Лекція

з дисципліни: «Клієнт-серверне програмування»

на тему: «*Компіляція програми. GCC/Clang компілятори.*»

ПЛАН ЛЕКЦІЇ

1. Трансляція програми.
2. Модульність.
3. GNU Compiler Collection, GCC/Clang.
4. Розбиття програми на різні файли.
5. Компіляція.
6. Бібліотеки.
7. Встановлення та використання бібліотек

1. Трансляція програми.

Трансляція програми – перетворення програми, представленої на одній з мов програмування, в програму на іншій мові. Транслятор зазвичай виконує також діагностику помилок, формує словники ідентифікаторів, видає для друку текст програми і т.д.

Мова, на якій представлена вхідна програма, називається вхідною мовою, а сама програма – вхідним кодом. Вихідна мова називається цільовою мовою, а вихідна (результуюча) програма – об'єктним кодом.

Мета трансляції – перетворення тексту з однієї мови на мову, зрозумілу адресату. При трансляції комп'ютерної програми адресатом може бути:

- пристрій – процесор (трансляція називається компіляцією);
- програма – інтерпретатор (трансляція називається інтерпретацією).

Види трансляції:

- компіляція;
- інтерпретація;
- динамічна компіляція.

Мова процесора називається машинною мовою, машинним кодом. Код на машинній мові виконується процесором.

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.

Переваги компіляції:

- компіляція програми виконується один раз;
- наявність компілятора на пристрої, для якого компілюється програма, не потрібна.

Недоліки компіляції:

- компіляція – повільний процес;
- при внесенні змін до початкового коду, потрібна повторна компіляція.

Асемблер – компілятор, що перетворює текст з мови асемблера на машинну мову. Мова асемблера – мова, близька до машинної мови, мова низького рівня.

Інтерпретація – процес читання і виконання вихідного коду. Реалізується програмою-інтерпретатором.

Інтерпретатор може працювати двома способами:

- читати код і виконувати його відразу (чиста інтерпретація);
- читати код, створювати в пам'яті проміжне представлення коду (байт-код або р-код), виконувати проміжне представлення коду (змішана реалізація).

У першому випадку трансляція не використовується, а в другому – використовується трансляція вихідного коду в проміжний код.

Етапи роботи інтерпретатора:

- лексичний аналіз;
- синтаксичний аналіз;
- семантичний аналіз;
- створення проміжного представлення коду (при чистій інтерпретації не виконується);
- виконання.

Інтерпретатор моделює віртуальну машину, реалізує цикл вибірки-виконання команд машини. Інтерпретатор можна назвати виконавцем мови віртуальної машини.

Переваги інтерпретаторів в порівнянні з компіляторами:

- можливість роботи в інтерактивному режимі;
- відсутність необхідності перекомпіляції вихідного коду після внесення змін і при перенесенні коду на іншу платформу.

Недоліки інтерпретаторів в порівнянні з компіляторами:

- низька продуктивність;
- необхідність наявності інтерпретатора на пристрої, на якому планується інтерпретація програми;
- виявлення помилок синтаксису на етапі виконання (актуально для чистих інтерпретаторів).

Динамічна або JIT компіляція – трансляція, при якій вихідний або проміжний код перетворюється (компілюється) в машинний код безпосередньо під час виконання, «на льоту». Компіляція кожної ділянки

коду виконується тільки один раз; скомпільований код зберігається в кеші і при необхідності використовується повторно.

Переваги динамічної компіляції в порівнянні з компіляцією:

- швидкість роботи динамічно компільованих програм близька до швидкості роботи компільованих програм;
- відсутність необхідності перекомпіляції програми при перенесенні на іншу платформу.

Недоліки динамічної компіляції в порівнянні з компіляцією і чистою інтерпретацією:

- велика складність реалізації;
- великі вимоги до ресурсів.

Динамічна компіляція добре підходить для веб-додатків.

Динамічна компіляція з'явилася і підтримується в тій чи іншій мірі в реалізаціях Java, .NET Framework, Perl, Python.

2. Модульність.

Для створення програмних доданків, які вирішують реальні практичні задачі, часто використовується великий обсяг тексту а сам програмний доданок складається з сотень різних функцій. Досвід показує, що записувати всі функції програми в один файл незручно, оскільки зі збільшенням програми в такому файлі стає важко орієнтуватися. Крім того, застосування командної роботи передбачає написання окремих елементів програми різними програмістами. Тому грамотна технологія програмування полягає в тому, щоб розбивати велику програму на частини (так звані модулі), кожен з яких складається з порівняно невеликої кількості функцій.

В першому наближенні модуль являє собою окремий файл, в якому містяться визначення, описи структур та класів, тіла декількох функцій. На даний час існує багато доступних рішень у вигляді готових шаблонів (патернів) і бібліотек, які можуть використовуватись при реалізації певних задач.

Крім того, до складу одного програмного продукту цілком можуть входити модулі, написані різними мовами програмування. Мови програмування бувають спеціалізованими: кожною мовою найзручніше вирішуються задачі своєї певної області. Якщо ж треба вирішити комплексну задачу, яка розпадається на частини, то найкраще кожну підзадачу реалізувати окремим модулем, написаним найзручнішою для цієї підзадачі мовою.

Мо́дульне програмува́ння — парадигма програмування, орієнтована на зменшення складності програмних продуктів та можливості перенесення окремих рішень з одних програмних проектів у інші. Побудована як наслідок модульна архітектура підкреслює поділ функціональності програми на незалежні змінні модулі, таких, що кожен з них містить все необхідне, щоб

виконати тільки один аспект необхідної функціональності. Модулі, як правило, включені в програму через інтерфейси.

Парадигма базується на механізмах, що дозволяють будувати програму з окремих модулів шляхом забезпечення:

- реалізації окремих елементів програми (описів структур даних, реалізації функцій або класів) в окремих файлах (для окремих мов програмування — також у каталогах) з можливістю об'єднання модулів в програмний продукт на етапі компіляції або виконання;
- логічної завершеності модулів, коли кожен окремий модуль містить елементи, що реалізують деякі логічно закінчені описи та алгоритми;
- обмеженості модулів у функціональності та розмірах, що дозволяє найкращим способом організувати модулі у програму або переносити модулі між програмами та підтримувати їх;
- замкнутості модулів, коли кожен модуль надає свій чітко визначений інтерфейс іншим модулям і використовує інші модулі через їх чітко визначений інтерфейс при повній незалежності одних модулів від значень даних, що містяться в інших модулях;
- універсальності модулів, коли параметри обробки та дані для обробки передаються з інших модулів або файлів конфігурації, а не містяться у модулі як константи або змінні;
- мінімалістичності модулів, коли кожен модуль надає якомога простий інтерфейс.

3. GNU Compiler Collection, GCC/Clang.

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.

Компілятор кожної мови є окремою програмою, яка отримує початковий текст і породжує вихід на мові асемблера. Всі компілятори мають загальну внутрішню структуру:

frontend, який проводить синтаксичний розбір і породжує абстрактне синтаксичне дерево,

і backend, який конвертує дерево в Register Transfer Language (RTL), виконує різні оптимізації, потім породжує програму на мові асемблера, використовуючи архітектурно-залежне зіставлення зі зразком.

GCC майже повністю написаний на Сі, хоча значна частина front-end для Ади написана на Ада.

gcc виконує:

- попередня обробка,
- компіляція,
- збірка,
- зв'язування.

Опції компіляції

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 провадиться.
-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 C++ 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** **-O3**	Різні рівні оптимізації.
-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.

Щоб відкомпілювати вихідний код C++, що знаходиться у файлі **name.cc** і створити об'єктний файл **name.o**, необхідно виконати команду:

gcc -c name.cc

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 name -o name1.o name2.o

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

Опція Wall показує більшість попереджень, пов’язаних із, можливо, неправильним кодом.

```
#include <stdio.h>

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

```
[root@ksp cpp]# gcc -Wall main.c -o main
main.c: In function ‘main’:
main.c:5:6: warning: unused variable ‘unused_var’ [-Wunused-variable]
  int unused_var;
      ^
```

-Wall - це комбінація великого загального набору параметрів -W all. Зазвичай вони включають:

- невикористані змінні;
- можливо, неініціалізовані змінні під час першого використання;
- типи повернення за замовчуванням;
- відсутність фігурних дужок у певному контексті, що робить його неоднозначним;
- тощо.

```
#include <stdio.h>

int main()
{
    int unused_var;
    printf("Hello linux world\n");
    printf("var = %d", unused_var);
    return 0;
}
```

```
[root@ksp cpp]# gcc -Wall main1.c -o main1
main1.c: In function ‘main’:
main1.c:7:8: warning: ‘unused_var’ is used uninitialized in this
 function [-Wuninitialized]
  printf("var = %d", unused_var);
        ^
```

Завжди рекомендований варіант, щоб уберегти ваш код від деяких «прихованих» помилок.

При виправленні помилки отримаємо:

```
#include <stdio.h>

int main()
{
    int unused_var = 7;
    printf("Hello linux world\n");
    printf("var = %d\n", unused_var);
    return 0;
}
```

```
[root@ksp cpp]# gcc -Wall main1.c -o main1
[root@ksp cpp]# ./main1
Hello linux world
var = 7
```

Різні рівні оптимізації коду -O, -O1, -O2, -O3, -O0, -Os

Від -O1 до -O3 – різні ступені оптимізації швидкості

Якщо додається -O, то враховується розмір коду

-O0 означає «без оптимізації»

-Os націлює на розмір згенерованого коду (змушує не використовувати оптимізацію, яка призводить до збільшення коду).

У процесі компонування часто доводиться використовувати бібліотеки. Бібліотекою називають набір об'єктних файлів, згрупованих у єдиний файл та проіндексованих. Коли команда компонування виявляє деяку бібліотеку у списку об'єктних файлів для компонування, вона перевіряє, чи містять вже скомпоновані об'єктні файли виклики для функцій, визначених у одному з файлів бібліотек. Якщо такі функції знайдені, відповідні виклики зв'язуються з кодом об'єктного файлу з бібліотеки. Бібліотеки можуть бути підключені за допомогою опції виду -lname. У цьому випадку в стандартних каталогах, таких як /lib, /usr/lib, /usr/local/lib, буде проведено пошук бібліотеки у файлі з ім'ям libname.a. Бібліотеки повинні бути перераховані після вихідних або об'єктних файлів, які містять виклики до відповідних функцій.

Clang — це новий компілятор для C-подібних мов, створений спеціально для роботи на базі LLVM.

На відміну від GCC, Clang спочатку спроектований для максимального збереження інформації у процесі компіляції, у тому числі збереження «зовнішнього вигляду» вихідного коду. Ця особливість дозволяє Clang створювати розгорнуті контекстно-орієнтовані повідомлення про помилки, зрозумілі як для програмістів, так і для середовищ розробки. Модульний дизайн компілятора дозволяє застосовувати його у складі середовищ розробки для індексування коду, підсвітки синтаксису і рефакторингу.

Clang підтримує більшість розповсюджених опцій GCC.

4. Розбиття програми на різні файли.

Коли потрібно працювати з достатньо великою програмою часто потрібно розбити її на якийсь певні логічні, функціональні частини які можна, а часто й потрібно винести в різні файли. Наприклад, це робиться для того щоб декілька програмістів працювало над одним проектом, відокремити частини функціоналу та інтерфейсів для різних підпроектів і таке інше.

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));
}
```

Тут доцільно розбити програму на 2 частини - з функцією add (файл add.c) та головною функцією (файл main.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));
}
```

Однак, якщо ми просто скомпілюємо тепер файл - то програма може не запрацювати:

```
[root@ksp cpp]# gcc mains.c -o mains
/tmp/cc2DUDoi.o: In function `main':
mains.c:(.text+0x14): undefined reference to `add'
collect2: error: ld returned 1 exit status
```

undefined reference to `add' coll ect 2: error: ld returned 1 exit status

Повідомлення каже нам, що невизначене посилання на `add`.

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

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

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

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.

Для використання функціоналу інших програмних файлів в мовах С та С++ використовується директива (макрокоманда) #include .

Синтаксис даної директиви наступний: після #include йде ім'я файлу (з розширенням для С та без розширення для С++), взяте або в кутові дужки (< >), або в подвійні лапки (“ ”), тобто має одну з двох форм:

1. #include < filename.h>
2. #include "filename.h"

Перша форма вказує транслятору, що файл, ім’я якого вказано в кутових дужках, треба шукати в спеціальній директорії (директоріях), що вказуються як системні шляхи, які містять стандартні або ті, що ми хочемо щоб вони трактувались як стандартні, заголовні файли. Саме тому в кутових дужках вказують імена таких заголовних файлів, як stdio.h , math.h і т.п., які використовуються компілятором. Друга форма макрокоманди означає, що транслятор повинен шукати файл в директорії з програмою, або в певній користувацький директорії (можливо, зокрема, у тій самій директорії, що й програма модуля).

Тобто директива include дає транслятору команду знайти файл з відповідним іменем та підставити весь вміст цього файлу на місце макрокоманди.

Більш точно процес роботи директиви виглядає так:

• транслятор читає файл рядок за рядком;
• коли транслятор бачить в тексті директиву #include, він:
• тимчасово припиняє обробляти поточний файл;
• шукає той файл, ім'я якого вказано в лапках після слова include;
• та починає так само, рядок за рядком, опрацьовувати його;
• дійшовши до кінця заголовного файлу, транслятор знов повертається до недочитаного файлу та продовжує обробляти його з того самого місця, де припинив.

Іншим варіантом може бути компіляція окремо кожного модуля а потім збірка всього у виконуваний файл.

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

Виконали компіляцію файлу add.c

```
[root@ksp cpp]# gcc -c add.c
```

Результатом буде файл add.o

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

Виконали компіляцію файлу func.c

```
[root@ksp cpp]# gcc -c func.c
```

Результатом буде файл func.o

А далі виконаємо збірку у виконуваний бінарний файл func_bin

```
[root@ksp cpp]# gcc add.o func.o -o func_bin
```

Результат виконання

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

5. Компіляція.

Для компіляції використовуємо команду:

gcc <ім'я файлу>

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

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

```
-rwxr-xr-x. 1 root root 8360 Mar  9 14:31 a.out
```

Зазвичай програми на мовах C або C++ є сукупністю кількох .c (.cpp) файлів з реалізаціями функцій і .h файлів з прототипами функцій і визначеннями типів даних. Як правило, кожному .c файлу відповідає .h файл з тим же ім'ям.

Припустимо, що розробляється програма, яка називається general і вона складається з файлів gen_file1.c, gen_file1.h, gen_file2.c, gen_file2.h, gen_file3.c, gen_file3.h. Розробка програми ведеться в POSIX-середовищі з використанням компілятору GCC.

Найпростіший спосіб скомпілювати програму - вказати всі вихідні .c файли в командному рядку gcc:

gcc gen_file1.c gen_file2.c gen_file3.c -o general

Компілятор gcc виконає всі етапи компіляції вихідних файлів програми і компоновку виконуваного файлу general. Зверніть увагу, що в командному рядку gcc вказуються тільки .c файли і ніколи не вказуються .h файли.

Компіляція і компоновка за допомогою перерахування всіх вихідних файлів в аргументах командного рядка GCC допустима лише для зовсім простих програм.

З ростом числа вихідних файлів ситуація дуже швидко стає некерованою. Крім того, кожен раз всі вихідні файли будуть компілюватися від початку до кінця, що в разі великих проектів займає багато часу. Тому зазвичай компіляція програми виконується в два етапи: компіляція об'єктних файлів і компоновка виконуваної програми з об'єктних файлів. Кожному .c файлу тепер відповідає об'єктний файл, ім'я якого в POSIX-системах має суфікс .o. Таким чином, в даному випадку програма general компонується з об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o наступною командою:

gcc gen_file1.o gen_file2.o gen_file3.o -o general

Кожен об'єктний файл повинен бути отриманий з відповідного вихідного файлу за допомогою такої команди:

gcc -c gen_file1.c

Зверніть увагу, що явно задавати ім'я вихідного файлу необов'язково. Воно буде отримано з імені компільованого файлу заміною суфікса .c на суфікс .o. Отже, для компіляції програми general тепер необхідно виконати чотири команди:

gcc -c gen_file1.c
gcc -c gen_file2.c
gcc -c gen_file3.c
gcc gen_file1.o gen_file2.o gen_file3.o -o general

Хоча тепер для компіляції програми необхідно виконати чотири команди замість однієї, натомість з'являються такі переваги:

• якщо якась зміна внесена в один файл, наприклад, в файл gen_file3.c, немає необхідності перекомпілювати файли gen_file2.o або gen_file1.o; досить перекомпілювати файл gen_file3.o, а потім виконати компоновку програми general;

• компіляція об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o не залежить один від одного, тому може виконуватися паралельно на багатопроцесорному (багатоядерному) комп'ютері.

6. Бібліотеки.

Бібліотека - це пакет коду, який призначений для повторного використання багатьма програмами. Як правило, бібліотека C ++ складається з двох частин:

1) Файл заголовка, який визначає функціональні можливості, які бібліотека виставляє (пропонує) програмам, що використовують її.

2) Скомпільований бінарний файл, який містить реалізацію цієї функціональності, попередньо скомпільовану в машинну мову.

«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.

Статичні бібліотеки містять об'єктний код, пов'язаний з додатком кінцевого користувача, а потім вони стають частиною виконуваного файлу. Ці бібліотеки спеціально використовуються під час компіляції, що означає те, що бібліотека повинна бути у правильному місці, коли користувач хоче скомпілювати свою програму C або C ++.

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.

Однак, оскільки копія бібліотеки стає частиною кожного виконуваного файлу, це може призвести до великої кількості втраченого простору. Статичні бібліотеки також не можуть бути легко оновлені - для оновлення бібліотеки необхідно замінити весь виконуваний файл.

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.

Спільні або динамічні бібліотеки потрібні лише під час виконання, тобто користувач може компілювати свій код без використання цих бібліотек. Коротше кажучи, ці бібліотеки пов'язані між собою під час компіляції для вирішення невизначених посилань, а потім поширюються до програми, щоб програма могла завантажувати її під час виконання. Наприклад, коли ми відкриваємо наші папки з іграми, можна знайти багато файлів .dll (динамічних бібліотек). Оскільки ці бібліотеки можуть спільно використовуватися кількома програмами, вони також називаються спільними бібліотеками.

Оскільки динамічні бібліотеки не пов'язані з вашою програмою, програми, що використовують динамічні бібліотеки, повинні явно завантажувати та взаємодіяти з динамічною бібліотекою. Цей механізм може бути заплутаним і робить незручною взаємодію з динамічною бібліотекою.

7. Встановлення та використання бібліотек.

Процес, необхідний для використання бібліотеки:

Для кожної бібліотеки:

1) Закачайте бібліотеку. Наприклад, завантажте з веб-сайту або за допомогою менеджера пакетів.

2) Встановіть бібліотеку. Розпакуйте її до каталогу або встановіть за допомогою менеджера пакетів, або проінсталюйте її (наприклад, запустить як C/C++ - програму).

3) Вкажіть компілятору, де слід шукати файл(и) та заголовок(ки) для бібліотеки.

Для кожного проекту:

4) Вкажіть лінкеру (компонувальнику), де шукати файл(и) бібліотеки для бібліотеки.

5) Вкажіть лінкеру (компонувальнику), які статичні файли або файли імпортувати.

6) Включіть файли заголовків бібліотеки у вашу програму.

7) Переконайтеся, що програма знає, де можна знайти будь-які динамічні бібліотеки, які використовуються.

Встановлення бібліотеки на C/C++ зазвичай включає чотири кроки:

1) Завантаження бібліотеки. Найкращим варіантом є завантаження попередньо скомпільованого пакета для вашої операційної системи (якщо він існує), тому вам не доведеться самостійно компілювати бібліотеку. Якщо для

вашої операційної системи немає такого пакету, вам доведеться завантажити доданок з вихідним кодом і скомпілювати його самостійно (компіляція проекту).

У Windows бібліотеки, як правило, розповсюджуються у вигляді файлів .zip або безпосередньо dll. У Linux бібліотеки зазвичай розподіляються як пакети (наприклад, .RPM) або теж архівом. Менеджер пакетів може мати деякі з найпопулярніших бібліотек (наприклад, SDL), перероблені вже для легкого встановлення, тому перевірте їх там і встановить, наприклад, командою apt install.

2) Встановіть бібліотеку. У Linux це зазвичай передбачає виклик менеджера пакетів і дозвіл (наприклад надання системних прав) йому виконати всю роботу. У Windows це зазвичай передбачає розпакування бібліотеки в потрібний каталог.

3) Переконайтеся, що компілятор знає, де шукати файли заголовків для бібліотеки. У Windows, як правило, це підкаталог include каталогу, до якого ви встановили файли бібліотеки. На Linux бібліотеки зазвичай встановлюються в /usr/include, які вже повинні бути частиною шляху пошуку файлів. Однак, якщо файли встановлені в іншому місці, вам доведеться сказати компілятору, де їх знайти.

4) Вкажіть компонувальнику, де шукати файли бібліотек. Як і в кроці 3, це зазвичай передбачає додавання каталогу до списку місць, де лінкер шукає бібліотеки, або додайте бібліотеки в ті місця де проект вже шукає бібліотеки (зокрема в системних шляхах). У Windows, це типово підкаталог /lib каталогу, до якого ви встановили файли бібліотеки. В Linux бібліотеки зазвичай встановлюються в /usr/lib, які вже повинні бути частиною шляху пошуку вашої бібліотеки.

Після того, як бібліотека встановлена та IDE знає, де її слід шукати, для кожного проекту, який бажає використовувати бібліотеку, як правило, потрібно виконати наступні 3 кроки:

5) Якщо ви використовуєте статичні бібліотеки або імпортовані бібліотеки, повідомте компонувальнику, які файли бібліотек повинні зв'язати (gcc -L(l) *).

6) Включіть файли заголовків бібліотеки у вашу програму (директива include). Це повідомляє компілятору про всі функціональні можливості, які пропонує бібліотека, щоб програма могла правильно скомпілюватись.

7) Якщо використовуються динамічні бібліотеки, переконайтеся, що програма знає, де їх можна знайти. У Linux, бібліотеки, як правило, встановлюються в /usr/lib, який знаходиться в шляху пошуку за замовчуванням або каталоги змінної середовища PATH. У Windows шлях пошуку за замовченням включає в себе каталог, з якого запускається програма, каталоги, встановлені за допомогою виклику SetDllDirectory(), каталоги Windows, System і System32, а також каталоги змінної середовища PATH. Найпростіший спосіб використовувати .dll - скопіювати .dll в розташування виконуваного файлу.

8. Компіляція.

Для компіляції використовуємо команду:

gcc <ім'я файлу>

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

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

```
-rwxr-xr-x. 1 root root 8360 Mar  9 14:31 a.out
```

Зазвичай програми на мовах C або C++ є сукупністю кількох .c (.cpp) файлів з реалізаціями функцій і .h файлів з прототипами функцій і визначеннями типів даних. Як правило, кожному .c файлу відповідає .h файл з тим же ім'ям.

Припустимо, що розробляється програма, яка називається general і вона складається з файлів gen_file1.c, gen_file1.h, gen_file2.c, gen_file2.h, gen_file3.c, gen_file3.h. Розробка програми ведеться в POSIX-середовищі з використанням компілятору GCC.

Найпростіший спосіб скомпілювати програму - вказати всі вихідні .c файли в командному рядку gcc:

gcc gen_file1.c gen_file2.c gen_file3.c -o general

Компілятор gcc виконає всі етапи компіляції вихідних файлів програми і компоновку виконуваного файлу general. Зверніть увагу, що в командному рядку gcc вказуються тільки .c файли і ніколи не вказуються .h файли.

Компіляція і компоновка за допомогою перерахування всіх вихідних файлів в аргументах командного рядка GCC допустима лише для зовсім простих програм.

З ростом числа вихідних файлів ситуація дуже швидко стає некерованою. Крім того, кожен раз все вихідні файли будуть компілюватися від початку до кінця, що в разі великих проектів займає багато часу. Тому зазвичай компіляція програми виконується в два етапи: компіляція об'єктних файлів і компоновка виконуваної програми з об'єктних файлів. Кожному .c файлу тепер відповідає об'єктний файл, ім'я якого в POSIX-системах має суфікс .o. Таким чином, в даному випадку програма general компонується з об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o наступною командою:

gcc gen_file1.o gen_file2.o gen_file3.o -o general

Кожен об'єктний файл повинен бути отриманий з відповідного вихідного файлу за допомогою такої команди:

gcc -c gen_file1.c

Зверніть увагу, що явно задавати ім'я вихідного файлу необов'язково. Воно буде отримано з імені компільованого файлу заміною суфікса .c на суфікс .o. Отже, для компіляції програми general тепер необхідно виконати чотири команди:

gcc -c gen_file1.c
gcc -c gen_file2.c
gcc -c gen_file3.c
gcc gen_file1.o gen_file2.o gen_file3.o -o general

Хоча тепер для компіляції програми необхідно виконати чотири команди замість однієї, натомість з'являються такі переваги:

• якщо якась зміна внесена в один файл, наприклад, в файл gen_file3.c, немає необхідності перекомпілювати файли gen_file2.o або gen_file1.o; досить перекомпілювати файл gen_file3.o, а потім виконати компоновку програми general;

• компіляція об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o не залежить один від одного, тому може виконуватися паралельно на багатопроцесорному (багатоядерному) комп'ютері.

У разі декількох вихідних .c і .h файлів і відповідних проміжних .o файлів відстежувати, який файл потребує перекомпіляції, стає складно, і тут на допомогу приходить програма make. За описом файлів і команд для компіляції програма make визначає, які файли потребують перекомпіляції, і може виконувати перекомпіляцію незалежних файлів паралельно.

Питання для самоконтролю

1. Яка різниця між компіляторами та інтерпретаторами?
2. Що собою представляє програмний модуль?
3. В чому особливість JIT компіляції?
4. Наведіть три опції компіляції.
5. Поясніть функції опції –Wall.
6. Для чого виконується розбиття програми на окремі файли?
7. Яка різниця між статичною та динамічною бібліотеками?