ТЕОРІЯ МОВ ПРОГРАМУВАННЯ

Лекція 7

Імплементація ООП (початок)

Весна 2021

Вже починаючи з цієї лекції ви дізнаєтеся про реалізацію віртуальної машини JCoCo, одночасно ознайомившись з Java та C++, двома статично обраними об'єктно-орієнтованими мовами програмування. Основна увага в перших трьох лекціях приділяється вивченню вдосконаленого об’єктно-орієнтованого програмування за допомогою Java та C ++. Статично обрані мови, такі як C ++ та Java, відрізняються від динамічно обраних мов, таких як Python, способом виявлення помилок типу. При запуску програми Python помилка типу може виникати в будь-якій гілці коду.

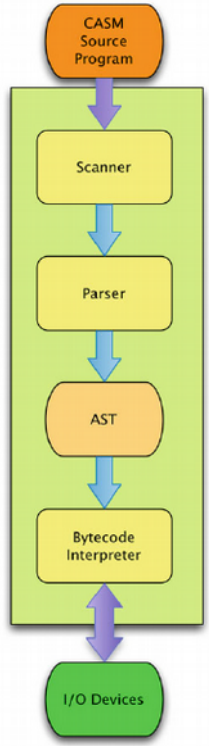
Однією з найбільших проблем програмування на Python є те, що ці помилки можуть існувати доти, доки не буде виконаний кожний можливий шлях у програмі на Python. Тестування коду Python вимагає значних зусиль, щоб забезпечити виконання всіх можливих шляхів. Хоча ретельне тестування - завжди гарна ідея, але ці помилки типу можуть бути виявлені лише набагато пізніше в циклі розробки.

Програми Java і C ++ набираються статично. Це означає, що помилки типу виявляються під час компіляції, коли програма перекладається у виконуваний формат, а не виконує програму взагалі. Програмісти повинні оголосити типи всіх значень, або компілятор повинен мати можливість вивести їх тип із контексту виразів у програмі. При оголошенні типів значень у програмах на C ++ та Java програміст отримує повідомлення, якщо будь-яка операція заборонена без виконання одного рядка коду. Помилки під час виконання все ще можливі, але ці помилки під час виконання виникають через логічні проблеми, а не через помилки типу.

Реалізація JCoCo буде гарним прикладом під час вивчення Java та C ++. Ми порівняємо Java та C ++, коли це доречно, щоб показати вам відмінності та подібності між двома мовами. Щоб побачити загальну картину, ми почнемо з огляду реалізації JCoCo, як показано на рис. 7.1. JCoCo читає вхідний файл, який повинен бути відформатований відповідно до граматики, зазначеної в додатку до лекцій A.1.

Дві частини реалізації JCoCo, сканер та синтаксичний аналізатор, відповідають за зчитування вхідного файлу. Сканер реалізований як кінцевий автомат. Синтаксичний аналізатор записаний як парсер зверху вниз. Синтаксичний аналізатор створює список визначень функцій та класів, що складають визначення програми абстрактного дерева синтаксису. Інша частина реалізації JCoCo становить інтерпретатор байт-коду.

Рис. 7.1 JCoCo


CASM
Source
Program
Scanner
Parser
AST
Bytecode
Interpreter
I/O Devices

Інтерпретатор байт-коду обчислює дерево абстрактного синтаксису (тобто AST), яке складається з об'єктів функції та класу. AST інтерпретується в контексті об'єктів кадру. Фрейм - це один запис активації у стеку часу виконання виконуваної програми. Коли програма починається, інтерпретатор починає виконувати основну функцію, створюючи для неї фрейм і починаючи виконання з першої інструкції.

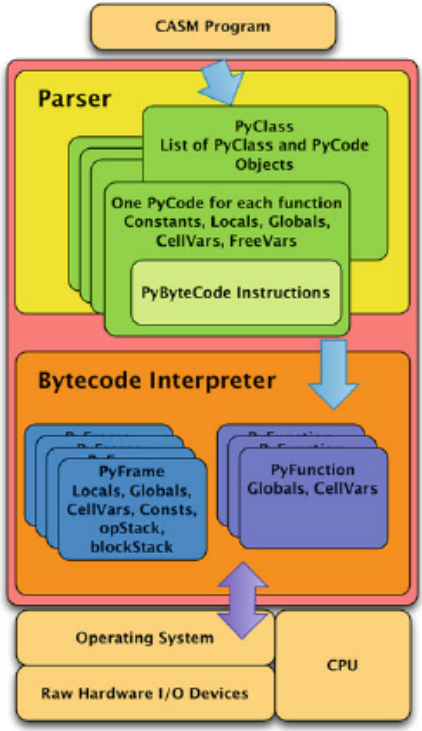
Приклади Java та C ++, використані в цих лекціях, походять із реалізації класів сканера, синтаксичного аналізатора, функції, класу та фрейму, а також класів для типів та екземплярів типів, таких як цілі числа, плаваючі, рядки та списки.

JCoCo написано на Java. Подібний проект, CoCo, раніше був написаний на C ++. JCoCo - це вдосконалена, більш повно розроблена версія CoCo, переписана на Java. JCoCo - це великий проект, що складається з 56 вихідних файлів та приблизно 8900 рядків коду. Не лякайтесь, багато коду повторюється. Для такої великої програми правильне її структурування є надзвичайно важливим. Віртуальна машина JCoCo є інтерпретатором інструкцій байт-коду, приблизно як віртуальна машина Java (тобто JVM). Інтерпретатор JCoCo читає вихідні файли мови збірки, які називаються файлами CASM, як ви дізналися про це в останніх лекціях.

Значна частина дизайну CoCo та JCoCo схожа. JCoCo включає підтримку визначених програмістом класів. CoCo не підтримує визначення класів. В іншому випадку дві реалізації дуже схожі. У першій частині цього розділу будуть представлені приклади реалізації Java та C ++, коли це доречно для порівняння і протиставляють дві мови. Пізніше в лекціях будуть розглянуті деталі реалізації Java JcoCo.

Як і інші інтерпретатори, реалізація JCoCo / CoCo розділена на деякі логічні компоненти: сканер, синтаксичний аналізатор та інтерпретатор байт-коду. Сканер зчитує символи з вихідного файлу CASM і створює об'єкти, звані маркерами. В останньому розділі було багато прикладів файлів CASM. Токени з файлу CASM, подібного до попередніх лекцій і зображені на рис. 7.2, включають ключове слово Function, двокрапку, основний ідентифікатор, скісну риску, ціле число 0, інше ключове слово Константи, іншу двокрапку, ключове слово None тощо. Ці маркери повертаються аналізатору по черзі, коли парсер запитує інший маркер.

Рис.7.2 Віртуальна машина JCoCo



Синтаксичний аналізатор читає маркери по одному зі сканера і використовує їх під час аналізу вихідного файлу відповідно до граматики для JCoCo, наведеної в Додатку А. Граматика, наведена там, є LL (1), тому синтаксичний аналізатор реалізований як парсер рекурсивного спуску . Кожен нетермінал граматики є функцією в синтаксичному аналізаторі. Права частина правил для кожного нетерміналу визначає тіло кожної функції в синтаксичному аналізаторі. Про це буде далі далі в наших лекціях. Результатом аналізу вихідного файлу є абстрактне дерево синтаксису (AST). Цей AST є внутрішнім представленням програми, яку слід інтерпретувати.

Інтерпретатор байт-коду JCoCo - це частина програми, враховуючи AST, яка інтерпретує вказівки байт-коду кожної функції. Коли інструкції виконуються, віртуальна машина взаємодіє з такими пристроями вводу-виводу, як клавіатура та екран. Інтерпретація байт-коду є відповідальністю кількох частин реалізації JCoCo, як ви прочитаєте далі. Остання частина цього розділу містить детальне пояснення реалізації віртуальної машини JcoCo.

Java Environment

У перших лекціях розповідалося, що Java виникла в рамках проекту Green. Сьогодні Java - це надійна мова, яка містить безліч функцій, які роблять її зручною для програмістів, а також ефективною та потужною. Java насправді складається з двох важливих інструментів: віртуальної машини Java (тобто JVM) та компілятора Java, який компілює з вихідного коду Java у байт-код Java (саме це виконує JVM).

Давайте розглянемо приклад світового привітання, написаний на Java. Програма Java наведена на рис. 7.3. Цю програму потрібно зберегти у файлі **HelloWorld.java**. Програма компілюється і запускається за допомогою таких команд.

```
My Mac> javac HelloWorld.java
My Mac> java HelloWorld
Hello World
My Mac>
```

Програма **`javac`** - це компілятор Java, який перекладає вихідну програму Java, що закінчується розширенням **`.java`**, у файл байт-коду, що закінчується на **`.class`**. Файл байт-коду - це двійковий файл, який читається машиною, але не читається людиною. Файл байт-коду читається віртуальною машиною Java або JVM, яка насправді є програмою під назвою **`java`** або **`java.exe`**, якщо ви працюєте на машині Windows. JVM взаємодіє з операційною системою для виконання файлу байт-коду.

```
public class HelloWorld {
 public static
  void main(String args[]) {
  System.out.println("Hello World");
 }
}
```

Рис.7.3 Хєлов ворд.

Як правило, програма Java складається з багатьох файлів вихідного коду, які компілюються у безліч файлів байт-коду. Під час програмування на Java кожен файл вихідного коду повинен називатися так само, як публічний клас у файлі вихідного коду. Наприклад, код на рис. 7.3 повинен знаходитись у файлі HelloWorld.java, оскільки загальнодоступний клас називається HelloWorld. Кожен файл вихідного коду Java може мати рівно один загальнодоступний клас.

Написання нетривіальної програми Java передбачає створення багатьох класів і, отже, багатьох вихідних файлів. Коли компілюється проект Java, компілятор Java переглядає дати всіх вихідних файлів і всіх файлів байт-коду. Щоразу, коли файл створюється або змінюється, змінюється дата останнього його модифікації. Це інформація, яку підтримує кожна операційна система, включаючи Mac OS X, Microsoft Windows та Linux.

Якщо виявляється, що будь-який файл байтового коду є старшим за відповідний вихідний файл, тоді вихідний файл перекомпілюється, щоб створити новий файл байт-коду з датою, що перевищує вихідний код. Цей механізм використання часових позначок для визначення того, що потрібно перекомпілювати, називається засобом створення з історичних причин, до якого ми переглянемо у наступних лекціях.

C++ Environment

C++ та Java мають багато спільного синтаксису. C++ був розроблений спочатку, а розробка Java приблизно через 10 років. Як згадувалося в перших лекціях, Б'ярн Струструп розробляв C++ на початку вісімдесятих. Він розробив мову таким чином, щоб вона була сумісною із C, тому для нього вже були прийняті деякі рішення, такі як необхідність окремої компіляції та наявність макропроцесора. C++ є однією з найбільш широко використовуваних об'єктно-орієнтованих мов сьогодні і продовжує розвиватися.

Комітет з питань стандартів зараз контролює C ++, регулярно переглядаючи мову, як-от версія C ++ 11, яка вийшла в 2011 році, і версія 2014 року, яка мала незначні зміни щодо версії 2011 року. Чергова версія була офіційно прийнята наприкінці 2017 року.
C ++ 20 - це назва останнього стандарту ISO / IEC мовної програми програмування C ++. Специфікація опублікована в грудні 2020 року.
Комітет за стандартом C ++ почав планувати C ++ 20 липня 2017 року. C ++ 20 є преемником C ++ 17.

Розробка мови C ++ триває.

Використання C / C++ для програмного проекту не обходиться без певних ризиків. Суттєвою проблемою, мабуть, найбільш стійкою проблемою з часом, є програма C / C++ - витік пам'яті. Програмісти C / C++ повинні бути дисциплінованими щодо розподілу та вивільнення пам'яті. Загальноприйняте, що програми, які працюють тривалий час, матимуть витік пам’яті, який слід відстежувати, що є складним завданням. Багато мов збирач сміття дбає про звільнення пам'яті, яка більше не потрібна програмі. Збірщик сміття не може бути безпечно включений до складу програм C та C++. І C, і C++ розроблені, щоб надати програмісту максимальний контроль.

Це означає, що більше відповідальності залишається за програмістом, і в результаті програмісти повинні бути дуже дисциплінованими при використанні C / C ++. Про це в лекції надалі.

C і C ++ мають багато застосувань, включаючи розробку операційної системи, критичне термінове програмне забезпечення та детальний апаратний доступ. Навчившись добре програмувати на C ++, ви знайдете довгий шлях до того, щоб стати чудовим програмістом на будь-якій мові. Цей розділ не навчить вас усьому, що вам потрібно знати, щоб стати програмістом на C ++. Це може бути і є темою багатьох книг.

Ця лекція познайомить вас з багатьма важливими концепціями та навичками, які вам знадобляться, щоб стати хорошим програмістом на C ++.

Як і Java, програми C ++ повинні бути скомпільовані, перш ніж ви зможете їх запускати. Програми Java компілюються в байт-код Java, і байт-код запускається на JVM. Програми C ++ компілюються на машинну мову центрального процесора, який їх буде виконувати. Операційна система комп'ютера, де працює програма C ++, відповідає за завантаження виконуваної програми та підготовку її до роботи, але в іншому випадку скомпільована програма C ++ працює безпосередньо на центральному процесорі машини, для якої вона була складена.

На рисунку 7.5 зображено процес компіляції програм на C ++. Розгляньте рис. 7.4, щоб порівняти це з виконанням програм Java. Середовище C ++ виглядає більш складним. Але все, що є в зеленому полі, насправді виконується за допомогою однієї команди компіляції. Рисунок 7.6 містить класичну програму **`hello world`**, написану на C ++.

Рис. 7.4 Java компілятор і віртуальна машина

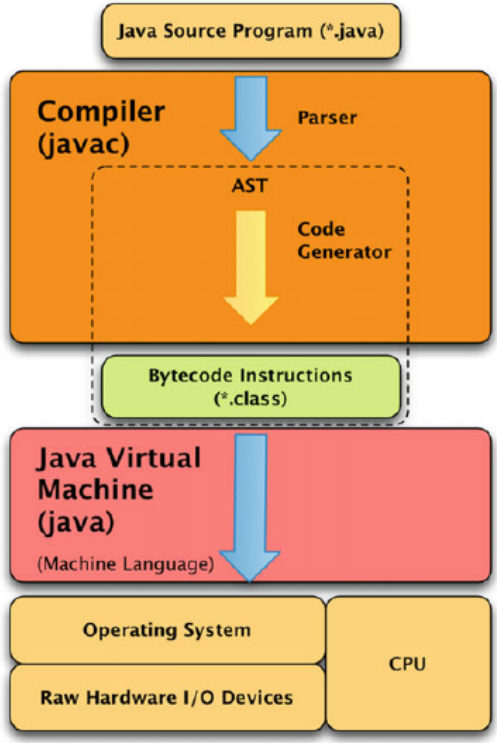


Рис.7.5 С++ компілятор

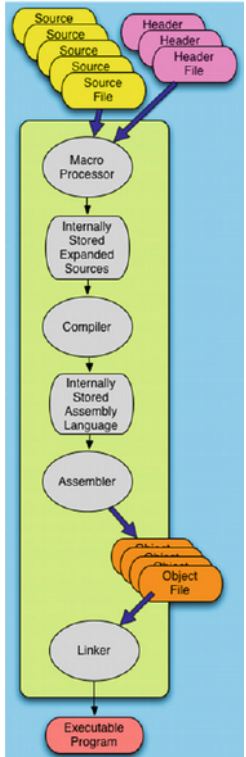


Рис.7.6.
hello.cpp

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  cout << "Hello World!"<< endl;
}
```

Рис.7.7.
Компіляція
hello.cpp

```
My Computer> g++ hello.cpp
My Computer> a.out
Hello World!
My Computer>
```

Рис.7.8.
Включіть
налагодження та ім'я

```
My Computer> g++ -g -o hello hello.cpp
My Computer> hello
Hello World!
My Computer>
```

Для запуску програми **hello world** її потрібно спочатку скомпілювати. На рисунку 7.5 показано процес складання програми на C ++, подібну до тієї, що зображена на рис. 7.6. Спочатку макропроцесор зчитує файл заголовка **iostream** і поєднує його з рештою вихідного файлу. Файл заголовка **iostream** не містить коду потоків. Він просто оголошує потоки та оператори, які використовувались для запису у вихідний потік. Об'єднаний текст програми надсилається компілятору, який аналізує програму та генерує код машинної мови за допомогою асемблера. Потім машинний код мови пов'язаний з бібліотекою iostream для створення виконуваного коду.

На щастя, весь цей процес укладено в одній команді. У наступних лекціях буде докладніше про процесор макросів та потоки вводу-виводу.

Виконання команди g ++ компілює програму, як показано на рис. 7.7. За замовчуванням g ++ створює програму під назвою **a.out**. Для запуску програми, яку ви вводите **a.out**, операційна система завантажить і запустить її. За замовчуванням a.out можна перейменувати або в команді компіляції вказати інше ім'я.

Параметр **`-g`** на рис. 7.8 повідомляє g ++ включати інформацію про налагодження в програму. **`-O`** говорить g ++ назвати виконувану програму привітанням замість **`a.out`**.

Для компіляції програми C ++ у вашій системі повинен бути встановлений компілятор C ++. Компілятор g ++, використаний на рис. 7.7, є компілятором GNU C ++. Цей компілятор доступний для Mac OS X, Linux та Microsoft Windows.

Macro Processor

Перший рядок програми на рис. 7.6 називається директивою макропроцесора. *Макропроцесор* - це частина компілятора C + +, яка відповідає за втягування інших файлів у вихідну програму, а іноді і за просте редагування вихідного файлу, щоб підготувати його до компіляції. У цій програмі макропроцесор включає інший файл або бібліотека з назвою **`iostream`**. Файл **`iostream`** називається файлом заголовка, оскільки він визначає функції та змінні, які існують у якійсь іншій бібліотеці чи коді в системі, де він компілюється.

Заголовкові файли визначають інтерфейси до цих інших бібліотек або коду. Коли файл заголовка укладено в кутові фігурні дужки, менші за / більші, ніж пара, це системний файл заголовка. Більше інформації про макропроцесор та файли заголовків можна знайти далі в лекції.

Make Tool

Файлова система - це програмне забезпечення та формат, який контролює спосіб зберігання файлів на жорсткому диску комп’ютера. Усі операційні системи мають власні файлові системи, а іноді підтримують декілька файлових систем. Microsoft Windows, зокрема, підтримує NTFS та Fat32. Підтримка Linux ext2, ext3, reiserfs та інші. Mac OS X підтримує кілька файлових систем, включаючи HFS +. Кожна з цих файлових систем зберігає атрибути кожного файлу, включаючи дату та час останньої зміни файлу.

Make - це програма, яка може бути використана для компіляції програм, які складаються з модулів та використовують окрему компіляцію. Програми C і C ++ використовують окрему компіляцію, і зазвичай ви пишете файл **make** для компіляції програм, написаних цими мовами, або використовуєте інструмент для автоматичного створення файлу **make**. Програми Java не компілюються через програму **make**, оскільки програма **make** вбудована в компілятор Java, як було згадано в попередніх лекціях.

Ідея проста. Щоразу, коли модуль компілюється компілятором C ++, він створює файл об'єкта. Наприклад, коли компілюється **`PyObject.cpp`**, компілятор C++ пише файл під назвою **`PyObject.o`**. Для кожного з цих файлів у файлі зберігаються дата та час останньої зміни або створення. Після компіляції дата на **`PyObject.cpp`** є старшою за дату на **`PyObject.o`**. Коли програміст змінює **`PyObject.cpp`**, його дата буде новішою, ніж дата **`PyObject.o`**.

Make використовує це просте спостереження разом із правилами make для виконання команд компіляції, необхідних для того, щоб дата **PyObject.o** була новішою, ніж дата **PyObject.cpp**. Ось правило створення для **PyObject.cpp**.

```
PyObject.o: PyObject.cpp PyObect.h
  g++ -g -c -std=c++0x PyObject.cpp
```

Це правило говорить, що **PyObject.o** повинен бути новішим за **PyObject.cpp** та **PyObject.h**. Якщо будь-який із цих двох файлів новіший, то **make** буде виконати команду в наступному рядку, який повинен бути з відступом під першим рядком.

Результатом виконання цієї команди компіляції є створення нового файлу **PyObject.o** з датою новішою, ніж будь-який із двох вихідних файлів.

Щоб зробити *кокос* виконуваним, усі об'єктні файли повинні бути пов'язані між собою. Щоб зв'язати все разом, перше правило пишеться так.

```
coco: main.o PyObject.o PyInt.o PyType.o ....
  g++ -o coco -std=c++0x main.o PyObject.o PyInt.o PyType.o ....
```

Тут мають бути перелічені всі 38 об’єктних файлів. Це говорить про те, що дата у виконуваній програмі **coco** повинна бути новішою за дату у всіх її об’єктних файлах.

Всі ці правила розміщуються у файлі, який називається **Makefile**, в тому ж каталозі, що і вихідні файли C ++. Коли буде викликано **make**, він буде шукати файл із назвою **Makefile**. Відстежуючи дати, будуть перекомпільовані лише вихідні файли, які були оновлені, а виконуваний файл coco буде відтворений шляхом зв'язування всіх об'єктних файлів.
Написати хороший **Makefile** іноді буває складно і майже завжди схильний до помилок, тому часто у **makefile** є правило, яке називається **clean**. Виконання **make clean** видалить усі об'єктні файли, щоб ви могли отримати нову компіляцію.

Існують також такі інструменти, як **autoreconf**, які автоматично генерують **Makefile** лише за допомогою декількох входів. Погляньте на сценарій відновлення в дистрибутиві CoCo, щоб побачити, як це можна використовувати. Щоб використовувати **autoreconf**, у вас повинні бути встановлені інструменти **automake** у вашій системі. Але якщо ви це зробите, ви можете виконати

```
./rebuild
./configure
make
```

побудувати всю віртуальну машину CoCo. Без інструментів **automake** ви все одно зможете виконувати налаштування та виконувати команди для побудови CoCo.

Окрема компіляція в програмах на C ++ означає, що кожен модуль у програмі компілюється окремо. Кожен об'єктний файл, створений компіляцією модуля, створюється незалежно від інших вихідних файлів. Це важливо, оскільки великі проекти на C ++ часто містять сотні вихідних файлів на C ++. Окрема компіляція означає, що потрібно перекомпілювати лише невеликий фрагмент, який програміст змінює, якщо інтерфейс (тобто файл заголовка) до інших модулів не змінюється. Після компіляції вихідних файлів до об'єктних файлів об'єктні файли можуть бути пов'язані між собою, щоб сформувати виконувану програму. Посилання - це дуже швидка операція порівняно із компіляцією.

Простори імен

Рядок 2 програми на рис. 7.6 відкриває стандартний простір імен у програмі. Перші два рядки цієї програми на C ++ схожі на імпорт модуля в Python або пакету на Java. Під час імпортування модуля в Python програміст пише оператор імпорту, як один із цих двох рядків.

```
from iostream import * # merges the namespace with the current module
import iostream # preserves the namespace while importing the module
```

На Java програміст написав би імпортну заяву приблизно так, хоча не зовсім так.

```
import java.iostream.cout
```

Еквівалент простору імен Python - це модуль. Модулі Python можна імпортувати одним із двох способів, зберігаючи простір імен або об’єднуючи його з існуючим простором імен. У Java-пакетах еквівалент простору імен, а вибрані класи та об'єкти можна імпортувати з пакета. Простори імен важливі в C ++, Python та Java, оскільки без них виникало б багато потенційних конфліктів імен між файлами заголовків та модулями, які могли б створювати помилки компіляції та не давати компілювати програми, а у випадку з Python - запускатись, які були б інакше правильні програми. У програмах на C ++, якщо ми не хочемо відкривати простір імен **`std`**, ми можемо переписати програму, як показано на рис. 7.9.

Найбезпечніший спосіб програмування - це не відкривати простори імен або об’єднувати їх між собою. Але це також незручно, оскільки кожне ім’я потрібно писати щоразу. Що правильно для вашої програми, залежить від програми, яку ви пишете.

```
1  #include <iostream>
2  int main(int argc, char* argv[]) {
3    std::cout <<
4       "Hello World!"<< std::endl;
5  }
```

Рис.7.9. Простір імен **std**

Динамічне зв’язування

Динамічне зв’язування пов’язане з просторами імен, модулями та пакетами. Сучасні мови програмування, такі як C ++ та Java, залежать від багатьох бібліотек, тому програмісти можуть вирішувати проблеми, а не переписувати код, який є спільним для декількох програм. Бібліотеки, що містять загальновживаний код, загалом доступні для використання програмами, написаними мовою високого рівня, включаючи програми C ++ та Java. Ці бібліотеки мають бути пов’язані з вашою програмою, щоб мати можливість ними користуватися. На рисунку 7.5 показано об’єктні файли (тобто модулі), пов’язані між собою в програму на C ++.

Проблема із зображенням на рис. 7.5. Програми раннього C могли бути самостійними програмами, які покладались на лише невелику кількість системних дзвінків з операційної системи Unix. Однак сучасні C, C ++, Java та майже будь-яка інша сучасна мова програмування залежать від такої кількості бібліотек, що зв'язування всіх їх між собою може бути проблемою у декількох напрямках.

- Розмір пов'язаної виконуваної програми був би величезним, що займав би багато місця в пам'яті під час її виконання.
- Будь-які зміни в будь-якій бібліотеці потребують повторного зв’язування кожної програми, яка використовує її, щоб отримати нову версію бібліотеки.
- Немає підстав мати кілька копій бібліотек, по одній для кожної програми, яка його використовує. Це витрачає простір на додаток до накладних витрат на управління кількома копіями бібліотек.

Сучасні мови не мають статичного зв’язку з усіма бібліотеками, які потрібні програмі. Вони динамічно пов’язують їх. Коли ви чуєте слово динамічний, вам слід подумати про час виконання. Бібліотеки, як правило, пов’язані під час роботи. Програмне забезпечення, часто входить до складу операційної системи, виявляє, коли бібліотека буде використана програмою, завантажує її в пам'ять і пов'язує з програмою, яка запитує її послуги під час виконання програми. Microsoft Windows називає ці динамічно пов'язані бібліотеки бібліотеками DLL. Windows включає служби, які дозволяють писати бібліотеки, щоб вони могли динамічно зв’язуватися з програмами під час їх виконання.

Mac OS X і Linux також мають можливість динамічно пов'язувати бібліотеки. Програми C ++ часто динамічно пов'язують з бібліотеками, які надаються бібліотеками часу виконання C ++ та іншими бібліотеками, які можуть знадобитися програмі, але надані разом із програмою.

Програми Java також використовують динамічне зв’язування. Насправді динамічне зв’язування вбудовано в основи Java. JVM завантажує файли байт-коду (тобто модулі) за необхідності у вашій програмі Java. Програми Java складаються з файлів .class, які називаються файлами байт-коду, які повинні знаходитися в поточному робочому каталозі або в каталозі на шляху до класу. Шлях до класу - це список каталогів або папок, де JVM шукає файли байт-кодів. Шлях до класу записується у змінну середовища під назвою **CLASSPATH**. Ось один із прикладів шляху до класу.

```
export CLASSPATH=./DBBrowser/lib/mysql-connector-
java-5.1.17-bin.jar:.:$CLASSPATH
```

Шлях до класу - це список папок або каталогів, де можна знайти ці динамічно завантажені файли **.class**. Іноді для реалізації якоїсь бібліотеки пишеться ціла група класів. Наприклад, цей шлях до класу включає **mysql-connector-java-5.1.17-bin.jar**. Це фактично те, що називається файлом **JAR**. Файл **JAR** розшифровується як Java Archive і являє собою набір файлів **.class**, що зберігаються у стислій формі. Динамічно пов'язані бібліотеки настільки загальні для програм Java, що файли **JAR** були додані як засіб для зручного групування та перерозподілу колекцій класів для програм Java.

Файли байт-коду, знайдені у файлі **JAR**, організовані в пакети. Імпортуємо щось на зразок

```
import java.io.BufferedInputStream
```

в програмі Java призведе до динамічного зв'язку класу **BufferedInputStream** із пакета **java.io**, який є бібліотекою, що надається середовищем виконання Java.

Визначення головної функції

Рядки 3–5 програми **`hello world`** на рис. 7.6 визначають основну функцію. Кожна програма C ++ повинна мати одну основну функцію і лише одну. Основна функція повинна повертати ціле число, і їй дається ціле число та масив символьних масивів, які є аргументами командного рядка. Аргументи командного рядка детально розроблені в розділі про масиви та покажчики далі в цій лекції.

Кожна програма Java також повинна мати основну функцію.

Однак під час запуску програми повинен бути вказаний клас, основну функцію якого ви хочете запустити. Таким чином, кожен клас потенційно може мати головну функцію. Основний із зазначеного класу буде запущений першим. Основну функцію програми Java **`hello world`** можна знайти на рис. 7.3.

I/O Streams

Рядок 4 програми на рис. 7.6 друкує Hello World на екран. Якщо бути трохи точнішим, cout представляє те, що називається потоком в C++. Ви можете уявити потік C++ як справжній потік, у якому знаходиться вода. Ви можете помістити речі в потік, і вони будуть нестись за течією. Щоб помістити щось у потік C++, ви використовуєте оператор **<<**. Пишемо

```
cout << "Hello World";
```

поміщає рядок “**Hello World**” у потік **cout**. Цей вираз повертає потік **cout**. Це означає, що кілька операторів **<<** можуть бути зв'язані ланцюгом. Рядок 4 - це як писати

```
(cout << "Hello World") << endl;
```

У цьому прикладі дужки не потрібні, оскільки **<<** вже асоціативно ліворуч. Але вони були включені, так що ви можете бачити, що виклик функції **<<** повертає потік, який можна використовувати в наступному **<<** операторі праворуч.

Існує три потоки, автоматично пов’язані з програмами. Ці три потоки пов'язані з будь-якою програмою, будь то C ++, Python, Java чи інша мова. У C ++ перший потік називається **`cout`**, і за замовчуванням він пише на екран. За замовчуванням потік **`cerr`** також записує на екран. За замовчуванням потік **`cin`** читає з клавіатури. Оператором зчитування з потоку є оператор зсуву вправо, записаний змінною **`cin >>`**, де змінна буде містити значення свого типу, яке було прочитано з потоку. У кожному з цих випадків ці потоки можуть бути перенаправлені на читання або запис у різні місця. Перенаправлення вводу та виводу - це функція операційної системи, яка насправді не пов’язана з певною мовою програмування.

Ви можете шукати в Інтернеті інформацію про переспрямування стандартного виводу, стандартну помилку або стандартне введення, якщо вам цікаво дізнатись більше про переспрямування.

Програми Java мають однакові три потоки. **`System.out`** - це назва стандартного вихідного потоку. **`System.err`** - це стандартний потік помилок. **`System.in`** - це вхідний потік. Рисунок 7.3 демонструє запис у стандартний вивід у програмі Java. Оператори зсуву праворуч і зсуву вліво не використовуються для читання та запису в потоки в Java. Натомість використовується більш традиційний синтаксис виклику функції.

Мусорозбірник

Збір сміття відбувається, коли динамічно виділений простір потрібно повернути до пулу доступного місця в пам'яті. Цей простір, доступний для розподілу під час виконання, називається купою. Кожного разу, коли створюється об'єкт, трохи пам'яті пам'яті комп'ютера має бути зарезервовано для зберігання інформації про стан цього об'єкта. Коли об’єкт більше не потрібен, слід звільнити простір у купі, щоб пізніше він міг використовувати інший об’єкт.

Такі мови, як Java та Python, забезпечують збір сміття як частину базової моделі обчислень. Вони можуть це зробити, оскільки ці мови обережно ставляться до впливу покажчиків на програміста. Насправді покажчики називаються посиланнями на цих мовах, щоб відрізнити їх від покажчиків на таких мовах, як C та C ++. Компромісом є те, що ці мови беруть певний контроль над програмістом. Java, Python та багато мов, що забезпечують збір сміття, вимагають віртуальної машини для виконання своїх програм, а віртуальна машина піклується про управління та звільнення невикористаної пам'яті.

Вивіз сміття може вплинути на продуктивність системи під час роботи. Такі мови, як Java та Python, не так підходять для програм реального часу, де час є критичним. У цих мовах збір сміття може відбуватися в будь-який час. Зазвичай запуск програми не є критичним за часом, а час, необхідний для збору сміття, незначний. Переваги збору сміття, як правило, значно перевищують можливість витоків пам'яті, але не в термінах критичних програм.

Існування системи виконання, яка підтримує збір сміття, як віртуальні машини Java та Python, означає, що ці програми мають менший доступ до базового обладнання машини. Щоб безпечно звільнити невикористовувану пам’ять, будь-яка система збору сміття повинна обмежити використання покажчиків у програмах, і в результаті програми, написані такими мовами, як Java та Python, мають менший доступ до деталей апаратної платформи. Знову ж таки, це, як правило, не є проблемою для більшості програм, але є випадки, коли прямий доступ до обладнання важливий.

Такі програми, як операційні системи, як правило, не написані на Python або Java. Щоб уникнути помилок, програми для Android написані на Java, але сама операційна система Android базується на ядрі Linux, яке реалізовано в C.

Програми на C ++ повинні керувати розподілом та звільненням купи. Але не завжди ясно, коли об’єкт більше не використовуватиметься. Витік пам'яті відбувається, коли пам'ять ніколи не звільняється, навіть якщо програма C ++ виконується за її допомогою. Існує додаткова робота, пов’язана з написанням класів C ++, щоб забезпечити звільнення об’єктів, коли вони більше не потрібні. У випадку з віртуальною машиною CoCo безпечно звільняти об’єкти після їх створення, оскільки в CoCo немає підрахунку посилань, щоб вирішити, коли об’єкт більше не використовується.

Оскільки об’єкти створюються і на них часто посилаються з декількох частин програми CASM, безпечно просто звільняти об’єкти в CoCo. Справжній збір сміття необхідний у реалізації CoCo на C ++, щоб зробити це справді корисна віртуальна машина. Поки CoCo працює для запуску коротких програм, але не підходить для тривалих програм.

Для програм Java збирач сміття - це потік, який запускається час від часу і перевіряє, скільки посилань все ще посилаються на об’єкт. Якщо в програмі немає частин, що використовують об'єкт, її можна звільнити. Часом у вас може з’явитися група об’єктів, які не використовуються, але всі, схоже, використовують один одного. У цьому випадку збирач сміття може сформувати графік залежностей і з’ясувати, що об’єкти, що беруть участь, утворюють цикл, і жодні інші об’єкти поза циклом не залежать від групи об’єктів у циклі. У цьому випадку всі об’єкти в циклі можуть бути звільнені. Існування збирача сміття значно спрощує написання програм Java, включаючи JCoCo.

Потоки

У попередньому розділі згадувалося, що збирач сміття JVM працює в іншому потоці. Потік - це запущена послідовність вказівок, яка ділить доступ до об’єктів з іншими потоками. Кожен потік працює здебільшого незалежно від інших потоків у тій же програмі. Ви можете розглядати кожен потік як по суті незалежну програму, яка працює з іншими потоками в тій самій більшій програмі для виконання певної роботи.

Java була побудована з нуля, щоб бути багатопотоковою мовою програмування. Кожен об'єкт у Java містить блокування, яке можна використовувати для синхронізації поведінки декількох потоків. Коли працює більше одного потоку, його робота не повинна скасовувати або змінювати роботу, яку виконує інший потік. Коли працює більше одного потоку, потрібно вирішити дві проблеми: синхронізація потоків та зв’язок між потоками.

Замки на об'єктах дозволяють потокам Java одночасно синхронізувати свою роботу та структуровано спілкуватися між собою. З кожним об'єктом у Java пов'язаний замок. У Java також є такі ключові слова, як синхронізовані, які забезпечують це на об'єкті може працювати лише один метод одночасно. Цей текст не навчить вас про створення потоків Java, але він є важливою темою і повинен бути вивчений колись.

C ++ також має підтримку потоків через клас потоків у стандартному просторі імен. Однак підтримка потоків на C ++ дещо відрізняється від підтримки Java. Наприклад, C ++ не має ключових слів, які дозволяють синхронізовані методи, такі як Java. Потоки в C ++ вимагають трохи більше роздумів і роботи. C ++ та Java однаково потужно підтримують багатопотокові програми, але, якщо вибрати вибір, Java є мовою для багатопотокових програм.

На наступній лекції продовжимо розгляд імплементації об’єкто-орієнтованого програмування.