(Ghi chú)

Ngôn ngữ lập trình cpp

**Thực hiện: Nguyễn Văn Hào**

**Email:** [hao2205tb@gmail.com](mailto:hao2205tb@gmail.com)

(Tài liệu được biên xoạn lại từ nhiều nguồn và không dùng trong mục đích kinh doanh kiếm tiền)

# Mục lục

[Mục lục 1](#_Toc126889360)

[Set 12](#_Toc126889361)

[Tổng quan 12](#_Toc126889362)

[Set trong c++ là gì 12](#_Toc126889363)

[Cấu trúc dữ liệu set trong c++ 13](#_Toc126889364)

[std::set trong C++ 14](#_Toc126889365)

[Khai báo set trong C++ 14](#_Toc126889366)

[Khai báo 1 set trong C++ 14](#_Toc126889367)

[Khai báo đồng thời nhiều set trong C++ 15](#_Toc126889368)

[Khởi tạo set trong C++ 15](#_Toc126889369)

[Khai báo set 2 chiều trong C++ 16](#_Toc126889370)

[Truy cập phần tử trong set C++ 17](#_Toc126889371)

[vector vs set trong C++ 19](#_Toc126889372)

[Duyệt set 20](#_Toc126889373)

[Duyệt set trong C++ bằng vòng lặp dựa trên phạm vi 20](#_Toc126889374)

[Duyệt set trong C++ bằng iterator 21](#_Toc126889375)

[Lấy kích thước set trong C++ bằng hàm size 22](#_Toc126889376)

[Thêm chèn phần tử vào set trong C++ 23](#_Toc126889377)

[Chèn 1 phần tử vào set trong C++ bằng hàm insert 23](#_Toc126889378)

[Chèn nhiều phần tử vào set trong C++ bằng hàm insert 25](#_Toc126889379)

[Xoá phần tử 26](#_Toc126889380)

[Xóa 1 phần tử trong set bằng hàm erase c++ 26](#_Toc126889381)

[Xóa các phần tử trong một phạm vi chỉ định bằng set erase c++ 28](#_Toc126889382)

[Sao chép set trong C++ 29](#_Toc126889383)

[Hoán đổi 2 set trong C++ 30](#_Toc126889384)

[Tính tổng 33](#_Toc126889385)

[Tính tổng các phần tử trong set C++ bằng std::accumulate 33](#_Toc126889386)

[Tính tổng các phần tử trong set C++ bằng vòng lặp 33](#_Toc126889387)

[Set trống 34](#_Toc126889388)

[Kiểm tra set trống trong C++ bằng hàm empty 34](#_Toc126889389)

[Làm trống 1 set trong C++ bằng hàm clear 35](#_Toc126889390)

[Multiset 36](#_Toc126889391)

[Tổng quan 36](#_Toc126889392)

[Multiset trong c++ là gì 36](#_Toc126889393)

[Cấu trúc dữ liệu multiset trong c++ 37](#_Toc126889394)

[std::multiset trong C++ 38](#_Toc126889395)

[Khai báo multiset trong C++ 38](#_Toc126889396)

[Khai báo 1 multiset trong C++ 38](#_Toc126889397)

[Khai báo đồng thời nhiều multiset trong C++ 40](#_Toc126889398)

[Khởi tạo multiset trong C++ 40](#_Toc126889399)

[Khởi tạo multiset 2 chiều trong C++ 40](#_Toc126889400)

[Truy cập phần tử trong multiset C++ 42](#_Toc126889401)

[vector vs multiset trong C++ 44](#_Toc126889402)

[Duyệt phần tử 44](#_Toc126889403)

[Duyệt multiset trong C++ bằng vòng lặp dựa trên phạm vi 44](#_Toc126889404)

[Duyệt multiset trong C++ bằng iterator 46](#_Toc126889405)

[Lấy kích thước 46](#_Toc126889406)

[Lấy kích thước multiset trong C++ bằng hàm size 46](#_Toc126889407)

[Chèn, thêm phần tử 48](#_Toc126889408)

[Chèn 1 phần tử vào multiset trong C++ bằng hàm insert 48](#_Toc126889409)

[Chèn nhiều phần tử vào multiset trong C++ bằng hàm insert 49](#_Toc126889410)

[Xoá phần tử 50](#_Toc126889411)

[Xóa 1 phần tử trong multiset bằng hàm erase c++ 50](#_Toc126889412)

[Xóa các phần tử trong một phạm vi chỉ định bằng multiset erase c++ 51](#_Toc126889413)

[Sao chép phần tử 53](#_Toc126889414)

[Hoán đổi 2 multiset trong C++ 54](#_Toc126889415)

[Tính tổng 56](#_Toc126889416)

[Tính tổng các phần tử trong multiset C++ bằng std::accumulate 56](#_Toc126889417)

[Tính tổng các phần tử trong multiset C++ bằng vòng lặp 57](#_Toc126889418)

[Tìm phần tử 57](#_Toc126889419)

[Tìm phần tử trong multiset C++ bằng hàm find 57](#_Toc126889420)

[Tìm phần tử trong multiset C++ bằng hàm equal\_range 58](#_Toc126889421)

[Tìm phần tử trong multiset C++ bằng hàm lower\_bound 59](#_Toc126889422)

[Tìm phần tử trong multiset C++ bằng hàm upper\_bound 61](#_Toc126889423)

[Đếm số lần xuất hiện của phần tử trong multiset C++ bằng hàm count 62](#_Toc126889424)

[Multiset trống 62](#_Toc126889425)

[Kiểm tra multiset trống trong C++ bằng hàm empty 62](#_Toc126889426)

[Làm trống 1 multiset trong C++ bằng hàm clear 63](#_Toc126889427)

[Map 64](#_Toc126889428)

[Map trong c++ là gì 64](#_Toc126889429)

[Cấu trúc dữ liệu map trong c++ 65](#_Toc126889430)

[std::map trong C++ 67](#_Toc126889431)

[Khai báo map trong C++ 67](#_Toc126889432)

[Khai báo 1 map trong C++ 67](#_Toc126889433)

[Khai báo đồng thời nhiều map trong C++ 68](#_Toc126889434)

[Gán giá trị cho map trong C++ 68](#_Toc126889435)

[Khởi tạo map trong C++ 69](#_Toc126889436)

[Truy cập phần tử trong map C++ 71](#_Toc126889437)

[Truy cập phần tử trong map C++ bằng toán tử [] 71](#_Toc126889438)

[Truy cập phần tử trong map C++ bằng hàm at() 72](#_Toc126889439)

[set và map trong c++ 72](#_Toc126889440)

[Duyệt map 73](#_Toc126889441)

[Duyệt map trong C++ bằng vòng lặp dựa trên phạm vi 73](#_Toc126889442)

[Duyệt map trong C++ bằng iterator 73](#_Toc126889443)

[Lấy kích thước 74](#_Toc126889444)

[Lấy kích thước map trong C++ bằng hàm size 74](#_Toc126889445)

[Thêm phần tử 75](#_Toc126889446)

[Thêm 1 phần tử vào map trong C++ bằng toán tử [] 75](#_Toc126889447)

[Chèn 1 phần tử vào map trong C++ bằng hàm insert 76](#_Toc126889448)

[Thêm chèn 1 phần tử vào map trong C++ bằng hàm emplace 79](#_Toc126889449)

[Chèn nhiều phần tử vào map trong C++ bằng hàm insert 80](#_Toc126889450)

[Xoá phần tử 81](#_Toc126889451)

[Xóa 1 phần tử trong map bằng hàm erase c++ 81](#_Toc126889452)

[Xóa các phần tử trong một phạm vi chỉ định bằng map erase c++ 83](#_Toc126889453)

[Sao chép, hoán đổi phần tử 85](#_Toc126889454)

[Sao chép map trong C++ 85](#_Toc126889455)

[Hoán đổi 2 map trong C++ 86](#_Toc126889456)

[Tìm phần tử 89](#_Toc126889457)

[Tìm phần tử trong map C++ bằng hàm find 89](#_Toc126889458)

[Tìm phần tử trong map C++ bằng hàm equal\_range 90](#_Toc126889459)

[Tìm phần tử trong map C++ bằng hàm lower\_bound 91](#_Toc126889460)

[Tìm phần tử trong map C++ bằng hàm upper\_bound 92](#_Toc126889461)

[Đếm số lần xuất hiện của phần tử trong map C++ bằng hàm count 94](#_Toc126889462)

[Map trống 94](#_Toc126889463)

[Kiểm tra map trống trong C++ bằng hàm empty 94](#_Toc126889464)

[Làm trống 1 map trong C++ bằng hàm clear 95](#_Toc126889465)

[Multimap 96](#_Toc126889466)

[Tổng quan 96](#_Toc126889467)

[Multimap trong c++ là gì 96](#_Toc126889468)

[Cấu trúc dữ liệu multimap trong c++ 97](#_Toc126889469)

[std::multimap trong C++ 98](#_Toc126889470)

[Khai báo multimap trong C++ 98](#_Toc126889471)

[Khai báo 1 multimap trong C++ 98](#_Toc126889472)

[Khai báo đồng thời nhiều multimap trong C++ 99](#_Toc126889473)

[Khởi tạo multimap trong C++ 100](#_Toc126889474)

[Truy cập phần tử trong multimap C++ 101](#_Toc126889475)

[multimap vs map, unordered\_map trong c++ 102](#_Toc126889476)

[Duyệt phần tử 102](#_Toc126889477)

[Duyệt multimap trong C++ bằng vòng lặp dựa trên phạm vi 102](#_Toc126889478)

[Duyệt multimap trong C++ bằng iterator 103](#_Toc126889479)

[Lấy kích thước 104](#_Toc126889480)

[Lấy kích thước multimap trong C++ bằng hàm size 104](#_Toc126889481)

[Thêm phần tử 105](#_Toc126889482)

[Sự khác biệt khi thêm chèn phần tử vào multimap với các container khác 105](#_Toc126889483)

[Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm insert 106](#_Toc126889484)

[Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm emplace 108](#_Toc126889485)

[Chèn nhiều phần tử vào multimap trong C++ bằng hàm insert 109](#_Toc126889486)

[Xoá phần tử 111](#_Toc126889487)

[Xóa 1 phần tử trong multimap bằng hàm erase c++ 111](#_Toc126889488)

[Xóa các phần tử trong một phạm vi chỉ định bằng multimap erase c++ 113](#_Toc126889489)

[Sao chép, hoán đổi phần tử 114](#_Toc126889490)

[Sao chép multimap trong C++ 114](#_Toc126889491)

[Hoán đổi 2 multimap trong C++ 116](#_Toc126889492)

[Tìm phần tử 118](#_Toc126889493)

[Tìm phần tử trong multimap C++ bằng hàm find 118](#_Toc126889494)

[Tìm phần tử trong multimap C++ bằng hàm equal\_range 119](#_Toc126889495)

[Tìm phần tử trong multimap C++ bằng hàm lower\_bound 121](#_Toc126889496)

[Tìm phần tử trong multimap C++ bằng hàm upper\_bound 122](#_Toc126889497)

[Đếm số lần xuất hiện của phần tử trong multimap C++ bằng hàm count 123](#_Toc126889498)

[Multimap trống 124](#_Toc126889499)

[Kiểm tra multimap trống trong C++ bằng hàm empty 124](#_Toc126889500)

[Làm trống 1 multimap trong C++ bằng hàm clear 125](#_Toc126889501)

[Unordered\_map 126](#_Toc126889502)

[Tổng quan 126](#_Toc126889503)

[Unordered\_map trong c++ là gì 126](#_Toc126889504)

[Cấu trúc dữ liệu unordered\_map trong c++ 127](#_Toc126889505)

[std::unordered\_map trong C++ 128](#_Toc126889506)

[Khai báo unordered\_map trong C++ 128](#_Toc126889507)

[Khai báo 1 unordered\_map trong C++ 128](#_Toc126889508)

[Khai báo đồng thời nhiều unordered\_map trong C++ 129](#_Toc126889509)

[Gán giá trị cho unordered\_map trong C++ 130](#_Toc126889510)

[Khởi tạo unordered\_map trong C++ 130](#_Toc126889511)

[Truy cập phần tử trong unordered\_map C++ 132](#_Toc126889512)

[Truy cập phần tử trong unordered\_map C++ bằng toán tử [] 132](#_Toc126889513)

[Truy cập phần tử trong unordered\_map C++ bằng hàm at() 133](#_Toc126889514)

[Duyệt phần tử 134](#_Toc126889515)

[Duyệt unordered\_map trong C++ bằng vòng lặp dựa trên phạm vi 134](#_Toc126889516)

[Duyệt unordered\_map trong C++ bằng iterator 134](#_Toc126889517)

[Lấy kích thước 135](#_Toc126889518)

[Lấy kích thước unordered\_map trong C++ bằng hàm size 135](#_Toc126889519)

[Thêm, chèn phần tử 137](#_Toc126889520)

[Thêm 1 phần tử vào unordered\_map trong C++ bằng toán tử [] 137](#_Toc126889521)

[Chèn 1 phần tử vào unordered\_map trong C++ bằng hàm insert 138](#_Toc126889522)

[Thêm chèn 1 phần tử vào unordered\_map trong C++ bằng hàm emplace 141](#_Toc126889523)

[Chèn nhiều phần tử vào unordered\_map trong C++ bằng hàm insert 142](#_Toc126889524)

[Xoá phần tử 143](#_Toc126889525)

[Xóa 1 phần tử trong unordered\_map bằng hàm erase c++ 143](#_Toc126889526)

[Xóa các phần tử trong một phạm vi chỉ định bằng unordered\_map erase c++ 145](#_Toc126889527)

[Sao chép phần tử 147](#_Toc126889528)

[Sao chép unordered\_map trong C++ 147](#_Toc126889529)

[Hoán đổi 2 unordered\_map trong C++ 148](#_Toc126889530)

[Tìm phần tử 150](#_Toc126889531)

[Tìm phần tử trong unordered\_map C++ bằng hàm find 150](#_Toc126889532)

[Tìm phần tử trong unordered\_map C++ bằng hàm equal\_range 151](#_Toc126889533)

[Đếm số lần xuất hiện của phần tử trong unordered\_map C++ bằng hàm count 153](#_Toc126889534)

[Unordered\_map trống 154](#_Toc126889535)

[Kiểm tra unordered\_map trống trong C++ bằng hàm empty 154](#_Toc126889536)

[Làm trống 1 unordered\_map trong C++ bằng hàm clear 155](#_Toc126889537)

[Bucket 155](#_Toc126889538)

[Bucket trong C++ là gì 155](#_Toc126889539)

[Lấy vị trí bucket chứa phần tử có khóa chỉ định bằng hàm unordered\_map bucket 156](#_Toc126889540)

[Lấy số bucket tạo ra bởi unordered\_map bằng hàm bucket\_count 157](#_Toc126889541)

[Lấy số phần tử lưu trữ trong bucket bằng hàm bucket\_size 158](#_Toc126889542)

[File 159](#_Toc126889543)

[Sử dụng hàm thừa kế từ ngôn ngữ C 159](#_Toc126889544)

[File trong C++ là gì 159](#_Toc126889545)

[Xử lý file trong C++ 160](#_Toc126889546)

[Tạo con trỏ file 161](#_Toc126889547)

[Mở file 161](#_Toc126889548)

[Đọc dữ liệu từ file 161](#_Toc126889549)

[Ghi file 162](#_Toc126889550)

[Đóng file 163](#_Toc126889551)

[Xử lý lỗi khi thao tác với file trong C++ 163](#_Toc126889552)

[Mở và đóng file 165](#_Toc126889553)

[Mở file trong C++ bằng hàm fopen 165](#_Toc126889554)

[Mở file binary file trong C++ 167](#_Toc126889555)

[Mở file trong C++ bằng hàm fopen\_s 167](#_Toc126889556)

[Sự khác biệt giữa hàm fopen và fopen\_s 169](#_Toc126889557)

[Xử lý lỗi khi mở file trong C++ 170](#_Toc126889558)

[Tạo file mới trong C++ 170](#_Toc126889559)

[Đóng file trong C++ bằng hàm fclose 171](#_Toc126889560)

[Đọc file trong C++ (fgetc, fgets, fscanf, sscanf) 172](#_Toc126889561)

[Đọc từng ký tự trong file bằng hàm fgetc 173](#_Toc126889562)

[Đọc từng dòng file trong C++ bằng hàm fgets 174](#_Toc126889563)

[Đọc từng dòng file khi đối số size nhỏ hơn số ký tự 176](#_Toc126889564)

[Đọc từng dòng file theo định dạng chỉ định bằng hàm fscanf 178](#_Toc126889565)

[Đọc từng dòng file theo định dạng chỉ định bằng hàm sscanf 181](#_Toc126889566)

[Đọc file csv trong C++ 183](#_Toc126889567)

[Ghi file trong C++ (fputc, fputs,fprintf) 187](#_Toc126889568)

[Ghi ký tự vào file bằng hàm fputc 187](#_Toc126889569)

[Ghi chuỗi vào file trong C++ bằng hàm fputs 188](#_Toc126889570)

[Ghi dữ liệu vào file trong C++ theo định dạng bằng hàm fprintf 189](#_Toc126889571)

[Ứng dụng hàm fprintf, chúng ta có thể thực hiện ghi mảng vào file trong C++. 191](#_Toc126889572)

[Ứng dụng hàm fprintf, chúng ta có thể thực hiện ghi cấu trúc vào file trong C++. 193](#_Toc126889573)

[Ghi file CSV trong C++ 194](#_Toc126889574)

[Thao tác với file trong C++ bằng phương pháp mới 196](#_Toc126889575)

[Phương pháp xử lý file mới trong C++ 196](#_Toc126889576)

[ofstream trong C++ 196](#_Toc126889577)

[ifstream trong C++ 201](#_Toc126889578)

[fstream 204](#_Toc126889579)

[Đọc file trong C++ bằng toán tử >> trong ifstream 204](#_Toc126889580)

[Đọc file trong C++ bằng bằng hàm getline 206](#_Toc126889581)

[Đọc file csv trong C++ 210](#_Toc126889582)

[Ghi file trong C++ bằng toán tử << trong ofstream 213](#_Toc126889583)

[Ghi file trong C++ bằng hàm write trong fstream 215](#_Toc126889584)

[Ghi mảng vào file trong C++ 216](#_Toc126889585)

[Ghi cấu trúc vào file trong C++ 218](#_Toc126889586)

[Ghi file CSV trong C++ 220](#_Toc126889587)

[OOP 221](#_Toc126889588)

[Hướng đối tượng trong C++ là gì 221](#_Toc126889589)

[Đối tượng trong c++ là gì 223](#_Toc126889590)

[Class 224](#_Toc126889591)

[Class trong C++ là gì 224](#_Toc126889592)

[Khai báo class trong C++ 225](#_Toc126889593)

[Định nghĩa class trong C++ | class.h 225](#_Toc126889594)

[Tạo hàm thành viên trong class | class.cpp 226](#_Toc126889595)

[Khai báo gộp class trong C++ 227](#_Toc126889596)

[Cách sử dụng class trong C++ 228](#_Toc126889597)

[Sử dụng class trong C++ bằng cách tạo instance 228](#_Toc126889598)

[Sử dụng class trong C++ không tạo instance 229](#_Toc126889599)

[Biến thành viên 231](#_Toc126889600)

[Khai báo biến thành viên trong C++ 231](#_Toc126889601)

[Khai báo biến thành viên trong C++ với const 232](#_Toc126889602)

[Hàm thành viên 233](#_Toc126889603)

[Khai báo hàm thành viên trong C++ 234](#_Toc126889604)

[Chia file khai báo và file mô tả hàm thành viên 234](#_Toc126889605)

[Khai báo nội tuyến hàm thành viên trong C++ 235](#_Toc126889606)

[Hàm thành viên dạng const trong C++ 236](#_Toc126889607)

[Gọi hàm thành viên trong class C++ 236](#_Toc126889608)

[Hàm khởi tạo 237](#_Toc126889609)

[Hàm khởi tạo trong c++ là gì 237](#_Toc126889610)

[Khai báo hàm khởi tạo trong C++ 238](#_Toc126889611)

[Hàm khởi tạo mặc định trong C++ (Default constructor) 239](#_Toc126889612)

[Hàm khởi tạo có tham số trong C++ 241](#_Toc126889613)

[Hàm huỷ 243](#_Toc126889614)

[Hàm hủy (Destructor) trong class C++ là gì 243](#_Toc126889615)

[Khai báo hàm hủy trong C++ 244](#_Toc126889616)

[Toán tử delete và cách gọi hàm hủy trong C++ 245](#_Toc126889617)

[Static 247](#_Toc126889618)

[Static trong class C++ là gì 247](#_Toc126889619)

[Biến static trong class C++ 247](#_Toc126889620)

[Hàm static trong class C++ 249](#_Toc126889621)

[This 251](#_Toc126889622)

[This trong class c++ là gì 251](#_Toc126889623)

[Cách viết this trong class c++ 251](#_Toc126889624)

[Access modifier 253](#_Toc126889625)

[Access modifier trong C++ là gì 253](#_Toc126889626)

[Kết thừa 254](#_Toc126889627)

[Kế thừa trong C++ là gì 254](#_Toc126889628)

[Cú pháp kế thừa trong C++ 254](#_Toc126889629)

[Truy cập biến và hàm thành viên của superclass trong C++ 256](#_Toc126889630)

[Kế thừa constructor trong c++ 256](#_Toc126889631)

[Đa kế thừa trong C++ 259](#_Toc126889632)

[Friend 262](#_Toc126889633)

[Friend trong C++ là gì 262](#_Toc126889634)

[Lớp bạn (Friend class) trong C++ 263](#_Toc126889635)

[Hàm bạn (Friend function) trong C++ 265](#_Toc126889636)

[Tham khảo 268](#_Toc126889637)

# Set

## Tổng quan

### Set trong c++ là gì

Set trong C++ là một tập hợp các phần tử duy nhất được sắp xếp theo thứ tự cụ thể, và được sử dụng làm tiêu chuẩn để xử lý các đối tượng chứa nhiều phần tử trong C++.

Mỗi phần tử trong set có giá trị phải là duy nhất, có nghĩa là nó không được trùng lặp với các giá trị còn lại trong set. Ngoài ra thì phần tử trong set không thể thay đổi giá trị, tuy nhiên chúng có thể được chèn hoặc xóa khỏi set.

Về mặt nội bộ, các phần tử trong set luôn được sắp xếp theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của nó. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một set, chúng sẽ tự động sắp xếp lại theo giá trị trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với set là multiset khi các phần tử trong chúng luôn được sắp xếp theo thứ tự. Tuy nhiên thì khác với multiset vốn cho phép các phần tử có thể trùng nhau cùng tồn tại, thì các phần tử trong set không được trùng nhau và luôn phải là duy nhất.

Nói tóm lại thì set trong C+++ sẽ giống như một mảng với các phần tử không trùng lặp và luôn được sắp xếp.

Về mặt tốc độ xử lý thì set có khả năng thêm, xóa, tìm kiếm dữ liệu với tốc độ cực cao là O(log N), và nó còn cao hơn cả vector với tốc là O(1). Tuy nhiên thì do các phần tử được quản lý dạng cây nhị phân nên tốc độ truy cập ngẫu nhiên tới một phần tử chỉ định trong set lại cực thấp.

Do đó, trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng set thay cho vector sẽ có lợi nhiều hơn.

Loại Truy cập ngẫu nhiên Thêm xóa tìm kiếm ngẫu nhiên

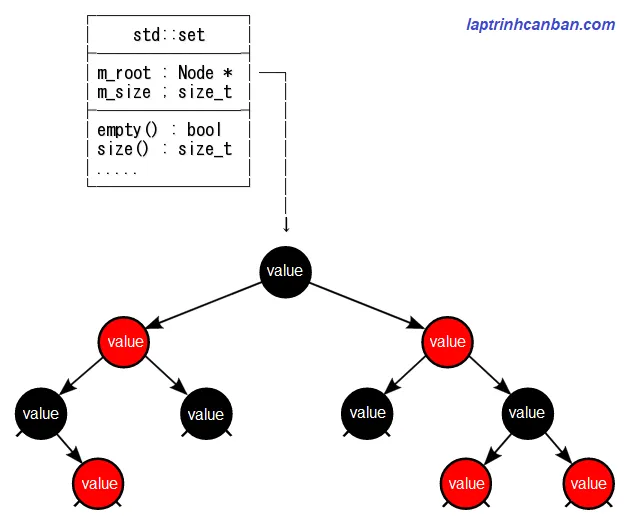
vector O(1) O(N)

set, multiset chậm O(log N)

### Cấu trúc dữ liệu set trong c++

Cấu trúc dữ liệu set trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu set trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ giá trị (value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái < Giá trị Node cha < Giá trị của Node con bên phải. Do trong set không tồn tại giá trị trùng nhau nên dấu < được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

### std::set trong C++

std::set được cài sẵn trong header file set và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <set> vào đầu chương trình.

## Khai báo set trong C++

### Khai báo 1 set trong C++

Cú pháp:

std::set<type> st;

Trong đó st là tên biến set và type là kiểu dữ liệu. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong set các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<double> name; //Khai báo set name kiểu double

    set<int> age;     //Khai báo set age kiểu int

}

Khi khai báo 1 set không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo set thuộc kiểu struct\*/

std::set<Person> st;

### Khai báo đồng thời nhiều set trong C++

Cú pháp:

set<type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<string> name, job, sex;

    set<int> age;

}

## Khởi tạo set trong C++

Cú pháp:

std::set<type> st {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* st là tên biến set
* value là các giá trị của set

VD:

std::set<string> user{"Kiyoshi", "male", "Tokyo"};

//{"Kiyoshi", "male", "Tokyo"}

### Khai báo set 2 chiều trong C++

Cú pháp:

using namespace std;

set<set<type> > st {l1, l2, l3, ...};

Trong đó:

* st là tên biến set 2 chiều
* l là các set 1 chiều được sử dụng như phần tử của set 2 chiều

Lưu ý, chúng ta cần phải viết thêm dấu cách giữa cặp dấu > > khi khai báo set 2 chiều. Lý do là để phân biệt với toán tử >> được sử dụng để dịch chuyển bit trong C++.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khai báo set 2 chiều\*/

    set<set<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

        {"Ajinomoto", "female", "Osaka"}};

    return 0;

}

Chúng ta cũng có thể khởi tạo các set 1 chiều trước rồi dùng chúng để khai báo set 2 chiều như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khởi tạo các set 1 chiều làm phần tử trong set 2 chiều\*/

    set<string> user1{"Kiyoshi", "male", "Hanoi"};

    set<string> user2{"Honda", "male", "Tokyo"};

    set<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo set 2 chiều\*/

    set<set<string> > all\_user{ user1, user2, user3};

    return 0;

}

## Truy cập phần tử trong set C++

Khác với vector hay mảng, do cấu trúc của set theo dạng cây chứ không phải dạng mảng nên chúng ta không thể truy cập ngẫu nhiên vào phần tử bất kỳ trong một set.Do vậy chúng ta cũng không thể sử dụng index của các phần tử để truy cập vào nó theo cách thông thường được.

Thay vào đó, chúng ta cần phải tiến hành truy cập tuần tự vào các phần tử của set, thông qua vòng lặp hoặc là trình lặp.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<string> user{"Kiyoshi", "male", "Tokyo"};

    int n=0, index = 2;

    for (string x: user) {

        /\*Truy cập vào phần tử thứ 2 trong set và kết thúc khi tìm thấy\*/

        if (n == index) {

            cout << x << endl;

            break;

        }

        ++n ;

    }

    return 0;

}

male

Chúng ta cũng có thể truy cập và in ra toàn bộ các phần tử trong set 1 chiều như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<string> user{"Kiyoshi", "male", "Tokyo"};

    for (string x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

Tokyo

male

Tương tự khi chúng ta cần truy cập vào phần tử trong set 2 chiều trong C++:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khởi tạo các set 1 chiều làm phần tử trong set 2 chiều\*/

    set<string> user1{"Kiyoshi", "male", "Hanoi"};

    set<string> user2{"Honda", "male", "Tokyo"};

    set<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo set 2 chiều\*/

    set<set<string> > all\_user{ user1, user2, user3};

    for (auto x: all\_user) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

    return 0;

}

Honda

male

Tokyo

Ajinomoto

female

Osaka

## vector vs set trong C++

Như đã phân tích ở trên thì giữa set và vector trong C++ có một số điểm khác biệt như sau:

Vector là mảng động còn set có cấu trúc cây nhị phân tạo bởi các Node

Vector chấp nhận phần tử trùng lặp còn set thì không. Phần tử trong set phải là duy nhất

Phần tử trong vector không được sắp xếp còn phần tử trong set thì được tự động sắp xếp thứ tự cụ thể.

Vector có tốc độ truy cập ngẫu nhiên nhanh hơn set, nhưng có tốc độ thêm xóa tìm kiếm ngẫu nhiên kém hơn set.

Thông thường trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng set thay cho vector sẽ có lợi nhiều hơn.

## Duyệt set

### Duyệt set trong C++ bằng vòng lặp dựa trên phạm vi

Bằng cách sử dụng vòng lặp dựa trên phạm vi, chúng ta có thể duyệt set trong C++ với cú pháp như sau:

for ( auto& x : v) {

    // Xử lý

}

Trong đó:

* v là tên set
* x là tên một biến dùng để gán từng phần tử được lấy từ set
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ set

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> data{5, 6, 88, -2};

    for (auto x: data) {

        cout << x << endl;

    }

}

-2

5

6

88

Nếu trong set chỉ chứa các phần tử thuộc kiểu dữ liệu nguyên thủy, chúng ta cũng có thể thay thế auto bằng tên kiểu, ví dụ set ở trên chỉ chứa các phần tử thuộc kiểu int nên chúng ta có thể viết:

set<int> data{5, 6, 88, -2};

for (int x: data) {

    cout << x << endl;

}

Một cách tương tự chúng ta cũng có thể dùng vòng lặp dựa trên phạm vi để duyệt set 2 chiều trong C++ như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<set<int> > data{{5, 2}, {6,3}, {88, -2}};

    for (auto x: data) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

}

-2

88

2

5

3

6

### Duyệt set trong C++ bằng iterator

Bằng cách sử dụng iterator, chúng ta có thể duyệt set trong C++ với cú pháp như sau:

for(auto itr = st.begin(); itr != st.end(); ++itr) {

cout << \*itr << endl;

}

Trong đó:

* st là tên set
* itr là tên iterator

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> data{5, 6, 88, -2};

    for(auto itr = data.begin(); itr != data.end(); ++itr) {

        cout << \*itr << endl;

    }

}

-2

5

6

88

## Lấy kích thước set trong C++ bằng hàm size

Cú pháp:

st.size();

Trong đó st là set cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 1, 4,5};

    cout<< st.size() <<endl;

    return 0;

}

4

Lưu ý, số phần tử hay kích thước của set ở đây được tính sau khi các phần tử được kiểm tra trùng lặp và sắp xếp trong set, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo set.

Ví dụ, nếu khi khai báo set mà tồn tại phần tử trùng lặp thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 3,  1, 4, 4, 5};

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout<< "\n "<< st.size() ;

    return 0;

}

1 3 4 5

4

## Thêm chèn phần tử vào set trong C++

### Chèn 1 phần tử vào set trong C++ bằng hàm insert

Cú pháp:

st.insert(value);

Trong đó st là set ban đầu, và value là phần tử cần chèn.

Hàm set insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp của set kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    //chèn phần tử đã tồn tại trong set

    st.insert(3);

    dump(st);

    //chèn phần tử chưa tồn tại trong set

    st.insert(4);

    dump(st);

    return 0;

}

1 3 5 6 7

1 3 5 6 7

1 3 4 5 6 7

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    auto r = st.insert(3);

    std::cout << r.second << "\n";

    // 0 : không chèn

    r = st.insert(4);

    std::cout << r.second << "\n";

    // 1 : đã chèn

    return 0;

}

### Chèn nhiều phần tử vào set trong C++ bằng hàm insert

Cú pháp:

st.insert(il);

Trong đó st là set ban đầu, và il là một initializer\_list chứa các phần tử cần chèn vào set.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Khai báo initializer\_list chứa phần tử cần chèn \*/

    int myints[]= {2,10,15};

    /\*Chèn initializer\_list vào set\*/

    st.insert (myints,myints+3);

    dump(st);

    return 0;

}

1 3 5 6 7

1 2 3 5 6 7 10 15

## Xoá phần tử

### Xóa 1 phần tử trong set bằng hàm erase c++

Cú pháp:

st.erase(itr);

OR

st.erase(value);

Trong đó st là set ban đầu, itr và value là trình lặp hoặc là giá trị của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(value) thì hàm sẽ tìm phần tử có giá trị bằng với value trong list và tiến hành xóa nó đi.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của set\*/

    auto itr = st.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong set

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    st.erase(itr);

    dump(st);

    //xóa phần tử có giá trị bằng 6 trong set

    st.erase(6);

    dump(st);

    return 0;

}

1 3 5 6 7

1 5 6 7

1 5 7

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    auto r = st.erase(2); //Xóa phần tử không tồn tại trong set

    std::cout << r << "\n";

    // 0

    r = st.erase(6); //Xóa phần tử tồn tại trong set

    std::cout << r << "\n";

    // 1

    return 0;

}

### Xóa các phần tử trong một phạm vi chỉ định bằng set erase c++

Cú pháp:

st.erase( iterator\_first, iterator\_last);

Trong đó st là set ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Lưu ý là phạm vi xóa là [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong set, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo set.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5, 9, 2};

    dump(st);

    int n;

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = st.begin();

    auto itr\_end = st.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    st.erase(itr\_start, itr\_end);

    dump(st);

    return 0;

}

1 2 3 5 6 7 9

1 2 7 9

## Sao chép set trong C++

set trong C++ thuộc kiểu dữ liệu đối tượng, do vậy khác với các kiểu dữ liệu nguyên thủy, chúng ta không thể sử dụng toán tử bằng = để gán và sao chép một set vào một set mới.

Thay vào đó, chúng ta sẽ sử dụng cách copy constructor trong set với cú pháp như sau:

std::set<type> st\_dest(st\_src);

Trong đó type là kiểu dữ liệu, st\_src là set nguồn để copy và st\_dest là set đích dùng để dán kết quả sao chép.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st\_src{3, 1, 4,5};

    cout << "Orginary set" << endl;

    dump (st\_src);

    /\*Sao chép set\*/

    set<int> st\_dest(st\_src);

    cout << "Copy set" << endl;

    dump (st\_dest);

    return 0;

}

Orginary set

1 3 4 5

Copy set

1 3 4 5

### Hoán đổi 2 set trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 set cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st1{3, 1, 4, 1, 5};

    set<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    st1.swap(st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 3 4 5

7 8 9

After swap

7 8 9

1 3 4 5

Ngoài cách dùng hàm set swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 set với nhau, cũng như là để hoán đổi các đối tượng khác như map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

VD:

#include <iostream>

#include <utility>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st1{3, 1, 4, 1, 5};

    set<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    swap(st1,st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

## Tính tổng

### Tính tổng các phần tử trong set C++ bằng std::accumulate

Để sử dụng hàm std::accumulate, chúng ta cần include header file numeric vào đầu chương trình như sau:

Cú pháp:

std::accumulate(st.begin(), st.end(), 0);

Trong đó st là tên set cần tính tổng các phần tử trong thứ tự các phần tử, còn st.begin() và st.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của set. Đối số cuối cùng (0) là giá trị khởi tạo của tổng.

VD:

#include <iostream>

#include <numeric>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 1, 4, 2, 5};

    int sum = accumulate(st.begin(), st.end(), 0);

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

### Tính tổng các phần tử trong set C++ bằng vòng lặp

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> st{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: st) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

## Set trống

### Kiểm tra set trống trong C++ bằng hàm empty

Cú pháp:

st.empty();

Trong đó st là set cần kiểm tra.

Hàm empty sẽ trả về true nếu set đã cho là set trống, cũng trả về false, nếu set đã cho có chứa phần tử.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    std::set<int> st;

    if( st.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    std::set<int> st2{1, 2, 3};

    if( st2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 set trong C++ bằng hàm clear

Cú pháp:

st.clear();

Trong đó st là set cần làm trống.

Khác với vector thì hàm set clear ngoài việc làm trống set chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    std::set<int> st{3, 1, 4};

    dump(st);

    st.clear();

    dump(st);

}

1 3 4

# Multiset

## Tổng quan

### Multiset trong c++ là gì

multiset trong C++ là một tập hợp các phần tử có thể trùng lặp được sắp xếp theo thứ tự cụ thể, và được sử dụng làm tiêu chuẩn để xử lý các đối tượng chứa nhiều phần tử trong C++.

Các phần tử trong multiset có thể trùng với các phần tử khác, ngoài ra thì phần tử trong multiset không thể thay đổi giá trị, tuy nhiên chúng có thể được chèn hoặc xóa khỏi multiset.

Về mặt nội bộ, các phần tử trong multiset luôn được sắp xếp theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của nó. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một multiset, chúng sẽ tự động sắp xếp lại theo giá trị trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với multiset là set khi các phần tử trong chúng luôn được sắp xếp theo thứ tự. Tuy nhiên thì khác với set vốn không cho phép các phần tử có thể trùng nhau cùng tồn tại, thì các phần tử trong multiset lại có thể trùng với các phần tử khác nó.

Nói tóm lại thì multiset trong C+++ sẽ giống như một mảng với các phần tử luôn được sắp xếp theo một thứ tự cụ thể.

Về mặt tốc độ xử lý thì multiset có khả năng thêm, xóa, tìm kiếm dữ liệu với tốc độ cực cao là O(log N), và nó còn cao hơn cả vector với tốc là O(1). Tuy nhiên thì giống với set thì do các phần tử được quản lý dạng cây nhị phân nên tốc độ truy cập ngẫu nhiên tới một phần tử chỉ định trong multiset lại cực thấp.

Do đó, trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng multiset thay cho vector sẽ có lợi nhiều hơn.

Loại Truy cập ngẫu nhiên Thêm xóa tìm kiếm ngẫu nhiên

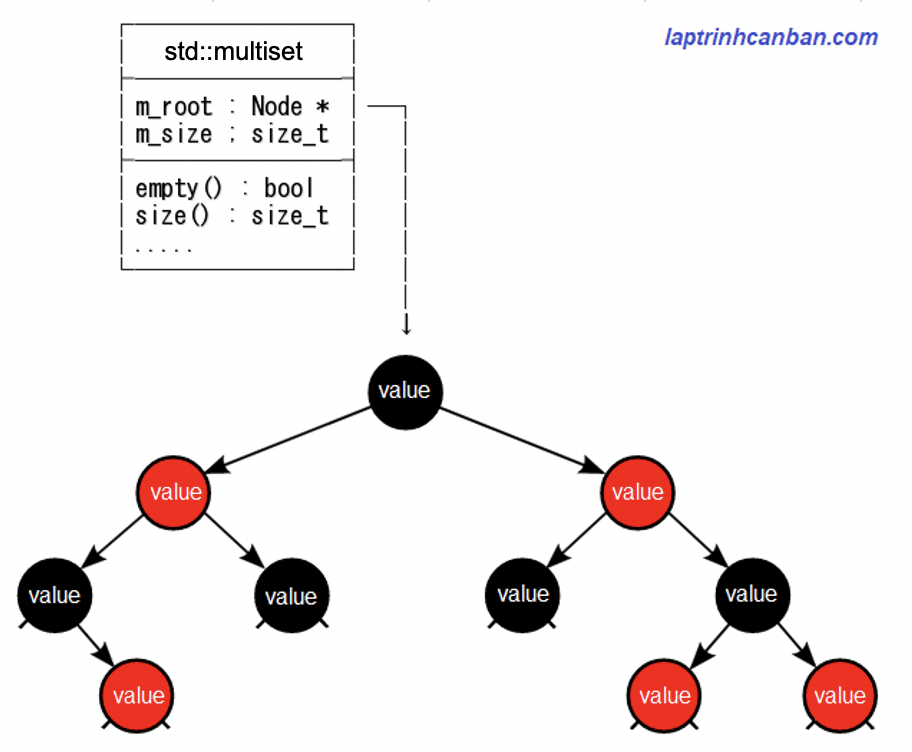
vector O(1) O(N)

multiset, set chậm O(log N)

### Cấu trúc dữ liệu multiset trong c++

Giống với set thì cấu trúc dữ liệu multiset trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu multiset trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ giá trị (value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái <= Giá trị Node cha <= Giá trị của Node con bên phải. Do trong multiset có thể trùng nhau nên dấu <= được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

### std::multiset trong C++

std::multiset trong C++ là một thư viện chuẩn được sử dụng để xử lý multiset trong C++.

Giống như std::set thì std::multiset được cài sẵn trong header file set và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <set> vào đầu chương trình.

Cần đặc biệt lưu ý ở đây là #include <set> chứ không phải là #include <multiset> .Khác với các container khác vốn có header file riêng biệt để quản lý thì multiset lại dùng chung header file với set.

#include <set>

int main()

{

    std::multiset<int> st1;

    std::multiset<double> st2;

}

## Khai báo multiset trong C++

### Khai báo 1 multiset trong C++

Cú pháp:

std::multiset<type> st;

Trong đó st là tên biến multiset và type là kiểu dữ liệu.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong multiset các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<double> name; //Khai báo multiset name kiểu double

    multiset<int> age;     //Khai báo multiset age kiểu int

}

Khi khai báo 1 multiset không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo multiset thuộc kiểu struct\*/

std::multiset<Person> st;

### Khai báo đồng thời nhiều multiset trong C++

Cú pháp:

multiset<type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<string> name, job, sex;

    multiset<int> age;

}

## Khởi tạo multiset trong C++

Cú pháp:

std::multiset<type> st {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* st là tên biến multiset
* value là các giá trị của multiset

VD:

std::multiset<string> user{"Kiyoshi", "male", "Tokyo"};

//{"Kiyoshi", "male", "Tokyo"}

## Khởi tạo multiset 2 chiều trong C++

Cú pháp:

multiset<multiset<type> > st {l1, l2, l3, ...};

Trong đó:

* st là tên biến multiset 2 chiều
* l là các multiset 1 chiều được sử dụng như phần tử của multiset 2 chiều

Lưu ý, chúng ta cần phải viết thêm dấu cách giữa cặp dấu > > khi khai báo multiset 2 chiều. Lý do là để phân biệt với toán tử >> được sử dụng để dịch chuyển bit trong C++.

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khai báo multiset 2 chiều\*/

    multiset<multiset<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

        {"Ajinomoto", "female", "Osaka"}};

    return 0;

}

Chúng ta cũng có thể khởi tạo các multiset 1 chiều trước rồi dùng chúng để khai báo multiset 2 chiều như sau:

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khởi tạo các multiset 1 chiều làm phần tử trong multiset 2 chiều\*/

    multiset<string> user1{"Kiyoshi", "male", "Hanoi"};

    multiset<string> user2{"Honda", "male", "Tokyo"};

    multiset<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo multiset 2 chiều\*/

    multiset<multiset<string> > all\_user{ user1, user2, user3};

    return 0;

}

## Truy cập phần tử trong multiset C++

Khác với vector hay mảng, do cấu trúc của multiset theo dạng cây chứ không phải dạng mảng nên chúng ta không thể truy cập ngẫu nhiên vào phần tử bất kỳ trong một multiset.

Do vậy chúng ta cũng không thể sử dụng index của các phần tử để truy cập vào nó theo cách thông thường được.

Thay vào đó, chúng ta cần phải tiến hành truy cập tuần tự vào các phần tử của multiset, thông qua vòng lặp hoặc là trình lặp mà Kiyoshi đã giới thiệu trong bài Duyệt multiset trong C++.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<string> user{"Kiyoshi", "male", "Tokyo"};

    int n=0, index = 2;

    for (string x: user) {

        /\*Truy cập vào phần tử thứ 2 trong multiset và kết thúc khi tìm thấy\*/

        if (n == index) {

            cout << x << endl;

            break;

        }

        ++n ;

    }

    return 0;

}

// male

Chúng ta cũng có thể truy cập và in ra toàn bộ các phần tử trong multiset 1 chiều như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<string> user{"Kiyoshi", "male", "Tokyo"};

    for (string x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

Tokyo

male

Tương tự khi chúng ta cần truy cập vào phần tử trong multiset 2 chiều trong C++:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khởi tạo các multiset 1 chiều làm phần tử trong multiset 2 chiều\*/

    multiset<string> user1{"Kiyoshi", "male", "Hanoi"};

    multiset<string> user2{"Honda", "male", "Tokyo"};

    multiset<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo multiset 2 chiều\*/

    multiset<multiset<string> > all\_user{ user1, user2, user3};

    for (auto x: all\_user) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

    return 0;

}

Honda

male

Tokyo

Ajinomoto

female

Osaka

## vector vs multiset trong C++

Như đã phân tích ở trên thì giữa multiset và vector trong C++ có một số điểm khác biệt như sau:

Vector là mảng động còn multiset có cấu trúc cây nhị phân tạo bởi các Node

Phần tử trong vector không được sắp xếp còn phần tử trong multiset thì được tự động sắp xếp thứ tự cụ thể.

Vector có tốc độ truy cập ngẫu nhiên nhanh hơn multiset, nhưng có tốc độ thêm xóa tìm kiếm ngẫu nhiên kém hơn multiset.

Thông thường trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng multiset thay cho vector sẽ có lợi nhiều hơn.

## Duyệt phần tử

### Duyệt multiset trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : v) {

    // Xử lý

}

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<int> data{5, 6, 5, 88, -2};

    for (auto x: data) {

        cout << x << endl;

    }

}

-2

5

5

6

88

Nếu trong multiset chỉ chứa các phần tử thuộc kiểu dữ liệu nguyên thủy, chúng ta cũng có thể thay thế auto bằng tên kiểu, ví dụ multiset ở trên chỉ chứa các phần tử thuộc kiểu int nên chúng ta có thể viết:

multiset<int> data{5, 6, 5, 88, -2};

for (int x: data) {

    cout << x << endl;

}

Một cách tương tự chúng ta cũng có thể dùng vòng lặp dựa trên phạm vi để duyệt multiset 2 chiều trong C++ như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<multiset<int> > data{{5, 2}, {6,3}, {88, -2}};

    for (auto x: data) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

}

-2

88

2

5

3

6

### Duyệt multiset trong C++ bằng iterator

Cú pháp:

for(auto itr = st.begin(); itr != st.end(); ++itr) {

cout << \*itr << endl;

}

Trong đó:

* st là tên multiset
* itr là tên iterator

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<int> data{5, 6, 5, 88, -2};

    for(auto itr = data.begin(); itr != data.end(); ++itr) {

        cout << \*itr << endl;

    }

}

-2

5

5

6

88

## Lấy kích thước

### Lấy kích thước multiset trong C++ bằng hàm size

Cú pháp:

st.size();

Trong đó st là multiset cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 1, 4, 5};

    cout<< st.size() <<endl;

    return 0;

}

4

Khác với set thì trong multiset cho phép tồn tại phần tử trùng lặp, nên khi đếm số phần tử có trong multiset thì hàm size() sẽ trả về toàn bộ các phần tử mà chúng ta đã dùng để khai báo set. Ví dụ:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 3,  1, 4, 4, 5};

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout<< "\n "<< st.size() ;

    return 0;

}

1 3 3 4 4 5

6

## Chèn, thêm phần tử

### Chèn 1 phần tử vào multiset trong C++ bằng hàm insert

Cú pháp:

st.insert(value);

Trong đó st là multiset ban đầu, và value là phần tử cần chèn.

Do multiset cho phép tồn tại các phần tử trùng lặp trong nó, nên chúng ta có thể chèn một phần tử không phân biệt việc nó đã tồn tại hay chưa trong multiset ban đầu.

Lại nữa, phần tử trong multiset được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multiset sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 3, 6, 7, 5};

    dump(st);

    //chèn phần tử đã tồn tại trong multiset

    st.insert(3);

    dump(st);

    //chèn phần tử chưa tồn tại trong multiset

    st.insert(4);

    dump(st);

    return 0;

}

1 3 5 6 7

1 3 3 5 6 7

1 3 3 4 5 6 7

### Chèn nhiều phần tử vào multiset trong C++ bằng hàm insert

Cú pháp:

st.insert(il);

Trong đó st là multiset ban đầu, và il là một initializer\_list chứa các phần tử cần chèn vào multiset.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Khai báo initializer\_list chứa phần tử cần chèn \*/

    int myints[]= {2, 3, 10};

    /\*Chèn initializer\_list vào multiset\*/

    st.insert (myints,myints+3);

    dump(st);

    return 0;

}

1 3 5 6 7

1 2 3 3 5 6 7 10

## Xoá phần tử

### Xóa 1 phần tử trong multiset bằng hàm erase c++

Cú pháp:

st.erase(itr);

OR

st.erase(value);

Trong đó st là multiset ban đầu, itr và value là trình lặp hoặc là giá trị của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(value) thì hàm sẽ xoá đi tất cả các phần tử có giá trị bằng với value trong list và tiến hành xóa nó đi.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    dump(st); // 1 2 3 3 5 6 7 9

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của multiset\*/

    auto itr = st.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong multiset với giá trị bằng 2

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    st.erase(itr);

    dump(st); //1 3 3 5 6 7 9

    //xóa tất cả các phần tử có giá trị bằng 3 trong multiset

    st.erase(3);

    dump(st); // 1 5 6 7 9

    return 0;

}

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    auto r = st.erase(4); //Xóa phần tử không tồn tại trong multiset

    std::cout << r << "\n";

    // 0

    r = st.erase(3); //Xóa phần tử tồn tại trong multiset

    std::cout << r << "\n";

    // 2

    return 0;

}

### Xóa các phần tử trong một phạm vi chỉ định bằng multiset erase c++

Cú pháp:

st.erase( iterator\_first, iterator\_last);

Trong đó st là multiset ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Lưu ý là phạm vi xóa là [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong multiset, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo multiset.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    dump(st);    // 1 2 3 3 5 6 7 9

    int n;

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = st.begin();

    auto itr\_end = st.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    st.erase(itr\_start, itr\_end);

    dump(st);   // 1 2 6 7 9

    return 0;

}

## Sao chép phần tử

Cú pháp:

std::multiset<type> st\_dest(st\_src);

Trong đó type là kiểu dữ liệu, st\_src là multiset nguồn để copy và st\_dest là multiset đích dùng để dán kết quả sao chép.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st\_src{3, 1, 4, 1, 5};

    cout << "Orginary multiset" << endl;

    dump (st\_src);

    /\*Sao chép multiset\*/

    multiset<int> st\_dest(st\_src);

    cout << "Copy multiset" << endl;

    dump (st\_dest);

    return 0;

}

Orginary multiset

1 1 3 4 5

Copy multiset

1 1 3 4 5

## Hoán đổi 2 multiset trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 multiset cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st1{3, 1, 4, 1, 5};

    multiset<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    st1.swap(st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 1 3 4 5

7 8 9

After swap

7 8 9

1 1 3 4 5

Ngoài cách dùng hàm multiset swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 multiset với nhau, cũng như là để hoán đổi các đối tượng khác như map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

#include <iostream>

#include <utility>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st1{3, 1, 4, 1, 5};

    multiset<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    swap(st1,st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 1 3 4 5

7 8 9

After swap

7 8 9

1 1 3 4 5

## Tính tổng

### Tính tổng các phần tử trong multiset C++ bằng std::accumulate

Để sử dụng hàm std::accumulate, chúng ta cần include header file numeric vào đầu chương trình như sau:

Cú pháp:

std::accumulate(st.begin(), st.end(), 0);

Trong đó st là tên multiset cần tính tổng các phần tử trong thứ tự các phần tử, còn st.begin() và st.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của multiset. Đối số cuối cùng (0) là giá trị khởi tạo của tổng.

VD:

#include <iostream>

#include <numeric>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 1, 4, 2, 5, 2};

    int sum = accumulate(st.begin(), st.end(), 0);

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 17

### Tính tổng các phần tử trong multiset C++ bằng vòng lặp

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<int> st{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: st) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 17

## Tìm phần tử

### Tìm phần tử trong multiset C++ bằng hàm find

Cú pháp:

st.find(val);

Trong đó val là giá trị của phần tử cần tìm trong multiset st.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong multiset. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multiset.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của phần tử đó trong multiset, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 1, 4, 5, 2, 9};

    dump(st);

    //Tìm phần tử có giá trị bằng 2 trong multiset

    auto itr = st.find(2);

    //Xóa phần tử vừa tìm thấy

    st.erase (itr);

    dump(st);

    return 0;

}

1 2 3 4 5 9

1 3 4 5 9

### Tìm phần tử trong multiset C++ bằng hàm equal\_range

VD:

st.equal\_range(val);

Trong đó val là giá trị phần tử cần tìm trong multiset st.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có giá trị giống giá trị chỉ định.

VD:

#include <iostream>

#include <set>

using namespace std;

typedef multiset<int>::iterator It;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 1, 2, 5, 7, 2, 3};

    dump(st);

    //Tìm phạm vi chứa phần tử có giá trị bằng 3 trong multiset

    pair<It,It> ret = st.equal\_range(2);

    //Xoá các phần tử trong phạm vi tìm thấy

    st.erase(ret.first,ret.second);

    dump(st);

    return 0;

}

1 2 2 3 3 5 7

1 3 3 5 7

Giống như trên, toàn bộ phạm vi chứa các phần tử có giá trị chỉ định đã được xác định bởi hàm equal\_range() và được xóa đi.

### Tìm phần tử trong multiset C++ bằng hàm lower\_bound

Cú pháp:

st.lower\_bound(val);

Trong đó val là giá trị của phần tử cần tìm trong multiset st.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng với giá trị chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multiset.

Trong trường hợp chỉ có một phần tử trong multiset có giá trị giống với giá trị chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó. Còn nếu có nhiều phần tử có giá trị giống với giá trị chỉ định, vị trí của phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định sẽ được hàm lower\_bound() trả về.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 3, 1, 4, 2, 5, 2, 9};

    dump(st);

    /\*Tìm vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng 2\*/

    auto itr1 = st.lower\_bound(2); // itr1 trỏ đến 2

    ///Tìm vị trí phần tử đầu tiên có giá trị lớn hơn 5 trong multimap

    auto itr2 = st.upper\_bound(5); // itr2 trỏ đến 9

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    st.erase (itr1, itr2);

    dump(st);

    return 0;

}

1 2 2 3 3 4 5 9

1 9

### Tìm phần tử trong multiset C++ bằng hàm upper\_bound

Cú pháp:

st.upper\_bound(key);

Trong đó key là giá trị của phần tử cần tìm trong multiset st.

Hàm upper\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multiset có giá trị giống với giá trị chỉ định thì hàm upper\_bound sẽ trả về con trỏ chỉ đến phần tử ngay sau nó. Còn nếu có nhiều phần tử có giá trị giống với giá trị chỉ định, vị trí của phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định sẽ được hàm upper\_bound() trả về.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 3, 1, 4, 2, 5, 2, 9};

    dump(st);

    /\*Tìm vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng 3\*/

    auto itr1 = st.lower\_bound(3); //itr1 trỏ đến 3

    //Tìm vị trí phần tử đầu tiên có giá trị lớn hơn 5

    auto itr2 = st.upper\_bound(5); //itr2 trỏ đến 9

    //In các phần tử trong phạm vi [itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << \*it << ' ';

    return 0;

}

1 2 2 3 3 4 5 9

3 3 4 5

### Đếm số lần xuất hiện của phần tử trong multiset C++ bằng hàm count

Cú pháp:

st.count(val);

Trong đó val là giá trị của phần tử cần đếm số lần xuất hiện trong multiset st.

VD:

#include <iostream>

#include <set>

using namespace std;

int main() {

    multiset<int> st{3, 1, 3, 5, 2, 3};

    //Đếm số lần xuất hiện của phần tử tồn tại trong set

    cout << st.count(3) <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong set

    cout << st.count(8) <<endl;

    return 0;

}

3

0

## Multiset trống

### Kiểm tra multiset trống trong C++ bằng hàm empty

Cú pháp:

st.empty();

Trong đó st là multiset cần kiểm tra.

Hàm empty sẽ trả về true nếu multiset đã cho là multiset trống, cũng trả về false, nếu multiset đã cho có chứa phần tử.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    std::multiset<int> st;

    if( st.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    std::multiset<int> st2{1, 2, 3};

    if( st2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 multiset trong C++ bằng hàm clear

Cú pháp:

st.clear();

Trong đó st là multiset cần làm trống.

Khác với vector thì hàm multiset clear ngoài việc làm trống multiset chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    std::multiset<int> st{3, 1, 4};

    dump(st);

    st.clear();

    dump(st);

}

1 3 4

# Map

### Map trong c++ là gì

Map trong C++ là một tập hợp các phần tử được sắp xếp theo thứ tự cụ thể, mà mỗi phần tử trong đó được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), với mỗi khóa là duy nhất trong map.

Trong map, các khóa (key) được sử dụng để sắp xếp và xác định giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong map là duy nhất và không được phép trùng lặp. Các giá trị trong map thì có thể trùng lặp, chúng có thể thay đổi giá trị, cũng như là được chèn hoặc xóa khỏi map.

Ví dụ cụ thể, một map chứa thông tin về một người sẽ gồm các khóa như name, old, weight. Tương ứng với các khóa này là các giá trị là thông tin cụ thể của người đó như sau:



Về mặt nội bộ, các phần tử trong map luôn được sắp xếp theo khóa của nó theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của map. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một map, chúng sẽ tự động sắp xếp lại theo khóa trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với map là unordered\_map khi các phần tử cũng được hình thành bởi các cặp khóa và giá trị. Tuy nhiên thì khác với unordered\_map có các phần tử không được sắp xếp, thì phần tử trong một map luôn luôn được sắp xếp theo khóa của chúng.

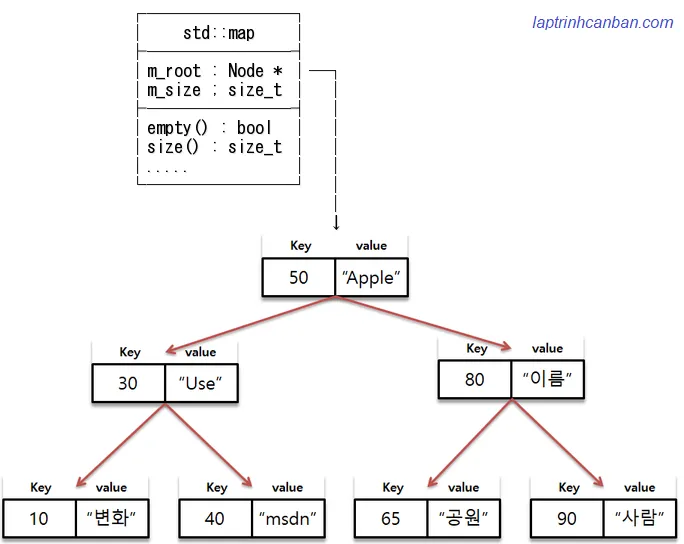
Về mặt tốc độ xử lý thì map có khả năng tìm kiếm dữ liệu theo khóa với tốc độ cực khá cao là O(log N). Tuy nhiên thì nó vẫn kém unordered\_map với tốc độ còn cao hơn là O(1), do unordered\_map sử dụng bảng hash.

Tuy nhiên thì map có lợi thế khi các phần tử của nó được sắp xếp sẵn, nên chúng ta nên dùng map trong các trường hợp cần ưu tiên tính chất này.

### Cấu trúc dữ liệu map trong c++

Cấu trúc dữ liệu map trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu map trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ cặp khóa:giá trị (key & value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái < Giá trị Node cha < Giá trị của Node con bên phải. Do trong map không cho phép các khóa trùng nhau nên dấu < được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

Nhờ vào cấu trúc dữ liệu kiểu này mà chúng ta có thể tìm kiếm nhị phân trong map, qua đó có thể tìm kiếm trong map với tốc độ cao O(Log N).

### std::map trong C++

std::map trong C++ là một thư viện chuẩn được sử dụng để xử lý map trong C++.

std::map được cài sẵn trong header file map và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <map> vào đầu chương trình.

#include <map>

int main()

{

    std::map<std::string, int> mp;

    std::map<char, double> mp2;

}

## Khai báo map trong C++

### Khai báo 1 map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp;

Trong đó mp là tên biến map và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong map các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<double, string> user;

    map<int, char> info;

}

Còn khi khai báo 1 map không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo map thuộc kiểu struct\*/

std::map<Person, int> mp;

### Khai báo đồng thời nhiều map trong C++

Cú pháp:

map<k\_type, v\_type> name1, name2, name3, ... ;

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string, int> name, job, sex;

    map<int, char> age;

}

## Gán giá trị cho map trong C++

Sau khi đã khai báo map, chúng ta có thể gán giá trị các phần tử vào nó bằng cách sử dụng toán tử [] với cú pháp sau đây:

mp[key] = value;

VD:

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

mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

mp["Honda"] = 2;   // {"Honda", 2}

mp["Suzuki"] = 3;  // {"Suzuki", 3}

Lại nữa, giá trị của phần tử có thể được ghi đè và thay thế bởi một giá trị mới nhiều lần như sau:

mp["Kiyoshi"] = 4; // {"Kiyoshi", 4}

mp["Kiyoshi"] = 8; // {"Kiyoshi", 8}

mp["Kiyoshi"] = 88;// {"Kiyoshi", 88}

Trong trường hợp thay đổi giá trị nhiều lần, thì giá trị trong lượt thay đổi giá trị cuối cùng sẽ được sử dụng làm giá trị phần tử.

## Khởi tạo map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến map
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý do mỗi khóa trong map là duy nhất, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì dù giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên sẽ được lưu vào trong map mà thôi.

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        std::cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

alpha: 10

beta: 20

gamma: 30

Trong trường hợp chỉ định các phần tử có khóa giống nhau, bất kể giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên được lưu vào map như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    return 0;

}

// alpha: 20

// beta: 20

// gamma: 30

## Truy cập phần tử trong map C++

### Truy cập phần tử trong map C++ bằng toán tử []

Cú pháp:

mp[key]

Nếu như key tồn tại trong map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, giá trị 0 sẽ được trả về

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị\*/

    std::map<std::string, int> mp;

    mp["Kiyoshi"] = 1;

    /\*Truy cập vào phần tử vừa gán bằng key\*/

    cout << mp["Kiyoshi"] <<   //1

    /\*Truy cập vào phần tử không tồn tại trong map\*/

    cout << mp["Honda"];       //0

    return 0;

}

Ngoài việc xuất giá trị, chúng ta cũng có thể thay đổi giá trị của một phần tử trong map với cách này. Ví dụ:

/\*Khai báo và gán giá trị\*/

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

mp["Kiyoshi"] = 1;

/\*Truy cập và thay đổi giá trị\*/

mp["Kiyoshi"] = 2;

### Truy cập phần tử trong map C++ bằng hàm at()

Hàm at() trong C++ là một hàm thành viên trong class std:map, có tác dụng truy cập vào phần tử trong map thông qua key của nó.

Nếu như key tồn tại trong map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, hàm at() sẽ trả về lỗi out\_of\_range.

Cú pháp:

mp.at(key);

Trong đó mp là tên map và key là khóa của phần tử cần truy cập.

/\*Khai báo và gán giá trị\*/

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

mp["Kiyoshi"] = 1;

/\*Truy cập vào phần tử vừa gán bằng key\*/

cout << mp.at("Kiyoshi"); //1

/\*Truy cập vào phần tử không tồn tại trong map\*/

cout << mp.at("Honda"); // throwing an instance of 'std::out\_of\_range

Có thể thấy rõ sự khác biệt giữa cách truy cập phần tử trong map bằng hàm at() và toán tử [] chính là ở kết quả khi chỉ định một key không tồn tại trong map.

## set và map trong c++

Giống với map thì kiểu set trong C++ cũng chứa các phần tử duy nhất được sắp xếp theo thứ tự trong nó. Và cấu trúc dữ liệu của 2 kiểu này cũng đều thuộc dạng cây nhị phân Red–black tree (cây đỏ đen).

Tuy vậy thì giữa chúng có 3 điểm khác biệt rất rõ ràng như sau:

Phần tử trong set tạo bởi 1 giá trị, trong khi trong map tạo bởi một cặp khóa và giá trị.

Phần tử trong set được sắp xếp theo giá trị của chúng, còn trong map thì theo khóa của chúng

Vùng bộ nhớ chứa set sẽ nhỏ hơn so với một map có cùng số phần tử, do trong map ngoài giá trị thì còn lưu trữ cả key nữa

Do vậy chúng ta có thể sử dụng một trong 2 kiểu này để xử lý các đối tượng mà cần có sự sắp xếp phần tử trong nó, tùy theo nhu cầu của mình.

## Duyệt map

### Duyệt map trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “: “ << x.second << endl;

}

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

}

alpha: 10

beta: 20

gamma: 30

### Duyệt map trong C++ bằng iterator

Cú pháp:

for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

cout << itr->first << ": "<< itr->second << "\n";

}

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

        cout  << itr->first << ": "<< itr->second << "\n";

    }

}

alpha: 10

beta: 20

gamma: 30

## Lấy kích thước

### Lấy kích thước map trong C++ bằng hàm size

Cú pháp:

mp.size();

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Lưu ý, số phần tử hay kích thước của map ở đây được tính sau khi các phần tử được kiểm tra key có trùng lặp và sắp xếp trong map, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo map.

Ví dụ, nếu khi khai báo map mà tồn tại các phần tử trùng lặp key thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

alpha: 20

beta: 20

gamma: 30

3

## Thêm phần tử

### Thêm 1 phần tử vào map trong C++ bằng toán tử []

Cú pháp:

mp[key] = value;

VD:

map<char,int> foo,bar;

foo['x']=100; //()

foo['y']=200;

Lưu ý do mỗi key trong map đều là duy nhất nên nếu chúng ta thêm một phần tử mới vào map nhưng lại có key trùng với một phần tử đã tồn tại trước đó, thì giá trị của phần tử mới này sẽ được dùng để ghi đè lên phần tử trước đó.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    dump(mymap);        // x:100 y:200 z:300

    mymap['z']=888;

    dump(mymap);        // x:100 y:200 z:888

}

### Chèn 1 phần tử vào map trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là map ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm map insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp trỏ đến map kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

Bởi vì các phần tử trong một map là duy nhất, nên thao tác chèn sẽ kiểm tra xem mỗi phần tử được chèn đã tồn tại trong map hay chưa. Nếu chưa tồn tại thì phần tử đó sẽ được chèn và ngược lại nếu đã tồn tại thì không được chèn.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    // duyệt map ban đầu

    dump(mp);

    //chèn phần tử với key chưa tồn tại trong map

    mp.insert(pair<char,int>('c',300));

    dump(mp);

    //chèn phần tử với key đã tồn tại trong map

    mp.insert(pair<char,int>('a',3));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

a:100 b:200

a:100 b:200 c:300

a:100 b:200 c:300 // Do key đã tồn tại nên sẽ không được chèn

a:100 b:200 c:300 x:50

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    //chèn phần tử với key chưa tồn tại trong map

    auto r = mp.insert(pair<char,int>('z',300));

    cout << r.second << "\n";   // Trả về 1 do được chèn

    //chèn phần tử với key tồn tại trong map,

    r = mp.insert(pair<char,int>('a',3));

    cout << r.second << "\n";  // Trả về 0 do không được chèn

    return 0;

}

### Thêm chèn 1 phần tử vào map trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là map ban đầu
* k,v là key và value.

Hàm map emplace sẽ thực hiện việc chèn phần tử nếu key của nó chưa tồn tại trong map. Ngược lại nếu key đó đã tòn tại, việc chèn thất bại và false sẽ được trả về

Và Hàm map emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào, nếu việc chèn thành công.

Lại nữa, phần tử trong map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    std::map<char,int> mymap;

    mymap.emplace('x',100);

    mymap.emplace('y',200);

    mymap.emplace('z',100);

    dump(mymap);

    return 0;

}

x:100 y:200 z:100

### Chèn nhiều phần tử vào map trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là map ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một map khác vào map ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một map khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo map mp1\*/

    map<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt map mp1

    dump(mp1);

    /\*Khai báo map mp2\*/

    map<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 4

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(4));

    // duyệt map mp2

    dump(mp2);

    return 0;

}

1: a 2: b 3: c 4: d 5: e

1: a 2: b 3: c

## Xoá phần tử

### Xóa 1 phần tử trong map bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là map ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong map và tiến hành xóa nó đi.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong map, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

Lưu ý phần tử trong map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử còn lại sau khi xóa phần tử trong map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ map.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mp;

    mp['x']=100;

    mp['y']=200;

    mp['z']=300;

    mp['t']=400;

    dump(mp);

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của map\*/

    auto itr = mp.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong map

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    mp.erase(itr);

    dump(mp);

    //Tìm vị trí và xóa phần tử có key bằng 'z' trong map

    mp.erase('z');

    dump(mp);

    //xóa phần tử có có key bằng 'y' trong map

    mp.erase('y');

    dump(mp);

    return 0;

}

t: 400 x: 100 y: 200 z: 300

t: 400 y: 200 z: 300

t: 400 y: 200

t: 400

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

map<char,int> mp;

mp['x']=100;

mp['y']=200;

mp['z']=300;

mp['t']=400;

auto r = mp.erase('t'); /\*r là số phần tử có key bằng 't' bị xóa\*/

std::cout << r;

//1

### Xóa các phần tử trong một phạm vi chỉ định bằng map erase c++

Cú pháp:

mp.erase( iterator\_first, iterator\_last);

Trong đó mp là map ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Lưu ý là phạm vi xóa sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong map, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo map.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=500;

    mp['f']=600;

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

a: 100 b: 200 c: 300 d: 400 e: 500 f: 600

a: 100 b: 200 f: 600

## Sao chép, hoán đổi phần tử

### Sao chép map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp\_des( mp\_src );

Trong đó type là kiểu dữ liệu, st\_src là map nguồn để copy và st\_dest là map đích dùng để dán kết quả sao chép.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    map<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary map" << endl;

    dump (mp\_src);

    /\*Sao chép map\*/

   map<string,int> mp\_des(mp\_src);

    cout << "Copy map" << endl;

    dump (mp\_des);

    return 0;

}

Orginary map

alpha: 10 beta: 20 gamma: 30

Copy map

alpha: 10 beta: 20 gamma: 30

### Hoán đổi 2 map trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 map cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

x: 100 y: 200

a: 11 b: 22 c: 33

After swap

a: 11 b: 22 c: 33

x: 100 y: 200

Ngoài cách dùng hàm map swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 map với nhau, cũng như là để hoán đổi các đối tượng khác như map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

VD:

#include <iostream>

#include <utility>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

x: 100 y: 200

a: 11 b: 22 c: 33

After swap

a: 11 b: 22 c: 33

x: 100 y: 200

## Tìm phần tử

### Tìm phần tử trong map C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong map. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong map

    auto ret = mp.equal\_range('c');

    //Xóa phần tử vừa tìm thấy

    mp.erase (ret.first,ret.second);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:2

a:1 b:2 d:1 e:2

### Tìm phần tử trong map C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Tuy nhiên do trong map thì các khóa là duy nhất, do đó nếu khóa tồn tại trong map thì hàm equal\_range sẽ trả về một khoảng chứa giá trị duy nhất đó mà thôi.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong map

    auto ret = mp.equal\_range('c');

    cout << "lower bound points to: ";

    cout << ret.first->first << " => " << ret.first->second << '\n';

    cout << "upper bound points to: ";

    cout << ret.second->first << " => " << ret.second->second << '\n';

    return 0;

}

a:1 b:2 c:3 d:1 e:2

lower bound points to: c => 3

upper bound points to: d => 1

### Tìm phần tử trong map C++ bằng hàm lower\_bound

Cú pháp:

mp.lower\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng với khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Và trong trường hợp chỉ có một phần tử trong map có khóa giống với khóa chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    dump(mp);

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong map\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa lớn hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    mp.erase (itr1, itr2);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:2 f:3

a:1 f:3

### Tìm phần tử trong map C++ bằng hàm upper\_bound

Hai hàm upper\_bound và lower\_bound đều tìm kiếm phần tử trong một danh sách đã xắp xếp (theo chiều tăng dần như set, map). Nhưng upper\_bound tìm kiếm phần tử đầu tiên có giá trị lớn hơn > giá trị tìm kiếm trong khi lower\_bound tìm kiếm phần tử đầu tiên có giá trị lớn hơn hoặc bằng giá trị trị tìm kiếm.

Tham khảo: <https://www.youtube.com/watch?v=dB2DWSKGLj8> đoạn 38p30’

Cú pháp:

mp.upper\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=300;

    mp['e']=300;

    mp['f']=300;

    /\*Duyệt map mp\*/

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl; //a:1 b:2 c:3 d:1 e:2

    //Tìm phần tử có khóa bằng 'b' trong map

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:200

    //Tìm vị trí phần tử  đầu tiên có khóa lớn hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:300

    //In các phần tử trong phạm vi (itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << (\*it).first << ":" << (\*it).second << ' ';

    return 0;

}

a:100 b:200 c:300 d:300 e:300 f:300

b:200 c:300 d:300 e:300

### Đếm số lần xuất hiện của phần tử trong map C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong map mp.

Do trong map các khóa là duy nhất, nên một phần tử nếu tồn tại cũng chỉ có xuất hiện 1 lần duy nhất trong map mà thôi. Do vậy, kết quả trả về của hàm count() cũng chỉ là 0 tương ứng với phần tử không tồn tại, hoặc 1 tương ứng với phần tử tồn tại trong map đó.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    //Đếm số lần xuất hiện của phần tử tồn tại trong map

    cout << mp.count('b') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong map

    cout << mp.count('f') <<endl;

    return 0;

}

1

0

## Map trống

### Kiểm tra map trống trong C++ bằng hàm empty

Cú pháp:

mp.empty();

Trong đó mp là map cần kiểm tra.

Hàm empty sẽ trả về true nếu map đã cho là map trống, cũng trả về false, nếu map đã cho có chứa phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    map<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 map trong C++ bằng hàm clear

Cú pháp:

mp.clear();

Trong đó mp là map cần làm trống.

Khác với vector thì hàm map clear ngoài việc làm trống map chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

# Multimap

## Tổng quan

### Multimap trong c++ là gì

Multimap trong C++ là một tập hợp các phần tử được sắp xếp theo thứ tự cụ thể, với mỗi phần tử được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), trong đó nhiều phần tử có thể có các khóa giống nhau.

Trong multimap, các khóa (key) được sử dụng để sắp xếp và xác định các giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong multimap được phép trùng lặp khiến nhiều phần tử có thể có các khóa tương đương. Do vậy, một khóa trong multimap có thể dùng để xác định nhiều giá trị được liên kết với nó.

Về mặt nội bộ, các phần tử trong multimap luôn được sắp xếp theo khóa của nó theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của multimap. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một multimap, chúng sẽ tự động sắp xếp lại theo khóa trước khi được lưu trữ nội bộ.

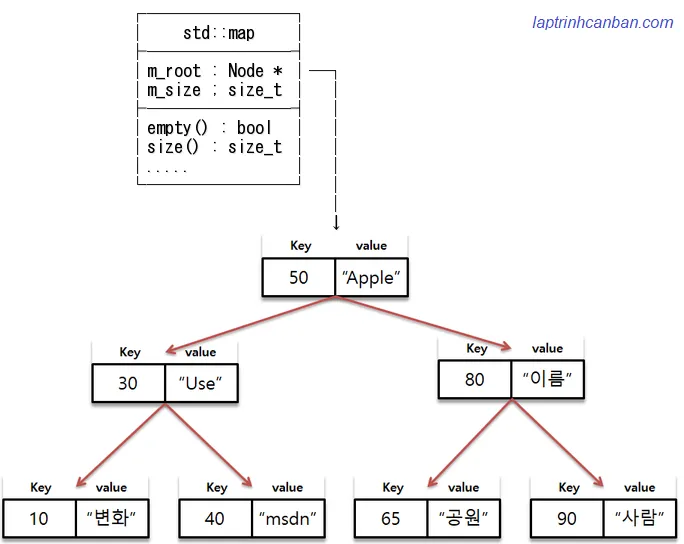
Khác với map không cho phép các khóa trùng lặp thì trong một multimap nhiều phần tử có thể có các khóa giống nhau.

Về mặt tốc độ xử lý thì multimap được cho là chậm hơn unordered\_map khi cần truy cập vào một phần tử ngẫu nhiên, nhưng lại nhanh hơn unordered\_map trong việc thực thi các trình lặp trỏ tới chúng.

### Cấu trúc dữ liệu multimap trong c++

Tương tự như kiểu map thì cấu trúc dữ liệu multimap trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu multimap trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ cặp khóa:giá trị (key & value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái <= Giá trị Node cha <= Giá trị của Node con bên phải. Do trong multimap cho phép khóa trùng nhau nên dấu <= được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

Nhờ vào cấu trúc dữ liệu kiểu này mà chúng ta có thể tìm kiếm nhị phân trong multimap, qua đó có thể tìm kiếm trong multimap với tốc độ cao O(Log N).

### std::multimap trong C++

std::multimap trong C++ là một thư viện chuẩn được sử dụng để xử lý multimap trong C++.

Giống như std::map thì std::multimap được cài sẵn trong header file map và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <map> vào đầu chương trình.

#include <map>

int main()

{

    std::multimap<std::string, int> mp;

    std::multimap<char, double> mp2;

}

## Khai báo multimap trong C++

### Khai báo 1 multimap trong C++

Cú pháp:

std::multimap<k\_type, v\_type> mp;

Trong đó mp là tên biến multimap và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong multimap các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<double, string> user;

    multimap<int, char> info;

}

Còn khi khai báo 1 multimap không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo multimap thuộc kiểu struct\*/

std::multimap<Person, int> mp;

### Khai báo đồng thời nhiều multimap trong C++

Cú pháp:

using namespace std;

multimap<k\_type, v\_type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string, int> name, job, sex;

    multimap<int, char> age;

}

## Khởi tạo multimap trong C++

Cú pháp:

std::multimap<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến multimap
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý khác với map thì khóa trong multimap có thể trùng lặp, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì tất cả chúng cũng sẽ được lưu vào map.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

alpha: 10

alpha: 20

alpha: 10

beta: 20

gamma: 30

## Truy cập phần tử trong multimap C++

Để truy cập vào phần tử trong multimap, cần sử dụng vòng lặp

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        if(x.first == "alpha") cout << x.first <<": " << x.second << '\n';

    }

    return 0;

}

## multimap vs map, unordered\_map trong c++

Trong C++ tồn tại 3 loại map gồm map, unordered\_map và multimap. Ba loại map này đều có phần tử được tạo thành bởi một cặp khóa và giá trị (key & value), tuy nhiên giữa chúng có những điểm hoàn toàn khác nhau như sau:

Phần tử trong map và multimap được sắp xếp theo thứ tự, còn unordered\_map thì không có thứ tự.

Phần tử trong map và multimap được lưu theo thứ tự của khóa, còn trong unordered\_map thì theo thứ tự giá trị hash của khóa.

map và multimap tuân theo cấu trúc cây nhị phân Red–black tree, còn unordered\_map thì theo giá trị trong bảng hash.

Key trong map và unordered\_map là duy nhất, còn trong multimap thì có thể trùng nhau.

Về tốc độ xử lý thì unordered\_map có tốc độ truy cập phần tử ngẫu nhiên nhanh hơn map và multimap, tuy nhiên tốc độ xử lý vòng lặp lại kém hơn so với 2 anh em của nó.

## Duyệt phần tử

### Duyệt multimap trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “: “ << x.second << endl;

}

Trong đó:

* mp là tên multimap.
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ multimap.
* x là tên một biến dùng để gán từng phần tử được lấy từ multimap.
* x.first và x.second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "alpha", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

}

alpha: 10

alpha: 20

beta: 20

gamma: 30

### Duyệt multimap trong C++ bằng iterator

Cú pháp:

for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

cout << itr->first << ": "<< itr->second << "\n";

}

Trong đó:

* mp là tên multimap
* itr là tên iterator dùng để trỏ đến phần tử
* itr->first và itr->second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "alpha", 20 },

                { "gamma", 30 } };

    for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

        cout  << itr->first << ": "<< itr->second << "\n";

    }

}

alpha: 10

alpha: 20

beta: 20

gamma: 30

## Lấy kích thước

### Lấy kích thước multimap trong C++ bằng hàm size

Cú pháp:

mp.size();

Trong đó mp là multimap cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Khác với map thì trong multimap cho phép tồn tại phần tử trùng lặp, nên khi đếm số phần tử có trong multiset thì hàm size() sẽ trả về toàn bộ các phần tử mà chúng ta đã dùng để khai báo map. Ví dụ:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

alpha: 20

alpha: 10

alpha: 10

beta: 20

gamma: 30

5

## Thêm phần tử

### Sự khác biệt khi thêm chèn phần tử vào multimap với các container khác

Để thêm chèn phần tử vào multimap trong C++, chúng ta sử dụng tới hàm thành viên insert() hoặc emplace(). Do khi chèn phần tử vào multimap thì vị trí chèn sẽ được tự động quyết định tùy thuộc vào bộ sắp xếp trong multimap, nên lưu ý khác với các containers khác như list hay vector thì trong multimap không tồn tại các hàm push\_back() hay push\_front() để thêm phần tử vào đầu hay cuối multimap.

Và khác với map thì trong multimap cũng không tồn tại toán tử [] nên chúng ta chỉ có thể dùng 2 hàm thành viên ở trên để thêm chèn phần tử mà thôi.

### Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là multimap ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm multimap insert sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào.

Lại nữa, phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    // duyệt multimap ban đầu

    dump(mp);

    //chèn phần tử bằng insert và pair

    mp.insert(pair<char,int>('z',40));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

a: 10 b: 20 c: 30

a: 10 b: 20 c: 30 z: 40

a: 10 b: 20 c: 30 x: 50 z: 40

Chúng ta cũng có thể lấy vị trí đã chèn của phần tử vào multimap bằng trình lặp thu về từ hàm, rồi truy xuất nó thông qua trình lặp như sau:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    auto itr = mp.insert(make\_pair('x',50));

    cout << (\*itr).first << " => " << (\*itr).second ;

    return 0;

}

x => 50

### Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là multimap ban đầu
* k,v là key và value.

Hàm multimap emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào.

Lại nữa, phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    // duyệt multimap ban đầu

    dump(mp);

    //chèn phần tử bằng emplace

    mp.emplace('z',40);

    dump(mp);

    //chèn và lấy khóa và giá trị của phần tử vừa chèn

    auto itr = mp.emplace('x',50);

    cout << (\*itr).first << " => " << (\*itr).second ;

    return 0;

}

a:10 b:20 c:30

a:10 b:20 c:30 z:40

x => 50

### Chèn nhiều phần tử vào multimap trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là multimap ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một multimap khác vào multimap ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một multimap khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo multimap mp1\*/

    multimap<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt multimap mp1

    cout <<"mp1= ";

    dump(mp1);

    /\*Khai báo multimap mp2\*/

    multimap<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 4

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(4));

    // duyệt multimap mp2

    cout <<"mp2= ";

    dump(mp2);

    return 0;

}

mp1= 1: a 2: b 3: c 4: d 5: e

mp2= 1: a 2: b 3: c

## Xoá phần tử

### Xóa 1 phần tử trong multimap bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là multimap ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong multimap và tiến hành xóa nó đi.

Hàm multimap erase sẽ trả về số lượng phần tử đã được xóa đi từ multimap ban đầu.

Lưu ý phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử còn lại sau khi xóa phần tử trong multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ multimap.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 5));

    mp.insert(make\_pair('d', 5));

    mp.insert(make\_pair('e', 6));

    dump(mp);

    /\*Tìm trình lặp trỏ đến phần tử có khóa bằng 'c' \*/

    auto itr = mp.find('c');

    //xóa phần tử tại vị trí itr chỉ đến

    mp.erase(itr);

    dump(mp);

    //xóa phần tử có khóa bằng 'd' trong multimap

    mp.erase('d');

    dump(mp);

    return 0;

}

a:1 b:2 c:5 d:5 e:6

a:1 b:2 d:5 e:6

a:1 b:2 e:6

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

VD:

auto r = mp.erase('d'); /\*r là số phần tử có key bằng 'd' bị xóa\*/

cout << r; //2

### Xóa các phần tử trong một phạm vi chỉ định bằng multimap erase c++

Cú pháp:

mp.erase( iterator\_first, iterator\_last);

Trong đó mp là multimap ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Lưu ý là phạm vi xóa bằng [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong multimap, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo multimap.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong multiset, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 5));

    mp.insert(make\_pair('d', 5));

    mp.insert(make\_pair('e', 6));

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

a:1 b:2 c:5 d:5 e:6

a:1 b:2

## Sao chép, hoán đổi phần tử

### Sao chép multimap trong C++

multimap trong C++ thuộc kiểu dữ liệu đối tượng, do vậy khác với các kiểu dữ liệu nguyên thủy, chúng ta không thể sử dụng toán tử bằng = để gán và sao chép một multimap vào một multimap mới.

Thay vào đó, chúng ta sẽ sử dụng cách copy constructor trong multimap với cú pháp như sau:

std::multimap<k\_type, v\_type> mp\_des( mp\_src );

Trong đó type là kiểu dữ liệu, st\_src là multimap nguồn để copy và st\_dest là multimap đích dùng để dán kết quả sao chép.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    multimap<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary multimap" << endl;

    dump (mp\_src);

    /\*Sao chép multimap\*/

   multimap<string,int> mp\_des(mp\_src);

    cout << "Copy multimap" << endl;

    dump (mp\_des);

    return 0;

}

Orginary multimap

alpha: 10 beta: 20 gamma: 30

Copy multimap

alpha: 10 beta: 20 gamma: 30

### Hoán đổi 2 multimap trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 multimap cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khởi tạo các multimap là foo và bar\*/

    multimap<char,int> foo = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    multimap<char,int> bar = {

            { 'x', 11 },

            { 'y', 21 } };

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

a: 10 b: 20 c: 30

x: 11 y: 21

After swap

x: 11 y: 21

a: 10 b: 20 c: 30

Ngoài cách dùng hàm multimap swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 multimap với nhau, cũng như là để hoán đổi các đối tượng khác như multimap, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

VD:

#include <iostream>

#include <map>

#include <utility>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khởi tạo các multimap là foo và bar\*/

    multimap<char,int> foo = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    multimap<char,int> bar = {

            { 'x', 11 },

            { 'y', 21 }};

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

a: 10 b: 20 c: 30

x: 11 y: 21

After swap

x: 11 y: 21

a: 10 b: 20 c: 30

## Tìm phần tử

### Tìm phần tử trong multimap C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong multimap. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multimap.

Lưu ý là hàm find() chỉ trả về một trình lặp cho một phần tử duy nhất mà thôi. Để có được toàn bộ phạm vi các phần tử có khóa chỉ định, hãy dùng hàm equal\_range mà Kiyoshi sẽ giới thiệu ở phần dưới.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của một phần tử thông qua khóa trong multimap, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    dump(mp);

    //Tìm phần tử có khóa bằng 'b' trong multimap

    auto itr = mp.find('e');

    //Xóa phần tử vừa tìm thấy

    mp.erase (itr);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:5 e:2 e:8 f:3

a:1 b:2 c:3 d:1 e:2 e:8 f:3

Giống như trên, mặc dù có 3 phần tử có khóa ‘e’ giống với khóa chỉ định, nhưng hàm find() chỉ trả về trình lặp trỏ đến phần tử đầu tiên mà thôi.

### Tìm phần tử trong multimap C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Bằng cách ứng dụng hàm equal\_range(), chúng ta có thể tìm ra phạm vi chứa kết quả tìm kiếm, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    dump(mp);

    //Tìm phần tử có khóa bằng 'b' trong multimap

    auto ret = mp.equal\_range('e');

    //Xóa phần tử vừa tìm thấy

    mp.erase (ret.first,ret.second);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:5 e:2 e:8 f:3

a:1 b:2 c:3 d:1 f:3

Giống như trên, toàn bộ phạm vi chứa các phần tử có khóa giống khóa ‘e’ chỉ định đã được xác định bởi hàm equal\_range() và được xóa đi.

### Tìm phần tử trong multimap C++ bằng hàm lower\_bound

Cú pháp:

mp.lower\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng với khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multimap có khóa giống với khóa chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó. Còn nếu có nhiều phần tử có khóa giống với khóa chỉ định, vị trí của phần tử đầu tiên có khóa lớn hơn khóa chỉ định sẽ được hàm lower\_bound() trả về.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    mp.insert(make\_pair('b', 9));

    dump(mp);   //a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong multimap\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa lớn hơn 'e' trong multimap

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    mp.erase (itr1, itr2);

    dump(mp);   //a:1 f:3

    return 0;

}

a:1 b:2 c:3 d:1 e:2 f:3

a:1 f:3

### Tìm phần tử trong multimap C++ bằng hàm upper\_bound

Cú pháp:

mp.upper\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm upper\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multimap có khóa giống với khóa chỉ định thì hàm upper\_bound sẽ trả về con trỏ chỉ đến phần tử ngay sau nó. Còn nếu có nhiều phần tử có khóa giống với khóa chỉ định, vị trí của phần tử đầu tiên có khóa lớn hơn khóa chỉ định sẽ được hàm upper\_bound() trả về.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    mp.insert(make\_pair('b', 9));

    /\*Duyệt map mp\*/

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

    //a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong map\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa nhỏ hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //In các phần tử trong phạm vi [itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << (\*it).first << ":" << (\*it).second << ' ';

    return 0;

}

a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

b:2 b:9 c:3 d:1 e:5 e:2 e:8

### Đếm số lần xuất hiện của phần tử trong multimap C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong multimap mp.

Hàm count() sẽ trả về số phần tử có khóa giống khóa chỉ định được tìm thấy trong multimap.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 100));

    mp.insert(make\_pair('b', 200));

    mp.insert(make\_pair('e', 500));

    mp.insert(make\_pair('c', 300));

    mp.insert(make\_pair('d', 100));

    mp.insert(make\_pair('e', 200));

    mp.insert(make\_pair('f', 300));

    //Đếm số lần xuất hiện của phần tử tồn tại trong multimap

    cout << mp.count('e') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong multimap

    cout << mp.count('x') <<endl;

    return 0;

}

2

0

## Multimap trống

### Kiểm tra multimap trống trong C++ bằng hàm empty

Cú pháp:

mp.empty();

Trong đó mp là multimap cần kiểm tra.

Hàm empty sẽ trả về true nếu multimap đã cho là multimap trống, cũng trả về false, nếu multimap đã cho có chứa phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    multimap<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 multimap trong C++ bằng hàm clear

Cú pháp:

mp.clear();

Trong đó mp là multimap cần làm trống.

Khác với vector thì hàm multimap clear ngoài việc làm trống multimap chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mymap = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

# Unordered\_map

## Tổng quan

### Unordered\_map trong c++ là gì

Unordered\_map trong C++ là một tập hợp các phần tử không có thứ tự, mà mỗi phần tử trong đó được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), với các khóa không được trùng nhau.

Trong unordered\_map, các khóa (key) được sử dụng để xác định giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong unordered\_map là duy nhất và không được phép trùng lặp. Các value trong unordered\_map thì có thể trùng lặp, chúng có thể thay đổi giá trị, cũng như là được chèn hoặc xóa khỏi unordered\_map.

Khác với map trong C++ với các phần tử được sắp xếp và lưu trữ theo thứ tự của các khóa, thì các phần tử trong unordered\_map lại được lưu trữ dựa trên giá trị hash của khóa chứ không phải dựa trên thứ tự của các khóa.

Ở đây, giá trị hash của khóa là giá trị thu về sau khi tính toán khóa bằng một hàm hash. Còn hàm hash là giải thuật nhằm sinh ra các giá trị hash tương ứng với mỗi khối dữ liệu (có thể là một chuỗi ký tự, một đối tượng trong lập trình hướng đối tượng, v.v…) để phân biệt các khối dữ liệu với nhau.

Điều đó có nghĩa là giá trị chỉ định làm key trong unordered\_map phải thuộc kiểu giá trị có thể được hash bằng hàm hash.

Về mặt tốc độ xử lý thì unordered\_map có khả năng truy cập vào các phần tử riêng lẻ nhanh hơn so với map, tuy nhiên thì việc lặp trên một phạm vi trong đó thì lại chậm hơn map.

Tuy nhiên thì trong unordered\_map các phần tử không được sắp xếp sẵn, nên để xử lý một số lượng phần tử không quá lớn thì chúng ta chỉ cần dùng tới map là đủ rồi.

### Cấu trúc dữ liệu unordered\_map trong c++

Khác với map hay multimap vốn có cấu trúc dữ liệu unordered\_map trong C++ thuộc dạng Red–black tree (cây đỏ đen), thì cấu trúc dữ liệu của unordered\_map trong c++ lại hoàn toàn khác.

Nó dựa vào hash table- một bảng hash chứa các giá trị đã được hash để so sánh các key của phần tử, và quyết định thứ tự lưu giữ các phần tử này bên trong nó.

Chúng ta cần đặc biệt lưu ý 3 điểm sau đây về Cấu trúc dữ liệu unordered\_map trong c++:

* Khác với vector thì unordered\_map không dựa vào index mà dựa vào key để truy cập phần tử.
* Các phần tử trong unordered\_map được lưu vào dựa vào giá trị sau khi đã hash của các key, chứ không phải là được sắp xếp theo giá trị của key.
* Giống với map thì key trong unordered\_map là duy nhất và một key chỉ có thể dùng để xác định một giá trị duy nhất mà thôi.

Và cũng nhờ việc phần tử trong unordered\_map được lưu giữ theo giá trị hash của key mà chúng ta có thể truy cập vào phần tử riêng lẻ trong nó với tốc độ tức thì O(1).

### std::unordered\_map trong C++

std::unordered\_map trong C++ là một thư viện chuẩn được sử dụng để xử lý unordered\_map trong C++.

std::unordered\_map được cài sẵn trong header file unordered\_map và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <unordered\_map> vào đầu chương trình.

#include <iostream>

#include <unordered\_map>

int main()

{

    std::unordered\_map<std::string, int> mp;

    std::unordered\_map<char, double> mp2;

}

## Khai báo unordered\_map trong C++

### Khai báo 1 unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp;

Trong đó mp là tên biến unordered\_map và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value.

Tuy nhiên đối với kiểu của key, chúng ta chỉ có thể chỉ định các kiểu dữ liệu có khả năng được hash bởi hash function mà thôi. Ví dụ như các kiểu dữ liệu nguyên thủy thuộc kiểu số, kiểu chữ, hoặc là con trỏ chẳng hạn.

Nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải thêm vào đối số thứ ba, là một hàm hash sử dụng để hash các dữ liệu này.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<double, string> user;

    unordered\_map<int, char> info;

}

Còn khi khai báo 1 unordered\_map không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải thêm vào đối số thứ 3 là hash function như ví dụ sau:

class HashVI {  // Hàm hash object

public:

    size\_t operator()(const vector<int> &x) const {

        const int C = 997;

        size\_t t = 0;

        for (int i = 0; i != x.size(); ++i) {

            t = t \* C + x[i];

        }

        return t;

    }

};

std::unordered\_map<vector<int>, int, HashVI> mp2;

### Khai báo đồng thời nhiều unordered\_map trong C++

Cú pháp:

using namespace std;

unordered\_map<k\_type, v\_type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string, int> name, job, sex;

    unordered\_map<int, char> age;

}

## Gán giá trị cho unordered\_map trong C++

Cú pháp:

mp[key] = value;

Trong đó

* mp là tên biến unordered\_map
* key và value là khóa và giá trị của phần tử cần gán vào unordered\_map

VD:

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

mp["Honda"] = 2;   // {"Honda", 2}

mp["Suzuki"] = 3;  // {"Suzuki", 3}

Lại nữa, giá trị của phần tử có thể được ghi đè và thay thế bởi một giá trị mới nhiều lần như sau:

mp["Kiyoshi"] = 4; // {"Kiyoshi", 4}

mp["Kiyoshi"] = 8; // {"Kiyoshi", 8}

mp["Kiyoshi"] = 88;// {"Kiyoshi", 88}

Trong trường hợp thay đổi giá trị nhiều lần, thì giá trị trong lượt thay đổi giá trị cuối cùng sẽ được sử dụng làm giá trị phần tử.

## Khởi tạo unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến unordered\_map
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý do mỗi khóa trong unordered\_map là duy nhất, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì dù giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên sẽ được lưu vào trong unordered\_map mà thôi.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        std::cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

gamma: 30

beta: 20

alpha: 10

Trong trường hợp chỉ định các phần tử có khóa giống nhau, bất kể giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên được lưu vào unordered\_map như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    return 0;

}

// beta: 20

// gamma: 30

// alpha: 20

## Truy cập phần tử trong unordered\_map C++

### Truy cập phần tử trong unordered\_map C++ bằng toán tử []

Cú pháp:

mp[key]

Trong đó

* mp là tên biến unordered\_map
* key là khóa của phần tử cần truy cập trong unordered\_map

Nếu như key tồn tại trong unordered\_map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, giá trị 0 sẽ được trả về

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị\*/

    unordered\_map<string, int> mp;

    mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

    mp["Honda"] = 2;   // {"Honda", 2}

    mp["Suzuki"] = 3;  // {"Suzuki", 3}

    /\*Truy cập vào phần tử vừa gán bằng key\*/

    cout << mp["Kiyoshi"] ;   //1

    /\*Truy cập vào phần tử không tồn tại trong unordered\_map\*/

    cout << mp["Honda"];       //0

    return 0;

}

Ngoài việc xuất giá trị, chúng ta cũng có thể thay đổi giá trị của một phần tử trong unordered\_map với cách này. Ví dụ:

/\*Khai báo và gán giá trị\*/

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập và thay đổi giá trị\*/

mp["Kiyoshi"] = 2;

### Truy cập phần tử trong unordered\_map C++ bằng hàm at()

Cú pháp:

mp.at(key);

Trong đó mp là tên unordered\_map và key là khóa của phần tử cần truy cập.

Nếu như key tồn tại trong unordered\_map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, hàm at() sẽ trả về lỗi out\_of\_range.

VD:

/\*Khai báo và gán giá trị\*/

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập vào phần tử vừa gán bằng key\*/

cout << mp.at("Kiyoshi"); //1

/\*Truy cập vào phần tử không tồn tại trong unordered\_map\*/

cout << mp.at("Honda"); // throwing an instance of 'std::out\_of\_range

## Duyệt phần tử

### Duyệt unordered\_map trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “:” << x.second << endl;

}

Trong đó:

* mp là tên unordered\_map.
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ unordered\_map.
* x là tên một biến dùng để gán từng phần tử được lấy từ unordered\_map.
* x.first và x.second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ":" << x.second << endl;

    }

}

gamma:30

beta:20

alpha:10

### Duyệt unordered\_map trong C++ bằng iterator

Cú pháp:

for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

cout << itr->first << ":"<< itr->second << "\n";

}

Trong đó:

* mp là tên unordered\_map
* itr là tên iterator dùng để trỏ đến phần tử
* itr->first và itr->second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for(auto itr = mp.begin(); itr != mp.end(); ++itr) {

        cout  << itr->first << ":"<< itr->second << "\n";

    }

}

gamma:30

beta:20

alpha:10

## Lấy kích thước

### Lấy kích thước unordered\_map trong C++ bằng hàm size

Cú pháp:

mp.size();

Trong đó mp là unordered\_map cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Lưu ý, số phần tử hay kích thước của unordered\_map ở đây được tính sau khi các phần tử được kiểm tra key có trùng lặp và sắp xếp trong unordered\_map, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo unordered\_map.

Ví dụ, nếu khi khai báo unordered\_map mà tồn tại các phần tử trùng lặp key thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ":" << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

beta:20

gamma:30

alpha:20

3

## Thêm, chèn phần tử

### Thêm 1 phần tử vào unordered\_map trong C++ bằng toán tử []

Cú pháp:

mp[key] = value;

Trong đó

* mp là tên biến unordered\_map
* key và value là khóa và giá trị của phần tử cần thêm vào unordered\_map

VD:

unordered\_map<char,int> foo,bar;

foo['x']=100; //()

foo['y']=200;

Lưu ý do mỗi key trong unordered\_map đều là duy nhất nên nếu chúng ta thêm một phần tử mới vào unordered\_map nhưng lại có key trùng với một phần tử đã tồn tại trước đó, thì giá trị của phần tử mới này sẽ được dùng để ghi đè lên phần tử trước đó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    dump(mymap);        // z:300 y:200 x:100

    mymap['z']=888;

    dump(mymap);        // z:888 y:200 x:100

}

### Chèn 1 phần tử vào unordered\_map trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là unordered\_map ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm unordered\_map insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp trỏ đến unordered\_map kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

Bởi vì các phần tử trong một unordered\_map là duy nhất, nên thao tác chèn sẽ kiểm tra xem mỗi phần tử được chèn đã tồn tại trong unordered\_map hay chưa. Nếu chưa tồn tại thì phần tử đó sẽ được chèn và ngược lại nếu đã tồn tại thì không được chèn.

Trong trường hợp tất cả các phần tử chèn vào vốn đã tồn tại từ trước trong unordered\_map, thì bản thân unordered\_map ban đầu sẽ được trả về.

Lại nữa, phần tử trong unordered\_map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào unordered\_map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    // duyệt unordered\_map ban đầu

    dump(mp);

    //chèn phần tử với key chưa tồn tại trong unordered\_map

    mp.insert(pair<char,int>('c',300));

    dump(mp);

    //chèn phần tử với key đã tồn tại trong unordered\_map

    mp.insert(pair<char,int>('a',3));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

b:200 a:100

c:300 b:200 a:100

c:300 b:200 a:100 // Do key đã tồn tại nên sẽ không được chèn

x:50 c:300 b:200 a:100

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    //chèn phần tử với key chưa tồn tại trong unordered\_map

    auto r = mp.insert(pair<char,int>('z',300));

    cout << r.second << "\n";   // Trả về 1 do được chèn

    //chèn phần tử với key tồn tại trong unordered\_map,

    r = mp.insert(pair<char,int>('a',3));

    cout << r.second << "\n";  // Trả về 0 do không được chèn

    return 0;

}

### Thêm chèn 1 phần tử vào unordered\_map trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là unordered\_map ban đầu
* k,v là key và value.

Hàm unordered\_map emplace sẽ thực hiện việc chèn phần tử nếu key của nó chưa tồn tại trong unordered\_map. Ngược lại nếu key đó đã tồn tại, việc chèn thất bại và false sẽ được trả về.

Hàm unordered\_map emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào, nếu việc chèn thành công.

Lại nữa, phần tử trong unordered\_map được lưu vào dựa theo giá trị hash của các khóa, nên các phần tử cần chèn thêm vào unordered\_map sẽ được tự động quyết định vị trí dựa trên giá trị hash ở trên bảng hash, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    std::unordered\_map<char,int> mymap;

    mymap.emplace('x',100);

    mymap.emplace('y',200);

    mymap.emplace('z',100);

    dump(mymap);

    return 0;

}

z:100 y:200 x:100

### Chèn nhiều phần tử vào unordered\_map trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là unordered\_map ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một unordered\_map khác vào unordered\_map ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một unordered\_map khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo unordered\_map mp1\*/

    unordered\_map<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt unordered\_map mp1

    dump(mp1);

    /\*Khai báo unordered\_map mp2\*/

    unordered\_map<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 2

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(2));

    // duyệt unordered\_map mp2

    dump(mp2);

    return 0;

}

5: e 4: d 3: c 2: b 1: a

3: c 4: d 5: e

## Xoá phần tử

### Xóa 1 phần tử trong unordered\_map bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là unordered\_map ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong unordered\_map và tiến hành xóa nó đi.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong unordered\_map, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

Lưu ý do phần tử trong unordered\_map không được sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ unordered\_map.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=400;

    dump(mp);

    /\*Tìm trình lặp trỏ đến phần tử có khóa bằng 'c' \*/

    auto itr = mp.find('c'); // Chỉ đến c:300

    //xóa phần tử tại vị trí itr chỉ đến

    mp.erase(itr);

    dump(mp);

    //xóa phần tử có khóa bằng 'd' trong multimap

    mp.erase('d');

    dump(mp);

    return 0;

}

e:400 d:400 c:300 b:200 a:100

e:400 d:400 b:200 a:100

e:400 b:200 a:100

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

unordered\_map<char,int> mp;

mp['x']=100;

mp['y']=200;

mp['z']=300;

mp['t']=400;

auto r = mp.erase('t'); /\*r là số phần tử có key bằng 't' bị xóa\*/

std::cout << r;

//1

### Xóa các phần tử trong một phạm vi chỉ định bằng unordered\_map erase c++

Cú pháp:

mp.erase( iterator\_first, iterator\_last);

Trong đó mp là unordered\_map ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Lưu ý là phạm vi xóa là [iterator\_first,iterator\_last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Ứng dụng điều này, chúng ta có thể chỉ định trình lặp và xóa đi các phần tử trong phạm vi chỉ định từ unordered\_map như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=500;

    mp['f']=600;

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

f:600 e:500 d:400 c:300 b:200 a:100

f:600 e:500 a:100

## Sao chép phần tử

### Sao chép unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp\_des( mp\_src );

Trong đó type là kiểu dữ liệu, st\_src là unordered\_map nguồn để copy và st\_dest là unordered\_map đích dùng để dán kết quả sao chép.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    unordered\_map<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary unordered\_map" << endl;

    dump (mp\_src);

    /\*Sao chép unordered\_map\*/

   unordered\_map<string,int> mp\_des(mp\_src);

    cout << "Copy unordered\_map" << endl;

    dump (mp\_des);

    return 0;

}

Orginary unordered\_map

gamma:30 beta:20 alpha:10

Copy unordered\_map

gamma:30 beta:20 alpha:10

### Hoán đổi 2 unordered\_map trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 unordered\_map cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

y:200 x:100

c:33 b:22 a:11

After swap

c:33 b:22 a:11

y:200 x:100

Ngoài cách dùng hàm unordered\_map swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 unordered\_map với nhau, cũng như là để hoán đổi các đối tượng khác như unordered\_map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

#include <iostream>

#include <utility>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

y:200 x:100

c:33 b:22 a:11

After swap

c:33 b:22 a:11

y:200 x:100

## Tìm phần tử

### Tìm phần tử trong unordered\_map C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong unordered\_map mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong unordered\_map. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong unordered\_map.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của phần tử đó trong unordered\_map, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khoá bằng 'c' trong multiset

    auto itr = mp.find('c');

    //Xóa phần tử vừa tìm thấy

    mp.erase (itr);

    dump(mp);

    return 0;

}

e:2 d:1 c:3 b:2 a:1

e:2 d:1 b:2 a:1

### Tìm phần tử trong unordered\_map C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong unordered\_map mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Tuy nhiên do trong unordered\_map thì các khóa là duy nhất, do đó nếu khóa tồn tại trong unordered\_map thì hàm equal\_range sẽ trả về một khoảng chứa giá trị duy nhất đó mà thôi.

Bằng cách ứng dụng hàm equal\_range(), chúng ta có thể tìm ra phạm vi chứa phần tử có khóa chỉ định trong unordered\_map như sau

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong unordered\_map

    auto ret = mp.equal\_range('c');

    cout << "lower bound points to: ";

    cout << ret.first->first << ":" << ret.first->second << '\n';

    cout << "upper bound points to: ";

    cout << ret.second->first << ":" << ret.second->second << '\n';

    return 0;

}

e:2 d:1 c:3 b:2 a:1

lower bound points to: c:3

upper bound points to: b:2

### Đếm số lần xuất hiện của phần tử trong unordered\_map C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong unordered\_map mp.

Do trong unordered\_map các khóa là duy nhất, nên một phần tử nếu tồn tại cũng chỉ có xuất hiện 1 lần duy nhất trong unordered\_map mà thôi.

Do vậy, kết quả trả về của hàm count() cũng chỉ là 0 tương ứng với phần tử không tồn tại, hoặc 1 tương ứng với phần tử tồn tại trong unordered\_map đó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main() {

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    //Đếm số lần xuất hiện của phần tử tồn tại trong unordered\_map

    cout << mp.count('b') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong unordered\_map

    cout << mp.count('f') <<endl;

    return 0;

}

1

0

## Unordered\_map trống

### Kiểm tra unordered\_map trống trong C++ bằng hàm empty

Cú pháp:

mp.empty();

Trong đó mp là unordered\_map cần kiểm tra.

Hàm empty sẽ trả về true nếu unordered\_map đã cho là unordered\_map trống, cũng trả về false, nếu unordered\_map đã cho có chứa phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    unordered\_map<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 unordered\_map trong C++ bằng hàm clear

Cú pháp:

mp.clear();

Trong đó mp là unordered\_map cần làm trống.

Khác với vector thì hàm unordered\_map clear ngoài việc làm trống unordered\_map chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

## Bucket

### Bucket trong C++ là gì

Bucket trong C++ là một không gian bộ nhớ trong bảng hash của container (unordered\_map), trong đó các phần tử sẽ được chỉ định dựa trên giá trị hash của khóa của chúng. Phạm vi giới hạn hợp lệ của các bucket là từ 0 đến bucket\_count - 1.

### Lấy vị trí bucket chứa phần tử có khóa chỉ định bằng hàm unordered\_map bucket

Cú pháp:

mp.bucket(k);

Trong đó mp là unordered\_map và k là khóa cần tìm số bucket chứa phần tử được xác định bởi nó.

Hàm bucket sẽ trả về vị trí (số thứ tự) của bucket chứa phần tử có khóa k được chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mymap = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

  for (auto& x: mymap) {

    std::cout << "Phần tử [" << x.first << ":" << x.second << "]";

    std::cout << " nằm tại bucket số #" << mymap.bucket (x.first) << std::endl;

  }

  return 0;

}

Phần tử [gamma:30] nằm tại bucket số #2

Phần tử [beta:20] nằm tại bucket số #0

Phần tử [alpha:10] nằm tại bucket số #0

### Lấy số bucket tạo ra bởi unordered\_map bằng hàm bucket\_count

Cú pháp:

mp.bucket\_count();

Trong đó mp là unordered\_map cần đếm số bucket đã được tạo ra từ nó.

Hàm bucket\_count sẽ trả về số bucket được tạo ra từ unordered\_map chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    unsigned n = mp.bucket\_count();

    cout << "Có tất cả " << n << " buckets.\n";

    for (unsigned i=0; i<n; ++i) {

        cout << "bucket #" << i << " chứa: ";

        for (auto it = mp.begin(i); it!=mp.end(i); ++it)

            cout << "[" << it->first << ":" << it->second << "] ";

        cout << "\n";

    }

  return 0;

}

Có tất cả 13 buckets.

bucket #0 chứa:

bucket #1 chứa:

bucket #2 chứa:

bucket #3 chứa:

bucket #4 chứa:

bucket #5 chứa:

bucket #6 chứa: [a:1]

bucket #7 chứa: [b:2]

bucket #8 chứa: [c:3]

bucket #9 chứa: [d:1]

bucket #10 chứa: [e:2]

bucket #11 chứa:

bucket #12 chứa:

### Lấy số phần tử lưu trữ trong bucket bằng hàm bucket\_size

Cú pháp:

mp.bucket\_size(i);

Trong đó mp là unordered\_map và i là số thứ tự của bucket cần đếm số phần tử trong nó.

Hàm bucket\_size sẽ trả về số phần tử của unordered\_map được lưu trữ trong bucket có số thứ tự chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    unsigned nbuckets = mp.bucket\_count();

    cout << "Có tất cả " << nbuckets << " buckets.\n";

    for (unsigned i=0; i<nbuckets; ++i) {

        cout << "bucket #" << i << " có " << mp.bucket\_size(i) << " phần tử.\n";

    }

  return 0;

}

Có tất cả 13 buckets.

bucket #0 có 0 phần tử.

bucket #1 có 0 phần tử.

bucket #2 có 0 phần tử.

bucket #3 có 0 phần tử.

bucket #4 có 0 phần tử.

bucket #5 có 0 phần tử.

bucket #6 có 1 phần tử.

bucket #7 có 1 phần tử.

bucket #8 có 1 phần tử.

bucket #9 có 1 phần tử.

bucket #10 có 1 phần tử.

bucket #11 có 0 phần tử.

bucket #12 có 0 phần tử.

# File

## Sử dụng hàm thừa kế từ ngôn ngữ C

### File trong C++ là gì

Trong thế giới máy tính, file là một loại “Tài liệu” nhằm lưu dữ dữ liệu và thông tin. Có rất nhiều kiểu file khác nhau nhằm lưu dữ các kiểu dữ liệu và thông tin khác nhau, ví dụ như file text, file Excel, file Json, file XML, hay là file CSV v.v…

Để có thể thao tác với các loại file này, người dùng thông thường cần các phần mềm chuyên dùng để mở, đọc ghi và lưu chúng. Tuy nhiên thì với các lập trình viên như chúng ta thì bằng cách sử dụng sử dụng ngôn ngữ C++, chúng ta cũng có thể dễ dàng thực hiện các thao tác tương tự với file như vậy.

Cũng giống như mảng, chuỗi hay biến thì file trong C++ là một loại dữ liệu trong chương trình, và một file khi nhập vào chương trình cũng sẽ được lưu giữ tại một vùng nào đó trên bộ nhớ máy tính. Tuy nhiên khác với các loại dữ liệu khác có thể gán vào một biến để xử lý, thì để thao tác với file, chúng ta cần tạo ra một thực thể của kiểu cấu trúc FILE để chứa thông tin của file cần thao tác, sau đó sử dụng tới một con trỏ để chỉ đến vị trí của thực thể FILE chứa thông tin file đó trên bộ nhớ, qua đó thực hiện các thao tác với file thông qua các hàm có sẵn, với các chức năng cụ thể như đóng, mở, ghi hay lưu file.

Do vậy muốn thao tác được với file, bạn cần nắm vững các kiến thức cơ bản về con trỏ và kiểu cấu trúc trong C++. Bạn có thể tham khảo các bài viết chi tiết về con trỏ trong chuyên đề Con trỏ trong C++, cũng như về kiểu cấu trúc tại chuyên đề Kiểu cấu trúc trong C++.

### Xử lý file trong C++

Tương tự như với ngôn ngữ C thì quy trình xử lý file trong C++ sẽ gồm các bước như sau:



### Tạo con trỏ file

Mỗi file trong chương trình C++ được xử lý dưới dạng một thực thể của kiểu cấu trúc FILE - một kiểu cấu trúc được quy định sẵn trong ngôn ngữ C++ phục vụ cho việc xử lý file.

Về căn bản thì cách sử dụng kiểu cấu trúc này cũng tương tự như các kiểu cấu trúc mà chúng ta đã học trong chuyên đề Kiểu cấu trúc trong C++.

Để thao tác với file, trước tiên chúng ta cần phải tạo một con trỏ chỉ đến thực thể của kiểu cấu trúc FILE chứa thông tin của file đó trên bộ nhớ với cú pháp sau đây:

FILE \*fp;

Sau đó, bằng cách truy cập vào địa chỉ này thông qua con trỏ, chúng ta mới có thể thực hiện các thao tác với file.

### Mở file

Thông thường khi mở file trong máy tính, chúng ta có thể click đúp chuột vào nó, hoặc là mở nó trong một chương trình đặc định. Tuy nhiên khi mở file trong chương trình C++, chúng ta cần phải sử dụng tới một số hàm chuyên dụng như fopen() hay fopen\_s().

Việc mở file và gán địa chỉ file vào con trỏ thường được tiến hành đồng thời. Ví dụ bạn có thể mở một file sample.txt bằng hàm fopen() và gán vào con trỏ file fp như sau:

fp = fopen("sample.txt", "r");

### Đọc dữ liệu từ file

Sau khi mở file, chúng ta đã có thể đọc các dữ liệu từ file vào trong chương trình để xử lý.

Có nhiều phương pháp để đọc dữ liệu từ một file. Ví dụ bạn có thể đọc từng dòng của file, hoặc đọc toàn bộ nội dung file. Nội dung file có thể được đọc dưới dạng từng ký tự, dạng chuỗi, hoặc dưới dạng mảng để có thể dễ dàng xử lý trong chương trình.

Để đọc file trong C++, chúng ta có thể sử dụng các hàm kế thừa từ C như là fgets() hay fgetc() chẳng hạn.

Ngoài ra với từng định dạng file mà chúng ta sẽ có các phương pháp đọc file khác nhau trong C++. Ví dụ như cùng sử dụng hàm fscanf() hoặc hàm sscanf() để đọc file, nhưng cú pháp đọc file txt sẽ khác cách đọc file CSV chẳng hạn.

Ví dụ cụ thể, sau khi mở file sample.txt ở trên, chúng ta có thể đọc từng dòng file đó bằng hàm fgets như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char arr[128];

    //Mở file bằn hàm fopen

    fp= fopen("sample.txt", "r");

    fgets(arr, 128, fp);

    cout <<  arr;

    return 0;

}

### Ghi file

Chúng ta có thể tạo một file mới rồi ghi nội dung vào file đó, hoặc là mở một file sẵn và ghi thêm nội dung vào đó.

Chúng ta có thể sử dụng các hàm có sẵn như fputc,fputs và fprintf để làm được việc này.

Ví dụ cụ thể, sau khi mở file sample.txt ở trên, chúng ta có thể ghi dòng chữ “Hello Vietnam!” vào file đó bằng hàm fprintf như sau:

FILE \* fp = NULL;

fp = fopen("sample.txt", "r");

fprintf(fp, "%s", "Hello Vietnam!");

### Đóng file

Sau khi đã xử lý xong file trong chương trình, chúng ta cần phải đóng file đó lại. Việc đóng file sẽ giúp kết thúc phiên làm việc với file, và giải phóng bộ nhớ.

Nếu không đóng file thì file đó vẫn tồn tại trên bộ nhớ, dẫn đến xảy ra các sự cố về bộ nhớ trong chương trình.

Để đóng một file trong C++, chúng ta cần dùng đến hàm fclose() với cú pháp sau đây:

fclose(fp);

Trong đó fp là con trỏ dùng để mở file.

VD:

FILE \* fp = NULL;

fp = fopen("sample.txt", "r");

fprintf(fp, "%s", "Hello Vietnam!");

fclose(fp);

### Xử lý lỗi khi thao tác với file trong C++

Chúng ta có thể sử dụng hàm fopen() để mở một file trong C++, nhưng trong quá trình mở có thể gặp một số lỗi nào đó, dẫn dến việc không phải lúc nào bạn cũng có thể mở file đó thành công.

Sẽ tất nguy hiểm khi thực hiện quá trình thao tác với một tập tin mà không thể mở được, do đó thay vì để chương trình tiếp tục chạy như cũ thì chúng ta cần phải thiết kế các xử lý để chương trình tự xử lý lỗi, và chạy chương trình một cách chính xác.

Nếu hàm fopen() không thể mở file chính xác, nó sẽ trả về giá trị NULL. Chúng ta có thể sử dụng điều này để phán đoán khi nào mở file thất bại và xử lý lỗi khi cần.

Giả sử bạn đã mở một file sample.txt và gán vào con trỏ file fp như sau:

fp = fopen("sample.txt", "r");

Khi đó, nếu mở file thất bại thì giá trị của fp sẽ trở thành NULL, và chúng ta có thể viết xử lý tránh lỗi như sau:

if (fp == NULL) {

//Nội dung xử lý khi lỗi mở file xảy ra

}

Về nội dung xử lý lỗi, thông thường chúng ta sẽ xuất một dòng thông báo “Đã xảy ra lỗi”, và sau đó buộc chương trình phải kết thúc.

Và để buộc chương trình phải kết thúc, chúng ta thường sử dụng tới hàm exit() trong header file cstdlib nhằm cưỡng chế kết thúc chương trình. Để sử dụng hàm này thì chúng ta cần include header file vào đầu chương trình, và gọi hàm tại bất cứ vị trí nào muốn kết thúc chương trình với cú pháp sau đây:

exit(status);

Trong đó status là trạng thái thoát chương trình theo cách bình thường hay bất thường. Đối số status có thể chỉ định bằng một trong hai giá trị là 0 và 1, tương ứng với việc kết thúc bình thường và kết thúc bất thường.

Ngoài ra, chúng ta cũng có thể chỉ định giá trị của status là EXIT\_SUCCESS tương ứng với 0, và EXIT\_FAILURE tương ứng với 1.

Ví dụ, các cách viết sau đều OK khi sử dụng hàm exit() để kết thúc chương trình C++.

// Kết thúc chương trình bình thường

exit(0);

exit(EXIT\_SUCCESS);

// Kết thúc chương trình bất thường

exit(1);

exit(EXIT\_FAILURE);

Lưu ý nếu bạn mở file bên trong hàm main(), thì thay vì dùng hàm exit() thì bạn cũng có thể dùng lệnh return như sau để xử lý lỗi khi mở file trong C++:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname <<" file not open!\n";

    return -1;

}

Ứng dụng các kiến thức trên, chúng ta có thể viết xử lý lỗi khi mở file trong C++ như sau:

#include <iostream>

#include < cstdlib>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    fp = fopen("sample.txt", "r");

    if (fp == NULL) {

        cout << "Error opening file!"<<endl;

        exit(0);

    }

    fp2 = fopen("sample2.txt", "r");

    if (fp2 == NULL) {

        cout << "Error opening file!"<<endl;

        exit(1);

    }

}

## Mở và đóng file

### Mở file trong C++ bằng hàm fopen

Có 2 loại hàm fopen trong C++ là hàm fopen thông thường và hàm fopen\_s cao cấp. Hàm fopen\_s là một phiên bản an toàn và bảo mật hơn của hàm fopen, và chúng ta sẽ tìm hiểu về nó ở phần sau.

Chúng ta sử dụng hàm fopen thông thường trong C++ với cú pháp sau đây:

fp = fopen(filepath, mode);

Trong đó:

* fp là con trỏ file dùng để gán kiểu cấu trúc FILE được trả về từ hàm nếu mở file thành công
* filepath là đường dẫn tới file cần mở. Đường dẫn này có thể là đường dẫn tương đối, hoặc là đường dẫn tuyệt đối.
* mode là chế độ mở file.

Hàm fopen sẽ trả về một thực thể được tạo ra từ kiểu cấu trúc FILE chứa các thông tin của file đã mở để chúng ta có thể xử lý file. Chúng ta cũng gọi thực thể này là file handle - một đối tượng để xủ lý file trong C++. Nếu mở file thất bại, một giá trị NULL sẽ được trả về.

VD:

#include <iostream>

using namespace std;

int main()

{

    //Khai báo con trỏ file

    FILE \* fp = NULL;

    //Mở file và gán file handle vào con trỏ file

    fp = fopen("sample.txt", "r");

    return 0;

}

Các mode để mở file trong C++

Các chế độ có thể sử dụng để mở file bằng hàm fopen trong C++ như sau:

| **Mode** | **Xử lý** | **Chức năng** |
| --- | --- | --- |
| r | Mở để đọc | Chỉ cho phép đọc file Nếu file không tồn tại thì trả về NULL |
| w | Mở để ghi đè | Xoá nội dung cũ và ghi đè nội dung mới Nếu file không tồn tại thì tạo file mới |
| a | Mở để ghi chèn | Ghi chèn nội dung mới vào cuối file Nếu file không tồn tại thì tạo file mới |
| r+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì trả về NULL |
| w+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì tạo file mới |
| a+ | Mở để đọc và ghi chèn | Cho phép cả đọc và ghi chèn Nếu file không tồn tại thì tạo file mới |

### Mở file binary file trong C++

Lại nữa, bằng cách thêm ký tự b vào đằng trước tên mode thì chúng ta cũng có thể mở các binary file - tập tin nhị phân trong C++. Khác với các tập tin văn bản mà chúng ta có thể đọc được các chữ cái, văn bản ghi trong đó, thì Binary File là một tập tin chứa nội dung là các chuỗi nhị phân như sau:

binary file



Các mode dùng để mở binary file trong C++ sẽ là:

| **Mode** | **Xử lý** | **Chức năng** |
| --- | --- | --- |
| rb | Đọc | Chế độ đọc file binary |
| wb | Ghi đè | Chế độ ghi đè file binary |
| ab | Ghi chèn | Chế độ ghi chèn file binary |

### Mở file trong C++ bằng hàm fopen\_s

Hàm fopen\_s trong C++ là một phiên bản an toàn và bảo mật hơn của hàm fopen, và hàm này cũng giúp chúng ta mở file trong C++.

Chúng ta sử dụng hàm fopen\_s trong C++ với cú pháp sau đây:

errno\_t err = fopen\_s(fp, filepath, mode);

Trong đó:

* errno\_t là kiểu lỗi trong C++, và err là tên biến để gán giá trị lỗi nếu mở file thất bại
* fp là con trỏ file dùng để gán kiểu cấu trúc FILE được trả về từ hàm nếu mở file thành công
* filepath là đường dẫn tới file cần mở. Đường dẫn này có thể là đường dẫn tương đối, hoặc là đường dẫn tuyệt đối.
* mode là chế độ mở file. Các mode có thể dùng tương tự như với hàm fopen mà chúng ta đã học ở trên.

Hàm fopen\_s sẽ trả về một số tự nhiên biểu thị số hiệu của lỗi khi mở file. Nếu số này bằng 0 thì việc mở file thành công, và nếu số này khác 0 thì chúng ta có thể tìm ra nội dung lỗi bằng cách đối chiếu số hiệu của lỗi với bảng lỗi.

Trong trường hợp không cần xác định tới kiểu lỗi nếu có khi mở file, chúng ta cũng có thể lược bỏ đi việc gán errno\_t err mà dùng hàm trực tiếp với cú pháp sau đây:

fopen\_s(fp, filepath, mode);

Sau khi mở file thành công, thực thể được tạo ra từ kiểu cấu trúc FILE chứa các thông tin của file đã mở sẽ được gán vào con trỏ fp đã khai báo. Và chúng ta có thể thực hiện các thao tác với file thông qua con trỏ file này như bình thường.

#include <iostream>

using namespace std;

int main(){

    //Mở file bằng hàm fopen\_s mà không kiểm tra lỗi

    FILE \* fp1 = NULL;

    fopen\_s(&fp1, "sample.txt", "r");

    fclose(fp1);

    //Mở file bằng hàm fopen\_s có kiểm tra lỗi

    FILE \* fp2 = NULL;

    errno\_t no2;

    no2 = fopen\_s(&fp2, "memo.txt", "w");

    if (no2 != 0) cout << "Mo file thanh cong";

    fclose(fp2);

}

### Sự khác biệt giữa hàm fopen và fopen\_s

Hàm fopen\_s là một phiên bản an toàn và bảo mật hơn của hàm fopen. Do đó, hàm này cấm mở một file cùng lúc khi đang mở file đó ở chế độ ghi.

Bởi vậy, nếu chúng ta đã mở một file bằng hàm fopen\_s rồi thì chương trình tiếp sẽ kết thúc bất thường vào lần gọi thứ hai của hàm fopen\_s.

Sự khác biệt lớn nhất giữa giữa hàm fopen và fopen\_s cũng chính là điều này. Chúng ta có thể mở một file cùng lúc khi đang mở file đó ở chế độ ghi bằng hàm fopen, nhưng sẽ xảy ra lỗi nếu chúng ta làm việc này với hàm fopen\_s.

Ví dụ, do hàm fopen\_s được gọi 2 lần với cùng môt file, nên chương trình sẽ tự động kết thúc như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp1 = NULL;

    FILE \* fp2 = NULL;

    errno\_t no1;

    errno\_t no2;

    //Lần mở file đầu tiên với fopen\_s

    no1 = fopen\_s(&fp1, "memo.txt", "w");

    if (no1 != 0)

    {

        cout << "ERR("<<no1<<"): mo lan 1"<<endl;

    }

    //Lần mở cùng một file thứ 2 với fopen\_s

    no2 = fopen\_s(&fp2, "memo.txt", "w");

    if (no2 != 0)

    {

        cout << "ERR("<<no2<<"): mo lan 2"<<endl;

    }

    return 0;

}

RR(13): mo lan 2

### Xử lý lỗi khi mở file trong C++

Như đã phân tích trong bài Xử lý file trong C++ thì chúng ta cần phải xử lý lỗi khi mở một file thất bại trong C++, để chương trình tự xử lý lỗi, và chạy chương trình một cách chính xác.

VD:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname<<" file not open!\n";

    exit(0);

}

Lưu ý nếu bạn mở file bên trong hàm main(), thì bằng cách dùng lệnh return như sau thì chúng ta cũng có thể xử lý lỗi khi mở file trong C++:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname<<" file not open!\n";

    return -1;

}

### Tạo file mới trong C++

Bằng cách sử dụng một trong 2 hàm mở file trên, chúng ta có thể tạo một file mới và mở nó ra để xử lý. Ví dụ chúng ta sẽ tạo một file mới có tên memo2.txt bằng hàm fopen() với mode w như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL; //Khai báo con trỏ file

    fp = fopen("memo2.txt", "w"); // Mở file và gán file handle vào con trỏ file

    return 0;

}

Bạn cũng có thể sử dụng mode a+ để tạo file mới trong C++:

#include <iostream>

using namespace std;

int main()

{

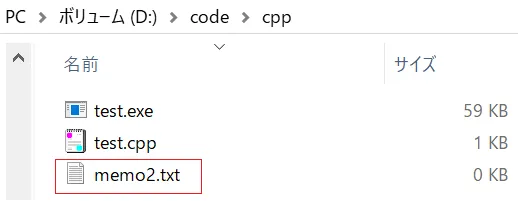
    FILE \* fp = NULL; //Khai báo con trỏ file

    fp = fopen("memo2.txt", "a+"); // Mở file và gán file handle vào con trỏ file

    return 0;

}

Sau khi compile và chạy chương trình, một file trống memo2.txt với kích thước 0 KB sẽ được tạo ra như dưới đây:



### Đóng file trong C++ bằng hàm fclose

Sau khi đã thao tác xong với file trong C++, chúng ta cần đóng file đó lại để kết thúc phiên làm việc với file và giải phóng bộ nhớ trong chương trình.

Và để đóng file trong C++ chúng ta sẽ dùng tới hàm fclose.

Hàm fclose là một hàm có sẵn trong thư viện chuẩn, có tác dụng đóng một file đã mở từ trước đó. Tên hàm fclose được viết tắt bởi cụm từ file và close, được dịch theo tiếng Việt chính xác là hàm đóng file.

Chúng ta sử dụng hàm fclose trong C++ với cú pháp sau đây:

int fclose(FILE \* fp);

Trong đó fp là con trỏ của file cần đóng.

Hàm fclose sẽ trả về 0 trong trường hợp kết thúc file bình thường. Và một giá trị kiểu EOF được trả lại trong trường hợp kết thúc file bất thường.

Ví dụ cụ thể, chúng ta đóng một file sau khi đã mở và hoàn thành các xử lý với nó như sau:

#include <iostream>

using namespace std;

int main(){

    //Mở file

    FILE \* fp = NULL;

    fp = fopen("memo.txt", "w");

    //Đóng file

    fclose(fp);

    return 0;

}

## Đọc file trong C++ (fgetc, fgets, fscanf, sscanf)

Ở đây, mode đọc file chính là thông tin về những việc cần làm với một file. Các mode có thể dùng để đọc file trong C++ như sau:

| **Mode** | **Xử lý** | **Chức năng** |
| --- | --- | --- |
| r | Mở để đọc | Chỉ cho phép đọc file Nếu file không tồn tại thì trả về NULL |
| r+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì trả về NULL |
| w+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì tạo file mới |
| a+ | Mở để đọc và ghi chèn | Cho phép cả đọc và ghi chèn Nếu file không tồn tại thì tạo file mới |

### Đọc từng ký tự trong file bằng hàm fgetc

Cú pháp:

int fgetc(FILE \* fp);

Trong đó fp là con trỏ của file cần đọc, được tạo ra từ việc mở file ở phần trên.

Hàm fgetc sẽ trả về mã ASCII của 1 ký tự được đọc ra từ file. Trong trường hợp vị trí đọc ký tự đã là cuối file, hoặc là việc đọc file thất bại thì giá trị EOF sẽ được trả về.

Ví dụ cụ thể, chúng ta có file sample.txt với nội dung sau đây:

Hello

World

Chúng ta sẽ dùng hàm fopen để mở file, sau đó đọc 1 ký tự trong file bằng hàm fgetc như sau. Lưu ý là để chuyển kết quả hàm fgetc là một mã ASCII sang kiểu ký tự thì chúng ta sử dụng thêm hàm char() bên trong chương trình.

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    //Mở file bằn hàm fopen

    fp= fopen("sample.txt", "r");

    cout << char(fgetc(fp)) <<endl;

    cout << char(fgetc(fp)) <<endl;

}

H

e

Bạn có thể thấy chúng ta chỉ có thể đọc từng ký tự từ file trong mỗi lần chạy hàm fgetc mà thôi.

Để có thể đọc tất cả các ký tự từ trong file, chúng ta sẽ cần tạo ra một vòng lặp để đọc từng ký tự từ đầu file cho tới cuối file, cho tới khi kết quả trả về là giá trị EOF như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    //Mở file bằn hàm fopen

    fp= fopen("sample.txt", "r");

    char a;

    //Đọc từng ký tự từ file cho tới khi gặp EOF

    while ((a = fgetc(fp)) != EOF)

    {

        //cout << fgetc(fp)<<endl;

        //Xuất từng ký tự ra màn hình

        cout << a;

    }

    fclose(fp);

    return 0;

}

Hello

World

### Đọc từng dòng file trong C++ bằng hàm fgets

Hàm fgets trong C++ là một hàm có sẵn trong thư viện chuẩn, có tác dụng đọc từng dòng trong file chỉ định. Tên hàm fgets được viết tắt bởi cụm từ file, get và string, được dịch theo tiếng Việt chính xác là hàm đọc từng dòng trong file.

Cú pháp:

char \* fgets(char \* buf, int size, FILE \* fp);

Trong đó:

* fp là con trỏ của file cần đọc, được tạo ra từ việc mở file
* buf là con trỏ tới nơi lưu trữ chuỗi đã đọc từ dòng trong file. Thông thường chúng ta chỉ định buf bằng một mảng.
* size là kích thước (số ký tự) lớn nhất có thể đọc từ dòng trong file.

Hàm fgets sẽ trả về con trỏ lưu địa chỉ trên bộ nhớ của chuỗi được đọc từ dòng trong file. Trong trường hợp vị trí đọc ký tự đã là cuối file, hoặc là việc đọc file thất bại thì con trỏ NULL sẽ được trả về.

Lại nữa, hàm fgets sẽ đọc một dòng từ file, và dòng đó được tính từ đầu dòng cho tới khi gặp ký tự xuống dòng \n.

Ví dụ cụ thể, chúng ta có file nums.txt với nội dung sau đây:

1234567

89abc

def

Chúng ta sẽ dùng hàm fopen để mở file, sau đó đọc từng dòng trong file bằng hàm fgets như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char arr[128];

    //Mở file bằn hàm fopen

    fp= fopen("nums.txt", "r");

    //Đọc dòng 1

    fgets(arr, 128, fp);

    cout <<  arr;

    //Đọc dòng 2

    fgets(arr, 128, fp);

    cout <<  arr;

    return 0;

}

1234567

89abc

Bạn có thể thấy chúng ta chỉ có thể đọc từng dòng từ file trong mỗi lần chạy hàm fgetc mà thôi.

Để có thể đọc tất cả các dòng từ trong file, chúng ta sẽ cần tạo ra một vòng lặp để đọc từng dòng từ đầu file cho tới cuối file, cho tới khi kết quả trả về là giá trị NULL như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char arr[128];

    //Mở file bằn hàm fopen

    fp= fopen("nums.txt", "r");

    //Đọc từng dòng từ file cho tới khi gặp NULL

    while (fgets(arr, 128, fp) != NULL)

    {

        //Xuất từng dòng ra màn hình

        cout <<  arr;

    }

    fclose(fp);

    return 0;

}

1234567

89abc

def

### Đọc từng dòng file khi đối số size nhỏ hơn số ký tự

Đối số size trong hàm fgets giúp chúng ta chỉ định số ký tự lớn nhất có thể được đọc từ một dòng trong file. Thông thường chúng ta chỉ định giá trị của size bằng với số phần tử của mảng chuẩn bị để lưu nội dung đọc từ dòng đó.

Lưu ý trong trường hợp giá trị của size nhỏ hơn số ký tự thực có trong dòng, thì chỉ có size -1 ký tự thực được đọc ra từ dòng đó mà thôi. Chỗ trống của ký tự còn lại sẽ được tự động lấp chỗ bằng một ký tự kết thúc chuỗi \0.

Ví dụ, chúng ta sẽ đọc lại file nums.txt ở trên, nhưng với chỉ định số phần tử lớn nhất có thể đọc lại nhỏ hơn số ký tự thực trong dòng như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char arr[128];

    //Mở file bằng hàm fopen

    fp= fopen("nums.txt", "r");

    //Đọc dòng 1

    fgets(arr, 3, fp);

     cout <<  arr <<endl;

    //Đọc dòng 2

    fgets(arr, 4, fp);

    cout <<  arr <<endl;

    return 0;

}

12

345

Vì sao lại chỉ đọc được 2 ký tự ở lần một từ dòng, và tại sao ở lần hai lại bắt đầu đọc dòng từ ký tự 3 chứ?

Câu trả lời là ở lần đọc 1 chúng ta chỉ định đọc 3 ký tự từ dòng, tuy nhiên chỉ có 2 ký tự thực là 12 được đọc, chỗ trống còn lại được dành để điền ký tự kết thúc chuỗi \0 rồi.

Và ở lần đọc thứ 2, hàm fgets cũng sẽ bắt đầu đọc từ cùng một dòng ban đầu do dòng này chưa được đọc hết ký tự. Và vị trí bắt đầu được đọc chính là từ ký tự kết thúc chuỗi \0 mới được điền ở lần đọc đầu tiên.

### Đọc từng dòng file theo định dạng chỉ định bằng hàm fscanf

Hàm fscanf trong C++ là một hàm có sẵn trong thư viện chuẩn, có tác dụng đọc từng dòng file theo định dạng chỉ định. Hàm fscanf không đọc toàn bộ nội dung của dòng, mà sẽ phân tách nội dung dòng đó theo từng định dạng chỉ định, qua đó có thể truy xuất các thông tin cần thiết mà người dùng muốn lấy ra từ dòng đó, ví dụ như là chỉ lấy các số trong dòng, hoặc là chỉ lấy các chuỗi trong dòng được viết theo một quy luật (định dạng) cụ thể.

Chúng ta sử dụng hàm fscanf trong C++ với cú pháp sau đây:

int fscanf(FILE \* fp, "fo1 fo2 fo3", add1, add2, add3);

Trong đó

* fp là con trỏ của file cần đọc, được tạo ra từ việc mở file ở phần trên.
* Các cặp fo và add tương ứng là định dạng (format) của dữ liệu cần đọc từ dòng, và địa chỉ của tên biến (con trỏ biến) dùng để lưu dữ liệu đó trong bộ nhớ.

Hàm fscanf sẽ trả về một số thuộc kiểu int, chính là số mục có thể đọc được từ dòng. Trong trường hợp dòng được đọc đã là dòng cuối file, hoặc là việc đọc file thất bại thì giá trị EOF sẽ được trả về.

Lại nữa, định dạng (format) của dữ liệu cần đọc cũng như kiểu của biến lưu dữ liệu được tóm tắt trong bảng dưới đây:

| **Định dạng chuyển đổi** | **Kiểu biến** | **Chi tiết** |
| --- | --- | --- |
| %hhd | char unsigned char | Chuyển về dạng thập phân và lưu trữ trong biến 1 byte |
| %hd | short unsigned short | Chuyển về dạng thập phân và lưu trữ trong biến 2 byte |
| %d | int unsigned int | Chuyển về dạng thập phân và lưu trữ trong biến kiểu int |
| %ld | long unsigned long | Chuyển về dạng thập phân và lưu trữ trong biến 4 byte |
| %hhx | char unsigned char | Chuyển về hệ thập lục phân và lưu trữ trong biến 1 byte |
| %hx | short unsigned short | Chuyển về hệ thập lục phân và lưu trữ trong biến 2 byte |
| %x | int unsigned int | Chuyển về hệ thập lục phân và lưu trữ trong biến kiểu int |
| %lx | long unsigned long | Chuyển về hệ thập lục phân và lưu trữ trong biến 4 byte |
| %f | float | Chuyển về số thực dấu phẩy động và lưu trữ trong biến kiểu float |
| %lf | double | Chuyển về số thực dấu phẩy động và lưu trữ trong biến kiểu double |
| %c++ | char | Chuyển về 1 ký tự và lưu trữ trong biến kiểu char |
| %s | char \* | Chuyển về chuỗi ký tự và lưu trữ trong biến kiểu mảng char |
| %p | void \* | Chuyển về địa chỉ và lưu trữ trong con trỏ |

#### Đọc số từ file trong C++

Ví dụ, chúng ta có file user.txt với nội dung sau đây:

Kiyoshi 32 172.5cm A

Honda 24 185.3cm O

Suzuki 63 153.8cm B

VD:

#include <iostream>

using namespace std;

int main(){

    FILE \* fp = NULL;

    char    name[32] = { 0 };    // Tên

    int        age = 0;            // Tuổi

    double    height = 0;            // Chiều cao

    char    blood = 0;            // Nhóm máu

    //Mở file bằng hàm fopen

    fp= fopen("user.txt", "r");

    //Đọc từng dòng trong file cho tới khi gặp EOF

    while (fscanf(fp, "%s %d %lfcm %c", name, &age, &height, &blood) != EOF)

    {

        //Xuất các dữ liệu số cần đọc

        cout <<  age <<" "<< height <<endl;

    }

    return 0;

}

32 172.5

24 185.3

63 153.8

Lưu ý là khi đọc từng dòng trong file bằng hàm fscanf thì chúng ta cần đọc tất cả các mục có thể đọc ra, tuy nhiên khi cần xuất dữ liệu số thì chúng ta chỉ cần chỉ định các dữ liệu đó mà thôi.

#### Đọc chuỗi từ file trong C++

Ví dụ, chúng ta cũng đọc file user.txt với nội dung sau đây:

Kiyoshi 32 172.5cm A

Honda 24 185.3cm O

Suzuki 63 153.8cm B

Giả sử chúng ta chỉ muốn đọc dữ liệu chuỗi bao gồm cột tên và nhóm máu trong từng dòng file trên, khi đó chúng ta dùng hàm fscanf như sau:

#include <iostream>

using namespace std;

int main(){

    FILE \* fp = NULL;

    char    name[32] = { 0 };    // Tên

    int        age = 0;            // Tuổi

    double    height = 0;            // Chiều cao

    char    blood = 0;            // Nhóm máu

    //Mở file bằng hàm fopen

    fp= fopen("user.txt", "r");

    //Đọc từng dòng trong file cho tới khi gặp EOF

    while (fscanf(fp, "%s %d %lfcm %c", name, &age, &height, &blood) != EOF)

    {

        //Xuất các dữ liệu số cần đọc

        cout <<  name <<" "<< blood <<endl;

    }

    return 0;

}

Kiyoshi A

Honda O

Suzuki B

Lưu ý là khi đọc từng dòng trong file bằng hàm fscanf thì chúng ta cần đọc tất cả các mục có thể đọc ra, tuy nhiên khi cần xuất dữ liệu chuỗi thì chúng ta chỉ cần chỉ định các dữ liệu đó mà thôi.

### Đọc từng dòng file theo định dạng chỉ định bằng hàm sscanf

Hàm sscanf trong C++ là một hàm có sẵn trong thư viện chuẩn, có tác dụng truy xuất thông tin theo định dạng chỉ định từ một chuỗi ký tự. Bằng cách ứng dụng hàm sscanf, chúng ta có thể lấy ra các thông tin cần thiết từ chuỗi theo một định dạng chỉ định nào đó.

Cú pháp sử dụng hàm sscanf trong C++ như sau:

int sscanf (buff, "fo1 fo2 fo3", add1, add2, add3 );

Trong đó

* buff là con trỏ tới chuỗi ký tự cần phân tích để lấy ra thông tin theo định dạng
* Các cặp fo và add tương ứng là định dạng (format) của dữ liệu cần đọc từ chuỗi ký tự, và địa chỉ của tên biến dùng để lưu thông tin được tách ra từ chuỗi trên bộ nhớ.

Và định dạng (format) sử dụng trong hàm sscanf thì cũng tương tự như với hàm fscanf mà Kiyoshi đã giới thiệu ở trên.

Cách sử dụng của sscanf rất giống với hàm fscanf ở trên, ngoại trừ việc nó sẽ xử lý nội dung dòng được đọc bởi hàm fgets mà thôi.

Ví dụ cụ thể, chúng ta có file test.txt với nội dung sau đây:

test01 1.0 1.1 1.2 1.3 1.4

test02 2.0 2.1 2.2 2.3 2.4

test03 3.0 3.1 3.2 3.3 3.4

VD:

#include <iostream>

using namespace std;

#define N 256 //Chỉ định số ký tự lớn nhất có thể đọc từ một dòng

int main() {

    FILE \*fp;

    char fname[] = "test.txt";

    char line[N];

    char str[16];

    float f1, f2, f3, f4, f5;

    //Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

    fp = fopen(fname, "r");

    if(fp == NULL) {

        cout << fname<<" file not open!\n";

        return -1;

    }

    //Đọc từng dòng trong file bằng hàm fgets

    while(fgets(line, N, fp) != NULL) {

        //Truy xuất thông tin cần thiết từ nội dung đọc được bằng hàm sscanf

        sscanf(line, "%s %f %f %f %f %f", str, &f1, &f2, &f3, &f4, &f5);

        cout << str <<" " << f1<<" "<< f2<<" "<< f3<<" "<< f4<<" "<< f5<< endl;

    }

    fclose(fp); //Đóng file

    return 0;

}

test01 1 1.1 1.2 1.3 1.4

test02 2 2.1 2.2 2.3 2.4

test03 3 3.1 3.2 3.3 3.4

### Đọc file csv trong C++

File csv thường có định dạng cố định, trong đó giữa các dữ liệu trong một hàng có thể được phân cách bởi một dấu phẩy, hoặc là bằng dấu cách. Và để đọc các file có định dạng cố định như thế này thì chúng ta có thể dùng hàm fscanf, hoặc là tổ hợp hàm fgets và hàm sscanf mà Kiyoshi đã giới thiệu ở trên.

Sau đây, hãy cùng tìm hiểu cách dùng hàm fscanf để đọc file csv trong C++ nhé.

#### Mở file CSV có các cột phân tách bởi dấu cách

Giả sử chúng ta có file test.csv với nội dung sau đây:

test01 1.0 1.1 1.2 1.3 1.4

test02 2.0 2.1 2.2 2.3 2.4

test03 3.0 3.1 3.2 3.3 3.4

Chúng ta sẽ mở file này và đọc từng dòng trong file theo định dạng như sau:

#include <iostream>

using namespace std;

int main() {

    FILE \*fp;

    char fname[] = "test.csv";

    char str[16];

    float f1, f2, f3, f4, f5;

    fp = fopen(fname, "r"); //Mở file bằng hàm fopen

    //Xử lý lỗi khi mở file thất bại

    if(fp == NULL) {

        cout << fname<<" file not open!\n";

        return -1;

    }

    //Đọc từng dòng theo định dạng bằng hàm fscan

    while(fscanf(fp, "%s %f %f %f %f %f", str, &f1, &f2, &f3, &f4, &f5) != EOF) {

        cout << str <<" " << f1<<" "<< f2<<" "<< f3<<" "<< f4<<" "<< f5<< endl;

    }

    fclose(fp); //Đóng file

    return 0;

}

test01 1.0 1.1 1.2 1.3 1.4

test02 2.0 2.1 2.2 2.3 2.4

test03 3.0 3.1 3.2 3.3 3.4

Có thể thấy bằng cách thêm dấu cách bên trong định dạng dữ liệu mở file mà chúng ta đã có thể đọc các dữ liệu cách nhau bởi dấu cách như thể từ từng dòng trong file CSV.

#### Mờ file CSV có các cột phân tách bởi dấu phẩy

Giả sử chúng ta có file test.csv với nội dung sau đây:

test01,1.0,1.1,1.2,1.3,1.4

test02,2.0,2.1,2.2,2.3,2.4

test03,3.0,3.1,3.2,3.3,3.4

Một cách tương tự thì chúng ta cũng mở file này và đọc từng dòng trong file theo định dạng như sau:

#include <iostream>

using namespace std;

int main() {

    FILE \*fp;

    char fname[] = "test.csv";

    char str[16];

    float f1, f2, f3, f4, f5;

    fp = fopen(fname, "r"); //Mở file bằng hàm fopen

    //Xử lý lỗi khi mở file thất bại

    if(fp == NULL) {

        cout << fname<<" file not open!\n";

        return -1;

    }

    //Đọc từng dòng theo định dạng bằng hàm fscan

    while(fscanf(fp, "%[^,],%f,%f,%f,%f,%f", str, &f1, &f2, &f3, &f4, &f5) != EOF) {

        cout << str <<" " << f1<<" "<< f2<<" "<< f3<<" "<< f4<<" "<< f5;

    }

    fclose(fp); //Đóng file

    return 0;

}

test01 1 1.1 1.2 1.3 1.4

test02 2 2.1 2.2 2.3 2.4

test03 3 3.1 3.2 3.3 3.4

#### Đọc mảng từ file trong C++

Ví dụ, chúng ta có file test.csv có nội dung sau đây:

test01,1.0,1.1,1.2,1.3,1.4

test02,2.0,2.1,2.2,2.3,2.4

test03,3.0,3.1,3.2,3.3,3.4

Có thể thấy với file CSV này thì các ô trong hàng được cách nhau bởi dấu phẩy, trong đó ô đầu tiên ở dạng chuỗi ký tự, và các ô sau thì ở dạng số thực.

VD:

#include <iostream>

using namespace std;

#define N 256 //Định nghĩa N là số ký tự lớn nhất có thể đọc từ một dòng

#define ROW 3 //Định nghĩa số dòng sẽ đọc trong file

// Khai báo cấu trúc mảng để lưu chuỗi và số trích xuất từ dòng

typedef struct str {

    char str[16];

    float f\_data[5];

} data;

int main() {

    FILE \*fp; //

    char fname[] = "test.csv";

    char line[N];

    char str[16];

    float f1, f2, f3, f4, f5;

    int i = 0;

    //Khởi tạo thực thể từ kiểu cấu trúc data

    data data[ROW];

    fp = fopen(fname, "r"); //Mở file bằng hàm fopen, trả về NULL khi thất bại

    if(fp == NULL) {

        cout << fname<<" file not open!\n";

        return -1;

    }

    //Đọc từng dòng trong file bằng hàm fgets

    while(fgets(line, N, fp) != NULL) {

        //Trích xuất thông tin từ nội dung file bằng hàm sscanf

        sscanf(line, "%[^,],%f,%f,%f,%f,%f", str, &f1, &f2, &f3, &f4, &f5);

        //Gán các nội dung trích xuất được lần lượt vào trong thực thể của cấu trúc

        for(int j = 0; j  < sizeof(data[i].str) / sizeof(data[i].str[0]); j++) {

            data[i].str[j] = str[j];

        }

        float tmp[] = {f1, f2, f3, f4, f5};

        for(int j = 0; j  < sizeof(data[i].f\_data) / sizeof(data[i].f\_data[0]); j++) {

            data[i].f\_data[j] = tmp[j];

        }

        //Xuất giá trị các thành viên trong thực thể của cấu trúc

        cout << data[i].str<<" "<< data[i].f\_data[0]<<" "<< data[i].f\_data[1]<<" "<<

        data[i].f\_data[2]<<" "<< data[i].f\_data[3]<<" "<< data[i].f\_data[4]<<endl;

        i++;

    }

    fclose(fp); //Đóng file

    return 0;

}

test01 1 1.1 1.2 1.3 1.4

test02 2 2.1 2.2 2.3 2.4

test03 3 3.1 3.2 3.3 3.4

Lưu ý trong định dạng dữ liệu sử dụng trong hàm sscanf, chúng ta đã dùng tới biểu thức chính quy với cách viết %[^,], có tác dụng lấy các dữ liệu chuỗi ngoại trừ dấu phẩy. Lý do là dấu phẩy được công nhận là một phần của chuỗi, do đó, chúng ta cần dùng biểu thức chính quy để loại bỏ dấu phẩy này từ trong kết quả đọc chuỗi.

## Ghi file trong C++ (fputc, fputs,fprintf)

### Ghi ký tự vào file bằng hàm fputc

Cú pháp:

int fputc(int char, FILE \* fp);

Trong đó:

* fp là con trỏ của file cần ghi, được tạo ra từ việc mở file ở phần trên.
* char là ký tự cần ghi vào file

Hàm fputc sẽ trả về ký tự đã được ghi vào file. Trong trường việc ghi file thất bại thì giá trị EOF sẽ được trả về.

Ví dụ cụ thể, chúng ta sẽ ghi dòng text Hello vào file trống sample.txt. Chúng ta sẽ dùng hàm fopen để mở file, sau đó ghi từng ký tự có trong dòng nội dung này vào file bằng hàm fputc như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    fopen\_s(&fp, "sample.txt", "w");

    fputc('H', fp);

    fputc('e', fp);

    fputc('l', fp);

    fputc('l', fp);

    fputc('o', fp);

    fputc('\n', fp);

    fclose(fp);

    return 0;

}

Hello

### Ghi chuỗi vào file trong C++ bằng hàm fputs

Cú pháp:

int fputs(const char \* str, FILE \* fp);

Trong đó:

* fp là con trỏ của file cần ghi, được tạo ra từ việc mở file
* str là con trỏ tới nơi lưu trữ chuỗi đã ghi từ dòng vào file. Thông thường chúng ta chỉ định str bằng một chuỗi.

Hàm fputs sẽ trả về 0 nếu việc ghi chuỗi vào file thành công. Trong trường hợp việc ghi file thất bại thì EOF sẽ được trả về.

Lưu ý là hàm fputs không tự động ghi ký tự xuống dòng \n sau khi đã ghi chuỗi, do đó khi muốn xuống dòng khi ghi chuỗi thì chúng ta phải tự điền ký tự \n vào vị trí cần xuống dòng trong chuỗi.

Ví dụ cụ thể, chúng ta sẽ ghi nội dung sau vào file trống Hello.txt.

Hello World!

I am Kiyoshi

Code:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    fopen\_s(&fp, "Hello.txt", "w");

    fputs("Hello World!\nI am Kiyoshi", fp);

    fclose(fp);

    return 0;

}

Hello World!

I am Kiyoshi

### Ghi dữ liệu vào file trong C++ theo định dạng bằng hàm fprintf

Cú pháp:

int fprintf(fp, "fo1 fo2 fo3", var1, var2, var3);

Trong đó

* fp là con trỏ của file cần ghi, được tạo ra từ việc mở file ở phần trên.
* Các cặp fo và var tương ứng là định dạng (format) ghi vào file của dữ liệu, và tên biến chứa dữ liệu đó.

Cơ chế hoạt động của hàm fprintf là lấy dữ liệu từ biến var, định dạng nó theo định dạng ghi trong fo để tạo ra một chuỗi. Sau đó thì ghi chuỗi vừa tạo này vào file mà con trỏ fp chỉ đến.

Hàm fprintf sẽ trả về một số thuộc kiểu int, chính là số ký tự có trong chuỗi được tạo. Trong trường hợp việc ghi file thất bại thì giá trị EOF sẽ được trả về.

Định dạng dữ liệu sử dụng trong hàm fprintf

| **Định dạng chuyển đổi** | **Kiểu biến** | **Chi tiết** |
| --- | --- | --- |
| %hhd | char unsigned char | Chuyển về dạng thập phân và lưu trữ trong biến 1 byte |
| %hd | short unsigned short | Chuyển về dạng thập phân và lưu trữ trong biến 2 byte |
| %d | int unsigned int | Chuyển về dạng thập phân và lưu trữ trong biến kiểu int |
| %ld | long unsigned long | Chuyển về dạng thập phân và lưu trữ trong biến 4 byte |
| %hhx | char unsigned char | Chuyển về hệ thập lục phân và lưu trữ trong biến 1 byte |
| %hx | short unsigned short | Chuyển về hệ thập lục phân và lưu trữ trong biến 2 byte |
| %x | int unsigned int | Chuyển về hệ thập lục phân và lưu trữ trong biến kiểu int |
| %lx | long unsigned long | Chuyển về hệ thập lục phân và lưu trữ trong biến 4 byte |
| %f | float | Chuyển về số thực dấu phẩy động và lưu trữ trong biến kiểu float |
| %lf | double | Chuyển về số thực dấu phẩy động và lưu trữ trong biến kiểu double |
| %c | char | Chuyển về 1 ký tự và lưu trữ trong biến kiểu char |
| %s | char \* | Chuyển về chuỗi ký tự và lưu trữ trong biến kiểu mảng char |
| %p | void \* | Chuyển về địa chỉ và lưu trữ trong con trỏ |

Ví dụ ghi dữ liệu vào file trong C++

Giả sử chúng ta có các kiểu thông tin như [chuỗi], [ký tự] và [số] được lưu vào các biến tương ứng. Bằng cách sử dụng hàm fprintf, chúng ta có thể định dạng các thông tin này và ghi vào file như chương trình sau đây:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char name[] = "Kiyoshi";

    char sex = 'M';

    int  old = 30;

    fopen\_s(&fp, "user.txt", "w");

    // Ghi dữ liệu theo định dạng chỉ định vào file

    fprintf(fp, "%s %c %d\n", name, sex, old);

    fprintf(fp, "%s,%c,%d\n", name, sex, old);

    fprintf(fp, "name: %s, sex: %c, old: %d\n", name, sex, old);

    fclose(fp);

    return 0;

}

Kiyoshi M 30

Kiyoshi,M,30

name: Kiyoshi, sex: M, old: 30

### Ứng dụng hàm fprintf, chúng ta có thể thực hiện ghi mảng vào file trong C++.

Do các phần tử trong mảng C++ đều có cùng kiểu dữ liệu, nên chúng ta có thể chỉ định cùng kiểu định dạng để ghi các phần tử từ mảng vào file bằng hàm hàm fprintf. Ví dụ cụ thể:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    int num[] = {10, 20, 44, 60, 82};

    fopen\_s(&fp, "num.txt", "w");

    // Ghi dữ liệu theo định dạng chỉ định vào file

    fprintf(fp, "%d %d %d %d %d", num[0], num[1], num[2], num[3], num[4]);

    fclose(fp);

    return 0;

}

10 20 44 60 82

Một cách tương tự thì chúng ta cũng có thể ghi các mảng 2 chiều vào file trong C++. Khi đó mỗi mảng con 1 chiều trong mảng 2 chiều sẽ được coi như một dòng để ghi vào file, và chúng ta cần phải thêm ký tự xuống dòng \n khi ghi từng mảng dòng vào file như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    int bangdiem[][3] = {

    {7, 9, 8} ,

    {8, 6, 7} ,

    {5, 7, 6} ,

    {4, 9, 5} ,

    {5, 8, 7} ,

    {6, 9, 3}

} ;

    //Lấy độ dài của mảng 2 chiều

    int y = sizeof(bangdiem) / sizeof(bangdiem[0]);

    int x = sizeof(bangdiem[0]) / sizeof(bangdiem[0][0]);

    fopen\_s(&fp, "bangdiem.txt", "w");

    // Ghi từng mảng 1 chiều trong mảng 2 chiều vào file

    for (int i = 0; i < y; i++){

        fprintf(fp, "Toan: %d Van: %d Anh: %d\n", bangdiem[i][0], bangdiem[i][1], bangdiem[i][2]);

    }

    fclose(fp);

    return 0;

}

Toan: 7 Van: 9 Anh: 8

Toan: 8 Van: 6 Anh: 7

Toan: 5 Van: 7 Anh: 6

Toan: 4 Van: 9 Anh: 5

Toan: 5 Van: 8 Anh: 7

Toan: 6 Van: 9 Anh: 3

### Ứng dụng hàm fprintf, chúng ta có thể thực hiện ghi cấu trúc vào file trong C++.

Do các thành viên trong cấu trúc C++ có thể có kiểu dữ liệu khác nhau, nên chúng ta cần chỉ định kiểu định dạng phù hợp cho từng thành viên để ghi giá trị của chúng từ mảng vào file bằng hàm hàm fprintf.

Ví dụ chúng ta ghi dữ liệu từ mảng cấu trúc vào file trong C++ bằng hàm fprintf như sau:

#include <iostream>

using namespace std;

#define PERSON\_NUM 5

//Khai báo kiểu struct

typedef struct {

    char name[20];

    char sex;

    int age;

    double height;

    double weight;

} person\_t;

int main()

{

    //Khởi tạo thực thể struct là một mảng struct

    person\_t p[PERSON\_NUM] = {{"Bob",      'M', 19, 165.4, 72.5},

                              {"Alice",    'F', 19, 161.7, 44.2},

                              {"Tom",      'M', 20, 175.2, 66.3},

                              {"Stefany",  'F', 18, 159.3, 48.5},

                              {"Leonardo", 'M', 19, 172.8, 67.2}};

    //Tạo con trỏ file

    FILE \* fp = NULL;

    //Mở file

    fopen\_s(&fp, "people.txt", "w");

    //Ghi header vào file

    fprintf(fp, "%s %s %s %s %s\n", "name", "sex", "age", "height", "weight");

    // Ghi từng thực thể của cấu trúc như một dòng vào file

    for (short i = 0; i < PERSON\_NUM; i++) {

           fprintf(fp, "%s %c %d %.2f %.2f \n",p[i].name, p[i].sex,p[i].age,p[i].height,p[i].weight);

    }

    fclose(fp);

    return 0;

}

name sex age height weight

Bob M 19 165.40 72.50

Alice F 19 161.70 44.20

Tom M 20 175.20 66.30

Stefany F 18 159.30 48.50

Leonardo M 19 172.80 67.20

### Ghi file CSV trong C++

Ở phần trên, chúng ta đã tiến hành ghi nội dung các kiểu dữ liệu phức tạp như mảng hay cấu trúc vào file dưới dạng file txt rồi. Tuy nhiên chúng ta cũng có thể ghi dữ liệu trên vào file CSV trong C++, đơn giản bằng cách thay đổi đuôi file từ .txt sang .csv là xong.

Ví dụ cụ thể, chúng ta ghi dữ liệu từ mảng cấu trúc vào file CSV trong C++ bằng hàm fprintf như sau. Lưu ý là chúng ta cần viết định dạng các dữ liệu cách nhau bởi dấu phẩy để có thể ghi từng dữ liệu vào từng ô trong file CSV.

#include <iostream>

using namespace std;

#define PERSON\_NUM 5

typedef struct {

    char name[20];

    char sex;

    int age;

    double height;

    double weight;

} person\_t;

int main()

{

    person\_t p[PERSON\_NUM] = {{"Bob",      'M', 19, 165.4, 72.5},

                              {"Alice",    'F', 19, 161.7, 44.2},

                              {"Tom",      'M', 20, 175.2, 66.3},

                              {"Stefany",  'F', 18, 159.3, 48.5},

                              {"Leonardo", 'M', 19, 172.8, 67.2}};

    FILE \* fp = NULL;

    int num[] = {10, 20, 44, 60, 82};

    fopen\_s(&fp, "people.csv", "w");

    // Ghi từng mảng 1 chiều trong mảng 2 chiều vào file

   fprintf(fp, "%s,%s,%s,%s,%s\n", "name", "sex", "age", "height", "weight");

   for (short i = 0; i < PERSON\_NUM; i++) {

           fprintf(fp, "%s,%c,%d,%.2f,%.2f\n",p[i].name, p[i].sex,p[i].age,p[i].height,p[i].weight);

    }

    fclose(fp);

    return 0;

}

## Thao tác với file trong C++ bằng phương pháp mới

### Phương pháp xử lý file mới trong C++

Ngoài việc sử dụng thư viện chuẩn nhập xuất file được kế thừa từ ngôn ngữ C thì trong C++ có thêm 3 class phục vụ cho việc nhập xuất file là ifstream ,ofstream và fstream.

Trong đó:

* ifstream dùng để nhập file trong C++
* ofstream dùng để xuất file trong C++
* fstream được gộp lại từ 2 class trên, dùng để nhập xuất file trong C++

Để dùng được 3 class này, chúng ta cần phải include các header file tương ứng có cùng tên là ifstream ,ofstream và fstream vào đầu chương trình.

### ofstream trong C++

ofstream là một class cung cấp chức năng của một luồng file đầu ra. ofstream được viết tắt từ các cụm từ out file stream trong tiếng Anh, dịch sang tiếng Việt có nghĩa là luồng file đầu ra.

Giống như trong C chúng ta mở file bằng cách tạo ra một cấu trúc FILE để chứa thông tin của file cần đọc, thì trong C++ chúng ta dùng class ofstream để tạo ra một thực thể stream (instance) chứa thông tin về file cần mở, và sau đó thao tác với file thông qua stream này.

Và stream tạo ra bởi ofstream sẽ có tác dụng ghi file trong C++.

Lưu ý là để sử dụng được fstream trong C++, chúng ta cần include header file fstream hoặc là ofstream vào đầu chương trình.

#### Mở file để ghi bằng ofstream

Để sử dụng ofstream, trước hết chúng ta cần thêm header file fstream vào chương trình, sau đó thì sử dụng cú pháp sau đây để mở file để ghi bằng ofstream trong C++,

std::ofstream ofs(filepath);

Trong đó filepath là đường dẫn của file cần mở và ofs là stream được tạo ra nếu mở file thành công.

Nếu mở file thành công bằng ofstream thì ofs sẽ được trả về. Tuy nhiên nếu mở file thất bại thì ofs sẽ không tồn tại.

VD:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("test.txt");

}

#### Xử lý lỗi khi mở file thất bại bằng ofstream

Nếu mở file thành công bằng ofstream thì ofs sẽ được trả về. Tuy nhiên nếu mở file thất bại thì ofs sẽ không tồn tại. Ứng dụng điều này, chúng ta có thể viết xử lý tránh lỗi như sau:

if (ofs) //Mở file thành công

{

    //Viết xử lý khi mở file thành công

}

if (!ofs) //Mở file thất bại

{

    //Viết xử lý khi mở file thất bại

}

#### Mở và đóng file trong C++ bằng ofstream

Ngoài việc thao tác trực tiếp với file bằng ofstream thì chúng ta cũng có thể sử dụng các hàm thành viên trong ofstream để mở và đóng file. Hai hàm được sử dụng là open() và close() tương ứng với việc mở và đóng file bằng ofstream.

Để mở file bằng hàm open trong ofstream, trước hết chúng ta cần khai báo một stream bằng ofstream, sau đó thì truyền giá trị của đường dẫn file vào hàm như sau:

std::ofstream ofs;

ofs.open(filepath);

Và để đóng file được mở bởi hàm open, chúng ta đơn giản viết như sau:

ofs.close();

VD:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    std::ofstream ofs; //Khai báo stream

    ofs.open(test.txt);//Mở file bằng stream

    ofs << "abc" << std::endl; //Ghi vào file

    ofs.close();//Đóng file

}

Mode mở file trong ofstream

Cũng giống như khi sử dụng hàm fopen() để mở file, thì trong ofstream cũng chuẩn bị sẵn các mode để mở file bằng hàm thành viên open() như sau:

| **mode** | **Cách dùng** |
| --- | --- |
| std::ios::out | Mở để ghi(chế độ mặc định) Khi viết, nội dung cũ bị xóa |
| std::ios::in | Mở để đọc |
| std::ios::app | Mở để ghi chèn |
| std::ios::ate | Sau khi mở thì di chuyển về vị trí cuối file |
| std::ios::trunc | Mở để ghi, tương tự mode out |
|  |  |
| std::ios::binary | Mở file nhị phân |

Do ofstream mở file để ghi, nên chúng ta có thể dùng bất kỳ mode nào ở trên khi muốn ghi file bằng ofsstream trong C++.

#### Ghi file trong C++ bằng ofstream

VD:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("test.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //ghi và in lần lượt các ký tự vào file

    ofs << "I am a big big girl!"<< endl;

    ofs << "in a big big World";

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, file test.txt được tạo ra với nội dung như sau:

I am a big big girl!

in a big big World

#### Lấy và thay đổi vị trí đọc file trong C++

Để lấy và thay đổi vị trí đang đọc và ghi file hiện tại, chúng ta sử dụng tới hàm thành viên seekp() trong class ofstream.

Dưới đây là các mode có thể sử dụng trong hàm seekp():

| **mode** | **Cách dùng** |
| --- | --- |
| std::ios::beg | Di chuyển tới đầu file |
| std::ios::cur | Lấy vị trí hiện tại trong file |
| std::ios::end | Di chuyển tới cuối file |

Để lấy vị trí hiện tại đang đọc hoặc ghi trong file, chúng ta cũng có thể sử dụng tới hàm thành viên tellp().

Sau đây, hãy cùng xem ví dụ lấy và thay đổi vị trí đọc và ghi file hiện tại trong ofstream bằng các hàm seekp và tellp.

Giả sử chúng ta có file hello.txt có nội dung sau đây:

Hello

Hãy lấy vị trí cuối file, sau đó thêm nội dung vào file và kiểm tra lại vị trí này như sau:

VD:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file và di chuyển tới cuối file

    std::ofstream ofs("hello.txt", std::ios::in | std::ios::ate);

    if (!ofs)

    {

        cout << "Cant open file" << std::endl;

        return 0;

    }

    //Hiển thị vị trí hiện tại trong file

    cout << "Vi tri hien tai: " << ofs.tellp() << std::endl;

    //Ghi chèn dòng abcde vào file

    ofs << "\nVietnam" << std::endl;

    cout << "Vi tri hien tai: " << ofs.tellp() << std::endl;

    //Dịch chuyển vị trí hiện tại về đầu file 2 ký tự

    ofs.seekp(-2, std::ios::cur);

    cout << "Vi tri hien tai: " << ofs.tellp() << std::endl;

    return 0;

}

Vi tri hien tai: 5

Vi tri hien tai: 16

Vi tri hien tai: 14

Và nội dung file cũng được ghi thêm như sau:

Hello

Vietnam

### ifstream trong C++

ifstream là một class cung cấp chức năng của một luồng file đầu vào. ifstream được viết tắt từ các cụm từ in, file và stream trong tiếng Anh, dịch sang tiếng Việt có nghĩa là luồng file đầu vào.

chúng ta dùng class ifstream để tạo ra một thực thể stream (instance) chứa thông tin về file cần mở, và sau đó thao tác với file thông qua stream này.

Lưu ý là sử dụng được fstream trong C++, chúng ta cần include header file fstream hoặc là ifstream vào đầu chương trình.

#### Mở file để đọc bằng ifstream

Để sử dụng ifstream, trước hết chúng ta cần thêm header file fstream vào chương trình, sau đó thì sử dụng cú pháp sau đây để mở file để đọc bằng ifstream trong C++,

std::ifstream ifs(filepath);

Trong đó filepath là đường dẫn của file cần mở và ifs là stream được tạo ra nếu mở file thành công.

Nếu mở file thành công bằng ifstream thì ifs sẽ được trả về. Tuy nhiên nếu mở file thất bại thì ifs sẽ không tồn tại.

VD:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ifstream

    ifstream ifs("test.txt");

}

#### Ghi file trong C++ bằng fstream

1234567

89abc

def

VD:

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

int main()

{

    string filename("sample.txt");

    // Khai báo vector để lưu các dòng đọc được

    vector<string> lines;

    string line;

    //Mở file bằng ifstream

    ifstream input\_file(filename);

    //Kiểm tra file đã mở thành công chưa

    if (!input\_file.is\_open()) {

        cerr << "Could not open the file - '"

             << filename << "'" << endl;

        return EXIT\_FAILURE;

    }

    //Đọc từng dòng trong

    while (getline(input\_file, line)){

        lines.push\_back(line);//Lưu từng dòng như một phần tử vào vector lines.

    }

    //Xuất từng dòng từ lines và in ra màn hình

    for (const auto &i : lines)

        cout << i << endl;

    //Đóng file

    input\_file.close();

    return 0;

}

Kết quả đọc file

1234567

89abc

def

### fstream

### Đọc file trong C++ bằng toán tử >> trong ifstream

Toán tử >> trong ifstream được sử dụng để gán từng nội dung đọc từ file vào một biến, hoặc một chuỗi chỉ định.

Cú pháp:

ifstream ifs >> variable;

Trong đó ifs là luồng chứa file được tạo ra khi mở file bằng ifstream, và variable là một biến kiểu string hoặc char để lưu dữ liệu đọc từ file ra.

Nếu chỉ định variable là một biến thuộc kiểu string thì chúng ta có thể đọc được từng từ trong file. Và nếu chỉ định variable là một biến thuộc kiểu char thì chúng ta có thể đọc được từng ký tự trong file.

Lưu ý, toán tử >> trong ifstream chỉ có thể đọc từng từ hoặc từng ký tự từ file trong C++, do đó chúng ta cần sử dụng toán tử này kết hợp với một vòng lặp để có thể đọc toàn bộ dữ liệu trong file.

#### Đọc từng ký tự trong file bằng toán tử >>

Chúng ta chỉ định variable là một biến thuộc kiểu char để đọc từng ký tự trong file bằng toán tử >>.

Ví dụ cụ thể, chúng ta có file sample.txt với nội dung sau đây:

123abc

Chúng ta sẽ đọc từng ký tự từ file này và xuất ra màn hình bằng cách sử dụng toán tử >> trong C++ như sau:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ifstream

    ifstream ifs("sample.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ifs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //Khai báo biến kiểu char để lưu ký tự đọc được

    char chr;

    //Đọc và in lần lượt các ký tự trong file

    while(ifs  >> chr){

        cout << chr << endl;

    }

    //Đóng file

    ifs.close();

    return 0;

}

1

2

3

a

b

c

#### Đọc từng từ trong file bằng toán tử >>

Chúng ta chỉ định variable là một biến thuộc kiểu string để đọc từng ký tự trong file bằng toán tử >>.

Ví dụ cụ thể, chúng ta có file sample.txt với nội dung sau đây:

I am a big big girl!

Chúng ta sẽ dùng ifstream để mở file, sau đó đọc từng ký tự từ file này và xuất ra màn hình bằng cách sử dụng toán tử >> trong C++ như sau:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ifstream

    ifstream ifs("sample.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ifs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //Khai báo biến kiểu string để lưu từng từ đọc được

    string str;

    //Đọc và in lần lượt các từ trong file

    while(ifs  >> str){

        cout << str << endl;

    }

    //Đóng file

    ifs.close();

    return 0;

}

I

am

a

big

big

girl!

### Đọc file trong C++ bằng bằng hàm getline

Hàm getline() trong C++ là một hàm có sẵn trong thư viện chuẩn, có tác dụng đọc từng dòng trong file chỉ định. Hàm getline đọc liên tục các ký tự từ file và lưu trữ nó trong một chuỗi cho đến khi nó tìm thấy ký tự phân cách. Chúng ta có thể chỉ định ký tự phân cách này, và giá trị mặc định của ký tự phân cách là ký tự xuống dòng \n.

Chúng ta sử dụng hàm getline trong C++ với cú pháp sau đây:

std::getline(input, str, delimiter );

Trong đó:

* input là luồng chứa file được tạo ra khi mở file bằng ifstream
* str là chuỗi để lưu kết quả đọc file
* delimiter là ký tự phân cách, và chúng ta có thể lược bỏ đối số này khi muốn sử dụng ký tự phân cách mặc định là ký tự xuống dòng \n.

Hàm getline sẽ trả về chuỗi đọc được, và trong trường hợp đọc file thất bại, giá trị 0 sẽ được trả về.

Do ký tự phân cách mặc định của hàm getline là ký tự xuống dòng \n nên chúng ta hay dùng hàm này để đọc từng dòng file trong C++. Tuy nhiên bằng cách chỉ định ký tự phân cách, chúng ta cũng có thể đọc file theo nhiều cách khác nhau.

Lưu ý, hàm getline chỉ có thể đọc từng dòng file trong C++, do đó chúng ta cần sử dụng hàm này kết hợp với một vòng lặp để có thể đọc toàn bộ các dòng trong file.

#### Đọc từng dòng file trong C++ bằng hàm getline

Khi lược bỏ đối số delimiter trong hàm getline, khi đó hàm sẽ sử dụng ký tự phân cách mặc định là ký tự xuống dòng \, và chúng ta có thể sử dụng hàm getline để đọc từng dòng file trong C++.

Ví dụ cụ thể, chúng ta có file sample.txt với nội dung sau đây:

1234567

89abc

def

Chúng ta sẽ dùng ifstream để mở file, sau đó đọc từng dòng trong file bằng hàm getline như sau. Lưu ý là chúng ta sử dụng vòng lặp while để có thể chạy số lần hàm getline tương ứng với số dòng cần đọc, và sử dụng vector để lưu lần lượt các dòng đọc được vào bộ nhớ.

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

int main()

{

    string filename("sample.txt");

    // Khai báo vector để lưu các dòng đọc được

    vector<string> lines;

    string line;

    //Mở file bằng ifstream

    ifstream input\_file(filename);

    //Kiểm tra file đã mở thành công chưa

    if (!input\_file.is\_open()) {

        cerr << "Could not open the file - '"

             << filename << "'" << endl;

        return EXIT\_FAILURE;

    }

    //Đọc từng dòng trong

    while (getline(input\_file, line)){

        lines.push\_back(line);//Lưu từng dòng như một phần tử vào vector lines.

    }

    //Xuất từng dòng từ lines và in ra màn hình

    for (const auto &i : lines)

        cout << i << endl;

    //Đóng file

    input\_file.close();

    return 0;

}

1234567

89abc

def

#### Đọc từng từ trong file bằng hàm getline

Khi chỉ định đối số delimiter trong hàm getline, khi đó hàm sẽ sử dụng ký tự phân cách chỉ định này, và chúng ta có thể sử dụng hàm getline để đọc từng từ trong file.

Ví dụ cụ thể, chúng ta có file sample.txt với nội dung sau đây:

I am a big big girl!

Các từ trong file này được phân cách bởi ký tự trống (dấu cách), do vậy nếu chỉ định đối số delimiter trong hàm getline bằng ký tự trống ' ', chúng ta có thể đọc từng từ trong file bằng hàm getline như sau:

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

int main()

{

    string filename("sample.txt");

    // Khai báo vector để lưu các dòng đọc được

    vector<string> lines;

    string line;

    //Mở file bằng ifstream

    ifstream input\_file(filename);

    //Kiểm tra file đã mở thành công chưa

    if (!input\_file.is\_open()) {

        cerr << "Could not open the file - '"

             << filename << "'" << endl;

        return EXIT\_FAILURE;

    }

    //Đọc từng từ trong file phân cách bởi ký tự trống

    while (getline(input\_file, line, ' ')){

        lines.push\_back(line);//Lưu từng dòng như một phần tử vào vector lines.

    }

    //Xuất từng dòng từ lines và in ra màn hình

    for (const auto &i : lines)

        cout << i << endl;

    //Đóng file

    input\_file.close();

    return 0;

}

I

am

a

big

big

girl!

### Đọc file csv trong C++

#### Đọc file CSV có các cột phân tách bởi dấu cách

Giả sử chúng ta có file test.csv với nội dung sau đây:

test01 1.0 1.1 1.2 1.3 1.4

test02 2.0 2.1 2.2 2.3 2.4

test03 3.0 3.1 3.2 3.3 3.4

Chúng ta sẽ mở file này và đọc từng ô trong file CSV theo định dạng như sau:

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

int main()

{

    string filename("test.csv");

    // Khai báo vector để lưu các dòng đọc được

    vector<string> lines;

    string line;

    //Mở file bằng ifstream

    ifstream input\_file(filename);

    //Kiểm tra file đã mở thành công chưa

    if (!input\_file.is\_open()) {

        cerr << "Could not open the file - '"

             << filename << "'" << endl;

        return EXIT\_FAILURE;

    }

    //Đọc từng dòng trong

    while (getline(input\_file, line, ' ')){

        lines.push\_back(line);//Lưu từng dòng như một phần tử vào vector lines.

    }

    //Xuất từng dòng từ lines và in ra màn hình

    for (const auto &i : lines)

        cout << i << endl;

    //Đóng file

    input\_file.close();

    return 0;

}

test01

1.0

1.1

1.2

test02

2.0

2.1

2.2

test03

3.0

3.1

3.2

#### Mờ file CSV có các cột phân tách bởi dấu phẩy

Giả sử chúng ta có file test.csv với nội dung sau đây:

test01,1.0,1.1,1.2,1.3,1.4

test02,2.0,2.1,2.2,2.3,2.4

test03,3.0,3.1,3.2,3.3,3.4

Một cách tương tự thì chúng ta cũng mở file này và đọc từng ô trong file CSV theo định dạng như sau:

#include <iostream>

#include <fstream>

#include <vector>

using namespace std;

int main()

{

    string filename("test.csv");

    // Khai báo vector để lưu các dòng đọc được

    vector<string> lines;

    string line;

    //Mở file bằng ifstream

    ifstream input\_file(filename);

    //Kiểm tra file đã mở thành công chưa

    if (!input\_file.is\_open()) {

        cerr << "Could not open the file - '"

             << filename << "'" << endl;

        return EXIT\_FAILURE;

    }

    //Đọc từng dòng trong

    while (getline(input\_file, line, ',')){

        lines.push\_back(line);//Lưu từng dòng như một phần tử vào vector lines.

    }

    //Xuất từng dòng từ lines và in ra màn hình

    for (const auto &i : lines)

        cout << i << endl;

    //Đóng file

    input\_file.close();

    return 0;

}

test01

1.0

1.1

1.2

test02

2.0

2.1

2.2

test03

3.0

3.1

3.2

### Ghi file trong C++ bằng toán tử << trong ofstream

Cú pháp:

ofs << data;

Trong đó ofs là luồng chứa file được tạo ra khi mở file bằng ofstream , và data là dữ liệu cần ghi vào file. Dữ liệu này có thể là một ký tự, hoặc một chuỗi trong C++.

Lưu ý, toán tử << trong ofstream chỉ có thể ghi từng ký tự hoặc từng chuỗi vào file trong C++, do đó chúng ta cần sử dụng toán tử này kết hợp với một vòng lặp để có thể ghi dữ liệu nhiều lần vào file.

#### Ghi từng ký tự vào file bằng toán tử <<

Ví dụ cụ thể, chúng ta muốn tạo file sample.txt và ghi vào file với nội dung sau đây:

123abc

Code:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("test.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //Ghi lần lượt các ký tự vào file

    ofs << 'H';

    ofs << 'e';

    ofs << 'l';

    ofs << 'l';

    ofs << 'o';

    ofs << endl;

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, file test.txt được tạo ra với nội dung như sau:

Hello

#### Ghi từng dòng vào file trong C++ bằng toán tử <<

Ví dụ cụ thể, chúng ta muốn tạo file sample.txt và ghi vào file với nội dung sau đây:

I am a big big girl!

in a big big World

code

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("test.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //ghi và in lần lượt các ký tự vào file

    ofs << "I am a big big girl!"<< endl;

    ofs << "in a big big World";

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, file test.txt được tạo ra với nội dung như sau:

I am a big big girl!

in a big big World

### Ghi file trong C++ bằng hàm write trong fstream

Chúng ta sử dụng hàm write trong C++ với cú pháp sau đây:

std::write (const char\* s, streamsize n);

Trong đó:

* s là con trỏ tới một chuỗi có ít nhất n ký tự.
* n là số lượng ký tự cần lấy từ chuỗi để ghi vào file

Hàm write sẽ trả về luồng, và trong trường hợp ghi file thất bại thì hàm sẽ trả về các flags nhằm thông báo lỗi.

Lưu ý, hàm write chỉ có thể ghi từng dòng file trong C++, do đó chúng ta cần sử dụng hàm này nhiều lần nếu muốn ghi nhiều dòng, và đừng quên ghi thêm cả ký tự xuống dòng ở cuối chuỗi để xuống dòng trong file.

#### Ghi chuỗi vào file trong C bằng hàm write

Chúng ta mở file cần ghi bằng ofstream, sau đó ghi từng dòng vào file trong C++ bằng hàm write như ví dụ sau đây:

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

    string text1("I am a big big girl!\n");

    string text2( "in a big big World");

    //Mở file bằng ofstream

    ofstream ofs("test2.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    //Ghi từng dòng vào file bằng hàm write

    ofs.write(text1.data(), text1.size());

    ofs.write(text2.data(), text2.size());

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, file test2.txt được tạo ra với nội dung như sau:

I am a big big girl!

in a big big World

### Ghi mảng vào file trong C++

Chúng ta sử dụng vòng lặp để lấy giá trị từng phần tử, sau đó ghi vào file bằng toán tử << như sau:

#include <iostream>

#include <fstream>

using namespace std;

//Tạo macro tìm số phần tử trong mảng

#define SIZE\_OF\_ARRAY(array) (sizeof(array)/sizeof(array[0]))

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("nums.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    int num[] = {10, 20, 44, 60, 82};

    //Ghi từng phần tử từ mảng vào file

    for (int i =0; i < SIZE\_OF\_ARRAY(num); i++)

        ofs << num[i] << ' ';

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, nội dung mảng sẽ được ghi vào file như sau:

10 20 44 60 82

Một cách tương tự thì chúng ta cũng có thể ghi các mảng 2 chiều vào file trong C++. Khi đó mỗi mảng con 1 chiều trong mảng 2 chiều sẽ được coi như một dòng để ghi vào file, và chúng ta cần phải thêm ký tự xuống dòng \n khi ghi từng mảng dòng vào file như sau:

#include <iostream>

#include <fstream>

using namespace std;

//Tạo macro tìm số phần tử trong mảng

#define SIZE\_OF\_ARRAY(array) (sizeof(array)/sizeof(array[0]))

int main()

{

    //Mở file bằng ofstream

    ofstream ofs("bangdiem.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    int bangdiem[][3] = {

    {7, 9, 8} ,

    {8, 6, 7} ,

    {5, 7, 6} ,

    {4, 9, 5} ,

    {5, 8, 7} ,

    {6, 9, 3}

} ;

     //Lấy độ dài của mảng 2 chiều

    int y = sizeof(bangdiem) / sizeof(bangdiem[0]);

    int x = sizeof(bangdiem[0]) / sizeof(bangdiem[0][0]);

    for (int i =0; i < y; i++){

        for (int j =0; j < x; j++){

            ofs << bangdiem[i][j] << ' ';

        }

        ofs << endl;

    }

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, từng dòng sẽ được ghi vào file như sau:

7 9 8

8 6 7

5 7 6

4 9 5

5 8 7

6 9 3

### Ghi cấu trúc vào file trong C++

Do các thành viên trong cấu trúc C++ có thể có kiểu dữ liệu khác nhau, nên chúng ta cần chỉ định kiểu định dạng phù hợp cho từng thành viên để ghi giá trị của chúng từ mảng vào file bằng hàm hàm fprintf.

Ví dụ chúng ta ghi dữ liệu từ mảng cấu trúc vào file trong C++ bằng hàm fprintf như sau:

#include <iostream>

#include <fstream>

using namespace std;

#define PERSON\_NUM 5

//Khai báo kiểu struct

typedef struct {

    char name[20];

    char sex;

    int age;

    double height;

    double weight;

} person\_t;

int main()

{

    //Khởi tạo thực thể struct là một mảng struct

    person\_t p[PERSON\_NUM] = {{"Bob",      'M', 19, 165.4, 72.5},

                              {"Alice",    'F', 19, 161.7, 44.2},

                              {"Tom",      'M', 20, 175.2, 66.3},

                              {"Stefany",  'F', 18, 159.3, 48.5},

                              {"Leonardo", 'M', 19, 172.8, 67.2}};

    //Mở file bằng ofstream

    ofstream ofs("people.txt");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    // Ghi từng thực thể của cấu trúc như một dòng vào file

    for (short i = 0; i < PERSON\_NUM; i++) {

        ofs << p[i].name << ' ' <<  p[i].sex<< ' '  << p[i].age<< ' '

        << p[i].height<< ' '  << p[i].weight << endl;

    }

    //Đóng file

    ofs.close();

    return 0;

}

Kết quả, nội dung mảng cấu trúc sẽ được ghi vào file như sau:

Bob M 19 165.4 72.5

Alice F 19 161.7 44.2

Tom M 20 175.2 66.3

Stefany F 18 159.3 48.5

Leonardo M 19 172.8 67.2

### Ghi file CSV trong C++

Ở phần trên, chúng ta đã tiến hành ghi nội dung các kiểu dữ liệu phức tạp như mảng hay cấu trúc vào file dưới dạng file txt rồi. Tuy nhiên chúng ta cũng có thể ghi dữ liệu trên vào file CSV trong C++, đơn giản bằng cách thay đổi đuôi file từ .txt sang .csv là xong.

Ví dụ cụ thể, chúng ta ghi dữ liệu từ mảng cấu trúc vào file CSV trong C++ bằng hàm fprintf như sau. Lưu ý là chúng ta cần viết định dạng các dữ liệu cách nhau bởi dấu phẩy để có thể ghi từng dữ liệu vào từng ô vào file CSV.

#include <iostream>

#include <fstream>

using namespace std;

#define PERSON\_NUM 5

//Khai báo kiểu struct

typedef struct {

    char name[20];

    char sex;

    int age;

    double height;

    double weight;

} person\_t;

int main()

{

    //Khởi tạo thực thể struct là một mảng struct

    person\_t p[PERSON\_NUM] = {{"Bob",      'M', 19, 165.4, 72.5},

                              {"Alice",    'F', 19, 161.7, 44.2},

                              {"Tom",      'M', 20, 175.2, 66.3},

                              {"Stefany",  'F', 18, 159.3, 48.5},

                              {"Leonardo", 'M', 19, 172.8, 67.2}};

    //Mở file bằng ofstream

    ofstream ofs("people.csv");

    //Kiểm tra file đã mở thành công hay chưa

    if(!ofs){

        cerr << "Error: file not opened." << endl;

        return 1;

    }

    // Ghi từng thực thể của cấu trúc như một dòng vào file

    for (short i = 0; i < PERSON\_NUM; i++) {

        ofs << p[i].name << ',' <<  p[i].sex<< ','  << p[i].age<< ','

        << p[i].height<< ','  << p[i].weight << endl;

    }

    //Đóng file

    ofs.close();

    return 0;

}

# OOP

## Hướng đối tượng trong C++ là gì

C++ chính là phiên bản của ngôn ngữ C được thêm vào lập trình hướng đối tượng.

POP- lập trình thủ tục trong C++

Lập trình thủ tục (Procedure Oriented Programming - POP) là một kỹ thuật lập trình truyền thống, nói một cách đơn giản thì giống như một bản viết trực tiếp các mệnh lệnh cần thiết để xử lý các thủ tục trong chương trình. Các thủ tục ở đây có thể là một chương trình con, hoặc một hàm nào đó trong chương trình, và chúng ta cần phải viết trực tiếp ra tất cả các đoạn mã cần thiết để xử lý chúng.

Ví dụ, chúng ta có chương trình in ra 5 số ngẫu nhiên. Với phương pháp lập trình thủ tục, chúng ta sẽ cần viết trực tiếp tất cả các mã lệnh trong chương trình ra như sau:

#include <iostream>

#include <cstdlib>

using namespace std;

int main(){

    for (int i = 0;i < 5;i++) {

        int newNum =  rand();

        cout << newNum <<endl;

    }

    return 0;

}

41

18467

6334

26500

19169

Trong chương trình trên, một biến newNum được tạo ra để lưu giữ số ngẫu nhiên được tạo ra. Với mỗi số ngẫu nhiên được tạo mới thì giá trị của nó sẽ được gán vào newNum, rồi sau đó sẽ được xuất ra màn hình. Mà một vòng lặp sẽ được thực hiện 5 lần để in ra 5 số ngẫu nhiên khác nhau.

Giống như thế, để đạt được mục đích in ra 5 lần 5 số ngẫu nhiên khác nhau, chúng ta cần viết trực tiếp ra toàn bộ các mã lệnh ra và thực thi chúng. Và phương pháp viết chương trình như vậy thì được gọi là lập trình thủ tục.

Trong lập trình thủ tục, để thu được kết quả mong muốn chúng ta chỉ cần viết các mã lệnh theo đúng thứ tự và thực thi chúng là xong. Không chỉ giới hạn trong chương trình đơn giản trên, trong các chương trình phức tạp với nhiều chương trình con hay hàm trong đó cũng vậy, chúng ta chỉ cần viết mã lệnh đúng thứ tự và thực thi chúng, thì sẽ ra được kết quả cuối cùng.

Lập trình thủ tục không phải là xấu. Thực tế thì chúng ta vẫn dùng nó hàng ngày, khi cần xử lý các bài tập lập trình, các chương trình nhỏ, bởi vì nó đơn giản, chỉ cần viết đúng và đủ xử lý thì sẽ cho ra kết quả cuối cùng.

Nhưng nó là chưa đủ. Hãy tưởng tượng với một chương trình lớn hơn, mà với bất kỳ thủ tục nào trong đó bạn cũng phải viết toàn bộ mã lệnh xử lý, thì tiến trình hoàn thành chương trình của bạn sẽ có tốc độ tồi tệ đến thế nào. Thay vì đó, hãy học cách tư duy và cách lập trình mới, để làm sao có thể viết ít dùng nhiều. Và do đó, chúng ta mới cần phải học và nắm vững về OOP - Lập trình hướng đối tượng trong C++.

Lập trình hướng đối tượng có tên tiếng anh là OOP (Object-Oriented Programming), là một phương pháp lập trình bằng cách đóng gói dữ liệu vào trong đối tượng và dùng chính đối tượng đó để thao tác dữ liệu.

Nói một cách đơn giản thì dữ liệu trong chương trình không đơn giản chỉ được coi là dữ liệu, mà chúng sẽ được mô hình hóa thành các đối tượng, có những thuộc tính riêng biệt, cũng như là các phương thức riêng để xử lý chúng.

## Đối tượng trong c++ là gì

Đối tượng trong c++ hay còn gọi là object trong C++, là các thực thể trừu tượng được tạo ra trong chương trình có tác dụng đóng gói và thao tác dữ liệu.

Một đối tượng sẽ gồm 2 thông tin là thuộc tính của nó và phương thức để xử lý nó.

Đặc tính của lập trình hướng đối tượng

Có 4 đặc tính quan trọng của lập trình hướng đối tượng trong C++ mà chúng ta cần nắm vững sau đây.

* Inheritance (Tính kế thừa ) trong lập trình hướng đối tượng có ý nghĩa, một class có thể kế thừa các thuộc tính của một class khác đã tồn tại trước đó. Khi một class con được tạo ra bởi việc kế thừa thuộc tính của class cha thì chúng ta sẽ gọi class con đó là subclass trong C++, và class cha chính là superclass trong C++.
* Abstraction (Tính trừu tượng) trong lập trình hướng đối tượng là một khả năng mà chương trình có thể bỏ qua sự phức tạp bằng cách tập trung vào cốt lõi của thông tin cần xử lý. Điều đó có nghĩa, bạn có thể xử lý một đối tượng bằng cách gọi tên một phương thức và thu về kết quả xử lý, mà không cần biết làm cách nào đối tượng đó được các thao tác trong class.
* Polymorphism (Tính đa hình) trong lập trình hướng đối tượng là một khả năng mà một phương thức trong class có thể đưa ra các kết quả hoàn toàn khác nhau, tùy thuộc vào dữ liệu được xử lý. Ví dụ đơn giản, cùng là một class quản lý dữ liệu là các con vật, thì hành động sủa hay kêu của chúng được định nghĩa trong class sẽ cho ra kết quả khác nhau, ví dụ nếu là con mèo thì kêu meo meo, còn con chó thì sủa gâu gâu chẳng hạn.
* Encapsulation (Tính đóng gói) và information hiding (che giấu thông tin ). Encapsulation (Tính đóng gói) trong lập trình hướng đối tượng có ý nghĩa không cho phép người sử dụng các đối tượng thay đổi trạng thái nội tại của một đối tượng, mà chỉ có phương thức nội tại của đối tượng có thể thay đổi chính nó. Điều đó có nghĩa, dữ liệu và thông tin sẽ được đóng gói lại, giúp các tác động bên ngoài một đối tượng không thể làm thay đổi đối tượng đó, nên sẽ đảm bảo tính toàn vẹn của đối tượng, cũng như giúp dấu đi các dữ liệu thông tin cần được che giấu. Ví dụ đơn giản, khi bạn dùng một cái iphone, bạn không thể thay đổi các cấu trúc bên trong của hệ điều hành iOS, mà chỉ có Apple mới có thể làm được điều này thôi.

# Class

## Class trong C++ là gì

Class trong C++ hay còn gọi là lớp trong C++ là các bản thiết kế ra các đối tượng dùng để đóng gói và xử lý dữ liệu trong lập trình hướng đối tượng C++.

Cấu trúc class trong C++ bao gồm 2 thành phần là biến thành viên (member variables) và hàm thành viên (member function) như sau:

Trong đó biến thành viên là các biến sử dụng trong nội bộ class có tác dụng lưu trữ thông tin chính là thuộc tính của đối tượng tạo ra từ class. Biến thành viên còn được gọi là field trong class c++.

Còn hàm thành viên là các hàm có tác dụng định nghĩa các xử lý có thể được sử dụng cho các đối tượng được tạo ra từ class. Hàm thành viên còn được gọi là method hay còn gọi là phương thức trong class C+++.

## Khai báo class trong C++

Để khai báo class (lớp) trong C++, chúng ta sẽ cần làm 2 việc, đó là định nghĩa class và tạo hàm thành viên trong class.

Trong đó việc định nghĩa class sẽ giúp khai báo tên class cũng như tên các biến thành viên và hàm thành viên có trong nó. Còn việc tạo hàm thành viên trong class sẽ giúp mô ta các xử lý cụ thể của từng hàm thành viên.

Thông thường thì 2 công việc này được viết riêng vào 2 file với đuôi file khác nhau lần lượt là classname.h và classname.cpp, tuy nhiên trong các chương trình nhỏ thì chúng ta có thể viết chúng trong cùng một file .cpp.

### Định nghĩa class trong C++ | class.h

Để định nghĩa class trong C++ chúng ta viết từ khóa class, sau đó đến tên class, rồi khai báo các thành viên của nó bên trong cặp dấu ngoặc nhọn {}, rồi lưu vào một file có tên classname.h như sau:

class classname

{

Khai báo tên biến và hàm thành viên;

};

Chúng ta có thể lược bỏ đi phần khai báo các thành viên trong class và tạo ra một class trống. Class này không có gì trong nó cả và chỉ có tác dụng giữ chỗ mà thôi.

VD:

#include <string>

using namespace std;

class Class1

{

private:

    string name; //Khai báo biến thành viên

public:

    void print1(); //Khai báo hàm thành viên print1

    void print2(); //Khai báo hàm thành viên print2

};

### Tạo hàm thành viên trong class | class.cpp

Sau khi đã khai báo tên các hàm thành viên như ở phần trên, chúng ta sẽ viết các xử lý cụ thể của các hàm thành viên đó, và lưu vào một file có tên là classname.cpp với cú pháp như sau:

#include “classname.h”

type classname::funcname()

{

    //Các xử lý trong hàm funcname

};

Trong đó:

* # include "classname.h" có tác dụng import file định nghĩa class đã tạo
* type là kiểu dữ liệu trả về của hàm thành viên funcname
* classname và funcname lần lượt là tên class và tên hàm thành viên.

Lưu ý giữa classname và funcname chúng ta sử dụng toán tử :: - toán tử phân giải phạm vi (Scope resolution operator) nhằm biểu thị phạm vi của hàm thành viên đó thuộc class nào.

Ví dụ, chúng ta tạo hàm thành viên của class1 và lưu vào file class1.cpp với nội dung như sau:

#include "Class1.h"

#include <iostream>

using namespace std;

void Class1::print1()

{

    name = "print1";

    cout << name << " đang được xử lý\n";

}

void Class1::print2()

{

    name = "print2";

    cout << name << " đang được xử lý\n";

}

Với 2 hàm thành viên ở trên thì do không trả về kết quả từ hàm, nên kiểu của hàm sẽ là void. Tuy nhiên thì chúng dùng biến name đã được khai báo trong phần định nghĩa class ở trên.

### Khai báo gộp class trong C++

Như đã nói ở trên, thì trong các chương trình lớn và phức tạp, chúng ta cần thiết phải chia việc khai báo class ra thành 2 phần và lưu trong 2 file khác nhau như trên, để dễ quản lý cũng như thuận tiện trong chia sẻ công việc nhóm.

Tuy nhiên đối với các chương trình nhỏ, chúng ta cũng hoàn toàn có thể viết gộp các công đoạn trên vào trong cùng một file.

Khi đó, chúng ta sẽ không cần phải include file classname.h nữa mà sẽ viết toàn bộ nội dung file này gộp vào file classname.cpp.

Ví dụ, chúng ta viết gộp cả 2 phần của class1 ở trên vào trong file class1.cpp như sau:

#include <string>

using namespace std;

/\*Định nghĩa class\*/

class Class1

{

private:

    string name; //Khai báo biến thành viên

public:

    void print1()    //Khai báo và viết xử lý hàm thành viên print1

    {

        name = "print1";

        cout << name << " đang được xử lý\n";

    }

    void print2()   //Khai báo và viết xử lý hàm thành viên print2

    {

        name = "print2";

        cout << name << " đang được xử lý\n";

    }

};

## Cách sử dụng class trong C++

Tùy thuộc vào việc có tạo đối tượng (instance) từ class hay không mà chúng ta có 2 phương pháp sử dụng class trong C++.

### Sử dụng class trong C++ bằng cách tạo instance

Để sử dụng class trong C++ bằng cách tạo ra instance, chúng ta có thể sử dụng cách tạo trực tiếp hoặc dùng toán tử new các cú pháp sau đây:

classname ins\_name;

OR

type \*ins\_name = new classname();

Trong đó type là kiểu dữ liệu, và ins\_name là tên của instance cần tạo ra từ classname. Lưu ý chúng ta cần viết dấu hoa thị trước tên biến \*ins\_name với ý nghĩa tạo ra con trỏ chỉ đến địa chỉ mà instance được lưu trong bộ nhớ.

Phương pháp này có ý nghĩa, chúng ta tạo ra một instance cụ thể từ class, rồi dùng các phương thức và thuộc tính trong class thông qua instance đó.

Ví dụ, chúng ta tạo ra 2 instance khác nhau trên từ class1 đã khai báo ở phần trên như sau:

#include <iostream>

#include <string>

using namespace std;

/\*Định nghĩa class\*/

class Class1

{

private:

    string name; //Khai báo biến thành viên

public:

    void print1() //Khai báo hàm thành viên print1

    {

        name = "print1";

        cout << name << " đang được xử lý\n";

    }

    void print2() //Khai báo hàm thành viên print2

    {

        name = "print2";

        cout << name << " đang được xử lý\n";

    }

};

int main() {

    /\*Tạo trực tiếp instance\*/

    Class1 cla1;

    cla1.print1(); //print1 đang được xử lý

    cla1.print2(); //print2 đang được xử lý

    /\*Tạo instance bằng toán tử new\*/

    Class1 \*cla2 = new Class1();

    cla2->print1(); //print1 đang được xử lý

    cla2->print2(); //print2 đang được xử lý

    delete cla2;

    return 0;

}

### Sử dụng class trong C++ không tạo instance

Ở phần trên, chúng ta đã truy cập và sử dụng các hàm thành viên trong class thông qua một instance được tạo ra từ class.

Và instance đó sẽ được thừa hưởng tất cả các thuộc tính lẫn phương thức của class nguồn tạo ra nó.

Nhưng thực thế trong chương trình, không phải lúc nào chúng ta cũng cần sử dụng tất cả các chức năng có trong một class, mà có thể chỉ dùng một phần nhỏ của class đó.

Khi đó, thay vì tạo một instance với đầy đủ chức năng của class để rồi chỉ sử dụng một phần rất nhỏ của nó, thì chúng ta có thể truy cập trực tiếp vào trong class và sử dụng chức năng, các hàm thành viên mà mình muốn. Lưu ý là chỉ có các biến hay hàm thành viên thuộc kiểu public mới có thể được truy cập từ bên ngoài class bằng phương pháp này mà thôi.

Với cách sử dụng class trong C++ không tạo instance này, chúng ta sẽ sử dụng trực tiếp các hàm thành viên trong class thông qua việc truy cập từ bên ngoài vào trong class đó.

Do đó, để tránh xung đột tên của hàm hay tên biến ở trong và ngoài một class, chúng ta cần thêm từ khóa static vào đầu tên hàm thành viên hoặc biến thành viên, với ý nghĩa rằng các hàm và biến này chỉ thuộc phạm vi bên trong class mà thôi. Khi đó, giả sử nếu có một hàm hoặc biến bên ngoài class có trùng tên chăng nữa thì chương trình cũng sẽ phân biệt được chúng với nhau.

Lưu ý đối với phương pháp này, do không tạo ra instance nên chúng ta cần phải khởi tạo giá trị của biến thành viên để có thể sử dụng như sau:

#include <iostream>

#include <string>

using namespace std;

/\*Định nghĩa class\*/

class Class1

{

public:

    static string name; //Khai báo biến thành viên kiểu static

public:

    static void print1() //Khai báo hàm thành viên print1 kiểu static

    {

        name = "print1";

        cout << name << " đang được xử lý\n";

    }

    static void print2() //Khai báo hàm thành viên print2 kiểu static

    {

        name = "print2";

        cout << name << " đang được xử lý\n";

    }

};

/\*Khởi tạo biến thành viên\*/

string Class1::name = "";

int main() {

    /\*Truy cập vào hàm thành viên trong class mà không cần tạo instance\*/

    Class1::print1(); //print1 đang được xử lý

    Class1::print2(); //print2 đang được xử lý

    return 0;

}

Còn đối với các chương trình không sử dụng tới biến thành viên trong đó thì chúng ta có thể lược bỏ đi phần này như sau:

#include<iostream>

using namespace std;

class MyMath {

public:

    static int max(int var1 , int var2) {

        if (var1 > var2) return var1;

        else return var2;

    }

};

int main() {

    cout << MyMath::max(100 , 101);

    return 0;

}

## Biến thành viên

Biến thành viên trong C++ có tên tiếng anh là member variable, là các biến được khai báo bên trong class, có tác dụng lưu trữ các thông tin của đối tượng được tạo ra từ class.

Trong biến thành viên lưu trữ thông tin của đối tượng, và chúng ta cũng có thể thay đổi trạng thái của đối tượng, thông qua việc thay đổi giá trị của các biến thành viên này.

Thông thường, các biến thành viên trong class được khai báo ở dạng private, do vậy chúng không thể được truy cập hay thay đổi từ bên ngoài class, mà chỉ có thể được tham chiếu truy cập hay thay đổi bằng hàm thành viên mà chúng ta đã được học trong bài Hàm thành viên trong C++.

### Khai báo biến thành viên trong C++

Biến thành viên thường được khai báo bên trong file myclas.h cùng với các thành phần khác của khi chúng ta khai báo class trong C++.

Trong khối class, chúng ta chỉ định Access Modifier của biến, sau đó là kiểu, tên biến, và giá trị nếu có của chúng như sau:

class Person {

/\*Khai báo biến thành viên\*/

private:

    int    m\_a;

    double  m\_b = 20;

/\*Khai báo hàm thành viên\*/

public:

    void  myfunc()

    //...

};

Access Modifier là từ khóa quy định một biến hay hàm thành viên trong class có thể được truy cập từ bên ngoài class hay không. Nếu chỉ định Access Modifier là public thì chúng ta có thể truy cập và sử dụng hàm thành viên đó từ bên ngoài class. Còn nếu chỉ định Access Modifier là private hoặc protected thì chúng ta sẽ không thể truy cập từ ngoài class vào hàm thành viên đó.

Ở ví dụ trên, chúng ta đã chỉ định Access Modifier của biến thành viên là private, do đó chỉ có thể truy cập vào biến thành viên từ bên trong class mà thôi.

Lại nữa, chúng ta thường sử dụng cụm ký tự \_m hoặc \_ đặt trước tên biến thành viên, với ý nghĩa biểu thị đây là member variable, nhằm phân biệt biến thành viên với các loại biến khác trong chương trình. Việc sử dụng cụm từ nào không phải là bắt buộc mà sẽ tuỳ thuộc vào quy định của từng team trong phát triển dự án phần mềm.

### Khai báo biến thành viên trong C++ với const

Tương tự như với Hàm thành viên trong C++ thì chúng ta cũng có thể dùng từ khoá const để chỉ định một biến có thể thay đổi giá trị được hay không, hay chỉ có thể dùng để đọc mà thôi.

Và một biến thành viên ở dạng const, sẽ mất đi khả năng thay đổi giá trị của nó, và chỉ có thể dùng để đọc giá trị.

Để biến một biến thành viên thành dạng const, chúng ta chỉ cần thêm từ khóa này vào trước kiểu của biến đó khi khai báo ví dụ dưới đây:

/\*Hàm thành viên không phải dạng const\*/

int    m\_a1;

/\*Hàm thành viên dạng const\*/

const int  m\_a2;

## Hàm thành viên

Hàm thành viên trong C++ có tên tiếng anh là member function, là các hàm hoặc toán tử được khai báo bên trong class, có tác dụng thực hiện các xử lý đối với đối tượng được tạo ra từ class.

Trong hàm thành viên miêu tả các xử lý tham chiếu và thay đổi các biến thành viên có trong class.

Thông thường, các biến thành viên trong class được khai báo ở dạng private, do vậy chúng không thể được truy cập hay thay đổi từ bên ngoài class, mà chỉ có thể được tham chiếu truy cập hay thay đổi bằng hàm thành viên mà thôi.

Do đó, bằng việc sử dụng hàm thành viên trong class++, chúng ta có thể từ bên ngoài một class mà thực hiện gián tiếp các xử lý truy cập lẫn thay đổi biến thành viên có trong nó.

Hai kiểu hàm thành viên trong C++

* Hàm thành viên dạng const: chỉ cho phép tham chiếu thuộc tính của đối tượng.
* Hàm thành viên không thuộc dạng const: cho phép tham chiếu và thay đổi thuộc tính của đối tượng. Đây là loại hàm thành viên thường sử dụng trong C++.

### Khai báo hàm thành viên trong C++

Chúng ta khai báo hàm thành viên trong trong khối của class, cũng như viết mô tả các xử lý trong nó với cú pháp tương tự như các hàm thông thường trong C++ như sau:

class Person {

private:

    double  m\_height;

    double  m\_weight;

public:

    void  myfunc(){  // Khai báo hàm thành viên

        // Các xử lý trong hàm

        std::cout << m\_height << "\n";

        std::cout << m\_weight << "\n";

    };

    //...

};

Trong đó myfunc() chính là một hàm thành viên. Khi khai báo hàm này, chúng ta cần lưu ý phải chỉ định Access Modifier cho nó, ví dụ như trên là public chẳng hạn.

Access Modifier là từ khóa quy định một biến hay hàm thành viên trong class có thể được truy cập từ bên ngoài class hay không. Nếu chỉ định Access Modifier là public thì chúng ta có thể truy cập và sử dụng hàm thành viên đó từ bên ngoài class. Còn nếu chỉ định Access Modifier là private hoặc protected thì chúng ta sẽ không thể truy cập từ ngoài class vào hàm thành viên đó.

### Chia file khai báo và file mô tả hàm thành viên

Đối với các chương trình nhỏ, chúng ta có thể viết gộp cả khai báo và các xử lý trong hàm thành viên như trên. Tuy nhiên đối với các chương trình lớn cần làm việc theo nhóm, thì thông thường việc khai báo và việc miêu tả xử lý trong hàm thành viên sẽ được viết riêng vào 2 file với đuôi file là myclass.h và myclass.cpp, cùng với các thành phần khác của class.

person.h:

class Person {

private:

    double  m\_height;

    double  m\_weight;

public:

    void  myfunc();

    //...

};

person.cpp:

#include "person.h"

void Person::myfunc()

{

    std::cout << m\_height << "\n";

    std::cout << m\_weight << "\n";

}

//...

Lưu ý với cách viết này, chúng ta cần thêm dòng #include "person.h" để import nội dung từ file person.h vào file person.cpp.

### Khai báo nội tuyến hàm thành viên trong C++

Đối với các hàm thành viên đơn giản, chúng ta có thể thực hiện việc khai báo nội tuyến (inline) trong đó việc khai báo và mô tả hàm được thực hiện cùng lúc trên một dòng.

VD:

class Person {

private:

    double  m\_height;

    double  m\_weight;

public:

    double  getHeight() const { return m\_height; }     //Trả về chiều cao

    void    setHeight(double ht) { m\_height = ht; }   // Thay đổi chiều cao

    //.....

};

### Hàm thành viên dạng const trong C++

Như đã trình bày ở trên thì bằng việc sử dụng từ khóa const trong khai báo hàm thành viên mà chúng ta có thể chỉ định hàm đó có khả năng thay đổi biến thành viên trong class hay không.

Và một hàm thành viên ở dạng const, sẽ mất đi khả năng thay đổi giá trị của biến thành viên trong class, và chỉ có thể đọc được giá trị đó mà thôi.

Để biến một hàm thành viên thành dạng const, chúng ta chỉ cần thêm từ khóa này vào trước cặp dấu {} khi khai báo hàm như ví dụ dưới đây:

/\*Hàm thành viên không phải dạng const\*/

void myfunc1(){

     // các xử lý

};

/\*Hàm thành viên dạng const\*/

void myfunc2() const{

     // các xử lý

};

Đối với một hàm thành viên ở dạng const, do nó không có khả năng thay đổi biến thành viên trong class, nên nếu tồn tại xử lý nào thay đổi biến thành viên của class trong nó, thì lỗi sẽ xảy ra.

### Gọi hàm thành viên trong class C++

Trong bài class trong C++ chúng ta đã biết cách sử dụng toán tử new hoặc không sử dụng nó để tạo ra instance khi sử dụng classs.

Và tùy thuộc vào cách chúng ta có sử dụng new hay không mà việc gọi hàm thành viên trong class C++ cũng sẽ khác nhau như sau:

#### Gọi hàm thành viên khi không dùng new

Để gọi hàm thành viên trong class trong trường hợp không dùng toán tử new, hãy sử dụng tới toán tử dấu chấm để gọi hàm thành viên với cú pháp sau đây:

obj.func();

Trong đó obj là tên instance và func là tên hàm thành viên cần gọi.

VD:

int main() {

    MyClass obj;  //Tạo instance tên obj từ MyClass

    obj.func();  // gọi hàm thành viên

    return 0;

}

#### Gọi hàm thành viên khi dùng new

Để gọi hàm thành viên trong class trong trường hợp dùng toán tử new, hãy sử dụng tới toán tử mũi tên để gọi hàm thành viên với cú pháp sau đây:

ptr->func();

Trong đó ptr là tên instance và func là tên hàm thành viên cần gọi.

VD:

int main() {

    MyClass \*ptr = new MyClass;  //Tạo instance tên ptr từ MyClass

    ptr->func();// gọi hàm thành viên

    return 0;

}

## Hàm khởi tạo

### Hàm khởi tạo trong c++ là gì

Hàm khởi tạo trong c++ hay còn gọi là constructor trong C++ là các hàm thành viên được tự động được thực thi khi chương trình tạo ra một đối tượng (instance) từ class, và được sử dụng với mục đích chính là khởi tạo các biến thành viên trong class đó.

Hàm khởi tạo trong c++ có 2 tính chất như sau:

* Tên hàm khởi tạo phải trùng với tên của class
* Hàm không mang kiểu dữ liệu trong nó, cũng như không sử dụng void khi khai báo nó.

Hàm khởi tạo, về bản chất cũng là một hàm thành viên trong class, tuy nhiên khác với các hàm thành viên khác thì nó chỉ có thể được gọi 1 lần duy nhất khi chúng ta tạo một instance từ class mà thôi. Sau khi đã được gọi và khởi tạo xong các biến thành viên, nó sẽ không thể được gọi thêm một lần nào nữa.

Có 2 loại hàm khởi tạo trong C++ phụ thuộc vào việc có tham số trong hàm hay không. Và nếu một hàm khởi tạo không chứa tham số trong nó thì chúng ta gọi nó là Hàm tạo mặc định (default constructor) trong C++.

### Khai báo hàm khởi tạo trong C++

Hàm khởi tạo (constructor) lần lượt được khai báo, và viết các xử lý chi tiết trong các file classname.h và classname.cpp với cú pháp như sau:

“MyClass.h”:

class MyClass {

public:

    MyClass();   //Khai báo hàm khởi tạo cùng tên với class

};

“MyClass.cpp”:

#include "MyClass.h"

MyClass::MyClass()

{

    //.....  //Viết lệnh khởi tạo các biến thành viên trong hàm

}

Tất nhiên đối với các chương trình nhỏ thì chúng ta cũng có thể viết gộp chúng lại trong một file .cpp duy nhất như sau:

class MyClass {

public:

    MyClass()    //Khai báo hàm khởi tạo cùng tên với class

    {

        //.....   // Viết lệnh khởi tạo các biến thành viên trong hàm

    }

};

Giống như trên thì hàm khởi tạo không mang kiểu dữ liệu của nó, cũng như không sử dụng tới void khi khai báo.

### Hàm khởi tạo mặc định trong C++ (Default constructor)

Chúng ta gọi một hàm khởi tạo không chứa tham số trong nó là Hàm tạo mặc định (Default constructor) trong C++.

#### Khai báo hàm khởi tạo mặc định trong C++

Để khai báo hàm khởi tạo mặc định trong C++, chúng ta viết toán tử bằng = giữa tên biến thành viên và giá trị của biến đó trong dấu ngoặc với cú pháp sau đây:

class MyClass {

private:

    int   m\_a;

public:

    MyClass()

    {

       m\_a = 123;

    }

};

Trong đó m\_a là tên biến thành viên và 123 là giá trị dùng để khởi tạo nó.

Trong trường hợp có nhiều biến thành viên trong class, chúng ta khai báo các biến cách nhau bởi dấu chấm phẩy như sau:

class MyClass {

private:

    int    m\_a;

    int    m\_b;

public:

    MyClass()

    {

        m\_a = 123;

        m\_b = 45;

    }

};

#### Sử dụng class chứa hàm khởi tạo mặc định trong C++

Do hàm khởi tạo mặc định không chứa tham số, nên lưu ý khi tạo một instance từ class, chúng ta sẽ không truyền đối số vào hàm và viết trực tiếp tên của instance tạo ra từ class như sau:

int main()

{

    MyClass sc1;

}

Đây là điểm rất khác biệt khi chúng ta sử dụng một class mà hàm khởi tạo của nó chứa tham số.

Ví dụ, chúng ta dùng hàm khởi tạo để khởi tạo giá trị của biến thành viên number của class sau đây với giá trị bằng 0 như sau:

#include <iostream>

#include <string>

using namespace std;

class SimpleClass

{

private:

    int number;

    string name;

public:

    //Khai báo hàm khởi tạo constructor

    SimpleClass()

    {

        //Khởi tạo giá trị biến thành viên number

        number = 0; // number(0);

    }

    int getNumber() { return number; } /\*Tạo hàm getNumber để lấy giá trị number\*/

    void setNumber(int n) { number = n; } /\*Tạo hàm setNumber để đổi giá trị number\*/

    string getName() { return name; }/\*Tạo hàm getName để lấy giá trị name\*/

    void setName(char\* s) { name = s; }/\*Tạo hàm setName để đổi giá trị name\*/

};

int main()

{

    SimpleClass sc;

    //Sau khi tạo instance sc, thì tại thời điểm này sc.number được khởi tạo bằng 0

    cout << sc.getNumber(); //0

}

### Hàm khởi tạo có tham số trong C++

Như các hàm khác thì chúng ta chỉ định các biến làm tham số trong hàm khởi tạo trong C++ như sau:

class MyClass {

private:

    int    m\_a;

public:

    MyClass(int a)

    {

        m\_a = a ;       // Sử dụng giá trị biến a để khởi tạo biến thành viên m\_a

    }

};

Với hàm khởi tạo kiểu này, khi tạo instance thì chúng ta cần truyền giá trị từ bên ngoài vào tham số trong classs, bằng cách viết giá trị truyền vào giữa cặp dấu ngoặc () như sau:

int main()

{

    MyClass sc1(123);

}

VD:

#include <iostream>

#include <string>

using namespace std;

class SimpleClass

{

private:

    int number;

    string name;

public:

    /\*Hàm khởi tạo mặc định không chứa tham số\*/

    SimpleClass()

    {

        number = 0;

    }

    /\*Hàm khởi tạo chứa tham số\*/

    SimpleClass(int n, string s)

    {

        number = n;

        name = s;

    }

    int getNumber() { return number; }

    void setNumber(int n) { number = n; }

    string getName() { return name; }

    void setName(string s) { name = s; }

};

int main()

{

    /\*Taọ instance với hàm khởi tạo mặc định\*/

    SimpleClass sc1;

    //number: 0 name:

    cout << "number: " << sc1.getNumber();

    cout << "\nname: " << sc1.getName() << endl;

    /\*Taọ instance với hàm khởi tạo chứa tham số, chúng ta cần truyền giá trị từ ngoài vào class\*/

    SimpleClass sc2(1, "Kiyoshi");

    cout << "number: " << sc2.getNumber();

    cout << "\nname: " << sc2.getName() << endl;

}

number: 0

name:

number: 1

name: Kiyoshi

## Hàm huỷ

### Hàm hủy (Destructor) trong class C++ là gì

Hàm hủy trong class C++ hay còn gọi là Destructor trong C++ là các hàm thành viên được tự động được thực thi khi chương trình hủy một đối tượng (instance) được tạo ra từ class, và được sử dụng với mục đích xóa và giải phóng đối tượng đó khỏi bộ nhớ.

Hàm hủy trong c++ có 3 tính chất như sau:

* Tên hàm hủy giống tên của class nhưng phải đặt kèm toán tử ~ đằng trước thành ~classname.
* Hàm hủy không mang kiểu dữ liệu trong nó, cũng như không sử dụng void khi khai báo nó.
* Hàm hủy không có tham số, cũng không trả về giá trị từ nó.

Nhiệm vụ chính của hàm hủy là giải phóng vùng bộ nhớ đã được sử dụng để lưu đối tượng sau khi nó bị hủy.

Tuy nhiên đối với các kiểu dữ liệu thuộc loại POD như int, double hay kiểu vector thì do chúng có cơ chế tự động giải phóng bộ nhớ nên chúng ta không cần dùng hàm hủy đối với các loại dữ liệu này.

Còn đối với các loại dữ liệu như mảng chẳng hạn vốn không có cơ chế trên, nên sau khi hủy đối tượng thì bắt buộc chúng ta cần phải dùng hàm hủy (Destructor) trong class C++ để giải phóng bộ nhớ đã dùng để lưu giữ chúng.

Trong trường hợp chúng ta không khai báo hàm hủy trong class, chương trình sẽ tự động tạo ra một hàm hủy trống (không có chứa xử lý) và cũng tự động được thực thi khi một đối tượng tạo ra từ class bị xóa khỏi chương trình.

### Khai báo hàm hủy trong C++

Cú pháp:

class MyClass {

public:

MyClass(); // Hàm khởi tạo

~MyClass(); // Hàm hủy

};

Trong đó MyClass là tên class, và ~MyClass với sự kết hợp của toán tử ~ và tên class chính là tên của hàm hủy.

Các xử lý trong hàm sẽ được viết kết hợp với toán tử delete bên trong khối của hàm hủy.

Ví dụ cụ thể, chúng ta tạo ra một hàm hủy trong class C++ để giải phóng bộ nhớ đã dùng để lưu mảng như sau:

#include <iostream>

using namespace std;

/\*Khai báo class\*/

class MyClass {

private:

    char \*m\_data;

public:

    MyClass()      // Hàm khởi tạo

    {

        m\_data = new char[100];    //Vùng bộ nhớ được tạo ra để lưu mảng

    }

    ~MyClass()    // Hàm hủy

    {

        delete [] m\_data;          //Giải phóng vùng bộ nhớ

        cout << "Ham huy duoc goi" << endl;

    }

};

/\*Bắt đầu hàm main\*/

int main()

{

    MyClass arr1;

    //Ham huy duoc goi

}

Giống như trên, khi tạo ra một đối tượng từ class thì biến thành viên m\_data sẽ được hàm khởi tạo khai báo giá trị bạn đầu là một mảng 100 phần tử. Để lưu giữ mảng này thì một vùng bộ nhớ sẽ được tự động sử dụng.

Và sau khi đối tượng được tạo ra bị xóa khỏi chương trình, hàm hủy sẽ được tự động thực thi. Khi đó toán tử delete trong hàm sẽ xóa và giải phóng vùng bộ nhớ đã dùng để lưu biến thành viên ban đầu, và dòng “Ham huy duoc goi” cũng được in ra màn hình.

### Toán tử delete và cách gọi hàm hủy trong C++

Toán tử delete trong C++ là một toán tử có tác dụng xóa và giải phóng vùng bộ nhớ đã sử dụng để lưu giữ một đối tượng, sau khi đối tượng đó bị xóa khỏi chương trình.

Toán tử delete sẽ có tác dụng ngược hoàn toàn với toán tử new mà chúng ta đã học trong bài Class trong C++. Toán tử new sẽ tạo ra instance từ class, còn toán tử delete sẽ xóa và giải phóng bộ nhớ đã dùng để lưu instance đó.

Nói cách khác thì toán tử delete cũng có tác dụng gọi hàm hủy trong class C++. Tuy nhiên có một điểm đặc biệt là chúng ta có thể sử dụng toán tử delete bên trong hàm hủy (trong class), hoặc là sử dụng nó độc lập bên ngoài class cũng được.

Hãy cùng nhớ lại trong bài Class trong C++ chúng ta đã biết có 2 cách để tạo ra instance từ class, đó là sử dụng toán tử new và không sử dụng toán tử new.Và tương ứng với 2 cách trên mà chúng ta cũng cần sử dụng toán tử delete để gọi hàm hủy trong C++ một cách tương ứng. Quy tắc ở đây là:

Nếu tạo instance từ class mà không dùng toán tử new, thì hàm hủy trong class sẽ tự động được gọi và giải phóng bộ nhớ sau khi kết thúc phiên làm việc với instance đó. Nói cách khác thì instance sẽ tồn tại cho tới khi kết thúc phiên làm việc với nó.

Nếu tạo instance từ class mà dùng toán tử new, thì hàm hủy trong class sẽ không tự động được gọi, mà phải chờ cho tới khi toán tử delete xuất hiện. Nói cách khác thì instance sẽ tồn tại ngay cả sau khi đã kết thúc phiên làm việc với nó, cho tới khi toán tử delete được gọi.

Ví dụ chúng ta có class sau đây:

#include <iostream>

using namespace std;

class MyClass {

private:

    char  \*m\_data;

public:

    MyClass()      // Hàm khởi tạo

    {

        m\_data = new char[100];    //Vùng bộ nhớ được tạo ra để lưu mảng

    }

    ~MyClass()    // Hàm hủy

    {

        delete [] m\_data;          //Giải phóng vùng bộ nhớ

        cout << "Ham huy duoc goi" << endl;

    }

};

Hãy so sánh kết quả khi có và không sử dụng toán tử new khi tạo instacne từ class như sau:

int main()

{

MyClass arr1;

// "Ham huy duoc goi"

MyClass \*arr2 = new MyClass();

// hàm hủy không được gọi, nên ko có gì xảy ra

}

Có thể thấy rõ ở lần tạo instance thứ nhất không dùng toán tử new, nên hàm hủy được tự động gọi, và dòng “Ham huy duoc goi” được in ra.

Tuy nhiên ở lần tạo instance thứ hai, do dùng toán tử new dẫn đến hàm hủy không được tự động gọi, nên không có gì in ra màn hình cả.

Bây giờ, hãy xem khi chúng ta thêm toán tử delete thì điều gì xảy ra:

int main()

{

MyClass arr1;

// "Ham huy duoc goi"

MyClass \*arr2 = new MyClass();

// hàm hủy không được gọi, nên ko có gì xảy ra

delete arr2;

// "Ham huy duoc goi"

}

## Static

### Static trong class C++ là gì

Static trong C++ là một toán tử có tác dụng chỉ định một biến hoặc hàm thành viên trong class tồn tại ở dạng tĩnh.

Dạng tĩnh ở đây có ý nghĩa, vùng bộ nhớ dùng để lưu trữ các dữ liệu này sẽ là bất biến, không thay đổi. Đối ngược với dạng tĩnh chính là dạng động (dynamic), ví dụ như khi chúng ta tạo ra các mảng động có thể tự thay đổi kích thước chẳng hạn.

Bằng việc sử dụng static trong class C++, chúng ta sẽ có được biến static và hàm static, là các thành viên được lưu tại các vùng bộ nhớ bất biến. Vì ở dạng tĩnh nên chúng sẽ tồn tại duy nhất trong class, và do đó chúng có thể được sử dụng chung cho tất cả các đối tượng được tạo ra từ class.

Khác với biến static và hàm static là các hàm và biến thành viên thông thường được tạo ra khi chúng ta tạo một đối tượng (instance) từ class. Các thành viên này được lưu giữ tại các vùng bộ nhớ khác nhau và chỉ có thể được sử dụng từ chính đối tượng đã tạo ra chúng mà thôi.

### Biến static trong class C++

Biến static trong C++ là biến thành viên ở dạng tĩnh và tồn tại duy nhất trong class. Biến static có khả năng được sử dụng chung cho tất cả các đối tượng được tạo ra từ class. Ngoài ra do nó ở dạng tĩnh nên chúng ta có thể sử dụng trực tiếp nó mà không cần phải tạo đối tượng từ class.

Để tạo biến static trong class C++, chúng ta thêm toán tử static vào đằng trước tên biến khi khai báo biến trong class như sau. Lưu ý là access modifier của biến static phải ở dạng public để cho phép truy cập nó từ ngoài phạm vi class.

class TestClass

{

public:

    int num;          //Khai báo biến thông thường

    static int sNum;  //Khai báo biến static

//...

};

Ưu điểm lớn nhất của biến static đó chính là chúng ta có thể sử dụng trực tiếp nó mà không cần phải tạo ra đối tượng (instance) từ class. Tuy nhiên do không tạo ra instance dẫn đến việc giá trị ban đầu của biến static cũng không được khởi tạo, nên khi dùng biến static mà không tạo ra đối tượng, chúng ta cần phải khởi tạo giá trị cho nó ở bên ngoài phạm vi class.

VD:

#include <iostream>

#include <string>

using namespace std;

class TestClass

{

public:

    int num;          //Khai báo biến thông thường

    static int sNum;  //Khai báo biến static

public:

    TestClass(){ num = 10;}

};

//Khởi tạo giá trị cho biến static

int TestClass::sNum = 100;

int main()

{

    /\*Truy cập các biến thành viên mà không tạo ra instance\*/

    cout << TestClass::sNum; // 100

    cout << TestClass::num;  // error: invalid use of non-static data member

}

Giống như ví dụ trên, do sNum là biến static nên chúng ta có thể trực tiếp sử dụng giá trị của nó mà không cần phải tạo ra đối tượng từ class. Tuy nhiên biến num thì chỉ là biến thành viên bình thường nên chúng ta không thể sử dụng trực tiếp nó vì lỗi sẽ xảy ra.

Lưu ý cách viết TestClass::sNum như trên sử dụng toán tử : nhằm biểu thị biến thành viên của một class.

### Hàm static trong class C++

Hàm static trong C++ là hàm thành viên ở dạng tĩnh và tồn tại duy nhất trong class. Giống với biến static thì hàm static có khả năng được sử dụng chung cho tất cả các đối tượng được tạo ra từ class, và chúng ta có thể sử dụng trực tiếp nó mà không cần phải tạo đối tượng từ class.

Để tạo hàm static trong class C++, chúng ta thêm toán tử static vào đằng trước tên hàm khi khai báo hàm trong class như sau. Lưu ý là access modifier của hàm static phải ở dạng public để cho phép truy cập nó từ ngoài phạm vi class.

class TestClass

{

public:

    void func1() { cout << "func1 called" <<endl }          //Khai báo hàm thông thường

    static void func2() {  cout << "func2 called" <<endl}   //Khai báo hàm static

//...

};

Ưu điểm lớn nhất của hàm static đó chính là chúng ta có thể sử dụng trực tiếp nó mà không cần phải tạo ra đối tượng (instance) từ class.

VD:

#include<iostream>

using namespace std;

class TestClass

{

public:

    static void func1()  { cout<<"func1 called"; }  //Khai báo hàm static

    void func2()  { cout<<"func2 called"; }         //Khai báo hàm thông thường

};

int main()

{

    /\*Gọi các hàm thành viên mà không tạo ra instance\*/

    TestClass::func1(); //func1 called

    TestClass::func2(); //error: cannot call member function

}

Giống như ví dụ trên, do func1 là hàm static nên chúng ta có thể trực tiếp gọi nó mà không cần phải tạo ra đối tượng từ class. Tuy nhiên hàm fucn2 thì chỉ là hàm thành viên bình thường nên chúng ta không thể sử dụng trực tiếp nó vì lỗi sẽ xảy ra.

Lưu ý đối với các hàm static có xử lý các biến thành viên, do chúng ta không tạo ra instance dẫn đến giá trị của các biến thành viên này sẽ không được khởi tạo, nên khi sử dụng chúng, bắt buộc phải có bước khởi tạo giá trị cho các biến thành viên. Và đương nhiên để làm được điều đó, các biến thành viên này cũng phải là biến static.

#include<iostream>

using namespace std;

class TestClass

{

public:

    static int sNum;  //Khai báo biến static

public:

    // static member function

    static void func1()  { cout<<sNum; } ////Khai báo hàm static

};

//Khởi tạo giá trị cho biến static

int TestClass::sNum = 100;

int main()

{

    /\*Truy cập các hàm thành viên mà không tạo ra instance\*/

    TestClass::func1(); //100

}

## This

### This trong class c++ là gì

This trong class c++ hay còn gọi là con trỏ this trong class c++ là một con trỏ đặc biệt ẩn dấu bên trong hàm thành viên của class, có tác dụng trỏ đến đối tượng đã gọi đến hàm thành viên đó.

Điều đó có nghĩa, khi một đối tượng gọi tới một hàm thành viên nào đó trong class thì con trỏ this sẽ trỏ đến vị trí của đối tượng đó để xác định đối tượng đang trong phiên làm việc, qua đó nó có thể giúp đối tượng đó có thể tiếp tục truy cập và xử lý các biến thành viên có trong class.

Mặc dù con trỏ this được tự động khai báo và cống hiến thầm lặng cho chúng ta sau khi tạo hàm thành viên trong class, nhưng thông thường thì nó sẽ được ẩn đi, do vậy chúng ta không mấy để ý tới sự tồn tại của nó. Tuy nhiên nếu viết rõ ra con trỏ this trong khai báo hàm thì sẽ giúp chúng ta hình dung rõ hơn về các xử lý trong class C++.

### Cách viết this trong class c++

Ví dụ chúng ta có một class đơn giản giúp tính tổng 2 số như sau:

#include <iostream>

using namespace std;

class MyClass

{

private:

    int x;

    int y;

public:

    MyClass(int tmpx, int tmpy)

    {

        x = tmpx;

        y = tmpy;

    };

    void sum() { cout << x + y ;}

};

int main() {

    MyClass dc(2,3);

    dc.sum(); //5

    return 0;

}

Trong class trên, hàm khởi tạo MyClass sẽ nhận 2 giá trị truyền vào khi chúng ta tạo đối tượng (instance) từ class. Sau đó 2 giá trị này sẽ được dùng để khởi tạo giá trị cho 2 biến thành viên tương ứng là x và y. Cuối cùng thì dùng 2 biến này để tính tổng bên trong hàm sum() và in ra kết quả ra màn hình.

Với chương trình trên, thực tế là con trỏ this đã bị giấu đi rồi, mặc dù nó vốn tồn tại. Và chúng ta sẽ viết rõ con trỏ this trong class C++ thông qua từ khoá this kèm với toán tử mũi tên -> vào các vị trí nó tồn tại như sau:

#include <iostream>

using namespace std;

class MyClass

{

private:

    int x;

    int y;

public:

    MyClass(int tmpx, int tmpy)

    {

        this->x = tmpx;

        this->y = tmpy;

    };

    void sum() { cout << this->x + this->y ;}

};

int main() {

    MyClass dc(2,3);

    dc.sum(); //3

    return 0;

}

Cả 2 cách viết trên đều đưa ra kết quả giống nhau. Tuy nhiên hãy cùng so sánh sự khác biệt giữa 2 cách viết như sau:

Có thể thấy, do con trỏ this sẽ trỏ tới đối tượng đang gọi hàm, trong trường hợp này là instance dc, nên con trỏ this hiện sẽ đại diện cho instance dc.

Do vậy, bằng cách ghi this -> x và this -> y , chúng ta có thể hiểu được instance dc đang thực hiện tham chiếu tới các biến thành viên trong class là x và y, qua đó có thể hiểu rõ hơn các xử lý trong class hiện đang phục vụ cho đối tượng nào trong chương trình.

## Access modifier

### Access modifier trong C++ là gì

Access modifier trong C++ là các toán tử có tác dụng chỉ định quyền truy cập đối với các biến và hàm thành viên có trong một class.

Nhờ có Access modifier mà chúng ta có thể quyết định việc một thành phần ở bên ngoài một class có thể truy cập tới các thành viên bên trong class hay không, qua đó bảo vệ được thông tin chứa trong class đó.

Có 3 kiểu Access modifier là public protected private trong C++, tương ứng với chúng là các quyền truy cập được mô tả như bảng dưới đây:

Toán tử Từ trong class Từ class kế thừa Từ ngoài class

public Có thể Có thể Có thể

private Có thể Không thể Không thể

protected Có thể Có thể Không thể

Lưu ý trong chương trình, nếu chúng ta lược bỏ hoặc không chỉ rõ access modifier khi khai báo thành viên của lớp, thì mặc định access modifier Private sẽ được chỉ định.

#### Public trong C++

Public trong C++ là Access modifier có tác dụng chỉ định thành viên của class có thể được truy cập từ bất cứ đâu trong chương trình.

Public dịch sang tiếng Việt nghĩa là công khai, do vậy tất cả các thành viên của lớp được khai báo dưới dạng public sẽ có thể được truy cập từ bất cứ đâu trong chương trình. Ví dụ như từ bên trong class, bên ngoài class, hoặc trong một class và hàm khác chẳng hạn.

#### Private trong C++

Private trong C++ là Access modifier có tác dụng chỉ định thành viên của class chỉ có thể được truy cập từ bên trong class mà thôi.

Private dịch sang tiếng Việt nghĩa là riêng tư, do vậy tất cả các thành viên của lớp được khai báo dưới dạng private sẽ được giấu đi với phần bên ngoài class, và chỉ có thể được truy cập từ các hàm thành viên trong class đó mà thôi.

#### Protect trong C++

Protect trong C++ là Access modifier có tác dụng chỉ định thành viên của class chỉ có thể được truy cập từ bên trong class và từ các class con (class kế thừa) mà thôi.

Protect dịch sang tiếng Việt nghĩa là bảo vệ, do vậy tất cả các thành viên của lớp được khai báo dưới dạng protect sẽ được bảo vệ, và chỉ cho phép truy cập từ class cũng như các class con của nó mà thôi.

## Kết thừa

### Kế thừa trong C++ là gì

Kế thừa trong C++ là một chức năng đặc biệt quan trọng của class trong C++, giúp chúng ta có thể tạo ra một class bằng cách kế thừa các thuộc tính và chức năng của một class khác đã tồn tại trước đó.

Khi một class con được tạo ra bởi việc kế thừa thuộc tính của class cha thì chúng ta sẽ gọi class con đó là subclass trong C++, và class cha chính là superclass trong C++.

### Cú pháp kế thừa trong C++

Để viết kế thừa trong C++, chúng ta sử dụng cú pháp sau đây:

class BaseClass

{

};

class DerivedClass : public BaseClass

{

};

Trong đó BaseClass và DerivedClass lần lượt là class cha và class con.

Lưu ý chúng ta cần phải khai báo class cha trước rồi mới khai báo class con sau, nếu làm ngược lại thì lỗi sẽ bị xảy ra.

VD:

#include <iostream>

using namespace std;

/\*Class cha\*/

class BaseClass

{

public:

    void print()

    {

        cout << "BassClass\n";

    }

};

/\*Class con được kế thừa từ class cha\*/

class DerivedClass : public BaseClass

{

public:

    /\*Thêm chức năng mới vào class con\*/

    void printNew()

    {

        cout << "DerivedClass\n";

    }

};

int main()

{

    //Tạo instance từ class con

    DerivedClass dc;

    //Do class con được kế thừa từ cha nó,

    //nên chúng ta có thể gọi và sử dụng tất cả các hàm thành viên từ trong class cha

    dc.print();          //BassClass

    //Và cũng có thể gọi chức năng chỉ có trong class con

    dc.printNew();         //DerivedClass

}

Giống như ví dụ trên, mặc dù chúng ta không khai báo hàm print() bên trong class DerivedClass, nhưng do nó được kế thừa từ class BaseClass, nên nó có khả năng gọi hàm print() được kế thừa từ class cha của nó.

### Truy cập biến và hàm thành viên của superclass trong C++

Về cơ bản thì class con (subclass) có thể kế thừa các biến và hàm thành viên từ class cha (superclass), ngoại trừ những biến và hàm thành viên có Access modifier là private.

Nói cách khác, từ class con, chúng ta chỉ có thể truy cập vào các biến và hàm thành viên có Access modifier là public hoặc protect trong class cha (superclass) mà thôi.

### Kế thừa constructor trong c++

Thứ tự thực thi constructor và Destructor khi kế thừa class

Khi chúng ta tạo ra một instance từ một class con được kế thừa từ class cha, tất nhiên thì constructor (hàm khởi tạo) của class con sẽ được gọi, nhưng thực tế thì trước khi constructor của class con được gọi thì constructor của class cha đã được thực thi rồi.

Thế nên, thứ tự xử lý khi chúng ta tạo một instance từ một class con sẽ là như sau:

Gọi constructor của class cha => Gọi constructor của class con => tạo instance của class con

Ở chiều ngược lại thì khi phá huỷ một instacne của class con, các Destructor (hàm huỷ) sẽ được gọi theo thứ tự như sau :

Gọi Destructor của class con => Gọi Destructor của class cha => Xoá instance của class con

VD:

#include <iostream>

using namespace std;

class BaseClass

{

public:

    BaseClass()

    {

        cout << "BassClass Constructor" << endl;

    }

    ~BaseClass()

    {

        cout << "BassClass Destructor" << endl;

    }

};

class DerivedClass : public BaseClass

{

public:

    DerivedClass()

    {

        cout << "DerivedClass Constructor" << endl;

    }

    ~DerivedClass()

    {

        cout << "DerivedClass Destructor" << endl;

    }

};

int main()

{

    DerivedClass \*dc = new DerivedClass();

    delete dc;

}

BassClass Constructor

DerivedClass Constructor

DerivedClass Destructor

BassClass Destructor

#### Cú pháp kế thừa constructor trong c++

Để kế thừa constructor trong C++, chúng ta sử dụng cú pháp sau đây:

class BaseClass

{

BaseClass()

};

class DerivedClass : public BaseClass

{

DerivedClass():BaseClass()

};

Trong đó BaseClass và DerivedClass lần lượt là class cha và class con. Và DerivedClass() và BaseClass() lần lượt là các constructor (hàm khởi tạo) của class con và class cha.

Lưu ý giữa 2 constructor, chúng ta sử dụng toán tử : biểu thị mối liên hệ kế thừa giữa chúng.

#### Ví dụ về kế thừa constructor trong c++

Chúng ta có thể kế thừa constructor trong c++, khi tạo ra một instance của class con, khi nó được kế thừa constructor từ class cha như ví dụ sau:

#include <iostream>

using namespace std;

/\*Khai báo class cha\*/

class BaseClass

{

public:

    int number;

    std::string name;

    BaseClass(int n = 0, const char \*s = "")

    {

        number = n;

        name = s;

    }

    void print()

    {

        cout << number << endl;

        cout << name << endl;

    }

};

/\*Khai báo class con được kế thừa từ class cha\*/

class DerivedClass : public BaseClass

{

public:

    //Do được kế thừa nên nó có thể gọi constructor được kế thừa từ class cha

    DerivedClass(int n = 0, const char \*s = "") : BaseClass(n, s)

    {

    }

};

int main()

{

    DerivedClass dc(1, "Kiyoshi");

    dc.print();

}

1

Kiyoshi

### Đa kế thừa trong C++

Ở các ví dụ trên, chúng ta đã hình dung ra được cách mà một class con kế thừa các biến và hàm thành viên từ class cha của nó. Hình thức kế thừa một con một cha như vậy được gọi là đơn kế thừa trong C++.

Nhưng có một điều rất đặc biệt trong C++, đó chính là khả năng một class không những có thể nhận kế thừa từ một class, mà nó còn có khả năng nhận kế thừa từ nhiều class khác nhau. Và đó chính là tính đa kế thừa trong C++.

Đa kế thừa trong C++ có tên tiếng anh là Multiple Inheritance, là một chức năng đặc biệt quan trọng của class trong C++, trong đó một class có thể nhận kế thừa từ nhiều class khác nhau.

Nhờ có tính đa kế thừa trong C++ mà chúng ta có thể tái sử dụng rất nhiều các class có sẵn để có thể tạo ra một class mới, thông qua việc kế thừa từ chúng, qua đó có thể rút ngắn thời gian viết code và nâng cao hiệu năng phát triển phần mềm.

#### Cú pháp đa kế thừa trong C++

Khi tạo một class con kế thừa từ nhiều class cha, chúng ta sử dụng cú pháp đa kế thừa trong C++ sau đây:

class BaseClass1

{

};

class BaseClass2

{

};

class BaseClass3

{

};

class DerivedClass : public BaseClass1, public BaseClass2, public BaseClass3

{

};

Trong đó DerivedClass là class con được nhận đa kế thừa từ các class cha BaseClass. Lưu ý chúng ta viết các class cha cách nhau bởi dấu phẩy như trên.

Ví dụ cụ thể về đa kế thừa trong C++

Hãy cùng xem ví dụ đơn giản về đa kế thừa, đó là kế thừa 2 lớp trong c++ với một class con được kế thừa từ 2 lớp cha như sau:

#include <iostream>

using namespace std;

/\*Khai báo class cha\*/

class BaseClass1 {

public:

    //Khai báo Constructor của BaseClass1

    BaseClass1() {

      cout << "BaseClass1 constructor called" << endl;

    }

    //Khai báo hàm thành viên của BaseClass1

    void print() {cout<<"BaseClass1 called" <<endl;}

};

/\*Khai báo class mẹ\*/

class BaseClass2 {

public:

    //Khai báo Constructor của BaseClass2

    BaseClass2() {

      cout << "BaseClass2 constructor called" << endl;

    }

};

/\*Khai báo class con được kế thừa từ class cha\*/

class DerivedClass: public BaseClass1, public BaseClass2 {

public:

    //Khai báo hàm thành viên của DerivedClass

    void print2() {cout<<"DerivedClass called"<< endl;}

};

int main() {

    /\*Khi tạo insstance của class con thì 2 constructor của class cha cũng được gọi\*/

    DerivedClass dc;   //BaseClass1 constructor called //BaseClass2 constructor called

    /\*Do được kế thừa nên có thể gọi hàm thành viên của class cha\*/

    dc.print();  //BaseClass1 called

    /\*Và cũng có thể tự gọi các hàm thành viên của chính class con\*/

    dc.print2(); //DerivedClass called

    return 0;

}

Có thể thấy rõ, class con DerivedClass do được đa kế thừa từ 2 class là BaseClass1 và BaseClass2, nên khi instance của nó được tạo ra thì các hàm khởi tạo Constructor của 2 hàm cha cũng sẽ được gọi.

Và nó có thể gọi hàm thành viên của chính nó, hoặc là các hàm thành viên được kế thừa từ class cha mẹ của nó như trên.

## Friend

### Friend trong C++ là gì

Trong class C++, chúng ta sử dụng các access modifier để quyết định quyền truy cập vào các dữ liệu chứa trong nó, trong đó cần chú ý nhất là public - quyền truy cập công khai và private- quyền truy cập riêng tư.

Ví dụ đối với một biến thành viên của class ở dạng private, khi đó chúng ta chỉ có thể truy cập vào nó ở bên trong phạm vi class mà thôi. Do nó là riêng tư, không công khai nên chúng ta không thể truy cập từ ngoài class vào biến đó được.

Vấn đề đặt ra là, khi cần phải truy cập một biến riêng tư từ bên ngoài, mà không muốn thay đổi quyền truy cập tới biến đó thì chúng ta cần phải làm thế nào? Và đó là lý do mà chức năng Friend trong C++ đã ra đời.

Friend là một chức năng rất quan trọng trong C++, giúp cho phép một hàm hay một class có quyền truy cập vào một thành viên riêng tư của một class khác.

Nhờ có chức năng friend mà các hàm và class có thể dùng chúng dữ liệu riêng tư của một class có sẵn, qua đó rút gọn thời gian viết code và nâng cao hiệu suất phát triển phần mềm.

### Lớp bạn (Friend class) trong C++

Giống như vậy, một friend class trong C++ hay còn gọi là lớp bạn trong C++, chính là một class có quyền truy cập vào một thành viên riêng tư (private) của một class khác.

Để chỉ định một class làm lớp bạn của một class sẵn có, chúng ta viết cụm từ khóa friend class và tên của friend class này vào class sẵn có, với cú pháp sau đây:

class BaseClass

{

friend class FriendClass;

};

class FriendClass

{

};

Trong đó BaseClass là class ban đầu, và FriendClass là lớp bạn của nó.

VD:

/\*Khai báo class nguồn\*/

class BaseClass{

private:

    string m\_b;

public:

    BaseClass()

    {

        m\_b = "BaseClass called";

    }

    //Chỉ định class bạn của BaseClass là FriendClass

    friend class FriendClass;

};

/\*Khai báo class bạn của class nguồn\*/

class FriendClass{

public:

    void print(BaseClass arg)

    {

        cout<<arg.m\_b;

    }

};

Trong ví dụ trên, FriendClass là class bạn của BaseClass, do vậy nó có quyền truy cập vào biến private của BaseClass là biến m\_b.

Khi sử dụng Lớp bạn (Friend class) trong C++, chúng ta sẽ truy cập vào biến private của lớp nguồn từ lớp bạn.

VD:

#include <iostream>

using namespace std;

/\*Khai báo class nguồn\*/

class BaseClass{

private:

    string m\_b;

public:

    BaseClass()

    {

        m\_b = "BaseClass called";

    }

    friend class FriendClass;

};

/\*Khai báo class bạn của class nguồn\*/

class FriendClass{

public:

    void print(BaseClass arg)

    {

        cout<<arg.m\_b;

    }

};

int main()

{

    /\*Tạo instance của class nguồn và class bạn\*/

    BaseClass b;

    FriendClass f;

    /\*Truy cập vào biến private của lớp nguồn\*/

    f.print(b);//BaseClass called

    return 0;

}

### Hàm bạn (Friend function) trong C++

Tương tự như lớp bạn thì một friend function trong C++ hay còn gọi là hàm bạn trong C++, chính là một hàm có quyền truy cập vào một thành viên riêng tư của một class có sẵn.

Để chỉ định một hàm làm hàm bạn của một class sẵn có, chúng ta viết từ khóa friend đằng trước phần khai báo hàm này vào class nguồn, với cú pháp sau đây:

class MyClass

{

friend void func(MyClass obj);

};

void func(MyClass obj)

{

};

Trong đó MyClass là tên class ban đầu, obj là đối tượng tạo bởi class và func là hàm bạn của class đó.

VD:

#include <iostream>

using namespace std;

/\*Khai báo class\*/

class MyClass

{

private:

    int num1, num2;

public:

    /\*Khai báo hàm tạo\*/

    MyClass(){num1 = 10; num2=20; }

    //Chỉ định hàm bạn của class là hàm func

    //với tham số là đối tượng ob tạo ra từ class

    friend void func(MyClass ob);

};

/\*Khai báo hàm func, với tham số là đối tượng tạo từ class\*/

void func(MyClass ob)

{

    cout<< ob.num1 +  ob.num2;

}

int main()

{

    /\*Tạo đối tượng (instance) từ class\*/

    MyClass c;

    //Do là hàm bạn của class nên func có thể truy cập

    //vào 2 biến private của class là num1 và num2 và tính tổng

    func(c); //30

}

Trong ví dụ trên, FriendClass là class bạn của BaseClass, do vậy nó có quyền truy cập vào biến private của BaseClass là biến m\_b.

#### Hàm bạn của class nằm trong một class khác

Ở phần trên chúng ta đã học cách dùng một hàm bạn của class khi hàm đó được khai báo độc lập.

Tuy nhiên cũng có những trường hợp hàm bạn này lại được khai báo ở bên trong của một class khác.

Khi đó, ngoài việc chỉ định hàm bạn của class, chúng ta cũng cần phải chỉ định thêm class chứa hàm bạn đó cũng là lớp bạn của class ban đầu.

VD:

#include <iostream>

using namespace std;

class MyClass

{

private:

    int num1, num2;

public:

    MyClass(){num1 = 10; num2=20; }

    friend class FriendClass;     //Chỉ định class FriendClass là hàm bạn của class

    friend void func(MyClass ob); //Chỉ định hàm func là hàm bạn của class

};

/\*Khai báo class chứa hàm bạn của class MyClass\*/

class FriendClass{

public:

    void func(MyClass ob)

    {

        cout<< ob.num1 +  ob.num2;

    }

};

int main()

{

    MyClass c;

    FriendClass d;

    d.func(c); //30

}

Nếu chúng ta bỏ qua việc chỉ định class chứa hàm bạn cũng là lớp bạn của class ban đầu, kết quả lỗi chương trình sẽ bị xảy ra.

# Tham khảo

Lập trình căn bản:

<https://laptrinhcanban.com/cpp/lap-trinh-cpp-co-ban/>

Thuật toán tìm kiếm nhị phân:

<https://www.youtube.com/watch?v=dB2DWSKGLj8>