



Лабораторна робота №11

Тема: «Етапи компіляції C-програм та автоматизація побудови C-програм»

Викладач: Олександр А. Блажко,

доцент кафедри ІС Одеської політехніки, blazhko@ieee.org

Мета роботи: дослідити багатоетапний процес побудови C-програм компілятором *gcc*, отримати навички створення C-програм взаємодії із СКБД *PostgreSQL* та отримати навички із автоматизації побудови C-програм командою *make*

1 Теоретичні відомості

1.1 Особливості компіляції програм на мові програмування C

1.1.1 Початок боротьби за *Unix*-свободу та *GNU*-програмні інструменти

Всі редакції ОС *Unix* до 6-ї редакції (до 1975 року) вільно розповсюджувалися серед університетів світу. Компанія *Bell Labs (AT&T Corporation)* у 1975 році випустила 6-ту редакцію *Unix* як першу комерційну версію, хоча і з можливістю доступу до *Source*-кодів програмного забезпечення.

У 1983 році з'явилася одна з перших комерційно успішних версій ОС *Unix* - *AT&T UNIX System V*. *GNU* (рекурсивне скорочення від фрази «*GNU is Not Unix*») – проєкт зі створення вільної *Unix*-подібної ОС, започаткований у 1983 році Річардом Столменом та в подальшому підтримуваний Фондом вільного програмного забезпечення (*Free Software Foundation, FSF*).

В рамках проєкту *GNU* було розроблено велику кількість високоякісного та поширеного вільного програмного забезпечення, включаючи текстовий редактор *Emacs*, збірку компіляторів *GNU (GCC — GNU Compiler Collection)* і *GNU Debugger (GDB)*.

GNU toolchain — набір необхідних пакетів програм для компіляції та генерації *executable*-файлу із сирцевих текстів програм.

Компілятори *GNU* розроблені і підтримуються спільнотою *GNU*. Це вільне програмне забезпечення, яке розповсюджується *FSF*. Вони використовуються для компіляції більшості програм проєкту і багатьох інших.

GNU C складається з двох частин:

- набір компіляторів з різних мов в абстрактне дерево, незалежне від мови і процесора – так звані *frontend compilers*;
- набір компіляторів, які перетворюють абстрактне дерево в об'єктний код для різних процесорів - так звані *backend compilers*.

Така схема дозволяє робити код мобільним: будь-який код, скомпільований для одного процесора, швидше за все скомпілюється і для інших.

Програми проекту *GNU* поширюються в першу чергу у вигляді текстів програмного коду - *source*-коду.

1.1.2 GCC - *General Public License Compiler Collection*

В *Unix*-подібних ОС для компіляції програм часто використовується *GCC*-набір (*GCC* - *General Public License Compiler Collection*) - набір компіляторів для різних мов програмування.

Спочатку так називався компілятор з мови *C*. Пізніше він був розширений для підтримки мов *C++*, *Fortran*, *Java*, *Ada*, *Objective-C* та інших.

Компіляція програми на мові *C* за допомогою команди *gcc* передбачає такі послідовні етапи:

- *preprocessing*-етап – попередня обробка програмного коду;
- *compilation*-етап – трансляція програмного коду у код на мові програмування *Assembler*;
- *assembly*-етап – трансляція програмного коду з мови програмування *Assembler* в об'єктний модуль з машинними командами;
- *linking*-етап – збирання всіх об'єктних модулів, підключення статичних програмних бібліотек та встановлення зв'язків із функціями динамічних програмних бібліотек.

Для створення *executable*-файлу *gcc*-компілятором достатньо вказати опцію *-o* для визначення назви *executable*-файлу. Компілятор автоматично виконає усі етапи, але їх виклики можна відслідковувати додавши опцію *-v* до команди *gcc*.

Розглянемо кожен з даних етапів на прикладі файлу *hello.c*, програмний код якого представлено на рисунку 1.

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

Рис. 1 – Приклад файлу *helloworld.c*

Розглянувши кожен етап окремо можна краще зрозуміти роботу компілятора. Програма *helloworld* є простою програмою, але в ній використано зовнішні бібліотеки, тому під час компіляції будуть пройдені усі вище вказані етапи.

1.1.3 *Preprocessing*-етап

Preprocessing-етап виконує наступні дії:

- розгортає макроси-константи – знаходить директиви *#define* як умовні функції обробки та заміни програмного коду та замінює їх на макровизначення;
- розгортає зміст *header*-файлів – знаходить директиви *#include* та замінює їх на зміст відповідних *header*-файлів.

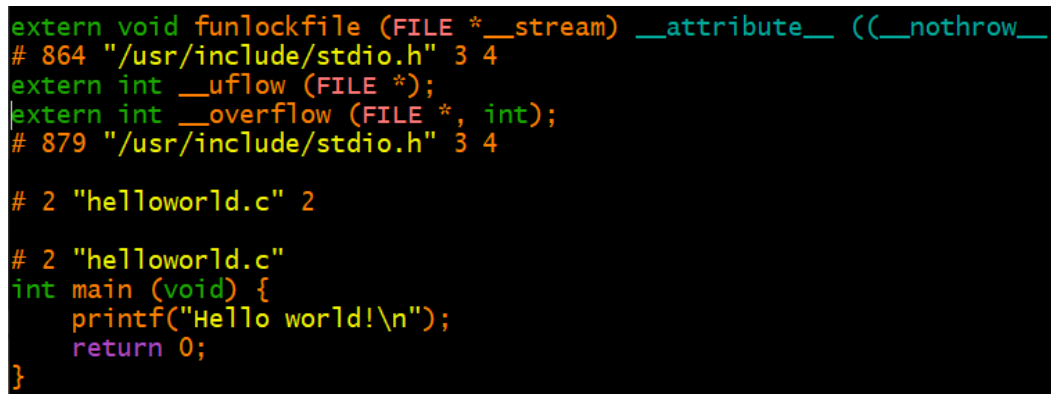
Для перегляду результату створення *gcc* викликає таку команду:

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

Результатом виконання команди є файл *helloworld.i*, для якого:

- додано *source*-код з розгорнутими макросами і вставленими *header*-файлами;
- видалено всі коментарі зі старого *source*-коду;
- додано спеціальні коментарі, які вказують на номери рядків старого *source*-коду.

Фрагмент файлу *helloworld.i* представлено на рисунку 2.



```
extern void funlockfile (FILE *__stream) __attribute__((__nothrow__  
# 864 "/usr/include/stdio.h" 3 4  
extern int __uflow (FILE *);  
extern int __overflow (FILE *, int);  
# 879 "/usr/include/stdio.h" 3 4  
  
# 2 "helloworld.c" 2  
  
# 2 "helloworld.c"  
int main (void) {  
    printf("Hello world!\n");  
    return 0;  
}
```

Рис. 2 – Фрагмент файлу *helloworld.i*

1.1.4 *Compilation*-етап

Результатом виконання *compilation*-етапу є файл *helloworld.s*, який містить програмний код на мові *Assembler*. Файл *hello.s* є проміжним програмним кодом, тому компілятор за замовчуванням його видаляє. Щоб зберегти *helloworld.s* і переглянути його вміст необхідно виконати таку команду:

```
gcc -S helloworld.i -o helloworld.s
```

Зміст файлу *helloworld.s*, створеного з файлу *helloworld.i*, представлено на рисунку 3.

При створенні асемблерного файлу можна провести оптимізацію програмного коду, використовуючи опцію *-O3*, наприклад:

```
gcc -O3 -S helloworld.i -o helloworld.s
```

```

.file "helloworld.c"
.text
.section .rodata
.LC0:
.string "Hello world!"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 8.5.0 20210514 (Red Hat 8.5.0-4)"
.section .note.GNU-stack,"",@progbits

```

Рис. 3 – Приклад файлу *hello.s*

На рисунку 4 наведено приклад файлу *helloworld.s* після оптимізації коду. Якщо порівняти код на рисунку 3 з кодом на рисунку 4, можна помітити зменшення кількості рядків коду та зміни окремих команд.

```

.file "helloworld.c"
.text
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "Hello world!"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
subq $8, %rsp
.cfi_def_cfa_offset 16
movl $.LC0, %edi
call puts
xorl %eax, %eax
addq $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 8.5.0 20210514 (Red Hat 8.5.0-4)"
.section .note.GNU-stack,"",@progbits

```

Рис. 4 – Приклад файлу *helloworld.s* після оптимізації коду

1.1.5 *Assembly*-етап

Метою *Assembly*-етапу є перетворення коду на мові програмування *Assembler* в об'єктний код з машинними кодами та генерація *object*-файлу. Якщо у коді є виклики зовнішніх функцій, *Assembler* залишає адреси початку цих зовнішніх функцій невизначеними. Їх значення будуть заповнені на *Linking*-етапі.

Для створення *object*-файлу необхідно виконати додати опцію `-c`:

```
gcc -c helloworld.s -o helloworld.o
```

Результатом роботи команди є файл *helloworld.o*, який містить програму *helloworld* у вигляді машинних команд, де є поки що невизначена адреса початку функції *printf()*.

1.1.6 *Linking*-етап

Останнім етапом зі створення *executable*-файлу є *Linking*-етап, який використовує окрему команду *ld*.

На практиці *executable*-файли потребують багатьох зовнішніх функцій і динамічних (*run-time*) бібліотек *C*, які враховують особливості ОС та апаратного забезпечення. Тому команди лінковки, які внутрішньо автоматично виконує *gcc*, можуть бути складними.

Для того, щоб дізнатися про опції команди *ld*, які використовує *gcc*, можна при створенні *executable*-файлу вказати *gcc* опції `-Wl, -v`, наприклад:

```
gcc -Wl,-v helloworld.o -o helloworld
```

Результатом виконання може бути наступний рядок:

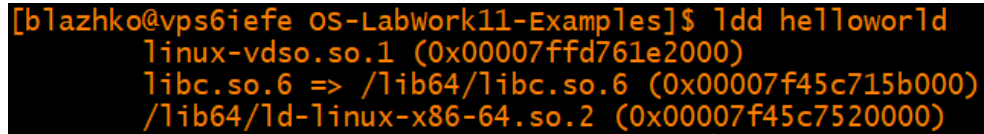
```
/usr/bin/ld -plugin /usr/libexec/gcc/x86_64-redhat-  
linux/8/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-  
redhat-linux/8/lto-wrapper -plugin-opt=-  
fresolution=/tmp/ccFUcWp0.res -plugin-opt=-pass-through=-lgcc -  
plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -  
plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s  
--build-id --no-add-needed --eh-frame-hdr --hash-style=gnu -m  
elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o  
helloworld /usr/lib/gcc/x86_64-redhat-  
linux/8/../../../../lib64/crt1.o /usr/lib/gcc/x86_64-redhat-  
linux/8/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-  
linux/8/crtbegin.o -L/usr/lib/gcc/x86_64-redhat-linux/8 -  
L/usr/lib/gcc/x86_64-redhat-linux/8/../../../../lib64 -  
L/lib/../../lib64 -L/usr/lib/../../lib64 -L/usr/lib/gcc/x86_64-redhat-  
linux/8/../../../../ -v helloworld.o -lgcc --as-needed -lgcc_s --no-  
as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed  
/usr/lib/gcc/x86_64-redhat-linux/8/crtend.o /usr/lib/gcc/x86_64-  
redhat-linux/8/../../../../lib64/crtn.o
```

Вказана команда лінкує *object*-файл *helloworld.o* із стандартною бібліотекою *C* і створює вихідний *executable*-файл *helloworld*.

Для перевірки списку підключених динамічних (*run-time*) бібліотек використовується команда *ldd*, наприклад:

```
ldd helloworld
```

На рисунку 5 представлено результат виконання команди *ldd*



```
[blazhko@vps6iefe OS-Labwork11-Examples]$ ldd helloworld
linux-vdso.so.1 (0x00007ffd761e2000)
libc.so.6 => /lib64/libc.so.6 (0x00007f45c715b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f45c7520000)
```

Рис. 5 – Приклад виконання команди *ldd*

Таким чином процес створення *executable*-файлу *helloworld* можна зобразити як послідовність підпроцесів: *helloworld.c -> PREPROCESSOR(helloworld.c) -> helloworld.i -> COMPILER(helloworld.i) -> helloworld.s -> ASSEMBLER(helloworld.s) -> helloworld.o -> LINKER(helloworld.o) -> helloworld*

1.2 Програмування на мові C програм взаємодії із сервером СКБД *PostgreSQL*

1.2.1 Приклади C-функцій встановлення та завершення з'єднання із сервером СКБД *PostgreSQL*

Всі визначення C-функцій взаємодії з СКБД *PostgreSQL* описано у заголовному файлі *libpq-fe.h*.

Для встановлення з'єднання із сервером СКБД використовується функція *PqsetdbLogin* із наступними аргументами:

```
PGconn* PqsetdbLogin(
    const char *pghost, // адреса серверу
    const char *pgport, // порт прослуховування сервера
    const char *pgoptions, // додаткові параметри
    const char *pgtty, // раніше визначав потік виводу
    const char *dbName, // назва БД
    const char *login, // ім'я користувача СКБД
    const char *pwd // пароль користувача СКБД
);
```

Якщо встановлюється з'єднання із сервером СКБД в межах сервера Linux, тоді достаньо використати параметри *dbName*, *login* та *pwd* (якщо встановлено).

Для перевірки встановлення з'єднання використовується функція *PQstatus*:

```
ConnStatusType PQstatus(const PGconn * conn);
```

Статус може приймати одне з значень: *CONNECTION_OK* та *CONNECTION_BAD*.

Для закриття з'єднання з сервером використовується функція *PQfinish*:

```
void PQfinish(PGconn * conn);
```

На рисунку 6(a) наведено приклад програмного коду програми *db_connect.c* із вбудованими значеннями назви бази даних та імені користувача, а на рисунку 6(b) назви бази даних та ім'я користувача передаються через параметри командного рядку.

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

int main (void) {
    PGconn *conn;
    char db_name[] = "blazhko";
    char user_name[] = "blazhko";

    conn = PQsetdbLogin(NULL, NULL, NULL, NULL, db_name, user_name, NULL);
    if (PQstatus(conn) == CONNECTION_OK )
        printf("Connection to database %s is successful!\n", db_name);
    else {
        fprintf(stderr, "Connect to database %s failed: %s",
            db_name, PQerrorMessage(conn));
        return EXIT_FAILURE;
    }
    PQfinish(conn);
    return EXIT_SUCCESS;
}
```

(a) приклад із вбудованими значеннями назви бази даних та імені користувача

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "libpq-fe.h"

int main (int argc, char * argv[]) {
    PGconn *conn;
    char db_name[32];
    char user_name[32];
    if(argc < 3) {
        fprintf(stderr, "Usage: %s database user\n", argv[0]);
        return EXIT_FAILURE;
    }
    strcpy(db_name, argv[1]);
    strcpy(user_name, argv[2]);
    conn = PQsetdbLogin(NULL, NULL, NULL, NULL, db_name, user_name, NULL);
    if (PQstatus(conn) == CONNECTION_OK )
        printf("Connection to database %s is successful!\n", db_name);
    else {
        fprintf(stderr, "Connect to database %s failed: %s\n",
            db_name, PQerrorMessage(conn));
        return EXIT_FAILURE;
    }
    PQfinish(conn);
    return EXIT_SUCCESS;
}
```

(b) назва бази даних та ім'я користувача – параметри командного рядку

Рис. 6 – Приклад програмного коду з функцією *PQsetdbLogin*

Для передачі на сервер *SQL*-команд використовується функція *Pqexec* з наступними аргументами:

```
PGresult *Pqexec(  
    PGconn *conn, // змінна з описом успішного підключення  
    const char *command // рядок з SQL-командою  
);
```

Для перевірки успішності результату виконання команди використовується функція *PqresultStatus*, яка містить наступні аргументи:

```
ExecStatusType PqresultStatus(const PGresult *res);
```

Варіанти статусу відповіді:

– *PGRES_COMMAND_OK* – успішне завершення команди, яка не повертає даних, наприклад, для команд *LOCK TABLE*, *INSERT*, *UPDATE*, *DELETE*;

– *PGRES_TUPLES_OK* – успішне завершення команди, яка повертає дані, наприклад, для команди *SELECT*.

Для отримання повідомлення про помилку, пов'язаного з останньою командою, використовується функція *PqErrorMessage*, яка містить наступні аргументи:

```
char *PqErrorMessage(PGconn *conn);
```

Для звільнення області пам'яті, пов'язаної із *Pgresult*, використовується функція *PQclear*, яка містить наступні аргументи:

```
void PQclear (PGresult * res);
```

На рисунку 7 наведено приклад програмного коду транзакції із командою видалення рядків таблиці.


```

ExecStatusType delete_bills(PGconn* conn) {
    PGresult *res;
    res = PQexec(conn, "START TRANSACTION");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "START TRANSACTION failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "LOCK TABLE bills IN EXCLUSIVE MODE");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        fprintf(stderr, "LOCK TABLE failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "DELETE FROM bills");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        fprintf(stderr, "DELETE failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "COMMIT");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "COMMIT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    printf("Deleting bills is successfull!\n");
    return PGRES_COMMAND_OK;
}

```

Рис. 7 – Приклад програмного коду транзакції із командою видалення рядків таблиці

На рисунку 8 наведено приклад програмного коду транзакції із командою внесення рядку таблиці.

```

ExecStatusType add_bill(PGconn* conn) {
    PGresult *res;
    res = PQexec(conn, "START TRANSACTION");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "START TRANSACTION failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "LOCK TABLE bills IN EXCLUSIVE MODE");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        fprintf(stderr, "LOCK TABLE failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "INSERT INTO bills VALUES (1, 100, 'Blazhko')");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        fprintf(stderr, "DELETE failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "COMMIT");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "COMMIT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    printf("Adding bills is successful!\n");
    return PGRES_COMMAND_OK;
}

```

Рис. 8 – Приклад програмного коду транзакції із командою внесення рядку таблиці

Для отримання кількості рядків *SELECT*-запиту, використовується функція *PQntuples*, яка містить наступні аргументи:

```
int PQntuples(const PGresult *res);
```

Для отримання значення одного стовпчика за номером *column_number* (0 – перший стовпчик) з рядка таблиці за номером *row_number* (0 – перший рядок), яке вже міститься у структурі *Pgresult*, використовується функція *Pqgetvalue*, яка містить наступні аргументи:

```
char *Pqgetvalue(const PGresult *res,
                  int row_number,
                  int column_number);
```

На рисунку 9 наведено приклад програмного коду транзакції із командою отримання рядків таблиці.

```

ExecStatusType get_bills(PGconn* conn) {
    PGresult *res;
    res = PQexec(conn, "START TRANSACTION");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "START TRANSACTION failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "LOCK TABLE bills IN SHARE MODE");
    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
        fprintf(stderr, "LOCK TABLE failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    PQclear(res);
    res = PQexec(conn, "SELECT n_bill, bill_count, name FROM bills");
    if (PQresultStatus(res) != PGRES_TUPLES_OK) {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        return PGRES_BAD_RESPONSE;
    }
    printf("%-10s%-12s%-10s\n", "n_bill", "bill_count", "name");
    for (int i = 0; i < PQntuples(res); i++)
        printf("%-10s%-12s%-10s", PQgetvalue(res, i, 0), PQgetvalue(res, i, 1),
            PQgetvalue(res, i, 2));
    printf("\n");
    PQclear(res);
    return PGRES_COMMAND_OK;
}

```

Рис. 9 – Приклад програмного коду транзакції із командою отримання рядків таблиці

1.2.2 Особливості компіляції C-програм взаємодії з СКБД *PostgreSQL*

Для взаємодії із СКБД *PostgreSQL* C-програми використовують динамічну програмну бібліотеку *libpq*, яка містить важливі функції:

- встановлення/закриття з'єднання із сервером;
- виконання *SQL*-запитів та обробки відповідей на ці запити.

Для підключення функцій цієї бібліотеки необхідно:

- увімкнути заголовний файл *libpq-fe.h*;
- повідомити компілятор *gcc* назву каталогу, в якому розміщений файл заголовків, використовуючи аргумент командного рядка *-I каталог*;
- повідомити компілятор *gcc* про назву каталогу, в якому розміщено файл програмної бібліотеки, використовуючи аргумент командного рядка *-L каталог*;
- повідомити компілятор *gcc* про назву програмної бібліотеки, використовуючи аргумент командного рядка *-l файл*, при цьому врахувати, що назва бібліотеки не включає приставку *lib*, тобто *-l pq*

Для швидкого визначення назви каталогу з файлами заголовків СКБД *PostgreSQL*, встановленими в ОС, можна використовувати команду *pg_config --includedir*, наприклад:

```
pg_config --includedir
```

Результат виконання команди:

```
/usr/include
```

Для швидкого визначення назви каталогу з файлами бібліотеки СКБД *PostgreSQL*, встановленими в ОС, можна використовувати команду *pg_config --libdir*, наприклад:

```
pg_config --libdir
```

Результат виконання команди:

```
/usr/lib64
```

Приклад командного рядку для створення програми *db_connect.c*:

```
gcc db_connect.c -o db_connect -I/usr/include \
-L/usr/lib64 -lpq
```

1.3 Модульне програмування та автоматизація побудови С-програм

1.3.1 Модульне програмування

Відомо, що модульне програмування – це організація програми у вигляді множини невеликих незалежних програмних блоків (модулів), яка зменшує час на створення програми.

Для реалізації модульного програмування необхідно:

- всі чотири функції розмістити в окремих *c*-файлах, приклад однієї з функцій показано на рисунку 10;
- створити *header*-файлу *bill.h*, в якому задекларувати назви всіх функцій, як це показано на рисунку 11, та додати його до файлів з функціями (приклад на рисунку 12);
- для кожного *c*-файлу створити об'єктний файл;
- створити *executable*-файл, об'єднавши всі об'єктні файли.

```
#include "bill.h"

PGconn* connect_db (void) {
    PGconn *conn;
    char db_name[] = "blazhko";
    char user_name[] = "blazhko";

    conn = PQsetdbLogin(NULL, NULL, NULL, NULL, db_name, user_name, NULL);
    if (PQstatus(conn) == CONNECTION_OK ) {
        printf("Connection to database %s is successfull!\n", db_name);
        return conn;
    }
    else {
        fprintf(stderr, "Connect to database %s failed: %s",
            db_name, PQerrorMessage(conn));
        return NULL;
    }
}
```

Рис. 10 – Приклад програмного модуля з функцією *connect_db*

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

PGconn* connect_db (void);
ExecStatusType remove_bills(PGconn* conn);
ExecStatusType add_bill(PGconn* conn);
ExecStatusType get_bills(PGconn* conn);
```

Рис. 11 – Приклад *header*-файлу *bill.h*

```
#include "bill.h"

int main (void) {
    PGconn *conn;
    conn = connect_db();
    if (conn == NULL)
        return EXIT_FAILURE;
    if (remove_bills(conn) != PGRES_COMMAND_OK)
        return EXIT_FAILURE;
    if (add_bill(conn) != PGRES_COMMAND_OK)
        return EXIT_FAILURE;
    if (get_bills(conn) != PGRES_COMMAND_OK)
        return EXIT_FAILURE;
    PQfinish(conn);
    return EXIT_SUCCESS;
}
```

Рис. 12 – Приклад програмного модуля із *main*-функцією

1.3.2 Автоматизація побудови С-програм

В перших редакціях *Unix*-подібних ОС побудова С-програм забезпечувалася через опис команд компіляції, які зберігалися у скрипт-файлах, що ускладнювалося керування проєктів з великою кількістю програмних модулів, створених за модульним принципом програмування. Тому у 1977 році Стюартом Фельдманом, співробітником компанії *Bell Labs*, для версії *PWB/UNIX (for Programmer's Workbench)* була створена команда *make* як утиліта для автоматичної побудови програм. Дії, що повинні виконати *make*, описують в конфігураційному файлі з назвою *Makefile*.

Сьогодні існує багато інших систем керування проєктами, але команда *make* залишається популярною серед розробників вбудованих систем з використанням мови С.

Makefile містить «правила» у такій формі:

target: dependencies

<табуляція> system command(s)

Табуляція є обов'язковою частиною синтаксиса.

Метою (*target*) може бути:

- ім'я файлу, який генерується програмою, наприклад, *executable*-файл або об'єктний файл;

- дія, яку необхідно додатково виконати через перелік *bash*-команд, наприклад, «*install*», «*clean*».

Залежність (*dependencies*) – це множина файлів, які використовуються як вхідні дані для створення мети.

На рисунку 13 представлено приклад конфігураційного файлу *Makefile*:

- використовуються змінні *GCC*, *INCLUDE*, *LIB* та *OBJ* для спрощення подальшого процесу їх можливої зміни та зменшення овжини рядків описи мети;

- для команди *gcc* вказано опцію *-Wall*, яка вказує на виведення всіх попереджень під час побудови програми;

- для кожного програмного модуля з описом функцій створено свій опис мети;

- створено опис мети *install* для спрощення процесу розміщення *executable*-файлу у каталозі, який присутній у змінній *PATH* для подальшого виконання *executable*-файлу без вказування повного шляху;

- створено опис мети *clean* для очистки каталогу від всіх *object*-файлів та *executable*-файлу.

```
GCC = gcc -Wall
INCLUDE = -I/usr/include
LIB = -L/usr/lib64 -lpq
OBJ = connect_db.o remove_bills.o add_bill.o get_bills.o db_program2.o
db_program2: $(OBJ)
    $(GCC) $(OBJ) -o db_program2 $(INCLUDE) $(LIB)
db_program2.o: db_program2.c
    $(GCC) -c -o db_program2.o db_program2.c $(INCLUDE) $(LIB)
connect_db.o: connect_db.c
    $(GCC) -c -o connect_db.o connect_db.c $(INCLUDE) $(LIB)
remove_bills.o: remove_bills.c
    $(GCC) -c -o remove_bills.o remove_bills.c $(INCLUDE) $(LIB)
add_bill.o: add_bill.c
    $(GCC) -c -o add_bill.o add_bill.c $(INCLUDE) $(LIB)
get_bills.o: get_bills.c
    $(GCC) -c -o get_bills.o get_bills.c $(INCLUDE) $(LIB)
install:
    cp ./db_program2 /home/blazhko/bin/
clean:
    rm -f *.o
    rm db_program2
```

Рис. 13 – Приклад конфігураційного файлу *Makefile*

На рисунку 14 наведено приклади виконання команди *make*:

- якщо команда виконується без вказування наві мети, тоді виконується перша мета із файлу Makefile;
- якщо повторно виконати команду *make*, вона перевіре всі залежності на наявність файлів, для яких дата останньої зміни менше дати запуску команди, і, якщо всі файли із залежностей присутні та є свіжими, команда повідомить про це фразою «*is up to date*».

```
[blazhko@vps6iefe OS-Labwork11-Examples]$ make clean
rm -f *.o db_program2
[blazhko@vps6iefe OS-Labwork11-Examples]$ make
gcc -Wall -c -o connect_db.o connect_db.c -I/usr/include -L/usr/lib64 -lpq
gcc -Wall -c -o remove_bills.o remove_bills.c -I/usr/include -L/usr/lib64 -lpq
gcc -Wall -c -o add_bill.o add_bill.c -I/usr/include -L/usr/lib64 -lpq
gcc -Wall -c -o get_bills.o get_bills.c -I/usr/include -L/usr/lib64 -lpq
gcc -Wall connect_db.o remove_bills.o add_bill.o get_bills.o \
    -o db_program2 db_program2.c -I/usr/include -L/usr/lib64 -lpq
[blazhko@vps6iefe OS-Labwork11-Examples]$ make install
cp ./db_program2 /home/blazhko/bin/
[blazhko@vps6iefe OS-Labwork11-Examples]$ make
make: 'db_program2' is up to date.
[blazhko@vps6iefe OS-Labwork11-Examples]$ |
```

Рис. 14 – Приклади виконання команди *make*

2 Завдання лабораторної роботи

Виконати наступні дії з підготовки до виконання завдань роботи:

- 1) встановити з'єднання з *Linux*-сервером з *IP*-адресою = 46.175.148.116;
- 2) перейти до каталогу *Git*-репозиторія;
- 3) створити нову *Git*-гілку з назвою «*Laboratory-work-11*»;
- 4) перейти до роботи зі створеною гілкою;
- 5) створити каталог з назвою «*Laboratory-work-11*»;
- 6) перейти до каталогу «*Laboratory-work-11*»;
- 7) в каталозі «*Laboratory-work-11*» створити файл *README.md* та додати до файлу рядок тексту із темою лабораторної роботи «*Етапи компіляції C-програм та автоматизація побудови C-програм*» як заголовок 2-го рівня *Markdown*-форматування;
- 8) виконати операції з оновлення *GitHub*-репозиторію змінами *Git*-репозиторія через послідовність *Git*-команд *add*, *commit* із коментарем «*Changed by Local Git*» та *push*;
- 9) на веб-сервісі *GitHub* перейти до створеної гілки «*Laboratory-work-11*»;
- 10) перейти до каталогу «*Laboratory-work-11*» та розпочати процес редагування файлу *README.md* ;
- 11) в подальшому за результатами рішень кожного наступного розділу завдань до файлу *README.md* додавати рядки як заголовки 3-го рівня *Markdown*-форматування з назвами розділу та знімки екрану з підписами до знімків екранів з описом пунктів завдань.

2.1 Побудова програми з'єднання з СКБД *PostgreSQL* на основі монолітної C-програми

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.1.1 Створити C-програму з назвою «*db_connect.c*», яка:

- встановлює з'єднання із СКБД *PostgreSQL*;
- під час з'єднання використовує назву БД та користувача з попередньої лабораторної роботи, які вбудовані в програмний код;
- обробляє результат з'єднання (успішне та помилкове), виводячі на екран відповідні повідомлення, які враховують назву БД;
- увесь програмний код розміщується лише у функції *main*.

2.1.2 Скомпілювати C-програму, враховуючи каталоги з *header*-файлами та бібліотеками СКБД *PostgreSQL*.

Перевірити роботу *executable*-файла.

2.1.3 Створити C-програму з назвою «*db_connect_param.c*», яка повторює всі дії C-програми з назвою «*db_connect.c*», але назву бази даних та ім'я користувача програма повинна брати як параметри командного рядку.

2.1.4 Скомпілювати C-програму, враховуючи каталоги з *header*-файлами та бібліотеками СКБД *PostgreSQL*.

Перевірити роботу *executable*-файла за двома варіантами назви БД: правильна назва та будь-яка неправильна БД.

2.2 Побудова програми з'єднання з СКБД *PostgreSQL* за модульним принципом програмування

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.2.1 Змінити код C-програми, враховуючи модульний принцип програмування:

- створити C-файл з назвою «*connect_назва таблиці.c*», який містить програмний код з'єднання із СКБД *PostgreSQL* у вигляді функції з назвою «*connect_назва таблиці*», де «*назва таблиці*» - назва реляційної таблиці з попередньої лабораторної роботи;

- створити *header*-файл за шаблоном «*назва таблиці.h*» та додати до файлу декларацію створеної функції;

- створити C-файл з назвою «*назва таблиці.c*», який містить *main*-функцію з викликом функції з назвою «*connect_назва таблиці*».

2.2.2 Побудувати *executable*-файл через кроки:

- створити *object*-файл з назвою «*connect_назва таблиці.o*» для «*connect_назва таблиці.c*», враховуючи каталоги з *header*-файлами, бібліотеками СКБД *PostgreSQL*

- створити *object*-файл з назвою «*назва таблиці.o*» для файлу «*назва таблиці.c*», враховуючи каталоги з *header*-файлами, бібліотеками СКБД *PostgreSQL*

- створити *executable*-файл з назвою «*назва таблиці*», враховуючи каталоги з *header*-файлами, бібліотеками СКБД *PostgreSQL*, а також створені *object*-файли з назвою «*connect_назва таблиці.o*» та «*назва таблиці.o*».

Перевірити роботу *executable*-файла.

2.3 Побудова програми з'єднання з СКБД *PostgreSQL* через команду *make*

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.3.1 Створити *Makefile*, який містить наступний опис мети:

- мета створення *object*-файлу з назвою «*connect_назва таблиці.o*»;
- мета створення *object*-файлу з назвою «*назва таблиці.o*»;
- мета створення *executable*-файлу з назвою «*назва таблиці*».

2.3.2 Викнати команду *make* для побудови програми.

2.4 Побудова програми видалення рядку реляційної таблиці

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.4.1 Створити файл як програмний модуль з назвою «*remove_назва таблиці.c*» із описом функції видалення рядку реляційної таблиці, яка повинна містити:

- команди транзакції (*START TRANSACTION, LOCK TABLE назва таблиці ..., DELETE FROM назва таблиці ..., COMMIT*);
- повідомлення про результат виконання кожної команди.

Команду *DELETE FROM* створити за прикладом з попередньої лабораторної роботи.

2.4.2 Оновити раніше створені файли:

- додати до файлу «*назва таблиці.h*» декларацію нової функції;
- виконати виклик нової функції із *main*-функції файлу «*назва таблиці.c*»
- додати опис нової мети у файл *Makefile*

2.4.3 Скомпілювати C-файли програмних модулів командою *make*.

Перевірити роботу *executable*-файла.

2.5 Побудова програми додавання рядку реляційної таблиці

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.5.1 Створити файл як програмний модуль з назвою «*add_назва таблиці.c*» із описом функції додавання рядку реляційної таблиці, яка повинна містити:

- команди транзакції (*START TRANSACTION, LOCK TABLE назва таблиці ..., INSERT INTO назва таблиці ..., COMMIT*);
- повідомлення про результат виконання кожної команди.

Команду *INSERT INTO* створити за прикладом з попередньої лабораторної роботи.

2.5.2 Оновити раніше створені файли:

- додати до файлу «*назва таблиці.h*» декларацію нової функції;
- виконати виклик нової функції із *main*-функції файлу «*назва таблиці.c*»
- додати опис нової мети у файл *Makefile*

2.5.3 Скомпілювати C-файли програмних модулів командою *make*.

2.6 Побудова програми перегляду рядків реляційної таблиці

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.6.1 Створити файл як програмний модуль з назвою «*get_назва_таблиці*».с із описом функції перегляду рядків реляційної таблиці, яка повинна містити:

- команди транзакції (*START TRANSACTION, LOCK TABLE назва_таблиці ..., SELECT ... FROM назва_таблиці ..., COMMIT*);
- повідомлення про результат виконання кожної команди.

Команду *SELECT ... FROM* створити за прикладом з попередньої лабораторної роботи.

2.6.2 Оновити раніше створені файли:

- додати до файлу «*назва_таблиці*».h декларацію нової функції;
- виконати виклик нової функції із *main*-функції файлу «*назва_таблиці*».с
- додати опис нової мети у файл *Makefile*

2.6.3 Скомпілювати С-файли програмних модулів командою *make*.

Перевірити роботу *executable*-файла.

2.7 Додаткове налаштування процесу керування файлами через команду *make*

Знаходячись у каталозі «*Laboratory-work-11*», виконати наступні завдання.

2.7.1 Додати до файлу *Makefile* наступні описи мети:

install – копіювання *executable*-файлу до каталогу */home/ваш_користувач/bin* (попередньо створити такий каталог);

clean – очистка всіх *object*-файлів та *executable*-файлу.

2.7.2 Виконати команду *make* з метою *clean*.

Перевірити відсутність всіх *object*-файлів та *executable*-файлу.

2.7.3 Скомпілювати С-файли програмних модулів командою *make*.

2.7.4 Виконати команду *make* з метою *install*.

Перевірити роботу *executable*-файла без необхідності вказувати до нього шлях доступу.

2.7.5 Повторно скомпілювати С-файли програмних модулів командою *make*. Надати висновки щодо повідомлення команди.

2.8 Огляд етапів побудови C-програми GNU-компілятором GCC

Припустимо, що у каталозі «*Laboratory-work-11*» є файл «*назва таблиці.c*», створений у розділі 2.2.1. Виконати наступні завдання.

2.8.1 Виконати *preprocessing*-етап для вказаного файлу, зберігши результат у файлі «*назва таблиці.i*»

2.8.2 Виконати *compilation*-етап для файлу «*назва таблиці.i*», зберігши результат у файлі «*назва таблиці.s*»

2.8.3 Повторити *compilation*-етап для файлу «*назва таблиці.i*» з оптимізацією програмного коду, зберігши результат у файлі «*назва таблиці_opt.s*», та визначити відсоток зменшення кількості рядків після оптимізації.

2.8.4 Виконати *assembly*-етап для файлу «*назва таблиці.i*», зберігши результат у файлі «*назва таблиці.o*»

2.8.5 Визначити командний рядок виконання *linking*-етапу для файлу «*назва таблиці.o*» та зберегти результат у файл-скрипті *ld.sh*

2.8.6 Виконати *linking*-етап через виконання створеного раніше файл-скрипту *ld.sh*.

2.8.7 Переглянути список файлів динамічних бібліотек, пов'язаних зі створеним *executable*-файлом.

2.9 Підготовка процесу *Code Review* для надання рішень завдань лабораторної роботи на перевірку викладачем

Примітка: Рішення на завдання 2.1, 2.2, 2.3 можуть бути надані під час *Online*-заняття для отримання відповідних балів. За межами *Online*-заняття виконуються всі рішення.

На веб-сервісі *GitHub* зафіксувати зміни у файлі *README.md*.

Скопіювати файли, які було створено у попередніх розділах завдань, в каталог «*Laboratory-work-11*» *Git*-репозиторію.

Оновити *Git*-репозиторій змінами нової гілки «*Laboratory-work-11*» з *GitHub*-репозиторію. Оновити *GitHub*-репозиторій змінами нової гілки «*Laboratory-work-11*» *Git*-репозиторію. Виконати запит *Pull Request*.

Примітка: Увага! Не натискайте кнопку «Merge pull request»!

Це повинен зробити лише рецензент, який є вашим викладачем!

Може бути виконано два запити *Pull Request*: під час *Online*-заняття та за межами *Online*-заняття. Якщо запит *Pull Request* було зроблено під час *Online*-заняття, тоді рецензент-викладач перегляне рішення, надасть оцінку та закриє *Pull Request* без операції *Merge*. Коли буде зроблено остаточний запит *Pull Request*, тоді рецензент-викладач

перегляне ваше рішення та виконає злиття нової гілки та основної гілки через операцію *Merge*. Якщо рецензент знайде помилки, він повідомить про це у коментарях, які з'являться на сторінці *Pull request*.

2.9 Оцінка результатів виконання завдань лабораторної роботи

Оцінка	Умови
+2 бали	під час <i>Online</i> -заняття виконано правильні рішення завдань 2.1, 2.2 або на наступному <i>Online</i> -занятті чи на консультації отримано правильну відповідь на два запитання, які стосуються рішень
+2 бали	1) всі рішення роботи відповідають завданням 2) <i>Pull Request</i> представлено не пізніше найближчої консультації після офіційного заняття із захисту лабораторної роботи
-0.5 балів за кожну омилку	в рішенні є помилка, про яку вказано в <i>Code Review</i>
-1 бал	<i>Pull Request</i> представлено пізніше часу завершення найближчої консультації після офіційного заняття із захисту лабораторної роботи за кожний тиждень запізнення
+2 бали	на наступному <i>Online</i> -занятті або на консультації отримано правильну відповідь на два запитання, які стосуються рішень

Література

1. Блажко О.А. Відео-запис лекції «Етапи компіляції С-програм та автоматизація побудови С-програм». URL: <https://youtu.be/1OxZtfO2r-8>