C++

Virtual Function

```
class base{
public:
   virtual void print(){...}
   void show(){...}
};
class derived: public base{
public:
   void print(){...}
   void show(){...}
};
int main(){
   base* bptr;
   derived d;
   bptr = &d;    //向上轉型是沒問題的，所以base可以指向d
   bptr→print(); //會執行derived的print()，因為實際指向的物件型別是derived
   bptr→show();  //會執行base的print()，因為指標設定指向的型別是base
```

1. 由於print()在base(父類別)裡是virtual，因此要使用derived還是base的print()是**執行**的時候才會決定；由於show()在base(父類別)裡不是virtual，因此要用誰的show()在**編譯**的時候就決定了。
2. 要用**指標**或**參考**的方式呼叫，才有改變的餘地喔
3. 用處是base和derived同時有的函式，會優先使用derived的，其他情況下會用base自己的

Virtual Succeed

```
class D {
public:
   int x = 100;
};
class B : public D {};
class C : public D {};
```

```
class A : public B, public C {};

int main() {
    A a;
    a.B::x = 1;
    a.C::x = 2;
    cout << a.B::x << ", " << a.C::x << endl;  // ➜ 1, 2 ❗
}
```

```
class D {
public:
    int x = 100;
};

class B : virtual public D {};
class C : virtual public D {};
class A : public B, public C {};

int main() {
    A a;
    a.x = 42;
    cout << a.B::x << ", " << a.C::x << ", " << a.x << endl;  // ➜ 42, 42, 42 ✅
}
```

💡 此時就會共用同一份D的資料，因此不必再用 `::` 去指定要呼叫誰的 `D`

Dynamic cast

💡

1. 常用在父類別轉成子類別的時候，因為此類過程較不安全
2. dynamic cast仰賴父類別一定要有至少一個virtual，這樣編譯器才知道這個類別要使用**執行時型別資訊（RTTI）**
3. 子類別轉父類別是安全的，且為什麼子父類別間互轉不會因為含有的資料型別不同而發生問題？因為在繼承的時候，子類別就會騰出父類別所需要的記憶體

<time.h>

函式名	功能	回傳型別
`time()`	取得目前時間（從 1970 起秒數）	`time_t`
`localtime()`	把 `time_t` 轉成「當地時間的結構」	`struct tm*`
`gmtime()`	把 `time_t` 轉成「UTC 時間結構」	`struct tm*`
`strftime()`	將時間格式化成字串	`size_t`
`difftime()`	比較兩個時間的差（秒）	`double`
`clock()`	測量 CPU 執行時間	`clock_t`
`asctime()`	把 `struct tm*` 轉成字串	`char*`
`ctime()`	把 `time_t` 轉成字串（= `asctime(localtime(...))`）	`char*`
`mktime()`	把 `struct tm` 轉成 `time_t`（反向）	`time_t`

```
//取得目前時間
time_t t = time(NULL);
struct tm* ptr = localtime(&t);
cout << asctime(ptr); //Wed May 07 17:57:19 2025
cout << ptr→tm_sec; //輸出現在秒數
```

```
//程式執行時間
time_t start, end;
start = time(NULL);
/*.....*/
end = time(NULL);
difftime(end, start);
```

```
struct tm {
    int tm_sec;   // 秒數（0-60，包含閏秒）
    int tm_min;   // 分鐘（0-59）
    int tm_hour;  // 小時（0-23）
```

```
    int tm_mday;  // 一個月中的第幾天（1-31）
    int tm_mon;   // 月份（0-11，0 = 一月）
    int tm_year;  // 自 1900 年起的年數（所以 2024 年要寫成 124）
    int tm_wday;  // 一週的第幾天（0-6，0 = 星期日）
    int tm_yday;  // 一年中的第幾天（0-365）
    int tm_isdst; // 夏令時間旗標（>0 = 有夏令時間，0 = 沒有，<0 = 不知道）
};
```

```
//測量程式執行時間
clock_t start = clock();
/*.....*/
clock_t end = clock();
cout << double(end - start) / CLOCKS_PER_SEC;
```

#ifndef / define / endif

```
//headers file
#ifndef MY_HEADER_H
#define MY_HEADER_H
void say_hello();
#endif
```

Constructor

```
//COPY constructor
class A {
public:
        int x, y;
        A(){ //Default constructor的模樣
            int x = 0, y = 0;
        }
    A(const A& other) { //使用左值參考
        cout << "Copy Constructor\n";
        x = other.x;
    }
};
//此為自訂copy constructor，預設copy constructor會複製class中的所有資料型別
```

```
//MOVE constructor
class A {
public:
        string name;
    int* data;
    A(A&& other) { //使用右值參考
            name = std::move(other.name);
        data = other.data;    //指標不支援move，要自己處理
        other.data = nullptr;
    }
};
```

💡 若class裡已經有寫constructor，則系統則不會幫你補上default constructor，所以若你沒自己補寫default constructor，則無法使用 `A a;` 這種宣告方法

Destructor

```
class Myclass{
public:
        int *ptr;
        Myclass(int val){ ptr = new int(val); }
        ~Myclass() {
        delete ptr;         // ❗ 若沒這行，data 永遠不會被釋放，因為預設的destructor只會
    }
}
//此為自訂destructor，會在 物件超出作用域 刪除物件指標時 自動觸發
//ptr本身的記憶體會在整個物件銷毀時被釋放
```

💡 小心有經過copy和move的兩物件中資料若有指標，可能會重複delete指標，因為經過copy和move，兩物件的指標會指向同個記憶體

Friend class / function

💡 Properties

1. 可以access該A class的 `private` 和 `protected`
2. 不屬於A class的member

```
class anotherClass{
public:
      int foo();
}
class A{
   friend class F;
   friend void foo();
   friend int anotherClass::voo();
private:
   int private_data;
};
class F{
   void show(A& a){ cout << a.private_data; }
};
void foo(){ ... }
int anotherClass::voo(){ ... }
int main(){
   foo();
   anotherClass B;
   B.voo();
}
```

this

💡 Properties

1. `this` 是呼叫成員函式的物件本身
2. `this` 只能在類別的**成員函式**裡使用

```
class Person{
public:
```

```
    string name;
    int val = 0;
    void setname(string& name){ this→name = name; } //這樣就不會搞混
    Counter& add(int x) { //可以做鏈式呼叫 ex:C.add(1).add(2).add(3);
      val += x;
    return *this;  //回傳「目前這個物件本身」
  }
```

Operator Overloading

💡
1. class自己可定義operator
2. `.` `*` 等等pointer to member operator、`?:` `sizeof()` `typeid()` 無法自訂義

```
class Vec2 {
public:
    int x, y;
    Vec2 operator+(const Vec2& other) const { //operator前面需要修飾詞
        return Vec2(x + other.x, y + other.y);
    }
    operator float() const { //operator前面不用修飾詞
        return sqrt(x * x + y * y);
    }
};
//const放在那邊意思是不能修改目前物件的成員變數
```

```
//自訂義內建class的operator
class Myclass{
    friend ostream& operator<<(ostream&, const Myclass&);
private:
    int a = 2;
};
ostream& operator<<(ostream& output, const Myclass& a){
    output << "djiewjdi: " << a.a << endl; //千萬不能再寫<<a，不然會陷入無窮遞迴
    return output; //使支援鏈式呼叫
}
```

▼ Supplement

`cin.ignore()` 可以吃掉緩衝區的第一個字元

`cin.ignore(num, char)` 持續吃掉輸入直到遇到特定char

寫 `cin` 要用 `istream`

```
class Date{
public:
    Date& operator++();
private:
    int year, month, day;
}
Date& Date::operator++(){
    day++;
    return *this; //就跟i++會回傳i現在的值一樣
}
```

File processing

#include <fstream>

```
std::ofstream fout("file.txt", std::ios::out);
if (!fout.is_open()){
    std::cerr << "Failed to open file\n";
    return 1;
}
const char* data = "Hello World!";
fout.write(data, 5); //寫入 "Hello"
fout.put('A');
fout.close();
```

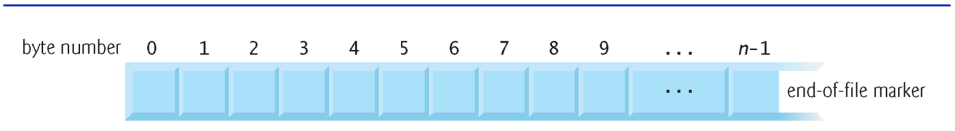

Fig. 14.1 | C++'s simple view of a file of *n* bytes.

```
std::ofstream fout("file.txt"); //預設是std::ios::out
fout << "Hello World";
fout.seekp(6);          // 移到 "W" 的位置
fout << "C++";          // 寫成 "Hello C++ld"
fout.close();
```

Mode	Description
`ios::app`	*Append* all output to the end of the file.
`ios::ate`	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written *anywhere* in the file.
`ios::in`	Open a file for *input*.
`ios::out`	Open a file for *output*.
`ios::trunc`	*Discard* the file's contents (this also is the default action for `ios::out`).
`ios::binary`	Open a file for binary, i.e., *nontext*, input or output.

Fig. 14.3 | File-open modes.

```
std::ifstream fin("file.txt", std::ios::in);
if (!fin.is_open()){
   std::cerr << "Failed to open file\n";
   return 1;
}
std::string word;
fin >> word; //讀到空白或換行為止
char c;
fin.get(c);
getline(fin, word);
fin.close();
```

1. `cin` 會留下 `\n` ，若在後面接 `getline` 會出事
2. `getline` 第三個參數可決定要讀到哪個字元，預設是 `\n`

```
std::fstream f("file.dat", std::ios::in | std::ios::out | std::ios::binary); //同時讀寫二進位資
Student s; //存取資料庫檔案中的struct
f.read(reinterpret_cast<char*>(&s), sizeof(word));
```

```
int x = 42;
f.write(reinterpret_cast<char*>(&x), sizeof(x)); //write實際上會以byte為單位寫入資料
//reinterpret_cast會把指標改變呈現方式
f.seekp(5); //移動put指標到第5個byte
f.seekg(10, std::ios::cur); //移動get指標到從目前位置往後10bytes的位置
f.seekp(0, std::ios::end); //移動put指標到結尾
f.seekg(0, std::ios::beg); //回到檔案開頭
std::cout << f.tellp() << f.tellg(); //輸出put指標和get指標的位置
if (f.eof) std::cout << "end!"; //讀取失敗才會回傳true
f.close();
```

- `read()` 就是把資料以1byte為單位讀進來一個變數，然後要怎麼解釋這個變數任由你發揮
- 讀指標和寫指標可以同時在不同的地方
- `read()` 和 `write()` 只能寫 `char*` 型別的資料，因為本身只支援二進位輸入

stringstream

```
//字串↔數字
#include <iostream>
#include <sstream>
using namespace std;

int main() {
    stringstream ss;
    int num = 123;

    // 數字轉字串(可以把資料存進字串)
    ss << num;
    string str = ss.str();
    cout << "字串內容：" << str << endl;

    // 清空再來一次
    ss.str(""); // 清內容
    ss.clear(); // 重設狀態
```

```
    // 字串轉數字
    ss << "456";
    int x;
    ss >> x;
    cout << "轉回數字：" << x << endl;
}
```

```
//切割字串
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string data = "apple banana cherry";
    stringstream ss(data);

    string word;
    while (ss >> word) {
        cout << "拿到單字：" << word << endl;
    }
}
```

📌 很像用 `cin >> word`，但輸入對象是字串！

```
//連續讀入多種型別
#include <iostream>
#include <sstream>
using namespace std;

int main() {
    string input = "James 21 182.5";
    stringstream ss(input);

    string name;
    int age;
    float height;

    ss >> name >> age >> height;
```

```
    cout << name << " 年齡：" << age << " 身高：" << height << endl;
}
```

用法	說明
`ss.str()`	🔍 取得目前的完整字串內容
`ss.str("...")`	✍️ 直接設定新的字串進來當資料流

```
//如果你想重複使用 stringstream，除了 .str("新字串")，還要加上 .clear()
ss.str("new content");
ss.clear();  // 清除 EOF 或錯誤狀態（⚠️ 不清可能無法再讀）//
```

💡 一樣可以用 `getline(ss, string)`

Lambda

```
[capture](parameter_list) -> return_type {
    // 函式內容
}
//return type可省略(給編譯器自行偵測)
//會回傳匿名函式物件
```

💡 **Capture**

- `[]` 不使用任何外部變數
- `[=]` 以copy的方式使用所有外部變數
- `[x]` 以copy的方式使用外部變數x
- `[&]` 以reference的方式使用所有外部變數
- `[&x]` 以reference的方式使用外部變數x
- `[=, &x]` 除了x用reference以外，其他所有外部變數都用copy的方式

```
//Example
auto doubleIt = [](int x) {
    return x * 2;  // 自動推斷是 int
```

```
};
std::cout << doubleIt(5);  // 印出 10
```

```
//Example
vector<int> v = {5, 2, 9, 1};
sort(v.begin(), v.end(), [](int a, int b) {
    return a > b;  // 降序
});
```

```
// for_each用法
array<int, 5> values{1, 2, 3, 4, 5};
int sum = 0;
for_each(values.cbegin(), values().cend(), [&sum](auto i){ if (i > 0) sum += i; });
//cbegin()和cend()是 const 開頭迭代器，不能'*'修改位置資料
```

Ostream Iterator & copy

#include <iterator>

```
//將一串輸入存進vector
vector<int> v;
copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));

//將讀取的資料存進vector
ifstream fin("data.txt");
vector<double> values;
copy(istream_iterator<double>(fin), istream_iterator<double>(), back_inserter(value

//輸出vector
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));

//將vector寫入檔案
ofstream fout("ppt.txt");
copy(values.begin(), values.end(), ostream_iterator<double>(fout, ","));
```

```
//copy()用法
set<int> s = {9, 4, 7};
vector<int> v(5);
copy(s.begin(), s.end(), v.begin()+1);
```

```
//只要有資料結構支援begin()、end(),就能用copy()
//注意資料要有足夠的空間存入
```

STL Functions

#include <algorithm>

函式名稱	功能分類	用途	參數格式
fill	填充	將區間全部填入指定值	fill(first, last, value)
fill_n	填充	從起始位置填入 n 個指定值	fill_n(first, n, value)
generate	產生	用函式產生元素填滿區間	generate(first, last, gen)
generate_n	產生	用函式產生 n 個元素	generate_n(first, n, gen)
equal	比對	判斷兩區間內容是否相等	equal(first1, last1, first2)
mismatch	比對	找出兩區間第一個不同元素	mismatch(first1, last1, first2)
lexicographical_compare	比對	字典順序比較兩區間	lexicographical_compare(first1, last1, first2, last2)
remove	移除	移除符合條件的元素(該元素以外的元素移往前，元素都還會在，只是被覆蓋)	remove(first, last, value)
remove_copy	移除	複製不包含指定值的元素	remove_copy(first, last, dest, value)
remove_if	移除	移除符合條件的元素（用 predicate）	remove_if(first, last, pred)
remove_copy_if	移除	複製不符合條件的元素	remove_copy_if(first, last, dest, pred)
replace	取代	將區間中指定值改為新值	replace(first, last, old_value, new_value)
replace_copy	取代	將指定值替換後複製到新區間	replace_copy(first, last, dest, old_value, new_value)
replace_if	取代	將符合條件的元素改為新值	replace_if(first, last, pred, new_value)

replace_copy_if	取代	符合條件元素替換後複製	replace_copy_if(first, last, dest, pred, new_value)
count	統計	統計某值在區間中出現次數	count(first, last, value)
count_if	統計	統計符合條件的元素數量	count_if(first, last, [pred])
find	查找	尋找指定值的位置	find(first, last, value)
find_if	查找	尋找第一個符合條件的元素	find_if(first, last, [pred])
find_if_not	查找	尋找第一個不符合條件的元素	find_if_not(first, last, [pred])
sort	排序	將區間排序	sort(first, last)
binary_search	搜尋	判斷已排序區間中是否存在某值	binary_search(first, last, value)
lower_bound	搜尋	回傳第一個不小於目標值的位置	lower_bound(first, last, value)
upper_bound	搜尋	回傳第一個大於目標值的位置	upper_bound(first, last, value)
equal_range	搜尋	回傳等於目標值的範圍(回傳[lower_bound, upper_bound])	equal_range(first, last, value)
all_of	條件判斷	是否所有元素都符合條件	all_of(first, last, [pred])
any_of	條件判斷	是否任一元素符合條件	any_of(first, last, [pred])
none_of	條件判斷	是否所有元素皆不符合條件	none_of(first, last, [pred])
accumulate	累加	從初始值開始累加區間值	accumulate(first, last, init)
transform	轉換	將每個元素套用函式並存入新區間	transform(first, last, dest, [unary_op])
min_element	最值	回傳區間中最小值位置	min_element(first, last)
max_element	最值	回傳區間中最大值位置	max_element(first, last)
minmax_element	最值	同時回傳 min/max 位置	minmax_element(first, last)

swap	交換	交換兩個變數的值(vector會交換指標)	swap(a, b)
iter_swap	交換	交換兩個 iterator 指向的內容(用於容器內部交換時)	iter_swap(it1, it2)(一定得是指標)
swap_ranges	交換	交換兩個區間對應的內容	swap_ranges(first1, last1, first2)
copy_backward	複製	從尾到頭複製元素	copy_backward(first, last, dest_end)
reverse	反轉	反轉區間順序	reverse(first, last)
reverse_copy	反轉	反轉後複製到新區間	reverse_copy(first, last, dest)
unique	去重	移除相鄰重複元素	unique(first, last)
unique_copy	去重	移除重複元素並複製	unique_copy(first, last, dest)
merge	合併	合併兩個排序好的區間	merge(first1, last1, first2, last2, dest)
inplace_merge	合併	就地合併兩段已排序區間	inplace_merge(first, middle, last)
includes	集合運算	A 是否包含 B 所有元素	includes(first1, last1, first2, last2)
set_union	集合運算	求聯集	set_union(first1, last1, first2, last2, dest)
set_intersection	集合運算	求交集	set_intersection(first1, last1, first2, last2, dest)
set_difference	集合運算	A 減 B 的差集	set_difference(first1, last1, first2, last2, dest)
set_symmetric_difference	集合運算	對稱差集（非交集部分）	set_symmetric_difference(first1, last1, first2, last2, dest)
minmax	最值	找出兩值中最小與最大者	minmax(a, b)

▼ 注意事項

`accumulate` 要 `#include <numeric>`

`set_` 系列的函式都要求**排序好**的序列

💡 有些需要判斷的函式都可以自訂 `comp()`

Character-handling Functions

函式名稱	用途	參數格式
isdigit	Return 1 if c is a digit and 0 otherwise	isdigit(c)
isalpha	Return 1 if c is a letter and 0 otherwise	isalpha(c)
isalnum	Return 1 if c is a digit or a letter and 0 otherwise	isalnum(c)
isxdigit	Return 1 if c is a hexadecimal digit character and 0 otherwise	isxdigit(c)
islower	Return 1 if c is a lowercase letter and 0 otherwise	islower(c)
isupper	Return 1 if c is a uppercase letter and 0 otherwise	isupper(c)
tolower	Change a letter into lowercase	tolower(c)
toupper	Change a letter into uppercase	toupper(c)
isspace	Return 1 if c is a whitespace character (‘\n’, ‘ ‘, ‘f’, ‘\r’, ‘\t’, ‘\v’) and 0 otherwise	isspace(c)

Memory operation Functions

函式名稱	用途	語法	補充說明
memcpy	複製記憶體資料	memcpy(dest, src, size)	複製 src 的資料到 dest，⚠️ 不可重疊 不能複製內有指向外部資源的資料 ex: array可以,vector不行
memmove	安全複製記憶體（可重疊）	memmove(dest, src, size)	跟memcpy()功能一樣 ✅ 可處理 src/dest 區塊重疊的情況
memcmp	比較兩塊記憶體	memcmp(ptr1, ptr2, num)	逐 byte 比較，回傳 0 表示相等，<0/>0 表示大小關係(字典大小)
memchr	尋找記憶體中第一個指定 byte	memchr(ptr, ch, size)	找 ch 第一次出現的位置(void*)，找不到則回傳 nullptr ex: 轉換成char*用 static_cast<char*> 可以尋找位元資料
memset	將記憶體填滿某個 byte 值	memset(ptr, value, size)	常用來初始化成 0（注意是填 byte，不是整數） 只適合原始資料，不適合STL容器或class

```
struct Student {
    int id;
    char name[20];
};
Student s = {42, "James"};
char buffer[sizeof(Student)];
memcpy(buffer, &s, sizeof(Student)); //將s的記憶體資料複製進buffer
Student* ps = reinterpret_cast<Student*>(buffer); //將buffer改變詮釋方式
cout << ps→id << " " << ps→name << endl;
```

C-string Functions

函式原型	回傳型別	用途說明
`double stof(const char* or string)`	`double`	將字串轉為 `double`。若無法轉換則回傳 0。
`int stoi(const char* or string)`	`int`	將字串轉為 `int`。若無法轉換則回傳 0。
`long stol(const char* or string)`	`long int`	將字串轉為 `long`。若無法轉換則回傳 0。
`double strtod(const char* nPtr, char** endPtr)`	`double`	將字串轉為 `double`，`endPtr` 會指向第一個不是數字的地方。轉換失敗回傳 0。 ex: char* end, a = “2.34abc”; double f = strtod(a, &end);
`long strtol(const char* nPtr, char** endPtr, int base)`	`long`	將字串轉為 `long`，可指定進位（2–36），`endPtr` 會指向未處理部分。
`unsigned long strtoul(const char* nPtr, char** endPtr, int base)`	`unsigned long`	將字串轉為 `unsigned long`，同樣可指定進位，轉換失敗回傳 0。
`char* strpbrk(const char* s1, const char* s2)`	`char*`	回傳s2中任意字元出現在s1中的第一個位置

💡 `to_string(auto)` 可將int, double, float,long long轉成字串

deque

#include <deque>

常用函式	功能
push_back()	在尾端新增元素
push_front()	在前端新增元素
pop_back()	移除尾端元素
pop_front()	移除前端元素

front()	取得前端元素
back()	取得尾端元素
at[i] / [i]	存取第 i 個元素
size()	取得大小
empty()	檢查是否為空

queue

#include <queue>

常用函式	功能
push()	加入元素（到尾端）
pop()	移除元素（從前端）
front()	查看最前端元素
back()	查看最後一個元素
size()	回傳元素個數
empty()	判斷是否為空佇列