Тема

Ознайомлення із основами створення програм з використанням мови C++. Опції компіляції. Типи збірок. Виконавчі файли, статичні та динамічні бібліотеки

Мета

Засвоїти принципи створення програм написаних мовою C++, навчитись перетворювати текст програми на виконавчий файл за допомогою командного рядка. Навчитись задавати опції компіляції, створювати різні типи збірок - виконавчі файли, статичні та динамічні бібліотеки

Теоретичні відомості

Структура програми

Програма чи бібліотека складається із однієї чи багатьох *одиниць трансляції* (translation unit). Зазвичай, одиницею трансляції є файл із розширенням .c для коду C та із розширенням .cpp чи .cxx для коду C++.

Кожна одиниця трансляції складається із функцій та змінних. Вони можуть бути як внутрішні (для використання лише в межах однієї одиниці трансляції), так і зовнішні (для зв'язків між одиницями та доступу ззовні).

Для зручності використання функцій та змінних між різними одиницями трансляції існують *файли-заголовки* (header files) із розширенням .h чи .hpp. Їхній вміст підставляється препроцесором у місце директиви #include.

Файл-заголовок може відповідати якійсь одиниці трансляції, або існувати сам по собі та зазвичай містить оголошення типів, змінних та функцій, які визначені в якійсь одиниці трансляції.

Нижче подано приклад структури C++ програми, що складається із двох файлів-заголовків та трьох одиниць трансляції.

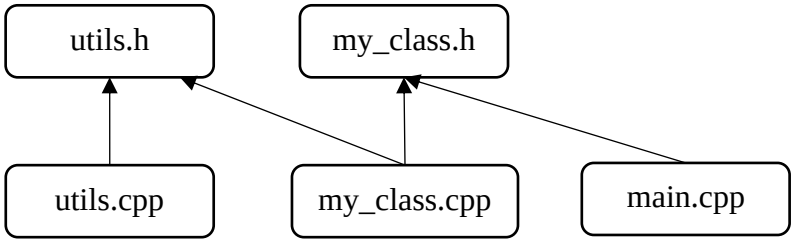


Рис. 1. Приклад структури програми. Стрілками позначено включення файлів-заголовків за допомогою директиви #include.

Перетворення тексту програми у фінальний файл (виконавчий чи бібліотеку) проходить у кілька етапів:

- компіляція окремих одиниць трансляції в об'єктні файли (.o/.obj);
- створення із об'єктних файлів статичних бібліотек (.a/.lib);
- компонування об'єктних фалів та статичних бібліотек у виконавчі файли чи динамічні бібліотеки (.exe/.so/.dll).

Нижче подано загальну схему процесу збірки C++ програми:

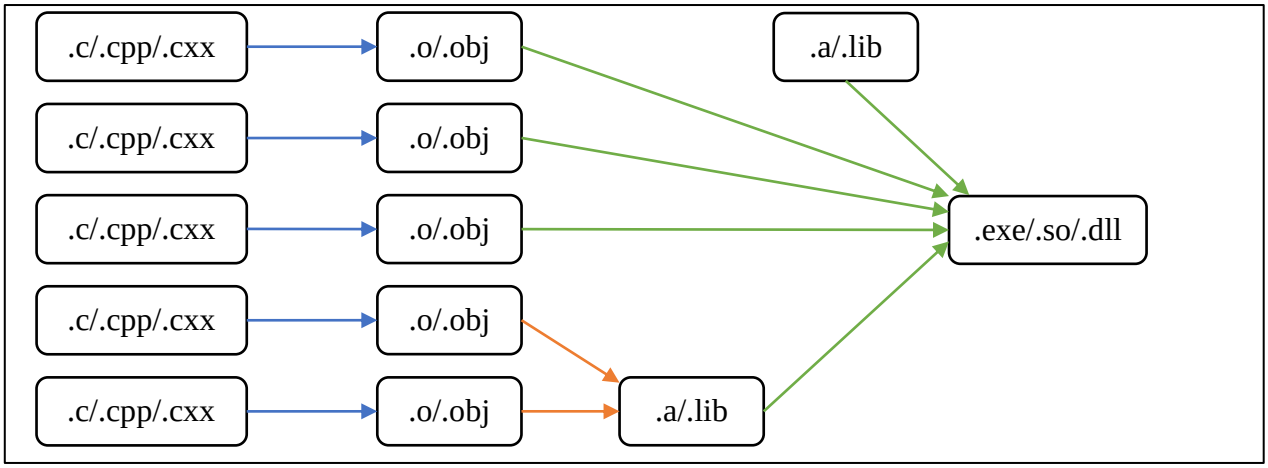


Рис. 2. Схема збірки C++ програми. Сині стрілки – компіляція, оранжеві – створення статичних бібліотек, зелені – компонування.

У наступних розділах детальніше зупинимось на кожному із цих етапів

Структура С файлу

Для початку варто розуміти відмінність між *оголошенням* (declaration) та *визначенням* (definition).

Визначення пов'язує ім'я із реалізацією цього імені – даними або кодом:

- визначення змінної вказує компілятору виділити певне місце під цю змінну та, можливо, заповнити його певним значенням;
- визначення функції вказує компілятору згенерувати код цієї функції.

Оголошення повідомляє компілятору, що визначення із таким ім'ям та типом існує деінде у програмі, ймовірно, в іншому С файлі.

Для змінних, визначення поділяють на два види:

- *глобальні змінні*, які існують протягом усього життя програми і які доступні з багатьох різних функцій;
- *локальні змінні*, які існують лише протягом виконання певної функції та доступні лише з цієї функції.

Під «доступні» мається на увазі, що можна використовувати ім'я із визначення для звертання до цих змінних.

Також є кілька особливих випадків:

- статичні (static) локальні змінні є насправді глобальними змінними, бо вони існують протягом усього часу виконання програми, незважаючи на те, що вони доступні лише із однієї функції;
- статичні (static) глобальні змінні є також глобальними змінними, не зважаючи, що вони доступні лише для функцій із одного С файлу.

Для функцій, як і для глобальних змінних, ключове слово «static» обмежує доступ до неї по імені лише до функцій із того ж С файлу.

Для глобальних та локальних змінних, ми також розрізняємо, ініціалізована змінна чи ні, тобто чи місце, пов'язане із конкретним ім'ям, попередньо заповнене певним значенням.

Нарешті, дані можуть зберігатись у пам'яті, що виділена динамічно за допомогою malloc чи new. Вона не пов'язана із певним ім'ям, доступ до неї відбувається за допомогою вказівників – іменованих змінних, які містять адресу неіменованої адреси у пам'яті. Ці частини пам'яті звільняються за допомогою free або delete.

Таблиця 1. Типи оголошень та визначень

			Оголошення	**Визначення**
Код			int func(int x);	int func(int x) { ... }
Дані	Глобальні	Ініціалізовані	extern int x;	int x = 1; (в області файлу)
		Неініціалізовані	extern int x;	int x; (в області файлу)
	Локальні	Ініціалізовані	---	int x = 1; (в області функції)
		Неініціалізовані	---	int x; (в області функції)
	Динамічні		---	int * p = (int *)malloc(sizeof(int));

Нижче наведено приклад коду для кращого розуміння різних типів оголошень та визначень.

```
// Визначення неініціалізованої глобальної змінної
int g_uninit_x;

// Визначення ініціалізованої глобальної змінної
int g_init_x = 1;

// Визначення неініціалізованої глобальної змінної, доступної лише в цьому файлі
static int g_uninit_y;

// Визначення ініціалізованої глобальної змінної, доступної лише в цьому С файлі
static int g_init_y = 2;

// Оголошення глобальної змінною, що існує деінде в програмі
extern int g_z;

// Оголошення фукнції, що існує деінде в програмі
//  (можна додати "extern" спереду, але це не обов'язково)
int func_a(int x, int y);

// Визначення фукнції, яка доступна лише в цьому С файлі
static int func_b(int x)
{
        return x + 1;
}

// Визначення фунції
// Параметр функції вважається локальною змінною
int fn_c(int local_x)
{
        // Визначення неініціалізованої локальної змінної
        int local_uninit_y;
        // Визначення ініціалізованої локальної змінної
        int local_init_y = 3;

        // Код, що звертається до локальних та глобальних змінних та інших функцій по імені
        g_uninit_x = func_a(local_x, g_init_x);
        local_uninit_y = func_a(local_x, local_init_y);
        local_uninit_y += func_b(g_z);
        return (g_uninit_y + local_uninit_y + g_init_y);
}
```

Рис 3. sample.c

Компіляція

Завдання компілятора (compiler) – перетворити текст С файлу в *об'єктний файл*. На UNIX платформах такі файли мають розширення .o, на Windows - .obj.

Вміст об'єктного файлу – це здебільшого два типи сутностей:

- код, що відповідає визначенням функцій у С файлі;

- дані, що відповідають визначенням глобальних змінних у С файлі (для ініціалізованих глобальних змінних також містить їхні значення).

Екземпляри вище наведених сутностей мають пов’язані з ними імена – імена змінних чи функцій, визначення яких було у С файлі.

Код в об’єктному файлі – це послідовність машинних інструкцій, що відповідають тілу функцій, написаних у С файлі. Ці інструкції маніпулюють змінними та можуть звертатись до інших ділянок коду (інших функцій програми).

Звертання до змінних чи функцій можливе лише, якщо вони були перед тим оголошені чи визначені. Оголошення – це своєрідна обіцянка, що визначення десь існує у програмі (можливо, в іншому файлі).

Компілятор, генеруючи об’єктний файл, залишає порожні значення для тих імен, які були оголошені, але не визначені.

Враховуючи вище написане, можна зобразити об’єктний файл, що відповідає коду з Рис.3.

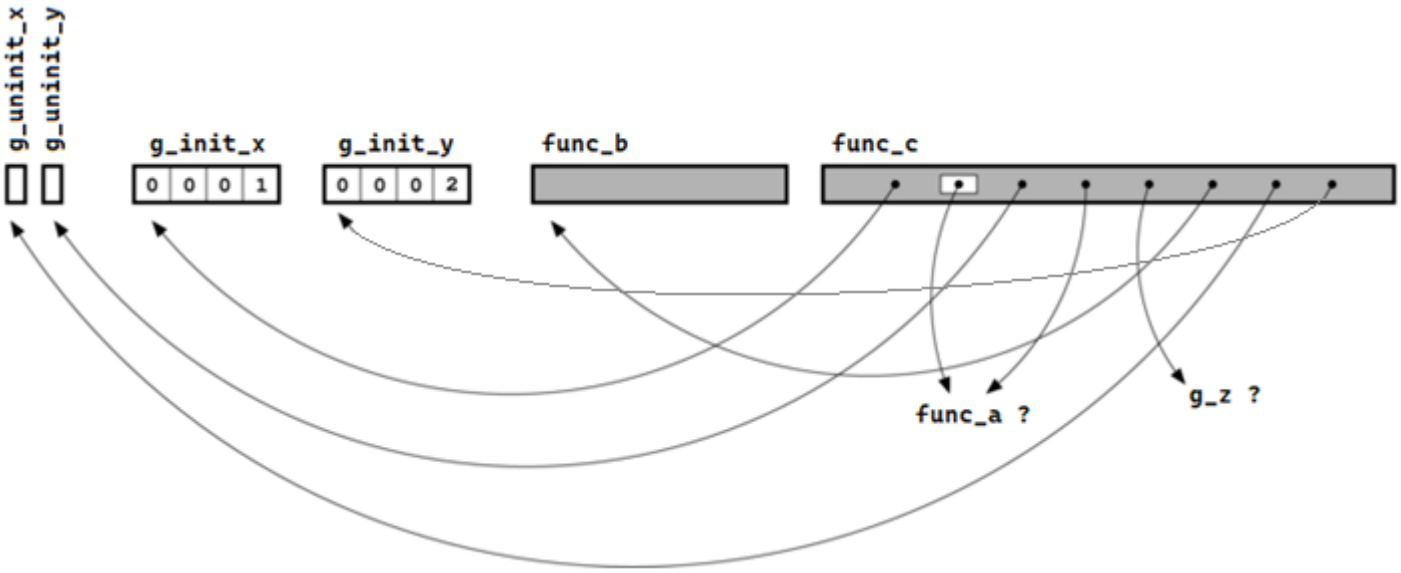


Рис. 4. Структура об’єктного файлу для sample.c

На практиці структуру об’єктного файлу можна дослідити використовуючи утиліту nm, яка повертає інформацію про символи (глобальні змінні та функції) об’єктного файлу на UNIX платформах. На Windows утиліта DUMPBIN із опцією /SYMBOLS друкує подібну інформацію.

Нижче подано вивід утиліти nm із опцією ‘-f sysv’ для sample.o на системі CentOS 7.9.

```
Symbols from sample.o:

Name                  Value           Class        Type         Size             Line  Section

fn_c                |000000000000000f|   T  |            FUNC|0000000000000059|      |.text
func_a              |                |   U  |          NOTYPE|                |      |*UND*
func_b              |0000000000000000|   t  |            FUNC|000000000000000f|      |.text
g_init_x            |0000000000000000|   D  |          OBJECT|0000000000000004|      |.data
g_init_y            |0000000000000004|   d  |          OBJECT|0000000000000004|      |.data
g_uninit_x          |0000000000000000|   B  |          OBJECT|0000000000000004|      |.bss
g_uninit_y          |0000000000000004|   b  |          OBJECT|0000000000000004|      |.bss
g_z                 |                |   U  |          NOTYPE|                |      |*UND*
```

Рис. 5. Вивід програми nm

Значення поля Class може мати такі значення:

- U – невизначені посилання. Це і є ті порожні значення, які генерує компілятор для оголошених, але не визначених змінних та функцій. В нашому прикладі їх два: func_a та g_z. На деяких системах nm також друкує колонку Section із значенням *UND* або UNDEF в такому випадку.
- t – код, локальний для цього файлу. Тобто функція була оголошена із ключовим словом static. Секція для цього класу - .text.
- T – код, до якого можна звертатись із інших файлів. Функція була оголошена без ключового слова static або із ключовим словом extern. Секція для цього класу - .text.
- d – ініціалізована глобальна змінна, локальна для цього файлу. Вона була оголошена із ключовим словом static. Секція для цього класу - .data.
- D - ініціалізована глобальна змінна, до якої можна звертатись із інших файлів. Вона була оголошена без ключового слова static або із ключовим словом extern. Секція для цього класу - .data.
- b – неініціалізована глобальна змінна (або ініціалізована нулем), локальна для цього файлу. Вона була оголошена із ключовим словом static. Секція для цього класу - .bss.
- B або C – неініціалізована глобальна змінна (або ініціалізована нулем), до якої можна звертатись із інших файлів. Вона була оголошена без ключового слова static або із ключовим словом extern. Секція для цього класу - .bss або *COM*.

В об’єктних файлах можуть бути також присутні інші символи, відповідника яких нема у С файлі. Зазвичай, вони потрібні для внутрішніх механізмів компілятора та компонувальника, щоб зібрати програму.

Компонування (лінкування)

Позаяк оголошення функції чи змінної є для компілятора лише обіцянкою, що десь у програмі існує визначення для цієї функції чи змінної, задача компонувальника полягає в тому, щоб виконати цю обіцянку, тобто знайти відповідні визначення та розв’язати залежності.

Для ілюстрації розглянемо ‘main.c’ файл супутній до ‘sample.c’.

```
// ініціалізована глобальна змінна
int g_z = 11;

// друга змінна з назвою g_init_y, але вони обидві статичні
static int g_init_y = 2;

// ологошення іншої глобальної змінної
extern int g_init_x;

int func_a(int x, int y)
{
        return(x + y);
}

int main(int argc, char* argv[])
{
        return func_a(11, 12);
}
```

Рис. 6. main.c

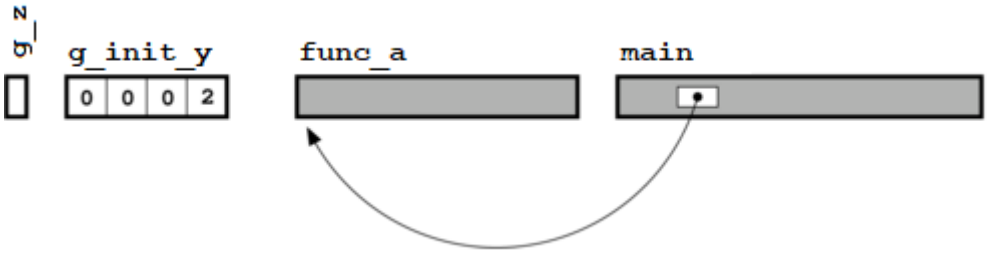


Рис. 7. Структура об’єктоного файлу для main.c

Маючи ці дві діаграми, очевидно, що усі зв’язки можуть бути встановлені (якби це було неможливо, то компонувальник би закінчив роботу неуспішно та і надрукував повідомлення про помилку). Кожен символ має своє місце, кожне місце пов’язане із якимось символом, компонувальник заповнив всі «порожні» місця.

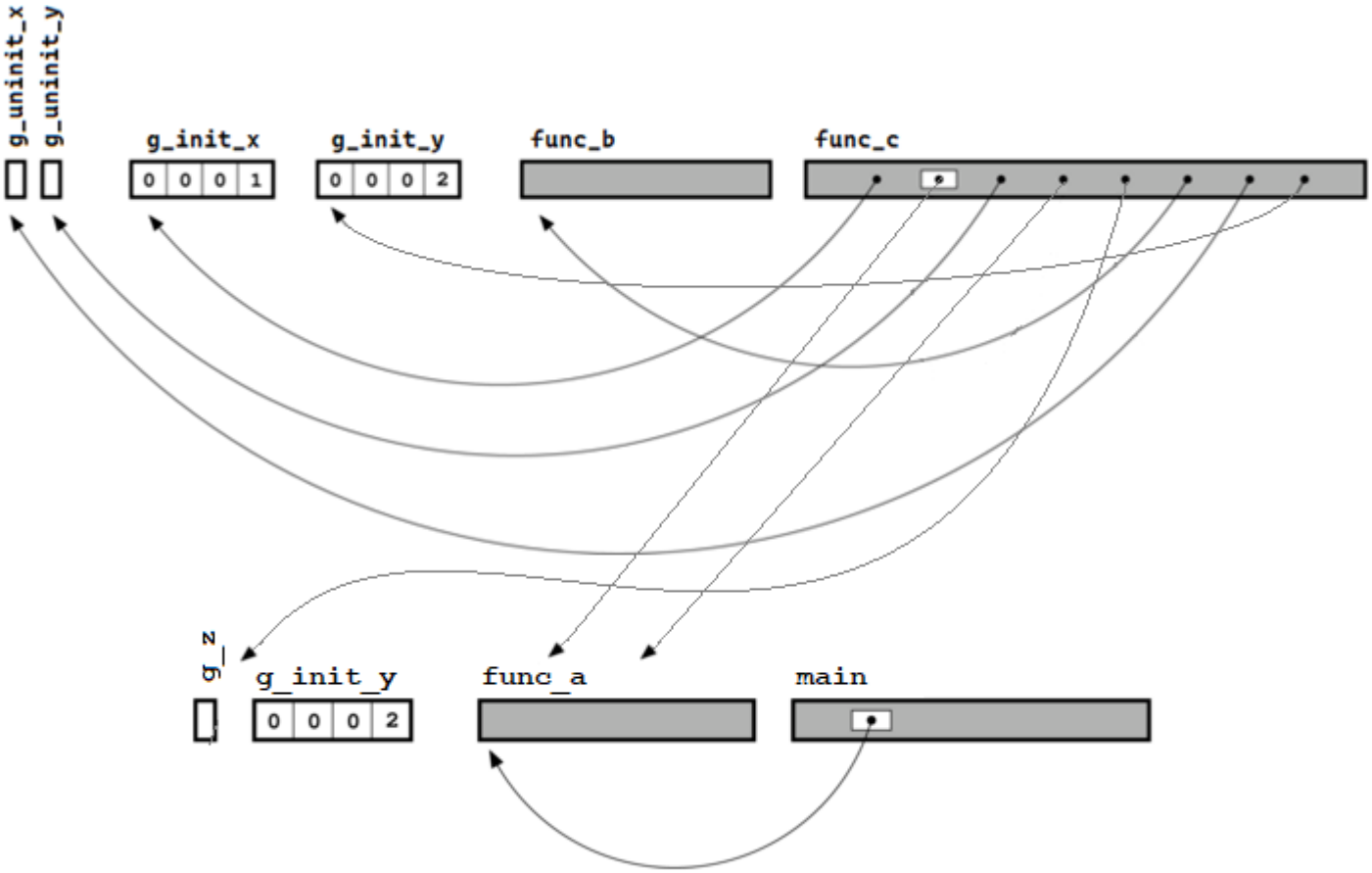


Рис. 8. Структура програми, що створена компонуванням sample.c та main.c

Статичні бібліотеки

Найпростішим типом бібліотек є статична бібліотека. Принцип статичної бібліотеки полягає у перевикористанні готових об’єктних файлів. Статичну бібліотеку можна уявити як набір об’єктних файлів об’єднаних в один файл, аналогічно з архівацією.

На UNIX системах команда для створення статичної бібліотеки – ar, вона генерує файл із розширенням .a. Файли бібліотек зазвичай мають префікс lib та передаються компонувальнику за допомогою прапорця -l, за яким йде назва бібліотеки без префіксу lib (наприклад -lmyprogram підключить ‘libmyprogram.a’).

На Windows статичні бібліотеки мають розширення .lib та створюються LIB утилітою.

Коли компонувальник обробляє список об’єктних файлів, він створює список символів, для яких ще немає визначення. Коли він закінчив обробляти всі явно передані об’єктні файли, то він починає

шукати визначення символів (які ще не були знайдені в об’єктних файлах) у бібліотеках. Якщо він знаходить визначення символу серед якогось об’єктного файлу всередині, то цей об’єктний файл додається до переліку об’єктних файлів програми і процес компонування продовжується.

Гранулярність додавання символів із статичної бібліотеки – об’єктний файл. Тобто якщо потрібно хоча б один символ, то до програми додається цілий об’єктний файл. Це може привезти до необхідності пошуку визначення для інших символів, на які є посилання у новододаному об’єктному файлі.

Динамічні бібліотеки

Одним з недоліків статичних бібліотек є те, що кожна програма місить копію коду. Це займає зайве місце на диску та у пам’яті під час виконання.

Іншим недоліком є те, що код бібліотеки є закріпленим всередині програми. Якщо виникає потреба змінити код бібліотеки, щоб, приміром, виправити якусь помилку, то доведеться перезібрати як бібліотеку, так і кожну програму, яка з нею скомпонована.

Щоб вирішити ці та інші проблеми було введено динамічні бібліотеки (Dynamic library, Shared library). Файли динамічних бібліотек мають розширення .so (Linux), .dll (Windows) чи .dylib (Mac). Під час компонування програми із такими бібліотеками, символи позначаються такими, які потребують компонування на етапі виконання.

Якщо компонувальник знаходить визначення символу у динамічній бібліотеці, то він не включає його до складу фінального виконавчого файлу. Натомість, компонувальник записує назву символу та в якій бібліотеці він присутній.

Ці залишки роботи компонувальника відкладаються на час початку виконання програми. Перед тим, як виконання перейде до функції main, менша версія компонувальника (ld.so) виконує пошук та завантаження динамічних бібліотек та знаходження потрібних у них символів.

Це значить, що жодна програма не містить у собі копію бібліотечного коду. Нові зміни до бібліотеки тепер доступні у програмі автоматично – шляхом оновлення файлу бібліотеки та перезапуску програми.

Ще однією відмінністю динамічних бібліотек є гранулярність – якщо хоча б один символ потребує завантаження динамічної бібліотеки, то вона завантажується у пам’ять програми цілком, а не окремими об’єктними файлами, як у статичній.

Для UNIX систем, всі символи зі всіх об’єктних файлів динамічної бібліотеки є автоматично доступні для користувачів. На Windows програміст має явно позначити такі символи доступними – експортувати їх. Це можна зробити кількома способами:

- в коді оголосити символ як __declspec(dllexport)
- через опцію /export компонувальника link.exe
- через опцію /def компонувальника link.exe та спеціально створений файл експорту.

Особливості C++

C++ надає деякі додаткові можливості поруч із C, тому має деякі відмінності у роботі компілятора та компонувальника.

Перш за все, у зв’язку із підтримкою перевантаження функцій (function overloading), просторів імен (namespaces), класів та іншого виникає потреба розрізняти функції із однаковим ім’ям.

Цю задачу розв’язує так зване декорування імен (name mangling). Вся інформація про сигнатуру функції та її розміщення в просторах імен, класах кодується в текстову форму і символ з таким іменем бачить компонувальник.

Наприклад, розглянемо код трьох функцій max, які відрізняються типами параметрів.

```
int max(int x, int y)
{
        if (x > y) return x;
        else return y;
}

float max(float x, float y)
{
        if (x > y) return x;
        else return y;
}

double max(double x, double y)
{
        if (x > y) return x;
        else return y;
}
```

Рис. 9. Код файлу mangling.cpp

Нижче подано вивід утиліти nm із опцією ‘-f sysv’ для mangling.o на системі CentOS 7.9.

```
Symbols from mangling.o:

Name                  Value           Class        Type         Size             Line  Section

_Z3maxdd             |0000000000000043|   T  |              FUNC|0000000000000032|     |.text
_Z3maxff             |000000000000001c|   T  |              FUNC|0000000000000027|     |.text
_Z3maxii             |0000000000000000|   T  |              FUNC|000000000000001c|     |.text
```

Рис. 10. Вивід утиліти nm для файлу mangling.o

Як видно, всі три функції max насправді мають різні імена в об’єктному файлі. Можна здогадатись, що символи d, f та i після назви max позначають типи параметрів – double, float та int відповідно. Формат стає набагато складніший з класами, просторами імен, шаблонами та перевизначених операторах.

Зазвичай існує спосіб перетворити декоровані імена, з якими працює компонувальник, у дружні для людини. Це може бути окрема утиліта, як от c++filt, чи опція, на зразок --demangle для nm.

Нижче наведено вивід утиліти nm із опцією –demangle для файлу mangling.o:

```
Symbols from mangling.o:

Name                  Value           Class        Type         Size             Line  Section

max(double, double) |0000000000000043|   T  |              FUNC|0000000000000032|     |.text
max(float, float)   |000000000000001c|   T  |              FUNC|0000000000000027|     |.text
max(int, int)       |0000000000000000|   T  |              FUNC|000000000000001c|     |.text
```

Рис. 11. Вивід утиліти nm для файлу mangling.o із опцією --demangle

Ця особливість декорування імен має значення під час змішування коду C та C++. Всі символи створені компілятором C++ - мають змінені імена. Всі символи створені компілятором C мають імена такі як у тексті програм. Відповідно, виникає проблема у компонуванні об’єктних файлів, якщо деякий код C має викликати функцію C++, або якщо деякий код C++ має викликати функцію C. Для розв’язування цієї проблеми у C++ існує ключове слово extern "C". Воно вказує компілятору C++, що назву цієї функції не треба декорувати, вона має бути в об’єктному файлі такою, як в тексті програми.

Іншою особливістю C++ є необхідність ініціалізації глобальних об’єктів, що мають нетривіальні конструктори. Для цього компілятор включає додаткову інформацію до об’єктних файлів – як от

список конструкторів/деструкторів, які потрібно викликати перед початком виконання функції main або після неї (для деструкторів).

Ще одна особливість пов'язана із підтримкою шаблонів у C++. Особливість роботи механізмів шаблонів вимагає забезпечити компонування об'єктних файлів, які містять однакові символи, утворені в результаті інстанціювання шаблонів. Це зазвичай реалізовано або за допомогою позначення таких символів класом W (Weak), що дозволяє компонувальнику ігнорувати дублікати, або за допомогою генерації коду для шаблонів під час роботи компонувальника.

Завдання

Виконайте завдання подані у частинах 1-3 нижче.

Оформіть звіт, що включає:

- Тему, мету, теоретичні відомості
- Завдання
- Результати виконання завдань (по скріншоту вікна терміналу на кожну частину, діаграму, таблицю розмірів збірки для випуску та збірки для відлагодження)
- Висновки

Результати виконання завдань мають містити знімки екрану, на яких видно задані команди рядка та результати їх роботи, таблиці, графіки тощо.

Частина 1. Компіляція. Компонування

1. Завантажити та встановити LLVM (див. Додаток 1)
2. У файловій системі створити директорію 'oop_lab1_prog' та файли 'lab1_main.cpp', 'lab1_utils.h', 'lab1_utils.cpp'. Текст файлів подано у Додатку 2. В усіх файлах необхідно замінити NN у суфіксі '_NN' на номер варіанту, наприклад _01 для першого варіанту, _02 для другого і так далі.
3. Відкрити командний рядок та перейти у директорію 'oop_lab1_prog'. Скомпілювати файл 'lab1_main.cpp' виконавши команду, подану нижче. Створиться об'єктний файл 'lab1_main.o'

```
clang -c lab1_main.cpp -o lab1_main.o
```

4. Проглянути вміст об'єктного файлу 'lab1_main.o' виконавши команду, подану нижче. Пояснити значення колонки Class для кожного символу. Рядок для символу '@feat.00' можна пропустити.

```
llvm-nm -f sysv lab1_main.o
```

5. Скомпілювати файл 'lab1_utils.cpp' виконавши команду, подану нижче. Створиться об'єктний файл 'lab1_utils.o'

```
clang -c lab1_utils.cpp -o lab1_utils.o
```

6. Проглянути вміст об’єктного файлу ‘lab1_utils.o’ виконавши команду, подану нижче. Пояснити значення колонки Class для кожного символу. Рядок для символу ‘@feat.00’ та символів, що починаються символом підкреслення, можна пропустити.

```
,
llvm-nm -f sysv lab1_utils.o
```

7. Скомпонувати два об’єктних файли у виконавчий виконавши команду, подану нижче. Створиться файл ‘lab1.exe’

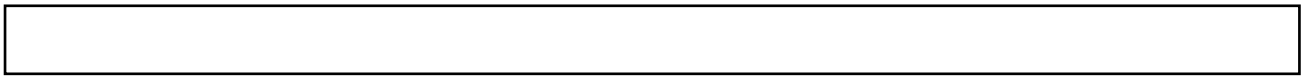
```
clang lab1_main.o lab1_utils.o -o lab1.exe
```

8. Виконайте отриманий lab1.exe передаючи йому різну кількість параметрів і переконайтесь, що програма працює коректно. Для цього перевірте код виконання програми за допомогою інструкції echo %errorlevel%. Змінна %errorlevel% містить код завершення програми, що виконувалась останньою.

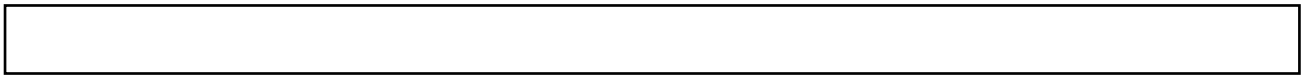
```
lab1.exe
echo %errorlevel%
0

lab1.exe a b c
echo %errorlevel%
3
```

9. Побудувати діаграму структури програми, на якій відобразити об’єктні файли із символами та зв’язки між ними

lab1_main.o



lab2_utils.o



Частина 2. Опції компіляції, типи збірок

Створіть два типи збірки – для випуску (Release) та відлагодження (Debug).

Для створення збірки для випуску виконайте такі кроки:

1. Відкрити командний рядок та перейти у директорію ‘oop_lab1_prog’. Скомпілювати файл ‘lab1_main.cpp’ виконавши команду, подану нижче. Створиться об’єктний файл

‘lab1_main_rel.o’. Опція -O3 задає третій (найвищий) рівень оптимізації. Опція -flto вмикає оптимізацію під час компонування.

```
clang -c lab1_main.cpp -O3 -flto -o lab1_main_rel.o
```

2. Скомпілювати файл ‘lab1_utils.cpp’ виконавши команду, подану нижче. Створиться об’єктний файл ‘lab1_utils_rel.o’.

```
clang -c lab1_utils.cpp -O3 -flto -o lab1_utils_rel.o
```

3. Скомпонувати два об’єктних файли у виконавчий файл, застосувавши команду, подану нижче. Створиться файл ‘lab1_rel.exe’.
   Прапорець -fuse-ld=lld-link потрібен для того, щоб використовувати lld-link компонувальник, який вміє працювати із опцією -flto.

```
clang -fuse-ld=lld-link -flto lab1_main_rel.o lab1_utils_rel.o -o lab1_rel.exe
```

Для створення збірки для відлагодження виконайте такі кроки:

1. Відкрити командний рядок та перейти у директорію ‘oop_lab1_prog’. Скомпілювати файл ‘lab1_main.cpp’ виконавши команду, подану нижче.
   Створиться об’єктний файл ‘lab1_main_dbg.o’. Опція -O0 задає нульовий рівень оптимізації (тобто оптимізація вимкнена). Опція -g вмикає генерацію потрібної для відлагодження інформації.

```
clang -c lab1_main.cpp -O0 -g -o lab1_main_gdb.o
```

2. Скомпілювати файл ‘lab1_utils.cpp’, виконавши команду, подану нижче. Створиться об’єктний файл ‘lab1_utils_dbg.o’.

```
clang -c lab1_utils.cpp -O0 -g -o lab1_utils_dbg.o
```

3. Скомпонувати два об’єктних файли у виконавчий, застосувавши команду, подану нижче. Створиться файл ‘lab1_gdb.exe’.

```
clang -g lab1_main_dbg.o lab1_utils_dbg.o -o lab1_dbg.exe
```

Складіть таблицю із розмірами об’єктних та виконавчих файлів для збірок ‘Release’ та ‘Debug’ як показано нижче:

Таблиця 2. Розміри файлів для збірок ‘Release’ та ‘Debug’

	Розмір для збірки ‘Release’ (байт)	Розмір для збірки ‘Debug’ (байт)
lab1_main.o	*візьміть розмір lab1_main_rel.o*	*візьміть розмір lab1_main_dbg.o*
lab1_utils.o	*візьміть розмір lab1_utils_rel.o*	*візьміть розмір lab1_utils_dbg.o*
lab1.exe	*візьміть розмір lab1_rel.exe*	*візьміть розмір lab1_dbg.exe*

Проаналізуйте та поясніть різницю у розмірах.

Частина 3. Статичні та динамічні бібліотеки.

1. У файловій системі поруч із директорією ‘oop_lab1_prog’ створити директорію ‘oop_lab1_lib’ та файли ‘lab1_lib.h’, ‘lab1_lib.cpp’, ‘lab1_utils.cpp’. Текст файлів подано у Додатку 2.
2. Відкрити командний рядок та перейти у директорію ‘oop_lab1_lib’. Скомпілювати файл ‘lab1_lib.cpp’ виконавши команду, подану нижче. Створиться об’єктний файл ‘lab1_lib.o’.

```
clang -c lab1_lib.cpp -o lab1_lib.o
```

3. Скомпілювати файл ‘lab1_utils.cpp’ виконавши команду, подану нижче. Створиться об’єктний файл ‘lab1_utils.o’.

```
clang -c lab1_utils.cpp -o lab1_utils.o
```

4. Об’єднайте об’єктні файли ‘lab1_lib.o’ та ‘lab1_utils.o’ у статичну бібліотеку виконавши команду, подану нижче. Створиться файл ‘lab1_lib.lib’.

```
llvm-ar rcs lab1_lib.lib lab1_lib.o lab1_utils.o
```

5. Прогляньте вміст статичної бібліотеки та переконайтесь, що в ній містяться файли ‘lab1_lib.o’ та ‘lab1_utils.o’

```
llvm-ar t lab1_lib.lib
```

6. Перекомпілюйте файл ‘lab1_lib.cpp’, вказавши прапорець -DLAB1_LIB_BUILD_AS_SHARED_LIB. Для цього виконайте команду, подану нижче.

```
clang -DLAB1_LIB_BUILD_AS_SHARED_LIB -c lab1_lib.cpp -o lab1_lib.o
```

7. Скомпонуйте об’єктні файли ‘lab1_lib.o’ та ‘lab1_utils.o’ у динамічну бібліотеку виконавши команду, подану нижче. Створиться файл ‘lab1_lib.dll’.

```
clang -shared lab1_lib.o lab1_utils.o -o lab1_lib.dll
```

Контрольні питання

1. З яких частин складається програма написана мовою C чи C++?
2. Що таке одиниця трансляції?
3. Що таке об’єктний файл? Який його вміст?
4. Що таке символ в об’єктному файлі? Які є класи символів? Як код C/C++ впливає на клас символів в об’єктному файлі.
5. Яку функцію виконує компонувальник? В яких випадках від завершує виконання з помилкою?
6. Які особливості компіляції та компонування C++ в порівнянні із C?
7. Що таке декорування імен (name mangling)?
8. Як позбутись декорування імен для певних функцій?

Додаток 1

1. Перейдіть за адресою https://github.com/llvm/llvm-project/releases/ та знайдіть останній (Latest) випуск LLVM

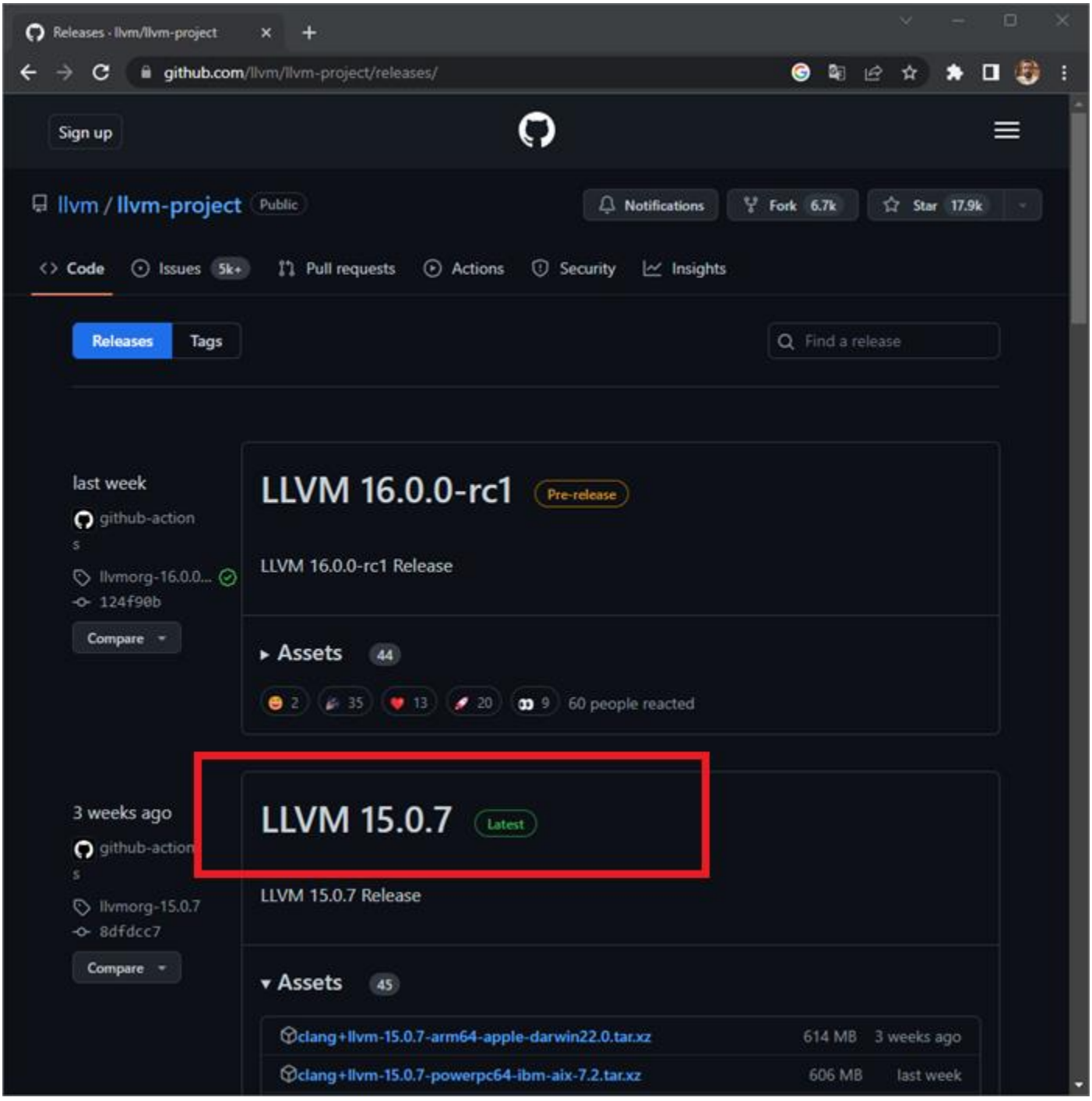


Рис. 12. Сторінка випусків LLVM

2. В переліку Assets натисніть на “Show all NN assets” щоб отримати перелік всіх файлів.

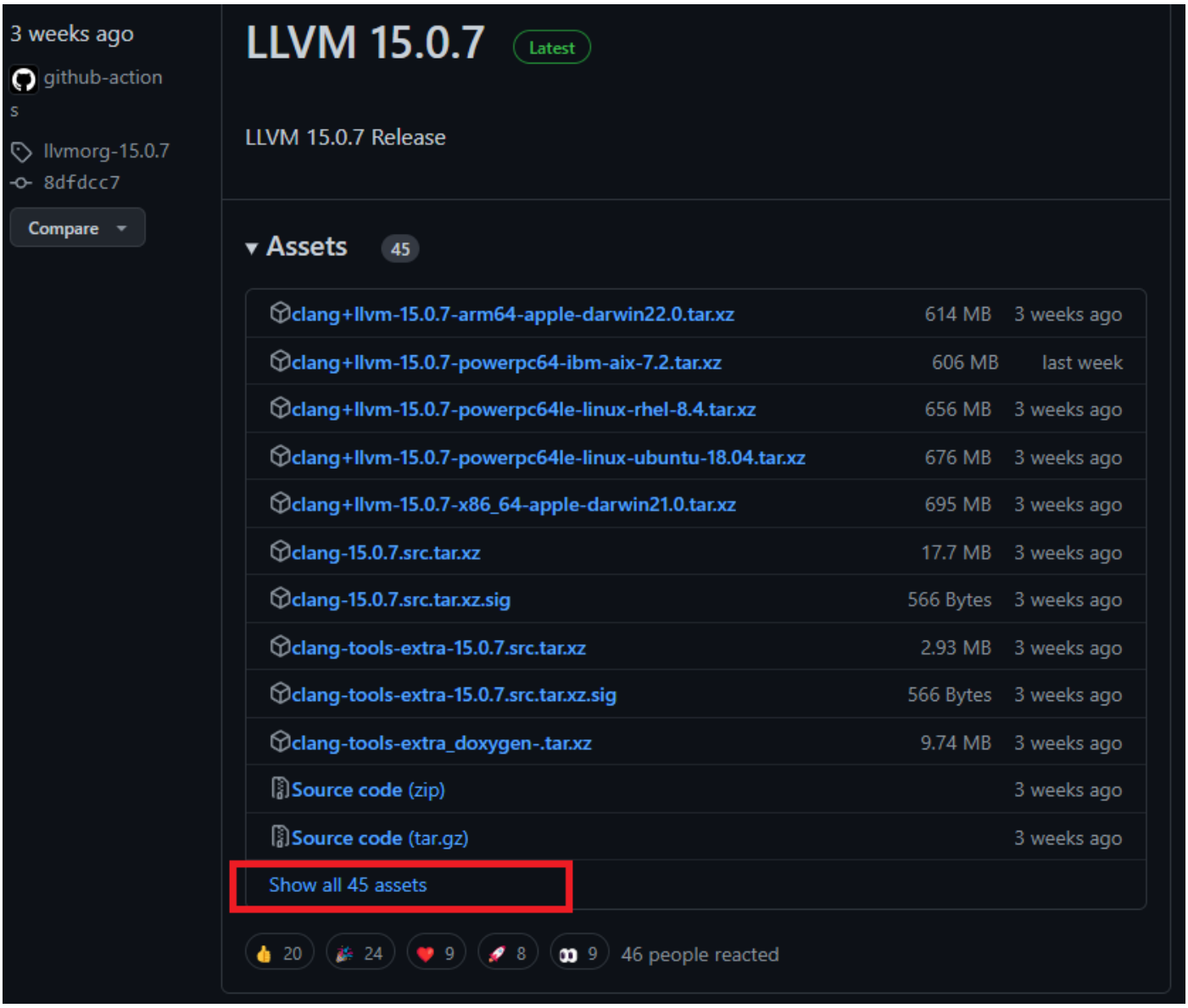


Рис. 13. Опція показу усіх файлів викуску

3. В отриманому переліку знайдіть та натисніть на LLVM-*-win64.exe (якщо у вас 32 бітна система, тоді оберіть LLVM-*-win32.exe).

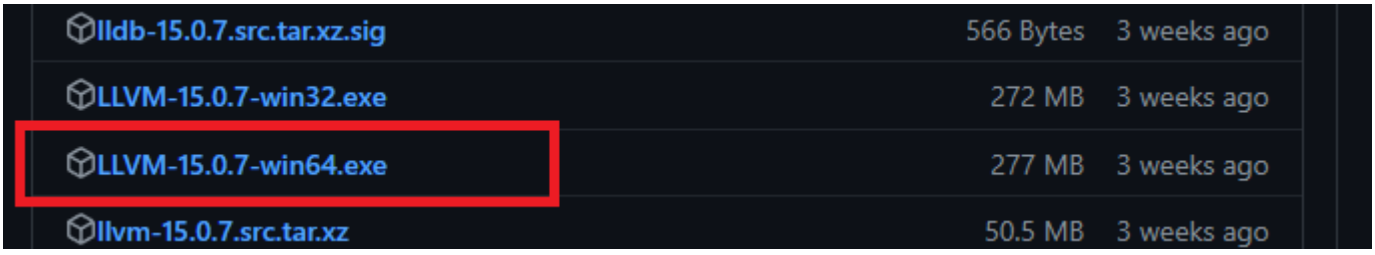


Рис. 14. Вибір встановлювача LLVM для 64-бітної платформи Windows

4. Після того, як файл завантажиться, виберіть його, щоб запустити майстер встановлення.

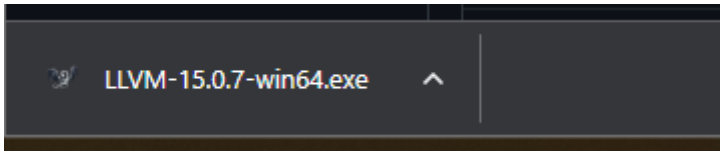


Рис. 15. Завантажений встановлювач LLVM

5. Встановіть LLVM керуючись вказівками майстра та зображеннями із налаштуваннями нижче.

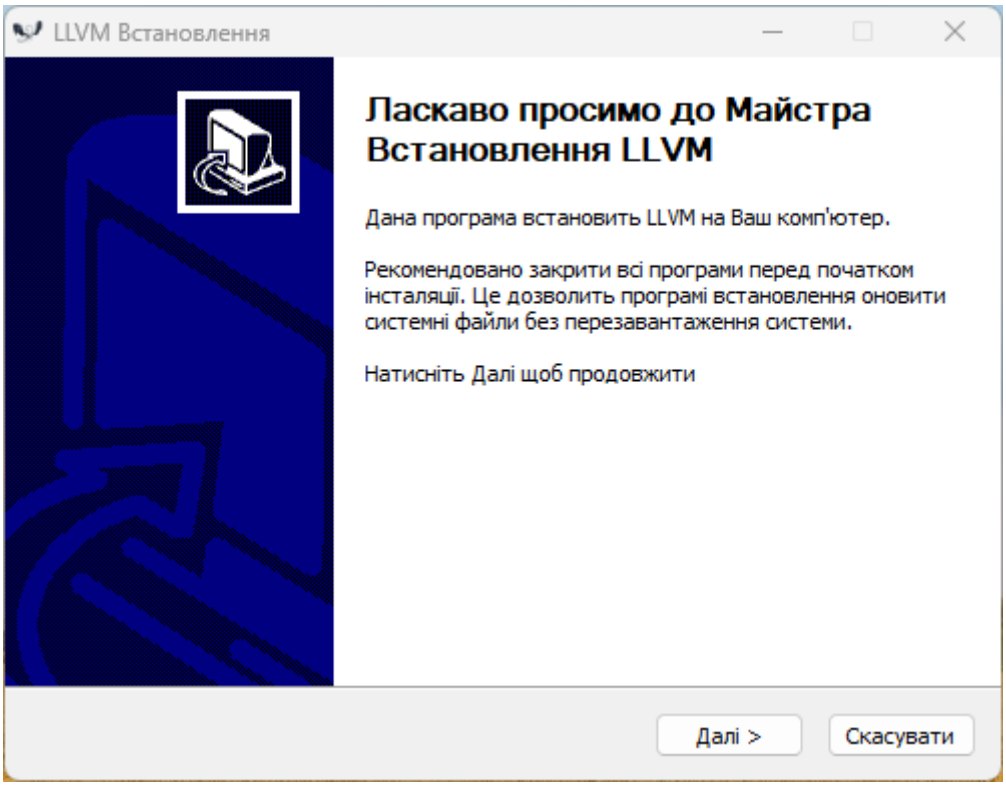


Рис. 16. Початок роботи майстра встановлювача

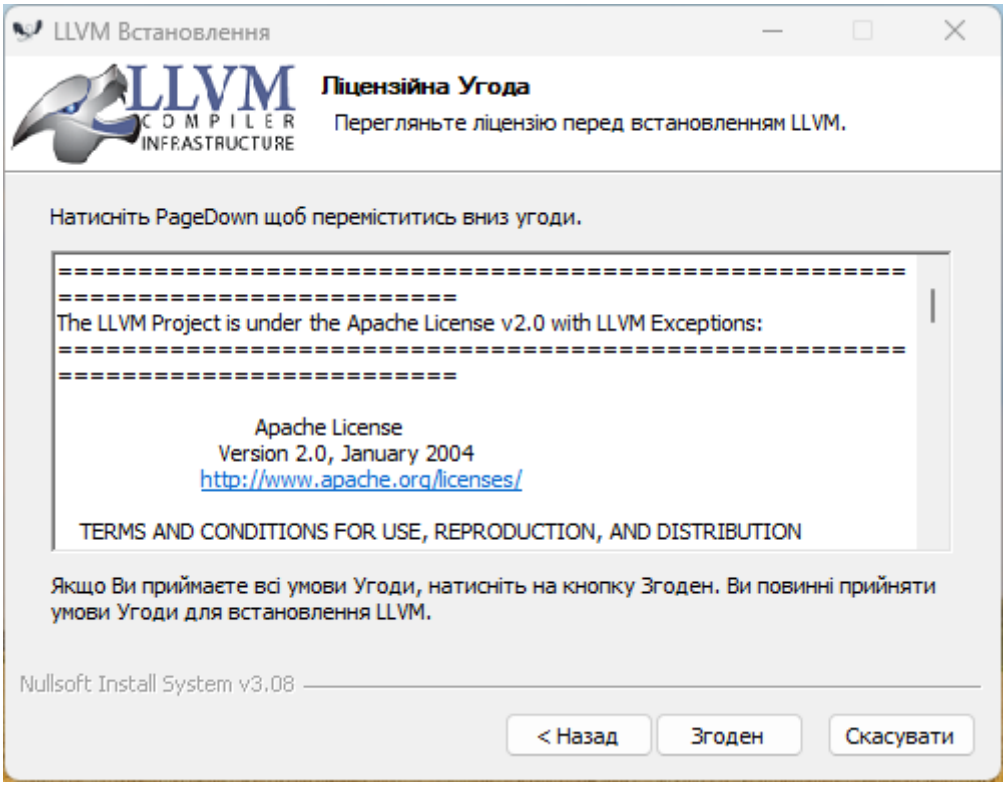


Рис. 17. Ліцензійна угода

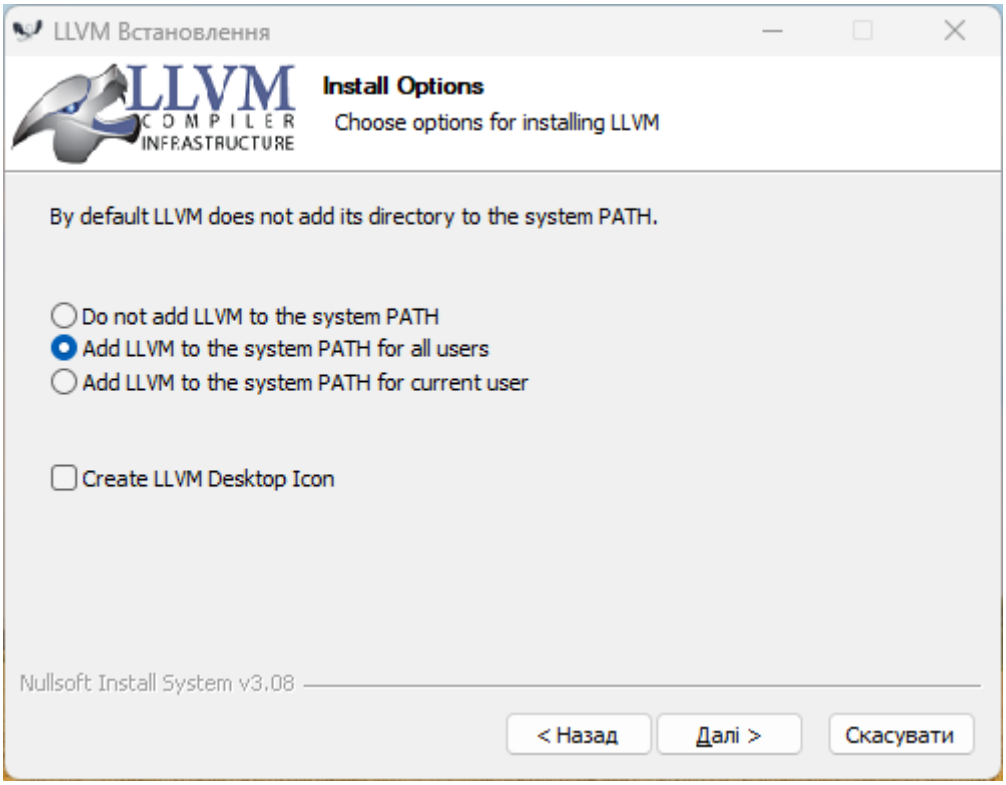


Рис. 18. Налаштування додавання шляху LLVM в системну змінну PATH

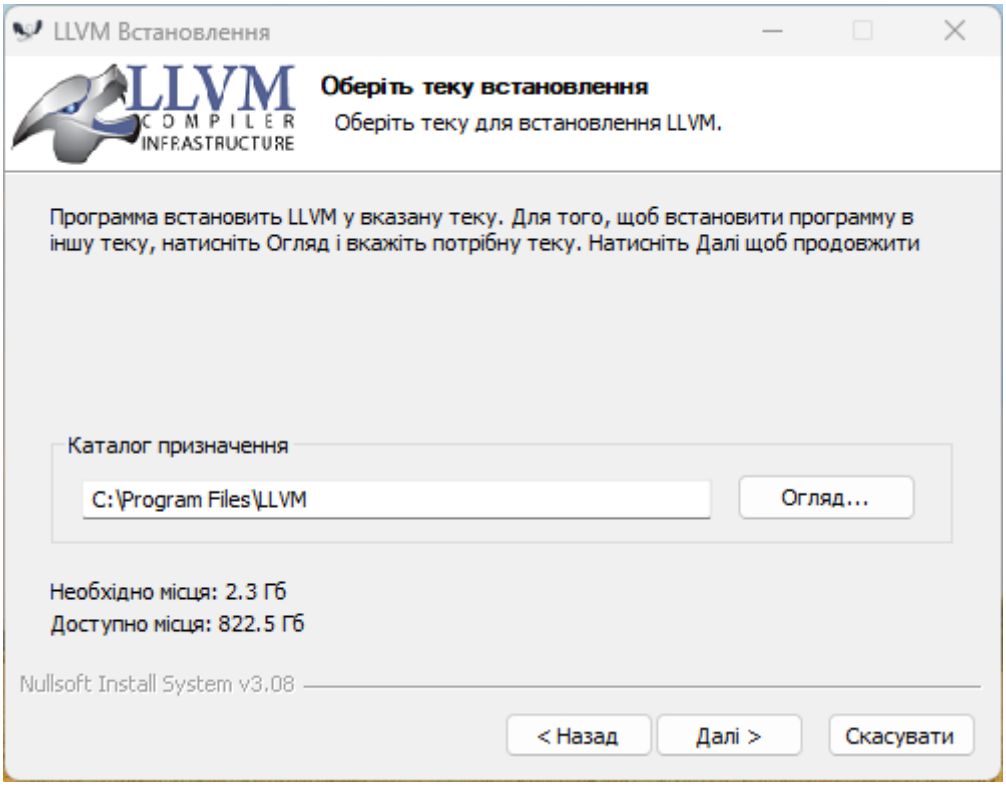


Рис. 19. Налаштування каталогу призначення

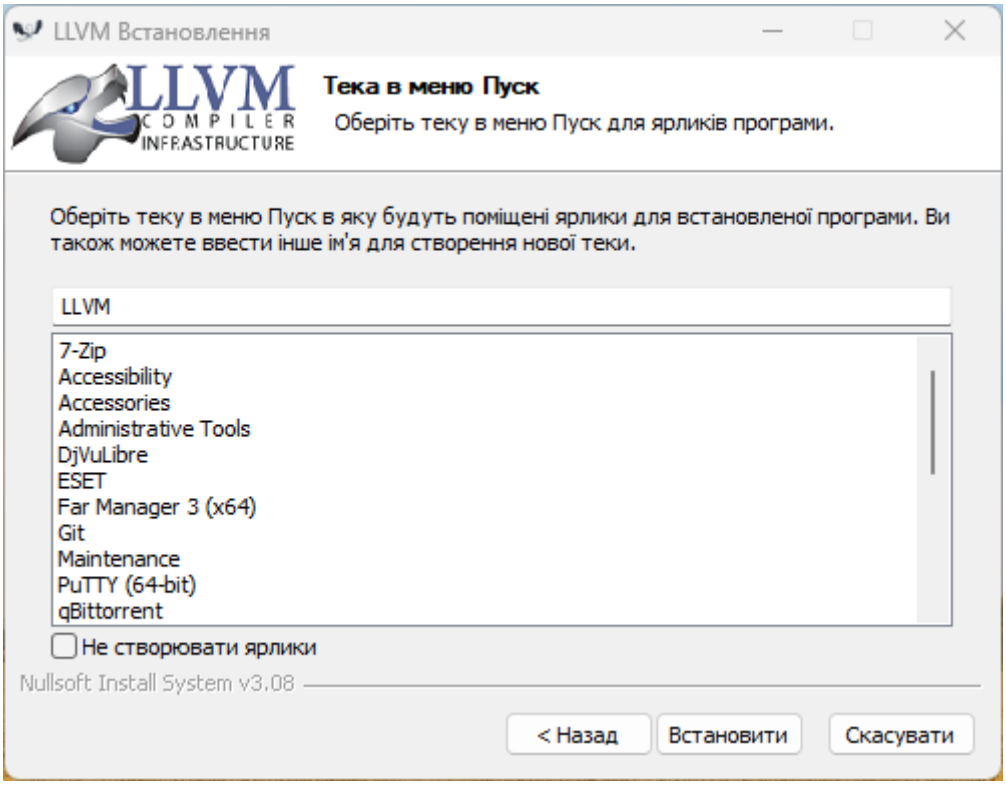


Рис. 20. Налаштування теки в меню Пуск

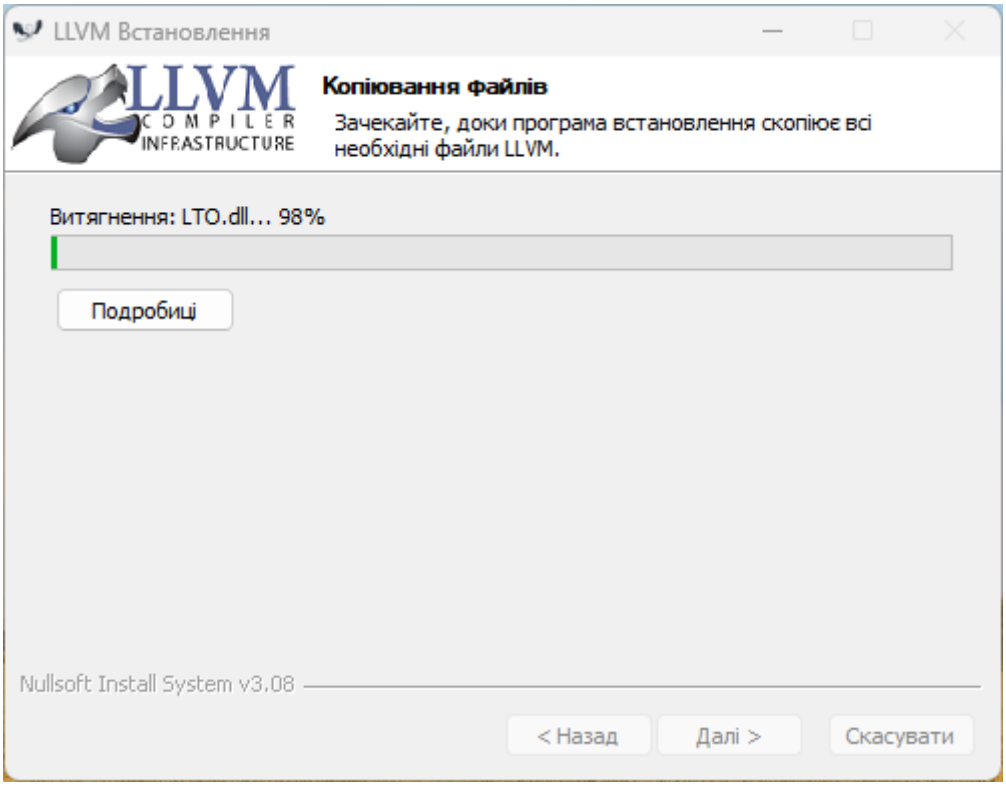


Рис. 21. Процес встановлення LLVM

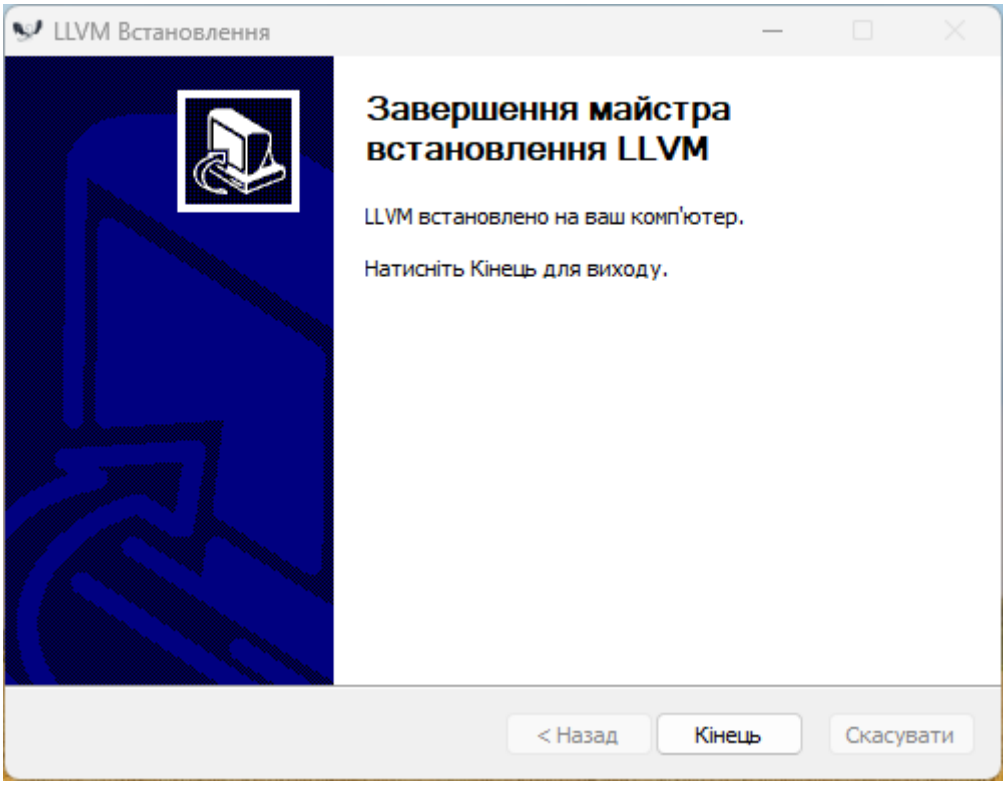


Рис. 22. Завершення майстра встановлення

Додаток 2

Увага! При виконанні лабораторної роботи у всіх файлах необхідно замінити NN у суфіксі ‘_NN’ на номер варіанту, наприклад _01 для першого варіанту, _02 для другого і так далі.

Текст файлу lab1_main.cpp

```
#include "lab1_utils.h"

int main(int argc, char **)
{
	return GetNumberOfArgumentsByArgc_NN(argc);
}
```

Текст файлу lab1_utils.h

```
#ifndef _LAB1_UTILS_H__INCLUDED_
#define _LAB1_UTILS_H__INCLUDED_

extern int g_iNumberOfCalls_NN;

int GetNumberOfArgumentsByArgc_NN(int argc);

double GetTriangleAreaByHeron_NN(double sideA, double sideB, double sideC);

#endif // _LAB1_UTILS_H__INCLUDED_
```

Текст файлу lab1_utils.cpp

```
#include <cmath>
#include "lab1_utils.h"

int g_iNumberOfCalls_NN = 0;

int GetNumberOfArgumentsByArgc_NN(int argc)
{
	++g_iNumberOfCalls_NN;

	return argc - 1;
}

static bool IsTriangleValid_NN(double sideA, double sideB, double sideC)
{
	const bool cbRes =
		(sideA > (sideB + sideC)) && (sideB > (sideA + sideC)) && (sideC > (sideA + sideB));

	return cbRes;
}

double GetTriangleAreaByHeron_NN(double sideA, double sideB, double sideC)
{
	++g_iNumberOfCalls_NN;

	double dRes = NAN;

	if (IsTriangleValid_NN(sideA, sideB, sideC))
	{
		const double cbSemiP = (sideA + sideB + sideC) * 0.5;
		dRes = sqrt(cbSemiP * (cbSemiP - sideA) * (cbSemiP - sideB) * (cbSemiP - sideC));
	}

	return dRes;
}
```

Текст файлу lab1_lib.h

```
#ifndef _LAB1_LIB_H__INCLUDED_
#define _LAB1_LIB_H__INCLUDED_

#ifdef LAB1_LIB_BUILD_AS_SHARED_LIB
#define LAB1_LIB_EXPORT __declspec(dllexport)
#else
#define LAB1_LIB_EXPORT
#endif

// list of exported functions
LAB1_LIB_EXPORT double GetTriangleArea_NN(double sideA, double sideB, double sideC);

#endif // _LAB1_LIB_H__INCLUDED_
```

Текст файлу lab1_lib.cpp

```
#include "lab1_utils.h"
#include "lab1_lib.h"

double GetTriangleArea_NN(double sideA, double sideB, double sideC)
{
	return GetTriangleAreaByHeron_NN(sideA, sideB, sideC);
}
```