови використовуються для написання невеличких системних додатків, драйверів пристроїв, та ін., тобто коли вимагаються компактність, висока швидкодія і можливість прямого доступу до апаратних ресурсів.

Мови програмування високого рівня значно ближчі і зрозуміліші людині, ніж комп'ютеру. Особливості конкретних комп'ютерних архітектур в них не враховуються. Розробляти програми значно простіше, помилок значно менше.

Ознаки мов програмування високого рівня

Мова програмування (англ. Programming language) — це штучна мова, створена для передачі команд машинам, зокрема комп'ютерам.

Мови програмування високого рівня грають роль засобу зв’язку між програмістом і машиною, а також між програмістами. Ця обставина накладає на мову багато обов’язків:

1.Мова повинна бути близькою до тих фрагментів природних мов, які забезпечують конкретну предметну область діяльності людини; (Мова, яка орієнтована на ділові сфери використань, повинна містити поняття, які використовуються у цьому виді діяльності: рахунок, база даних і т.п.).

2.Всі засоби мови повинні бути формалізовані у такому степені, щоб їх можна було реалізувати як машинні програми; (наприклад, речення “Знайти документ X у базі Y” повинно генерувати програму в машинній мові, яка здійснює потрібний пошук).

3.Мова програмування не тільки підтримує предметно-орієнтовну діяльність , але і стимулює її розвиток (поняття бази даних, обчислювальної мережі привело до революції у діловій діяльності).

4.Мова програмування – дещо більше, ніж засіб опису алгоритмів: він несе в собі систему понять, на основі яких людина може обдумувати свої задачі, і нотацію, за допомогою якої він може виразити свої розуміння з приводу рішення задачі.

Вивчаючи нову мову програмування, краще всього до неї відноситися, як до будь-якої іншої іноземної мови, без постійної практики – не можливо отримати хороші результати.

Покоління мов програмування

Мови програмування прийнято ділити на п'ять поколінь. В перше покоління входять мови, які були створенні на початку 50-х років, коли з'явилися перші комп'ютери. Це була перша мова Асемблера, створена по принципу "одна інструкція - один рядок".

Розквіт другого покоління мов програмування припав на кінець 50-х - початок 60-х років. Тоді був розроблений Асемблер, у якому появилося поняття змінної. Він став першою повноцінною мовою програмування. Завдяки його появі відчутно зросли швидкість розробки і надійність програм.

Появу третього покоління мов програмування прийнято відносити на 60-і роки. В цей час з'явилися універсальні мови програмування високого рівня. Такі якості нових мов, як відносна простота, незалежність від конкретного комп'ютера і можливість використовувати потужні синтаксичні конструкції, дозволили різко підвищити продуктивність праці програмістів. Зрозуміла структура мов програмування привернула увагу спеціалістів із некомп'ютерних областей до написання невеличких програм. Багато мов програмування цього покоління використовується і тепер.

З початку 70-х років до теперішнього часу продовжується період мов програмування четвертого покоління. Ці мови призначені для реалізації великих проектів. Вони зазвичай орієнтовані на спеціалізовані області використання, де гарних результатів можна досягти, використовуючи не універсальні, проблемно-орієнтовані мови, що оперують конкретними поняттями вузької предметної області. Як правило, в ці мови вбудовуються потужні оператори, що дозволяють одним рядком описати такі функції, для реалізації яких на мовах молодших поколінь потребувалось би тисячі рядків вихідного коду.

Народження мов програмування п'ятого покоління відбулося в середині 90-х років. Це системи автоматичного створення прикладних програм за допомогою візуальних засобів розробки без знання програмування. Головна ідея цих мов програмування - можливість автоматичного формування результуючого тексту на універсальних мовах програмування (який потім потрібно відкомпілювати). Інструкції вводяться в комп'ютер в максимально наглядному вигляді з допомогою методів, найбільш зручних для людини, не знайомої з програмуванням.

Огляд мов програмування високого рівня

Фортран (50-і роки) - перша мова високого рівня, створена Джимом Бекусом. Фортран продовжує активно використовуватися в багатьох організаціях. Зараз завершується робота по створенню чергового стандарту Фортрану F2K.

Кобол (60-і роки) - використовується в економічних областях. На цій мові створено дуже багато додатків, які активно експлуатуються сьогодні. Найбільшу зарплату в США отримують програмісти, які створюють програми на мові програмування Кобол.

Алгол (60-і роки) - не використовується. Версія Алгол-68 по своїм можливостям і сьогодні випереджає багато мов програмування, але із-за більш складної структури ніж Фортран не отримав широкого розповсюдження.

Паскаль (70-і роки) - створений основоположником багатьох ідей сучасного програмування Ніклаусом Віртом, багато в чому нагадує Алгол, дозволяє з успіхом використовувати його для створення великих проектів.

Бейсік (60-і роки) - для мови є компілятори і інтерпретатори. По популярності займає перше місце у світі.

C - планувався для заміни асемблера, щоб мати можливість створювати ефективні і компактні програми, і у той же час не залежати від конкретного типу процесора. Схожий на Паскаль. На цій мові у 70і роки написано багато прикладних і системних програм і ряд відомих операційних систем (Unix).

C++ - об'єктно-орієнтоване розширення мови C (1980 рік). Нові можливості мови дозволили підвищити продуктивність програмістів.

Java (90-і роки) - на основі C++ шляхом виключення низькорівневих можливостей. Займає сьогодні друге місце по популярності після мови програмування Бейсік. Напрямки використання - підтримка мобільних пристроїв і мікрокомп'ютерів, що вбудовуються у побутову техніку та створення програм для глобальних та локальних мереж.

Із універсальних мов програмування найбільш популярними є:

-Бейсік - для засвоєння потрібна початкова підготовка (загальноосвітня школа);

- Паскаль - потребує спеціальної підготовки;

- C++, Java - потребують професійної підготовки.

Мови програмування баз даних, мови програмування для Інтернету являються інтерпретаторами, які розповсюджуються безкоштовно, а самі програми - у вихідних текстах. Такі мови називають скрипт-мовами (HTML, Perl тощо).

Поняття про систему програмування

Розглянемо послідовність змін, що відбуваються з програмою під час виконання (у процесі компіляції).

Текст програми, який написаний мовою програмування, називається вихідним модулем. Достатньо складні програми можуть складатися з кількох модулів, взаємодіючих один з одним. Вихідний модуль – це вхідний потік для програми-компілятора, яка:

– здійснює лексичний аналіз вхідного потоку [блок лексичного аналізу];

– здійснює синтаксичний аналіз вхідного потоку [блок синтаксичного аналізу];

– генерує машинний код, який є перекладом вихідного модуля на мову комп’ютера в умовних адресах [генератор коду].

В результаті цих перетворень на виході отримується об’єктний модуль.

Навіть якщо в програмі один вихідний модуль, для успішного виконання програми необхідно “зв’язати” його з деякими іншими програмами (наприклад, з стандартними процедурами введення-виведення, які реалізовані у мові). Ці функції виконує програма – редактор зв’язків. Вихідний потік цієї програми називають завантаженим модулем.

Процес перетворення тексту вихідного модуля в модуль, який виконується, можна зобразити схематично (рисунок 1.4):

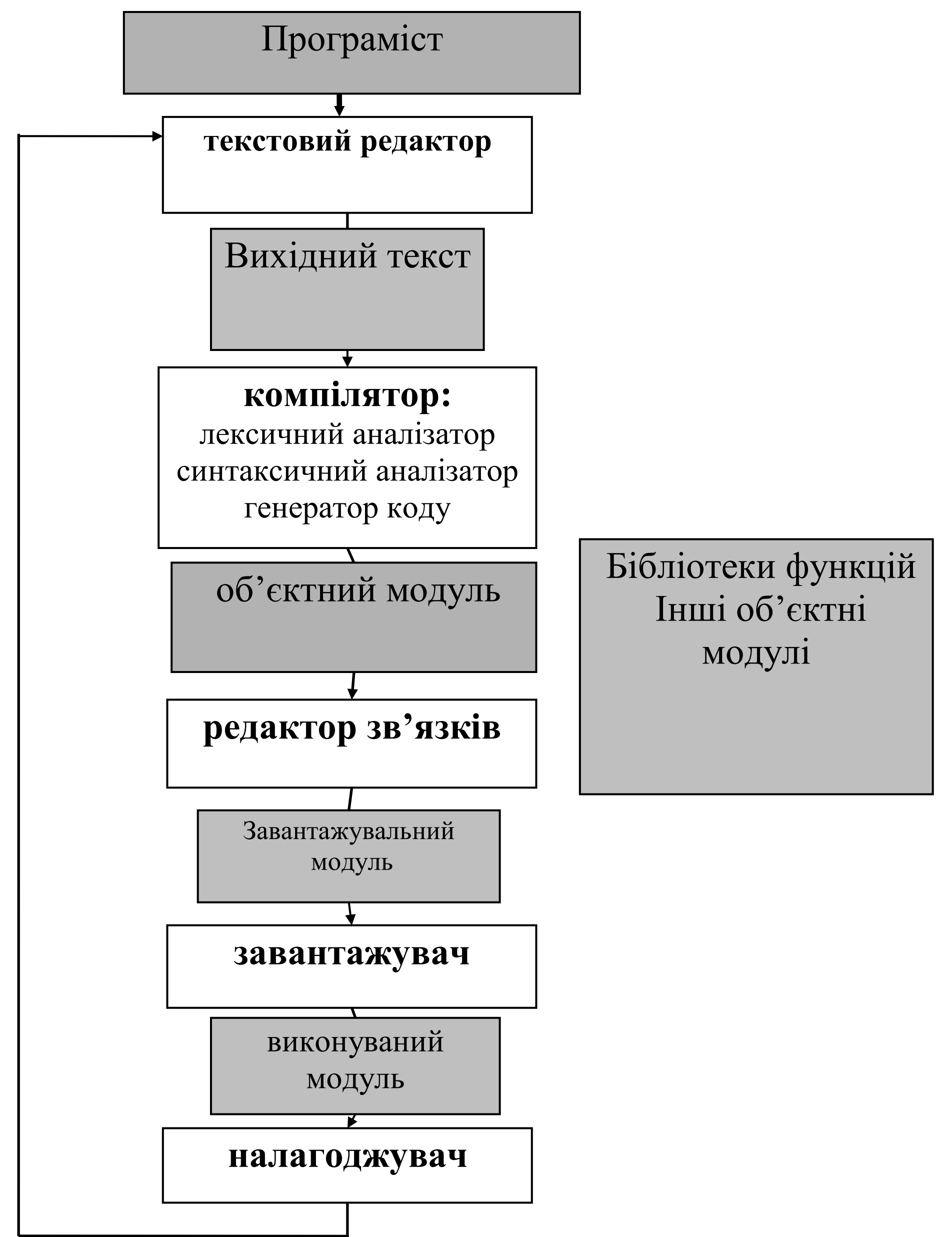

Рисунок 1.4 – Послідовність дій при створенні програмного додатку

Сучасна технологія застосовування комп'ютера потребує, щоб виконуючу програму можна було розміщувати в довільному місці ОЗП. Тому і завантажений модуль написаний в умовних адресах. Розміщенням завантаженого модуля в пам'ять займається програма – завантажувач. Як правило, програми, які щойно написані, містять безліч помилок. Помилки бувають:

а) синтаксичні та лексичні (виявляються на етапах лексичного та синтаксичного аналізу). Наприклад, помилка y = sos(x) замість y= cos(x) – лексична, а помилка в операторі if (x < 0) { y= 0;} else {y= 1} - синтаксична.

б) семантичні (виявляються на етапі налагодження). Наприклад, помилка в операторі присвоювання – ділення на нуль: x= y; z= 1/ (x - y).

в) логічні (виявляються на етапі контрольних випробувань). До логічних ми відносимо такі помилки, в результаті яких програма не виконує того, що ми від неї чекаємо. Для автоматизації процесу пошуку і усунення семантичних і частково логічних помилок використовуються спеціальні програми, які називають відлагоджувачами.

Звичайно в склад системи програмування включають власний текстовий редактор, інші сервісні програми. Для керування роботою будь-яких частин системи програмування використовують керуючу програму, яку називають оболонкою.

Для роботи в мові програмування використовуються спеціальні пакети програм, які називають системами програмування (СП). У склад СП входять:

- оболонка;
- текстовий редактор;
- компілятор;
- редактор зв’язків;
- завантажувач;
- налагоджувач;
- бібліотеки стандартних процедур і функцій;
- сервісні програми.

Історія С/С++

Мова С була створена на початку 70-х років Деннісом Рітчі, який працював в компанії Bell Telephone Laboratories. Родовід мови С бере свій початок від мови Алгол і включає в себе Паскаль.

У 1978 році Річі і Керніган опублікували першу редакцію книги «Мова програмування С». Ця книга, відома серед програмістів як «K & R», служила багато років неформальній специфікацією мови.

В кінці 1970-х років С почав витісняти Бейсік з позиції провідного мови для програмування мікрокомп'ютерів. У 1980-х роках він був адаптований для використання в IBM PC, що призвело до різкого зростання його популярності. У той же час Бярн Страуструп та інші в лабораторіях Bell Labs розпочали роботу по додаванню в С можливостей об'єктно-орієнтованого програмування. Мова, яку вони в результаті зробили, C++, в даний час є найпоширенішою мовою програмування для платформи Microsoft Windows. С залишається більш популярним в UNIX-подібних системах.

У 1983 році Американський Національний Інститут Стандартизації (ANSI) сформував комітет для розробки стандартної специфікації С. По закінченні цього довгого і складного процесу в 1989 році він був нарешті затверджений як «Мова програмування С» ANSI X3.159-1989. Цю версію мови прийнято називати ANSI С.

ANSI С зараз підтримують майже всі існуючі компілятори. Майже весь код С, написаний останнім часом, відповідає ANSI С.

Мова програмування, відома як C++ - це надбудова мови С. Реально вона не є новою мовою, так як включає всі оператори і засоби мови С, додавши лише деякі нові. Вивчаючи С, ви здебільшого одночасно вивчаєте і мову C++. Перевага C++ в тому, що вона дозволяє з більшою легкістю розробляти великі складні програми за рахунок більш модульного підходу та деяких інших удосконалень. Крім того, C++ є мовою об'єктно-орієнтованого програмування.

Відмінні особливості мови С

С стала найбільш популярною серед всіх комп'ютерних мов, і обумовлено це трьома вагомими причинами. Ці причини: швидкість, переносимість та структурування.

Швидкість. Можна сказати, що С є мовою більш близьким до асемблеру, ніж інші мови високого рівня, так як багато інструкції С адресовані безпосередньо апаратної частини комп'ютера. Тому програма на С виконується дуже швидко. Фактично, вона працює настільки швидко, що С може бути використаний для написання операційних систем, комунікаційних і інженерних додатків і навіть компіляторів.

Крім того, більшість компіляторів С генерують високо оптимізовані коди. Ви пам'ятаєте, що комп'ютеру необхідні двійкові коди? Чим менше цих кодів генерує компілятор, тим більше оптимізованим є код і тим швидше працює програма. Багато компілятори інших мов генерують менш оптимізовані коди, так що їх програми працюють повільніше.

Переносимість. Зрозуміло, можна створювати дуже швидко працюючі програми, якщо писати їх на асемблері. Однак мнемонічні асемблерні коди не однакові для кожного сімейства мікропроцесорів. Якби ви написали на асемблері програму для IBM PC або сумісного з ним комп'ютера, а потім вирішили виконати ті ж самі процедури на Apple Macintosh, вам довелося б переписати всю програму.

С використовує стандартні набори ключових слів. У загальному випадку ви пишете програму один раз, безвідносно того на якій платформі (з яким комп'ютером або операційною системою) збираєтеся її використовувати. Тим не менше, хоча вихідний файл зберігається без змін, необхідні дві компілятора: один, щоб перевести програму у двійкові коди, які розуміє IBM, інший, щоб перевести програму у двійкові коди для Apple. Але сам текст програми ви створили раз і назавжди.

Вищесказане також означає, що раз вже ви вивчили Сі, то вам не треба вивчати інші мови, щоб програмувати на інших комп'ютерах. Ви легко

перенесете свої навички з однієї платформи на іншу без перенавчання. Зрештою, важко передбачити заздалегідь, куди вас заведе тільки що розпочате вивчення програмування, і краще бути готовим до всього.

Структурування. Якою б легкою для вивчення не була мова С, у неї є свої вимоги. Мова С має свою структуру і правила створення програми, які змушують програміста мислити логічно. Можна обійтися без серйозного структурування та швидко написати «простеньку» програму, порівняно просту та невеликого розміру, але щоб створити дійсно серйозну програму на С, потрібно перш за все добре подумати. Необхідність структуризації, однак, далеко не є тягарем. Завдяки цій якості програму на С дуже легко проектувати, підтримувати та налагоджувати.

Бібліотеки функцій. Мова С як такої містить невелику кількість операцій. На відміну від інших мов, С не має вбудованих засобів введення та виведення інформації або роботи з рядками (рядок-це послідовність символів, що утворюють слово або фразу). Наприклад, первинний стандарт мови С мав тільки 27 ключових слів (keywords) або команд.

Вся міць мови С забезпечується бібліотеками функцій, які поставляються разом з компілятором.

Функцією називають послідовний набір інструкцій для вирішення спеціального завдання.

Бібліотека - це окремий файл, що додається до компілятору і містить функції для вирішення поширених завдань. Деякі бібліотечні файли містять попередньо відкомпілювалися коди. Коли компілятор стикається з ім'ям функції з такої бібліотеки, йому не доводиться займатися перетворенням інформації в двійкові коди, так як це перетворення вже виконано. Під час процесу компонування програми двійкові коди функції (інструкції для виконання функції, що містяться в бібліотеці) об'єднуються з об'єктним файлом і створюється виконується програма. Оскільки функції були відкомпільовані заздалегідь, вони являють собою дуже ефективні коди.

Виробник компілятора вже «очистив» функції, так що вони повністю оптимізовані.

Існують функції, які використовуються так часто, що разом з багатьма компіляторами поставляються їх вихідні тексти. Вони містяться в файлах, які називаються файлами заголовків (header file) і зазвичай мають розширення .h. Файли заголовків також містять директиви компілятору та інструкції, що вказують використовувати конкретні визначення. Під час процесу компіляції вміст файлу заголовків додається до вашої власної програмі і для нього також створюються об'єктні коди.

Файли заголовків, на відміну від бібліотечних файлів, не відкомпільовані. Так само як і ваш вихідний файл з текстом програми на С, їх можна читати, друкувати на принтері і редагувати. Проте вам слід остерігатися вносити зміни в файли заголовків, що поставляються з компілятором. Якщо ви зробите помилку, то компілятор більше не зможе генерувати для них об'єктні коди.

Контрольні запитання:

1) Що таке мова програмування?
2) Назвіть властивості мов програмування високого рівня.
3) Перелічіть етапи створення програмного додатку.
4) Що таке середовище розробки програмних засобів?
5) Назвіть особливості мови програмування С++.
6) Що таке бібліотека функцій?

Програмні засоби розробки

Поняття оператора

Одним з ключових понять будь-якої мови програмування є поняття "оператор". Без оволодіння цим поняттям повною мірою написання програм не представляється можливим. Від того, наскільки швидко і правильно програміст засвоїть, що таке оператори і як вони застосовуються в програмі, залежить, як скоро він почне писати програми.

Оператор - це складова частина програми, фраза алгоритмічної мови, що пропонує певний порядок перетворення інформації.

Будь-яка програма містить оператори. Найбільш близькою аналогією оператору є речення. Оператори утворюють програму так само, як звичайні речення утворюють текст розповіді або повісті.

Розрізняються два види властивостей операторів - загальні та власні.

Всі оператори мають одну загальну властивість - вони виконуються.

Можна сказати, що оператор - це інструкція, що містить керівництво до дії (опис наказу). Для комп'ютера виконувати запущену на ньому програму означає (послідовно переходячи від одного оператора до іншого) виконувати приписи (інструкції, накази), що містяться в операторах.

Власне оператор - це тільки запис, деяка послідовність символів. У операторі немає важелів, проводів або комірок пам'яті. Тому, коли комп'ютер виконує програму, в самих операторах нічого не відбувається, вони продовжують залишатися в програмі в тому вигляді, в якому їх склав програміст. Але перетворення відбуваються в комп'ютері, що має осередки пам'яті та канали зв'язку між ними. Якщо комп'ютер виконав деякий перетворення інформації відповідно до містяться в операторі приписом, то говорять, що оператор виповнився.

Власні властивості операторів. Існує кілька різних видів операторів. Кожен вид оператора володіє власним властивістю. Наприклад, властивістю

оператора присвоювання є здатність присвоєння зазначеної змінної деякого значення, властивістю оператора циклу є його багаторазове виконання.

Розрізняють два типи операторів - прості і складені.

Прості оператори в мові С закінчуються знаком ";" (крапка з комою). За допомогою цього роздільника комп'ютер може визначити, де закінчується один і починається інший оператор. Знак ";" (Крапка з комою) так само необхідний у програмі, як "." (Звичайна точка) необхідна в звичайному тексті для розділення пропозицій. Один оператор може розташовуватися в декількох рядках, кілька операторів можна розташовувати в одному рядку. Наприклад:

C=5;

Складні оператори. Складний оператор складається з декількох простих, розділених знаком ";", і оформляється фігурними дужками. Щоб можна було використовувати кілька операторів там, де очікується присутність тільки одного, передбачається складений оператор (який також називають "блоком"). Список операторів в складеному операторі виділяється фігурними дужками, а свідченням закінчення складеного оператора є наявність закриває фігурної дужки.

Вимоги до операторів. Оператори повинні бути записані в текст програми відповідно до правил форматування (подання в програмі). Жоден оператор не може бути складений поза цих правил. Якщо ж програма містить оператор, складений всупереч правилам форматування, то при компіляції програми редактор видасть повідомлення про помилку. Це означає, що в такому вигляді програма (що містить помилковий оператор) не може використовуватися.

Вираз "Формат оператора" слід розуміти як набір правил форматування, властивих виду оператора. Кожен вид оператора має свій формат.

Порядок виконання операторів. Важливою характеристикою будь-якої програми є порядок виконання операторів. Оператори не можуть

виконуватися просто так, без усякої системи. У мові С прийнято наступне правило виконання операторів:

Оператори виконуються в тому порядку, в якому вони зустрічаються в програмі. Напрямком послідовності виконання операторів прийнято напрямок зліва направо і зверху вниз.

Це означає, що і прості і складові оператори виконуються просто поспіль, один за іншим (подібно до того, як ми читаємо рядки віршів - спочатку верхній рядок, потім наступний нижче, наступну тощо). Якщо в одному рядку знаходиться декілька операторів, то вони виконуються послідовно один за іншим зліва направо, після чого виконуються оператори в наступному рядку, розташованої нижче.

Оператори, що входять у складений оператор, виконуються так само: будь-який оператор блоку починає виконуватися після того, як виповниться попередній.

Поняття компілятора, інтерпретатора.

З допомогою мови програмування створюється не готова програма, а тільки її текст, що описує раніше розроблений алгоритм. Щоб отримати виконавчу програму, потрібно цей текст або автоматично перевести в машинний код (для чого служать програми-компілятори) і потім використовувати окремо від вихідного тексту, або ж виконувати команди мови, вказані у тексті програми (цим займаються програми-інтерпретатори).

Перекладачі бувають двох типів: компілятори (compiler) та інтерпретатори (interpreter).

Компілятори повністю обробляють увесь текст програми (його часто називають вихідним кодом програми). Компілятори переглядають текст програми в пошуках синтаксичних помилок (інколи декілька разів), виконують певний змістовий аналіз, а потім автоматично переводять (транслюють) на машинну мову - генерують машинний код. В результаті виконавча програма виходить компактною і ефективною, працює швидше,

може бути перенесена на інші комп'ютери з процесором, що підтримують відповідний машинний код. Недолік - труднощі трансляції мов програмування, орієнтованих на опрацювання даних складної структури.

Процес компіляції складається з двох частин: аналізу (analysis) і синтезу (synthesis).Аналізує частина компілятора розбиває вихідну програму на складові її елементи (конструкції мови) і створює проміжне представлення вихідної програми. Синтезує частина з проміжного представлення створює нову програму, яку комп'ютер в змозі зрозуміти. Така програма називається об'єктної програмою. Об'єктна програма може надалі виконуватися без перетрансляції. В якості проміжного представлення зазвичай використовуються дерева, зокрема, так звані дерева розбору. Під деревом розбору розуміється дерево, кожен вузол якого є певною операції, а сини цього вузла - операнда.

На відміну від компілятора, інтерпретатор не створює ніякої нової програми, а просто виконує кожне речення мови програмування.

Інтерпретатор бере наступний оператор мови із тексту, аналізує його структуру і відразу виконує. Тільки після його успішного виконання інтерпретатор переходить до наступного оператора. Один і той же оператор інтерпретатор виконує так, начебто зустрів його вперше. При цьому, деякі програми працюють повільно й завжди потрібно мати інтерпретатор.

В реальних системах програмування змішані технології, що використовують як компіляцію, так і інтерпретацію програм.

Поняття компілятора.

Створюючи на завершальному етапі певну програму, будь-якому програмісту доводиться звертатися до послуг компілятора. У технічній документації цій програмі відведено досить скромне визначення як утиліті, що виконує компіляцію.

Компіляція – це процес перетворення програми, написаної мовою, зрозумілою людині (мові високого рівня), у команди, зрозумілі для машини (низькорівнева мова).

В результаті отримуємо програму, яка близька машинного коду. Вона може виглядати як об’єктний модуль, абсолютний код. Іноді така програма схожа на мову асемблера.

Таким чином, компіляція – це коли вхідна інформація (вихідний код), що представляє опис алгоритму або написана на проблемно-орієнтованій мові програма, переписується в еквівалентний перелік команд, представлених в об’єктних коді (машинно-орієнтованій мові).

Якщо ще спростити визначення, то компілювати – це транслювати машинну програму з проблемно-орієнтованого в машинно-орієнтована мова.

Незважаючи на прозорість і простоту визначення, компіляція – це процес досить багатоплановий. Існує кілька її видів:

- Пакетна компіляція здійснюється над кількома вихідними модулями в одному пункті завдання.

- Пострічкова компіляція – це те ж саме, що і інтерпретація (покрокова незалежна компіляція кожного наступного оператора).

- Умовна компіляція. У такому випадку трансльований текст має залежність від умов, які задані у вихідній програмі директивами компілятора.

Міняючи значення певної константи, можна регулювати включення або виключення трансляції частини тексту програми.

Для зручності програмістів при вирішенні різних завдань застосовуються найбільш зручні і пристосовані компілятори. Якщо провести їх класифікацію, то можна виділити кілька видів подібних утиліт.

Компілятор векторизуется виробляє трансляцію вихідного коду в машинний комп’ютерний код, підлаштовуючись під векторні процесори.

Гнучкий компілятор був розроблений на основі модульного принципу. Його управління здійснюється таблицями. Запрограмований він на

високорівневої мовою. Також можлива його реалізація за допомогою компілятора компіляторів.

Компілятор інкрементальний здійснює повторне транслювання фрагментів програми та доповнень до неї, при цьому перекомпіляція всієї програми виключається.

Інтерпретує або покроковий компілятор використовує принцип послідовного виконання незалежної компіляції для кожного окремого оператора або команди з вихідної програми.

Компілятор компіляторів – це транслятор, який сприймає формальний опис для мови програмування. Він здатний самостійно генерувати компілятор для конкретної мови.

Відладочний компілятор може самостійно усувати деякі види синтаксичних помилок.

Резидентність компілятору відведено постійне місце в оперативній пам'яті, і він доступний при повторному використанні широким спектром завдань.

Існують самокомпілюючіся компілятори. Вони пишуться на тій же мові, з якої відбувається трансляція.

Універсальний компілятор має в основі формальний опис семантики і синтаксису вхідної мови. Він складається з ядра, синтаксичного і семантичного завантажувачів.

Найбільш часто зустрічаються задачі, де компілятори знаходять собі застосування, – це компіляція ядра для платформи Linux. Операція ця дозволяє вирішити широкий спектр проблем, пов'язаних з погодженням обладнання та налаштування найбільш прийнятною версії платформи.

Інтерпретатор. Директиви препроцесора

Обробка програми препроцесором здійснюється перед її компіляцією. На цьому етапі попередньої обробки можуть виконуватися наступні дії: включення у файл, що компілюється інших файлів, визначення символічних

констант і макросів, встановлення режиму умовної компіляції програми і умовного виконання директив препроцесора.

Директивами називаються інструкції препроцесора. Всі директиви повинні починатися з символу #, перед яким в рядку можуть розташовуватися тільки пробільні символи.

Примітка. Після директив препроцесора крапка з комою не ставиться.

Інтерпретатор переводить комп'ютера всі інструкції безпосередньо в момент їх виконання.

Програма, що обробляється інтерпретатором, існує тільки у вигляді вихідного текстового файлу.

Наприклад, на початкових етапах навчання, інтерпретатор дозволяє без додаткових зусиль писати та відразу ж тестувати програму рядок за рядком. Компілятору ж, щоб приступити до переведення, необхідно мати повністю завершений текст всієї програми, або, як мінімум, окремої виконуваної її частини.

У силу того, що інтерпретатор робить процес створення програми дуже легким, у користувача з'являється спокуса зневажити стадією попереднього планування і проектування, необхідної для того, щоб створити працюючу програму. Він упевнено приступає до справи, намагаючись писати програму з нальоту, а потім проводити довгі непотрібні годинник, вносячи зміни методом проб і помилок. Це не найкращий спосіб роботи, і оскільки ви тільки приступаєте до вивчення програмування, вам краще вчитися працювати грамотно з самого початку.

Крім того, що інтерпретуються мови працюють повільніше. Необхідно завантажити інтерпретатор в пам'ять комп'ютера, потім переводити і виконувати кожну окрему рядок програми. Компілятор перетворить весь текст програми відразу, а після цього відкомпілювати запускається програма існує у вигляді двійкових кодів, адресованих безпосередньо комп'ютеру.

Інтегроване середовище підготовки програм на C++ чи компілятор мови обов'язково містять препроцесор для обробки тексту програми перед компіляцією.

Стадії та команди препроцесорної обробки. Препроцесорна обробка відповідно до вимог стандарту мови C++ містить кілька стадій, що виконуються поступово. Конкретна реалізація транслятора може поєднувати кілька стадій, але результат повинен бути таким, неначе вони виконуються послідовно:

- всі системно-залежні позначення (наприклад, системно-залежний індикатор кінця рядка) перекодуються у стандартні коди;
- кожна пара символів '\' та "кінець рядка" видаляються і, наступний рядок вхідного файла приєднується до рядка, у якому знаходилась вказана пара символів;
- у тексті розпізнаються директиви препроцесора, а кожен коментар замінюється одним символом порожнього пропуску;
- виконуються директиви препроцесора та макропідстановки;
- ESC-послідовності у символьних константах та символьних рядках, наприклад, '\n', замінюються на їх еквіваленти (на відповідні числові коди);
- суміжні символьні рядки конкатенуються, тобто поєднуються в один рядок .

Знайомство з переліченими задачами препроцесорної обробки пояснює деякі узгодження синтаксису мови. Наприклад, зрозумілим є суть ствердження: кожен символьний рядок можна у файлі переносити на наступний рядок, якщо використати символ '\' чи "два символьні рядки, записані послідовно, сприймаються як один рядок".

На стадії обробки директив препроцесора можливі такі дії:

- заміна ідентифікаторів наперед підготовленими послідовностями символів;
- приєднання до програми текстів із вказаних файлів;
- видалення з програми окремих частин її тексту (умовна компіляція);

- макропідстановка, тобто заміна позначення параметризованим текстом, що формується препроцесором з урахуванням конкретних параметрів (аргументів).

Для керуванням препроцесором (вказування потрібних дій) використовуються команди (директиви) препроцесора, кожна з яких міститься в окремому рядку та починається символом #. Визначені наступні препроцесорні директиви: #define, #include, #undef, #if, #ifdef, #ifndef, #else, #endif, #elif, #line, #error, #pragma, #.

Директива #define має кілька модифікацій. Вони передбачають визначення макросів чи препроцесорних ідентифікаторів, кожному з яких відповідає деяка символьна послідовність. У наступному тексті програмі препроцесорні ідентифікатори замінюються наперед запланованими послідовностями символів.

Директива #include приєднує до тексту програми текст вибраного файла.

Дирекстива #undef відміняє дію команди #define, яка визначила до цього ім’я препроцесорного ідентифікатора.

Директива #if та її модифікації #ifdef, #ifndef сумісно з директивами #else, #endif, #elif організують умовну обробку тексту програми, коли компілюється не весь текст, а лише ті його частини, які деяким способом виділені названими директивами.

Директива #line дає змогу керувати нумерацією рядків у файлі з програмою. Ім’я файла та початковий номер рядка вказуються безпосередньо у директиві #line.

Директива #error дає змогу вказати текст діагностичного повідомлення, яке виведеться на екран при виникненні помилок.

Директива #pragma викликає дії, які залежать від реалізації.

Директива #- порожня директива, не викликає ніяких ефектів, ігнорується.

Розглянемо можливості перерахованих команд при вирішенні типових задач, що доручаються препроцесору.

Директива підключення #include

Препроцесор приєднує до програми засоби зв'язку з бібліотекою введення-виведення, пошук файла iostream ведеться у стандартних системних каталогах.

Синтаксис:

#include <ім'я_файла> //ім'я в кутових дужках

#include "ім'я_файла" //ім'я у лапках

Директива #include використовується для включення копії вказаного файла в те місце програми, де знаходиться ця директива.

Різниця між двома формами директиви полягає в методі пошуку препроцесором файла, що включається.

Якщо ім'я файла розміщене в "кутових" дужках < >, то послідовність пошуку препроцесором заданого файла в каталогах визначається встановленими каталогами включення (include directories).

Якщо ж ім'я файла заключне в лапки, то препроцесор шукає в першу чергу файл у поточній директорії, а потім вже у каталогах включення.

Робота директиви #include зводиться практично до того, що директива #include прибирається, а на її місце заноситься копія вказаного файла.

За узгодженням суфікс .h надається файлам, які записані у заголовку програми.

Окрім такого дещо стандартного файла. Яким є iostream, у заголовок програми можна приєднувати інші файли (стандартні чи підготовлені спеціально).

Заголовочні файли - досить ефективний засіб при модульній розробці великих програм, коли зв'язок між модулями, що розташовані у різних файлах, реалізується не лише через параметри, але й через зовнішні об'єкти, глобальні для кількох чи всіх модулів. Оголошення таких зовнішніх об'єктів

(змінних, масивів, структур) записуються у одному файлі, який директивами #include приєднуються до всіх модулів, де потрібні зовнішні об’єкти. До того ж файлу можна приєднати і директиву приєднання бібліотеки функцій введення-виведення..

Текст файла, що включається може містити директиви препроцессора, і директиву #include зокрема. Це означає, що директива #include може бути вкладеною. Допустимий рівень вкладеності директиви #include залежить від конкретної реалізації компілятора.

#include <stdio.h> / приклад 1*/*

#include "defs.h" / приклад 2*/*

В першому прикладі у головний файл включається файл з ім'ям stdio.h. Кутові дужки повідомляють компілятору, що пошук файла необхідно здійснювати в директоріях, вказаних в командному рядку компіляції, а потім в стандартних директоріях.

В другому прикладі в головний файл включається файл з ім'ям defs.h. Подвійні лапки означають, що при пошуку файла спочатку повинна бути переглянута директорія, що містить поточний файл.

В С є також можливість задавати ім'я шляху в директиві #include за допомогою іменованої константи. Якщо за словом include слідує ідентифікатор, то препроцесор перевіряє, чи не іменує він константу або макровизначення. Якщо ж за словом include слідує рядок, що заключений в лапки або в кутові дужки, то С не буде шукати в ній ім'я константи.

#define myincl "c:\test\my.h"

#include myincl

При програмуванні на С++ зустрічається і дещо зворотна ситуація. Коли у програмі використовується кілька функцій, тоді іноді зручно текст кожної з них зберігати в окремому файлі. При підготовці програми користувач приєднує до неї тексти використовуваних функцій командою #include.

Директива #define

Синтаксис:

#define ідентифікатор текст

#define ідентифікатор (список_параметрів) текст

Директива #define заміняє всі входження ідентифікатора у програмі на текст, що слідує в директиві за ідентифікатором. Цей процес називається макропідстановкою. Ідентифікатор замінюється лише в тому випадку, якщо він представляє собою окрему лексему. Наприклад, якщо ідентифікатор є частиною рядка або більш довгого ідентифікатора, він не замінюється. Якщо за ідентифікатором слідує список параметрів, то директива визначає макровизначення з параметрами.

Текст представляє собою набір лексем, таких як ключові слова, константи, ідентифікатори або вирази. Один або більше пробільних символів повинні відділяти текст від ідентифікатора (або заключених в дужки параметрів). Якщо текст не вміщується в рядку, то він може бути продовжений на наступному рядку; для цього слід набрати в кінці рядка символ обернений слеш \ і зразу за ним натиснути клавішу Enter.

Текст може бути опущений. В такому разі всі екземпляри ідентифікатора будуть вилучені з тексту програми. Але сам ідентифікатор розглядається як визначений і при перевірці директива #if дає значення 1.

Список параметрів, якщо він заданий, містить один або більше ідентифікаторів, розділених комами. Ідентифікатори в рядку параметрів повинні відрізнятися один від одного. Їх область дії обмежена макровизначенням. Список параметрів повинен бути заключений в круглі дужки. Імена формальних параметрів у тексті відмічають позиції, в які повинні бути підставлені фактичні аргументи макровиклику. Кожне ім'я формального параметра може з'явитися в тексті довільне число раз.

В макровиклику за ідентифікатором записується в круглих дужках список фактичних аргументів, що відповідають формальним параметрам із списку параметрів. Текст модифікується шляхом заміни кожного

формального параметра на відповідний фактичний параметр. Списки фактичних параметрів і формальних параметрів повинні містити одне і те ж число елементів.

Примітка. Не слід плутати підстановку аргументів в макровизначеннях з передачею параметрів у функціях. Підстановка в препроцесорі носить чисто текстовий характер. Ніяких обчислень при перетворенні типу при цьому не виконується.

Вище вже говорилося, що макровизначення може містити більше одного входження даного формального параметра. Якщо формальний параметр представлений виразом з "побічним ефектом" і цей вираз буде обчислюватися більше одного разу, разом з ним кожний раз буде виникати і "побічний ефект". Результат виконання в цьому випадку може бути помилковим.

Всередині тексту в директиві #define можуть знаходитися вкладені імена інших макровизначень або констант.

Після того, як виконана макропідстановка, отриманий рядок знову переглядається для пошуку інших імен констант і макровизначень. При повторному перегляді не розглядається ім'я раніше проведеної макропідстановки. Тому директива

#define a a

не призведе до за циклювання препроцесора.

Приклад 1:

#define WIDTH 80

#define LENGTH (WIDTH+10)

В даному прикладі ідентифікатор WIDTH визначається як ціла константа із значенням 80, а ідентифікатор LENGTH - як текст (WIDTH+10). Кожне входження ідентифікатора LENGTH у програму буде замінено на текст (WIDTH+10), який після розширення ідентифікатора WIDTH перетвориться на вираз (80+10). Дужки дозволяють уникнути помилок в операторах, подібних наступному:

*val=LENGTH*20;*

Після обробки програми препроцесором текст набуде вигляду:

*val=(80+10)*20;*

Значення, яке буде присвоєно змінній val рівне 1800.

При відсутності дужок значення val буде рівне 280.

*val=80+10*20;*

Приклад 2:

#define MAX(x,y) ((x)>(y))?(x):(y)

В даному прикладі визначається макровизначення MAX. Кожне входження ідентифікатора MAX в тексті програми буде замінено на вираз ((x)>(y))?(x):(y), в якому замість формальних параметрів x та y підставляються фактичні. Наприклад, макровиклик:

MAX(1,2)

заміниться на вираз ((1)>(2))?(1):(2).

Директива #undef

Синтаксис:

#undef ідентифікатор

Визначення символічних констант і макросів можуть бути анульовані за допомогою директиви препроцесора #undef. Таким чином, область дії символічної константи або макросу починається з місця їх визначення і закінчується явним їх анулюванням директивою #undef або кінцем файла.

Після анулювання ідентифікатор може бути знову використаний директивою #define.

Приклад:

```
#define WIDTH 80
/* … */
#undef WIDTH
/* … */
#define WIDTH 20
```

Умовна компіляція та макропідстановки засобами препроцесора.

Умовна компіляція у мові С++ забезпечена набором команд, які керують не компіляцією, а препроцесорною обробкою:

#if константний_вираз – деякий текст компілюється лише при виконанні умови

#ifdef ідентифікатор – перевіряє, чи визначений на поточну мить ідентифікатор у#define

#else ідентифікатор – на компіляцію передається інший текст, заданий ідентифікатором

#endif – при відсутності директиви #else весь текст від #if до #endif пропускається

#elif

Перші три команди перевіряють умови, дві наступні – визначають діапазон дії перевіреної умови. Остання директива організує мультирозгалуження при обробці препроцесором вхідного тексту програми: #elif константний_вираз.

Макрос – це засіб заміни однієї послідовності символів іншою.

Для виконання заміни потрібно задавати відповідні макровизначення. Найпростіші макровизначення вводились при розгляді директиви #define ідентифікатор рядок_заміщення. Вона зручна у використанні, проте має недолік – рядок заміщення фіксований, тому використовують макровизначеннія з параметрами

:#define ім’я(список_параметрів) рядок_заміщення

Тут ім’я – ім’я макроса (ідентифікатор), список_параметрів – список відокремлених комами ідентифікаторів. Між іменем макроса та списком параметрів не можна ставити пробіли. Макровизначення:#define max(a,b) (a<b?b:a) створює у програмі вираз, що визначає максимальне з двох значень аргументів. Тепер max(X,Y) замінюється виразом (X<Y?Y:X), а max (Z,4) створить вираз (Z<4?4:Z). При істинному значенні X<Y повертається

значення Y, інакше – значення X. У другому прикладі Z порівнюється з константою 4 і вибирається більше значення.

Заголовні файли

Визначення і оголошення спільно використовуються кількома файлами. Можливо, централізувати ці оголошення і визначення так, щоб для них була тільки одна копія. Тоді програму в процесі її розвитку буде легше і виправляти, і підтримувати в робочому стані. Для цього заголовну інформацію треба розташувати в заголовні файли (*.h), який по мірі необхідності можна включати в інші файли. (Директивою # include)

Для програм, що не перевищують деякого середнього розміру, мабуть, найкраще мати один заголовний файдл, в якому зібрані разом всі об'єкти, кожен з яких використовується більш ніж в одному файлі. Для програм великих розмірів потрібно більше складна організація з великим числом заголовків файлів.

Моделі пам'яті в мові програмування C++ - спосіб вказати припущення, які має зробити компілятор при генерації коду для платформ з сегментної адресацією пам'яті або з сторінкової пам'яттю.

Контрольні запитання:

1) Що таке компілятор?
2) Що таке інтерпретатор?
3) Дайте визначення поняттю препроцесор.
4) Для чого використовується деректива “#include”?
5) Для чого використовується деректива “#define”?
6) Для чого використовуються заголовочні файли?

Оператори мови програмування C++

Алфавіт мови, або її символи - це основні неподільні знаки, за допомогою яких пишуться всі тексти на мові програмування.

Лексема, або елементарна конструкція - мінімальна одиниця мови, яка має самостійний зміст.

Вираз задає правило обчислення деякого значення.

Оператор задає кінцевий опис деякої дії.

Алфавіт мови C++

Алфавіт мови C++ включає:

- великі та малі літери латинської абетки;
- арабські цифри;
- пробільні символи : пробіл, символи табуляції, символ переходу на наступний рядок тощо;
- символи , . ; : ? ' ! | / \ ~ ( ) [ ] { } < > # % ^ & - + * =

Основою системи числення називається кількість різних символів (цифр), що використовуються в кожному з розрядів числа для його зображення в даній системі числення.

Двійкова система: 0,1

Десяткова система: 0,1,2,3,4,5,6,7,8,9

Шістнадцятирічна система: 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F

У різних версіях можуть використовуватися різні набори символів. Зараз широко використовується набір символів коду ASCII (American Standard Code for Information Interchange).

Цей код передбачає розширення для національних алфавітів, символів псевдографіки, які можуть змінюватись від версії до версії.

Програма мовою C складається з синтаксичних конструкцій які називаються команди (оператори, вказівки).

Команди будуються з лексем.

Лексема – неподільний елемент мови (слово, число, символи операцій).

Ідентифікатор – це назва (ім’я), яку користувач надає об’єктам, наприклад змінним, сталим, функціям.

Ідентифікатори записуються латинськими буквами, цифрами, знаком підкреслення. Розпочинаються ідентифікатори з латинських літер та знаку підкреслення.

A, a, max, Max, MAX, _max, max1, max_znach – правильно записані ідентифікатори.

1max, max-znach, max znach, a..b – неправильно записані ідентифікатори

При написанні імені ідентифікатора враховується регістр. (MAX,Max,max- три різні ідентифікатори).

Для поліпшення читаності програми слід давати об'єктам осмислені імена. Існує угода про правила створення імен, що називається угорською нотацією(оскільки запропонував її співробітник компанії Microsoft угорець за національністю), по якому кожне слово, що становить ідентифікатор, розпочинається з прописної букви а спочатку ставиться префікс, що відповідає типу величини, наприклад, iMaxLength IpfnSetFirstDialog. Інша традиція - розділяти слова, що становлять ім'я, знаками підкреслення : maxjength, number _ of _ galosh.

Довжина ідентифікатора за стандартом не обмежена, але деякі компілятори і компонувальники накладають на неї обмеження. Ідентифікатор створюється на етапі оголошення змінної, функції, типу і т. п., після цього його можна використати у подальших операторах програми. При виборі ідентифікатора необхідно мати на увазі наступне:

- ідентифікатор не повинен співпадати з ключовими словами і іменами використовуваних стандартних об'єктів мови;

- не рекомендується розпочинати ідентифікатори з символу підкреслення, оскільки вони можуть співпасти з іменами системних функцій або змінних і, крім того, це знижує мобільність програми;

- на ідентифікатори, використовувані для визначення зовнішніх змінних накладаються обмеження компонувальника (використання різних компонувальників або версій компонувальника накладає різні вимоги на імена зовнішніх змінних).

Ключові слова

Ключові слова - це зарезервовані ідентифікатори, які наділені певним змістом. Їх можна використовувати тільки у відповідності зі значенням відомим компілятору мови С.

Ідентифікатор (ім'я) – це послідовність букв та цифр довільної довжини. Перший символ - буква, символ підкреслення "_" - також буква. Великі та малі букви є різні. Мова С++ не обмежує кількість символів в імені, проте у її реалізаціях містяться компоненти, параметри яких не узгоджені з виробником транслятора (наприклад, завантажувач) і можуть обмежувати довжину імені. Також деякі системні програми, потрібні для виконання програми на С++, збільшують чи зменшують допустиму кількість символів у ідентифікаторі. Наприклад, використання $ в імені погіршує можливість перенесення програми на іншу операційні системи. Не можна використовувати як імена також службові слова С++.

Всі імена, які починаються символом підкреслення, відведені для використання у реалізації мови чи у програмах сумісного використання з програмою користувача, тому його не можна використовувати у іменах програми користувача. Ідентифікатори, які містять подвійне підкреслення "__", відведені для реалізації С++ та стандартних бібліотеках системи програмування, тому користувач не може їх використовувати для створення власних імен.

Ключові слова. Наведені нижче ідентифікатори фіксуються тільки як службові слова.

asm	*continue*	*float*	*new*	*signed*	*try*
auto	*default*	*for*	*operator*	*sizeof*	*typedef*
break	*delete*	*friend*	*private*	*static*	*union*
case	*do*	*goto*	*protected*	*struct*	*unsigned*
catch	*double*	*if*	*public*	*switch*	*virtual*
char	*else*	*inline*	*register*	*template*	*void*
class	*enum*	*int*	*return*	*this*	*volatile*
const	*extern*	*long*	*short*	*throw*	*while*

Ключові слова не можуть бути використані в якості ідентифікаторів.

Константами називають сталі величини, тобто такі, які в процесі виконання програми не змінюються. В мові С існує чотири типи констант : цілі, дійсні, рядкові та символьні.

1. Цілі константи можуть бути десятковими, вісімковими або шістнадцятковими.

Якщо ви хочете виробляти математичні операції, то повинні використовувати числовий тип даних. Мова С має кілька типів числових даних в залежності від значення, яке може бути їм присвоєно, і займаного обсягу пам'яті.

Цілі числа (int, від англійського integer) - це числа, що не мають дробової частини. Їх значення може бути позитивним, негативним або нульовим, але ніколи не має у своєму складі знаків після точки. У мові С є проста аксіома, яка говорить: «Використовуйте для підрахунків цілі числа». Використовуйте цілі числа завжди, коли є можливість представити якесь значення у вигляді цілого числа, наприклад, при підрахунку кількості повторів певної події.

Кожен елемент цілочисельних даних займає в пам'яті стільки ж місця, скільки два елементи символьних, незалежно від величини самого числа (і число 2, і число 2000 вимагають для зберігання однакового обсягу пам'яті). Але для того щоб займане місце не перевищувала двох елементів пам'яті,

величина цілочисельних даних в мові С обмежена. До цілочисловим даними (власне тип int) відносяться величини, значення яких знаходиться в проміжку між -32768 і 32767. Величини, значення яких виходить за ці межі, вимагають для зберігання більше двох елементів пам'яті.Для того щоб забезпечити можливість роботи з числами будь-якої величини, в більшості компіляторів С визначено декілька типів цілочисельних даних. Межі для різних типів можуть розрізнятися залежно від конкретного компілятора.

За замовчуванням усі цілочисельні типи вважаються знаковими, тобто специфікатор signed можна опускати.

Константам, яке трапляється в програмі, приписується той або інший тип відповідно до їх видом. Якщо цей тип з якихось причин не влаштовує програміста, він може явно вказати потрібний тип за допомогою суфіксів L, 1 (long) і U, u (unsigned). Наприклад, константа 321 буде мати тип long і займати 4 байти. Можна використовувати суфікси L і U одночасно, наприклад, 0x22UL або 05lu.

Типи short int, long int, signed int і unsigned int можна скорочувати до short, long, signed, і unsigned відповідно.

Десяткова константа - послідовність десяткових цифр (від 0 до 9), яка починається не з нуля якщо це число не нуль. Приклади десяткових констант: 10, 132, 1024.

Вісімкові константи починаються з символу 0, після якого розміщуються вісімкові цифри (від 0 до 7). Наприклад: 023. Запис константи вигляду 08 буде сприйматися компілятором як помилка, так як 8 не є вісімковою цифрою.

Шістнадцяткові константи починаються з символів 0x або 0X, після яких розміщуються шістнадцяткові цифри (від 0 до F, можна записувати їх у верхньому чи нижньому регістрах). Наприклад: 0XF123.

Дійсні константи складаються з цілої частини, десяткової крапки, дробової частини, символу експоненти (e чи E) та показника степеня. Дійсні константи мають наступний формат представлення:

[ціла_частина][. дробова_частина][E[-]степінь]

У записі константи можуть бути опущені ціла чи дробова частини (але не обидві разом), десяткова крапка з дробовою частиною чи символ E (e) з показником степеня (але не разом). Приклади дійсних констант: 2.2, 220e-2, 22.E-1, .22E1.

Якщо потрібно сформувати від'ємну цілу або дійсну константу, то перед константою необхідно поставити знак унарного мінуса.

Числа, які можуть містити десяткову частину (речові), називаються числами з плаваючою точкою (floating-point values). Для роботи з ними в мові С використовується тип даних з плаваючою точкою (float). Так як числа з плаваючою точкою можуть бути надзвичайно маленькими або великими, для їх запису часто використовують експоненційну форму, наприклад, значення числа з плаваючою точкою може дорівнювати 3.4e +38. Розшифрувати це можна таким чином: «пересунути точку вправо на 38 пунктів, додавши відповідну кількість нулів». Існують додаткові типи даних для роботи в дуже широких межах величин:

Тип даних з плаваючою точкою має межу точності, діапазон якого залежить від компілятора. Наприклад, число 6.12345678912345 в межах допустимого діапазону для чисел типу float може бути записано комп'ютером тільки як 6.12345. Цей тип, як прийнято говорити, є типом з одинарною точністю, що означає, що точність його обмежена п'ятьма або шістьма знаками після крапки. Тип double прийнято називати типом з подвійною точністю, він має 15-16 знаків після точки.

Символьні константи. Символьна константа - це один або декілька символів, які заключені в апострофи. Якщо константа складається з одного символу, вона займає в пам'яті 1 байт (тип char). Двосимвольні константи займають в пам'яті відповідно 2 байти (тип int).

Послідовності символів, які починаються з символу \ (обернений слеш) називаються керуючими або escape-послідовностями (таблиця 1.2).

Таблиця 1.2 Escape-послідовності

Спеціальний символ	Шістнадцятковий код	Значення
\a	07	звуковий сигнал
\b	08	повернення на 1 символ
\f	0C	переведення сторінки
\n	0A	перехід на наступний рядок
\r	0D	повернення каретки
\t	09	горизонтальна табуляція
\v	0B	вертикальна табуляція
\\	5C	символ \
\'	27	символ '
\"	22	символ "
\?	3F	символ ?
\0	00	нульовий символ
\0ddd	-	вісімковий код символу
\0xddd	ddd	шістнадцятковий код символу

4. Рядкові константи записуються як послідовності символів, заключених в подвійні лапки."Це рядковий літерал!\n"

Для формування рядкових констант, які займають декілька рядків тексту програми використовується символ \ (обернений слеш):

*"Довгі рядки можна розбивати на *

частини"

Загальна форма визначення іменованої константи має вигляд:const тип ім'я = значення ;

Модифікатор const попереджує будь-які присвоювання даному об'єкту, а також інші дії, що можуть вплинути на зміну значення. Наприклад:

```
const float pi = 3.14l5926;
const maxint = 32767;
char *const str="Hello,P...!"; /* покажчик-константа */
char const *str2= "Hello!"; /* покажчик на константу */
```

Використання одного лише модифікатору const еквівалентно const int.

Назва константи	Значення
M_PI	π
M_PI_2	π/2
M_PI_4	π/4
M_1_PI	1/π
M_2_PI	2*π
M_1_SQRTPI	1/√π
M_2_SQRTPI	2/√π
M_E	E
M_LOG2E	log(e)
M_LOG10E	log10(e)
M_LN2	Ln(2)
M_LN10	Ln(10)
M_SQRT	2√2
M_SQRT_2	√2/2

Розширений символьний тип (wchar_t)

Тип wchar_t призначений для роботи з набором символів, для кодування яких недостатньо 1 байта, наприклад, Unicode. Розмір цього типу залежить від реалізації; як правило, він відповідає типу short. Рядкові константи типу wchar_t записуються з префіксом L, наприклад, L "Gates".

Логічний тип (bool)

Величини логічного типу можуть приймати тільки значення true і false, які є зарезервованими словами. Внутрішня форма представлення значення false - 0 (нуль). Будь-яке інше значення інтерпретується як true. При перетворенні до цілого типу true має значення 1.

Змінні

Суть фактично будь-якої програми зводиться до введення, зберігання, модифікації і виведенню деякої інформації. Для того, щоб програма могла на протязі свого виконання зберігати певні дані і оперувати ними, використовуються змінні та константи.

Змінну можна уявити собі як чарунку пам’яті комп’ютера, в якій може зберігатися деяке значення, доступне для використання в програмі. Пам’ять комп’ютера можна розглядати як ряд чарунок. Змінна може займати одну або кілька чарунок, в яких зберігаються певні дані.

У будь-якій алгоритмічній мові кожна константа, змінна, результат обчислення виразу або функції повинні мати певний тип.

Змінну найчастіше визначають як пару «ім’я”=”значення”. Імені відповідає адреса дільниці пам’яті, відведеної змінній, а значення - її вміст. Іменем є ідентифікатор, а значення відповідає типу змінної, який визначає множину допустимих значень та набір операцій, для яких змінна може бути операндом. Множина допустимих значень змінної найчастіше співпадає з множиною допустимих констант того ж типу. Таким чином, вводяться дійсні, цілі та символьні змінні, до того ж символьні (char) іноді відносять до цілих. Цілочислові та дійсні вважаються арифметичними типами, які (разом з символьним) є окремим випадком скалярних типів, куди відносять ще вказівники, посилання та переліки.

Тип даних визначає:

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

Визначення та оголошення змінних в С++

Визначення відрізняється від оголошення тим, що, окрім введення змінної у текст програми, передбачає на основі даного визначення відведення їй пам’яті компілятором. Визначення та оголошення змінних основних типів записується ключовими словами:

Основні (стандартні) типи даних часто називають арифметичними, оскільки їх можна використовувати в арифметичних операціях. Для опису основних типів визначені наступні ключові слова:

int (ціле);

char (символьний);

wchar_t (розширений символьний);

bool (логічний);

float (речовинний);

double (речовинний з подвійною точністю).

void – відсутність значення.

Перші чотири типи називають цілочисельними (цілими), останні два - типами з плаваючою крапкою. Код, який формує компілятор для обробки цілих величин, відрізняється від коду для величин з плаваючою крапкою.

Існує чотири специфікатора типу, уточнюючих внутрішнє представлення та діапазон значень стандартних типів:

short (короткий);

long (довгий);

signed (знаковий);

unsigned (беззнакові).

При визначенні змінним можна надавати початкових значень, які записуються у відведену для них пам’ять при ініціюванні. Наприклад, визначення (оголошення з ініціюванням):

char newsimbol = ‘\n’;

long filebagin = 0;

double pi = 3.14159265358979323 85;

У позначенні типу можна використовувати одночасно кілька службових слів. Наприклад, визначення: long double zebra, stop; оголошує змінні з іменами zebra та stop дійсного типу підвищеної точності, проте явно не надає їм ніяких початкових значень.

Службові слова unsigned (беззнаковий) та signed (знаковий) дають змогу вибирати спосіб урахування знакового розряду для арифметичного чи символьного типу :

unsigned int i,j,k; // Значення від 0 до 65535

unsigned long L,M,N; // Значення від 0 до 4294967295

unsigned char c,s; // Значення від 0 до 255;

cout<<"\n sizeo(double)="<<sizeof(d); /*8*/ cout<<"\t sizeof (long double)="<<sizeof(ld);}/*10*/

При оголошенні змінної для неї виділяється деяка область пам'яті, розмір якої залежить від конкретного типу змінної.

Константи, як і змінні, представляють собою область пам'яті для збереження даних, лише з тією різницею, що значення, яке було надано константі не може бути змінено на протязі виконання всієї програми. Константи можуть бути літеральними і типізованими, причому літеральні поділяються ще на символьні, рядкові, цілі та дійсні.

Кожна змінна має тип даних. Саме тип даних визначає які дані вона може містити та які операції можна виконувати над змінною, а також інтерпретацію цих операцій (таблиця 1.3).

Таблиця 1.3 - Вбудовані типи даних C++

Назва	Позначення	Діапазон значень
Символ	char	від -128 до +127
Без знака	unsigned char	від 0 до 255
Коротке ціле число	short	-32768 до +32767
Коротке ціле число без знаку	unsigned short	від 0 до 65535
Ціле число	int	– 2147483648 до + 2147483647
Ціле число без знаку	unsigned int (або просто unsigned)	від 0 до 4294967295
Довге ціле число	long	від – 2147483648 до + 2147483647
Довге ціле число без знаку	unsigned long	від 0 до 4294967295
Дійсне число одинарної точності	float	±3.4e-38 до ±3.4e+38 (7 значучих цифр)
Дійсне число подвійної точності	double	±1.7e-308 до ±1.7e+308 (15 значучих цифр)
Дійсне число збільшеної точності	long double	±1.2e-4932 до ±1.2e+4932
Логічне значення	bool	значення true або false

Перетворення типів

В C++ існує явне та неявне перетворення типів.

В загальному випадку перетворення типів записується у двох варіантів

1) (новий_тип) вираз;

2) новий_тип (вираз).

Обидва варіанти перетворення виглядають так:

```
char letter = 'a';
int nasc = int (letter);
long iasc = (long) letter;
```

У загальному випадку неявне перетворення типів зводиться до участі у виразі змінних різного типу (так звана арифметика змішаних типів). Якщо подібна операція відбувається над змінними базових типів, вона може спричинити помилки: у випадку, наприклад, якщо результат займає в пам'яті більше місця, ніж відведено під приймаючу змінну, неминуча втрата розрядів.

Для явного перетворення змінної одного типу в інший перед ім'ям змінної у дужках вказується тип, що їй привласнюється. Наприклад:

```
#include <iostream>
using namespace std;
int main()
{
int integer=54;
float floating = 15.873;
integer = (int) floating; // явне перетворення типу
cout<<"New integer:";
cout<<Integer<<'\n';
return 0;
}
```

У наведеному прикладі після визначення відповідних змінних (цілочисельної та дійсної) відбувається явне перетворення типу з плаваючою крапкою до цілочисельного.

Приклад неявного перетворення:

```
#include <iostream>
using namespace std;
{
int integer=0;
 float floating = 15.873;
Integer = floating; // неявне перетворення типу
cout<<"New integer:";
cout<<Integer<<'\n';
return 0;
}
```

На відміну від попереднього варіанту програми, в даному випадку після визначення та ініціалізації змінних відбувається привласнення значення змінної з плаваючою крапкою цілочисельній змінній. Результат роботи обох програм виглядає наступним чином:

New integer: 15

Тобто у другому прикладі неявно відбулося відсікання дробової частини дійсного числа.

Операції, пріоритети, правила та приклади виконання.

Для здійснення маніпуляцій над даними C++ має широкий набір операцій. Операції представляють собою деякі дії, які виконуються над одним (унарна) або кількома(бінарна) операндами, в результаті якої повертається значення.

Арифметичні операції. До базових арифметичних операцій можна віднести операції додавання (+), віднімання (-), множення (*), ділення (/) та

взяття за модулем (%), тобто обчислення залишку від ділення лівого операнду на правий.

Для ефективного використання повертаємого операціями значення застосовується оператор присвоєння (=) та його модифікації: додавання з присвоєнням (+=), віднімання з присвоєнням (-=), множення з присвоєнням (*=), ділення з присвоєнням (/=), модуль з присвоєнням (%=) та інші.

У наведеному лістінгу представлені приклади використання операцій.

```
#include <iostream>
using namespace std;
int main()
{
        int a=0, b=4, c=90;
        char z='\n';
        a=b;  //a=4
        cout<<a<<z;
        a=b+c+c+b;        //a=4+90+90+4=188
        cout<<a<<z;
        a=b-2;        //a=4-2=2
cout<<a<<z;
a=c/(b+6);  //a=90/(4+6)=9
cout<<a<<z;
cout<<a%2<<z;           //9%2=1
a+=b;         //a=a+b=9+4=13
a*=c-50;     //a=a*(c-50)=13*(90-50)=520
return 0;}
```

В С++ існує ефективний засіб для збільшення та зменшення значення операнда на одиницю – унарні операції інкремента(++) та декремента(--).

По відношенню до операнда даний вид операторів може буте префіксний або постфіксний. Префіксний оператор застосовується до

операнда перед використанням отриманого значення. Постфіксний оператор застосовується до операнда після використання операнда. Поняття префікса та постфіксу має значення лише у виразах з присвоєнням. Приклад:

int x=1, y=3;

x=y++;   //x=3, y=4

x=--y;   //y=3, x=3

x++; //x=4

Порозрядні логічні операції

Для того щоб програміст міг адресуватися до окремих бітів використовуються порозрядні логічні операції.

Операція	Описання
&	Логічне І (множення)
|	Логічне АБО (додавання)
^	Виключне АБО
~	Логічне НІ (інверсія)

Логічні операції

Не потрібно плутати порозрядні логічні операції з просто логічними операціями && - логічне І, || - логічне АБО, ! – логічне НІ. Їх результатом може бути 0 або 1 і вони в основному використовуються в умовних виразах операторів if, while або for.

Операція «кома»

Операція «кома» пов’язує між собою кілька виразів таким чином, що останні розглядаються компілятором як один цілий вираз.

Завдяки використанню даної операції при написані програм досягається висока ефективність. У наведеному прикладі спочатку відбудеться виклик функції з присвоєнням результату змінній і, а потім відбудеться порівняння значення і з 7.

if (i=CallFunc(), i<7)

…

Ще більшої ефективності вдається досягти при використанні операції «кома» у операторі циклу for.

Пріоритет операцій

Як і в арифметиці, в C++ операції виконуються у визначеному порядку. Для того, щоб компілятор міг розібратися яку дію виконувати першою при виконанні операцій, останні в свою чергу мають важливу властивість – пріоритет. В першу чергу виконуються операції з найбільшим пріоритетом. В таблиці наводяться операції, які використовуються в C++, впорядковані за спаданням пріоритету (таблиця 1.4)

Таблиця 1.4 – Приорітет операцій на мові C++

::	Розширення області видимості	class_name :: member
::	Глобальне	:: name
.	вибір члена	object . member
->	Вибір члена	pointer -> member
[]	індексування	pointer [expr]
()	Виклик функції	expr (expr_list)
()	Структурне значення	type (expr_list)
Sizeof	Розмір об’єкта	sizeof expr
Sizeof	Розмір типу	sizeof (type)
++	Постфіксний інкремент	lvalue ++
++	Префіксний інкремент	++ lvalue
--	Постфіксний декремент	lvalue --
--	Префіксний декремент	-- lvalue
~	доповнення	~ expr
!	Логічне НІ	! expr
-	Унарний мінус	- expr
+	Унарний плюс	+ expr
&	Взяття адреси	& lvalue
*	вказівник	* expr
New	Створення (розміщення)	new type
Delete	Видалення (звільнення)	delete pointer
delete[]	Видалення масиву	delete[] pointer
()	Приведення типу	(type) expr
*	множення	expr * expr
/	ділення	expr / expr
%	Остача від ділення	expr % expr
+	Додавання (плюс)	expr + expr
-	Віднімання (мінус)	expr – expr

Продовження таблиці 1.4

<<	Зсув вліво	expr << expr
>>	Зсув вправо	expr >> expr
<	менше	expr < expr
<=	Менше або дорівнює	expr <= expr
>	більше	expr > expr
>=	Більше або дорівнює	expr >= expr
==	дорівнює	expr == expr
!=	Не дорівнює	!= expr
&	Порозрядке І	expr & expr
^	Порозрядне виключаюче АБО	expr ^ expr
|	Порозрядне включаюче АБО	expr | expr
&&	Логічне І	expr && expr
||	Логічне АБО	expr || expr
? :	Операція умови	expr1? expr2 : expr3
=	Просте присвоєння	lvalue = expr
*=	Присвоєння з множенням	lvalue *= expr
/=	Присвоєння з діленням	lvalue /= expr
%=	Присвоєння із взяттям остачі від ділення	lvalue %= expr
+=	Присвоєння з додаванням	lvalue += expr
-=	Присвоєння з відніманням	lvalue -= expr
<<=	Присвоєння з ссувом вліво	lvalue <<= expr
>>=	Присвоєння з ссувом вправо	lvalue >>= expr
&=	Присвоєння з порозрядним І	lvalue &= expr
|=	Присвоєння з порозрядним включаючим АБО	lvalue |= expr
^=	Присвоєння з порозрядним виключаючим АБО	lvalue ^= expr
,	Кома (послідовність)	expr , expr

Контрольні запитання:

1) Що таке оператор?
2) Що таке змінна?
3) Що таке константа?
4) Перелічіть типи данних в С++.
5) Які логічні операції є в С++?
6) Яка система приорітетності операцій підтримується в С++?

РОЗДІЛ ІІ Основи мови програмування С++

Функції вводу/виводу на С++

Структура програми на С++

Функція main()

Усі програми, написані на мові С, повинні містити в собі хоча б одну функцію.

Функція main() - вхідна точка будь-якої програмної системи, причому немає різниці, де її розміщувати. Але потрібно пам'ятати наступне: якщо вона буде відсутня, завантажувач не зможе зібрати програму, про що буде виведене відповідне попередження. Перший оператор програми повинен розміщуватися саме в цій функції.

Коментар – це фрагмент тексту який призначено для пояснення програми або окремих фрагментів. Коментар записують:

```
/*   початок коментарю
       коментар може бути записаний
       в декількох стрічках
*/   кінець коментарю
// коментар записаний в одній стрічці
```

Узагальнена структура програми на мові С++ має вигляд:

```
 #include   //опис бібліотек
void main()              //головна функція
{
              тіло програми;
}
```

На приклад:

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

```
int main()
{
cout<<"My first program";
return 0;
}
```

Функція починається з імені. В даному прикладі вона не має параметрів, тому за її ім'ям розташовуються порожні круглі дужки (). Далі обидві фігурні дужки {...} позначають блок або складений оператор, з яким ми працюватимемо, як з єдиним цілим. У Паскалі аналогічний зміст мають операторні дужки begin ... end.

Мінімальна програма має лише один оператор - оператор повернення значення return. Він завершує виконання програми та повертає в нашому випадку деяке ціле значення (ненульове значення свідчить про помилки в програмі, нульове про успішне її завершення). Виконання навіть цієї найпростішої програми, як і решти багатьох, проходить у декілька етапів.

код запуску → функція main() → код завершення

Засоби вводу/виводу С++

Програмний додаток, написаний будь-якою мовою програмування, повинний взаємодіяти з навколишнім світом. Інакше користі від нього не буде. Як правило, така взаємодія здійснюється за допомогою введення/виведення інформації на монітор або в файл. Правда, є деяке множина програм, які не використовують файловий або консольний ввід-вивід, це програми, що здійснюють низькорівневу взаємодію з апаратною частиною комп'ютера та периферією (ядро ОС, драйвери тощо).

У стандартному С ++ існує два основних шляхи введення-виведення інформації:

- за допомогою традиційної системи введення/виведення, успадкованої від С;

- за допомогою потоків, реалізованих у STL (Standard Template Library).

Якщо копнути трохи глибше, то виявиться, що і потоки, і традиційна система вводу/виводу для здійснення необхідних дій використовують виклики операційної системи.

Традиційна система вводу/виводу.

Під традиційними функціями вводу/виводу будемо розуміти функції, що згруповані в бібліотеці cstdio (C Standard Input and Output Library. Бібліотека c*stdio* надає необхідний набір функцій для введення і виведення інформації як у текстовому, так і в двійковому поданні.

Приклад використання бібліотеки c*stdio*:

```
#include <cstdio>
int main()
{
printf("Hello, world!\n");
return 0;
}
```

При виводі інформації тут використовуються потоки даних, зокрема потік для виведення на стандартний пристрій виводу інформації, в загальному випадку це є монітор.

Потік даних (англ. data stream) в програмуванні - це послідовність кодованих пакетів даних в цифровій формі, яка використовується, щоб передавати або отримати інформацію. У програмуванні з потоком даних часто асоціюється потік - абстракція, яка використовується для читання чи запису в єдиній манері файлів, сокетів тощо. Потоки є зручним уніфікованим програмним інтерфейсом для читання або запису файлів (у тому числі спеціальних і, зокрема, пов'язаних з пристроями), сокетів і передачі даних між процесами. Підтримка потоків включена в більшості мов програмування

і чи не в усі сучасні операційні системи. При запуску процесу йому надаються так звані стандартні потоки.

При запуску консольного програмного додатку неявно відкриваються три потоки:

stdin - для введення з клавіатури;

stdout - для буферизованного виводу на монітор;

stderr - для небуферізованной виводу на монітор повідомлень про помилки.

Основні властивості потоків. Потоки мають деякі властивості, які визначають, які функції можуть бути використані для організації введення/виводу і яким чином буде здійснюватися обмін даними через потоки введення або виведення. Більшість з цих властивостей визначаються в момент, коли потік, пов'язаний з файлом, відкритий за допомогою функції fopen.

- Доступ потоку до читання або запису. Ця властивість визначає, чи має даний потік доступ до читання і (або) запису на фізичних носіях.

Текст або двійковий код. Потоки, як вважається, представляють собою набір текстових рядків, кожен з яких закінчується символом нового рядка. Залежно від середовища, в якому програма запускається, символи нового рядка можуть відрізнятися, тому виникає необхідність адаптувати деякі спеціальні символи в текстовому файлі, згідно специфікаціям використовуваної системи. З іншого боку, двійковий потік - це послідовність символів, записувана або зчитувана з фізичного середовища без всякого перетворення даних.

Буфер тимчасового зберігання даних. Буфер блоку пам'яті, де дані накопичуються, перш ніж фізично зчитуються або записуються на відповідний файл або пристрій. Потоки можуть бути або з повною буферизацією, або без буферизації. Якщо повна буферизація, то дані читання / запису фізично переносяться або змінюються, коли буфер заповнюється. Буфер вважається заповненим, якщо в потік потрапляє символ нового рядка.

Небуферізованной потоки символів, також призначені для читання / запису, але буферизація в них виконується, по можливості, якомога швидше.

Індикатори потоків введення / виводу. Потоки мають певні внутрішні показники, які визначають їх поточний стан і які впливають на поведінку деяких операцій введення:

Індикатор помилки. Цей індикатор сигналізує про те, що сталася помилка в ході виконання операції, пов'язаної з потоком. Цей показник може бути перевірений функцією ferror, і може бути скинутий шляхом виклику функції clearerr або будь-якою функцією позиціонування (rewind, fseek і fsetpos).

End-Of-File індикатор. Якщо даний індикатор сигналізує про те, що остання операція читання або запису з потоком досягла кінця файлу. Це можна перевірити за допомогою функції feof. Даний індикатор може бути скинутий шляхом виклику функції clearerr або будь-якою функцією позиціонування (rewind, fseek і fsetpos).

Індикатор положення. Це внутрішній покажчик кожного потоку, який вказує на наступний символ, який повинен бути лічений або записаний в наступній операції введення / виводу. Його значення може бути отримано функціями ftell і fgetpos.

В бібліотеці cstdio, серед інших, є наступні функції для організації вводу/виводу :

*int printf(const char *format, ...) // фоматований консольний вивід*

*int sprintf(char *s, const char *format, ...) // форматований вивід в буфер (стрічку)*

*int scanf(const char *format, ...) // фоматований консольний ввід*

*int sscanf (const char *s, const char *format, ...) // форматований ввід в буфер (стрічку)*

int getchar(void) // читання символу з stdin

*char *gets(char *s) // читання стрічки з stdin*

int putchar(int c) // запис символу в stdout

*int puts(const char *s) // запис стрічки в stdout*

Функція printf () дозволяє виводити інформацію на екран при програмуванні в консольному режимі. Дана функція визначена в бібліотеці cstdio і має наступний синтаксис:

*int printf (const char * format [, argument] ...);*

Тут перший аргумент * format визначає рядок, яка виводиться на екран і може містити спеціальні керуючі символи для виведення змінних. Потім, слідує список необов'язкових аргументів. Функуція повертає або число відображених символів, або негативне число у разі своєї некоректної роботи.

printf ("Hello world");

Однак з її допомогою можна виводити змінні різного типу: починаючи з числових і закінчуючи рядковими. Для виконання цієї операції використовуються спеціальні керуючі символи, які називаються специфікаторами і які починаються з символу%.

Серед керуючих символів є наступні:

Код	Формат виводу
% c	Символ
%d	Десяткове ціле число зі знаком
%i	Десяткове ціле число зі знаком
%e	Науковий формат (мала літера е)
%E	Науковий формат (мала літера Е)
%f	Десяткове число з плаваючою точкою
%g	Залежно від того, який запис коротший, застосовується або %e, або %f
%G	Залежно від того, який запис коротший, застосовується або %E, або %f
%o	Вісімкове число без знака
%s	Рядок символів
%u	Десяткове ціле число без знака
%x	Шістнадцяткове число без знака (малі літери)
%X	Шістнадцяткове число без знака (прописні літери)
%p	Вказівник
%%	Знак %

В якості прикладу розглянемо наступний запис:

printf("Hello %c %d %s",'f ', 10, "world!");

В результаті виконання даної функції на екран буде виведено “Hello f 10 world”.

Крім специфікаторів у функції printf () використовуються керуючі символи, такі як перевід каретки на новий рядок \ n, табуляції \ t та ін. Наприклад:

printf ("num_i =% d, \ n num_f =% f", num_i, num_f);

В даному випадку вивід значень буде проводитись в вигляді стовпців.

Коди формату можуть мати модифікатори, що задають ширину поля, точність і ознака вирівнювання по лівому краю. Ціле число, вказане між знаком відсотка% і кодом формату, діє як специфікатор мінімальної ширини поля. Він заповнює поле виведення пробілами або нулями, забезпечуючи її мінімальну ширину. Якщо рядок або число більше мінімальної ширини поля, вони все одно виводяться повністю. За умовчанням як заповнювач використовується прогалину. Якщо порожнє поле виведення необхідно заповнити нулями, перед специфікатором ширини поля слід поставити символ 0. Наприклад, специфікатор% 05d заповнить нулями пустощі позиції поля виводу, якщо кількість цифр в цілому числі, підметі висновку, буде менше п'яти.

Точний зміст модифікатора точності залежить від коду модифікує формату. щоб додати модифікатор точності, після специфікатора ширини поля необхідно поставити десяткову точку і вказати точність. Для форматів e, E і f модифікатор точності означає кількість цифр після десяткової крапки. Наприклад, форматний код% 10.4f виведе на екран число, у якого кількість цифр не перевищує 10, чотири з яких розміщуються після десяткової крапки. Якщо модифікатор застосовується до специфікаторами формату% g або% G, він задає кількість значущих цифр.

Якщо модифікатор точності застосовується до цілим типами, він задає мінімальну кількість цифр, з яких має складатися число. Якщо число

складається з меншої кількості цифр, воно доповнюється провідними нулями.

Якщо модифікатор використовується для виведення рядків, він задає максимальну довжину поля. Наприклад, специфікатор% 5.7s означає, що на екран буде виведена рядок, що складається як мінімум з п'яти символів, довжина якої не перевищує семи. Якщо рядок виявиться довшим, останні символи будуть відкинуті.

За замовчуванням висновок вирівнюється по правому краю. Інакше кажучи, якщо ширина поля більше, ніж виводяться дані, результати притискаються до правого краю. Висновок можна вирівняти по лівому краю, поставивши перед символом% знак "мінус". Наприклад, специфікатор% -10.2f вирівнює число з двома знаками після крапки по лівому краю поля, що складається з 10 позицій.

Функція scanf(). Для введення інформації з клавіатури зручно використовувати функцію scanf () бібліотеки cstdio, яка має наступний синтаксис:

*int scanf (const char * format [, argument] ...);*

Тут, як і для функції printf (), змінна * format визначає форматну рядок для визначення типу даних, що вводяться і може містити ті ж специфікатори що і функція printf (). Потім, слід список необов'язкових аргументів. Робота функції scanf () демонструється на лістингу:

```
int age;
float weight;
printf ("Введіть інформацію про Ваш вік:");
scanf ("% d", & age);
printf ("Введіть інформацію про Вашу вагу:");
scanf ("% f", &weigth);
printf ("Ваш вік =% d, Ваша вага =% f", age, weigth);
```

Основною відмінністю застосування функції scanf () від функції printf () є знак & перед ім'ям змінної, в яку записуються результати введення.

Функція scanf () може працювати відразу з декількома змінними. Припустимо, що необхідно ввести два цілих числа з клавіатури. Формально для цього можна двічі викликати функцію scanf (), проте краще скористатися такою конструкцією:

scanf ("% d,% d", & n, & m);

Функція scanf () інтерпретує це так, як ніби очікує, що користувач введе число, потім - кому, а потім - друге число. Функція scanf () повертає число успішно лічених елементів. Якщо операції зчитування не відбувалося, що буває в тому випадку, коли замість очікуваного цифрового значення вводиться якась буква, то повертається значення одно 0.

Ввід/вивід на основі потоків, реалізованих у STL.

Для використання об'єктно-орієнтованого консольного вводу-виводу за допомогою потоків (stream) STL в програму необхідно включити заголовний файл <iostream> а для файлового ще й <fstream>. Приклад программи з використанням потоків STL виглядає:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, world!\n";
return 0;}
```

При запуску консольного додатку неявно відкриваються чотири потоки:

cin - для введення з клавіатури;

cout - для буферизованного виводу на монітор;

cerr - для небуферізованной виводу на монітор повідомлень про помилки;

clog - буферізованние аналог *cerr*.

Ці чотири символи визначені за допомогою <iostream>. Потоки *cin, cout* і *cerr* відповідають потокам *stdin, stdout* і *stderr* відповідно.

Для того щоб використовувати стандартні потоки для введення і виведення, необхідно включити заголовний файл <iostream>. Для введення використовується операція >>, для виведення - операція <<. Компілятор визначає тип введеної / виведеної змінної і відповідним чином форматує її.

```
#include <iostream>
using namespace std;
cin >> x; // Введення значення в змінну x зі стандартного потоку cin
cout << x; // Вивід значення змінної x в стандартний потік cout
cin >> x >> y; // Введення двох змінних
cout << "x =" << x << "\ ny =" << y << endl; // Функція endl здійснює перевід на новий рядок
```

Якщо при введенні або виведенні сталася помилка, у змінній стану потоку встановлюється відповідний прапор. Стан потоку визначається набором з 4 прапорів - badbit, failbit, eofbit, goodbit. Всі ці прапори визначені в класі ios_base, який є базовим для всіх класів потоків. Перші два означають виникнення помилки на останній операції вводу-виводу (причому, як пише сам творець мови, відмінність між ними незначне і цікаво в основному тим, хто створює оператори введення-виведення для нових класів). Третій означає досягнення кінця файлу. Четвертий прапор - goodbit - означає відсутність трьох перших, це ознака бездоганно пройшла операції вводу-виводу.

Приклад перевірки винекнення помилок під час вводу інформації, преевірка здійснюється на наявність встановленог опрапорця failbit. Інші перевірки можна здійснити аналогічно.

```
cin >> x;
if (cin.fail ()) cout << "Сталася помилка при введенні \ n";
```

Можливість управляти введенням-виводом в С ++, забезпечують форматуючі функції-члени, прапори та маніпулятори. Прапори, функції і

маніпулятори виконують одне і те ж завдання - задають певний формат введення / виводу інформації в потоках. Введення / вивід на екран / з екрану в С ++ здійснюється за допомогою операторів cin і cout відповідно, а значить маніпулятори форматування використовуються спільно з цими операторами введення / виводу. Різницю між функціями прапорами і маніпуляторами форматування полягає в способі їх застосування. Тепер розглянемо способи застосування об'єктів форматування.

// Основні форматує функції-члени:

cout.fill ('/ * symbol * /'); // Встановлює символ заповнювач

// де symbol - символ заповнювач, символ передається в одинарних лапках

cout.width (/ * width_field * /); // Задає ширину поля

// де width_field - кількість позицій (одна позиція вміщує один символ)

cout.precision (/ * number * /); // Задає кількість знаків після десяткової крапки

// де number - кількість знаків після десяткової крапки

Доступ до функцій здійснюється через операцію крапка, а в круглих дужках передається аргумент. Аргумент функції fill () може передаватися у вигляді символу, обрамленого одинарними лапками, або у вигляді числа (код символу). Одних функцій не достатньо для форматування потоків введення / виводу, тому в С ++ передбачений ще один спосіб форматування — прапори.

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

// Установка прапора виведення

cout.setf (ios :: / * name_flag * /);// де name_flag - це ім'я прапора

Доступ до функцій оператора виведення виконується через операцію крапка. Метод setf () приймає один аргумент - ім'я прапора. Прапори

виведення оголошені в класі ios, тому, перед тим, як звернутися до прапора, необхідно написати ім'я класу - ios, після якого, за допомогою операції дозволу області дії, викликати потрібний прапор.

// Зняття прапора виведення

cout.unsetf (ios :: / * name_flag * /);// де name_flag - це ім'я прапора

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

// Установка декількох прапорів

cout.setf (ios :: / * name_flag1 * / | ios :: / * name_flag2 * / | ios :: / * name_flag_n * /);

// Зняття декількох прапорів

cout.unsetf (ios :: / * name_flag1 * / | ios :: / * name_flag2 * / | ios :: / * name_flag_n * /);

Ще один спосіб форматування - форматування за допомогою маніпуляторів. Маніпулятор - об'єкт особливого типу, який управляє потоками вводу / виводу, для форматування переданої в потоки інформації. Почасти маніпулятори доповнюють функціонал, для форматування введення / виводу. Але більшість маніпуляторів виконують точно, те ж саме, що і функції з прапорами форматування. Є випадки, коли простіше користуватися прапорами або функціями форматування, а іноді зручніше використовувати маніпулятори форматування.

Приклади використання форматованого виводу: для t∈ [0; 3] з кроком 0,5 обчислити значення y = cos (t).

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

```
HANDLE h;
h = GetStdHandle(STD_OUTPUT_HANDLE);
SetConsoleTextAttribute(h,FOREGROUND_RED | BACKGROUND_BLUE);
 cout<<fixed;
 for (double t = 0; t <= 3; t += 0.5) {
   cout.width(3);
   cout.precision(1);
   cout << t;
   cout.width(8);
   cout.precision(3);
   cout << cos(t) << endl;  }
 system("pause");
 return 0;}
```

До переваг використання бібліотеки cstdio слід віднести невеликі розміри (розмір програми “Hello world” з використанням даної бібліотеки буде мати приблизно 15Кб, а використання iostream – приблизно мегабайт) та швидкість обробки введеної/виведеної інформації.

До переваг використання iostream слід віднести простоту, більшу гнучкість та лаконічність записів. Для роботи деяких маніпуляторів слід підключати додаткові бібліотеки, наприклад <iomanip>.

Контрольні запитання:

1) Наведіть узагальнену структуру програми на мосі С++.
2) Назвіть основні бібліотеки для організації вводу/виводу в С++.
3) Охарактеризуйте особивості організації вводу/виводу на основі потоків.
4) Які засоби використовуються для організації форматованого виводу інформації?

Обробка масивів в С++

Одномірні масиви

Масив - однорідна, фіксована за розміром і конфігурацією, послідовність елементів простий або складової структури, упорядкованих за індексам і розташованих у пам'яті поспіль. Елементи цієї послідовності даних називаються елементами масиву. Нумерація елементів виконується індексними послідовностями: кожен елемент масиву має свій індекс, чим і відрізняється від інших елементів. Масив визначається ім'ям і розмірністю - кількістю індексів, необхідних для зазначення місцезнаходження елемента масиву (для одновимірного масиву - Одним, для двомірного - двома і т.д.). Масив називається багатовимірним при кількості індексів більше одного.

У С ++ ознакою одновимірного масиву є наявність парних дужок [], в яких вказується кількість елементів масиву (його довжина, розмір), зафіксоване у визначенні масиву і в процесі роботи програми не змінюване. Розмір може бути заданий тільки цілої позитивної константою або константним виразом.

Визначення одновимірного масиву:

тип ім'я [розмір масиву];

У деяких випадках допустимо опис масиву без зазначення кількості його елементів:

extern int array [];

До окремого елементу масиву можна звернутися за допомогою змінної з індексом: імені масиву (єдиного для всіх елементів) і одного або декількох індексів, залежно від розмірності масиву. Індекс може мінятися від 0 до n-1, де n - кількість елементів масиву, його розмір. Максимальний індекс елемента завжди на 1 менше розміру масиву.

Наприклад:

char buffer [81]; // масив складається з елементів buffer [0], ... buffer [80] типу char;

int key [4]; // масив складається з елементів key [0], ... key [3] типу int;

float a [10]; // масив складається з елементів key [0], ... key [3] типу float;

Доступ до елемента масиву:

buffer [80] key [3] a [1] a [9]

Автоматичний контроль виходу індексу за межі масиву не проводиться, тому програміст повинен стежити за цим самостійно. Типовою помилкою при використанні масивів є звернення до неіснуючого елементу, тобто вихід індексу за допустиме значення.

При визначенні масиву його розмір може бути не зазначений. У цьому випадку повинен бути присутнім ініціалізатор, і компілятор виділить пам'ять за кількістю ініціалізували значень. Ініціалізували значення для масивів записуються у фігурних дужках. Значення елементам присвоюються по порядку.

Якщо елементів у масиві більше, ніж ініціалізаторов, елементи, для яких значення не вказані, обнуляються:

*int b [5] = {3, 2, 1}; // масив цілих значень, займає 5 * 4 = 20 байтів*

// b [0] = 3; b [1] = 2; b [2] = 1; b [3] = 0; b [4] = 0;

*float f [10]; // масив займає 10 * 4 = 40 байтів*

*double x [] = {1.5, 2.8, 3.4, 0.5, 7.3};// масив дійсних значень займає 5 * 8 = 40 байтів*

char str [] = {'a', 'b', '+'}; // масив символів, займає 3 байти

Якщо елементів у масиві менше, ніж ініціалізаторов, то визначення масиву буде помилковим:

double B []; // помилка у визначенні - ні розміру double B [3] = {1, 2, 3, 4, 5}; // помилка ініціалізації

Для статичних або зовнішніх масивів компілятор виконує ініціалізацію за замовчуванням (всім елементам присвоюються нульові значення).

Найбільш часто масив використовується для зберігання значень векторів. Наприклад, структура з трьох значень типу float, проіндексованих заданих діапазоном цілих чисел від 0 до 2 оголошується записом: float V [3];

У пам'яті елементи V [0], V [1], V [2] розташовуються послідовно:

V [0] V [1] V [2]

Адреса масиву в пам'яті збігається з адресою його першого елемента (елемента зі значенням індексу рівним лівій межі діапазону індексів).

Будь поліном можна описати масивом його коефіцієнтів a0, a1, ..., ak-1, ak, в якому роль індексу грає ступінь змінної x: a [0], a [1], ... a [k-1], a [k] .

Характеристики масиву

В якості характеристик масиву виступають:

-Тип - загальний тип елементів масиву.

- Розмірність - кількість індексів масиву (a [10] - одновимірний масив; b [3] [5] - двомірний масив; з [2] [3] [5] - тривимірний і т.д.)

- Розмір (кількість елементів) по кожній з розмірностей.

- Тип компонентів. Тип індексів

Елементами масиву можуть бути дані будь-якого типу, включаючи похідні (масиви, рядки, структури, файли). В якості індексів можуть використовуватися цілі позитивні константи або вирази зі сталими. Рекомендується задавати розміри масиву у вигляді іменованих або препроцесорних констант. При необхідності такі константи легко змінити, адаптуючи їх до різних наборам вихідних даних, хоча і з повторною трансляцією програми. Крім того, завдання значення такої константи достатньо великим, дозволить працювати з масивами даного типу будь-якого меншого розміру і як би ними нівелює змінну довжину статичного масиву.

Приклад використання при визначенні масиву попередньо визначених констант:

#define k 50 const int n = 10; const int m = 20; double a [n]; char b [m]; double ar [k];

Двовимірні масиви

Характерним об'єктом програмування є двовимірні масиви (матриці). Часто розглядаються матриці спеціального виду - поодинокі, діагональні, трикутні, розріджені та ін. Подання матриці кожного з видів в програмі може бути різним; наприклад, діагональні матриці можуть представлятися одновимірним масивом.

Запам'ятаємо деякі властивості квадратних матриць:

- якщо номер рядка і елемента матриці А збігається з номером стовпця j, тобто і == j, це означає, що елемент A [i] [i] лежить на головній діагоналі матриці;

- якщо номер рядка і елемента матриці А перевищує номер стовпчика j, тобто i> j, це означає, що елемент A [i] [j] знаходиться нижче головної діагоналі,

- якщо номер стовпчика j елемента матриці А більше номера рядка і, тобто і <j, це означає, що елемент A [i] [j] знаходиться вище головної діагоналі;

-якщо індекси елемента A [i] [j] матриці А задовольняють рівності і == Nj-1,це означає, що елемент A [i] [j] лежить на побічній діагоналі;

- якщо індекси елемента A [i] [j] матриці А задовольняють нерівності і <Nj-1,це означає, що елемент A [i] [j] знаходиться вище побічної діагоналі;

- якщо індекси елемента A [i] [j] матриці А задовольняють нерівності i> Nj-1,це означає, що елемент A [i] [j] знаходиться нижче побічної діагоналі.

Приклад заповнення двомірного масиву значеннями символів

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

```
const int n=3;
const int m=4;
char a[n][m];
int i, j;
for (i=0; i< 4; i++)
        for( j=0; j< 3; j++)
                a[i][j] = char (97+rand() %26);
for (i=0; i< 4; i++)
for( j=0; j< 3; j++)
        cout<<a[i][j]<<"\n";
_getch();
return 0;
}
```

Багатомірні масиви

За визначенням, багатовимірні масиви як такі в С ++ не існують, масив завжди вважається одномірним. Однак в С ++ дозволено оголошувати одномірні масиви масивів (тобто багатовимірні масиви). Вони задаються зазначенням розміру кожної розмірності в квадратних дужках.

Наприклад, опис двовимірного масиву з 6 рядків по 8 стовпців: int matr[6][8]; інтерпретується як одновимірний масив з ім'ям matr з 6 елементів типу int [8].

Тривимірний масив double prim[6][4][2]; інтерпретується як одновимірний масив з ім'ям prim, що включає 6 елементів, кожен з яких має тип double[4][2]. У свою чергу, кожен з цих елементів є одновимірний масив з чотирьох елементів типу double[2]. І, нарешті, кожен з цих елементів є масивом з двох елементів типу double.

Для доступу до елементу багатовимірного масиву вказуються всі його індекси,наприклад:

matr [i] [j], prim [i] [j] [k] ;.

Форми ініціалізації масиву:

При структурній ініціалізації багатовимірний масив представляється як масив масивів, при цьому кожен масив полягає в свої фігурні дужки. При безструктурній ініціалізації задається загальний список елементів у тому порядку, в якому елементи розташовуються в пам'яті.

Багатовимірні масиви можуть ініціалізовуватись і без зазначення розміру самої лівої розмірності. Компілятор в цьому випадку визначає число елементів за кількістю членів у списку ініціалізації.

Приклади:

структурна ініціалізація:

int mass [] [2] = {{1,1}, {0, 2}, {1, 0}};

int array [2] [3] = {{1, 2, 3}, {2, 3, 4}};

int a [2] [3] [4] = {{{1}, {2, 3}, {4, 5, 6}}, {{1, 2}, {2, 3, 4}, {4 , 5, 6, 7}}}; // результат ініціалізації:

1000 2300 4560

1200 2340 4567

безструктурна ініціалізація:

int mass [3] [2] = {1, 1, 0, 2, 1, 0};

int a [2] [3] [4] = {1, 0, 0, 0, 2, 3, 0, 0, 4, 5, 6, 0, 1, 0, 0, 0, 2, 3, 0 , 0, 4, 5, 6, 0};

int m [] [3] = {00, 01, 02, 10, 11, 12, 20, 21, 22}

У пам'яті багатовимірний масив розташовується в послідовних комірках по шарах (стрічкам). Елементи з меншими значеннями індексу зберігаються в більш низьких адресах пам'яті. Багатовимірні масиви розташовуються таким чином, що самий правий індекс зростає найпершим. Наприклад, якщо є масив int array[10][3], то в пам'яті за зростанням адрес будуть розміщені елементи:

array [0] [0], array [0] [1], array [0] [2], array [1] [0], array [1] [1], array [1] [2], ..., array [9] [0], array [9] [1], array [9] [2].

Розглянемо тривимірний масив: int M[10][2][3].

M[0], M[9] - одномірні масиви, елементи яких є двовимірні масиви (матриці) int [2] [3], що займають (2 x 3 x sizeof (int)) байт;

M[0][0], M[0][1] - одномірні масиви, елементи яких є одномірні масиви (вектора) int [3], що займають (3 x sizeof (int)) байт;

M[0][1][1], M[0][1][0] - значення типу int, що займають sizeof (int) байт.

У разі опису масиву double M[10][2][3], елементи M[0][1][0], M[0][1][1] є значення типу double, що займають sizeof (double) байт.

Операції над масивами

Операції над масивами - це операції над їх окремими елементами: ініціалізація, введення і виведення значень, перестановка значень, копіювання. Для роботи з масивами, як правило, використовуються цикли.

Найбільш часто при роботі з масивами вирішуються наступні типи завдань:

1. аналіз масиву (всього або частини) для знаходження деякої його характеристики;

2. пошук в масиві, тобто визначення елемента (першого, останнього, всіх) з деяким умовою і знаходження його індексу;

3. побудова масиву по деякому правилу, використовуючи індекси, числа або масиви;

4. перетворення масиву (зміна значення, перестановка елементів, додавання або видалення елементів і т.п.);

5. сортування масиву за деяким критерієм;

6. вивід (представлення) масиву в спеціальному вигляді.

Розглянемо більш детально роботу з масивами.

Ініціалізація масиву - присвоювання кожному елементу масиву значення, відповідного базового типу. Можлива явна ініціалізація масиву тільки при його визначенні:

int intarray [5] = {31, 54, 77, 52, 93};

або:

int ar [5] = {0, 0, 0, 0, 0};

Розмір масиву вказувати необов'язково (компілятор обчислить його за кількістю ініціалізували значень):

int V [] = {12, 1, -5, 22, -4};

Розмір масиву визначається з аналізу умови задачі. Розмірність і розмір масиву разом з типом його елементів визначають загальний обсяг пам'яті, необхідний для розміщення масиву, яке виконується на етапі компіляції.

За допомогою операції sizeof (імя_масива) можна визначити розмір масиву в байтах (тобто розмір ділянки пам'яті, виділеного для масиву).

Тому що всі елементи масиву мають однаковий розмір, то кількість елементів у масиві дозволяє визначити вираз:

sizeof (імя_масива) / sizeof (імя_масива [0]).

Введення масиву реалізується введенням в циклі значень його елементів. При цьому розмір масиву задається константою, наприклад:

```
int age [4];
for (int j = 0; j <4; j ++)
{
cout << "Enter an age:";
cin >> age [j];
}
```

Якщо точну кількість елементів у масиві невідомо, то пам'ять можна виділити по максимуму (наприклад, const int n = 100;), а потім заповнювати тільки частина її. Незважаючи на те, що значення константи n визначається з запасом, надійна програма повинна обов'язково містити перевірку на кількість вводяться елементів.

```
#include <iostream>
#include <conio.h>
#include <iomanip>
void Vvid (int *, int);
```

```
void Druk (int *, int);
const int n = 1000;
int a[1000];
int kol_a;
using namespace std;
int main ()
{
cout << "kol_a = ???";
cin >> kol_a;
if (kol_a > n) {
cout << "size> n" << endl;
_getch ();
return 1;
};
Vvid (a, kol_a);
Druk (a, kol_a);
_getch();
return 0;
};
void Vvid (int *mas, int kol_a)
{for (int j = 0; j <kol_a; j ++) {cout << "Enter an a:";
cin >> mas[j];};};
void Druk (int *mas, int kol_a) {int j;
for (j = 0; j <kol_a; j ++){
cout << setw (4) << mas[j]; if (!((j + 1)% 5)) cout << endl;}
}
```

Вивід елементів масиву можна організувати по-різному.

Вивід елементів одновимірного масиву по одному в рядку:

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

```
using namespace std;
int main ()
{
int intarray [5] = {31, 54, 77, 52, 93};
for (int j = 0; j <5; j ++)
cout << intarray [j] << endl;
getch ();
return 0;}
```

Вивід елементів одновимірного цілочисельного масиву по K в рядку

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
const int n = 10;
int K = 5;
int A [n];
for (int j = 0; j <n; j ++)
A [j] = j * 10;// заповнення масиву
for (int i = 0; i <n; i ++)
{cout << setw (7) << A [i]; // на елемент відводиться 7 позицій
if (! ((i + 1)% K)) cout << endl; // зміна рядка відбувається після виведення K = 5
}
}
```

Наприклад, у припущенні, що на значення відводиться 8 позицій - разом зі знаком і відступом від попереднього значення, маємо:

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

```
#include <iomanip>
using namespace std;
const int n = 10;
int main ()
{
int Random_array [n];
int a = 5, b = 10;
srand(time(0));
for (int index = 0; index <n; index ++) // заповнення масиву
Random_array [index] = a + rand ()% b;
for (int i = 0; i <n; i ++) {cout << setw (8);
cout << Random_array [i];
if (! ((i + 1)% 5)) cout << endl; // вивід елементів масиву по 5 в рядок
}
cout << endl;
getch ();
return 0;
}
```

У цьому випадку елементи масиву будуть розташовані чіткими стовпцями.

Пошук в масиві

Пошук - це перегляд деякого набору однотипних даних з метою знаходження одного або більше елементів, що володіють певним властивістю, або з метою встановлення факту відсутності таких елементів. Всі методи пошуку поділяються на дві групи: пошук в неврегульованих наборі; пошук в упорядкованому наборі.

Пошук в неврегульованих наборі полягає в послідовному переборі елементів масиву і порівнянні їх значень з ключем до того моменту, поки результат порівняння не дасть значення «істина» (зрозуміло, шуканого

значення в масиві може і не бути). При наявності великої кількості елементів у наборі процес пошуку може затягнутися. Найбільш ефективні методи пошуку працюють тільки на впорядкованих наборах даних. Якщо передбачається вести пошук у великому наборі, цей набір необхідно попередньо відсортувати.

Пошук у впорядкованому наборі може виконуватися різними методами. Розглянемо двійковий пошук (бінарний пошук, пошук діленням навпіл). Алгоритм двійкового пошуку допустимо використовувати для знаходження заданого елемента тільки в упорядкованих масивах. Розглянемо його на прикладі масиву, впорядкованого за спаданням.

Принцип двійкового пошуку. Вихідний масив ділиться навпіл і для порівняння вибирається середній елемент. Якщо він збігається з шуканим, то пошук закінчується. Якщо ж середній елемент менше шуканого, то всі елементи лівіше його також будуть менше шуканого. Отже, їх можна виключити із зони подальшого пошуку, залишивши тільки праву частину масиву. Аналогічно, якщо середній елемент більше шуканого, то відкидається права частина, а залишається ліва. На другому етапі виконуються аналогічні дії над залишилася половиною масиву. В результаті після другого етапу залишається ¼ частина масиву. І так далі, поки або елемент буде знайдений, або довжина зони пошуку стане рівною нулю. В останньому випадку шуканий елемент знайдений не буде.

Алгоритм:

1. Визначити left = 0, right = n;

2. Поки left <right виконувати:

знайти індекс середнього елемента в масиві mid = (left + right) / 2;

якщо для елемента масиву з індексом mid значення ключа> шуканого, покласти right = mid-1, інакше якщо значення ключа <шуканого, покласти left = mid + 1, інакше вийти з циклу (елемент з індексом mid знайдений);

3. Якщо значення ключа елемента з індексом mid збігається з заданим, пошук успішний, інакше - шуканого елемента в масиві немає.

```
#include <stdlib.h>
#include <iostream>
#include <conio.h>
#include <iomanip>
using namespace std;
int poshyk (int * mas, int n, int find);
int main ()
{
const int n = 10; int s [n];
int find, i;
for (i = 0; i <n; i ++) s [i] = i * 10; // заповнення масиву числами
for (i = 0; i <n; i ++) cout << setw (4) << s [i]; // вивід масиву
cout << endl;
cout << "\n value of find:"; cin >> find; // введення числа для пошуку
int res = poshyk (s, n, find);
if (res == - 1)
cout << "\n no element";
else
cout << "\n element nomer" << res;
cout << endl;
getch ();
return 0;
}
int poshyk (int * mas, int n, int find)
{int left = 0, right = n-1, mid = 0;
while (left <= right)
{mid = (left + right + 1) / 2;
if (mas [mid]> find) right = mid-1;
else if (mas [mid] <find) left = mid + 1;
else break;
```

```
}
if (mas [mid] == find) return mid;
else return -1;
}
```

Сортування масивів

Сортування - це процес упорядкування деякої послідовності нумерованих об'єктів відповідно до заданих ознакою. Мета сортування - полегшити подальший пошук елементів у відсортованій послідовності.

При вирішенні задачі сортування зазвичай висувається вимога мінімального використання додаткової пам'яті, тобто зокрема, неприпустимість застосування додаткових масивів. Для оцінки швидкодії алгоритмів різних методів сортування, як правило, використовують два показники: кількість присвоювань і кількість порівнянь.

Всі методи сортування можна розділити на дві великі групи: прямі методи сортування і поліпшені методи сортування. Прямі методи сортування за принципом, який лежить в основі методу, поділяються на три підгрупи:

сортування обміном («бульбашкова» сортування); сортування вставкою (включенням); сортування вибором (виділенням).

Прямі методи на практиці використовуються досить рідко, оскільки мають відносно низьку швидкодію. Однак вони добре показують суть заснованих на них поліпшених методів. У той же час, в деяких випадках (як правило, при невеликій довжині масиву і / або особливому вихідному розташуванні елементів масиву) деякі з прямих методів можуть навіть перевершити поліпшені методи). Поліпшені методи сортування, ґрунтуючись на тих же принципах, що і прямі, використовують деякі оригінальні ідеї для прискорення процесу сортування. З точки зору програміста найбільший інтерес представляють сортування масивів, рядків, файлів.

Методи вставки і вибору виявляються в середньому приблизно еквівалентними і в кілька разів (в залежності від довжини масиву) краще, ніж метод обміну («бульбашки).

Дослідження алгоритмів прямих методів сортування показали, що як за кількістю порівнянь, так і по числу присвоювань вони мають квадратичну залежність від довжини масиву n. Виняток становить метод вибору, який має число присвоювань порядку n * ln (n).

Контрольні запитання:

1) Що таке масив?
2) Назвіть основні характеристики масивів.
3) Що таке багатомірні масиви?
4) Назвіть способи задання масивів.
5) Назвіть основні операції над масивами
6) Перелічіть алгоритми пошуку в масиві.
7) Перелічіть алгоритми сортування масивів

Оператори вибору

Оператори умови (керуючі оператори) це механізми, за допомогою яких можна змінювати порядок виконання програми. C++ надає три категорії операторів управління програмою: ітераційні оператори (оператори циклів), оператори вибору і оператори переходів. Ітераційні оператори - це while, for і do / while. Вони найчастіше називаються циклами. Оператори вибору або умовні оператори - це if і switch. Оператори переходу - це break, continue і goto. (Оператор return, в принципі, також є оператором переходу, оскільки він впливає на програму.) Функція exit () вона також впливає на виконання програми.

Вирази C/C++ можна розбити на декілька категорій:

- порожні або нуль- вирази;
- оголошення та декларації;
- складені вирази;
- послідовні вирази;
- умовні вирази (if, if...else);
- оператори вибору (switch);
- циклічні оператори (for, while, do...while);
- оператор переривання (break);
- оператор goto;
- оператор повернення return;
- вирази перехоплення та обробки виключень.

Вирази закінчуються символом ‘;’ складені вирази закінчуються символом ‘}’. Вирази можуть починатись з мітки (кажуть, що такий вираз є поміченим).

Порожні вирази.

Порожні вирази або Null – вирази містять лише синтаксичний роздільник – символ ‘;’. Виконання такого виразу не дає жодного ефекту, проте порожні вирази мають сенс. Наприклад, в процедурах швидкого сортування масивів, коли шукається індекс останнього елемента масиву, який менший заданого значення v, наприклад

while (a[++i] < v);

Тут тіло циклу є порожнім, тому основне завдання – пошуку індексу – є вирішеним через перебігання по елементах масиву в циклі. Тут null- вираз є необхідним; в противному випадку будь-який вираз, який буде розміщений в тілі циклу повинен бути обов’язково виконаний на кожному кроці циклу.

Оголошення

В даному розділі мова буде вестись про оголошення змінних. Оголошення функцій розглядатиметься далі.

Оголошення визначає тип змінної і її атрибути. Зазвичай це об’єднується з визначенням змінної, тобто її розміщення в області пам’яті, де вона буде фізично зберігатись. Декларації мають вид

modifiers Type ident_list;

де:

modifiers – модифікатори (які є опційними) визначають атрибути декларованої змінної. Ці атрибути ще називають класами пам’яті змінної. Атрибути (тобто const, static і ін.) детально розглядались в попередній лекції. Якщо модифікаторів є декілька, то вони оголошуються в порядку слідування з розділенням між собою пропуском, а не комою.

Type визначає тип декларованої змінної (тобто, double, string, unsigned short і ін.) – тут див. лекцію про типи).

ident_list є список ідентифікаторів декларованих змінних, які розділяються між собою комою. Якщо оголошення об’єднано з визначенням,

то ініціалізація можу бути здійснена безпосередньо при оголошенні, наприклад

```
double x = 17.5, y, z = 1;
static const double PI = 3.14;
extern double PIHALF;
const double * const pPi = &PI;
```

В третьому рядку прикладу є декларація змінної без ініціалізації. Очевидно, що змінна PIHALF повинна бути визначена (тобто оголошена без ідентифікатора extern, або з явною ініціалізацією) в іншому модулі програми.

Складені вирази

Складені вирази використовуються там, де синтаксично вимагається один, а треба виконати послідовність виразів. Для вирішення цього формується складений вираз у формі послідовності простих виразів, які охоплюються фігурними дужками. Будь-який вираз, який входить до складу складеного, у свою чергу, також може бути складеним. Символу ‘;’ по закінченню складеного виразу не потрібно. Найбільш наочним прикладом складеного виразу є тіло функції.

Як вже згадувалось, тіло складеного виразу містить блок: усі змінні, які оголошені в тілі блоку мають областю дії тільки цей блок (фактично ділянку від моменту оголошення і до завершення блоку). Коли управління виходить за межі блоку усі локальні змінні будуть автоматично знищені, наприклад:

```
{
    int i = 5;
    {
        int k = fun(i);
        i += k;
    }
    cout << "i=" << i << endl;
}
```

В наведеному прикладі складений вираз містить три вирази, один з яких, у свою чергу, є також складеним. Коли весь блок виконається, змінні і та k перестануть існувати і їх імена можуть використовуватись як ідентифікатори для будь-яких інших потреб.

Обчислювальні вирази

Обчислювальними виразами вважаються вирази, які повертають значення. Іноді, кажуть, що обчислювальний вираз – це вираз, який володіє значенням. Протягом виконання такого виразу це значення обчислюється і може бути опущеним, тобто ігноруватись. У випадку ігнорування сенс виразу полягає у виконання дій, які в нього закладені. Прикладами таких ігнорувань є оператори присвоєння, інкременту та декременту, виклики функцій тощо. Приклад:

```
1.  #include <iostream>
2.  #include <cstdio>
3.  using namespace std;
4.
5.  int main()
6.  {
7.     int k = 7, m = 8;
8.     ++k;
9.     k+1;
10.    k = 5;
11.    printf("Continue?");
12.    k > m ? ++k : --m;
13.    new double(3.5);
14. }
```

В наведеному прикладі є декілька обчислювальних виразів: інкремент у рядку 8, присвоєння у рядку 11, тенарний оператор у рядку 12 та створення

об’єкту у рядку 13 (без присвоєння адреси новоствореного об’єкта довільній змінній). Вираз в рядку 9 (k+1) не має жодного ефекту і може бути проігнорований компілятором (в мові java він був би помилковим). Створення об’єкту в рядку 13 також не має жодного сенсу, оскільки новостворений динамічний об’єкт є недоступним. Проте компілятор не може ігнорувати такі порожні за змістом дії, а тому будуть виконані усі дії по створенню динамічного об’єкта. З іншого боку в конструктор новостворюваного об’єкта можуть бути закладені деякі важливі дії. В цьому випадку сенс в створенні об’єкта є. Але при цьому треба пам’ятати, що динамічна зменшилась у розмірі і в ній з’явилось недоступне сміття (в мові java менеджер пам’яті автоматично знищить об’єкт, на який не існує жодного посилання).

Умовний вираз

Більшість операторів управління програмою в будь-яких комп'ютерних мовах, включаючи C++, грунтуються на перевірці умов, що визначають, якого роду дію необхідно виконати. В результаті перевірки умов можна отримати істину або хибу. На противагу іншим мовам, де вводиться спеціальний тип для зберігання істини(true) і хиби(false), в C++ істині відповідає будь-яке ненульове значення, включаючи від'ємні числа. Хибному твердженню (брехні, false) відповідає нуль. Даний спосіб опису істини і хиби реалізований в C++ від початку, оскільки він надає можливість легкого написання ефективних підпрограм.

Умовний вираз має дві форми. Найпростіша з них має вид

if (b) stmt

де b є логічним виразом, а stmt - звичайним. Напротивагу мові Java в мові C не існує логічного типу, тобто логічний вираз не повертає значення типу bool. Тут істинним вважається будь-яке значення, яке не рівне нулю (очевидно, що для типу, який використовується, повинен бути визначений нуль, зокрема для вказівників таким значенням є константа NULL).

В мові C++ з’явився тип bool із двома логічним значенням true та false. При цьому true відповідає ненульовому значенню, а false – нульовому. Ці значення також можуть використовуватись в операторі if. При цьому підхід мови C до обчислення логічних значень повністю підтримується.

Після обчислення значення b виконання оператора if може бути перерване без виконання stmt, якщо результатом b було хибне значення (тобто нуль);

виконається вираз stmt, якщо результатом b було істинне значення (true).

Зауважимо, що синтаксично вимагається тільки один вираз після закриваючої круглої дужки. Якщо необхідно, щоб виконалось декілька, то треба формувати складений оператор, наприклад

```
if ( s != 0 )
cerr << "Something went wrong. Quitting." << endl;
exit(1);
```

У наведеному прикладі функція exit завжди негайно перерве виконання програми незалежно від результату виразу (s != 0). Якщо модифікувати програму

```
if ( s != 0 ) {
cerr << "Something went wrong. Quitting." << endl;
exit(1);
}
```

То програма буде перервана лише у випадку істинності виразу (s != 0).

Типовою помилкою є розміщення символу ‘;’ після круглих дужок, які визначають умову оператора if:

```
if ( s != 0 );
{
cerr << "Something went wrong. Quitting." << endl;
exit(1);
}
```

В такому випадку виразом оператора if буде порожній оператор; оператор cerr і функція exit виконаються завжди.

Підхід мови С до обчислення логічних значень може бути причиною логічних помилок, які не виявляються компілятором (таке неможливо в мові Java), наприклад

if (a = b) stmt // probably WRONG !!

Якщо у наведеному прикладі a і b є змінними цілого типу, а присвоєння є обчислювальним виразом, то результатом дії оператора присвоєння ' a = b' є ціле число, яке рівне значенню лівої частини оператора присвоєння, тобто b. З іншого боку, вирази типу (a = b) у операторі if мають на увазі оператор порівняння, тобто рівність значення змінної a змінній b. Для вирішення цього завдання треба використовувати булівський оператор порівняння на рівність, тобто замість (a = b) треба писати (a == b).

Інша форма умовного виразу передбачає використання вітки else на той випадок, якщо логічний вираз поверне хибне значення

if (b) stmt1

else stmt2

Тут b є обчислювальний вираз, який повертає логічне значення, stmt1 та stmt2 є виразами (у випадку потреби складеними). Найпершим обчислюється значення b, а тоді:

виконається вираз stmt1 і про ігнорується вираз smt2, якщо результатом b є true;

виконається вираз stmt2 і про ігнорується вираз smt1, якщо результатом b є false.

Обидва вирази stmt1 і stmt2 можуть також бути умовними. Це дозволяє генерувати дуже розгалужені умовні вирази, які можуть бути складними для розуміння. Для того, щоб спростити їх розуміння, треба дотримуватись наступного правила: кожна поява else, закриваючи існуючий оператор if, відкриває новий, наприклад, наступний вираз

if (b1) stmt0 else if (b2) stmt1 if (b3) stmt2

else stmt3 else stmt4

є еквівалентний такому

```
if ( b1 )                // 1
    stmt0
else                     // 1
    if ( b2 )            // 2
        stmt1
        if ( b3 )        // 3
            stmt2
        else             // 3
            stmt3
    else                 // 2
        stmt4
```

Тут кожному оператору if відповідає власне else. Ця пара відмічена однаковим числом, яке задане в коментарях.

Інший приклад

```
1. if ( val >= 0 )
2. {
3. if ( val > 9 ) cout << "Too large" << endl;
4. }
5. else
6. cout << "Too small" << endl;
```

В цьому прикладі оператор if з рядка 3 повинен обов’язково взятий в дужки, оскільки оператор else з рядка 5 буде стосуватись саме його, а не найпершого оператора if. А тому, якщо по контексту завдання потрібно відношення до if з рядка 1, то обов’язковим є використання вкладеного if як складеного оператора.

Оператор вибору - switch

Оператор вибору (switch - вираз) можу бути завжди замінений умовними виразами. Проте з точки зору читабельності коду пропонується використовувати саме його. Цей оператор відповідає оператору case мову Pascal чи Ada, незважаючи на те, що його реалізація є ближчою до обчислень мови Fortran. В загальному випадку оператор switch має вид

```
switch (integ_expr)
{
case const1: list1;
case const2: list2;
// ...
 default: list;
}
```

Тут integ_exp є вираз, який повертає значення цілого типу; const1, const2, ..., є константними виразами (тобто їх значення є відомі під час компіляції) з цілими значеннями; list, list1, list2, ..., є набором виразів (можливо порожніми). Константи const1, const2, і ін. можуть бути задані літеральними значеннями цілого типу, або виразами, що повертають значення цих типів, наприклад

```
const i =10;
switch (k)
{
case i+1: cout<<"i++";
case 10: cout<<"10" ;
}
```

Кількість операторів case (тобто кількість варіантів вибору) є необмеженою. Константи, які в них використовуються повинні бути унікальними. Оператор default є опційним і не може появлятись більше одного разу. Це не означає, що він повинен бути останнім оператором.

Коли оператор switch виконаний, то напочатку виконання буде обчислено значення integ_expr. Якщо це значення рівне значенню однієї з констант операторів case, то управління буде негайно передано виразу цього оператора. Якщо значення integ_exp не збігається із значенням жодною з констант, то управління буде передане виразу оператора default; якщо це оператор є відсутній - то оператор switch завершить своє виконання.

Якщо управління передано виразу одного з case чи default, то будуть виконані усі оператори, які слідують за символом двокрапки і до закінчення тіла оператора switch (тобто оператори case і default є "прозорими"). Якщо є потреба перервати виконання операторів раніше, то треба використовувати оператори break, return або goto.

Використання оператора switch ілюструє наведена нижче програма. В ній функція sw під час виконання друкує певну кількість символів '*'. Власне друк реалізується викликом функції g(). Різна кількість викликів функції g реалізує різну кількість надрукованих на екрані символів. При цьому ця різна кількість викликів, у свою чергу залежить від аргументу функції sw: якщо аргумент рівний 4, то 4 виклики і відповідно 4 символи '*'; якщо аргумент рівний 5, то 2 символи '*'; якщо аргумент рівний 2 або 3, то 0 символів '*'; для усіх решту значень цілого типу буде надруковано 3 символи.

Приклад

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

 void g( )
 {
   cout << '*';
  }
 void show(int k)
 {
    cout << k << ": ";
```

```
   switch ( k )
   {
     default: g( ); g( ); g( ); break;
            case  5: g( ); g( );
     case  3:
     case  2: break;
     case  1: g( ); g( ); g( ); g( );

   }
   cout << endl;
 }
 int main()
 {
   show (9);
   show (5);
   show (4);
   show (3);
   show (2);
   show (1);
   show(0);
   return 0;
 }
```

Сесія програми є такою:

```
9: ***
5: **
4: ***
3:
2:
1: ****
0: ***
```

Зазначимо, що для аргументів відмінних від 1, 2, 3, 5 управління буде передано виразу оператора default (рядок 14); функція g буде викликано один раз; оскільки після цього немає переривання оператора switch, то управління буде передано виразу оператора case 5, вираз якого складається з двох викликів функції g. Далі аналогічно управління до виразу оператора case 3. оскільки він порожній, то до виразу оператора case 2. Вираз цього оператора складається з оператора break, а це означає, що на цьому виконання оператора switch закінчиться. Тому для значень аргументу, які відмінні від 1, 2, 3, чи 5, буде надруковано три символи ‘*’. Якщо замість оператора break для переривання виконання switch було б використано оператор return, то б відбулось негайне завершення виконання функції show(int).

Функція hexVal з програми, яка наводиться нижче, переводить символьне представлення числа у формат цілого числа (або -1 якщо символу не відповідає жодне ціле число):

Приклад

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

int hexVal(char c)
{
  switch ( c )
  {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      return c - '0';

    case 'a': case 'b': case 'c':
    case 'd': case 'e': case 'f':
      return 10 + c - 'a';
    case 'A': case 'B': case 'C':
```

```
        case 'D': case 'E': case 'F':
            return 10 + c - 'A';
        default: return -1;
    }
}
int main()
{
    cout << "A = " << hexVal('A') << endl;
    cout << "f = " << hexVal('f') << endl;
    cout << "9 = " << hexVal('9') << endl;
    cout << "b = " << hexVal('b') << endl;
    cout << "Z = " << hexVal('Z') << endl;
return 0;
}
```

В рядках 8 і 9 групуються декілька порожніх операторів case в залежності від ідентичності завдань, які повинні виконуватись для кожної групи. Сесія програми є такою:

```
A = 10
f = 15
9 = 9
b = 11
Z = -1
```

Оператор ?:

Оператор? може використовуватися для заміни стандартної конструкції if / else:

if (умова) вираз;

else вираз;

Обмеженням в даному випадку є використання єдиного виразі як після if, так і після else.

Оператор? називається тріадним оператором, оскільки йому потрібно три операнда і він має наступний вигляд:

вираз1? вираз2: вираз3

де вираз1, вираз2 і вираз3 - це вирази.

Оператор? працює таким чином. Обчислюється вираження1. Якщо воно істинне, обчислюється вираження2 і вся конструкція одержує обчислене вираження. Якщо вираження1 хибне, обчислюється вираз3 і вся конструкція одержує обчислене вираження. Наприклад:

```
x = 10;
y = x> 9? 100: 200;
```

В даному прикладі у отримує значення 100. Якби х було менше, ніж 9, то у отримало б значення 200. Нижче наведено фрагмент програми, що виконує такі ж дії, але з використанням операторів if / else:

```
x = 10;
if (x> 9) y = 100;
else y = 200;
```

Використання оператора? для заміни if / else не обмежується привласненням. Треба пам'ятати, що всі функції (крім оголошених як void) можуть повертати значення. Отже, допустимо використання одного або декількох викликів функції у виразах. Коли зустрічається ім'я функції, функція відповідно виконується для визначення значення, що повертається. Тому можливо виконати одну або кілька функцій, використовуючи оператор?, Помістивши їх у вирази, що утворюють операнди. Наприклад:

```
#include <stdio.h>
int f1 (int n), f2 (void);
int main (void)
{
int t;
printf (":");
scanf ("%d", &t);
```

```
t? f1 (t) + f2 (): printf ("Zero Entered");
return 0;
}
int f1 (int n)
{
printf ("%d", n);
return 0;
}
int f2 (void)
{
printf ("entered");
return 0;
}
```

В даному прикладі, якщо ввести 0, викличеться функція printf () і з'явиться повідомлення «Zero Entered». Якщо ввести будь-яке інше число, то виконуються функції f1 () і f2 (). Слід зауважити, що в даному прикладі значення, що повертається оператором?, Відкидається.Навіть якщо ні f1 (), ні f2 () не повертають інформативного значення, вони не можуть бути визначені як повертають тип void, оскільки це не дозволить застосовувати їх у виразі. Тому функції просто повертають 0.

Використовуючи оператор?, Можливо переписати нашу програму в такий спосіб:

```
#include <stdio.h>
int main (void)
{
int magic = 123; /* Шукане число */
int guess;
printf ("Enter your guess:");
scanf ("%d", &guess);
if (guess == magic)
```

```
{
printf ("** Right **");
printf ("%d is the magic number", magic);
}
else
guess> magic? printf ("High"): printf ("Low");
return 0;
}
```

Мітки

Будь-який активний вираз (якщо він не оголошення чи декларація) може бути відмічений (помічений) міткою. Мітка визначається будь-яким ідентифікатором, за яким слідує символ ‘:’, а далі – вираз. Мітки використовуються для організації прямих переходів з оператора goto, наприклад:

Приклад:

```
#include <iostream>
 using namespace std;
 int main(void)
 {
    int tab[2][2][2] = {1,2,3,4,5,6,7,8};
    bool present = false;
     for (int i = 0; i < 2; i++)
       for (int j = 0; j < 2; j++)
           for (int k = 0; k < 2; k++)
              if (tab[i][j][k] == 5)
              {
                 present = true;
                 goto LAB;
              }
```

```
LAB:if(present)
        cout <<  "5 is present in the array" << endl;
    else
        cout << "there is no 5 in the array" << endl;  }
```

В наведеній програмі здійснюється пошук елемента масиву розмірністю 3, який рівний значенню 5. Якщо такий елемент знайдено (рядок 11), то змінна present набуде значення true і здійсниться перехід на оператор, який помічений міткою LAB. Оскільки ідентифікатор LAB знаходиться за межами циклів, то, тим самим, здійсниться вихід за межі циклів. Зазначимо, що у випадку використання оператора break, вихід б здійснювався тільки за межі самого вкладеного циклу, який саме і містив break. Якщо елемента масиву tab, який був би рівний значенню 5 не знайдено, то б повністю виконались усі цикли і змінна present зберегла б значення рівне false. В операторі switch може використовуватись інший тип міток, але про них мова буде вестись в розділі про оператор вибору switch.

Контрольні запитання:

1) Що таке порожній вираз?
2) Що таке складений вираз?
3) Дайте визначення алгоритму розгалуження.
4) Особливості використання оператора вибору if.
5) Особливості застосування оператору множинного вибору.
6) Що таке тернарний оператор?
7) Особливості використання міток.

Циклічні оператори

Ітераційні вирази (або циклічні оператори) зазвичай використовуються тоді, коли треба повторити виконання деякої частини коду. Цей фрагмент коду буде виконуватись до тих пір, доки умову повторення буде мати істинне значення. Як тільки умова циклу набуде хибного значення, то цикл негайно завершиться. Уся відповідальність за керуванням значення умови циклу покладена на програміста.

Ще одним шляхом завершення циклу є використання операторів return, break або goto.

В мові С/С++ існує три види циклічних операторів. Кожен з них може бути трансформований в інший. Вибір конкретного виду повністю залежить від розробника і може бути найкращим в конкретних випадках.

Цикл while

Цей вид циклічного оператора називається оператором з передумовою. Його загальний вид є таким

while (b) stmt

Тут b є виразом (умовою повторення), який володіє логічним значенням; stmt – вираз, який буде повторюватись (тіло циклу). Наведений синтаксис приводить тільки один вираз тіла циклу. Якщо є потребу у повторення набору виразів, то треба сформувати складений оператор.

Алгоритм виконання оператора з передумовою є таким:

- на початку обчислюється значення виразу b і конвертується у логічне значення за схемою, яка приводилась у розділі про оператор if;
- якщо значення b є хибним, то виконання оператора while переривається і управління передається до наступного після тіла оператора (тобто stmt не виконається);
- якщо значення b є істинним, то stmt виконається і управління знову буде передано оператору while (тобто на перший крок алгоритму.

Наприклад, наведений нижче фрагмент коду буде шукати перше натуральне число у формі 2n, яке є більше за наперед задане натуральне число lim:

```
int numb = 1;
while ( numb <= lim ) numb *= 2;
```

Наступний код

```
int age = 0;
while ( age < 5 || age > 100) cin >> age;
```

буде читати з стандартного потоку вводу введені числа з діапазону [5, 100].

В даній формі циклу важливим є те, що умова циклу, принаймні один раз виконається, а це є важливим, якщо в цю умову закладені потрібні далі дії. Ще одним моментом є те, що тіло циклу може жодного разу не виконатись (якщо умова повторення циклу матиме хибне значення вже при першому входженні в while).

Цикл do-while

Цей тип циклічного оператора називають оператором з післяумовою. Він має вид

do stmt while (b)

де b – умова повторення; stmt – тіло ператора. Подібно до попереднього випадку, синтаксис говорить про один вираз в тілі оператора. Якщо в циклі необхідне виконання набору операторів, то треба сформувати складений оператор.

Алгоритм виконання оператора з післяумовою є таким:

– на початку виконується stmt;
– обчислюється значення b і конвертується в логічне значення;
– якщо значення b є хибним, то виконання циклу закінчується і управління передається наступному за оператором while виразу;

– якщо значення b є істинним, то управління повертається до першого оператора stmt і увесь цикл повторюється.

В операторі do...while, на відміну від while..do, вираз stmt на початку обчислюється, а умова виконання циклу визначається вже після цього. Тобто stmt принаймні один раз виконається завжди.

Наведена нижче програма симулює підкидання двох гральних кубиків. Значення кожного кубика отримується, через генератор випадкових чисел. Гра закінчується тоді, коли сума значень кубиків стане рівна 12, тобто значення кожного кубика буде рівна 6:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    int x, y, roll = 0;
    srand(unsigned(time(0)));
    do {
        x =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
        y =  (int)(rand()/(RAND_MAX + 1.)*6)+1;
        cout << "Roll no " << ++roll   << ": ("
             << x << ", " << y << ")" << endl;
    } while (x + y != 12);
}
```

Одним з варіантів виконання програми може бути таким (все залежить від результатів вибору випадкових чисел):

Roll no 1: (1, 5)

Roll no 2: (5, 3)

Roll no 3: (6, 2)

Roll no 4: (5, 3)

Roll no 5: (1, 5)

Roll no 6: (3, 2)

Roll no 7: (3, 3)

Roll no 8: (5, 4)

Roll no 9: (3, 2)

Roll no 10: (6, 6)

Цикл For

Циклічний оператор for називають ітераційним і він має вид

for (init ; b ; incr) stmt

де stmt є виразом, або тілом циклу (може бути також складеним, або порожнім). Вираз b повинен повертати значення, яке повинно конвертуватись в булівське значення і відгравати роль умови завершення циклу. Цей вираз може бути опущеним і тоді по замовчуванні в цикл for буде передаватись значення true. Вираз init може бути

- пропущеним;
- одним або набором операторів – оголошень змінних, які між собою розділяються комами, з/без явної ініціалізації;
- будь-яким набором розділених між собою комами операторів.

'Інкрементна' частина (у синтаксисі оголошена як incr) може бути також пропущеною, одним або набором операторів, які між собою розділяються комою.

Якщо хоча б одна з частин є опущена, то вона все рівно закривається символом ‘;’ або круглою дужкою.

Алгоритм виконання циклічного оператора є таким:

Якщо значення init є непорожнім, то воно виконається. Якщо це є послідовність операторів, то виконаються усі в порядку зліва направо і їх значення будуть ігноруватись. Якщо init є набором оголошень змінних, то треба пам’ятати, що їх область дії тільки даний оператор (разом з тілом). Тобто за межами цикли ці змінні перестануть існувати.

Значення b буде обчисленим і переведеним у тип bool (якщо b є порожнім то у for буде передано true). Якщо b буде мати значення false, то цикл буде перервано.

Тіло циклу (вираз stmt) виконається.

Якщо incr є непорожнім, то виконаються усі вирази цієї частини оператора for, а їх значення будуть проігноровані.

Управління повернеться на пункт 2.

Заборонено оголошення змінних різного типу в частини init. Наприклад, наступний приклад оператора for є неправильним:

```
for (double x = 0, int k = size-1; x < k; x++, k--) {
   // ... WRONG !!!
}
```

Якщо цей код переписати, наприклад, у наступному виді, то він буде правильним:

```
for (int i = 0, k = size-1; i < k; i++, k--) {
   // OK
}
```

Оператор for є найбільш загальною формою циклічного оператора. Вона дуже зручна у випадках, коли наперед відома кількість циклів. Зазвичай на початку ініціалізується змінна, яка надалі відіграє роль лічильника циклу і інкрементується чи декрементується в розділі incr.

Наведемо приклад програми, яка в зворотному порядку сортує елементи масиву цілих чисел:

```
#include <iostream>
 using namespace std;
  void reverse(int *tab, int size) {
    if ( size < 2 ) return;
   for (int i = 0, k = size-1, pom; i < k; i++, k--) {
        pom    = tab[i];
       tab[i] = tab[k];
```

```
        tab[k] = pom;
    }
}
void printTab(int *tab, int size) {
    cout << "[ ";
   for (int i = 0; i < size; i++)
        cout << tab[i] << " ";
    cout << "]" << endl;
}
int main() {
   int tab[] = { 1, 3, 5, 7, 2, 4, -9, 12 };
    int size = sizeof(tab)/sizeof(tab[0]);
    printTab(tab,size);
    reverse (tab,size);
    printTab(tab,size);
}
```

Оператори continue і break

Цих два оператори використовуються всередині циклів будь-якого виду: for, do...while або while (оператор break також може використовуватись всередині оператора switch).

Оператор continue примусово завершує виконання поточного кроку циклу, при цьому управління з циклу не виходить, а переходить на кінець тіла циклу. При цьому надалі цикл буде виконуватись у нормальному режимі. Якщо використовується оператор for, то після оператора continue управління перейде до частини incr. Звертаємо увагу, що примусово завершується лише поточний цикл оператора, а не увесь оператор. Тобто, оператор continue не може використовуватись для переривання виконання циклічного оператора.

В наведеній нижче програмі функція sumPos підсумовує значення усіх невід'ємних елементів таблиці; від'ємні значення ігноруються і в сумуванні ігноруються за допомогою оператора continue (рядок 7):

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

int sumPos(int *tab, int size) {
   int sum = 0;
  for (int i = 0; i < size; i++) {
      if ( tab[i] <= 0 ) continue;
     sum += tab[i];
  }
   return sum;
}

int main() {
   int tab[] = { 1, -3, 5, -7, 2, 0, 9 };
   int sum = sumPos(tab,sizeof(tab)/sizeof(tab[0]));
   cout << "Sum: " << sum << endl;
}
```

Оператор break перериває виконання найближчого циклу, при цьому управління передається наступному за циклічним оператором виразу. Модифікована попередня програма, сумує усі додані лементи масиву до тих пір, доки не зустрінеться перший від'ємний елемент. При цьому усі решта елементів буде проігноровано:

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

int sumuntil(int *tab, int size) {
   int sum = 0;
```

```
    for (int i = 0; i < size; i++) {
        if ( tab[i] <= 0 ) break;
        sum += tab[i];
    }
    return sum;
}

int main() {
    int tab[] = { 1, 3, 5, 7, 0, 4, 9 };
    int sum = sumuntil(tab, sizeof(tab)/sizeof(tab[0]));
    cout << "Suma: " << sum << endl;
}
```

В рядку 7 зустрічання від’ємного елемента масиву перериває виконання циклу і сумування елементів масиву.

Оператор break також може використовуватись всередині оператора switch, де він призводить до переривання виконання цілого оператора switch (див. розділ про оператор switch). Очевидно, що, оскільки continue не перериває виконання поточного оператора, то немає жодного сенсу використовувати цей оператор в тілі switch.

Для тих, хто добре володіє мовою java треба пам’ятати, що варіантів операторів break і continue з мітками в мові C/C++ не підтримується.

Оператор goto

Оператор переходу goto передає управління іншій частині коду і має вид

```
goto label;
```

де label є міткою іншого виразу і тілі тієї самої функції. Після виконання оператора goto управління буде передано саме цьому виразу.

Мова дозволяє будь-які переходи, тобто навіть на блоки, які простим оголошенням змінних навіть без явної ініціалізації. Але такою гнучкістю не

треба зловживати. А тому рекомендується використовувати оператор goto для переходу між блоками, які реалізують якісь дії. Але тут треба пам'ятати, що використання оператора goto ускладнює розуміння програми і може бути причиною дуже складних логічних помилок:

```
#include <iostream>
using namespace std;
int main()
{
    const int st_size = 5;
    char grades[][st_size] = {{ 'A', 'A', 'B', 'C', 'B' },
                             { 'A', 'C', 'C', 'F', 'D' },
                             { 'A', 'C', 'B', 'B', 'A' }};
    int gr_size = sizeof(grades)/sizeof(grades[0]);
    bool isF = false;
    for (int group = 0; group < gr_size; group++)
      for (int student = 0; student < st_size; student++)
        if ( grades[group][student] == 'F' )
        {
          isF = true;
          goto THEEND;
        }
  THEEND:
    if (isF) cout << "There was an \'F\'" << endl;
    else     cout << "There was no \'F\'" << endl; }
```

Оператор return.

Оператор return перериває поточне виконання функції і дає змогу повернути з функції значення. При цьому поточне управління до зовнішньої функції, а саме до оператора, який слідує за викликом функції, виконання якої переривалося оператором return. Якщо функція оголошена як void, то оператор допускається, але без значення повертання. З іншого боку для void-

функцій використання оператора return є необов’язковим. Якщо функція повертає відмінне від void, то використання оператора return є обов’язковим, причому формат використання повинен бути

return val;

де val є вираз, значення якого, є тим значенням, яке функція поверне після свого виконання. А тому воно повинно збігатись за типом з типом, значення, яке функція повинна повернути.

```
#include <stdio.h>
int mul(int a, int b);
int main(void)
{
  int x, y, z;
  x = 10;   y = 20;
  z = mul(x, y);          /* 1 */
  printf("%d", mul(x,y));  /* 2 */
  mul(x, y);              /* 3 */
  return 0;
}
int mul(int a, int b)
{  return a*b;}
```

Контролні запитання:

1) Дайте визначення циклічного алгоритму.
2) Назвіть основні види циклічних алгоритмів.
3) Назвіть відмінності використання циклів з перед і після умовою.
4) Для чого використовується оператор goto?
5) Особливості запису та використання оператора return.

Обробка рядків символів

Поняття стрічки в С++

Стрічка - послідовність (масив) символів. Якщо у виразі зустрічається одиночний символ, він повинен бути обмежений одинарними лапками. При використанні у виразах, рядок необхідно обмежити подвійними лапками. Ознакою кінця рядка є нульовою символ \ 0. В С ++ рядок можна описати за допомогою масиву символів (масив елементів типу char), в якому слід передбачити місце для зберігання ознаки кінця рядка.

Наприклад, опис рядки з 25 символів має виглядати так:

char s [25];

Тут елемент s [24] призначений для зберігання символу кінця рядка.

char s [7] = "Привіт";

Можна описати і масив рядків:

char s [3] [25] = {"Приклад", "використання", "стрічок"};

Визначено масив з 3 рядків по 25 байт в кожній.

Для роботи з вказівниками можна використовувати тип даних (char *). Адреса першого символу буде початковим значенням сказівника.

Розглянемо приклад оголошення і виведення рядків.

```
#include <iostream>
#include <Windows.h>
using namespace std;
int main ()
{ SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
// Описуємо 3 рядка, s3- вказівники
char s2 [20], * s3, s4 [30];
cout << "s2 ="; cin >> s2; // ввід рядка s2
cout << "s2 =" << s2 << endl;
// Запис в s3 адреси рядка, де зберігається s4. Тепер в змінних
```

```
// (вказівниках) s3 і s4 зберігається значення однієї і тієї ж адреси
s3 = s4;
cout << "s3 ="; cin >> s3; // ввід рядка s3
// Вивід на екран рядків s3 і s4, хоча в результаті присвоювання s3 = s4;
// Тепер s3 і s4 - це одне і теж
cout << "s3 =" << s3 << endl;
cout << "s4 =" << s4 << endl;
system ("pause");
return 0;
}
```

Функція cin вводить рядки до зустрівся пробілу. Більш універсальною функцією є getline.

*cin.getline (char * s, int n);*

Функція призначена для введення з клавіатури рядка s з пробілами, в рядку не повинно бути більш n символів. Отже, для коректного введення рядків, що містять пробіл, необхідно в нашій програмі замінити cin >> s на cin.getline (s, 80).

Операції над рядками

Рядок можна обробляти як масив символів, використовуючи алгоритми обробки масивів або за допомогою спеціальних функцій обробки рядків, деякі з яких наведено нижче. Для роботи з цими рядками необхідно підключити бібліотеку cstring.

Деякі функції роботи з рядками:

Прототип функції	Опис функції
size_t strlen (const char * s)	обчислює довжину рядка s в байтах.
char * strcat (char * dest, const char * scr)	приєднує рядок src в кінець рядка dest, отримана терміну повертається в якості результату

char * strcpy (char * dest, const char * scr)	копіює рядок scr в місце пам'яті, на яке вказує dest
char strncat (char * dest, const char * dest, size_t maxlen)	приєднує рядок maxlen символів рядка src в кінець рядка dest
char * strncpy (char * dest, const char * scr, size_t maxlen)	копіює maxlen символів рядка src в місце пам'яті, на яке вказує dest
int ctrcmp (const char * s1, const char * s2)	порівнює два рядки в лексикографічному порядку з урахуванням відмінності великих і малих літер, функція повертає 0, якщо рядки збігаються, повертає - 1, якщо s1 розташовується в упорядкованому за алфавітом порядку раніше, ніж s2, і 1 - в протилежному випадку.
int strncmp (const char * s1, const char * s2, size_t maxlen)	порівнює maxlen символів двох рядків в лексикографічному порядку, функція повертає 0, якщо рядки збігаються, повертає - 1, якщо s1 розташовується в упорядкованому за алфавітом порядку раніше, ніж s2, і 1 - в протилежному випадку.
double atof (const char * s)	перетворить рядок в дійсне число, в разі невдалого перетворення повертається число 0
long atol (const char * s)	перетворить рядок в довге ціле число, в разі невдалого перетворення повертається 0
char * strchr (const char * s, int c);	повертає покажчик на перше входження символу c в рядок, на яку вказує s. Якщо символ c не знайдений, повертається NULL
char * strupr (char * s)	перетворює символи рядка, на яку вказує s, в символи верхнього регістру, після чого повертає її

Тип даних string

Крім роботи з рядками, як з масивом символів, в С ++ існує спеціальний тип даних string. Для введення змінних цього типу можна використовувати cin, або спеціальну функцію getline.

getline (cin, s);

Тут s - ім'я введеної перемінної типу string.

При описі змінної цього типу можна відразу присвоїти значення цієї змінної.

string var (s);

Тут var - ім'я змінної, s - строкова константа. В результаті цього оператора створюється змінна var типу string, і в неї записується значення строкової константи s. Наприклад,

string v («Hello»);

Створюється рядок v, в яку записується значення Hello.

Доступ до i-го елементу рядка s типу string здійснюється стандартним чином s [i].Над рядками типу string визначені наступні операції:

- присвоювання, наприклад s1 = s2;

- об'єднання рядків (s1 + = s2 або s1 = s1 + s2) - додає до рядка s1 рядок s2, результат зберігатися в рядку s1, приклад об'єднання рядків:

```
#include <iostream>
#include <Windows.h>
#include <string>
using namespace std;
int main () {
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
string a, b;
cout << "a ="; getline (cin, a);
cout << "b ="; getline (cin, b);
a += b;
```

```
cout << "a =" << a << endl;
return 0; }
```

- порівняння рядків на основі лексикографічного порядку: s1 = s2, s1! = s2, s1 <s2, s1> s2, s1 <= s2, s1> = s2 - результатом буде логічне значення;

При обробці рядків типу string можна використовувати наступні функції:

- s.substr (pos, length) - повертає підрядок з рядка s, починаючи з номера posдовгою length символів;

- s.empty () - повертає значення true, якщо рядок s порожня, false - в іншому випадку;

- s.insert (pos, s1) - вставляє рядок s1 в рядок s, починаючи з позиції pos;

- s.remove (pos, length) - видаляє з рядка s підрядок length довгою posсимволів;

- s.find (s1, pos) - повертає номер першого входження рядка s1 у рядок s, пошук починається з номера pos, параметр pos може бути відсутнім, в цьому випадку пошук йде з початку рядка;

- s.findfirst (s1, pos) - повертає номер першого входження будь-якого символу з рядка s1 у рядок s, пошук починається з номера pos, який може бути відсутнім.

Кодування символів

Інформацію в комп'ютерах найлегше представляти числами. Щоб зберегти в комп'ютері літеру, треба домовитися про відповідність літер і чисел. Після деякого часу суперечок в англомовному світі прийшли до 7-бітного (тобто старший біт в байті з 8 біт - завжди 0) кодування ASCII. Оскільки сучасні комп'ютери походять від тих, то основою всіх сучасних кодувань є саме ASCII.

В різних мовах дуже багато різних символів; наприклад, китайських ієрогліфів близько 100 тисяч. Як же їх кодують, якщо в байті всього 8 біт і 256 значень?

Стара ідея полягала в тому, що до ASCII додавали ще 128 значень (зі старшим бітом 1) і кодували ними інші алфавіти, а програма (чи операційна система, якщо програма не уточнює) визначала, яке конкретно кодування застосовується, і чи число 200 означає "х", "Č" чи "И". Звісно, це не дозволяло використовувати одночасно кілька мов, і не задовільняло потреби ієрогліфічних Китая, Японії та Кореї. Тому виникла ідея багатобайтового кодування - юнікода, де різні символи займають різну кількість байтів (що ускладнює роботу з текстом). Втім, слід враховувати, що існує кілька варіантів представлення юнікода, хоча всі вони використовують одне й те саме кодування.

Варіанти кириличних (зокрема, українських) кодувань:

KOI8-U - різновид KOI8-R; відрізняється тим, що якщо занулити перший біт, більшість літер стануть схожими літерами ASCII, наприклад, "Привіт" стане "pRIW&T". При цьому, звісно, повністю губиться алфавітний порядок символів.

CP866 (інші назви - "альтернативне кодування", "кирилиця DOS") - розробка компанії Microsoft; відрізняється більш-менш алфавітним розташуванням літер (маленькі літери розірвані на дві групи, щоб зберегти символи псевдографіки, які активно використовувалися в DOS) і відсутністю питомо українських літер і, ї, є, ґ. Для забезпечення роботи цих літер робилися різні спроби ввести схожі кодування, але жодне з них не прижилося, і навіть зараз в консолі Windows 7 неможливо повноцінно використовувати літеру і.

CP1251 - стандартне кодування старих версій Windows(9x); сучасні версії також її розуміють. Алфавіт майже збережено (знову ж таки, "особливі" і, ї, є, ґ винесені за межі алфавіту).

ISO 8859-5 - затверджене міжнародним стандартом, алфавіт майже збережено, не містить літери ґ, використовується дуже мало.

Юнікод - як вже сказано, багатобайтовий. Для символів кирилиці виділені коди від U+0400 до U+052F.

В старих версіях це було залишено на програміста, стандартних функцій для роботи з кодуваннями не було. Звісно, таке становище дуже шкодило переносимості програм, і були розроблені стандартні засоби для роботи з різними кодуваннями, для C - бібліотека <locale.h> (clocale в C++) із функцією

char setlocale (int category, const char* locale);*

Ця функція змінює поведінку стандартних функцій C для роботи з рядками у відповідності до локалі і категорії, причому категорії бувають:

LC_COLLATE - впливає на функції strcoll and strxfrm

LC_CTYPE - впливає на функції з cctype, крім isdigit and isxdigit

LC_MONETARY - впливає на інформацію про грошові одиниці з функції localeconv.

LC_NUMERIC - впливає на форматування чисел при вводі-виводі (зокрема, на десяткову кому) і включає LC_MONETARY

LC_TIME - впливає на strftime

LC_ALL - включає все попереднє

Другий параметр - назва локалі - залежить від ОС; Windows розуміє прості назви виду "Ukrainian" чи "Russian"; іншим системам треба давати більш точні вказівки типу "uk_UA.cp1251" чи "en_US.utf8" (у форматі мова_країна.кодування).

Контрольні запитання:

1) Що таке стрічка?
2) Назвіть основні операції над стрічками.
3) В чому особливість використання типу даних string?
4) Особливості запису криличних символів.

Функції

Основні поняття та властивості функції

При програмуванні на мові С++ функція є основним поняттям. По-перше, кожна програма обов'язково містить єдину функцію іменем main (головна функція). Саме функція mian забезпечує точку входа у відкомпілбовану програму. Окрім функції з іменем main, до програми входить довільна кількість неголовних, виконання яких ініціюється прямо чи через посередників викликами з функції main. Всім іменам функцій програми по замовчуванні надається клас пам'яті extern, тобто кожна функція має зовнішній тип компоновки та статичну тривалість існування. Як об'єкт з класом пам'яті extern, кожна функція є глобальною, тобто при визначених умовах доступна у модулі і навіть у всіх модулях програми. Для доступності у модулі функція повинна визначатися у ньому чи оголошуватися перед першим викликом.

Таким чином, кожна програма на мові С++ - це сукупність функцій, кожна з яких повинна визначатися чи принаймні оголошуватися перед її використанням у конкретному модулі програми. У визначенні функції вказується послідовність дій, які виконуються при її виклику, ім'я функції, тип функції (тип значення, яке повертається, тобто тип результату) та сукупність формальних параметрів (аргументів). Кожен формальний параметр не лише перелічується, але й специфікується, тобто для нього вказується тип. Сукупність формальних параметрів визначає сигнатуру функції. Цей термін активно використовується при перевантаженні функцій. Сигнатура функції залежить від кількості параметрів, від їх типів та порядка розташування у специфікації формальних параметрів.

Визначення функції, у якому виділяються дві частини – заголовок та тіло, має такий формат:

тип_функції ім'я_функції (специфікація_формальних_параметрів)

{ тіло_функції }

Тут тип_функції - тип значення, яке повертається функцією, серед них і void, якщо функція не повертає значення. ім'я_функції – ідентифікатор. Імена функцій як імена зовнішні (тип extern) повинні бути фнікальними серед інших імен з модулів, у яких використовуються функції. Специфікація формальних параметрів – це чи порожня, чи void, чи список специфікацій окремих параметрів, у кінці якого може ставитись багато крапок. Специфікація кожного параметра у визначенні функції має вигляд:

тип ім'я_параметра

тип ім'я_параметра= значення_по_замовчуванні

Як вказує формат, для параметра можна задавати (чи ні) значення при замовчуванні. Більше того, синтаксис мови дозволяє параметри без імен, коли останні не використовуються у тілі функції. Використання специфікації параметра без імені відводить місце у спису параметрів. .Далі такий параметр вводиться у функції бз зміни інтерфейса, тобто без зміни програми, що викликає. Така можливість зручна при розвиненні вже існуючої програми завдяки зміні функцій, що входять до неї.

тіло_функції - це завжди блок чи складений оператор. Тобто послідовність оголошень та операторів, записаних у фігурних дужках. Дуже важливим оператором тіла функції є оператор повернення у точку виклику: return вираз; чи return;

вираз у операторі return визначає значення. Яке повертає функція. Саме це значення буде результатом звернення до функції. Тип значення, що повертається, визначається типом функції. Якщо функція не повертає ніякого значення, тобто має тип void, тоді вираз в операторі return не записується. У такому випадку необов'язковим є і оператор return у тілі функції. Потрібні кодикоманд повернення у точку виклику компілятор мови C++ додасть у об'єктний модуль функції автоматично. У тілі функції може бути кілька операторів return. Оператор return можна використати і у функції main. Якщо тип значення, що повертається функцією main, відрізняється від void, тоді це значення аналізується операційною системою. Прийнято, що при

нормальному завершенні програми повертається значення 0. У противному випадку повертається значення, що відрізняється від нуля.

Навіть тоді, коли функція не повинна виконувати ніяких дій і не повинна повертати ніяких значень, тіло функції буде складатися з фігупних дужок {}. Таку функцію можна використовувати при відладці програми.

Приклади визначень функцій з різними сигнатурами:

```
void print (char *name, int value) //Нічого не повертає
{ cout<<”\n”<<name<<value; } // Немає оператора return
float min(float a,float b) // У функції два оператори повернення
{ if (a<b) return a; return b;} // Повертає мінімальне значення з аргументів
float cube (float x) // Повертає значення типу float
{ return x*x*x; } // Підводить до куба дійсного числа
int max(int n,int m) // Поверне значення типу int
{return n<m?m:n; } // Поверне максимальне з значень аргументів
void write(void) // Нічого не повертає і нічого не одержує
{ cout,,”\nНазва :“; } // Завжди друкує одне й те ж
```

Заголовок останньої функції можна записати без типу void у списку формальних параметрів void write() //Відсутність параметрів еквівалентна void.

Наступна функція обчислює на площі дещо незвичайну відстань між двома точками, координати яких передаються як значення параметрів типу double:

```
double norma (double X1, double Y1, double X2, double Y2, double)
{return X2-X1>Y2-Y1?X2-x1:Y2-Y1;}
```

Останній параметр, специфікований типом double, у тілі функції norma() не використовуєтся. Пізніше без зміни інтерфейса можна змінити функцію norma(), додавши до неї ще один параметр. Наприклад, функція може обчислювати відстань між точками (X1,Y1), (X2,Y2) на площині.

При зверненні до функції формальні параметри замінюються фактичними, при цьому суворо дотримується відповідність параметрів.

Суворе узгодження за типами між формальними та фактичними параметрами вимагає запису визначення чи оголошення (прототипу) функції, яке містить відомості про її тип (тип результату) та типи всіх параметрів до її першого виклику. Саме наявність прототипу чи повного визначення дає змогу компілятору контролювати відповідність типів. Прототип (оголошення) функції може зовні майже повністю співпадати з заголовком її визначення:

тип_функції ім'я_функції (специфікація_формальних_параметрів);

Основна різниця – крапка з комою в кінці оголошення (прототипа). Друга відмінність – необов'язковість імен формальних параметрів у прототипу навіть тоді, коли вони є у заголовку визначення функції. Приклади прототипів:

*void print(char *,int); // Не вказані імена параметрів*

float min(float a,float b);

float cube(float x);

int max(int,int m); // Не вказане одне ім'я

void write(void); // Списка параметрів може не бути

double norma(double,double,double,double,double);)

В останньому випадку прототип функції norma() не містить імен всіх параметрів. Компілятор читаючи параметри прототипу функції не може розпізнати конкретні імена формальних параметрів, тому не будуть видаватися попередження при зверненні до функції norma (), яка містить 5 фактичних параметрів, замість перших чотирьох, які насправді використовуються у тілі функції.

Звернення до функції(виклик функції) –це вираз з операцією "круглі дужки". Операндами є ім'я функції(чи вказівник на функцію) та список фактичних параметрів:

ім'я_функції (список_фактичних_параметрів);

Значенням виразу “виклик функції” є поверенене функцією значення, тип якого відповідає типу функції.

Фактичний параметр (аргумент) функції – це у загальному випадку вираз. Відповідність між формальними та фактичними параметрами встановлюється за їх взаємним розташуванням у списках. Фактичні параметри передаються з викликаючої програми у функцію за значенням, тобто обчислюється значення кожного виразу, що є аргументом, і саме це значення використоввується у тілі функції замість відповідного формального параметра. Таким чином, список_фактичних_параметрів є або порожнім, або void, або відокремлені комами фактичні параметри.

Приклад запису функції. Програма, до якої відносяться деякі функції, записані в одному програмному модулі з основною функцією.

```
#include <iostream>
using namespace std;
int max(int n, int m)
{return n<m?m:n;}
void printMY(char*,int);
float cube(float x=0);
int main() //
{
int sum=5, k=2;
sum=max((int)cube(float(k)),sum);
char st[] = "\nsum=";
printMY(st,sum);
return 0;} // sum=8
void printMY(char *name,int value)
{cout<<"\n"<<name<<value;}
float cube(float x) {return x*x*x;}
```

У наведеній програмі виникає потреба у перетворенні типів фактичних параметрів при виклику функцій max() та cube(). Перетворення потрібні для узгодження типів формальних та фактичних параметрів. Для ілюстрації вибрані різні форми запису перетворень – канонічна та функціональна (int)cube(float(k). Для функції max() прототип не потрібен, оскільки вона визначена у тому ж файлі перед викликом функції. Прототипи функцій print() та cube() у програмі потрібні, оскільки їх визначення розташовані після звернення до них. Якщо закоментувати (виділити символами /* */ чи //) прототип довільної з функцій print() чи cube(), тоді компілятор повідомить про помилку. Такі ж повідомлення виникнуть , коли у програмі перенести визначення max() у кінець модуля і не записати прототип у main.

При наявності прототипів функції, що викликаються, не повинні знаходитись у одному файлі (модулі) з функцією, що їх викликає, а можуть записуватися у вигляді окремих модулів чи знаходитися у вже відкомпільованому вигляді у бібліотеці об’єктних модулів. Сказане відноситься не лише до функцій, які пише програмист для приєднання до власної програми, але й до функцій зі стандартних бібліотек компілятора, де вони записані у вигляді об’єктних модулів (відкомпільовані). Оголошення у вигляді прототипів потрібно приєднувати до програми додатково. Найчастіше це роблять за допомогою препроцесорних команд #include <ім’я_файла>.

Подібно до того, як це зроблено у бібліотеці стандартних функцій компілятора, можна оформляти розробку власних програм, що складаються з досить великої кількості функцій, розташованих у різних модулях. Прототипи функцій та оголошення зовнішніх об’єктів (змінних, масивів, тощо) розташовують у окремий файл, який препроцесорною командою #include “ім’я_файла” приєднують на початку кожного з модулів програми. На відміну від бібліотечних функцій компілятора ім’я такого заголовочного файла у команді #include записують не в кутових дужках <>, а в лапках. При цьому не потрібно піклуватися про збільшення розміру створеної програми.

Прототипи функцій потрібні лише при компіляції і не переносяться до об'єктного модуля, тобто не збільшують машинний код. А прототипи тих функцій, які не викликаються у модулі, взагалі не використовуються компілятором.

Наприклад, для наведених вище функцій можна записати такий заголовочний файл:

*void printMy(char *=" номер сторінки", INT=1);*

float min(float a, float b);

float cube(float x=1);

int max(int,int m=0);

void write();

double Norma(double,double,double,double);

Початкові значення параметрів (при замовчуванні).

При визначенні функції специфікація параметра може містити параметри при замовчуванні. Ці значення використовуються тоді, коли при зверненні до функції відповідний параметр не вказується. При запису початкових параметрів (при замовчуванні) потрібно дотримуватися таких узгоджень. Якщо параметр має значення при замовчуванні, тоді всі параметри, які записані праворуч від нього, також повинні мати такі значення. Наприклад, можна таким чином визначити функцію друкування:

void printMy (char name=" Номер будинку", int value=1)*

{cout<<"\n"<<name<<value;}

Залежно від кількості та значень фактичних параметрів у викликах функції на екран виведуться такі повідомлення:

printMy (); //"Номер будинку: 1

printMy ("Номер кімнати: "); // Номер кімнати : 1

printMy (,15);//Помилка–можно не вказувати лише кінцеві параметри списку

Тому зручнішим буде таке розташування параметрів :

*void display (int value=1,char *name="Номер будинку")*

cout<<"\n"<<name<<value;}

Тоді звернення до неї будуть такими:

display(); // Номер будинку: 1

display (15); // Номер будинку: 15

display (6,"Вимірність простору:"); // Вимірність простору: 6

Аргументи функції main (): argv і argc

Іноді при запуску програми буває корисно передати їй яку-небудь інформацію. Зазвичай така інформація передається функції main () за допомогою аргументів командного рядка. Аргумент командного рядка - це інформація, яка вводиться в командному рядку операційної системи слідом за ім'ям програми.

Щоб взяти аргументи командного рядка, використовуються два спеціальних вбудованих аргументу: argc і argv. Параметр argc містить кількість аргументів у командному рядку і є цілим числом, причому він завжди не менше 1, тому що першим аргументом вважається ім'я програми. А параметр argv є покажчиком на масив покажчиків на рядки. В цьому масиві кожен елемент вказує на який-небудь аргумент командного рядка. Всі аргументи командного рядка є строковими, тому перетворення яких би то не було чисел в потрібний двійковий формат має бути передбачено в програмі при її розробці.

Ось простий приклад використання аргументу командного рядка. На екран виводяться слово Привіт і ваше ім'я, яке треба вказати у вигляді аргументу командного рядка.

```
#include <stdio.h>
 #include <stdlib.h>
 int main (int argc, char * argv [])
 {
   if (argc != 2) {
     printf ("What is you name?");
```

```
    exit (1);
  }
  printf (" HI %s", argv [1]);
  return 0; }
```

Якщо ви назвали цю програму name (ім'я) і ваше ім'я Том, то для запуску програми слід в командний рядок ввести name Том. В результаті виконання програми на екрані з'явиться повідомлення Привет, Том.

Якщо в рядку є прогалини, то, щоб з неї не вийшло декілька аргументів, в деяких середовищах цей рядок можна укладати в подвійні лапки. В результаті вся рядок вважатиметься одним аргументом. Щоб докладніше дізнатися, як у вашій операційній системі задаються параметри командного рядка, вивчіть документацію цієї системи.

Дуже важливо правильно оголошувати argv. Ось як це роблять частіше за все:

*char * argv [];*

Порожні квадратні дужки вказують на те, що у масиву невизначена довжина. Тепер отримати доступ до окремих аргументів можна за допомогою індексації масиву argv. Наприклад, argv [0] вказує на першу символьну рядок, якій завжди є ім'я програми; argv [1] вказує на перший аргумент і так далі.

Щоб отримати доступ до окремого символу одного з аргументів командного рядка, введіть в argv другий індекс. Наприклад, наступна програма посимвольно виводить всі аргументи, з якими її викликали:

```
#include <stdio.h>
int main (int argc, char * argv [])
{
  int t, i;
  for (t = 0; t <argc; ++ t) {
    i = 0;
    while (argv [t] [i]) {
      putchar (argv [t] [i]);
```

```
      ++ i;
     }
    printf ("\n");
   }
  return 0;
 }
```

Пам'ятайте, перший індекс argv забезпечує доступ до рядка, а другий індекс - доступ до її окремих символам.

Зазвичай argc і argv використовують для того, щоб передати програмі початкові команди, які знадобляться їй при запуску. Наприклад, аргументи командного рядка часто вказують такі дані, як ім'я файлу, параметр або альтернативне поведінку. Використання аргументів командного рядка додає вашій програмі "професійний зовнішній вигляд" і полегшує її використання в пакетних файлах.

Імена argc і argv є традиційними, але не обов'язковими. Ці два параметри у функції main () ви можете назвати як завгодно. Крім того, в деяких компіляторах для main () можуть підтримуватися-додаткові аргументи, тому обов'язково вивчіть документацію до вашого компілятору.

Коли для програми не потрібні параметри командного рядка, то найчастіше явно декларують функцію main () як що не має параметрів. В такому випадку в списку параметрів цієї функції використовують ключове слово void.

Повернення з функції

Функція може завершувати виконання і здійснювати повернення в зухвалу програму двома способами. Перший спосіб використовується тоді, коли після виконання останнього оператора в функції зустрічається закриває фігурна дужка (}). (Звичайно, це просто жаргон, адже в сьогоденні об'єктному коді фігурної дужки немає!) Наприклад, функція pr_reverse () у

наведеній нижче програмі просто виводить на екран у зворотному порядку рядок Мені подобається С, а потім повертає керування викликає програмі.

```
#include <string.h>
#include <stdio.h>
void pr_reverse (char*);
int main (void)
{
  char str[]="I like C++";
  pr_reverse (str);
  return 0; }
void pr_reverse (char *s)
{
  register int t;
  for (t = strlen(s) -1; t>= 0; t--) putchar (s[t]); }
```

Як тільки рядок виведена на екран, функції pr_reverse () "робити більше нічого", тому вона повертає керування туди, звідки вона була викликана.

Але на практиці не так вже й багато функцій використовують саме такий спосіб завершення виконання. У більшості функцій для завершення виконання використовується оператор return - або тому, що необхідно повернути значення, або щоб зробити код функції простіше й ефективніше.

У функції може бути декілька операторів return. Наприклад, в наступній програмі функція find_substr () повертає початкову позицію підрядка в рядку або ж повертає -1, якщо підрядок, навпаки, не знайдена. У цій функції для спрощення кодування використовуються два оператора return.

```
#include <stdio.h>
int find_substr (char * s1, char * s2);
int main (void)
{
  if (find_substr ("C++ is cool", "is") != -1)
```

```
    printf ("Substring find");
  return 0; }
/* Повернути позицію першого, входження s2 в s1.  */
 int find_substr (char * s1, char * s2)
 {
  register int t;
  char * p, * p2;
  for (t = 0; s1 [t]; t ++) {
    p = & s1 [t];
    p2 = s2;
    while (* p2 && * p2 == * p) {
      p ++;
      p2 ++;     }
    if (! * p2) return t;  /* 1-й оператор return */
  }
   return -1;  /* 2-й оператор return */
 }
```

Повернення значень

Всі функції, крім тих, які відносяться до типу void, повертають значення. Це значення вказується виразом в операторі return. Стандарт С89 допускає виконання оператора return без вказівки вирази всередині функції, тип якої відмінний від void. В цьому випадку все одно відбувається повернення якогось довільного значення. Але такий стан справ, м'яко кажучи, нікуди не годиться! Тому в Стандарті С99 (та й в C ++) передбачено, що у функції, тип якої відмінний від void, в операторі return необхідно обов'язково вказати повертається значення. Тобто, згідно C99, якщо для будь-якої функції зазначено, що вона повертає значення, то всередині цієї функції у будь-якого оператора return має бути своє вираження. Однак якщо функція, тип якої відмінний від void, виконується до самого кінця (тобто до

закриває її фігурної дужки), то повертається довільне (непередбачуване з точки зору розробника програми!) Значення. Хоча тут немає синтаксичної помилки, це є серйозним упущенням і таких ситуацій необхідно уникати.

Якщо функція не оголошена як має тип void, вона може використовуватися як операнд у виразі. Тому кожне з наступних виразів є правильним:

x = power (y);

if (max (x, y)> 100) printf ("більше");

for (ch = getchar (); isdigit (ch);) ...;

Загальноприйняте правило говорить, що виклик функції не може перебувати в лівій частині оператора присвоювання. Вираз:

*swap (x, y) = 100; / * Неправильне вираз * /*

є неправильним. Якщо компілятор C в якій-небудь програмі знайде такий вираз, то позначить його як помилкове і програму компілювати не буде.

У програмі можна використовувати функції трьох видів. Перший вид - прості обчислення. Ці функції призначені для виконання операцій над своїми аргументами і повертають отримане в результаті цих операцій значення. Обчислювальна функція є функцією "в чистому вигляді". Як приклади можна назвати стандартні бібліотечні функції sqrt () і sin (), які обчислюють квадратний корінь і синус свого аргументу відповідно.

Другий вид включає в себе функції, які обробляють інформацію і повертають значення, яке показує, чи успішно було виконано ця обробка. Прикладом є бібліотечна функція fclose (), яка закриває файл. Якщо операція закриття була завершена успішно, функція повертає 0, а в разі помилки вона повертає EOF.

У функцій останнього, третього виду немає явно повертаються значень. По суті, такі функції є чисто процедурними і ніяких значень видавати не повинні. Прикладом є exit (), яка припиняє виконання програми. Всі функції, які не повертають значення, повинні оголошуватися як повертають значення

типу void. Оголошуючи функцію як повертає значення типу void, ви забороняєте її застосування у виразах, запобігаючи таким чином випадкове використання цієї функції не за призначенням.

Іноді функції, які, здавалося б, фактично не видають змістовний результат, все ж повертають якесь значення. Наприклад, printf () повертає кількість виведених символів. Повторимо, що присвоювання аж ніяк не є обов'язковим, причому відсутність його не стане причиною будь-яких неприємностей. Якщо повертається значення не входить ні в один з операторів присвоювання, то це значення буде просто відкинуто. Відзначимо також, що таке відкидання значення зустрічається дуже часто. Проаналізуйте наступну програму, в якій використовується функція mul ():

```
int mul (int a, int b);
int main (void) {
int x = 10; y = 20, z;
   z = mul (x, y);  /* 1 */
   printf ("% d", mul (x, y));  /* 2 */
   mul (x, y);  / * 3 * /
   return 0; }
 int mul (int a, int b)
 {   return a * b; }
```

Контрольні запитання:

1) Що таке функція?
2) Основні властивості функцій?
3) Види функцій та особивості їх використання.
4) Особливості функції main()/
5) Передача параметрів в функції.
6) Повернення значень з функцій.

РОЗДІЛ ІІІ Об’єктно-орієнтований підхід в С++

Основи об’єктно-орієнтованого підходу в програмуванні

Об’єктно-орієнтоване програмування (ООП) — парадигма програмування, в основу якої покладено принципи інкапсуляції (encapsulation), наслідування (inheritance), поліморфізму (polymorphism) — три ключові принципи ООП —та у якій центральними поняттями є клас (class) та об’єкт (object). Також можна зустріти підходи, які включають у визначення ООП крім трьох згаданих принципів ще три: абстракція (abstraction), пересилка повідомлень (messaging) та модульність (modularity).

Історично із розвитком ідеології процедурного програмування на зміну популярній свого часу концепції структурного програмування прийшла концепція ООП. Початком мов ООП стали мови Симула (з’явилася у 1967 році) та згодом Smalltalk (перша широко розповсюджена мова на основі ООП). Популярність ООП призвела до тотальної переорієнтації багатьох старих мов програмування на підтримку ООП (наприклад, мови С/С++ чи Pascal) та появі мов, заснованих виключно на ООП (наприклад, Java чи C#).

У основі об’єктно-орієнтованого підходу лежить ідея про взаємодію між собою об’єктів, що існують у деякому середовищі. Спрощено, методологія ООП реалізує моделі «реального світу»: у програмі породжуються та існують впродовж деякого часу об’єкти (екземпляри класів), що пов’язані між собою деякими взаємодіями, внаслідок здійснення яких повинні вирішуватися завдання, поставлені перед програмою. З точки зору програмування, об’єкти є представниками деяких класів, що є типовими поєднанням даних та функцій, що застосовуються для маніпулювання цими даними (принцип інкапсуляції). Крім того, об’єкти можуть адаптуватися до умов функціонування програми згідно з тими правилами, які визначені для них класами, до яких ці об’єкти належать (принцип поліморфізму). У свою чергу класи можуть бути пов’язані ієрархічними «родинними» зв’язками, що

дозволяють класам-нащадкам успадковувати та розвивати необхідним чином властивості класів-предків (принцип наслідування). При цьому для класів не є необхідною умовою обов'язковість виконання якихось конкретних дій чи маніпуляцій — вони можуть уособлювати якісь виокремлені уявлення програміста про природу чи суть об'єктів, які у підсумку покликані вирішити поставлені завдання (принцип абстракції). Водночас об'єкти можуть і повинні бути автономними одиницями у програмному коді (принцип модульності) і разом з тим об'єкти можуть обмінюватися інформацією, необхідною для виконання покладених на них обов'язків (принцип пересилки повідомлень).

Складність програмного забезпечення

Існують складні і не дуже складні (прості) програмні системи. Наприклад, є програми, і навіть програмні системи, які проектуються і розробляються, супроводжуються і використовуються однією людиною. Основною особливістю подібних систем є те, що вони зазвичай мають досить обмежену область застосування і короткий час життя, а тому їх відносять до класу простих систем. Для більшості ж користувачів і програмістів інтерес представляють промислові програмні продукти, відмінною рисою яких є відносно високий рівень їх складності, коли один розробник практично не в змозі охопити всі аспекти такої програмної системи.

Складність властива всім великим програмним системам в тому сенсі, що з нею можна впоратися, але позбавитися від неї не можна. Виділяють наступні чотири основні причини, з яких, програмне забезпечення (ПЗ) має таку властивість як складність:

1) складність реальної предметної області, для якої розробляється ПО;

2) труднощі управління процесом розробки;

3) необхідність забезпечити достатню гнучкість програми;

4) проблема опису поведінки великих систем.

Говорячи про складність реальної предметної області, слід зазначити, що проблеми, які намагаються вирішити за допомогою ПЗ, часто містять складні підзадачі, а до відповідним програмним комплексам пред'являється велика кількість різних, нерідко взаємовиключних вимог. Крім того, практично завжди виникають "тонкі" місця, пов'язані із взаємодією між користувачами системи та її розробниками, оскільки користувачі насилу можуть пояснити в повному обсязі та у зрозумілій розробникам формі, що ж у кінцевому підсумку необхідно зробити. Додаткові складності виникають і в результаті змін вимог до програмної системі вже в процесі розробки і впровадження.

Труднощі керування процесом розробки пов'язана з тим, що реалізація програмної системи припускає залучення колективу розробників, а тому неминуче виникають проблеми, пов'язані з організацією колективної розробки. При колективному виконанні проекту головним завданням керівництва є підтримка єдності і цілісності розробки.

Гнучкість програмного забезпечення проявляється в тому, що розробник сам може створювати різні базові конструктивні елементи, на основі яких надалі буде проводити реалізацію програмної системи. З цієї причини програмні розробки залишаються трудомістким заняттям.

Проблема опису поведінки великих дискретних систем пов'язана з тим, що на сьогоднішній день немає ні математичного апарату, ні інтелектуальних можливостей для повного моделювання поведінки великих дискретних систем, а тому доводиться задовольнятися розумним рівнем впевненості в їх правильності. Відзначимо, що програмні системи (ПС) відносяться до типу дискретних, так як всередині них існують сотні змінних і декілька потоків управління. Повний набір цих змінних, їх поточних значень, поточних адрес, значення характеристик потоків управління описує стан прикладної ПС в кожен момент часу. Оскільки виконання програм здійснюється на комп'ютері, то ми маємо систему з дискретними станами.

Ознаки складної системи

Навколо нас існує велике число складних систем. Прикладом є персональні комп'ютери (ПК), які в своїй більшості складаються з одних і тих же складових елементів: системної плати, монітора, клавіатури, вінчестера, миші і т.д. Можна взяти будь-який з цих елементів і також розкласти його на складові. Наприклад, системна плата містить оперативну пам'ять, центральний процесор, шину для підключення периферійних пристроїв. У свою чергу кожен з цих елементів можна розкласти на складові: центральний процесор складається з арифметико-логічного пристрою (АЛП) і пристрій керування (ПК).

В даному випадку ми маємо приклад складної ієрархічної системи. ПК нормально працює завдяки чіткому спільному функціонуванню всіх його елементів, які разом утворюють єдине логічне ціле. Користувач може зрозуміти, яким чином працює комп'ютер, тільки тому, що може розглянути окремо кожну його складову (наприклад, можна незалежно один від одного вивчити пристрій монітора і клавіатури).

Тут важливо не тільки те, що складна система ПК ієрархічна, але і те, що рівні цієї ієрархії представляють різні рівні абстракції, причому один надбудований над іншим і кожен може бути розглянутий окремо.

Виходячи з представленого прикладу і способу вивчення складної системи, можна вивести п'ять загальних ознак будь-якої складної системи:

1. Складні системи часто є ієрархічними і складаються з взаємозалежних підсистем, які в свою чергу також можуть бути розділені на підсистеми, і т.д., аж до найнижчого рівня. (Важливо зрозуміти, що архітектура складних систем складається і з компонентів, і з ієрархічних відносин цих компонентів);

2. Вибір, які компоненти в даній системі вважаються елементарними, відносно довільний і більшою мірою залишається на розсуд дослідника. (Низький рівень для одного спостерігача може виявитися відносно високим для іншого);

3. Внутрішньокомпонентний зв'язок звичайно сильніше, ніж між компонентами. Дана обставина дозволяє відокремлювати "високочастотні" взаємодії усередині компонентів від "низькочастотної" динаміки взаємодії між компонентами. (Це розходження внутрішньокомпонентних і міжкомпонентних взаємодій обумовлює поділ функцій між частинами системи і дає можливість відносно ізольовано вивчати кожну частину)

4. Ієрархічні системи зазвичай складаються з небагатьох типів підсистем, по-різному скомбінованих і організованих. (Іншими словами, різні складні системи містять однакові структурні компоненти, які в свою чергу можуть використовувати загальні більш дрібні підкомпоненти);

5. Будь-яка працююча складна система є результатом розвитку більш простої системи. Складна система, спроектована з нуля, ніколи не заробить, а тому слід починати з працюючої відносно простий системи. (У процесі розвитку системи об'єкти, спочатку розглядаються як складні, стають елементарними, і з них будуються більш складні системи. Більш того, неможливо відразу правильно створити елементарні об'єкти: з ними треба спочатку повозитися, щоб більше дізнатися про реальний поведінці системи, і потім вже удосконалювати їх).

Декомпозіція при проектуванні складаний систем

Можливості розробників складних систем обмежуються можливостями статистично середньої людини. Зрозуміло, що одній людині неможливо в тонкощах мати уявлення про складну систему, і тому людство вже давно користується способом управління складними системами, який звучить так: «Розділяй і володарюй!».

Стосовно до питання проектування складної програмної системи це означає необхідність розділяти її на все менші й менші підсистеми, кожну з яких можна надалі незалежно доводити до потрібного рівня досконалості. У цьому випадку розробникам і користувачам доводиться одночасно тримати в

розумі інформацію лише про небагатьох частинах системи. Такий процес називається декомпозицією.

Стосовно до розробки ПЗ декомпозиція може здійснюватися по-різному. Використовуючи технологію структурного програмування (або проектування), декомпозицію сприймають як звичайне поділ алгоритмів, коли кожен модуль системи виконує один з етапів загального процесу. У даному випадку говорять про алгоритмічної декомпозиція. Альтернативний спосіб декомпозиції заснований на використанні об'єктів і називається об'єктно-орієнтованої декомпозицією. Складна система представляється сукупністю автономних діючих об'єктів, які взаємодіють один з одним, щоб забезпечити поведінку системи, що відповідає більш високому абстрактному рівню.

Досвід використання цих двох способів при проектуванні складних програмних систем показав, що об'єктно-орієнтована декомпозиція має ряд важливих переваг перед алгоритмічної. По-перше, об'єктна декомпозиція дозволяє зменшити розмір програмних систем за рахунок повторного використання загальних механізмів, що приводить до значної економії виразних засобів. По-друге, об'єктно-орієнтовані системи більш гнучкі й простіше еволюціонують з часом, оскільки їх схеми базуються на стійких проміжних формах. По-третє, використання об'єктів дозволяє застосовувати технологію подолання складності на основі абстрагування. Будучи не в змозі повністю відтворити складну систему, вдається ігнорувати не надто важливі деталі і мати справу з узагальненою моделлю системи, яка носить прозорий характер. Використання абстракції дозволяє маніпулювати одиницями інформації істотно більшого об'єму. Це особливо підтверджується, коли розгляд навколишнього світу відбувається через призму об'єктно-орієнтованого підходу, оскільки об'єкти як абстракції реального світу являють собою окремі насичені зв'язні інформаційні одиниці.

Слід звернути увагу і на те, що при проектуванні складних систем число абстракцій може бути більшим, а тому значні спрощення в розумінні

складних систем досягається за рахунок утворення з абстракцій ієрархічної структури. В даному випадку, ієрархія - упорядкування абстракцій, розташування їх за рівнями. Основними видами ієрархічних структур стосовно до складних програмним системам є структура класів і структура об'єктів.

Об’єктно-орієнтована парадигма (інкапсуляція, поліморфізм, наслідування).

Для розуміння переваг об'єктно-орієнтованого програмування необхідно розібратися з трьома ключовими поняттями ООП:

• інкапсуляція (encapsulation)

• наслідування (inheritance)

• поліморфізм (polymorphism)

Інкапсуляція - це механізм, який об'єднує дані і методи, що маніпулюють цими даними, і передбачає захист методів і даних від зовнішнього втручання або неправильного використання.

Об'єднання методів і даних таким способом дозволило реалізувати принципово новий тип даних - клас. Породжувані на основі таких класів змінні отримали окреме найменування - об'єкти.

Якщо задатися питанням, для чого ж потрібна інкапсуляція, то можна запропонувати наступний відповідь: програмісту властиво помилятися, а застосування інкапсуляції забезпечує захист даних і методів, від можливих помилок, які можуть виникнути при прямому доступі до них.

Іншими словами суть інкапсуляції така: змінні стану об'єкта приховані від зовнішнього світу. Зміна станів об'єкта (значень його даних) можливо тільки за допомогою його методів (функцій, процедур). Це запобігає можливість введення об'єкта в неприпустиме стан та / або несанкціоноване руйнування цього об'єкту.

Наслідування - це механізм, за допомогою якого, один клас може успадковувати властивості іншого (інших) класу (класів) і додавати до них риси, характерні тільки для нього.

Суть поняття можна продемонструвати на наступному прикладі успадкування з реального життя: натуралісти левову частку часу витрачають на класифікацію об'єктів у відповідності з певними особливостями. Так, наприклад, в біології існують спеціальні принципи класифікації та розбиття на класи, підкласи (види і підвиди). В результаті утворюється ієрархія з однією загальною категорією в корені і розгалужуються підкатегоріями.

Так проводячи класифікацію нових об'єктів (тварин і т.д.) необхідно виявляти, в чому проявляється подібність і відмінність з іншими об'єктами загального класу. Кожен клас при цьому визначається набором відповідних характеристик. Починаючи з кореневого класу, у результаті формується ієрархія класів. Більш високі рівні містять більш узагальнені характеристики, а кожен наступний рівень є більш специфічним, і менш загальним. Коли деяка характеристика визначена в поточному класі, то всі класи нижче в ієрархії автоматично включають цю характеристику. Тому, коли мова заходить про представника того чи іншого конкретного класу, то в першу чергу увагу приділяють специфічним особливостям в рамках цього класу.

Сенс і універсальність спадкування полягає в тому, що не треба щоразу заново (з нуля) описувати новий клас. Можна легко створити новий клас, вказавши батьківський (базовий клас) і описавши відмітні особливості створюваного класу. В результаті, цей клас буде володіти всіма властивостями батьківського класу плюс своїми власними відмінними компонентами.

Відзначимо, що в деяких об'єктно-орієнтованих мовах програмування визначені механізми спадкування, що дозволяють успадковувати безпосередньо не тільки з одного класу. Крім того, слід мати на увазі, що реалізації навіть простої спадкування можуть розрізнятися залежно від конкретної мови.

Говорячи про термінологію, слід звернути увагу на те, що в описах мов ООП прийнято клас, з якого успадковують називати батьківським класом (parent class) або базовим (base class). Клас, який виходить в результаті наслідування, називається породженим класом (derived or child class). Батьківський клас є більш узагальненим, а породжений клас - більш суворий і конкретний, що робить його більш зручним у застосуванні.

Об'єктно-орієнтоване програмування можна розглядати як процес побудови ієрархії класів. Одним з найбільш важливих властивостей ООП є механізм успадкування, по якому породжені класи можуть успадковувати дані і методи з більш узагальнених класів. Спадкування забезпечує спільність функцій, в той же час, допускаючи стільки особливостей, скільки необхідно.

Поліморфізм (у мовах програмування) - взаємозамінність об'єктів з однаковим інтерфейсом. Мова програмування підтримує поліморфізм, якщо класи з однаковою специфікацією можуть мати різну реалізацію - наприклад, реалізація класу може бути змінена в процесі успадкування.

У загальному сенсі, концепцією поліморфізму є ідея "один інтерфейс, безліч методів". Таким чином, можна створювати загальний інтерфейс для групи близьких за змістом дій.

Перевага поліморфізму проявляється в тому, що він допомагає знижувати складність програм, вирішуючи використання загального інтерфейсу для єдиного класу дій.

Контрольні запитання:

1) Що таке параддигма програмування?
2) Особливості об’єктно-орієнтованого підходу.
3) Ознаки складних програмних систем.
4) Що таке інкапсуляція?
5) Особливості використання наслідування при розробці програм.
6) Поліморізм та приклади його застосування.

Клас і їх опис в С++

Отже, для подальшого коректного розуміння понять ООП формалізуємо визначення понять.

Клас (class) — це тип (структура даних), що описує внутрішнє влаштування об’єктів (реальних об’єктів) та способи їх взаємодії із оточуючим середовищем (програмним середовищем). Клас, по суті, є програмною моделлю об’єктів реального світу.

Об’єкт (object), або екземпляр класу, або інстанція (instance) — сутність, що існує і функціонує у адресному просторі обчислювальної системи та є фактичною реалізацією класу із конкретними значеннями параметрів та характеристик, передбачених даним класом.

Інкапсуляція (encapsulation) — властивість системи, що дозволяє поєднати дані і методи роботи з ними, приховавши деталі реалізації від користувача. При цьому методи та дані класу, що залишаються доступними для безпосереднього використання користувачем, називають інтерфейсними або, у сукупності, інтерфейсом класу. Дані, що описують клас, називають, як правило, його атрибутами чи полями, а функції, що описують методи обробки цих даних, називають операціями або методами.

Наслідування (inheritance) — властивість системи, що дозволяє описати новий клас на основі вже існуючого класу, використовуючи часткове чи повне запозичення його функціональності. Клас, від якого здійснювалося наслідування, називають батьківським, базовим або предком. Новий клас при цьому називають нащадком, спадкоємцем або породженим класом. Множину класів, пов’язаних відношеннями наслідування, називають ієрархією класів. Властивість успадкування одночасно від декількох різних класів називають множинним наслідуванням.

Поліморфізм (polymorphism) — властивість системи використовувати об’єкти з однаковим інтерфейсом без інформації про тип та внутрішню структуру об’єкта. Проявами поліморфізму є, зокрема, застосування

перевантажених методів класу, використання класів-нащадків на місці їх класів-предків (в тому числі застосування поліморфних функцій) і т.п.

Абстракція (abstraction) — це властивість системи або процес, що дозволяє представити дані та програми подібними їх розумінню (чи призначенню), приховуючи або відокремлюючи деталі реалізації. Це доволі складне поняття, суть якого полягає у наданні можливості якнайпростіше поглянути на об’єкт. Під абстракцією часто розуміють сукупність лише значимих характеристик об’єкту, відкидаючи усі властивості об’єкту, що є несуттєвими для його розгляду з деякої точки зору.

Пересилка повідомлень або обмін повідомленнями (messaging) — принцип взаємодії між об’єктами шляхом обміну інформацією з допомогою пересилання повідомлень, що можуть бути як простими, так і складними структурами даних або, навіть, спеціалізованими об’єктами. Справжній обмін повідомленнями як спеціалізованими об’єктами є дуже узагальненим механізмом, що вимагає додаткових обчислювальних зусиль, які не завжди виправдані, тому у більшості сучасних мов його замінено на звичайне використання інтерфейсних методів та полів класу. Зокрема, механізм віртуальних функцій є розвитком цієї концепції для забезпечення механізму обміну повідомленнями при використанні поліморфних змінних.

Модульність (modularity) — принцип, згідно із яким програмний засіб розбивається на окремі поіменовані сутності, модулі, що являють собою якусь логічно відокремлену програмну одиницю або набір програмних одиниць (структури даних, бібліотеки функцій, класи, сервіси, тощо), які реалізують деяку функціональність та надають інтерфейс до неї. Модульність як модульне програмування є концепцією без чіткого визначення, але із надзвичайно широким спектром застосування, характерним не лише для парадигми ООП, але й для більшості інших парадигм програмування.

Звичайно, у парадигму ООП логічно інтегровано і поняття парадигм, що передували її появі. Подані тут основні поняття ООП визначають природу і суть цієї парадигми програмування, що вирізняє її серед інших.

Екземпляри класів або об’єкти

Клас, як основа ООП, володіючи строгою стуктурою, може бути:

- попердньо-оголошеними;

- похідним або базовим. Похідний клас (derived class) є клас, який породжений від базового (батьківського, base class) з унаслідуванням властивостей останнього. Похідний клас детально розглядається в теорії наслідування.

- протокласом. Протоклас (protoclass) є одним із різновидів батьківських класів. Це важливий вид класу у теорії проектування програмних систем. Розглядатись при вивченні поняття наслідування в ієрархіях класів;

- абстрактним. Абстрактним (abstract class) вважається клас, який містить принаймі одну порожню віртуальну функції.

- віртуальним. Віртуальні базові класи (virtual base class) є однію з основ побудови складних ієрархій класів.

- поліморфічним. Поліморфічним класом (polymorphic class) є клас, який містить принаймі одну віртуальну функцію.

- класом-шаблони. Клас-шаблон (template class) є формалізованим записом, за яким можна оголошувати множини класів. Шаблони є основою побудови бібліотек класів і розлядаються в окремій лекції;

- порожнім класом. Клас може не містити жодного члена. Такий клас називається порожнім (empty class). C++ допускає оголошення порожнього класу і створення об'єктів таких класів, наприклад

class A{} ;

A a;

- дружнім класом (friend class);

- вкладеним класом. Клас оголошений всередині іншого класу називається вкладеним (member class). Механізм вкладення не надає додаткових привілеїв доступу жодному з класів.

Синтаксис і семантика ООП на мові C++

Мова C++ не є за своїм походженням об'єктно-орієнтованою (як, наприклад, Java чи C#), тобто підтримка ООП є у них набутою із плином часу. Насправді, C++ є яскравим прикладом мультипарадигменної мови програмування.

Опис класу

Мовна конструкція, що описує користувацький тип даних, який іменують класом виглядає так:

class НазваКласу

{

***private**:*

/ описи закритих полів та методів — доступні лише у класі */*

***protected**:*

/ описи захищених полів та методів — доступні у класі і його нащадках */*

***public**:*

/ описи відкритих (інтерфейсних) полів та методів — доступні назовні */*

};

Важливо звернути увагу на те, що опис класу обов'язково повинен завершитися символом «крапка з комою» (після закриваючої фігурної дужки). Крім того, специфікатор доступу private може бути пропущений (звичайно ж, разом із двокрапкою, що за ним слідує), і описані таким чином поля та методи (тобто такі, що не мають явно визначеного специфікатора доступу) автоматично вважаються закритими. Також слід використовувати лише ті специфікатори доступу (private, protected або public), які доцільні у конкретному класі (це означає, зокрема, що можна пропустити непотрібні специфікатори при описі класу). Порядок слідування специфікаторів доступу із відповідними описами можна обирати довільний.

Нехай нам потрібно описати сутність, що моделює мережі. Приклад опису класу «Мережа»:

class Net

{

float Comp; / зберігає відсоток активних комп'ютерів в мережі */*

string NameOfNet; / зберігає назву організації в якій прокладена мережа */*

public:

void TestComp (void); / метод для перевірки чи комп'ютер в мережі включений */*

string CompName(float); / метод для встановлення рівня приорітету комп'ютера в мережі */*

Net(); / конструктор за замовчуванням */*

Net(float); / конструктор з параметром */*

~ Net(); / деструктор */*

};

Наведений щойно опис, насправді, ще не є завершеним описом класу — це лише декларація. Завершеним опис буде тоді, коли ми опишемо усі задекларовані методи, включно із конструкторами та деструкторами. Коментарі, записані при декларації класу, повинні полегшити у подальшому пр оцес розробки, нагадуючи про призначення чи особливості застосування полів та методів класу.

Ім'я класу не є обов'язковим, проте його краще завжди вказувати, так як ім'я класу стає новим типом даних, воно необхідно для подальшого об'явлення об'єктів цього класу.

Функції та змінні, об'явлені всередині класу, стають членами цього класу. Змінні називаються даними-членами класу, а функції функціями-членами, або методами класу. По замовченню, всі функції і змінні, об'явлені в класі являються закритими. Це означає що вони доступні тільки для інших членів цього класу. Для об'явлення відкритих членів класу використовується

ключове слово public, за яким слідує двокрапка. Всі функції і змінні об'явлені після слова public, доступні і для інших членів класа і для будь якої частини програми, в якій міститься клас.

У наведеному вище прикладі зазначені тільки прототипи функцій, для визначення цих функцій необхідно зв'язати ім'я класу, частиною якого являється дана функція, з іменем функції. Це досягається шляхом вказання імені функції після імені класу з двома двокрапками. Нагадаємо, що операція у вигляді двох двокрапок називається розширенням області видимості. Наприклад:

```
void Net:: CompName(float a)
{
return a;
}
```

Об'явлення класу не приводить до створення його об'єктів. Для того, щоб створити об'єкт відповідного класу необхідно просто використати клас як тип даних:

```
Net obj;
```

Після того, як створено об'єкт класу, можна звертатися до його відкритих членів за допомогою операції «крапка», так само як до полів структури.

```
Net. CompName (10);
```

У прикладі розглянуто присвоєння змінній а значення 10. Кожен об'єкт має свою копію всіх даних, об'явлених в класі.

Мова C++ накладає певні заборони на дані-члени класу:

Дані-члени класу не можуть об'являтися з модифікаторами auto, extern або register.

Даним-членом класа не може бути об'єкт цього ж класу (проте даним-членом класу може бути вказівник або посилання на об'єкт цього класу або сам об'єкт іншого класу)

Імена всіх елементів класу мають своєю областю дії цей клас. Це означає що будь яка функція-член може звертатися до будь якого елементу класу просто за його ім'ям.

Функції та процедури класу називаються методами і містять початковий код, який використовується для обробки внутрішніх даних об'єкта даного класу.

Специфікатори доступу (private, protected, public).

У відповідності до синтаксису мови С++ кожний компонент класу має статус доступу. Таких статуси три: загальнодоступний (public), власний (private) та захищений (protected). За специфікаторами доступу (public, private, protected) йде двокрапка. Дія специфікаторів на компоненти класу починається з моменту написання до нового специфікатора або до кінця описання класу.

private – мітка для закритих членів класу які доступні тільки функціям-членам даного класу або членам класів-"друзів". Це специфікація за замовчанням.

protected – для свого класу ця мітка рівнозначна private, але члени із цією міткою доступні для членів даного класу й класів-друзів, а також для членів похідних класів.

public – члени із цією міткою є відкритими, тобто доступні при звертанні до об'єкта з будь-якого місця програми, де даний клас є видимим.

Специфікатор доступу private використовується для завдання статусу доступу до елементів даних класу, що дозволяє вирішити проблему захисту даних. Власні дані становляться доступними тільки для методів свого класу. Специфікатор доступу public часто використовується для завдання загальнодоступного доступу методів класу, які організують зв'язок об'єкта даного класу з зовнішнім світом. Статус захищений (protected) використовується в класах при використанні механізму успадкування класів.

При відсутності успадкування специфікатор protected еквівалентний специфікатору private.

Усі компоненти класу, введені за допомогою ключових слів struct і union є за замовченням загально доступними, а за допомогою ключового слова class – власними, тобто недоступними для зовнішніх викликів. Для заміни статусу доступу компонентів класу, описаних за допомогою ключових слів class та union, необхідно використовувати специфікатори доступу. Класи, описані за допомогою ключового слова union, не можуть використовуватися як базові класи при успадкуванні. У об'єктів, об'явлених на основі подібного класу, для елементів даних виділяється загальне місце в пам'яті. Статус компонентів у таких класів змінити неможна.

Одним з найважливіших механізмів в C++ є механізм успадкування. Успадкування в будь-якій сучасній мові програмування виконує дві ролі: з одного боку, попереджає дублювання кодів, а з іншого – допомагає розвивати роботу в необхідному направленні. При успадкуванні обов'язково є клас-родитель та клас-нащадок. В C++ батьківський-клас прийнято називати базовим, а клас-нащадок – похідним. Відносини між батьківським класом та його нащадками називають ієрархією класів.

Простим (або одиночним) називається успадкування, при якому похідний клас має тільки одного батька. Формально успадкування одного класу от іншого можна завдати наступною конструкцією:

Class ім'я_класу-нащадника: [модифікатор_доступу] ім'я_базового_класу

{тіло_класу}

Клас-нащадок успадковує структуру (всі елементи даних) та поведінку (всі функції-методи) базового класу. Модифікатор_доступу визначає доступність елементів базового класу в класі-нащадку. Квадратні дужки говорять о том, що цей модифікатор може бути відсутнім. Цей модифікатор називається модифікатором успадкування.

Існують чотири варіанти успадкування: клас від класу, клас від структури, структура від структури та структура від класу. В залежності від модифікаторів доступу при об'яві базового класу та при успадкуванні, доступність об'єктів базового класу із класів-нащадків змінюється.

В Таблиці 3.1 приведені усі варіанти доступності елементів базового класу в похідному класі.

Таблиця 3.1 - Доступ до елементів базового класу в класах-нащадках

Модифікатор в базовому класі	Модифікатор успадкування	Доступ в похідному класі	
		struct	class
public	відсутній	public	private
protected	відсутній	public	private
private	відсутній	недоступний	недоступний
public	public	public	public
protected	public	protected	protected
private	public	недоступний	недоступний
public	protected	protected	protected
protected	protected	protected	protected
private	protected	недоступний	недоступний
public	private	private	private
protected	private	private	private
private	private	недоступний	недоступний

Якщо у якості специфікатора доступу записано слово public, то таке успадкування називається відкритим. Відповідно, при використанні модифікатора protected маємо захищене успадкування, а слово private визначає закрите успадкування.

Конструктор і деструктор

При створенні об'єктів однією з найбільш широко використовуваних операцій яку виконують у програмах, є ініціалізація елементів даних об'єкта. Щоб спростити процес ініціалізації елементів даних класу, C++ використовує спеціальну функцію, звану конструктором, яка запускається для кожного створюваного вами об'єкта. Подібним чином C++ забезпечує функцію, звану деструктором, яка запускається при знищенні об'єкта:

Конструктор являє собою метод класу, який полегшує програмам ініціалізацію елементів даних класу. Конструктор має таке ж ім'я, як і клас. Конструктор не має значення, що повертається. Кожного разу, коли ваша програма створює змінну класу, C++ викликає конструктор класу, якщо конструктор існує.

Багато об'єктів можуть розподіляти пам'ять для зберігання інформації; коли ви знищуєте такий об'єкт, C++ буде викликати спеціальний деструктор, який може звільняти цю пам'ять, очищаючи її після об'єкта.

Деструкція має таке ж ім'я, як і клас, за винятком того, що ви повинні випереджати його ім'я символом тильди (~). Деструкція не має значення, що повертається. Приклад конструктора:

```
class employee
{
public:
  employee(char *, long, float); //Конструктор
  void show_employee(void);
  int change_salary(float);
  long get_id(void);
```

```
private:
  char name [64];
  long employee_id;
  float salary;
};
```

Деструктор автоматично запускається кожного разу, коли програма знищує об'єкт.

Кожна зі створених програм створювала об'єкти на самому початку свого виконання, просто оголошуючи їх. При завершенні програма знищує об'єкти. Якщо визначити деструктор всередині програми, то він буде автоматично викликатися для кожного об'єкта, коли програма завершується (тобто коли об'єкти знищуються). Подібно до конструктора, деструктор має таке ж ім'я, як і клас об'єкта. Однак у випадку деструктора ви передували його ім'я символом тильди (~), як показано нижче:

```
~class_name (void) //-----------> деструктор
{
// Оператори деструктора
}
```

Перевантаження конструкторів та деструкторів класу

Одним із найпоширеніших прикладів поліморфізму майже у кожній програмі, що декларує та описує класи, є перевантаження функцій, виражене у перевантаженні методів, операцій (операторів) та конструкторів класу. У наведеному прикладі класу зустрічаємо перевантаження конструктора: є конструктор за замовчуванням (без жодних параметрів) та конструктор з параметром.

```
Net (); /* конструктор за замовчуванням */
Net (float); /* конструктор з параметром */
```

Взагалі конструктори можна розглядати як специфічні методи класу, тому їх перевантаження фактично не відрізняється від перевантаження

звичайних методів класу. В результаті застосування перевантажених методів виникає важливий ефект (власне у ньому тут виникає поліморфізм) — результат звертання до перевантаженого методу залежатиме від того, який саме метод буде викликано. У нашому прикладі, коли перевантажено конструктор, цей конструктор викликається неявно при створенні екземпляру класу. Усе тут залежатиме від параметрів, а у нашому випадку відсутності чи наявності параметра при створенні об’єкту.

Також, можна створити лише один конструктор з параметром, який буде підміняти дію конструктора за замовчуванням таким чином:

Net (float Comp =0,5); /* конструктор з параметром */

Вказівник this

Ім'я this є службовим (ключовим) словом. Явно описати чи визначити вказівник this не можна. Відповідно до неявного визначення this є константним вказівником, тобто змінювати його не можна, однак у кожної приналежної класу функції він указує саме на той об'єкт, для якого функція викликається. Говорять, що вказівник this є додатковим словам, при вході в тіло приналежній класу функції вказівник this ініціалізується значенням адреси того об'єкта, для якого викликана функція. Об'єкт, що адресується вказівником this, стає доступним усередині приналежної класу функції саме за допомогою вказівника this.

Коли функція, що належить класу, викликається для обробки даних конкретного об'єкта, цієї функції автоматично і неявно передається вказівник на той об'єкт, для якого функція викликана. Цей вказівник має фіксоване ім'я this і непомітно для програміста ("таємно") визначено у кожної функції класу наступним чином:

імя_класса * const this = адресу оброблюваного об'єкта;

Ім'я this є службовим (ключовим) словом. Явно описати чи визначити вказівник this не можна і не потрібно. Відповідно до неявним визначенням this є константним покажчиком, тобто змінити його не можна, проте в кожній

належить класу функції він указує саме на той об'єкт, для якого функція викликається.

1. Вказівник this ініціалізується значенням адреси об'єкта, для якого викликаний метод, перед початком виконання коду цього методу.

2.Імя this є службовим (ключовим) словом.

3.Явно описати або визначити вказівник this не можна.

4.У відповідності з неявним визначенням this є константним вказівником, тобто змінювати його не можна, однак у кожної приналежної класу функції він указує саме на той об'єкт, для якого функція викликається.

5.Об'ект, який адресується покажчиком this, стає доступним усередині приналежної класу функції саме за допомогою вказівника this.

6.Всередині функції - члена класу можна явно використовувати цей покажчик.

Покажчик this є дуже корисним, а інколи просто незамінним. Наприклад, в наступному коді покажчик this дозволяє компілятору розібратися в ситуації, коли ім'я компонента класу збігається з ім'ям формального параметра, що належить методу:

```
class Student        // Класс студент.
{       char name[50]; // ім’я
        char surname[50]; // фамілія
        int age; // вік
public: //Конструктор:
        Student(char name[],char surname[],int age)
        {       // Компоненти та однойменні параметри:
                strcpy(this->name,name);
                strcpy(this->surname,surname);
                this->age=age;
        }
        void Show()
        { // тут this необов’язковий, проте його можна використовувати
```

```
            cout << "\nNAME - " << this->name;
            cout << "\nSURNAME - " << this->surname;
            cout << "\nAGE - " << this->age;
            cout << "\n\n";
        }
};
```

Кажуть, що вказівник this є додатковим (схованим) параметром кожної нестатичні компонентної функції. Іншими словами, при вході в тіло належить класу функції вказівник this ініціалізується значенням адреси того об'єкта, для якого викликана функція. Об'єкт, який адресується вказівником this, стає доступним усередині приналежної класу функції саме за допомогою вказівник this. При роботі з компонентами класу усередині приналежної класу функції можна було б скрізь використовувати цей покажчик.

Особливості перезавантаження операторів в класі

Перевантаження операторів задумане з метою спрощення запису та уніфікації виразів. Більше того, перевантажені оператори можуть наслідуватись. Власне поширення концепції перевантаження операторів еквівалентне звертанню до функцій і є новим в C++. Це дозволяє не просто створювати легкий для розуміння програмний код, але й забезпечити інтеграцію створених нами класів з іншими класами. Одразу зазначимо, що перевантаження операторів можливо трьома способами:

1. Звичайний перевантажений оператор.
2. Дружній перевантажений оператор.
3. Перевантажений оператор, визначений як член класу.

Перший варіант передбачає, що оператор описаний десь за межами класу і просто приймає посилання на екземпляр класу як параметр. У цьому випадку оператор не матиме доступу до закритих та прихованих полів і методів класу. А це може бути проблемою, якщо такий доступ насправді буде потрібен для виконання дій, що передбачає оператор.

Другий варіант передбачає, що оператор, задекларований у класі як дружній і, так само як і будь-який дружній елемент (функція, інший клас чи окремий метод іншого класу), отримає доступ до закритих та прихованих полів і методів нашого класу. Але, так0 само як і для поперенього випадку, одним із операндів для виклику оператора повинно бути посилання на наш клас. Тим не менше, це найбільш універсальний з трьох способів перевантажити оператор. У третьому варіанті ми гарантовано матимемо прямий доступ до закритих та прихованих полів класу, але ми будемо обмежені тим, що повинні будемо використовувати оператор для роботи саме з тим екземпляром класу, який викликає його. Наприклад, такими операторами можуть бути присвоєння, інкремент, декремент, оператор виклику функції, оператор індексації тощо. Основна їх особливість тут у тому, що вони здійснюють дії над об’єктом, для яким викликані.

Контрольні запитання:

1) Що таке клас?
2) Що таке об’єкт класу?
3) Проілюструйте узагальнену структуру запису класу на мові С++.
4) Що таке модифікатори доступу? Назвіть їх відмінності.
5) Для чого використовується конструктор класу?
6) Поняття та основні завдання деструктора класу
7) Для чого використовується механізм перезавантаження конструктора?
8) Вказівник this та особливості його застосування.

Перезавантаження операторів

У математиці зміст операції залежить від її операндів. Якщо a і b є цілими числами, їх сума a+b обчислюється по одним правилах, якщо матрицями — по іншим. Людина, що знає природу операндів, не випробує утруднень, інтерпретуючи зміст операції +. Як відомо, кожен убудований тип даних зв'язаний з визначеним набором операцій, які можна до нього застосовувати. Ці операції позначаються відповідними символами, зміст яких відомий компілятору заздалегідь. Однак цілком природно спробувати розширити застосування операторів на класи, створювані програмістом. Наприклад, матричні обчислення стають набагато наочніше, якщо сума двох матриць записується як a+b, а не як summa(a,b). Для цього в мові C++ існує механізм перевантаження операторів.

Спочатку необхідно відзначити, що не всі символи операцій можна перевантажити. Нижче перераховані оператори, дозволені до перевантаження.

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr>
<tr><td>&</td><td>|</td><td>~</td><td>!</td><td>=</td><td><</td></tr>
<tr><td>></td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td></tr>
<tr><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td>>>=</td></tr>
<tr><td><<=</td><td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td></tr>
<tr><td>||</td><td>++</td><td>--</td><td>->*</td><td>,</td><td>-></td></tr>
<tr><td>[]</td><td>()</td><td>new</td><td>new[]</td><td>delete</td><td>delete[]</td></tr>
</table>

Існують також оператори, заборонені до перевантаження. Зміна їх змісту зруйнувало би логіку програми. До таких операторів належать :: (оператор дозволу області видимості), . (“точка” — оператор доступу до члена класу), ?: (тернарний оператор), .* (доступ до розіменованого вказівника-члена класу), sizeof, typeid, static_cast, dynamic_cast, const_cast і reinterpret_cast. Крім того, не рекомендується перевантажувати логічні

оператори && і ||, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів. (Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються.)

Синтаксис операторних функцій виглядає в такий спосіб.

тип_значення_що_повертається operator символ_операції(параметри)

{

...

}

Наприклад, операторна функція, що перевантажує операцію +, називається operator+(). Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями. Необхідно пам'ятати про обмеження, що супроводжують застосування перевантажених операторів.

1. Перевантажені функції не можуть змінити пріоритет операторів.
2. Кількість операндів фіксована: жодного, один чи два.
3. Значення операндів не можна задавати за замовчуванням.

Перевантаження операторів за допомогою функцій-членів

Оператори можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний — два. Нагадаємо, що до унарних операторів, що перевантажуються, належать такі оператори, як +, -, ++, --, &, ~ і !. До бінарних операторів, що перевантажуються, належать всі інші оператори, перераховані в приведеній вище таблиці.

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника this. Отже, така функція-член класу не має явних параметрів.

Розглянемо приклад перевантаження унарного мінуса. (Операція унарного плюса є "порожній". Вона включена в мову для симетрії.) Перевантаження унарних операторів "плюс" і "мінус"

```
#include <cstdio>
class TComplex
{
double Re;
double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void print(){ printf("%lf + i*%lf\n", Re, Im);}
;
TComplex operator-() {Re = -Re; Im = -Im; return *this;}
};
int main()
{
TComplex z(1,1),u(0,0);
z.print();
u=-z;
u.print();
return 0;}
```

Помітимо, що операторна функція може повертати об'єкт класу (точніше, розіменований вказівник this) чи посилання на об'єкт, а може і нічого не повертати. Вибираючи тип значення, що повертається, необхідно керуватися здоровим глуздом. Якщо унарний оператор повинний використовуватися усередині виразів, то операторна функція повинна повертати чи об'єкт посилання. Якщо ж оператор використовується ізольовано, функція може нічого не повертати.

Рядок програми

u = - z;

можна переписати в еквівалентному виді.

u = z.operator-();

Цей рядок демонструє, що виклик операторної функції operator-() здійснюється об'єктом z. Варто мати на увазі, що хоча зміст оператора перевантажувати можна, його природу змінювати заборонено. Це значить, що унарні оператор не можна перевантажити як бінарний і навпаки. Розглянемо найбільш важливі приклади унарних операторів.

Оператори інкремента і декремента

У мові C++ передбачено дві форми операторів інкремента і декремента: префиксна і постфиксна. Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій — вводиться фіктивний цілочисловий параметр. Наприклад, для класу TComplex прототипи відповідних операторних функцій можуть виглядати в такий спосіб.

```
#include <cstdio>
#include <iostream>
class TComplex{
double Re;
double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void print(){printf("%lf + i*%lf\n", Re, Im);};
TComplex& operator++()
{++Re;
++Im;
printf("Префіксна форма ++ \n");
return *this;}
const TComplex operator++(int i)
```

```
{++Re;
++Im;
printf("Постфіксна форма ++ %d\n",i);
return *this;}
TComplex& operator--()
{--Re;
--Im;
printf("Префіксна форма -- \n");
return *this;}
const TComplex operator--(int i)
{--Re;
--Im;
printf("Постфіксна форма -- %d\n",i);
return *this;
}};
int main()
{
setlocale (LC_ALL,"ukr");
TComplex z(1,1);
++z;
z.print();
z++;
z.print();
--z;
z.print();
z--;
z.print();
return 0;
}
```

У функції main() до об'єкта z послідовно застосовуються префіксна і постфіксна форми операторів ++ і --.

Префіксна форма ++

2.000000 + i*2.000000

Постфіксна форма ++ 0

3.000000 + i*3.000000

Префіксна форма --

2.000000 + i*2.000000

Постфіксна форма -- 0

1.000000 + i*1.000000

Якщо символ операції ++ стоїть перед операндом, викликається операторна функція operator++(), якщо після — операторна функція operator++(int i). Змінна і відіграє роль прапора, що повідомляє компілятору, що дана функція перевантажує постфиксну форму оператора інкремента і декремента.

Незважаючи на те що програміст вільний вільно трактувати перевантажений оператор, прагнучи зберегти аналогію з його убудованими аналогами, необхідно дотримувати визначені правила. Наприклад, як відомо, оператор інкрементації цілих чисел повертає посилання на неконстантний об'єкт. Ця властивість повинна зберігатися і при перевантаженні.

```
TComplex& operator++()
{
++Re;
++Im;
printf("Префіксна форма ++ \n");
return *this;}
```

У той же час постфіксний оператор повертає константне значення. Саме тому в С++ неможливі вираження х++++. Цілком природно зажадати, щоб перевантажений оператор мав таку ж властивість.

```
const TComplex operator++(int i)
```

```
{++Re;
++Im;
printf("Постфіксна форма ++ %d\n",i);
return *this;}
```

Крім того, оскільки у вихідному варіанті постфиксний оператор іінкрементації реалізується через префиксний, це також бажано врахувати при перевантаженні. У цьому випадку реалізація операторної функції ++ для постфиксної форми може виглядати так.

```
const TComplex operator++(int i)
{TComplex Z;
++*this;
printf("Постфиксна форма ++ %d\n",i);
return Z;}
```

Унарні оператори !, & і ~

Оператори заперечення (!), узяття адреси (&) і побітового заперечення (~) допускають перевантаження, але не мають універсальних альтернатив, що варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора ! можна позначати операцію звертання матриці, а за допомогою символу ~ — її транспонування. Щоправда, застосування тильди закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанину. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.

Перевантаження оператора **->**

Оператор посилання на член об'єкта є унарним. Операторна функція, що перевантажуєйого, виглядає в такий спосіб.

об'єкт -> елемент

Цей запис еквівалентний наступному вираженню.

об'єкт.operator->(елемент);

Функція operator- >() повинна бути нестатичним членом класу. Як параметр вона одержує об'єкт чи класу посилання на нього, повертаючи вказівник this на об'єкт, де виконується виклик, або посилання на об'єкт будь-якого іншого класу, у якому визначений оператор ->. її зручно використовувати в контейнерних класах, що містять усередині себе вказівник на інший клас. Основний зміст перевантаження оператора -> полягає в додатковій функціональності, що розширює можливості звичайних вказівників.

Наприклад, у приведеній нижче програмі функція operator- >() веде підрахунок посилань на кожен об'єкт класу.

Перевантаження оператора **->**

```
#include <cstdio>
class TClass{
int n;
int counter;
public:
TClass(int x):n(x),counter(0) { }
TClass* operator->();
int get(void) { return n;}
int ref(void) { return counter; }
};
TClass* TClass::operator ->()
{counter++;
return this;}
int main()
{TClass a(1), b(2);
printf("n = %d \n",a->get());
printf("n = %d \n",b->get());
printf("n = %d \n",a->get());
```

```
printf("counter = %d \n",a->ref());
printf("counter = %d \n",b->ref());
return 0;}
```

У результаті роботи програми на екран виводяться наступні рядки.

n = 1

n = 2

n = 1

counter = 3

counter = 2

Зверніть увагу на те, що функція operator- >() у даному прикладі повертає значення типу TClass*.

Перевантаження арифметичних операторів

Продемонструємо, як здійснюються ланцюжки обчислень над комплексними числами. Арифметичні дії над комплексними числами

```
#include <cstdio>
class TComplex
{double Re;
double Im;
public:
TComplex(double x, double y):Re(x),Im(y){ }
TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
~TComplex(){}
void print(){printf("%lf + i*%lf\n", Re, Im);};
TComplex operator+(const TComplex& z) // Додавання
{
TComplex w(0,0);
w.Re = Re+z.Re;
w.Im = Im+z.Im;
printf("Operator + \n");
```

```
return w;}
TComplex operator-(const TComplex& z) // Вирахування
{TComplex w(0,0);
w.Re = Re-z.Re;
w.Im = Im-z.Im;
printf("Operator - \n");
return w;}
TComplex operator*(const TComplex& z) // Множення
{TComplex w(0,0);
w.Re = Re*z.Re-Im*z.Im;
w.Im = Re*z.Im+Im*z.Re;
printf("Operator * \n");
return w;}
TComplex operator-()
{Re = -Re;
Im = -Im;
printf("Unary operator - \n");
return *this;}
TComplex operator~()
{TComplex w(0,0);
w.Re = Re;
w.Im = -Im;
printf("Unary operator ~ \n");
return w;}
TComplex operator/(TComplex& z)
{TComplex w(0,0);
w=(*this)*(~z);
w.Re = w.Re/modul2(z);
w.Im = w.Im/modul2(z);
printf("Operator / \n");
```

```
return w;}
double modul2(const TComplex& z)
{return z.Re*z.Re+z.Im*z.Im;}
};
int main()
{TComplex u(1,1),v(2,2),z(0,0);
z=(u+v)*u/v;
z.print();
return 0;}
```

У класі TComplex операція розподілу двох комплексних чисел реалізована через обчислення сполученого числа (операція ~) і розподіл дійсної і мнимої частин на квадрат модуля дільника (функція modul2()).

От у якому порядку виконувалися зазначені оператори.

Operator +

Operator *

Unary operator ~

Operator *

Operator /

У підсумку, як і випливало очікувати, одержуємо наступне число.

1.500000 + i*1.500000

Реалізуючи ланцюжок обчислень, необхідно обережно використовувати ключове слово const. Безперечно, захист параметра від необережної модифікації необхідний. У той же час повернення константних об'єктів не завжди виправданий.

Перевантаження операторів new і delete

Програміст може керувати виділенням пам'яті, перевантажуючи оператори new і delete. Перевантажена операторна функція має наступний вид.

```
void* operator new(size_t size);
```

Вона виділяє size байт пам'яті і повертає адресу виділеної пам'яті. Конструктор і деструктор об'єктів викликаються автоматично. Тип size_t є цілочисловим. Перевантажений оператор delete звільняє пам'ять, виділену перевантаженим оператором new. Розглянемо програму, у якій оператор new реалізує виділення пам'яті за допомогою функції malloc(). Таке перевантаження може виявитися корисної при сполученні декількох модулів, написаних на мовах C і C++. Як відомо, одночасне використання механізмів розподілу пам'яті, передбачених у мовах C і C++, тобто використання пар new — free() чи malloc() — delete, може привести до непередбачених результатів. Отже, перевантаження оператора new, зазначена вище, може привести різні модулі "до загального знаменника". Розглянемо програму, у якій перевантажені оператори new і delete, що використовують функції malloc() і free(). У класі передбачений статична змінна — індикатор зайнятої пам'яті. Як тільки обсяг виділеної пам'яті перевищує припустимий, програма видає повідомлення на екран.

Перевантаження операторів new і delete

```
#include <iostream >
#define MAX_SIZE 1024
class TClass
{char array[500];
public:
TClass(){cout << "Ctor" << endl;}
~TClass(){cout << "Dtor" << endl;}
static void* operator new(size_t);
static void operator delete(void*);
static long counter;};
void* TClass::operator new(size_t size)
{if( counter >= MAX_SIZE - sizeof(TClass))
{cout << "Пам'ять вичерпана" << endl;
return 0;}
```

```
else
{counter += sizeof(TClass);
cout << "New " << endl;
cout << "Зайняте: " << counter << " байт" << endl;
return malloc(size);}}
void TClass::operator delete(void* p)
{free(p);
counter-=sizeof(TClass);
cout << "Delete " << endl;
cout << "Зайнято: " << counter << " байт" << endl;};
long TClass::counter = 0;
int main()
{TClass* q[3];
for(int i = 1; i<=3; i++) { q[i] = new TClass;}
for(int i = 1; i<=3; i++) { delete q[i];}
return 0;}
```

Ця програма веде облік зайнятої пам'яті, збільшує розмір лічильника, а потім звільняє пам'ять і зменшує лічильник на відповідну величину. Зверніть увагу на те, що перевантажені оператори new і delete повинні бути статичними членами класу.

```
New
Зайнято: 500 байт
Ctor
New
Зайнято: 1000 байт
Ctor
Пам'ять вичерпана
Dtor
Delete
Зайнято: 500 байт
```

Dtor

Delete

Зайнято: 0 байт

Тепер застосування операторів new і delete до об'єктів класу TClass приведе до виклику перевантажених версій, а для убудованих типів використовуються звичайні варіанти цих операторів. Оператор new має особливу форму перевантаження, що називається синтаксисом розміщення. Вона дозволяє створювати об'єкт, розміщаючи його в осередку з заданою адресою. Нагадаємо, що саме в цьому випадку необхідно явно викликати деструктор.

Перевантаження оператора [ ]. Бінарний оператор доступу до члена масиву перевантажується за допомогою наступної операторної функції, що повинна бути нестатичним членом класу.

```
тип_ щоповертається_значення& ім′я_класу::operator[](інтегральний_тип i)
{
// ...
}
```

Відзначимо два моменти. По-перше, оскільки операція доступу застосовується до індексованих масивів, параметр операторної функції повинний бути цілочисловим. По-друге, елементи масиву можуть стояти як у лівої, так і в правій частині оператора присвоювання. Отже, функція operator[]() повинна повертати посилання або вказівник.

Хоча зовні функція operator[]() виглядає унарною, насправді вона є бінарною. Її перший параметр явно задає індекс елемента, а другий параметр, що представляє собою вказівник this на об'єкт,де виконується виклик, передається неявно. От типовий приклад використання цього оператора. Перевантаження оператора [ ].

class TMatrix

{

```
int n;
int m;
double *p;
public:
TMatrix(int x,int y);
TMatrix();
~TMatrix() { if (p!=NULL) delete p; p=NULL; }
double* operator[](int i) { return p+i*m; }
void output();
};
TMatrix::TMatrix(int x,int y):n(x),m(y)
{
p=new double[n*m];
for (int i=0; i<n; i++)
for (int j=0; j<m; j++)
if(i==j)p[i*m+j]=1; else p[i*m+j]=0;
}
void TMatrix::output()
{for (int i=0; i<n; i++)
{for (int j=0; j<m; j++)
{
printf("%6.3lf ", (*this)[i][j]);}
printf("\n");}
printf("\n");}
int main()
{TMatrix c(3,3);
c[1][2]=2.0;
c[2][1]=2.0;
c.output();
return 0;}
```

Як відомо, у мові C++ за замовчуванням не передбачена перевірка виходу індексу масиву за припустимі межі. З цієї причини визначення класу TMatrix варто було б доповнити генеруванням відповідних виняткових ситуацій.

Отож, на основі вище сказаного можна зробити наступні висновки:

Не всі символи операцій можна перевантажити.

До операторів, які не можна перевантажити, належать :: (оператор дозволу області видимості), .("точка" — оператор доступу до члена класу), ?: (тернарний оператор), .* (доступ дорозіменованого вказівника-члена класу), sizeof, typeid, static_cast, dynamic_cast, const_cast і reinterpret_cast. Крім того, не рекомендується перевантажувати логічні оператори && і ||, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів. (Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються.)

Синтаксис операторних функцій виглядає в такий спосіб.

тип_значення_що_повертається operator символ_операції(параметри)

{...}

Наприклад, операторна функція, що перевантажує операцію +, називається operator+().

Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями.

Необхідно пам'ятати про обмеження, що супроводжують застосування перевантажених операторів:

1) перевантажені функції не можуть змінити пріоритет операторів, 2) кількість операндів фіксована: жодного, один чи два, 3) значення операндів не можна задавати за замовчуванням.

Оператори можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний — два. Нагадаємо, що до унарних операторів, що

перевантажуються, належать такі оператори, як +, -, ++, --, &, ~ і !. До перевантажуваного бінарного належать всі інші оператори, перераховані в приведеній вище таблиці.

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника this. Отже, така функція-член класу не має явних параметрів.

У мові C++ передбачено дві форми операторів інкремента і декремента: префіксна і постфіксна. Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій — вводиться фіктивний цілочисловий параметр. Якщо символ операції ++ стоїть перед операндом, викликається операторна функція operator++(), якщо після — операторна функція operator++(int i).

Змінна і відіграє роль прапора, що повідомляє компілятору, що дана функція перевантажує постфиксну форму оператора інкремента і декремента.

Незважаючи на те що програміст вільний вільно трактувати перевантажений оператор, прагнучи зберегти аналогію з його убудованими аналогами, необхідно дотримувати визначені правила.

Наприклад, як відомо, оператор інкрементації цілих чисел повертає посилання на неконстантний об'єкт. Ця властивість повинна зберігатися і при перевантаженні. Постфіксний оператор повертає константне значення. Саме тому в C++ неможливі вираження x++++. Цілком природно зажадати, щоб перевантажений оператор мав таку ж властивість.

Крім того, оскільки у вихідному варіанті постфіксний оператор інкрементації реалізується через префіксній, це також бажано врахувати при перевантаженні. У цьому випадку реалізація операторної функції ++ для постфіксної форми може виглядати так.

Оператори заперечення (!), узяття адреси (&) і побітового заперечення (~) допускають перевантаження, але не мають універсальних альтернатив, що варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора ! можна

позначати операцію звертання матриці, а за допомогою символу ~ — її транспонування.

Оператор посилання на член об'єкта є унарним. Операторна функція, що перевантажуєйого, виглядає в такий спосіб.

об'єкт -> елемент

Цей запис еквівалентний наступному виразенню.

об'єкт.operator->(елемент);

Контрольні запитання:

1) Поясніть принципи функціонування механіки перезавантаження.
2) Які оператори можна перезавантажувати?
3) Назвіть особливості перезавантажування унарних операторів.
4) Назвіть особливості перезавантажування арифметичних операторів.
5) Особливості перезавантаження операторів new та delete.

Віртуальні функції. Абстрактні класи

Мова C++ забезпечує як статичний, так і динамічний поліморфізм. Як указувалося в попередніх розділах, статичний поліморфізм досягається за допомогою перевантаження функцій і операторів. Динамічний поліморфізм реалізується на основі спадкоємства і віртуальних функцій. Саме ця тема знаходиться в центрі уваги даного розділу.

Раннє та пізнє скріплення (link)

Перш ніж торкнутися самого вживання віртуальних функцій необхідно розглянути такі поняття як раннє і пізнє скріплення (link). Порівняємо два підходи до покупки, наприклад, комп’ютера. У першому випадку заздалегідь визначимо точну конфігурацію робочої станції та її підсумкову вартість. Тому для покупки ми візьмемо відому суму грошей та транспортний засіб відповідних габаритів. У другому випадку необхідно придбати комп’ютер з наперед не відомими характеристиками. Тому в другому випадку, нам необхідно взяти якомога більше грошей та максимально великий транспортний засіб. Оскільки вартість, розміри та вага придбаного комп’ютера не відомі, тому необхідно передбачити всі можливі варіанти.

Наведений приклад до певної міри відображає сенс вживання раннього і пізнього скріплення, відповідно. Вочевидь, що для даного прикладу перший варіант оптимальний. У другому випадку ми надто багато всього передбачили, але нам це не знадобилося. З іншого боку, якщо в магазині виникне потреба змінити конфігурації комп’ютера або ціна на деякі комплектуючі змінилась, то в першому випадку придбати обновку не буде можливості. У другому ж випадку – у нас не виникне жодних труднощів.

А тепер розглянемо цей приклад з точки зору програмування. При вживанні раннього скріплення, ми як би говоримо компілятору: "Я точно знаю, чого я хочу. Тому жорстко (статично) зв'язуй всі виклики функцій". При вживанні механізму пізнього скріплення ми як би говоримо

компілятору: "Я доки не знаю чого я хочу. Коли прийде час, я повідомлю, що і як я хочу".

Таким чином, під час раннього скріплення викликаючий метод та метод що викликаються, зв'язуються при першій слушній нагоді, зазвичай при компіляції.

При пізньому скріпленні методу, що викликається, і викликаючого методу вони не можуть бути зв'язані під час компіляції. Тому реалізований спеціальний механізм, який визначає, як відбуватиметься скріплення методів, що викликаються і викликають, коли виклик буде зроблений фактично.

Вочевидь, що швидкість і ефективність при ранньому скріпленні вищі, ніж при використанні пізнього скріплення. В той же час, пізнє скріплення забезпечує деяку універсальність скріплення.

Віртуальні функції

Віртуальна функція (virtual function) — це функція-член, оголошена в базовому класі і перевизначена в похідному. Щоб створити віртуальну функцію, слід вказати ключове слово virtual перед її оголошенням в базовому класі. Похідний клас перевизначає цю функцію, пристосовувавши її для своїх потреб. По суті, віртуальна функція реалізує принцип "один інтерфейс, декілька методів", що лежить в основі поліморфізму. Віртуальна функція в базовому класі визначає вид інтерфейсу, тобто спосіб виклику цієї функції. Кожне перевизначення віртуальної функції в похідному класі реалізує операції, властиві лише даному класу. Інакше кажучи, перевизначення віртуальної функції створює конкретний метод (specific method).

Адреса віртуальної функції відома лише у момент виконання програми. Коли відбувається виклик віртуальної функції, його адреса береться з таблиці віртуальної функції свого класу. Таким чином, викликається те, що потрібне.

Перевага вживання віртуальної функції полягає в тому, що при цьому використовується саме механізм пізнього скріплення, який допускає обробку об'єктів, тип яких невідомий під час компіляції.

При звичайному виклику віртуальні функції нічим не відрізняються від решти функцій-членів. Особливі властивості віртуальних функцій виявляються при їх виклику за допомогою покажчиків. Покажчики на об'єкти базового класу можна використовувати для посилання на об'єкти похідних класів. Якщо покажчик на об'єкт базового класу встановлюється на об'єкт похідного класу, що містить віртуальну функцію, вибір необхідної функції ґрунтується на типі об'єкту, на який посилається покажчик, причому цей вибір здійснюється в ході виконання програми. Таким чином, якщо покажчик посилається на об'єкт різних типів, то будуть викликані різні віртуальні функції. Це відносите і до посилань на об'єкти базового класу.

Розглянемо спершу наступний приклад.

```
#include <iostream>
using namespace std;
class base {
        public:
        virtual void vfunc() { cout << "Функція vfunc() з класу base.\n";}};
class derived1 : public base {
        public:
        void vfunc() { cout « "Функція vfunc() з класу derived1.\n";}};
class derived2 : public base {
        public:
        void vfunc() { cout « " Функція vfunc() з класу derived2.\n" ; }};
int main() {
        base *p, b;
        derived1 d1;
        derived2 d2 ;
        P = &b;
        p->vfunc(); // Виклик функції vfunc() з класу base.
        p = &d1;
        p->vfunc(); // Виклик функції vfunc(j з класу derived1.
```

```
p = &d2;
p->vfunc(); // Виклик функції vfunc() з класу derived2.
return 0;}
```

Ця програма виводить на екран наступні рядки.

Функція vfunc() з класу base.

Функція vfunc() з класу derived1.

Функція vfunc () з класу derived.2 .

Як показує ця програма, усередині класу base оголошена віртуальна функція vfunc (). Зверніть увагу на ключове слово virtual в оголошенні функції. При перевизначені функції vfunc () у класах derived1 і derived2 ключове слово virtual не потрібне. (Проте його використання не є помилкою, просто воно не обов'язкове.)

У даній програмі класи derived1 і derived2 є похідними від класу base. Усередині кожного з цих класів функція vfunc () перевизначається наново відповідно до нового призначення. У програмі main() оголошені чотири змінні.

p - Вказівник на базовий клас.

b - Об'єкт базового класу.

d1 - Об'єкт класу derived1

d2 - Об'єкт класу derived2

Крім того, вказівнику p привласнюється адреса об'єкту b, а функція vfunc() викликається за допомогою вказівника p. Оскільки вказівник p посилається на об'єкт класу base, виконується варіант функції vfunc() з базового класу. Потім вказівнику p привласнюється адреса об'єкту d1, і функція vfunc() знову викликається з його допомогою. Цього разу вказівник p посилається на об'єкт класу derived1. Отже, викликається функція derived1:: vfunc(). В результаті вказівнику p привласнюється адреса об'єкту d2, тому вираз p->vfunc() приводить до виклику функції vfunc() з класу derived2. Принципово важливо, що варіант функції, що викликається, визначається

типом об'єкту, на який посилається покажчик p. Крім того, вибір відбувається в ході виконання програми, що забезпечує основу динамічного поліморфізму.

Віртуальну функцію можна викликати звичайним способом, використовуючи ім'я об'єкту і оператора ".", проте поліморфізм досягається тільки при зверненні до неї через вказівник. Наприклад, наступний фрагмент програми є абсолютно правильним.

d2.vfunc(); // Викликається функція vfunc() з класу derived2

Не дивлячись на те що такий виклик віртуальної функції помилкою не є, ніяких переваг він не надає.

На перший погляд, перевизначення віртуальної функції в похідному класі мало відрізняється від звичайного перевантаження функцій. Проте це не так, і термін перевантаження непридатний до перевизначеня віртуальних функцій з кількох причин. Найбільш важлива відмінності:

1. прототип віртуальної функції, що перевизначається, повинен точно співпадати з прототипом, визначеним в базовому класі. Віртуальні функції відрізняються від перевантажених, які відрізняються типами і кількістю параметрів. (Фактично при перевантаженні функцій типи і кількість і параметрів повинні відрізнятися! Саме ці відмінності дозволяють компілятору вибрати правильний варіант переобтяженої функції.) При перевизначенні віртуальної функції всі аспекти їх прототипів повинні бути однаковими. Якщо не дотримував правило, компілятор вважатиме ці функції просто перезавантаженими, а їх віртуальна природа буде втрачена;
2. що віртуальні функції не можуть бути статичними членами класів;
3. віртуальні функції не можуть бути дружніми функціями;
4. конструктори не можуть бути віртуальними, хоча на деструкції це обмеження не розповсюджується.

Із-за перерахованих обмежень для перевизначення віртуальної функції в похідному класі використовується термін заміщення (overriding).

Віртуальні функції описуються за допомогою ключового слова virtual в базовому класі. Це означає, що в похідному класі цей метод може бути заміщений методом, більш відповідним для цього похідного класу.

Оголошений віртуальним в базовому класі, метод залишиться віртуальним для всіх похідних класів.

Якщо в похідному класі віртуальний метод не буде перевизначений, то при виклику буде знайдений метод з таким ім'ям вгору за ієрархією класів (тобто в базовому класі).

Виклик віртуальної функції з допомогою посилання на об'єкт базового класу.

У попередньому прикладі віртуальна функція викликалася за допомогою вказівника на об'єкт базового класу, проте поліморфна природа віртуальних функцій збережена і при їх виклику за допомогою посилання на об'єкт базового класу. Посилання є неявним вказівником. Таким чином, посилання на об'єкт базового класу можна використовувати для звернення до об'єкту базового або будь-якого похідного класу. Якщо віртуальна функція викликається за допомогою посилання на об'єкт базового класу, її варіант залежить від типу об'єкту, на який встановлено посилання у момент виклику.

В більшості випадків віртуальна функція за допомогою посилання викликається для передачі параметрів. Розглянемо ще один варіант попередньої програми.

```
void f (base &r)
    {r.vfunc() ;}
int main() {
    base b;
    derived1 d1;
    derived2 d2 ;
    f(b); // Функції f() передається об'єкт класу base.
    f(d1); // Функції f() передається об'єкт класу derived1.
```

```
f(d2); // Функції f() передається об'єкт класу derived2.
return 0;}
```

Програма виводить на екран ті ж повідомлення, що і раніше. У даному прикладі функція f() отримує як параметр посилання на об'єкт класу base. У функції main() ця функція викликається за допомогою об'єктів класів base, derived1 і derived2. Конкретний варіант функції vfunc() вибирається усередині функції f() залежно від типу її параметра, простоти в решті прикладів цього розділу віртуальні функції викликаються з покажчиків, причому результат нічим не відрізняється від виклику за допомогою посилань.

Атрибут virtual успадковується

При наслідуванні віртуальної функції її віртуальна природа також успадковується. Це означає, що якщо похідний клас, що успадкував віртуальну функцію від базового класу, стає базовим по відношенню до іншого похідного класу, віртуальна функція може як і раніше заміщатися. Інакше кажучи, не має значення, скільки разів успадковувалася віртуальна функція, вона все одно залишається віртуальною. Розглянемо наступну програму.

```
#include <iostream>
using namespace std;
class base {
    public:
    virtual void vfunc () { cout << "Функція vfunc() з класу base.\n";}
class derived1 : public base {
    public:
    void vfunc () { cout << " Функція vfunc() з класу derived1.\n";}
    };
class derived2 : public derived1 {
    public:
```

void vfunc() {cout « " Функція vfunc() з класу derived2.\n"; }};

В даному випадку клас derived2 є спадкоємцем класу derived1, а класу base, але функція vfunc() залишається віртуальною.

Віртуальні функції є ієрархічними

Як відомо, якщо функція оголошена віртуальною в базовому класі, її можна змінювати в похідному класі. Проте віртуальну функцію не обов'язково змінювати. В цьому випадку викликається функція, визначена в базовому класі.. Спадкоємство в мові С++ організоване за ієрархічним принципом, тому віртуальні функції також повинні бути ієрархічними. Це означає, що якщо віртуальна функція не змінюється, викликається її попередня перевизначена версія. Наприклад, в наступній програмі клас derived2 є спадкоємцем класу derived1, який, у свою чергу, є похідним від класу base. Функція vfunc() у класі derived2 не заміщається. Отже, найближча до класу derived2 версія функції vfunc() визначена в класі derived1. Таким чином, виклик функції vfunc() за допомогою об'єкту класу derived2 відносить до функції derived1:: vfunc().

Чисто віртуальні функції

Якщо віртуальна функція не змінюється в похідному класі, то викликається її версія з базового класу. Проте у багатьох випадках неможливо створити розумну версію віртуальної функції в базовому класі. Крім того, в деяких ситуаціях необхідно гарантувати, що віртуальна функція буде змінена у всіх похідних класах. Для таких ситуацій в мові С++ передбачені чисто віртуальні функції.

Чисто віртуальна функція (pure virtual function) – *це функція, що не має визначення в базовому класі, у неї немає тіла, а є лише декларація про її існування.*

Для оголошення чисто віртуальної функції використовується наступна синтаксична конструкція.

virtual тип імя_функції (список_параметрів) = 0;

Чисто віртуальні функції повинні перевизначатися в кожному похідному класі, інакше виникне помилка компіляції.

Слід мати на увазі, що всі похідні класи зобов'язані перевизначати чисто віртуальну функцію. Якщо цього не зробити, виникне помилка компіляції.

Абстрактні класи

Клас, що містить хоч би одну чисто віртуальну функцію, називається абстрактним (abstract class). Оскільки абстрактний клас містить одну або декілька функцій, що не мають визначення (тобто чисто віртуальні функції), його об'єкт створити неможливо. Отже, абстрактні класи можна чи використовувати як основу для похідних класів.

Не дивлячись на те що об'єкти абстрактного класу не існують, можна створювати покажчики і посилання на абстрактний клас. Це дозволяє застосовувати абстракту класи для підтримки динамічного поліморфізму і вибирати відповідаю віртуальну функцію залежно від типу покажчика або посилання.

```
#include <iostream>
using namespace std;
class base {
    public:
    virtual void vfunc() =0;
class derived1 : public base {
    public:
    void vfunc() { cout « "Функція vfunc() з класу derived1.\n";}
    };
class derived2 : public base {
    public:
    void vfunc() { cout « " Функція vfunc() з класу derived2.\n" ; }};
```

Абстрактні класи не бувають ізольованими, тобто завжди абстрактний клас має бути успадкованим. Оскільки в чисто віртуального методу немає тіла, то створити об'єкт абстрактного класу неможливо. Крім того, щоб уникнути появи помилки при виклику чистого віртуального методу, похідний клас повинен містити і декларування, і тіло чистого віртуального методу.

Абстрактним класом можна назвати клас, спеціально визначений для забезпечення спадкоємства характеристик породженими класами.

Таблиці віртуальних методів (функцій)

Таблиця віртуальних методів (функцій) (ТВМ) (англ. virtual method table, VMT) - координуюча таблиця або vtable - механізм, використовуваний в мовах програмування для підтримки динамічної відповідності (або методу пізнього скріплення).

Координуюча таблиця об'єкту містить адреси динамічно зв'язаних методів об'єкту. Метод викликається при вибірці адреси методу з таблиці. Координуюча таблиця буде тією ж самою для всіх об'єктів, що належать тому ж класу, тому допускається її спільне використання. Об'єкти, що належать класам, сумісним за типом (наприклад, що стоять на одному рівні в ієрархії спадкоємства), матимуть схожі координуючі таблиці: адреса даного методу зафіксується з одним і тим же самим зсувом для всіх класів, сумісних за типом.

У стандартах C++ немає чіткого визначення, як повинна реалізовуватися динамічна координація, але компілятори частенько використовують деякі варіації однієї і тієї ж базової моделі.

Зазвичай компілятор створює окрему ТВМ для кожного класу. Після створення об'єкту генерується віртуальний табличний вказівників (або vpointer) на цю vtable, він додається як прихований член даного об'єкту (а частенько як перший член). Компілятор також генерує "прихований" код в

конструкторі кожного класу для ініціалізації vpointer'ів його об'єктів адресами відповідної vtable.

Приклад розглянемо наступні оголошення класу в синтаксисі C++:

```
class B1{
    public:
    void f0() {}
    virtual void f1() {}
    int int_in_b1;
    };
class B2 {
    public:
    virtual void f2() {}
    int int_in_b2;
    };
```

використовуємо для створення наступного класу множинне успадкування:

```
class D : public B1, public B2 {
    public:
    void d() {}
    void f2() {} // перевизначаємо B2::f2()
    int int_in_d;
    };
```

в наступному фрагменті C++ коду:

```
B2 *b2 = new B2();//покажчик на об'єкт класу B2
D *d = new D();//покажчик на об'єкт класу D
```

створюється 32-бітова схема пам'яті для об'єкту b2:

b2:

+0: покажчик на таблицю віртуальних методів B2

+4: значення int_in_b2

таблиця віртуальних методів B2:

+0: B2::f2()

а для об'єкту d схема пам'яті буде такою:

d:

+0: покажчик на ТВМ D (для B1)

+4: значення int_in_b1

+8: покажчик на ТВМ D (для B2)

+12: значення int_in_b2

+16: значення int_in_d

Всього: 20 bytes.

ТВМ D (для B1):

+0: B1::f1() // B1::f1() не перевизначена

ТВМ D (для B2):

+0: D::f2() // B2::f2() перевизначена D::f2()

Необхідно відзначити, що невіртуальні функції (такі як f0()) в загальному випадку не можуть з'являтися в vtable, але в деяких випадках є виключення (як, наприклад, конструктор за умовчанням).

Перевизначення методу f2() у класі D реалізується дублюванням ТВМ B2 і заміною вказівника на B2::f2() вказівником на D::f2().

Контрольны запитання:

1. Що таке віртуальні функції?
2. Яка особливість чисто віртуальних функцій?
3. Що таке абстрактні класи?
4. Особливості успадкування атрибуту virtual.
5. Таблиця віртуальних функцій.

Адреси, вказівники

Спеціальними об’єктами у програмах на мовах С та С++ є вказівники. Існують два типи вказівників: вказівники-змінні та вказівники-константи. Значеннями вказівників служать адреси дільниць пам’яті, відведених для об’єктів конкретних типів. Саме тому у визначеннях та оголошеннях вказівників завжди присутні позначення відповідного типу, що дає змогу за допомогою вказівника одержати доступ до сього об’єкту, що зберігається.

Вказівники діляться на дві категорії – вказівники на об’єкти та вказівники на функції, що передбачає різні властивості та правила використання.

Вказівники на об’єкти

Визначення та оголошення вказівника на змінну має вигляд:
*type *ім’я_вказівника;*

де type – позначення типу; ім’я_вказівника – ідентифікатор; * - унарна операція розкриття посилання (операція розйменування; операція звернення за адресою), операндом якої повинен бути вказівник (саме тому слідом за “*” записане ім’я_вказівника).

Ознакою вказівника при розгляді визначення чи оголошення служить символ “*”, вміщений перед іменем, що означає “вказівник на об’єкт даного типу. Таким чином, при потребі визначити кілька вказівників на об’єкти одного типу цей символ записують перед кожним іменем. Наприклад, визначення int *i1p, *i2p, *i3p, i; вводить три вказівники на об’єкти цілого типу i1p, i2p, i3p та одну змінну i цілого типу. Змінній i відведеться 2 байти пам’яті, а вказівникам i1p, i2p, i3p - дільниця пам’яті, визначена реалізацією, часто 2 байти.

Оскільки при визначенні у більшості випадків доцільно виконати ініціювання, тому формат може бути таким: type *ім’я_вказівника ініціювач;

Ініціювач має дві форми визначення вказівників:

type *ім’я_вказівника= вираз_ініціювання;

type *ім’я_вказівника (вираз_ініціювання);

вираз_ініціювання повинен бути константним виразом, наприклад:

- явно вказана адреса дільниці пам’яті;
- вказівник, якому вже надане значення;
- вираз, отримання адресу об’єкта за допомогою операції ‘&’.

Якщо значення константного виразу дорівнює 0, тоді це нульове значення перетвориться на порожній (нульовий) вказівник. Синтаксис мови “гарантує, що цей вказівник відрізняється від вказівника на довільний об’єкт”. Окрім того, внутрішнє (бітове) відтворення порожнього вказівника може відрізнятися від бітового відтворення цілого значення 0. Приклади визначення вказівників:

char cc = ‘d’; //Символьна змінна (типу char)

char *pc = &cc; // Ініційований вказівник на об’єкти типу char

char *ptr(NULL); // Нульовий вказівник на об’экт типу char

char *p; // Неініційований вказівник на об’єкт типу char

Змінна cc ініційована значенням символьної константи ‘d’. Після визначення (з ініціюванням) вказівника pc доступ до значення змінної cc можливий як через ім’я, так і через адресу, що є значенням вказівника на змінну pc. У останньому випадку застосовується операція розіменування “*” (одержання значення через вказівник). Таким чином, при виконанні оператора cout<< “\n cc дорівнює “<<cc<<” і pc=”<<pc; виведеться cc дорівнює d та*pc=d. Вказівники ptr та p, визначені у прикладі, мають різні “права”. Вказівник ptr має нульове початкове значення (порожній вказівник), і спроба розйменування не виконується.

Порожній вказівник, не поєднаний з дільницею пам’яті, повинен мати нульову адресу і зберігати нульове значення, що не забезпечується синтаксисом мови C++. Проте надавши ptr значення адреси існуючого об’єкта, можна коректно застосовувати операцію розйменування. Наприклад, довільний з операторів надання значення ptr=&cc;чи ptr=pc; поєднає ptr з

дільницею пам'яті, відведеної для змінної cc , тобто після їх виконання значенням *ptr буде 'd'.Надавши вказівнику адресу конкретної дільниці пам'яті, можна за допомогою операції розйменування не лише читати, але й змінювати її вміст. Наприклад, оператори надання :*pc='+'; чи ptr=pc; *ptr='+';нададуть змінній cc значення '+'.

Унарний вираз *вказівник володіє правами імені змінної, тобто *pc та *ptr є синонімами (псевдонімами, іншими іменами) імені cc. Вираз *вказівник можна використовувати практично скрізь, де допустиме використання імен об'єктів того типу, до якого відноситься вказівник. Проте це твердження справедливе лише тоді, коли вказівник ініційований при визначенні явним способом. У наведеному прикладі не ініційований вказівник p. Тому спроби використати вираз *p у лівій частині оператора надання чи в операторі введення неправомірні. Значення вказівника p невідоме, а результат запису значення у невизначену дільницю пам'яті непередбачена і іноді призводить до аварійних подій.

*p='%' // Помилкове використання неініційованого p.

При наданні вказівнику адреси конкретного об'єкта (p=&cc;) чи значення вже ініційованого вказівника (p=pc;) *p перетвориться на синонім існуючого імені об'єкта.

Для поєднання неініційованого вказівника з новою дільницею пам'яті, ще не відведеної ніяким об'єктам програми, використовується оператор new чи надається вказівнику явна адреса: p=new char;

//Відвели пам'ять змінній типу char та поєднали вказівник p з дільницею пам'яті

p=(char *) 0xb8000000;// Початкова адреса відеопам'яті для кольорового монітора у текстовому режимі

Числове значення перетворюється на адресу (char *).

Після довільного з записаних операторів можна використовувати *p для запису у пам'ять потрібних симовльних значень. Наприклад, допустимими будуть оператори: *p='&'; чи cin>>p;

Числове значення адреси може використовуватися не лише при наданні вказівнику значення при виконанні програми, але й при ініціюванні вказівника при його визначенні. Потрібно лише не забувати про потребу явного перетворення типів. Наприклад, у наступному визначенні вказівник з іменем Computer при ініціюванні отримає значення адреси того байта, у якому містяться відомості про тип комп'ютера, на якому виконується програма (тільки для IBM – сумісних комп'ютерів): char *Computer = (char *)0xF000FFFF;

При визначенні вказівника його та його значення можна оголошувати константами. Для цього використовують модифікатор const:

type const *const і'мя_вказівника ініціювач;

Модифікатори const - це не обов'язкові елементи визначення. Найближчий до імені вказівника модифікатор const перед символом '*'визначає "константність" початкового значення, поєднаного з вказівником. Мнемоніка зрозуміла, оскільки вираз *ім'я_вказівника є зверення до вмісту відповідного вказівника дільниці пам'яті. Таким чином, визначення константного вказівника має такий формат:

type *const ім'я_вказівника ініціювач;

Для прикладу визначимо вказівник-константу key_byte та поєднаємо його з байтом, що відтворює поточний стан клавіатури ПЕОМ IBM PC:char *const key_byte=(char *)1047;

Значення вказівника key_byte не можна змінити, він завжди вказує на байт з адресою 1047 (шістнадцятковий запис 0x0417). Це основний байт стану клавіатури.

Оскільки значення вказівника-константи змінити неможливо, тому ім'я вказівника можна вважати найменуванням конкретної фіксованої адреси дільниці основної пам'яті. Вміст цієї дільниці пам'яті за допомогою розйменування вказівника-константи у загальному випадку доступне як для читання, так і для зміни.

Таким чином, вміст дільниці пам'яті, на яку "дивиться" вказівник-

константа, можна явно змінити. Спробу змінити значення власне вказівника-константи, тобто операцію типу key_byte=NULL; не допустить компілятор і видасть повідомлення про помилку: Error…:Cannot modify a const object. Формат визначення вказівника на константук:

type const * ім'я_вказівника ініціювач; Наприклад, введемо вказівник на константу цілого типу зі значенням 0: const int zero=0; // Визначення константи

int const *point_to_const=&zero; // Вказівник на константу 0

Оператори вигляду point_to_const=1; cin>>poiny_to_const; - недопустимі, оскільки кожен з них – це спроба змінити значення константи 0.

Працюючи з вказівниками, постійно використовують операцію & - одержання адреси об'єкта. Для неї існують певні обмеження:

- не можна визначати адресу неіменованої константи, тобто недопустимі вирази &3.1411593 чи & '?'.

- не можна визначати адресу значення, що одержується при обчисленні скалярних виразів, тобто недопустимі конструкції &(44*x-z) чи &(a+b) !=12.

- не можна визначати адресу зміної, що відноситься до класу пам'яті register. Тому помилковою буде послідовність операторів:

int register Numb=1; int *prt_Numb=&Numb;

Стандарт мови С++ повідомляє, що операцію & можна застосовувати до об'єктів, що мають ім'я та розташовані в пам'яті. ЇЇ не можна застосовувати до виразів, неіменованих констант, бітових полів структур та об'єднань, до регістрових змінних та зовнішніх об'єктів (файлів).

Адресна арифметика, типи вказівників та операції над ними.

Операція & одержання адреси об'єкта завжди дає однозначний результат, який залежить від розташування об'єкта в пам'яті, а операція розіменування * вказівник залежить не лише від значення вказівника, але й від його типу. Справа у тому, що при доступі до пам'яті через розіменування вказівника потрібна інформація не лише про розташування, але й про розміри

дільниці пам’яті, які будуть використовуватися. Цю додаткову інформацію компілятор одержує з типу вказівника. Явне перетворення типів при роботі з різними вказівниками в одному виразі потрібне для всіх вказівників, окрім тих, що мають тип void *. При використанні вказівника типу void *операція перетворення типів застосовується при замовчуванні. На відміну від інших типів, тип void передбачає відсутність значення. Вказівник типу void *відрізняється від інших вказівників відсутністю відомостей про розмір відповідної йому дільниці пам’яті.

Операції над вказівниками можна згуртувати таким чином:

- операція розйменування чи доступу за адресою (*);
- перетворення типів (приведення типів);
- надання;
- одержання значення адреси (&);
- додавання та віднімання (адитивні операції);
- інкремент чи автозбільшення (++);
- декремент чи автозменшення (--);
- операції відношення (операції порівняння).

Розіменування, приведення типів, надання розглядалися вище. Про одержання адрси вказівника можна сказати: вказівник є об’єкт і тому має адресу відповідної йому дільниці пам’яті. Значення цієї адреси доступне через операцію &.

Віднімання застосовується до вказівників на об’єкти одного типу і до вказівника та цілої константи. Застосовуючи операцію віднімання до двох вказівників одного типу, можна визначати “відстань” між двома дільницями пам’яті. «Відстань» визначається в одиницях, кратних довжині (у байтах) об’єкта того типу, до якого віднесений вказівник. Таким чином, різниця однотипних вказівників, що адресують два суміжних об’єкти довільного типу, за абсолютним значенням завжди дорівнює 1. Подібно до описаного виконується операція додавання вказівника до цілочислового значення. Додавати два вказівники заборонено синтаксисом мови С++.

Декремент та інкремент вказівників не мають ніяких нових особливостей. Як і віднімання одиничної константи, операція – змінює конкретне числове значення вказівника типу type на величину sizeof(type), де type *- тип вказівника. Таким чином, вказівник пересувається до сусіднього об'єкту з меншою адресою. Подібна до описаного вище декременту операція одиничного автозбільшення ++. Залежно від положення (перед операндом-вказівником чи за ним) виконання унарних операцій ++ та – відбувається або перед або після використання значення вказівника.

Оскільки вказівник – це об'єкт в пам'яті, тому можна визначчати вказівник на вказівник і т.д. скільки завгодно разів. Наприклад, у наступній програмі визначені такі вказівники і через низх виконується доступ до начення змінної:

```
#include <iostream>
using namespase std;
int main() { int i=88; int *pi=&i; int **ppi=&pi; int ***pppi=&ppi;
cout<<"\n ***pppi="<<***pppi;
return 0;}
```

В результаті виконання програми виконується наступна відповідь:

***pppi = 88

Контрольні запитання:

1. Що таке вказівники і для чого вони використовуються?
2. Які операції можна проводити над вказівниками?
3. Переваги та недоліки використання вказівників.
4. Які обмеження накладаються на виокристання вказівників?

РОЗДІЛ IV Приклади проектування та реалізації типових задач засобами С++

Структури та об’єднання

З основних типів мови С++ користувач може конструювати похідні типи, наприклад, структури та об'єднання. Разом з масивами та класами структури та об’єднання відносяться до структуризованих типів.

Структура як тип та сукупність даних.

Структура – це об’єднана в одне ціле множина поіменованих елементів у загальному випадку різних типів. Порівнюючи структуру з масивом, потрібно відмітити, що масив – це сукупність однорідних об’єктів, що має загальне ім’я – ідентифікатор масива. Іншими словами, сві елементи масива є об’єктами одного типу. Це не завжди зручно, оскільки виникає потреба у використанні єдиного цілого, де всі дані мають різну довжину та різні типи. Об’єднати такі різнорідні дані зручно у структурі. Кожна структура містить у собі один чи кілька об’єктів (змінних, масивів, вказівників, структур) , що називаються елементами структури. Відомості про дані, що входять у структуру, можна відтворити структурним типом.

Відповідно до синтаксису мови визначення структурного типу починається службовим словом struct, слідом за яким записується вибране користувачем ім’я типу. Оголошення елементів, що відносіться до структури, записуються у фігурних дужках, слідом за якими записується крапка з комою. Елементи структури можуть бути як базових, так і похідних типів.

Якщо структура визначається одноразово, тобто не потрібно у різних фрагментах програми визначати чи оголошувати однакові за внутрішнім складом структуровані об’єкти, тоді можна не вводити іменований структурний тип, а безпосередньо визначати структури з визначенням їх компонентного складу.

Для звернення до об’єктів, що є елементами структури, найчастіше використовуються уточнені імена. Загальною формою уточненого імені елемента структури є наступна конструкція:

ім’я_структури.ім’я_елемента_структури

Наприклад, якщо визначити структуру, що описує бібліографічну картку:

*struct card {char *autor; // П.І.Б. автора книги*

*char *title; //Заголовок книги*

*char *city; //Місце видання*

*char *firm; //Видавництво*

int year; //Рік видання

int pages; //Кількість сторінок

} c;

тоді, звернутися до елемента year структури card можна через змінну с таким чином: c.year=1998. Можна також надрукувати це значення: cout<<c.year;

При визначенні структур можливе ініціювання, тобто надання початкових значень їх елементам. Наприклад, оголосивши структурний тип card, можна таким чином визначити та ініціювати конкретну структуру:

card distionary= {“Hornby A.S.”, “Oxford distionary of Current English”, \

“Oxford”, “Oxford University”, 1984, 769};

Таке визначення еквівалентне наступній послідовності операторів:

card distionary; distionary.author=”Hornby A.S.”;

distionary.title=“Oxford students distionary of Current English”;

distionary.city= “Oxford”; distionaru.year=1984; distionary.pages=769;

Важливо розрізняти ім’я конкретної структури distionary та ім’я структурного типу card. З іменем структурного типу не поєднаний ніякий об’єкт, і тому через нього не можна звертатись за уточненим іменем елементів. Визначення структурного типу оголошує лише шаблон (формат, внутрішню побудову) структур. Ідентифікатор card у нашому прикладі – це

назва структурного типу, тобто “ярлик” чи “етикетка” структур, які будуть визначені у програмі. Для пояснення різниці між іменем структурного типу та іменем конкретних структур, що мають такий тип, у англомовній літературі по мові С++ для позначення структурного типу використовують тармін tag (ярлик, етикетка, бірка). У прикладі визначення структурного типу поєднане з визначенням конкретної структури такого типу с.

Оскільки ім’я структурного типу має всі права імен типів, тому можна визначати вказівники на структури:

ім’я_структурного типу * ім’я_вказівника_на_структуру;

Вказівник, що визначається, можна ініціювати. Значенням кожного вказівника на структуру може бути адреса структури того типу, скажемо так, номер байта, починаючи від якого структура розташована в пам’яті. Структурний тип вказує її розміри, визначаючи, на яку величину (на скільки байтів) зміниться значення вказівника на структуру, якщо до нього додати 1 (чи відняти від нього 1).

Після визначення такого вказівника виникає ще одна можливість доступу до елементів структури *с* через операцію ‘->’ доступу до елемента структури, з якою на дану мить поєднаний вказівник. Формат відповідного виразу такий: ім‘я_вказівника -> ім’я_елемента_структури

Наприклад с->firm

Інша можливість звернення до елемента структури через вказівник, що її адресує – це розйменування вазівника ‘*’ та створення уточненого імені такого вигляду: (*ім’я_вказівника).ім’я_елемента_структури

Важливим є використання круглих дужок, оскільки операція розйменування повинна відноситися лише до імені вказівника, а не до уточненого імені елемента структури. Таким чином, наступні три вирази еквівалентні:

card ptrcard=&c; (ptrcard).firm

ptrcard->firm

c.firm

Всі вони відносяться до одного елементу firm конкретної структури card. Як і для інших об‘єктів, для структур можна визначати посилання:

ім’я_структурного_типу& ім’я_посилання_на структуру ініціювач

Наприклад, для введеного вище структурного типу можна ввести посилання card& fc=c; і після такого визначення до елемента year можна звертатися як через c.year, так і через fc.year.

Всі елементи структури розташовані в пам’яті послідовно, їм загалом відводиться саме стільки пам’яті, скільки структурі в цілому.

У відношенні до елементів структур існує практично лише одне обмеження – елемент структури не може мати той же тип, що й визначений структурний тип., тобто помилковим буде визначення: struct v{ v s; int n;}; Але можливе оголошення вказівника на структуру, що визначається, наприклад: struct cor{cor *pc; long f;};

Елементом структури, що визначається, може бути структура, визначена раніше, наприклад: struct beg{int k;char *h;} st; struct nex{beg b; float d;};

Якщо у визначенні структурного типу потрібно використати вказівник на структуру іншого типу як елемент, тоді дозволяється така послідовність визначень struct A; //Неповне визначення структурного типу

*struct B {struct A *pa;};*

*struct A{struct B *pb;};*

Неповне визначення структурного типу A можна використовувати у визначенні структурного типу B, оскільки визначення вказівника pa на структуру типу A не вимагає відомостей про розмір структури типу A.

Наступне визначення у тій же програмі структурного типу A обов’язкове. Використання у структурах типу A вказівників на структури вже введеного типу B не потребує пояснень.

Розглядаючи “взаємовідношення” структур та функцій, маємо дві можливості: значення, що повертає функція, та параметри функції. Функція може повертати структуру як результат: struct h{char *n; int num;};

h func(void); // Прототип функції

Функція може повертати вказівник на структуру:

h *fun(void); //Прототип функції

Функція може повертати посилання на структуру:

h& f(void); //Прототип функції

Через аппарат параметрів інформація про структуру може передаватися у функцію або безпосередньо, або через вказівник, або через посилання:

void f1 (h str); //Пряме використання

*void f2(h *pst); //Через вказівник*

void f3 (h& rst); //Через посилання

Застосування посилання на об'єкт як параметра дає змогу уникнути повторення об'єкта в пам'яті. До структур застосовується операція new відведення динамічної пам'яті, операндом для неї може бути структурний тип. У такому випадку відводиться пам'ять для структури заданого типу, і операція new повертає вказівник на відведену пам'ять. Пам'ять можна відводити і для масиву структур, наприклад:

L=new card[20];// Пам'ять для масива структур

У такому випадку операція new повертає вказівник на початок масива.

Об'єднання різнотипних даних.

Об'єднання оголошуються через службове слово union, наприклад, union{ long L; int t,r; char c[4];} un;

Всі елементи об'єднання мають одну початкову адресу, розміри елементів відповідають їх типам, а розмір об'єднання визначається максимальним розміром його елементів.

Таким чином, об'єднання можна розглядати як структру, всі елементи якої при розташуванні в пам'яті мають нульове зміщення від початку, тобто всі елементи об'єднання розташовуються на одній дільниці пам'яті. Розмір дільниці пам'яті, відведеної для об'єднання, визначається максимальною з довжин його елементів.Для об'єднань вводиться поєднуючий тип:

```
union ім’я_поєднуючого типу {елементи_об’єднання}
```

Приклад поєднуючого типу:

```
union mix {double d; long[2]; int K[4];};
```

Після оголошення типу об’єднання можна визначати конкретні об’єднання, їх масиви, а також вказівники та посилання на об’єднання:

```
mix mA, mB[4]; // Об’єднання та масив об’єднань
mix *pmix; //Вказівник на об'єднання
mix& rmix=mA; //Посилання на об’єднання
```

Для звернення до елемента об’єднання використовується уточнене ім’я: ім’я_об’єднання.ім’я_елементаабо конструкцію, що містить вказівник:

```
вказівник_на об’єднання->ім’я_елемента
(*вказівник_на_об’єднання).ім’я_елемента
```

або конструкцію, що містить посилання:

```
посилання_на_об’єднання.ім’я_елемента
```

Приклади:

```
mA.d=64.8; mB[2].E[1]=10L; pmix=&mB[0]; pmix->E[0]=66;
cin>>(*pmix).K[1]; cin>>rmix.E[0];
```

Основна перевага об’єднання – можливість по-різному розпізнавати один вміст (код) дільниці пам’яті, тому основним призначенням об’єднань є доступ до однієї дільниці пам’яті через об’єкти різних типів. Потреба у такому механізмі виникає, наприклад, при віділенні з внутрішнього відтворення (з коду0 об’єкта заданого фрагмента.

Елементами об’єднань можуть бути масиви та структури. При визначенні конкретних об’єднань дозволене їх ініціювання, проте ініціюється лише перший елемент об’єднання, наприклад:

```
union comp {long L; int I[2]; char C[4];}; comp m1={11111111 };
```

Дозволяється створювати масиви об’єднань та ініціювати їх:

```
comp mi[]={1L,2L,3L,4L };
```

Тут для кожного елемента mi введеного масива з чотирьох об'єднань типу comp ініціюється лише перший компонент об'єднання. Тобто початкове значення отримав кожен елемент mi[i].L.

Бітові поля структур та об'єднань.

У структурах та об'єднаннях можна використовувати як компоненти (елементи) бітові поля. Кожне бітове поле є цілим чи беззнаковим цілим значенням, яке розташоване в пам'яті у фіксованій кількості бітів (у компіляторі BC++ від 1 до 16 бітів). Бітові поля можуть бути лише елементами структур, об'єднань та класів (побачимо пізніше), тобто бітові поля не можна оголошувати як самостійні об'єкти програми. Бітові поля не мають адрес, тобто для них не визначена операція '&', немає посилань та вказівників на бітові поля. Їх не можна об'єднувати у масиви. Призначення бітових полів – забезпечити зручний доступ до окремих бітів даних. Через бітові поля можна створювати об'єкти з довжиною внутрішньої побудови, не кратною байту. Це дає змогу стискати інформацію з метою заощадження пам'яті.Структура з бітовими полями визначається таким чином:

struct {тип_поля ім'я_поля:ширина_поля;

тип_поля ім'я_поля:ширина_поля; ...}ім'я_структури;

Тут тип_поля – один з базових цілих типів int, unsigned int (скорочено unsigned), signed int(скорочено signed), char, short, long та їх знакові та беззнакові варіанти.

ім'я_поля – ідентифікатор, що вибирається користувачем;

ширина_поля – ціле невід'ємне десяткове число, значення якого не перевищує довжини слова конкретної ЕОМ.

Приклад запису бітових полів у структурі: struct {int a:10; int b:14; } xx, *px;

Діапазон можливих значень ширини_поля та порядок розташування полів структури суттєво залежать від реалізації.

Рекурсія

Загалом, рекурсія значить самоповторюваний шаблон. В математиці це може бути функція визначена через себе. Інакше кажучи, це функція, що викликає сама себе. Кожна рекурсивна функція має умову завершення; інакше вона буде викликати себе безперестанку, і ця умова може бути названою *базова умова*.

Зазвичай, рекурсія це трошки складна для розуміння більшості студентів концепція.

Типи рекурсії

В C++, типи рекурсії можна визначити більш ніж за одним виміром. З одного боку, їх можна категоризувати як рекурсії час виконання і як рекурсії часу компіляції через використання шаблонів.

Рекурсії часу виконання найбільш уживана техніка в C++. Вона може бути здійснена через функції (функції члени), які викликають самі себе.

В C++, ми також можемо реалізувати рекурсії часу компіляції за допомогою шаблонів. Коли ви інстанціюєте клас (або структуру) шаблон в C++, компілятор створює код цього класу під час компіляції. Як і рекурсії часу виконання, клас шаблон може інстанціонувати себе самого для забезпечення рекурсії. Так само потрібна умова завершення; інакше інстанціонування триватиме безупинно, щонайменше теоретично, хоча звісно в дійсності цей процес обмежений ресурсами комп’ютера і компілятора. В такому шаблоні, ви можете визначити умову завершення (базову умову) за допомогою спеціалізації шаблона або часткової спеціалізації, залежно від умови завершення.

Дехто може подумати, що він зможе реалізувати рекурсію за допомогою макросов препроцесора C++, бо вони теж заміняються під час компіляції. Насправді, препроцесор заміняє макроси ще перед компіляцією. Також препроцесор має багато обмежень; через просту заміну тексту, але

найбільш критичне обмеження те, що макроси не можуть викликати себе рекурсивно; тож ви не можете програмували рекурсії за допомогою макросів, накшталт мета-програмуванню.

Інший погляд на рекурсію це погляд на те як реалізовані алгоритм рекурсії. Алгоритм рекурсії може бути реалізованим більш ніж одним способом. Можливі варіанти рекурсії це лінійна, хвостова, обопільна, двійкова і вкладена. Ви можете здійснити їх або під час компіляції через використання шаблонів або під час виконання через використання функцій або функцій членів.

Існують такі типи діаграм:

- вкладена
- двійкова
- обопільна
- хвостова
- лінійна

Зараз ви зможете дослідити різні типи алгоритмів один за одним і побачити їх реалізації часу виконання і часу компіляції.

Лінійна рекурсія

Лінійна рекурсія це найпростіший вид рекурсії і можливо найбільш вживана рекурсія. В цій рекурсії, одна функція просто викликає себе доти, доки не досягне умови завершення (також відомої як базова умова); цей процес відомий як *намотування*. Як тільки виконана умова завершення, виконання програми повертається до викликальника; це зветься *розмотуванням*.

Впродовж намотування і розмотування функція може виконувати якісь додаткові корисні задачі; у випадку факторіала вона множить вхідне значення на значення повернуте під час фази розмотування. Цей процес може бути зображений у вигляді наступної діаграми (знизу), яка показує обидві фази функції обчислення факторіала із використанням лінійної рекурсії.

Математично, ви можете написати функцію обчислення факторіала таким чином; інакше кажучи, коли значення “n” нуль, повертати одиницю і, коли значення “n” більше ніж нуль, викликати функцію рекурсивно з “n-1″ і помножити результат на “n”.

```
int Factorial(int no)
{  // перевірка на помилковий параметр
  if (no < 0)
  return -1;
  // умова завершення
  if (0 == no)
  return 1;
  // лінійний рекурсивний виклик
  return no * Factorial(no - 1);
}
```

Попередня програма являє собою реалізацію лінійної рекурсії часу виконання. Тут ми маємо умову завершення у вигляді 0; програма починає виконувати розмотування коли досягає умови завершення. В цій програмі присутня перевірка на введення від’ємного числа, яке могло б спричините нескінченну рекурсію. Ця функція просто поверне -1 як помилку якщо параметр від’ємний.

Хвостова рекурсія

Хвостова рекурсія це спеціальна форма лінійної рекурсії, де рекурсивний виклик зазвичай іде останнім у функції. Цей тип рекурсії здебільшого більш ефективний, бо розумні компілятори автоматично перетворять таку рекурсію в цикл задля уникнення вкладених викликів функцій. Через те, що рекурсивний виклик функції зазвичай останнє, що робить функція, вона не має потреби ще щось робити під час розмотування; натомість, вона просто повертає значення отримане через рекурсивний виклик. Ось приклад тої самої програми реалізованої як хвостова рекурсія.

Ви можете визначити хвостову рекурсію математично через наступну формулу; інакше кажучи, коли значення “n” нуль, просто повернути значення “a”; якщо значення “n” більше ніж нуль, викликати рекурсивну функцію з параметрами “n-1″ і “n*a”. Також, можна зауважити, що під час фази розмотування кожна рекурсивно викликана функція просто повертає значення “a”.

```
int Factorial(int no, int a)
{   // перевірка на помилковий параметр
  if (no < 0)
    return -1;
  // умова завершення
  if (0 == no || 1 == no)
    return a;
  // хвостовий рекурсивний виклик
  return Factorial(no - 1, no * a);
}
```

Це змінена версія програми з лінійною рекурсією. Ви виконуєте всі обчислення до виклику рекурсивної функції, і просто повертаєте значення отримане з цього виклику. Тут, порядок обчислення зворотній до порядку за лінійної рекурсії. У випадку лінійної рекурсії, ви спочатку множите 1 на 2; отриманий результат на 3 і так далі. З іншого боку, тут ви множите n на n-1, і тоді на n-2 доки не досягнете 0.

Хвостова рекурсія дуже корисна і часом неуникна в функціональних мовах програмування, бо деякі з них можуть не підтримувати циклічні конструкції. Тоді, зазвичай, цикли реалізуються за допомогою хвостової рекурсії. За допомогою хвостової рекурсії ви можете робити майже все, що можна зробити з циклом, але в зворотньому напрямку це часто не вірно. От дуже простий приклад, що демонструє цикл через хвостову рекурсію.

```
// реалізація циклу через хвостову рекурсію
// проста версія
```

```
void RecursiveLoop(int n)
{   // умова завершення
  if (0 == n)
    return;
  // дія
  cout << n << endl;
  // хвостовий рекурсивний виклик
  return RecursiveLoop(--n);}
```

Обопільна рекурсія

Обопільна рекурсія також відома як *непряма рекурсія*. В цьому типі рекурсії, дві або більше функції викликають одна одну циклічно. Це єдиний шлях для здійснення рекурсії в мовах, що не дозволяють вам викликати функції рекурсивно. Умова завершення в такій рекурсії може знаходитись в одній або всіх функціях.

```
bool isEven(int no)
{
  // умова завершення
  if (0 == no)
    return true;
  else
    // взаємний рекурсивний виклик
    return isOdd(no - 1);}
bool isOdd(int no)
{   // умова завершення
  if (0 == no)
    return false;
  else
    // взаємний рекурсивний виклик
    return isEven(no - 1);}
```

Це дуже примітивний приклад обопільної рекурсії. Ви знаєте, що нуль це парне число, одиниця непарне. Якщо ви хочете дізнатися чи парне число, ви можете використати ці функції; на внутрішньому рівні вони викликають одна одну і віднімають одиницю з вхідного значення доки не досягнуть виконання базової умови. Звісно, це не найліпший шлях для реалізації цього алгоритму; тут потрібна велика кількість ресурсів для визначення чи число парне. Крім того, якщо передати від'ємне число, функції будуть викликати одна одну доки не відбудеться переповнення стека.

Визначення парності числа за допомогою обопільної рекурсії не дуже добра ідея. Більш цікавим прикладом є чоловіча і жіноча послідовності.

Наведемо приклади часу виконання і часу компіляції чоловічої і жіночої функцій із використанням обопільної рекурсії.

```
int MaleSequence(int n)
{
  // умова завершення
  if (0 == n)
    return 0;
  // взаємний рекурсивний виклик
  return n - FemaleSequence(MaleSequence(n-1));}
int FemaleSequence(int n)
{   // умова завершення
  if (0 == n)
    return 1;
  // взаємний рекурсивний виклик
  return n - MaleSequence(FemaleSequence(n-1));}
```

Двійкова рекурсія

У випадку двійкової рекурсії функція викликає себе двічі, замість одного разу. Такий тип рекурсії дуже корисний при роботі з деякими структурами даних, наприклад при обході дерева в прямому, зворотньому або центрованому порядку або генерації чисел Фібоначчі і так далі.

Двійкова рекурсія – це особлива форма експонентної рекурсії, де одна функція викликає себе більш ніж один раз (у випадку двійкової рекурсій).

```
int Fib(int no)
{   // перевірка на помилковий параметр
  if (no < 1)
    return -1;
  // умова завершення
  if (1 == no || 2 == no)
    return 1;
  // подвійний рекурсивний виклик
  return Fib(no - 1) + Fib(no - 2);}
```

Ось проста реалізація послідовності Фібоначчі, що викликає рекурсивну функцію двічі. Тут ми маємо два базові випадкі; коли значення параметру на вході є 1 чи 2. Це, звісно, не найкраща реалізація послідовності Фібоначчі і ви можете перетворити її в хвостову рекурсію трошки змівнивши її. Але, перед перетворенням її в хвостову рекурсію, погляньте на версію часу компіляції двійкової рекурсії.

```
int Fib(int n, int a = 0, int b = 1)
{
  // умова завершення
  if (1 == n)
    return b;
  else
    // хвостовий рекурсивний виклик
    return Fib(n-1, b, a+b);}
```

Тут ви перетворюєте двійкову рекурсію в хвостову. Ви просто робите обчислення перед рекурсивним викликом; звідси, ви не маєте двічі робити рекурсивний виклик.

ЗМІСТ

ПЕРЕДМОВА 4

ВСТУП 5

РОЗДІЛ І Основні поняття програмування 7

Парадигми програмування. 8

Процедурне програмування. 11

Об'єктно-орієнтований підхід в програмуванні. 14

Основи алгоритмізації та програмування 17

Поняття алгоритму 17

Класифікація алгоритмів 20

Способи задання алгоритмів 22

Поняття мови програмування 29

Рівні мов програмування 29

Покоління мов програмування 30

Поняття про систему програмування 33

Історія C/C++ 35

Відмінні особливості мови С 37

Програмні засоби розробки 40

Поняття оператора 40

Поняття компілятора, інтерпретатора. 42

Поняття компілятора. 43

Інтерпретатор. Директиви препроцесора 45

Директива підключення #include 49

Директива #define.. 51

Директива #undef 53

Заголовні файли 55

Оператори мови програмування С++ 56

Алфавіт мови С++.. 56

Ключові слова.. 58

Змінні.. 63

Визначення та оголошення змінних в С++ 64

Перетворення типів.. 67

Операції, пріоритети, правила та приклади виконання. 68

РОЗДІЛ II Основи мови програмування С++ 73

Функції вводу/виводу на С++.. 73

Структура програми на С++ 73

Засоби вводу/виводу С++.. 74

Традиційна система вводу/виводу. 75

Ввід/вивід на основі потоків, реалізованих у STL......................... 81

Обробка масивів в С++.. 86

Одномірні масиви 86

Двовимірні масиви.. 89

Багатомірні масиви 90

Операції над масивами 92

Пошук в масиві.. 96

Оператори вибору 101

Порожні вирази. 102

Оголошення 102

Складені вирази 103
Обчислювальні вирази 104
Умовний вираз 105
Оператор вибору - switch 109
Оператор ?: 113
Мітки 116
Циклічні оператори 118
Цикл while 118
Цикл do-while 119
Цикл For 121
Оператори continue і break 123
Оператор goto 125
Оператор return. 126
Обробка рядків символів 128
Поняття стрічки в C++ 128
Операції над рядками 129
Тип даних string 131
Кодування символів 132
Функції 135
Основні поняття та властивості функції 135
Аргументи функції main (): argv і argc 142
Повернення з функції 144
Повернення значень 146
РОЗДІЛ III Об’єктно-орієнтований підхід в С++ 149
Основи об’єктно-орієнтованого підходу в програмуванні 149

Складність програмного забезпечення 150

Ознаки складної системи 152

Декомпозіція при проектуванні складаний систем 153

Об’єктно-орієнтована парадигма (інкапсуляція, поліморфізм, наслідування). 155

Клас і їх опис в С++ 158

Екземпляри класів або об’єкти 160

Синтаксис і семантика ООП на мові С++ 161

Специфікатори доступу (private, protected, public). 164

Конструктор і деструктор 167

Перевантаження конструкторів та деструкторів класу................ 168

Вказівник this.. 169

Особливості перезавантаження операторів в класі 171

Перезавантаження операторів 173

Перевантаження операторів за допомогою функцій-членів 174

Оператори інкремента і декремента 176

Унарні оператори !, & і ~ 179

Перевантаження оператора -> 179

Перевантаження арифметичних операторів................................. 181

Перевантаження операторів new і delete 183

Віртуальні функції. Абстрактні класи 191

Раннє та пізнє скріплення (link).. 191

Віртуальні функції 192

Виклик віртуальної функції з допомогою посилання на об'єкт базового класу. 196

Атрибут virtual успадковується 197
Віртуальні функції є ієрархічними... 198
Чисто віртуальні функції... 198
Абстрактні класи 199
Таблиці віртуальних методів (функцій) 200
Адреси, вказівники.. 203
Вказівники на об’єкти 203
Адресна арифметика, типи вказівників та операції над ними. 207
РОЗДІЛ IV Приклади проектування та реалізації типових задач засобами С++ 210
Структури та об’єднання... 210
Структура як тип та сукупність даних... 210
Об’єднання різнотипних даних. 214
Бітові поля структур та об’єднань.. 216
Рекурсія 217
Типи рекурсії 217
Лінійна рекурсія 218
Хвостова рекурсія 219
Обопільна рекурсія 221
Двійкова рекурсія.. 222
ЗМІСТ.. 224
Список використаної літератури 229
Додаток А.. 232

Список використаної літератури

1. ISO/IEC 9899:1999. Programming Languages — C. — ISO/IEC, 1999. — xiv + P. 538.

2. Абрамов С. А. Задачи по программированию / С. А. Абрамов, Г. Г. Гнездилова, Е. Н. Капустина, М. И. Селюн. — М.: Наука, 1988. - 356с.

3. Березин Б.И. Начальный курс С и С++../ Б.И. Березин, С.Б. Березин/ — М.: ДИАЛОГ-МИФИ, 1996.

4. Бертран Мейер. Объектно-ориентированное конструирование программных систем, 2-е издание. Русская редакция, 2005

5. Бондарев В.М., Рублинецкий В.И., Качко Е.Г. Основы программирования. —Харьков: Фолио, Ростов н/Д: Феникс, 1997.

6. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1981.

7. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989.

8. Генри С. Уоррен, мл. Алгоритмические трюки для программистов. Вильямс, 2007

9. Герб Саттер и Андрей Александреску. Стандарты программирования на С++. Вильямс, 2008

10. Гради Буч. Объектно-ориентированный анализ и проектирование, 3-е издание. Вильямс, 2008

11. Дагене В.А., Григас Г. К., Аугутис К.Ф. 100 задач по программированию. — М.: Просвещение, 1993.

12. Джеффри Рихтер, Кристоф Назарр. Windows via C/C++. 5-е издание. Питер, Русская Редакция, 2009

13. Дональд Э. Кнут. Искусство программирования. Том 1. Основные алгоритмы. 3-е издание. Вильямс, 2008

14. Дьюхэрст С. Скользкие места С++. Как избежать проблем при проектировании и компиляции ваших программ. - М.: ДМК Пресс, 2006.- 264с

15. Дэвид Вандевурд, Николай М. Джосаттис. Шаблоны C++. Справочник разработчика. Вильямс, 2008

16. Иванова Г.С. Основы программирования: Учебник для вузов. - М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. - 416 с.

17. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. — М.: Финансы и статистика, 1992.

18. Коплиен Дж. Программирование на C++. Классика CS. - СПб.: Питер, 2005.-479с

19. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. - М.: МЦНМО, 2009. - 960с.

20. Культин Н. Б. C/C++ в задачах и примерах. — СПб.: БХВ-Петербург, 2001. - 288 с.

21. Левитин В. Алгоритмы: введение в разработку и анализ. : Пер. с англ. - М. : Вильяме, 2006.

22. Липпман С. Б. Основы программирования на C++: Пер. с англ. — М.: Вильямс, 2002. — 256 с.

23. Липпман С. Б., Лажойе Ж. Язык программирования C++. Вводный курс: Пер. с англ. — 3-е изд. — М.: ДМК, 2001. — 1104 с.

24. Мейерс С. Эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. - М.: ДМК Пресс; СПб.: Питер, 2006.-296с

25. Окулов С.М. Основы программирования. - 3-е изд. - М.: БИНОМ. Лаборатория знаний, 2006. - 440с.

26. Подбельский В.В. Язык СИ++. 5-е издание. - М.: Финансы и статистика, 2001.-560с

27. Прата С. Язык программирования C++. Лекции и упражнения. 5-е издание.- .: Издательский дом "Вильямс", 2007.-1184с

28. Сабуров С., Язык программирования C и C++, М.: Бук-пресс, 2006. -647 с.

29. Саттер Г. Новые сложные задачи на C++ (серия C++ in Depth). - М.: Издательский дом "Вильямс", 2005.-272с

30. Стефенс Д.Р., Диггинс К., Турканис Д., Когсуэлл Д. C++. Сборник рецептов. - М.: КУДИЦ ПРЕСС, 2007.-624с

31. Страуструп Б. Дизайн и эволюция языка C++. Объектно-ориентированный язык программирования: Пер. с англ. — М.: ДМК пресс, Питер, 2006. — 448 с.

32. Страуструп Б. Язык программирования C++: Пер. с англ. — 3-е спец. изд. — М.: Бином, 2003. — 1104 с.

33. Хеннер Е.К. Информатика: Учеб. пособие для студ. пед. вузов / Под ред. Е. К. Хеннера. — М.: Изд. центр «Академия», 1999.

34. Эккель Б. Философия C++. Введение в стандартный C++: Пер. с англ. — 2-е изд. — СПб.: Питер, 2004. — 572 с.

35. Эккель Б., Эллисон Ч. Философия C++. Практическое программирование: Пер. с англ. — СПб.: Питер, 2004. — 608 с.

36. Элджер Дж. C++: Библиотека программиста. - СПб.: Питер, 1999.-320с

37. Эрик Гамма, Ральф Джонсон, Ричард Хелм, Джон Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Питер, 2007

Додаток А

Завдвння №1.

Розробити та графічно представити блок-схему алгоритму згідно індивідуального завдання:

1. Алгоритм знаходження суми елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

2. Алгоритм знаходження максимального елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

3. Алгоритм знаходження мінімального елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

4. Алгоритм знаходження периметру квадрата. Довжини сторін вводиться з клавіатури.

5. Алгоритм знаходження суми від'ємних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

6. Алгоритм знаходження мінімального від'ємного елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

7. Алгоритм знаходження максимального непарного елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

8. Алгоритм знаходження суми непарних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

9. Алгоритм знаходження площі кола. Радіус вводиться з клавіатури.

10. Алгоритм знаходження добутку парних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

11. Алгоритм знаходження суми додатних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

12. Алгоритм знаходження максимального парного елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

13. Алгоритм знаходження суми елементів кратних 5 одномірної матриці. Розмірність матриці вводиться з клавіатури.

14. Алгоритм знаходження мінімального від’ємного елементу кратного 3 одномірної матриці. Розмірність матриці вводиться з клавіатури.

15. Алгоритм знаходження середнього елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

16. Алгоритм знаходження мінімального парного елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

17. Алгоритм знаходження площі квадрата. Довжини сторін вводиться з клавіатури.

18. Алгоритм знаходження периметру трапеції. Розміри кутів та довжин сторін вводиться з клавіатури.

19. Алгоритм знаходження суми парних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

20. Алгоритм знаходження мінімального непарного елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

21. Алгоритм знаходження площі трапеції. Розміри кутів та довжин сторін вводиться з клавіатури.

22. Алгоритм сортування масиву по зростанню на основі алгоритму бульбашки.

23. Алгоритм знаходження периметра кола. Радіус вводиться.

24. Алгоритм знаходження добутку елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

25. Алгоритм знаходження добутку непарних елементів одномірної матриці. Розмірність матриці вводиться з клавіатури.

26. Алгоритм знаходження мінімального додатнього елементу одномірної матриці. Розмірність матриці вводиться з клавіатури.

27. Алгоритм знаходження периметра прямокутника. Довжини сторін вводиться з клавіатури.

28. Алгоритм сортування масиву по спаданню на основі алгоритму бульбашки.

29. Алгоритм обчислення кількості парних чисел одномірного масиву.

Завдання 2

Розробити програму для розв'язку математичного виразу

N вар	Формули для *y*	Параметри
1	1) $t1=\frac{ax}{y}+\frac{b}{y^2}\lg(yx+c)$ 2) $y=\begin{cases}2u+\sqrt[3]{a\cdot c}, & u\le a-c^2\\ \ln(a\cdot c)+Cos^2u, & u>a-c^2\end{cases}$	$a=3.6;$ $c=2.1;$ $u=Sinx$
2	1) $t1=\frac{1}{c}[\frac{b}{a}\ln(ax+b)+\frac{d}{y}\ln(yx+d)]$ 2) $y=\begin{cases}abxCos^2zx, & x<3{,}5a\\ (a+bx)^2-Ln(zx), & 3{,}5a\le x\le b\\ \sqrt{(a+bx-zx^2)}, & x>b\end{cases}$	$a=0.4;$ $b=2.3;$ $z=e^{2x}$
3	1) $t1=\frac{1}{c}(\frac{1}{ax+b}+\frac{y}{c}\ln(\frac{yx+a}{ax+b})$ 2) $y=\begin{cases}Sin(bm+Cosnx), & \lvert bm\rvert>n^2\\ Cos(bm-Sinx), & \lvert bm\rvert<n^2\\ \sqrt{e^{\lvert Lnx\rvert}+\sqrt{\lvert bmx\rvert}}, & \lvert bm\rvert=n^2\end{cases}$	$b=-1.6;$ $m=0.9;$ $n=-1.4$
4	1) $t1=\frac{b}{(a-b)(x+b)}-\frac{a}{(a-b)^2}\ln(\frac{a+x}{b+x})$ 2) $y=\begin{cases}aSin^2x+bCoszx, & x<-Ln(a)\\ a^b-Cos^3(a+zx), & -Ln(a)<x\le b\\ \sqrt{2.5a^3+(b-zx^2)^6}, & x>b\end{cases}$	$a=0.2;$ $b=0.5;$ $z=e^{ax}$

N вар	Формули для *y*	Параметри
5	1) $t1=\frac{1}{a}(\frac{-1}{(n-2)x^{n-2}}-\frac{b}{(n-1)x^{n-1}})$ 2) $y=\begin{cases} Sin\left(e^{a+b}\right)+x^2, & e^{a+b}>e^c \\ arctg(abc)+\sqrt[3]{x}, & e^{a+b}=e^c \\ Cos\left(\sqrt{\lvert x+abc\rvert}\right), & e^{a+b}<e^c \end{cases}$	$a=-4{,}2;$ $b=5.3;$ $c=1.5$
6	1) $t1=\frac{-1}{(a-b)^2}(\frac{1}{a+x}+\frac{1}{1+x})+\frac{2}{(a-b)^3}\ln\frac{a+x}{b+a}$ 2) $y=\begin{cases} 2.8Sin^2 ax-bx^3z, & x<a \\ zCos(ax+b)^2+Ln(z), & a\le x\le b^2 \\ e^{2{,}5ax}+zabx, & x>b^2 \end{cases}$	$a=-5;$ $b=2.5;$ $z=Ln\lvert bx^3\rvert$
7	1) $t1=\frac{1}{a^3}(\ln x+\frac{2b}{x}-\frac{b^2}{2x^2})$ 2) $y=\begin{cases} xe^a+e^{\lvert bc\rvert}, & \lvert 1-b^2\rvert=a+c \\ Sin^2ax+Cosbc, & \lvert 1-b^2\rvert>a+c \\ \sqrt{ab^4+\sqrt{cx},} & \lvert 1-b^2\rvert<a+c \end{cases}$	$a=3.2;$ $b=-0.7;$ $c=2.2$
8	1) $t1=\frac{1}{a^4}(\frac{x^3}{3}-3bx+3b^2\ln x+\frac{b^3}{x})$ 2) $y=\begin{cases} Ln\lvert mx+n\rvert, & k^2>m+n \\ e^{Ln\lvert mx-n\rvert}, & k^2=m+n \\ \sqrt[3]{k^2+Cos^2x}, & k^2<m+n \end{cases}$	$k=3.1;$ $m=5.15;$ $n=-0.5$
9	1) $t1=\frac{1}{b^2}(\ln\frac{y}{x}+\frac{ax}{y})$ 2) $y=\begin{cases} aSin^{2.5}x+bCos(zx+a), & x<a^3 \\ (a+bx)^2-Sin(a+zx), & a^3\le x\le b \\ \sqrt{(Sin(a+bx+z)-x)}, & x>b \end{cases}$	$a=-2.2;$ $b=7.2;$ $z=e^x$

N вар	Формули для *y*	Параметри
10	1) $t1=\frac{1}{b^3}(\ln\frac{y}{x}-\frac{a^2x^2}{2y^2})$ 2) $y=\begin{cases}\sqrt[3]{b^2+\sqrt{\lvert x+c\rvert}}, & Lga+Lgb<Lgc\\ Cos(x-a+b-c), & Lga+Lgb=Lgc\\ Sin(x+a-b+c), & Lga+Lgb>Lgc\end{cases}$	$a=101;$ $b=9;$ $c=1112$
11	1) $t1=a(\frac{1}{b^2y}+\frac{1}{ab^2x}-\frac{2}{b^3}\ln\frac{y}{x})$ 2) $y=\begin{cases}e^{ax}-3.5Cos^2(z+bx), & x\le a\\ a+Ln\lvert a+bx\rvert-2x, & a<x\le b^{3.5}\\ a+Cos^{3.5}(a+bxz), & x>b^{3.5}\end{cases}$	$a=-1;$ $b=3.4;$ $z=tgbx$
12	1) $t1=\frac{1}{b^3}(a^2\ln\frac{y}{x}+\frac{2ax}{y}+\frac{y^2}{2x^2})$ 2) $y=\begin{cases}Ln(Lg\lvert kx+mn\rvert), & 3k>\lvert m+n\rvert\\ Sinkmx+\sqrt{\lvert nx\rvert}, & 3k=\lvert m+n\rvert\\ e^{kCosx}+e^{m+n}, & 3k<\lvert m+n\rvert\end{cases}$	$k=4;$ $m=-14.7;$ $n=-0.7$
13	1) $t1=\frac{1}{b^4}(3a^3\ln\frac{y}{x}+\frac{a^2x}{y}-\frac{3ay}{x})$ 2) $y=\begin{cases}x^2e^{2k}+Ln\lvert rx\rvert, & Cosk=Cosrs\\ \sqrt[3]{x^2}+\lvert\sqrt{k+rsx}\rvert, & Cosk>Cosrs\\ arctg(kx+rs), & Cosk<Cosrs\end{cases}$	$k=1.33;$ $r=0.85;$ $s=3,5$
14	1) $t1=\frac{1}{2(n-1)x^{n-1}}+\frac{a}{2nx^n})$ 2) $y=\begin{cases}2.5b^2+ax-4.5Cosxz, & x\le 5a\\ (a^2-5.4x)^3+Ln(xz), & x>b\\ \sqrt{6.5b^2+(a-x^3z)}, & 5a<x\le b\end{cases}$	$a=0.5;$ $b=4.5;$ $z=e^{ax}$

N вар	Формули для *y*	Параметри
15	1) $t1=\frac{1}{4a^2x^2}+\frac{1}{2a^4x}+\frac{1}{2a^6}\ln\frac{y^2}{x})$ 2) $y=\begin{cases}\sqrt{ax-Cos^2b^3x+5.1c^2}, & \lvert 1-b^2\rvert=a+c\\ e^{0.007x}+Ln\lvert b^5C''sx\rvert, & \lvert 1-b^2\rvert>a+c\\ Cos^2b^3x^2+Ln\lvert bx-a^2\rvert, & \lvert 1-b^2\rvert<a+c\end{cases}$	$a=3.5$; $b=-0.73$; $c=2.5$
16	1) $t1=\frac{1}{a^2c^2+b^2}[c\ln(b+cx)-\frac{c}{2}\ln y]$ 2) $y=\begin{cases}3.5Sin(bx+z)-e^{3.5a}, & x\le a\\ Ln(a+b^3x)+a, & a<x\le b^{2.5}\\ Cos^2(a^b+xz)+a^2, & x>b^{2.5}\end{cases}$	$a=0.2$; $b=0.5$; $z=e^{2.5ax}$
17	1) $t1=\frac{1}{3a^3y}+\frac{1}{3a^6}\ln\frac{x^3}{y}$ 2) $y=\begin{cases}a+Sinbx+Cosx^2, & x<a\\ \sqrt{a+bx}+Sinzx, & a<x<Lnb\\ Ln(a+bx+z), & x>Lnb\end{cases}$	1.$a=0.2$; $b=0.75$; $z=Ln(bx)$
18	1) $t1=\frac{1}{6a}\ln\frac{a^2-ax+x^2}{(a+x)^2}+\frac{1}{a\sqrt{3}}$ 2) $y=\begin{cases}(3.5a-7.3bx+Sinzx), & x<-Ln\lvert a\rvert\\ a^b-Cos^3(a+zx), & -Ln\lvert a\rvert\le x<b\\ \sqrt{\lvert tga-x\rvert}-x^2, & x>b\end{cases}$	$a=6$; $b=3.2$; $z=e^{1.5ax}$
19	1) $t1=\frac{1}{6a^3}\ln\frac{a+x}{a-x}+\frac{1}{2a^3}$ 2) $y=\begin{cases}cSinb^2x+bLn(cx+a), & x\le a\\ a+Ln(bx)-Sin^2(a+cx), & a\le x<b\\ \sqrt{Cos(a+bx)+cx^2}, & x>b\end{cases}$	$a=2.2$; $b=2.4$; $c=Ln\lvert bx\rvert$

N вар	Формули для *y*	Параметри
20	1) $t1=\frac{1}{4a^3}\ln\frac{a^2+x^2}{a^2-x^2}$ 2) $y=\begin{cases} e^{ax}+fCos^{3.5}bx, & x\le a \\ a+Cos^2bx-Ln(fx), & a\le x<b^2 \\ Cos^2(a+bfx), & x>b^2 \end{cases}$	$a=0.8;$ $b=2.4;$ $f=e^{1.5ax}$
21	1) $t1=\frac{2\sqrt{x}}{3b^2}-\frac{2a^2\sqrt{x}}{b^4}+\frac{2a^3}{b^5}y$ 2) $y=\begin{cases} aCos^2x+bSinzx, & x\le a \\ atg(ax+z)+Sin^2bx, & a<x\le 4.5b \\ Ln(ax-b)+z^2, & x>4.5b \end{cases}$	$a=4.5;$ $b=8.4;$ $z=tgbx^2$
22	1) $t1=\frac{2}{a^2y\sqrt{x}}+\frac{3b^2\sqrt{x}}{a^4y}$ 2) $y=\begin{cases} a+bx+Sin^2zx^{3.5}, & x<a \\ a+Ln\left\|ab-zx^3\right\|+Lnx, & a\le x\le b^2 \\ \sqrt{\left\|a+tgzx\right\|}+bSinx, & x>b^2 \end{cases}$	$a=0.3;$ $b=0.9;$ $z=Sinx^2$
23	1) $t1=\frac{1}{2a\sqrt{2}}\ln\frac{x+a\sqrt{2x+a^2}}{x-a\sqrt{2x+a^2}}$ 2) $y=\begin{cases} Ln\left\|bzx\right\|+za^{2.5}, & a^5<x\le b \\ ax^2+bz^a+Sin^2zx, & x>b \\ Cos(ax+b)+Ln\left\|zx\right\|, & x\le a^5 \end{cases}$	$a=1.5;$ $b=6.4;$ $z=Ln\left\|bx^3\right\|$
24	1) $t1=\frac{2(3ax-2b)\sqrt{x^3}}{15a^2}$ 2) $y=\begin{cases} xe^x+(z+7.7abx), & x<a \\ tg(ax+z)+Cos^2bx, & a\le x\le b^2 \\ Ln(Sin(a+bx+zx^2)), & x>b^2 \end{cases}$	$a=3.7;$ $b=8.7;$ $z=tgbx$

N вар	Формули для *y*	Параметри
25	1) $t1=\frac{1}{\sqrt{b}}\ln\frac{\sqrt{x}-\sqrt{b}}{\sqrt{x}+\sqrt{b}}$ 2) $y=\begin{cases} a+z\cos^2 bx^3, & x<a \\ a+Sin^2 b^2+Ln(zx), & a<x<b \\ \sqrt[3]{0.3b+\sqrt{\lvert(a-z^2-\cos x)\rvert}}, & x>b \end{cases}$	$a=1.5;$ $b=5.7;$ $z=Ln\lvert tg\,bx\rvert$
26	1) $t1=\frac{2}{35a^2}(5\sqrt{x^5}-7b\sqrt{x^7})$ 2) $y=\begin{cases} c\cdot a^2+b\cdot\cos^3(a\cdot x), & a>x \\ \ln\lvert a\cdot c\cdot x\rvert+\sqrt{b}, & a\le xb \\ \cos(a+b\cdot x\cdot c+a\cdot c^2), & x>b \end{cases}$	$a=0.5;$ $b=6;$ $c=-5$
27	1) $t1=\frac{2}{a^3}(\frac{\sqrt{x^3}}{3}-2b\sqrt{x}-\frac{b^2}{\sqrt{x}})$ 2) $y=\begin{cases} a^2+\sqrt{b^4+1.7}, & \lvert Cosx\rvert<0.2 \\ arctg(2^x-\lvert p\rvert), & \lvert Cosx\rvert=0.2 \\ \sqrt[3]{Ln\lvert a\rvert+4.3}, & \lvert Cosx\rvert>0.2 \end{cases}$	$a=0.5;$ $b=1.5;$ $p=-4$
28	1) $t1=\frac{\sqrt{y}}{2x^2}+\frac{1}{2a}\ln\frac{a+\sqrt{y}}{x})$ 2) $y=\begin{cases} a\cdot\sin^2 x+\ln\lvert b\cdot c\cdot x^2\rvert, & x<\frac{\pi}{2} \\ \sin^2(x^3+3a+\sqrt{b\cdot c}), & x=\frac{\pi}{2} \\ \cos(x/a)+b\sqrt{c}, & x>\frac{\pi}{2} \end{cases}$	$a=5;$ $b=5;$ $c=10$
29	1) $t1=\frac{2}{\sqrt{ay}}\ln(\sqrt{ax}+\sqrt{by})$ 2) $y=\begin{cases} a\cdot x-b^2+a\cdot\cos(c\cdot x), & x<a \\ c(a+b\cdot x)+\cos(b\cdot x), & x=a \\ a\cdot x^b+e^{c\cdot x}+\sin^2 x, & x>b \end{cases}$	$a=1{,}4$ $b=1{,}7$ $c=3{,}7$

Завдання 3.

Розробити програму для розв'язку задачі з матрицями згідно номера:

№	Варіант завдання
1.	Визначити максимальний елемент одновимірної матриці $M[n]$, де $n[10;50]$ та значення вводиться з клавіатури.
2.	Визначити середній елемент одновимірної матриці $M[n]$, де $n[10;50]$ та значення вводиться з клавіатури.
3.	Визначити мінімальний елемент одновимірної матриці $M[n]$, де $n[10;50]$ та значення вводиться з клавіатури.
4.	Обчислити добуток матриць A і B. Вхідні дані: цілі, додатні числа n, масиви чисел A і B розмірності nxn
5.	Дана матриці A розмірності nxm, знайти суму чисел кратним 2.
6.	Дана матриця A = (a_{ij}) розмірністю m на n і матриця B = (b_{ij}) розмірністю m на n. Сформувати матрицю C = (c_{ij}) розмірністю m на n: $C_{ij} = a_{ij} - b_{ij}$.
7.	Дана матриці A розмірності nxm, знайти суму непарних чисел.
8.	Заповнити матрицю випадковими числами. На побічній діагоналі розмістити суми елементів, які лежать на тому ж рядку і стовпці
9.	Обчислити скалярний добуток вектора A на вектор B. Вхідні дані: ціле позитивне число n, масиви чисел A і B розмірності n, ціле позитивне число K від 1 до n / 2.
10.	Дана матриці A розмірності nxm, знайти суму парних чисел кратним 3.
11.	Знайти стовбець і рядок з мінімальними сумами в матриці A. Вхідні дані: цілі позитивні числа n і к, масив чисел A розмірності nxк
12.	Визначити чи збігається хоча б одна пара - сума i-го рядка і j-го стовпця матриці A. Вхідні дані: ціле додатнє число n, масив чисел A розмірності nxn.

13.	Дана матриця А розмірності nxn. Вхідні числа повинні бути менші 0. Знайти максимальний відємний елемент.
14.	У матриці А знайти рядок з максимальним добутком. Вхідні дані: ціле позитивне число n, масив чисел А nxn.
15.	Дана матриця А = (a_{ij}) розмірністю m на n і матриця В = (b_{ij}) розмірністю m на n. Сформувати матрицю С = (c_{ij}) розмірністю m на n: C_{ij} = a_{ij} - b_{ij} . Знайти в матриці С = (c_{ij}) рядок з максимальним середнім арифметичним значенням її елементів.
16.	Дана матриці А розмірності nxm, знайти суму парних чисел.
17.	Визначити, чи рівні між собою суми двох головних діагоналей в матриці А. Вхідні дані: ціле позитивне число n, масив чисел А розмірності n.
18.	Дана матриці А розмірності nxm, знайти суму чисел кратним 5.
19.	Знайти максимальний елемент у матриці А. Вхідні дані: цілі додатні числа n і к, масив чисел А розмірності n і к.
20.	Дано матрицю А, розмірності nxm. Вивести найбільш повторюване число матриці на екран.
21.	Визначити чи є матриця А магічним квадратом. Вхідні дані: ціле позитивне число n, масив чисел А розмірності nxn. (Матриця є магічним квадратом, коли рівні між собою суми всіх рядків і суми всіх стовпців.)
22.	Дано матрицю А розмірності nxm. Вхідні дані ціле число від 1 до 26. Вивести матрицю на екран. При натисканні клавіши enter, вивести матрицю, яка має замінюватиме 1 на літеру A, 2 на літеру B, 3 на літеру C і так далі.
23.	Дана матриця А, розмірності nxn, вхідні числа повинні бути менші 0. Знайти суму елементів по діагоналях.

24.	Дана матриця А розмірності nxn. Визначити кількість елементів рівних 0.
25.	Дана матриця А, розмірності nxm. Вхідні дані цілі числа в діапазоні [-20;20]. Замінити від'ємні числа будь яким літерним символом, що вводиться з клавіатури.
26.	Дана матриця А розмірності nxn. Обрахувати суму кожного рядка і вивести максимальну суму по рядках.
27.	Дана матриця А, розмірність nxn. Знайти суму парних чисел по діагоналі.
28.	Дана матриця А, розмірності nxn. Вхідні дані: ціле число діапазоном [-10,10]. Замінити всі від'ємні елементи на додатні та навпаки.
29.	Дана матриця А, розмірності nxn. Вхідні дані будь яке ціле додатнє число. Вивести цю матрицю. Далі посортувати її по зростанню цифр по рядках.
30.	Дана матриця А розмірності nxn. Вхідними даними є літери від A до Z. Програма повинна замінювати голосні літери цифрами, при тому що a=1,e=2,i=3 і т.д. Вивести вихідну матрицю на екран.

Завдання 4

Розробити програму для розв'язку задачі з стрічками.

№ завдання	Варіант завдання
1	В стрічці, що вводиться з клавіатури, знайти максимальне за довжиною слово, та перевести всі букви даного слова в вверхній регістр. Отриману нову стрічку вивести на екран.
2	В стрічці, що вводиться з клавіатури, порахувати кількість розділових знаків. Отримані результати вивести вивести на екран у форматі: "." - n раз, "," - m раз, і т.д. Де n,m — кількість входжень розділового знаку в стрічку.
3	Задана стрічка А, яка містить декілька слів кількістю букв не менше 7. Підрахувати кількість голосних букв.
4	Задана стрічка А, яка містить великі і малі літери. Замінити великі літери на малі, а малі на великі. Отриману стрічку вивести.
5	Дано стрічку А, яка містить не менше 10 слів. Вивести найдовше слово і підрахувати у ньому скільки є приголосних літер.
6	Дана стрічка А, яка містить не менше 12 слів. Посортувати слова по зростанню(по кількості букв). Вивести кінцеву стрічку.
7	Задана стрічка А, яка містить не менше 15 слів. Обрахувати кількість приголосних літер.
8	Заданий стрічка S, що містить не менше 10 слів. Необхідно знайти слово, що містить максимальну кількість входжень символу а. Вхідні дані: рядок S, символ а.
9	Задана стрічка А, яка містить не менше 10 слів. Обрахувати кількість голосних літер.
10	Дана стічка А, яка містить не менше 15 слів. Замінити кожен пробіл на символ, який вводиться з клавіатури. Вивести кінцеву стрічку.

11	Задана стрічка А, яка містить слово кількістю букв не менше 50. Підрахувати кількість приголосних букв.
12	Дано стрічку А, яка містить не менше 4-ох слів. Необхідно знайти символ, який найчастіше зустрічається. Вивести цей символ і слово, де воно найбільш зустрічається.
13	Дано стрічку А, яка містить не менше трьох слів. Необхідно замінювати символи кирилиці на символи латиниці. Вивести отриману стрічку на екран.
14	Задається стрічка шлях до деякого файлу. Розбити дану стрічку на диск, шлях ім'я і тип файлу.
15	Задана стрічка, яка містить набір чисел. Посортувати цисла по зростанню.
16	Відсортувати введену з клавіатури стрічку букв кирилиці за алфавітом.
17	Задану стрічку, яка містить імя, прізвище і по – батькові. Перетворити задану стрічку у вигляді «Прізвище ініціали». Вивести кінцеву стрічку на екран.
18	Задана стрічка, яка містить назву місяця. Вивести кінцеву стрічку яка має заміняти назву місяця на число , наприклад, січень - 01, серпень - 08.
19	Розробити програму "склеювання" слів у стрічки. Кількість слів мінімум 20, що вводяться з клавіатури. При повторному введені слова, воно повинно нехтуватись.
20	Розробити програму переводу кириличного тексту на латиницю. Наприклад: "а" - "a", "б" - "b", "ж" - "zh".
21	Дана стрічка, яка повинна містити число від 0 до 15. Вивести на екран кінцеву стрічку, яка має заміняти число в буквенному форматі, наприклад, 5- п'ять, 0 – нуль.

22	Написати програму перекодування римських цифр в арабські. Наприклад “I”-”1”, “V”-”5”,“X”-”10”. Максимальне число 1 000 000.
23	Написати прграму для перевертання вхідних стрічок. Перекодування проводити за принципом перший символ замінює останній, другий на передостанній і т.д.
24	Написати програму для конвертації одиничної метричної системи С з символьного позначення в стрічкове. Наприклад “см” - “сантиметри”.
25	Написати програму для розрізання вхідної стрічки на підстрічки, довжина підстрічки задається з клавіатури.
26	Написати програму перекодування арабські цифр в римських. Наприклад “1”-”I”, “5”-”V”,“10”-”X”. Максимальне число 1 000 000.
27	Задана стрічка, яка містить число від 01 до 12. Вивести кінцеву стрічку яка має заміняти число на назву місяця, наприклад, 1- січень, 8 - серпень.
28	Розробити програму для підрахунку розділових знаків в вхідній стрічці. Досжина стрічки не менше 10 слів.
29	Дана стрічка, яка повинна містити назву цифр від 1 до 20. Вивести на екран кінцеву стрічку, яка має заміняти назву цифри в саму цифру, наприклад, п’ять – 5,нуль - 0.
30	Написати програму, що буде преводити в верхній регістр зарезервовані слова мови програмування С++. В вхідній стрічці не менше 15 слів.

Завдання 5

Розробити програму для розв'язку задачі з файлами.

№ завдання	Варіант завдання
1	Записати у файл список студентів навчальної групи. Вказати наступні данні: ім'я, фамілія, по-батькові, рік народження, середній бал. Інформацію подати у вигляді таблиці.
2	Записати у файл інформацію про робочу станцію. Вказати наступні данні: тип процесора, тип відеокарти, об'єм ОЗУ, розмір жорсткого диску, наявність мережевого обладнання. Інформацію подати у вигляді таблиці.
3	Записати у файл список студентів навчальної групи. Вказати наступні дані: ім'я, фамілія, по-батькові, рік народження, середній бал. Інформацію подати у вигляді таблиці у алфавітному порядку
4	Записати у файл інформацію про периферійні пристрої. Вказати наступні дані: мишка, екран, принтер, сканер. Інформацію подати у вигляді таблиці
5	Записати у файл інформацію про відеокарту. Вказати наступні дані: фірма виготовлення, пам'ять, пропускна здатність, частота, наявність вентиляторів. Вивести на екран відеокарти у яких пам'ять більше 512мб у вигляді таблиці
6	Записати у файл інформацію мережеве обладнання. Вказати наступні дані: швидкість, марку виготовлення, пропускна здатність, тип з'єднання. Вивести на екран список мережевих карт відсортованих у порядку збільшення пропускної здатності.
7	Записати у файл інформацію про сервер. Вказати наступні дані: розмір жорсткого диску, об'єм ОЗУ, тип процесора. Інформацію подати у вигляді списку. Забезпечити можливість вибору серверних станцій за об'ємом постійної пам'яті.

8	Записати у файл список викладачів навчальної групи. Вказати наступні дані: ім’я, фамілія, науковий ступінь, стаж роботи. Вивести на екран Фамілії викладачів у алфавітному порядку у вигляді таблиці.
9	Записати у файл інформацію про тварин. Вказати наступні дані: вид, тривалість проживання, середня маса, вид локації проживання, тип харчування. Вивести на екран середню масу і тип харчування у вигляді таблиці.
10	Записати у файл інформацію про флеш накопичувачі. Вказати наступні дані: об’єм, пропускна здатність, швидкість передавання інформації, надійність. Інформацію подати у вигляді списків
11	Записати у файл інформацію про маршрутки Тернополя. Вказати наступні дані: марка маршрутки, вартість проїзду, місткість пасажирів, квота на пільгові місця. Інформацію подати у вигляді таблиці.
12	Записати у файл список викладачів вашої кафедри. Вказати наступні дані, ім’я, фамілію, науковий ступіть, вік, назва предмета. Інформацію подати у вигляді таблиці у алфавітному порядку
13	Записати у файл список ПК у вашій аудиторії. Вказати наступні дані: порядковий номер ПК, наявність мережевого обладнання, тип процесора, розмір жорсткого диску. Інформацію подати у вигляді таблиці.
14	Запасати у файл список продукції Microsoft. Вказати наступні дані: тип продукції, ціна, дата випуску, термін експлуатації. Інформацію подати у вигляді списків
15	Записати у файл інформацію про мережеві кабелі. Вказати наступні дані: ціна, швидкість передачі інформації, матеріал, тип кабеля. Інформацію подати у вигляді таблиці

16	Записати у файл список провайдерів вашого міста. Вказати наступні дані: назва, використання кабелів, ціна підключення, швидкість виконання замовлення. Інформацію подати у вигляді таблиці
17	Записати у файл інформацію про wifi-роутери. Вказати наступні дані: фірма, швидкість, область охоплення, швидкість передачі інформації. Вивести на екран область охоплення і швидкість у зростаючому порядку у вигляді таблиці
18	Записати у файл інформацію про мобільний телефон. Вказати наступні дані: пам'ять, підтримка карт пам'яті, об'єм ОЗУ, розмір екрану, екран сенсорний/не сенсорний. Вивести на екран усі телефони з сенсорним екраном і яких об'єм ОЗУ більше 512мб у вигляді таблиці.
19	Записати у файл інформацію про мишки. Вказати наступні дані: провідна\безпровідна, з колесом\без колеса, кількість клавіш, dpi. Вивести на екран безпровідні мишки з dpi більше 500 у вигляді таблиці.
20	Записати у файл інформацію про дисководи. Вказати наступні дані: CD\DVD\Blu-Ray, привід, швидкість, зчитування\ записування. Вивести на екран дисководи які підтримують DVD і мають властивість зчитування і записування у вигляді таблиці
21	Записати у файл інформацію про клавіатуру. Вказати наступні дані: тип підключення, кількість клавіш, сенсорна\не сенсорна, дротова\бездротова, колір, вага, ціна. Вивести на екран клавіатури чорні з бездротовим з'єднанням у вигляді таблиці.
22	Записати у файл інформацію про вентилятори. Вказаним наступні дані: розмір, швидкість обертання, вдування\видування, тип підключення. Вивести на екран вентилятори зі швидкістю обертання менше 10\сек у вигляді таблиці.

23	Записати у файл інфромацію про автомобілі. Вказати наступні дані: максимальна швидкість, передній\задній привід, об’єм двигуна, марка, вмістіть людей, вольтаж акумулятора. Вивести на екран автомобілі з максимальною швидкістю більше 240км\год
24	Записати у файл інформацію про екрани. Вказати наступні дані: розмір, розширення, бітність зображення, тип кристалів, кількість кольорів. Вивести на екран екрани з розширенням більше 1280\1024 і вказати їх тип кристалів у вигляді таблиці
25	Записати у файл інформацію про жорсткий диск. Вказати наступні дані: розмір, швидкість обертання, тип, форм фактор. Вивести на екран жорсткі диски з максимальною швидкістю обертання і вказати їх тип у вигляді таблиці
26	Записати у файл інформацію про принтер. Вказати наступні дані: швидкість, тип, спосіб підключення. Вивести на екран інформацію у вигляді таблиці
27	Записати у файл інформацію про свічі. Вказати наступні дані: кількість портів, присутність джерела віфі, пропускна здатність, тип живлення. Інформацію подати у вигляді таблиці.
28	Записати у файл інформацію про оперативну пам'ять. Вказати наступні дані: тип, об’єм, пропускна здатність, форма, DDR\DDR2\DDR3. Інформацію подати у вигляді таблиці
29	Записати у файл інформацію про файлову систему. Вказати наступні дані: тип, максимальна кількість секторів, максимальний об’єм сектора, застосування ос. Вивести на екран тип і максимальну кількість серверів у алфавітному порядку.
30	Записати у файл інформацію про веб-камеру. Вказати наступні дані: якість зображення, тип підключення, ціна, спосіб прикріплення, колір, розмір. Вивести на екран якість зображення, тип підключення і ціну в зростаючому порядку

Завдання 6

Розробити програму для розв’язку задачі з файловою системою.

№ завдання	Варіант завдання
1	Пошуку файлу в заданій директорії.
2	Визначення кількості файлів у директорії. Шлях та назва директорії. Задаються за допомогою клавіатури.
3	Організувати пошук заданого файлу у вказаній директорії і її піддиректорії.
4	Розробити функцію підрахунку сумарного розміру всіх файлів заданого шаблону у вказаній директорії.
5	Розробити функцію підрахунку кількості піддиректорій в заданій директорії.
6	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність слів. Функція повинна знайти та вивести на екран всі слова, що містять парну кількість символів.
7	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить повне ім'я файлу (шлях та ім'я). Функція повинна визначати чотири параметри оточення - буква диска, шлях, ім'я файлу і тип файлу.
8	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить повне математичний вираз (формат $X = A + B$, де A, B- деякі цілі числа). Функція повинна обчислити та вивести на екран значення виразу.
9	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність символів. Функція повинна знайти та вивести на екран всі можливі комбінації утворені за допомогою введеного набору символів.

10	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність чисел (від 0 до 9). Функція повинна знайти та вивести на екран всі можливі числа, утворені за допомогою введеного набору чисел.
11	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність чисел (від 0 до 9). Функція повинна знайти та вивести на екран всі можливі числа, утворені за допомогою введеного набору чисел які будуть лежати в межах від 1 до 100.
12	Розробити функцію копіювання каталогів
13	Розробити функцію копіювання файлів за деяким шаблоном
14	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність слів. Функція повинна знайти та вивести на екран кількість повторень слово S, значення якого користувач вводить з клавіатури.
15	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність чисел (від 0 до 9). Функція повинна знайти та вивести на екран всі можливі числа, утворені за допомогою введеного набору чисел які будуть кратні 5.
16	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність слів. Функція повинна знайти та вивести на екран всі слова, що починаються з приголосних літер (алфавіт - латина).
17	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність слів. Функція повинна знайти та вивести на екран всі слова, що починаються з великої літери (алфавіт - латина).

18	Розробити функцію підрахунку розміру директорії.
19	Розробити функцію пошуку файлів за введеним користувачем шаблоном.
20	Розробити функцію підрахунку сумарної кількості всіх файлів заданого типу у вказаній директорії.
21	Розробити функцію перейменування файлів за деяким шаблоном
22	Розробити функцію видалення файлів за деяким шаблоном
23	Розробити функцію видалення каталогів за деяким шаблоном
24	Розробити функцію створення каталогів з ім'ям поточного часу.
25	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність чисел (від 0 до 9). Функція повинна знайти та вивести на екран всі можливі числа, утворені за допомогою введеного набору чисел.
26	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність чисел (від 0 до 9). Функція повинна знайти та вивести на екран всі можливі числа, утворені за допомогою введеного набору чисел які будуть лежати в межах від 1 до 100.
27	Розробити функцію розбору заданої текстового рядка (задається параметром командного рядка), що містить довільну послідовність слів. Функція повинна знайти та вивести на екран кількість повторень слово S, значення якого користувач вводить з клавіатури.
28	Розробити функцію підрахунку сумарного розміру всіх файлів заданого шаблону у вказаній директорії.
29	Розробити функцію підрахунку кількості піддиректорій в заданій директорії.
30	Розробити функцію підрахунку розміру директорії.

Завдання 7

Розробити програму для розв'язку задачі з класами.

№ завдання	Варіант завдання
1	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення середнього значення масиву, обчислення максимального значення елементу масиву, обчислення мінімального значення елементу масиву.
2	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування масиву в порядку зростання, сортування масиву в порядку спадання.
3	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, виконання математичних дій додавання і віднімання елементу масиву на константу.
4	Розробити клас TМобільний_телефон. Атрибути класу: марка, фірма виробник, ціна, колір, тип корпусу. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування назв мобільних телефонів в алфавітному порядку.
5	Розробити клас TАвтомобіль. Атрибути класу: марка, колір, рік виготовлення, типу кузова, пробіг. Методи класу: конструктор без параметрів, конструктор з параметрами,визначення технічних характеристик автомобіля.

6	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення суми елементів масиву.
7	Розробити клас TЄвросоюз. Атрибути класу: назва країни, площа, кількість населеня, масив сусідніх країн, середнє тривалість життя. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор,сортувати країни в алфавітному порядку.
8	Розробити клас TАлфавіт. Атрибути класу: тип алфавіту, регістр букв, список розділових знаків, витота букв. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування букв в зворотньому порядку.
9	Розробити клас TСтудент . Атрибути класу: Ім'я, Прізвище, група, масив оцінок, факультет стать, вік. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення кількості студентів в групі.
10	Розробити клас TСтудент. Атрибути класу: Ім'я, Прізвище, група, масив оцінок, факультет стать, вік. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення середнього значення оцінок.
11	Розробити клас TКабінет. Атрибути класу: висота, ширина, довжина, кількість робочих столів, номер, площа для одного стола. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення значення чи в кабінеті достатня кількість робочих столів.
12	Розробити клас TСтудент. Атрибути класу: Прізвище, група, масив оцінок, факультет стать, вік, зріст. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування студентів за зростом.

13	Розробити клас ТБібліотека. Атрибути класу: назва, адреса, кількість книг в книгосховищі, масив книг. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, функцію сортування книг в бібліотеці.
14	Розробити клас ТВідеокарта. Атрибути класу: Виробник, марка, вартість. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, визначення технічних характеристик відеокарти.
15	Розробити клас ТМова програмування. Атрибути класу: назва, розмірність алфавіту, масв ключових слів, дата ствренння. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування мов програмування в порядку зростання за характеристикою: дата випуску.
16	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, обчислення середнього значення елементів масиву.
17	Розробити клас ТМузика. Атрибути класу: назва, виконавець, автор слів, автор музики, назва альбому, рік випуску. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування пісень в алфавітному порядку .
18	Розробити клас ТОпераційна система. Атрибути класу: назва ОС, тип ОС, фірма виробник, вартість. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, визначеня технічних характеристик операційної системи.
19	Розробити клас ТБібліотека. Атрибути класу: назва, адреса, кількість книг в книгосховищі, масив книг. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, метод перегляду наявності книг в книгосховищі.

20	Розробити клас ТОперативна пам’ять. Атрибути класу: виробник, марка, вартість, об’єм . Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, визначення технічних характеристик оперативної пам’яті.
21	Розробити клас ТДокументи. Атрибути класу: назва, власник, тип, дата подання, віза секретаря. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування документів на поступлення в алфавітному порядку.
22	Розробити клас ТФігура. Атрибути класу: тип фігури, масив довжин сторін, колір, периметр. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, визначення площі фігури.
23	Розробити клас ТМобільний телефон. Атрибути класу: марка, фірма виробник, ціна, колір, тип корпусу. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, сортування мобільних телефонів за ціною.
24	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, виконання математичних дій множення і ділення елементу масиву.
25	Розробити клас TMatrix. Атрибути класу: розмірність масиву, однорідний масив значень. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, знаходження спільних елементів масивів.
26	Розробити клас ТНоутбук. Атрибути класу: назва, фірма виробник, вартість, масив характеристик. Методи класу: конструктор без параметрів, конструктор з параметрами, деструктор, функція вибору ноутбуків, що задовільняють введеним з клавіатури характеристикам.