PHẦN HAI: CHƯA BIẾT ĐẶT TÊN GÌ

**§1: DEBUG VỚI CODE::BLOCKS**

**1. Debug là gì ?**

Lập trình vốn là một công việc nặng nhọc và căn thẳng, vì thế không phải lúc nào người viết code cũng thật sự tỉnh táo để thực hiện chương trình của mình một cách trơn tru nhất. Đa phần trong chúng ta đều không thể tránh khỏi những lỗi khi lập trình, do đó sinh ra “bug”.

Bug hiểu theo nghĩa đen là những con bọ, ám chỉ các lỗi xảy ra trong logic, hay bất kì vấn đề gì khiến cho chương trình không thực hiện đúng như ý muốn của người lập trình. Bug luôn tìm tàn khắp mọi nơi, trên từng dòng code và có thể nói là chúng rất khó kiểm soát.

Ví dụ: *Viết chương trình tính tổng hai số được nhập vào từ bàn phím.*

Ta xét đoạn code:

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {  int a, b;  cout << “Nhap a va b: “;  cin >> a >> b;  cout << a + b;  return 0;  } |

Khi nhập a = 3, b = 4 thì hiển nhiên ta sẽ được kết quả là 7. Tuy nhiên ta thử nhập 3.1 và 4.1 thì kết quả vẫn là 7, đây là kết quả không chính xác. Do đó ta xem như đoạn chương trình này toàn tại bug khiến cho kết quả của chương trình là không đúng.

Để khắc phục lỗi trên, ta có thể kiểm tra cụ thể từng dòng code để xem lỗi xảy ra ở đâu. Tuy nhiên, với những chương trình lớn hằng trăm, hàng nghìn dòng code thì việc này là dường như bất khả thi. Do đó trên hầy hết các IDE hiện đại đều hỗ trợ khả năng gỡ rối (debug) chương trình bằng cách chạy từng dòng lệnh (step by step) và xem xét sự thay đổi gia trị của biến (watch variables).

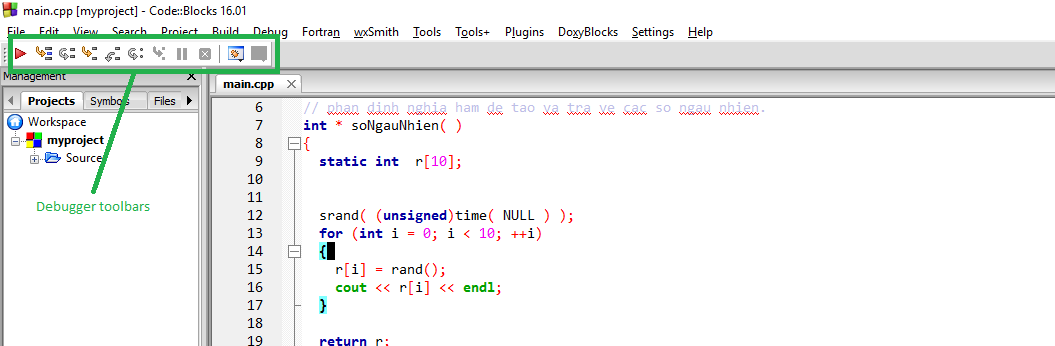
**2. Debug với Code::Blocks**

***a. Debugger toolbars***

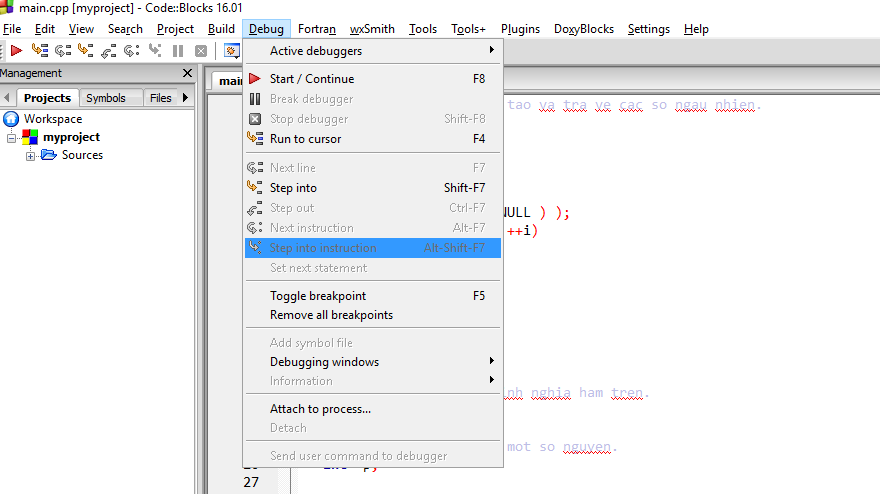
Để mở debugger toolbars trong Code::Blocks ta lần lượt thao tác như sau:

**View → Toolbars → Debugger**

Khi đó ta sẽ thấy xuất hiện những công cụ của debugger trên thanh toolbars.



Từ đây ta có thể sử dụng toolbars này để debug hoặc dử dụng option **Debug**



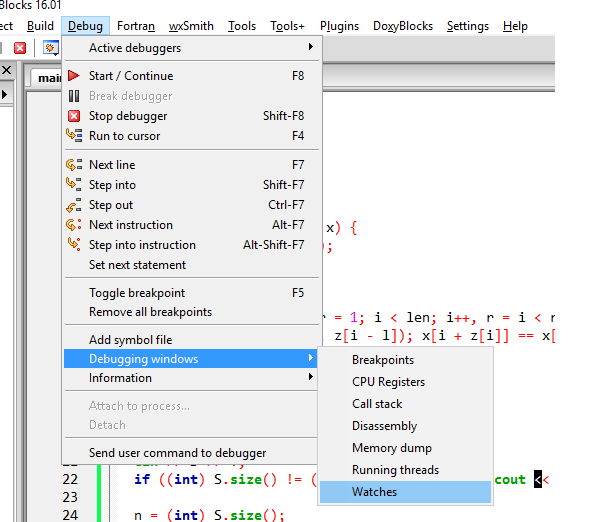
Hoặc sử dụng các phím tắt thường dùng như đã được mô tả trong option **Debug:**

* **F8**: Bắt đầu hoặc tiếp tục việc debug còn dang dở.
* **Shift + F8:** Dừng việc debug.
* **F4:** Chuyển đến dòng đang chứa con trỏ.
* **F7:** Đến dòng chứa lệnh tiếp theo.
* **Shift + F7:** Vào thân hàm xuất hiện tại dòng hiện tại.
* **Ctrl + F7:** Thoát khỏi hàm.
* **F5:** Đặt hoặc xóa bỏ breakpoints.

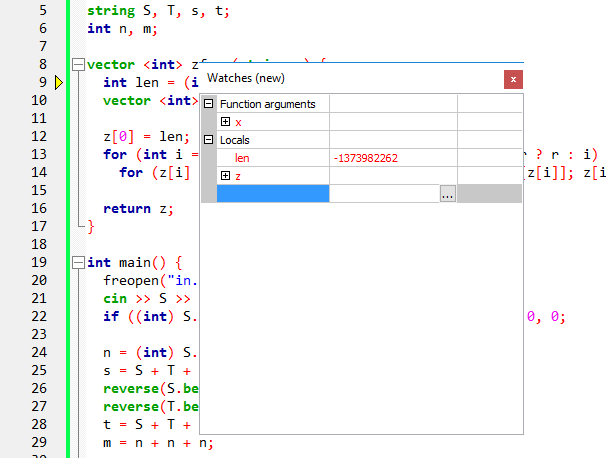
***b. Những cách debug thường dùng***

**Watches windows:** Đây là một cửa sổ cho phép ta xem giá trị của biến hoặc thậm chí là kết quả của một số biểu thức với biến. Để mở **watches windows** ta phải đảm bảo debugger đã được chạy và thao tác:

**Debug → Debugging windows → watches**

****

Watches windows sẽ hiện ra:

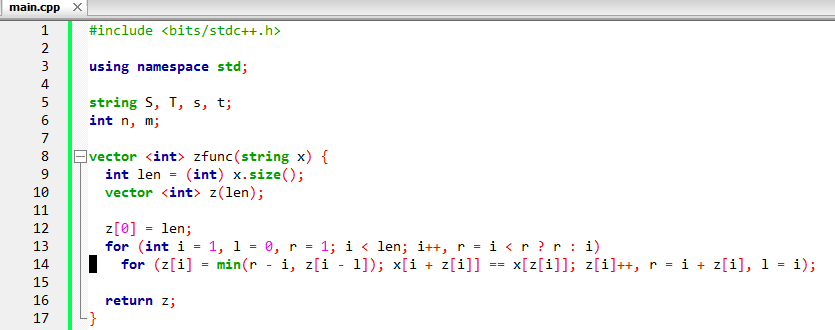


Bảng này sẽ bao gồm:

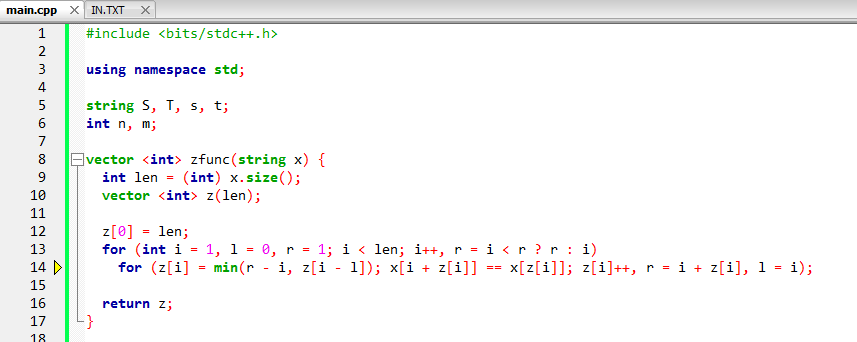
* Function arguments: Danh sách các tham số của hàm chứa con trỏ debugger.
* Locals: Chứa danh sách biến cục bộ trong hàm chứa con trỏ debugger.

Ta có thể gõ bất kì biến hợp lệ nào vào ô màu xanh để debugger hiện ra giá trị của biến, hoặc thậm chí là một biểu thức hợp lệ.

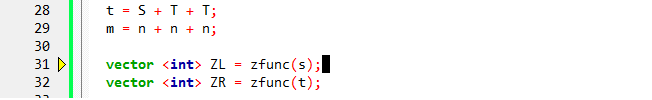
Dưới đây là chi tiết cách thức hoạt động của một số phương thức thông dụng.

* **F8/Shift + F8:** Đây chỉ đơn giản là bắt đầu hoặc dừng việc debug.
* **F4 (Run to cursor):** Dịch nôm na là chuyển đến dòng chứa con trỏ hiện tại. Ví dụ: 

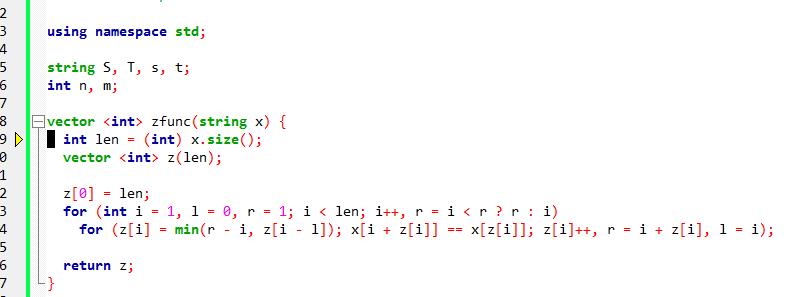
Ta nghi ngờ dòng số 14 là nguyên nhân gây ra lỗi. Cho nên ta đặt con trỏ chuột ở dòng thứ 14. Sau đó nhấn **F4** thì ta sẽ được con trỏ debug sẽ dừng lại ở dòng 14.



* **Shift + F7** và **F7:** Có thể nói đây là hai tổ hợp phím dùng nhiều nhất khi debug. Để bắt đầu debug ta nhấn tổ hợp phím **Shift + F7** chức năng của nó là (Step into) và vì main là một hàm cho nên nó sẽ dừng lại ở ngay dòng đầu tiên của hàm main để kết thúc việt debug ta có thể nhấn **F8** hoặc **Shift + F8**. Để dịch chuyển đến dòng lệnh tiếp theo thì nhấn **F7.** Còn nếu muốn vào một hàm nào đó thì phải nhấn **Shift + F7** nếu không debugger sẽ tự động bỏ qua việc debug bên trong hàm đó. Ví dụ tại dòng 31, muốn xem hàm zfunc(s) hoạt động như thế nào thì ta phải nhấn **Shift + F7.**

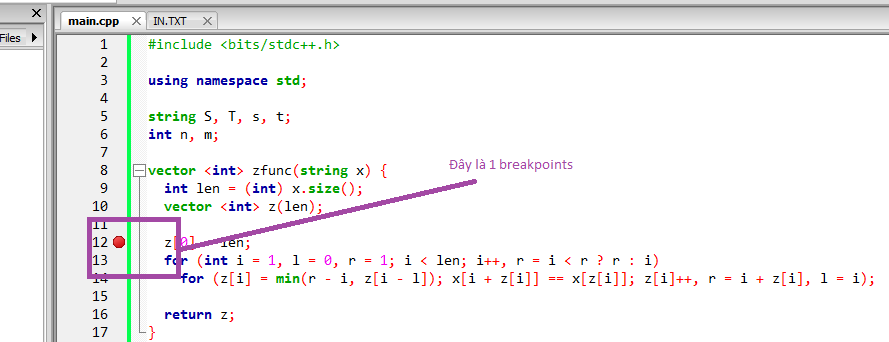


Khi đó, con trỏ debugger sẽ dừng lại ở dòng đầu tiên bên trong hàm zfunc(s):



Ngược lại, tại một vị trí bất kì trong hàm thì để thoát ra khỏi hàm đó ta chỉ cần nhấn **Ctrl + F7**.

***c. Breakpoints***

Breakpoints là vị trí mà chương trình sẽ dừng lại để người lập trình xem xét sự thay đổi của các biến qua từng dòng lệnh, từ đó phát hiện ra vị trí dòng code bị lỗi. Breakpoint được kí hiệu bằng chấm tròn màu đỏ ở đầu dòng code. Để tạo ra một breakpoint, cách đơn giản nhất là click chuột vào đầu dòng code (như trong hình). Để huỷ breakpoint, chỉ cần click chuột vào breakpoint đó một lần nữa. Ngoài ra cũng có thể tạo/huỷ breakpoint bằng phím F5 

Công dụng của break point là khi ta nhấn phím **F8** để debug toàn bộ chương trình thì chương trình sẽ dùng lại ở vị trí được đặt breakpoints, giúp chúng ta có thể nhanh chóng tìm ra lỗi hơn là **F7** từng dòng code.

**§2: STANDARD TEMPLATE LIBRARY (STL)**

Có thể nói STL chính là một trong những lí do chính giúp C++ đang dần thay thế pascal trong các cuộc thi lập trình ở Việt Nam. Việc sử dụng thành thạo STL sẽ là rất quan trọng nếu chúng ta có ý định tham gia các kì thi về Tin Học. Như thầy Lê Minh Hoàng đã từng nói “STL sẽ nối dài khả năng lập trình của các bạn”. Đúng như vậy, STL là một kho thư viện khổng lồ chứa các công cụ giúp học sinh giải được bài toán một cách dễ dàng hơn rất nhiều. Tuy nhiên, khi học STL ta cũng phải chắc rằng mình đã nắm vững những thuật toán được sử dụng trong STL, bời vì trong nhiều trường hợp việc sử dụng STL sẽ trở thành con dao hai lưỡi nếu ta lạm dụng quá nhiều, do STL bao gồm các template được quy định bởi ngôn ngữ lập trình nên việc debug, gỡ rối là không hề dễ dàng và chúng rất khó kiểm soát. Trong bài bày ta sẽ tìm hiểu một số STL hay dùng nhất trong C++.

**1. STL là gì ?**

Standard Tamplate Library (STL) trong C++ là một tập hợp các lớp Template mạnh mẽ trong C++ để cung cấp các lớp và các hàm được tạo theo khuôn mẫu cho mục đích lập trình tổng quát, mà triển khai nhiều thuật toán và cấu trúc dữ liệu được sử dụng phổ biến và thông dụng như vector, list, queue và stack, ….

Thư viện chuẩn C++ (C++ Standard Library) có thể chia thành hai phần chính:

* Thư viện hàm chuẩn (Standard Function Library): Thư viện này gồm các hàm mang mục đích tổng quát mà không là một phần của bất kì lớp nào (class). SFL mang tính kế thừa từ C. Ví dụ: I/O, xử lí chuỗi kí tự, toán học, Date, Time, Place, cáp phát động, hỗn hợp, các hàm cho wide-character, ..
* Thư viện lớp hướng đối tượng (Object Oriented Class Library): Đây là một tập hợp của các lớp và các hàm liên kết. Ví dụ: các lớp I/O chuẩn C++, string, numberic, stl container, stl algorithm, các đôi tượng thuộc stl function, stl iterator, stl allocator, localization, các lớp exception handling, thư viện hàm hỗn hợp, …

Tuy nhiên, ba cấu trúc thực sự đã tạo ra sự mạnh mẽ và khác biệt cho C++ đó chính là Containers, Algorihtms và Iterators.

**2. Một số lưu ý khi dùng STL**

Khi sử dụng một STL nào đó, ta buộc phải khai báo bằng cú pháp sau ở đầu chương trình:

**#include <tên thư viện>;**

Ví dụ, nếu muốn sử dụng thư viện vector ta phải khai báo:

**#include <vector>;**

Ta nên khai báo namespace std, vì đa số các thư viện dùng chung namespace này, ta khai báo như sau:

**using namespace std;**

Ngoài ra, khi khai báo một đối tượng lưu trữ với kiểu dữ liệu nào đó thì ta phải khai báo theo cú pháp tương tự như khi báo một vector với kiểu int như sau:

**vector <int> a;**

Hãy đặt biệt chú ý đến cặp dấu “<>”, khi khai báo một cấu trúc đối tượng trong đối tượng thì những cặp dấu “<>” không được viết liền nhau để tránh nhầm lẫn với toán tử trích luồng. Ví dụ

**vector < vector<int> > N; // Khai báo đúng**

**vector <vector<int>> N; // Khai báo sai**

Tuy nhiên, kể từ phiên bản C++ 11 thì đều phiền toái này không xảy ra nữa.

Để truy cập hàm hoặc phương thức thành viên của một STL, ta có thể thực hiện tương tự như ví dụ dưới đây.

**vector <int> v;**

**v.clear(); // Phương thức này cho phép xóa toàn bộ phần tử của vector v**

**3. Iterator**

Trong C++, một biến lặp là một đối tượng bất kì, trỏ tới một số phần tử trong 1 phạm vi của các phần tử (như mảng hoặc container), có khả năng để lặp các phần tử trong phạm vi bằng cách sử dụng một tập các toán tử (operators) (như so sánh, tăng (++),...). Dạng rõ ràng nhất của iterator là một con trỏ: Một con trỏ có thể trỏ tới các phần tử trong mảng, và có thể lặp thông qua sử dụng toán tử tăng (++). Tuy nhiên, cũng có các dạng khác của iterator. Ví dụ: mỗi loại container (chẳng hạn như vector) có một loại iterator được thiết kế để lặp các phần tử của nó một cách hiệu quả.

Để khai báo iterator, ta dùng cú pháp:

|  |
| --- |
| **<containers>::iterator it;**  **<containers>::reverse\_iterator rit; // Đối với iterator ngược** |

Chẳng hạn:

|  |
| --- |
| **vector<int>::iterator it;**  **vecotr<int>::reverse\_iterator rit;** |

Ta có thể thực hiện một số thao tác với iterator như:

- So sánh: “==” và “!=”.

- Gán “=”.

- Cộng trừ.

- Lấy giá trị “\*”.

Ta có một vài iterator hay dùng trong các containers như.

- begin(): Trả về iterator đầu tiên của containers.

- end(): Trả về iterator cuối cùng của containers.

- rbegin(): Trả về iterator đầu tiên của vector theo chiều ngược.

- rend(): Trả về iterator cuối cùng của vector theo chiều ngược.

**4. Containers**

Containers thực chất là một nhóm các thư viện cung cấp “dịch vụ” lưu trữ các đối tượng cụ thể. Ở mỗi containers sẽ cho phép thực hiện một số thao tác nhất định với đối tượng được containers lưu trữ. Các containers có thể chia thành những nhóm sau:

* **Sequence containers:** Vector, list, deque, …
* **Associative containers:** Set, multiset, map, multimap, …
* **Containers adapters:** Stack, queue, priority\_queue, …
* **String:** String, rope, …
* **Bitset**

***a. Vector***

Vector thật chất là một cấu trúc mảng động, người lập trình không cần quy định kích thước của vector như một mảng thông thường. Vector sẽ rất hữu ích khi cài đặt những mảng có số lượng phần tử không biết trước, tuy nhiên chúng ta vẫn có thể quy định kích thước của vector. Ví dụ

|  |
| --- |
| **vector <int> v; // Khai báo vector khi chưa biết số lượng phần tử**  **vector <int> v(10); // Khai báo vector với 10 phần tử rỗng**  **vector <int> v(10, 1); // Khai báo vector với 10 phần tử có giá trị 1** |

Lưu ý là cũng như mảng, index của vector cũng bắt đầu từ 0.

Dưới đây là một số hàm thành viên của STL vector:

* empty(): Trả về giá trị true nếu vector rỗng, ngược lại là giá trị false.
* size(): Trả về giá trị là số lượng phần tử của vector.
* resize(n): Xóa phần tử dư hoặc thêm một số phần tử để số lượng phần tử của vector đúng bằng n.
* at(i): Trả về giá trị tại vị trí i của vector, v.at(i) tương đương với v[i].
* front(): Trả về giá trị đầu tiên của vector từ trái sang phải. v.font() = v[0].
* push\_back(value): Thêm phần tử value vào cuối vector.
* pop\_back(): Xóa phần tử cuối của vector.
* assign(n, t): Khởi tạo n phần tử đầu tiên của vector với giá trị bằng t.
* swap(v2): Hoán đổi với vector v2.
* erase(v): Xóa phần tử tại vị trí v (v là một iterator). Hoặc có thể dùng erase(v1, v2) để xóa phần tử từ vị trí v1 đến v2 của vector (v1, v2 là iterator). Thậm chí là xóa toàn bộ vector với erase().
* insert(i, T): Chèn phần tử T tại iterator i. Tương tự như erase ta cũng có insert(i, n, T) = chèn n phần tử T bắt đầu từ iterator i. Hoặc insert(i, v1, v2) tức là chèn vào vị trí iterator i những phần tử từ iterator v1 đến iterator v2 (chú ý v1, v2 có thể là iterator của một vector khác).

Ví dụ:

|  |
| --- |
| **#include <iostream>**  **#include <vector>**  **using namespace std;**  **void view(vector <int> vec) {**  **for (vector<int>::iterator it = vec.begin(); it != vec.end(); it++)**  **cout << \*it << " ";**  **cout << "\n";**  **}**  **int main() {**  **vector <int> v(10, 1);**  **view(v); // 1 1 1 1 1 1 1 1 1 1 1**  **cout << v.size() << "\n"; // 10**  **cout << v.front() << "\n"; // 1**  **for (int i = 0; i < v.size(); i++)**  **v[i] = i;**  **v.erase(v.begin() + 5);**  **view(v); // 0 1 2 3 4 6 7 8 9**  **v.erase(v.begin() + 5, v.end());**  **view(v); // 0 1 2 3 4**  **vector <int> V;**  **for (int i = 1; i <= 10; i++)**  **V.push\_back((i + 1) \* (i - 1));**  **V.swap(v);**  **view(V); // 0 1 2 3 4**  **V.swap(v);**  **view(V); // 0 3 8 15 24 35 48 63 80 99**  **vector<int>::iterator it = v.begin();**  **V.insert(V.begin() + 5, it, it + 3);**  **view(V); // 0 3 8 15 24 0 1 2 35 48 63 80 99**  **V.pop\_back();**  **view(V); // 0 3 8 15 24 0 1 2 35 48 63 80**  **V.resize(10);**  **view(V); // 0 3 8 15 24 0 1 2 35 48**  **V.assign(10, 0);**  **view(V); // 0 0 0 0 0 0 0 0 0 0**  **return 0;**  **}** |

***b. List***

List thực chất là cấu trúc dữ liệu Double-Linked List, trước đây chúng ta thường phải cài đặt bằng con trỏ và làm việc với nó khá nhọc nhằn. Tuy nhiên, C++ đã cung cấp hẳn một thư viện hỗ trợ cho cấu trúc dữ liệu này. Nếu chưa biết về cách hoạt động của Double-Linked List, các bạn có thể tìm hiểu ở một tài liệu khác. Ở đây chúng ta chỉ nói về cách sử dụng STL list trong C++.

Cách khai báo một list: **list<***kiểu dữ liệu***>** *tên biến***;**

Ví dụ: **list<int> mylist;**

Do cùng là sequence containers cho nên list cũng có một số phương thức giống như vector và hoạt động tương tự: front(), back(), push\_back(), pop\_back(), assign(), size(), empty(), swap(), erase(), resize(), size(), …

Ngoài ra, ta cũng có một số phương thức mà list có còn vector thì không:

- push\_front(value): Thêm phần tử value vào đầu list.

- pop\_front(): Xóa phần tử đầu của list.

- sort(comparision\_function): Hàm sắp xếp list theo thứ tự được quy định bởi compare\_function (Cách viết compare function sẽ được đề cập ở một bài khác). Mặt định nếu không có comparision\_function hàm sort sẽ sắp xếp list theo thứ tự tăng dần.

- merge(s, comparison\_function): Trộn list hiện tại với list s. Sau khi trộn ta sẽ được một list đã được sắp xếp. Lưu ý bản chất hàm này dựa trên thuật toán sắp xệp trộn (merge sort) cho nên hai list trước khi trộn phải là list đã được sắp xếp.

Ví dụ:

|  |
| --- |
| **#include <iostream>**  **#include <list>**  **using namespace std;**  **void view(list<int> L) {**  **for (list<int>::iterator it = L.begin(); it != L.end(); it++)**  **cout << \*it << " ";**  **cout << "\n";**  **}**  **bool comparision(const int &first, const int &second) {**  **return first > second;**  **}**  **int main() {**  **list<int> myList;**  **for (int i = 1; i <= 10; i++)**  **myList.push\_back(i);**  **view(myList); // 1 2 3 4 5 6 7 8 9 10**  **myList.push\_front(100);**  **myList.pop\_back();**  **view(myList); // 100 1 2 3 4 5 6 7 8 9**  **list<string> L;**  **L.push\_back("one");**  **L.push\_back("two");**  **L.push\_back("three");**  **L.sort();**  **list<string>::iterator it;**  **for (it = L.begin(); it != L.end(); it++)**  **cout << \*it << " ";**  **cout << "\n"; // one three two**  **list<int> L2;**  **for (int i = 1; i <= 10; i++)**  **L2.push\_front(i);**  **view(L2);**  **myList.sort();**  **view(myList); // 1 2 3 4 5 6 7 8 9 100**  **myList.sort(comparision);**  **view(myList); // 100 9 8 7 6 5 4 3 2 1**  **myList.merge(L2, comparision);**  **view(myList); // 100 10 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1**  **return 0;**  **}** |

***c. Pair***

Đôi khi trong lúc lập trình, ta cần lưu giữ một đối tượng với một cặp giá trị. Khi đó ta thường sẽ nghĩ đến struct. Ví dụ:

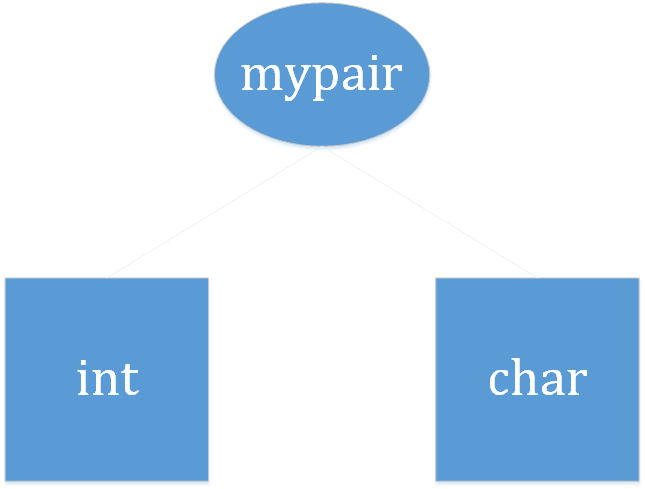
|  |
| --- |
| **struct p {**  **int a;**  **char b;**  **};** |

Tuy nhiên, có một cách khác đó là dùng pair. Pair cho phép ta gộp 2 đối tượng thành một cặp, có thể 2 đối tượng cùng kiểu hoặc khác kiểu, với thuộc tính "first" là đối tượng đầu tiên, "second" là đối tượng thứ 2 trong cặp đối tượng "position/ data". Để khai báo pair ta dùng cú pháp:

**pair <***kiểu giá trị 1***,** *kiểu giá trị 2***>** *tên đối tượng***;**

Ví dụ: **pair <int, char> mypair;**

Khi đó mypair sẽ có cấu trúc:



Để lấy dữ liệu trong pair, ta dùng hai thuộc tính là first và second. Để tạo một pair với một cặp dữ liệu có sẵn ta dùng constructor make\_pair(x, y), trong đó x là giá trị của thuộc tính first, y là giá trị của thuộc tính second. Ví dụ:

|  |
| --- |
| **pair<int, char> p = make\_pair(2, ‘D’);**  **cout << p.first << “ “ << p.second; // 2 D** |

Hoặc ta cũng có thể khởi tạo pair bằng cú pháp:

|  |
| --- |
| **pair<int, char> p(2, ‘D’);** |

Ví dụ về sử dụng pair:

|  |
| --- |
| **#include <cstdio>**  **#include <iostream>**  **#include <vector>**  **using namespace std;**  **int main() {**  **pair<int, string> p;**  **p = make\_pair(3, "NHTN");**  **cout << p.first << " " << p.second << "\n"; // 2 NHTN**  **vector< pair<int, int> > p2;**  **for (int i = 1; i <= 10; i++)**  **p2.push\_back(make\_pair(i, 10 - i));**  **vector < pair<int, int> >::iterator it;**  **for (it = p2.begin(); it != p2.end(); it++)**  **cout << (\*it).first << " " << (\*it).second << "\n";**  **/\***  **1 9**  **2 8**  **3 7**  **4 6**  **5 5**  **6 4**  **7 3**  **8 2**  **9 1**  **10 0**  **\*/**  **return 0;**  **}** |

***d. Map***

Map thuộc loại Associative containers (Cấu trúc liên kết) được thiết kế như là một cây đỏ-đen tiểu chuẩn (red-black tree). Map được tổ chức theo kiểu key-value tức là một key sẽ được ánh xạ với một value. Các key sẽ được sắp xếp theo một thứ tự do hàm Compare quy định. Các thao tác chèn, xóa, tìm kiếm, … với map được thực hiện với độ phức tạp logarit. Cú pháp khai báo:

**map<***kiểu dữ liệu***,** *kiểu dữ liệu***>** *tên biến***;**

Ví dụ:

|  |
| --- |
| **map<string, int> m;**  **m[“duy”] = 1;**  **m[“crush”] = 100000;** |

Trong ví dụ trên, ta đã khai báo map m với key là kiểu string và value thuộc kiểu int. Ta có thể duyệt qua các phần tử của map bằng iterator, và các hàm thành viên cũng hoạt động giống như vector và list. Ví dụ với hàm insert:

|  |
| --- |
| **map<int, char\*> mymap, copymymap;**  **mymap[0] = "a";**  **mymap[1] = "b";**  **mymap[5] = "c";**    **// chèn vào copymymap cặp đối tượng (10, "c")**  **copymymap.insert(pair<int, char\*>(10, "c"));**  **// chèn (-1, "d") vào copymymap từ vị trí bắt đầu của copymymap**  **copymymap.insert(copymymap.begin(), pair<int, char\*>(-1, "d"));**  **// chèn mymap vào copymymap**  **copymymap.insert(mymap.begin(), mymap.end());**  **// => copymymap = {(-1,"d"),(0,"a"),(1,"b"),(5,"c"),(10,"c")}** |

Tuy nhiên, thật ra việc chèn ở một vị trí nào đó là vô nghĩa, Vì các key trong map đều được sắp xếp theo một quy tắc được quy định sẳn. Như ở trên key là các số nguyên nên chúng được sắp xếp tăng dần, cho nên dù không cần xác định vị trí iterator cần chèn thì vị trí của value được chèn vào vẫn được chương trình tự xác định.

Hàm thành viên erase() cho phép ta xóa một đối tượng hay hàm clear() cho phép ta xóa tất cả đối tượng trong "map", hàm erase() được override với các chức năng trong ví dụ sau:

|  |
| --- |
| **map<int, char\*> mymap, copymymap;**  **mymap[0] = "a";**  **mymap[1] = "b";**  **mymap[5] = "c";**  **mymap[7] = "d";**  **mymap[9] = "e";**    **// xóa cặp đối tượng với "position" là 5**  **mymap.erase(5);**  **// => mymap = {(0,"a"),(1,"b"),(7,"d"),(9,"e")}**    **map<int, char\*>::iterator var = mymap.begin();**  **// xóa cặp đối tượng mà var đang truy cập**  **mymap.erase(var); // => mymap = {(1,"b"),(7,"d"),(9,"e")}**    **var = mymap.find(7); // => var truy cập đến (7,"d")**    **// xóa từ vị trí var đang truy cập cho đến (9,"e")**  **mymap.erase(var, mymap.end()); // => mymap = {(1,"b")}** |

Hàm thành viên find() cho phép ta tìm kiếm theo "position" của cặp giá trị "position/data"

|  |
| --- |
| **map<int, char\*> mymap;**  **mymap[0] = "a";**  **mymap[1] = "b";**  **mymap[5] = "c";**  **mymap[9] = "e";**  **map<int, char\*>::iterator var = mymap.find(5);**  **// var -> (5,"c")** |

***d. Set***

Set là một cấu trúc dữ liệu liên kết lưu trữ một danh sách các đối tượng được sắp xếp theo thứ tự được quy định bởi hàm compare tiêu chuẩn hoặc do người lập trình quy định. Ứng dụng mạnh nhất của set là thêm vào set một đối tượng nào đó hoặc kiểm tra đối tượng đó có xuất hiện trong set hay chưa với độ phức tạp logarit.

Một số hàm hữu ích:

* insert(object): Chèn đối tượng object vào set.
* erase(v): Xóa đối tượng khỏi erase (v ở đây có thể là object, iterator). Ví dụ:

|  |
| --- |
| **// erasing from set**  **#include <iostream>**  **#include <set>**  **int main ()**  **{**  **std::set<int> myset;**  **std::set<int>::iterator it;**  **// insert some values:**  **for (int i=1; i<10; i++) myset.insert(i\*10); // 10 20 30 40 50 60 70 80 90**  **it = myset.begin();**  **++it; // "it" points now to 20**  **myset.erase (it);**  **myset.erase (40);**  **it = myset.find (60);**  **myset.erase (it, myset.end());**  **std::cout << "myset contains:";**  **for (it=myset.begin(); it!=myset.end(); ++it)**  **std::cout << ' ' << \*it;**  **std::cout << '\n';**  **return 0;**  **}** |

* find(value): Trả về iterator chứa value. Ví dụ:

|  |
| --- |
| **// set::find**  **#include <iostream>**  **#include <set>**  **int main ()**  **{**  **std::set<int> myset;**  **std::set<int>::iterator it;**  **// set some initial values:**  **for (int i=1; i<=5; i++) myset.insert(i\*10); // set: 10 20 30 40 50**  **it=myset.find(20);**  **myset.erase (it);**  **myset.erase (myset.find(40));**  **std::cout << "myset contains:";**  **for (it=myset.begin(); it!=myset.end(); ++it)**  **std::cout << ' ' << \*it;**  **std::cout << '\n';**  **return 0;**  **}** |

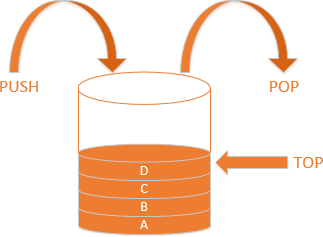
* count(value): Trả về số lượng phần tử bằng với value. Ví dụ:

|  |
| --- |
| **// set::count**  **#include <iostream>**  **#include <set>**  **int main ()**  **{**  **std::set<int> myset;**  **// set some initial values:**  **for (int i=1; i<5; ++i) myset.insert(i\*3); // set: 3 6 9 12**  **for (int i=0; i<10; ++i)**  **{**  **std::cout << i;**  **if (myset.count(i)!=0)**  **std::cout << " is an element of myset.\n";**  **else**  **std::cout << " is not an element of myset.\n";**  **}**  **return 0;**  **}** |

***e. Stack***

Stack và queue là hai cấu trúc dữ liệu quan trọng trong lập trình. Trong C++, hai cấu trúc dữ liệu này cũng được cung cấp với khá đầy đủ những chức năng cần thiết. Là một.

Stack là một kiểu cấu trúc dữ liệu và cơ chế của nó là LIFO (Last In First Out) nghĩa là vào sau ra trước. Ta có thể hình dung Stack như một chồng đĩa ,ta chỉ có thể lấy chiếc đĩa ra hoặc thêm một chiếc đĩa khác vào trên đỉnh của nó và chiêc đĩa nằm trên đỉnh đó được gọi là Top.



Các phương thức quan trọng của stack là:

* empty(): Dùng để kiểm ra stack có rỗng hay không. Nếu stack rỗng trả về true và ngược lại trả về false.
* push(value): Thêm một phần tử value và đỉnh stack.
* top(): Trả về giá trị của phần tử ở đỉnh stack.
* pop(): Lấy phần tử ở đỉnh stack ra ngoài (xóa).
* clear(): Xóa toàn bộ stack.

Ví dụ về cách dùng stack:

|  |
| --- |
| **#include <iostream>**  **#include <stack>**  **using namespace std;**  **int main() {**  **stack <int> st;**  **for (int i = 1; i <= 5; i++)**  **st.push(i);**  **// st = 1 2 3 4 5**  **cout << st.top() << "\n"; // 5**  **st.pop();**  **cout << st.top() << "\n"; // 4**  **while (!st.empty()) {**  **cout << st.top() << " ";**  **st.pop();**  **}**  **// 4 3 2 1**  **}** |

***f. Queue***

Tương tự như stack, nhưng queue tổ chức theo cấu trúc FIFO (First In First Out), đó gọi là cấu trúc hàng đợi. Các phương thức quan trọng trong queue là:

* empty(): Giống như stack hàm này cũng dùng để kiểm tra liệu queue có rỗng hay không.
* push(value): Đẩy giá trị value vào hàng đợi.
* pop(): Lấy giá trị ra khỏi hàng đợi.
* front(): Lấy giá trị đang chờ ở đỉnh trong hàng đợi.

Ví dụ:

|  |
| --- |
| **// queue::push/pop**  **#include <iostream> // std::cin, std::cout**  **#include <queue> // std::queue**  **int main ()**  **{**  **std::queue<int> myqueue;**  **int myint;**  **std::cout << "Please enter some integers (enter 0 to end):\n";**  **do {**  **std::cin >> myint;**  **myqueue.push (myint);**  **} while (myint);**  **std::cout << "myqueue contains: ";**  **while (!myqueue.empty())**  **{**  **std::cout << ' ' << myqueue.front();**  **myqueue.pop();**  **}**  **std::cout << '\n';**  **return 0;**  **}** |

***g. Priority queue (Hàng đợi ưu tiên)***

Priority queue là một loại container adaptor, được thiết kế đặc biệt để phần tử ở đầu luôn luôn lớn nhất (theo một quy ước về độ ưu tiên nào đó) so với các phần tử khác. Priority Queue rất giống với heap, cho nên nó thường được dùng thay cho heap. Phép toán mặc định là less, có nghĩa là phần tử lớn nhất sẽ ở đầu. Tuy nhiên để linh hoạt ta có thể viết hàm so sánh (ở bài sau) để chủ động trong việc sử dụng priority queue. Khai báo:

|  |
| --- |
| **priority\_queue <int> pq; // Dùng phép toán mặc định less**  **priority\_queue <int, vector<int>, greater<int> > pq; // Phép toán greater** |

Các phương thức quan trọng:

* empty(): Kiểm tra pq có rỗng hay không.
* push(value): Thêm phần tử value vào pq.
* pop(): Loại bỏ phần tử ở đỉnh pq (phần tử có ưu tiên cao nhất).
* top(): Trả về phần tử ở đỉnh pq (phần tử có ưu tiên cao nhất).

Ví dụ:

|  |
| --- |
| **// priority\_queue::push/pop**  **#include <iostream> // std::cout**  **#include <queue> // std::priority\_queue**  **int main ()**  **{**  **std::priority\_queue<int> mypq;**  **mypq.push(30);**  **mypq.push(100);**  **mypq.push(25);**  **mypq.push(40);**  **std::cout << "Popping out elements...";**  **while (!mypq.empty())**  **{**  **std::cout << ' ' << mypq.top();**  **mypq.pop();**  **}**  **std::cout << '\n';**  **return 0;**  **}** |

**5. STL Algorithm (Thư viện thuật toán).**

Thư viện này chứa khá nhiều thuật toán hay và quan trọng. Ta sẽ chỉ xem xét ngắn gọn một số thuật toán hữu ích trong việc lập trình giải các bài toán trong thực tế.

***a. Min/Max***

Min/Max sẽ trả về giá trị bé/lớn hơn khi so sánh hai đối tượng cùng kiểu. ví dụ:

min(‘a’, ‘b’) = ‘a’;

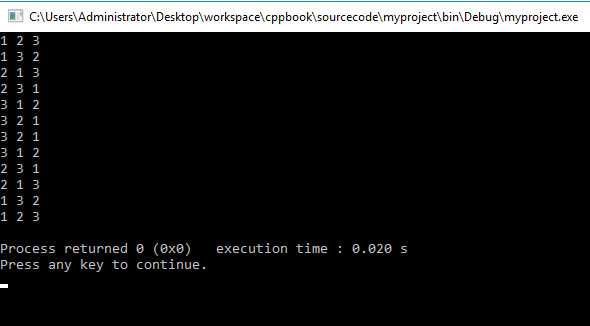
min(2, 3) = 2

***b. prev\_permutation/next\_permutation***

next\_permutation trả về kết quả là true nếu có hoán vị tiếp theo, ngược lại thì kết quả là false. Hàm prev\_permutation thì trả về true nếu hoán vị đang xét không phải là hoán vị đầu tiên của dãy các hoán vị được xếp theo thứ tự từ điển. Ví dụ:

|  |
| --- |
| **#include <iostream>**  **#include <cstdio>**  **#include <algorithm>**  **using namespace std;**  **int main() {**  **int myint[] = {1, 2, 3};**  **do {**  **cout << myint[0] << " " << myint[1] << " " << myint[2] << "\n";**  **} while (next\_permutation(myint, myint + 3));**  **vector <int> a;**  **a.push\_back(3);**  **a.push\_back(2);**  **a.push\_back(1);**  **do {**  **cout << a[0] << " " << a[1] << " " << a[2] << "\n";**  **} while (prev\_permutation(a.begin(), a.end()));**  **}** |

Output:



***c. sort***

Hàm sort cho phép ta sắp xếp các phần tử theo một thứ tự ưu tiên nào đó với độ phức tạp là . Ví dụ:

|  |
| --- |
| **#include <iostream>**  **#include <cstdio>**  **#include <algorithm>**  **using namespace std;**  **int main() {**  **int myint[5] = {1, 3, 2, 5, 4};**  **// sử dụng toán tử mặc định less**  **// => sắp xếp tăng dần**  **sort(myint, myint + 5);**  **for (int i = 0; i < 5; i++)**  **cout << myint[i] << " "; // 1 2 3 4 5**  **cout << "\n";**  **// sư dụng toán tử greater**  **// => sắp xếp giảm dần**  **sort(myint, myint + 5, greater<int>());**  **for (int i = 0; i < 5; i++)**  **cout << myint[i] << " "; // 5 4 3 2 1**  **cout << "\n";**  **}** |

***d. binary\_search***

Tìm kiếm nhị phân xem khóa có trong đoạn cần tìm không với độ phức tạp . Đoạn tìm kiếm phải là đoạn có thứ tự (đã sắp xếp). Nếu có thì trả về true, ngược lại là false. Ví dụ:

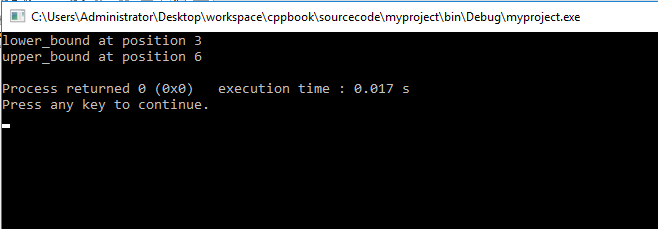
|  |
| --- |
| **// binary\_search example**  **#include <iostream>**  **#include <algorithm>**  **#include <vector>**  **using namespace std;**  **bool myfunction (int i,int j) { return (i<j); }**  **int main () {**  **int myints[] = {1,2,3,4,5,4,3,2,1};**  **vector<int> v(myints,myints+9); // 1 2 3 4 5 4 3 2 1**    **// Sử dụng toán tử so sánh mặc định**  **sort (v.begin(), v.end());**    **cout << "looking for a 3... ";**    **if (binary\_search (v.begin(), v.end(), 3))**  **cout << "found!\n"; else cout << "not found.\n";**    **// sử dụng hàm so sánh tự định nghĩa:**  **sort (v.begin(), v.end(), myfunction);**    **cout << "looking for a 6... ";**    **if (binary\_search (v.begin(), v.end(), 6, myfunction))**  **cout << "found!\n"; else cout << "not found.\n";**    **return 0;**  **}** |

***e. lower\_bound và upper\_bound***

Hàm lower\_bound và upper\_bound tương đối giống nhau. Hàm lower\_bound sẽ trả về phần tử đầu tiên lớn hơn hoặc bằng khóa cho trước. Còn upper\_bound thì trả về phần tử đầu tiên lớn hơn hẳn khóa cho trước. Cả hai hàm này đều có độ phức tạp . Ví dụ:

|  |
| --- |
| **// lower\_bound/upper\_bound example**  **#include <iostream>**  **#include <algorithm>**  **#include <vector>**  **using namespace std;**  **int main () {**  **int myints[] = {10,20,30,30,20,10,10,20};**  **vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20**  **vector<int>::iterator low,up;**  **sort (v.begin(), v.end()); // 10 10 10 20 20 20 30 30**  **low = lower\_bound (v.begin(), v.end(), 20); // ^**  **up = upper\_bound (v.begin(), v.end(), 20); // ^**  **cout << "lower\_bound at position " << int(low- v.begin()) << endl;**  **cout << "upper\_bound at position " << int(up - v.begin()) << endl;**  **return 0;**  **}** |

Output:



**6.Kết**

STL là một thư viện cực kì lớn, được xây dựng trong một thời gian dài, do đó việc nắm vũng STL trong ngày một, ngày hai là đều không thể. Cho nên ở trên đây, tôi đã nêu ra những cái cơ bản và ngắn gọn nhất của STL, phần còn lại là do người học tự tìm tòi và nghiên cứu, kiến thức sẽ được tích lũy trong một thời gian dài.

**§3: COMPARISION FUNCTION**

Ở bài trước ta đã xét đến một số hàm thường dùng trong STL có dùng đến thứ tự ưu tiên như là sort, lower\_bound, upper\_bound, … Tất cả những hàm đó xác định tính ưu tiên dựa trên một hàm so sánh mặc định, tuy vậy trong nhiều trường hợp ta không thể hoặc không muốn dùng hàm so sánh mặc định thì ta vẫn có thể viết cho mình một hàm so sánh riêng. Ở bài này, ta sẽ tìm hiểu những cách viết các hàm đó.

**1. Định nghĩa toán tử “bé hơn” (less operator)**

Toán tử bé hơn “<” trong C++ đã được mặc định sẳn với những kiểu dữ liệu và đối tượng (object) tiêu chuẩn. Tuy nhiên, khi chúng ta sử dụng các đối tượng tự định nghĩa như struct, class, … thì toán tử “<” không hề có sẳn trong trường hợp này. Do đó ta phải định nghĩa lại toán tử “<” cho đối tượng mà ta đã tạo ra. Ví dụ ta có một struct biểu diễn một cạnh trong đồ thị như sau:

|  |
| --- |
| **struct Edge {**  **int from, to, weight;**  **};** |

Giả sử vấn đề là ta có danh sách cạnh và muốn sắp xếp chúng giảm dần để thực hiện thuật toán Kruskal. Và như đã biết thuật toán sort trong thư viện algorithm có chiều sắp xếp mặc định là tăng dần, tức là sử dụng hàm less trong thư viện functional. Hàm này có dạng f(x, y) = true nếu x < y, ngược lại thì false. Do đó với đối tượng Edge ta phải định nghĩa một cạnh như thế nào gọi là bé hơn cạnh kia. Ta có thể làm như sau:

|  |
| --- |
| **struct Edge {**  **int from, to, weight;**  **bool operator < (Edge other) const {**  **return weight < other.weight;**  **}**  **}** |

Khi đó, chương trình sẽ hiểu khi so sánh với hai đối tượng Edge a và b: a < b = true nếu a.weight < b.weight. Sau đó chỉ cần viết hàm sort một cách rất bình thường như khi sắp xếp tăng dần một dãy số. Ví dụ:

|  |
| --- |
| **struct Edge {**  **int from, to, weight;**  **bool operator < (Edge other) const {**  **return weight < other.weight;**  **}**  **} e[N];**  **…**  **sort(e, e + N);** |

Thật ra cách định nghĩa toán tử này là một cách tổng quát, ta có thể định nghĩa các toán tử khác với ý nghĩa theo ý muốn của chúng ta như: +, -, \*, /, … cho một đối tượng. Tuy nhiên, ở đây tôi trình bày cách định nghĩa toán tử “<” vì đây là toán tử so sánh mặc định cho đa số các hàm hay containers trong STL. Chẳng hạn ta có thể viết:

|  |
| --- |
| **struct Edge {**  **int from, to, weight;**  **bool operator > (Edge other) const {**  **return weight < other.weight; // a lớn hơn b nếu a.weight < b.weight**  **}**  **int operator + (Edge other) const {**  **return weight – other.weight; // a + b = a.weight – b.weight**  **}**  **int operator / (Edge other) const {**  **return weight \* other.weight; // a / b = a.weight \* b.weight**  **}**  **}** |

**2. Định nghĩa một hàm so sánh của riêng mình**

Trong nhiều trường hợp, ta sử dụng những kiểu dữ liệu có sẳn trong thư viện. Tuy nhiên ta lại không muốn sử dụng hàm so sánh mặc định của kiểu dữ liệu đó. Mặt khác ta cũng không thể tự ý thay đổi cơ chế mặc so sánh mặc định của một kiểu dữ liệu có sẵn như kiểu dữ liệu tự định nghĩa ở trên. Cho nên ta cần phải viết một hàm so sánh của riêng mình với mục đích chuyên biệt. Hàm này sẽ có dạng:

|  |
| --- |
| **bool cmp(T a, T b)** |

Trong đó, T là một kiểu dữ liệu. Hàm cmp(a, b) sẽ trả về true nếu a ưu tiên cao hơn b vào ngược lại. Ví dụ với đối tượng Edge ở trên nếu không muốn định nghĩa toán tử “<” thì ta có thể viết một hàm cmp như sau:

|  |
| --- |
| **struct Edge {**  **int from, to, weight;**  **} e[N];**  **…**  **bool cmp(Edge a, Edge b) {**  **return a.weight < b.weight;**  **}**  **int main() {**  **…**  **sort(e, e + N, cmp);**  **}** |

**3. Functor**

Trong C++ có những kiểu lập trình khá “lạ”, trong đó có Functor hay Function Object. Functor được thiết kế để có thể dùng như một hàm và cũng có thể dùng như một đối tượng (bản chất nó là một object trong lập trình hướng đối tượng). Vì liên quan đến lập trình hướng đối tượng nên ta sẽ không đi sâu vào functor mà chỉ tìm hiểu cách dùng nó để viết một hàm so sánh.

Trong các containers có cấu trúc liên kết như map hay set thì thứ tự rất quan trọng. Vì các đối tượng trong cùng một tập đối tượng luôn được sắp xếp theo một thứ tự mặc định. Nhưng không phải lúc nào thứ tự này cũng đáp ứng như cầu của chúng ta. Vì vậy, functor là một giải pháp giúp ta tùy chỉnh thứ tự ưu tiên của các khóa trong một tập đối tượng. Một functor dùng với mục đích xác định thứ tự ưu tiên có thể viết như sau:

|  |
| --- |
| **struct cmp {**  **bool operator()(int a, int b) {**  **return a > b;**  **}**  **}** |

Khi đó ta có thể tùy chỉnh các containers theo thứ tự mà mình mong muốn. Ví dụ:

|  |
| --- |
| **set<int, cmp> s; // tập hợp s với các giá trị được xếp giảm dần**  **priority\_queue<int, vector<int>, cmp> pq; // hàng đợi ưu tiên với phần tử có ưu tiên cao nhất là phần tử bé nhất**  **map<int, char, cmp> m; // map m với khóa int được sắp xếp giảm dần** |

Hoặc có thể dùng với các hàm binary\_search, lower\_bound, upper\_bound, … Ví dụ:

|  |
| --- |
| **sort(v.begin(), v.end(), cmp);**  **binary\_search(v.begin(), v.end(), 6, cmp); // tìm xem có phần tử nào lớn hơn 6 trong v hay không** |

Ngoài ra, C++ cũng cung cấp sẵn một số function như less<T>, greater<T>, … Trong đó, như đã nói less<T> làm hàm so sánh mặc định của đa số các containers hay hàm.

**§4: CHỈ THỊ TIỀN XỬ LÝ TRONG C++**

Tác giả: PHẠM HOÀI NGUYÊN – STDIO.VN

**1. Chỉ thị tiền xử lý là gì ?**

Chỉ thị tiền xử lý (preprocessor directives) là những chỉ thị cung cấp cho trình biên dịch để xử lí những thông tin trước khi bắt đầu biên dịch thật sự. Tất cả các chỉ thị tiền xử lý đều bắt đầu với “#” và các chỉ thị tiền xử lí không phải là lệnh trong C++ cho nên không kết thúc bằng dấu “;”. Ta thường có ba nhóm chỉ thị tiền xử lý chính đó là:

* Chỉ thị khai báo header file (#include).
* Chỉ thị định nghĩa cho tên (#define marco).
* Chỉ thị có điều kiện (#if, #else, #elif, #endif).

**2. Chỉ thị khai báo header file**

Có duy nhất một chỉ thị để khai báo header file đó là #include. Chỉ thị này cho phép ta chèn một file khác vào file chúng ta đang làm việc. Có 2 cú pháp để thực hiện chỉ thị này:

**#include <***file\_name***>** ví dụ **#include <iostream>**

Khi dùng cú pháp này, bộ tiền xử lý (preprocessor) sẽ tìm file\_name được cung cấp trong IDE để biên dịch cùng với file bạn đang làm việc, nếu không tìm thấy trình biên dịch sẽ báo lỗi.

**#include “***file\_name***”** ví dụ **#include “iostream”**

Khi dùng cú pháp này, cách bộ tiền xử lý làm việc có khác đôi chút đó là trước khi tìm xem file\_name có được cung cấp bởi IDE hay không thì preprocessor sẽ tìm trong máy của chúng ta trước. Nếu kết quả vẫn là hoàn toàn không có thì trình biên dịch sẽ báo lỗi.

**3. Chỉ thị định nghĩa cho tên (#define marco)**

Có hai chỉ thị hay dùng trong nhóm này đó là #define và #undef

Chỉ thị #define có cú pháp như sau:

**#define ten\_moi ten\_duoc\_thay\_the**

Chỉ thị này có tác dụng báo cho trình biên dịch biết rằng ten\_moi khi biên dịch sẽ được thay thế bằng ten\_duoc\_thay\_the. Nếu ten\_duoc\_thay\_the quá dài thì ở cuối mỗi dòng ta thêm “\”. Ví dụ:

|  |
| --- |
| **#define ll long long**  **#define view(x) for (int i = 0; i < x.size(); i++)\**  **cout << x[i] << “ “;\** |

Từ đó khi biên dịch chương trình cứ hễ bắt gặp “ll” thì trình biên dịch sẽ hiểu là “long long” hay nếu bắt gặp view(x) thì trình biên dịch sẽ hiểu là cả dòng for phía sau. Phạm vi của tên được định nghĩa bởi #define là lúc từ khi nó được định nghĩa cho đến cuối tệp. Có thể dùng #define định nghĩa như tên hàm, một biểu thức, một đoạn chương trình bằng một tên, với cách sử dụng này thì chương trình của chúng ta sẽ ngắn gọn và dễ hiểu hơn. Ví dụ ta có thể thay cả một đoạn code chỉ bằng 1 từ như sau:

|  |
| --- |
| **#define HELLO { printf(“Hello STDIO\n”); printf(“stdio.vn”); }**  **void main() {**  **bool x = true;**  **if(x) HELLO;**  **}** |

Hay thậm chí là thay cho một hàm:

|  |
| --- |
| **#define SUM(x,y) (x)+(y)** |

Chỉ thị #undef được dùng khi ta cần định nghĩa lại một tên trước đó ta đã định nghĩa, múc đích của #undef là hủy định nghĩa đó để định nghĩa lại bằng #undef. Ví dụ:

|  |
| --- |
| **#define STDIO "Hello STDIO" // Định nghĩa cho tên STDIO là "Hello STDIO"**  **#undef STDIO // Hủy bỏ định nghĩa cho tên STDIO**  **#define STDIO "Welcome to STDIO" // Định nghĩa lại cho tên STDIO là "Welcome to STDIO"** |

**4. Chỉ thị biên dịch có điều kiện**

Ở nhóm này gồm các chỉ thị #if, #elif, #else, #ifdef, #ifndef.

Đầu tiên ta xét các chỉ thị: #if, #elif, #else.

Cú pháp:

|  |
| --- |
| **#if constant-expression\_1**  **// Đoạn chương trình 1**  **#elif constant-expression\_2**  **// Đoạn chương trình 2**  **#else**  **//Đoạn chương trình 3**  **#endif** |

Nếu constant-expression\_1 true thì chỉ có đoạn chương trình 1 sẽ được biên dịch, trái lại nếu constant-expression\_1 false thì sẽ tiếp tục kiểm ta đến constan-expression\_2. Nếu vẫn chưa đúng thì đoạn chương trình trong chỉ thị #else được biên dịch . Các constant-expression là biểu thức mà các toán hạng trong đó đều là hằng, các tên đã được định nghĩa bởi các #define cũng được xem là các hằng.

Các chỉ thị #ifdef, #ifndef. Một cách biên dịch có điều kiện khác đó là sử dụng #ifdef và #ifndef, được hiểu như là Nếu đã định nghĩa và Nếu chưa được định nghĩa.

Chỉ thị #ifdef

|  |
| --- |
| **#ifdef identifier**  **//Đoạn chương trình 1**    **#else**  **//Đoạn chương trình 2**    **#endif** |

Nếu indentifier đã được định nghĩa thì đoạn chương trình 1 sẽ được thực hiện. Ngược lại nếu indentifier chưa được định nghĩa thì đoạn chương trình 2 sẽ được thực hiện.

Chỉ thị #ifndef

|  |
| --- |
| **#ifndef identifier**  **//Đoạn chương trình 1**    **#else**  **//Đoạn chương trình 2**    **#endif** |

**§5: MỘT SỐ KĨ THUẬT TRONG C++11**

Tác giả: ALFONSO PETERSSEN – CODEFORCES

Từ trước đến giờ ta đã học về C++ dựa trên nền tảng C++98, nhưng bây giờ khi mà C++ 11 đã được chấp nhận trong các kì thi quốc gia, quốc tế thì việc tìm hiểu về các kĩ thuật mới trong C++11 là một điều hết sức cần thiết. Lưu ý bài viết này chỉ mang tính giới thiệu để người đọc nghiên cứu chứ không nhầm mục đích hướng dẫn chi tiết.

**1. Từ khóa “auto”**

Có thể nói từ khóa “auto” là một từ khóa cực kì hay trong C++11. Nói nôm na là nó có thể tự động nhận diện kiểu dữ liệu của một biến nào đó. Ví dụ trước đây để khai báo và khởi tạo một map với C++98 ta phải viết:

**map< string, pair< int, int > > somyLongTypeName = map< string, pair< int, int > >();**

Tuy nhiên nếu dùng auto thì mọi chuyện sẽ đơn giản hơn:

|  |
| --- |
| **auto longTypeNamesAreHistory = map< string, pair< int, int > >();**  **Hoặc với iterator:**  **for (map< string, pair< int, int > >::iterator it = m.begin(); it != m.end(); ++it) {**  **/\* do something with it \*/**  **}** |

Có thể thay bằng:

|  |
| --- |
| **for (auto it = m.begin(); it != m.end(); ++it) {**  **/\* do something with it \*/**  **}** |

**2. Khởi tạo một danh sách**

Chúng ta dễ dàng khởi tạo các containers dạng danh sách chỉ với cặp dấu “{}”:

|  |
| --- |
| **vector< int > primeDigits = {2, 3, 5, 7};**  **vector< string > days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};**  **pair< int, int > p = {1, 2}; // equivalent to make\_pair(1, 2)**  **map< int, string > numeral = {**  **{0, "zero"},**  **{1, "one"},**  **{2, "two"},**  **{3, "three"},**  **{4, "four"},**  **{5, "five"},**  **{6, "six"},**  **{7, "seven"},**  **{8, "eight"},**  **{9, "nine"}**  **};** |

**3. Duyệt toàn bộ contarners**

Phương pháp cũ:

|  |
| --- |
| **for (containers::iterator it = container.begin(); it != container.end(); ++it) {**  **cout << \*it << endl;**  **}** |

Tuy nhiên với C++11 thì:

|  |
| --- |
| **for (auto x : container) {**  **cout << x << endl;**  **}** |

Chúng ta thậm chí có thể tay đổi giá trị của một containers, hãy thử đoạn code dưới đây

|  |
| --- |
| **vector< int > numbers = {2, 3, 5, 7};**  **for (auto& x : numbers)**  **x \*= 2;**  **for (auto x : numbers)**  **cout << x << endl;** |

**4. Kiểu mảng mới**

C++11 xây dựng một kiểu mảng hẳn hoi là array, nó trong như thế này:

|  |
| --- |
| **auto arr = array< int, 10 >{5, 8, 1, 9, 0, 3, 4, 2, 7, 6}; // array of 10 ints**  **sort(arr.begin(), arr.end()); // yes, same as sort(arr, arr + n) for normal arrays**  **arr[0] += arr[5]; // normal [] indexing**  **for (auto i: range(arr.size())) {**  **/\* do something \*/**  **}**  **auto matrix = array< array<double, 10>, 10>(); // 10 by 10 matrix (not the same as double[10][10])** |

Thực sự thì nó trông không tốt hơn cách cài đặt mảng truyền thống là mấy, nếu không muốn nói là rườm rà hơn.

**5. Lambda functions**

Trước C++11, chúng ta thường hay sử dụng functor (function object) để tạo một hàm so sánh cho các containers và method trong algorithm. Tuy nhiên, từ C++11 trở đi ta có một dạng function mới đó là Lambda function. Chúng ta cùng xét ví dụ:

|  |
| --- |
| **auto heights = vector< int >{ ...some values here... };**  **auto order = vector< int >(heights.size());**  **for (int i: range(heights.size()))**  **order[i] = i;**  **sort(order.begin(), order.end(),**  **[&] (const int& a, const& int b) -> bool {**  **return heights[a] < heights[b];**  **}**  **);** |

Hoặc với for\_each:

|  |
| --- |
| **for\_each(v.begin(), v.end(),**  **[&] (int x) {**  **cout << x << endl;**  **}**  **)** |

**§6: TEMPLATE TRONG C++**

Tác giả: NGUYỄN HOÀNG MINH TRUNG – STDIO.VN

**1. Template là gì ?**

“Template” là từ khóa trong C++, chúng ta có thể hiểu rằng là nó một kiểu dữ liệu trừu tượng, đặc trưng cho các kiểu dữ liệu cơ bản. “Template” là từ khóa báo cho trình biên dịch rằng đoạn mã sau đây định nghĩa cho nhiều kiểu dữ liệu và mã nguồn của nó sẽ được compile sinh ra tương ứng cho từng kiểu dữ liệu trong quá trình biên dịch.

Có 2 loại “template” cơ bản:

* Function template: là một khuôn mẫu hàm, cho phép định nghĩa các hàm tổng quát thao tác cho nhiều kiểu dữ liệu.
* Class template: là một khuôn mẫu lớp, cho phép định nghĩa các lớp tổng quát cho nhiều kiểu dữ liệu.

**2. Function template**

|  |
| --- |
| **void Swap( int &x, int &y)**  **{**  **int Temp = x;**  **x = y;**  **y = Temp;**  **}** |

Với đoạn code trên dùng để hoán vị giữa 2 số nguyên, nếu ta cần hoán vị giữa 2 số kiểu float hoặc double,… thì ta lại phải định nghĩa các hàm cho từng loại kiểu dữ liệu như thế. Tôi tự hỏi có cách nào mà có thể định nghĩa một hàm duy nhất mà có thể dùng cho nhiều kiểu dữ liệu hay không, sau một hồi tìm hiểu thì tôi đã có câu trả lời đó là “template”, với từ khóa “template” ta chỉ cần định nghĩa một hàm duy nhất cho các kiểu dữ liệu: int, float, double, …

|  |
| --- |
| **template <class Type>**  **void Swap( Type &x, Type &y)**  **{**  **Type Temp = x;**  **x = y;**  **y = Temp;**  **}**    **void main()**  **{**  **int x = 5, y = 10;**  **float a = 5.5f, b = 3.0f;**    **Swap(x, y);**  **Swap(a, b);**  **}** |

Vậy cách thức nó hoạt động ra sao, ta hãy xét hoạt động của trình biên dịch khị gặp lời gọi hàm Swap(x, y) với tham số truyền vào là kiểu int, trước hết khi gặp lời gọi hàm Swap(x, y) thì trình biên dịch tìm xem có hàm Swap() nào đã được khai báo với tham số kiểu int hay chưa, nếu có thì sẽ liên kết với hàm đó, nếu chưa nhưng lại tìm thấy từ khóa “template” với hàm Swap() được truyền vào 2 tham số cùng kiểu với nhau (lúc này là kiểu Type), trình biên dịch chỉ cần kiểm tra xem lời gọi hàm Swap(x, y) có 2 tham số có cùng kiểu dữ liệu với nhau hay không( trong ví dụ trên là 2 tham số kiểu int -> cùng kiểu dữ liệu), nếu cùng kiểu thì trình biên dịch lại kiểm tra xem hàm Swap() với 2 tham số kiểu int đã được sinh ra trước đó hay chưa, nếu có thì lời gọi hàm sẽ liên kết với hàm Swap() đã được sinh ra, nếu chưa thì khi đó trình biên dịch sẽ sinh ra hàm Swap(int &x, int &y), tương tự với Swap(a, b) kiểu float cũng tương tự như vậy. Vì vậy trong quá trình biên dịch sẽ sinh ra 2 hàm Swap() cho 2 loại kiểu dữ liệu trên. Trong ví dụ trên tôi đã chỉ ra cách khai báo hàm mẫu, trong đó:

* “Type” chỉ là một tên riêng thể hiện cho một kiểu dữ liệu tổng quát.
* “class” ta có thể thay thế bằng “typename”, ở đây nó không có sự khác biệt.

Ngoài ra ta có thể dùng prototype cho nguyên mẫu hàm giống như ta làm cho các hàm thông thường

|  |
| --- |
| **template <class T> void Swap( T &x, T &y);**    **void main()**  **{**  **int x = 3, y = 4;**  **float a = 1.2f, b = 2.2f;**    **Swap(x, y);**  **Swap(a, b);**  **}**    **template <class T> void Swap( T &x, T &y)**  **{**  **T z = x;**  **x = y;**  **y = z;**  **}** |

Trong ví dụ trên ta chỉ hoán ví giữa 2 tham số cùng kiểu, vậy với khác kiểu thì sau? Ta chỉ cần khai báo thêm một kiểu dữ liệu tổng quát

|  |
| --- |
| **template <class T, class X> void Swap( T &x, X &y);** |

**3. Overloading Function Templates**

Nguyên mẫu hàm (function template) đều có tính chất của một hàm thông thường, nó cho phép chúng ta nạp chồng (overload function).

|  |
| --- |
| **template <class T, class X>**  **T Sum( T x, X y)**  **{**  **T sum = x + y;**  **return sum;**  **}**    **template <class T, class X>**  **X Sum( T x, X y, T z)**  **{**  **X sum = x + y + z;**  **return sum;**  **}** |

Lưu ý: muốn nạp chồng hàm thì các tham số truyền vào ở các hàm phải khác nhau

**4. Class template**

Ta xét ví dụ sau

|  |
| --- |
| **class Point**  **{**  **int x;**  **int y;**  **};** |

Nếu muốn khai báo khuôn mẫu lớp với kiểu tùy ý ta làm như sau

|  |
| --- |
| **template <class Type>**  **class Point**  **{**  **Type x;**  **Type y;**  **};** |

Lúc này nếu như muốn khai báo một thể hiện template Point: **Point<int> P1;**

Trong ví dụ trên là khai báo Point với 2 thuộc tính cùng kiểu với nhau, chúng ta có thể định nghĩa với 2 thuộc tính có kiểu dữ liệu khác nhau​

|  |
| --- |
| **template <class TypeOne, class TypeTwo>**  **class Point**  **{**  **TypeOne x;**  **TypeTwo y;**  **Public:**  **Point();**  **};**    **template <class TypeOne, class TypeTwo>**  **Point<TypeOne, TypeTwo>::Point()**  **{**  **x = 0;**  **y = 0;**  **}** |

Trong ví dụ trên, khi muốn định nghĩa hàm của một lớp mẫu ở file khác thì ta phải khai báo template với các tham số kiểu trước mỗi hàm ta muốn định nghĩa None-type template parameters None-type template parameters là tham số mẫu mặc định, được xác định và trình biên dịch không sinh ra kiểu khác trong suốt thời gian biên dịch.

|  |
| --- |
| **template <class Type, int N>**  **class Stack**  **{**  **//do something**  **};** |

Trong ví dụ trên N là kiểu int đã được xác định rõ từ trước và không thay đổi trong thời gian biên dịch

**5. Default type**

Xét ví dụ sau

|  |
| --- |
| **template <class Type = int> //defaults to type int**  **class Point**  **{**  **Type x;**  **Type y;**  **};**    **Point<> point;** |

Trong ví dụ trên khi khai báo thể hiện của khuôn mẫu lớp là Point<> point; do không có kiểu truyền vào, lúc này trình biên dịch sẽ mặc định là kiểu đó là int do lúc đầu đã gán tham số kiểu mặc định là int khi khai báo kiểu trong template. Lưu ý: tham số kiểu mặc định phải nằm ngoài cùng bên phải trong danh sách các tham số kiểu mẫu