***5 Розділ переклад***

Однією з найважливіших особливостей Стандарту C++11 є щось, що більшість програмістів можливо навіть не помітить. Це не нові можливості синтаксису та не нові функції бібліотеки, а нова модель пам'яті, яка свідома багатопотоковості. Без цієї моделі пам'яті, яка точно визначає, як працюють основні будівельні блоки, не можна було б сподіватися, що будь-які із розглянутих мною засобів будуть надійно працювати. Звісно, є причина, чому більшість програмістів не помічають цього: якщо ви використовуєте м'ютекси для захисту ваших даних та умовні змінні або об'єкти futures для сигналізації подій, то деталі того, чому вони працюють, не є важливими. Це стає важливим лише тоді, коли ви починаєте намагатися наблизитися до "машини", і саме в цьому випадку точні деталі моделі пам'яті мають значення.

Незважаючи на все інше, C++ - це мова системного програмування. Одна з метою Комітету зі стандартів полягає в тому, що не повинно бути потреби у мові рівня нижче, ніж C++. Програмістам має бути надано достатньо гнучкості всередині C++, щоб вони могли робити все, що їм потрібно, без перешкод мови, дозволяючи їм наблизитися до "машини", коли це стає необхідним. Атомарні типи та операції саме це й дозволяють, надаючи можливості для операцій низькорівневої синхронізації, які зазвичай можуть зводитися до одного або двох інструкцій процесора.

У цій главі я спочатку розгляну основи моделі пам'яті, потім перейду до атомарних типів і операцій, і, нарешті, розгляну різновиди синхронізації, доступні з операціями на атомарних типах. Це досить складно: якщо ви не плануєте писати код, який використовує атомарні операції для синхронізації (наприклад, безблокуючі структури даних у главі 7), вам не потрібно знати ці деталі.

Давайте почнемо з основ моделі пам'яті.

***5.1 Memory model basics***

Модель пам'яті має два аспекти: основні структурні аспекти, які відносяться до того, як речі розміщуються у пам'яті, і аспекти конкурентності. Структурні аспекти важливі для конкурентності, особливо коли ви розглядаєте низькорівневі атомарні операції, тому я розпочну з них. У C++ все стосується об'єктів і місць у пам'яті.

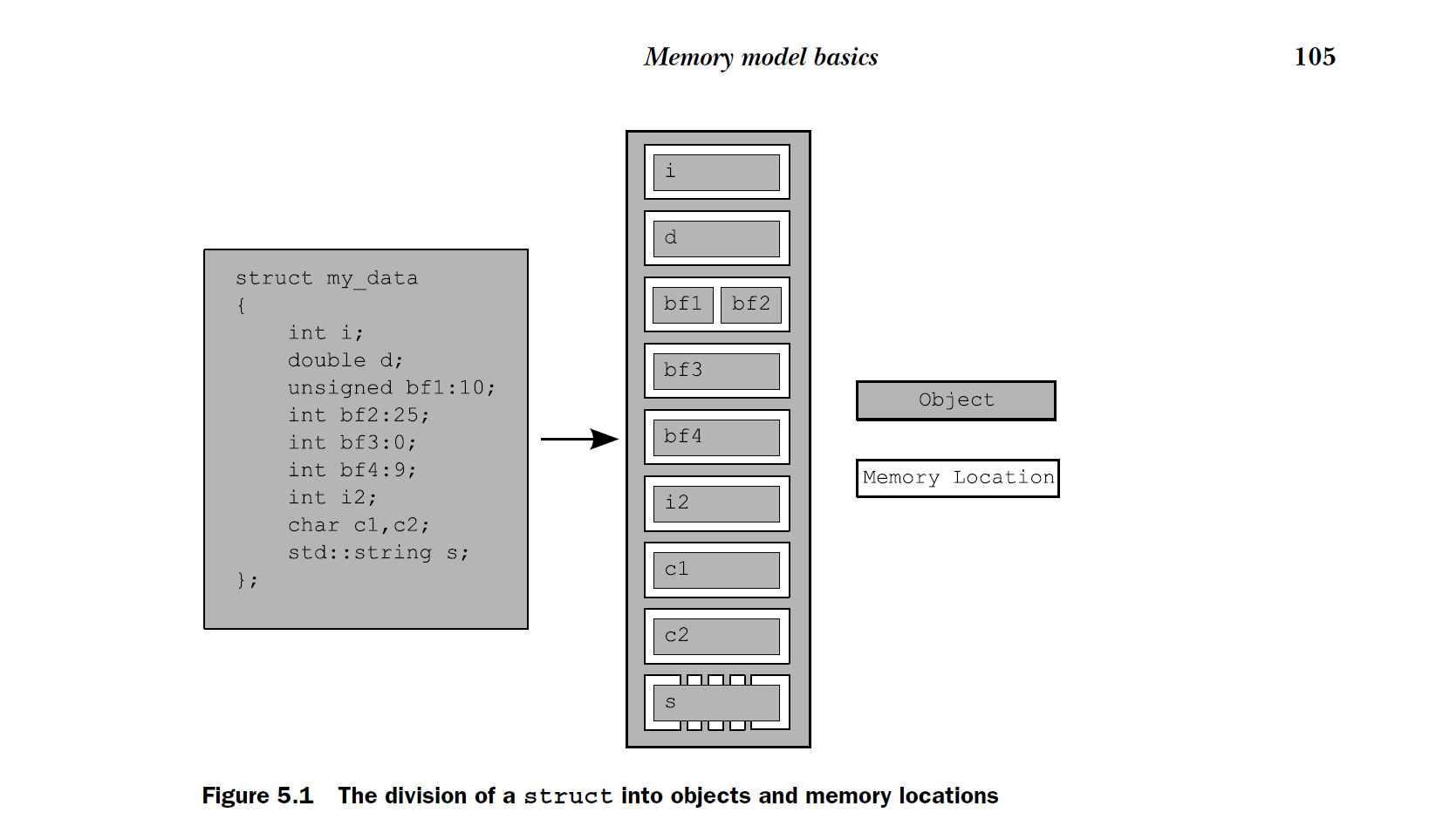
***5.1.1 Objects and memory locations***

Усі дані в програмі на C++ складаються з об'єктів. Це не означає, що ви можете створити новий клас, який походить від int, або що фундаментальні типи мають члени-функції, чи будь-які інші наслідки, які часто вказуються, коли люди говорять "все - це об'єкт", обговорюючи мови, наприклад, Smalltalk або Ruby. Це просто заява про будівельні блоки даних у C++. Стандарт C++ визначає об'єкт як "область пам'яті", хоча йде далі, надаючи властивості цим об'єктам, такі як їх тип та час життя.

Деякі з цих об'єктів - це прості значення фундаментального типу, такого як int чи float, тоді як інші - екземпляри класів, які визначені користувачем. Деякі об'єкти (наприклад, масиви, екземпляри похідних класів та екземпляри класів із нестатичними членами даних) мають підоб'єкти, а інші - ні.

Незалежно від їхнього типу, об'єкт зберігається в одному чи кількох місцях у пам'яті. Кожне таке місце у пам'яті є або об'єктом (або підоб'єктом) скалярного типу, такого як unsigned short або my\_class\*, або послідовністю суміжних бітових полів. Якщо ви використовуєте бітові поля, це важливий момент: хоча суміжні бітові поля є відмінними об'єктами, вони все ще рахуються як те саме місце у пам'яті. На рисунку 5.1 показано, як структура розподіляється на об'єкти та місця у пам'яті.

Спочатку вся структура - це один об'єкт, який складається з декількох підоб'єктів, одного для кожного члена даних. Бітові поля bf1 та bf2 ділять одне місце у пам'яті, а об'єкт std::string s складається з кількох місць у пам'яті внутрішньо, але в іншому випадку кожен член має своє власне місце в пам'яті. Зверніть увагу, як нульове бітове поле bf3 розділяє bf4 на його власне місце у пам'яті.



З цього можна зробити чотири важливі висновки:

■ Кожна змінна - це об'єкт, включаючи ті, що є членами інших об'єктів.

■ Кожен об'єкт займає принаймні одне місце в пам'яті.

■ Змінні фундаментального типу, такі як int чи char, складають одне місце в пам'яті, незалежно від їхнього розміру, навіть якщо вони є суміжними чи частинами масиву.

■ Суміжні бітові поля є частиною того самого місця у пам'яті.

Я впевнений, що ви цікавитеся, який це має відношення до конкурентності, тому давайте розглянемо це.

***5.1.2 Objects, memory locations, and concurrency***

Отже, ось та частина, яка є вирішально важливою для багатопотокових застосувань в C++: все залежить від тих місць у пам'яті. Якщо два потоки мають доступ до різних місць у пам'яті, немає проблем: все працює добре. З іншого боку, якщо два потоки мають доступ до того самого місця у пам'яті, вам потрібно бути обережними. Якщо ж жоден із потоків не оновлює місце в пам'яті, то все в порядку; дані, що лише читаються, не потребують захисту чи синхронізації. Якщо один із потоків модифікує дані, може виникнути можливість виникнення "гонки за ресурсами", як описано в главі 3.

Щоб уникнути гонки за ресурсами, потрібно встановити порядок між доступами в двох потоках. Один спосіб забезпечити визначений порядок - використовувати м'ютекси, як описано в главі 3; якщо перед обома доступами замикати той самий м'ютекс, тільки один потік може отримати доступ до місця в пам'яті, отже, один з них повинен відбутися після іншого. Інший спосіб - використовувати властивості синхронізації атомарних операцій (див. розділ 5.2 для визначення атомарних операцій) на тому самому чи іншому місці в пам'яті для забезпечення порядку між доступами в двох потоках. Використання атомарних операцій для встановлення порядку описано в розділі 5.3.

Якщо більше ніж два потоки мають доступ до того самого місця в пам'яті, кожна пара доступів повинна мати визначений порядок.

Якщо між двома доступами до одного місця в пам'яті з різних потоків не встановлений визначений порядок, один чи обидва цих доступи не є атомарними, і один чи обидва з них є записами, то це є гонка за даними і викликає невизначену поведінку.

Ця заява має критичне значення: невизначена поведінка - це одна з найбрудніших сторінок C++. Згідно із стандартом мови, якщо додаток має хоча б одну невизначену поведінку, тоді все може бути не передбачуваним; поведінка всього додатка тепер невизначена, і він може робити що завгодно. Я знаю про випадок, коли певний випадок невизначеної поведінки спричинив спалення монітора у когось. Хоча ймовірність того, що це станеться з вами, вельми мала, гонка за даними - це безперечно серйозна помилка, яку слід уникати за будь-яку ціну.

Існує ще один важливий момент у цій заяві: ви також можете уникнути невизначеної поведінки, використовуючи атомарні операції для доступу до місця в пам'яті, що залучено до гонки за ресурсами. Це не запобігає самій гонці - не визначено, яка з атомарних операцій торкнеться місця в пам'яті першою - але це повертає програму в межі визначеної поведінки.

Перш ніж розглядати атомарні операції, є ще одне поняття, яке важливо зрозуміти про об'єкти та місця в пам'яті: порядок модифікації.

***5.1.3 Modification orders***

Кожен об'єкт у програмі на C++ має визначений порядок модифікації, який складається з усіх записів у цей об'єкт з усіх потоків у програмі, починаючи з ініціалізації об'єкта. У більшості випадків цей порядок буде різнитися між запусками програми, але в будь-якому конкретному виконанні програми всі потоки в системі повинні узгоджувати цей порядок. Якщо об'єкт, що розглядається, не є одним із атомарних типів, описаних в розділі 5.2, вам слід забезпечити достатню синхронізацію, щоб гарантувати, що потоки узгоджуються щодо порядку модифікації кожної змінної. Якщо різні потоки бачать різні послідовності значень для однієї змінної, ви маєте гонку за даними та невизначену поведінку (див. розділ 5.1.2). Якщо ви використовуєте атомарні операції, компілятор відповідає за те, щоб забезпечити необхідну синхронізацію.

Це вимога означає, що певні види спекулятивного виконання не допускаються, оскільки, як тільки потік бачить певний елемент у порядку модифікації, подальші читання з цього потоку повинні повертати значення з пізнішого часу, і подальші записи з цього потоку до цього об'єкта повинні відбуватися пізніше в порядку модифікації. Також читання об'єкта, яке слідує за записом до цього об'єкта в тому самому потоці, повинно повертати або значення, записане, або інше значення, яке відбувається пізніше в порядку модифікації цього об'єкта. Хоча всі потоки повинні узгоджувати порядок модифікації кожного окремого об'єкта у програмі, вони не обов'язково повинні узгоджувати порядок операцій для окремих об'єктів. Дивіться розділ 5.3.3 для отримання додаткової інформації про порядок операцій між потоками.

Тепер давайте розглянемо, що означає атомарна операція і як їх можна використовувати для встановлення порядку.

***5.2 Atomic operations and types in C++***

Атомарна операція - це нерозривна операція. Ви не можете спостерігати за такою операцією, яка залишилася незавершеною, з будь-якого потоку в системі; вона або виконана, або не виконана. Якщо операція завантаження, яка зчитує значення об'єкта, є атомарною, і всі модифікації цього об'єкта також є атомарними, то ця операція завантаження візьме або початкове значення об'єкта, або значення, збережене однією з модифікацій.

Протилежність цьому полягає в тому, що незаатомарна операція може бути сприйнята іншим потоком як незавершена. Якщо ця операція - це збереження, значення, спостережене іншим потоком, може бути або не значення перед збереженням, або не значення, що було збережено, а щось інше. Якщо незаатомарною операцією є завантаження, вона може витягнути частину об'єкта, інший потік може змінити значення, а потім витягнути залишок об'єкта, таким чином отримуючи не ані перше значення, ані друге, а якусь комбінацію обох. Це простий проблемний випадок "гонки за ресурсами", як описано в главі 3, але на цьому рівні він може бути гонкою за даними (див. розділ 5.1) і, таким чином, викликати невизначену поведінку.

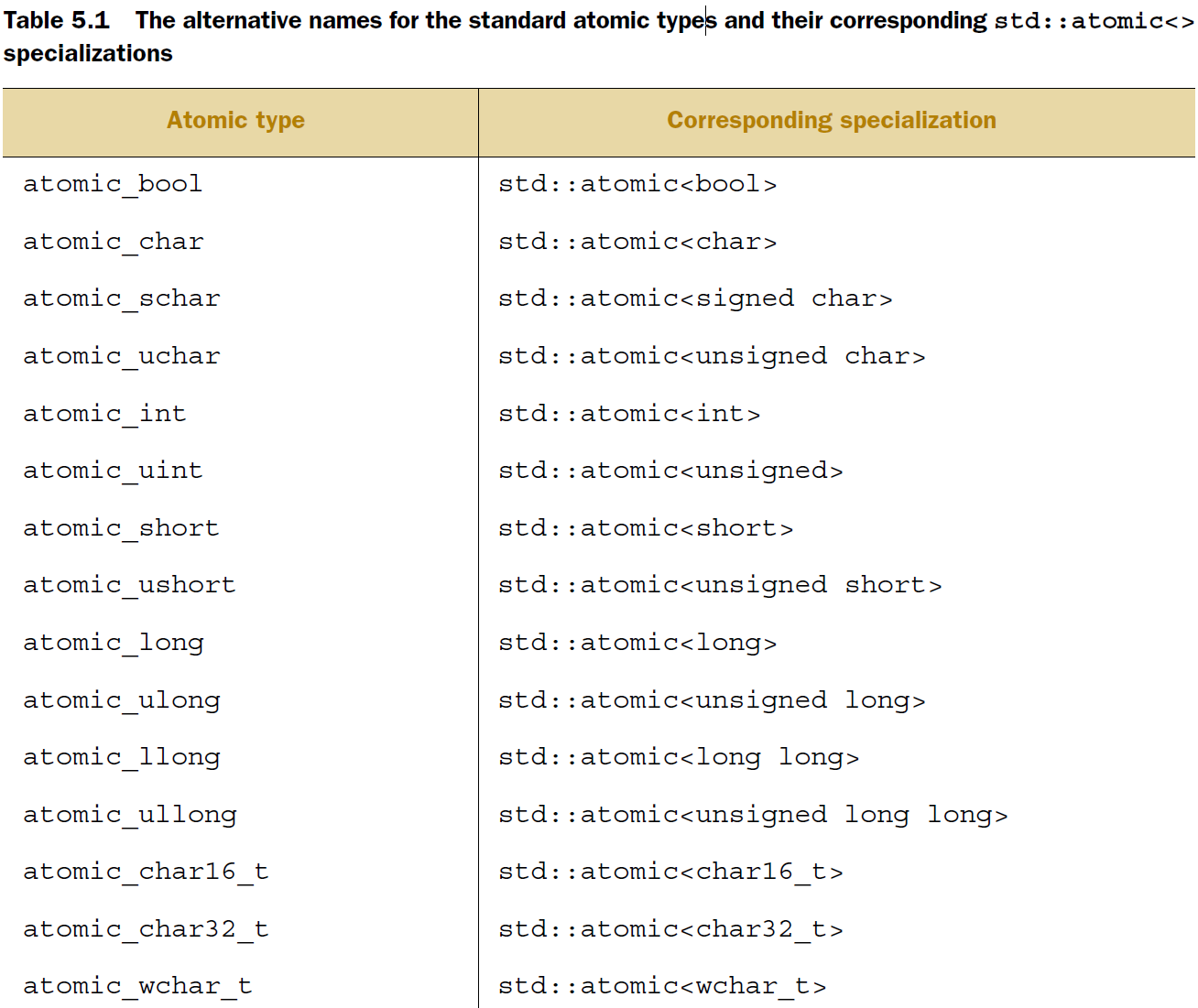
У C++ вам потрібно використовувати атомарний тип для отримання атомарної операції у більшості випадків, тому давайте розглянемо їх.

***5.2.1 The standard atomic types***

Стандартні атомарні типи можна знайти у заголовочному файлі **<atomic>**. Всі операції з такими типами є атомарними, і лише операції з цими типами є атомарними в сенсі визначення мови, хоча ви можете використовувати м'ютекси, щоб інші операції здавалися атомарними. Насправді, стандартні атомарні типи можуть використовувати таку емуляцію: у них (майже) всі є член-функція is\_lock\_free(), яка дозволяє користувачеві визначити, чи виконуються операції з даного типу безпосередньо із використанням атомарних інструкцій (x.is\_lock\_free() повертає true), чи вони виконуються за допомогою замка внутрішнього для компілятора і бібліотеки (x.is\_lock\_free() повертає false).

Єдиний тип, який не надає член-функцію is\_lock\_free(), - це **std::atomic\_flag**. Цей тип - це дуже простий булевий прапорець, і операції з цим типом повинні бути безблокуючими; як тільки у вас є простий безблокуючий булевий прапорець, ви можете використовувати його для реалізації простого замка і таким чином реалізовувати всі інші атомарні типи, використовуючи його як основу. Коли я сказав "дуже простий", я мав на увазі саме це: об'єкти типу **std::atomic\_flag** ініціалізуються як очищені, і їх можна перевіряти та встановлювати (за допомогою члена-функції **test\_and\_set()**) або очищати (за допомогою члена-функції **clear()**). Це все: жодного присвоєння, ні копіювальної конструкції, ні операцій перевірки та очищення, абсолютно жодних інших операцій.

Решта атомарних типів доступні через спеціалізації класу-шаблону **std::atomic<>** і трошки більш повнофункціональні, але можуть не бути безблокуючими (як пояснено раніше). На більшості популярних платформ очікується, що атомарні варіанти всіх вбудованих типів (таких як **std::atomic<int>** та **std::atomic<void\*>**) є безблокуючими, але це не є обов'язковим. Як ви побачите незабаром, інтерфейс кожної спеціалізації відображає властивості типу; операції за побітовими операціями, такими як &=, не визначені для звичайни



A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

Крім основних атомарних типів, стандартна бібліотека C++ також надає набір псевдонімів для атомарних типів, які відповідають різним неатомарним псевдонімам стандартної бібліотеки, таким як **std::size\_t**. Їх показано в таблиці 5.2. Є досить простий шаблон: для стандартного псевдоніму **T** відповідний атомарний тип має ту саму назву з префіксом **atomic\_**: **atomic\_T**. Те ж саме стосується вбудованих типів, за винятком того, що **signed** скорочується як **s**, **unsigned** - як **u**, а **long long** - як **llong**. Зазвичай проще використовувати **std::atomic<T>** для будь-якого **T**, з яким ви хочете працювати, аніж використовувати альтернативні назви.

Стандартні атомарні типи не можна копіювати або присвоювати у звичайному розумінні, оскільки вони не мають конструкторів копіювання або операторів копіювання. Проте, вони підтримують присвоєння та неявне перетворення до відповідних вбудованих типів, а також прямі методи завантаження (**load()**) та збереження (**store()**), **exchange()**, **compare\_exchange\_weak()**, та **compare\_exchange\_strong()**. Вони також підтримують складні оператори присвоєння, де це відповідно: **+=**, **-=** , **\*=**, **|=** та інші, а також цілочисельні типи та спеціалізації **std::atomic<>** для покажчиків підтримують **++** та **--**. Ці оператори також мають відповідні іменовані методи-члени з тією ж самою функціональністю: **fetch\_add()**, **fetch\_or()** та інші. Значення, що повертається від операторів присвоєння та методів-членів, - це або значення, збережене (у випадку операторів присвоєння), або значення перед операцією (у випадку іменованих функцій). Це уникне можливих проблем, які можуть виникнути від звичайної звички операторів присвоєння повертати посилання на об'єкт, який повинен бути присвоєний. Для отримання збереженого значення з такого посилання код повинен виконати окремий читання, що дозволяє іншому потоці змінювати значення між операцією присвоєння та читанням, відчиняючи двері для гонки за даними.

Клас-шаблон **std::atomic<>** - це не просто набір спеціалізацій. Він має основний шаблон, який можна використовувати для створення атомарної варіанту користувацького типу. Оскільки він є загальним класом-шаблоном, операції обмежені **load()**, **store()** (і присвоєнням та перетворенням до користувацького типу), **exchange()**, **compare\_exchange\_weak()**, та **compare\_exchange\_strong()**. Кожна з операцій над атомарними типами має необов'язковий аргумент порядку пам'яті, який можна використовувати для вказівки потрібних семантик порядку пам'яті. Точні семантики опцій порядку пам'яті розглядаються в розділі 5.3. Зараз досить знати, що операції розділені на три категорії:

* Операції збереження, які можуть мати порядок пам'яті **memory\_order\_relaxed**, **memory\_order\_release**, або **memory\_order\_seq\_cst**.
* Операції завантаження, які можуть мати порядок пам'яті **memory\_order\_relaxed**, **memory\_order\_consume**, **memory\_order\_acquire**, або **memory\_order\_seq\_cst**.
* Операції читання-модифікації-запису, які можуть мати порядок пам'яті **memory\_order\_relaxed**, **memory\_order\_consume**, **memory\_order\_acquire**, **memory\_order\_release**, **memory\_order\_acq\_rel**, або **memory\_order\_seq\_cst**. За замовчуванням порядок виконання для всіх операцій - **memory\_order\_seq\_cst**. Давайте тепер розглянемо операції, які можна виконати на кожному зі стандартних атомарних типів, починаючи з **std::atomic\_flag**.

***5.2.2 Operations on std::atomic\_flag***

**std::atomic\_flag** - це найпростіший стандартний атомарний тип, який представляє собою булевий прапорець. Об'єкти цього типу можуть перебувати в одному з двох станів: встановлений або очищений. Це намірено базовий тип і призначений лише як базовий компонент. Таким чином, я ніколи не очікував би його використання, крім дуже специфічних випадків. Проте він слугує початковою точкою для обговорення інших атомарних типів, оскільки він показує деякі загальні принципи, які стосуються атомарних типів.

Об'єкти типу **std::atomic\_flag** повинні бути ініціалізовані за допомогою **ATOMIC\_FLAG\_INIT**. Це ініціалізує прапорець в очищений стан. Вибору в цьому немає; прапорець завжди починається з очищеного стану:

A group of white letters on a black background

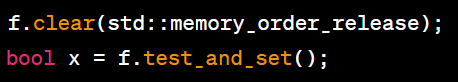
Description automatically generated

Це стосується будь-де, де об'єкт оголошено і який би не мав він області видимості. Це єдиний атомарний тип, який вимагає такої спеціальної обробки для ініціалізації, але він також єдиний тип, що гарантовано є безблокуючим. Якщо об'єкт типу **std::atomic\_flag** має статичний час життя, гарантується його статична ініціалізація, що означає відсутність проблем з порядком ініціалізації; він завжди буде ініціалізований до моменту першої операції з прапорцем.

Як тільки у вас є ініціалізований об'єкт прапорця, існують лише три дії, які ви можете виконати з ним: знищити його, очистити його, встановити його та перевірити попереднє значення. Ці дії відповідають деструктору, методу-члену **clear()** та методу-члену **test\_and\_set()**, відповідно. Як методи **clear()**, так і **test\_and\_set()** можуть мати вказане замовлення пам'яті. **clear()** - це операція запису, і, отже, вона не може мати семантику **memory\_order\_acquire** або **memory\_order\_acq\_rel**, але **test\_and\_set()** - це операція читання-модифікації-запису, і вона може мати будь-які з тегів впорядкування пам'яті.

Як і в усіх атомарних операціях, за замовчуванням для обох варіантів є **memory\_order\_seq\_cst**.

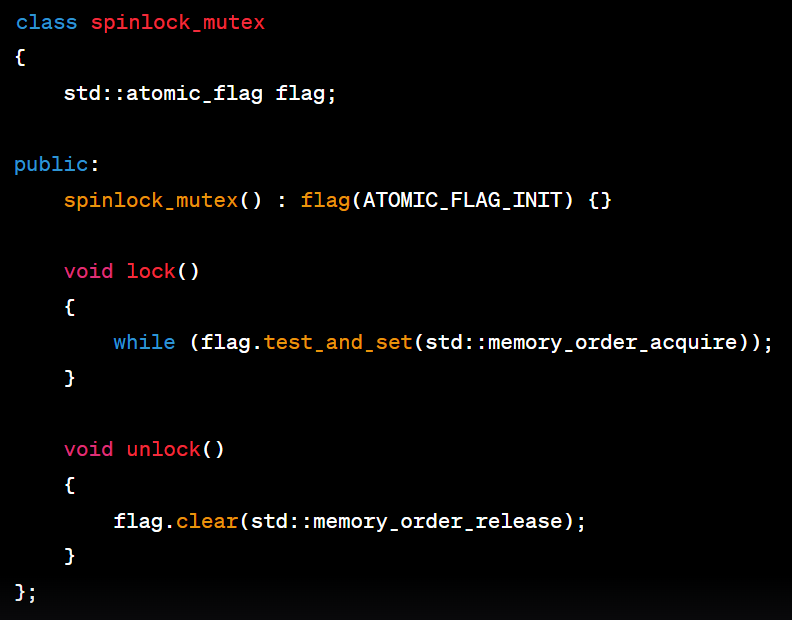
Наприклад:



Тут виклик **clear()** явно вказує, що прапорець очищається зі семантикою випуску (release semantics), тоді як виклик **test\_and\_set()** використовує типове упорядкування пам'яті для встановлення прапорця та отримання старого значення.

Ви не можете скопіювати інший об'єкт **std::atomic\_flag** з першого об'єкта і не можете присвоїти один **std::atomic\_flag** іншому. Це не є щось особливим для **std::atomic\_flag**, це загальне для всіх атомарних типів. Усі операції з атомарним типом визначені як атомарні, і присвоєння та копіювання об'єктів включають два об'єкти. Одна операція з двома різними об'єктами не може бути атомарною. У випадку копіювання або присвоєння значення спочатку має бути прочитано з одного об'єкта і потім записано в інший. Це дві окремі операції над двома різними об'єктами, і комбінація не може бути атомарною. Тому ці операції не дозволяються.

Обмежений набір функцій робить **std::atomic\_flag** ідеальним для використання як м'ютекс типу "спінлок". Спочатку прапорець є очищеним, і м'ютекс розблокований. Для блокування м'ютексу використовуйте цикл навколо **test\_and\_set()**, доки старе значення не буде false, що вказує, що цей потік встановив значення в true. Розблокування м'ютексу - це просто очищення прапорця. Така реалізація показана в наступному прикладі.



Такий м'ютекс є дуже простим, але цього достатньо для використання з **std::lock\_guard<>** (див. розділ 3). За своєю природою він виконує активне очікування в **lock()**, тому це поганий вибір, якщо ви очікуєте будь-якої кількості конкуренції, але цього достатньо для забезпечення взаємовиключення. Коли ми розглянемо семантику упорядкування пам'яті, ви побачите, як це гарантує необхідне виконання вимоги щодо встановлення порядку, яка супроводжує блокування м'ютекса. Цей приклад розглядається в розділі 5.3.6.

**std::atomic\_flag** настільки обмежений, що його навіть не можна використовувати як загальний логічний прапорець, оскільки він не має простої операції запиту, яка не змінює значення. Для цього краще використовувати **std::atomic<bool>**, тому далі я розгляну його.

***5.2.3 Operations on std::atomic<bool>***

Найпростіший з атомарних цілих типів - **std::atomic<bool>**. Це більш функціональний булевий прапорець, ніж **std::atomic\_flag**, як ви можливо очікували. Хоча його неможливо скопіювати або присвоїти значення з копії, ви можете конструювати його зі звичайного натомість **bool**, тому він може бути початково істинним або хибним. Ви також можете присвоїти значення екземплярам **std::atomic<bool>** звичайним **bool**:

A black background with white text

Description automatically generated

Ще одне, на що варто звернути увагу, це те, що оператор присвоєння з звичайного **bool** відрізняється від загальної конвенції повертання посилання на об'єкт, який йому присвоюється: він повертає **bool** із значенням, яке було присвоєно. Це ще один загальний патерн для атомарних типів: оператори присвоєння, які вони підтримують, повертають значення (відповідного типу без атомарності), а не посилання. Якщо б поверталося посилання на атомарну змінну, будь-який код, що залежить від результату присвоєння, міг би явно завантажувати значення, існує ймовірність отримання результату зміни іншим потоком. Повертаючи результат присвоєння як звичайне значення, ви можете уникнути цього додаткового завантаження, і ви знаєте, що отримане значення - це фактичне збережене значення. На відміну від обмежувальної функції **clear()** для **std::atomic\_flag**, записи (як істинні, так і хибні) виконуються за допомогою виклику **store()**, хоча семантика упорядкування пам'яті все ще може бути вказана. Також **test\_and\_set()** був замінений більш загальною функцією-членом **exchange()**, яка дозволяє вам замінити збережене значення новим вибором і атомарно отримати початкове значення. **std::atomic<bool>** також підтримує простий опитувальний запит значення з імпліцитною конвертацією в звичайний **bool** або з явним викликом **load()**. Як і можна очікувати, **store()** є операцією запису, тоді як **load()** - це операція читання. **exchange()** - це операція читання-модифікації-запису:

A black background with white text

Description automatically generated

**exchange()** - це не єдина підтримувана операція читання-модифікації-запису для **std::atomic<bool>**; вона також вводить операцію для збереження нового значення, якщо поточне значення дорівнює очікуваному значенню.

STORING A NEW VALUE (OR NOT) DEPENDING ON THE CURRENT VALUE

Нова операція називається порівняння/обмін (compare/exchange), і вона надається у вигляді член-функцій **compare\_exchange\_weak()** та **compare\_exchange\_strong()**. Операція порівняння/обмін є основою програмування з атомарними типами; вона порівнює значення атомарної змінної з передбаченим значенням і зберігає задане бажане значення, якщо вони рівні. Якщо значення не рівні, передбачуване значення оновлюється фактичним значенням атомарної змінної. Типом повернення функцій порівняння/обміну є **bool**, який дорівнює **true**, якщо операція запису виконана, і **false** у протилежному випадку.

У випадку **compare\_exchange\_weak()**, запис може бути неуспішним навіть якщо оригінальне значення було рівним передбаченому значенню, у такому випадку значення змінної залишається незмінним, і значення, повернене **compare\_exchange\_weak()**, дорівнює **false**. Це найімовірніше трапляється на машинах, які не мають інструкції одноразового порівняння та обміну, якщо процесор не може гарантувати атомарність операції - можливо через те, що потік, який виконує операцію, був замінений серед відповідної послідовності інструкцій, та інший потік запущений його місцем операційної системи, коли потоків більше, ніж процесорів. Це називається випадковою помилкою, оскільки причина відмови полягає в хронометражу, а не в значеннях змінних.

Оскільки **compare\_exchange\_weak()** може трапитися випадковою відмовою, його зазвичай слід використовувати в циклі:

A black background with white text

Description automatically generated

У цьому випадку ви продовжуєте цикл, доки **expected** залишається **false**, що вказує на те, що виклик **compare\_exchange\_weak()** виявився невдалим через випадкову відмову.

З іншого боку, **compare\_exchange\_strong()** гарантує повернення **false** тільки у випадку, якщо фактичне значення не рівне передбаченому значенню. Це може усунути потребу у циклах, подібних показаному вище, де ви просто хочете знати, чи успішно змінили ви змінну чи інший потік вже там перший.

Якщо ви хочете змінити змінну незалежно від початкового значення (можливо, з оновленим значенням, яке залежить від поточного значення), оновлення **expected** стає корисним; кожен раз під час циклу **expected** перезавантажується, тому якщо інший потік не модифікує значення протягом цього часу, виклик **compare\_exchange\_weak()** або **compare\_exchange\_strong()** повинен бути успішним наступного разу в циклі. Якщо обчислення значення для зберігання є простим, можливо, корисно використовувати **compare\_exchange\_weak()**, щоб уникнути подвійного циклу на платформах, де **compare\_exchange\_weak()** може випадково відмовити (і тому **compare\_exchange\_strong()** містить цикл). З іншого боку, якщо обчислення значення для зберігання саме час витрачає, можливо, має сенс використовувати **compare\_exchange\_strong()**, щоб уникнути необхідності перерахування значення для зберігання, коли очікуване значення не змінилося. Для **std::atomic<bool>** це не так важливо - все-таки тут можливі всього два можливих значення - але для більших атомарних типів це може мати значення.

Функції порівняння/обміну також незвичайні тим, що вони можуть приймати два параметри для упорядкування пам'яті. Це дозволяє відмінність семантики упорядкування пам'яті у випадку успіху та невдачі; може бути бажаним, щоб успішний виклик мав семантику **memory\_order\_acq\_rel**, тоді як невдалий виклик мав семантику **memory\_order\_relaxed**. Невдалий виклик порівняння/обміну не виконує операції запису, тому він не може мати семантику **memory\_order\_release** або **memory\_order\_acq\_rel**. Таким чином, неможливо вказати ці значення як упорядкування для невдачі. Також не можна вказати строгіше упорядкування пам'яті для невдачі, ніж для успіху; якщо ви хочете семантику **memory\_order\_acquire** або **memory\_order\_seq\_cst** для невдачі, ви повинні вказати їх для успіху. Якщо ви не вказуєте упорядкування для невдачі, вважається, що воно таке саме, як і для успіху, за винятком того, що частина вивільнення упорядкування відкидається: **memory\_order\_release** стає **memory\_order\_relaxed**, і **memory\_order\_acq\_rel** стає **memory\_order\_acquire**. Якщо ви не вказуєте жодного із них, вони за замовчуванням стають **memory\_order\_seq\_cst**, яке забезпечує повний послідовний порядок як для успіху, так і для невдачі. Наступні два виклики **compare\_exchange\_weak()** еквівалентні:

A black background with white text

Description automatically generatedA black background with white text

Description automatically generated

Причини вибору упорядкування пам'яті залишаються для розділу 5.3.

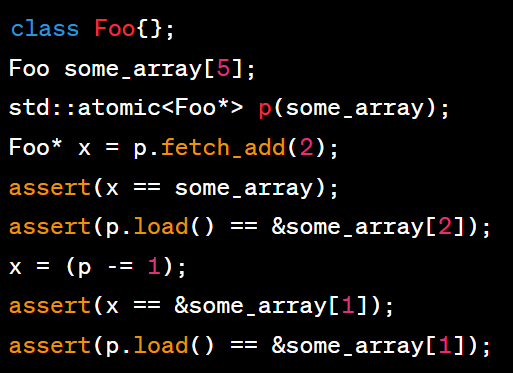
Ще одна різниця між **std::atomic<bool>** і **std::atomic\_flag** полягає в тому, що **std::atomic<bool>** може бути не lock-free; реалізація може внутрішньо здійснювати замовчування м'ютекс, щоб забезпечити атомарність операцій. Для випадків, коли це важливо, ви можете використовувати член-функцію **is\_lock\_free()**, щоб перевірити, чи є операції з **std::atomic<bool>** lock-free. Це є ще однією функцією, спільною для всіх атомарних типів, окрім **std::atomic\_flag**.

Наступні за складністю атомарні типи - це атомарні спеціалізації вказівників **std::atomic<T\*>**, тому розглянемо їх наступними.

***5.2.4 Operations on std::atomic<T\*>: pointer arithmetic***

Атомний вигляд вказівника на деякий тип **T** - це **std::atomic<T\*>**, так само, як і атомний вигляд **bool** - це **std::atomic<bool>**. Інтерфейс є суттєво таким самим, хоча він працює зі значеннями відповідного вказівникового типу, а не значеннями **bool**. Так само, як і **std::atomic<bool>**, він не може бути скопійований чи присвоєний за допомогою конструкторів копіювання, але його можна створити та призначити відповідні значення вказівників. Крім обов'язкового члена-функції **is\_lock\_free()**, **std::atomic<T\*>** також має методи **load()**, **store()**, **exchange()**, **compare\_exchange\_weak()**, і **compare\_exchange\_strong()**, які мають схожі семантичні значення з **std::atomic<bool>**, знову ж таки, працюючи з типами **T\*** замість **bool**.

Нові операції, які надає **std::atomic<T\*>**, - це операції арифметики вказівників. Основні операції надаються методами **fetch\_add()** і **fetch\_sub()**, які роблять атомарні операції додавання та віднімання збереженої адреси, а також оператори **+=** і **-=** та пре- та пост-інкремент і декремент з **++** і **--**, які надають зручні обгортки. Оператори працюють точно так, як ви очікуєте від вбудованих типів: якщо **x** - це **std::atomic<Foo\*>** для першого елемента масиву об'єктів **Foo**, то **x+=3** змінює його так, що він вказує на четвертий елемент і повертає звичайний вказівник **Foo\***, який також вказує на цей четвертий елемент. **fetch\_add()** і **fetch\_sub()** трохи відрізняються тим, що вони повертають початкове значення (таким чином, **x.fetch\_add(3)** оновить **x** для вказівки на четвертий елемент, але поверне вказівник на перший елемент у масиві). Цю операцію також називають "обмін та додавання", і вона є атомарною операцією читання-модифікації-запису, подібно **exchange()** та **compare\_exchange\_weak()**/**compare\_exchange\_strong()**. Так само як і в інших операціях, повернене значення - це звичайне значення типу **T\***, а не посилання на об'єкт **std::atomic<T\*>**, щоб викликаючий код міг виконувати дії на основі попереднього значення:



Функціональні форми також дозволяють вказати семантику порядку пам'яті як додатковий аргумент виклику функції:



Оскільки обидві **fetch\_add()** і **fetch\_sub()** є операціями читання-модифікації-запису, вони можуть мати будь-які теги порядку пам'яті і можуть брати участь у вивільненні послідовності. Вказівка семантики порядку, що не вдається для операторних форм, оскільки немає способу надання цієї інформації: ці форми завжди мають семантику **memory\_order\_seq\_cst**.

Решта основних атомних типів суттєво схожі між собою: всі вони є атомними цілими типами та мають той самий інтерфейс, але відповідний вбудований тип різний. Ми розглянемо їх як групу.

***5.2.5 Operations on standard atomic integral types***

Крім звичайних операцій (load(), store(), exchange(), compare\_exchange\_weak() та compare\_exchange\_strong()), атомні цілі типи, такі як **std::atomic<int>** або **std::atomic<unsigned long long>**, мають досить широкий набір доступних операцій: **fetch\_add()**, **fetch\_sub()**, **fetch\_and()**, **fetch\_or()**, **fetch\_xor()**, операції складеного призначення для цих операцій (**+=**, **-=** , **&=**, **|=** та **^=**) і пре- та пост-інкременти та декременти (**++x**, **x++**, **--x** і **x--**). Це не зовсім повний набір операцій складеного призначення, які можна виконати для звичайного цілого типу, але це дуже близько: відсутні тільки операції ділення, множення та зсуву. Оскільки атомні цілі значення зазвичай використовуються як лічильники або маски бітів, це не є особливо помітною втратою; додаткові операції можна легко виконати за допомогою **compare\_exchange\_weak()** у циклі, якщо це потрібно.

Семантика дуже близька до **fetch\_add()** і **fetch\_sub()** для **std::atomic<T\*>**. Названі функції атомарно виконують свою операцію і повертають старе значення, тоді як оператори складеного призначення повертають нове значення. Пре- та пост-інкременти та декременти працюють як зазвичай: **++x** збільшує змінну і повертає нове значення, тоді як **x++** збільшує змінну і повертає старе значення. Як і очікується на цьому етапі, результатом є значення асоційованого цілого типу в обох випадках.

Тепер ми розглянули всі основні атомні типи; все, що залишається, це загальний основний клас-шаблон **std::atomic<>**, а не його спеціалізації, так що давайте розглянемо його наступним.

***5.2.6 The std::atomic<> primary class template***

Наявність первинного шаблону дозволяє користувачеві створювати атомний варіант власного типу даних, додатково до стандартних атомних типів. Проте не можна використовувати будь-який користувацький тип з **std::atomic<>**. Тип має задовольняти певні критерії. Щоб використовувати **std::atomic<UDT>** для користувацького типу UDT, цей тип повинен мати тривіальний оператор копіюваної присвоєння. Це означає, що тип не повинен мати жодних віртуальних функцій або віртуальних базових класів і повинен використовувати генерований компілятором оператор копіюваної присвоєння. Крім того, кожен базовий клас і нестатичний член даних користувацького типу повинен також мати тривіальний оператор копіюваної присвоєння. Це в суті дозволяє компілятору використовувати операцію memcpy() або еквівалентну операцію для операцій присвоєння, оскільки немає користувацького коду для виконання.

Зрештою, тип повинен бути порівнюваний на рівні бітів. Це йде поруч з вимогами щодо присвоєння; не тільки ви повинні мати можливість скопіювати об'єкт типу UDT за допомогою memcpy(), але ви також повинні мати можливість порівняти екземпляри на рівні бітів за допомогою memcmp(). Ця гарантія потрібна для роботи операцій compare/exchange.

Мотивація за такими обмеженнями повертається до одного з принципів з розділу 3: не передавати вказівники та посилання на захищені дані за межі області блокування, передавши їх як аргументи до функцій, які надаються користувачем. Загалом компілятор не зможе генерувати код без блокування для **std::atomic<UDT>**, отже для всіх операцій йому доведеться використовувати внутрішнє блокування. Якщо користувацькі оператори копіюваної присвоєння або порівняння були б дозволені, це потребувало б передачі посилання на захищені дані як аргумент до функції, що надається користувачем, порушуючи таким чином цей принцип. Крім того, бібліотека може використовувати одне блокування для всіх атомних операцій, які йому потрібні, і дозвільність викликати користувацькі функції під час утримання цього блокування може спричинити взаємоблокування або призвести до блокування інших потоків через те, що операція порівняння займає дуже багато часу. Нарешті, ці обмеження збільшують ймовірність того, що компілятор зможе використовувати атомні інструкції безпосередньо для **std::atomic<UDT>** (і, таким чином, зробити певний екземпляр безблокуваним), оскільки він може просто розглядати користувацький тип як набір сирових байтів.

Слід зауважити, що ви можете використовувати **std::atomic<float>** або **std::atomic<double>**, оскільки вбудовані типи з плаваючою точкою задовольняють вимоги для використання **memcpy** та **memcmp**. Однак поведінка може бути неочікуваною у випадку **compare\_exchange\_strong**. Операція може не вдалося, навіть якщо старе збережене значення було рівне значенню в порівнянні, якщо збережене значення мало іншу репрезентацію. Зверніть увагу, що немає атомних арифметичних операцій на значеннях з рухомою комою. Аналогічна поведінка буде із **compare\_exchange\_strong**, якщо ви використовуєте **std::atomic<>** з користувацьким типом, для якого визначений оператор порівняння за рівністю, і цей оператор відрізняється від порівняння за допомогою **memcmp** - операція може не вдалося через те, що рівні значення мають різну репрезентацію.

Якщо ваш користувацький тип даних має такий самий розмір, як (або менший, ніж) **int** або **void\***, більшість типових платформ зможе використовувати атомні інструкції для **std::atomic<UDT>**. Деякі платформи також зможуть використовувати атомні інструкції для користувацьких типів, які вдвічі більше за розмір **int** чи **void\***. Зазвичай це ті платформи, які підтримують інструкцію подвійного порівняння і обміну (DWCAS), що відповідає функціям **compare\_exchange\_xxx**. Як ви побачите в главі 7, така підтримка може бути корисною при написанні безблокового коду.

Ці обмеження означають, що ви не можете, наприклад, створити **std::atomic<std::vector<int>>**, але ви можете використовувати його з класами, які містять лічильники, прапорці або вказівники, або навіть просто масивами простих елементів даних. Це не є особливою проблемою; чим складніше структура даних, тим ймовірніше, ви захочете виконувати інші операції з нею, крім простого присвоєння та порівняння. Якщо це так, вам краще використовувати **std::mutex**, щоб забезпечити належний захист даних для бажаних операцій, як описано в главі 3. При інстанціюванні користувацького типу T інтерфейс **std::atomic<T>** обмежується набором операцій, доступних для **std::atomic<bool>**: **load()**, **store()**, **exchange()**, **compare\_exchange\_weak()**, **compare\_exchange\_strong()**, і присвоєння від і до екземпляра типу T. Таблиця 5.3 показує операції, доступні для кожного атомного типу.

***5.2.7 Free functions for atomic operations***

A screenshot of a table

Description automatically generated

До цього часу я фокусувався на описі членських функцій операцій для атомних типів. Однак існують також еквівалентні функції, які не є членами класу для всіх операцій з різних атомних типів. Загалом назви нечленних функцій схожі на їхні аналоги-члени, але з префіксом **atomic\_**. Наприклад, **std::atomic\_load()** - це нечленна версія **load()**, і **std::atomic\_store\_explicit()** відповідає **store\_explicit()**. Ці функції перегружені для кожного атомного типу. Якщо потрібно вказати мітку порядку в пам'яті, існують дві версії цих функцій: одна без мітки і одна з суфіксом **\_explicit**, яка приймає додаткові параметри для мітки або міток порядку в пам'яті.

На відміну від членських функцій, які неявно посилаються на атомний об'єкт, нечленні функції приймають вказівник на атомний об'єкт як перший параметр. Наприклад, **std::atomic\_is\_lock\_free()** - це функція нечлена, яка повертає той самий значення, що і **a.is\_lock\_free()** для об'єкта атомного типу **a**. Аналогічно, **std::atomic\_load(&a)** еквівалентно **a.load()**, а еквівалент **a.load(std::memory\_order\_acquire)** - це **std::atomic\_load\_explicit(&a, std::memory\_order\_acquire)**.

Ці нечленні функції спроектовані так, щоб бути сумісними з С, тому вони використовують вказівники замість посилань у всіх випадках. Наприклад, перший параметр у членських функціях **compare\_exchange\_weak()** та **compare\_exchange\_strong()** (очікуване значення) - це посилання, у той час як перший параметр у **std::atomic\_compare\_exchange\_weak()** (перший - це вказівник на об'єкт) - це вказівник. **std::atomic\_compare\_exchange\_weak\_explicit()** також вимагає вказівки як для успішних, так і для неуспішних міток в пам'яті, на відміну від членських функцій, які мають одну форму мітки пам'яті (за замовчуванням **std::memory\_order\_seq\_cst**), а також перезавантаження, яке приймає мітки пам'яті порядку окремо.

Назви операцій для **std::atomic\_flag** трохи відрізняються. Вони явно включають "флаг" у свої назви, наприклад, **std::atomic\_flag\_test\_and\_set()**, **std::atomic\_flag\_clear()**, хоча додаткові варіанти, які вказують порядок в пам'яті, знову мають суфікс **\_explicit**, наприклад, **std::atomic\_flag\_test\_and\_set\_explicit()** і **std::atomic\_flag\_clear\_explicit()**.

Крім того, стандартна бібліотека C++ надає нечленні функції для взаємодії з екземплярами **std::shared\_ptr<>** атомарним чином. Це відхід від принципу, що атомні операції підтримуються лише атомними типами, оскільки **std::shared\_ptr<>** цілком точно не є атомним типом. Однак комітет стандартів C++ вважав це досить важливим, щоб надати ці додаткові функції. Атомні операції для **std::shared\_ptr<>** включають завантаження, зберігання, обмін і порівняння/обмін, які подаються як перевантаження тих самих операцій для стандартних атомних Так само, як і з атомними операціями для інших типів, надаються **\_explicit** варіанти для можливості вказати потрібний порядок в пам'яті, і функція **std::atomic\_is\_lock\_free()** може бути використана для перевірки того, чи використовує реалізація блокування для забезпечення атомарності.

Як описано в уведенні, стандартні атомні типи роблять більше, ніж просто уникнуть невизначеного поведінки, що пов'язана із гонкою за даними; вони дозволяють користувачеві накладати порядок операцій між потоками. Цей зазначений порядок є основою можливостей захисту даних і синхронізації операцій, таких як **std::mutex** і **std::future<>**. З цим на увазі, перейдемо до основної частини цієї глави: подробиць з аспектів конкурентності моделі пам'яті та того, як атомні операції можна використовувати для синхронізації даних та накладення порядку.

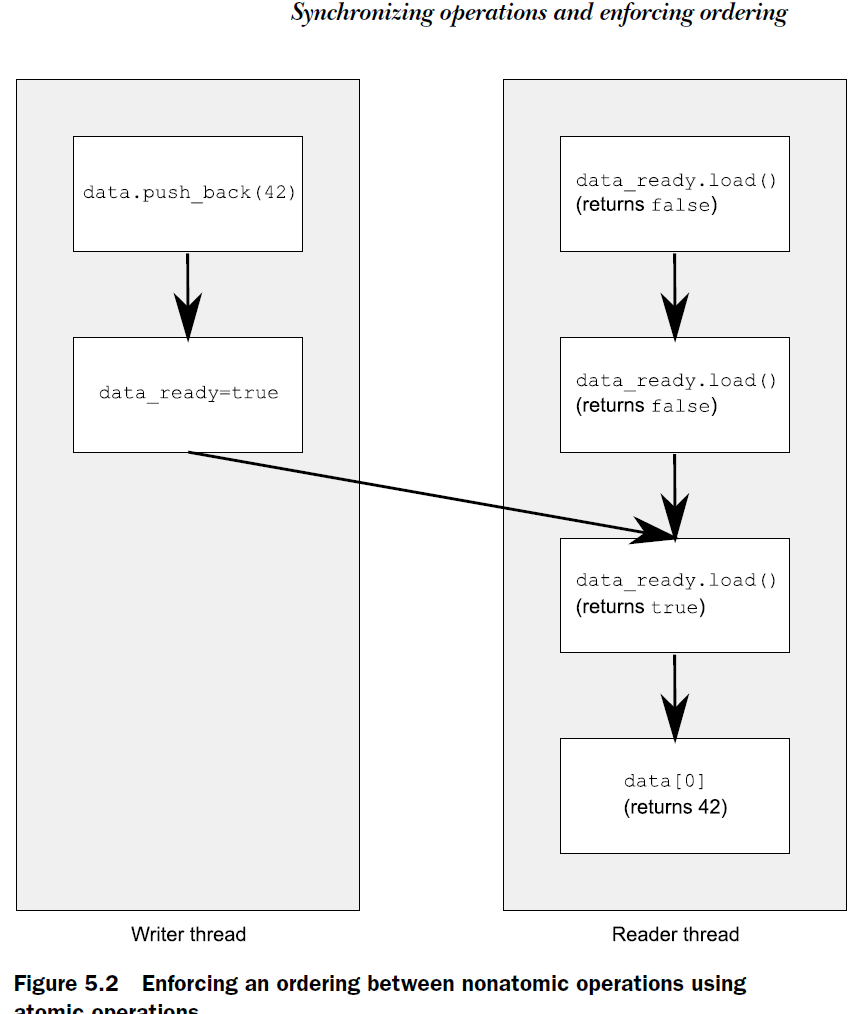
типів. Ці функції приймають **std::shared\_ptr<>\*** як перший аргумент.

***5.3 Synchronizing operations and enforcing ordering***

A screenshot of a computer program

Description automatically generated

Припустімо, у вас є два потоки, один з яких наповнює структуру даних для подальшого читання другим потоком. Щоб уникнути проблемної гонки за даними, перший потік встановлює прапорець для позначення готовності даних, і другий потік не читає дані, поки прапорець не встановлено. Наступний лістинг показує такий сценарій. Залишаючи осторонь неефективність циклу, який чекає, доки дані будуть готові B, вам дійсно потрібно, щоб це працювало, оскільки в іншому випадку обмін даними між потоками стає неможливим: кожен об'єкт даних мусить бути атомарним. Ви вже вивчили, що це є невизначене поведінка мати nonatomic читання c та записи d, які звертаються до тих самих даних без накладеного порядку, тому для того, щоб це працювало, повинен бути затверджений порядок десь. Необхідний затверджений порядок надають операції на змінній std::atomic<bool> data\_ready; вони забезпечують необхідний порядок завдяки відносинам моделі пам'яті "відбулося перед" (happens-before) і "синхронізується з" (synchronizes-with). Запис даних d відбувається перед записом прапорця data\_ready e, а читання прапорця B відбувається перед читанням даних c. Коли значення, прочитане з data\_ready B, дорівнює true, запис синхронізується з цим читанням, створюючи відносину "відбулося перед". Оскільки "відбулося перед" є транзитивним, запис даних d відбувається перед записом прапорця e, який відбувається перед читанням true значення з прапорця B, що відбувається перед читанням даних c, і у вас є затверджений порядок: запис даних відбувається перед читанням даних, і все в порядку. На рисунку 5.2 показані важливі відносини "відбулося перед" в обох потоках. Я додав кілька ітерацій циклу while з читачем потоку. Це може здатися досить інтуїтивним: звісно, операція, яка записує значення, відбувається перед операцією, яка читає це значення! Зі стандартними атомними операціями це дійсно так (це і є причина за замовчуванням), але це потрібно вказати: атомні операції також мають інші варіанти вимог порядку, про які я скоро розповім. Тепер, коли ви побачили, як працюють "відбулося перед" і "синхронізується з", настав час подивитися на те, що вони насправді означають. Я почну з "синхронізується з".



***5.3.1 The synchronizes-with relationship***

Відносина "синхронізується з" - це щось, що можна отримати лише між операціями з атомними типами. Операції над структурою даних (наприклад, блокування мютекса) можуть надати цю відносину, якщо структура даних містить атомні типи і операції над цією структурою даних виконують відповідні атомні операції всередині неї, але фундаментально вона виникає лише від операцій над атомними типами.

Основна ідея полягає в наступному: відповідно позначена атомна операція запису W змінної x синхронізується з відповідно позначеною атомною операцією читання x, яка читає значення, збережене або цією запису (W), або подальшою атомною операцією запису x цим самим потоком, який виконав початковий запис W, або послідовністю атомних операцій читання-зміни-запису в x (наприклад, fetch\_add() або compare\_exchange\_weak()) будь-яким потоком, де значення, прочитане першим потоком в цій послідовності, є значенням, записаним W (див. розділ 5.3.4).

Залиште поки осторонь частину "відповідно позначеної", оскільки всі операції з атомними типами за замовчуванням є відповідно позначеними. Це, в суті, означає те, що можна очікувати: якщо потік A зберігає значення, і потік B читає це значення, між зберіганням у потоці A та читанням у потоці B існує відносина "синхронізується з", так само, як у лістингу 5.2.

Я впевнений, що ви вже здогадалися, що всі нюанси полягають в частині "відповідно позначеної". Модель пам'яті C++ дозволяє застосовувати різні обмеження порядку до операцій над атомними типами, і це є позначенням, на яке я посилаюся. Різні опції для впорядкування пам'яті та їх відносини з відносиною "синхронізується з" розглядаються в розділі 5.3.3. Спочатку давайте відійдемо назад і розглянемо відносину "відбулося перед".

***5.3.2 The happens-before relationshipA screenshot of a computer code

Description automatically generated***

A computer screen shot of text

Description automatically generated

Відносина "відбулося перед" є основним будівельним блоком упорядкування операцій у програмі; вона вказує, які операції бачать ефекти інших операцій. Для одного потоку це великою мірою є досить простим: якщо одна операція відбувається перед іншою в послідовності виконання програми, то вона також відбувається перед нею. Це означає, що якщо одна операція (A) відбувається у виразі перед іншою (B) у вихідному коді, то A відбувається перед B. Ви бачили це в лістингу 5.2: запис в data d відбувається перед записом в data\_ready e.

Якщо операції відбуваються в одному виразі, взагалі немає відносини "відбулося перед" між ними, оскільки вони не впорядковані. Це лише інший спосіб сказати, що порядок не визначений. Ви знаєте, що програма в наступному лістингу виведе "1,2" або "2,1", але не визначено, яке саме, оскільки порядок двох викликів get\_num() визначено не визначено.

В багатопотоковому програмуванні розуміння відносин між операціями є важливим для уникнення гонок даних та забезпечення правильності вашого коду. Концепції **послідовно-передує**, **міжпотоковий порядок відбувається перед**, та **синхронізується із** відіграють важливу роль у встановленні порядку операцій між потоками.

**Послідовно-передує** - це відношення, яке виникає у межах одного потоку. Якщо одна операція A передує іншій операції B у межах того самого потоку, то A передує B. Це відношення є властивістю порядку операцій, вказаних у вашому коді.

**Міжпотоковий порядок відбувається перед** - це відношення між операціями з різних потоків. Це фундаментальна концепція, яка допомагає забезпечити порядок операцій між потоками. Якщо операція A на одному потоці синхронізується з операцією B на іншому потоці, то A міжпотоковий порядок відбувається перед B. Це відношення забезпечує видимість змін, здійснених операцією A, операції B.

Відношення **синхронізується із** встановлює прямий міжпотоковий порядок відбувається перед. Якщо операція A на одному потоці синхронізується із операцією B на іншому потоці, то A міжпотоковий порядок відбувається перед B. Це відношення забезпечує видимість змін, здійснених операцією A, операції B.

**Транзитивність** є важливою властивістю цих відносин. Якщо A міжпотоковий порядок відбувається перед B і B міжпотоковий порядок відбувається перед C, то A міжпотоковий порядок відбувається перед C. Ця транзитивна властивість допомагає встановити послідовний порядок подій в багатопотоковому середовищі.

Крім цих відносин, теги порядку пам'яті використовуються в атомарних операціях для подальшої уточнення порядку операцій між потоками. Ці теги надають явні інструкції щодо того, як повинен відбуватися доступ до пам'яті щодо атомарних операцій. Теги порядку пам'яті включають в себе:

* **memory\_order\_relaxed**: Вказує відсутність обмежень порядку. Це забезпечує атомарність, але дозволяє перенос неатомарних операцій навколо атомарної операції.
* **memory\_order\_acquire**: Забезпечує те, що пам'ять, яку зчитують після атомарної операції у програмному порядку, не можна переносити перед атомарною операцією. Цей тег часто використовується з операціями зчитування.
* **memory\_order\_release**: Забезпечує те, що пам'ять, яку записують до атомарної операції у програмному порядку, не можна переносити після атомарної операції. Цей тег часто використовується з операціями запису.
* **memory\_order\_acq\_rel**: Забезпечує як зчитування, так і запис. Цей тег забезпечує відповідність порядку для зчитувань і записів пам'яті навколо атомарної операції.
* **memory\_order\_seq\_cst**: Вказує послідовну консистентність, забезпечуючи загальний порядок операцій, який відповідає програмному порядку. Цей тег надає найсильніші гарантії порядку, але може призвести до погіршення продуктивності на деяких платформах через більші обмеження.

Розуміння цих відносин та тегів порядку пам'яті є важливим для написання правильних та ефективних багатопотокових програм. Вони надають необхідні інструменти для координації дій між потоками, запобігання гонкам даних та забезпечення очікуваної поведінки вашого конкурентоспроможного коду.

***5.3.3 Memory ordering for atomic operations***

Є шість опцій впорядкування пам'яті, які можна застосовувати до операцій на атомарних типах в C++: **memory\_order\_relaxed**, **memory\_order\_consume**, **memory\_order\_acquire**, **memory\_order\_release**, **memory\_order\_acq\_rel** та **memory\_order\_seq\_cst**. Ці опції представляють три відмінні моделі впорядкування пам'яті: послідовно узгоджене впорядкування (**memory\_order\_seq\_cst**), впорядкування при здобутті-вивільненні (**memory\_order\_consume**, **memory\_order\_acquire**, **memory\_order\_release**, **memory\_order\_acq\_rel**) та спрощене впорядкування (**memory\_order\_relaxed**).

1. **Послідовно Узгоджене Впорядкування (memory\_order\_seq\_cst):**
   * Забезпечує найстрогіше впорядкування операцій.
   * Гарантує загальний порядок всіх операцій, роблячи всі операції видимими для всіх потоків в одному глобальному порядку.
   * Надає найвищий рівень синхронізації та послідовності.
   * За замовчуванням встановлено для атомарних операцій, якщо вони не вказані явно.
2. **Впорядкування при Здобутті-Вивільненні (memory\_order\_consume, memory\_order\_acquire, memory\_order\_release, memory\_order\_acq\_rel):**
   * Пропонує проміжні гарантії між послідовно узгодженим та спрощеним впорядкуванням.
   * Операції здобуття переконуються, що після них операції стають видимими.
   * Операції вивільнення переконуються, що перед ними операції стали видимими.
   * Операції здобуття-вивільнення (Acq-Rel) поєднують в собі здобуття та вивільнення.
   * Забезпечують баланс між синхронізацією та продуктивністю, що підходить для багатьох сценаріїв.
3. **Спрощене Впорядкування (memory\_order\_relaxed):**
   * Надає найменші гарантії впорядкування.
   * Не налагає жодних обмежень синхронізації чи порядку на інші читання або записи.
   * Має найкращу продуктивність, але втрачає більшість гарантій синхронізації.
   * Корисне, коли потрібна атомарність, але немає строгих вимог до порядку.

Вибір моделі впорядкування пам'яті може впливати на продуктивність на різних архітектурах процесорів. Системи з дрібним керуванням видимістю можуть вимагати додаткових інструкцій синхронізації для послідовно узгодженого чи впорядкованого впорядкування та для спрощеного впорядкування. На процесорах, таких як x86 чи x86-64 (які поширені в настільних ПК), впорядкування при здобутті-вивільненні не вносить додаткових витрат, окрім вимог до атомарності. Послідовно узгоджене впорядкування вносить невеликі додаткові витрати на записи, але його легше розуміти.

Експерти можуть вибирати конкретні моделі впорядкування пам'яті з урахуванням вимог щодо продуктивності та розуміння взаємозв'язків. Послідовно узгоджене впорядкування є за замовчуванням і легше для використання, а впорядкування при здобутті-вивільненні забезпечує гарний баланс між продуктивністю та гарантіями синхронізації. Спрощене впорядкування найефективніше, але надає мінімальні гарантії синхронізації.

Розуміння цих опцій дозволяє програмістам приймати обдумані рішення про вибір моделі, збалансовані вимоги до синхронізації та продуктивності у багатопотоковому коді.

SEQUENTIALLY CONSISTENT ORDERING

За замовчуванням впорядкування називається "послідовно узгодженим", оскільки це передбачає, що поведінка програми узгоджується з простим послідовним виглядом світу. Якщо всі операції над атомарними типами є послідовно узгодженими, то поведінка багатопотокової програми виглядає так, ніби всі ці операції були виконані у певній послідовності одним потоком. Це найпростіша модель впорядкування пам'яті для розуміння, і саме тому вона є за замовчуванням: всі потоки повинні бачити один і той самий порядок операцій. Це спрощує аналіз поведінки коду, написаного з використанням атомарних змінних. Ви можете записати всі можливі послідовності операцій різними потоками, вилучити ті, що несумісні, і перевірити, що ваш код веде себе так, як очікується в інших випадках. Це також означає, що операції не можуть бути перевпорядковані; якщо ваш код має одну операцію перед іншою в одному потоці, це впорядкування повинно бути бачене всіма іншими потоками.

З точки зору синхронізації, послідовно узгоджене збереження синхронізується з послідовно узгодженим завантаженням тієї ж самої змінної, яке читає збережене значення. Це надає одне впорядкування для операцій двох (або більше) потоків, але послідовна узгодженість діє більш потужно. Будь-які послідовно узгоджені атомарні операції, які виконуються після цього завантаження, також повинні з'являтися після збереження для інших потоків у системі, які використовують послідовно узгоджені атомарні операції. Приклад у лістингу 5.4 показує це впорядкування в дії. Це обмеження не поширюється на потоки, які використовують атомарні операції зі спрощеним порядком пам'яті; вони можуть бачити операції в іншому порядку, тому вам слід використовувати послідовно узгоджені операції на всіх ваших потоках для отримання переваги.

Однак ця легкість розуміння може мати свою ціну. На слабко узгоджених машинах з великою кількістю процесорів це може призвести до помітного падіння продуктивності, оскільки загальна послідовність операцій повинна залишатися стійкою між процесорами, можливо, вимагаючи обширних (і дорогих!) операцій синхронізації між процесорами. Проте деякі архітектури процесорів (такі як поширені архітектури x86 і x86-64) пропонують послідовну узгодженість відносно дешево, тому якщо ви хвилюєтеся через вплив використання послідовно узгодженого впорядкування на продуктивність, перевірте документацію для архітектур вашого цільового процесора.

Наступний лістинг показує послідовно узгоджене впорядкування в дії. Завантаження та збереження x та y явно позначені як memory\_order\_seq\_cst, хоча в цьому випадку цей тег можна було б пропустити, оскільки він є за замовчуванням.

A screenshot of a computer program

Description automatically generated