Лабораторна робота №4
Засоби командного рядка для набору та збірки програм мовою С у UNIX-подібних системах

Мета роботи: вивчення основних прийомів та отримання практичних навичок виконання набору та збірки програми, написаної мовою *C* на *UNIX*-подібних системах.

Зміст

Короткі теоретичні відомості

Загальні поняття

Програма – це детальний та закінчений опис алгоритму засобами мови програмування. Виконавцем програми є комп'ютер. Для виконання комп'ютером програма повинна бути подана у машинному коді – послідовності чисел, які зрозумілі процесору, як відповідні інструкції та/або дані. Написати програму в

машинних кодах вручну досить складно. Тому сьогодні практично всі програми створюються за допомогою мов програмування вищого рівня, які за своїми синтаксисом та семантикою наближені до природної людської мови. Це знижує трудомісткість програмування. Однак, текст програми, записаний за допомогою мови програмування, повинен бути перетворений в машинний код. Ця операція виконується автоматично за допомогою спеціальної службової програми, яка називається транслятором.

Транслятори можна поділити на інтерпретатори та компілятори.

Інтерпретатори перетворюють в машинний код і виконують почергово оператори (команди) програми безпосередньо після запуску програми. Якщо команда повторюється, то інтерпретатор розглядає її так само, як і ту що зустрів вперше. Перевага інтерпретаторів – їх компактність, можливість зупинити в будь-який момент виконання програми, виконати різні перетворення даних і продовжити роботу програми. Якщо деякі оператори програми не знадобилися під час виконання, інтерпретатор не звертатиметься до них і не перетворюватиме їх.

Компілятор перетворює увесь вихідний текст програми повністю у машинний код. Ця операція виконується заздалегідь, і після отримання виконуваного модуля компілятор більше не потрібен. Виконуваний модуль також можна переміщувати на інші машини з подібною архітектурою. Тому перевага компіляторів – швидкодія і автономність отриманих програм.

Мова програмування C – відмінний вибір тих, хто починає знайомитися із програмуванням оскільки, *C* – це відносно проста мова, але вона є потужною та широко використовується. Також мова *C* є основою для багатьох інших мов програмування, і, отже, досвід, отриманий з *C*, можна застосувати і до інших мов. Крім того, досвід роботи з *C* корисний для глибокого розуміння *Linux* та *Unix*-подібних операційних систем, оскільки вони значною мірою написані на мові *C*.

Linux є досить сприятливим середовищем для написання програм. Це пояснюється тим, що, на відміну від деяких популярних комерційних операційних систем, немає необхідності купувати будь-яке дороге програмне

забезпечення для програмування, а в багатьох випадках відповідне програмне забезпечення вже встановлено на комп'ютері. Більше того, більшість основних дистрибутивів *Linux* включають засоби програмування; такі інструменти можна дуже легко встановити під час встановлення системи або окремо пізніше.

Створення, збірка та налагодження програм у UNIX-подібних системах

Найчастіше для створення програм на *C* потрібні три речі:

1. текстовий редактор,
2. компілятор,
3. стандартна бібліотека C.

Текстовий редактор – це програма, в якій можна створити файл, записати та зберегти текст програми (*source code*) на C та виконати редагування тексту програми за потреби. Програми на C можна писати за допомогою будь -якого із багатьох текстових редакторів, доступних для *Linux*, таких як *vi, gedit, gedit, kedit* або *emacs*.

Принаймні один текстовий редактор є вбудованим у кожну *Unix*-подібну операційну систему, і більшість таких систем містять кілька текстових редакторів. Щоб побачити, чи існує певний текстовий редактор у системі, все, що потрібно, це ввести його ім'я у командний рядок (тобто, текстовий користувальницький інтерфейс), а потім натиснути клавішу ***ENTER***. Якщо він існує, редактор з'явиться у існуючому вікні, якщо це редактор командного рядка, наприклад ***vi***. Якщо це редактор із графічним інтерфейсом користувача, наприклад ***gedit***, то він відкриється у новому вікні.

Наприклад, якщо потрібно побачити, чи є ***vi*** у системі (редактор ***vi*** майже завжди є у системі), потрібно лише ввести таку команду та натиснути клавішу ***ENTER***:

vi

Компілятор – це спеціалізована програма, яка перетворює вихідний код в машинний код (його також називають об'єктним кодом), який зрозумілий центральному процесору. Дуже добрий компілятор *C* включений до *GNU Compiler Collection (GCC)* – одного з компонентів більшості сучасних дистрибутивів *Linux*. GNU – це поточний проект *Free Software Foundation (FSF)* для створення повного, сумісного з *Unix*, високопродуктивного та вільно розповсюджуваного середовища.

Аби переконатися, що *GCC* вже встановлено у системі, слід ввести таку команду та натиснути клавішу *ENTER*:

gcc

повідомлення про помилку виду *«gcc: no input files»*, вказує на те, що *GCC* встановлено та готовий до використання. Якщо повідомлення про помилку виглядає наступним чином: *«command not found»*, то це вказує на те, що *GCC* не було встановлено. У разі відсутності *GCC* можна встановити так само, як і інше програмне забезпечення.

glibc – це реалізовані проектом *GNU* стандартні бібліотеки *C*. Як і у випадку з *GCC*, це також один із найважливіших компонентів більшості сучасних дистрибутивів *Linux*, які використовують його як свою офіційну стандартну бібліотеку *C*.

Бібліотека – це сукупність підпрограм, які може використати будь-який програміст, щоб зменшити кількість складного коду який часто повторюється у певних програмах. У стандартній бібліотеці зазвичай містяться модулі для роботи з файловою системою, дисплеєм та клавіатурою, набір підпрограм для обчислення математичних функцій і т.ін. Кожній *Unix*-подібній операційній системі потрібна бібліотека *C*.

Командою *locate* можна скористатися для підтвердження того, що бібліотеки *glibc* вже встановлено в системі. Перевірка наявності бібілотек можна виконати наступним чином:

locate glibc

Якщо файл із такою назвою присутній, то ця команда виведе на екран шлях до нього.

Інформацію про будь-які команді чи програми, включаючи *GCC*, *vi* та *gedit*, можна отримати за допомогою команди *man*. Для виходу із режиму перегляду інформації про програму, потрібно натиснути клавішу *q*.

Компіляція С-програми за допомогою GCC

Після того, як текст програми було написано та перевірено на наявність очевидних помилок, вона готова до компіляції. Компіляція складається з двох основних кроків: трансляції та компоновки (лінковки); *gcc* у своєму складі містить як транслятор, так і компоновщик.

Програму *program.c* можна зібрати за допомогою *GCC* наступним чином:

gcc -o program program.c

Ключ *-o* інформує компілятора про бажане ім'я для виконуваного файла, який буде створено після компіляції. Якщо параметр *-o* опущено, компілятор використає ім’я ***a.out*** за замовчуванням.

Час компіляції залежить від потужності комп’ютера, який виконує компіляцію, об’єму та складності програми, яка компілюється.

Після компіляції краще за все перевірити вміст каталогу, для того щоб переконатися, що було створено виконуваний файл програми. Для цього можна скористатися або командою *ls*, або файловим менеджером *Midnight Commander*.

Помилки, які можуть виникнути під час компіляції

При написанні програм можуть виникати помилки різного типу – синтаксичні, логічні та інші. Зазвичай під час компіляції виявляються синтаксичні помилки. Сучасні компілятори зазвичай надають досить докладні

повідомлення про помилки, які полегшують знаходження та виправлення помилок у коді програми.

Компілятор *GCC*, при виявленні помилки при компіляції, видає повідомлення `error:` , після якого йде детальний опис помилки. Приклад такого повідомлення наведено нижче:

```
.c:6:13: error: expected ';' before numeric constant
```

При появі помилок при компіляції, файл що виконується не буде створено.

Також при компіляції можуть з'явитися повідомлення про попередження: `warning:` із їх описом.

Деякі конструкції можуть бути цілком припустимі за одних обставин і небезпечними за інших. Компілятор може видавати попередження при використанні такого коду, для цього скористайтеся ключем *-Wall*.

Наявність попереджень може не містити загрози для нормальної роботи програми. Однак, програмісти намагаються писати код, який збирається без помилок та попереджень. Хоч це і не гарантує відсутності логічних помилок у програмі, загалом якість коду стає вищою. Іноді, щоб примусити програміста переглядати потенційно небезпечний код, використовується ключ *-Werror*, завдяки якому попередження компілятора інтерпретуються як помилки, і, таким чином, запобігають успішній збірці програми.

Виконання скомпільованої програми

Після того, як компілятор створить виконуваний файл, він готовий до роботи. Для запуску файла програми на виконання треба набрати його повний шлях у запрошенні оболонки, вказати необхідні ключі та аргументи і натиснути *ENTER*. Така процедура є громіздкою, тому зазвичай команди запускаються на виконання за їхніми іменами, а пошук власне виконуваних файлів відбувається у наперед заданих каталогах, перелічених у змінній оточення *$PATH*. Наприклад, у нашому *sandbox* за замовчуванням встановлено шлях пошуку:

```
sandbox:~$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Зверніть увагу, що до цього переліку за звичайних умов не входить домашній каталог користувача. Тому спроба запустити файл на виконання призведе до помилки пошуку. Щоб не вводити повний шлях до щойно створеної програми від кореневого каталога, можна скористатися спеціальними псевдонімами. Так, символ «.» (точка) означає «поточний каталог». Тому для запуску програми з поточного каталогу можна скористатися командою

```
./program
```

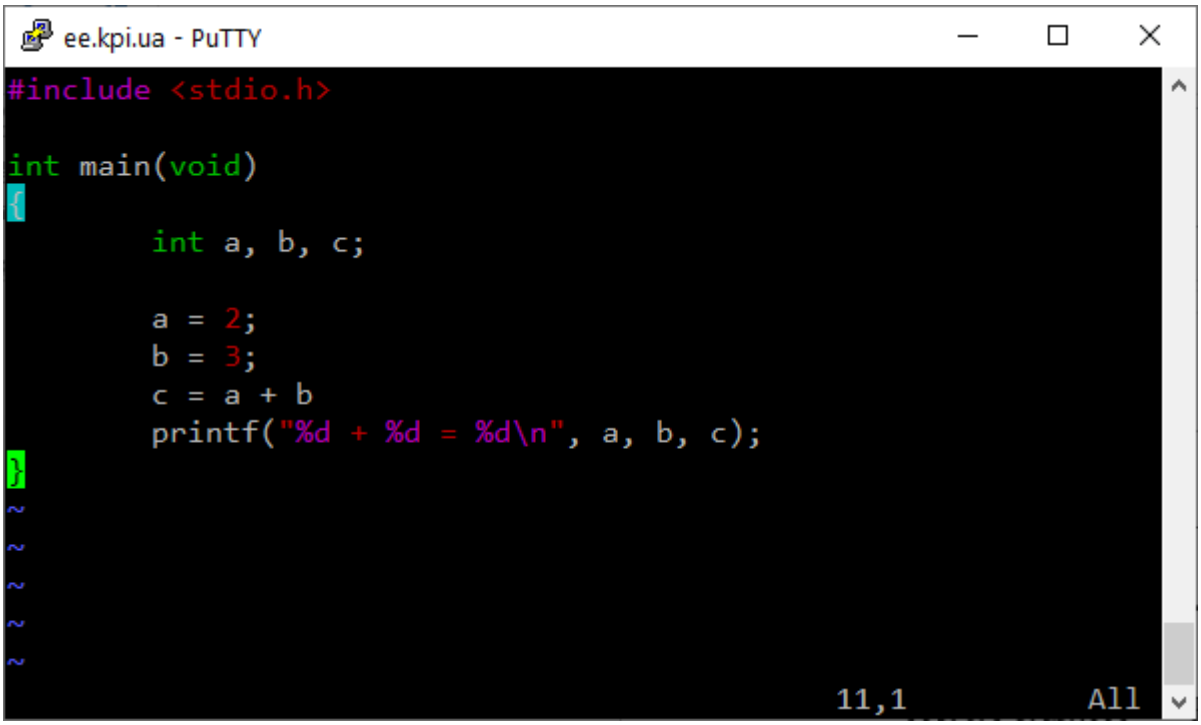
Результати виконання можна буде побачити у тому ж вікні починаючи із наступного рядка. Після закінчення виконання програм знову з’явиться запрошення командного рідка.

Приклад створення та налагодження програми у командному рядку

Скористаємося редактором vi для редагування файла *demo4.c*:

```
vi demo4.c
```

Наберемо текст програми, як показано на малюнку:



Зверніть увагу, за розширенням .c редактор розпізнав текст програми мовою *C* і виконує підсвічування синтаксису: вбудовані типи даних (`int, void`), числа (`2, 3`) і т.п. Також, коли курсор стоїть на правій фігурній дужці в останньому рядку, то підсвічується відповідна їй ліва фігурна дужка у четвертому рядку.

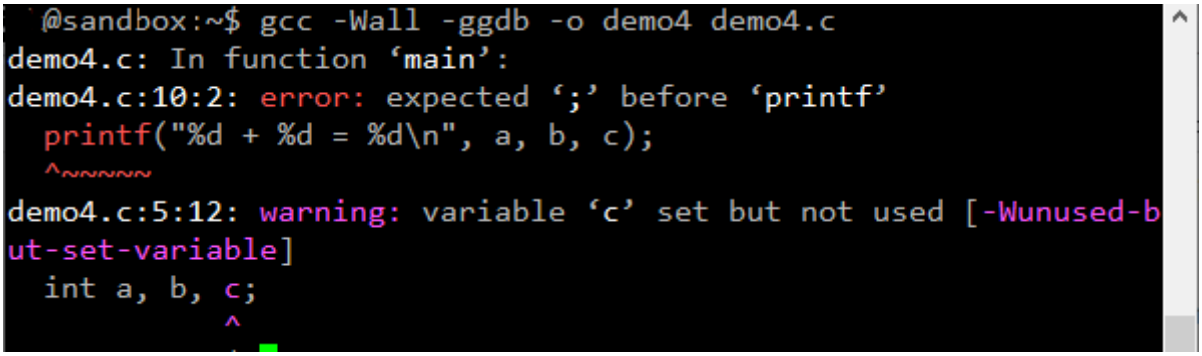
Збережемо файл з текстом, закриємо редактор і перейдемо до збірки програми командою:

```
gcc -Wall -ggdb -o demo4 demo4.c
```

`-Wall` – ключ, який дозволяє виводити на екран усі попередження, які виникають при компіляції програми,

`-ggdb` – ключ, який створює службову інформацію для програми налагодження *GDB*.

У вікні терміналу:



Як бачимо, збірка завершилася невдало: у десятому рядку виявлено помилку, а у п’ятому – попередження. Пояснення до помилки повідомляє, що у десятому рядку, другому стовпчику очікується символ «;» (крапка з комою) перед словом «*printf*». Якщо придивитися до тексту програми на попередньому малюнку, то можна виявити, що насправді крапка з комою була втрачена у дев’ятому рядку, а в її пошуках компілятор дійшов до десятого, де і зустрів слово «*printf*». За допомогою текстового редактора виправимо помилку і повторимо спробу. Цього разу компілятор не вивів ніяких повідомлень. Командою *ls* переконаємося, що у поточному каталозі справді створено файл *demo4*. Запустимо його на виконання командою

```
./demo4
```

У терміналі виводиться

```
  @sandbox:~$ ./demo4
2 + 3 = 5
```

У такій простій програмі логічні помилки малоймовірні, проте складніші програми, у переважній більшості, з першого разу реалізуються невдало і для пошуку помилок вдаються до налагодження програми. Цей процес передбачає зупинку програми на певних рядках, перевірку контрольних значень або встановлення змінних, покрокове виконання і т.п. Ці сервіси у тому чи іншому вигляді надаються усіма системами програмування. Розглянемо приклад налагодження простої програми у текстовій консолі за допомогою програми налагодження *gdb*. Запустимо налагодження командою

```
gdb ./demo4
```

У консолі буде виведено інформацію про програму, підказка щодо пошуку документації на командний рядок налагодження:

```
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./demo4...done.
(gdb)
```

Виконаємо пошук точки входу в програму, для програми мовою *C* це функція main(). Для цього у командному рядку наберемо *l* (скорочення від *list*), в консоль буде виведено наступні 10 рядків тексту програми:

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./demo4...done.
(gdb) l
1       #include <stdio.h>
2
3       int main(void)
4       {
5               int a, b, c;
6
7               a = 2;
8               b = 3;
9               c = a + b;
10              printf("%d + %d = %d\n", a, b, c);
(gdb)
```

Щоб почати покрокове налагодження програми, зупинимо її виконання одразу на точці входу за допомогою точки зупинки (англійською – *breakpoint*). Для цього у командному рядку наберемо

```
b main
```

Символ ‘*b*’ – це скорочення від команди ‘*break*’, а параметр вказує, що потрібно зупинитися на функції *main*. У консолі з’явиться таке повідомлення:

```
Reading symbols from ./demo4...done.
(gdb) l
1       #include <stdio.h>
2
3       int main(void)
4       {
5               int a, b, c;
6
7               a = 2;
8               b = 3;
9               c = a + b;
10              printf("%d + %d = %d\n", a, b, c);
(gdb) b main
Breakpoint 1 at 0x652: file demo4.c, line 7.
(gdb)
```

Зверніть увагу, що реально точка зупинки встановлена аж у 7 рядку, позаяк попередні рядки не містять коду, який можна налагодити. Запустимо програму на виконання командою *r* (від *run*), у консолі буде виведено:

```
Reading symbols from ./demo4...done.
(gdb) l
1       #include <stdio.h>
2
3       int main(void)
4       {
5               int a, b, c;
6
7               a = 2;
8               b = 3;
9               c = a + b;
10              printf("%d + %d = %d\n", a, b, c);
(gdb) b main
Breakpoint 1 at 0x652: file demo4.c, line 7.
(gdb) r
Starting program: /home/  /demo4

Breakpoint 1, main () at demo4.c:7
7               a = 2;
(gdb)
```

Як бачимо, виконання програми зупинилося у 7 рядку. Переглянемо значення змінної a командою «*print a*»:

```
(gdb) l
1       #include <stdio.h>
2
3       int main(void)
4       {
5               int a, b, c;
6
7               a = 2;
8               b = 3;
9               c = a + b;
10              printf("%d + %d = %d\n", a, b, c);
(gdb) b main
Breakpoint 1 at 0x652: file demo4.c, line 7.
(gdb) r
Starting program: /home/  /demo4

Breakpoint 1, main () at demo4.c:7
7               a = 2;
(gdb) print a
$1 = 32767
```

Змінна *a* містить «дивне» значення, що можна пояснити тим, що оператор присвоювання у сьомому рядку ще не було виконано. Виконаємо 7 рядок командою «*n*» (від *next*, мається на увазі наступний рядок) і знову надрукуємо значення змінної *a*:

```
Breakpoint 1, main () at demo4.c:7
7               a = 2;
(gdb) print a
$1 = 32767
(gdb) n
8               b = 3;
(gdb) p a
$2 = 2
(gdb)
```

Цього разу програма зупинилася на 8 рядку, де буде оператор «*b = 3;*», а значення змінної *a* вже дорівнює 2. Для зупинки програми у на подію оновлення значення змінної скористаємось командами

watch b

watch c

```
(gdb) p a
$2 = 2
(gdb) watch b
Hardware watchpoint 2: b
(gdb) watch c
Hardware watchpoint 3: c
(gdb)
```

Продовжимо виконання програми командою «*c*» (*continue*):

```
(gdb) c
Continuing.

Hardware watchpoint 2: b

Old value = 0
New value = 3
main () at demo4.c:9
9               c = a + b;
(gdb)
```

Система друкує колишнє та нове значення змінної, яка переглядається і зупиняє виконання програми. Спробуємо змінити значення змінної *a* командою

```
(gdb) set variable a = 5
(gdb)
```

Переглянемо значення усіх локальних змінних:

```
(gdb) info locals
a = 5
b = 3
c = 0
(gdb)
```

Продовжимо виконання програми командою «*c*» (*continue*):

```
(gdb) c
Continuing.

Hardware watchpoint 3: c

Old value = 0
New value = 8
main () at demo4.c:10
10              printf("%d + %d = %d\n", a, b, c);
(gdb)
```

Відбулося оновлення значення змінної *c*, яка була поставлена «під нагляд» командою *watch*, тому виконання програми зупинилося і в консоль виведено інформацію про оновлення значення змінної *c*. Продовжимо виконання програми командою «*c*» (*continue*), у консолі буде надруковано:

```
(gdb) c
Continuing.
5 + 3 = 8
```

Решта повідомлень перевершують наші поточні знання, залишимо їх без розгляду. Закінчимо сеанс налагодження командою «*q*» (*quit*). У будь-який можна скористатися командою *help*, щоб отримати підказку про команди та їх формати.

Засоби автоматизації збірки make

У процесі розробки цієї простої програми нам довелося скористатися кількома іншими програмами та викликати їх з різними ключами, які, звісно, треба пам’ятати. Якщо програма складається більше, ніж з одного файлу вихідного коду, завдання стає ще складнішим. Тому для автоматизації збірки програм розроблено програмне забезпечення *make* із командними файлами. Детальний виклад синтаксису командних файлів та можливостей *make* перевершує наші поточні знання, тому обмежимося простим прикладом. Для

щойно розглянутого прикладу можна створити такий командний файл з іменем *Makefile* (з великої літери *M*):

```
# Default target
.PHONY: all
all: demo4

# Rule to build executable
demo4: demo4.c
	gcc -Wall -ggdb -o demo4 demo4.c

.PHONY: clean
clean:
	rm -f demo4

.PHONY: debug
debug: demo4
	gdb demo4

```

Команди у *Makefile* починаються із символа табуляції. Маючи такий командний файл збірку програми можна виконати командою

`make` або `make all`

```
@sandbox:~$ make all
gcc -Wall -ggdb -o demo4 demo4.c
```

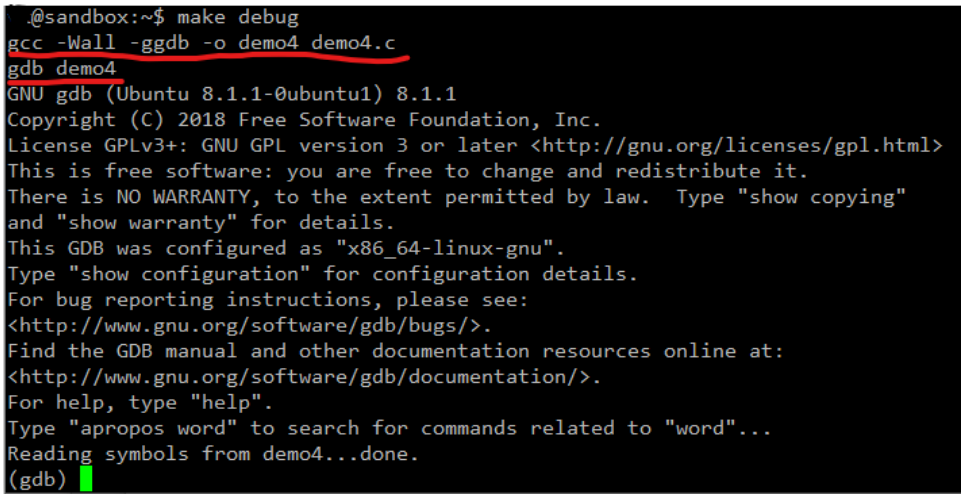
Якщо відредагувати файл вихідного коду *demo4.c*, то система *make* помітить це і виконає повторну компіляцію. Якщо вихідні файли не мінялися, буде надруковане відповідне повідомлення.

Для видалення згенерованих файлів можна скористатися командою очищення:

```
@sandbox:~$ make clean
rm -f demo4
```

Для запуску програми на налагодження можна скористатися командою:

`make debug`



Зверніть увагу, що оскільки для правила *debug* встановлено залежність від *demo4* (виконуваний файл програми), то спочатку було скомпільовано виконуваний файл, а лише потім запущено налагодження. Для цієї роботи охочі можуть скористатися наданим *Makefile* та переписати його на сервер до свого робочого каталога, як описано у наступному розділі.

Копіювання файлів між сервером і комп'ютером

SCP (Secure Copy Protocol) – команда в *Linux* для захищеного копіювання файлів або папок на віддалений комп'ютер (сервер) або з нього, використовуючи для цього протокол *SSH* (*Secure Shell*). *SCP* є складовою частиною пакету *OpenSSH*. Завдяки використанню *ssh, SCP* є не тільки зручним, але і безпечним способом копіювання файлів.

Команда копіювання локального *SourceFile* на віддалений хост:

```
scp SourceFile user@host:/directory/TargetFile
```

Команда копіювання *SourceFile* з віддаленого хоста:

```
scp user@host:/directory/SourceFile TargetFile
```

Для копіювання файлів між сервером та комп'ютером також можна скористатися графічним клієнтом *WinSCP*, як це описано в попередніх лабораторних роботах.

Завдання для виконання

Частина 1

В пунктах завдання 1-10 цієї лабораторної роботи ми НЕ використовуємо файловий менеджер *WinSCP*.

1. Зайти на сервер *sandbox.ee.kpi.ua*, скориставшись інформацією із попередніх лабораторних робіт.
2. Перейти в каталог *PROG_SEM1*, а якщо не має такого каталогу, то створити його.
3. Відкрити будь-який текстовий редактор та створити текстовий файл із ім'ям *laba4.c*, в якому набрати наступний код програми:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

4. Виконати компіляцію файлу *laba4.c* та створити файл що виконується із ім'ям *laba4_1*.
5. Переглянути вміст поточного каталогу (будь-яким зручним для вас способом) та переконатися, що файл *laba4_1* з'явився.
6. Запустити на виконання файл *laba4_1*.
7. Зайти в файл *laba4.c* в режимі редагування та змінити напис `return 0;` на `returnnn 0;`, додавши букви *n* в кінці слова *return*.
8. Виконати компіляцію файлу *laba4.c* та створити файл що виконується із ім'ям *laba4_2*.
9. Переглянути повідомлення про помилки при спробі скомпілювати цей файл.
10. Переконатися, що після виконання компіляції файл із ім'ям *laba4_2* НЕ з'явився в поточній директорії.

Частина 2

11. На персональному комп'ютері, на якому виконується лабораторна робота, створити файл із ім'ям *new_lab4.c*. В файл розмістити наступний текст програми:

```
#include <stdio.h>
#define PI 3.14

int main()
{
    foat r, p;
    r = 3;
    p = 2*PI*r;
    printf("r = %f, p = %f\n", r, p);
    return 0;
}
```

12. Запустити графічний клієнт для копіювання файлів *WinSCP* на персональному комп'ютері, на якому виконується лабораторна робота та зайти *sandbox.ee.kpi.ua* за допомогою *WinSCP*.
13. Скопіювати файл *new_lab4.c* із персонального комп'ютера, на якому виконується лабораторна робота, на сервер *sandbox.ee.kpi.ua,* в каталог із ім'ям *PROG_SEM1*.
14. Виконати компіляцію файлу *new_laba4.c* та створити файл що виконується із ім'ям *new_laba4_1*. Якщо буде виявлено помилки – виправити їх.
15. Переглянути вміст поточного каталогу (будь-яким зручним для вас способом) та переконатися, що файл *new_laba4_1* з'явився.
16. Запустити на виконання файл *new_laba4_1*. Переконатися що результати виконання програми коректно виводяться на екран.
17. Виконати програму покрокового налагодження програми за допомогою *gdb*:
    - ✓ Перекомпілювати файл *new_laba4.c* таким чином, щоб можна було скористатися програмою покрокового налагодження *gdb*.
    - ✓ Запустити файл *new_laba4_1* в режимі налагодження.
    - ✓ Встановити *breakpoint* на початку функції *main*.

- ✓ Запустити програму на виконання та подивитися поточне значення змінної *p*.
- ✓ Виконати покроково 4 наступні рядочки програми та знову подивитися поточне значення змінної *p*.
- ✓ Закінчити виконання програми *new_laba4_1*.

Контрольні питання

1. Які типи трансляторів ви знаєте?
2. Для чого використовується програма *GCC*?
3. Як називаються реалізовані проектом *GNU* стандартні бібліотеки *C*?
4. Який тип помилок в програмі частіш за все знаходить компілятор?
5. Які програми копіювання файлів між сервером і комп’ютером ви знаєте?
6. Чи дозволяє програма *WinSCP* зберігати налаштування із сервером у певні профілі?