МЕТОДИЧНА РОЗРОБКА
для проведення лабораторного заняття з дисципліни
«Клієнт-серверне програмування»

Лабораторна робота

Компіляція програм. GCC компілятор. Створення бібліотек.

Мета роботи: набуття навичок використання компіляторів програм, створювання користувацьких бібліотек.

1. Теоретичні відомості

Компілятор – транслятор, що перетворює вихідний код з будь-якої мови програмування на машинну мову.

Процес компіляції, як правило, складається з декількох етапів:

- лексичний аналіз;
- синтаксичний аналіз;
- семантичний аналіз;
- створення на основі результатів аналізів проміжного коду;
- оптимізація проміжного коду;
- створення об'єктного коду, в даному випадку машинного.

GNU Compiler Collection, (GCC) — набір компіляторів для різних мов програмування. GCC — вільне програмне забезпечення, розроблене Фондом Вільних Програм під ліцензією GNU GPL та GNU LGPL, і є ключовою складовою набору знарядь розробки GNU (GNU development toolchain). Це стандартний компілятор для вільних Unix-подібних операційних систем, і деяких пропрієтарних систем, що з них розвинулись, наприклад Mac OS X.

Спочатку називався GNU Компілятор Сі, оскільки підтримував лише одну мову програмування — C. Пізніше був розширений для підтримки C++, Fortran, Java (компілятор GCJ), Ada, D, та інших.

Зовнішній інтерфейс GCC є стандартом для компіляторів на платформі Unix. Користувач викликає управляючу програму, яка називається gcc. Вона інтерпретує аргументи командного рядка, визначає і запускає для кожного вхідного файлу свої компілятори потрібної мови, запускає, якщо необхідно, асемблер і компонувальник.

Бібліотека - це пакет коду, який призначений для повторного використання багатьма програмами.

«Програмна бібліотека» — це простий файл, що містить скомпільований код (і дані), який пізніше буде включено в програму; бібліотеки програм дозволяють програмам бути більш модульними, їх швидше перекомпілювати та легше оновлювати.

Деякі бібліотеки можуть бути розділені на кілька файлів та/або мати декілька файлів заголовків.

Бібліотека - це місце, де реалізована фактична функціональність, тобто вони містять тіло функції.

Програмні бібліотеки можна розділити на типи:

- статичні (static libraries);
- динамічні (dynamically loaded libraries).

Статичні бібліотеки включаються у виконуваний файл програми перед запуском програми.

Динамічні бібліотеки динамічно завантажуються під час роботи виконуваного файлу.

Бібліотеки попередньо компілюються з кількох причин.

По-перше, оскільки бібліотеки нечасто змінюються, їх не потрібно часто перекомпілювати. Було б марно витрачати час на перекомпіляцію бібліотеки кожного разу, коли ви написали програму, яка їх використовує.

По-друге, оскільки попередньо скомпільовані об'єкти знаходяться на машинній мові, це запобігає доступу людей або зміні вихідного коду.

Статична бібліотека (також відома як архів) складається з процедур, які компілюються і безпосередньо пов'язані з вашою програмою. Під час компіляції програми, яка використовує статичну бібліотеку, вся функціональність статичної бібліотеки, що використовує наша програма, стає частиною виконуваної програми. У Windows статичні бібліотеки зазвичай мають розширення .lib, тоді як у Unix-подібних систем статичні бібліотеки зазвичай мають розширення .a (архіву). Однією з переваг статичних бібліотек є те, що потрібно завантажувати або розповсюджувати лише виконуваний файл, щоб користувачі могли запускати вашу програму. Оскільки бібліотека стає частиною вашої програми, це гарантує, що з вашою програмою завжди використовується правильна версія бібліотеки.

Динамічна бібліотека (яка також називається спільною бібліотекою, shared library) складається з процедур, які завантажуються у вашу програму під час виконання. Коли ви збираєте програму, яка використовує динамічну бібліотеку, бібліотека не стає частиною вашого виконуваного файлу - вона залишається окремою одиницею. У Windows динамічні бібліотеки, як правило, мають розширення .dll (динамічна бібліотека зв'язків), тоді як у Linux динамічні бібліотеки зазвичай мають розширення .so (shared object). Однією з переваг динамічних бібліотек є те, що багато програм можуть спільно використовувати одну копію, що економить простір. Можливо, більшою перевагою є те, що динамічну бібліотеку можна оновити до більш нової версії без заміни всіх виконуваних файлів, які використовують її.

2. Компіляція

Для виконання компіляції програм потрібно встановити сам компілятор. Для цього необхідно виконати команду:

```
yum install gcc
```

```
[root@ksp LIB]# yum install gcc
```

```
Встановлено:
  cpp-14.2.1-7.el10.x86_64              gcc-14.2.1-7.el10.x86_64
  glibc-devel-2.39-32.el10.x86_64       kernel-headers-6.12.0-56.el10.x86_64
  libmpc-1.3.1-7.el10.x86_64            libxcrypt-devel-4.4.36-10.el10.x86_64
  make-1:4.4.1-9.el10.x86_64
```

Серед безлічі опцій компіляції та компонування найчастіше використовуються такі:

Опція	Призначення
-c	Ця опція означає, що потрібна лише компіляція. З вихідних файлів програми створюються об'єктні файли як **name.o** . Компонування не проводиться.
-Dname=value	Визначити ім'я name в програмі, що компілюється, як значення v **alue** . Ефект такий, як наявність рядка **#define name value** на початку програми. Частина **=value** може бути опущена, у разі значення за умовчанням дорівнює 1.
-o file-name	Використовувати **file-name** як ім'я для створюваного файлу.
-lname	Використовувати під час компонування бібліотеку libname.so
-Llib-path **-Iinclude-path**	Додати до стандартних каталогів пошуку бібліотек та заголовних файлів шляху lib-path та include-path відповідно.
- g	Помістити в об'єктний або виконуваний файл налагоджувальну інформацію для налагоджувача **gdb** . Опція повинна бути вказана і для компіляції, і компонування. У поєднанні **– g** рекомендується використовувати опцію відключення оптимізації **–O0** (див.нижче)
-MM	Вивести залежності від заголовних файлів, що використовуються в Сі або С++ програмі, у форматі, що підходить для утиліти **make** . Об'єктні чи виконувані файли не створюються.
-pg	Помістити в об'єктний або виконуваний файл інструкції профілювання для генерації інформації, що використовується утилітою **gprof** . Опція повинна бути вказана і для компіляції, і компонування. Зібрана з опцією **-pg** програма під час запуску генерує файл статистики. Програма **gprof** на основі цього файлу створює розшифровку, яка вказує час, витрачений на виконання кожної функції.
-Wall	Виведення повідомлень про всі попередження чи помилки, які виникають під час компіляції програми.
-O1 **-O2**	Різні рівні оптимізації.

-O3	
-O0	Не оптимізувати. Якщо ви використовуєте численні опції **-O** з номерами або без номерів рівня, дійсною є остання така опція.
-I	Використовується для додавання ваших власних каталогів для пошуку заголовних файлів у процесі збирання
-L	Передається компонувальнику. Використовується для додавання ваших власних каталогів для пошуку бібліотек у процесі збирання.
-l	Передається компонувальнику. Використовується для додавання власних бібліотек для пошуку в процесі збирання.

Для компіляції використовуємо команду:

gcc <ім'я файлу>

```
[root@ksp cpp]# gcc sum.c
```

За замовчуванням створюється скомпільований файл **a.out**

Приклад 1. Команда ***gcc***

```
[root@ksp LIB]# cat sum.c
#include <stdio.h>
int add(int x, int y){
return x+y;
}
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
[root@ksp LIB]# gcc sum.c
[root@ksp LIB]# ls
a.out  sum.c
[root@ksp LIB]#
```

Виконаємо файл **a.out**.

```
[root@ksp LIB]# ./a.out
The sum of 3 and 4 is: 7
 [root@ksp LIB]#
```

Приклад 2. Команда ***gcc -c name.c***

Відбувається компіляція вихідного коду C++, що знаходиться у файлі **name.c** і створюється об'єктний файл **name.o**.

Опція -c означає "тільки компіляція".

Щоб скомпонувати один або кілька об'єктних файлів, отриманих з вихідного коду - **name1.o**, **name2.o**, ... - в єдиний файл **name**, що виконується, необхідно ввести команду:

gcc name1.o name2.o -o name

Опція -o задає ім'я файлу, що виконується.

Приклад 3. Створимо файл **main.c**

```
#include <stdio.h>
int main()
{
	int unused_var;
	printf("Hello linux world\n");
	return 0;
}
```

```
  GNU nano 8.1                                  main.c
#include <stdio.h>
int main()
{
        int unused_var;
        printf("Hello linux world\n");
        return 0;
}
[root@ksp LIB]# ls
a.out  main.c  sum.c
```

Виконаємо компіляцію файлу **main.c** .

```
[root@ksp LIB]# gcc -c main.c
[root@ksp LIB]# ls
a.out  main.c  main.o  sum.c
```

Та його компонування під ім’ям **main**.

```
[root@ksp LIB]# ls
a.out  main  main.c  main.o  sum.c
```

Виконаємо файл **main**.

```
[root@ksp LIB]# ./main
Hello linux world
```

Приклад 3. Виконаємо компіляцію та компонування файлу **main.c** з використанням опції ***-Wall***.

```
[root@ksp LIB]# gcc -Wall main.c -o main
main.c: У функції «main»:
main.c:4:13: попередження: Невикористана змінна «unused_var» [-Wunused-variable]
    4 |         int unused_var;
      |             ^~~~~~~~~~
[root@ksp LIB]# ./main
Hello linux world
```

Отримуємо попередження про помилку, але компіляція і компонування відбулось.

Приклад 4. Розбиття програми на різні файли

Розглянемо просту програму:

```
#include <stdio.h>
int add(int x, int y){
return x+y;
}
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
```

}

Дану програму доцільно програму на 2 частини - з функцією add (файл add.c) та головною функцією (файл mains.c):

```
// файл 1 add.c
int add(int x, int y){
return x+y;
}
// файл 2: mains.c
#include <stdio.h>
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

```
[root@ksp LIB]# nano add.c
[root@ksp LIB]# nano mains.c
[root@ksp LIB]# ls
add.c  a.out  main  main.c  main.o  mains.c  sum.c
```

Виконаємо компіляцію і компонування

```
[root@ksp LIB]# gcc mains.c -o mains
mains.c: У функції «main»:
mains.c:3:41: помилка: Неявне оголошення функції «add» [-Wimplicit-function-declar
ation]
    3 | printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
      |                                         ^~~
```

Для того, щоб компіляція відбулася без попереджень, можна додати включення першого файлу у другий та виконати компіляцію.

```
#include <stdio.h>
#include "add.c"
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

```
  GNU nano 8.1                                  mains.c
#include <stdio.h>
#include "add.c"
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

В цьому випадку директива #include "add.c" просто додає код файлу add.c до другого файлу mains.c, а отже команда компіляції (наприклад gcc mains.c) виконає початкову версію файлу.

Виконаємо компіляцію і компонування

```
[root@ksp LIB]# gcc mains.c -o mains
[root@ksp LIB]# ls
add.c  a.out  main  main.c  main.o  mains  mains.c  sum.c
```

Виконаємо файл **mains**.

```
[root@ksp LIB]# ./mains
The sum of 3 and 4 is: 7
```

Приклад 4. Зазвичай компіляція програми виконується в два етапи: компіляція об'єктних файлів і компоновка виконуваної програми з об'єктних файлів.

```
// файл  add.c
int add(int x, int y){
return x+y;
}

// файл 2: func.c
#include <stdio.h>
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Виконаємо компіляцію файлу **add.c** та **func.c**

```
[root@ksp LIB]# gcc -c add.c
[root@ksp LIB]# gcc -c func.c
func.c: У функції «main»:
func.c:3:41: помилка: Неявне оголошення функції «add» [-Wimplicit-function-decla
ration]
    3 | printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
      |                                         ^~~
```

Для того, щоб винести функцію або змінну в окремий файл треба перед нею поставити зарезервоване слово **extern**.

```
#include <stdio.h>
extern int add ();
int main()
{
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

```
  GNU nano 8.1                                  func.c
#include <stdio.h>
extern int add ();
int main(){
printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Виконаємо компіляцію файлу **func.c** та отримаємо об’єктний файл **func.o**

```
[root@ksp LIB]# gcc -c func.c
[root@ksp LIB]# ls
add.c  a.out   func.c  main    mains    sum.c
add.o  comp.c  func.o  main.c  mains.c
```

А далі виконаємо збірку у виконуваний бінарний файл **func_bin**

```
[root@ksp LIB]# gcc add.o func.o -o func_bin
[root@ksp LIB]# ls
add.c  a.out   func_bin  func.o  main.c  mains.c
add.o  comp.c  func.c    main    mains   sum.c
```

Результат виконання

```
[root@ksp LIB]# ./func_bin
The sum of 3 and 4 is: 7
```

3. Створення програмної бібліотеки

«Програмна бібліотека» — це простий файл, що містить скомпільований код (і дані), який пізніше буде включено в програму; бібліотеки програм дозволяють програмам бути більш модульними, їх швидше перекомпільовувати та легше оновлювати.

Програмні бібліотеки можна розділити на типи:

- статичні (static libraries);
- динамічні (dynamically loaded libraries).

Статичні бібліотеки включаються у виконуваний файл програми перед запуском програми.

Динамічні бібліотеки динамічно завантажуються під час роботи виконуваного файлу.

Бібліотеки попередньо компілюються з кількох причин.

По-перше, оскільки бібліотеки нечасто змінюються, їх не потрібно часто перекомпілювати. Було б марно витрачати час на перекомпіляцію бібліотеки кожного разу, коли ви написали програму, яка їх використовує.

По-друге, оскільки попередньо скомпільовані об'єкти знаходяться на машинній мові, це запобігає доступу людей або зміні вихідного коду.

Приклад 1. Створення статичної бібліотеки

Розглянемо просту програму:

Створимо файл ***func1.c***

```
#include <stdio.h>
void say_hello()
{
printf("Hello from func\n");
}
int addOne(int x) // x - це змінна-посилання
{
return x = x + 1;
}
int main()
{
   say_hello();
   int a = 7;
   printf(" a = %d \n", addOne(a));
   return 0;
}
```

Після компіляції та виконання отримаємо:

```
[root@ksp statlib]# gcc func1.c -o func1
[root@ksp statlib]# ls
func  func1  func1.c  func2.c  func.c  func.h  main1.c  main1.o  main.c
[root@ksp statlib]# ./func1
Hello from func
 a = 8
```

Для створення статичних бібліотек існує спеціальна проста програма звана ar (скор. від archiver - архіватор). Вона використовується для створення,

модифікації та перегляду об'єктних файлів в статичних бібліотеках, які насправді представляють із себе прості архіви.

Розділимо вихідний код на два файли ***func2.c*** та ***main2.c***

Файл ***func2.c***

```
#include <stdio.h>
void say_hello()
{
printf("Hello from func\n");
}
int addOne(int x) // x - це змінна-посилання
{
return x = x + 1;
}
```

Файл ***main2.c***

```
#include <stdio.h>
#include <math.h>
#include "func.h"
int main()
{
    printf("Hello linux world\n");
    say_hello();
    printf("x + 1 = %d\n", addOne(8));
    printf("sin(2) = %lf\n", sin(2));
}
```

Для початку компілюємо ці файли:

```
[root@ksp statlib]# gcc -c func2.c
[root@ksp statlib]# gcc -c main2.c
[root@ksp statlib]# ls
func   func1.c  func2.o  func.h   main2.c  main.c
func1  func2.c  func.c   main1.c  main2.o
```

Для того, щоб створити бібліотеку з об'єктів файлів треба викликати програму ***ar rc libім'я_бібліотеки. a [список_ *. o_файлів]***

```
[root@ksp statlib]# ar rc libfunc2.a func2.o
[root@ksp statlib]# ls
func   func1.c  func2.o  func.h      main1.c  main2.o
func1  func2.c  func.c   libfunc2.a  main2.c  main.c
```

Ключом -L ми задаємо шлях де компілятор повинен шукати нашу бібліотеку. Оскільки, ми вказали відносний шлях ./, то компілятор буде шукати бібліотеку у поточному каталозі;

Ключ -l (l маленьке) вказує компілятору, яку бібліотеку потрібно підключити без **префікса** lib і **закінчення**.

[root@ksp statlib]# gcc main2.o -l func2 -L ./ -o mainlib

```
[root@ksp statlib]# gcc main2.o -l func2 -L ./ -o mainlib
[root@ksp statlib]# ls
func   func1.c  func2.o  func.h      main1.c  main2.o  mainlib
func1  func2.c  func.c   libfunc2.a  main2.c  main.c
```

Результат виконання

```
[root@ksp statlib]# ./mainlib
Hello linux world
Hello from func
x + 1 = 9
sin(2) = 0.909297
```

Приклад 2. Створення динамічної бібліотеки

Спільні бібліотеки завантажуються під час запуску програми та можуть використовуватися різними програмами. Усі програми можуть мати спільний доступ до спільної бібліотеки та будуть оновлені, якщо встановлено нову версію бібліотеки.

Кожна бібліотека має назву «soname», яка починається з префікса «lib», за яким іде назва бібліотеки, розширення «.so», потім крапка та номер версії. Кожна спільна бібліотека також має «справжнє ім'я», яке є назвою файлу з фактичним кодом бібліотеки. Це «справжнє ім'я» — це «soname» із додатковою крапкою та номером другорядної версії, а також, необов'язково, крапкою та номером випуску.

Кілька різних програм можуть використовувати одну бібліотеку, і кожна з них розташовується в різному адресному просторі. Тому потрібно, щоб переходи у функціях бібліотеки (операції goto на асемблері) використовували не абсолютну адресацію, а відносну. Тобто генерований компілятором код повинен бути незалежним від адрес, така технологія отримала назву PIC - Position Independent Code. У компіляторі gcc дана можливість вмикається ключем -fPIC.

Використаємо вихідний код з попереднього прикладу:

Файл ***func2.c***

```
#include <stdio.h>
void say_hello()
{
printf("Hello from func\n");
}
int addOne(int x) // x - це змінна-посилання
{
return x = x + 1;
}
```

Файл ***main2.c***

```
#include <stdio.h>
#include <math.h>
#include "func.h"
int main()
{
    printf("Hello linux world\n");
    say_hello();
    printf("x + 1 = %d\n", addOne(8));
    printf("sin(2) = %lf\n", sin(2));
```

}

Тепер компілювання наших файлів буде мати вигляд:

gcc -Wall -fpic -c func2.c -o funcdyn.o

Для того, щоб створити динамічну бібліотеку треба використовувати ключ *-shared:*

gcc -shared funcdyn.o -o libfunc2.so

```
[root@ksp dynlib]# gcc -Wall -fpic -c func2.c -o funcdyn.o
[root@ksp dynlib]# ls
func2.c  funcdyn.o  main2.c
[root@ksp dynlib]# gcc -shared funcdyn.o -o libfunc2.so
[root@ksp dynlib]# ls
func2.c  funcdyn.o  libfunc2.so  main2.c
```

У результаті отримаємо динамічну бібліотеку libfunc2.so, яка буде динамічною версією бібліотеки libfunc2.a, з попереднього прикладу. Тепер, щоб компілювати результуючий файл з використанням динамічної бібліотеки нам треба зібрати файл командою:

gcc -Wall -L ./ main2.c -lfunc2 -o main2

```
func2.c  funcdyn.o  func.h  libfunc2.so  main2.c
[root@ksp dynlib]# gcc -Wall -L ./ main2.c -lfunc2 -o main2
[root@ksp dynlib]# ls
func2.c  funcdyn.o  func.h  libfunc2.so  main2  main2.c
```

Проблема з цим прикладом полягає в тому, що ми маємо фіксований шлях для бібліотеки.

```
[root@ksp dynlib]# ./main2
./main2: error while loading shared libraries: libfunc2.so: cannot open shared o
bject file: No such file or directory
```

Розташування спільних бібліотек може відрізнятися залежно від системи.

При використанні ОС Linux програма, що використовує динамічні бібліотеки, шукає всі бібліотеки, від яких вона залежить, за шляхами, прописаними у змінній середовища LD_LIBRARY_PATH.

Отримати значення цієї змінної можна такою вказівкою Терміналу:

echo $LD_LIBRARY

```
[root@ksp dynlib]# echo $LD_LIBRARY

[root@ksp dynlib]#
```

Може трапитися, що змінну середовища LD_LIBRARY_PATH ще не визначено. У цьому випадку буде виведено порожній рядок.

Перевірити доступ відкомпільованого файлу **main2** до замовлених динамічних бібліотек можна такою вказівкою Терміналу:

ldd main2

```
[root@ksp dynlib]# ldd main2
        linux-vdso.so.1 (0x00007f6696097000)
        libfunc2.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x00007f6695eb2000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f6696099000)
```

Після виконання вказівок Терміналу:

LD_LIBRARY_PATH=/home/dynlib

export LD_LIBRARY_PATH

буде надано можливість доступу до динамічних бібліотек, розташованих у папці /home/dynlib. Після цього можна компілювати код програми main2.c і запустити її на виконання вказівкою Терміналу з повним шляхом до програми.

```
[root@ksp dynlib]# LD_LIBRARY_PATH=/home/dynlib
[root@ksp dynlib]# export LD_LIBRARY_PATH
[root@ksp dynlib]# ldd main2
        linux-vdso.so.1 (0x00007fcb23ec9000)
        libfunc2.so => /home/dynlib/libfunc2.so (0x00007fcb23ebe000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fcb23cdf000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcb23ecb000)
```

Виконуємо файл

```
[root@ksp dynlib]# ./main2
Hello linux world
Hello from func
x + 1 = 9
sin(2) = 0.909297
```

Описаний вище встановлення значення змінної LD_LIBRARY_PATH поширюється лише на активний сеанс терміналу.

Якщо ви хочете, щоб ваша бібліотека була встановлена в системі, ви можете скопіювати файли .so в один із стандартних каталогів - /usr/lib і запустити ldconfig.

Будь-яка програма, яка використовує вашу бібліотеку, може просто використовувати –lfunc2 і система знайде її в /usr/lib.

4. **Завдання на виконання**.

1. Ознайомитися з теоретичними матеріалом по лабораторній роботі.
2. Встановити компілятор ***gcc***.
3. Використовуючи наведений код (або використовуючи власний), створіть скомпільований файл **a.out**. Запустіть на виконання та зробіть скріни.
4. Використовуючи наведений код (або використовуючи власний), виконайте компіляцію та компонування файлу **main.c** з використанням опції -**Wall** (зробити скрін).
5. Використовуючи наведений код (або використовуючи власний), розбийте програму на різні файли.
6. Виконайте компіляцію окремо кожного файлу та отримайте об’єктні файли.
7. Зробіть компонування отриманих об’єктних файлів та отримайте виконуваний фай. Запустіть на виконання (зробити скріни).
8. Використовуючи наведений код (або використовуючи власний), створіть два файли **func2.c** та **main2.c**.
9. Виконайте компіляцію окремо кожного файлу.
10. Зі скомпільованого файлу **func2.o** створіть статичну бібліотеку **lib func2. a**.
11. Підключіть отриману бібліотеку до файлу **main2.c** та виконайте компонування.
12. Запустіть на виконання (зробити скріни).
13. Використовуючи наведений код (або використовуючи власний), створіть два файли **func2.c** та **main2.c**.

14. Генерований компілятором код повинен бути незалежним від адрес, тому використайте ключ -fPIC для отримання обєктного файлу **funcdyn.o**.

15. Створшть динамічну бібліотеку **libfunc2.so**, використовувати ключ -shared:

16. Виконайте компіляцію та компонування для отримання результуючого файлу з використанням динамічної бібліотеки.

17. Надайте доступ до динамічної бібліотеки та запустіть виконуваний файл (зробити скріни).

Підготувати звіт про виконання лабораторної роботи та зробіть висновки.

4. Результати виконання роботи та оформлення звіту

Зміст звіту:

1) титульний аркуш;
2) короткі теоретичні відомості;
3) скріни з виконаними завданнями та висновки.

Контрольні питання

1. Наведіть три опції компіляції.
2. Поясніть функції опції –Wall.
3. Для чого виконується розбиття програми на окремі файли?
4. Яка різниця між статичною та динамічною бібліотеками?
5. Команди для створення статичної бібліотеки.
6. Команди для для створення динамічної бібліотеки.
7. Додавання динамічної бібліотеки на поточний сеанс.
8. Встановлення динамічної бібліотеки в системі для постійного використання.

Методичну розробку склав
доцент кафедри інформаційних технологій
та систем електронних комунікацій Юрій БОРЗОВ