
# Трансляція та виконання: компілятор, інтерпретатор, компоновувальник

Цей документ представляє собою опорний конспект з теми "Трансляція та виконання: компілятор, інтерпретатор, компоновувальник", який охоплює ключові аспекти перетворення програмного коду від написання до виконання. У конспекті розглядаються принципи роботи компіляторів та інтерпретаторів, етапи компіляції, роль компонувальників, використання бібліотек, методи оптимізації коду та сучасні тенденції в галузі трансляції програм. Матеріал містить теоретичні відомості, практичні приклади та тестові завдання для ефективної підготовки до ЄФВВ з інформаційних технологій.

# Компілятори: основні принципи роботи

Компілятор – це спеціальна програма, яка перетворює вихідний код, написаний мовою програмування високого рівня, на еквівалентний код низького рівня (машинний код або асемблер), який може бути безпосередньо виконаний процесором комп'ютера. Основним призначенням компілятора є забезпечення можливості використання зручних для людини мов програмування, з подальшим їх перетворенням у форму, зрозумілу для комп'ютера.

## Архітектура типового компілятора

Типовий компілятор має модульну структуру і складається з наступних основних компонентів:

- Лексичний аналізатор (scanner) - розбиває вихідний текст на лексеми
- Синтаксичний аналізатор (parser) - перевіряє відповідність коду граматиці мови
- Семантичний аналізатор - перевіряє смислову коректність програми
- Генератор проміжного коду - створює платформо-незалежне представлення
- Оптимізатор - покращує ефективність коду
- Генератор цільового коду - створює машинний код для конкретної платформи

## Статична та динамічна компіляція

Статична компіляція передбачає повне перетворення вихідного коду на машинний код перед виконанням програми. Це традиційний підхід, який використовується в таких мовах, як C і C++. Динамічна (JIT) компіляція передбачає перетворення коду безпосередньо під час виконання програми, що дозволяє застосовувати оптимізації на основі інформації, доступної лише під час виконання.

### Переваги компіляторів

- Висока швидкість виконання готових програм
- Відсутність необхідності в додатковому ПЗ для запуску
- Можливість глибокої оптимізації коду
- Раннє виявлення багатьох типів помилок

### Недоліки компіляторів

- Необхідність повної перекомпіляції при зміні коду
- Складності з крос-платформною розробкою
- Довший час розробки через цикл "редагування-компіляція-запуск"
- Обмежена інтерактивність під час розробки

### Приклади сучасних компіляторів

- GCC (GNU Compiler Collection) для C, C++, Fortran та інших мов
- Clang/LLVM - сучасна інфраструктура компіляції з модульною архітектурою
- MSVC (Microsoft Visual C++) для розробки під Windows
- Intel C++ Compiler - оптимізований для процесорів Intel

Сучасні компілятори часто інтегруються в більш складні системи, що поєднують функції компіляції, компонування, оптимізації та інструменти налагодження для забезпечення ефективного розвитку програмного забезпечення. Вони постійно вдосконалюються, щоб забезпечити кращу продуктивність, безпеку та підтримку нових стандартів мов програмування.

# Етапи компіляції програм

Процес компіляції програми — це складний, багатоетапний процес перетворення вихідного коду у виконуваний машинний код. Кожен етап виконує специфічну функцію та має власні методи аналізу та перетворення коду. Розглянемо детально кожен з основних етапів компіляції.

### Лексичний аналіз (tokenization)

На цьому етапі вихідний текст програми розбивається на послідовність лексем (tokens) — найменших значущих одиниць мови програмування. Лексеми включають ключові слова, ідентифікатори, літерали, оператори та роздільники. Лексичний аналізатор також видаляє коментарі та ігнорує пробіли.

### Синтаксичний аналіз

На цьому етапі лексеми групуються в граматичні структури згідно з правилами граматики мови програмування. Результатом синтаксичного аналізу є побудова дерева розбору або абстрактного синтаксичного дерева (AST), яке відображає структуру програми.

### Семантичний аналіз

Цей етап перевіряє коректність використання типів даних, визначення та використання змінних, перевіряє контекстні обмеження мови. Створюється таблиця символів, яка містить інформацію про ідентифікатори та їхні характеристики.

### Оптимізація проміжного коду

На цьому етапі компілятор перетворює AST або дерево розбору в проміжне представлення (IR), яке оптимізується для покращення ефективності майбутнього машинного коду. Застосовуються різні методи оптимізації: видалення спільних підвиразів, згортання константних виразів, оптимізація циклів тощо.

### Генерація машинного коду

Фінальний етап, на якому оптимізований проміжний код перетворюється на машинний код цільової платформи. Враховуються особливості архітектури процесора, системи команд, регістрів та моделі пам'яті.

## Приклад проходження коду через етапи компіляції

| Етап | Вихідний код: **int sum = a + b;** |
|---|---|
| Лексичний аналіз | [INT], [SUM], [=], [A], [+], [B], [;] |
| Синтаксичний аналіз | DECLARATION(TYPE(int), ASSIGN(ID(sum), ADD(ID(a), ID(b)))) |
| Семантичний аналіз | Перевірка: чи оголошені змінні a і b; перевірка сумісності типів |
| Проміжний код | temp = a + b; sum = temp |
| Оптимізація | sum = a + b (без тимчасової змінної) |
| Машинний код | MOV EAX, [a]; ADD EAX, [b]; MOV [sum], EAX |

Варто зазначити, що в сучасних компіляторах ці етапи можуть бути інтегровані або розбиті на більш дрібні кроки. Наприклад, оптимізація часто виконується на різних рівнях абстракції. Також, деякі компілятори можуть генерувати не машинний код, а проміжне представлення (як у LLVM) або асемблер, який потім буде оброблений асемблером та компонувальником.

Розуміння етапів компіляції є важливим не тільки для розробників компіляторів, але й для програмістів, оскільки це допомагає ефективніше писати код та розуміти природу багатьох помилок, з якими вони стикаються під час розробки програмного забезпечення.

# Інтерпретатори: принципи роботи

Інтерпретатор – це програмне забезпечення, яке безпосередньо виконує інструкції, написані мовою програмування високого рівня, без попереднього перетворення їх у машинний код. На відміну від компілятора, інтерпретатор аналізує та виконує код рядок за рядком у реальному часі, що надає певних переваг і обмежень для процесу розробки та виконання програм.

## Алгоритм роботи інтерпретатора

Типовий інтерпретатор працює за наступним алгоритмом:

1. Зчитування рядка або блоку вихідного коду
2. Аналіз коду (лексичний, синтаксичний, семантичний)
3. Безпосереднє виконання проаналізованого коду
4. Збереження результатів виконання в пам'яті інтерпретатора
5. Перехід до наступного рядка або блоку коду

Цей процес повторюється для кожної частини програми, що виконується. При цьому інтерпретатор підтримує середовище виконання, що містить поточний стан змінних, функцій та об'єктів програми.

## Види інтерпретаторів

### Чисті інтерпретатори

Безпосередньо інтерпретують вихідний код без будь-якого проміжного перетворення (наприклад, старі версії BASIC).

### Інтерпретатори з проміжним кодом

Спочатку перетворюють вихідний код у байт-код, який потім інтерпретується (Python, Java до JIT-компіляції).

### JIT-компілятори

Динамічно компілюють часто виконувані фрагменти коду в машинний код під час виконання для підвищення продуктивності (V8 для JavaScript).

## Переваги інтерпретації

- Немає необхідності в окремому етапі компіляції – програму можна запустити відразу після написання
- Крос-платформність – один і той же код може виконуватися на різних платформах без змін
- Динамічне типізування та виконання спрощують швидке прототипування
- Легше реалізувати інтерактивний режим розробки та налагодження
- Можливість змінювати код програми під час її виконання (метапрограмування)

## Недоліки інтерпретації

- Нижча продуктивність порівняно з скомпільованими програмами через накладні витрати на аналіз коду під час виконання
- Вища споживання пам'яті, оскільки інтерпретатор повинен бути завантажений разом із програмою
- Деякі помилки можуть проявитися лише під час виконання, що ускладнює налагодження
- Менші можливості для оптимізації коду порівняно з компіляторами
- Вихідний код програми доступний користувачам, що може створювати проблеми з інтелектуальною власністю

## Приклади популярних інтерпретаторів

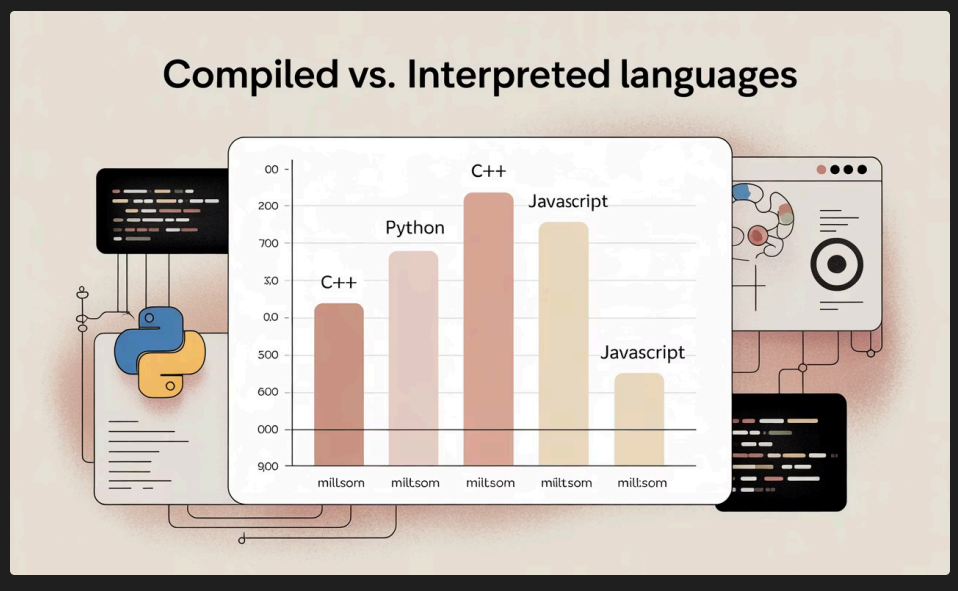
Серед найвідоміших інтерпретаторів можна виділити Python (CPython, PyPy), JavaScript (V8 у Chrome, SpiderMonkey у Firefox), Ruby (MRI, YARV), PHP, Perl, R та інші. Багато сучасних "інтерпретаторів" фактично використовують гібридний підхід, поєднуючи елементи інтерпретації та компіляції для досягнення оптимального балансу між зручністю розробки та продуктивністю.

# Порівняння компіляторів та інтерпретаторів

Компілятори та інтерпретатори представляють два фундаментально різних підходи до трансляції та виконання програмного коду. Кожен з них має свої переваги та недоліки, які роблять їх більш придатними для різних сценаріїв використання. Розуміння цих відмінностей є критично важливим для правильного вибору інструментів розробки та оцінки ефективності програмного забезпечення.

## Швидкість виконання програми

Програми, скомпільовані в машинний код, зазвичай виконуються значно швидше, ніж інтерпретовані програми. Це пов'язано з тим, що компілятор виконує всю роботу з аналізу та перетворення коду заздалегідь, а під час виконання процесор безпосередньо працює з оптимізованим машинним кодом. Інтерпретатор, навпаки, має аналізувати та перетворювати вихідний код під час виконання, що створює додаткові накладні витрати.




Однак сучасні технології JIT-компіляції (Just-In-Time) у таких системах, як Java HotSpot VM або .NET CLR, значно зменшують розрив у продуктивності між компільованими та інтерпретованими мовами. JIT-компілятори аналізують виконання програми в реальному часі та динамічно компілюють "гарячі" ділянки коду, що часто використовуються, в ефективний машинний код.

Також варто зазначити, що продуктивність залежить не тільки від методу трансляції, але й від інших факторів, таких як дизайн мови, ефективність реалізації певних алгоритмів та наявність оптимізованих бібліотек.

## Особливості налагодження

Інтерпретовані мови зазвичай пропонують більш інтерактивне середовище налагодження. Оскільки програма виконується рядок за рядком, розробник може легко зупинити виконання, перевірити значення змінних, змінити код і продовжити виконання. У скомпільованих мовах цикл "редагування-компіляція-запуск" більш тривалий, хоча сучасні IDE значно спрощують цей процес.

## Ресурсні вимоги та ефективність

| Характеристика | Компілятори | Інтерпретатори |
|---|---|---|
| Споживання пам'яті | Менше під час виконання | Більше (потрібно зберігати інтерпретатор і середовище виконання) |
| Розмір виконуваного файлу | Більший (містить весь необхідний код) | Менший (тільки вихідний код) |
| Час запуску | Швидший (код вже готовий до виконання) | Повільніший (потрібна ініціалізація інтерпретатора) |
| Використання CPU | Ефективніше | Менш ефективне через накладні витрати на інтерпретацію |

## Портативність програмного коду

Інтерпретовані мови зазвичай забезпечують кращу портативність, оскільки один і той же вихідний код може виконуватися на будь-якій платформі, де доступний відповідний інтерпретатор. Для скомпільованих мов потрібна перекомпіляція програми для кожної цільової платформи, або використання крос-компіляції.

Мови, що використовують байт-код (Java, C#), займають проміжне положення: вони компілюються в платформо-незалежний байт-код, який потім виконується віртуальною машиною. Це дозволяє досягти балансу між портативністю та продуктивністю.

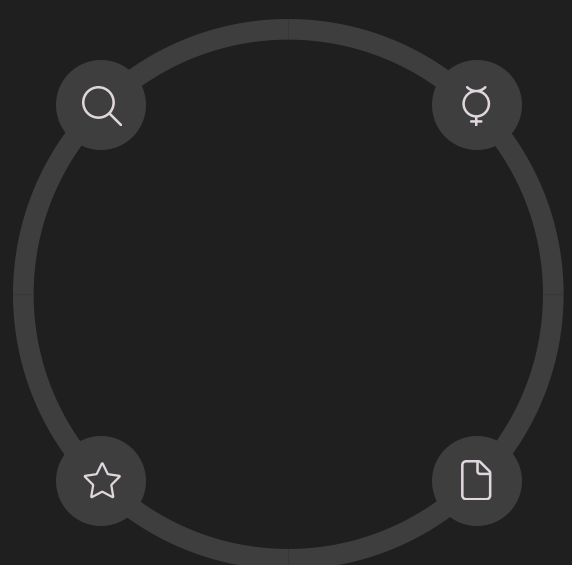
## Порівняння мов програмування за типом трансляції

### Компільовані мови

- C/C++
- Rust
- Go
- Fortran

### Спеціалізовані підходи

- LLVM (інфраструктура компіляції)
- WebAssembly
- PyPy (JIT для Python)



### Гібридні підходи

- Java (байт-код + JIT)
- C# (.NET CLR)
- Kotlin

### Інтерпретовані мови

- Python
- JavaScript
- Ruby
- PHP

Вибір між компілятором та інтерпретатором часто залежить від конкретних вимог проекту: якщо пріоритетом є максимальна продуктивність та ефективність використання ресурсів, то перевага віддається компільованим мовам; якщо ж важливіша швидкість розробки, крос-платформність та гнучкість, то інтерпретовані мови можуть бути кращим вибором.

# Байт-код та віртуальні машини

Байт-код – це проміжне представлення програми, яке є результатом компіляції вихідного коду, але не є машинним кодом для конкретного процесора. Байт-код розроблений для ефективного виконання віртуальною машиною, яка абстрагує програму від деталей конкретної апаратної платформи. Ця технологія поєднує переваги компіляції та інтерпретації, забезпечуючи баланс між портативністю та продуктивністю.

## Концепція байт-коду

Байт-код отримав свою назву тому, що інструкції зазвичай складаються з одного байта для операційного коду (opcode), після якого можуть йти додаткові байти для операндів. Такий формат забезпечує компактність та ефективність виконання. Байт-код є більш абстрактним, ніж машинний код, але більш конкретним, ніж вихідний код програми.

Основні характеристики байт-коду:

- Незалежність від апаратної платформи
- Компактність порівняно з вихідним кодом
- Можливість ефективної верифікації та оптимізації
- Простіша структура інструкцій порівняно з високорівневим кодом

## Віртуальні машини

### Java Virtual Machine (JVM)

Виконує байт-код Java (.class файли). Використовується для Java, Kotlin, Scala, Groovy та інших JVM-мов. Забезпечує портативність за принципом "скомпілюй один раз, запускай будь-де" (write once, run anywhere).

### Common Language Runtime (CLR)

Основа платформи .NET. Виконує проміжний код IL (Intermediate Language). Підтримує мови C#, F#, VB.NET. Забезпечує керування пам'яттю, обробку винятків та безпеку типів.

### V8 (JavaScript)

Рушій JavaScript, розроблений Google для Chrome. Компілює JavaScript безпосередньо в машинний код без генерації байт-коду, але використовує проміжні представлення для оптимізації.

### Python Virtual Machine

CPython компілює код Python в байт-код (.pyc файли), який потім виконується. Альтернативні реалізації, такі як PyPy, використовують JIT-компіляцію для підвищення продуктивності.

## JIT-компіляція: принципи роботи

Just-In-Time (JIT) компіляція – це техніка, яка поєднує переваги компіляції та інтерпретації. JIT-компілятор працює під час виконання програми та перетворює байт-код у машинний код "на льоту".

### Профілювання виконання

JIT-компілятор спочатку дозволяє програмі виконуватися в режимі інтерпретації, збираючи статистику про "гарячі" ділянки коду, які виконуються найчастіше.

### Компіляція "гарячого" коду

Виявлені "гарячі" ділянки компілюються в оптимізований машинний код, що значно підвищує продуктивність.

### Адаптивна оптимізація

JIT-компілятор продовжує моніторити виконання і може перекомпілювати код з більш агресивними оптимізаціями, якщо він виконується дуже часто.

### Деоптимізація при потребі

Якщо припущення, на яких базувалася оптимізація, стають недійсними (наприклад, через динамічне завантаження нових класів), JIT може повернутися до менш оптимізованого коду.

## Порівняння продуктивності

Продуктивність виконання байт-коду з JIT-компіляцією часто наближається до продуктивності нативного коду, особливо для довготривалих програм, де JIT має достатньо часу для оптимізації. Проте існують і відмінності:

| Аспект | Нативний код | Байт-код з JIT |
|---|---|---|
| Час запуску | Швидший (вже скомпільований) | Повільніший (потрібна ініціалізація VM та початкова компіляція) |
| Пікова продуктивність | Висока, стабільна | Може досягати нативної після "розігріву", але з більшою варіативністю |
| Оптимізація | Статична, на етапі компіляції | Динамічна, з урахуванням реального виконання |
| Використання пам'яті | Зазвичай менше | Більше (VM + кеш скомпільованого коду) |

Приклади використання байт-коду в різних мовах включають: Java (.class файли), C# (MSIL у .exe та .dll файлах), Python (.pyc файли) та багато інших. Ця технологія стала стандартом для сучасних платформ розробки, забезпечуючи ефективний баланс між продуктивністю, портативністю та безпекою виконання.

# Компонувальники (лінкери): призначення та функції

Компонувальник (лінкер) – це програмний інструмент, який приймає один або кілька об'єктних файлів, згенерованих компілятором, та об'єднує їх у єдиний виконуваний файл, бібліотеку чи інший об'єктний файл. Він відіграє критичну роль у процесі побудови програмного забезпечення, виконуючи останній етап перетворення вихідного коду на виконувану програму.

## Визначення компонувальника та його роль

Основною функцією компонувальника є розв'язання символьних посилань між різними об'єктними файлами та бібліотеками. Коли програма складається з кількох модулів, функції та змінні з одного модуля можуть викликати або посилатися на функції та змінні з інших модулів. Компілятор генерує об'єктний файл для кожного модуля окремо, залишаючи "заглушки" для зовнішніх символів. Компонувальник знаходить ці символи у інших об'єктних файлах та бібліотеках і встановлює правильні адреси для таких посилань.

## Статичне та динамічне компонування

### Статичне компонування

При статичному компонуванні весь необхідний код з бібліотек включається безпосередньо у виконуваний файл під час компонування. Такий підхід має наступні характеристики:

- Повна самодостатність виконуваного файлу
- Відсутність залежностей від зовнішніх бібліотек під час виконання
- Більший розмір виконуваного файлу
- Ускладнене оновлення бібліотечного коду (потрібна перекомпіляція програми)

### Динамічне компонування

При динамічному компонуванні до виконуваного файлу додається лише інформація про необхідні бібліотеки, а сам код завантажується під час виконання. Особливості:

- Менший розмір виконуваного файлу
- Економія пам'яті при використанні однієї бібліотеки кількома програмами
- Можливість оновлення бібліотек без перекомпіляції програм
- Залежність від наявності необхідних бібліотек у системі ("DLL hell")

## Процес розв'язання символьних посилань

Процес розв'язання символьних посилань є однією з ключових функцій компонувальника і включає наступні етапи:

1. Сканування вхідних об'єктних файлів та формування глобальної таблиці символів
2. Визначення невирішених (зовнішніх) символів у кожному об'єктному файлі
3. Пошук визначень цих символів у таблиці символів
4. Обчислення остаточних адрес для кожного символу
5. Оновлення всіх посилань на символи відповідними адресами
6. Обробка переміщень (relocations) – коригування адрес при зміні розташування коду та даних

## Формати об'єктних файлів та виконуваних файлів

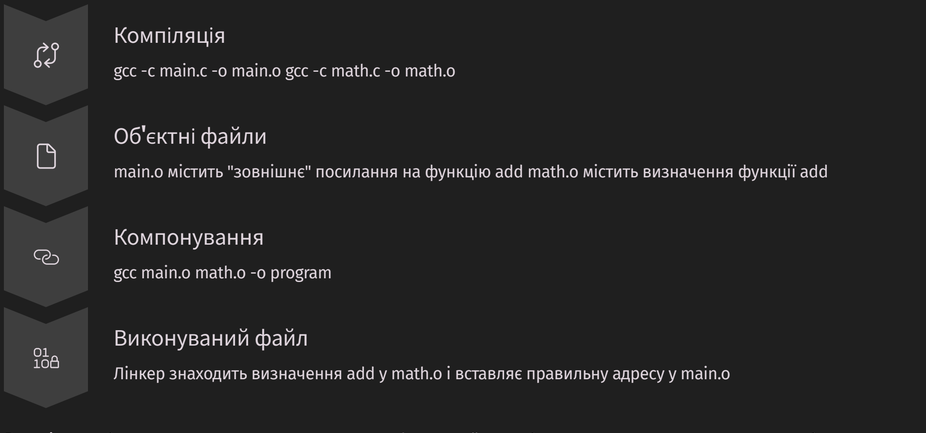
| Платформа | Формат об'єктного файлу | Формат виконуваного файлу |
|---|---|---|
| Windows | COFF, PE-COFF | PE (Portable Executable) |
| Linux, Unix | ELF | ELF (Executable and Linkable Format) |
| macOS | Mach-O | Mach-O (Mach Object) |

## Приклад роботи лінкера з вихідним кодом

Розглянемо простий приклад:

```
// main.c
extern int add(int a, int b);
int main() {
    return add(5, 3);
}

// math.c
int add(int a, int b) {
    return a + b;
}
```




Розуміння роботи компонувальника важливе для ефективної розробки великих програмних систем, особливо при модульній структурі проектів та використанні сторонніх бібліотек. Воно також допомагає розв'язувати типові проблеми, пов'язані з помилками лінкування, такі як "undefined reference" або "unresolved external symbol".

# Бібліотеки та їх використання в процесі компонування

Бібліотеки є важливою складовою сучасної розробки програмного забезпечення, оскільки вони дозволяють повторно використовувати код, розділяти функціональність між різними програмами та ефективно організовувати великі проекти. У процесі компонування бібліотеки відіграють ключову роль, забезпечуючи доступ до попередньо скомпільованого коду, який можна використовувати в різних програмах.

## Статичні бібліотеки: особливості використання

Статичні бібліотеки (з розширеннями .a для Unix/Linux, .lib для Windows) – це архіви, що містять набір об'єктних файлів. Вони використовуються при статичному компонуванні, коли весь необхідний код безпосередньо включається у виконуваний файл.

### Переваги статичних бібліотек

- Відсутність залежностей від зовнішніх бібліотек під час виконання
- Повна самодостатність виконуваного файлу
- Відсутність проблем з версіями бібліотек
- Потенційно швидше виконання програми (немає накладних витрат на завантаження бібліотек)

### Недоліки статичних бібліотек

- Збільшений розмір виконуваного файлу
- Неефективне використання пам'яті, якщо кілька програм використовують одну бібліотеку
- Необхідність перекомпіляції програми при оновленні бібліотеки
- Потенційно більш тривалий час компонування

Створення статичної бібліотеки зазвичай включає компіляцію вихідних файлів в об'єктні файли, а потім використання архіватора (ar в Unix/Linux) для об'єднання їх у єдиний архівний файл.

## Динамічні бібліотеки

Динамічні бібліотеки (з розширеннями .so для Unix/Linux, .dll для Windows, .dylib для macOS) завантажуються під час виконання програми. Вони можуть бути завантажені як при запуску програми (implicit linking), так і в процесі її виконання за запитом (explicit linking).

### Час завантаження

Динамічні бібліотеки можуть завантажуватися під час запуску програми (implicit linking) або динамічно під час виконання (explicit linking), що дозволяє реалізувати плагінні архітектури та економити ресурси при частковому використанні функціональності.

### Спільне використання коду

Одна фізична копія динамічної бібліотеки в пам'яті може використовуватися кількома програмами одночасно, що значно економить ресурси системи, особливо для бібліотек з великим обсягом коду.

### Оновлення

Динамічні бібліотеки можна оновлювати незалежно від програм, які їх використовують, що спрощує підтримку та виправлення помилок без необхідності перекомпіляції всіх залежних програм.

### Проблеми сумісності

Зміни в API бібліотек можуть призвести до проблем з версіями (DLL hell), коли програми, розраховані на роботу з однією версією бібліотеки, не працюють з іншою версією.

## Механізми завантаження динамічних бібліотек

Існує два основних механізми завантаження динамічних бібліотек:

1. **Неявне зв'язування (Implicit linking)**: бібліотеки завантажуються автоматично під час запуску програми. Інформація про необхідні бібліотеки вбудована у виконуваний файл.
2. **Явне зв'язування (Explicit linking)**: програма завантажує бібліотеки в процесі виконання за допомогою спеціальних API-функцій (dlopen у POSIX-системах, LoadLibrary у Windows).

## Залежності між бібліотеками та їх управління

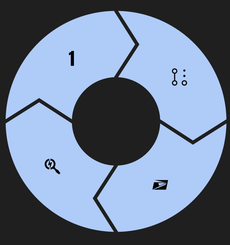
Складні програми можуть мати розгалужену структуру залежностей між бібліотеками. Управління такими залежностями є важливим аспектом розробки та підтримки програмного забезпечення.

### Визначення залежностей

Використання інструментів ldd (Linux), otool (macOS), depends (Windows) для аналізу залежностей бібліотек.

### Пошук бібліотек

Налаштування змінних середовища LD_LIBRARY_PATH, PATH для пошуку бібліотек.




### Управління версіями

Використання семантичного версіонування та механізмів soname/symlinks в UNIX-системах.

### Системи управління пакетами

Використання систем типу apt, yum, brew, vcpkg для контролю залежностей.

## Приклади підключення різних типів бібліотек

```
/* Створення та використання статичної бібліотеки в Unix */

// math.c - код для бібліотеки
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

// Компіляція в об'єктний файл
$ gcc -c math.c -o math.o

// Створення статичної бібліотеки
$ ar rcs libmath.a math.o

// main.c - використання бібліотеки
#include
extern int add(int a, int b);
int main() {
    printf("%d\n", add(10, 5));
    return 0;
}

// Компіляція та лінкування зі статичною бібліотекою
$ gcc main.c -L. -lmath -o program
```

```
/* Використання динамічної бібліотеки в C++ для Windows */

// math.cpp - код для бібліотеки
#include "math.h"
MATH_API int add(int a, int b) { return a + b; }

// math.h - заголовок бібліотеки
#ifdef MATH_EXPORTS
  #define MATH_API __declspec(dllexport)
#else
  #define MATH_API __declspec(dllimport)
#endif
MATH_API int add(int a, int b);

// Компіляція в DLL
$ cl /LD math.cpp /DMATH_EXPORTS /Femath.dll

// main.cpp - використання бібліотеки
#include "math.h"
#include
int main() {
    std::cout << add(10, 5) << std::endl;
    return 0;
}

// Компіляція та лінкування з DLL
$ cl main.cpp math.lib /Feprogram.exe
```

Розуміння особливостей роботи з різними типами бібліотек та їх інтеграції в процес компонування є важливим аспектом розробки надійного та ефективного програмного забезпечення, особливо для великих проектів з численними залежностями.

# Оптимізація коду та її рівні

Оптимізація коду – це процес модифікації програмного коду для покращення його характеристик без зміни функціональності. Компілятори здійснюють автоматичну оптимізацію коду під час трансляції, застосовуючи різноманітні техніки для покращення ефективності, зменшення розміру виконуваного файлу та підвищення швидкодії програми. Розуміння принципів та рівнів оптимізації дозволяє розробникам створювати більш ефективні програми.

## Типи оптимізацій

### Машинно-незалежні оптимізації

Це оптимізації, які застосовуються на рівні проміжного представлення коду і не залежать від конкретної архітектури процесора. Вони зосереджені на поліпшенні логіки програми та структури коду:

- Видалення мертвого коду – усунення недосяжних фрагментів коду
- Згортання констант – обчислення виразів з константами під час компіляції
- Розгортання циклів – зменшення накладних витрат на обробку циклів
- Підстановка функцій (інлайнінг) – заміна виклику функції її тілом
- Видалення спільних підвиразів – обчислення повторюваних виразів один раз
- Оптимізація хвостових викликів – перетворення рекурсивних викликів на ітерації

### Машинно-залежні оптимізації

Ці оптимізації враховують особливості конкретної апаратної архітектури і спрямовані на максимальне використання її можливостей:

- Планування інструкцій – оптимальне розміщення інструкцій для конвеєрної обробки
- Розподіл регістрів – ефективне використання доступних регістрів процесора
- Оптимізація коду для кешування – підвищення локальності даних для кращої продуктивності кешу
- Використання специфічних інструкцій процесора (SSE, AVX, NEON)
- Вирівнювання даних – оптимальне розташування даних у пам'яті
- Специфічні для процесора евристики для гілок та переходів

## Рівні оптимізації в сучасних компіляторах

Сучасні компілятори, такі як GCC, Clang та MSVC, пропонують різні рівні оптимізації, які визначають баланс між часом компіляції, розміром вихідного коду та продуктивністю:

| Рівень | Опис | Використання |
|---|---|---|
| O0 | Відсутність оптимізації. Компіляція найшвидша, код найбільш буквально відповідає вихідному. | Налагодження, коли важлива відповідність вихідного і виконуваного коду. |
| O1 | Базова оптимізація. Застосовуються прості оптимізації, які не потребують значного часу компіляції. | Коли потрібен певний баланс між швидкістю компіляції та продуктивністю. |
| O2 | Помірна оптимізація. Включає більшість оптимізацій, крім тих, що суттєво збільшують розмір коду. | Зазвичай використовується для релізних версій програм. |
| O3 | Агресивна оптимізація. Застосовуються всі доступні оптимізації, навіть якщо вони збільшують розмір коду або час компіляції. | Для критичних за продуктивністю ділянок коду. |
| Os | Оптимізація для розміру. Зменшує розмір виконуваного файлу за рахунок деяких оптимізацій продуктивності. | Для вбудованих систем з обмеженою пам'яттю. |
| Ofast | Максимальна оптимізація швидкості, що може порушувати стандарти мови (наприклад, для операцій з плаваючою точкою). | Для високопродуктивних обчислень, де точність обчислень не критична. |

## Автовекторизація та розпаралелювання

Сучасні компілятори здатні автоматично використовувати векторні інструкції процесора (SIMD) та розпаралелювати код для багатоядерних систем:

### Автовекторизація

Перетворює послідовні операції над даними на векторні інструкції, які обробляють кілька елементів одночасно. Це особливо ефективно для обробки масивів та матриць.

### Автоматичне розпаралелювання

Розподіляє обчислення між кількома ядрами процесора. Може застосовуватися до незалежних ітерацій циклів та інших паралельних операцій.

### Оптимізація конвеєрів

Реорганізує інструкції для ефективного використання конвеєрної архітектури процесора, мінімізуючи простої через залежності даних.

## Інлайнінг функцій та розгортання циклів

Ці техніки оптимізації є особливо ефективними для підвищення продуктивності програм:

### Інлайнінг функцій

Полягає у заміні виклику функції її вмістом, що усуває накладні витрати на виклик функції. Особливо ефективний для невеликих функцій, які викликаються часто.

```
// До оптимізації
int square(int x) { return x * x; }
int sum_squares(int a, int b) {
    return square(a) + square(b);
}

// Після інлайнінгу
int sum_squares(int a, int b) {
    return (a * a) + (b * b);
}
```

### Розгортання циклів

Техніка, яка зменшує накладні витрати на цикли шляхом дублювання тіла циклу. Це зменшує кількість перевірок умови та операцій інкрементування лічильника.

```
// До оптимізації
for (int i = 0; i < 100; i++) {
    array[i] = i * 2;
}

// Після розгортання (частково)
for (int i = 0; i < 100; i += 4) {
    array[i] = i * 2;
    array[i+1] = (i+1) * 2;
    array[i+2] = (i+2) * 2;
    array[i+3] = (i+3) * 2;
}
```

## Приклади покращення коду після оптимізації

Розглянемо приклад оптимізації простого алгоритму обчислення суми елементів матриці:

```
// Вихідний код
double sum = 0.0;
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        sum += matrix[i][j];
    }
}
```

Після застосування оптимізацій компілятор може перетворити цей код таким чином:

- Зміна порядку доступу до елементів для кращої локальності кешу
- Розгортання внутрішнього циклу для зменшення накладних витрат
- Використання векторних інструкцій для паралельного сумування кількох елементів
- Розпаралелювання зовнішнього циклу для роботи на кількох ядрах

Оптимізований псевдокод може виглядати приблизно так:

```
// Оптимізований код
double sum1 = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0;
#pragma omp parallel for reduction(+:sum1,sum2,sum3,sum4)
for (int i = 0; i < rows; i += 2) {
    for (int j = 0; j < cols; j += 4) {
        // Векторне завантаження 4 елементів одночасно
        sum1 += matrix[i][j] + matrix[i][j+1] + matrix[i][j+2] + matrix[i][j+3];
        sum2 += matrix[i+1][j] + matrix[i+1][j+1] + matrix[i+1][j+2] + matrix[i+1][j+3];
    }
}
double sum = sum1 + sum2 + sum3 + sum4;
```

Такі оптимізації можуть суттєво підвищити продуктивність програм, особливо для обчислювально інтенсивних задач. Розуміння принципів оптимізації дозволяє розробникам писати код, який компілятор зможе ефективно оптимізувати, а також застосовувати ручні оптимізації там, де автоматичні методи недостатньо ефективні.

# Крос-компіляція та мультиплатформна розробка

Крос-компіляція – це процес компіляції програми на одній платформі (хост-система) для виконання на іншій платформі (цільова система). Цей підхід є фундаментальним для розробки програмного забезпечення для вбудованих систем, мобільних пристроїв та різноманітних операційних систем. Мультиплатформна розробка, у свою чергу, фокусується на створенні програм, які можуть працювати на різних платформах з мінімальними змінами у вихідному коді.

## Визначення та призначення крос-компіляції

Крос-компіляція застосовується у наступних сценаріях:

- Розробка для вбудованих систем, де цільова платформа має обмежені ресурси або не підтримує середовище розробки
- Розробка для мобільних пристроїв (Android, iOS) з використанням настільних комп'ютерів
- Підготовка програм для різних архітектур процесорів (x86, ARM, RISC-V, MIPS)
- Компіляція програм для іншої операційної системи (наприклад, Windows-програми на Linux або навпаки)
- Мінімізація часу збірки для великих проектів шляхом використання потужніших хост-систем

## Архітектурні особливості цільових платформ

При крос-компіляції важливо враховувати численні відмінності між платформами:

| Характеристика | Вплив на компіляцію | Приклади відмінностей |
|---|---|---|
| Порядок байтів (endianness) | Впливає на представлення багатобайтових значень | Little-endian (x86, ARM) vs Big-endian (деякі MIPS, PowerPC) |
| Розрядність | Визначає розмір вказівників і деяких типів даних | 32-bit vs 64-bit архітектури |
| Набір інструкцій | Впливає на генерацію машинного коду | CISC (x86) vs RISC (ARM, RISC-V) |
| Вирівнювання даних | Вимоги до розміщення даних у пам'яті | Деякі архітектури вимагають строгого вирівнювання, інші - ні |
| ABI (Application Binary Interface) | Правила виклику функцій, передачі параметрів | Різні конвенції для x86, ARM, MIPS |
| Системні виклики | Інтерфейси з операційною системою | Різні API в Windows, Linux, macOS |

## Інструменти для крос-компіляції

### Інструментальні ланцюжки (toolchains)

Набори інструментів, що включають компілятор, асемблер, лінкер, бібліотеки та інші утиліти для певної цільової платформи. Приклади: GCC cross-toolchains, LLVM/Clang, Android NDK.

### Системи збірки

Інструменти для автоматизації процесу збірки, які підтримують крос-компіляцію. Приклади: CMake, Meson, Bazel, які дозволяють гнучко налаштовувати параметри компіляції для різних платформ.

### Контейнеризація та віртуалізація

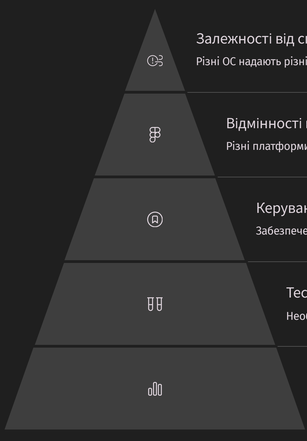
Docker, QEMU та інші інструменти для створення ізольованих середовищ збірки та тестування, які емулюють цільові платформи.

### Мультиплатформні фреймворки

Qt, Flutter, Xamarin, Electron - фреймворки, які абстрагують відмінності між платформами та спрощують крос-платформну розробку.

## Проблеми мультиплатформної розробки

При розробці програм для різних платформ розробники стикаються з численними викликами:



### Залежності від специфічних API

Різні ОС надають різні інтерфейси для доступу до системних ресурсів

### Відмінності в GUI

Різні платформи мають власні стандарти інтерфейсу користувача

### Керування бібліотеками

Забезпечення сумісних бібліотек для різних платформ

### Тестування

Необхідність тестування на всіх цільових платформах

### Оптимізація

Врахування особливостей кожної платформи для досягнення оптимальної продуктивності

Для вирішення цих проблем застосовуються різні підходи, зокрема:

- Використання абстрактних шарів, які ізолюють платформо-залежний код
- Умовна компіляція з використанням препроцесора (#ifdef) для різних платформ
- Впровадження поліморфізму для реалізації платформо-специфічного коду
- Використання крос-платформних бібліотек, які інкапсулюють відмінності між платформами

## Приклади крос-компіляції для різних архітектур

```
# Встановлення GCC крос-компілятора для ARM на Ubuntu
$ sudo apt install gcc-arm-linux-gnueabihf

# Компіляція простої програми для ARM
$ arm-linux-gnueabihf-gcc hello.c -o hello_arm

# Використання CMake для крос-компіляції проекту
$ mkdir build && cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=../arm-toolchain.cmake ..
$ make

# Вміст типового файлу налаштувань toolchain для CMake
# arm-toolchain.cmake
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
set(CMAKE_FIND_ROOT_PATH /usr/arm-linux-gnueabihf)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

Крос-компіляція для Android вимагає використання Android NDK (Native Development Kit):

```
# Налаштування Android NDK
$ export ANDROID_NDK=/path/to/android-ndk

# Компіляція C/C++ коду для Android з використанням ndk-build
$ $ANDROID_NDK/ndk-build APP_ABI=armeabi-v7a,arm64-v8a,x86,x86_64

# Або з використанням CMake
$ mkdir build && cd build
$ cmake -DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK/build/cmake/android.toolchain.cmake \
      -DANDROID_ABI=armeabi-v7a -DANDROID_PLATFORM=android-21 ..
$ make
```

Крос-компіляція та мультиплатформна розробка стають все більш важливими в епоху різноманітних пристроїв та операційних систем. Володіння інструментами та методиками, необхідними для ефективної крос-компіляції, є цінною навичкою для розробників програмного забезпечення, що дозволяє створювати продукти, доступні для широкої аудиторії користувачів на різних платформах.

# Сучасні тенденції в трансляції програм

Технології трансляції програм постійно еволюціонують для задоволення зростаючих вимог до продуктивності, ефективності розробки та підтримки різноманітних платформ. У цьому розділі розглянемо найважливіші сучасні тенденції та інноваційні підходи в області компіляторів, інтерпретаторів та пов'язаних технологій.

## AOT-компіляція в мовах з динамічною типізацією

Ahead-of-Time (AOT) компіляція стає все більш популярною навіть для мов, які традиційно використовують інтерпретацію або JIT-компіляцію:

- Python: проекти типу Nuitka, Cython та MyPyC перетворюють Python-код у компільований код
- JavaScript: TypeScript дозволяє додати статичну типізацію, а інструменти на кшталт Google's Closure Compiler оптимізують JS-код перед виконанням
- Ruby: YARV (Yet Another Ruby VM) використовує AOT-компіляцію для підвищення продуктивності
- PHP: проект KPHP від VK компілює PHP у C++ для значного підвищення продуктивності

AOT-компіляція для динамічних мов вирішує проблему "холодного старту" (cold start), яка є недоліком JIT-компіляції, та забезпечує стабільнішу продуктивність без часу, необхідного для "розігріву" JIT-компілятора.

## Інкрементальна компіляція та її переваги

Інкрементальна компіляція – це техніка, яка дозволяє перекомпілювати тільки ті частини програми, які змінилися, замість повної перекомпіляції:

### Швидший цикл розробки

Значно зменшує час між внесенням змін до коду та можливістю тестування цих змін, що підвищує продуктивність розробників.

### Економія ресурсів

Мінімізує використання процесора та пам'яті під час компіляції, що особливо важливо для великих проектів.

### IDE інтеграція

Дозволяє інтегрованим середовищам розробки надавати миттєвий зворотний зв'язок та підсвічувати помилки в реальному часі.

### Послідовні бінарні артефакти

Забезпечує більш стабільні та передбачувані результати компіляції, уникаючи відмінностей, які можуть виникати при повній перекомпіляції.

Сучасні мови програмування, такі як Rust (з cargo), Kotlin, Swift та Go, мають вбудовану підтримку інкрементальної компіляції, що значно покращує досвід розробки.

## Розподілена компіляція

Розподілена компіляція використовує кілька комп'ютерів для паралельної компіляції різних частин великих проектів:

### Підготовка

Вихідний код розділяється на незалежні модулі

### Розподіл

Модулі розподіляються між серверами компіляції

### Паралельна компіляція

Кожен сервер компілює свою частину коду

### Об'єднання

Скомпільовані артефакти збираються разом для фінального лінкування

Системи розподіленої компіляції, такі як DistCC, Bazel, BuildGrid та Goma від Google, дозволяють значно скоротити час компіляції великих проектів і стають все більш важливими в корпоративному середовищі розробки.

## Нові підходи до оптимізації коду

Сучасні компілятори застосовують все більш складні техніки оптимізації:

### Профіле-орієнтована оптимізація (PGO)

Використовує дані профілювання реального виконання програми для прийняття рішень щодо оптимізації. Компілятор отримує інформацію про те, які шляхи виконання використовуються найчастіше, і оптимізує саме їх.

### Міжпроцедурна оптимізація (IPO/LTO)

Link-Time Optimization (LTO) та Interprocedural Optimization (IPO) дозволяють компілятору оптимізувати код на рівні всієї програми, а не лише окремих файлів, що відкриває нові можливості для оптимізації.

### Поліморфна інлайн-кеш (PIC)

Техніка, що використовується в JIT-компіляторах для оптимізації викликів віртуальних методів шляхом кешування найчастіше використовуваних реалізацій.

### Спекулятивна оптимізація

Компілятор робить припущення про ймовірну поведінку програми та оптимізує код відповідно, додаючи перевірки для випадків, коли припущення не виконуються (deoptimization).

### Автовекторизація та автопаралелізація

Автоматичне перетворення послідовного коду у векторні інструкції або паралельний код для багатоядерних процесорів та GPGPU.

## Машинне навчання в компіляторних технологіях

Машинне навчання стає потужним інструментом для покращення компіляторів та процесів оптимізації:

### Інтелектуальний вибір оптимізацій

ML-моделі передбачають найбільш ефективні оптимізації для конкретного коду

### Адаптивна оптимізація під час виконання

Динамічне налаштування оптимізацій на основі поведінки програми

### Автоматична генерація коду

Використання ML для створення оптимізованих реалізацій алгоритмів

4

### Прогнозування продуктивності

Передбачення ефекту оптимізацій без фактичного виконання коду

Проекти, такі як TVM від Apache, MLIR від LLVM, та Google's TensorFlow Compiler (XLA), демонструють потенціал машинного навчання для підвищення ефективності компіляції та оптимізації, особливо для областей штучного інтелекту та високопродуктивних обчислень.

## Інші значущі тенденції

- **WebAssembly (Wasm)**: бінарний формат для виконання коду в веб-браузерах, що дозволяє запускати додатки, написані різними мовами, в браузері з продуктивністю, близькою до нативних програм
- **Компіляція в LLVM IR**: більшість сучасних мов використовують LLVM як backend, що спрощує створення нових мов та забезпечує високу якість генерації коду
- **JIT-компіляція в керованих середовищах**: такі платформи як .NET Core, Java HotSpot VM та V8 продовжують вдосконалювати свої JIT-компілятори для забезпечення продуктивності, близької до нативного коду
- **Інструменти статичного аналізу**: інтеграція статичного аналізу в процес компіляції для виявлення помилок та вразливостей на ранніх етапах
- **Формальна верифікація**: компілятори з підтвердженою коректністю, які гарантують, що оптимізації не змінюють семантику програми

Ці тенденції відображають постійний прогрес у галузі технологій трансляції програм, спрямований на підвищення продуктивності, ефективності та надійності програмного забезпечення в контексті все більш складних та різноманітних обчислювальних середовищ.

# Практичні приклади використання компіляторів та інтерпретаторів

У цьому розділі ми розглянемо практичні приклади використання різних інструментів трансляції програмного коду. Ці приклади демонструють процеси компіляції, інтерпретації, лінкування та аналізу коду, які використовуються у повсякденній практиці програмування.

## Компіляція та виконання програми на C/C++

Розглянемо процес компіляції простої програми на C:

```
// hello.c
#include

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

### Препросесинг

Розширення директив препроцесора, включення заголовочних файлів

```
$ gcc -E hello.c -o hello.i
```

### Компіляція

Перетворення коду C в асемблерний код

```
$ gcc -S hello.i -o hello.s
```

### Асемблювання

Перетворення асемблерного коду в об'єктний файл

```
$ gcc -c hello.s -o hello.o
```

### Лінкування

Створення виконуваного файлу з об'єктних файлів та бібліотек

```
$ gcc hello.o -o hello
```

Ці команди можна об'єднати в одну, і компілятор виконає всі етапи послідовно:

```
$ gcc hello.c -o hello
```

Для C++ процес аналогічний, але використовується компілятор g++ (або clang++ для LLVM):

```
$ g++ hello.cpp -o hello
```

Додавання оптимізацій та інших параметрів:

```
$ gcc -O2 -Wall -std=c11 hello.c -o hello
```

## Інтерпретація коду Python та JavaScript

Python та JavaScript є чудовими прикладами інтерпретованих мов з різними підходами до виконання коду.

### Python

Стандартний інтерпретатор Python (CPython) компілює вихідний код у байт-код, який потім виконується віртуальною машиною.

```
# hello.py
def greet(name):
    return f"Привіт, {name}!"

print(greet("Світ"))

# Запуск інтерпретатора
$ python hello.py

# Компіляція в байт-код
$ python -m py_compile hello.py
# Створює __pycache__/hello.cpython-xx.pyc

# Дизасемблювання байт-коду
$ python -m dis hello.py
```

### JavaScript

JavaScript зазвичай виконується в браузері або в Node.js. Сучасні рушії JavaScript використовують JIT-компіляцію для підвищення продуктивності.

```
// hello.js
function greet(name) {
    return `Привіт, ${name}!`;
}

console.log(greet("Світ"));

// Запуск у Node.js
$ node hello.js

// Запуск з V8 з відображенням оптимізацій
$ node --trace-opt hello.js

// Запуск з Babel для транспіляції
$ npx babel hello.js --out-file hello.transpiled.js
```

## Трансляція та виконання програм на Java/C#

Java та C# використовують подібний підхід: вони компілюються у байт-код, який потім виконується віртуальною машиною з JIT-компіляцією.

### Java

```
// Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Привіт, світ!");
    }
}

// Компіляція в байт-код
$ javac Hello.java
// Створює Hello.class

// Запуск програми через JVM
$ java Hello

// Перегляд байт-коду
$ javap -c Hello.class
```

### C#

```
// Hello.cs
using System;

class Hello {
    static void Main() {
        Console.WriteLine("Привіт, світ!");
    }
}

// Компіляція в IL
$ csc Hello.cs
// Створює Hello.exe

// Запуск програми через CLR
$ Hello.exe
// або
$ dotnet Hello.dll

// Перегляд IL
$ ildasm Hello.exe
```

## Робота з компонувальником: вирішення типових помилок

При роботі з компонувальником можуть виникати різні помилки. Розглянемо типові проблеми та їх вирішення:

| Помилка | Причина | Вирішення |
|---|---|---|
| undefined reference / unresolved external symbol | Символ (функція, змінна) використовується, але не визначений | Додати відсутній об'єктний файл або бібліотеку до лінкування |
| multiple definition / duplicate symbol | Символ визначено кілька разів | Використовувати extern або static, виправити дублювання у вихідному коді |
| library not found | Компонувальник не може знайти вказану бібліотеку | Додати шлях до бібліотеки: -L/path/to/lib |
| relocation truncated to fit | Невідповідність розміру адреси (зазвичай 32/64-біт проблеми) | Використовувати правильні прапорці компіляції: -m32 або -m64 |

Приклад вирішення помилки "undefined reference":

```
// main.c
extern int add(int a, int b);
int main() {
    return add(5, 3);
}

// math.c
int add(int a, int b) {
    return a + b;
}

// Спроба компіляції тільки main.c призведе до помилки
$ gcc main.c -o program
// main.c:(.text+0x14): undefined reference to `add'

// Правильне лінкування з обома файлами
$ gcc main.c math.c -o program
```

## Аналіз проміжного коду та дизасемблювання

Аналіз проміжного та машинного коду є важливим інструментом для розуміння роботи компіляторів та оптимізації програм.

### Дизасемблювання об'єктних файлів

Інструменти: objdump, ndisasm, IDA Pro

```
$ objdump -d hello.o
```

### Аналіз байт-коду

Інструменти: javap (Java), dis (Python), monodis (C#/.NET)

```
$ javap -c -verbose Hello.class
```

### Перегляд проміжного представлення

Інструменти: llvm-ir (LLVM), -fdump-tree-* (GCC)

```
$ clang -S -emit-llvm hello.c -o hello.ll
```

### Аналіз оптимізацій

Інструменти: -fopt-info (GCC), -Rpass (Clang)

```
$ gcc -O2 -fopt-info-all hello.c
```

Приклад аналізу асемблерного коду, згенерованого GCC:

```
// Наш вихідний код:
int square(int x) {
    return x * x;
}

// Компіляція з виведенням асемблера:
$ gcc -S -O2 -masm=intel sample.c -o sample.s

// Результуючий асемблерний код:
square:
    mov     eax, edi
    imul    eax, edi
    ret
```

Аналіз проміжного коду та дизасемблювання допомагає зрозуміти, як компілятор перетворює високорівневий код у машинні інструкції, які оптимізації застосовуються, та як можна покращити продуктивність програми шляхом змін у вихідному коді.

Практичне розуміння процесів трансляції та виконання програм дозволяє розробникам ефективніше налагоджувати код, оптимізувати продуктивність та вирішувати проблеми, пов'язані з компіляцією та виконанням програм на різних платформах.

# Тестові завдання для перевірки знань

Перевірте свої знання з теми "Трансляція та виконання: компілятор, інтерпретатор, компоновувальник" за допомогою наступних тестових завдань. Оберіть одну правильну відповідь з чотирьох запропонованих варіантів.

1

## Питання

Яка основна відмінність між компілятором та інтерпретатором?

1. Компілятор працює швидше, ніж інтерпретатор
2. Компілятор перетворює весь код у машинний перед виконанням, а інтерпретатор виконує код рядок за рядком
3. Інтерпретатор може працювати з будь-якою мовою програмування, а компілятор - ні
4. Компілятор завжди створює більш оптимізований код, ніж інтерпретатор

**Правильна відповідь: B**

**Пояснення:** Основна відмінність полягає в тому, що компілятор переводить вихідний код програми в машинний код до початку виконання програми, створюючи окремий виконуваний файл, тоді як інтерпретатор аналізує та виконує код послідовно під час виконання програми, рядок за рядком, без створення окремого машинного файлу.

2

## Питання

Який етап компіляції відповідає за перевірку правил мови програмування?

1. Лексичний аналіз
2. Синтаксичний аналіз
3. Генерація коду
4. Оптимізація

**Правильна відповідь: B**

**Пояснення:** Синтаксичний аналіз (парсинг) перевіряє, чи дотримується код граматичних правил мови програмування. На цьому етапі будується дерево розбору або абстрактне синтаксичне дерево, що представляє структуру програми відповідно до правил граматики мови.

3

## Питання

Яка роль компонувальника (лінкера) у процесі компіляції?

1. Перетворює вихідний код на машинний
2. Оптимізує виконання програми
3. Об'єднує об'єктні файли та бібліотеки у виконуваний файл
4. Виконує аналіз синтаксичних помилок

**Правильна відповідь: C**

**Пояснення:** Компонувальник (лінкер) об'єднує об'єктні файли, згенеровані компілятором, та необхідні бібліотеки в єдиний виконуваний файл. Він розв'язує символьні посилання між різними модулями програми, встановлюючи правильні адреси для викликів функцій та доступу до даних.

4

## Питання

Що таке JIT-компіляція?

1. Компіляція вихідного коду безпосередньо в машинний код без проміжних етапів
2. Компіляція частин програми під час її виконання
3. Компіляція з використанням попередньо скомпільованих шаблонів
4. Компіляція, що виконується на різних комп'ютерах одночасно

**Правильна відповідь: B**

**Пояснення:** JIT (Just-In-Time) компіляція - це техніка, при якій байт-код або проміжний код компілюється в машинний код під час виконання програми, а не заздалегідь. Це дозволяє застосовувати оптимізації на основі інформації, доступної тільки під час виконання, та забезпечує баланс між швидкістю виконання та портативністю.

5

## Питання

Яка перевага статичних бібліотек перед динамічними?

1. Менший розмір виконуваного файлу
2. Можливість оновлення бібліотеки без перекомпіляції програми
3. Відсутність залежностей під час виконання програми
4. Економія пам'яті при використанні кількома програмами

**Правильна відповідь: C**

**Пояснення:** Основною перевагою статичних бібліотек є те, що вони включаються безпосередньо у виконуваний файл програми при компіляції. Це забезпечує відсутність зовнішніх залежностей під час виконання - програма містить всі необхідні функції та не потребує наявності бібліотек у системі користувача.

6

## Питання

Що таке байт-код?

1. Машинний код, оптимізований для конкретного процесора
2. Проміжне представлення програми для виконання віртуальною машиною
3. Послідовність байтів, що представляють вихідний код програми
4. Код, написаний мовою асемблера

**Правильна відповідь: B**

**Пояснення:** Байт-код - це проміжне представлення програми, розроблене для ефективного виконання віртуальною машиною. На відміну від машинного коду, який виконується безпосередньо процесором, байт-код інтерпретується або компілюється "на льоту" віртуальною машиною. Типові приклади - байт-код Java (.class файли) та Python (.pyc файли).

7

## Питання

Для чого використовується крос-компіляція?

1. Для одночасної компіляції програми на різних мовах програмування
2. Для компіляції програми на одній платформі для виконання на іншій
3. Для паралельної компіляції різних частин великої програми
4. Для компіляції програми з різними рівнями оптимізації

**Правильна відповідь: B**

**Пояснення:** Крос-компіляція - це процес компіляції програми на одній платформі (хост-системі) для виконання на іншій платформі (цільовій системі). Це необхідно при розробці програм для вбудованих систем, мобільних пристроїв або інших платформ, де безпосередня компіляція на цільовій системі неможлива або неефективна.

8

## Питання

Яке твердження щодо рівнів оптимізації компілятора (-O0, -O1, -O2, -O3) є правильним?

1. Вищий рівень оптимізації завжди призводить до швидшого виконання програми
2. Рівень -O0 забезпечує найшвидшу компіляцію та найкращу підтримку налагодження
3. Рівень -O3 завжди створює код, який займає менше пам'яті
4. Рівні оптимізації впливають тільки на швидкість виконання програми, але не на її розмір

**Правильна відповідь: B**

**Пояснення:** Рівень оптимізації -O0 (або відсутність оптимізації) забезпечує найшвидшу компіляцію та найбільш пряму відповідність між вихідним та виконуваним кодом, що полегшує налагодження. На цьому рівні компілятор не витрачає час на аналіз та оптимізацію коду, що прискорює процес компіляції, хоча і призводить до менш ефективного виконуваного коду.

9

## Питання

Які мови програмування зазвичай використовують інтерпретацію?

1. C та C++
2. Python та JavaScript
3. Java та C#
4. Fortran та COBOL

**Правильна відповідь: B**

**Пояснення:** Python та JavaScript традиційно є інтерпретованими мовами. Python використовує інтерпретатор, який виконує байт-код (.pyc файли), а JavaScript інтерпретується безпосередньо в браузері або середовищі Node.js. Хоча сучасні реалізації цих мов часто використовують JIT-компіляцію для підвищення продуктивності, вони все ще належать до категорії інтерпретованих мов.

10

## Питання

Що означає помилка "undefined reference" при компонуванні?

1. У вихідному коді є синтаксична помилка
2. Компілятор не може знайти заголовочний файл
3. Компонувальник не може знайти визначення для символу, який використовується в програмі
4. У програмі використовується змінна без ініціалізації

**Правильна відповідь: C**

**Пояснення:** Помилка "undefined reference" (або "unresolved external symbol" у деяких компіляторах) виникає, коли компонувальник (лінкер) не може знайти визначення для функції, змінної або іншого символу, який використовується в програмі. Це може статися, якщо відсутній об'єктний файл або бібліотека, які містять це визначення, або якщо символ неправильно оголошений (наприклад, різні сигнатури функції в оголошенні та визначенні).

Ці тестові завдання допоможуть вам перевірити своє розуміння ключових концепцій трансляції та виконання програм, особливостей роботи компіляторів, інтерпретаторів та компонувальників. Якщо ви успішно відповіли на більшість питань, ви маєте хороше розуміння теми. Якщо ж у вас виникли труднощі, рекомендується повернутися до відповідних розділів конспекту для повторення матеріалу.