Метапрограмування

Мультипарадигмовий підхід в інженерії програмного забезпечення

Київ, 2015 — 2022

Аннотація

Усі програми - це дані. Одні дані інтерпретуються, як значення, а інші - як типи цих значень, треті - як інструкції з обробки перших двох. Будь- які парадигми та техніки програмування - це лише спосіб формувати метадані, які дають правила і послідовність потоку обробки інших даних. Мультипарадигменне програмування бере краще з всіх парадигм і будує з них синтаксичні конструкції, що дозволяють більш зрозуміло та зручно описати предметну область. Ми зв'язуємо високорівневі DSL (доменні мови) із низькорівневими машинними інструкціями через безліч шарів абстракцій. Тут важливо не фанатично слідувати одній парадигмі, а найефективніше відображати завдання для виконання на машинному рівні. Найефективніше - це з меньшою кількістю шарів та залежностей, найбільш зручно для розуміння людиною, для підтримки та модифікації, забезпечення надійності та тестованості коду, розширюваності, перевикористання, зрозумілості та гнучкості конструкцій метаданих на кожному рівні. Ми вважаємо, що такий підхід дозволить отримувати як швидкі перші результати в розробці кожного завдання, так і не втрачати темпів при більшому потоку змін на етапах, коли проект вже достиг високої зрілості та складності. Ми постараємося розглянути прийоми та принципи програмування з різних парадигм через призму метапрограмування та не стільки змінити цим програмну інженерію, як її осмислення новими поколіннями інженерів.

Зміст

1. Вступ

Постійне переосмислення своєї діяльності, навіть найпростішої, має супроводжувати інженера все його життя. Звичка записувати свої думки словами та відточувати формулювання дуже допомагає у цьому. Текст цей з'явився як мої уривчасті нотатки, написані в різні роки, які я накопичував і критично вичитував десятки разів. Часто, я не погоджувався з собою, перечитуючи уривок після того, як він полежить деякий час. Тому я доводив текст до того, поки сам не погоджувався з написаним після тривалих періодів витримки матеріалу. Своїм завданням я прийняв писати якомога коротше і неодноразово переписував великі фрагменти, знаходячи, що їх можна виразити коротше. Це дозволило брати участь у формуванні книги всім охочим із спільноти Метархія, швидко знаходити помилки та неточності завдяки читачам, а також багатьом просто зручніше сприймати у вигляді книги. Актуальну версію завжди можна знайти на https:// github.com/ HowProgrammingWorks/ Book, вона доповнюватиметься постійно. Прошу надсилати запити на виправлення та доповнення до [issues] (https:// github.com/ HowProgrammingWorks/ Book/ issues) англійською мовою, нові ідеї у [discussions] (https:// github.com/ HowProgrammingWorks/ Book/ discussions) будь- якою мовою, а свої доповнення та виправлення оформляти у вигляді pull-request у репозиторій книги.

Програмування – це мистецтво та інженерія розв'язання задач за допомогою обчислювальної техніки. Інженерія, тому що воно покликане отримувати користь із знань, а мистецтво, тому що знаннями програмування на сучасному етапі розвитку, на жаль, не обмежується і змушене вдаватися до інтуїції та слабко осмисленого особистого досвіду. Завдання програміста не у знаходженні математично вірного рішення, а у відшуканні узагальненого механізму рішення, здатного нас приводити до знаходження прийнятного рішення за обмежений час у якомога більшому класі завдань. Іншими словами, у знаходженні абстрактного класу рішень. Не всі парадигми програмування припускають рішення покрокове, але фізична реалізація обчислювальної техніки та природа людського мислення передбачають покроковість. Складність у цьому, що ці дії далеко не завжди зводяться до машинних операцій і залучають зовнішню взаємодію з пристроями введення/ виводу і датчиками, а через них, із зовнішнім світом і людиною. Ця обставина створює велику невизначеність, яка не дозволяє математично суворо довести правильність способу

вирішення всіх завдань і, тим більше, суворо вивести таке рішення з аксіом, як це характерно для точних наук. Однак, окремі алгоритми можуть бути виведені аналітично, якщо вони зводяться до чистих функцій. Тобто, до функцій, які у будь-який момент для певного набору вхідних даних однозначно дають однаковий результат. Чиста функція не має історії (пам'яті або стану) і не звертається до зовнішніх пристроїв (які можуть мати такий стан), може звертатися тільки до інших чистих функцій. Від математики програмування успадкувало можливість шукати точні рішення аналітично, та й сама обчислювальна машина функціонує строго в рамках формального математичного апарату. Але процес написання програмного коду не завжди може бути зведений до формальних процедур, ми змушені приймати рішення в умовах великої невизначеності та конструювати програми інженерно. Програміст обмежений і часом розробки програми, тому ми зменшуємо невизначеність завдяки введенню конструктивних обмежень, які не є суворо виведеними із завдання і засновані на інтуїції та досвіді конкретного спеціаліста. Простіше кажучи, через відсутність оптимального алгоритму, програміст може вирішувати завдання будь-яким способом, який дає прийнятні результати за розумний час, і який може бути реалізований за такий час, поки завдання ще залишається актуальним. У таких умовах ми повинні брати до уваги не тільки міру наближення рішення до оптимального, а й знання програміста, володіння інструментарієм та інші наявні ресурси. Адже навіть доступ до знань вже готових програмних рішень обмежений авторським правом, правами володіння вихідним кодом та документацією, що відповідають ліцензійними обмеженнями, не тільки на програмні продукти, а й на книги, відео, статті, навчальні матеріали тощо. Все це суттєво ускладнює та уповільнює розвиток галузі, але згодом доступність знань незворотно зростає, вони просочуються у вільне ходіння в мережі через популяризаторів, ентузіастів та рух вільного програмного забезпечення.

1.1. Підходи до вивчення програмування

Багато хто думає, що головне уміння програміста — це писати код. Насправді програмісти частіше читають код і виправляють його. А основні критерії якості, коду - зрозумілість, читабельність та простота.

Головне уміння програміста - це читання та виправлення

коду.

Кожна тема містить приклади хорошого коду та поганого коду. Ці приклади зібрані з практики програмування та ревью проектів. Спеціально заготовлені приклади поганого коду будуть працездатні, але сповнені антипаттернів та проблем, які потрібно виявити та виправити. Навіть сама перша практична робота в курсі буде пов'язана з виправленням коду, підвищення його читабельності. Якщо давати традиційні завдання (написати функцію по сигнатурі, алгоритм, клас), то початківець, очевидно, реалізує його не найкращим чином, але захищатиме свій код, бо це перше, що він написав. А якщо завдання буде "взяти приклад чужого поганого коду, знайти проблеми та виправити", не переписати з нуля, а покращити за кілька кроків, фіксуючи та усвідомлюючи ці кроки, то включається критичний підхід.

Виправлення поганого коду - один із найефективніших способів навчання.

Початківець одержує приклади рев'ю коду і за аналогією прагне виправити своє завдання. Такі ітерації повторюються неодноразово, не втрачаючи критичного настрою. Дуже добре, якщо буде наставник, який спостерігає за покращеннями, і може коригувати та підказувати. Але наставник у жодному разі не повинен виконувати роботу за новачка, а скоріше наштовхувати його на те, як треба думати про програмування і де шукати рішення.

Наставник – незамінний на будь-якому етапі професійного зростання.

Далі будуть завдання по написанню свого коду. Дуже рекомендуємо початківцям обмінюватися між собою цими рішеннями для перехресного ревью. Звичайно, перед цим потрібно застосувати лінтери та форматери коду, які проаналізують синтаксис, знаходячи в ньому помилки, і виявлять проблемні місця за великою кількістю шаблонів коду. Потрібно домогтися того, щоб колега розумів виражену тобою думку, а не витрачав час на синтаксис і форматування.

Застосовуйте дружнє ревью коду, перехресне ревью, лінтери та форматери.

Переходимо до вправ на зниження зв'язуваності між кількома абстракціями, потім між модулями, тобто зробити так, щоб потрібно було якнайменше знати про структури даних однієї частини програми з іншої її частини. Зниження мовного фанатизму досягається паралельним вивченням з самого початку кількох мов програмування та перекладами з однієї мови на іншу. Між **JavaScript** та **Python** перекладати дуже просто, а **C** складніше буде, але ці три мови, які б вони не були, не можна не включити в курс.

З перших кроків не допускайте жодного фанатизму: мовного, фреймворкового, парадигменного.

Зниження фреймворкового фанатизму — заборона для початківців використовувати бібліотеки та фреймворки, і зосередитись на максимально нативному коді без залежностей. Зниження парадигмального фанатизму - намагатися комбінувати процедурне, функціональне, ООП, реактивне та автоматне програмування. Ми спробуємо показати, як ці комбінації дозволяють спростити патерни та принципи з GoF та SOLID.

Наступна важлива частина курсу – вивчення антипаттернів та рефакторингу. Спочатку ми дамо огляд, а потім практикуватимемося на реальних прикладах коду з живих проектів.

1.2. Приклади на мовах JavaScript, Python та C

No translation

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

```
#include <stdio.h>

int main() {
```

```
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

```
first_num = 2;
secord_num = 3;
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

1.3. Моделювання: абстракції та повторне використання

В основі будь- якого програмування лежить моделювання, тобто створення моделі розв'язання задачі або моделі об'єктів і процесів у пам'яті машини. Мови програмування надають синтаксиси для конструювання обмежень під час створення моделей. Будь- яка конструкція та структура, покликана розширити функціональність та введена в модель, призводить до додаткових обмежень. Підвищення рівня абстракції, навпаки, може знімати частину обмежень і зменшувати складність моделі і коду програми, що виражає цю модель. Ми постійно балансуємо між розширенням функцій і згортанням їх у більш узагальнену модель. Цей процес може та повинен бути багаторазово ітеративним.

Дивно, але людина здатна успішно вирішувати завдання, складність яких перевищує можливості її пам'яті та мислення, за допомогою побудови моделей та абстракцій. Точність цих моделей визначає їх користь для прийняття рішень та вироблення керуючих впливів. Модель завжди не точна і відображає лише малу частину реальності, одну чи кілька її сторін чи аспектів. Однак, в обмежених умовах використання, модель може бути невідмінною від реального об'єкта предметної області. Є фізичні, математичні, імітаційні та інші моделі, але нас цікавитимуть насамперед інформаційні та алгоритмічні моделі.

Абстракція, це спосіб узагальнення, що зводить безліч різних, але схожих між собою випадків до однієї моделі. Нас цікавлять абстракції даних та абстрактні алгоритми. Найпростіші приклади

абстракції в алгоритмах, це цикли (ітераційне узагальнення) та функції (процедури та підпрограми). За допомогою циклу ми можемо описати багато ітерацій одним блоком команд, припускаючи його повторюваність кілька разів, з різними значеннями змінних. Функції також повторюються багато разів з різними аргументами. Приклади абстракції даних, це масиви, асоціативні масиви, списки, множини тощо. У застосунках абстракції потрібно поєднувати в рівні – шари абстракцій. Низькорівневі абстракції вбудовані у мову програмування (змінні, функції, масиви, події). Абстракції більш високого рівня містяться у програмних платформах, рантаймах, стандартних бібліотеках, та зовнішніх бібліотеках або їх можна побудувати самостійно із простих абстракцій. Абстракції так називаються тому, що вирішують абстрактні узагальнені завдання загального призначення, не пов'язані з предметною областю.

Побудова шарів абстракцій це чи не найважливіша задача програмування від успішного вирішення якої залежать такі характеристики програмного рішення, як гнучкість налаштування, простота модифікації, здатність до інтеграції з іншими системами та період життя рішення. Всі шари, які не прив'язані до предметної області та конкретних прикладних завдань, ми називатимемо системними. Над системними шарами програміст надбудовує прикладні шари, абстракція яких навпаки знижується, універсальність зменшується та конкретизується застосування, прив'язуючись до конкретних завдань.

Абстракції різних рівнів можуть бути як в одному адресному просторі (одному процесі або одному застосунку), так і в різних. Відокремити їх один від одного і здійснити взаємодію між ними можна за допомогою програмних інтерфейсів, модульності, компонентного підходу і просто зусиллям волі, уникаючи прямих викликів із середини одного програмного компонента в середину іншого, якщо мова програмування або платформа не дбають про запобігання такій можливості. Так слід чинити навіть усередині одного процесу, де можна було б звертатися до будь-яких функцій, компонентів та модулів з будь-яких інших, навіть якщо вони логічно відносяться до різних шарів. Причина цього в необхідності знизити пов'язаність шарів та програмних компонентів, забезпечивши їх взаємозамінність, повторне використання та роблячи можливою їх роздільну розробку. Одночасно потрібно підвищувати зв'язність усередині шарів, компонентів та модулів, що забезпечує зростання продуктивності коду, простоту його читання, розуміння та

модифікації. Якщо ж нам вдасться уникати зв'язаності між різними рівнями абстракцій, і за допомогою декомпозиції домогтися того, щоб один модуль завжди міг бути повністю охоплений увагою одного інженера, процес розробки стає масштабованим, керованим і більш передбачуваним. Подібна ідея покладена в основу архітектури мікросервісів, але більш загальний принцип застосовується для будь-яких систем, і не важливо, чи це будуть незалежно запущені мікросервіси або модулі, запущені в одному процесі.

Слід зазначити, що чим краще система розподілена, тим краще вона централізована. Тому, як вирішення завдань у таких системах знаходяться на адекватних рівнях, де вже достатньо інформації для прийняття рішень, обробки та отримання результату, відсутня жорстка зв'язаність моделей різного рівня абстракції. При такому підході немає зайвих ескалацій завдання на верхні рівні, уникаються “перегріви” вузлів прийняття рішень, мінімізована передача даних і підвищена оперативна швидкодія.

1.4. Алгоритм, программа, синтаксис, мова

No translation

1.5. Декомпозиція та поділ відповідальності

No translation

1.6. Огляд спеціальності інженер-програміст

Навколо програмування, як і навколо будь-якої сфери своєї діяльності, людина встигла побудувати величезну кількість забобонів і омани. Найперше джерело проблеми, це термінологія, адже різні парадигми, мови та екосистеми насаджують свою термінологію, яка не тільки суперечить між собою, але і нелогічна навіть всередині окремої спільноти. Більше того, багато програмістів самоучки і одинаки, видумують самобутню, ні на що не схожу термінологію і концепції, дублюючи один одного. Ошаленілі маркетологи теж деструктивно впливають на IT галузь у цілому, на формування світогляду і термінології. Викручуючи очевидні речі, заплутуючи та ускладнюючи концепції, вони забезпечують невичерпну лавину проблем, на якій тільки і підтримується весь

софтверний бізнес. Переманюючи користувачі і програмістів на свої технології, гіганти промисловості часто створюють дуже привабливі і правдоподібні концепції, що ведуть в підсумку до несумісності, війни стандартів і до явної залежності від постачальника програмної платформи. Угруповання, що захоплюють увагу людей, десятиліттями паразитують на їх увазі та бюджетах. Ведені гординею та марнославством, деякі розробники й самі розповсюджують сумнівні, а іноді й свідомо тупикові ідеї. Адже робити програмне забезпечення добре – зовсім не вигідно для виробника. Ситуація значно краща у сфері вільного ПЗ та відкритого коду, але децентралізовані ентузіасти надто роз'єднані, щоб ефективно протидіяти потужній пропаганді гігантів галузі.

Як тільки якась технологія або екосистема розвивається достатньо, щоб на ній можна було створювати хороші рішення, вона обов'язково застаріває або виробник припиняє її підтримку або стає занадто складною. На моїй пам'яті вже змінилося понад п'ять таких технологічних екосистем.

1.7. Огляд парадигм програмування

Математика розглядає програму, як функцію, яку можна декомпозувати (розділити) на більш прості функції так, щоб програма- функція була їх суперпозицією. Тобто, грубо кажучи, програма є складною формулою, перетворювачем даних, коли на вхід подаються умови завдання, а на виході ми отримуємо рішення. Не всякий програміст знайомий із цією точкою зору, хоч вона й не ідеальна, але корисна для переосмислення своєї діяльності. Більш поширена протилежна точка зору, яку легко одержати з практики програмування. Полягає вона в написанні програм виходячи з уявлення користувача, з малюнків екранів інтерфейсу користувача, і з інструментарію, мови, платформи і бібліотек. В результаті ми отримуємо не програму- функцію, а велику систему станів, в якій відбувається комбінаторний вибух переходів і поведінка якої непередбачувана навіть для автора, не те що для користувача. Але не можна відразу відкидати цей, здавалося б, жахливий підхід. У ньому є конструктивне зерно, і полягає воно в тому, що не всі програми можна за короткий термін реалізувати у функціональній парадигмі, як перетворювачі даних. Тим більше, що людська діяльність вся складається з кроків і зміни станів навколишніх предметів за принципом покрокових маніпуляцій ними, а уявити її у вигляді функцій було б досить неприродним для нашого мислення.

Набір базових ідей, використовуваних інженером для побудови моделей, абстракцій та програмних систем, називається парадигмою. У цьому розділі ми розглянемо деякі з них поверхнево, а далі у книзі буде спеціальний розділ з більш детальним обговоренням кожної парадигми. Існують мови, які підтримують одну парадигму, а є мультипарадигмові мови. Ми звернемо увагу на різні мови та відмінності реалізації парадигм у них.

Для людини природно уявлення про будь-яку дію, як про набір кроків або алгоритм, це імперативний підхід. Ці кроки можуть бути або лінійними або прийняттям рішення про перехід до іншого кроку плану, замість того, щоб виконувати дії послідовно. Прийняття рішення для машини це операція, порівняння, що веде до розгалуження алгоритму, що дає варіанти (зазвичай два). Події можна умовно поділити на внутрішні та зовнішні. У внутрішніх беруть участь лише процесор та пам'ять, дія виконується відразу, без очікування, і має певний результат, який доступний безпосередньо на наступному етапі алгоритму. Зовнішні дії - це звернення до зовнішніх пристроїв введення-виводу (мережа, диски, інші пристрої), і вони вимагають очікування реакції від пристрою, яка прийде за час, зазвичай невідомий заздалегідь. Ми надсилаємо керуючий сигнал периферійним пристроям про те, що їм потрібно щось зробити і передаємо їм потрібні для цього дані. Далі у нас знову є два варіанти, або чекати результату, і це буде називатися блокуючим режимом вводу-виведення або перейти до наступного кроку алгоритму, не чекаючи результату, і це буде неблокуючий ввід-вивід. Такий поділ спричинено значною різницею у тривалості внутрішніх та зовнішніх дій. Більшість зовнішніх дій пов'язані з фізичними операціями над зовнішнім середовищем. Наприклад, передача даних через бездротову або провідну мережу, запис або читання з фізичного носія, взаємодія з датчиком, реле або приводом. Такі операції часто мають не цифрову, а аналогову природу, тому потрібно додаткове перетворення даних, очікування перехідного процесу, очікування потрібного показання датчика або сигналу від пристрою і т.д. Пристрої введення-виводу часто мають свій контролер, в якому виконується окремий потік операцій, а взаємодія між центральним процесором та пристроями введення-виводу також вимагає узгодження, що займає час і може закінчитися невдало.

2. Базові концепти

No translation

```
// Single-line comment
```

```
/*
  Multi-line
  comments
*/
```

```
# Single-line comment
```

```
"""
  Multi-line
  comments
"""
```

```
select name from PERSON -- comments in sql
```

```
<!-- commented block in xml and html -->
```

```
; Single-line comment in Assembler and LISP
```

2.1. Значення, ідентифікатор, змінна та константа, літерал, присвоєння

No translation

2.2. Типи даних, скалярні, посилання та структурні типи

No translation

2.3. Контекст та лексичне оточення

No translation

2.4. Оператор та вираз, блок коду, функція, цикл, умова

No translation

2.5. Процедурна парадигма, виклик, стек та куча

No translation

2.6. Функція вищого порядку, чиста функція, побічні ефекти

No translation

2.7. Замикання, функції зворотного виклику, обгортки та події

No translation

2.8. Винятки та обробка помилок

No translation

2.9. Мономорфний код у динамічних мовах

No translation

3. Стан застосунку, структури даних та колекції

No translation

3.1. Підходи до роботи зі станом: stateful and stateless

No translation

3.2. Структури та записи

```
#include <stdio.h>

struct date {
  int day;
  int month;
  int year;
};

struct person {
  char *name;
  char *city;
  struct date born;
};

int main() {
  struct person p1;
  p1.name = "Marcus";
  p1.city = "Roma";
  p1.born.day = 26;
  p1.born.month = 4;
  p1.born.year = 121;

  printf(
    "Name: %s\nCity: %s\nBorn: %d-%d-%d\n",
    p1.name, p1.city,
    p1.born.year, p1.born.month, p1.born.day
  );
```

```
  return 0;
}
```

Pascal

```
program Example;

type TDate = record
  Year: integer;
  Month: 1..12;
  Day: 1..31;
end;

type TPerson = record
  Name: string[10];
  City: string[10];
  Born: TDate;
end;

var
  P1: TPerson;
  FPerson: File of TPerson;

begin
  P1.Name := 'Marcus';
  P1.City := 'Roma';
  P1.Born.Day := 26;
  P1.Born.Month := 4;
  P1.Born.Year := 121;
  WriteLn('Name: ', P1.Name);
  WriteLn('City: ', P1.City);
  WriteLn(
    'Born: ',
    P1.Born.Year, '-',
    P1.Born.Month, '-',
    P1.Born.Day
  );
  Assign(FPerson, './record.dat');
```

```
  Rewrite(FPerson);
  Write(FPerson, P1);
  Close(FPerson);
end.
```

Rust

```
struct Date {
    year: u32,
    month: u32,
    day: u32,
}

struct Person {
    name: String,
    city: String,
    born: Date,
}

fn main() {
    let p1 = Person {
        name: String::from("Marcus"),
        city: String::from("Roma"),
        born: Date {
            day: 26,
            month: 4,
            year: 121,
        },
    };

    println!(
        "Name: {}\nCity: {}\nBorn: {}-{}-{}\n",
        p1.name, p1.city,
        p1.born.year, p1.born.month, p1.born.day
    );
}
```

TypeScript: Interfaces

```
interface IDate {
  day: number;
  month: number;
  year: number;
}

interface IPerson {
  name: string;
  city: string;
  born: IDate;
}
```

```
const personToString = (person: IPerson): string => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
  return fields.join('\n');
};
```

```
const person: IPerson = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

TypeScript: Classes

```
class DateStruct {
  day: number;
  month: number;
  year: number;
}

class Person {
  name: string;
  city: string;
  born: DateStruct;
}
```

JavaScript: Classes

```
class DateStruct {
  constructor(year, month, day) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}

class Person {
  constructor(name, city, born) {
    this.name = name;
    this.city = city;
    this.born = born;
  }
}
```

```
const personToString = (person) => {
  const { name, city, born } = person;
  const { year, month, day } = born;
  const fields = [
    `Name: ${name}`,
    `City: ${city}`,
    `Born: ${year}-${month}-${day}`,
  ];
```

```
  return fields.join('\n');
};
```

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);
console.log(personToString(person));
```

JavaScrip: Objects

```
const person = {
  name: 'Marcus',
  city: 'Roma',
  born: {
    day: 26,
    month: 4,
    year: 121,
  },
};

console.log(personToString(person));
```

JavaScrit: strucute serialization

```
const v8 = require('v8');
const fs = require('fs');
```

Take from previous example:

- class DateStruct - class Person

```
const date = new DateStruct(121, 4, 26);
const person = new Person('Marcus', 'Roma', date);

const v8Data = v8.serialize(person);
const v8File = './file.dat';
fs.writeFile(v8File, v8Data, () => {
```

```
  console.log('Saved ' + v8File);
});
```

File: file.dat

```
FF 0D 6F 22 04 6E 61 6D 65 22 06 4D 61 72 63 75
73 22 04 63 69 74 79 22 04 52 6F 6D 61 22 04 62
6F 72 6E 6F 22 03 64 61 79 49 34 22 05 6D 6F 6E
74 68 49 08 22 04 79 65 61 72 49 F2 01 7B 03 7B
03
```

Nested structures

```
#include <stdio.h>
#include <map>
#include <string>
#include <vector>

struct Product {
  std::string name;
  int price;
};

void printProduct(Product item) {
  printf("%s: %d\n", item.name.c_str(), item.price);
}

void printProducts(std::vector<Product> items) {
  for (int i = 0; i < items.size(); i++) {
    printProduct(items[i]);
  }
}

int main() {
  std::map<std::string, std::vector<Product>> purchase {
    { "Electronics", {
      { "Laptop", 1500 },
```

```
      { "Keyboard", 100 },
      { "HDMI cable", 10 },
    } },
    { "Textile", {
      { "Bag", 50 },
    } },
  };

  std::vector electronics = purchase["Electronics"];
  printf("Electronics:\n");
  printProducts(electronics);

  std::vector textile = purchase["Textile"];
  printf("\nTextile:\n");
  printProducts(textile);

  Product bag = textile[0];
  printf("\nSingle element:\n");
  printProduct(bag);

  int price = purchase["Electronics"][2].price;
  printf("\nHDMI cable price is %d\n", price);
}
```

Python

```
purchase = {
  'Electronics': [
    { 'name': 'Laptop', 'price': 1500 },
    { 'name': 'Keyboard', 'price': 100 },
    { 'name': 'HDMI cable', 'price': 10 },
  ],
  'Textile': [
    { 'name': 'Bag', 'price': 50 },
  ],
}

electronics = purchase['Electronics']
print({ 'electronics': electronics })
```

```
textile = purchase['Textile']
print({ 'textile': textile })

bag = textile[0]
print({ 'bag': bag })

price = purchase['Electronics'][2]['price']
print({ 'price': price })
```

JavaScript

```
const purchase = {
  Electronics: [
    { name: 'Laptop', price: 1500 },
    { name: 'Keyboard', price: 100 },
    { name: 'HDMI cable', price: 10 },
  ],
  Textile: [{ name: 'Bag', price: 50 }],
};

const electronics = purchase.Electronics;
console.log(electronics);

const textile = purchase['Textile'];
console.log(textile);

const bag = textile[0];
console.log(bag);

const price = purchase['Electronics'][2].price;
console.log(price);

const json = JSON.stringify(purchase);
console.log(json);
const obj = JSON.parse(json);
console.log(obj);
```

3.3. Масив, список, множина, кортеж

No translation

3.4. Словник, хеш-таблиця та асоціативний масив

No translation

3.5. Стек, черга, дек

No translation

3.6. Дерева та графи

No translation

3.7. Проекції та відображення наборів даних

No translation

3.8. Оцінка обчислювальної складності

No translation

4. Розширені концепції

No translation

4.1. Що таке технологічний стек

No translation

4.2. Середовище розробки та налагодження коду

No translation

4.3. Ітерування: рекурсія, ітератори та генератори

No translation

4.4. Структура додатку: файли, модулі, компоненти

No translation

4.5. Об'єкт, прототип та клас

No translation

4.6. Часткове застосування та каррування, композиція функцій

No translation

4.7. Чейнінг для методів та функцій

No translation

4.8. Домішки (mixins)

No translation

4.9. Залежності та бібліотеки

No translation