

Section 18 Exception Handling

What is an Exception? (什麼是例外?)

在程式執行期間，當發生錯誤或非預期的狀況（如除以零、開啟檔案失敗、記憶體不足）時，就會拋出一個 **exception**（例外）。例外是一種特殊的物件，用來通知程式發生了錯誤。

```
int divide(int a, int b) {  
    if (b == 0)  
        throw "Division by zero!";  
    return a / b;  
}
```

What is Exception Handling? (什麼是例外處理?)

Exception Handling 是一套機制，讓你能夠：

- 拋出 (throw) 錯誤
 - 捕捉 (catch) 錯誤
 - 處理 (handle) 錯誤
- 避免程式在錯誤時直接中斷執行。

使用三個關鍵字實作：

```
try {  
    // 嘗試執行的程式碼  
} catch (exceptionType e) {  
    // 錯誤處理區塊  
}
```

What do we throw and catch exceptions? (我們拋出與捕捉什麼?)

你可以 `throw` 任何類型 (如 `int` 、 `const char*` 、 `std::string` , 或自定類別)。

```
try {  
    throw std::string("Something went wrong");  
} catch (const std::string& e) {  
    std::cerr << "Caught exception: " << e << '\n';  
}
```

建議：實務上，應該拋出 類別物件，尤其是繼承自 `std::exception` 的類別，以利結構化錯誤處理。

How does it affect flow of control? (它如何影響控制流程?)

當拋出例外時：

- `throw` 會立即離開當前函式
- 程式會尋找對應的 `catch` 區塊來處理例外
- `try` 區塊後的程式碼將不會被執行

```
void test() {  
    std::cout << "Before throw\n";  
    throw 123;  
    std::cout << "This line will NOT run.\n";  
}
```

Defining our own exception classes (自定義例外類別)

自訂錯誤類別可以讓我們建立專屬的錯誤邏輯。做法是 繼承自 `std::exception` 並覆寫 `what()` 方法。

```

#include <exception>
#include <string>

class MyException : public std::exception {
    std::string msg;
public:
    MyException(const std::string& message) : msg(message) {}
    const char* what() const noexcept override {
        return msg.c_str();
    }
};

// 使用
try {
    throw MyException("Custom error occurred");
} catch (const MyException& e) {
    std::cerr << "Caught: " << e.what() << '\n';
}

```

The Standard Library Exception Hierarchy (標準例外階層)

C++ 標準函式庫提供了多個錯誤類別，全部繼承自 `std::exception`。以下是部分結構圖：

```

std::exception
├── std::logic_error
│   ├── std::invalid_argument
│   ├── std::domain_error
│   ├── std::length_error
│   └── std::out_of_range
├── std::runtime_error
│   ├── std::overflow_error
│   ├── std::underflow_error
│   └── std::range_error

```

這些類別讓你針對錯誤型態精細處理：

```
try {
    throw std::out_of_range("Index out of range");
} catch (const std::out_of_range& e) {
    std::cerr << e.what() << '\n';
}
```

`std::exception` and `what()` (例外訊息顯示)

`std::exception` 是所有標準例外類的基底，它定義了一個虛擬函式：

```
virtual const char* what() const noexcept;
```

這個方法會傳回錯誤訊息字串，可用來診斷錯誤：

```
try {
    throw std::runtime_error("File open failed");
} catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << '\n';
}
```

小結：Exception Handling 筆記總結

項目	說明
<code>try</code>	包裹可能拋出例外的區域
<code>throw</code>	拋出例外物件
<code>catch</code>	捕捉特定型別的例外
<code>std::exception</code>	所有標準例外的基底類別
<code>what()</code>	回傳錯誤訊息

非常好的問題！在 C++ 中，從函式中拋出例外（throwing an exception from a function）是 Exception Handling 的典型用法之一。

Throwing an Exception from a Function (從函式中拋出例外)

當函式內發生錯誤而無法正常繼續執行時，可以使用 `throw` 關鍵字來拋出例外，把錯誤狀況「通知」呼叫者處理。這樣做的好處是：

你不需要在錯誤發生的當下處理它，而是可以把控制權交給更高層級的呼叫者。

◆ 基本語法

```
ReturnType functionName(...) {  
    if (error_condition) {  
        throw exception_object; // 可為 string、int、或 exception 類別  
    }  
    // 其他正常流程  
}
```

範例一：拋出文字例外

```
void openFile(const std::string& filename) {  
    if (filename.empty()) {  
        throw "Filename is empty!";  
    }  
    // 假設後續要打開檔案  
}
```

使用方式：

```
try {  
    openFile("");  
}
```

```
} catch (const char* msg) {  
    std::cerr << "Caught exception: " << msg << '\n';  
}
```

範例二：拋出標準例外類別

```
#include <stdexcept>  
  
double divide(double a, double b) {  
    if (b == 0)  
        throw std::runtime_error("Division by zero!");  
    return a / b;  
}
```

使用方式：

```
try {  
    std::cout << divide(10, 0);  
} catch (const std::runtime_error& e) {  
    std::cerr << "Error: " << e.what() << '\n';  
}
```

為什麼這麼做？

1. 更清楚的錯誤分層處理：函式專注於其本職功能，錯誤由呼叫端負責處理。
2. 減少嵌套：不需要每層都加 if/else 處理錯誤。
3. 提升可維護性：不同錯誤類別可以使用不同 `catch` 處理方式。

注意：控制流程會被「中斷」

只要 `throw` 被執行，函式剩下的程式碼不再執行，並且直接跳轉至最近對應的 `catch` 區塊。

補充：函式宣告中可以（但不建議）使用 `noexcept` 或 `throw` 規格

舊語法（C++11 前）：

```
void f() throw(std::runtime_error); // 指明只會拋出這個型別的例外
```

現代 C++（建議使用）：

```
void f() noexcept; // 表示此函式保證「不會」丟出例外
```

如果在 `noexcept` 函式中拋出例外，程式會直接終止！

小結

主題	說明
函式內使用 <code>throw</code>	拋出錯誤並交給上層處理
好處	控制錯誤處理分離、提升可讀性
常見錯誤類別	<code>std::runtime_error</code> , <code>std::invalid_argument</code> , <code>std::out_of_range</code> 等
使用 <code>try/catch</code> 捕捉	在呼叫端捕捉錯誤來避免程式崩潰

什麼是 Stack Unwinding？

Stack Unwinding（堆疊解開）是指當程式拋出例外、離開當前函式時，C++ runtime 會自動「逐層清除呼叫堆疊」的過程，也就是：

➡ 依序呼叫每一層中區域變數的 `destructor`（解構子），以確保資源正確釋放。



Stack Unwinding 流程圖（簡化版）：

```

main() ↓
├── funcA() ↓
│   ├── funcB() ↓
│   │   └── throw exception 🚨
│   │       ↑ ↑ ↑
│   │       ~funcB's local objects destroyed
│   │       ~funcA's local objects destroyed
└── ~main() catches exception

```



範例：觀察 Stack Unwinding

```

#include <iostream>
#include <stdexcept>

class Test {
    std::string name;
public:
    Test(const std::string& n) : name(n) {
        std::cout << "Constructor: " << name << '\n';
    }
    ~Test() {
        std::cout << "Destructor: " << name << '\n';
    }
};

void funcB() {
    Test t("B");
    throw std::runtime_error("Something went wrong in B!");
}

void funcA() {
    Test t("A");
    funcB();
}

```



```
int main() {
    try {
        funcA();
    } catch (const std::exception& e) {
        std::cerr << "Caught exception: " << e.what() << '\n';
    }
}
```




輸出：

```
Constructor: A
Constructor: B
Destructor: B
Destructor: A
Caught exception: Something went wrong in B!
```

你會看到：

- 即使程式中斷於 `throw`，仍然自動呼叫了解構子
- 這就是 **stack unwinding** 的作用

為什麼 Stack Unwinding 很重要？

1.  自動釋放資源（RAII 原則）
2.  避免資源洩漏（如：檔案沒關、記憶體沒釋放）
3.  讓 **destructors** 有機會清理狀態

補充：什麼情況下不會進行 stack unwinding？

- 若你在 `noexcept` 函式中丟出例外，程式會立即 **terminate**，不會進行 stack unwinding！

```
void f() noexcept {
    throw std::runtime_error("Oops!"); //  terminate!
```

}

✓ 小結

名稱	說明
Stack Unwinding	拋出例外後，自動解構堆疊上所有區域物件的過程
用途	資源釋放、自動清理
關鍵技術	搭配 RAII，讓物件自己在生命週期結束時做清理
注意	<code>noexcept</code> 函式中拋例外會直接 terminate，不會 stack unwinding

Class Level Exception Handling（類別層級的例外處理）

我們來依據三個面向進行詳細說明：

1. 📦 Method（成員函式）
2. 🛠 Constructor（建構子）
3. 💣 Destructor（解構子）

📦 Method：從成員函式拋出例外（Works same as regular functions）

在 C++ 中，類別的成員函式（method）與一般函式一樣，可以使用 `throw` 拋出例外。

◆ 範例

```
class BankAccount {  
public:
```

```
void withdraw(double amount) {
    if (amount < 0)
        throw std::invalid_argument("Negative withdrawal not allowed");
    // ...
}
};
```

◆ 使用方式

```
BankAccount account;
try {
    account.withdraw(-100);
} catch (const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

✓ 優點：讓錯誤的責任由呼叫端處理，類別內只需報告錯誤。

Constructor：建構子可能會拋出例外 (Constructor may fail)

建構子本身沒有回傳值，因此無法透過 `return false` 等機制表示失敗。若無法正確初始化物件，應使用 `throw` 拋出例外。

◆ 範例

```
class FileHandler {
    std::ifstream file;
public:
    FileHandler(const std::string& filename) {
        file.open(filename);
        if (!file)
            throw std::runtime_error("Failed to open file: " + filename);
    }
};
```

◆ 使用方式

```
try {  
    FileHandler f("not_exist.txt");  
} catch (const std::exception& e) {  
    std::cerr << e.what() << '\n';  
}
```

✅ 重點：建構失敗的物件不會建立成功，也不會留任何殘留記憶體。

💣 Destructor：！千萬不要在解構子中拋出例外 (Do NOT throw in destructor)

C++ 明確規定：不應從解構子中拋出例外。如果在解構物件時發生例外，而且有另一個例外同時發生（例如在 stack unwinding 中），程式會 **直接 terminate**！

💣 錯誤示範：

```
class Dangerous {  
public:  
    ~Dangerous() {  
        throw std::runtime_error("Don't do this in destructor!");  
    }  
};
```

這樣做會讓你的程式完全無法控制錯誤流程。

✅ 正確方式：捕捉例外但不再拋出

```
~Dangerous() {  
    try {  
        // 可能會出錯的清理邏輯  
    } catch (...) {  
        // 記錄錯誤，不要再 throw  
        std::cerr << "Exception suppressed in destructor\n";  
    }  
}
```

```
}
}
```

✅ 小結：Class Level Exception Handling 筆記

範疇	可拋出例外？	建議處理方式
Method	✅ 可以	用來報告邏輯錯誤
Constructor	✅ 可以	用來處理初始化失敗
Destructor	❌ 不可以	應該用 try/catch 吞下例外並記錄

std::exception Class Hierarchy（標準例外類別階層）

在 C++ 標準函式庫中，所有標準例外類別都繼承自：

```
class std::exception （定義於 <exception>）
```

這個基底類別提供了所有例外都能共用的基本介面，最重要的是虛擬函式：

```
virtual const char* what() const noexcept;
```

為什麼使用 std::exception 作為基底類別？

1. ✅ 統一接口（Polymorphic interface）
 - 讓所有例外都可以被統一使用，例如：

```
catch (const std::exception& e) {
    std::cerr << e.what();
}
```

2.  多型捕捉（Catch by base class）  多型捕捉的用途
 - 使用 std::exception 可以捕捉任何繼承它的子類別。

3. 可擴充性 (Extendable)

- 你可以自訂類別繼承它，加入自定錯誤訊息、錯誤代碼等。

4. 與標準函式庫相容

例外類別階層圖 (簡化版)

```

std::exception
├── std::logic_error      // 編譯時可檢查錯誤
│   ├── std::invalid_argument
│   ├── std::domain_error
│   ├── std::length_error
│   └── std::out_of_range
├── std::runtime_error    // 執行期錯誤
│   ├── std::range_error
│   ├── std::overflow_error
│   └── std::underflow_error

```

範例：使用標準例外類別

```

#include <iostream>
#include <stdexcept>

void process(int index) {
    if (index < 0 || index >= 10)
        throw std::out_of_range("Index out of range");
}

int main() {
    try {
        process(100);
    } catch (const std::exception& e) {
        std::cerr << "Caught: " << e.what() << '\n';
    }
}

```

✅ 使用 `std::exception` 可以保證無論你丟的是哪一種標準例外，都可以透過 `what()` 取得錯誤訊息。

👤💻 延伸：自定例外也建議繼承 `std::exception`

```
#include <exception>
#include <string>

class MyException : public std::exception {
    std::string msg;
public:
    MyException(const std::string& m) : msg(m) {}
    const char* what() const noexcept override {
        return msg.c_str();
    }
};
```

✅ 小結：為什麼使用 `std::exception` ？

優點	說明
🔄 多型支援	所有標準例外都可用 <code>std::exception</code> 統一捕捉
📢 提供 <code>what()</code>	能夠回傳錯誤訊息，便於 debug
📦 架構清晰	按照邏輯錯誤 / 執行期錯誤分類
🔧 易於擴充	可自定例外並與標準類別整合





🚫 C++ `noexcept` 筆記整理

✅ `noexcept` 是什麼？

`noexcept` 是 C++11 引入的關鍵字，用來標示「這個函式不會拋出例外」。

```
void safe_func() noexcept;    // 保證不會 throw
void risky_func();           // 可能會 throw
void explicitly_throwing() noexcept(false); // 明確聲明可能 throw (不常用)
```

noexcept 的用途

目的	說明
 禁止例外	若真的 throw，會立刻呼叫 <code>std::terminate()</code>
 多型一致	子類覆寫基底 class 的 method 時，必須同樣標示 <code>noexcept</code> （例如 <code>what()</code> ）
 例外安全性	搭配 STL 或錯誤處理使用時，避免進一步崩潰
 STL 最佳化	STL containers 會針對 <code>noexcept move/swap</code> 做優化

為什麼像 `what()` 要標示 `noexcept` ？

原因：

因為 `what()` 是在「例外已被捕捉後」呼叫的，用來回傳錯誤訊息字串，若它再拋出例外，會導致：

double exception → 程式終止！

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override { // 必須 noexcept
        return "My custom exception";
    }
};
```




Double Exception 的錯誤範例

```

class BadException : public std::exception {
public:
    const char* what() const override { // ❌ 沒有 noexcept !
        throw std::runtime_error("oops in what()!");
    }
};

int main() {
    try {
        throw BadException();
    } catch (const std::exception& e) {
        std::cerr << e.what(); // ⚠️ 呼叫後又 throw → terminate !
    }
}

```



結果：

```
terminate called after throwing an instance of 'std::runtime_error'
```

這就是「例外處理中再次丟出例外」的 **double exception**，會導致程式立刻中止！



哪些地方推薦使用 noexcept

用法	建議
析構子 (destructor)	✅ 強烈建議 <code>~T() noexcept</code> ，避免終結過程 throw
<code>std::exception::what()</code> override	✅ 必須 <code>noexcept</code> 才能正確覆寫
<code>swap()</code> / 移動建構子	✅ <code>noexcept</code> 可啟用 STL swap/move 最佳化
錯誤報告用成員函式	✅ 避免二次錯誤 (如 <code>.message()</code> 、 <code>.error_code()</code>)

小結

- `noexcept` 是 例外安全性 的保證
- 一旦標註，就要保證不會拋出例外
- 對於例外類別中的 `what()`、`destructor` 等函式，加上 `noexcept` 是必須的安全措施

多型捕捉的用途

當你定義了多個不同的 exception 類別，但都繼承自 `std::exception`，你就可以只寫一個 catch block：

```
catch (const std::exception& e) {  
    std::cerr << e.what();  
}
```

這樣就能捕捉到任何子類別。

範例程式：多型例外捕捉

```
#include <iostream>  
#include <exception>  
  
// 自訂例外類別  
class FileNotFound : public std::exception {  
public:  
    const char* what() const noexcept override {  
        return "File not found";  
    }  
};  
  
class InvalidInput : public std::exception {  
public:
```

```

const char* what() const noexcept override {
    return "Invalid input";
}

};

void process(int code) {
    if (code == 1)
        throw FileNotFound();
    else if (code == 2)
        throw InvalidInput();
    else
        std::cout << "Processing success.\n";
}

int main() {
    try {
        process(2); // 試著丟出 InvalidInput
    } catch (const std::exception& e) { // ← 多型捕捉
        std::cerr << "Caught: " << e.what() << '\n';
    }
}

```



執行結果：

Caught: Invalid input



小結

行為	說明
throw 子類別	丟出 FileNotFound 或 InvalidInput
catch 父類別	使用 catch (const std::exception&) 來捕捉
多型實現	根據物件真實型別，呼叫對應的 what()

C++ 裡面這麼多 `std::exception` 的子類別，它們只是為了可讀性？還是真的對程式行為有影響？

答案是：兩者皆是，但主要目的是為了「語意分類」與「選擇性捕捉」，也就是：

子類別的主要目的

1. 語意分類（Semantic classification）

讓你明確知道是哪一類型的錯誤發生，例如：

例外類別	用途	說明
<code>std::out_of_range</code>	容器越界	用於 <code>at()</code> 函式
<code>std::invalid_argument</code>	無效輸入	例如建構時參數錯誤
<code>std::runtime_error</code>	執行時錯誤	檔案打不開等

➡ 可讀性提升，debug 時更容易判斷問題本質

2. 選擇性捕捉（Selective catching）

你可以針對不同錯誤型別做不同處理：

```
try {  
    throw std::invalid_argument("bad input");  
} catch (const std::out_of_range& e) {  
    std::cerr << "Range error: " << e.what();  
} catch (const std::invalid_argument& e) {  
    std::cerr << "Input error: " << e.what(); // ← 這個會被呼叫  
} catch (const std::exception& e) {  
    std::cerr << "General exception: " << e.what();  
}
```

➡ 控制流根據例外類型不同而不同，這會直接影響程式行為！

✓ 3. 提供更細緻的錯誤意圖與 API 訊號

許多 STL 或標準函式會明確指定它們會拋出哪一種錯誤，這對使用者來說：

- 有明確 API 文件行為
- 可以依據標準使用正確的 try-catch
- 像是：

```
std::vector<int> v;
v.at(10); // throws std::out_of_range
```

你就知道這不是 `std::runtime_error`，也不是 memory error，而是 index 超出界限。

🔍 額外行為上的差異：幾乎沒有

從功能面來說：

- `std::exception` 及其子類別只提供 `virtual const char* what() const noexcept`
- 這些子類別並沒有其他額外邏輯或資料成員（是 lightweight class）
- 所以除了 `what()` 提供的訊息不同外，程式執行邏輯並不會因此自動變化

！真正影響程式行為的，是你在 catch 時根據子類別作不同處理這件事

✓ 小結：子類別存在的價值

目標	是否影響行為？	說明
可讀性與除錯	✓ 閱讀時更清楚是什麼錯誤	
選擇性捕捉	✓ 你可以 catch 不同類別做不同事情	
提供 <code>what()</code> 不同內容	✓ 報錯時更清楚	
程式自動處理差異	✗ 除非你自己針對不同類別處理，否則行為一樣	

Demo

 範例：隨機丟出不同 `std::exception` 子類別，並針對性處理

 `main.cpp`

```
#include <iostream>
#include <stdexcept>
#include <cstdlib>
#include <ctime>

void throw_random_exception() {
    int n = std::rand() % 3;

    if (n == 0) {
        throw std::invalid_argument("Invalid argument: expected positive integer.");
    } else if (n == 1) {
        throw std::out_of_range("Out of range: index exceeds limit.");
    } else {
        throw std::runtime_error("Runtime error: unknown processing failure.");
    }
}

int main() {
    std::srand(static_cast<unsigned>(std::time(nullptr)));

    try {
        throw_random_exception();
    } catch (const std::invalid_argument& e) {
        std::cerr << "[Invalid Input Handler] " << e.what() << '\n';
    } catch (const std::out_of_range& e) {
        std::cerr << "[Index Error Handler] " << e.what() << '\n';
    } catch (const std::exception& e) {
        std::cerr << "[Generic Error Handler] " << e.what() << '\n';
    }
}
```

```
return 0;
}
```

執行結果（每次可能不同）

[Index Error Handler] Out of range: index exceeds limit.

或

[Invalid Input Handler] Invalid argument: expected positive integer.

或

[Generic Error Handler] Runtime error: unknown processing failure.

重點回顧

功能	說明
throw std::invalid_argument(...)	模擬輸入錯誤
throw std::out_of_range(...)	模擬容器索引錯誤
throw std::runtime_error(...)	模擬一般執行錯誤
catch 多型階層	根據子類別作不同處理，展現子類的意義與用處

如何編譯（使用 g++）

```
g++ -std=c++11 -o exception_demo main.cpp
./exception_demo
```