課程 4.1：共享資料的問題

第四階段：共享資料與競爭條件

課程 4.1：共享資料的問題

引言

多執行緒程式設計最大的挑戰在於**共享資料**。當多個執行緒同時存取同一份資料，且至少有一個執行緒進行寫入時，問題就會產生。本課深入分析這個問題的本質。

一、共享資料的類型

```
┌──────────────────────────────────────────────────────────┐
│ 共享資料的來源 │
├──────────────────────────────────────────────────────────┤
│ │
│ • 全域變數 │
│ • 靜態變數 │
│ • 堆積上的物件（透過指標/引用共享） │
│ • 類別成員變數（多執行緒存取同一物件） │
│ │
└──────────────────────────────────────────────────────────┘
```

二、讀取 vs 寫入

```
┌──────────────────────────────────────────────────────────┐
│ 存取類型與安全性 │
├──────────────────────────────────────────────────────────┤
│ │
│ 多執行緒同時讀取 → 安全 ✓ │
│ 一個寫入 + 一個讀取 → 不安全 ✗（資料競爭） │
│ 多執行緒同時寫入 → 不安全 ✗（資料競爭） │
│ │
└──────────────────────────────────────────────────────────┘
```

三、問題展示：計數器

```
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;

void increment() {
for (int i = 0; i < 100000; ++i) {
++counter;
}
}

int main() {
std::vector<std::thread> threads;
for (int i = 0; i < 10; ++i) {
threads.emplace_back(increment);
}
for (auto& t : threads) {
t.join();
}
std::cout << "預期: 1000000" << std::endl;
std::cout << "實際: " << counter << std::endl;
return 0;
}
```

輸出（每次不同）：

```
預期: 1000000
實際: 387432
```

四、為什麼會出錯？

++counter 看起來是一個操作，實際上是三步：

```
┌──────────────────────────────────────────────────────────────┐
│ ++counter 的真實步驟 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 1. 讀取：從記憶體讀取 counter 的值到 CPU 暫存器 │
│ 2. 修改：在暫存器中將值 +1 │
│ 3. 寫入：將暫存器的值寫回記憶體 │
│ │
│ 這三步不是原子的，可能被其他執行緒打斷！ │
│ │
└──────────────────────────────────────────────────────────────┘
```

五、交錯執行的災難

```
時間 執行緒A 執行緒B counter（記憶體）
──────────────────────────────────────────────────────────────
1 讀取 counter (0) 0
2 讀取 counter (0) 0
3 +1 得到 1 0
4 +1 得到 1 0
5 寫回 1 1
6 寫回 1 1
──────────────────────────────────────────────────────────────
```

結果：兩次 ++，但 counter 只變成 1！

六、更複雜的例子：銀行轉帳

```
#include <iostream>
#include <thread>

struct Account {
int balance = 1000;
};
```

```
Account accountA, accountB;

void transfer(Account& from, Account& to, int amount) {
if (from.balance >= amount) {
from.balance -= amount;
to.balance += amount;
}
}

int main() {
std::thread t1([&]() {
for (int i = 0; i < 1000; ++i) {
transfer(accountA, accountB, 1);
}
});
std::thread t2([&]() {
for (int i = 0; i < 1000; ++i) {
transfer(accountB, accountA, 1);
}
});
t1.join();
t2.join();
int total = accountA.balance + accountB.balance;
std::cout << "A: " << accountA.balance << std::endl;
std::cout << "B: " << accountB.balance << std::endl;
std::cout << "總額: " << total << " (應為 2000)" << std::endl;
return 0;
}
```

可能輸出：

```
A: 1042
B: 1036
總額: 2078 (應為 2000)
```

錢憑空產生了！這是嚴重的資料損毀。

七、資料競爭的定義

根據 C++ 標準，**資料競爭（Data Race）** 發生於：

```
┌──────────────────────────────────────────────────────────┐
│ 資料競爭的條件 │
├──────────────────────────────────────────────────────────┤
│ │
│ 同時滿足以下條件： │
│ │
│ 1. 兩個或多個執行緒同時存取同一記憶體位置 │
│ 2. 至少有一個是寫入操作 │
│ 3. 沒有同步機制保護 │
│ │
│ 結果：未定義行為（Undefined Behavior） │
│ │
└──────────────────────────────────────────────────────────┘
```

八、不只是錯誤的結果

資料競爭導致的是**未定義行為**，可能出現：

- 錯誤的計算結果
- 程式崩潰
- 資料損毀
- 看似正確但偶爾出錯（最難除錯）
- 在某些機器正常，換台機器就出錯

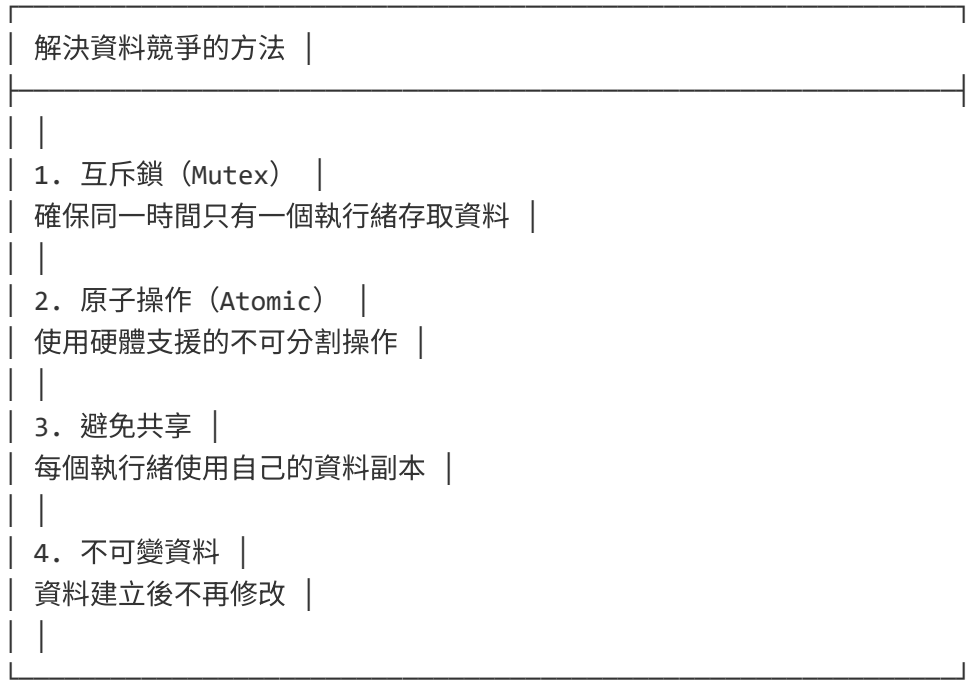
九、哪些操作是安全的？

```
#include <iostream>
#include <thread>

const int readOnlyData = 42; // 唯讀資料

void reader() {
// 安全：只有讀取，沒有寫入
std::cout << readOnlyData << std::endl;
}

int main() {
```

```
std::thread t2(reader);
t1.join();
t2.join();
return 0;
}
```

規則：只要沒有寫入，多執行緒讀取是安全的。

十、解決方案預覽



十一、本課重點回顧

1. 多執行緒存取共享資料是危險的
2. 讀取 + 寫入或多重寫入會造成資料競爭
3. `++counter` 不是原子操作，由三個步驟組成
4. 資料競爭導致**未定義行為**
5. 只有純讀取是安全的
6. 解決方案：互斥鎖、原子操作、避免共享

下一課預告
在 **課程 4.2：不變量與競爭條件** 中，我們將學習：

- 什麼是不變量（Invariant）
- 不變量被破壞的情況
- 如何維護資料的一致性

準備好繼續嗎？

課程 4.2：不變量與競爭條件

第四階段：共享資料與競爭條件

課程 4.2：不變量與競爭條件

引言

要理解競爭條件為何危險，必須先理解**不變量（Invariant）**的概念。不變量是資料結構必須永遠保持為真的條件，競爭條件的危害在於它會破壞不變量。

一、什麼是不變量

```
┌──────────────────────────────────────────────────────────┐
│ 不變量（Invariant） │
├──────────────────────────────────────────────────────────┤
│ │
│ 定義：資料結構在任何「可觀察」時刻都必須滿足的條件 │
│ │
│ 例子： │
│ • 雙向鏈結串列：A.next = B 則 B.prev = A │
│ • 銀行帳戶：轉帳前後總金額不變 │
│ • 二元搜尋樹：左子節點 < 父節點 < 右子節點 │
│ • 堆疊：size 等於實際元素數量 │
│ │
└──────────────────────────────────────────────────────────┘
```

二、不變量在操作中的暫時破壞

進行複合操作時，不變量可能**暫時**被破壞：

```
// 雙向鏈結串列插入節點
// 不變量：A.next->prev == A

// 初始狀態：A <-> C
// 目標：A <-> B <-> C

// 步驟 1：B.next = C
// A -> C B -> C 不變量暫時破壞！
// A <- C B.prev 還沒設定

// 步驟 2：B.prev = A
// A -> C B -> C
// A <- C B <- A 不變量仍破壞！

// 步驟 3：A.next = B
// A -> B -> C
// A <- C B <- A 不變量仍破壞！

// 步驟 4：C.prev = B
// A -> B -> C
// A <- B <- C 不變量恢復 ✓
```

三、單執行緒下沒問題

```
#include <iostream>

struct Node {
int data;
Node* prev = nullptr;
Node* next = nullptr;
};

void insertAfter(Node* a, Node* newNode, Node* c) {
// 不變量暫時被破壞，但沒關係
// 因為沒有其他人會看到中間狀態
```

```
newNode->next = c;
newNode->prev = a;
a->next = newNode;
c->prev = newNode;
// 不變量恢復
}

int main() {
Node a{1}, b{2}, c{3};
a.next = &c;
c.prev = &a;
insertAfter(&a, &b, &c); // 安全：單執行緒
std::cout << a.next->data << std::endl; // 2
return 0;
}
```

四、多執行緒下的災難

```
#include <iostream>
#include <thread>

struct Node {
int data;
Node* prev = nullptr;
Node* next = nullptr;
};

Node a{1}, b{2}, c{3};

void writer() {
// 插入 b 到 a 和 c 之間
b.next = &c;
b.prev = &a;
// ← 此時另一個執行緒可能讀取！
a.next = &b;
c.prev = &b;
}
```

```
void reader() {
// 嘗試遍歷鏈結串列
Node* current = &a;
while (current != nullptr) {
std::cout << current->data << " ";
current = current->next;
}
std::cout << std::endl;
}

int main() {
a.next = &c;
c.prev = &a;
std::thread t1(writer);
std::thread t2(reader);
t1.join();
t2.join();
return 0;
}
```

可能的輸出：

```
1 3 // 正常（在修改前讀取）
1 2 3 // 正常（在修改後讀取）
1 // 異常！讀到中間狀態
```

五、銀行帳戶的不變量

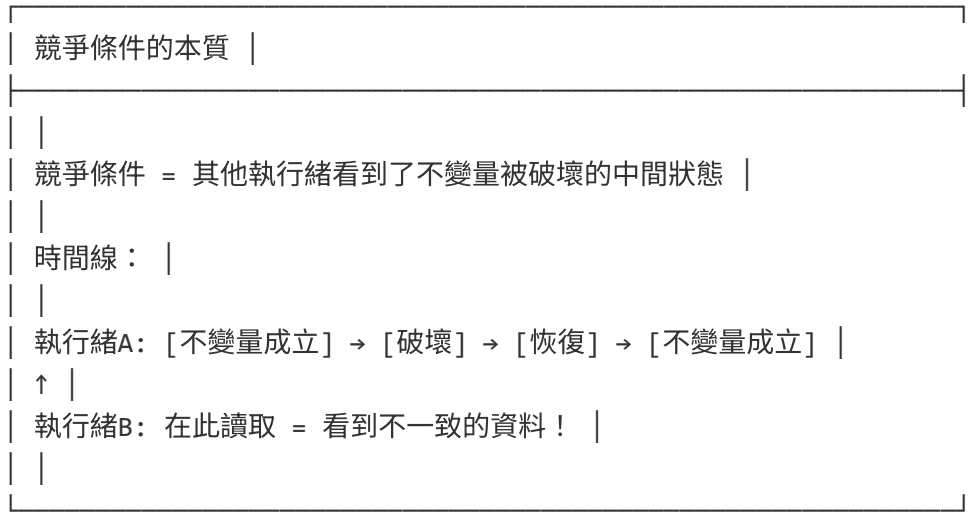
```
#include <iostream>
#include <thread>

struct Bank {
int accountA = 1000;
int accountB = 1000;
// 不變量：accountA + accountB == 2000
};
```

```
Bank bank;

void transfer(int amount) {
// 不變量暫時破壞
bank.accountA -= amount;
// ← 此刻總額不是 2000！
bank.accountB += amount;
// 不變量恢復
}

void audit() {
int total = bank.accountA + bank.accountB;
if (total != 2000) {
std::cout << "警告！總額異常: " << total << std::endl;
}
}

int main() {
std::thread t1([]() {
for (int i = 0; i < 1000; ++i) transfer(1);
});
std::thread t2([]() {
for (int i = 0; i < 1000; ++i) audit();
});
t1.join();
t2.join();
return 0;
}
```

可能輸出：

```
警告！總額異常: 1999
警告！總額異常: 2001
警告！總額異常: 1999
```

六、競爭條件的本質



七、常見的不變量破壞場景

場景一：容器的 size

```
#include <vector>
#include <thread>

std::vector<int> vec;
// 不變量：vec.size() == 實際元素數量

void unsafeAdd(int value) {
vec.push_back(value); // 可能破壞內部不變量
}
```

場景二：物件的狀態

```
class Connection {
bool connected = false;
int socket = -1;
// 不變量：connected == true 時，socket >= 0
public:
void connect() {
socket = openSocket(); // socket 已設定
// ← 此刻 connected 還是 false！
```

```
connected = true;
}
};
```

八、如何保護不變量

```
┌──────────────────────────────────────────────────────────────┐
│ 保護不變量的方法 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 1. 互斥鎖 │
│ 讓整個操作成為原子，其他執行緒看不到中間狀態 │
│ │
│ 2. 原子操作 │
│ 對於簡單資料，使用硬體保證的原子操作 │
│ │
│ 3. 事務性操作 │
│ 先準備好所有資料，最後一步原子切換 │
│ │
└──────────────────────────────────────────────────────────────┘
```

九、使用互斥鎖保護（預覽）

```
#include <iostream>
#include <thread>
#include <mutex>

struct Bank {
int accountA = 1000;
int accountB = 1000;
std::mutex mtx;
};

Bank bank;
```

```
void transfer(int amount) {
std::lock_guard<std::mutex> lock(bank.mtx);
// 不變量暫時破壞，但沒人看得到
bank.accountA -= amount;
bank.accountB += amount;
// 不變量恢復
}

void audit() {
std::lock_guard<std::mutex> lock(bank.mtx);
int total = bank.accountA + bank.accountB;
std::cout << "總額: " << total << std::endl; // 永遠是 2000
}
```

十、本課重點回顧

1. **不變量**是資料結構必須保持為真的條件
2. 複合操作會**暫時破壞**不變量
3. 單執行緒下暫時破壞沒問題，因為沒人看到
4. 多執行緒下，其他執行緒可能看到破壞的中間狀態
5. **競爭條件**的本質是看到了不一致的資料
6. 解決方案：讓整個操作對外呈現為原子

下一課預告

在 **課程 4.3：臨界區段概念** 中，我們將學習：

- 什麼是臨界區段（Critical Section）
- 如何識別需要保護的程式碼
- 臨界區段的設計原則

準備好繼續嗎？

課程 4.3：臨界區段概念

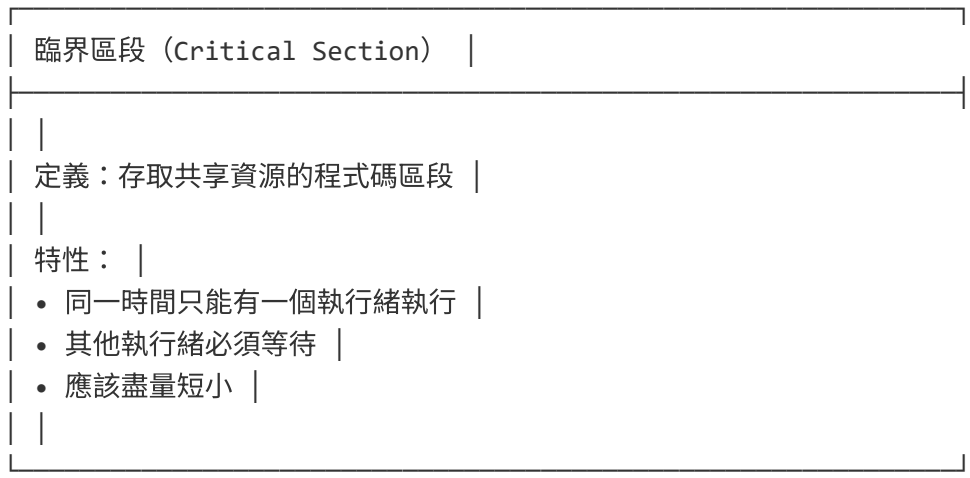
第四階段：共享資料與競爭條件

課程 4.3：臨界區段概念

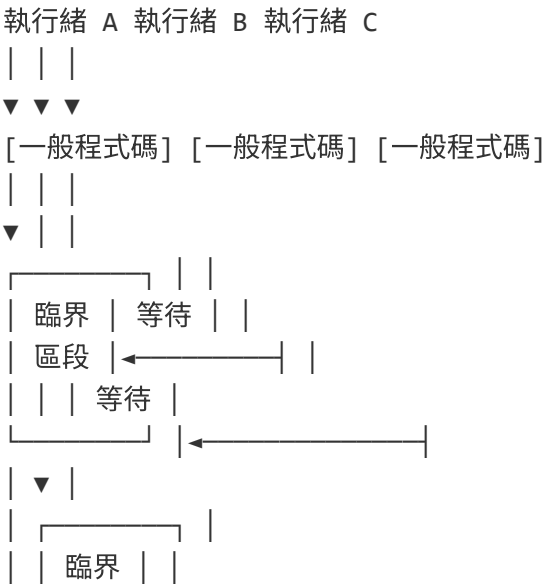
引言

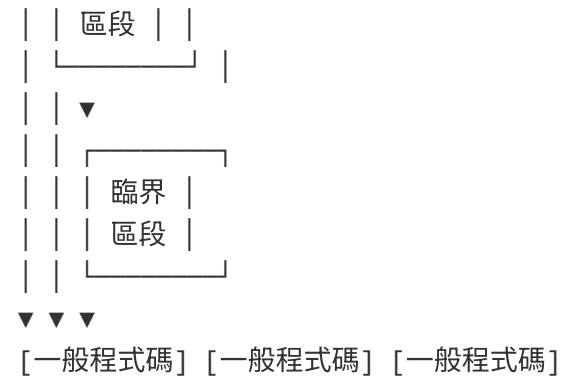
既然我們知道共享資料存取會造成問題，下一步就是識別哪些程式碼需要保護。**這些需要保護的程式碼區域稱為臨界區段（Critical Section）**。

一、臨界區段的定義



二、圖解臨界區段





三、識別臨界區段

```
#include <iostream>
#include <thread>

int sharedData = 0;

void worker() {
int localVar = 0; // 不是臨界區段：區域變數
localVar = 42; // 不是臨界區段：只存取區域變數
sharedData = localVar; // ← 臨界區段：寫入共享資料
int temp = sharedData; // ← 臨界區段：讀取共享資料（如有寫入者）
std::cout << temp; // 可能是臨界區段：cout 也是共享資源
}
```

四、臨界區段的範圍

範例：太大的臨界區段（不好）

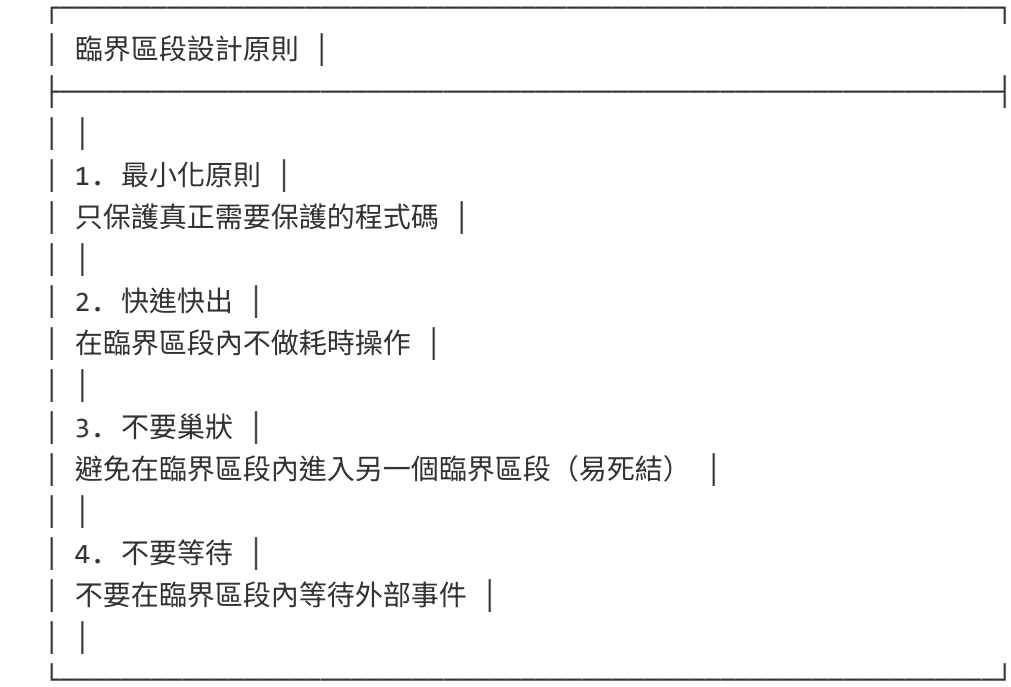
```
void badExample() {
lock();
// 臨界區段開始
int result = complexCalculation(); // 不需要鎖！
sharedData = result; // 這才需要鎖
logToFile(result); // 不需要鎖！
// 臨界區段結束
```

```
unlock();
}
```

範例：精確的臨界區段（好）

```
void goodExample() {
int result = complexCalculation(); // 在鎖外計算
lock();
sharedData = result; // 只保護必要的部分
unlock();
logToFile(result); // 在鎖外記錄
}
```

五、臨界區段設計原則



六、識別共享資源

```
#include <iostream>
#include <thread>
#include <vector>

// 共享資源
int globalCounter = 0; // 全域變數
static int staticCounter = 0; // 靜態變數
std::vector<int> sharedVector; // 全域容器

class MyClass {
int memberVar; // 若多執行緒存取同一物件
static int staticMember; // 靜態成員
public:
void method() {
// memberVar 是否為共享資源？
// 取決於是否多執行緒存取同一個 MyClass 物件
}
};

void function() {
int localVar = 0; // 不是共享資源
static int localStatic = 0; // 是共享資源！
thread_local int tlVar = 0; // 不是共享資源
}
```

七、常見的共享資源

資源類型	是否共享	需要保護？
全域變數	是	是
函式內 static 變數	是	是
類別 static 成員	是	是
堆積上的物件（共用指標）	是	是
區域變數	否	否
thread_local 變數	否	否
函式參數（傳值）	否	否
const 全域變數	是	否（唯讀）

八、程式碼標記練習

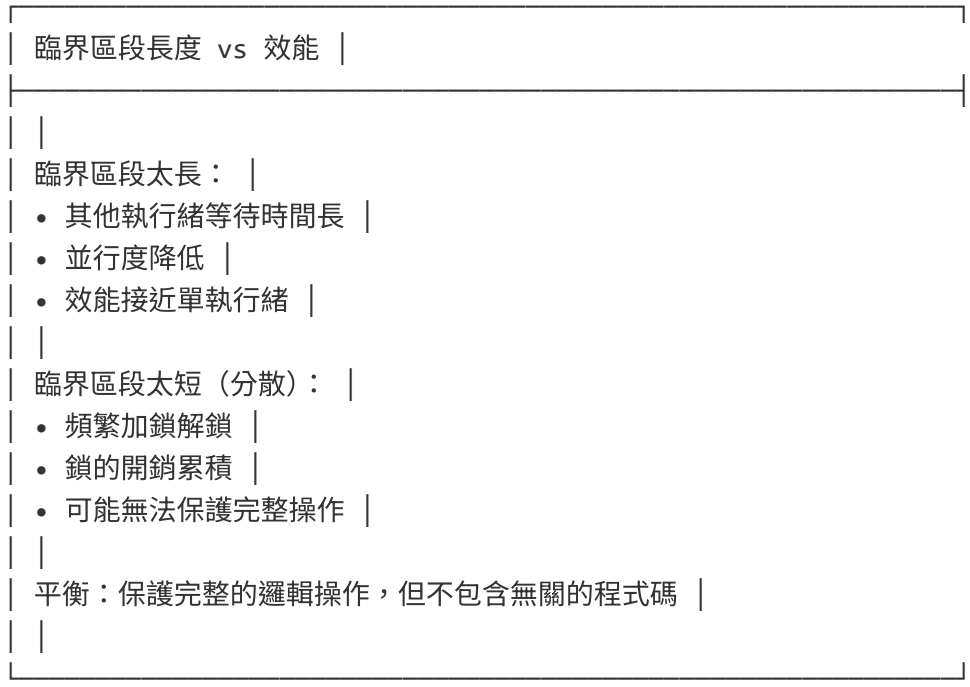
```
#include <thread>

int shared = 0;

void example(int param) {
int local = param; // A
local += 10; // B
shared = local; // C
local = shared; // D
int result = local * 2; // E
shared += result; // F
}
```

哪些是臨界區段？

A：不是（只存取區域變數和參數）
B：不是（只存取區域變數）
C：是（寫入共享資料）

D: 是（讀取共享資料，且有其他寫入者）
E: 不是（只存取區域變數）
F: 是（讀取 + 寫入共享資料）

九、臨界區段與效能



十、實際案例分析

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

std::vector<int> data;
std::mutex mtx;

// 不好：臨界區段太大
```

```
void badPush(int value) {
std::lock_guard<std::mutex> lock(mtx);
int processed = value * value; // 不需要鎖
data.push_back(processed); // 需要鎖
std::cout << "Added: " << processed; // 可能不需要鎖
}

// 好：臨界區段精確
void goodPush(int value) {
int processed = value * value; // 鎖外計算
{
std::lock_guard<std::mutex> lock(mtx);
data.push_back(processed); // 只保護必要操作
}
std::cout << "Added: " << processed; // 鎖外輸出
}
```

十一、本課重點回顧

1. **臨界區段**是存取共享資源的程式碼區域
2. 同一時間只能有一個執行緒執行臨界區段
3. 臨界區段應該**盡量短小**
4. 只保護真正需要保護的程式碼
5. 避免在臨界區段內做耗時操作
6. 區域變數和 thread_local 變數不需要保護

下一課預告

在 **課程 4.4：資料競爭範例分析** 中，我們將：

- 分析更多真實世界的資料競爭案例
- 學習如何發現潛在的競爭條件
- 探討競爭條件的除錯技巧

準備好繼續嗎？

課程 4.4：資料競爭範例分析

第四階段：共享資料與競爭條件

課程 4.4：資料競爭範例分析

引言

本課透過多個實際案例，分析資料競爭如何發生，以及如何識別程式碼中潛在的競爭條件。

一、案例一：Check-Then-Act 競爭

最常見的競爭模式：先檢查條件，再根據結果行動。

```
#include <iostream>
#include <thread>
#include <map>

std::map<int, std::string> cache;

// 危險！Check-Then-Act 競爭
std::string getValue(int key) {
if (cache.find(key) == cache.end()) { // 檢查
// ← 另一執行緒可能在此插入相同 key！
cache[key] = "computed_" + std::to_string(key); // 行動
}
return cache[key];
}

int main() {
std::thread t1([]() { getValue(1); });
std::thread t2([]() { getValue(1); });
t1.join();
t2.join();
return 0;
}
```

問題：兩個執行緒都可能認為 key 不存在，然後都嘗試插入。

二、案例二：Read-Modify-Write 競爭

讀取、修改、寫回的複合操作。

```
#include <iostream>
```

```
#include <thread>

int counter = 0;

void increment() {
// 這三步不是原子的！
// 1. 讀取 counter
// 2. +1
// 3. 寫回
for (int i = 0; i < 10000; ++i) {
counter++; // 非原子操作
}
}

int main() {
std::thread t1(increment);
std::thread t2(increment);
t1.join();
t2.join();
// 預期 20000，實際可能更小
std::cout << "counter = " << counter << std::endl;
return 0;
}
```

三、案例三：複合操作競爭

看似獨立的操作，實際上有關聯。

```
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> vec;

void unsafeAppend(int value) {
if (vec.size() < 10) { // 檢查
// ← 另一執行緒可能在此改變 size！
vec.push_back(value); // 可能超過限制
}
```

```
}

void unsafeAccess() {
if (!vec.empty()) { // 檢查
// ← 另一執行緒可能清空 vec！
int last = vec.back(); // 可能崩潰
std::cout << last << std::endl;
}
}
```

四、案例四：迭代器失效

```
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> data = {1, 2, 3, 4, 5};

void reader() {
for (auto it = data.begin(); it != data.end(); ++it) {
// ← 另一執行緒可能修改 data，導致迭代器失效！
std::cout << *it << " ";
}
}

void writer() {
data.push_back(6); // 可能導致重新配置，所有迭代器失效
}

int main() {
std::thread t1(reader);
std::thread t2(writer);
t1.join();
t2.join();
return 0;
}
```

五、案例五：延遲初始化

```
#include <iostream>
#include <thread>

class Singleton {
static Singleton* instance;
public:
// 危險！雙重檢查鎖定的錯誤實作
static Singleton* getInstance() {
if (instance == nullptr) { // 第一次檢查
// ← 多執行緒可能同時通過這裡！
instance = new Singleton(); // 可能建立多個實例
}
return instance;
}
};

Singleton* Singleton::instance = nullptr;
```

六、案例六：共享物件的部分更新

```
#include <iostream>
#include <thread>
#include <string>

struct Person {
std::string firstName;
std::string lastName;
int age;
};

Person person{"John", "Doe", 30};

void writer() {
// 更新不是原子的
```

```
person.firstName = "Jane";
// ← 此刻資料不一致！
person.lastName = "Smith";
person.age = 25;
}

void reader() {
// 可能讀到 "Jane Doe 30" 這種不一致狀態
std::cout << person.firstName << " "
<< person.lastName << " "
<< person.age << std::endl;
}
```

七、競爭條件識別清單

```
┌──────────────────────────────────────────────────────────┐
│ 競爭條件警示信號 │
├──────────────────────────────────────────────────────────┤
│ │
│ ⚠️ if (condition) { action } │
│ 條件和行動之間狀態可能改變 │
│ │
│ ⚠️ variable++, variable += x │
│ Read-Modify-Write 不是原子的 │
│ │
│ ⚠️ 讀取多個相關變數 │
│ 可能讀到不一致的組合 │
│ │
│ ⚠️ 迭代容器時修改容器 │
│ 迭代器可能失效 │
│ │
│ ⚠️ 物件狀態的部分更新 │
│ 可能讀到中間狀態 │
│ │
└──────────────────────────────────────────────────────────┘
```

八、如何發現競爭條件

方法一：程式碼審查

問自己：

- 這個變數是否被多執行緒存取？
- 有沒有寫入操作？
- 操作是否原子？

方法二：使用工具

```
# ThreadSanitizer（你已熟悉）
g++ -fsanitize=thread -g -o program program.cpp
./program
```

方法三：壓力測試

```
// 大量重複執行，增加發現問題的機會
for (int i = 0; i < 10000; ++i) {
runTest();
}
```

九、ThreadSanitizer 輸出範例

```
// 編譯：g++ -fsanitize=thread -g -o race race.cpp -pthread

#include <thread>

int counter = 0;

int main() {
std::thread t1([]() { counter++; });
std::thread t2([]() { counter++; });
t1.join();
t2.join();
return 0;
}
```

TSan 輸出：

```
WARNING: ThreadSanitizer: data race (pid=12345)
Write of size 4 at 0x000000601040 by thread T2:
#0 main::$_1::operator()() race.cpp:8
Previous write of size 4 at 0x000000601040 by thread T1:
#0 main::$_0::operator()() race.cpp:7
Location is global 'counter' of size 4 at 0x000000601040
```

十、競爭條件的特性

```
┌──────────────────────────────────────────────────────────┐
│ 競爭條件為何難以除錯 │
├──────────────────────────────────────────────────────────┤
│ │
│ 1. 非確定性 │
│ 同樣的程式碼，每次執行結果可能不同 │
│ │
│ 2. 時機敏感 │
│ 只在特定時序下發生 │
│ │
│ 3. 難以重現 │
│ 加上 printf 除錯可能改變時序，問題消失 │
│ │
│ 4. 環境依賴 │
│ 在某台機器正常，換台機器就出錯 │
│ │
└──────────────────────────────────────────────────────────┘
```

十一、本課重點回顧

1. **Check-Then-Act**：檢查和行動之間狀態可能改變
2. **Read-Modify-Write**：複合操作不是原子的
3. **複合操作**：多個相關操作必須一起保護
4. **迭代器失效**：修改容器時迭代器可能無效
5. **部分更新**：物件可能處於不一致狀態
6. **TSan** 是發現競爭條件的有力工具

下一課預告

在 **課程 4.5：競爭條件的檢測** 中，我們將深入學習：

- ThreadSanitizer 的詳細使用
- 靜態分析工具
- 除錯策略與技巧

準備好繼續嗎？

課程 4.5：競爭條件的檢測

第四階段：共享資料與競爭條件

課程 4.5：競爭條件的檢測

引言

競爭條件難以重現和除錯，幸運的是有專門的工具可以幫助我們檢測。本課介紹實用的檢測工具和技巧，特別是你已經熟悉的 ThreadSanitizer。

一、檢測工具總覽

```
┌─────────────────────────────────────────────────────────┐
│ 競爭條件檢測工具 │
├─────────────────────────────────────────────────────────┤
│ │
│ 動態分析（執行時檢測） │
│ • ThreadSanitizer (TSan) – 最常用 │
│ • Helgrind (Valgrind) │
│ • Intel Inspector │
│ │
│ 靜態分析（編譯時檢測） │
│ • Clang Static Analyzer │
│ • Coverity │
│ • PVS-Studio │
│ │
└─────────────────────────────────────────────────────────┘
```

二、ThreadSanitizer 基本使用

```
// 檔案：race_demo.cpp
#include <thread>

int counter = 0;

void increment() {
for (int i = 0; i < 1000; ++i) {
counter++;
}
}

int main() {
std::thread t1(increment);
std::thread t2(increment);
t1.join();
t2.join();
return 0;
}
```

編譯與執行：

```
g++ -std=c++17 -fsanitize=thread -g -o race_demo race_demo.cpp -pthread
./race_demo
```

三、TSan 輸出解讀

```
==================
WARNING: ThreadSanitizer: data race (pid=12345)
Write of size 4 at 0x000000601040 by thread T2:
#0 increment() race_demo.cpp:7 (race_demo+0x...)
#1 void std::__invoke_impl<...>
```

```
Previous write of size 4 at 0x000000601040 by thread T1:
#0 increment() race_demo.cpp:7 (race_demo+0x...)
#1 void std::__invoke_impl<...>

Location is global 'counter' of size 4 at 0x000000601040

SUMMARY: ThreadSanitizer: data race race_demo.cpp:7 in increment()
==================
```

關鍵資訊：

- **Write of size 4**：4 位元組的寫入操作
- **race_demo.cpp:7**：問題發生在第 7 行
- **global 'counter'**：問題變數是全域的 counter

四、TSan 常見報告類型

```
┌──────────────────────────────────────────────────────────┐
│ TSan 報告類型 │
├──────────────────────────────────────────────────────────┤
│ │
│ data race │
│ → 兩個執行緒同時存取，至少一個寫入 │
│ │
│ thread leak │
│ → 執行緒結束前未 join 或 detach │
│ │
│ lock-order-inversion (potential deadlock) │
│ → 鎖的獲取順序不一致，可能死結 │
│ │
│ use of uninitialized mutex │
│ → 使用未初始化的互斥鎖 │
│ │
└──────────────────────────────────────────────────────────┘
```

五、更複雜的案例

```
// 檔案：complex_race.cpp
```

```
#include <iostream>
#include <thread>
#include <vector>

std::vector<int> data;

void producer() {
for (int i = 0; i < 100; ++i) {
data.push_back(i);
}
}

void consumer() {
for (int i = 0; i < 100; ++i) {
if (!data.empty()) {
int val = data.back();
data.pop_back();
}
}
}

int main() {
std::thread t1(producer);
std::thread t2(consumer);
t1.join();
t2.join();
return 0;
}
```

TSan 會報告 vector 內部的多處競爭。

六、TSan 的限制

```
┌──────────────────────────────────────────────────────────────┐
│ TSan 的限制與注意事項 │
├──────────────────────────────────────────────────────────────┤
│ │
│ 效能影響 │
│ • 執行速度慢 5-15 倍 │
│ • 記憶體使用增加 5-10 倍 │
```

```
│ │
│ 檢測限制 │
│ • 只能檢測實際執行到的程式碼 │
│ • 無法檢測所有可能的交錯 │
│ • 需要足夠的測試覆蓋率 │
│ │
│ 使用建議 │
│ • 開發和測試時使用，不用於生產環境 │
│ • 結合壓力測試增加發現機率 │
│ │
└──────────────────────────────────────────────────────────────┘
```

七、Helgrind（Valgrind 工具）

另一個選擇，不需要重新編譯：

```
# 正常編譯
g++ -std=c++17 -g -o race_demo race_demo.cpp -pthread

# 使用 Helgrind 執行
valgrind --tool=helgrind ./race_demo
```

輸出：

```
==12345== Possible data race during write of size 4
==12345== at 0x401234: increment() (race_demo.cpp:7)
==12345== This conflicts with a previous write
==12345== at 0x401234: increment() (race_demo.cpp:7)
```

八、TSan vs Helgrind 比較

特性	TSan	Helgrind
速度	**較快（5-15x）**	**較慢（20-100x）**
需要重新編譯	是	否
誤報率	低	較高
支援 C++11 原子	完整	部分

九、手動檢測技巧

技巧一：插入延遲

```
void suspiciousFunction() {
// 在可疑位置插入延遲，增加競爭發生機率
checkCondition();
std::this_thread::sleep_for(std::chrono::milliseconds(1));
doAction();
}
```

技巧二：壓力測試

```
#include <iostream>
#include <thread>
#include <vector>

void stressTest() {
for (int trial = 0; trial < 1000; ++trial) {
// 重設狀態
counter = 0;
std::vector<std::thread> threads;
for (int i = 0; i < 10; ++i) {
threads.emplace_back(increment);
}
for (auto& t : threads) {
t.join();
}
```

```
std::cout << "競爭檢測！Trial " << trial
<< " counter=" << counter << std::endl;
}
}
}
```

十、檢測清單

```
┌─────────────────────────────────────────────────────────┐
│ 競爭條件檢測清單 │
├─────────────────────────────────────────────────────────┤
│ │
│ □ 使用 TSan 編譯並執行測試 │
│ □ 確保測試覆蓋多執行緒路徑 │
│ □ 進行壓力測試（多次重複執行） │
│ □ 審查所有共享變數的存取 │
│ □ 檢查 Check-Then-Act 模式 │
│ □ 檢查 Read-Modify-Write 操作 │
│ □ 確認所有複合操作都有適當保護 │
│ │
└─────────────────────────────────────────────────────────┘
```

十一、本課重點回顧

1. **ThreadSanitizer** 是最實用的競爭條件檢測工具
2. 使用 `-fsanitize=thread -g` 編譯
3. TSan 會報告資料競爭的位置和相關執行緒
4. TSan 會降低效能，僅用於開發和測試
5. **Helgrind** 不需重新編譯，但較慢
6. 結合**壓力測試**增加發現問題的機率

第四階段完成！

恭喜你完成了共享資料與競爭條件階段！你已經學會：

- ✅ 共享資料的問題本質
- ✅ 不變量與競爭條件的關係

- ✅ 臨界區段的識別與設計
- ✅ 常見競爭條件模式
- ✅ 使用工具檢測競爭條件

下一階段預告

第五階段：互斥鎖基礎 (std::mutex) 將學習如何解決這些問題：

- 課程 5.1：std::mutex 基本操作
- 課程 5.2：互斥鎖的工作原理
- 課程 5.3：try_lock() 非阻塞鎖定
- ...

準備好進入第五階段嗎？