

Section 14-1 Operator Overloading

C++ Operator Overloading（運算子重載）介紹

什麼是 Operator Overloading？

Operator Overloading 是 C++ 的一項語言特性，允許你為自訂類別（class）定義或改寫內建運算子（如 +、-、*、=、== 等）的行為，使這些運算子可以作用在自訂類別的物件上，達到像內建型態一樣方便使用的效果。

簡單說：

- 讓你可以用 + 來加兩個自訂類別的物件
- 用 == 比較兩個物件
- 用 [] 操作物件像陣列一樣

為什麼要用 Operator Overloading？

1. 提升程式的可讀性與直觀性
使用自訂類別時，也能用熟悉的運算子操作，讓程式碼更自然。
2. 達成物件導向的設計理念
讓物件自己定義如何和其他物件互動。
3. 方便數學或複雜資料型態操作
例如向量、矩陣、複數等數學物件的加減乘除。

怎麼實作 Operator Overloading？

1. 成員函式（Member Function）重載

```
class Vector {  
public:  
    int x, y;  
  
    Vector(int x, int y) : x(x), y(y) {}  
};
```

```
// 重載 + 運算子，讓兩個 Vector 可以相加
Vector operator+(const Vector& v) {
    return Vector(x + v.x, y + v.y);
}
};
```

使用時：

```
Vector v1(1, 2);
Vector v2(3, 4);
Vector v3 = v1 + v2; // v3 是 (4, 6)
```

2. 非成員函式（Friend 或 普通函式）重載

有時候為了對稱性或特定需求，會用非成員函式：

```
class Vector {
public:
    int x, y;

    Vector(int x, int y) : x(x), y(y) {}

    friend Vector operator+(const Vector& v1, const Vector& v2);
};

Vector operator+(const Vector& v1, const Vector& v2) {
    return Vector(v1.x + v2.x, v1.y + v2.y);
}
```

哪些運算子可以重載？

- 幾乎所有運算子都可以重載（+ - * / % == != < > <= >= [] () -> 等）
- 注意：不能重載以下幾個：
 - :: （作用域運算子）
 - . （成員選取運算子）

- `*` (成員指標運算子)
- `?:` (三元條件運算子)

簡單範例：重載 `<<` 方便輸出

```
#include <iostream>
using namespace std;

class Vector {
public:
    int x, y;
    Vector(int x, int y) : x(x), y(y) {}

    friend ostream& operator<<(ostream& os, const Vector& v);
};

ostream& operator<<(ostream& os, const Vector& v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}

int main() {
    Vector v(10, 20);
    cout << v << endl; // 輸出: (10, 20)
}
```

小結 Summary

中文說明	English Description
運算子重載讓自訂類別也能使用運算子	Operator overloading lets custom classes use operators like built-in types.
提升程式直觀與可讀性	Improves code readability and intuitiveness.


中文說明	English Description
可以透過成員函式或非成員函式實作	Implemented via member or non-member functions.
不能重載少數幾個運算子	Certain operators like <code>::</code> and <code>.</code> cannot be overloaded.

常見運算子重載實例整理

+ 運算子：兩個物件相加

```
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}

    Point operator+(const Point& p) const {
        return Point(x + p.x, y + p.y);
    }
};
```


 使用：

```
Point p1(1, 2), p2(3, 4);
Point p3 = p1 + p2; // (4, 6)
```

== 運算子：比較兩個物件是否相等

```
class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}
```


```
bool operator==(const Point& p) const {
    return x == p.x && y == p.y;
}
};
```

 使用：

```
Point p1(1, 2), p2(1, 2);
if (p1 == p2) {
    cout << "Equal!";
}
```

[] 運算子：像陣列一樣取值

```
class MyArray {
    int data[10];
public:
    int& operator[int index](int%20index) {
        return data[index];
    }
};
```

 使用：

```
MyArray arr;
arr[0] = 42;
cout << arr[0]; // 輸出 42
```


() 運算子：使物件像函式一樣可呼叫

```
class Multiply {
    int factor;
public:
```

```

Multiply(int f) : factor(f) {}
int operator()(int x) const {
    return x * factor;
}
};

```

 使用：

```

Multiply times3(3);
cout << times3(5); // 15

```

++ 運算子：自訂前置或後置遞增

前置：

```

class Counter {
    int count;
public:
    Counter() : count(0) {}
    Counter& operator++() { // 前置
        ++count;
        return *this;
    }
};

```

後置：

```

class Counter {
    int count;
public:
    Counter() : count(0) {}
    Counter operator++(int) { // 後置
        Counter temp = *this;
        count++;
        return temp;
    }
};

```

```
}
};
```


✓ << 運算子：友善輸出（stream）

```
#include <iostream>
using namespace std;

class Point {
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}

    friend ostream& operator<<(ostream& os, const Point& p);
};

ostream& operator<<(ostream& os, const Point& p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}
```

 使用：

```
Point p(5, 10);
cout << p << endl; // (5, 10)
```

✓ = 運算子：指定自訂的複製邏輯（深拷貝）


```
class DeepCopy {
    int* data;
public:
    DeepCopy(int value) {
        data = new int(value);
    }
}
```

```

DeepCopy& operator=(const DeepCopy& other) {
    if (this != &other) {
        delete data;
        data = new int(*other.data);
    }
    return *this;
}

~DeepCopy() {
    delete data;
}
};

```

 使用：

```

DeepCopy a(10), b(20);
b = a;


```

-> 運算子：用於 smart pointer 類別

```

class SmartPointer {
    int* ptr;
public:
    SmartPointer(int* p) : ptr(p) {}
    int* operator->() {
        return ptr;
    }
};

```

 使用：

```

SmartPointer sp(new int(5));
cout << *(sp.operator->()); // 5

```


✓ < 運算子：用於排序或比較大小

```
class Person {
public:
    int age;
    Person(int age) : age(age) {}

    bool operator<(const Person& other) const {
        return age < other.age;
    }
};
```

✏ 使用（如在 `std::sort` 中）：

```
vector<Person> people = {Person(30), Person(20)};
sort(people.begin(), people.end()); // 按年齡升序
```

📝 總結

運算子	常見用途	可否重載
<code>+ - * /</code>	數學運算	✓
<code>== != < ></code>	比較運算	✓
<code>[]</code>	陣列存取	✓
<code>()</code>	仿函式物件 (function object)	✓
<code><< >></code>	輸出入串流操作	✓
<code>=</code>	自訂複製邏輯（深拷貝）	✓
<code>++ --</code>	遞增遞減	✓
<code>-></code>	智慧指標模擬	✓
<code>:: .* ?:</code>	作用域 / 成員 / 條件運算	✗ 不可重載



friend 的角色

- friend 不是函式定義，而是授權：允許某個函式或類別存取 private/protected 成員。
- 出現在類別內部，只是告訴編譯器「這個外部函式/類別有存取權」。
- 函式的實作仍然需要在類別外部定義。

```
class MyClass {
private:
    int secret = 42;
    friend void reveal(const MyClass& obj); // 授權
};

void reveal(const MyClass& obj) { // 定義
    std::cout << obj.secret << std::endl;
}
```



Operator Overload 的兩種方式

1 Member function overload

- 左操作元 (lhs) 預設是呼叫物件本身 (this)。
- 編譯器會將 `a + b` 翻譯成 `a.operator+(b)`。

```
class Point {
private:
    int x;

public:
    Point(int x) : x(x) {}

    // member function
    Point operator+(const Point& other) const {
        return Point(this->x + other.x);
    }
};
```

```
Point a(10), b(20);
Point c = a + b; // => a.operator+(b)
```

👉 特性：

- 左邊必須是該類別物件。
- 封裝性較好，不需使用 `friend`。

2 Friend function overload

- 左右操作元都需要明確傳入參數。
- 編譯器會將 `a + b` 翻譯成 `operator+(a, b)`。

```
class Point {
private:
    int x;

public:
    Point(int x) : x(x) {}
    friend Point operator+(const Point& lhs, const Point& rhs);
};

Point operator+(const Point& lhs, const Point& rhs) {
    return Point(lhs.x + rhs.x);
}

Point a(10), b(20);
Point c = a + b; // => operator+(a, b)
```

👉 特性：

- 左操作元不一定要是該類別，可以處理 `int + Point`、`ostream << obj`。
- 需要授權才能存取 `private` 成員。

差異比較表

特性	Member function	Friend function
語法轉換	<code>a + b</code> \rightarrow <code>a.operator+(b)</code>	<code>a + b</code> \rightarrow <code>operator+(a, b)</code>
左操作元	呼叫物件 (<code>this</code>)	函式參數明確傳入
可否處理 <code>int + Point</code>	✗ 不行	✓ 可以
封裝性	✓ 不需暴露 <code>private</code>	✓ 需 <code>friend</code> 授權
彈性	⚠ 左邊必須是類別	✓ 更靈活

設計準則

- **Member function**：當左操作元必然是該類別時（如 `p1 + p2`）。
- **Friend function**：當需要存取 `private`，或左操作元不是該類別（如 `int + Point`，`ostream << obj`）。