ЗМІСТ

ВСТУП 4
1 ЗАГАЛЬНА ЧАСТИНА 5
1.1 Оцінювання знання теоретичного матеріалу 5
1.2 Оцінювання виконання практичної частини роботи 6
1.3 Техніка безпеки при виконанні робіт 7
1.4 Програмне забезпечення, необхідне для роботи 7
1.4.1 Короткий опис середовища розробки програм Dev-CPP 7
1.4.2 Опис програми термінального клієнта Xshell 14
1.4.3 Використання програми Midnight Commander 19
1.4.4 Робота з компілятором GCC 31
1.4.5 Призначення програми make 52
2 ЛАБОРАТОРНА РОБОТА №1 СТВОРЕННЯ ПРОГРАМНОГО ЕМУЛЯТОРА МЕНЕДЖЕРА ПАМ'ЯТІ 66
2.1 Теоретична частина 66
2.2 Завдання до лабораторної роботи 74
2.3 Деякі рекомендації до виконання роботи 76
2.4 Завдання для підготовки до лабораторної роботи 84
2.5 Контрольні запитання до лабораторної роботи 85
ПЕРЕЛІК ПОСИЛАНЬ 87
ДОДАТОК А ФОРМА ТИТУЛЬНОГО АРКУША І ЗМІСТ ПРОТОКОЛУ РОБОТИ 88

ВСТУП

Методичні вказівки призначені для студентів третього курсу факультету комп'ютерних наук, всіх спеціалізацій, котрі вивчають дисципліну «Операційні системи» з виконанням відповідних лабораторних робіт.

Виконання лабораторних робіт з цієї дисципліни бажано робити за допомогою мов C/C++, оскільки ці мови, на відміну від мови Java, надають можливість безпосередньо звертатись до системних викликів, а також власними засобами мають доступ практично до всіх ресурсів як операційних систем, так і до ресурсів безпосередньо комп'ютера. Взагалі написання якісних програм, навіть рівня користувача, неможливе без знання архітектури операційних систем, володіння різними системними викликами та розуміння основ роботи з обладнанням комп'ютера програмними інструментальними засобами.

Питання, які виносяться на лабораторні роботи досить складні, і з огляду на обмеженість часу, відведеного на лекційну частину курсу, треба відзначити, що вони потребують вивчення за допомогою додаткової літератури, роль якої і повинні виконувати дані методичні вказівки.

Тому ***метою*** даних методичних вказівок є надання студентам як можна більш повної інформації з теоретичних питань до відповідних лабораторних робіт і, крім того, ознайомити студентів з програмно-інструментальними засобами, що рекомендуються при виконанні робіт і допомагають при компіляції розроблених програм мовами C/C++. Ця інформація повинна допомогти студентам при створенні не тільки лабораторних робіт з операційних систем, але і при виконанні досить складних розробок у курсових та дипломних проектах.

Завданням методичних вказівок є навчити студентів вибирати зі всіх можливих засобів розробки додатків ті, котрі є найбільш придатними і ефективними у кожному конкретному випадку їх застосування.

1 ЗАГАЛЬНА ЧАСТИНА

В цьому розділі описані загальні питання, щодо виконання робіт, а саме:

- методика оцінювання знань з теоретичного матеріалу стосовно лабораторної роботи;
- методика оцінювання виконання практичної частини завдання;
- питання додержання правил техніки безпеки;
- опис програмного забезпечення, що застосовується при виконанні робіт;
- деякі основні відомості з програмування мовами C/C++ на відміну від мови Java.

1.1 Оцінювання знання теоретичного матеріалу

До початку лабораторного заняття студент, за рахунок часу, відведеного на самостійну роботу, повинен підготуватися до виконання роботи шляхом вивчення відповідних розділів курсу по підручнику [1], а також за допомогою додаткової літератури [2, 3] (всі джерела розміщені у електронній бібліотеці ОДЕКУ). Підготовка мусить містити у собі: ознайомлення з метою роботи, вивчення теоретичних питань, що відносяться до неї, відповіді на контрольні питання, котрі містяться в методичних вказівках наприкінці теоретичної частини до кожної лабораторної роботи, оформлення титульного аркуша й заповнення розділів 1 і 2 протоколу лабораторної роботи (див. додаток А).

Необхідні сторінки підручника, а подеколи і додаткової літератури, вказуються безпосередньо після контрольних запитань перед завданням на практичну роботу до кожної лабораторної роботи.

Перед проведенням заняття викладач зобов'язаний перевірити в студента наявність підготовленого й попередньо заповненого протоколу. *Якщо підготовлений протокол відсутній* ***студент не допускається*** *до виконання роботи* з виставлянням нульової оцінки за практичну частину роботи. Після перевірки наявності й правильності заповнення протоколу викладач повинен провести опитування знань студента й допустити останнього до роботи з відповідною оцінкою за теоретичну частину. Опитування проводиться по контрольних питаннях до даної лабораторної роботи (3 – 4 питання студентові), повна, якісна відповідь на них гарантує одержання 60% балів за теоретичну частину роботи. Ця частина опитування може поводиться або в усній формі, або у письмовому вигляді. Для оцінки ступеня знайомства студента з додатковою літературою та

одержання більше високої оцінки студентові необхідно відповісти на кілька додаткових питань (1 – 2 питання) за змістом цієї самої додаткової літератури.

1.2 Оцінювання виконання практичної частини роботи

Практична частина роботи оцінюється в такий спосіб. На кожному занятті викладач оцінює відсоток виконаної роботи. Якщо лабораторна робота розрахована на чотири години й більше, то загальна оцінка за роботу прямо пропорційна відсотку виконання завдання на першому лабораторному занятті. Наприклад, якщо на першому занятті студент написав 60% програмного коду, а на другому ті 40%, що залишилися, та здав на третьому занятті повністю оформлений протокол – він одержує 100% балів за дану роботу. Навпаки, якщо на першому занятті виконано 10% завдання на другому 20%, а на заключному занятті зданий протокол з повністю виконаною лабораторною роботою, то студент може розраховувати на одержання тільки 30% від можливих балів за практичну частину роботи. На оцінку практичної частини роботи впливає й строк здачі протоколу. Для одержання максимуму балів повністю оформлений протокол повинен бути зданий на останньому занятті, котре відведено для виконання роботи або наступного дня після виконання роботи. Затримка зі здачею протоколу знижує оцінку на 5% за кожний повний і неповний тиждень затримки. Тобто студент, котрий за ритмічність виконання роботи міг одержати, наприклад, 70% відсотків можливих балів, але без поважної причини здав протокол через 16 днів після її завершення, одержить тільки 70-15=55% балів за виконання роботи.

Максимальна кількість балів за практичну і теоретичну частини кожної роботи становить 100 балів.

Допуск до підсумкового контролю знань можливий тільки при одержанні не менш ніж 50% окремо за практичну та теоретичну частину ***всіх*** запланованих лабораторних робіт. Виконання половини робіт на 100%, а другої половини на 40% оцінюється тільки в 40%, а не 70% і, таким чином, студент не одержує допуск до підсумкового контролю знань.

Наприкінці семестру виводиться середня оцінка студента за всіма заняттями. Ця оцінка є підсумковою оцінкою за виконання практичного модуля і повинна складати не менш ніж 50% від максимально можливої сумарної оцінки за всі лабораторні роботи.

1.3 Техніка безпеки при виконанні робіт

При виконанні лабораторних робіт необхідно додержуватись правил, котрі встановлені в лабораторіях кафедри за узгодженням зі службою охорони праці і техніки безпеки університету. На початку виконання курсу робіт (на початку семестру) студенти повинні шляхом проведення інструктажу ознайомитися з цими правилами і розписатись у відповідному журналі.

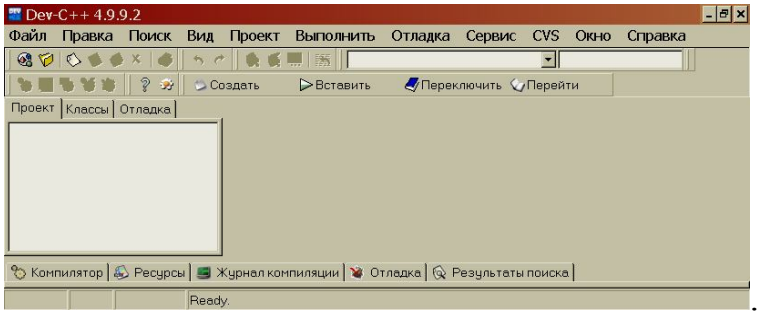
Особливо слід звернути увагу на те, що при роботі за комп'ютером обов'язково треба робити 10-ти хвилинну перерву на кожну годину, що проведена за комп'ютером. Не можна працювати за комп'ютером більш ніж 4 години поспіль без тривалої перерви. Не рекомендується наближатись с тильного боку до моніторів, в яких застосовані електронно-променеві трубки. В разі виявлення несправності комп'ютера, ***категорично забороняється самостійно*** вживати заходів з діагностування та спроби з її усунення, в цьому випадку слід повідомити про несправність викладача або навчально-допоміжний персонал лабораторії.

1.4 Програмне забезпечення, необхідне для роботи

В цьому розділі розглянуте програмне забезпечення, знання якого і вміння роботи з яким є необхідним для якісного виконання курсу лабораторних робіт.

1.4.1 Короткий опис середовища розробки програм Dev-CPP

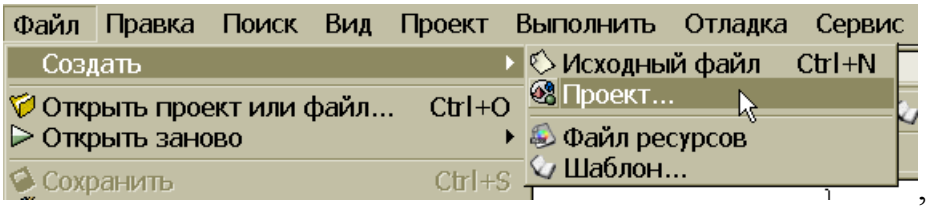
Лабораторні роботи мовою C/C++ в ОС Windows виконуються в середовищі розробки Dev-CPP. Це досить потужне середовище, що володіє засобами не тільки для виконання лабораторних робіт, але й для здійснення досить складних проектів. У той же час це середовище розробки дуже просте й, до того ж, є безкоштовним тому, що поставляється під ліцензією GPL. Запуск програми здійснюється стандартним способом через меню кнопки «Пуск» або подвійним клацанням по іконці на робочому столі. Робоче вікно програми після її запуску показано на рисунку



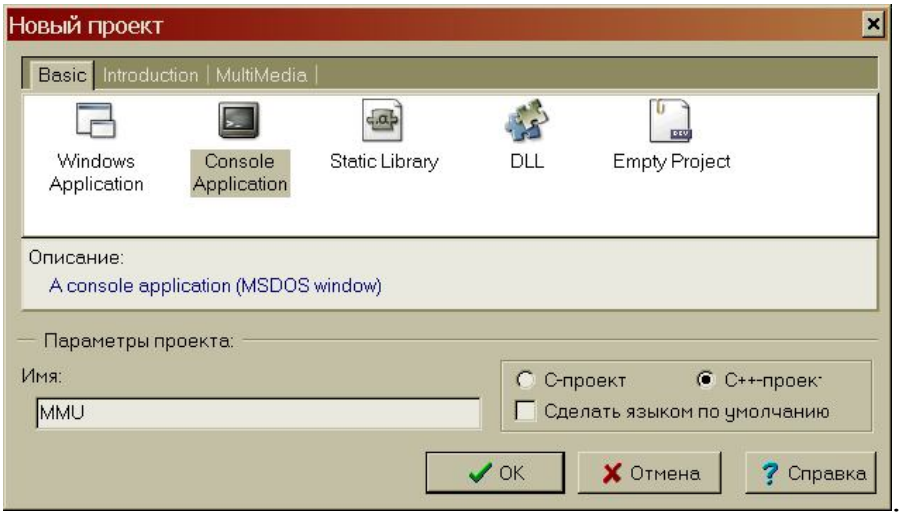
Як звичайно, інтерфейс програми містить рядок меню, кілька панелей інструментів і рядок статусу. Крім того в правій частині вікна розташовані три панелі, що перемикаються вкладками «Проект», «Классы» і «Отладка». Трохи незвичним елементом інтерфейсу є ряд вкладок, розташованих над рядком статусу. У цих вкладках під час роботи над проектом можна спостерігати різну інформацію, наприклад, повідомлення компілятора (вкладка «Компилятор») або управляти налагодженням програми (вкладка «Отладка»).

Вся інша частина вікна відведена під вкладки редактора вихідних файлів розроблювального проекту.

Робота над новим проектом починається з його створення, для цього за допомогою пункту меню «Файл | Создать | Проект...»



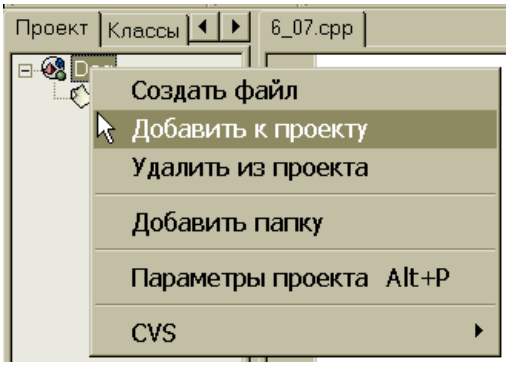
або за допомогою кнопки на панелі інструментів треба відкрити діалогове вікно створення проекту, у якому вказати ім’я й мову проекту

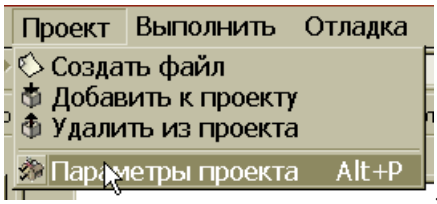


У верхній частині вікна на вкладці «Basic» потрібно вказати тип проекту. Для виконання лабораторних робіт необхідно вибрати тип проекту «Console Application».

Якщо після закриття діалогового вікна виявляється яка-небудь помилка в його створенні, наприклад, не введено ім’я проекту, то це можна виправити, скориставшись локальним меню на вкладці «Проект» і вибравши в ньому

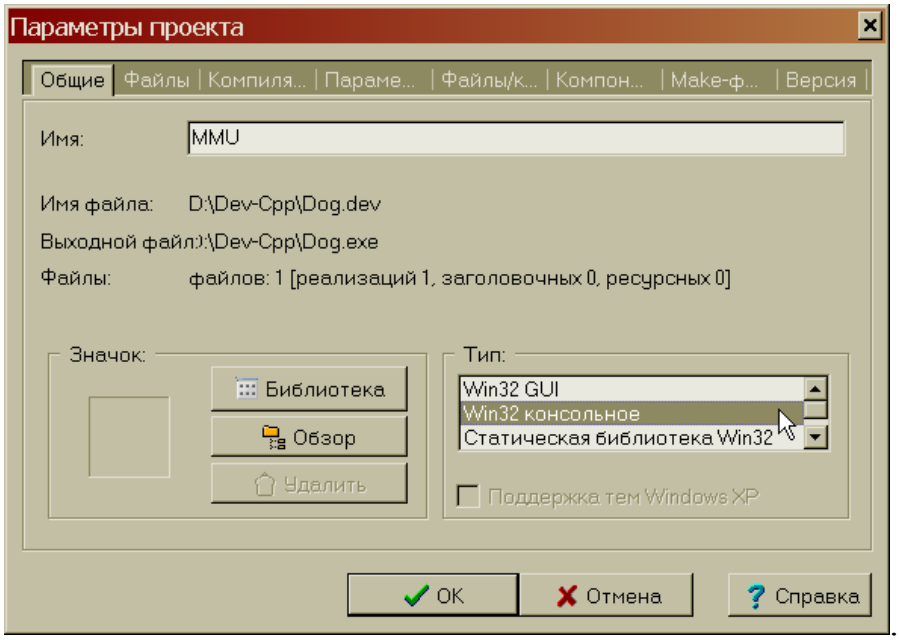
пункт «Параметры проекта»



або через меню


.

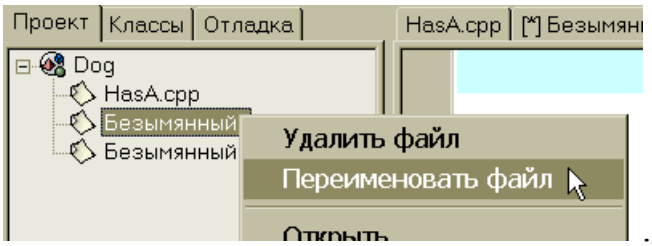
При цьому відкривається вікно майстра завдання параметрів проекту, у якому, перемикаючи вкладки, можна задати або змінити будь-які параметри



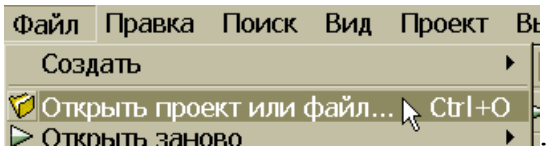
.

Тут можна вказати які дії виконувати з файлами проекту (вкладка «Файлы»), підключити зовнішній компілятор замість стандартного (вкладка «Компилятор»), параметри компонування (вкладка «Компоновка») та інше.

Після створення проекту в нього додаються файли, це можна зробити декількома способами: за допомогою кнопки на панелі інструментів або через пункти «Создать файл» або «Исходный файл» в меню, що описані вище. При цьому в дерево проекту в панелі «Проект» додається новий елемент із ім’ям «Безымянный» і відкривається нова вкладка з тим же ім’ям у панелі текстового редактора. Файл можна перейменувати через відповідний пункт локального меню в панелі «Проект»

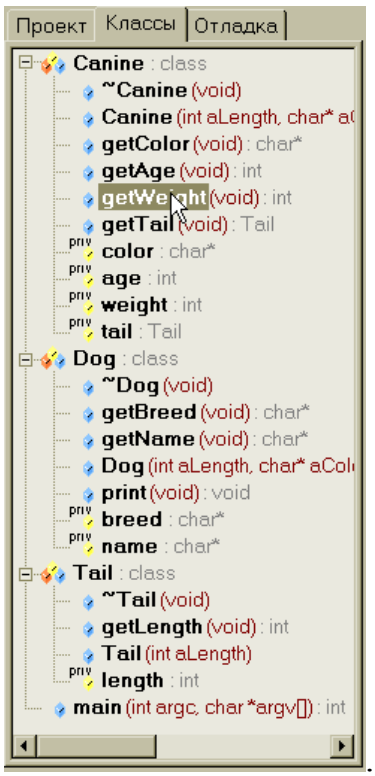


Раніше створений проект відкривається стандартним способом – кнопкою на панелі інструментів або через пункт меню «Файл | Открыть проект или файл»

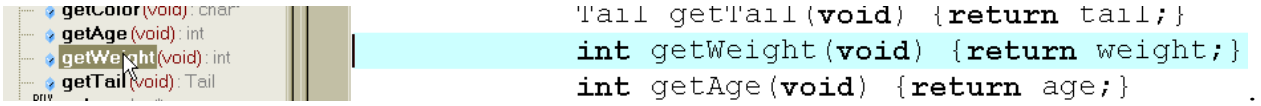


Є очевидним, що користуючись цим же пунктом меню можна відкрити в черговій вкладці редактора раніше створений файл вихідного коду. Крім того, додавання нового файлу в проект із одночасним його відкриттям можна здійснити через пункт у локальному меню панелі «Проект» – «Добавить к проекту».

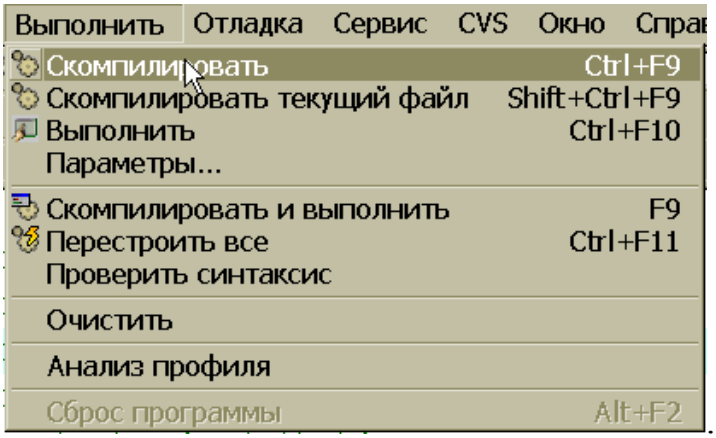
При написанні програми або при відкритті раніше створеного проекту, у вкладці «Классы» відображається ієрархія класів програми з усіма їхніми полями й методами



Вибір якого-небудь класу або методу в панелі «Классы» призводить до переміщення курсору поточного рядка у редакторі (позначається блакитним кольором фона) на початок відповідного класу або методу

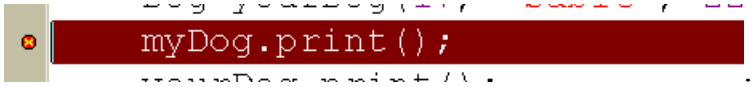


Компіляція всього створеного проекту або окремих його файлів виконується за допомогою кнопок на панелі інструментів або меню «Выполнить»

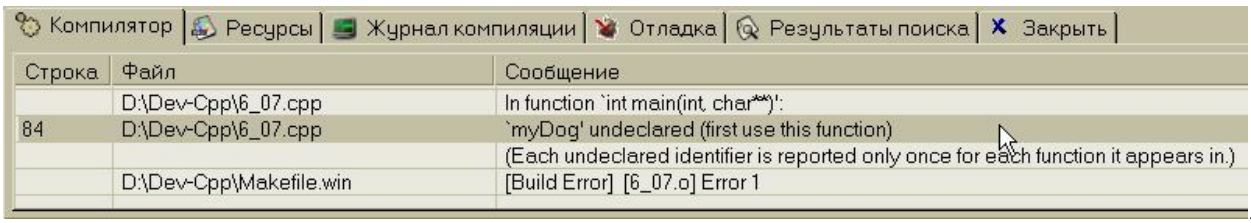


Кнопки на панелі розташовані в наступному порядку (ліворуч праворуч): «Скомпилировать», «Выполнить», «Скомпилировать и выполнить», «Перестроить все».

При компіляції програми можлива поява повідомлення про синтаксичну помилку. Рядок, у якому припускається помилка, позначається в редакторі тексту символом і виділяється коричневим кольором фона



Зміст помилки пояснюється у вкладці «Компилятор» унизу вікна



Остання в рядку кнопок компіляції – кнопка «Отладка». Ця кнопка відкриває вкладку «Отладка» у нижній частині вікна. Та сама вкладка відкривається й через меню «Отладка | Отладка». Використовуючи кнопки в цій вкладці або пункти меню «Отладка» можна управляти процесом налагодження програми.

Налагодження являє собою один з етапів створення працездатної програми, причому етап не менш важливий, ніж написання вихідного коду.

Проста істина полягає в тому, що рідко кому вдається з першого разу написати програму без яких-небудь логічних помилок. У програмуванні помилки є цілком нормальним явищем. У цьому немає нічого дивного. Однак поява помилки в програмі змушує програміста багаторазово перевіряти все із самого початку. Цей процес і називається «налагодженням». Іншими словами налагодження – це знаходження й виправлення помилок у програмі. Дуже часто пошук і усунення помилок займають більше часу, ніж саме написання програми. Налагодження не є точною наукою. Найкращий засіб налагодження – голова програміста. Проте, системний підхід до процесу налагодження може дати деякі переваги.

Процес налагодження в загальному випадку можна розділити на чотири етапи:

1. виявлення помилки;
2. пошук її місцезнаходження;
3. визначення причини помилки;
4. виправлення помилки.

Найбільш важким є другий етап. Він полягає в тім, щоб знайти місце в програмі, де знаходиться помилка. Просто неможливо тримати в голові всю програму відразу (якщо ця програма не дуже маленька). Кращий підхід – розбивати програму на частині й налагоджувати їх окремо. Особливих проблем при об'єктно-орієнтованому і навіть при модульному програмуванні з такою розбивкою не виникає, оскільки програми вже розбиті на окремі функції або об'єкти з їхніми методами. Тому фактично налагодження зводиться до локалізації помилки усередині основних алгоритмічних конструкцій у тій або іншій функції, у тім або іншого методі.

Можна, звичайно, для знаходження логічних помилок у вихідний текст програми вставляти додаткові (може бути не потрібні, але що допомагають знаходити помилки) виклики процедур печатки, наприклад, так

printf("*Значення змінної* x *на* %d-*му кроці циклу дорівнює* %f\n", i, x).

Але такий підхід досить скрутний і набагато зручніше користуватися мо-

жливостями налагоджувачів, що вбудовуються в інтегровані середовища розробки програм.

Налагоджувач середовища Dev-CPP допомагає в розв'язанні двох самих важких завдань налагодження – визначення місцезнаходження помилки й виявлення її причини. Досягається це завдяки широким можливостям припиняти виконання програми й перевіряти її стан у будь-якій точці. Можливості налагоджувача реалізуються такими функціями як трасування, покрокове виконання й спостереження.

Трасування «Шаг внутрь» – виконання програми по одному оператору із заходом у процедури й функції

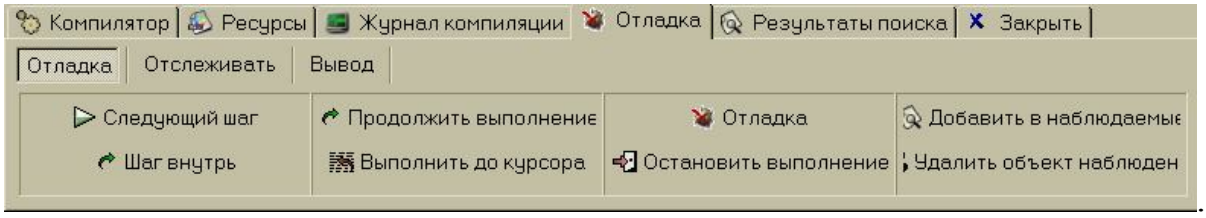
Покрокове виконання «Следующий шаг» – виконання програми по одному оператору. Виклики методів і функцій виконуються за один крок налагодження як єдине ціле. Значно прискорює налагодження.

Спостереження «Добавить в наблюдаемые» – вибір деяких змінних і спостереження за зміною їхніх значень по ходу виконання програми.

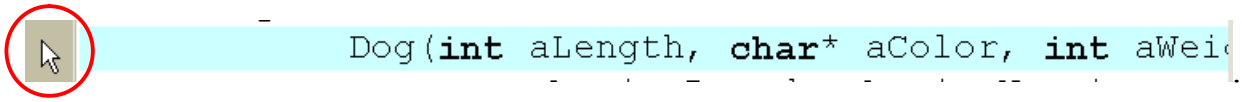
Ці засоби налагодження дозволяють розбивати програму на окремі частини й переконуватися в працездатності однієї частини, перш ніж переходити до наступній.

Слід зазначити, що при всіх своїх можливостях, налагоджувач не може думати за програміста. Налагоджувач це потужний інструмент, але якщо його використовувати бездумно, він навряд чи допоможе заощадити час і зусилля.

Вкладка «Налагодження» з усіма її кнопками й іншими елементами керування має такий вигляд



У довільних місцях програми можна встановлювати контрольні точки. Робиться це простим клацанням лівої кнопки миші в полі вікна редактора, ліворуч від оператора, на якому програма повинна зупинитися

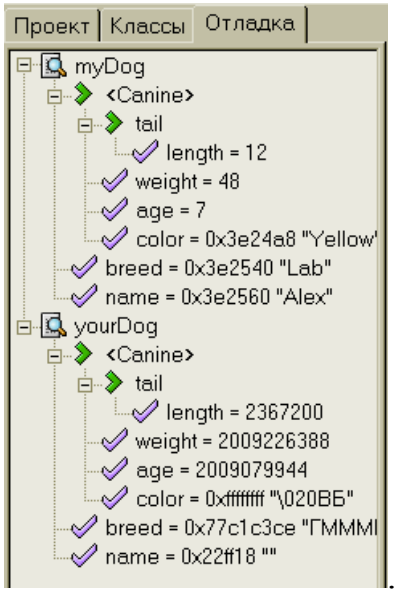


У редакторі контрольні точки позначаються символом і виділяються червоним кольором фона. Оператор, що буде виконуватися наступним при тра-

суванні або покроковому виконанні, позначається символом і синім кольором фона

```
Dog myDog(12, "Yellow", 48, 7, "Lab", "Alex");
Dog yourDog(17, "Sable", 22, 12, "Sheltie", "Ch
myDog.print();
```

Об’єкти (змінні), додані в спостережувані, відображаються в панелі «Отладка». Тут відображуються значення, що об’єкти мають у сучасний момент



Спостереження за значенням змінних при покроковому або поблочному виконанні програми дозволяє визначити місце вихідного коду, де з’являється логічна помилка.

1.4.2 Опис програми термінального клієнта Xshell

Лабораторні роботи з використанням ОС Linux виконуються на віддаленому Linux-сервері з робочої станції під управлінням ОС Windows. При подібній роботі необхідний якийсь інструмент, котрий дозволяє емулювати Linux-термінал, призначений для роботи на платформі Microsoft Windows. Такий програмний продукт за назвою Xshell [5] був створений компанією NetSarang Computer – провідним розроблювачем і постачальником програм, призначених для продуктивної взаємодії у мережах.

Програмний продукт Xshell підтримує протоколи SSH1 і SSH2, способи автентифікації користувачів із застосуванням відкритого ключа DSA/RSA і до-

зволяє кодувати трафік за допомогою різноманітних алгоритмів шифрування. Одним з основних аргументів на користь вибору цієї програми слугує те, що вона є безкоштовною для використання дома й у закладах освіти. Правда необхідно відзначити те, що існує й великою популярністю користується інший безкоштовний клієнт – PuTTY. Але інтерфейс користувача, що надається Xshell є більш звичним для людей, котрі працюють в Windows, ніж трохи спрощений інтерфейс PuTTY.

За твердженням авторів, програма працює в будь-якій операційній системі Windows починаючи з Windows 2000 до Windows 8 (новітня версія), можлива робота навіть в Windows 98 (більш ранні версії).

NetSarang Xshell надзвичайно потужний клієнт, котрий підтримує протоколи, що вже згадувалися – SSH1, SSH2, крім того підтримуються SFTP, TELNET, RLOGIN і SERIAL (останній для підключення тільки по COM портах), можливе здійснення роботи через proxy-сервер. Програма дозволяє одночасно підключатися до декількох серверів – у програмі убудована підтримка вкладок. Існує можливість швидко міняти розташування інструментів на панелях і їхній набір, для кожної вкладки, настроювати колірну схему, шрифти, перекодування й т.д. Всі схеми настроювань зберігаються, і надалі дуже просто повернутися до зручного інтерфейсу, що влаштовує користувача.

Безпосередньо робота із сервером організована так само дуже зручно – можна зберегти як окремі команди, так і їхній набір у панелі «Quick Commands». Набори зручно структуруються й у результаті можна зовсім позбутися нудотного набивання команд. Але навіть цим розроблювачі з NetSarang не обмежилися – в Xshell команди можна відправляти відразу на кілька сесій. Єдиним помітним “недоліком”, у роботі із клієнтом, можна назвати відсутність іншої мови інтерфейсу, крім англійської, але робота з ним не повинна спричиняти труднощі у студентів досить добре знайомих зі стандартним для Windows інтерфейсом вікон.

Все це різноманіття функцій має велике значення для системних адміністраторів, що віддалено працюють з Linux-серверами. Для безпосереднього виконання лабораторних робіт необхідно вміти користуватися тільки досить обмеженим набором цих функцій.

Тому нижче описані тільки запуск Xshell, створення з’єднання й здійснення деяких основних настроювань.

Запускається Xshell, стандартним для ОС Windows способом – подвійним клацанням лівої кнопки миші по іконці програми на робочому столі або, якщо такої немає, то через меню кнопки «ПУСК».

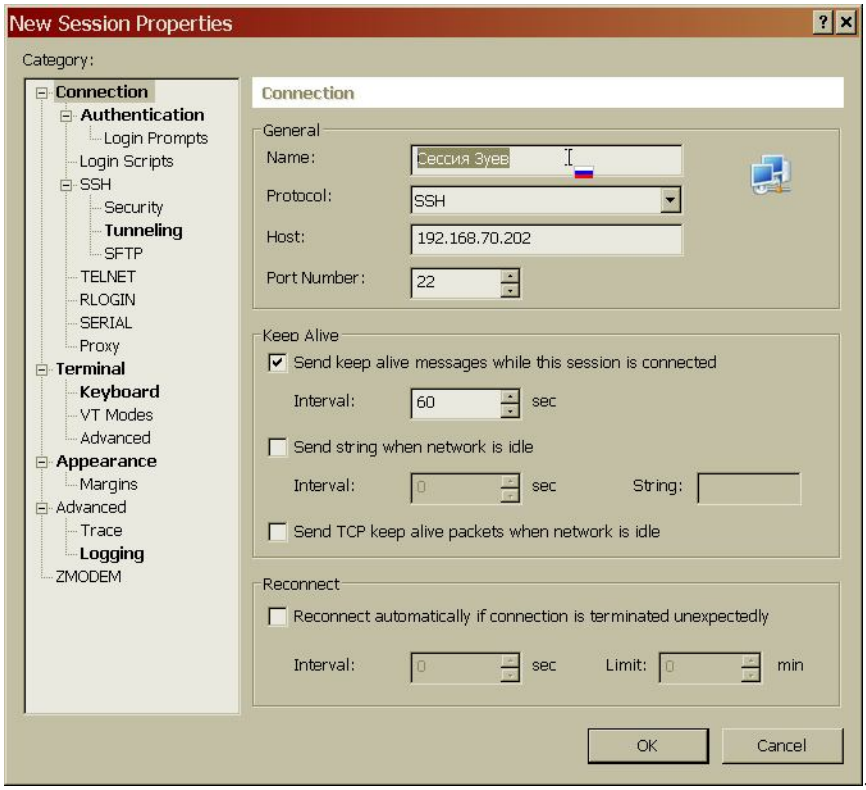
Вікно програми, котре відчиняється, хоча і являє собою емуляцію термі-

нала, але, у той же час, є звичайним вікном Windows з рядком меню й панеллю інструментів:

Докладний опис інтерфейсу програми не має особливого сенсу, тому більш доцільним буде перейти безпосередньо до опису процесу створення з'єднання з віддаленим сервером.

Створити нову сесію можна, як звичайно, скориставшись або меню «File | New...», або кнопкою «New» на панелі інструментів, або, нарешті, гарячою клавішею «Alt+N».

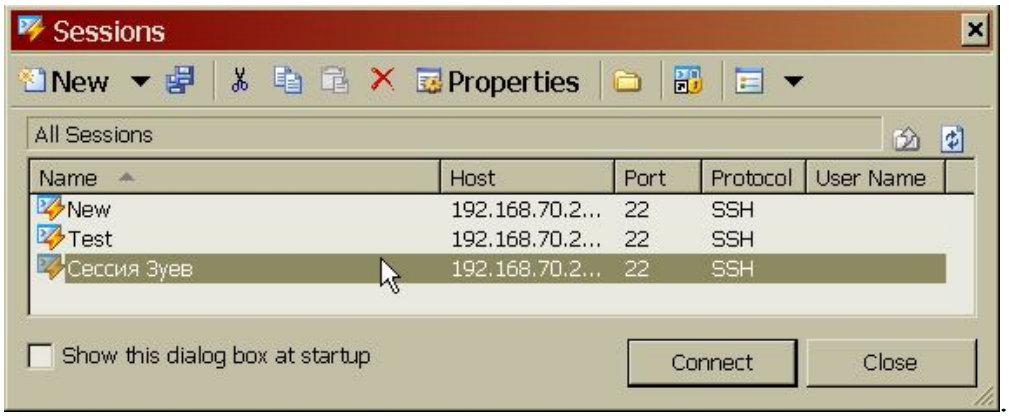
При цьому для створення з'єднання із сервером відкривається діалогове вікно «New Sessions Properties»



У полі вводу «Name» цього вікна необхідно ввести ім'я створюваної сесії це, наприклад, може бути логін користувача або будь-яке інше. Цим ім'ям на-

далі можна буде користуватися для повторного відкриття сесії з тими ж параметрами. У полі «Host» уводиться ім'я або IP-адреса сервера, найбільш імовірна адреса 192.168.70.202, але можливо й інше значення, про яке можна довідатися у викладача, що проводить заняття. Інші поля вводу «Protocol», «Port Number» та інші елементи керування залишаються без зміни. У цьому ж вікні, скориставшись полем «Category», можна задати спеціальні параметри конфігурації сесії. Параметри можна установити й пізніше вже після з'єднання й початку роботи в сесії.

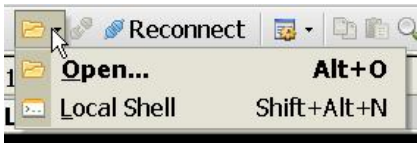
Після закінчення діалогу відкривається нове діалогове вікно – «Sessions», у якому відображається список всіх можливих сесій і пропонується відкрити тільки що створену, або будь-яку іншу на вибір користувача



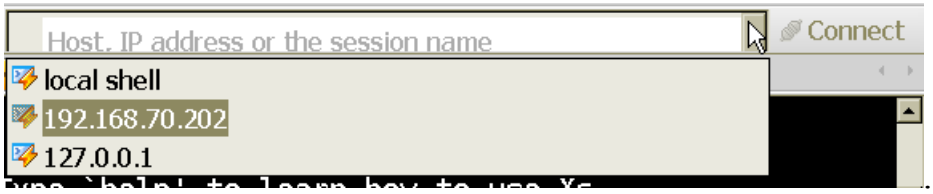
Після вибору й натискання на кнопку «Connect» здійснюється з'єднання із сервером і останній запитує логін і пароль користувача. Автентифікація виконується в термінальному режимі в рядку запрошення Linux, причому при введенні пароля він ніяк не відображається.

За результатом правильного уведення логіну й пароля користувач допускається до його домашнього каталогу й одержує можливість працювати на віддаленому сервері.

Якщо сесія вже була створена раніше, то діалогове вікно «Sessions» можна відкрити іншим шляхом. Для цього є чотири можливості – через пункт меню «File | Open», за допомогою кнопки «Open» на панелі інструментів, скористатися кнопкою «+» ліворуч в області заголовків вкладок поточних сесій або, на сам кінець, по гарячій клавіші «Alt+O»



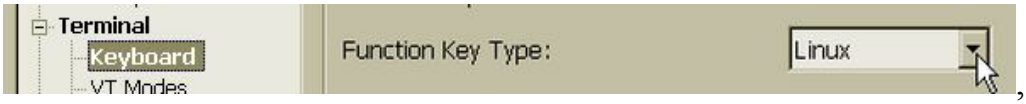
З’єднатися з хостом можна також і за допомогою рядка адреси, що розташований під панеллю інструментів. Для з’єднання потрібно ввести в рядок ім’я сесії або IP-адресу хосту або вибрати їх зі списку, що випадає, і натиснути кнопку «Connect»



Завдання параметрів сесії, як уже було сказано вище, при створенні нової сесії виконується у полі «Category» вікна «New Sessions Properties». Доступ до настроювань також можна одержати скориставшись пунктом меню

у вікні «Sessions». Для виконання лабораторних робіт досить завдання мінімальної кількості настроювань у цьому деревоподібному меню. Через пункт «Terminal» у полі «Encoding» зі списку, що випадає, вибирається код символів «Unicode (UTF-8)»

а в пункті «Keyboard» у полі «Function Key Type» зі списку, що випадає, вибирається «Linux»



хоча можна залишити й значення за замовчуванням «<Default>».

Тут же в пункті «Appearance» можна змінити тип і розмір шрифту термінала, але це простіше зробити за допомогою відповідної кнопки на панелі інструментів:

У даних методичних вказівках наведені тільки наймінімальніші відомості необхідні для початку роботи в програмі Xshell. Більш докладно з роботою в програмі можна ознайомитися в керівництві користувача [5] (англійською).

1.4.3 Використання програми Midnight Commander

Для управління файловою системою й для роботи з файлами в консольному режимі UNIX-сумісних операційних систем використовуються команди інтерпретатора системних команд ОС, наприклад такі, як **`pwd`**, **`ls`**, **`cd`**, **`mv`**, **`mkdir`**, **`rmdir`**, **`cp`**, **`rm`**, **`cat`**, **`more`** і т.д. Але набагато зручніше виконувати більшу частину роботи з обслуговування файлової системи за допомогою спеціальних командних оболонок. В ОС UNIX і сумісних з нею найбільш часто використовується програма Midnight Commander [4].

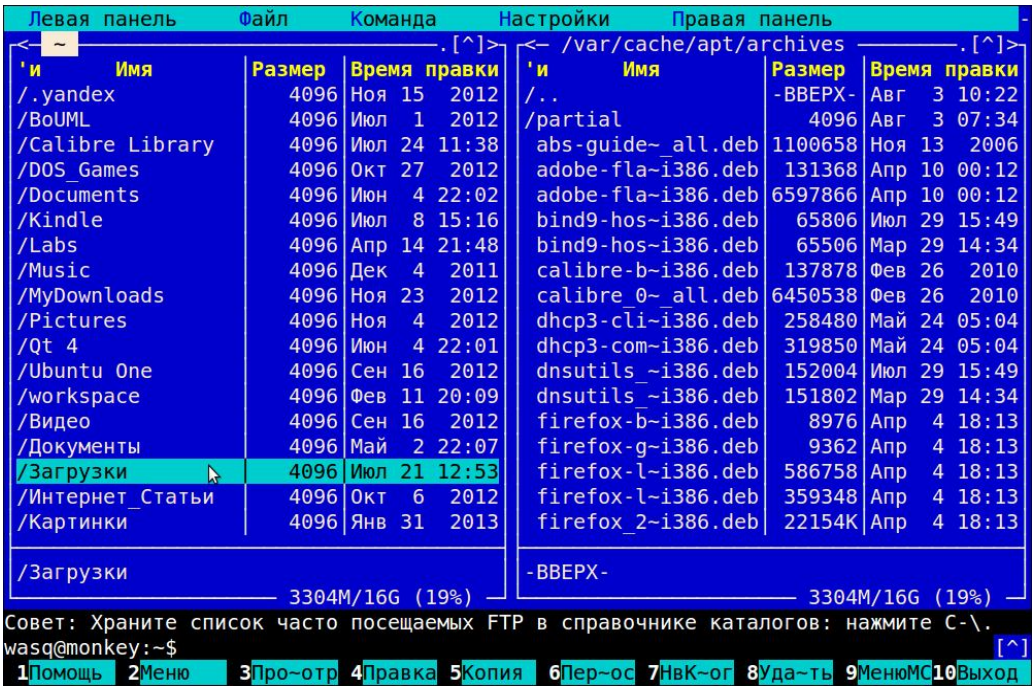
Midnight Commander – це програма, що призначена для перегляду вмісту каталогів і виконання основних функцій управління файлами в UNIX-подібних операційних системах. Програма Midnight Commander (або просто **`mc`**) надає можливість наочно представити всі виконувані дії, полегшуючи тим самим виконання операцій з файловою системою. Програма дозволяє переглянути структуру каталогів і виконати основні операції по управлінню файловою системою. Інакше кажучи, це файловий менеджер. В ОС типу Windows також існують аналоги цієї програми, наприклад FAR. Більш за те, Midnight Commander, як програма, розповсюджується під ліцензією GPL і перенесена в ОС Windows у вигляді файлу **`mc.exe`**. Студенти, що мають досвід роботи з FAR в Windows, легко зможуть працювати й з **`mc`**, оскільки навіть основні комбінації «гарячих клавіш» у цих програмах збігаються. Для тих же, хто не знаком з FAR, у даному розділі методичних вказівок наведений короткий опис роботи з файловим менеджером **`mc`**.

Необхідно сказати, що оскільки робота із програмою під час лабораторних робіт виконується в емуляторі термінала, а не в самому терміналі, можлива деяка розбіжність у реакції програми на натискання гарячих клавіш на відміну від звичайної реакції. Це пов’язане з тим фактом, що обробка деяких гарячих

клавіш перехоплюється терміналом і виконується за його правилами, наприклад, натискання клавіші <F10>, викликає запит на закриття термінала, а не програми **mc**. Але таке поводження характерно тільки при використанні гарячих клавіш, робота ж по управлінню файловою системою через меню програми до таких несподіванок не веде.

Запуск програми виконується шляхом набору команди «**mc**» в командному рядку термінала і натискання клавіші <Enter>.

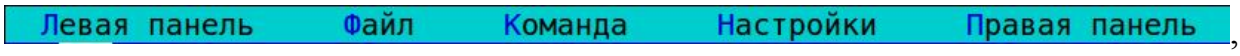
У результаті виконання команди, з'являється робоче вікно програми, котре виглядає так:



і має кілька структурних елементів.

Насамперед, це дві «панелі», котрі при роботі з Midnight Commander займають майже весь простір екрана й, найчастіше за все, відображають списки файлів двох каталогів. Ця частина, як і рядок вводу команди

(точніше – це командний рядок поточної оболонки ОС), де можна ввести й запустити на виконання будь-яку команду системи є основною й обов'язковою частиною в інтерфейсі програми, всі інші, тобто – головне меню



,

рядок порад програми

і рядок функціональних клавіш, яку можна вважати підказкою по використанню функціональних клавіш, а можна з її допомогою й безпосередньо запускати відповідні команди, клацаючи мишкою по екранній кнопці

можуть бути відключені при настроюванні зовнішнього вигляду програми для збільшення корисного простору файлових панелей.

При цьому доступ до рядка меню вибору команд можна одержати шляхом натискання клавіші <F9> або, якщо з якої-небудь причини вона не працює, шляхом послідовного натискання клавіш <Esc> і <9>, або за допомогою миші – клацанням її лівої кнопки по верхньому рядку файлера.

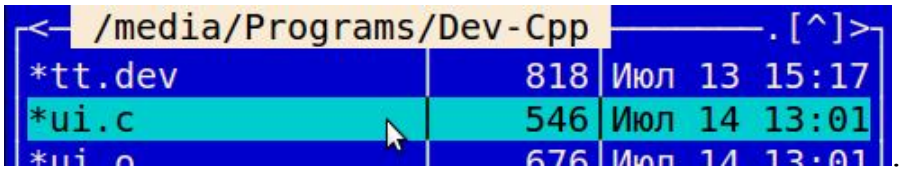
Використовуючи пункти основного меню Midnight Commander можна виконати будь-які доступні файлові команди. Навіть коли рядок функціональних клавіш (котрий можна розглядати просто як підказку) відключений, можливість виконання будь-якої операції з числа асоційованих з функціональними клавішами, крім <F1>, <F2> і <F9>, забезпечується через меню «Файл»

.

Кожна з двох панелей складається із заголовка, списку файлів якого-небудь каталогу й рядків «міні-статусу» (останній може бути відсутнім, це задається настроюваннями програми). У заголовку панелі вказується повний шлях до каталогу, уміст якого відображається в панелі, наприклад,

а також чотири екранні кнопки – <, ., [^] і >, які використовуються для управління програмою за допомогою миші. У рядках міні-статусу відображаються деякі дані про файлову систему, наприклад, ім'я файлу або каталогу, на який у цей момент указує підсвічування, розмір каталогу, вміст якого відображається в панелі, загальний розмір файлової системи й вільне місце в системі у відсотках (уміст міні-статусу задається в настроюваннях програми)

Одна з панелей є поточною (активною), про що свідчить підсвічування ім'я каталогу в заголовку панелі й підсвічування одного з її рядків

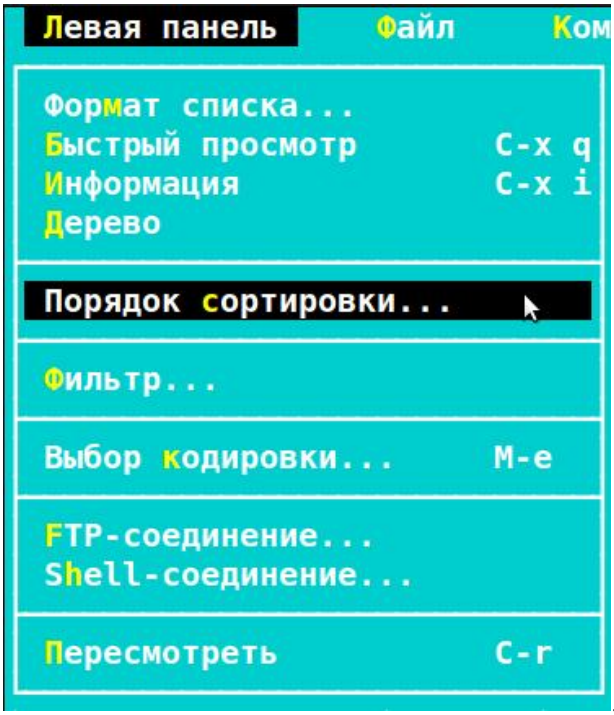


Відповідно, каталог, що відображується в активній панелі, є поточним для ОС у цей момент. Тому майже всі доступні файлові операції виконуються в цьому каталозі. При цьому каталог, що є відображуваним у другій панелі, слугує як цільовий каталог при операціях копіювання <F5> і перенесення <F6>. Файлові операції, крім команди створення нового каталогу – <F7>, виконуються над підсвіченим файлом або каталогом. Підсвічування в активній панелі можна переміщати за допомогою клавіш управління курсором.

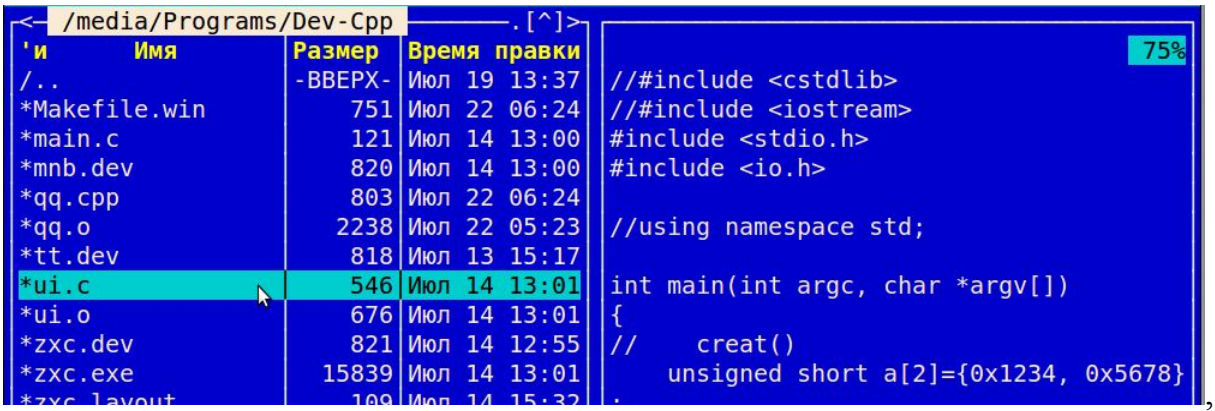
Панелі програми Midnight Commander найчастіше відображають уміст каталогів файлової системи (тому іноді вони називаються панелями каталогів). Однак на панель може бути виведена й деяка інша інформація.

Вигляд панелей, у яких відображаються списки файлів і підкаталогів, може бути змінений через команди меню, що випадає, лівої й правої панелі

«Левая панель» і «Правая панель» відповідно



Наприклад, переведення панелі в режим «Быстрый просмотр» дає можливість при переміщенні підсвічування в іншій панелі частково переглядати вміст файлів, на які вказує підсвічування



тим самим завжди легко можна знайти файл, котрій цікавить користувача.

У режимі «Информация» на панель виводиться докладна інформація про підсвічений в іншій панелі файл й про поточну файлову систему.

Використовуючи пункти меню панелей, можна змінити вигляд відображення списку файлів, на наведених вище рисунках використаним є «стандартний» формат, при якому відображаються ім’я файлу, його розмір і час останньої модифікації.

У меню кожної з панелей є команда «Перечитать». За цією командою оновлюється список файлів, котрий відображується у панелі, це буває корисно тоді, коли інші процеси незалежно від mc створюють або видаляють файли.

Ще одне підміню головного меню – «Команда» дозволяє виконати низку операцій з управління файловою системою, а також виконати деякі команди, що змінюють вигляд панелей Midnight Commander і відображувану в панелі інформацію

Команда «Писк файла» (гарячі клавіші <Meta-?> або <Esc><?>) викликає появу діалогового вікна, у якому необхідно вказати параметри файлу для розшуку (можна використовувати стандартні символи ОС для підставлення). Завершення діалогу дозволяє знайти на диску файл із заданими параметрами.

По команді «Отключить панели» (<Ctrl-O>) показується вивід команди оболонки shell, що виконувалась останньою.

По команді «Сравнить каталоги» (<Ctrl-X><D>) порівнюється вміст каталогів, відображуваних на лівій і правій панелях.

Докладний розгляд всіх можливих команд програми виходить далеко за межі даних методичних вказівок і може служити об’єктом самостійної роботи студентів. Тим більше що при роботі із програмою Midnight Commander практично в будь-який момент можна звернутися до інтерактивної підказки, виклик якої здійснюється натисканням клавіші <F1> (<Esc><1>). Підказка організована як гіпертекст, тобто в неї зустрічаються посилання на інші розділи. Такі посилання виділені блакитним фоном. Для переміщення у вікні перегляду підказ-

ки можна використовувати клавіші переміщення курсору або мишу. Крім того завжди можна скористатися системними командами «`man mc`» або «`info mc`» для одержання докладного посібника з користування програмою.

Для виконання ж лабораторних робіт цілком достатньо знати невеликий набір команд та пов'язаних з ними гарячих клавіш. Нижче наведені ці гарячі клавіші й дії, виконувані програмою при їхньому натисканні.

Клавіша або комбінація клавіш	Виконувана дія
<F1>	Виклик контекстно-залежної підказки
<F3>	Перегляд підсвіченого в активній панелі файлу
<F4>	Виклик текстового редактора для підсвіченого файлу
<F5>	Копіювання файлу або групи файлів з поточного каталогу в каталог на другій панелі. При копіюванні одного файлу можна поміняти його ім'я або вказати інший каталог
<F6>	Перенесення файлу або групи відзначених файлів. Як і при копіюванні, можна поміняти ім'я файлу або цільового каталогу
<F7>	Створення підкаталогу в поточному каталозі
<F8>	Видалення файлу/каталогу або групи файлів
<F9>	Виклик головного меню програми
<Tab>	Зміна активної панелі
<Ctrl-O>	Показ виводу останньої команди, що виконувалась
<Alt-?>, <Esc><?>	Пошук файлу
<Ctrl-X><D>	Порівняння каталогів
<Ctrl-X><Q>	Переведення неактивної панелі в режим швидкого перегляду файлів у активній
<Ctrl-X><I>	Переведення неактивної панелі в режим інформації про файли у активній
<Ctrl-X><C>	Перегляд і зміна прав доступу до файлу
<Alt-P>	Переміщення на одну команду назад у переліку команд, що запускалися раніше
<Alt-N>	Переміщення на одну команду вперед в історії команд
<Ctrl-X><T>	Перенесення ім'я підсвіченого файлу до командного рядка
<Ctrl-S>, <Alt-S>	Ініціалізація пошуку файлу в поточному каталозі за першими символами ім'я
<Shift-F4>	Створення й редагування нового файлу
<Insert>	Помітка файлу для групових операцій
<Gray +>	Виділення групи файлів для наступної групової операції
<Gray –>	Зняття виділення групи файлів
<Enter>	Запуск виконуваного файлу на виконання
<F10>, <Esc><Esc>	Вихід з Midnight Commander, відмова від пункту меню або діалогового вікна

При використанні комбінацій клавіш, встановленим має бути латинський регістр клавіатури, регістр великих або маленьких літер значення не має.

Коли в ОС запущений драйвер миші, програма Midnight Commander забезпечує й підтримку миші. Клацанням лівої кнопки можна перемістити підсвічування на будь-який файл кожної з панелей. Для позначки або зняття позначки будь-якого файлу досить клацнути правою кнопкою миші на імені файлу.

Подвійне клацання лівою кнопкою миші на імені файлу означає спробу запустити файл на виконання, якщо це виконувана програма (виконувані файли позначаються символом зірочки «*» перед ім'ям файлу).

Клацанням миші по функціональній кнопці можна також викликати програму, асоційовану з нею. Клацання по команді верхнього меню викликає підменю, що випадає.

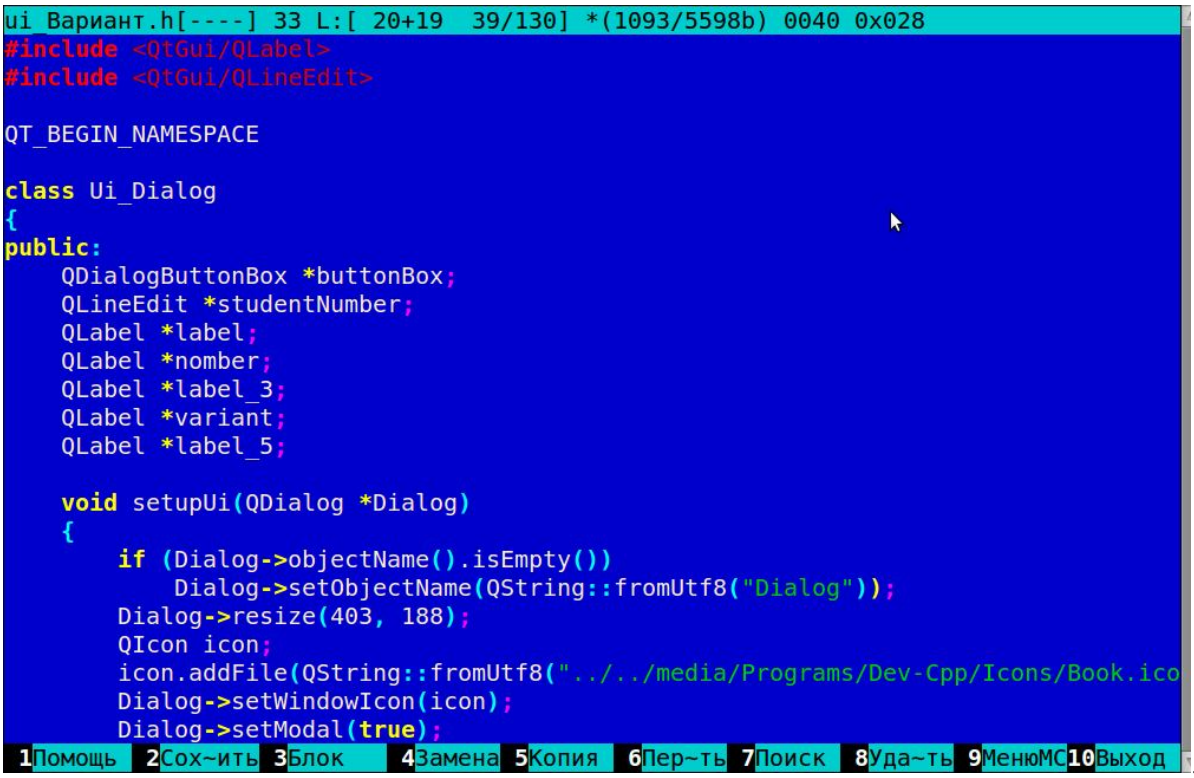
Тому що однією з основних дій при виконанні лабораторних робіт є створення й редагування файлів вихідного коду програм, то необхідно відзначити, що, в порівнянні з роботою через командній рядок, в Midnight Commander особливо зручно здійснювати перегляд, створення й редагування файлів, оскільки для цих цілей у ньому є вбудовані програми.

При роботі в командному рядку доводиться користуватися редактором `vi` або його трохи доробленими нащадками. Однак для тих користувачів, які звикли до текстових редакторів середовища Windows, ці редактори навряд-чи здадуться зручними. Причина в тім, що в них немає звичних меню й підказок і треба пам'ятати безліч клавіатурних комбінацій для уведення команд. Крім того робота в редакторі організована у вигляді двох роздільних режимів – уведення тексту й уведення команд, і новачок часто просто плутає режими. Серед користувачів навіть існує жарт: "Усе, що треба знати про редактор `vi` – це команда «**`:q`**»", тобто закінчити роботу й вийти з редактора.

У свою чергу, вбудований в Midnight Commander текстовий редактор CoolEdit – це проста у використанні програма, котра має майже звичний для більшості користувачів інтерфейс з меню й з більш-менш звичними комбінаціями управляючих клавіш.

Виклик редактора для зміни підсвіченого файлу в середовищі Midnight Commander здійснюється простим натисканням клавіші <F4>. При натисканні ж комбінації <Shift-F4> відкривається порожнє вікно редактора для створення й редагування нового файлу.

Зовнішній вигляд вікна редактора з відкритим для редагування файлом мовою програмування C++ показаний нижче



Цей редактор забезпечує виконання більшості функцій редагування, що є властивими для повноекранних редакторів тексту. З його допомогою можна редагувати файли практично будь-якого розміру, оскільки верхня границя для розміру файлу, що редагується, становить 16 МіБ.

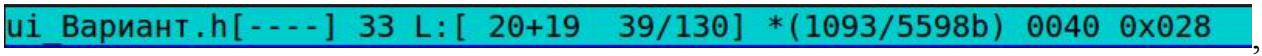
У редакторі підтримуються наступні можливості:

- копіювання, переміщення, видалення, вирізання й вставка блоків тексту;
- скасування попередніх операцій (по комбінації клавіш <Ctrl-U>);
- меню, що випадають;
- вставка файлів;
- макро-визначення;
- пошук і заміна по регулярних виразах (тобто по шаблонах, що сформовані з використанням спеціальних символів), а також власний варіант операцій пошуку й заміни, заснований на функціях scanf – printf;
- виділення тексту комбінацією клавіш <Shift-стрілки> у стилі MS Windows;
- переключення між режимами вставки/заміни символу;
- підсвічування синтаксису програм;
- та інше.

Редактор дуже простий і практично не потребує вивчання. Для того, щоб довідатися, які клавіші викликають виконання певних дій, досить переглянути

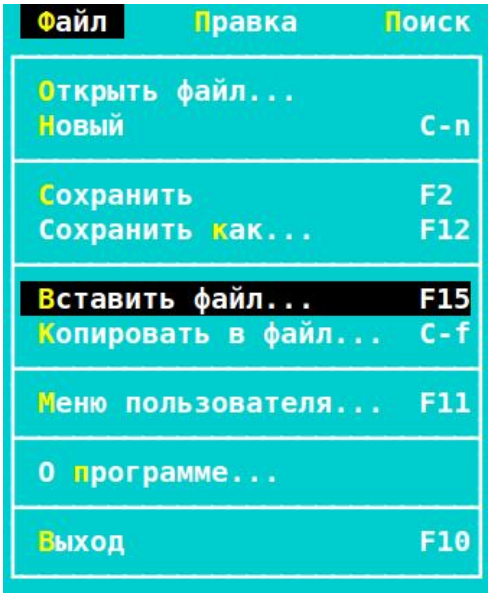
меню, що випадають:

У вихідному стані цей рядок у вікні редактора не показується, на його місці вгорі вікна знаходиться рядок статусу, в якому вказується ім'я файлу, котрий редагується, номер рядка, в яком знаходиться курсор, його позиція в рядку і ASCII код символу у позиції курсору.

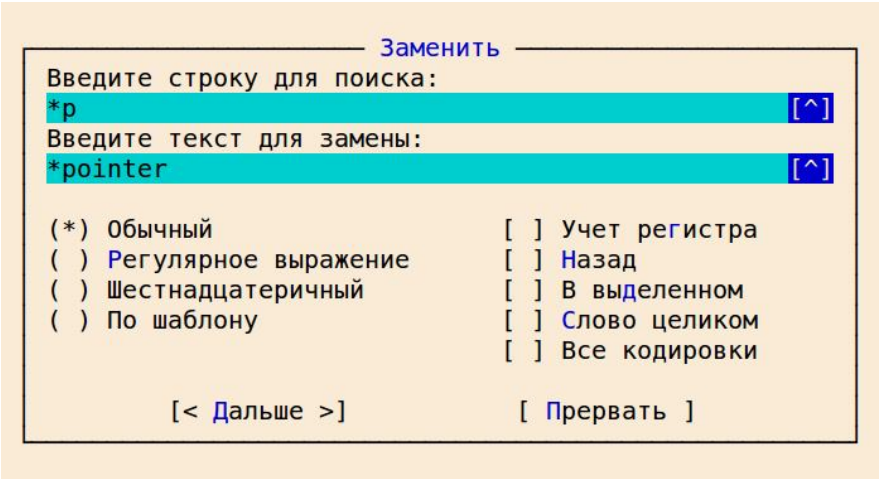


а меню з'являється лише при натисканні клавіші <F9>.

Нижче і на наступній сторінці показані три основних підменю програми – «Файл», «Правка» і «Команда», котрі частіше за все використовуються при роботі в редакторі. Меню «Форматирование» і «Настройка» при написанні тексту програм практично не використовуються, тому докладно розглядати їх не має сенсу. А меню «Поиск» спеціальних пояснень не потребує оскільки є досить простим й має лише три цілком зрозумілі пункти.



Як видно з наведених рисунків, практично кожному пункту меню відповідає або одна з функціональних клавіш <F1> – <F12>, або деяка керуюча комбінація клавіш. Як і раніш у Midnight Commander, функціональним клавішам <F1> – <F10> зіставлені у відповідність екранні клавіші. Натискання на функціональну клавішу або клацання мишею по екранній клавіші викликає реакцію цілком ясну з підписів на них. Так натискання клавіші <F2> викликає збереження файлу, що редагується, під тим же ім’ям, якщо файл новий, то до його збереження пропонується ввести ім’я файлу. Дія клавіші <F4> полягає у виведенні на екран діалогового вікна для здійснення пошуку й заміни тексту



,

натискання на <F7> викликає точно таке ж вікно, але без рядка заміни тексту.

Клавіші <F3>, <F5>, <F6> і <F8> призначені для роботи із блоками виді-

леного тексту, причому робота із блоками за допомогою цих клавіш трохи незвична. Так для виділення тексту необхідно встановити курсор на початок виділюваного блоку й клавішею <F3> позначити його, після цього курсор треба перемістити в кінець блоку й знову натиснути клавішу <F3>, за результатом цих дій блок виділяється і позначається блакитним кольором фона.

Для видалення виділеного блоку достатньо натиснути клавішу <F8>.

Для копіювання або переносу блоку в інше місце тексту потрібно після виділення блоку перемістити курсор у позицію, куди буде провадитися копіювання або переміщення, і натиснути <F5> або <F6> відповідно.

Скорочення управляючих комбінацій, котрі наведені в пунктах меню слід розуміти так:

- літера «M» означає клавішу <Meta> (на клавіатурах IBM це клавіша <Alt>), таким чином, позначення «M-e» – це одночасне натискання клавіш <Alt-E>;
- літера «C» означає клавішу <Ctrl>, тобто «C-n» відповідає натисканню <Ctrl-N>;
- «S» – це клавіша <Shift>, тим самим «S-Ins» означає <Shift-Insert>;
- «F13», «F15» – це натискання <Shift> одночасно з <F3> або <F5> відповідно.

Редактор так само підтримує виконання операцій із блоками в стилі редакторів MS Windows, тобто з використанням наступних комбінацій клавіш:

<Shift-клавіші стрілок> – виділення блоку тексту;

<Ctrl-Insert> – копіювання блоку в буфер обміну (файл cooledit.clip);

<Shift-Insert> – виконує вставку останнього скопійованого в буфер блоку в позицію курсору;

<Shift-Delete> – видаляє виділений блок тексту, запам’ятовуючи його в буфері.

Якщо в системі підтримується миша, то можна спробувати робити копіювання й вставку блоків тексту за допомогою миші. Для цього потрібно нажати клавішу <Shift> і, утримуючи її, виділити потрібну частину тексту, потім відпустити <Shift>, перенести курсор у нове місце, знову нажати <Shift> і клацнути правою кнопкою миші. Заради справедливості треба сказати, що можливо це не спрацює у вікні емулятора термінала.

У решті робота з текстом у редакторі CoolEdit досить проста, але серед інших функцій роботи з текстом можна відзначити видалення рядка, у якому розташований курсор за допомогою сполучення клавіш <Ctrl-Y> і скасування останнього здійсненої дії сполученням <Ctrl-U>, а не <Ctrl-Z> як це прийнято в

редакторах Windows.

Використання сполучення <Ctrl-Z> є категорично не припустимим, оскільки це може призвести до виходу не тільки з редактора, але й до повного закриття й виходу із програми Midnight Commander без збереження результатів редагування.

Для опису всіх функцій убудованого редактора, треба було б занадто багато місця. Але в цьому немає потреби, оскільки для його використання досить запам'ятати, що всі основні операції можна виконати через пункти меню, що викликається натисканням клавіші <F9> у вікні редагування. Крім того, можна прочитати man-сторінку по команді «`man mcedit`» або «`info mcedit`».

1.4.4 Робота з компілятором GCC

В ОС UNIX для компіляції програм використовується оригінальний компілятор C [6], котрий називається cc ("C Compiler"). При розробці проекту GNU оригінальний компілятор C, включений у цей проект, одержав назву gcc ("GNU C Compiler"). Ця абревіатура використовується й дотепер, але її значення дещо змінилося. Зараз "GCC" означає "GNU Compiler Collection", що переводиться як "набір компіляторів GNU" або як "збірний компілятор GNU". Це відбулося в результаті принципового розширення компілятора – включення в нього додаткових мов. Однак базовою структурою GCC як і раніше залишається компілятор C. На щастя, структура мови C власне заснована на досить низькорівневих і досить схожих на апаратні діях. Це дозволяє у верхньому рівні програмного забезпечення для компіляції коду надбудовувати структури компіляторів інших мов на основі, що підтримує мову C.

Розробка компілятора не припиняється. Експериментальна версія обновляється раз у кілька хвилин – вона змінюється безупинно. Деякі виправлення поправляють виявлені помилки, інші – додають нові мови й можливості, а деякі – забирають більш не застосовувані засоби.

За традицією, що є характерною для всіх UNIX-сумісних систем, GCC – це компілятор командного рядка. У різних ОС існує безліч надбудов над компілятором, які дозволяють користувачеві працювати в середовищі графічного оточення, але при цьому сам процес компіляції виконується «поза кадром» за допомогою компілятора GCC. Прикладом такої надбудови може служити описана раніш графічна оболонка Dev-C++, що заснована на GNU компіляторі minGW для ОС Windows

Як вже вказувалось вище, лабораторні роботи під керівництвом ОС Linux

виконуються на віддаленому Linux-сервері за допомогою робочій станції, котра емулірує текстовий термінал. Тому немає можливості застосування графічної оболонки для компілятора GCC і працювати доводиться у режимі командного рядка із зазначенням відповідних опцій.

Відносно необхідності роботи у режимі командного рядка можна навести висловлювання з одного з сайтів, котрий присвячений компіляції програм під керуванням ОС Linux (цитую мовою оригіналу): “Вызов происходит из консоли, если вы не знаете, что это такое, то вам нет смысла читать эту статью, вы всё равно ничего не поймёте, так же мне не понятно, как вы попали на этот сайт – каждый уважающий себя и других человек, считающий себя хотя бы подобием программиста обязан знать об этой вещи.”

Трохи пізніше будуть розглянуті опції командного рядка, котрі застосовуються при компіляції програм мовами C і C++, а поки що декілька загальних зауважень відносно опцій.

Кожна опція командного рядка починається зі знака дефіса «-» або пари дефісів. Приміром, наступна команда компілює програму **muxit.c** мовою C, написану у відповідності зі стандартом ANSI, і створює не скомпонований об’єктний файл із ім’ям **muxit.o**:

```
$ gcc -ansi -c muxit.c -o muxit.o
```

Опція, що складається з однієї букви, яка передує імені, котре відповідає опції, не обов’язково повинна включати пробіл між буквою опції та її параметром. Наприклад, опція **-omuxit.o** тотожна **-o muxit.o.**

Наступна команда використовує **-v** для виводу докладних описів та **--help** для роздрукування списку доступних опцій компілятора, по ній буде виведений вичерпний список опцій командного рядка, що включає й особливі опції для кожної мови:

```
$ gcc -v -ihelp
```

Можлива побудова команди з таким набором опцій, коли не виконується ніяких дій. Наприклад, наступна команда завантажує в компілятор об’єктний файл і потім застосовує опцію **-c**, щоб запобігти виклику компонувальника:

```
$ gcc -c brookm.o
```

Всі опції командного рядка приблизно укладаються в три категорії.

По-перше – специфічні для мови. Компілятор GCC здатний компілювати кілька мов, і деякі опції застосовуються тільки до одної або двох з них.

По-друге – специфічні для платформи. Компілятор GCC здатний генерувати об'єктний код для декількох платформ, деякі опції застосовуються лише тоді, коли створюється код для особливої платформи. Приміром, якщо цільовою платформою є Intel x86, то може бути використаний набір опцій **`-fp -ret -in -387`** щоб визначити, що числа із плаваючою крапкою, котрі повертаються функціями, що викликаються, повинні зберігатися в апаратних регістрах із плаваючою крапкою.

По-третє – загальні. Чимало опцій мають загальне значення для всіх мов і всіх платформ. Наприклад, опція **`-o`** указує компіляторові оптимізувати об'єктний код, що є результатом роботи компілятора.

Слід зауважити, що збірний компілятор GCC працює на багатьох платформах. Під платформою (platform) розуміється сполучення певного комп'ютерного процесора з певною операційною системою, що запускається на ньому.

З тисячі таких апаратно-системних сполучень, виділяються кілька основних і допоміжних платформ, які використовуються для тестування правильності роботи версій GCC, що випускаються. Ці платформи обрані через їхню популярність, і ще тому, що вони цілком представляють властивості інших підтримуваних GCC платформ.

З усією можливою старанністю розроблювачі домагаються правильної роботи GCC на основних платформах, але й чимала уважність також приділяється вторинним платформам.

Платформи тестування GCC

Процесор	Операційна система
Основні платформи	
Alpha	Red Hat Linux
HPPA	HPUX
Intel x86	Debian Linux, Red Hat Linux і FreeBSD
MIPS	IRIX
PowerPC	AIX
Sparc	Solaris
Вторинні платформи	
PowerPC	Linux
Sparc	Linux
ARM	Linux
Intel x86	MS Win32 + Cygwin

Компілятор GCC, як колекція компіляторів, складається з багатьох компонентів. У наступній таблиці представлений лише незначний перелік основних частин GCC, причому не всі з них завжди присутні. Деякі є специфічними до мови програмування, тому, якщо підтримка для відповідній їм мови не встановлена, то таких частин у системі немає.

Різні встановлювані частини компілятора GCC

Частина	Опис
c++	Версія **gcc**, що встановлює C++ як мову за замовчуванням і автоматично підключає стандартні бібліотеки C++ при компонуванні. Те ж саме, що й **g++**.
cc1	Діючий компілятор C.
cc1plus	Діючий компілятор C++.
cygwin1.dll	Поділювана бібліотека для Microsoft Windows, що забезпечує інтерфейс API (Application Programming Interface), емулюючи системні виклики UNIX.
g++	Версія **gcc**, що встановлює C++ як мову за замовчуванням і при компонуванні автоматично підключає стандартні бібліотеки C++. Те ж саме, що й **c++**.
gcc	Основна драйверна програма, що координує виконання компіляторів і компонувальників (linkers) для створення бажаного коду. Підтримує велику кількість опцій
gcj	Драйверна програма, котра використовується для компіляції програм мовою Java.
jc1	Діючий компілятор Java.
libjcc	Ця бібліотека містить функціональні блоки коду (routines), які можуть уважатися частиною компілятора, тому що вони компонуються практично в кожну виконувану програму. Це спеціальні підпрограми (routines), що компонуються до виконуваній програми для виконання таких ґрунтовних завдань, як арифметичні дії над числами із плаваючою крапкою й т.і. Ці функціональні блоки найчастіше залежать від платформи.
libjcj	Ця бібліотека підключається до програми на етапі її виконання (Real Time Library, RTL), і містить класи ядра Java.
libstdc++	Динамічна бібліотека (RTL), що містить класи й функції C++, визначені як частина стандарту мови.

Крім того, з компілятором пов’язана велика кількість додаткового програмного забезпечення для забезпечення процесу компіляції. Частина засобів, що стосуються до цієї категорії, наведенні в таблиці нижче. Деякі з них обов’язково необхідні (такі, як **as** і **ld**), інші, при всій своїй корисності, не є

строго обов’язковими.

Програмні засоби, що використовуються разом з GCC

Засіб	Опис
addr2line	Надаючи адреси, що містяться у виконуваному файлі, **addr2line** використовує налагоджувальну інформацію, що є у файлі, для перетворення адрес в ім’я вихідного файлу і номер рядка. Ця програма входить до пакету binutils.
ar	Програма для підтримки бібліотек. Додає, видаляє й витягає файли з архівів. Частиш за все використовується для створення й обслуговування об’єктних бібліотек, котрі використовуються компонувальником. Входить до пакету binutils.
as	Асемблер GNU. У дійсності являє собою ціле сімейство асемблерів, він здатний працювати з однієї з декількох різних платформ.
f2з	Програма-транслятор з мови Fortran на мову C. У пакет binutils не входить.
gcov	Профілюючий засіб, використовуваний разом з gprof для визначення ділянок програми, що потребують найбільшу частину часу виконання програми.
gdb	налагоджувач (debugger) GNU. Може використовуватися для перевірки дій і значень величин під час виконання програми.
gprof	Цей засіб забезпечує спостереження за виконанням програми, скомпільованої з убудованим до неї профілюючим кодом. Виробляє профіль для оптимізації, повідомляючи час виконання кожної функції. Входить до пакету binutils.
ld	Редактор зв’язків (компонувальник) GNU. Збирає об’єктні файли у готову виконувану програму. Входить до пакету binutils.
make	Утиліта, що читає сценарій компоновки (makefile script) для визначення частин програми, котрі вимагають компіляції й компонування, і потім видає команди, необхідні для таких дій. Вона зчитує сценарій, що зветься make-файлом (makefile або Makefile), і містить визначення відносин і залежностей файлів.
strings	Зчитує файл будь-якого типу й витягає з нього всі обумовлені рядки буквених символів (character strings). Входить до пакету binutils.
Windres	Компілятор для файлів ресурсів вікон (Window Resource Files). Входить до пакету binutils.

Як уже згадувалося вище, основна частина GCC призначена й використовується для компіляції програм, що написані алгоритмічною мовою C, в об’єктні файли, готові до запуску програми й бібліотеки.

Вихідні файли програм і проміжні файли, які виробляються в процесі роботи компілятора, мають цілком певні суфікси імен файлів (користувачі ОС типу Windows говорять “розширення імен файлів”). Перелік суфіксів імен файлів, які так чи інакше пов’язані з компіляцією й компонуванням програм мовою С наведений нижче.

Суфікси імен файлів, пов’язані з мовою С

Суфікс імені	Уміст файлу
.a	Статична об’єктна бібліотека (архів).
.c	Вихідний код мовою С, що підлягає обробці препроцесором.
.h	Вихідний код заголовного файлу мовою С (header file).
.i	Вихідний код мовою С, що не підлягає попередній обробці. Файли цього типу є проміжним продуктом компіляції.
.o	Об’єктний файл у форматі, котрий придатний для компонування (linker). Файли цього типу є проміжним продуктом компіляції.
.s	Асемблерний код. Файли цього типу є проміжним продуктом компіляції.
.so	Поділювана об’єктна бібліотека. (Shared object library)

Як приклади, можна розглянути використання компілятора GCC для компіляції різних варіантів найпростішої програми мовою С – "**hello, world**".

Звичайний приклад вихідного коду такої програми має вигляд:

```
/* helloworld.c */
#include <stdio.h>
int main(int argc.char *argv[]) {
  printf("hello, world\n");
  return(0);
}
```

Найпростішим і самим прямим способом компіляції цієї програми в готову до запуску форму є наступний спосіб. Зберегти цей вихідний код у файлі з ім’ям **helloworld.c** і в командному рядку ОС ввести таку команду:

```
$ gcc helloworld.c
```

Компілятор, перевіривши суфікс ім’я файлу, визначить, що зазначений файл містить вихідний код мовою С. Відповідно до визначеної за замовчуван-

ням послідовності дій GCC компілює вихідний файл в об'єктний, компонує його в готовий до запуску виконуваний файл, і потім видаляє тимчасовий об'єктний файл. У команді не призначене ім'я файлу для вироблюваної виконуваної версії програми, тому компілятор за замовчуванням назве файл **a.out** і збереже його в поточному каталозі. Уведення цього ім'я в командному рядку викличе запуск програми й виведення нею повідомлення:

```
$ a.out
hello, world
```

Для призначення іншого імені файлу готової до запуску програми потрібно використовувати опцію **-o.** За наступною командою буде створена виконувана програма з ім'ям **howdy**:

```
$ gcc helloworld.c -o howdy
```

Уведення цього ім'я програми в командному рядку викличе її запуск і роздрукує її повідомлення:

```
$ howdy
hello, world
```

Використання компілятора з опцією **-c** указує **gcc** скомпілювати вихідний код і залишити на диску тільки об'єктний файл, пропускаючи етап компонування об'єктного файлу в готову до запуску програму. У цьому випадку ім'я виведеного файлу за замовчуванням буде таким же, як і ім'я вихідного, але тільки із суфіксом **.o**. Наприклад, наступна команда створить об'єктний файл із ім'ям **helloworld.o**:

```
$ gcc -c helloworld.c
```

Можливе використання опції **-o** для перепризначення ім'я створюваного об'єктного файлу. За наступною командою буде створений файл з ім'ям **harumph.o**:

```
$ gcc -c helloworld.c -o harumph.o
```

Для побудови об'єктних бібліотек або просто, щоб створити набір

об’єктних файлів для їхнього наступного компонування, можна використовувати одну команду для створення об’єктних файлів з декількох вихідних. По наступній команді будуть створені об’єктні файли з іменами **arglist.o**, **ponder.o** та **listsort.o**:

```
$ gcc -c arglist.c ponder.c listsort.c
```

Якщо програма складається з декількох вихідних файлів, то й у цьому випадку GCC здатний виконати компонування виконуваного файлу автоматично. Приміром, наведений нижче вихідний код збережений у файлі **hellomain.c**, що, у свою чергу, викликає функцію **sayhello()**, збережену у файлі **sayhello.c.**

```
/* hellomain.c */
void sayhello(void);
int main (int argc, char *argv[]) {
  sayhello ();
  return(0);
}

/* sayhello.c */
#include <stdio.h>
void sayhello(void)
{
  printf("hello, world\n");
}
```

По наступній команді буде виконана компіляція двох цих вихідних файлів в об’єктні, компонування їх у готову до запуску програму з ім’ям **hello** й, потім, видалення тимчасових об’єктних файлів:

```
$ gcc hellomain.c sayhello.c -o hello
```

Опція **-E** вказує компіляторові **gcc** запустити тільки препроцесор **cpp** без створення об’єктного або виконуваного файлу. Наступна команда виконає передобробку (preprocessing) вихідного файлу **helloworld.c** і виведе результат на стандартний пристрій виведення.

```
$ gcc -E helloworld.c
```

Можливе використання опції **-o** для направлення результатів передобробки коду у файл. Як було вказано раніше, що вихідний код мовою С, котрий більше не потребує передобробки зберігається у файлі із суфіксом **.i**. Це може бути виконане в результаті застосування такої команди:

```
$ gcc -E helloworld.c -o helloworld.i
```

Опція **-S** указує компіляторові виробити тільки код мовою асемблера й на цьому зупинитися. Наступна команда з вихідного файлу **helloworld.c** створить файл з ім'ям **helloworld.s** мовою асемблера для поточної цільової машини:

```
$ gcc -S helloworld.c
```

Різновид мови асемблер залежить від вибору цільової платформи. При компіляції декількох вихідних файлів виробляється окремий модуль асемблерного коду для кожного з них.

За допомогою компілятора GCC можна створювати статичні бібліотеки, які являють собою набір файлів типу **.o**, створюваних компілятором звичайним шляхом. Компонування програми з об'єктними модулями бібліотеки – те ж саме, що й компонування її з об'єктними файлами, що перебувають у каталозі. Інша назва статичної бібліотеки – архів. Утиліта, що управляє вмістом статичної бібліотеки, називається **ar**.

Для побудови статичної бібліотеки спочатку необхідно скомпілювати всі об'єктні модулі, які повинні ввійти до неї. Наприклад, вихідний код наступної програми втримується у двох файлах **hellofirst.c** і **hellosecond. c**:

```
/* hellofirst.c */
#include <stdio.h>
void hellofirst() {
  printf{"The first hello\n");
}

/* hellosecond.c */
#include <stdio.h>
void hellosecond() {
  printf("The second hello\n");
}
```

Як вже вказувалося раніше, два цих вихідних файли можуть бути скомпільовані у відповідні їм об'єктні файли (із суфіксом **.o**) наступною командою:

```
$ gcc -c hellofirst.c hellosecond.c
```

Для створення нової бібліотеки й включення до неї об'єктних файлів слід використовувати утиліту **ar** з опцією **-r**. Опція **-r** створює вказану бібліотеку, якщо її ще немає, і додає до неї перелічені об'єктні модулі, при необхідності заміняючи вже присутні модулі їхніми новими версіями. Наступна команда створює бібліотеку **libhello.a**, що містить два об'єктних модулі:

```
$ ar -r libhello.a hellofirst.o hellosecond.o
```

Тепер бібліотека цілком закінчена й готова до використання. Наступний приклад програми, що перебуває у файлі **twohellos.c**, викликає обидві функції зі створеної раніше бібліотеки:

```
/* twohellos.c */
void hellofirst(void);
void hellosecond(void);
int main(int argc,char *argv[]) {
  hellofirst();
  hellosecond();
  return(0);
}
```

Програма **twohellos** може бути відкомпільована й скомпонована одною командою компіляторові із вказівкою бібліотеки в командному рядку:

```
$ gcc twohellos.c libhello.a -o twohellos
```

В Unix-системах існує угода про імена статичних бібліотек: вони повинні починатися із трьох букв **lib** і закінчуватися суфіксом **.a**. Всі системні бібліотеки відповідають цій угоді, що дозволяє використовувати в командному рядку скорочену форму імен бібліотек, застосовуючи опцію **-l** ("ель"). Наступний командний рядок відрізняється від попереднього тільки розташуванням, у якому **gcc** буде проводити пошук бібліотеки **libhello.a**:

```
$ gcc twohellos.c -lhello -o twohellos
```

Для компіляції програм, що написані мовою C++ виконуються практично ті ж самі дії, що й для програм мовою C. Компілятор GNU C++ також є повнофункціональним компілятором, що виробляє готовий до виконання об'єктний код на своєму виході. Первісний компілятор C++, випущений компанією AT&T, називався **cfront** і в дійсності був транслятором (прекомпілятором) коду мовою C++ у код C. Дотепер деякі компілятори C++ діють саме так. GCC був споконвічно компілятором C, тому язик C++ був у нього доданий без особливих проблем як обираного режиму компіляції.

Нижче в таблиці наведений список суфіксів імен файлів, задіяних у компіляції й компонуванні (linking) програм мовою C++. Список не э повним і містить тільки найбільше уживані суфікси.

Суфікси імен файлів, пов'язаних з компіляцією C ++

Суфікс	Уміст файлу
.a	Статична об'єктна бібліотека (архів).
.c, **.c++**, **.cc**, **.cp**, **.cpp**, **.cxx**	Вихідний код мовою C++, що підлягає обробці препроцесором.
.h	Заголовний файл, що включається в програми, (header file) мовою C або C++.
.ii	Вихідний код програми мовою C++, не підлягаючий передобробці. Файли цього типу виробляються на Проміжному етапі компіляції.
.o	Об'єктний файл у форматі, котрий підтримується компонувальником (linker). Файли цього типу виробляються на проміжному етапі компіляції.
.s	Вихідний код на асемблері. Файли цього типу виробляються на проміжному етапі компіляції.
без суфікса	Поділювана (динамічна) об'єктна бібліотека. (Shared object library.) Стандартні динамічні бібліотеки основного набору функцій мови C++ мають імена без суфікса.

Як і раніше, як приклад, можна розглянути вихідний код найпростішої програми, але вже мовою C++. Код збережений у файлі **helloworld.cpp**:

```
/* helloworld.cpp */
#include <iostream>
int main(int argc,char *argv[]) {
  std::cout << "hello, world\n";
  return(0);
}
```

Для видачі окремого рядка на стандартний пристрій виводу ця програма використовує функцію **cout**, визначену в стандартному заголовному файлі **iostream**. У готову для запуску форму ця програма може бути скомпільована наступною командою:

```
$ g++ helloworld.cpp
```

Компілятор **g++** розпізнає файл по суфіксу його імені як вихідний файл мовою C++. Відповідно до діючої угоди він повинен скомпілювати указаний вихідний файл в об'єктний, далі скомпонувати об'єктний файл у готову до запуску машинну програму, і потім видалити об'єктний файл. У зв'язку з тим, що в командному рядку не визначене ім'я для вихідного файлу, за замовчуванням буде створений файл із ім'ям **a.out**. Команда для запуску готової програми і її вихід виглядають так:

```
$ a.out
hello, world
```

Ім'я для виконуваного системою файлу, у який повинен бути скомпільований вихідний, передається компіляторові опцією **-o**. Наступна команда створить файл із ім'ям **helloworld**:

```
$ g++ helloworld.cpp -o helloworld
```

Уведення цього імені в командному рядку призведе до виконання програми:

```
$ helloworld
hello, world
```

Програма **g++** є особливим варіантом утиліти **gcc**, вона встановлює C++ як основну мову компіляції, що призводить до автоматичного використання стандартних бібліотек мови C++ замість застосовуваних за замовчуванням бібліотек мови C. При використанні угод про імена бібліотек і зазначенні відповідної бібліотеки в командному рядку можлива компіляція й компонування програми мовою C++ і командою **gcc**. Наприклад, такої:

```
$ gcc helloworld.cpp -lstdc++ -o helloworld
```

Опція **-l** (“ель”) заміняє наступне за нею ім’я, підставляючи до нього префікс **lib** і суфікс **.a**, що перетворює ім’я указаної бібліотеки в **libstdc++.a**. Потім пошук бібліотеки з таким ім’ям проводиться в каталогах для стандартних бібліотек. Подальший процес компіляції й вихідний файл точно відповідають дії команди **g++**.

На багатьох системах при інсталяції GCC створюється програма з ім’ям **c++**. Її використання є цілком рівносильним дії **g++**. Ось приклад такої команди:

```
$ c++ helloworld.cpp -o helloworld
```

Якщо в командному рядку вказується кілька вихідних файлів, то всі вони компілюються й компонуються разом в один виконуваний файл. Нижче наведений приклад програми, яка складається з 3-х файлів:

1. заголовного файлу (header file) з ім’ям **speak.h**, цей файл містить визначення класу, котрий має одну єдину функцію:

```
/* speak.h */
#include <iostream>
class Speak {
  public:
    void sayHello(const char *);
};
```

2. файлу **speak.cpp**, що містить код реалізації функції **sayHello()**:

```
/* speak.cpp */
#include "speak.h"
  void Speak::sayHello(const char *str) {
    std::cout << "Hello " << str << "\n";
}
```

3. файл **hellospeak.cpp** містить програму, котра використовує клас Speak:

```
/* hellospeak.cpp */
#include "speak.h"
int main(int argc,char *argv[]) {
  Speak speak;
  speak.sayHello("world");
  return(0);
}
```

Для компіляції й компонування цих трьох файлів у єдиний виконуваний файл використовується одна команда:

```
$ g++ hellospeak.cpp speak.cpp -o hellospeak
```

Для компіляції вихідного коду із придушенням компонування й, відповідно, виведення об'єктного файлу замість виконуваного, застосовується опція **-c**. За замовчуванням ім'я створюваного файлу збігається з ім'ям вихідного, тільки має суфікс **.o**. Наприклад, наступною командою вихідний файл **hellospeak.cpp** буде скомпільований в об'єктний файл **hellospeak.o**:

```
$ g++ -c hellospeak.cpp
```

Програма **g++** також розпізнає файли із суфіксом **.o** у якості вхідних і передає їх компонувальнику (linker):

```
$ g++ -c hellospeak.cpp
$ g++ -c speak.cpp
$ g++ hellospeak.o speak.o -o hellospeak
```

Варто помітити, що опція **-o** використовується не тільки для присвоєння імен виконуваним файлам. Її також можна використовувати для призначення імен і інших вихідних файлів компілятора. Наприклад, у результаті наступної послідовності команд буде створений той же файл із виконуваним кодом, що й у попередньому прикладі. З тією лише різницею, що імена проміжних об'єктних файлів будуть іншими:

```
$ g++ -c hellospeak.cpp -o hspk1.o
$ g++ -c speak.cpp -o hspk2.o
$ g++ hspk1.o hspk2.o -o hellospeak
```

Призначення опції **-E** вказує драйверу **g++** пропустити вихідний код через препроцесор **cpp** і не робити ніяких подальших дій. Наступна команда виконає послідовно передобробку вихідного коду з файлу **helloworld.cpp** і видачу результату передобробки на стандартний пристрій виведення:

```
$ g++ -E helloworld.cpp
```

Вихідний код програми у файлі **helloworld.cpp** складається всього із шести рядків і ця програма тільки й робить, що виводить рядок тексту. Але на виході препроцесора буде більш ніж 1200 рядків. Так багато тому, що буде включений заголовний файл **iostream.h**, а він, у свою чергу, включить до коду ще кілька заголовних файлів. У них визначається кілька великих класів, відповідальних за уведення й виведення.

Суфікс ім'я файлу, прийнятий в GCC для результатів передобробки – **.ii**. Такий файл може бути вироблений при використанні опції **-o**, як у наступному прикладі:

```
$ gcc -E helloworld.cpp -o helloworld.ii
```

Опція **-S** указує компіляторові згенерувати код мовою асемблера й на цьому зупинитися. Наступна команда створює файл із ім'ям **helloworld.s** з асемблерним кодом з вихідного файлу мовою C++:

```
$ g++ -S helloworld.cpp
```

Створений асемблерний код залежить від типу цільової платформи. Якщо переглядати отриманий у результаті такої компіляції код, то в ньому можна побачити не тільки виконувані інструкції й визначення даних, але також і адресні таблиці, що необхідні для забезпечення спадкування й компонування програм мовою C++.

Статична бібліотека (static library) – це архівний файл, що містить набір створених компілятором об'єктних файлів. Частини, що складають бібліотеку можуть містити використовувані функції, визначення класів і об'єкти, що є входженнями екземплярів (instances) визначень класів. У дійсності все, що може втримуватися в об'єктному файлі із суфіксом **.o**, може бути поміщене в бібліотеку.

У наступному прикладі створюються два об'єктних модулі, котрі викори-

стовуються потім для складання статичної бібліотеки. Заголовний файл (header file) містить необхідні програмі функції, визначення класів і об'єкти.

Заголовний файл **say.h** містить прототип функції **sayHello()** і визначення класу з ім'ям **Say**. Далі приводиться вміст цього файлу:

```
/* say.h */
#include <iostream>
void sayhello(void);
class Say {
  private:
    char *string;
  public:
    Say(char *str) {
    string = str;
}
void sayThis(const char *str) {
std::cout << str << " from a static library\n";
}
void sayString(void);
};
```

Наступний файл з ім'ям **say.cpp** – вихідний файл першого із двох об'єктних модулів, які повинні бути поміщені в бібліотеку. У ньому міститься тіло визначення функції **sayString()** класу **Say**. Він також містить оголошення об'єкта **librarysay**, що є екземпляром класу **Say**:

```
/* say.cpp */
#include "say.h"
void Say::sayString() {
  std::cout << string << "\n";
}
Say librarysay("Library instance of Say");
```

Файл **sayhello.cpp** містить вихідний код другого модуля, що також повинен бути поміщений у бібліотеку. У ньому перебуває визначення функції **sayhello()**:

```
/* sayhello.cpp */
#include "say.h"
void sayhello() {
  std::cout << "hello from a static library\n";
}
```

Наведена далі послідовність команд виконує компіляцію двох вихідних файлів в об'єктні й потім зберігає їх за допомогою утиліти **ar** у статичній бібліотеці:

```
$ g++ -c sayhello.cpp
$ g++ -c say.cpp
$ ar -r libsay.a sayhello.o say.o
```

Утиліта **ar**, використана з опцією **-r**, створює нову бібліотеку з ім'ям **libsay.a** і поміщає до неї перераховані в команді об'єктні модулі. При такому способі застосування **ar** створює нову бібліотеку із зазначеним ім'ям, якщо такий ще не існує. При наявності зазначеної бібліотеки утиліта **ar** заміняє в ній існуючі об'єктні модулі їхніми новими версіями.

Далі – уміст головного вихідного файлу програми **saymain.cpp**, що використовує статичну бібліотеку **libsay.a**:

```
/* saymain.cpp */
#include "say.h"
int main(int argc,char *argv[]) {
  extern Say librarysay;
  Say localsay = Say("Local instance of Say");
  sayhello();
  librarysay.sayThis("howdy");
  librarysay.sayString();
  localsay.sayString();
  return(0);
}
```

Ця програма компілюється й компонується наведеною далі командою. У команді визначається, що драйвер верхнього рівня компілятора **g++** дозволяє всі зовнішні посилання, наявні в **saymain.cpp**, через пошук визначень для необхідних символічних імен у бібліотеці **libsay.a**:

```
$ g++ saymain.cpp libsay.a -o saymain
```

Зовнішнє відношення до **librarysay** посилається на об'єкт, оголошений в **libsay.cpp** і зберігається в бібліотеці **libsay.a**. Як **librarysay.sayThis()**, так і **librarysay.sayString()** є викликами методів об'єкта бібліотеки. Також й **sayhello()** є викликом функції з

об’єктного файлу **sayhello.o**, що також є в цій бібліотеці. При запуску програми відбувається виведення наступних рядків:

```
hello from a static library
howdy from a static library
Library instance of Say
Local instance of Say
```

Крім статичних бібліотек можливе створення й поділюваних або динамічних бібліотек. Поділювана бібліотека (shared library) також містить набір об’єктних файлів. Тільки її об’єктні модулі повинні використовувати відносну адресацію, щоб їхній код міг бути динамічно завантажений у будь-яку область пам’яті й запущений без його надлишкового переміщення. Це дозволяє завантажувати виконується код, що, з поділюваної бібліотеки під час виконання програми, замість його статичного компонування у виконуваний файл.

Наведений далі заголовний файл із ім’ям **average.h** визначає клас, що повинен бути збережений у поділюваній бібліотеці:

```
/* average.h */
class Average {
  private:
   int count; double total;
  public:
    Average(void) {count = 0; total = 0.0;
    }
    void insertValue(double value);
    int getCount(void);
    double getTotal(void);
    double getAverage(void);
};
```

Вихідний файл **average.cpp**, що також повинен бути скомпільований і збережений у поділюваній бібліотеці, містить код функцій, визначених у класі **Average**:

```
/* average.cpp */
#include "average.h"
void Average::insertValue(double value){
  count++;
  total += value;
}
```

```
int Average::getCount(){
  return(count);
}
double Average::getTotal(){
  return(total);
}
double Average::getAverage(){
  return(total / (double)count);
}
```

Вихідні файли компілюються в об'єктні, які потім використовуються для побудови поділюваної бібліотеки. Це відбувається при виконанні наступних двох команд:

```
$ g++ -c -fpic average.cpp
$ gcc -shared average.o -o average.so
```

Перша команда використовує опцію **-c**, тому компілятор виробляє об'єктний файл **average.o** і не вживає спроби його компонування в здійсненний машинний код. Опція **-fpic** ("pic" – скорочення від англ. "Position Independent Code" – "незалежний від положення код") указує компіляторові виробляти придатний для поділюваної бібліотеки код. Тобто такий код, у якому внутрішня адресація обчислюється відносно початкової адреси завантаження модуля, коли з'являється необхідність у ньому. Друга команда використовує опцію **-shared**, що приводить до створення поділюваної бібліотеки з ім'ям **average.so** (зазначеним опцією **-o**). Для другої команди можна застосувати як команду **g++** , так і **gcc.** Створення поділюваної бібліотеки з об'єктних файлів при компіляції C++ не має ніяких особливих відмінностей від загальної процедури. Для складання бібліотеки з декількох файлів досить їх просто перелічити в командному рядку.

Дві попередні команди можна об'єднати в одну. По ній вихідні файли компілюються в об'єктні, з яких потім створюється поділювана бібліотека:

```
$ g++ -fpic -shared average.cpp -o average.so
```

Наступна програма використовує визначення класу, що перебуває в поділюваній бібліотеці, для створення екземпляра об'єкта, що застосовується для додавання в нього 4-х значень і розрахунку їхнього середнього:

```
/* showaverage.cpp */
#include <iostream>
#include "average.h"
int main(int argc,char *argv[]) {
  Average avg;
  avg.insertValue(30.2);
  avg.insertValue(88.8);
  avg.insertValue(3.002);
  avg.insertValue(11.0);
  std::cout << "Average=" << avg.getAverage() << "\n";
  return(0);
}
```

Далі наведена команда для компіляції й компонування програми з поділюваною бібліотекою. По цій команді виробляється файл **showaverage** з виконуваним машинним кодом:

```
$ g++ showaverage.cpp average.so -o showaverage
```

При запуску програми бібліотека **average.so** повинна бути встановлена в каталозі, що є дозволеним для пошуку поділюваної бібліотеки під час виконання програми.

Нижче у таблиці наведені найбільш часто використовувані опції GCC:

Опція	Призначення
-Wall	Вивести всі попередження при компіляції
-c	Скомпілювати об’єктний файл
-o	Указати ім'я підсумкового файлу, використання: **-o Output**, **-o Output.exe**, **-o Output.so**, **-o Output.dll**, **-o Output.lib**, **Output.o**
-I	Указати директорію для підключення заголовних файлів
-L	Указати директорію для підключення бібліотек
-l	Підключити бібліотеку (суфікс не пишеться), використання: **-lLibrary** без пробілу після літери **l**
-s	Скомпілювати Shared library (.so, .dll)

Процес компіляції досить простих прикладів, що розглянуті вище, як видно й сам не дуже складний. Але для більш-менш складних проектів задача компіляції доволі ускладнюється. Наприклад, необхідно написати програму звичайного текстового редактора. Такий проект вже може складатися з великої кількості файлів. Нехай для створення виконуваного файлу **edit** були написа-

ні наступні файли **main.c**, **kbd.c**, **command.c**, **display.c**, **insert.c**, **search.c**, **files.c** та **utils.c**. Крім того ще три заголовних файлів: **defs.h**, **command.h** та **buffer.h**.

Причому заголовний файл **defs.h** включається в усі файли з вихідним текстом. Заголовний файл **command.h** включається тільки в ті вихідні файли, які стосуються команд редагування, а файл **buffer.h** – тільки в "низькорівневі" файли, що безпосередньо оперують буфером редагування. Цілком є очевидним, що й у цьому випадку, можна скористатися наведеними вище командами і скомпілювати виконуваний файл програми, наприклад так:

```
$ gcc main.c kbd.c command.c display.c \
      insert.c search.c files.c utils.c -o edit
```

Але при такому підході можливе виникнення ситуації, коли компілятор не врахує змін внесених у який-небудь із заголовних файлів. Тому бажано, спочатку окремо створювати об'єктні файли, а потім зв'язувати їх у здійсненний файл:

```
$ gcc main.c kbd.c command.c display.c \
      insert.c search.c files.c utils.c
$ gcc main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o -o edit
```

Але й такий підхід має дуже істотний недолік, що полягає в тому, що різко збільшується час компіляції, оскільки компілюються відразу всі об'єктні модулі, хоча зміни могли бути внесені лише в деякі з них або навіть в один. Роздільне ж створення об'єктних модулів теж не рятує становища. Завдання створення виконуваного файлу, котре здається дуже простим, ще більше ускладнюється, якщо проект додається створення власних статичних або динамічних бібліотек і підключення до проекту зовнішніх бібліотек.

Як видно зі сказаного, компілювати й збирати проект таким чином досить незручно й утомливо, тому що можна забути відкомпілювати якийсь модуль, навіть якщо саме в ньому були зроблені всі зміни. Можна написати скрипт, що буде робити це автоматично, але в цьому випадку простіше й швидше буде використати утиліту **make** і написати **Makefile**.

1.4.5 Призначення програми make

Утиліта **make** [7][1] автоматично визначає які частини великої програми повинні бути перекомпільовані, і виконує необхідні для цього дії. У даному розділі описується програма **GNU make**, авторами якої є Richard Stallman і Roland McGrath. Починаючи з версії 3.76, розробку програми веде Paul D. Smith.

GNU make задовольняє вимогам розділу 6.2 стандарту IEEE Standard 1003.2-1992 (POSIX.2).

У прикладах, що наводяться далі, передбачається використання програм мовою С. Однак, **make** може бути використовувана з будь-якою мовою програмування для якої є компілятор, що працює з командного рядка. Насправді, область застосування **make** не обмежується тільки зборкою програм. Її можна використовувати для вирішення будь-яких завдань, де одні файли повинні автоматично оновлятися при зміні інших файлів.

Перед тим, як використовувати **make**, необхідно створити так званий make-файл (**makefile**), котрий описує залежності між файлами програми, і містит команди для відновлення цих файлів. Як правило, виконуваний файл програми залежить від об'єктних файлів, які, у свою чергу, утворюються у результаті компіляції відповідних файлів з вихідними текстами.

Після створення потрібного make-файлу, для виконання всіх необхідних перекомпіляцій, якщо які-небудь із вихідних файлів програми були змінені, достатньо надати просту команду:

```
make
```

Використовуючи інформацію з make-файлу, і, знаючи час останньої модифікації файлів, утиліта **make** вирішує, які з файлів потребують оновлення. Для кожного із цих файлів будуть виконані указані в make-файлі команди.

Зі сказаного є зрозумілим, що для роботи з утилітою **make**, необхідним є так званий make-файл (**makefile**), що містить опис потрібних дій. Як правило, make-файл описує, яким чином потрібно компілювати й компонувати програму.

Make-файл також може інструктувати **make** як виконувати ті або інші дії, коли явно буде викликане їхнє виконання (наприклад, видалити певні файли у

[1]Є переклад за Інтернет адресою: http://www.linuxlib.ru/prog/make_379_manual.html

відповідь на команду «очистка»).

При компіляції програми, будь-який файл із вихідним текстом, що був модифікований, повинен бути відкомпільований заново. Якщо був модифікований який-небудь із заголовних файлів, то, щоб уникнути проблем, повинні бути перекомпільовані всі вихідні файли, які містили в собі цей заголовний файл. Кожна компіляція вихідного файлу породить нову версію відповідного йому об'єктного файлу. І, нарешті, якщо які-небудь із вихідних файлів були перекомпільовані, то всі об'єктні файли (як «нові», так і ті, що залишилися від попередніх компіляцій) повинні бути заново скомпоновані для одержання нової версії виконуваного файлу програми.

Простий make-файл складається з «правил» (rules) наступного виду:

```
ціль: пререквізит ...
    команда
    ...
    ...
```

За звичай, ціль (target) являє собою ім'я файлу, що генерується в процесі роботи утиліти **make**. Прикладом можуть слугувати об'єктні й виконуваний файли програми, що підлягає зборці. Ціль також може бути ім'ям деякої дії, яку потрібно виконати (наприклад, **clean** – очистити).

Пререквізит (prerequisite) – це файл, що використовується як вихідні дані для досягнення цілі. Дуже часто ціль залежить відразу від декількох файлів.

Команда – це дія, виконувана утилітою **make**. У правилі може міститися кілька команд – кожна на своє власному рядку. Важливе зауваження: рядки, що містять команди обов'язково повинні починатися із символу табуляції! Не виконання цієї вимоги є характерною помилкою для багатьох починаючих користувачів.

За звичай, команди знаходяться у правилах із пререквізитами й служать для створення файлу-цілі, якщо який-небудь із пререквізитів був модифікований. Однак, правило, що має команди, не обов'язково повинне мати пререквізити. Наприклад, правило з метою «**clean**» («очистка»), що містить команди видалення, може не мати пререквізитів.

Правило (rule) описує, коли і яким образом треба оновляти файли, котрі зазначені в ньому як ціль. Для створення або відновлення цілі, **make** виконує зазначені в правилі команди, використовуючи пререквізити в якості вихідних даних. Правило також може описувати яким чином повинна виконуватися деяка дія.

Крім правил, make-файл може містити й інші конструкції, однак, простий make-файл може складатися з одних лише правил. Правила можуть виглядати більше складними, чим наведений вище шаблон, однак всі вони більш-менш відповідають йому за структурою.

Нижче наведений приклад простого make-файлу, у якому описується створення виконуваного файлу **edit**, котрий, як було описано раніш, залежить від восьми об'єктних файлів, які, у свою чергу, залежать від восьми відповідних вихідних файлів і трьох заголовних файлів.

```
edit: main.o kbd.o command.o display.o insert.o \
      search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
main.o: main.c defs.h
    cc -c main.c
kbd.o: kbd.c defs.h command.h
    cc -c kbd.c
command.o: command.c defs.h command.h
    cc -c command.c
display.o: display.c defs.h buffer.h
    cc -c display.c
insert.o: insert.c defs.h buffer.h
    cc -c insert.c
search.o: search.c defs.h buffer.h
    cc -c search.c
files.o: files.c defs.h buffer.h command.h
    cc -c files.c
utils.o: utils.c defs.h
    cc -c utils.c
clean:
    rm edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
```

Для того, щоб за допомогою цього make-файлу створити виконуваний файл **edit**, достатньо у командному рядку системи надати команду:

```
make
```

Для видалення з директорії проекту виконуваних і об'єктних файлів необхідно набрати:

```
make clean
```

У наведеному прикладі, цілями, зокрема, є об'єктні файли **main.o** і **kbd.o**, а, також, виконуваний файл **edit**. До пререквізитів належать такі файли, як **main.c** і **defs.h**. Кожний об'єктний файл, фактично, є одночасно й ціллю й пререквізитом. Прикладами команд можуть служити **cc -c main.c** та **cc -c kbd.c**.

У випадку, якщо ціль є файлом, цей файл повинен бути перекомпільований або перекомпонований щораз, коли був змінений який-небудь із його пререквізитів. Крім того, будь-які пререквізити, які самі генеруються автоматично, повинні бути оновлені першими. У наведеному прикладі виконуваній файл **edit** залежить від восьми об'єктних файлів; об'єктний файл **main.o** залежить від вихідного файлу **main.c** і заголовного файлу **defs.h**.

Кожному рядку, котрий містить ціль і пререквізити, слідує рядок з командою. Ці команди вказують, яким чином треба оновляти цільовий файл. На початку кожного рядка, що містить команду, повинен знаходитись символ табуляції. Саме наявність символу табуляції є ознакою, по якій **make** відрізняє рядки з командами від інших рядків make-файлу. Треба майти на увазі, що **make** не має ні найменшого уявлення про те, як працюють ці команди. Тому, відповідальність за те, що виконувані команди потрібним образом оновлять цільовий файл, цілком лягає на програміста. Утиліта **make** просто виконує зазначені в правилі команди якщо ціль має потребу у відновленні.

Ціль **clean** є не файлом, а ім'ям дії. Оскільки, при звичайній зборці програми ця дія не потрібно, ціль **clean** не є пререквізитом якого-небудь із правил. Отже, **make** не буде виконувати цього правила, якщо про це не попросити спеціально. Слід зазначити, що це правило не тільки не є пререквізитом, але й саме не містить яких-небудь пререквізитів. Таким чином, єдине призначення даного правила – виконання указаних у ньому команд. Цілі, які є не файлами, а іменами дій називаються абстрактними цілями (phony targets).

За замовчуванням, **make** починає свою роботу з першої зустрінутої цілі (крім цілей, чиє ім'я починається із символу «**.**» крапка). Ця ціль буде головною ціллю за замовчуванням (default goal). Головна ціль (goal) – це ціль, котру **make** прагне досягти як результат своєї роботи.

У прикладі, що наведений вище, головна мета полягає у відновленні виконуваного файлу **edit**, тому дане правило розташоване на початку make-файлу.

Таким чином, коли дається команда:

```
make
```

make читає make-файл із поточної директорії й починає його обробку з першого зустрінутого правила. У прикладі це правило забезпечує перекомпонування виконуваного файлу **edit**. Однак, перш ніж **make** зможе повністю обробити це правило, їй потрібно обробити правила для всіх файлів, від яких залежить **edit**. У цьому випадку – від всіх об'єктних файлів програми. Кожний із цих об'єктних файлів обробляється відповідно до свого власного правила. Ці правила говорять, що кожний файл із розширенням **.o** (об'єктний файл) утворюється у результаті компіляції відповідного йому вихідного файлу. Така компіляція повинна бути виконана, якщо вихідний файл або який-небудь із заголовних файлів, що перераховані як пререквізити, є «більш новими», ніж об'єктний файл, або такого об'єктного файлу взагалі не існує.

Інші правила обробляється завдяки тому, що їхні цілі прямо або побічно є пререквізитами для головної цілі. Якщо яке-небудь правило жодним чином не «зв'язано» з головною ціллю (тобто ні прямо, ні побічно не є його пререквізитом), то це правило не обробляється. Щоб виконати такі правила, доводиться явно вказувати **make** на необхідність їхньої обробки, наприклад, подібним образом:

```
make clean.
```

Перед перекомпіляцією об'єктного файлу, **make** розглядає необхідність оновлення його пререквізитів, у цьому випадку – файлу з вихідним текстом і заголовними файлами. У make-файлі прикладу не міститься ніяких інструкцій з відновлення цих файлів – файли з розширеннями **.c** та **.h** не є цілями яких-небудь правил. Таким чином, утиліта **make** не робить ніяких дій із цими файлами. Однак, **make** могла б автоматично оновляти й вихідні тексти, якби вони, наприклад, генерувалися за допомогою програм, подібних до Bison або Yacc, і для них були б визначені відповідні правила.

Після перекомпіляції об'єктних файлів, які мають потребу в цьому, **make** приймає рішення – потрібно чи ні перекомпоновувати файл **edit**. Це потрібно робити, якщо для файлу **edit** не існує об'єктних файлів або який-небудь з об'єктних файлів є більш «свіжим» у порівнянні з ним. Якщо який-небудь із об'єктних файлів тільки що був відкомпільований заново, то він буде «більш молодий», ніж файл **edit**. Відповідно, файл **edit** буде перекомпонований.

Таким чином, якщо був модифікований файл **insert.c** а потім запуще-

на **make**, цей файл буде скомпільований заново для відновлення об'єктного файлу **insert.o**, і, потім, файл **edit** буде перекомпонований. Якщо, наприклад, був змінений файл **command.h** і запущена **make**, то будуть перекомпільовані об'єктні файли **kbd.o**, **command.o** і **files.o**, а потім виконуваний файл **edit** буде скомпонований заново.

У наведеному вище прикладі, у правилі для **edit** довелося перераховувати двічі список об'єктних файлів програми:

```
edit: main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Подібне дублювання чревате помилками. При додаванні в проект нового об'єктного файлу, можна додати його в один список і забути про інший. Але подібний ризик можна усунути, і, одночасно, спростити make-файл, використовуючи змінні. Змінні (variables) дозволяють, один раз визначивши текстовий рядок, потім використовувати їх багаторазово в потрібних місцях.

Звичайною практикою при побудові make-файлів є використання змінної з ім'ям **objects**, **OBJECTS**, **objs**, **OBJS**, **obj**, або **OBJ**, котра містить список всіх об'єктних файлів програми. Подібну змінну з ім'ям **objects** можна визначити у такий спосіб:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Далі, щораз, коли з'являється необхідність у списку об'єктних файлів, то можна використовувати значення цієї змінної за допомогою запису **$(objects)**.

От так виглядає наведений раніше приклад з використанням змінної для зберігання списку об'єктних файлів:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit: $(objects)
    cc -o edit $(objects)
main.o: main.c defs.h
    cc -c main.c
kbd.o: kbd.c defs.h command.h
```

```
    cc -c kbd.c
command.o: command.c defs.h command.h
    cc -c command.c
display.o: display.c defs.h buffer.h
    cc -c display.c
insert.o: insert.c defs.h buffer.h
    cc -c insert.c
search.o: search.c defs.h buffer.h
    cc -c search.c
files.o: files.c defs.h buffer.h command.h
    cc -c files.c
utils.o: utils.c defs.h
    cc -c utils.c
clean:
    rm edit $(objects)
```

Треба визнати, що це нововведення не дуже спростило make-файл, але, насправді, немає необхідності явного зазначення команд компіляції окремо для кожного з вихідних файлів. Утиліта **make** сама може «здогадатися» про використання потрібних команд, оскільки в неї є, так зване, неявне правило (implicit rule) для відновлення файлів з розширенням **.o** з файлів з розширенням **.c**, за допомогою команди **cc -c**. Наприклад, вона б використовувала команду

```
cc -c main.c -o main.o
```

для перетворення файлу **main.c** у файл **main.o**. Таким чином, можна усунути явне зазначення команд компіляції в правилах, що описують побудову об'єктних файлів.

Коли файл із розширенням **.c** автоматично використовується подібним чином, він також автоматично додається до переліку пререквізитів «свого» об'єктного файлу. Тобто можна цілком усунути файли з розширенням **.c** зі списків пререквізитів об'єктних файлів.

Таким чином, новий варіант make-файлу, у який внесені обидві зміни, що описани вище, а також використовується змінна **objects** має вигляд:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit: $(objects)
    cc -o edit $(objects)
main.o: defs.h
kbd.o: defs.h command.h
```

```
command.o: defs.h command.h
display.o: defs.h buffer.h
insert.o: defs.h buffer.h
search.o: defs.h buffer.h
files.o: defs.h buffer.h command.h
utils.o: defs.h
.PHONY: clean
clean:
    rm edit $(objects)
```

Приблизно так і виглядають make-файли в реальній практиці. Для правила з **clean** тут використаний більш складний запис, котрий буде пояснений нижче.

Якщо для створення об'єктних файлів використовуються тільки неявні правила, то можна рекомендувати інший стиль написання make-файлів. У такому make-файлі записи групуються по їхніх пререквізитах, а не по їхнім цілям. От як може виглядати подібний make-файл:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
    cc -o edit $(objects)
$(objects): defs.h
kbd.o command.o files.o: command.h
display.o insert.o search.o files.o: buffer.h
```

Тут, заголовний файл **defs.h** оголошується пререквізитом для всіх об'єктних файлів програми. Файли **command.h** та **buffer.h** є пререквізитами для перерахованих об'єктних файлів.

Який стиль побудови make-файлів вибрати – є справою уподобання кожного програміста окремо. Альтернативний стиль більше компактний, однак він подобається не всім – багато хто вважає більш «природним» розташовувати інформацію про кожну ціль в одному місці, а не «розпорошувати» її по make-файлу.

Компіляція програми – не єдина річ, для якої можна написати правила. Часто, в make-файлі вказується, яким чином можна виконати деякі інші дії, що не відносяться до компіляції програми. Такою дією, наприклад, може бути видалення всіх об'єктних і виконуваних файлів програми для очищення каталогу в якому здійснюється компіляція.

Раніш в прикладі проекту програми вже вказувалось як пишеться прави-

ло для очищення каталогу:

```
clean:
    rm edit $(objects)
```

На практиці, швидше за все, це правило записується трохи складнішим способом, котрий припускає можливість непередбачених ситуацій:

```
.PHONY: clean
clean:
    rm edit $(objects)
```

Такий запис запобігає можливій плутанині коли, раптом, у каталозі міститься файл із ім'ям **clean**, а також дозволяє **make** продовжити роботу, навіть якщо команда **rm** завершиться з помилкою.

Подібне правило не слід поміщати в початок make-файлу, оскільки не бажано, щоб воно запускалося «за замовчуванням»! У прикладі, дане правило поміщено у кінець make-файлу, щоб головною ціллю за замовчуванням залишалося складання файлу **edit**.

Оскільки **clean** не є пререквізитом цілі **edit**, це правило не буде виконуватися, якщо викликати **make** без аргументів. Для запуску даного правила, як вже вказувалось раниш, потрібно буде набрати **make clean**.

Все сказане вище можна підсумувати наступним. Make-файл може складатися з конструкцій п'яти видів:

- явні правила;
- неявні правила;
- визначення змінних;
- директиви make-файлу;
- коментарі.

Явне правило (explicit rule) описує, коли і яким чином треба оновляти файли, котрі є цілями правила. У цьому правилі перераховуються файли, від яких залежить ціль правила (так звані пререквізити), а також можуть бути задані команди, які необхідно використати для створення або оновлення цілі.

Неявне правило (implicit rule) описує, коли і яким чином потрібно оновляти деяку групу файлів, імена котрих підходять під певний шаблон. Таке правило описує, як ціль може залежати від файлу з «подібним» ім'ям і задає команди для оновлення цілей.

Визначення змінної (variable definition) – це рядок make-файлу, у якому

змінній привласнюється певне текстове значення. Далі, це значення може бути «підставлене» у потрібному місці тексту. У прикладі make-файлу, змінна **objects** визначається як перелік об'єктних файлів програми.

Директива вказує програмі **make** на необхідність виконання деякої спеціальної дії під час інтерпретації make-файлу. Зокрема можливі, наприклад, такі дії:

– читання іншого make-файлу;

– прийняття рішення (на основі значення змінних) про використання або ігнорування частини make-файлу;

– визначення багаторядкової змінної, що складається з декількох рядків;

– та інші.

Символ «**#**» позначає початок коментарю. Весь текст, починаючи із цього символу й до кінця рядка, буде ігнорований. Коментар може бути продовжений у наступному рядку за допомогою одиночного символу зворотної косої риси «****», що перебуває наприкінці рядка. Коментарі можуть перебувати практично в будь-якому місці make-файлу за декількома виключеннями. Вони не можуть перебувати усередині директиви **define** і, можливо, усередині команд (оскільки, тут уже інтерпретатор командного рядка буде вирішувати – що саме є коментарем). Рядок make-файлу, що цілком є коментарем, розглядається як порожній й ігнорується.

За замовчуванням, коли make шукає make-файл для обробки, вона по черзі намагається знайти в поточному каталозі файли з наступними іменами (у зазначеному порядку): **GNUmakefile**, **makefile** і, на сам кінець, **Makefile**.

За звичай, має сенс давати make-файлу ім'я **makefile**, або **Makefile**. При чому рекомендується використовувати ім'я **Makefile**, тому що при виведенні вмісту каталогу, файл із таким ім'ям буде розташовуватися на початку списку, поряд з такими важливими файлами як **README**. Ім'я **GNUmakefile** – не може бути рекомендоване для більшості make-файлів. Це ім'я можна використовувати, якщо make-файл є специфічним тільки для **GNU make** і не буде оброблятися іншими версіями **make**.

У тому випадку, якщо **make** не може знайти файлів з перерахованими вище іменами, то вона намагається продовжити роботу без використання make-файлу. У такому випадку, при виклику **make**, програміст повинен явно вказати головну ціль й утиліта спробує досягти цієї цілі, використовуючи тільки «убудовані» до неї неявні правила.

Якщо є потреба у використовуванні «нестандартного» імені для make-

файлу, то його можна вказати у командному рядку, використовуючи опції «**-f**» або «**--file**». Аргументи «**-f ім'я_файлу**» або «**--file=ім'я_файлу**», указують програмі **make** на необхідність використовування файлу з ім'ям **ім'я_файлу** як make-файлу. У командному рядку можна задати обробку відразу декількох make-файлів, перелічивши їх у команді **make** з допомогою декількох опцій «**-f**» або «**--file**». Всі зазначені в такий спосіб make-файли логічно «поєднуються» у тім порядку, як вони були задані в командному рядку. При наявності в командному рядку опцій «**-f**» або «**--file**», автоматичного пошуку make-файлів з іменами **GNUmakefile**, **makefile** і **Makefile**, не виконується.

Програма **GNU make** працює по двопрохідній схемі. На першому проході робиться читання всіх make-файлів (у тому числі й тих, що підключаються), у ході якого вся інформація, котра міститься в них (змінні і їхні значення, явні й неявні правила) переводиться у внутрішнє представлення й будується граф залежностей для всіх цілей і їхніх пререквізитів. Далі, на другому проході, це внутрішнє представлення використовується для визначення того, які саме цілі мають потребу у відновленні й виконуються відповідні правила.

Розуміння такої двопрохідної схеми є дуже важливим, оскільки ця схема безпосередньо впливає на хід обчислення змінних і функцій; нерозуміння її, найчастіше, є джерелом непорозумінь при написанні make-файлів.

Треба чітко розуміти, що розділи цілей і пререквізитів обробляються негайно, а обробка команд, використовуваних для оновлення цілі, завжди відкладається. Це загальне правило діє для явних правил, шаблонових правил, суфіксних правил, статичних шаблонових правил і при простому визначенні пререквізитів.

Правила (rules) містяться в make-файлі й описують, коли і яким чином повинні будуть оновлені або створені деякі файли, котрі називаються цілями (targets). Найчастіше, кожне правило містить тільки одну ціль. У правилі перераховуються файли, які є пререквізитами (prerequisites) для цієї цілі й команди, які повинні бути виконані для створення або оновлення цілі.

Порядок розташування правил усередині make-файлі не має значення. Виняток становить лише вибір головної цілі за замовчуванням (default goal) – цілі, до якої буде прагнути **make**, якщо вона не задана явно. За замовчуванням, головною ціллю стає ціль з першого правила в першому оброблюваному make-файлі. Якщо це правило містить кілька цілей, то тільки перша з них стає головною ціллю. Тут є два винятки. По-перше, головними цілями, обираними за замовчуванням, не можуть стати цілі, ім'я яких починається із крапки

(якщо тільки вони не містять принаймні одного символу «**/**»). І, по-друге, із процесу вибору головної цілі виключаються шаблонові правила.

Тому, за звичай make-файли пишуться таким чином, щоб перше правило описувало процес збирання готової програми, або всіх програм, котрі описані у цьому make-файлі (часто, для цього використовується ціль з ім'ям **all**).

У загальному виді, правило виглядає так:

```
цілі: пререквізити
    команда
    ...
```

або так:

```
цілі: пререквізити; команда
    команда
    ...
```

Цілі (targets) – це імена файлів, які розділюються пробілами. В іменах цілей можуть бути використані шаблонові символи. Для файлів, що містяться в архіві, може бути використана спеціальна форма запису: «a(m)», де «a» – це ім'я архівного файлу, а «m» – ім'я файлу, що міститься в ньому. За звичай, у правилі міститься тільки одна ціль, однак, іноді має сенс задати кілька цілей в одному правилі.

Рядки, що містять команди, повинні починатися із символу табуляції. Перша команда може розташовуватися або в рядку із пререквізитами (і відділятися від них крапкою з комою), або в наступному рядку після пререквізитів (саме цей рядок повинен починатися із символу табуляції). В обох випадках, результат буде той самий.

Оскільки знак долара використовується для посилання на змінні, для використання його в правилах, потрібно писати «**$$**». Довгі рядки make-файлу можуть бути розділені на частині за допомогою символу «****», котрий вміщується наприкінці рядка. Це може підвищити зручність читання make-файлу, але «технічної» необхідності в цьому немає – **make** ніяк не обмежує довжину рядків make-файлу.

Правило містить інформацію про дві речі: коли треба вважати, що ціль «застаріла», і яким чином вона може бути оновлена при виникненні такої необхідності.

Критерій «старіння» обчислюється стосовно пререквізитів, які представ-

ляють із себе імена файлів, що розділені пробілами. В іменах пререквізитів можуть використовуватися шаблонові символи. Пререквізити також можуть бути файлами, що знаходяться в архівах. Ціль вважається «застарілою», якщо такого файлу не існує, або він «старше», чим який-небудь із пререквізитів (перевіряється час останньої модифікації файлу). Смисл тут у тім, що, оскільки цільовий файл будується на основі інформації з файлів-пререквізитів, то зміна хоча б одного з них може призвести до того, що вміст цільового файлу вже не буде «правильним».

Команди вказують на те, яким чином треба оновляти ціль. Це – просто рядки (з деякими додатковими можливостями), що виконуються інтерпретатором командного рядка.

При використанням шаблонових символів (wildcard characters), за допомогою одного ім'я можна задати цілу групу файлів. В make шаблоновими символами є «*****», «**?**» і «**[...]**» (як в оболонці Bourne). Наприклад, шаблон ***.c** буде відповідати всім файлам із суфіксом **.c**, котрі знаходяться в поточній директорії.

Символ «**~**» на початку ім'я файлу, також має спеціальне значення. Одиночний символ «**~**» або сполучення «**~/**» означає домашній каталог. Наприклад, вираз **~/bin** буде означати **/home/you/bin**. Якщо відразу за символом «**~**» слідує деяке ім'я, такий рядок являє собою домашню директорію користувача із цим ім'ям. Наприклад, рядок **~john/bin** буде означати **/home/john/bin**. У системах, де користувачі не мають свого домашнього каталогу (таких як MS Windows), така поведінка може емулюватися за допомогою установки змінної оточення **HOME**.

Шаблони також можуть бути корисні як пререквізити правил. У наступному прикладі, команда **make print** викличе друк всіх файлів з вихідними текстами (файлів з розширенням **.c**), які були модифіковані з того часу, як останній раз робилась роздруківка їх подібним чином:

```
print: *.c
     lpr -p $?
     touch print
```

У даному правилі, ціль **print** є порожньою ціллю (empty target file). Автоматична змінна **$?** використовується для друку тільки тих пререквізитів, які були змінені.

При завданні змінної, розкриття шаблонів не провадиться. Наприклад,

якщо записати:

```
objects = *.o
```

то значенням змінної **objects** буде рядок ***.o**. Однак, якщо значення змінної **objects** використовується в цілі, у пререквізиті або в команді, то в момент використання шаблона, буде зроблене його розширення. Щоб привласнити змінної **objects** значення, отримане після розширення шаблона, треба використовувати функцію **wildcard**:

```
objects:= $(wildcard *.o)
```

2 ЛАБОРАТОРНА РОБОТА №1
СТВОРЕННЯ ПРОГРАМНОГО ЕМУЛЯТОРА МЕНЕДЖЕРА ПАМ'ЯТІ

Мета роботи – ознайомлення студентів з принципами роботи менеджера віртуальної пам'яті, створення за допомогою методики об'єктно-орієнтованого моделювання та парадигми об'єктно-орієнтованого програмування засобами мови С++ простішого емулятора менеджера з однорівневою таблицею віртуальних сторінок.

Література до заняття: [1] – стор. 184 – 196, [2] – стор. 232 – 283, [3] – стор. 353 – 382 (розташовані у мережі кафедри та у електронній бібліотеці ОДЕКУ).

Оцінка роботи

Максимальна кількість балів за роботу становить:

- теоретична частина 100 балів;
- практична частина 100 балів.

Остаточна оцінка визначається за методикою, описаної в загальній частині даних методичних вказівок.

2.1 Теоретична частина

У середині минулого сторіччя програмісти витрачали дуже багато часу на те, щоб втиснути свої програми в наявну в їхньому розпорядженні дорогу й дуже малу пам'ять. Нерідко доводилося використовувати алгоритм, що працює набагато повільніше іншого алгоритму, оскільки кращий алгоритм був занадто великим за розміром й програма, у якій використовувався цей кращий алгоритм, могла не поміститися у пам'ять комп'ютера. У теперішній час програмістам не треба замислюватися про розмір пам'яті. що надається в їх розпорядження. Причому не треба думати, що всі суперсучасні комп'ютери мають пам'ять, вимірювану тисячами мебібайт. Комп'ютери, що працюють у системах реального часу найчастіше мають «величезну» пам'ять розміром від 2-х до 64-х кілобайт. Крім того розмір оперативної пам'яті, що операційна система виділяє тому або іншому процесу прямо залежить від завантаження комп'ютера й, найчастіше, становить 1 – 2 мебібайта для завдання яке вимагає пари сотень мебібайт. Але програмісти навіть не помічають такого обмеження у ресурсах, котрі їм потрібні, справедливо думаючи, що їм повинен бути доступним весь адресний простір, надаваний процесором (наприклад, 2^{32} або 2^{64} адрес).

Таке ставлення до розміру оперативної пам'яті стало доступним тому, що

ще в 1961 році група дослідників з Манчестера (Англія) запропонувала метод, що зараз називається «віртуальною пам'яттю», і повністю звільняє програміста від величезної кількості нудотної роботи з розподілу пам'яті під свої проекти. Уперше цей метод був використаний у низці комп'ютерів, випущених в 60-е роки. До початку 70-х років віртуальна пам'ять з'явилася в більшості комп'ютерів. У теперішній час навіть персональні комп'ютери на мікропроцесорах, мають дуже складні системи віртуальної пам'яті.

В основі віртуальної пам'яті лежить ідея про поділ понять «адресного простору» і «адрес пам'яті». Для приклада можна розглянути уявлюваний комп'ютер, команди якого мають 16-бітне поле адреси, а ОЗП складає 4096 слів. Число слів пам'яті, що адресуються, залежить тільки від числа бітів адреси й ніяк не пов'язане з числом реально доступних слів, тому програма, що працює на такому комп'ютері, мала б можливість звертатися до адресного простору в 65536 слів пам'яті, однак у дійсності комп'ютер має набагато менше слів у пам'яті.

До винаходу віртуальної пам'яті доводилося проводити жорстку відмінність між тими адресами, які менше 4096, і тими, які рівні або більші за 4096. Ці дві частини розглядалися як корисний адресний простір і даремний адресний простір (адреси вищі за 4095 – даремні, оскільки вони не відповідають реальним адресам пам'яті). Інакше кажучи, розбіжність між адресним простором і адресами пам'яті не проводилася, оскільки між ними малась на увазі взаємоодноначна відповідність.

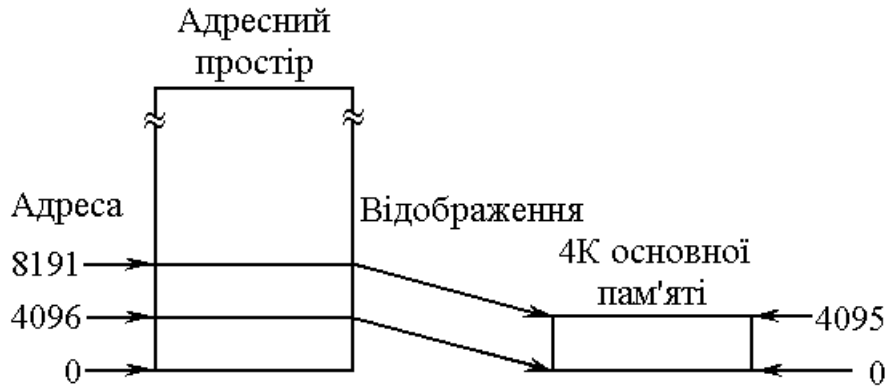


Рисунок 1 – Відображення віртуальних адрес від 4096 до 8191
в адреси основної пам'яті від 0 до 4096.

Ідея ж віртуальної пам'яті, як видно з рис. 1, полягає в тому, що в будь-який момент часу можна одержати прямий доступ до 4096 слів пам'яті, але це

не значить, що вони неодмінно повинні відповідати адресам пам'яті від 0 до 4095. Наприклад, можна повідомити комп'ютеру, що при звертанні до адреси 4096 повинне використовуватися слово пам'яті з адресою 0, при звертанні до адреси 4097 – слово з адресою 1, нарешті, при звертанні до адреси 8191 – слово з адресою 4095 і т.д. Інакше кажучи, так обумовлюється відображення адресного простору в дійсні адреси пам'яті, таке відображення схематично представлене на рисунку.

Тепер можна розглянути випадок, коли програма зробить перехід в одну з адрес від 8192 до 12287? У машині з віртуальною пам'яттю буде виконана наступна послідовність кроків:

1. слова від 4096 до 8191 будуть розміщені на диску;
2. слова від 8192 до 12287 будуть завантажені в основну пам'ять;
3. зміниться відображення адрес; тепер адреси від 8192 до 12287 відповідають коміркам пам'яті від 0 до 4095;
4. виконання програми продовжиться так, начебто нічого не сталося.

У машині ж без віртуальної пам'яті відбудеться помилка, і виконання програми зупиниться.

Така технологія автоматичного накладення називається «сторінковою організацією пам'яті», а частини програми, які зчитуються з диска, називаються «сторінками».

Можливі й інші, більш складні способи відображення адрес із адресного простору в реальні адреси пам'яті. Адреси, до яких програма може звертатися, називають «віртуальним адресним простором», а реальні адреси пам'яті в апаратному забезпеченні – «фізичним адресним простором». Для віртуальної пам'яті потрібен диск для зберігання повної програми та всіх її даних. Природно, що при зміні копії повинен змінюватися й оригінал. Передбачається, що на диску є досить місця для зберігання якщо не повного віртуального адресного простору то, принаймні, тієї його частини, що використовується в цей момент.

Програми пишуться так, начебто в основній пам'яті вистачає місця для розміщення всього віртуального адресного простору, навіть якщо це не відповідає дійсності. Програми можуть завантажувати слова з віртуального адресного простору або записувати слова у віртуальний адресний простір, незважаючи на те, що, насправді, фізичній пам'яті для цього не вистачає. Програміст може писати програми, навіть не усвідомлюючи, що віртуальна пам'ять існує. Просто створюється таке враження, що обсяг пам'яті даного комп'ютера досить великий.

Необхідно підкреслити й обов'язково звернути увагу на той факт, що

сторінкова організація пам’яті тільки створює повну ілюзію великої лінійної основної пам’яті такого ж розміру, що й адресний простір. У дійсності основна пам’ять може бути менша (або більша), ніж віртуальний адресний простір. З програми ні як не можна визначити те, що пам’ять великого розміру просто моделюється за допомогою сторінкової організації. При звертанні до будь-якої адреси завжди з’являються необхідні дані або потрібна команда. Цей механізм є повністю прозорим.

Операційна система створює ілюзію, що всі віртуальні адреси підтримуються реальною пам’яттю, навіть якщо це неправда. Тільки розроблювачам операційних систем і тим, хто вивчає операційні системи, потрібно знати, як створюється така ілюзія.

Віртуальний адресний простір розбивається на низку сторінок рівного розміру, зазвичай, від 512 байт до 64 КіБ, хоча іноді зустрічається й сторінки розміром в 4 МіБ. Розмір сторінки завжди дорівнює ступеню двійки. Фізичний адресний простір теж розбивається на частині точно такого ж розміру, як і сторінки. Ці частини основної пам’яті називаються «сторінковими кадрами».

Для того, щоб правильно співвіднести віртуальні адреси з фізичними адресами, будується так звана таблиця сторінок. Відображення повинно бути зроблене, оскільки, в остаточному підсумку, пам’ять сприймає тільки реальні адреси й не сприймає віртуальні. Кожний комп’ютер з віртуальною пам’яттю містить пристрій для здійснення відображення віртуальних адрес на фізичні. Цей пристрій називається «контролером управління пам’яттю» (MMU – Memory Management Unit). Він може розташовуватися як безпосередньо на мікросхемі процесора, так і на окремій мікросхемі. Для з’ясування того, як будується таблиця віртуальних сторінок і як працює контролер пам’яті, можна розглянути уявлюваний контролер управління пам’яттю, що, наприклад, відображає 32-бітну віртуальну адресу в 15-бітну фізичну адресу. Це означає, що йому потрібно мати 32-бітний вхідний регістр адреси й 15-бітний вихідний регістр.

Рис 2, наведений на наступній сторінці, ілюструє принцип роботи такого контролера управління пам’яттю. Коли в контролер надходить 32-бітна віртуальна адреса, він поділяє цю адресу на 20-бітний номер віртуальної сторінки і 12-бітне зміщення усередині цієї сторінки (передбачається, що сторінки у прикладі мають розмір 4 КіБ). Номер віртуальної сторінки використовується як індекс у таблиці сторінок для знаходження потрібної сторінки. У прикладі номер віртуальної сторінки дорівнює 3, тому з таблиці вибирається третій елемент.

З наведених даних випливає, що менеджер повинен управляти 2^{20} (при-

близно мільйоном) віртуальних сторінок при наявності всього 8 сторінкових кадрів. Очевидно, що не всі віртуальні сторінки можуть одночасно знаходитися в пам’яті. Контролер управління пам’яттю спочатку перевіряє біт присутності потрібної сторінки в пам’яті в теперішній момент. У прикладі цей біт дорівнює 1, тобто сторінка перебуває в пам’яті.

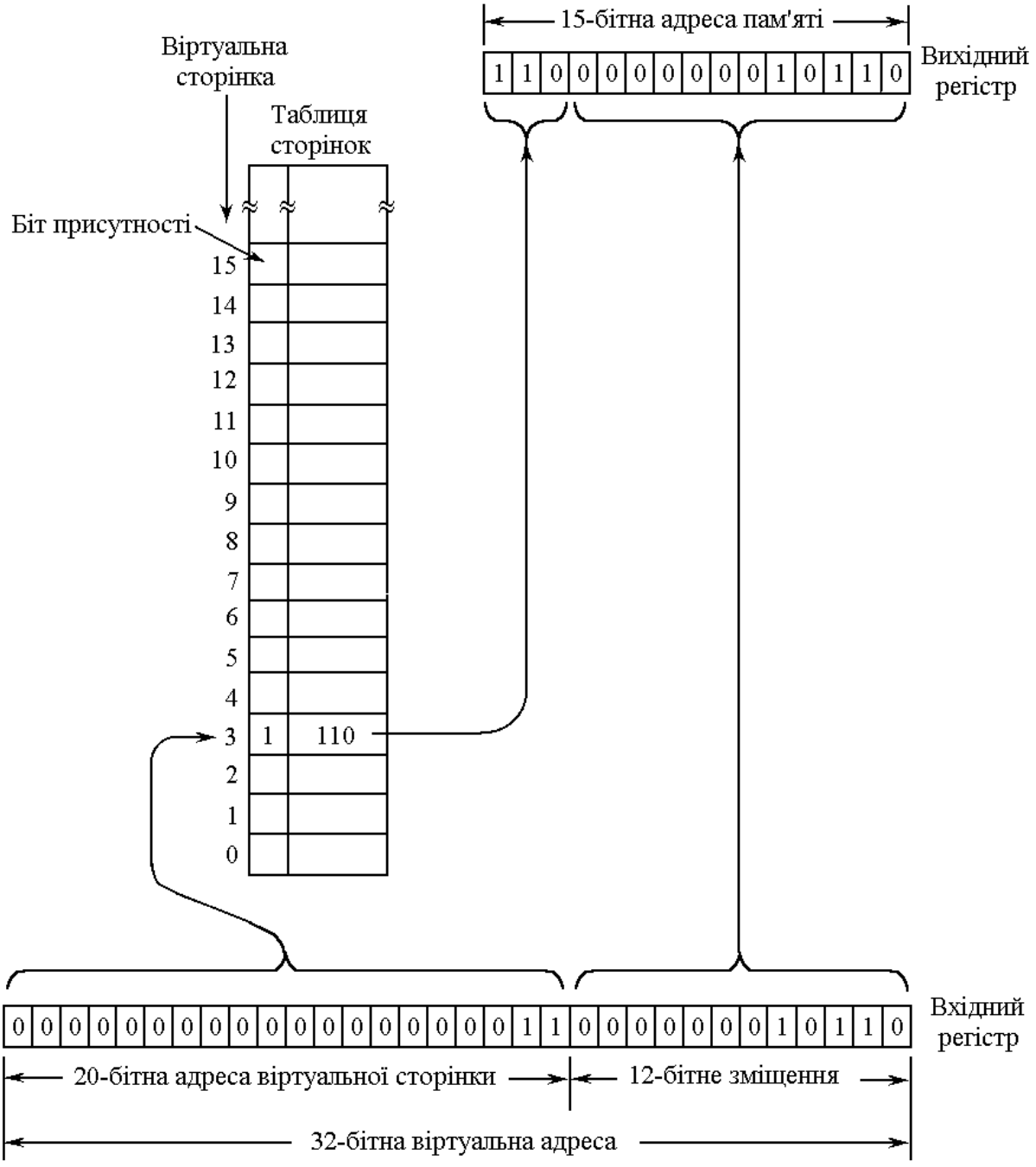


Рисунок 2 – Формування адреси основної пам’яті з адреси віртуальної пам’яті

Далі з обраного елемента таблиці береться значення сторінкового кадру (у прикладі – 6) і копіюється в старші три біти 15-бітного вихідного регістра.

Необхідна наявність саме трьох бітів, тому що у фізичній пам’яті є 8 сторінкових кадрів. Паралельно з цією операцією молодші 12 бітів віртуальної адреси (поле зміщення всередині сторінки) копіюються в молодші 12 бітів вихідного регістра. Потім отримана 15-бітна адреса виставляється на шину адреси.

Таким чином, відображення віртуальних сторінок у фізичні сторінкові кадри може бути зовсім довільним. Нижче на рис. 3 показане одне з таких можливих відображень. Віртуальна сторінка 0 знаходиться в сторінковому кадрі 1. Віртуальна сторінка 1 – у сторінковому кадрі 0. Сторінки 2 немає в основній пам’яті. 3-я сторінка міститься в сторінковому кадрі 2. Віртуальної сторінки 4 немає в пам’яті. Сторінка 5 поміщається в кадрі 6 і т.д.

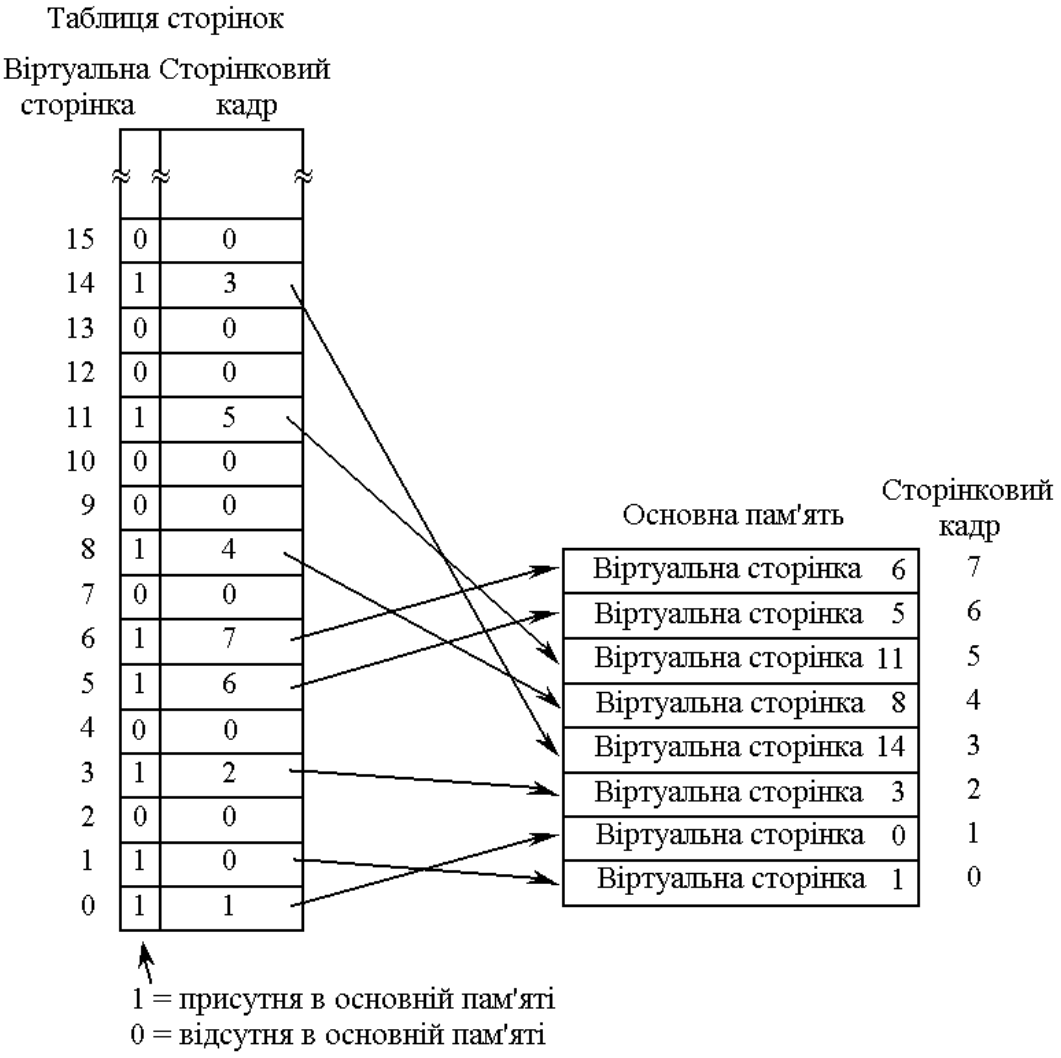


Рисунок 3 – Можливе відображення перших 16 віртуальних сторінок в основну пам’ять, що має 8 сторінкових кадрів

При звертанні до адреси, що знаходиться у сторінці якої немає в основній пам’яті, відбувається помилка через відсутність сторінки. У випадку такої по-

милки операційна система зчитує потрібну сторінку з диска, уводить у таблицю сторінок нову адресу фізичної пам'яті, а потім повторює команду, що викликала помилку.

На машині з віртуальною пам'яттю можна запустити програму навіть у тому випадку, коли жодній частини програми немає в основній пам'яті. Просто таблиця сторінок повинна вказувати, що абсолютно всі віртуальні сторінки перебувають у допоміжній пам'яті. Якщо центральний процесор намагається викликати першу команду, він відразу одержує помилку через відсутність сторінки, у результаті чого сторінка, що містить першу команду, завантажується у пам'ять й вноситься до таблиці сторінок. Після цього починається виконання першої команди. Якщо перша команда містить дві адреси й обидві ці адреси знаходяться на різних сторінках, причому обидві сторінки не є сторінкою, у якій перебуває команда, то відбудеться ще дві помилки через відсутність сторінки й ще дві сторінки будуть перенесені в основну пам'ять до завершення команди. Наступна команда може викликати ще кілька помилок і т.д.

Такий метод роботи з віртуальною пам'яттю називається «викликом сторінок на вимогу». При виклику сторінок на вимогу сторінки переносяться в основну пам'ять тільки якщо в цьому виникає потреба, але не заздалегідь.

Цілком очевидно, що використання алгоритму виклику сторінок на вимогу, має сенс тільки на самому початку при запуску програми. Коли програма проробить якийсь час, потрібні сторінки вже будуть зібрані в основній пам'яті.

Фахівцями з операційних систем було помічено, що більшість звертань до пам'яті зазвичай належать до невеликого числа сторінок. Тобто в кожний момент часу існує набір сторінок, які використовувалися при деякій кількості останніх звернень. Такий набір сторінок називають «робочою множиною». Робоча множина зазвичай змінюється дуже повільно.

В ідеалі набір сторінок (робоча множина), які постійно використовуються програмою**,** можна було б зберігати в пам'яті, щоб скоротити кількість помилок через відсутність сторінок. Однак навіть у цьому випадку програма може звернутися до сторінки, якої немає в основній пам'яті, і контролеру віртуальної пам'яті буде необхідно викликати її з диска. Отже в цьому разі, у пам'яті потрібно звільнити для неї місце, а для цього на диск потрібно відправити якунебудь іншу сторінку.

Вибирати таку сторінку просто навмання не можна, і, вочевидь, потрібний алгоритм для визначення, яку саме сторінку необхідно усунути з пам'яті. Більшість операційних систем намагаються завбачувати, які зі сторінок у пам'яті є найменш корисними в тому розумінні, що їхня відсутність не вплине

сильно на хід програми. Іншими словами, замість того щоб усувати сторінку, що незабаром знадобиться, операційна система намагається вибрати таку сторінку, котра не буде потрібна довгий час.

Існує безліч алгоритмів, що здійснюють вибір непотрібних сторінок [1 – 3], але в лабораторній роботі використовуються тільки два з них, що здійснюються найбільш просто.

Цілком логічно для вибору сторінки, що усувається, використовувати алгоритм FIFO (First-in First-out – першим прийшов – першим пішов). По цьому алгоритму усувається та сторінка, що завантажувалася раніш за всіх, незалежно від того, коли востаннє відбувалося звертання до цієї сторінки.

Для реалізації алгоритму з кожним сторінковим кадром зв'язується лічильник звертань. Від початку всі лічильники встановлюються у 0.

Після кожної помилки через відсутність сторінок, лічильники всіх сторінок, що перебувають у пам'яті, збільшуються на 1, а лічильник викликаної сторінки приймає значення 0. Коли потрібно вибрати сторінку для усунення, вибирається сторінка із самим великим значенням лічильника. Оскільки сторінка не завантажувалася у пам'ять дуже давно, існує велика ймовірність того, що вона не знадобиться найближчим часом.

Можна застосовувати й інший алгоритм – усувається та сторінка, котра використовувалася найбільш давно, оскільки ймовірність того, що вона буде в поточній робочій множині, дуже мала. Цей алгоритм називається LRU (Least Recently Used – алгоритм усунення елементів, що використовувалися дуже давно). Реалізація цього алгоритму, а точніше сказати його різновиду, що називається NFU (Not Frequently Used – сторінка, що рідко використовувалася) виконується за допомогою лічильника аналогічного лічильнику в алгоритмі FIFO. Але тепер скидання лічильника до 0 виробляється не при завантаженні сторінки у пам'ять, а просто при звертанні до неї. Зрозуміло, що й зараз максимальне значення лічильника буде мати сторінка, звертання до якої виробляється рідше за все.

Необхідно розуміти, що у випадку коли розмір робочої множини більш, ніж число доступних сторінкових кадрів, жоден з існуючих алгоритмів не дає гарних результатів, і помилки через відсутність сторінок виникають доволі часто, має місце так звана «пробуксовка» (thrashing). Очевидно, що пробуксовка є дуже небажаним явищем. І навпаки, навіть якщо програма використовує великий віртуальний адресний простір, але має невелику робочу множину, що змінюється повільно і вміщується в основну пам'ять, нічого страшного не відбувається. Це твердження має силу, навіть якщо програма використовує в сто ра-

зів більше слів віртуальної пам’яті ніж їх є у фізичній основній пам’яті.

Крім вибору сторінки, яку можна вивантажити з пам’яті, менеджер пам’яті повинен вирішити ще одну задачу: чи треба зберігати на диску сторінку, котра вивантажується? Дійсно, якщо сторінка, яку потрібно усунути, не змінювалася з тих пір, як її зчитали (наприклад, це цілком імовірно, якщо сторінка містить програму, а не дані), то необов’язково записувати її назад на диск, оскільки точна копія там вже існує. У противному випадку, якщо сторінка змінювалася, то копія на диску вже їй не відповідає, і її потрібно оновити.

У багатьох контролерах управління пам’яттю, урахування зміни сторінки здійснюється додаванням ще одного біта стану для кожної сторінки, котрий дорівнює 0 при завантаженні сторінки й набуває значення 1, коли мікропрограма або апаратне забезпечення змінюють цю сторінку. По цьому бітові операційна система визначає, змінювалася дана сторінка або ні й чи потрібно її перезаписувати на диск або ні.

2.2 Завдання до лабораторної роботи

Розробити найпростіший програмний емулятор менеджера віртуальної сторінкової пам’яті комп’ютера з однорівневою таблицею сторінок. Питання роботи менеджерів пам’яті розглянуті в літературі [1] на стор. 184 – 196, [2] на стор. 232 – 283, [3] на стор. 353 – 382.

У лабораторній роботі віртуальний адресний простір моделюється за допомогою звичайного бінарного файлу відповідної довжини (вказане у варіанті завдання), що містить чотирьохбайтні цілі числа без знака, старше слово яких вказує номер сторінки, а молодше – задає зміщення усередині сторінки, наприклад, чотири послідовних байти зі значеннями 0x99093D01 (при розташуванні даних у пам’яті діє правило «старший по старшому») задають номер сторінки 317 і зміщення усередині її 2457. Файл може бути створений окремим програмним модулем. Розроблена програма повинна забезпечувати:

- можливість вводу з клавіатури віртуальної адреси в межах від 0 до максимальної за завданням адреси без одиниці;
- перетворення адреси у формат сторінка/зміщення;
- відображення отриманої сторінки у відповідний сторінковий кадр (з обов’язковим урахуванням того, що сторінка вже може знаходитися в пам’яті);
- запис сторінки, що заміщається, на своє місце у файлі (якщо в цьому є необхідність);
- вивід результатів розрахунку в таблицю, приблизний вигляд якої по-

казаний нижче.

Приблизний вигляд представлення результатів роботи програми

Віртуальна адреса	Сторінка	Сторінковий кадр	Зміщення
25789	3	0	1213
417512	50	1	7912
17555	2	2	1171
213718	26	3	726
3206	0	4	3206

Таблиця може містити додаткові поля на розсуд розроблювача програми (наприклад, ознаку зміни сторінки).

Заповнення сторінкових кадрів спочатку виробляється на вимогу, а потім по одному з алгоритмів заміщення сторінок відповідно до варіанта завдання.

Ознаку зміни сторінки встановлювати, використовуючи генератор псевдовипадкових чисел.

Таблиця варіантів завдання

Варіант	1	2	3	4	5	6	7	8	9	10	11	12
Загальний віртуальний простір (адрес)	2^{22}	2^{24}	2^{23}	2^{21}	2^{20}	2^{20}	2^{23}	2^{22}	2^{21}	2^{24}	2^{24}	2^{23}
Розмір сторінкового кадру (подвійних слів)	2^{10}	2^{9}	2^{12}	2^{11}	2^{14}	2^{13}	2^{10}	2^{9}	2^{12}	2^{11}	2^{14}	2^{13}
Кількість сторінкових кадрів фізичної пам’яті	8	10	12	14	8	10	12	14	8	10	12	14
Алгоритм заміщення сторінок	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO	LRU	FIFO

Програма повинна бути написана з використанням об’єктно-орієнтованої технології, з обов’язковим здійсненням моделювання й представленням відповідних UML діаграм. Мова програмування C++, розробка в ОС Windows і в середовищі Dev-C++.

2.3 Деякі рекомендації до виконання роботи

У мові С, а відповідно й у мові С++, на відміну від мови Java, поняття «масив» і «покажчик» (у Java взагалі такого поняття немає) є дуже близькими. А оскільки в програмах постійно доводиться мати діло з масивами, в тому числі і в даній роботі, слід більш докладно розібратися в тім, чому у мові є й ті й інші і у чому вони все ж-таки відрізняються.

Відомо, що з точки зору програми або програміста масив – це набір однотипних змінних, доступ до яких виконується за допомогою індексів. Але з точки зору спеціаліста з операційних систем, з точки зору розподілу пам'яті, масив являє собою безперервну область байтів пам'яті за розміром рівною добутку кількості елементів в масиві на розмір одного елемента у байтах. Інакше кажучи масив вбудованих у мову елементів будь-якої вимірності відображається на фізичну структуру пам'яті, котра зветься «вектор». У свою чергу вектор характеризується початковою адресою, а до будь-якого елемента вектора (байта або слова пам'яті) можна звернутися знаючи його зміщення відносно цієї початкової адреси, тобто достатньо додати до початкової адреси зміщення і отримати адресу потрібного елемента.

Тому-то у мові С/С++ масиви і покажчики є майже ідентичними поняттями, це витікає з того, що адреси зберігаються у вбудованому типі даних – покажчику. Як і для інших вбудованих типів, для покажчиків визначені арифметичні операції.

Під «арифметикою покажчиків» (або «адресною арифметикою») розуміють правила застосування до покажчиків арифметичних операцій. Припустимими є операції: додавання покажчика із цілим числом, вирахування з покажчика цілого числа (до цих операцій входять також пред- і пост- інкремент і декремент), вирахування одного покажчика з іншого, а також операції порівняння покажчиків **>**, **>=**, **<**, **<=**, **==**, **!=**. Порівняння покажчиків, так само як і вирахування одного покажчика з іншого, можливо тільки у випадку однотипних покажчиків (покажчиків на об'єкти одного типу).

За винятком перерахованих, інші арифметичні операції над покажчиками не визначені – їхнє використання призведе до помилок компіляції програми. Так, помилку компіляції спричинить спроба додавання двох покажчиків або порівняння покажчиків на об'єкти різного типу, спроба вирахування з покажчика числа із плаваючою крапкою та ін. Ті ж арифметичні операції, котрі можна застосовувати до покажчиків, у переважній більшості випадків мають сенс тільки в контексті роботи з масивами (із цієї причини арифметика покажчиків і

масиви звичайно розглядаються одночасно).

Так, наприклад, з елементами масиву типу **char**, за допомогою покажчиків і арифметики покажчиків можна реалізувати наступні операції:

```
char s[100]; /* Масив зі 100 елементів типу char */
char *p = &s[0]; /* Покажчик на нульовий елемент */
*(p+10)='l'; /* Зміна десятого елемента */
/* "Установлення" покажчика на перший елемент */
p++;
*(p+10) = 'k'; /* Зміна 11 елемента */
/* Перевірка вмісту нульового елемента */
if (*(p-1))
  ...;
```

Але в цьому прикладі масив і покажчик мають тип **char**. Тобто, розмір елемента дорівнює одному байту і додавання 1 до адреси, призводить до переміщення в пам'яті саме на один байт. Якщо ж робота ведеться, наприклад, з масивом типу **int**, у якого розмір елементу 4 байти, то й у цьому випадку всі наведені вище арифметичні операції з покажчиками дають правильний результат. Тобто компілятор автоматично, без участі програміста, замість одиниці додає до адреси чотири:

```
/*
  Єдина заміна – замість типу char тип int.
  Усе як і раніше працює правильно
*/
int s[100]; /* Масив з 100 елементів типу int */
int *p = &s[0]; /* Покажчик на нульовий елемент */
*(p+10)='l'; /* Зміна десятого елемента */
/* "Установлення" покажчика на перший елемент */
p++;
*(p+10) = 'k'; /* Зміна 11 елемента */
/* Перевірка вмісту нульового елемента */
if (*(p-1))
  ...;
```

Це і є основна ідея арифметики покажчиків – додаючи (або віднімаючи) до покажчика якогось типу число, ми тим самим переміщаємо його на таке саме число елементів цього типу.

А щоб ще полегшити життя програмістів, C/C++ допускає деяку взаємозамінність масивів і покажчиків. Наприклад:

```
int s[100];
int *p;
p = s; /* Те ж саме, що й p=&s[0] */
p[10]=1; /* Те ж саме, що й *(p+10)=1 або s[10]=1 */
*s = 2; /* Те ж саме, що й s[0]=2 */
*(s+10)=0; /* Те ж саме, що й s[10]=0 */
```

Зрозуміло, це працює не тільки для **int**, але й для будь-яких інших типів, у тому числі й для типів, котрі вводяться самим користувачем.

До речі, часто виникають випадки, коли функція очікує покажчика, а їй хочеться передати масив, або навпаки. І завдяки взаємозамінності покажчиків і масивів зробити це вдається без зайвих турбот, як у такому прикладі:

```
#include <stdio.h>

void print1(char s[]) {
  printf(s);
}

void print2(char *s) {
  printf(s);
}

int main(void) {
  char str1[]="hello\n";
  char *str2="hello\n";
  print1(str2);
  print2(str1);
  return 0;
}
```

Для кращого розуміння застосування покажчиків та арифметики покажчиків необхідно розглянути ще декілька прикладів найпоширеніших ідіом мов C/C++, що стосуються роботи з покажчиками. У прикладах передбачається, що **p** являє собою покажчик на якийсь елемент масиву довільного типу.

В арифметиці покажчиків часто використовуються оператори інкременту й декременту. Щоб просто переставити покажчик на один елемент можна користуватися й постфіксною, і префіксною формами інкременту й декременту. Однак завдяки пріоритетам і асоціативності операторів **++**, **--** й *****, є можливість сполучити доступ до елемента й зсув покажчика в одному виразі. І при

цьому префіксна й постфіксна форми будуть давати різні результати. Наприклад:

```
*p++=5;
/* Число 5 записується в поточний елемент, потім
   покажчик переноситься на наступний елемент. */
*++p=8;
/* Спочатку покажчик переноситься на наступний
   елемент, потім до нього записується число. */
*p--=12;
*--p=7;
/* Дії аналогічні наведеним вище, але для
   пост- і пред- декрементів */
(*p)++;
(*p)--;
--*p;
++*p;
/* У цих операціях виконується інкремент
   і декремент змінної, на яку вказує покажчик*/
```

Іноді буває корисно знайти різницю між двома покажчиками. Це дає можливість обчислити число елементів масиву між ними (очевидно, що обидва покажчики повинні вказувати на елементи одного масиву). Наприклад, ось так обчислюється довжина рядка:

```
char s[]="Hello";
char *p=s;
while (*p++); // Цикл поки *p != '0' (кінець рядка)
int l = p - s - 1;
printf("Довжина рядка s %d символів\n",l);
```

Покажчики на той самий тип можна порівнювати між собою за допомогою операторів відносин. Найбільше часто це перевірки **==** або **!=**, але можна зустріти й інші:

```
int *p1, *p2;
if (p1 != p2) ...;
if (p1 < p2)  ...;
else if (p1 == p2) ...;
else  ...;  // p1 > p2
```

За допомогою оператора приведення типів значення покажчиків можна

записувати у змінні типу **long** і навпаки. Таке робиться, якщо потрібно працювати з адресою як зі звичайним числом:

```
unsigned long VGA_MEM_BASE = 0xA0000;
char *p = (char*) VGA_MEM_BASE;
unsigned long addr;
p[0] = 1; // Змінюється байт за адресою 0xA0000
addr = (unsigned long)p;
// Значення p запам'ятовується в змінній addr
```

У мові С за допомогою покажчиків є можливість звертатися до старших і молодших 16-бітних слів 32-бітних комірок пам'яті:

```
unsigned long a = 0x013D0999;
unsigned short* p = &a;
printf("%x/t%x/n", *p, *(p + 1));
/* Друкуються окремо молодше й
   старше слово подвійного слова */
```

Необхідно відзначити, що мова С++, завдяки більш суворої типизованості, не дозволяє виконувати подібні операції. Крім того, у мові С++ не можна застосовувати арифметику покажчиків для покажчиків на тип **void**, хоча вони й сумісні за типом з будь-якими іншими.

Наявність арифметики покажчиків разом із принципом реалізації оператора індексації й комутативністю операції додавання призводить до досить цікавої особливості масивів.

Якщо **v** – ім'я масиву, а **i** – індекс елемента цього масиву, то вираз **v[i]** тотожний до виразу **i[v]**. І в першому, і в другому випадку результатом буде значення **i-го** елемента масиву **v**, оскільки

```
i[v] = *(i + v) = *(v + i) = v[i].
```

Взагалі арифметичні операції дозволяють писати ефективний, компактний и елегантний код до того ж більш швидкий. Яскравим прикладом цьому слугує реалізація функцій роботи з рядками (зокрема функція копіювання рядків **strcpy**) у стандартній бібліотеці С/С++:

```
char* strcpy(char* t, const char* s)
  {
```

```
    char* p = t;
    while(*p++ = *s++);
    return t;
}
```

Поки що мова йшла тільки про одномірні масиви, але C/C++ дозволяють працювати з масивами будь-якої розмірності, і все, що було сказано вище про арифметику покажчиків і їхню спорідненість з масивами, справедливо й у багатомірному випадку. Однак справа отут трохи складніша, і, щоб правильно користуватися покажчиками у випадку багатомірних масивів, потрібно ясно уявляти собі, як улаштовані останні.

Як відомо для того, щоб створити, наприклад, двовимірний масив, приходиться використовувати два набори квадратних дужок, наприклад так:

```
double arr[2][4];
```

У цьому масиві 2 рядки по 4 елементи в кожному (або масив з 2-х рядків і 4-х стовпців). Тобто можна сказати, що такий двовимірний масив фактично являє собою два одномірних масиви з чотирьох елементів типу **double** у кожному, точніше – одномірний масив, двома елементами якого є одномірні масиви з елементами типу **double**.

Тому-то із взаємозамінністю покажчиків і масивів справа є декілька складнішою. Якщо у випадку одномірних масивів замість них можна використовувати покажчик на тип елемента, то в цьому випадку теж можна з рівним успіхом використовувати покажчик, але тип його відрізняється від типу покажчика на елемент. Якщо по звичці спробувати написати наступний код:

```
double arr[2][4];
double *p;
p=arr;,
```

то компілятор мови C видасть попередження про невірний тип використовуваного покажчика. Якщо проігнорувати це попередження, то в найкращому разі вийде неправильно працююча програма, а те й зовсім не працездатна. Компілятор C++ взагалі повідомить про помилку, і вихідні файли створені не будуть. Виходить, що у двовимірного масиву тип елемента не **double**, а якийсь інший. Власне кажучи, саме тому нові розмірності вказуються в нових наборах квадратних дужок. З урахуванням сказаного вище про те, що двовимірний масив насправді є одномірним з елементами теж масивами, то переходячи до покаж-

чиків можна написати:

```
double arr[2][4];
double *p;
p = arr[0]; /* Покажчик на нульовий рядок */.
```

Таким чином, виявляється, що тип елемента **arr** зовсім не **double**, а покажчик на **double**, тобто двовимірний масив через покажчик представляється як:

```
double (*parr)[4].
```

Круглі дужки в цьому виразі необхідні, якщо їх усунути, то замість покажчика на масив вийде масив із чотирьох покажчиків, і якщо написати такий, правильний, тип покажчика, його взаємозамінність із масивом відновиться.

Таким чином дуже важливим є розуміння основної ідеї – насправді програміст працює з абстрактною структурою «багатомірний масив», а компілятор і процесор – з одномірними масивами, масивами масивів, масивами масивів масивів і так далі. Виходячи із цього, за аналогією з одномірним масивом, до кожного елемента багатомірного масиву можна звернутися за допомогою відповідного покажчика на покажчик:

```
double **p2arr // Доступ до елемента 2-мірного масиву
double ***p3arr // – ″ – ″ елемента 3-мірного масиву
double ****p4arr // – ″ – ″ елемента 4-мірного масиву
```

Коли програмісти на C/C++ після одержання невеликого досвіду роботи освоюють основний принцип роботи з багатомірними масивами як з масивами покажчиків, або з покажчиками на покажчики, то стає очевидним, що такий підхід абсолютно не є складним (до речі, оголошення масивів з порожніми дужками – повністю ідентично оголошенню покажчика). Крім того такий підхід дозволяє більш гнучко працювати з кожним елементом масивів як з одномірним масивом. Крім того масиви другого, третього й так далі вимірів можуть мати різну довжину, що неможливо при звичайному завданні масивів. За допомогою одного покажчика, шляхом його перенастроювання, можна звертатися до елементів різних масивів, що містять елементи однакового типу, навіть іноді різного, використовуючи операції приведення типів. І, нарешті, програми з більшою кількістю більших багатомірних масивів, при доступі до їхніх елементів через покажчики, а не через індекси, працюють значно швидше.

Правда необхідно відзначити, що такий підхід вимагає трохи більшої пам'яті за рахунок виділення її для покажчиків.

Якщо прийняти до увазі той факт, що масиви і покажчики є взаємозамінні, то може виникнути цілком слушне запитання «Навіщо у мовах є й ті й інші і в чому вони, все ж таки, різняться?».

По-перше, поняття покажчика пов'язане не тільки зі зверненням до елементів масивів, але і дає змогу звертатись до будь якого структурного елемента даних програми через його адресу у пам'яті комп'ютера, а це є дуже важливим для системного програмування, оскільки воно передбачає необхідність доступу до всіх ресурсів апаратного забезпечення комп'ютера.

По-друге, адреса нульового елемента масиву (втім і будь-якого іншого) є покажчиком-константою, тому змінювати його, на відміну від покажчика-змінної, не можна ні в якому разі. Нижче наведений код ілюструє це твердження.

```
char s[]="Hello, world";
char *p;
p=s;
s+=10; // Помилка! Адреса масиву – покажчик-константа
p+=10; // Вірно, покажчик-змінну міняти можна
```

По-третє, коли масив створено, можна бути цілком упевненим, що компілятор обов'язково відведе для нього відповідну ділянку пам'яті, тобто в цього масиву обов'язково є деякі елементи, нехай навіть й не ініційовані і такі, що мають випадкові значення. В той самий час створення покажчика на тип означає виділення пам'яті під сам покажчик і ні яким чином не означає розташування масиву у пам'яті. В цьому випадку треба зробити так щоб покажчик шляхом або ініціалізації за допомогою оператора **new**, або присвоювання кудись вказував, наприклад так:

```
int s[10];
int *p;
s[0] = 0; // Нормально, у масиву s є елементи
p[0] = 0; // Не можна, p ще нікуди не указує
p=s;      // Відповідне присвоювання
p[0] = 0; // Тепер можна, p указує на s[0]
```

На закінчення опису роботи з покажчиками необхідно сказати наступне.

Значна частина помилок при роботі з пам'яттю пов'язана з її втратами,

тобто з не звільненням або неправильним звільненням раніше виділеної динамічної пам'яті. Зокрема, розповсюдженою помилкою є видалення оператором **`delete`** масиву об'єктів створеного оператором **`new`**, замість оператора **`delete[]`**. Така підміна операторів може ніяк не виявитися при видаленні масиву об'єктів убудованого типу, але при видаленні масиву складних об'єктів, наприклад масиву рядків (об'єктів типу **`std::string`**), використання оператора **`delete`** призведе до того, що вилученим виявиться тільки перший рядок, інші залишаться в пам'яті. Для виявлення помилок подібного роду створені спеціальні інструментальні засоби, опис яких не є метою даних методичних вказівок.

Гіршими виявляються справи з іншим видом помилок – помилками, викликаними ушкодженням (як говорять «протиранням») пам'яті. Такі помилки виникають при зверненнях по недійсних покажчиків, у тому числі, при виході в процесі обробки елементів масиву за його межі. Розіменування недійсного покажчика й наступна модифікація даних, на які він вказує, часто не закінчується аварійним завершенням роботи програми, а призводить до зміни стану об'єкта, що може в цей момент знаходитися на місці раніше вилученого об'єкта, а це неминуче порушує логіку роботи програми. Наслідки подібних помилок проявляються в зовсім непередбачених місцях, причому домогтися їхньої стабільної появи буває досить складно.

Таким чином, робота з пам'яттю вимагає особливої уваги й акуратності, некоректне поводження з нею призводить до помилок, що досить важко виявити, і до істотних втрат часу при розробці програмного забезпечення.

2.4 Завдання для підготовки до лабораторної роботи

1. Ознайомитися з коротким описом системи розробки програм Dev-CPP і вміти створити проект та виконати його компіляцію.
2. Розібратися в питаннях роботи менеджера віртуальної пам'яті, уяснити організацію алгоритмів заміни сторінок FIFO і LRU.
3. Створити попередню UML модель менеджера пам'яті і занести її до протоколу роботи.
4. За літературою з мови C++ (або конспектом [8]) ознайомитись з особливостями створення класів у мові C++ на відміну від Java.

2.5 Контрольні запитання до лабораторної роботи

1. Навіщо у комп'ютері повинен бути менеджер пам'яті, а чи можна обійтися без нього?
2. Що таке «віртуальна адреса»?
3. Чим фізична адреса відрізняється від віртуальної?
4. Що таке віртуальна сторінка і фізичний кадр, чим вони схожі і чим відрізняються?
5. Поясніть, чому на рисунку, що пояснює перетворення віртуальної адреси на фізичну, під номер сторінки відведено лише 3 біти, а під зміщення – 12?
6. Чому таблиця сторінок на рисунку, що пояснює перетворення віртуальної адреси на фізичну, повинна мати 2^{20} комірок?
7. Скільки є віртуальних сторінок якщо комп'ютер має 32-бітний віртуальний адресний простір з побайтовою адресацією, а сторінка має розмір 8 КіБ?
8. Може чи ні скластися така ситуація, коли віртуальна сторінка з адресами від 10240 до 12287 буде відображена у нульовий фізичний кадр, а сторінка з адресами від 0 до 2047 у перший?
9. Що таке робоча множина і як її розмір впливає на швидкість виконання програми?
10. Чим один від одного відрізняються алгоритми заміщення сторінок FIFO і LRU?
11. Який з двох алгоритмів заміщення сторінок, котрі розглядаються у лабораторній роботі, є кращім?
12. До супермаркету, що має обмежені площі для розташування товарів, надійшла партія нового товару, цілком є зрозумілим, що для його розміщення треба видалити деякий з тих, що вже є у наявності. За яким алгоритмом FIFO або LRU це краще здійснити?
13. ОЗП комп'ютера має три кадри, сторінки пам'яті викликаються у наступному порядку 8, 5, 7, 5, 12, 4, 6, 4, 14, 4 , які три сторінки і в якому порядку залишаться у пам'яті після закінчення роботи за алгоритмом FIFO?
14. ОЗП комп'ютера має три кадри, сторінки пам'яті викликаються у наступному порядку 7, 0, 10, 7, 2, 14, 2, 13, 10, 4, які три сторінки і в якому порядку залишаться у пам'яті після закінчення роботи за алгоритмом LRU?

15. Ініціалізація масиву **a** і покажчика **p** зроблена наступним чином:

```
int a[] = {7, 15, 31, 4, 9, 17, 22, 5, 8};
int *p = a;
```

чому дорівнюють наступні вирази?

```
*p+8
*(p+8)
*(p+9)
*p+9
*p+7
*(p+7)
```

16. Масив **a** і покажчик **p** описані так само, як у запитанні 15. Чи є різниця у виразах **p+3** і **p[3]**, а у виразах ***(p+3)** і **p[3]**?
17. Масив **a** і покажчик **p** описані так само, як у запитанні 15, крім того описана змінна **int n = 4**. Поясніть, чому вирази **p[n]** і **n[p]** повністю ідентичні, а вираз **4[p]** спричинить помилку при компіляції.

ПЕРЕЛІК ПОСИЛАНЬ.

1. Шеховцов В.А. Операційні системи. – К.: Видавнича група BHV, 2005. – 567 с.
2. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002. – 1040 с.
3. Таненбаум Э., Вудхалл А. Операционные системы: разработка и реализация. – СПб.: Питер, 2006. – 576 с.
4. Костромин В.А. Самоучитель Linux для пользователя – СПб.:БХВ-Петербург, 2003. – 672 с.
5. Xshell 4 User Guide Secure Terminal Emualtor – Seoul: NetSarang Computer, Inc., 2011. – 157 p.[Електронний ресурс] – Режим доступу: http://www.netsarang.com/docs/xshell4_manual.pdf
6. Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов: Пер. с англ. – К.: ООО «ТИД «ДО», 2004. – 624 с.
7. A Program for Directing Recompilation GNU make Version 3.82 / Richard M. Stallman, Roland McGrath, Paul D. Smith – Boston: Free Software Foundation, 2010 – 192 p. [Електронний ресурс] – Режим доступу: http://www.gnu.org/software/make/manual/make.pdf
8. Рольщиков В.Б. Об’єктно-орієнтоване програмування / Конспект лекцій. – Одеса: ОДЕКУ, 2009. – 215 с.

ДОДАТОК А
ФОРМА ТИТУЛЬНОГО АРКУША І ЗМІСТ ПРОТОКОЛУ РОБОТИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСКІЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ
Кафедра інформаційних технологій

ПРОТОКОЛ ЛАБОРАТОРНОЇ РОБОТИ № ___

З ОПЕРАЦІЙНИХ СИСТЕМ

тема або назва лабораторної роботи: _____

_____

виконав*(ла)* студент*(ка)* групи _____ _____
прізвище та ініціали

Оцінка за підготовку до роботи (теоретична частина)	Оцінка за виконання роботи (практична частина)	Оцінка за захист роботи (остаточна, загальна кількість балів)

1. Завдання до лабораторної роботи:

2. Коротка теоретична частина роботи, алгоритм програми або UML діаграма:

3. Текст програми:

4. Результати роботи програми:

5. Висновки: