

GIÁO TRÌNH LẬP TRÌNH NÂNG CAO



Nguyễn Văn Vinh, Phạm Hồng Thái, Trần Quốc Long
Khoa Công nghệ Thông tin - Trường Đại học Công nghệ - ĐHQG Hà Nội

MỤC LỤC

1	Mở đầu	1
1.1	Giải quyết bài toán bằng lập trình	1
1.1.1	Thuật toán	1
1.1.2	Thiết kế chương trình	2
1.1.3	Chu kỳ phát triển phần mềm	2
1.2	Tiêu chuẩn đánh giá một chương trình tốt	3
1.3	Ngôn ngữ lập trình và chương trình dịch	4
1.4	Môi trường lập trình bậc cao	4
1.5	Lịch sử C và C++	5
1.6	Chương trình đầu tiên trong C++: In dòng văn bản	7
2	Một số khái niệm cơ bản trong C++	11
2.1	Khai báo biến và sử dụng biến	11
2.1.1	Biến	11
2.1.2	Tên hay định danh	12
2.1.3	Câu lệnh gán	14
2.2	Vào ra dữ liệu	15
2.2.1	Xuất dữ liệu với <code>cout</code>	15
2.2.2	Chỉ thị biên dịch và không gian tên	17
2.2.3	Các chuỗi Escape	18
2.2.4	Nhập dữ liệu với <code>cin</code>	18
2.3	Kiểu dữ liệu và biểu thức	20
2.3.1	Kiểu <code>int</code> và kiểu <code>double</code>	20
2.3.2	Các kiểu số khác	22
2.3.3	Kiểu <code>C++11</code>	22
2.3.4	Kiểu <code>char</code>	24
2.3.5	Tương thích kiểu dữ liệu	25
2.3.6	Toán từ số học và biểu thức	26
2.4	Luồng điều khiển	28
2.5	Phong cách lập trình	31

2.6	Biên dịch chương trình với GNU/C++	32
3	Kiểm thử và gỡ rối chương trình	37
3.1	Kỹ thuật kiểm thử	37
3.1.1	Kiểm thử trong khi viết mã nguồn	38
3.2	Kỹ thuật gỡ rối chương trình	39
3.2.1	Khái niệm về gỡ rối chương trình	39
3.2.2	Phân loại lỗi	39
3.2.3	Một số kỹ thuật gỡ rối	40
3.2.4	Giải pháp và vấn đề liên quan đến C/C++	42
3.3	Lập trình không lỗi	44
4	Hàm	47
4.1	Thiết kế từ trên xuống (top-down)	47
4.2	Hàm	48
4.2.1	Ý nghĩa của hàm	48
4.2.2	Cấu trúc chung của hàm	48
4.2.3	Khai báo hàm	51
4.3	Cách sử dụng hàm	52
4.3.1	Lời gọi hàm	52
4.3.2	Hàm với đối mặc định	54
4.4	Biến toàn cục và biến địa phương	55
4.4.1	Biến địa phương (biến trong hàm, trong khối lệnh)	55
4.4.2	Biến toàn cục (biến ngoài tất cả các hàm)	56
4.4.3	Mức ưu tiên của biến toàn cục và địa phương	56
4.5	Tham đối và cơ chế truyền giá trị cho tham đối	60
4.5.1	Truyền theo tham trị	60
4.5.2	Biến tham chiếu	61
4.5.3	Truyền theo tham chiếu	63
4.5.4	Hai cách truyền giá trị cho hàm và từ khóa const	64
4.6	Ngăn xếp gọi hàm và các mẫu tin kích hoạt	64
4.7	Chồng hàm và khuôn mẫu hàm	68
4.7.1	Chồng hàm (hàm trùng tên)	68
4.7.2	Khuôn mẫu hàm	70
4.8	Lập trình với hàm đệ quy	72
4.8.1	Khái niệm đệ quy	72
4.8.2	Lớp các bài toán giải được bằng đệ quy	74
4.8.3	Cấu trúc chung của hàm đệ quy	75

5	Mảng	83
5.1	Lập trình và thao tác với mảng một chiều	83
5.1.1	Ý nghĩa của mảng	83
5.1.2	Thao tác với mảng một chiều	84
5.1.3	Mảng và hàm	88
5.1.4	Tìm kiếm và sắp xếp	93
5.2	Lập trình và thao tác với mảng nhiều chiều	98
5.2.1	Mảng 2 chiều	98
5.2.2	Thao tác với mảng hai chiều	99
5.3	Lập trình và thao tác với xâu kí tự	105
5.3.1	Khai báo	106
5.3.2	Thao tác với xâu kí tự	106
5.3.3	Phương thức nhập xâu (<code>#include <iostream></code>)	107
5.3.4	Một số hàm làm việc với xâu kí tự (<code>#include <cstring></code>)	108
5.3.5	Các hàm chuyển đổi xâu dạng số thành số (<code>#include <cstdlib></code>)	112
5.3.6	Một số ví dụ làm việc với xâu	113
6	Các kiểu dữ liệu trừu tượng	119
6.1	Kiểu dữ liệu trừu tượng bằng cấu trúc (<code>struct</code>)	119
6.1.1	Khai báo, khởi tạo	119
6.1.2	Hàm và cấu trúc	122
6.1.3	Bài toán Quản lý sinh viên (QLSV)	127
6.2	Kiểu dữ liệu trừu tượng bằng lớp (<code>class</code>)	134
6.2.1	Khai báo lớp	135
6.2.2	Sử dụng lớp	136
6.2.3	Bài toán Quản lý sinh viên	143
6.2.4	Khởi tạo (giá trị ban đầu) cho một đối tượng	146
6.2.5	Hủy đối tượng	152
6.2.6	Hàm bạn (friend function)	152
6.2.7	Tạo các phép toán cho lớp (hay tạo chồng phép toán - Operator Overloading)	156
6.3	Dạng khuôn mẫu hàm và lớp	159
6.3.1	Khai báo một kiểu mẫu	159
6.3.2	Sử dụng kiểu mẫu	160
6.3.3	Một số dạng mở rộng của khai báo mẫu	162
7	Con trỏ và bộ nhớ	167
7.1	Khái niệm con trỏ	167
7.2	Biến con trỏ	167
7.3	Cấp phát bộ nhớ động	171

7.4	Con trỏ và mảng động	172
7.4.1	Biến mảng và biến con trỏ	172
7.4.2	Biến mảng động	174
7.5	Truyền tham số của hàm như con trỏ	176
7.6	Con trỏ hàm	177
7.7	Lập trình với danh sách liên kết	178
7.7.1	Nút và danh sách liên kết	180
7.7.2	Danh sách liên kết của lớp	192
8	Vào ra dữ liệu	197
8.1	Dòng và vào ra file cơ bản	197
8.2	Vào ra file	198
8.2.1	Mở file	198
8.2.2	Đóng file	199
8.3	Vào ra với file văn bản	199
8.4	Vào ra với file nhị phân	201
8.5	Truy cập ngẫu nhiên	202
9	Xử lý ngoại lệ	205
9.1	Các vấn đề cơ bản trong xử lý ngoại lệ	205
9.1.1	Ví dụ xử lý ngoại lệ	205
9.1.2	Định nghĩa lớp ngoại lệ	207
9.1.3	Ném và bắt nhiều ngoại lệ	208
9.1.4	Ném ngoại lệ từ hàm	209
9.1.5	Mô tả ngoại lệ	211
9.2	Kỹ thuật lập trình cho xử lý ngoại lệ	211
9.2.1	Ném ngoại lệ ở đâu	211
9.2.2	Cây phả hệ ngoại lệ STL	212
9.2.3	Kiểm tra bộ nhớ	213
10	Tiền xử lý và lập trình nhiều file	215
10.1	Các chỉ thị tiền xử lý	215
10.1.1	Chỉ thị bao hàm tệp <code>#include</code>	215
10.1.2	Chỉ thị macro <code>#define</code>	216
10.1.3	Các chỉ thị biên dịch có điều kiện <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code>	217
10.2	Lập trình trên nhiều file	219
10.2.1	Tổ chức chương trình	219
10.2.2	Viết và kiểm tra các file include	220
10.2.3	Biên dịch chương trình có nhiều file	220

11 Lập trình với thư viện chuẩn STL	225
11.1 Giới thiệu thư viện chuẩn STL	225
11.2 Khái niệm con trỏ duyệt	225
11.2.1 Các thao tác cơ bản với con trỏ duyệt	226
11.2.2 Các loại con trỏ duyệt	228
11.3 Khái niệm vật chứa	230
11.3.1 Các vật chứa dạng dãy	231
11.3.2 Ngăn xếp và hàng đợi	236
11.3.3 Tập hợp và ánh xạ	239
11.3.4 Hàm băm, tập hợp và ánh xạ không thứ tự (C++11)	242
11.4 Các thuật toán mẫu	244
11.4.1 Thời gian chạy và ký hiệu “O-lớn”	244
11.4.2 Các thuật toán không thay đổi vật chứa	245
11.4.3 Các thuật toán thay đổi vật chứa	248
11.4.4 Các thuật toán tập hợp	250
11.5 Một số thư viện chuẩn khác trong STL	250
11.5.1 Xử lý xâu với <code><string></code>	251
11.5.2 Con trỏ thông minh và quản lý bộ nhớ với <code><memory></code> (C++11)	254
11.5.3 Tính toán thời gian với <code><chrono></code> (C++11)	255
11.5.4 Lập trình song song với <code><thread></code> (C++11)	256
A Bảng từ khóa của ngôn ngữ C++	261
B Thứ tự ưu tiên của phép toán	263
C Phong cách lập trình	265
D Hàm inline	269
Tài liệu tham khảo	271

Lời giới thiệu

Lập trình là cách thức diễn tả thuật toán (chương trình) giải quyết một vấn đề sao cho máy tính hiểu và thi hành thuật toán đó. Nắm bắt và hiểu rõ về các kỹ thuật lập trình giúp chúng ta viết chương trình hiệu quả hơn cũng như ít phát sinh lỗi hơn. Hơn nữa, thuật toán của một bài toán chỉ có thể được hiểu cặn kẽ thông qua chương trình thể hiện thuật toán đó trên máy tính.

Giáo trình **Lập trình nâng cao** cung cấp các nội dung cơ bản và nâng cao về các kỹ thuật lập trình cho sinh viên đại học chuyên ngành CNTT và những ai yêu thích về lập trình. Giáo trình cũng giới thiệu các kiến thức căn bản về ngôn ngữ C++ và sử dụng ngôn ngữ lập trình này để minh họa ví dụ về các kỹ thuật lập trình. Giáo trình này thích hợp nhất cho những ai đã có kiến thức cơ bản về lập trình.

Cách tiếp cận khi viết giáo trình này là trình bày các kỹ thuật lập trình để giải quyết bài toán chứ không đi sâu về giới thiệu ngôn ngữ lập trình. Hơn nữa giáo trình này cũng được thiết kế dành cho các bạn sinh viên đã có kiến thức cơ bản về lập trình, ví dụ như đã học qua môn học về nhập môn lập trình. Do đó, giáo trình chỉ đề cập các kiến thức cơ bản nhất trong ngôn ngữ C++ để minh họa các kỹ thuật lập trình.

Nội dung trong giáo trình đề cập các kỹ thuật lập trình cơ bản đến nâng cao giúp sinh viên lập trình giải quyết các bài toán một cách hiệu quả và giảm thiểu mắc lỗi trong chương trình. Các chương trong giáo trình bao gồm như sau: Chương 1 giới thiệu về các bước giải bài toán bằng lập trình và ngôn ngữ lập trình bậc cao C++. Chương 2 trình bày các khái niệm cơ bản trong C++. Kiểm thử và gỡ rối là các kỹ thuật quan trọng trong quá trình lập trình. Vì vậy, vấn đề này đã được đề cập trong chương 3 của giáo trình. Chương 4, 5 lần lượt đề cập đến lập trình sử dụng hàm và mảng trong C++. Chương 6 giới thiệu về các kiểu dữ liệu trừu tượng. Chương 7 trình bày về con trỏ, bộ nhớ cũng như các kỹ thuật lập trình dựa vào con trỏ. Chương 8, 9 trình bày thao tác vào ra dữ liệu và cách xử lý ngoại lệ trong C++. Nhằm cung cấp các kỹ thuật lập trình để phát triển các dự án lớn và phức tạp, chương 10 cung cấp các kiến thức về tiền xử lý và lập trình nhiều file. Chương cuối cùng trình bày về thư viện chuẩn nổi tiếng STL của C++ và cách thức lập trình sử dụng thư viện này. Ngoài ra còn có các phụ lục ở cuối giáo trình: bảng từ khóa C++, thứ tự ưu tiên các phép toán, phong cách lập trình.

Các tác giả chân thành cảm ơn TS. Trần Thị Minh Châu, TS. Lê Quang Minh, ThS. Trần Hồng Việt, ThS. Phạm Nghĩa Luân, ThS. Nguyễn Quang Huy cùng các đồng nghiệp và sinh viên tại khoa CNTT, Trường Đại học Công nghệ đã đọc bản thảo và đóng góp các ý kiến quý báu về nội dung cũng như hình thức trình bày. Đây là lần đầu tiên xuất bản nên chắc chắn giáo trình còn nhiều khiếm khuyết, chúng tôi rất mong nhận được các ý kiến góp ý để giáo trình được hoàn thiện hơn.

Nhóm tác giả

Chương 1

Mở đầu

Trong chương này, chúng tôi mô tả các thành phần cơ bản của máy tính cũng như các kỹ thuật cơ bản về thiết kế và viết chương trình trên máy tính. Tiếp theo, chúng tôi minh họa về một chương trình đơn giản trong ngôn ngữ C++ và mô tả chúng hoạt động như thế nào.

1.1 Giải quyết bài toán bằng lập trình

Trong phần này, chúng ta mô tả một số nguyên lý chung để sử dụng thiết kế và viết chương trình trên máy tính. Đây là các nguyên lý tổng quát ta có thể sử dụng cho bất cứ ngôn ngữ lập trình nào chứ không chỉ trên ngôn ngữ C++.

1.1.1 Thuật toán

Khi học ngôn ngữ lập trình đầu tiên, chúng ta thường dễ nhận ra rằng công việc khó khăn của giải quyết một bài toán trên máy tính là chuyển ý tưởng của chúng ta thành các ngôn ngữ cụ thể để được đưa vào máy tính. Phần khó khăn nhất của giải quyết bài toán trên máy tính là tìm ra giải pháp. Sau khi chúng ta tìm ra giải pháp, công việc thường lệ tiếp theo là chuyển giải pháp của bài toán thành ngôn ngữ được yêu cầu, có thể là C++ hoặc là một số ngôn ngữ lập trình khác. Vì vậy, điều hữu ích là tạm thời bỏ qua ngôn ngữ lập trình và thay vào đó là tập trung xây dựng các bước của giải pháp và viết chúng ra bằng ngôn ngữ tự nhiên (tiếng Việt, tiếng Anh, ...). Dãy các bước giải pháp như vậy được hiểu như là thuật toán.

Dãy các chỉ thị chính xác mà đưa ra giải pháp được gọi là thuật toán. Thuật toán có thể biểu diễn dưới dạng ngôn ngữ tự nhiên hoặc ngôn ngữ lập trình như C++. Một chương trình máy tính đơn giản là một thuật toán biểu diễn trên ngôn ngữ mà máy tính có thể hiểu và thi hành. Vì vậy, khái niệm thuật toán là tổng quát hơn so với khái niệm chương trình.

Tuy nhiên, khi chúng ta nói dãy các chỉ thị là thuật toán, chúng ta thường hiểu các chỉ thị này biểu diễn dưới dạng ngôn ngữ tự nhiên (mã giả), còn khi chúng biểu diễn dưới dạng ngôn ngữ lập trình thì chúng ta thường gọi đó là chương trình. Ví dụ dưới đây giúp ta hiểu rõ khái niệm này. Hình 1.1 mô tả thuật toán bằng ngôn ngữ tự nhiên. Thuật toán xác định số lần của một tên nào đó xuất hiện trong danh sách tên.

Thuật toán xác định số lần của một tên xuất hiện trong danh sách tên cho trước

1. Lấy danh sách tên
2. Lấy tên cần tính
3. Thiết lập SOLANTEN bằng 0
4. Thực hiện với mỗi tên trong danh sách tên
So sánh tên đó với tên cần tính, nếu cùng tên thì SOLANTEN tăng thêm 1.
5. Thông báo số lần tên cần tính là SOLANTEN

Hình 1.1: Thuật toán

1.1.2 Thiết kế chương trình

Thiết kế chương trình thường là nhiệm vụ khó. Không có một tập đầy đủ các qui tắc, thuật toán để nói với chúng ta viết chương trình như thế nào. Tuy nhiên, chúng ta cũng có một quá trình thiết kế chương trình tương đối tổng quát mô tả trong hình 1.2. Toàn bộ việc thiết kế chương trình được chia làm hai pha: pha giải quyết bài toán và pha thực hiện. Kết quả của pha giải quyết bài toán là thuật toán của bài toán biểu diễn dưới dạng ngôn ngữ tự nhiên. Để có chương trình trong ngôn ngữ lập trình như C++, thuật toán được chuyển đổi trong ngôn ngữ lập trình. Quá trình xây dựng chương trình từ thuật toán gọi là pha thực hiện.

Bước định nghĩa bài toán, chắc chắn là bài toán mà chương trình cần giải quyết là gì. Chúng ta cần mô tả đầy đủ và chính xác. Chúng ta cần định nghĩa bài toán dưới dạng tin học. Xác định rõ các ràng buộc, dữ liệu đầu vào và đầu ra của bài toán cũng như kiểu dữ liệu cần xử lý. Ví dụ, nếu là chương trình kế toán ngân hàng, chúng ta phải biết không chỉ là lãi suất mà còn lãi suất được cộng dồn từ hàng năm, hàng tháng, hàng ngày hay không.

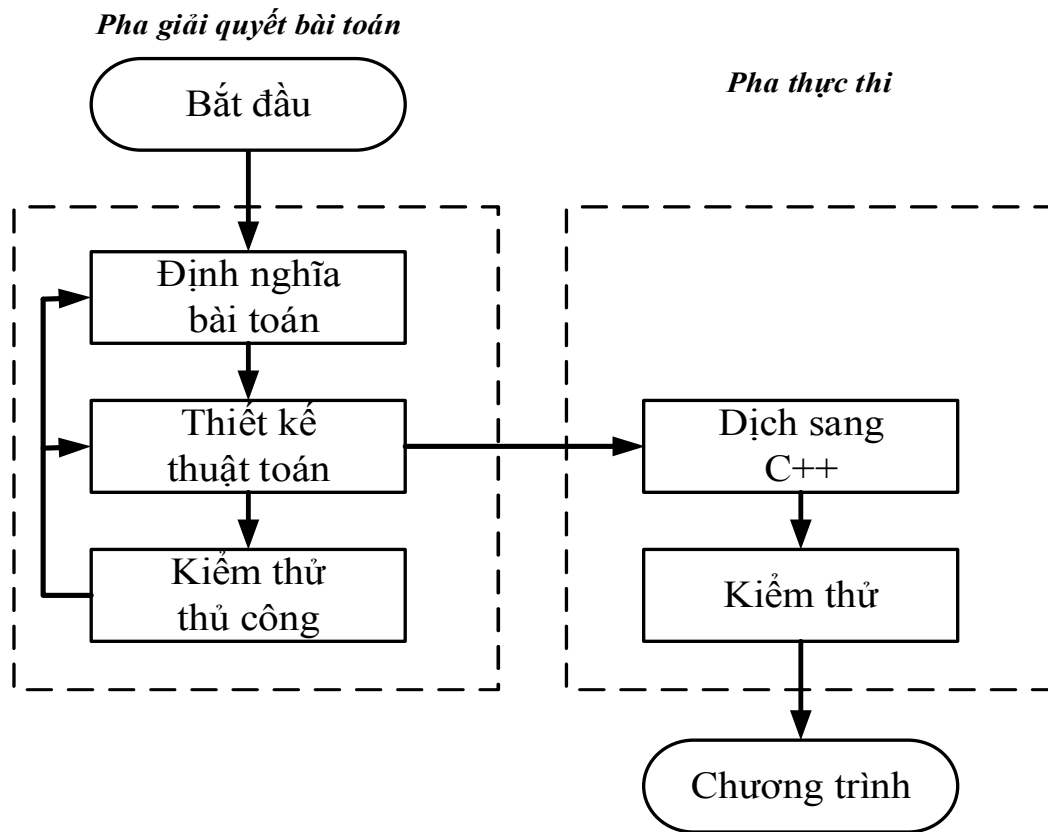
Pha thực hiện không phải là bước đơn giản. Có những chi tiết có thể được quan tâm và đôi khi có một số chi tiết có thể được tối ưu và thực hiện tinh tế nhưng nó đơn giản hơn so với pha đầu tiên. Khi chúng ta thành thạo với ngôn ngữ C++ hoặc bất cứ ngôn ngữ lập trình khác, việc chuyển đổi từ thuật toán sang chương trình trong ngôn ngữ lập trình trở thành công việc bình thường.

Như đề cập trong hình 1.2, kiểm thử xảy ra cả trong 2 pha. Trước khi chương trình được viết, thuật toán cần được kiểm thử. Khi thuật toán chưa hiệu quả, cần thiết kế thuật toán lại. Kiểm thử thủ công từng bước thực hiện và thi hành thuật toán là do chính chúng ta làm. Chương trình C++ được kiểm thử bằng cách dịch nó và chạy nó với một số bộ dữ liệu đầu vào. Trình biên dịch sẽ cho thông báo lỗi với một số loại lỗi cụ thể nào đó.

1.1.3 Chu kỳ phát triển phần mềm

Thiết kế hệ thống phần mềm lớn như trình biên dịch hoặc hệ điều hành thường chia quy trình phát triển phần mềm thành sáu pha được biết như là chu kỳ phát triển phần mềm. Sáu pha đó như sau:

1. Phân tích và đặc tả bài toán (định nghĩa bài toán)
2. Thiết kế phần mềm (thiết kế thuật toán và đối tượng)



Hình 1.2: Quá trình thiết kế chương trình

3. Lập trình
4. Kiểm thử
5. Bảo trì và nâng cấp của hệ thống phần mềm
6. Hủy không dùng nữa

1.2 Tiêu chuẩn đánh giá một chương trình tốt

Như thế nào là một chương trình tốt có lẽ là chủ đề tranh luận chưa bao giờ nguôi từ khi con người bắt đầu lập trình cho máy tính. Có thể nói, viết một chương trình tốt là một nghệ thuật nhưng qua kinh nghiệm của chúng tôi, một chương trình tốt thường có những đặc điểm sau:

1. **Dễ đọc:** Mã nguồn của một chương trình tốt phải giúp lập trình viên (cả người viết chương trình, người trong nhóm, hoặc người bảo trì chương trình) đọc chúng một cách dễ dàng. Luồng điều khiển trong chương trình phải rõ ràng, không làm khó cho người đọc. Nói một cách khác, chương trình tốt có khả năng *giao tiếp* với người đọc chúng.
2. **Dễ kiểm tra:** Các mô-đun, các hàm trong chương trình được viết sao cho chúng có thể dễ dàng đặt vào các bộ kiểm tra đơn vị chương trình (*unit test*).
3. **Dễ bảo trì:** Khi sửa lỗi hoặc cải tiến chương trình, thường chỉ cần tác động vào một vài bộ phận trong mã nguồn.

4. **Dễ mở rộng:** Khi cần thêm các chức năng hoặc tính năng mới, người viết chương trình dễ dàng viết tiếp mã nguồn mới để thêm vào mã nguồn cũ. Người mở rộng chương trình (có thể không phải người lập trình đầu tiên) khó có thể “*làm sai*” khi mở rộng mã nguồn của một chương trình tốt.

Tất nhiên, tất cả các đặc điểm trên là các đặc điểm **lý tưởng** của một chương trình tốt. Khi phát triển chương trình hoặc phần mềm, các điều kiện thực tế sẽ ảnh hưởng rất nhiều khả năng chúng ta đạt được những đặc điểm của một chương trình hoàn hảo. Ví dụ, đến hạn báo cáo hoặc nộp chương trình cho đối tác, chúng ta không kịp kiểm tra hết mọi tính năng. Hoặc chúng ta bỏ qua rất nhiều bước tối ưu mã nguồn và làm cho mã nguồn trong sáng, dễ hiểu. Thực tế làm phần mềm là quá trình cân bằng giữa lý tưởng (4 đặc điểm trên) và các yêu cầu khác. Hiếm khi chúng ta thỏa mãn được 4 đặc điểm này nhưng chúng sẽ luôn là cái đích chúng ta, những lập trình viên tương lai hướng tới.

1.3 Ngôn ngữ lập trình và chương trình dịch

Có nhiều ngôn ngữ lập trình để viết chương trình. Trong giáo trình này, chúng tôi giới thiệu và sử dụng ngôn ngữ lập trình C++ để viết chương trình. C++ là **ngôn ngữ lập trình bậc cao** và được sử dụng rộng rãi trong thực tế để phát triển phần mềm. Ngoài ra còn có các ngôn ngữ bậc cao thông dụng hiện nay như C, C#, Java, Python, PHP, Pascal. Ngôn ngữ lập trình bậc cao gần với ngôn ngữ tự nhiên của con người. Chúng được thiết kế để con người dễ dàng viết chương trình và con người cũng dễ dàng đọc chương trình được viết trên đó. Ngôn ngữ bậc cao như C++, bao gồm các chỉ thị phức tạp hơn rất nhiều so với các chỉ thị đơn giản mà các bộ vi xử lý của máy tính có thể hiểu và thi hành. Điều này để phân biệt với một loại ngôn ngữ mà máy tính có thể hiểu được thường gọi là **ngôn ngữ bậc thấp**.

Ngôn ngữ bậc thấp bao gồm ngôn ngữ máy và ngôn ngữ Assembly. Ngôn ngữ máy bao gồm các chỉ thị do phần cứng tạo ra nên máy có thể hiểu được ngay. Còn ngôn ngữ Assembly đã sử dụng các tập lệnh và qui tắc bằng tiếng Anh đơn giản để biểu diễn. Ngôn ngữ Assembly khá gần ngôn ngữ máy và nó cần được dịch bởi chương trình dịch đơn giản để thành ngôn ngữ máy.

1.4 Môi trường lập trình bậc cao

Phần này trình bày các bước để xây dựng và thi hành chương trình C++ sử dụng môi trường phát triển C++ (minh họa trong hình 1.3), hệ thống C++ bao gồm ba phần: môi trường phát triển, ngôn ngữ và thư viện chuẩn của C++. Chương trình C++ cơ bản có 6 pha: Soạn thảo, tiền xử lý, Dịch, liên kết, nạp và thi hành. Dưới đây sẽ mô tả chi tiết môi trường phát triển chương trình C++ cơ bản.

Pha 1: Xây dựng chương trình

Bước này bao gồm soạn thảo file trong trình soạn thảo. Chúng ta gõ chương trình C++ (có thể hiểu là mã nguồn) sử dụng trình soạn thảo này và lưu chương trình vào đĩa cứng. Tên của mã nguồn C++ thường kết thúc với đuôi mở rộng là .cpp, .cxx, .cc hoặc .C. Hai trình soạn thảo phổ

biến nhất trên hệ điều hành UNIX là vim và emacs. Đối với hệ điều hành Window, gói phần mềm C++ của Microsoft là Microsoft Visual C++ có trình soạn thảo tích hợp vào môi trường lập trình. Chúng ta cũng có thể sử dụng trình soạn thảo đơn giản như Notepad của Window để viết chương trình mã nguồn C++.

Pha 2 và 3: Tiền xử lý và biên dịch chương trình C++

Trong bước này, chúng ta thực hiện lệnh dịch chương trình mã nguồn C++. Trong hệ thống C++, trình tiền xử lý thi hành tự động trước khi bước thực thi của trình biên dịch. Trình tiền xử lý thực hiện các chỉ thị tiền xử lý. Các thao tác này được thực hiện trước khi biên dịch chương trình. Các thao tác thường bao gồm file văn bản khác được biên dịch hoặc thực hiện việc thay thế các đoạn văn bản khác nhau. Chỉ thị tiền xử lý thông dụng đề cập chi tiết trong phần phụ lục. Trong pha 3, trình biên dịch sẽ dịch chương trình C++ thành mã ngôn ngữ máy (mã đối tượng).

Pha 4: Liên kết

Chương trình C++ cơ bản bao gồm các tham chiếu tới hàm và dữ liệu được định nghĩa ở nơi khác như thư viện chuẩn hoặc thư viện của người dùng tự tạo ra. Mã đối tượng sinh ra do trình biên dịch C++ thường chứa “lỗ” do những phần thiếu. Trình liên kết sẽ liên kết mã đối tượng với mã của các hàm thiếu để tạo ra chương trình thi hành.

Pha 5: Nạp

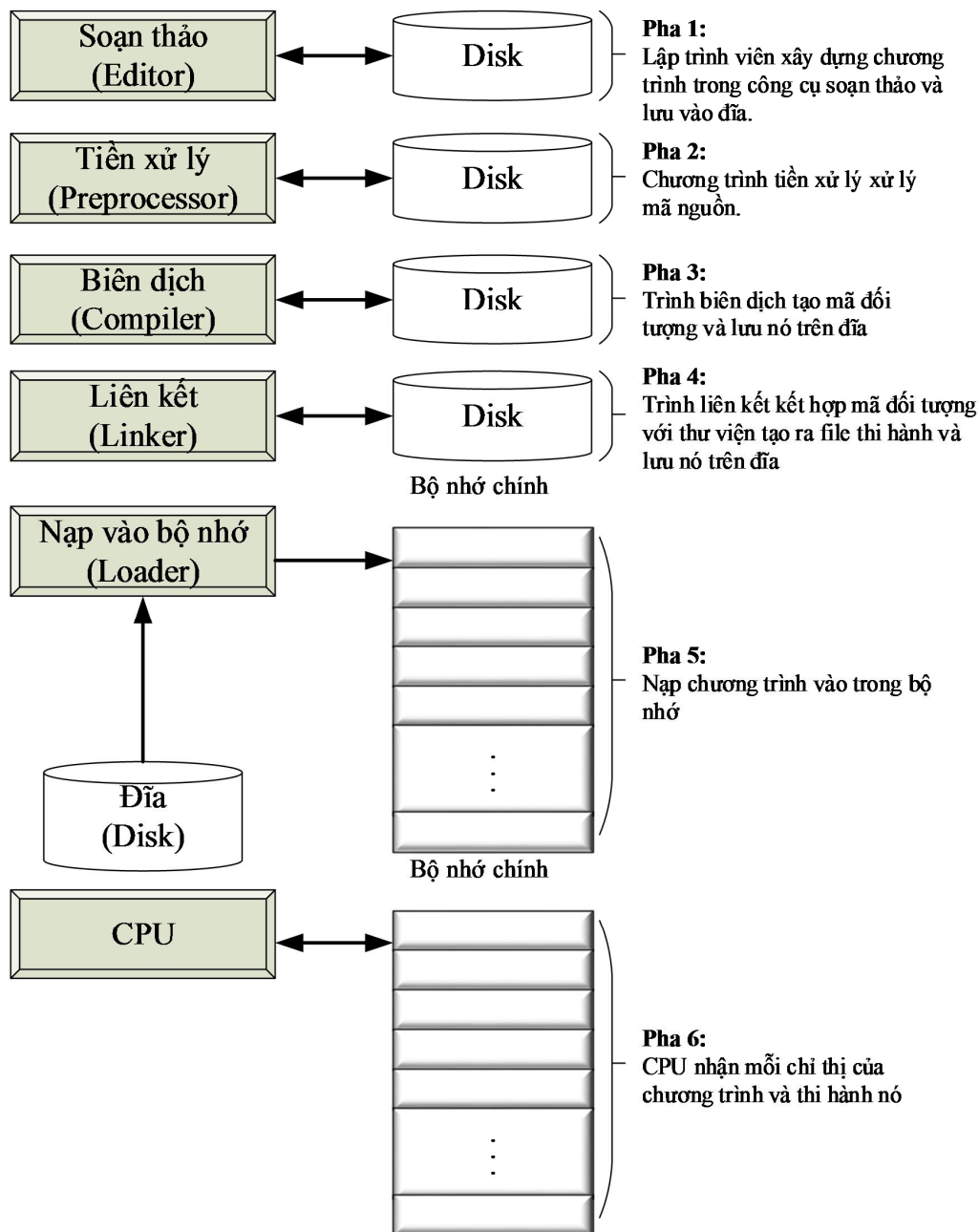
Trước khi chương trình có thể thi hành, đầu tiên nó phải được đặt trong bộ nhớ. Nó được thực hiện nhờ vào trình nạp bằng cách lấy hình ảnh của chương trình trên đĩa chuyển vào trong bộ nhớ. Các thành phần thêm từ các thư viện chia sẻ cũng được nạp vào để hỗ trợ chạy chương trình.

Pha 6: Thi hành

Cuối cùng, máy tính dưới sự điều khiển của CPU thi hành chương trình. Vấn đề có thể xuất hiện khi thi hành chương trình. Chương trình không phải luôn luôn làm việc đúng trong lần chạy thử đầu tiên. Mỗi pha trước có thể thất bại vì các lỗi khác nhau mà chúng tôi thảo luận trong suốt cuốn giáo trình này. Ví dụ, chương trình thi hành cố gắng thực hiện chia cho 0. Điều này sẽ nguyên nhân mà hệ thống sẽ thông báo lỗi. Khi đó, chúng ta phải quay lại pha soạn thảo và sửa lại lỗi chương trình rồi thực hiện lại các pha tiếp theo.

1.5 Lịch sử C và C++

Ngôn ngữ C++ phát triển từ ngôn ngữ C, trong đó C phát triển từ hai ngôn ngữ lập trình trước là ngôn ngữ BCPL và ngôn ngữ B. BCPL được phát triển vào năm 1967 bởi Martin Richards như một ngôn ngữ để viết các hệ thống phần mềm hệ điều hành và trình biên dịch cho các hệ thống điều hành. Ken Thompson đã mô hình nhiều tính năng trong ngôn ngữ B kế tiếp ngôn ngữ BCPL và sử dụng B để tạo ra phiên bản đầu tiên của hệ điều hành UNIX tại phòng thí nghiệm Bell vào năm 1970.



Hình 1.3: Các bước cơ bản để xây dựng một chương trình.

Ngôn ngữ C được phát triển từ B bởi Dennis Ritchie tại Bell Laboratories. C sử dụng nhiều khái niệm quan trọng của BCPL và B. C ban đầu được biết đến rộng rãi như là ngôn ngữ phát triển của hệ điều hành UNIX. Ngày nay, hầu hết các hệ điều hành được viết bằng C/C++. C có sẵn cho hầu hết các máy tính và phần cứng độc lập.

Ngôn ngữ C được thiết kế phù hợp với các máy tính. C sử dụng rộng rãi với nhiều máy tính khác nhau (các nền tảng phần cứng) có thể dẫn đến nhiều biến thể. Đây là một vấn đề đối với các nhà phát triển chương trình cần viết các chương trình có thể chạy trên nhiều nền tảng.

Cần thiết có một phiên bản tiêu chuẩn của C. Viện Tiêu chuẩn Quốc gia Hoa Kỳ (ANSI) đã phối hợp với Tổ chức Tiêu chuẩn Quốc tế (ISO) để chuẩn C trên toàn thế giới, tiêu chuẩn chung đã được công bố vào năm 1990 và gọi là chuẩn ANSI/ISO 9899:1990.

C99 là một chuẩn ANSI mới cho các ngôn ngữ lập trình C. Nó được phát triển từ ngôn ngữ

C để theo kịp sự phát triển mạnh mẽ của phần cứng và yêu cầu ngày càng cao của người dùng. C99 mang lại cho C nhiều sự thích hợp với C++. Để biết thêm thông tin về C và C99, tham khảo chi tiết trong cuốn sách [5]. Do ngôn ngữ C là một ngôn ngữ chuẩn, độc lập phần cứng, ngôn ngữ phổ biến, các ứng dụng viết bằng C có thể chạy với không có lỗi hoặc ít lỗi trên một phạm vi rộng.

Ngôn ngữ C++ mở rộng từ ngôn ngữ C, được phát triển bởi Bjarne Stroustrup vào đầu những năm 1980 tại Bell Laboratories. C++ cung cấp một số tính năng cải tiến mới từ ngôn ngữ C, nhưng quan trọng hơn, nó cung cấp khả năng lập trình hướng đối tượng.

Cuộc cách mạng diễn ra trong cộng đồng phần mềm. Xây dựng phần mềm một cách nhanh chóng, chính xác và kinh tế vẫn là một mục tiêu khó, và tại một thời điểm khi nhu cầu về phần mềm mới tốt hơn đang tăng lên. Các mục tiêu cơ bản là tái sử dụng các thành phần phần mềm đã được mô hình trong thế giới thực. Các nhà phát triển phần mềm đã phát hiện ra mô-đun, thiết kế hướng đối tượng có nhiều ưu điểm hơn so với các kỹ thuật lập trình cấu trúc phổ biến trước đây. Các chương trình hướng đối tượng dễ hiểu, chính xác và dễ sửa đổi.

1.6 Chương trình đầu tiên trong C++: In dòng văn bản

Phần này, chúng ta xem xét chương trình đơn giản in ra màn hình dòng văn ký tự trong hình 1.4. Đây là chương trình minh họa các đặc trưng quan trọng của ngôn ngữ lập trình C++. Chúng ta sẽ xem xét chi tiết.

```
1 // Text-printing program.
2
3 #include <iostream>
4
5 // function main begins program execution
6 int main()
7 {
8     std::cout << "Welcome to C++!\n"; // display message
9     return 0; // indicate that program ended successfully
10 }
```

Hình 1.4: Chương trình C++ đầu tiên.

Output chương trình Hình 1.4:

```
Welcome to C++!
```

Mỗi dòng bắt đầu bởi `//`, chỉ ra rằng phần còn lại của dòng là chú thích. Chúng ta thường chèn chú thích vào chương trình mã nguồn để giúp người khác đọc và hiểu được chúng. Chú thích không có tác dụng khi chương trình chạy. Chúng thường được bỏ qua bởi trình biên dịch C++. Bắt đầu chú thích với `//` được gọi là chú thích dòng đơn bởi vì nó kết thúc vào cuối dòng hiện thời. Chúng ta cũng có thể sử dụng chú thích nhiều dòng bắt đầu với `/*` và kết thúc với `*/`.

Dòng 3 là chỉ thị tiền xử lý, đây là thông báo tới bộ tiền xử lý C++. Dòng này bắt đầu với `#` được xử lý bằng bộ tiền xử lý trước khi chương trình được biên dịch. Dòng này thông báo với bộ tiền xử lý là bao gồm trong chương trình nội dung dòng vào ra trong file `<iostream>`. File này phải được sử dụng trong bất cứ chương trình mà xuất dữ liệu ra màn hình và nhập dữ liệu từ bàn phím sử dụng C++.

Dòng 4 đơn giản là dòng trống. Chúng ta sử dụng dòng trống, ký tự trắng, ký tự tab để làm cho chương trình dễ đọc hơn. Những ký tự như vậy được gọi chung là khoảng trắng. Các ký tự khoảng trắng thường được bỏ qua bởi trình biên dịch.

Dòng 5 là dòng chú thích đơn chỉ dẫn thi hành chương trình bắt đầu tại dòng tiếp theo.

Dòng 6 (**int main()**) là một phần của mọi chương trình C++. Dấu ngoặc sau **main** chỉ ra rằng **main** là một hàm. Chương trình C++ cơ bản bao gồm một hoặc nhiều hàm. Chính xác, một hàm trong mỗi chương trình C++ phải tên là **main**. Hình 1.4 bao gồm chỉ một hàm. Chương trình C++ bắt đầu thi hành tại hàm **main** ngay cả khi **main** không phải là hàm đầu tiên của chương trình. Từ khóa **int** bên trái hàm **main** chỉ ra rằng hàm **main** trả về giá trị nguyên.

Dấu ngoặc kép mở { (dòng 7) phải bắt đầu của thân mỗi hàm. Tương ứng là ngoặc kép đóng } (dòng 11) phải kết thúc tại mỗi thân của hàm. Dòng 8 chỉ thị cho máy tính thực hiện một hành động, đó là in ra xâu ký tự được chứa trong 2 dấu ngoặc kép. Toàn bộ dòng 8 bao gồm **std::cout**, toán tử **<<**, xâu ký tự **"Welcome to C++\n"** và dấu chấm phẩy **;** được gọi là câu lệnh. Mỗi câu lệnh C++ phải kết thúc bằng dấu chấm phẩy. Chú ý là chỉ thị tiền xử lý (như **#include**) thì không cần kết thúc bởi dấu chấm phẩy. Đầu vào và đầu ra trong C++ được thực hiện với các dòng ký tự. Vì vậy, khi câu lệnh trước được thi hành, nó sẽ gửi dòng ký tự **"Welcome to C++\n"** ra đối tượng dòng ra chuẩn (**std::cout**) mà dòng này kết nối với màn hình. Chúng ta sẽ thảo luận chi tiết nhiều tính chất của **std::cout** trong chương 2.

Bài tập

1. Hãy dùng các dấu hoa thị `*` để vẽ tên mình trên màn hình. Ví dụ

```

***          ***          *****          ***
***          ***          *****          ***
***          ***          ***              ***
***          ***          ***              ***
*** ***          ***              *****
*****          ***          *****

```

2. Hãy gõ lại và biên dịch chương trình sau thành file chạy `hello` (hoặc `hello.exe` trên Windows). Chương trình nhận tên người từ dòng lệnh và in ra câu chào "Hello, <tên người>".

```

#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello, " << argv[1] << endl;
    return 0;
}

```

Sau khi biên dịch chương trình trên, để chạy nó, bạn cần mở **cửa sổ dòng lệnh**.

- Trong Windows: nhấn phím cửa sổ + phím R và gõ `cmd` rồi ấn Enter.
- Trong Linux: chạy chương trình **Terminal**.

Trong cửa sổ dòng lệnh, bạn di chuyển đến thư mục chứa file chạy vừa biên dịch bằng lệnh `cd`. Sau đó, bạn chạy chương trình bằng lệnh

```
./hello Vinh (hoặc hello.exe Vinh trên Windows)
```

Bạn sẽ nhận được câu chào

```
Hello, Vinh
```


Chương 2

Một số khái niệm cơ bản trong C++

Trong chương này, chúng ta tập trung tìm hiểu các khái niệm cơ bản trong C++ như khai báo và thao tác biến, kiểu dữ liệu, biểu thức, ... thông qua một số chương trình C++. Từ đó cho phép bạn có thể xây dựng các chương trình được viết trên ngôn ngữ lập trình C++.

2.1 Khai báo biến và sử dụng biến

Dữ liệu được xử lý dùng trong chương trình gồm dữ liệu số và các ký tự. C++ và hầu hết các ngôn ngữ lập trình sử dụng các cấu trúc như các biến để đặt tên và lưu trữ dữ liệu. Biến là thành phần trung tâm của ngôn ngữ lập trình C++. Bên cạnh việc các chương trình phải có cấu trúc rõ ràng, một số đặc điểm mới sẽ được đưa ra để giải thích.

2.1.1 Biến

Một biến trong ngôn ngữ C++ có thể lưu trữ số hoặc dữ liệu thuộc các kiểu khác nhau. Ta tập trung vào biến dạng số. Các biến này có thể được viết ra và có thể thay đổi.

```
1 //Chương trình minh họa
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     int number_of_bars;
7     double one_weight, total_weight;
8
9     cout << "Enter the number of candy bars in a package\n";
10    cout << "and the weight in ounces of one candy bar.\n";
11    cout << "Then press return.\n";
12    cin >> number_of_bars;
13    cin >> one_weight;
14
15    total_weight = one_weight * number_of_bars;
16
17    cout << number_of_bars << " candy bars\n";
18    cout << one_weight << " ounces each\n";
19    cout << "Total weight is " << total_weight << " ounces.\n";
20
```

```

21     cout << "Try another brand.\n";
22     cout << "Enter the number of candy bars in a package\n";
23     cout << "and the weight in ounces of one candy bar.\n";
24     cout << "Then press return.\n";
25     cin >> number_ofBars;
26     cin >> one_weight;
27
28     total_weight = one_weight * number_ofBars;
29
30     cout << number_ofBars << " candy bars\n";
31     cout << one_weight << " ounces each\n";
32     cout << "Total weight is " << total_weight << "      ounces.\n";
33
34     cout << "Perhaps an apple would be healthier.\n";
35
36     return 0;
37 }

```

Hình 2.1: Chương trình minh họa thao tác với biến trong C++.

Trong ví dụ 2.1, `number_ofBars`, `one_weight`, và `total_weight` là các biến. Chương trình được chạy với đầu vào thể hiện trong các đối thoại mẫu, `number_ofBars` đã thiết lập giá trị 11 trong câu lệnh.

```
cin >> number_ofBars;
```

giá trị của biến `number_ofBars` được thay đổi đến 12 khi câu lệnh sao chép thứ hai được thực hiện.

Trong các ngôn ngữ lập trình, biến được thực hiện như địa chỉ trong bộ nhớ. Trình biên dịch sẽ gán một địa chỉ trong bộ nhớ (đề cập trong Chương 1) cho mỗi tên biến trong chương trình. Các giá trị của biến, một hình thức được mã hóa bao gồm bit 0 và 1, được lưu trữ theo địa chỉ bộ nhớ được gán cho biến đó. Ví dụ, ba biến trong ví dụ trong hình 2.1 có thể được gán địa chỉ trong bộ nhớ là 1001, 1003, và 1007. Các con số chính xác sẽ phụ thuộc vào máy tính, trình biên dịch và các yếu tố khác. Trình biên dịch sẽ lựa chọn giá trị cho các biến trong chương trình, có thể biểu diễn các địa chỉ trong bộ nhớ được gán qua các tên biến.

2.1.2 Tên hay định danh

Điều đầu tiên bạn có thể nhận thấy về tên của các biến trong ví dụ là dài hơn những tên thường dùng trong các lớp về toán học. Để làm cho chương trình dễ hiểu, nên sử dụng tên có ý nghĩa cho các biến. Tên của biến (hoặc các đối tượng khác được xác định trong một chương trình) được gọi là định danh.

Một định danh phải bắt đầu bằng chữ cái hoặc dấu `_`, và tất cả phần còn lại là chữ cái, chữ số, hoặc dấu `_`. Ví dụ, các định danh sau là hợp lệ:

```

x      x1      x_1      _abc      ABC123z7      sum
RATE   count    data2    Big_Bonus

```

Tất cả những cái tên được đề cập trước đó là hợp lệ và trình biên dịch chấp nhận, năm tên đầu tiên định danh kém vì không phải mô tả sử dụng định danh. Những định danh sau đây là không hợp lệ và không được trình biên dịch chấp nhận:

```
12  3X    % change
```

```
data-1    myfirst.c    PROG.CPP
```

Ba định danh đầu không được phép vì không bắt đầu bằng chữ cái hoặc dấu `_`. Ba định danh còn lại chứa các ký hiệu khác với chữ cái, chữ số và dấu `_`.

C++ là một ngôn ngữ lập trình chặt chẽ phân biệt giữa chữ hoa và chữ thường. Do đó ba định danh sau riêng biệt và có thể sử dụng để đặt tên cho ba biến khác nhau:

```
rate    RATE    Rate
```

Tuy nhiên, đây không phải là ý tưởng tốt để sử dụng trong cùng một chương trình vì có thể gây khó hiểu. Mặc dù nó không phải là yêu cầu của C++, các biến thường được viết với chữ thường. Các định danh được định nghĩa trước như: `main`, `cin`, `cout`, ... phải được viết bằng chữ thường.

Một định danh C++ có thể có chiều dài tùy ý, mặc dù một số trình biên dịch sẽ bỏ qua tất cả các ký tự sau một số quy tắc và số lượng lớn các ký tự khởi tạo ban đầu.

Có một lớp đặc biệt của định danh, gọi là từ khóa được định nghĩa sẵn trong C++ và không thể sử dụng để đặt tên cho biến hoặc dùng vào công việc khác. Các từ khóa được viết theo các cách khác nhau như: `int`, `double`. Danh sách các từ khóa được đưa ra trong Phụ lục 1.

Bạn có thể tự hỏi tại sao những từ khác, chúng định nghĩa như là một phần của ngôn ngữ C++ lại không phải là từ khóa. Những gì về những từ như `cin` và `cout`? Câu trả lời là bạn được phép xác định lại những từ này, mặc dù nó sẽ là khó hiểu để làm như vậy. Những từ này được xác định trước là không phải từ khóa. Tuy nhiên, chúng được định nghĩa trong thư viện theo yêu cầu của tiêu chuẩn ngôn ngữ C++.

Chúng tôi sẽ thảo luận về các thư viện sau trong cuốn sách này. Để bây giờ, bạn không cần phải lo lắng về thư viện. Không cần phải nói, việc dùng một định danh đã được xác định trước cho bất cứ điều gì khác hơn ý nghĩa tiêu chuẩn của nó có thể gây nhầm lẫn và nguy hiểm, và do đó nên được tránh.

Khai báo biến

Mỗi biến trong chương trình C++ phải được khai báo. Khi bạn khai báo một biến nghĩa là cho trình biên dịch biết và máy tính hiểu loại dữ liệu bạn sẽ được lưu trữ trong các biến. Ví dụ, hai khai báo sau của ví dụ 2.1 khai báo 3 biến được sử dụng trong chương trình:

```
int    number_ofBars;
double one_weight, total_weight;
```

Khi có nhiều hơn một biến trong khai báo, các biến cách nhau bởi dấu phẩy. Khai báo kết thúc bằng dấu chấm phẩy.

Từ `int` ở dòng đầu khai báo số nguyên. Khai báo `number_ofBars` là một biến kiểu `int`. Giá trị của `number_ofBars` phải là một số nguyên, như 1, 2, -1, 0, 37, hoặc -288.

Từ `double` ở dòng thứ hai khai báo `one_weight` và `total_weight` là biến kiểu `double`. Biến kiểu `double` có thể lưu giữ các con số với phần lẻ sau dấu thập phân (số dấu chấm động), như 1,75 hoặc -0,55. Các loại dữ liệu được tổ chức trong biến được gọi là kiểu và tên kiểu, như `int` hoặc `double`, được gọi là tên kiểu.

Mỗi biến trong một chương trình C++ phải được khai báo trước khi sử dụng. Có hai cách để khai báo biến: ngay trước khi sử dụng hoặc ngay sau khi bắt đầu hàm `main` của chương trình.

```
int main ()
{
```

Điều này làm cho chương trình rõ ràng hơn.

Khai báo biến

Tất cả các biến phải được khai báo trước khi sử dụng.

Cú pháp để khai báo biến như sau:

```
Type_name Variable_Name_1, Variable_Name_2, ...;
```

Ví dụ:

```
int count, number_of_dragons, number_of_trolls;  
double distance;
```

Khai báo biến cung cấp thông tin cho trình biên dịch để biết thể hiện của các biến. Trình biên dịch thể hiện các biến như bộ nhớ địa phương và giá trị của biến được gán cho biến đó. Các giá trị được mã hoá như các bit 0 và 1. Các kiểu khác nhau của biến yêu cầu kích thước trong bộ nhớ khác nhau và phương pháp khác nhau để mã hóa các giá trị các bit 0 và 1. Việc khai báo biến sẽ cho phép trình biên dịch phân bổ vị trí bộ nhớ, kích thước bộ nhớ cho các biến này để sử dụng trong chương trình.

2.1.3 Câu lệnh gán

Cách trực tiếp nhất để thay đổi giá trị của một biến là sử dụng câu lệnh gán. Một câu lệnh gán là một thứ tự để các máy tính biết, “thiết lập giá trị của biến này với những gì đã viết ra”. Các dòng sau trong chương trình 2.1 là một ví dụ về một câu lệnh gán

```
total_weight = one_weight * number_of_bars;
```

Khai báo này thiết lập giá trị của `total_weight` là tích của `one_weight` và `number_of_bars`. Một câu lệnh gán luôn bao gồm một biến phía bên trái dấu bằng và một biểu thức ở bên tay phải. Câu lệnh gán kết thúc bằng dấu chấm phẩy. Phía bên phải của dấu bằng có thể là một biến, một số, hoặc một biểu thức phức tạp hơn của các biến, số, và các toán tử số học như `*` và `+`. Một lệnh gán chỉ thị máy tính tính giá trị các biểu thức ở bên phải của dấu bằng và thiết lập giá trị của các biến ở phía bên trái dấu bằng với giá trị được tính.

Có thể sử dụng bất kỳ toán tử số học để thay phép nhân. Ví dụ, câu lệnh gán giá trị:

```
total_weight = one_weight + number_of_bars;
```

Câu lệnh này cũng giống như câu lệnh gán trong ví dụ mẫu, ngoại trừ việc nó thực hiện phép cộng chứ không phải nhân. Khai báo này thay đổi giá trị của `total_weight` bằng tổng giá trị của `one_weight` và `number_of_bars`. Nếu thực hiện thay đổi này trong chương trình hình 2.1, chương trình sẽ cho giá trị không đúng với mục đích, nhưng nó vẫn chạy.

Trong một câu lệnh gán, biểu thức ở bên phải của dấu bằng đơn giản có thể là một biến. Khai báo:

```
total_weight = one_weight;
```

thay đổi giá trị của `total_weight` giống giá trị của biến `one_weight`.

Nếu sử dụng trong chương trình hình 2.1, sẽ cho các giá trị không chính xác thấp hơn giá trị của `total_weight`.

Câu lệnh gán sau thay đổi giá trị của `number_of_bars` thành 37:

```
number_ofBars = 37;
```

Số 37 trong ví dụ gọi là một hằng số, không giống như biến, giá trị của nó không thể thay đổi. Các biến có thể thay đổi giá trị và phép gán là cách để thay đổi. Trước hết, các biểu thức ở bên phải của dấu bằng được tính toán, sau đó giá trị biến ở bên trái được gán bằng giá trị được tính toán ở bên phải. Nghĩa là, biến có thể ở cả hai bên của toán tử gán. Ví dụ, xét các câu lệnh gán:

```
number_ofBars = number_ofBars + 3;
```

Giá trị thực là “Giá trị của `number_ofBars` bằng với giá trị của `number_ofBars` cộng với ba” hay “Giá trị mới `number_ofBars` bằng với giá trị cũ của `number_ofBars` cộng với ba”. Dấu bằng trong C++ không được sử dụng theo nghĩa dấu bằng trong ngôn ngữ thông thường hoặc theo nghĩa đơn giản trong toán học.

Câu lệnh gán

Trong một khai báo, biểu thức đầu tiên ở bên phải của dấu bằng được tính toán, sau đó biến ở bên trái của dấu bằng được thiết lập với giá trị này.

Cú pháp

Biến = biểu thức;

Ví dụ

```
distance = rate * time;
```

```
count = count + 2;
```

2.2 Vào ra dữ liệu

Đối với chương trình C++ có nhiều cách để nhập và xuất dữ liệu. Ở đây, chúng ta sẽ mô tả cách gọi là luồng (**stream**). Một luồng nhập (**input stream**) được hiểu đơn giản là một dòng dữ liệu được đưa vào máy tính để sử dụng. Luồng cho phép chương trình xử lý dữ liệu đầu vào theo cùng một cách như nhau, bất kể chúng được nhập vào bằng hình thức nào. Luồng chỉ tập trung vào dòng dữ liệu mà không quan tâm đến nguồn gốc của dữ liệu. Trong phần này, chúng ta giả định dữ liệu được nhập vào bằng bàn phím và xuất ra màn hình. Trong chương 8, chúng ta sẽ tìm hiểu thêm về xuất và nhập dữ liệu từ tệp tin.

2.2.1 Xuất dữ liệu với `cout`

`cout` cho phép xuất ra màn hình giá trị của biến cũng như các chuỗi văn bản. Có nhiều kết hợp bất kỳ giữa biến và chuỗi văn bản để có thể xuất ra. Ví dụ: xem câu lệnh trong chương trình ở phần 2.1

```
cout << number_ofBars << " candy bars\n";
```

Câu lệnh này cho phép máy tính xuất ra màn hình hai mục: giá trị của biến `number_ofBars` và cụm từ trích dẫn "candy bars\n". Lưu ý rằng, bạn không cần thiết phải lặp lại câu lệnh `cout` cho mỗi lần xuất dữ liệu ra. Bạn chỉ cần liệt kê tất cả các dữ liệu đầu ra với biểu tượng mũi tên << phía trước. Câu lệnh `cout` ở trên tương đương với hai câu lệnh `cout` ở dưới đây:

```
cout << number_ofBars;  
cout << "candy bars\n";
```

Bạn có thể đưa công thức toán học vào câu lệnh `cout` được thể hiện ở ví dụ dưới đây, trong đó `price` và `tax` là các biến

```
cout << "The total cost is $" << (price + tax);
```

Đối với các biểu thức toán học như `price + tax` trình biên dịch yêu cầu phải có dấu ngoặc đơn.

Hai biểu tượng `<` được đánh sát nhau không có dấu cách và được gọi là **toán tử chèn**. Toàn bộ câu lệnh `cout` kết thúc bằng dấu chấm phẩy.

Nếu có hai lệnh `cout` cùng một dòng, bạn có thể kết hợp chúng lại thành một lệnh `cout` dài hơn. Ví dụ, hãy xem xét các dòng sau từ hình 2.1

```
cout << number_of_bars << " candy bars\n";
cout << one_weight << " ounces each\n";
```

Hai câu lệnh này có thể được viết lại thành một câu lệnh đơn và chương trình vẫn thực hiện chính xác như câu lệnh cũ

```
cout << number_of_bars << " candy bars\n" << one_weight
<< " ounces each\n";
```

Bạn nên tách câu lệnh thành hai hoặc nhiều dòng thay vì một câu lệnh dài để giữ cho câu lệnh không bị chạy khỏi màn hình.

```
cout << number_of_bars << " candy bars\n"
<< one_weight << " ounces each\n";
```

Bạn không cần phải cắt ngang chuỗi trích dẫn thành hai dòng, mặt khác, bạn có thể bắt đầu dòng mới của bạn ở bất kỳ chỗ nào trống. Những khoảng trống và ngắt dòng hợp lý sẽ được máy tính chấp nhận như trong ví dụ ở trên.

Bạn nên sử dụng từng lệnh `cout` cho từng nhóm dữ liệu đầu ra. Chú ý rằng chỉ có một dấu chấm phẩy cho một lệnh `cout`, ngay cả với những lệnh kéo dài.

Từ ví dụ đầu ra trong hình 2.1, cần chú ý rằng chuỗi trích dẫn phải có ngoặc kép. Đây là một ký tự ngoặc kép trên bàn phím, chứ không sử dụng hai ngoặc đơn để tạo thành ngoặc kép. Bên cạnh đó, cần chú ý ngoặc kép cũng được sử dụng để kết thúc chuỗi. Đồng thời, không có sự phân biệt giữa ngoặc trái và ngoặc phải.

Cũng cần chú ý đến khoảng cách bên trong các chuỗi trích dẫn. Máy tính không chèn thêm bất kỳ khoảng cách nào trước hoặc sau dòng dữ liệu ra bằng câu lệnh `cout`. Vì vậy, chuỗi trích dẫn mẫu thường bắt đầu và/hoặc kết thúc với một dấu cách. Dấu cách giữ cho các chuỗi ký tự và số có thể xuất hiện cùng nhau. Nếu bạn muốn có khoảng trống mà các chuỗi trích dẫn không có thì bạn có thể đặt thêm vào đó một chuỗi chỉ có khoảng trống như ví dụ dưới đây:

```
cout << first_number << " " << second_number;
```

Như đã nói ở chương 1, `\n` cho biết chúng ta sẽ bắt đầu một dòng xuất mới. Nếu bạn không sử dụng `\n` để xuống dòng, máy tính sẽ xuất dữ liệu trên cùng một dòng. Phụ thuộc vào cách cài đặt màn hình, dữ liệu xuất ra sẽ bị ngắt một cách tùy ý và chạy ra khỏi màn hình. Chú ý rằng `\n` phải được đặt trong chuỗi trích dẫn. Trong C++, lệnh xuống dòng được coi như một ký tự đặc biệt vì vậy nó được đặt ở trong chuỗi trích dẫn và không có dấu cách giữa hai ký tự `\` và `n`. Mặc dù có hai ký tự nhưng C++ chỉ coi `\n` như một ký tự duy nhất, gọi là ký tự xuống dòng.

2.2.2 Chỉ thị biên dịch và không gian tên

Chúng ta bắt đầu chương trình với 2 dòng sau đây:

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

Hai dòng ở trên cho phép người lập trình sử dụng thư viện `iostream`. Thư viện này bao gồm định danh của `cin` và `cout` cũng như nhiều định danh khác. Bởi vậy, nếu chương trình của bạn sử dụng `cin` và/hoặc `cout`, bạn nên thêm 2 dòng này khi bắt đầu mỗi tệp chứa chương trình của bạn.

Dòng dưới đây được xem là một “chỉ thị bao gồm”. Nó “bao gồm” thư viện `iostream` trong chương trình của bạn, vì vậy người sử dụng có thể dùng `cin` và `cout`:

```
#include <iostream>
```

Toán tử `cin` và `cout` được định danh trong một tệp `iostream` và dòng phía trên chỉ tương đương với việc sao chép tập tin chứa định danh vào chương trình của bạn. Dòng thứ hai tương đối phức tạp để giải thích về nó.

C++ chia các định danh vào các “không gian tên (namespace)”. Không gian tên là tập hợp chứa nhiều các định danh, ví dụ như `cin` và `cout`. Câu lệnh chỉ định không gian tên như ví dụ trên được gọi là sử dụng chỉ thị.

```
using namespace std;
```

Việc sử dụng chỉ thị cụ thể cho biết chương trình của bạn đang sử dụng không gian tên `std` (không gian tên chuẩn). Tức là những định danh mà bạn sử dụng được nhận diện trong không gian tên là `std`. Trong trường hợp này, điều quan trọng là khi những đối tượng như `cin` và `cout` được định danh trong `iostream`, các định danh của chúng cho biết chúng nằm trong không gian tên `std`. Vì vậy để sử dụng chúng, bạn cần báo với trình biên dịch bạn đang sử dụng không gian tên `std`.

Lý do C++ có nhiều không gian tên là do có nhiều đối tượng cần phải đặt tên. Do đó, đôi khi có hai hoặc nhiều đối tượng có thể có cùng tên gọi, điều đó cho thấy có thể có hai định danh khác nhau cho cùng một tên gọi. Để giải quyết vấn đề này, C++ phân chia những dữ liệu thành các tuyển tập, nhờ đó có thể loại bỏ việc hai đối tượng trong cùng một tuyển tập (không gian tên) bị trùng lặp tên.

Chú ý rằng, không gian tên không đơn giản là tuyển tập các định danh. Nó là phần thân của chương trình C++ nhằm xác định ý nghĩa của một số đối tượng, ví dụ như một số định danh hoặc/và khai báo. Chức năng của không gian tên chia tất cả các định danh của C++ thành nhiều tuyển tập, từ đó, mỗi định danh chỉ có một nhận dạng trong không gian tên.

Một số phiên bản C++ sử dụng chỉ dẫn như ở dưới. Đây là phiên bản cũ của “chỉ dẫn bao gồm” (không sử dụng không gian tên)

```
#include <iostream.h>
```

Nếu trình biên dịch của bạn không chạy với dòng chỉ dẫn:

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

thì thử sử dụng dòng chỉ dẫn dưới đây để thay thế:

```
#include <iostream.h>
```

Nếu trình biên dịch của bạn yêu cầu `iostream.h` thay vì `iostream`, thì bạn đang sử dụng một trình biên dịch phiên bản cũ và bạn nên có một trình biên dịch phiên bản mới hơn.

2.2.3 Các chuỗi Escape

Có nhiều kí tự được dùng cho các nhiệm vụ đặc biệt như dấu `'` (cho biểu diễn kí tự), dấu `"` (cho biểu diễn chuỗi). Các kí tự này nếu xuất hiện trong một số trường hợp sẽ gây lỗi, ví dụ để gán biến `letter` là kí tự `'` (single quote) ta không thể viết: `letter = ''`; vì dấu nháy đơn được hiểu như kí hiệu bao lấy kí tự. Tương tự câu lệnh: `cout << "This is double quote ("");` cũng sai. Để có thể biểu diễn được các kí tự này (cũng như các kí tự điều khiển không có mặt chữ, như kí tự xuống dòng) ta dùng cơ chế “thoát” bằng cách thêm kí hiệu `\` vào phía trước. Các dấu gạch chéo ngược, `\`, viết liền trước một ký tự cho biết các ký tự này không có ý nghĩa giống thông thường. Như vậy, các câu lệnh trên cần được viết lại:

```
letter = '\\';
cout << "This is double quote (\\)";
```

Và đến lượt mình, do dấu `\` được trưng dụng để làm nhiệm vụ đặc biệt như trên, nên để biểu thị `\` ta cần phải viết `\\`.

Chuỗi gồm dấu `\` đi liền cùng một kí tự bất kỳ, được gọi là chuỗi thoát. Sau `\` có thể là một kí tự bất kỳ, nếu kí tự này chưa qui định ý nghĩa thoát thì theo tiêu chuẩn ANSI hành vi của các chuỗi này là không xác định. Từ đó, một số trình biên dịch đơn giản bỏ qua dấu `\` và vẫn xem kí tự với ý nghĩa gốc, còn một số khác có thể “hiểu nhầm” và gây hiệu ứng không tốt. Vì vậy, bạn chỉ nên sử dụng các chuỗi thoát đã được cung cấp. Chúng tôi liệt kê một số chuỗi ở đây.

Thuật ngữ	Ký hiệu	Ý nghĩa
new line	<code>\n</code>	xuống dòng
horizontal tab	<code>\t</code>	dịch chuyển con trỏ một số dấu cách
alert	<code>\a</code>	tiếng chuông
backslash	<code>\\</code>	dấu <code>\</code>
single quote	<code>\'</code>	dấu <code>'</code>
double quote	<code>\"</code>	dấu <code>"</code>

2.2.4 Nhập dữ liệu với `cin`

Bạn sử dụng `cin` để nhập dữ liệu ít nhiều tương tự cách mà bạn sử dụng `cout` để xuất dữ liệu. Cú pháp là tương tự, trừ việc `cin` được thay thế cho `cout` và `<<` được thay bằng `>>`. Chương trình trong hình 2.1, biến `number_of_bars` và `one_weight` được nhập vào với lệnh `cin` như sau:

```
cin >> number_of_bars;
cin >> one_weight;
```

cũng tương tự `cout`, bạn có thể gộp hai dòng lệnh trên thành một và viết trên một dòng:

```
cin >> number_of_bars >> one_weight;
```

hoặc trên hai dòng liên tiếp nhau:

```
cin >> number_of_bars
    >> one_weight;
```




Và cũng chú ý với mỗi cin chỉ có một dấu chấm phẩy.




Cách nhập dữ liệu với >>

Khi gặp câu lệnh cin chương trình sẽ chờ bạn nhập dãy giá trị vào từ bàn phím và đặt giá trị của biến thứ nhất với giá trị thứ nhất, biến thứ hai với giá trị thứ hai ... Tuy nhiên, chỉ sau khi bạn nhấn Enter chương trình mới nhận lấy dòng dữ liệu nhập và phân bố giá trị cho các biến. Điều này có nghĩa bạn có thể nhập tất cả giá trị cho các biến (trong một hoặc nhiều câu lệnh cin >>) cùng một lần và chỉ với một dấu Enter), điều này tạo thuận lợi cho NSD kịp thời sửa chữa, xóa, bổ sung dòng dữ liệu nhập (nếu có sai sót) trước khi nhấn Enter. Các giá trị nhập cho các biến phải được cách nhau bởi ít nhất một dấu trắng (là dấu cách, dấu tab hoặc thậm chí là dấu xuống dòng – enter). Ví dụ cần nhập các giá trị 12 và 5 cho các biến number_of_bars và one_weight thông qua câu lệnh:

```
cin >> number_of_bars >> one_weight;
```


Có thể nhập

```
12 5 
12 
5 
```


Chương trình sẽ bỏ qua các dấu  , dấu  , dấu  và gán 12 cho number_of_bars và 5 cho one_weight.

Vì chương trình bỏ qua không gán các dấu trắng cho biến (kể cả biến xâu kí tự) nên giả sử candy_mark là xâu kí tự và ta có câu lệnh:

```
cin >> number_of_bars >> one_weight >> candy_mark;
```

và dòng nhập: 12 5 peanut candy 

thì biến candy_mark chỉ nhận được giá trị: "peanut" thay vì "peanut candy". Để xâu nhận được đầy đủ thông tin đã nhập ta cần lệnh nhập khác đối với xâu (xem chương 5).

Khi NSD nhập vào dãy byte nhiều hơn cần thiết để gán cho các biến thì số byte còn lại và kể cả dấu xuống dòng (nhập bằng phím ) sẽ nằm lại trong cin. Các byte này sẽ tự động gán cho các biến trong lần nhập sau mà không chờ NSD gõ thêm dữ liệu vào từ bàn phím. Ví dụ:

```
#include <iostream>
using namespace std;
int main( )
{
    char my_name;
    int my_age;
    cout << "Enter data: ";
    cin >> my_name >> my_age; // Gia su nhap A 15 B 16
    cout << "My name is " << my_name;
    cout << " and I am " << my_age << " years old.\n";

    char your_name;
    int your_age;
    cout << "Enter data: ";
    cin >> your_name >> your_age;
    cout << "Your name is " << your_name;
```

```

    cout << " and you is " << your_age << " years old.\n";
    return 0;
}

```

Chương trình trên gồm hai đoạn lệnh giống nhau, một nhập tên, tuổi và in ra màn hình cho nhân vật tôi, và đoạn còn lại cũng thực hiện giống hệt vậy cho nhân vật bạn. Giả sử đáp ứng lệnh nhập đầu tiên, NSD nhập: A 15 B 16 thì chương trình sẽ in luôn ra kết quả như hình dưới mà không cần chờ nhập cho lệnh nhập thứ hai. Dưới đây là output của chương trình trên

```

Enter data: A 15 B 16
My name is A and I am 15 years old.
Enter data: Your name is B and you is 16 years old.

```

Thông báo trước khi nhập dữ liệu (kết hợp cout với cin)

Khi gặp lệnh nhập dữ liệu chương trình chỉ đơn giản dừng lại chờ nhưng không tự động thông báo trên màn hình, do vậy ta cần “nhắc nhở” NSD nhập dữ liệu (số lượng, loại, kiểu ... cho biến nào ...) bằng các câu lệnh cout << đi kèm phía trước. Ví dụ:

```

cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press return.\n";
cin >> number_of_bars >> one_weight;

```

hoặc

```

cout << "Enter your name and age: ";
cin >> your_name >> your_age;

```

2.3 Kiểu dữ liệu và biểu thức

2.3.1 Kiểu int và kiểu double

Về mặt khái niệm thì hai số 2 và 2.0 đều cùng một số. Nhưng trong C++ hai số đó là hai số có kiểu dữ liệu khác nhau. Số 2 thuộc kiểu int, số 2.0 kiểu double vì có chứa phần thập phân (mặc dù phần thập phân có giá trị 0). Toán học trong lập trình máy tính hơi khác với toán học thông thường bởi vì các vấn đề thực tế trong máy tính đã làm cho các số này khác với các định nghĩa trừu tượng. Hầu hết các kiểu số trong C++ đáp ứng đủ các giá trị số để tính toán. Kiểu int không có gì đặc biệt, nhưng với giá trị của kiểu double có nhiều vấn đề bởi vì kiểu double bị giới hạn bởi số các chữ số, do đó máy tính chỉ lưu được giá trị xấp xỉ của số kiểu double. Các số kiểu int sẽ được lưu đúng giá trị. Độ chính xác của số kiểu double được lưu khác nhau trên các máy tính khác nhau, tuy nhiên chúng ta vẫn mong muốn các số đó lưu với độ chính xác ít nhất là 14 chữ số. Để đáp ứng các ứng dụng thì độ chính xác này là chưa đủ, mặc dù các vấn đề có thể xảy ra ngay cả với những trường hợp đơn giản. Do đó, nếu chúng ta biết trước được giá trị của các biến sử dụng là các số nguyên thì tốt nhất nên sử dụng kiểu dữ liệu int.

Kiểu double là gì?

Tại sao số có phần thập phân được gọi là **double**? Với kiểu dữ liệu "single" thì giá trị có bằng một nửa? Không, nhưng có một số thứ gần giống như vậy. Rất nhiều ngôn ngữ lập trình truyền thống sử dụng hai kiểu dữ liệu cho các số thập phân. Một kiểu sử dụng lưu trữ ít tốn bộ nhớ nhưng độ chính xác thấp (không cho phép sử dụng quá nhiều chữ số thập phân). Dạng thứ hai sử dụng gấp đôi dung lượng bộ nhớ và do đó chính xác hơn và cũng cho phép sử dụng số có giá trị lớn hơn (mặc dù người lập trình quan tâm nhiều đến độ chính xác hơn là kích thước bộ nhớ). Các số sử dụng gấp đôi kích thước bộ nhớ được gọi là số có độ chính xác kép; các số sử dụng ít bộ nhớ được gọi là số có độ chính xác đơn. Theo cách gọi này thì số có độ chính xác kép được gọi trong C++ là số **double**. Các số có độ chính xác đơn gọi là số **float**. C++ cũng có số dạng thứ ba gọi là **long double**, các số này được mô tả trong phần "Các kiểu dữ liệu khác". Tuy nhiên, chúng ta sẽ ít sử dụng kiểu **float** và **long double** trong cuốn sách này.

Các giá trị hằng số kiểu **double** được viết khác với các hằng kiểu **int**. Các giá trị hằng kiểu **int** không chứa các số thập phân. Nhưng các giá trị hằng kiểu **double** cần viết cả phần nguyên và phần thập phân (ví dụ 2.1, 2.0 ...). Dạng thức viết đơn giản của các giá trị hằng **double** giống như chúng ta viết các số thực hàng ngày. Khi viết dạng này, giá trị hằng **double** phải chứa cả phần thập phân.

Một cách viết phức tạp hơn của các giá trị hằng kiểu **double** gọi là ký hiệu khoa học hay ký hiệu dấu phẩy động để viết cho các số rất lớn hoặc rất bé. Ví dụ:

3.67×10^{17}

tương đương với

367000000000000000.0

và được biểu diễn trong C++ giá trị **3.67e17**. Với số

5.89×10^{-6}

tương đương với

0.0000589

và được biểu diễn trong C++ giá trị **5.89e-6**. Chữ **e** viết tắt của exponent và có nghĩa là số mũ của lũy thừa 10.

Ký hiệu **e** được sử dụng vì các phím trên bàn phím không thể biểu diễn các số ở bên trên mũ. Số đi sau chữ **e** chỉ cho ta hướng và số chữ số cần dịch chuyển dấu thập phân. Ví dụ, để thay đổi số **3.49e4** thành số không chứa ký tự **e**, ta di chuyển dấu thập phân sang bên phải 4 chữ số và ta được 34900.0, đây là cách viết khác của số ban đầu. Nếu như số sau ký tự **e** là số âm, ta di chuyển sang trái, thêm vào các số 0 nếu cần thiết. Do đó, số **3.49e-2** tương đương với 0.0349.

Giá trị trước ký tự **e** có thể chứa cả phần thập phân hoặc không. Nhưng giá trị sau ký tự **e** bắt buộc là giá trị không chứa phần thập phân.

Khi các máy tính bị giới hạn về kích thước bộ nhớ thì các số cũng được lưu trữ với số bytes giới hạn. Do đó, với mỗi kiểu dữ liệu sẽ có một giới hạn miền giá trị nhất định. Giá trị lớn nhất của kiểu **double** lớn hơn giá trị lớn nhất của kiểu **int**. Các trình biên dịch C++ cho phép giá trị lớn nhất của kiểu **int** là 2,147,483,647 và giá trị của kiểu **double** lên tới 10^{308} .

2.3.2 Các kiểu số khác

Trong C++ còn có các kiểu dữ liệu số khác ngoài kiểu `int` và `double`, các kiểu dữ liệu số này được trình bày trong bảng 2.2. Các kiểu dữ liệu này có miền giá trị số và độ chính xác khác nhau (tương ứng với nhiều hoặc ít số các chữ số thập phân). Trong bảng 2.2, mỗi kiểu dữ liệu mô tả đi kèm với kích thước bộ nhớ, miền giá trị và độ chính xác. Các giá trị này thay đổi trên các hệ thống khác nhau.

Mặc dù có một số kiểu dữ liệu được viết bằng hai từ, chúng ta vẫn khai báo các biến thuộc kiểu này giống như với kiểu `int` và `double`. Ví dụ sau đây khai báo một biến có kiểu `long double`:

```
long double big_number;
```

Kiểu dữ liệu `long` và `long int` là hai tên cho cùng một kiểu. Do đó, hai khai báo sau là tương đương:

```
long big_total;
```

tương đương với

```
long int big_total;
```

Trong một chương trình, bạn có thể chỉ sử dụng một trong hai kiểu khai báo trên cho biến `big_total`, nhưng chương trình không quan tâm bạn sử dụng kiểu dữ liệu nào. Do đó, kiểu dữ liệu `long` tương đương với `long int`, nhưng không tương đương với `long double`.

Các kiểu dữ liệu cho số nguyên như `int` và các kiểu tương tự được gọi là các kiểu số nguyên. Kiểu dữ liệu cho số có phần thập phân như kiểu `double` và một số kiểu tương tự gọi là các kiểu số thực (kiểu dấu phẩy động). Các kiểu này được gọi là kiểu dấu phẩy động bởi vì máy tính lưu số tương ứng với khi viết, ví dụ số 392.123, đầu tiên sẽ chuyển sang dạng ký hiệu e ta được 3.92123e2. Khi máy tính thực hiện biến đổi này, dấu phẩy động đã được dịch chuyển sang vị trí mới.

Chúng ta nên biết các kiểu dữ liệu số trong C++. Tuy nhiên, trong giáo trình này, chúng tôi chỉ sử dụng các kiểu dữ liệu `int`, `double` và `long`. Đối với các ứng dụng đơn giản, chúng ta không cần sử dụng các kiểu dữ liệu khác ngoài `int` và `double`. Nhưng khi bạn viết một ứng dụng cần sử dụng đến các số lớn thì có thể dùng sang kiểu `long`.

2.3.3 Kiểu C++11

Miền giá trị của kiểu số nguyên thay đổi trên các máy tính có hệ điều hành khác nhau. Ví dụ, trên máy có hệ điều hành 32-bit một số nguyên cần 4 bytes để lưu trữ, nhưng trên máy có hệ điều hành 64-bit một kiểu số nguyên cần 8 bytes. Điều này dẫn đến nhiều vấn đề nếu như bạn không hiểu chính xác miền giá trị được lưu trữ cho kiểu số nguyên. Để giải quyết vấn đề này, các kiểu số nguyên mới được thêm vào C++11 để chỉ rõ chính xác giá trị cho cả số có dấu và số không dấu. Để sử dụng các kiểu dữ liệu này cần thêm `<cstdint>` trên khai báo. Bảng 2.3 biểu diễn một số kiểu dữ liệu mới này.

C++11 cũng thêm vào một kiểu có tên là `auto`, khi đó chương trình tự suy ra kiểu dữ liệu tương ứng dựa vào biểu thức toán học bên phải phép gán. Ví dụ, dòng lệnh sau định nghĩa một biến `x` có kiểu dữ liệu tùy thuộc vào việc tính giá trị biểu thức từ “`expression`”:

```
auto x = expression;
```

Bảng 2.2: Một số kiểu dữ liệu số.

Kiểu	Kích thước	Miền giá trị	Độ chính xác
short (short int)	2 bytes	-32.768 đến 32.768	
int	4 bytes	-2.147.483.648 đến 2.147.483.647	
long (long int)	4 bytes	-2.147.483.648 đến 2.147.483.647	
float	4 bytes	xấp xỉ từ 10^{-38} đến 10^{38}	7 chữ số
double	8 bytes	xấp xỉ từ 10^{-308} đến 10^{308}	15 chữ số
long double	10 bytes	xấp xỉ từ 10^{-4932} đến 10^{4932}	19 chữ số

Trong bảng chỉ đưa ra một vài thông tin về sự khác nhau giữa các kiểu dữ liệu số. Các giá trị có thể khác nhau trên các hệ thống khác nhau. Độ chính xác để chỉ số các số phần thập phân. Miền giá trị cho các kiểu `float`, `double` và `long double` là miền giá trị cho số dương. Đối với số âm thì miền giá trị tương tự nhưng chứa dấu âm phía trước mỗi số.

Kiểu dữ liệu này không được sử dụng nhiều cho tới thời điểm này nhưng giúp cho chúng ta tiết kiệm những đoạn code sử dụng kiểu dữ liệu lớn hơn do tự chúng ta định nghĩa.

Bảng 2.3: Một số kiểu số nguyên trong C++11.

Kiểu	Kích thước	Miền giá trị
<code>int8_t</code>	1 bytes	-2^7 đến 2^7-1
<code>uint8_t</code>	1 bytes	0 đến 2^8-1
<code>int16_t</code>	2 bytes	-2^{15} đến $2^{15}-1$
<code>uint16_t</code>	2 bytes	0 đến $2^{16}-1$
<code>int32_t</code>	4 bytes	-2^{31} đến $2^{31}-1$
<code>uint32_t</code>	4 bytes	0 đến $2^{32}-1$
<code>int64_t</code>	8 bytes	-2^{63} đến $2^{63}-1$
<code>uint64_t</code>	8 bytes	0 đến $2^{64}-1$
<code>long long</code>	Ít nhất là 8 bytes	

C++11 đưa ra một cách thức mới để xác định kiểu của một biến hoặc một biểu thức. `decltype(expr)` là một dạng khai báo của biến hoặc một biểu thức:

```
int x = 10;
decltype(x*3.5) y;
```

Đoạn mã nguồn trên khai báo biến `y` có cùng kiểu dữ liệu với `x*3.5`. Biểu thức `x*3.5` là một số thực do đó `y` cũng được khai báo là một số thực.

2.3.4 Kiểu char

Trong máy tính cũng như trong C++, không chỉ sử dụng tính toán với dữ liệu số, do đó chúng tôi xin giới thiệu một số kiểu dữ liệu phi số khác thậm chí còn phức tạp hơn. Các giá trị của kiểu `char` viết tắt của từ `character` là các ký tự đơn giống như các chữ cái, chữ số và các ký tự chấm câu. Giá trị của kiểu dữ liệu này gọi là các ký tự và trong C++ gọi là `char`. Ví dụ, các biến `symbol` và `letter` có kiểu `char` được khai báo như sau:

```
char symbol, letter;
```

Các biến có kiểu `char` chứa bất kỳ một ký tự từ bàn phím. Ví dụ, biến `symbol` có thể lưu ký tự `'A'` hoặc một ký tự `'+'`. Chú ý các ký tự hoa và ký tự thường là hoàn toàn khác nhau.

Đoạn văn bản nằm trong dấu hai nháy ở câu lệnh `cout` được gọi là chuỗi ký tự. Ví dụ sau thực hiện trong chương trình hình 2.1 là một chuỗi ký tự:

```
"Enter the number of candy bars in a package\n"
```

Chú ý rằng, giá trị chuỗi ký tự đặt trong dấu nháy kép, một ký tự thuộc kiểu `char` thì đặt trong dấu nháy đơn. Hai dấu nháy này có ý nghĩa hoàn toàn khác nhau. Ví dụ, `'A'` và `"A"` là hai giá trị khác nhau. `'A'` là giá trị của biến kiểu `char`. `"A"` là chuỗi các ký tự. Mặc dù chuỗi ký tự chỉ chứa một ký tự nhưng cũng không thể biến chuỗi `"A"` có giá trị kiểu `char`. Chú ý, với cả chuỗi ký tự và ký tự, dấu nháy ở bên phải và bên trái đều như nhau.

Sử dụng kiểu `char` được mô tả trong chương trình hình 2.4. Chú ý rằng, khi người dùng gõ một khoảng trống giữa hai giá trị nhập, chương trình sẽ bỏ qua khoảng trống và nhập giá trị `'B'` cho biến thứ hai. Khi bạn sử dụng câu lệnh `cin` để đọc giá trị vào cho biến kiểu `char`, máy tính sẽ bỏ qua tất cả các khoảng trống và dấu xuống dòng cho đến khi gặp một ký tự khác khoảng trống và đọc ký tự đó vào biến. Do đó, không có sự khác biệt giữa các giá trị chứa khoảng trống hay không chứa khoảng trống. Chương trình trên hình 2.4 sẽ hiển thị giá trị ra màn hình với hai trường hợp người dùng gõ khoảng trống giữa các ký tự nhập vào và trường hợp không chứa khoảng trống dưới đây.

```
1 //Chương trình minh họa kiểu dữ liệu char
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     char symbol1, symbol2, symbol3;
7
8     cout << "Enter two initials, without any periods:\n";
9     cin >> symbol1 >> symbol2;
10    cout << "The two initials are:\n";
11    cout << symbol1 << symbol2 << endl;
12    cout << "Once more with a space:\n";
13    symbol3 = ' ';
14    cout << symbol1 << symbol3 << symbol2 << endl;
15    cout << "That's all.";
16    return 0;
17 }
```

Hình 2.4: Chương trình minh họa kiểu dữ liệu `char`.

Kiểu `bool` Kiểu dữ liệu chúng ta đề cập tiếp theo là kiểu `bool`. Kiểu dữ liệu này do ISO/ANSI (International Standards Organization/American National Standards Organization) đưa vào ngôn ngữ C++ năm 1998. Các biểu thức của kiểu `bool` được gọi là Boolean khi nhà toán học người Anh Geogre Boole (1815-1864) đưa ra các luật cho toán học logic. Các biểu thức boolean có hai giá trị đúng (`true`) hoặc sai (`false`). Các biểu thức boolean được sử dụng trong các câu lệnh rẽ nhánh và câu lệnh lặp, chúng ta sẽ đề cập trong phần 2.4.

2.3.5 Tương thích kiểu dữ liệu

Theo quy tắc thông thường, bạn không thể lưu giá trị thuộc kiểu dữ liệu này cho một biến thuộc kiểu dữ liệu khác. Ví dụ, phần lớn trình biên dịch không cho phép như sau:

```
int int_variable;  
int_variable = 2.99;
```

Vấn đề ở đây là không tương thích kiểu dữ liệu. Giá trị hằng `2.99` là kiểu `double` và biến `int_variable` là kiểu `int`. Tuy nhiên, không phải trình biên dịch nào cũng xử lý vấn đề này như nhau. Một số trình biên dịch sẽ trả ra thông báo lỗi, một số sẽ đưa ra cảnh báo, một số sẽ không chấp nhận một số kiểu. Nhưng ngay cả với những trình biên dịch cho phép bạn sử dụng phép gán như trên, thì biến `int_variable` chỉ nhận giá trị `2`, chứ không phải là `3`. Khi bạn không biết trình biên dịch có chấp nhận phép gán như trên hay không, tốt nhất bạn không nên gán giá trị số thực cho biến kiểu nguyên.

Vấn đề tương tự khi bạn gán giá trị của biến kiểu `double` thay cho giá trị hằng `2.99`. Phần lớn các trình biên dịch sẽ không chấp nhận phép gán như sau:

```
int int_variable;  
double double_variable;  
double_variable = 2.00;  
int_variable = double_variable;
```

Thực tế giá trị `2.00` không có gì khác biệt. Giá trị `2.00` là kiểu `double`, không phải kiểu `int`. Như chúng ta thấy, chúng ta có thể thay thế giá trị `2.00` bằng `2` trong phép gán giá trị cho biến `double_variable`, nhưng vẫn không đủ để cho phép gán ở dòng thứ 4 được chấp nhận. Các biến `int_variable` và biến `double_variable` thuộc kiểu dữ liệu khác nhau, đây mới là nguyên nhân của vấn đề.

Mặc dù trình biên dịch cho phép sử dụng nhiều kiểu dữ liệu trong phép gán, nhưng phần lớn trường hợp đó không nên sử dụng. Ví dụ, nếu trình biên dịch cho phép gán giá trị `2.99` cho biến kiểu nguyên, và biến này sẽ nhận giá trị `2` thay vì `2.99`. Vì thế, chúng ta rất dễ bị hiểu nhầm rằng chương trình đang nhận giá trị `2.99`.

Trong một số trường hợp, giá trị biến này được gán cho giá trị biến kia. Biến kiểu `int` có thể gán giá trị cho biến kiểu `double`. Ví dụ, câu lệnh sau đều hợp lệ:

```
double double_variable;  
double_variable = 2;
```

Đoạn lệnh trên thực hiện gán cho biến `double_variable` giá trị bằng `2.0`.

Mặc dù đây không phải là cách hay nhưng bạn có thể lưu giá trị số nguyên `65` vào biến kiểu `char` và lưu giá trị của ký tự `'Z'` vào biến kiểu `int`. C++ coi các ký tự là các số nguyên bé (`short`), vì C++ được kế thừa từ C. Lý do là vì các biến kiểu `char` tốn ít bộ nhớ hơn các biến kiểu `int` và

tính toán trên các biến kiểu `char` cũng sẽ tiết kiệm bộ nhớ. Tuy nhiên, chúng ta nên sử dụng kiểu `int` khi làm việc với các số nguyên và sử dụng kiểu `char` khi làm việc với các ký tự.

Quy tắc là bạn không thể thay thế giá trị của một kiểu bằng giá trị của biến có kiểu khác, nhưng vẫn có nhiều trường hợp ngoại lệ hơn là các trường hợp thực hiện theo quy tắc. Thậm chí trường hợp trình biên dịch không quy định chặt chẽ, thì cứ theo quy tắc là tốt nhất. Thay thế giá trị của một biến bằng một giá trị biến có kiểu dữ liệu khác có thể gây ra các vấn đề khi giá trị bị thay đổi để đúng với kiểu của biến, làm cho giá trị cuối cùng của biến không đúng như mong muốn.

2.3.6 Toán tử số học và biểu thức

Trong chương trình C++, bạn có thể kết hợp các biến và/hoặc các số sử dụng toán tử `+` cho phép cộng, `-` cho phép trừ, `*` cho phép nhân và `/` cho phép chia. Ví dụ, phép gán trong chương trình hình 2.1 sử dụng toán tử `*` để nhân các số nằm trong hai biến (kết quả được gán lại cho biến nằm bên trái dấu bằng)

```
total_weight = one_weight * number_of_bars;
```

Tất cả các toán tử số học có thể được sử dụng cho các số kiểu `int`, kiểu `double` và các kiểu số khác. Tuy nhiên, giá trị của mỗi kiểu dữ liệu được tính toán và giá trị chính xác phụ thuộc vào kiểu của các số hạng. Nếu tất cả toán hạng thuộc kiểu `int`, thì kết quả cuối cùng là kiểu `int`. Nếu một hoặc tất cả toán hạng thuộc kiểu `double`, thì kết quả cuối cùng là `double`. Ví dụ, nếu các biến `base_amount` và `increase` có kiểu `int`, thì biểu thức sau cũng có kiểu `int`:

```
base_amount + increase
```

Tuy nhiên, nếu một hoặc cả hai biến đều là kiểu `double` thì kết quả trả về là kiểu `double`. Tương tự với các toán tử `-`, `*` hoặc `/`.

Kiểu dữ liệu của kết quả phép tính có thể chính xác hơn những gì bạn nghĩ ngờ. Ví dụ, $7.0/2$ có một toán hạng kiểu `double`, là 7.0 . Khi đó, kết quả sẽ là kiểu `double` với giá trị bằng 3.5 . Tuy nhiên, $7/2$ có hai toán hạng là kiểu `int` và do đó kết quả có kiểu `int` với giá trị bằng 3 . Nếu kết quả chắc chắn vẫn có sự khác nhau ở đây. Ví dụ, nếu $6.0/2$ có một toán hạng kiểu `double`, toán hạng đó là 6.0 . Khi đó, kết quả có kiểu `double` và có giá trị bằng 3.0 là một số xấp xỉ. Tuy nhiên, $6/2$ có hai toán hạng kiểu `int`, kết quả trả về bằng 3 thuộc kiểu `int` và là số chính xác. Toán tử chia là một toán tử bị ảnh hưởng bởi kiểu của các đối số.

Khi sử dụng một trong hai toán hạng kiểu `double`, phép chia `/` cho kết quả như bạn tính. Tuy nhiên, khi sử dụng với các toán hạng kiểu `int`, phép chia `/` trả về phần nguyên của phép chia. Hay nói cách khác, phép chia số nguyên bỏ qua phần thập phân. Do đó, $10/3$ cho kết quả là 3 (không phải 3.3333), $5/2$ được 2 (không phải 2.5) và $11/3$ được 3 (không phải 3.66666). Chú ý rằng các số không được làm tròn, phần thập phân bị bỏ qua bất kể với giá trị lớn hay nhỏ.

Toán tử `%` được sử dụng với các toán hạng kiểu `int` để lấy lại phần giá trị bị mất khi sử dụng phép chia `/` với số nguyên. Ví dụ, 17 chia 5 được 3 dư 2 . Toán tử `/` trả về thương. Toán tử `%` trả về phần dư. Ví dụ, câu lệnh sau:

```
cout << "17 divided by 5 is " << (17/5) << endl;
cout << "with a remainder of " << (17%5) << endl;
```

cho kết quả:

```
17 divided by 5 is 3
```

```
with a remainder of 2
```

Khi sử dụng với số âm thuộc kiểu `int`, kết quả của phép chia `/` và phép lấy dư `%` có thể sẽ khác nhau trên các trình biên dịch C++ khác nhau. Do đó, bạn nên sử dụng `/` và `%` với giá trị nguyên chỉ khi bạn biết cả hai giá trị đều không âm.

Các biểu thức toán học có thể có các khoảng trống. Bạn có thể thêm các khoảng trống trước và sau các toán tử và các dấu ngoặc đơn hoặc có thể bỏ qua. Viết theo cách nào mà ta có thể dễ dàng đọc được nhất. Chúng ta có thể đưa ra thứ tự thực hiện phép toán bằng cách sử dụng các dấu ngoặc đơn như mô tả dưới đây:

```
(x + y) * z
x + (y * z)
```

Mặc dù bạn có thể sử dụng các công thức toán học có cả dấu ngoặc vuông và một số dấu ngoặc khác, nhưng những dấu ngoặc đó không được sử dụng trong C++. C++ chỉ cho phép dấu ngoặc đơn trong các biểu thức toán học.

Nếu bỏ qua dấu ngoặc đơn, máy tính sẽ thực hiện tính toán theo thứ tự ưu tiên như thứ tự của phép toán `+` và `*`. Thứ tự ưu tiên này tương tự với đại số và toán học. Ví dụ, phép nhân được thực hiện trước sau đó thực hiện phép cộng. Ngoại trừ một số trường hợp, như cộng các xâu kí tự hoặc phép nhân bên trong phép cộng, với cách này thì nên dùng thêm dấu ngoặc đơn. Dấu ngoặc đơn thêm vào để các biểu thức toán học dễ hiểu và để tránh lỗi lập trình. Bảng các thứ tự ưu tiên được mô tả trong phụ lục B.

Khi bạn sử dụng phép chia `/` cho hai số nguyên, kết quả cũng là một số nguyên. Sẽ có vấn đề nếu như bạn mong muốn kết quả là một số thực. Hơn nữa, vấn đề này lại khó phát hiện, kết quả của chương trình trông có vẻ là chấp nhận được nhưng khi tính toán cho ra kết quả không đúng. Ví dụ, giả sử bạn là một kiến trúc sư cầu đường được trả 5000\$ trên một dặm đường quốc lộ, và giả sử bạn biết chiều dài của con đường đo bằng feet. Giá bạn đổi được tính như sau:

```
total_price = 5000 * (feet/5280.0);
```

Phép toán này thực hiện được bởi vì 5280 feet trong một dặm. Nếu như chiều dài của đường quốc lộ bạn đang thi công là 15000 feet, công thức sẽ trả về cho bạn tổng giá trị là

```
5000 * (15000/5280.0)
```

Chương trình C++ của bạn nhận được giá trị cuối cùng như sau: 15000/5280.0 bằng 2.84. Sau đó chương trình nhân với 5000 với 2.84 được giá trị bằng 14200.00. Sử dụng chương trình C++ đó, bạn biết được phải trả 14.200\$ cho dự án.

Bây giờ giả sử biết `feet` là kiểu số nguyên, và bạn quên không viết thêm dấu chấm và số 0 vào sau 5280, câu lệnh gán được viết như sau:

```
total_price = 5000 * (feet/5280);
```

Câu lệnh dường như không có gì sai nhưng sẽ có vài vấn đề khi thực thi. Nếu bạn sử dụng phép gán thứ hai, bạn chia hai giá trị kiểu `int`, do đó kết quả phép chia `feet/5280` tương đương với 15000/2580 và được giá trị bằng 2 (thay vì giá trị 2.84 như lúc trước). Do đó giá trị được gán cho biến `total_cost` là 5000*2, bằng 10000.00. Nếu bạn quên dấu thập phân bạn sẽ trả 10.000\$.

Tuy nhiên, như chúng ta đã thấy, giá trị đúng phải là 14.200\$. Thiếu phần thập phân đã làm bạn mất đi 4200\$. Chú ý rằng bạn vẫn mất số tiền đó cho dù biến `total_price` có kiểu `int` hoặc `double`; giá trị đó đã bị biến đổi trước khi gán cho biến `total_price`.

2.4 Luồng điều khiển

Các chương trình phần mềm là một tập hợp thống nhất các câu lệnh đơn giản được hệ điều hành thực thi theo một thứ tự nào đó. Tuy nhiên, để viết các phần mềm phức tạp, bạn cần thực hiện nhiều câu lệnh hơn với thứ tự phức tạp hơn. Để làm được điều đó, bạn cần sử dụng các cấu trúc rẽ nhánh và các cấu trúc điều khiển. Trong phần này, chúng ta sẽ xem xét chúng, cấu trúc điều khiển đơn giản đó là cấu trúc **if-else** và **while (do-while)**.

Trên thực tế, việc bạn phải đưa ra lựa chọn là việc không thể tránh khỏi, điều đó cũng không phải là ngoại lệ trong lập trình. Bạn phải đưa ra quyết định lựa chọn câu lệnh nào sẽ được hệ điều hành thực thi, vậy bạn làm thế nào để thực hiện điều đó? C++ cung cấp cho chúng ta nhiều cách để làm điều đó, một trong số chúng là sử dụng cấu trúc rẽ nhánh **if-else**. Sử dụng cấu trúc này cho phép bạn lựa chọn thực hiện một hoặc một nhóm các câu lệnh dựa trên một điều kiện có sẵn nào đó.

Ví dụ, giả sử bạn là một ông chủ và bạn muốn viết một chương trình để tính lương tuần theo mỗi giờ cho nhân viên của mình. Công ty trả gấp rưỡi tiền lương cho mỗi giờ làm thêm, số giờ làm thêm được tính là số giờ làm việc sau số giờ làm việc bắt buộc (40 giờ làm việc bắt buộc một tuần). Thông thường, bạn sẽ dễ dàng tính được số tiền bạn phải chi trả cho một nhân viên của bạn như sau:

```
Gross_pay = rate * 40 + 1.5*rate*(hours-40)
```

Tuy nhiên, bạn sẽ nhận ra vấn đề khi nhân viên của bạn làm việc ít hơn 40 giờ mỗi tuần, nếu sử dụng công thức trên thì sẽ có vấn đề xảy ra. Trong trường hợp này, bạn phải sử dụng một công thức khác để tính, đó là:

```
Gross_pay = rate * hours
```

Công ty của bạn hiển nhiên có rất nhiều nhân viên, do đó cả hai trường hợp trên xảy ra là không thể tránh khỏi, vì vậy bạn cần sử dụng cả hai công thức trên, tuy nhiên vấn đề nằm ở chỗ, làm thế nào bạn biết khi nào bạn cần sử dụng công thức đầu tiên, khi nào bạn cần sử dụng công thức thứ hai trong chương trình của bạn? C++ cung cấp cho bạn cấu trúc **if-else** để làm điều này, việc đơn giản mà bạn cần làm là đặt chúng vào đúng vị trí của nó, việc đưa ra quyết định sử dụng cấu trúc **if-else** được đưa ra như sau:

```
if(hours > 40)
{
    Gross_pay=rate * 40 + 1.5*rate*(hours-40)
}
else{
    Gross_pay=rate * hours
}
```

Như vậy, bạn sẽ có thể tính toán chính xác số tiền phải trả cho nhân viên của mình cho dù anh ta làm việc ít hơn hay nhiều hơn 40 giờ mỗi tuần. Cấu trúc **if-else** là một cấu trúc rẽ nhánh đơn giản. Tuy nhiên, nó mang lại hiệu quả rất tốt đối với việc đưa ra quyết định dựa trên điều kiện nào đó, cú pháp như sau:

```
if (Boolean_Expression)
{
    True_Expression;
}
```

```

else
{
    False_Expression;
}

```

Boolean_Expression là một biểu thức logic hoặc tập hợp các biểu thức logic, nếu Boolean_Expression có giá trị true thì câu lệnh True_Expression sẽ được thực hiện, biểu thức này có thể là một câu lệnh đơn hoặc một tập hợp các câu lệnh khác. Nếu True_Expression là câu lệnh đơn, bạn có thể bỏ cặp dấu ngoặc mà C++ không báo lỗi. Nếu Boolean_Expression trả về giá trị false thì câu lệnh False_Expression sẽ được thực hiện.

Chương trình hoàn chỉnh của ví dụ trên như dưới đây.

```

1 //Chương trình minh họa câu lệnh if-else
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     int hours;
7     double gross_pay, rate;
8     cout << "Enter the hourly rate of pay: $";
9     cin >> rate;
10    cout << "Enter the number of hours worked,\n"
11    << "rounded to a whole number of hours: ";
12    cin >> hours;
13    if (hours > 40)
14        gross_pay = rate * 40 + 1.5 * rate * (hours - 40);
15    else
16        gross_pay = rate * hours;
17    cout.setf(ios::fixed);
18    cout.setf(ios::showpoint);
19    cout.precision(2);
20    cout << "Hours = " << hours << endl;
21    cout << "Hourly pay rate = $" << rate << endl;
22    cout << "Gross pay = $" << gross_pay << endl;
23    return 0;
24 }

```

Hình 2.5: Chương trình minh họa cấu trúc if-else.

Vòng lặp

Hầu hết các chương trình đều chứa một hoặc một số câu lệnh được thực hiện lặp lại nhiều lần. Ví dụ, chúng ta đã giả thiết rằng bạn là ông chủ của một công ty lớn, và bạn cần phải tính lương phải chi trả cho nhân viên của mình mỗi tuần, giả sử bạn có 1000 nhân viên, bạn phải thực hiện các phép tính ấy 1000 lần. Thông thường, trong chương trình của mình, bạn phải viết chúng lặp đi lặp lại 1000 lần. Tuy nhiên, C++ cung cấp cho chúng ta một cấu trúc cho phép thực hiện việc đó chỉ trong vài câu lệnh đơn giản, chúng được gọi là vòng lặp. Trong phần này, chúng ta chỉ xem xét đến vòng lặp while.

Cấu trúc lặp while cho phép bạn thực hiện câu lệnh trong phần “body” của nó cho tới khi nào biểu thức Boolean_Expression còn trả về giá trị true. Cú pháp như sau:

```

while (Boolean_Expression){
    Body statement;
}

```

```
}

```

Chương trình sau sẽ đưa ra màn hình chữ “Hello” với số lần được bạn nhập từ bàn phím sử dụng cấu trúc lặp `while`.

```
1 //Chương trình minh họa lặp while
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     int count_down;
7     cout << "How many greetings do you want? ";
8     cin >> count_down;
9
10    while (count_down > 0)
11    {
12        cout << "Hello ";
13        count_down = count_down - 1;
14    }
15    cout << endl;
16    cout << "That's all!\n";
17    return 0;
18 }
```

Hình 2.6: Chương trình minh họa vòng lặp `while`.

Cấu trúc `while` hoạt động dựa vào giá trị của `Boolean_Expression`, khi được thực thi, hệ điều hành sẽ kiểm tra điều kiện `Boolean_Expression` trước, nếu `Boolean_Expression` có giá trị `true` thì hệ điều hành sẽ thực hiện các câu lệnh trong “body”. Việc này có ưu điểm là `Boolean_Expression` sẽ được kiểm tra trước khi thực hiện bất cứ câu lệnh nào trong “body”. Tuy nhiên, trong một số trường hợp như bạn muốn hiển thị một menu chẳng hạn, bạn cần thực hiện các câu lệnh trong “body” ít nhất một lần dù `Boolean_Expression` là `True` hay `False`. Vậy bạn phải làm thế nào? Rất may, C++ cũng cung cấp một sự lựa chọn thay thế cho `while`, đó là cấu trúc `do-while`. Với cấu trúc này, các câu lệnh trong “body” sẽ được thực hiện ít nhất 1 lần. Cú pháp `do-while` như sau:

```
do {
    Body statement;
} while(Boolean_Expression);
```

Khi được thực thi, hệ điều hành sẽ thực thi các câu lệnh trong “body” trước khi kiểm tra `Boolean_Expression`. Ví dụ như dưới đây.

```
1 //Chương trình minh họa lặp do-while
2 #include <iostream>
3 using namespace std;
4 int main( )
5 {
6     char ans;
7     do
8     {
9         cout << "Hello\n";
10        cout << "Do you want another greeting?\n"
11        << "Press y for yes, n for no,\n"
12        << "and then press return: ";
```

```

13     cin >> ans;
14     } while (ans == 'y' || ans == 'Y');
15     cout << "Good-Bye\n";
16     return 0;
17 }

```

Hình 2.7: Chương trình minh họa vòng lặp do-while.

2.5 Phong cách lập trình

Tất cả các biến trong giáo trình này đều đã được lựa chọn để làm tiêu chuẩn cho các chương trình khác. Các chương trình, đoạn mã đều được đặt trong một định dạng cụ thể và thống nhất. Ví dụ như các khai báo, các câu lệnh đều được đặt thụt vào một khoảng nào đó bằng nhau. Một chương trình được viết cẩn thận không chỉ về logic mà còn thống nhất về hình thức thể hiện các đoạn mã lệnh sẽ dễ dàng cho người khác đọc, hiểu và sửa lỗi nếu có. Việc thay đổi chương trình cũng trở nên dễ dàng hơn.

Một chương trình sẽ dễ dàng đọc hơn, tất nhiên là đối với con người, nếu các đoạn mã lệnh được đặt một cách khoa học, các đoạn mã nên được đặt thụt vào một khoảng nào đó so với lề hoặc bố trí chúng theo từng nhóm.

Cặp dấu được sử dụng để phân biệt các đoạn mã dài trong một chương trình lớn. Bạn nên đặt chúng ở một dòng, dấu mở ngoặc `(` và đóng ngoặc `)` nên thụt dòng và cách lề một khoảng nhất định nào đó, việc này giúp bạn dễ dàng tìm được các cặp ngoặc tương ứng.

Để người khác dễ dàng hiểu được chương trình, có thể sử dụng chú giải để giải thích ngắn gọn đoạn mã đang viết. C++ cũng như hầu hết các ngôn ngữ lập trình khác cung cấp một cú pháp để có thể viết chú giải trong chương trình. Trong C++, sử dụng kí tự `//` để bắt đầu một chú thích với nội dung chỉ trên một dòng. Nếu chú thích là nhiều dòng bạn có thể sử dụng nhiều kí tự `//` hoặc có thể sử dụng một cặp kí tự `/*` để bắt đầu và `*/` để kết thúc. Ví dụ như sau:

```

/* Đây là chương trình tính Uscln của 2 số.
Sử dụng thuật toán Euclid */

```

Cần chú ý rằng, tất cả các chú giải sẽ không được trình biên dịch xử lý.

Trong lập trình, đặc biệt là các chương trình lớn, bạn sẽ phải làm việc với một số lượng rất lớn các biến. Trong số chúng, một số có giá trị là không thay đổi trong toàn bộ quá trình chương trình thực thi, chúng được gọi là hằng số. C++ cung cấp một từ khóa `const` cho phép bạn khai báo một biến với vai trò là một hằng số. Ví dụ,

```
const int WINDOW_COUNT=10;
```

Các hằng số thường được viết hoa toàn bộ các kí tự, nếu chúng gồm nhiều từ sẽ được nối với nhau bằng dấu gạch dưới (`_`). Thông thường, chúng được đặt ở đầu chương trình để thuận tiện cho việc kiểm soát cũng như thay đổi.

Sau khi khai báo hằng số bằng từ khóa `const`, bạn có thể sử dụng chúng ở bất kì đâu mà không cần khai báo lại chúng. Điều quan trọng bạn cần ghi nhớ đó là mỗi hằng số phải được gán một giá trị ngay khi chúng được khai báo, giá trị này là không thể thay đổi.

2.6 Biên dịch chương trình với GNU/C++

Trong các môi trường tích hợp được các nhà sản xuất chương trình dịch cung cấp, bạn có thể vừa soạn thảo chương trình, vừa dịch, liên kết và chạy chương trình kết hợp cùng một lúc (chỉ cần bạn nhấn một phím tắt nào đó - ví dụ F9 trong một số phiên bản của Dev-Cpp). Tuy nhiên, cũng có lúc bạn không có sẵn môi trường tích hợp hoặc cần dịch những chương trình, dự án lớn ra file thực thi (*.exe), khi đó bạn cần có một bộ chương trình dịch. Ở đây, chúng tôi trình bày cách dịch chương trình với bộ dịch của GNU/C++.

GNU/C++

Bộ trình dịch GNU (GCC: GNU Compiler Collection) là một tập hợp các trình dịch được thiết kế cho nhiều ngôn ngữ lập trình khác nhau, được nhiều hệ điều hành chấp nhận. Tên gốc của GCC là GNU C Compiler (Trình dịch C của GNU), do ban đầu nó chỉ hỗ trợ dịch ngôn ngữ lập trình C. GCC 1.0 được phát hành vào năm 1987, sau đó được mở rộng hỗ trợ dịch C++ vào tháng 12 cùng năm đó và tiếp tục mở rộng hỗ trợ dịch các ngôn ngữ khác như Fortran, Pascal, Objective C, Java, và Ada ... Bạn có thể tải về miễn phí trên mạng. Trong một số môi trường tích hợp như Dev-Cpp trình dịch GNU/C++ cũng được cài đặt sẵn (thư mục bin).

Câu lệnh dịch

Từ cửa sổ lệnh của hệ điều hành và trong thư mục chứa bộ dịch GNU (đã được cài đặt) bạn gõ câu lệnh sau:

```
g++ options source_file
```

trong đó, **source_file** là file bạn cần biên dịch (viết đầy đủ cả đường dẫn thư mục, tên lẫn phần mở rộng) và options là các lựa chọn chế độ dịch, có thể đặt ở trước hoặc sau **source_file**. Tên file kết quả được ngầm định là **a.exe** (trong môi trường DOS/Windows) hoặc **a.out** (trong môi trường Unix/Linux). Để đặt tên cụ thể cho file kết quả bạn sử dụng options:

```
-o target_file
```

Ví dụ: `g++ main.cpp -o main.exe` sẽ dịch file `main.cpp` ra mã máy, chạy được và đặt vào file có tên `main.exe`.

Dịch chương trình nhiều file

Để dịch chương trình nhiều file, bạn có thể liệt kê danh sách các file cần dịch và liên kết vào trong câu lệnh, GCC sẽ tạo file thực thi cuối cùng theo ý muốn. Ví dụ, ta có chương trình với 4 hàm lần lượt tính cộng (sum), trừ (sub), nhân (mul), chia (div) hai số nguyên và hàm main dùng để tính biểu thức $(a + b) * (a - b)$ với $a = 5$ và $b = 3$. Giả sử mỗi hàm được đặt trên 1 file có tên như tên hàm và đuôi cpp như đoạn mã bên dưới.

```
1 /* file "sum.cpp" */
2 int Sum(int value1, int value2)
3 {
4     return value1 + value2;
5 }
6
7 /* file "sub.cpp" */
8 int Sub(int value1, int value2)
9 {
10    return value1 - value2;
11 }
```



```

12
13 /* file "mul.cpp" */
14 int Mul(int value1, int value2)
15 {
16     return value1 * value2;
17 }
18
19 /* file "div.cpp" */
20 int Div(int value1, int value2)
21 {
22     return value1 / value2;
23 }

```

Hình 2.8: Chương trình với nhiều file.

```

1 /* file "main.cpp" */
2 #include <iostream>
3 using namespace std;
4
5 int Sum(int, int);
6 int Sub(int, int);
7 int Div(int, int);
8 int Mul(int, int);
9
10 int main() {
11     int a = 5, b = 3;
12     cout << Mul(Sum(a, b), Sub(a, b));
13     return 0;
14 }

```

Hình 2.9: Hàm main.

Khi đó, bạn có thể dùng câu lệnh: `g++ main.cpp sum.cpp sub.cpp div.cpp mul.cpp -o main.exe` để dịch chương trình này ra file `main.exe`.

Với câu lệnh trên bộ dịch đã thực hiện 2 bước: dịch từng file sang mã object (cùng tên file với đuôi `.o`) và sau đó liên kết các file này thành file mã thực thi (đuôi `.exe`)

Bạn có thể tách rời hai bước này bằng cách dịch sang mã object từng file một (với lựa chọn options là `-c`) và sau đó thực hiện liên kết chúng. Ví dụ:

Bước 1: Dịch sang object

```

g++ -c main.cpp      // cho ra file main.o
g++ -c sum.cpp       // cho ra file sum.o
g++ -c sub.cpp       // cho ra file sub.o
g++ -c mul.cpp       // cho ra file mul.o
g++ -c div.cpp       // cho ra file div.o

```

Bước 2: Liên kết các object

```

g++ main.o sum.o sub.o div.o mul.o -o main.exe

```

Tiện ích Make

Với chương trình nhiều file, mỗi lần cần dịch phải viết câu lệnh rất dài. Để thuận lợi, các thông tin trong câu lệnh này được đưa sẵn vào file có tên đặc biệt là `makefile`. Khi đó, mỗi lần dịch bằng câu lệnh **make**, GCC sẽ tự động tìm chạy file này.

Makefile gồm các nhóm thông tin, nhóm đầu tiên đặc tả file kết quả cuối cùng, các nhóm còn lại đặc tả các file thành viên. Mỗi đặc tả gồm 2 dòng. Dòng đầu gồm: tên file đích (*.o), dấu hai chấm và các file liên quan cần để dịch file này. Dòng thứ hai là câu lệnh dịch bắt đầu bằng dấu TAB. Riêng nhóm đầu tiên bắt đầu bằng từ all : và danh sách các file *.o cần liên kết để ra file cuối cùng. Ví dụ, makefile để dịch ví dụ trên như sau:

```
1 all : main.o sum.o sub.o mul.o div.o
2     g++ main.o sum.o sub.o mul.o div.o -o main.exe
3 main.o : main.cpp
4     g++ -c main.cpp
5 sum.o : sum.cpp
6     g++ -c sum.cpp
7 sub.o : sub.cpp
8     g++ -c sub.cpp
9 mul.o : mul.cpp
10    g++ -c mul.cpp
11 div.o : div.cpp
12    g++ -c div.cpp
13 clean :
14    rm *.o ; rm main.exe
```

Hình 2.10: Makefile.

Ngoài ra, trong Makefile còn có thể cài thêm một số câu lệnh khác, ví dụ câu lệnh clean (trong ví dụ trên) dùng để xóa các file object và file main.exe. Để dùng chức năng này, ta gõ lệnh : make clean (Cẩn thận khi dùng các lệnh xóa, có thể xóa nhầm các file object khác có sẵn trong thư mục).

Bài tập

- Trong mỗi bài toán trong các phần dưới đây, hãy cho biết sử dụng vòng lặp nào là hợp lý nhất (for, while, do-while):
 - Tính tổng dãy số sau: $1/2 + 1/3 + \dots + 1/2016$
 - Đọc danh sách điểm thi các môn học của một sinh viên.
 - Kiểm tra điều kiện đầu vào của biến nguyên dương a, b, c thỏa mãn điều kiện không vượt quá 1000.
 - Kiểm tra hàm xem nó thực hiện các giá trị khác nhau truyền vào như là tham số của hàm đó.
- Tìm ước số chung lớn nhất của 4 số nguyên được nhập từ bàn phím.
- Cho biết kết quả in ra màn hình của đoạn chương trình dưới đây:

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

- Viết chương tính e^x xấp xỉ theo công thức sau:

$$e^x = 1 + x + x^2/2! + x^3/3! + \dots + x^n/n!$$
- Viết chương trình in ra các số nguyên tố nhỏ hơn n (n là số nguyên dương, được nhập từ bàn phím).
- Giá trị Pi được tính xấp xỉ theo công thức sau:

$$Pi = 4[1 - 1/3 + 1/5 - 1/7 + 1/9 + \dots + ((-1)^n)/(2n + 1)]$$
 Hãy viết chương trình tính xấp xỉ giá trị Pi theo công thức trên. Chương trình nhập vào giá trị nguyên n và in giá trị Pi ra màn hình. Chương trình cũng cho phép người dùng lặp lại việc tính toán Pi với giá trị n mới cho đến khi người dùng dừng thì chương trình kết thúc.
- Viết chương trình đổi năm (giữa 1000 và 3000) từ số la mã sang số thập phân thông thường. Chương trình cho phép người dùng có thể lặp lại việc tính toán chuyển đổi năm cho đến khi muốn dừng thì chương trình kết thúc.
 Ví dụ: MCLM \rightarrow 1950, MCMLXXXIX \rightarrow 1989
- Quang chọn một số làm mật khẩu cho thẻ ngân hàng của mình. Hãy giúp Quang nhớ lại mật khẩu đó biết rằng mật khẩu là số có 4 chữ số thỏa mãn các điều kiện sau:
 - Tất cả các chữ số đều khác nhau.
 - Chữ số hàng nghìn bằng 3 lần chữ số hàng chục.
 - Là số lẻ.
 - Tổng các chữ số là 27.

Chương 3

Kiểm thử và gỡ rối chương trình

Chương này sẽ giới thiệu về các kỹ thuật kiểm thử cũng như gỡ rối chính khi viết chương trình và làm sao lập trình để có ít lỗi nhất.

3.1 Kỹ thuật kiểm thử

Kiểm thử là một thành phần chính của phát triển phần mềm để đảm bảo độ tin cậy và chất lượng của phần mềm. Phần mềm được kiểm thử theo nhiều cách khác nhau, một số trong các cách này được thực hiện bởi người lập trình (developer) và một số khác thường được thực hiện bởi người chuyên về kiểm thử (tester). Cụ thể hơn như sau:

- *Kiểm thử đơn vị (Unit testing)*: là sự thi hành của một lớp, chỉ thị hoàn chỉnh hoặc một chương trình nhỏ mà được viết bởi một lập trình viên hoặc một nhóm mà quá trình kiểm thử tách biệt;
- *Kiểm thử thành phần (Component testing)*: là sự thi hành của một lớp, modul hoặc một chương trình nhỏ được phát triển bởi nhiều người lập trình hoặc một nhóm lập trình mà quá trình kiểm thử tách biệt so với một hệ thống phần mềm hoàn chỉnh;
- *Kiểm thử tích hợp (Integration testing)*: là kết hợp các thành phần của một phần mềm và kiểm tra như một phần mềm đã hoàn thành. Trong khi kiểm thử đơn vị kiểm tra các thành phần và đơn vị riêng lẻ thì kiểm thử tích hợp kết hợp chúng lại với nhau và kiểm tra sự giao tiếp giữa chúng;
- *Kiểm thử hồi qui (Regression testing)*: Đơn thuần kiểm tra lại phần mềm sau khi có một sự thay đổi xảy ra, để bảo đảm phiên bản phần mềm mới thực hiện tốt các chức năng như phiên bản cũ và sự thay đổi không gây ra lỗi mới trên những chức năng vốn đã làm việc tốt;
- *Kiểm thử hệ thống (System testing)*:

Trong phần này, kiểm thử được đề cập là kiểm thử bởi lập trình viên, sẽ bao gồm kiểm thử đơn vị, kiểm thử thành phần và kiểm thử tích hợp.

Kiểm thử thường chia làm 2 loại chính là: **Kiểm thử hộp đen** và **kiểm thử hộp trắng**. Kiểm thử hộp đen là kỹ thuật mà người kiểm thử không biết công việc thực hiện bên trong của các hạng mục được kiểm thử. Rõ ràng, điều này không áp dụng khi bạn kiểm thử mã mà bạn đã viết.

Kiểm thử hộp trắng là kỹ thuật mà người kiểm thử biết rõ công việc thực hiện bên trong các hạng mục được kiểm thử. Đây là kỹ thuật kiểm thử mà bạn như là người phát triển phần mềm sử dụng để kiểm tra chính mã nguồn do chính bạn viết.

Một số lập trình viên sử dụng thuật ngữ “Kiểm thử” và “Gỡ rối” thay đổi cho nhau nhưng về cơ bản đây là 2 khái niệm khác nhau. Kiểm thử có nghĩa là phát hiện lỗi còn gỡ rối có nghĩa là chuẩn đoán, định vị và sửa nguyên nhân căn bản của lỗi mà đã được phát hiện.

Ở đây, chúng tôi trình bày một số kỹ thuật kiểm thử trong kiểm thử đơn vị.

3.1.1 Kiểm thử trong khi viết mã nguồn

Vấn đề lỗi càng sớm được tìm thấy thì tốt hơn trong việc viết mã nguồn. Nếu bạn nghĩ một cách hệ thống về cái gì chúng ta đang và sẽ viết nó, bạn có thể kiểm chứng các thuộc tính đơn giản của chương trình ngay khi nó được xây dựng. Với kết quả mà mã nguồn của bạn thông qua một vòng kiểm thử trước khi được biên dịch thì chắc chắn các loại lỗi sẽ được giảm thiểu một cách đáng kể.

Kiểm thử mã nguồn tại các cận (boundaries) của nó.

Một kỹ thuật là kiểm thử điều kiện cận: mỗi phần nhỏ của mã nguồn được viết như cấu trúc vòng lặp hoặc cấu trúc điều khiển. Quá trình này được gọi là kiểm thử điều kiện cận bởi vì chúng ta thường kiểm tra cận một cách tự nhiên trong chương trình và dữ liệu, ví dụ như đầu vào không tồn tại hoặc rỗng, cỡ của mảng, ... Ý tưởng là hầu hết các lỗi thường xuất hiện ở cận. Nếu một phần mã nguồn sẽ hỏng, nó có khả năng lỗi tại các cận. Ngược lại nếu nó thi hành tốt tại các cận thì nó cũng khả năng thi hành tốt tại các chỗ khác. Đoạn chương trình với hàm `fgets()` đọc các ký tự cho đến khi nó gặp ký tự dòng mới (newline) hoặc đầy bộ đệm:

```
int i;
char s[MAX];
for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i)
    s[--i] = '\0;
```

Hãy tưởng tượng rằng bạn vừa viết vòng lặp này. Cận đầu tiên được kiểm thử là khá đơn giản: dòng trống. Nếu bạn bắt đầu với dòng mà chỉ gồm 1 ký tự xuống dòng newline, nó sẽ dừng vòng lặp với `i = 0` và câu lệnh tiếp theo sẽ gán `i = -1` và vì vậy `s[-1]` sẽ được gán bằng ký tự `NULL`. Điều này sẽ gây ra lỗi bởi cận của mảng. Nếu chúng ta viết lại vòng lặp như sau:

```
for (i = 0; i < MAX-1; i++)
    if ((s[i] = getchar()) == '\n')
        break;
s[i] = '\0';
```

Lặp lại với kiểm thử ban đầu (dòng rỗng), dễ dàng xác nhận rằng ký tự xuống dòng (newline) được xử lý đúng. Nhưng cái gì xảy ra nếu đầu vào là rỗng vì khi đó thì lời gọi đầu tiên với hàm `getchar()` sẽ trả về `EOF`. Vì vậy, chúng ta phải kiểm tra điều đó:

```
for (i = 0; i < MAX-1; i++)
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)
        break;
s[i] = '\0';
```

Kiểm thử điều kiện cận rất hiệu quả khi tìm ra các lỗi như vậy.

Kiểm tra điều kiện trước và sau.

Một cách khác để kiểm thử chương trình là phải kiểm chứng đặc tính cần thiết hoặc mong đợi được nắm giữ trước và sau trong một số phần thi hành của mã nguồn. Đảm bảo chắc chắn giá trị đầu vào là trong phạm vi là ví dụ thường thấy trong kiểm thử điều kiện trước. Dưới đây, hàm `avg(...)` tính giá trị trung bình của `n` phần tử trong mảng có vấn đề nếu `n` nhỏ hơn hoặc bằng 0 :

```
double Avg(double a[], int n)
{
    double sum;
    sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum/n;
}
```

Hàm `avg` sẽ làm gì nếu `n` bằng 0. Mảng với không phần tử là khái niệm hợp lệ mặc dù giá trị trung bình là không xác định. Để kiểm thử điều này, chúng ta có thể cho giá trị trung bình là 0.0 nếu `n` nhỏ hơn hoặc bằng 0:

```
return (n <= 0) ? 0.0 : sum/n;
```

3.2 Kỹ thuật gỡ rối chương trình

3.2.1 Khái niệm về gỡ rối chương trình

Theo định nghĩa chung, gỡ rối là quá trình có phương pháp để tìm và giảm thiểu số lỗi và khiếm khuyết trong chương trình máy tính.

Tất cả nguồn gốc lỗi bắt nguồn từ tiền đề cơ bản: một cái gì đó nghĩ là đúng nhưng sự thực là sai. Bởi vì theo nguyên tắc đơn giản này, thực sự một số lỗi kỳ lạ có thể phi logic và do đó, việc gỡ rối phần mềm là công việc khó khăn và thách thức.

3.2.2 Phân loại lỗi

Lỗi được chia thành 3 loại lỗi chính:

- Lỗi cú pháp: Dễ dàng được phát hiện bởi trình biên dịch khi biên dịch chương trình. Nếu một câu lệnh mắc lỗi cú pháp, nó sẽ không được biên dịch và chương trình của bạn sẽ không được thực thi;
- Lỗi thực thi: Lỗi thực thi xảy ra khi máy tính được ra lệnh thực hiện một hành động lỗi. Khi xảy ra lỗi run-time, máy tính dừng thực thi chương trình và hiển thị thông báo chuẩn đoán tại dòng lệnh gây ra lỗi;
- Lỗi logic và lỗi ngữ nghĩa: Là loại lỗi khó tìm thấy và sửa chữa nhất vì dấu hiệu lỗi không thể hiện rõ ràng. Thông thường các chương trình chạy thành công, nhưng nó không trả về kết quả như mong đợi. Trình biên dịch không thể chuẩn đoán lỗi luận lý, do đó lập trình viên phải là người kiểm tra toàn bộ từng dòng code của mình và đảm bảo chương trình chạy đúng như mong đợi.

3.2.3 Một số kỹ thuật gỡ rối

1. Khai thác các đặc trưng của trình biên dịch.

Trình biên dịch tốt có thể làm một số phân tích tĩnh trên mã nguồn. Phân tích mã tĩnh là phân tích phần mềm mà thực hiện không cần chương trình thực thi được dịch từ mã nguồn phần mềm đó. Phân tích này có thể giúp phát hiện các vấn đề ngữ nghĩa cơ bản như sai kiểu, mã chết (code dead).

Ví dụ như `gcc`, trình biên dịch chuẩn đối với ngôn ngữ C trên hệ thống GNU/Linux có một số lựa chọn mà ảnh hưởng tới phân tích tĩnh gì được thực hiện. Chúng thường chia làm 2 loại: lựa chọn cảnh báo và cờ tối ưu. Theo như lựa chọn cảnh báo được đề cập, dưới đây là danh sách một số lựa chọn: `Wall`, `Wshadow`, ...

Trình biên dịch cũng hỗ trợ một số vấn đề tối ưu. Một số kích hoạt các trình biên dịch để làm phân tích dòng mã kỹ hơn và loại bỏ mã chết. Tuy nhiên các lập trình viên phải hiểu rằng công việc tối ưu ở một mức độ nào đó sẽ chống lại việc gỡ rối. Tối ưu là quá trình phân tích các dòng mã và sắp xếp lại các mã lệnh. Điều đó có nghĩa là khi tối ưu, mã có thể khác từ mã được viết ban đầu, điều đó sẽ khó khăn, thậm chí là không thể gỡ rối. Vì vậy, cờ tối ưu chỉ nên bật chỉ khi mã đã được gỡ rối xong.

2. Kỹ thuật gỡ rối dựa vào sử dụng `cout`.

Kỹ thuật `cout` xuất ra tên của các nội dung cần gỡ rối từ các lệnh của C++ vào thiết bị chuẩn như màn hình hoặc là file. Nó bao gồm thêm lệnh xuất (print) trong mã để theo dõi các luồng điều khiển. Đây là kỹ thuật gỡ rối khá thông dụng, đặc biệt với người mới lập trình. Ví dụ dưới đây minh họa kỹ thuật này. Lệnh xuất (printing) không nên sử dụng trực tiếp: nên định nghĩa macro để chúng ta có thể chuyển đổi chế độ gỡ rối được thuận tiện.

```

1 #ifndef DEBUG_H
2
3 #define DEBUG_H
4
5 #include <stdarg.h>
6
7 #if defined(NDEBUG) && defined(__GNUC__)
8
9 #define Pmsg(level, format, args ... ) ( ( void ) 0 )
10
11 #else
12
13 void Pmsg (int level , char * format , ... ) ;
14
15 /* p r i n t a message , i f i t i s c o n s i d e r e d s i g n i f i c a n t
    enough Adapted
16 from [ 9 ] , p . 174 */
17
18 #endif
19
20 #endif / * DEBUG_H * /

```

Hình 3.1: Ví dụ về kỹ thuật gỡ rối sử dụng `cout` - file định nghĩa.


```

1 #include <stdio.h>
2 #include "debug1.h"
3
4 extern int msglevel;
5
6 #if defined(NDEBUG) && defined(__GNUC__)
7
8 #else
9 void Pmsg(int level, char *format, ...){
10
11 #ifdef NDEBUG
12 #else
13     va_list args;
14     if (level > msglevel)
15         return;
16     va_start(args, format);
17     vfprintf(stderr, format, args);
18     va_end(args);
19
20 #endif
21 #endif
22 }

```

Hình 3.2: Ví dụ về kỹ thuật gỡ rối sử dụng **cout** - file thi hành.

Ở đây, **msglevel** là biến toàn cục được định nghĩa để điều khiển gỡ rối được in ra bao nhiêu. Sau đó, **pmsg(100, "Foo is %l\n", foo)** có thể được sử dụng để in giá trị của **foo** trong trường hợp **msglevel** nhận giá trị **100** hoặc lớn hơn. Chú ý rằng, tất cả mã gỡ rối này trong khi thi hành có thể bỏ qua bằng cách thêm **-DNDEBUG** vào cờ tiền xử lý.

3. Logging.

Logging là khái niệm in các thông điệp, đã được trình bày ở phần trước, tuy nhiên nó là khái quát hơn. Logging là trợ giúp thông dụng cho gỡ rối. Một ai đó thử ít nhất một lần để giải quyết một số vấn đề liên quan đến hệ thống đều biết file log hữu ích như thế nào. Logging nghĩa là tự động ghi lại các thông điệp thông tin hoặc sự kiện để giám sát trạng thái chương trình của bạn và chuẩn đoán những vấn đề gặp phải. Logging là giải pháp thực sự cho kỹ thuật **cout**.

4. Assertions.

Assertions là biểu thức để đánh giá là đúng tại một điểm nào đó trong mã nguồn. Nếu assertion là sai thì lỗi được tìm thấy. Viết assertions trong mã nguồn tạo cho các giả định tường minh. Trong C/C++ file **assert.h** phải được thêm vào chương trình và biểu thức bạn muốn khẳng định phải được viết như đối số macro, ví dụ **assert(var > 0)**. Chương trình sẽ hủy bỏ khi khẳng định thất bại và thông báo lỗi sẽ đưa ra chính xác dòng mã và file sử dụng **Assertion**. Bởi vì **assert** là macro nên nó có thể dễ dàng bỏ qua trong phiên bản cuối cùng khi biên dịch mã chương trình đó (dịch ở chế độ không gỡ rối). Nếu sử dụng **g++**, bạn phải sử dụng cờ tiền xử lý **-DNDEBUG**.

5. Các công cụ gỡ rối

Các công cụ gỡ rối cho phép làm việc thông qua từng dòng lệnh của mã để tìm ra lỗi ở đâu và tại sao. Nó cho phép làm việc tương tác, kiểm soát việc thực hiện các chương trình, dừng chương trình ở thời điểm khác nhau, kiểm tra các biến, thay đổi dòng mã trong khi đang thi hành chương trình. Để sử dụng công cụ gỡ rối, chương trình phải được biên dịch với thông tin gỡ rối được chèn vào. Thông tin này được cung cấp kí hiệu gỡ rối thông qua chương trình dịch trong mã nhị phân. Kí hiệu gỡ rối này mô tả hàm và biến ở đâu trong bộ nhớ. Thi hành chương trình với kí hiệu gỡ rối có thể chạy như bình thường, tuy nhiên nó chạy chậm hơn đôi chút.

3.2.4 Giải pháp và vấn đề liên quan đến C/C++

Phần này, chúng ta sẽ tập trung vào những vấn đề xuất hiện khi lập trình với ngôn ngữ C hoặc C++. Hiện nay, C++ là ngôn ngữ phổ biến nhất giải quyết các bài toán phức tạp vì vậy nó khá thích hợp để minh họa thực hiện một số kỹ thuật gỡ rối cho các bài toán chung mà các lập trình viên ít kinh nghiệm thường gặp phải khi lập trình. **Quá trình xây dựng chương trình trong C++.**

Trước khi kiểm tra các vấn đề phổ biến nhất được tạo ra bởi lập trình C/C++, rất hữu ích để tóm tắt lại các bước liên quan trong việc xây dựng và chạy một chương trình C++/C. Chương trình trong C++/C có thể được xây dựng từng bước, ví dụ nó có thể chia thành các bước nhỏ hơn. Trong môi trường Unix, xây dựng chương trình chia thành 5 bước:

- *Tiền xử lý*: Trong pha này bao gồm file header và macro được xử lý; đầu ra của pha tiền xử lý vẫn là mã nguồn C/C++.
- *Biên dịch*: Biên dịch là dịch mã nguồn C/C++ thành mã hợp ngữ.
- *Hợp ngữ*: Mã hợp ngữ được dịch thành mã đối tượng nhị phân. Kết quả thường là file với phần mở rộng là .o.
- *Liên kết*: Nhiệm vụ của liên kết là kết hợp các file đối tượng và thư viện thành file thi hành.
- *Nạp liên kết động*: Bước cuối cùng bao gồm nạp các thư viện (hoặc một phần thư viện) được yêu cầu bởi liên kết động để chạy file thi hành.

Cấp phát bộ nhớ động.

Trong C/C++, các lập trình viên có thể cấp phát hoặc giải phóng bộ nhớ động một cách minh bạch (thông qua **malloc/free** hoặc **new/delete**). Nếu bộ nhớ cấp phát hoặc giải phóng sai thì nó gây ra lỗi trong lúc chạy chương trình (mất bộ nhớ, hỏng bộ nhớ, ...).

Lỗi chung thường gặp là: cố gắng sử dụng bộ nhớ mà chưa được cấp phát; truy cập bộ nhớ đã được giải phóng; giải phóng bộ nhớ nhiều lần; cấp phát bộ nhớ mà không giải phóng chúng. Nghiêm trọng nhất xảy ra đối với vấn đề bộ nhớ là đổ vỡ chương trình. Nếu chương trình không đổ vỡ, hành vi của nó trở lên không thể đoán được bởi vì hỏng bộ nhớ như là cái bẫy: chương trình có thể chạy bình thường, giả như mọi thứ đều tốt, miễn là nó không rơi vào vùng hỏng bộ nhớ. Đôi khi vấn đề bộ nhớ không xuất hiện trong nhiều lần chạy chương trình và sau đó, bỗng nhiên chương trình đổ vỡ. May mắn, có một số công cụ để kiểm tra chương trình có vấn đề về bộ nhớ hay không. Chúng chia làm 2 loại như sau:

- Thư viện ngoài được thêm vào hoặc liên kết với chương trình thực thi;
- Chương trình thực thi mà điều khiển việc thực thi của chương trình.

Kiểm tra lỗi gọi hệ thống.

System call tracer là chương trình cho phép kiểm tra vấn đề tại ranh giới giữa mã của bạn và hệ điều hành. Chương trình của người sử dụng không thể tương tác trực tiếp với nhân của hệ điều hành. Nếu chương trình muốn truy cập đĩa cứng, nó không thể thực hiện trực tiếp mà nó phải gọi hàm hệ thống thích hợp chịu trách nhiệm chuyển dữ liệu giữa chương trình và đĩa cứng. Để tìm hàm nào được sử dụng bởi chương trình, chúng ta thường sử dụng công cụ System call tracer.

Công cụ System call tracer chuẩn trong GNU/Linux gọi là **strace**. Đây là công cụ mạnh mà đưa ra tất cả lời gọi hệ thống được sử dụng bởi chương trình người dùng. Dưới đây là ví dụ sử dụng strace để phát hiện lỗi chương trình.

```

1 #include <iostream>
2 #include <string>
3 #include <fstream>
4 #include <cstdlib>
5
6 using namespace std;
7 int main(int argc , char * argv[ ])
8 {
9     string filename;
10    string basename;
11    string extname;
12    string tmpname;
13    const string suffix("tmp");
14    /* for each commandline
15       argument ( which is an ordinary C string ) */
16
17    for (int i = 1; i < argc; ++i)
18    {
19        filename = argv[i]; // process argument as file name
20        string::size_type idx = filename.find('.'); // search period in name
21        if (idx == string::npos)
22        {
23            //file name does not contain any period
24            tmpname = filename; // HERE IS THE ERROR
25            // tmpname = filename + '.' + suffix;
26        }
27        else tmpname = filename;
28        // print file name and temporary name
29        // cout << filename << " => " << tmpname << endl ; // USEFUL
30    }
31    ifstream file(tmpname.c_str()) ;
32    if(!file)
33    {
34        cerr << "Can't open input file \"" << filename << ".tmp\"\n";
35        exit(EXIT_FAILURE);
36    }
37    char c ;
38    while (file.get(c))

```

```

39     cout.put(c);
40 }

```

Hình 3.3: Ví dụ về kỹ thuật gỡ rối sử dụng `strace`.

Đây là chương trình hoàn chỉnh, chúng ta có thể biên dịch với `g++` và chạy nó như sau:

```
g++ -o straceTest vidu3_chuong3.cpp
```

Chương trình đơn giản này cố gắng truy cập file văn bản với phần mở rộng của tên file là `tmp` nằm trong cùng thư mục. Tên file được mở phải được đưa vào như tham số dòng lệnh. Chương trình sẽ gắn thêm phần hậu tố vào tên và mở file đó. Để thực hiện chương trình này, file văn bản với tên thích hợp (ví dụ `list.tmp`) phải được tạo trước trong cùng thư mục với file chương trình này.

Chạy chương trình sẽ sinh ra thông báo lỗi `Can't open input file "list.tmp"`. Chương trình không thể tìm thấy file đầu vào, nhưng đây là hiện tượng khá lạ vì file đầu vào đó tồn tại trong thư mục. Như vậy là chương trình có lỗi.

Chúng ta sẽ sử dụng công cụ `strace` để tìm ra lỗi trong chương trình trên. Bắt đầu `strace` với tham số dòng lệnh:

```
strace -o strace.out ./straceTest list
```

Đầu ra của `strace` được viết trong file `strace.out`. Ví dụ như sau:

```

open("list", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
write(2, "Can\t open input file", 23) = 23
write(2, "list", 4) = 4
write(2, ".tmp\\n", 6) = 6
exit_group(1) = ?

```

Kiểm tra danh sách các lời gọi của hệ thống, chúng ta có thể thấy ngay rằng vấn đề lỗi ở đây là tên file bởi vì tham số trong hàm mở file cho chúng ta biết rằng cố gắng mở file là `list` chứ không phải là `list.tmp`.

3.3 Lập trình không lỗi

Lập trình không lỗi gần như là việc không thể làm được. Theo qui luật thì "Mọi chương trình đều có ít nhất 1 lỗi". Vì vậy, ta cần lập trình ít lỗi nhất có thể. Dưới đây là một số qui tắc để lập trình ít lỗi nhất:

- Thiết kế chương trình dễ hiểu, đơn giản. Nếu làm việc trên hệ thống có sẵn cần hiểu kỹ hơn cấu trúc của hệ thống này;
- Kiểm tra kỹ từng dòng mã mình viết;
- Quản lý môi trường: version cũ, ngay trước đó, version hiện tại.

Bài tập

1. Kiểm thử chương trình sau với điều kiện cận của nó.

(a) Hàm tính giai thừa:

```
int factorial (int n)
{
    int fac;
    fac = 1;
    while (n-->0)
        fac *= n;
    return fac;
}
```

(b) Hàm copy chuỗi ký tự từ con trỏ nguồn src vào con trỏ đích dest:

```
void strcpy(char *dest, char *src)
{
    for (int i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
}
```

(c) Một hàm copy chuỗi ký tự khác nhưng ở đây là copy **n** ký tự từ **s** vào **t**:

```
void strncpy(char *t, char *s, int n)
{
    while (n > 0 && *s != '\0'){
        *t = *s;
        t++; s++;
        n--;
    }
}
```


Chương 4

Hàm

4.1 Thiết kế từ trên xuống (top-down)

Làm thế nào để chúng ta bẻ gãy một bó đũa. Hay làm thế nào để có thể “ăn” hết được một con voi (câu đố trong một bài toán cổ). Câu trả lời thật dễ dàng: hãy chia nhỏ bó đũa và bẻ gãy từng chiếc một, cũng tương tự hãy chia nhỏ chú voi ra và ăn từng phần một. Đầu tiên ta có thể chia chú voi thành 4 phần: đầu, mình, tứ chi và đuôi. Dĩ nhiên một lúc không thể giải quyết hết phần đầu, vậy ta lại chia nhỏ đầu voi thành các bộ phận: tai mắt, mũi, họng, vòi ... , thậm chí chiếc vòi quá dài thì ta lại “phân khúc” tiếp. Đây là ý tưởng của việc thiết kế thuật toán từ trên xuống, tức chia nhỏ bài toán cần giải quyết thành nhiều phần, lại tiếp tục chia nhỏ các phần để có những phần nhỏ hơn, và tiếp tục như vậy cho đến khi bài toán đã trở thành tập hợp những bài toán nhỏ, đủ nhỏ để ta có thể giải quyết được trực tiếp không cần phải chia nữa. Ông tổ của C/C++ cũng đã đưa ra một ví dụ là hãy viết một chương trình để máy có thể đánh cờ với người. Thoạt đầu, ta cảm thấy bài toán rất khó, khó biết được phải bắt đầu từ đâu, tuy nhiên nếu kiên trì “chia nhỏ” bài toán đến một lúc nào đó ta sẽ thấy giải quyết bài toán này có lẽ không có gì khó khăn lắm. Đó chính là ưu điểm của chiến lược thiết kế top-down (thiết kế từ trên xuống).

Trong thực tế, thiết kế và lập trình từ trên xuống thường được kết hợp với thiết kế và lập trình từ dưới lên. Trong tiếp cận từ dưới lên, đầu tiên hãy giải bài toán đơn giản, những “trường hợp riêng” của bài toán tổng quát và lưu lại kết quả, sau đó kết hợp dần kết quả, nghiệm của các bài toán nhỏ thành nghiệm của bài toán lớn hơn và tiếp tục quá trình để thu được nghiệm của bài toán cuối cùng.

Khi viết một chương trình máy tính cũng vậy, với một chương trình lớn, thông thường ta nên chia nhỏ chương trình thành tập hợp các chương trình con nhỏ hơn, mỗi chương trình con này giải quyết một vấn đề cụ thể của chương trình, thường được gọi là hàm hay thủ tục. Trong ngôn ngữ C/C++ các chương trình con này được gọi chung là hàm. Và như vậy, một chương trình C/C++ là một tập hợp các hàm, mỗi hàm giải quyết một vấn đề và tập hợp các hàm này sẽ giải quyết tổng thể chương trình.

4.2 Hàm

4.2.1 Ý nghĩa của hàm

Hàm là một chương trình con trong chương trình lớn. Nhiệm vụ của hàm cũng giống như chương trình là giải quyết hoàn chỉnh một bài toán (con) nào đó. Do vậy, hàm cũng làm việc theo cách chung: *nhận hoặc không, nhận một hoặc nhiều đầu vào (là các tham số), xử lý các đầu vào này và cuối cùng trả lại (hoặc không) một giá trị kết quả nào đó cho những nơi gọi đến nó*. Như vậy, một chương trình C/C++ là tập hợp của các hàm, trong đó luôn luôn có một hàm chính với tên gọi `main()`, khi chạy chương trình, hàm `main()` sẽ được chạy đầu tiên và gọi đến hàm khác. Kết thúc hàm `main()` cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những môđun riêng rẽ, hoạt động độc lập với ngữ nghĩa của chương trình lớn, có nghĩa một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác. Khi một hàm đã được viết hoàn chỉnh để thực hiện một nhiệm vụ nào đó thì người sử dụng sẽ không cần quan tâm đến hàm *làm việc thế nào* mà chỉ cần biết hàm *cần gì (input)* và *giải quyết được việc gì (output)* để sử dụng vào chương trình của mình. Ví dụ khi cần tính căn bậc 2 của `x`, người sử dụng chỉ cần viết `sqrt(x)` và tin tưởng kết quả trả lại chắc chắn là căn bậc 2 của `x` mà không cần quan tâm hàm này thực hiện thế nào để cho ra kết quả đúng.

Về mặt kỹ thuật, hàm có một số đặc trưng:

- Nằm trong hoặc ngoài văn bản (file) có chương trình gọi đến hàm. Trong một văn bản có thể chứa nhiều hàm. Đối với người học ban đầu, có thể viết tất cả hàm (cùng với hàm `main()`) trong cùng một file văn bản.
- Được gọi (sử dụng) từ chương trình chính (`main()`), từ hàm khác hoặc từ chính nó. Trường hợp một hàm lại gọi đến chính nó thì ta gọi hàm là đệ quy.
- Có 3 cách truyền dữ liệu (đầu vào) cho hàm: Truyền theo giá trị (đối là các biến có kiểu thông thường), truyền theo tham chiếu (đối là biến tham chiếu) và truyền theo dẫn trỏ (đối là con trỏ).

4.2.2 Cấu trúc chung của hàm

Cấu trúc một hàm bất kỳ được bố trí cũng giống như hàm `main()`. Cụ thể cú pháp của một hàm được viết như sau:

```
type_returned function_name(list of arguments)
{
    declarations ;           // kiểu, hằng, biến, phục vụ riêng cho hàm này
    statements ;             // các câu lệnh
    return (expression);     // cũng có thể nằm ở vị trí khác
}
```

- Kiểu hàm (`type_returned`) là kiểu của kết quả mà hàm tính toán xong và trả lại. Nếu hàm chỉ làm một công việc nào đó mà không trả lại kết quả thì kiểu hàm được khai báo là `void`.

- Danh sách tham đối hình thức còn được gọi ngắn gọn là danh sách đối (list of arguments) gồm dãy các đối cách nhau bởi dấu phẩy, đối có thể là một biến thường, biến tham chiếu hoặc biến con trỏ, hai loại biến sau ta sẽ trình bày trong các phần tới. Mỗi đối được khai báo giống như khai báo biến, tức là cặp gồm **<type>** **<argument_name>**.
- Tên biến được khai báo trong thân hàm không được trùng với tên đối.
- Câu lệnh **return** có thể nằm ở vị trí bất kỳ. Khi gặp **return** chương trình tức khắc thoát khỏi hàm và trả lại giá trị của biểu thức sau **return** (nếu có). Nếu không có câu lệnh **return** hoặc sau câu lệnh **return** không có biểu thức thì kiểu của hàm phải được khai báo là **void**.

Đoạn mã 4.1 là hàm in 10 dấu **'*'**. Do hàm chỉ làm công việc đơn giản là in ra màn hình và không trả lại giá trị nên ta sẽ xây dựng hàm không đối và kiểu giá trị trả lại là **void**.

```

1 void PrintTenAsterisks()
2 {
3     for (int count = 1; count <= 10; count++)
4         cout << "*";
5     cout << endl;
6     return ;    // không tra lại kết quả tính toán nào, chỉ kết thúc
7 }
```

Hình 4.1: Xây dựng hàm để in 10 dấu **'*'**.

Vai trò của các đối trong hàm (như **x** trong hàm **sqrt(x)**, ...) là “cửa ngõ” để hàm giao tiếp với NSD theo nghĩa: hàm xử lý với số liệu vào bất kỳ (là các đối) theo ý muốn của NSD. Từ đó, ta có thể mở rộng hàm trên để cho phép in một dãy dấu **'*'** với độ dài bất kỳ bằng cách thêm cho hàm một đối đại diện cho số dấu sao cần in. Ta được hàm mới như Hình 4.2.

```

1 void PrintAsterik(int numAsteriks)
2 {
3     int count;
4     for (count = 1; count <= numAsteriks; count++)
5         cout << "*" ;
6     cout << endl;
7     return ;
8 }
```

Hình 4.2: Hàm in chuỗi dấu **'*'** với độ dài là tham số.

Trong hàm **PrintAsterik(int numAsteriks)**, ta đưa thêm tham đối **numAsteriks** với nghĩa hàm sẽ in ra **numAsteriks** dấu **'*'**, từ đó trong vòng lặp **for** thay cho biến đếm **count** chạy cố định từ 1 đến 10 như trong hàm **PrintTenAsterisks()**, ở đây **count** sẽ chạy từ 1 đến **numAsteriks**. Như vậy, hàm **PrintAsterik()** cung cấp cách in dãy **'*'** mềm dẻo hơn hàm **PrintTenAsterisk()**. Để in dãy 10 dấu **'*'** có thể gọi hàm **PrintTenAsterisk()** hoặc **PrintAsterik(10)**, tuy nhiên để in dãy 12 dấu **'*'** hàm **PrintTenAsterisk()** sẽ không dùng được.

Dưới đây là chương trình minh họa cách sử dụng hàm **PrintAsterik()** để in tam giác cân gồm toàn dấu **'*'**.

```

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

```

3
4 void PrintAsterik(int numAsteriks)
5 {
6     for (int count = 1; count <= numAsteriks; count++)
7         cout << "*" ;
8     cout << endl;
9     return ;
10 }
11
12 int main()
13 {
14     int edge_length;           // Chiều dài cạnh đáy của tam giác
15     cout << "Enter the length of the base of an equilateral triangle: ";
16     cin >> edge_length;
17     int count1, count2;
18     for (count1 = 1; count1 <= edge_length; count1 += 2)
19     {
20         for (count2 = 1; count2 <= (edge_length - count1)/2; count2++)
21             cout << " ";
22         PrintAsterik(count1);
23     }
24
25     return 0;
26 }

```

Hình 4.3: Hàm in tam giác cân bằng dấu '*'.

Do độ phân giải của màn hình nên ta chỉ in các dòng có số lẻ dấu '*', do đó trong chương trình biến đếm count1 sẽ chạy từ 1 và sau mỗi bước được tăng lên 2 đơn vị.

Ví dụ sau định nghĩa hàm tính lũy thừa n (với n nguyên) của một số thực bất kỳ. Hàm này có hai đầu vào (đối thực x và số mũ nguyên n) và đầu ra (giá trị trả lại) kiểu thực với độ chính xác gấp đôi là x^n . (Đây chỉ là ví dụ minh họa hàm do NSD tự xây dựng, trong C/C++ đã có sẵn hàm `pow(x, y)` để tính x^y với cả x, y đều là số thực).

```

1 double Power(double x, int n)
2 {
3     double res = 1.0 ;           // biến lưu kết quả
4     for (int count = 1; count <= n; count++)
5         res *= x ;
6     return res;                 // trả lại kết quả và kết thúc hàm
7 }

```

Hình 4.4: Hàm tính x^n .

Để sử dụng hàm power, ví dụ cần tính giá trị của biểu thức $x^4 + 2x^3 - 3x + 1$ ta có thể viết thêm hàm `main()` như sau:

```

1 #include <iostream>
2 using namespace std;
3
4 double Power(double x, int n)
5 { ... Installed above ... }
6
7 int main()

```

```

8 {
9     double x, res;
10    cout << "Enter value of x = "; cin >> x;
11    res = Power(x, 4) + 2*Power(x, 3) - 2*x + 1;
12    cout << "Value of x^4 + 2x^3 - 2x + 1 is " << res << endl;
13
14    return 0;
15 }

```

Hình 4.5: Sử dụng hàm `power()`.

4.2.3 Khai báo hàm

Về nguyên tắc, mọi loại đối tượng (kiểu, hằng, biến, ...) đều phải được khai báo trước khi được sử dụng, điều này cũng được yêu cầu đối với hàm. Có nghĩa, nếu trong hàm `main()` hoặc hàm `B()` nào đó có gọi đến hàm `A()` thì `A()` phải được viết trước hàm `main()` và hàm `B()`. Tuy nhiên, trong các chương trình lớn để che giấu bớt chi tiết ta chỉ cần *khai báo* `A()` trước `main()` hoặc `B()`, còn việc viết (*định nghĩa, cài đặt*) `A()` có thể được để sau.

```

1 #include <iostream>
2 using namespace std;
3
4 double Power(double, int);           // khai bao ham power (truoc)
5
6 int main()
7 {
8     cout << "Value of 5^2 is " << Power(5, 2) << endl;    // main gọi power
9     return 0;
10 }
11
12 double Power(double x, int n)        // cai dat ham power (sau)
13 {
14     double res = 1.0 ;
15     for (int count = 1; count <= n; count++)
16         res *= x ;
17     return res;
18 }

```

Hình 4.6: Khai báo hàm trước khi gọi.

Để khai báo hàm chỉ cần viết dòng tiêu đề của hàm và kết thúc bằng dấu `;`. Ngoài ra trong khai báo có thể để tên các đối hoặc không, như ví dụ trên ta có thể khai báo:

```
double power(double x, int n);    hoặc    double power(double, int);
```

Ví dụ :

```

int rand100();           // không doi, kiểu hàm là int
int square(int);         // một doi kiểu int, kiểu hàm là int
void alltrim(char[]) ;   // một doi kiểu xâu, hàm không trả lại giá trị
int sum(int, int);       // hai doi kiểu int, kiểu hàm là int

```

Chú ý về khai báo và định nghĩa hàm

- Danh sách đối trong khai báo hàm có thể chứa hoặc không chứa tên đối, thông thường ta chỉ khai báo kiểu đối chứ không cần khai báo tên đối, trong khi ở dòng đầu tiên khi cài đặt (định nghĩa) hàm phải có tên đối đầy đủ.
- Cuối khai báo hàm phải có dấu chấm phẩy ';', trong khi cuối dòng đầu tiên của định nghĩa hàm không có dấu chấm phẩy.
- Hàm có thể không đối (danh sách đối rỗng), tuy nhiên cặp dấu ngoặc sau tên hàm vẫn phải được viết. Ví dụ `PrintAsterisk()`, `lamtho()`, `vietgiaotrinh()`, ...
- Giả sử có 2 hàm `A()`, `B()` được cài đặt (định nghĩa) theo thứ tự `A()`, `B()`. Khi đó `B()` được phép có lời gọi đến `A()` (vì `A()` đã được biết trước `B()`), ngược lại `A()` không gọi được đến `B()` (`A()` chưa biết `B()`). Để `A()` gọi được `B()`, cần phải khai báo `B()` trước cài đặt của `A()`, khi đó cấu trúc của đoạn chương trình như sau:

```
Khai báo B;
Cài đặt A;    // trong A có lời gọi đến B
Cài đặt B;    // trong B có lời gọi đến A
```

Một cách tự nhiên, NSD thường cài đặt các hàm “con” trước và `main()` được cài đặt cuối cùng. Tuy nhiên đối với chương trình lớn, việc đặt `main()` ở cuối sẽ gây khó khăn cho việc tìm đến và đọc hiểu nội dung chính của chương trình. Vì vậy, hàm `main()` thường được viết đầu tiên sau đó mới đến các hàm khác. Để làm được việc này, cần có khai báo danh sách tất cả các hàm con lên đầu chương trình trước khi viết hàm `main()` (là nơi chứa nội dung chính của chương trình).

Để “ẩn giấu” bớt những chi tiết (không phải là nội dung chính), việc khai báo các biến, hằng, kiểu, hàm và cài đặt các hàm con có thể được viết trong một file khác. Chương trình nào cần sử dụng đến các đối tượng này có thể “`#include`” file khai báo này vào đầu chương trình của mình, sau đó có thể sử dụng mà không cần quan tâm đến cách làm việc cụ thể của đối tượng (ví dụ với hàm, NSD chỉ cần biết đầu vào, giá trị trả lại của hàm là gì (tức công dụng của hàm) để sử dụng đúng ngữ nghĩa của chương trình mà không cần biết hàm làm việc như thế nào). Chẳng hạn, NSD chỉ cần biết hàm `sqrt(x)` có công dụng tính căn bậc 2 của đối `x`, đã được khai báo và cài đặt sẵn trong file `math.h`, vì vậy để sử dụng `sqrt(x)`, chỉ cần đặt thêm câu khai báo `#include <math.h>` vào đầu chương trình.

Tóm lại, một chương trình lớn (để dễ đọc) thường được phân chia thành ít nhất 3 file: file khai báo các đối tượng, file cài đặt cụ thể các hàm và file chứa chương trình chính.

4.3 Cách sử dụng hàm

4.3.1 Lời gọi hàm

Một hàm sau khi khai báo, cài đặt sẽ được sử dụng thông qua “lời gọi hàm”. Lời gọi hàm được phép xuất hiện trong bất kỳ biểu thức, câu lệnh nào cần đến nó. Để gọi hàm, ta chỉ cần viết tên hàm và danh sách các giá trị thực sự để truyền cho các đối.

```
func_name(list_of_values) ;
```

Ví dụ cần gán `y` bằng căn bậc hai của 8, ta có thể viết `y = sqrt(8);`

- **list_of_values** là danh sách tham đối thực sự còn gọi là danh sách giá trị gồm các giá trị cụ thể (có thể là biểu thức) để gán lần lượt cho các đối hình thức của hàm. Khi hàm được gọi, việc đầu tiên chương trình sẽ tính toán các giá trị cụ thể, gán các giá trị này cho các tham đối hình thức theo thứ tự tương ứng trong danh sách đối của hàm, sau đó hàm tiến hành thực hiện các câu lệnh của hàm (để tính kết quả).
- Danh sách tham đối thực sự truyền cho tham đối hình thức có số lượng bằng với số lượng đối trong hàm và được truyền cho đối theo thứ tự tương ứng. Các tham đối thực sự có thể là các hằng, các biến hoặc biểu thức. Tên biến trong tham đối giá trị có thể trùng với tên của tham đối hình thức. Ví dụ ta có hàm `power(double x, int n);` và lời gọi hàm `power(a+1, 4);` thì `x` và `n` là các đối hình thức, `a+1` và `4` là các đối thực sự hoặc giá trị. Các đối hình thức `x` và `n` sẽ lần lượt được gán bằng các giá trị tương ứng là `x = a+1` và `n = 4` trước khi tiến hành các câu lệnh trong phần thân hàm. Giả sử `a` là biến trước đó nhận giá trị `2` thì lời gọi hàm `power(a+1, 4)` sẽ cho kết quả là $3^4 = 81$. Ngược lại, để tính 4^3 thì lời gọi hàm phải là `power(4, a+1)`.
- Các giá trị tương ứng được truyền cho đối phải có kiểu cùng với kiểu đối (hoặc C/C++ có thể tự động chuyển được về kiểu của đối).
- Khi một hàm được gọi, chương trình nơi gọi tạm thời chuyển điều khiển đến thực hiện hàm được gọi. Sau khi kết thúc thực hiện hàm, điều khiển lại được trả về thực hiện tiếp câu lệnh sau lệnh gọi hàm của nơi gọi.

Ví dụ: Để tìm phân số tối giản ta cần cài đặt hàm tìm Ước chung lớn nhất của 2 số và sau đó sử dụng hàm này để giản ước tử và mẫu của một phân số như trong chương trình sau.

```

1 #include <iostream>
2 using namespace std;
3
4 int GCD(int m, int n)    // Ham tim UCLN cua m va n
5 {
6     while (m != n)
7     {
8         if (m > n) m -= n;
9         else n -= m;
10    }
11    return m;
12 }
13
14 int main()
15 {
16     int numerator, denominator, gcd;
17     cout << "Enter the numerator = ";
18     cin >> numerator;
19     cout << "Enter the denominator = ";
20     cin >> denominator;
21     gcd = GCD(numerator, denominator);
22     cout << "Fraction in its lowest terms = "
23          << numerator/gcd << "/" << denominator/gcd << endl;
24
25     return 0;

```

Hình 4.7: Hàm tìm ước chung lớn nhất.

4.3.2 Hàm với đối mặc định

Trong phần trước, chúng ta đã khẳng định số lượng tham đối thực sự phải bằng số lượng tham đối hình thức khi gọi hàm. Tuy nhiên, trong thực tế rất nhiều lần hàm được gọi với các giá trị của một số tham đối hình thức được lặp đi lặp lại (cố định). Trong trường hợp như vậy lúc nào cũng phải viết một danh sách dài các tham đối thực sự giống nhau cho mỗi lần gọi là một công việc không mấy thú vị. Từ thực tế đó, C++ đưa ra một cú pháp mới về hàm sao cho một danh sách tham đối thực sự trong lời gọi không nhất thiết phải viết đầy đủ nếu một số trong chúng đã có sẵn những giá trị định trước. Cú pháp này được gọi là hàm với tham đối mặc định và được khai báo với cú pháp như sau:

```
type_name function_name(arg_1, ..., arg_k,
                        darg_1 = v_1, ..., darg_m = v_m)
```

- Các đối `arg_1, ..., arg_k` và đối mặc định `darg_1, ..., darg_m` đều được khai báo như cũ nghĩa là gồm có kiểu đối và tên đối.
- Riêng các đối mặc định `darg_1, ..., darg_m` có gán thêm các giá trị mặc định `v_1, ..., v_m`. Một lời gọi bất kỳ khi gọi đến hàm này đều phải có đầy đủ các tham đối thực sự ứng với các `arg_1, ..., arg_k` nhưng có thể có hoặc không các tham đối thực sự ứng với các đối mặc định `darg_1, ..., darg_m`. Nếu tham đối nào không có tham đối thực sự (trong lời gọi) thì nó sẽ được tự động gán giá trị mặc định `v_1, ..., v_m` đã khai báo.

Ví dụ :

- Xét hàm `double power(double x, int n = 2);` Hàm này có một tham đối mặc định là số mũ `n`, nếu lời gọi hàm bỏ qua số mũ này thì chương trình hiểu là tính bình phương của `x` (`n = 2`). Ví dụ lời gọi `power(4, 3)` được hiểu là 4^3 còn `power(4)` được hiểu là 4^2 .
- Hàm tính tổng 4 số nguyên: `int sum(int m, int n, int k = 0, int h = 0);` khi đó có thể tính tổng của 5, 2, 3, 7 bằng lời gọi hàm `sum(5, 2, 3, 7)` hoặc có thể chỉ tính tổng 3 số 4, 2, 1 bằng lời gọi `sum(4, 2, 1)` hoặc cũng có thể gọi `sum(6, 4)` chỉ để tính tổng của 2 số 6 và 4.

```
1 #include <iostream>
2 using namespace std;
3
4 int Sum(int m, int n, int k = 0, int h = 0)
5 {
6     return m + n + k + h;
7 }
8
9 int main()
10 {
11     cout << "5 + 2 + 3 + 7 equals to " << Sum(5, 2, 3, 7) << endl;    // 17
```

```

12     cout << "4 + 2 + 1 equals to " << Sum(4, 2, 1) << endl;           // 7
13     cout << "6 + 4 equals to " << Sum(6, 4) << endl;                 // 10
14     return 0;
15 }

```

Hình 4.8: Hàm `sum` tính tổng 4 số với 2 tham số mặc định.

Chú ý: Các đối ngầm định phải được khai báo liên tục và xuất hiện cuối cùng trong danh sách đối. Ví dụ:

```

int sum(int x, int y=2, int z, int t=1); // sai vì các đối mặc định không liên tục
void sub(int x=0, int y);                // sai vì đối mặc định không ở cuối

```

4.4 Biến toàn cục và biến địa phương

4.4.1 Biến địa phương (biến trong hàm, trong khối lệnh)

Biến địa phương (còn được gọi là biến cục bộ) là các biến được khai báo trong thân của hàm và chỉ có tác dụng trong hàm này. (kể cả các biến khai báo trong hàm `main()` cũng chỉ có tác dụng riêng trong hàm `main()`). Từ đó, tên biến trong các hàm khác nhau vẫn được phép trùng nhau. Các biến của hàm nào sẽ chỉ tồn tại trong thời gian hàm đó hoạt động. Khi bắt đầu hoạt động, các biến này được tự động sinh ra (trong bộ nhớ) và đến khi hàm kết thúc các biến này sẽ mất đi. Tóm lại, một hàm được xem như một đơn vị độc lập, khép kín.

Tham đối của các hàm cũng được xem như biến cục bộ. Dưới đây ta nhắc lại một chương trình nhỏ gồm 3 hàm: lũy thừa, xóa màn hình và `main()`. Mục đích để minh họa biến cục bộ.

```

1 #include <iostream>
2 using namespace std;
3
4 double Power(double x, int n)
5 {
6     int i;
7     double res = 1;
8     for (i = 1; i <= n; i++)
9         res *= x;
10    return res;
11 }
12
13 void ClearScreen(int n)           // xoa man hinh n lan
14 {
15     int i;                       // i, n la cuc bo cua clearScreen
16     for (i = 1; i <= n; i++)
17     {
18         cout << "Press any key to clear screen (" << i << "th time)";
19         cin.get();                // yeu cau doc vao phim bat ky
20         system("CLS");
21     }
22 }
23
24 int main()
25 {

```

```

26  double x; int n;                                // x, n là cục bộ của main
27  cout << "Enter x = "; cin >> x;
28  cout << "Enter n = "; cin >> n;
29  cin.ignore();                                   // xóa kí tự còn lưu trong bộ đệm bàn phím
30  ClearScreen(3);                                // xóa màn hình 3 lần
31  cout << "Result = " << Power(x, n) << endl;    // in x ^ n
32  return 0;
33 }

```

Hình 4.9: Biến cục bộ.

Qua ví dụ trên ta thấy, các biến **count**, đối **n** được khai báo trong hai hàm: **power()** và **clearScreen()**. Biến **res** được khai báo trong **power** và **main()**, ngoài ra các biến **x** và **n** của **main()** còn trùng tên với đối của hàm **power()**. Tuy nhiên, tất cả khai báo trên đều hợp lệ và đều được xem như khác nhau. Có thể giải thích như sau:

- Tất cả các biến trên đều cục bộ trong hàm nó được khai báo.
- **x** và **n** trong **main()** có thời gian hoạt động dài nhất: Trong suốt quá trình chạy chương trình, chúng chỉ mất đi khi chương trình chấm dứt. Đối **x** và **n** trong **power()** chỉ tạm thời được tạo ra khi hàm **power()** được gọi đến và độc lập với **x**, **n** trong **main()**, nói cách khác tại thời điểm đó trong bộ nhớ có hai biến **x** và hai biến **n**. Khi hàm **power()** chạy xong biến **x** và **n** của nó tự động biến mất, còn **x** và **n** của **main()** vẫn còn tồn tại cho đến khi chạy hết chương trình.
- Tương tự 2 đối **n**, 2 biến **i** trong **power()** và **clearScreen()** cũng độc lập với nhau, chúng chỉ được tạo và tồn tại trong thời gian hàm của chúng được gọi và hoạt động. Tức chúng là 2 biến khác nhau, không ảnh hưởng đến nhau tại cùng thời điểm hoạt động. Việc đặt lại giá trị của **i** trong hàm **power()** sẽ không làm thay đổi giá trị của **i** trong **clearScreen()**.

4.4.2 Biến toàn cục (biến ngoài tất cả các hàm)

Biến toàn cục (còn được gọi là biến toàn thể, biến dùng chung) là các biến được khai báo bên ngoài của tất cả các hàm. Vị trí khai báo của chúng có thể từ đầu văn bản chương trình hoặc tại một vị trí bất kỳ nào đó giữa văn bản chương trình. Thời gian tồn tại của chúng là từ lúc chương trình bắt đầu chạy đến khi kết thúc chương trình giống như các biến trong hàm **main()**. Tuy nhiên, về phạm vi tác dụng của chúng là bắt đầu từ điểm khai báo chúng cho đến hết chương trình, tức chỉ các hàm khai báo về sau mới có thể sử dụng và thay đổi giá trị của chúng. Như vậy, các biến ngoài được khai báo từ đầu chương trình sẽ có tác dụng lên toàn bộ các hàm của chương trình. Nếu **x** là biến toàn cục và khai báo từ đầu chương trình thì các hàm **A()**, **B()**, **C()**, **main()** ... bất kỳ đều được phép truy cập đến và thay đổi giá trị của **x**.

4.4.3 Mức ưu tiên của biến toàn cục và địa phương

Trong phần trên, ta đã thấy biến toàn cục được dùng chung trong tất cả các hàm. Ví dụ **x** được khởi tạo bằng **1**, sau đó trong **A()** có lệnh **x = x + 3**; thì **B()** sẽ nhìn thấy **x = 4**. Tuy nhiên, nếu trong hàm **A()** cũng khai báo một biến địa phương trùng tên với biến toàn cục **x** và có câu lệnh **x = 4** thì chỉ ảnh hưởng đến biến **x** của **A()** và không ảnh hưởng đến **B()** (

B() vẫn chỉ thấy biến **x** (toàn cục) với giá trị là **1**). Trường hợp này, ta cũng gọi là che giấu biến (biến **x** toàn cục được che giấu – không có tác dụng - trong **A()**)

Dưới đây là các ví dụ minh họa cho các giải thích trên.

Ví dụ 1: Biến toàn cục được dùng chung

Giả sử ta có trò chơi đơn giản sau: Đầu tiên, máy đưa ra một giá trị nguyên ngẫu nhiên. Sau đó, hai người chơi lần lượt cộng thêm một số (nguyên) vào kết quả trước. Ai cho ra kết quả là một số chia chắn cho 5 thì người đó thắng cuộc hoặc trò chơi chấm dứt nếu sau 3 lần cộng thêm mà số vẫn chưa chia hết cho 5.

Vì các kết quả được cộng thêm liên tục vào cùng một số nên chương trình sẽ dùng một biến **x** chung (toàn thể) để chứa kết quả này và 2 hàm cộng cho 2 người chơi.

```
1 #include <iostream>
2 using namespace std;
3
4 int x;
5
6 void PlayerOne()
7 {
8     int num;
9     cout << "First Player enters an integer : ";
10    cin >> num;
11    x += num;
12 }
13
14 void PlayerTwo()
15 {
16     int num;
17     cout << "Second Player enters an integer : ";
18     cin >> num;
19     x += num;
20 }
21
22 int main()
23 {
24     x = rand();
25     int time = 1;
26     while (time <= 3)
27     {
28         PlayerOne();
29         if (x % 5 == 0)
30         {
31             cout << "Your obtained number is " << x << ". You win" << endl;
32             break;
33         }
34         PlayerTwo();
35         if (x % 5 == 0)
36         {
37             cout << "Your obtained number is " << x << ". You win" << endl;
38             break;
39         }
40         time ++;
41     }
```

```

42     if (time > 3) cout << "Game Over\n";
43     return 0;
44 }

```

Hình 4.10: Trò chơi cộng số để được số chia hết cho năm.

Một vài tình huống:

- Nếu **x** chỉ được khai báo trong **main()**, chương trình sẽ báo lỗi vì 2 hàm con không nhận biết **x**.
- Nếu khai báo cả trong **main()** và 2 hàm con thì lời giải không đúng với mục đích trò chơi vì khi đó mỗi người đều chơi trên số của riêng mình.
- Tương tự, nếu **x** được khai báo toàn thể và trong 1 hàm con, khi đó 2 người sẽ chơi trên 2 biến khác nhau : một biến toàn thể và một biến cục bộ không đúng với yêu cầu trò chơi.

Ví dụ 2: Ảnh hưởng của biến dùng chung.

Chúng ta xét lại các hàm **power()** và **clearScreen()**. Chú ý rằng trong cả hai hàm này đều có biến **i** làm biến đếm cho câu lệnh **for**, vì vậy chúng ta có thể khai báo **i** một lần như một biến ngoài (để dùng chung cho **power()** và **clearScreen()**), ngoài ra **x** cũng có thể được khai báo như biến ngoài. Cụ thể:

```

1  #include <iostream>
2  using namespace std;
3
4  int i;                                // Khai bao bien i dung chung
5  double x;                             // Khai bao bien x dung chung
6  double Power(double x, int n);        // Khai bao ham power
7  void ClearScreen(int n);              // Khai bao ham clearScreen
8
9  int main()
10 {
11     x = 2;
12     i = 5;
13     ClearScreen(3);                    // xoa man hinh 3 lan
14     cout << x << " ^ " << i << " = " << Power(x, i) << endl;    // in 2 ^ 5
15     return 0;
16 }
17
18 double Power(double x, int n)          // Cai dat ham power
19 {
20     double res = 1;
21     for (i = 1; i <= n; i++)
22         res *= x;
23     return res;
24 }
25
26 void ClearScreen(int n)                // Cai dat ham xoa man hinh
27 {
28     for (i = 1; i <= n; i++)
29     {
30         cout << "Press any key to clear screen (" << i << "th time)";

```

```

31     cin.get();
32     system("CLS");
33 }
34 }

```

Hình 4.11: Ảnh hưởng của biến dùng chung.

Nhìn vào hàm `main()` ta thấy giá trị 2^5 được tính bằng cách đặt $x = 2$, $i = 5$ và gọi hàm `power(x, i)`. Kết quả ta mong muốn sẽ là giá trị 32, tuy nhiên không đúng như vậy. Trước khi in kết quả, hàm `clearScreen(3)` đã được gọi để xóa màn hình 3 lần. Hàm này sử dụng một biến ngoài `i` để làm biến đếm cho mình trong vòng lặp `for` và sau khi ra khỏi `for` (cũng là kết thúc `clearScreen(3)`), `i` nhận giá trị 4. Biến `i` ngoài này lại được sử dụng trong lời gọi `power(x, i)` của hàm `main()`, tức tại thời điểm này $x = 2$ và $i = 4$, kết quả in ra màn hình sẽ là $2^4 = 16$ thay vì 32 như mong muốn.

Tóm lại, “điểm yếu” dẫn đến sai sót của chương trình trên là ở chỗ lập trình viên đã “tranh thủ” sử dụng biến `i` cho 2 hàm `clearScreen()` và `main()` (bằng cách khai báo nó như biến ngoài) nhưng lại với mục đích khác nhau. Do vậy sau khi chạy xong hàm `clearScreen()`, biến `i` bị thay đổi khác với giá trị `i` được khởi tạo lúc ban đầu.

Để khắc phục lỗi trong chương trình trên, ta cần khai báo lại biến `i`: hoặc trong `main()` khai báo thêm `i` (nó sẽ che biến `i` ngoài), hoặc trong cả hai `clearScreen()` và `main()` đều có biến `i` (cục bộ trong từng hàm).

Từ đó, nên đề ra một vài nguyên tắc về biến dùng chung (toàn cục) và biến cục bộ:

- Nếu một biến chỉ sử dụng vì mục đích riêng của một hàm thì nên khai báo biến đó như biến cục bộ trong hàm. Ví dụ các biến đếm của vòng lặp, thông thường chúng chỉ được sử dụng thậm chí chỉ riêng trong vòng lặp chứ cũng chưa phải cho toàn bộ cả hàm, vì vậy không nên khai báo chúng như biến ngoài. Những biến cục bộ này sau khi hàm kết thúc chúng cũng sẽ kết thúc, không gây ảnh hưởng đến bất kỳ hàm nào khác. Một đặc điểm có lợi nữa cho khai báo cục bộ là chúng tạo cho hàm tính cách hoàn chỉnh, độc lập với mọi hàm khác, chương trình khác. Ví dụ hàm `clearScreen()` có thể mang qua chạy ở chương trình khác mà không phải sửa chữa gì nếu `i` đã được khai báo bên trong hàm. Trong ví dụ này, hàm `clearScreen()` vẫn hoạt động được nhưng trong chương trình khác nếu không có `i` như một biến ngoài (để `clearScreen()` sử dụng) thì hàm sẽ gây lỗi.
- Với các biến mang tính chất sử dụng chung rõ nét (đặc biệt với những biến kích thước lớn) mà nhiều hàm cùng sử dụng chúng với mục đích giống nhau thì nên khai báo chúng như biến ngoài. Điều này tiết kiệm được thời gian cho người lập trình vì không phải khai báo chúng nhiều lần trong nhiều hàm, tiết kiệm bộ nhớ vì không phải tạo chúng tạm thời mỗi khi chạy các hàm, tiết kiệm được thời gian chạy chương trình vì không phải tổ chức bộ nhớ để lưu trữ và giải phóng chúng. Ví dụ trong chương trình quản lý sinh viên, biến sinh viên được dùng chung và thống nhất trong hầu hết các hàm (xem, xóa, sửa, bổ sung, thống kê ...) nên có thể khai báo chúng như biến ngoài, điều này cũng tăng tính thống nhất của chương trình (mọi biến sinh viên là như nhau cho mọi hàm con của chương trình).
- Nguyên tắc tổng quát nhất là cố gắng tạo hàm một cách độc lập, khép kín, không chịu ảnh hưởng của các hàm khác và không gây ảnh hưởng đến hoạt động của các hàm khác đến mức có thể.

4.5 Tham đối và cơ chế truyền giá trị cho tham đối

Một hàm sau khi đã được khai báo, định nghĩa sẽ được phép sử dụng (gọi) trong những đoạn chương trình khác để gọi đến hàm cần cung cấp (truyền) các giá trị đầu vào cho danh sách tham đối. Có 3 cách truyền: Truyền thông qua biến thường (truyền theo tham trị), truyền thông qua biến tham chiếu (tham chiếu) và truyền thông qua biến con trỏ (tham trỏ). Mục này xét hai cách truyền theo tham trị và tham chiếu. Cách còn lại sẽ được đề cập đến trong chương sau.

4.5.1 Truyền theo tham trị

Ta xét lại ví dụ hàm `power(double x, int n)` để tính x^n . Giả sử trong chương trình chính ta có các biến `a`, `b`, `f` đang chứa các giá trị `a = 2`, `b = 3`, và `f` chưa có giá trị. Để tính a^b và gán giá trị tính được cho `f`, ta có thể gọi `f = power(a, b)`. Khi gặp lời gọi này, chương trình sẽ tổ chức như sau:

- Tạo 2 biến mới (tức 2 ô nhớ trong bộ nhớ) tương ứng với `x` và `n`. Gán nội dung các ô nhớ này bằng các giá trị trong lời gọi, tức `x` nhận giá trị của `a` (2) và `n` nhận giá trị của `b` (3).
- Tối phần khai báo (của hàm), chương trình tạo thêm các ô nhớ mang tên `res` và `i`.
- Tiến hành tính toán với `x = 2` và `n = 3`, gán kết quả cho `res`.
- Cuối cùng lấy kết quả trong `res` gán cho ô nhớ `f` (là ô nhớ có sẵn đã được khai báo trước, nằm bên ngoài hàm).
- Kết thúc, hàm quay về chương trình gọi. Do hàm `power` đã hoàn thành xong việc tính toán nên các ô nhớ cục bộ được tạo ra trong khi thực hiện hàm (gồm `x`, `n`, `res`, `i`) sẽ được xoá khỏi bộ nhớ. Kết quả tính toán được lưu giữ trong ô nhớ `f` (không bị xoá vì không liên quan gì đến hàm – được khai báo ngoài hàm).

Trên đây là truyền đối theo cách thông thường hay còn gọi là truyền theo tham trị. Với cách truyền này sau khi chạy xong hàm, giá trị của các biến bên ngoài sẽ không thay đổi (kể cả trong hàm có những câu lệnh thay đổi đối) như trong ví dụ sau:

```

1 #include <iostream>
2 using namespace std;
3
4 double Total_Bit(int bytes, int bits)
5 {
6     bits = 8*bytes + bits;
7     return bits;
8 }
9
10 int main()
11 {
12     int bytes = 4, bits = 3;
13     cout << bytes << " bytes and " << bits << " bits equals to " << Total_Bit(bytes,
14         bits) << " bits\n";
15     return 0;
16 }
```

Hình 4.12: Tính số bit.

Trong ví dụ trên ta cần tính tổng số bit của 4 byte và 3 bit thông qua hàm `Total_bit()`. Hàm này có 2 đối `bytes` và `bits`, trong hàm đối `bits` đã thay đổi giá trị (bằng 35), tuy nhiên biến `bits` ngoài hàm vẫn giữ giá trị như cũ (3).

Tương tự, ta xem thêm ví dụ về việc hoán đổi giá trị của hai biến `a = 3, b = 5` như sau.

```

1 #include <iostream>
2 using namespace std;
3
4 void Swap(int x, int y)
5 {
6     int temp ;
7     temp = x ;
8     x = y ;
9     y = temp ;
10    cout << "x = " << x << ", y = " << y << endl; // 5, 3 (x,y doi gia tri)
11 }
12
13 int main()
14 {
15     int a = 3, b = 5;
16     Swap(a, b) ;
17     cout << "a = " << a << ", b = " << b << endl; // 3, 5 (a,b van nhu cu)
18     return 0;
19 }
```

Hình 4.13: Trao đổi 2 số.

Màn hình gồm 2 dòng kết quả. Một dòng in ra giá trị `x = 5, y = 3`, đó là câu lệnh in của hàm `swap`, tức các tham đối `x` và `y` đã thay đổi giá trị, tuy nhiên dòng kết quả thứ hai (từ lệnh in của hàm `main`) giá trị của 2 biến `a, b` vẫn như cũ.

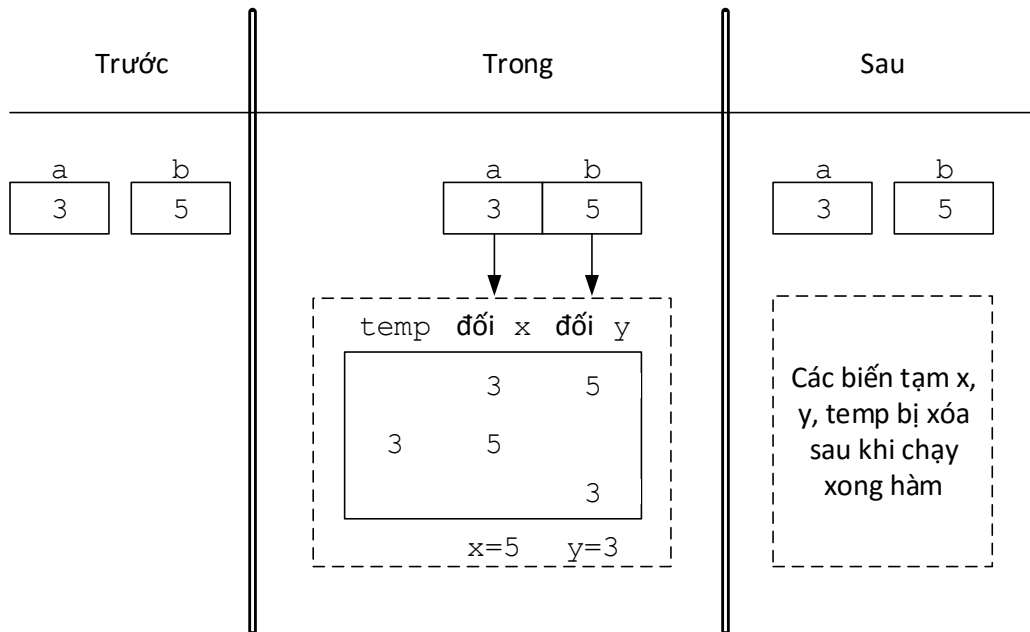
Tương tự như đã giải thích đối với lời gọi hàm `power()`, hình vẽ 4.14 minh họa cách làm việc của hàm `swap`, trước, trong và sau khi gọi hàm.

Như vậy, hàm `swap` cần được viết lại sao cho việc thay đổi giá trị không thực hiện trên các biến tạm (`x, y`) mà phải thực sự thực hiện trên các biến ngoài (`a, b`). Để thực hiện việc này, ta phải thông qua một dạng biến mới được gọi là biến tham chiếu. Hình ảnh đầu tiên trong ví dụ này là ta cho các đối `x, y` “tham chiếu” đến các biến ngoài `a, b`, hiệu ứng của việc này là các thay đổi trên `x, y` thực chất là sẽ làm thay đổi trên `a, b`.

4.5.2 Biến tham chiếu

Khai báo

Một biến có thể được gán cho một tên mới dưới dạng đặc biệt được gọi là bí danh hay tham chiếu của nó, khi đó chỗ nào xuất hiện biến (cũ) thì cũng tương đương như dùng bí danh (mới) và ngược lại. Nói cách khác là cho phép một loại biến đặc biệt được gọi là biến tham chiếu, “tham chiếu” tới biến thường nào đó để thay mặt biến thường này tham gia vào các thao tác, xử lý giá trị của các biến thường, tức sử dụng biến thường nhưng bằng tên của biến tham chiếu.



Hình 4.14: Các bước trao đổi biến của hàm **swap**.

Dưới đây là cú pháp khai báo một biến tham chiếu và cho nó tham chiếu đến một biến khác (đã có sẵn).

```
var_type    &reference_var_name = var_name;
```

Cú pháp khai báo thêm dấu và (**&**) trước tên biến cho phép tạo ra một biến tham chiếu mới (**reference_var_name**) và cho nó tham chiếu đến biến được tham chiếu (**var_name**) cùng kiểu và phải được khai báo từ trước. Khi đó, biến tham chiếu còn được gọi là bí danh của biến được tham chiếu.

Chú ý: Trong khai báo biến tham chiếu, luôn luôn phải cho biến này tham chiếu đến một biến nào đó (không có cú pháp khai báo chỉ tên biến tham chiếu mà không kèm theo khởi tạo). Ví dụ:

```
int hung, dung; // khai báo các biến nguyên hùng, dũng
int &ti = hung; // khai báo biến tham chiếu tí, tham chiếu đến hùng
int &teo = dung; // khai báo biến tham chiếu tèo, tham chiếu đến dũng
```

Từ vị trí này trở đi, việc sử dụng các tên hùng, tí hoặc dũng, tèo là như nhau.

Ví dụ:

```
hung = 3, dung = 5;
ti++; teo++; // tương đương hung++; dung++;
cout << hung << dung; // 4 6
```

Đặc trưng của biến tham chiếu

Trong bất kỳ biểu thức, câu lệnh, đều có thể thay tên biến thường bằng tên biến tham chiếu đến nó và ngược lại. Tuy nhiên, cách tổ chức bên trong của một biến tham chiếu khác với biến thường ở chỗ nội dung của nó là địa chỉ của biến thường mà nó đại diện, ví dụ câu lệnh `cout << dung ;` sẽ in nội dung (giá trị) của biến **dung**, nhưng `cout << teo;` sẽ không in nội dung của **teo** (là địa chỉ của **dung**) mà là in nội dung được chứa tại địa chỉ này, tức nội dung của biến **dung**. Cách thức làm việc của 2 câu lệnh trên là khác nhau nhưng cho cùng kết quả như nhau.

Tương tự, `dung++` làm tăng giá trị của `dung` và `teo++` cũng làm tăng giá trị của `dung` chứ không phải tăng giá trị của `teo`. Từ đó, khác biệt này có ích khi được sử dụng để truyền đối cho các hàm (`teo`) với mục đích làm thay đổi nội dung của biến ngoài (`dung`).

4.5.3 Truyền theo tham chiếu

Đối của một hàm có thể là các biến tham chiếu, khi đó việc thay đổi trên các đối này thực chất là thay đổi trên các biến ngoài mà nó tham chiếu. Các biến ngoài này được truyền cho các đối tham chiếu trong lời gọi hàm. Cách truyền này được gọi là truyền theo tham chiếu.

Ví dụ, hàm `swap` có thể được viết lại để thay đổi giá trị của các cặp số `a`, `b`. Khi đó, thay cho `x`, `y` là các đối thường như trong ví dụ trước, ở đây `x`, `y` sẽ là các đối tham chiếu. Các phần còn lại (nội dung của hàm, lời gọi hàm) không có gì thay đổi.

```

1 #include <iostream>
2 using namespace std;
3
4 void Swap(int &x, int &y)
5 {
6     int temp ;
7     temp = x ;
8     x = y ;
9     y = temp ;
10    cout << "x = " << x << ", y = " << y << endl; // 5, 3 (x,y doi gia tri)
11 }
12
13 int main()
14 {
15     int a = 3, b = 5;
16     Swap(a, b) ;
17     cout << "a = " << a << ", b = " << b << endl; // 5, 3 (a,b doi gia tri)
18     return 0;
19 }
```

Hình 4.15: Tráo đổi 2 số bằng tham chiếu.

Dưới đây là bảng chỉ ra sự khác biệt của 2 hàm swap.

Bảng 4.16: Khác biệt giữa truyền tham số theo tham trị và tham chiếu

	Tham trị	Tham chiếu
Khai báo đối	<code>void swap(int x, int y)</code>	<code>void swap(int &x, int &y)</code>
Câu lệnh	<code>t = x; x = y; y = t;</code>	<code>t = x; x = y; y = t;</code>
Lời gọi	<code>swap(a, b);</code>	<code>swap(a, b);</code>
Tác dụng	<code>a</code> , <code>b</code> không thay đổi	<code>a</code> , <code>b</code> có thay đổi

Sử dụng biến tham chiếu cho phép viết được mọi chương trình một cách bình thường như ý muốn. Ví dụ, viết hàm nhập giá trị cho các biến. Các biến này không thể là đối bình thường của hàm vì sau khi gọi hàm các biến cần nhập sẽ nhận giá trị mới, do vậy chúng cần được khai báo dưới dạng tham chiếu. Một trường hợp khác là khi hàm cần trả lại nhiều hơn một giá trị? Nếu số giá trị

trả lại bé thì ta có thể khai báo các đối tham chiếu để nhận các giá trị này (trường hợp giá trị trả lại là tập dữ liệu nào đó sẽ được bàn đến trong chương nói về mảng và cấu trúc).

Ví dụ hàm giải phương trình bậc 2. Hàm này có 3 đối là biến thường đại diện cho 3 hệ số **a**, **b**, **c** của phương trình. Vì hàm chỉ có thể trả lại một giá trị duy nhất trong khi phương trình có thể có 2 nghiệm. Do vậy trong danh sách đối của hàm cần thêm 2 đối tham chiếu để nhận giá trị của 2 nghiệm **x1**, **x2** này. Khi đó, hàm có thể không trả lại giá trị (**void**) hoặc cũng có thể trả lại giá trị là số lượng nghiệm của phương trình (0, 1 hoặc 2) (xem phần bài tập).

4.5.4 Hai cách truyền giá trị cho hàm và từ khóa **const**

Qua hai cách truyền theo tham trị và tham chiếu, có thể rút ra nhận xét về cách hoạt động của hàm khi có lời gọi:

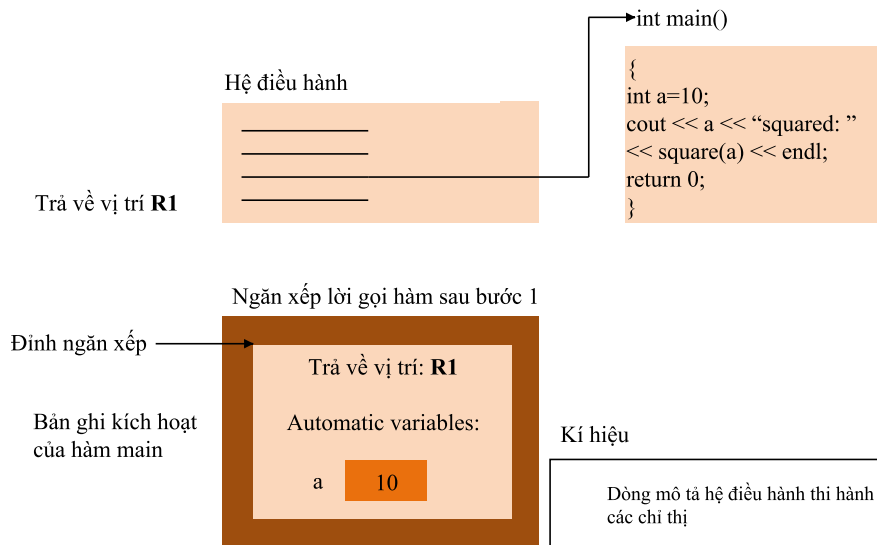
- Truyền theo tham trị : Đầu tiên, chương trình sẽ tạo ra các ô nhớ đại diện cho các đối của hàm, các ô nhớ này sẽ được lấp đầy giá trị bằng cách sao chép giá trị của các biến (hoặc biểu thức) tương ứng trong lời gọi (thao tác sao chép này sẽ mất thêm thời gian, đặc biệt đối với các biến có nội dung lớn). Tiếp theo, hàm thực hiện tính toán trên các giá trị đã sao chép (bản sao) và cuối cùng trả lại giá trị kết quả.
- Truyền theo tham chiếu: Mọi tính toán trên biến tham chiếu thực chất là trên các biến ngoài đã có sẵn. Hàm không mất thêm không gian (tạo ô nhớ tạm cho các đối) cũng không mất thêm thời gian (sao chép các biến). Như vậy, cách viết hàm dưới dạng đối tham chiếu có ưu điểm hơn các đối thường, ở chỗ:
 - Cho phép thay đổi giá trị biến ngoài.
 - Cải thiện tính hiệu quả của thuật toán (tốn ít bộ nhớ, chạy nhanh hơn).

Từ đó, liên quan đến các biến có nội dung lớn ta thường lợi dụng đối tham chiếu để tăng hiệu quả (tiết kiệm không gian, giảm thiểu thời gian thực hiện), tuy nhiên, cách viết này cũng vô tình cho phép thay đổi giá trị biến ngoài mà ta không mong muốn. Để ngăn cản việc thay đổi, C++ cho phép thêm từ khóa **const** trước các đối tham chiếu. Khi đó, nếu trong hàm có câu lệnh nào vô tình thay đổi giá trị của đối (được khai báo sau từ khóa **const**), chương trình dịch sẽ báo lỗi.

4.6 Ngăn xếp gọi hàm và các mẫu tin kích hoạt

Để hiểu cách gọi hàm trong C ++, đầu tiên ta cần xem xét một cấu trúc dữ liệu (tập hợp các mục dữ liệu có liên quan) được gọi là ngăn xếp (stack). Xem ngăn xếp như một chồng các đĩa ăn. Khi một đĩa được đặt vào chồng, nó thường được đặt ở phía trên (cho đĩa vào ngăn xếp). Tương tự như vậy, khi một đĩa được lấy ra khỏi chồng, nó thường được lấy ra từ phía trên (lấy đĩa ra khỏi ngăn xếp). Các ngăn xếp được biết như là vào cấu trúc dữ liệu sau ra trước (LIFO), phần tử cuối được đưa vào (chèn vào) ngăn xếp là phần tử đầu tiên được lấy ra khỏi ngăn xếp. Một trong những cơ chế quan trọng nhất cho sinh viên khoa học máy tính hiểu là ngăn xếp các lời gọi hàm (còn gọi là ngăn xếp thực thi chương trình). Cấu trúc dữ liệu này - làm việc "kịch bản phía sau" - hỗ trợ cơ chế gọi hàm/ trả về hàm. Nó cũng hỗ trợ việc khởi tạo, duy trì và hủy của các biến khi tự động gọi

Step 1: Hệ điều hành gọi hàm main để thực thi chương trình



Hình 4.17: Gọi stack sau khi hệ điều hành gọi hàm `main` để thực thi chương trình.

hàm. Giải thích hành vi vào sau ra trước của ngăn xếp (LIFO) với ví dụ xếp chồng đĩa lên nhau. Như trong hình 4.17 - 4.19, LIFO này chính là cách một hàm thực hiện khi trả về hàm gọi nó.

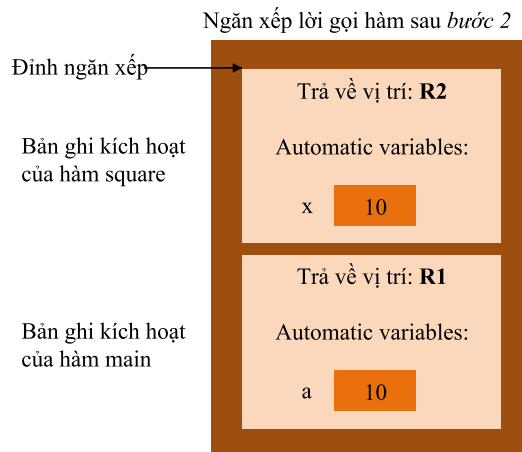
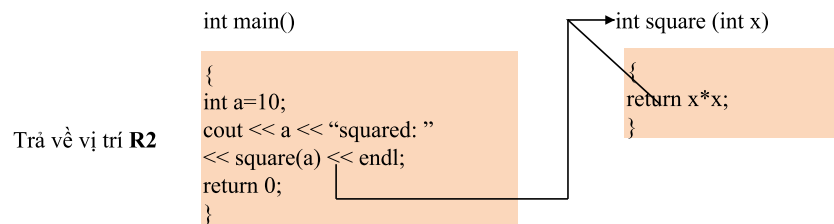
Theo mỗi hàm được gọi, có thể trong quá trình trả về, gọi các hàm khác - tất cả thực hiện trước bất cứ hàm trả về nào. Mỗi hàm phải trả lại quyền điều khiển các hàm gọi nó. Theo cách như vậy, phải giữ các địa chỉ trả về nơi các hàm cần trả lại quyền điều khiển đối với hàm gọi nó. Ngăn xếp là cấu trúc dữ liệu hoàn hảo để xử lý thông tin này. Mỗi lần hàm gọi một hàm khác, đối tượng được đẩy vào ngăn xếp. Đối tượng này được gọi là khung ngăn xếp hoặc mẫu tin kích hoạt, chứa địa chỉ trả về mà hàm được gọi cần để trả về hàm gọi. Nó cũng chứa thêm một số thông tin sẽ được thảo luận. Nếu hàm được gọi trả về, thay vì gọi một hàm khác trước khi trở về, mẫu tin kích hoạt cho việc gọi hàm được chuẩn bị, và điều khiển chuyển tới địa chỉ trả về trong mẫu tin kích hoạt được lấy ra.

Thuận tiện của ngăn xếp các lời gọi hàm là mỗi hàm được gọi, luôn có thông tin để trả về ở đỉnh ngăn xếp được gọi. Nếu một hàm gọi một hàm khác, một mẫu tin kích hoạt cho việc gọi hàm mới đơn giản là đẩy vào ngăn xếp gọi hàm. Do đó, địa chỉ trả về yêu cầu bởi hàm được gọi mới trả về lời gọi của nó nằm ở đỉnh ngăn xếp.

Mẫu tin kích hoạt có một trách nhiệm quan trọng khác. Hầu hết các hàm đều có khai báo các tham biến tự động và khai báo hàm cục bộ. Các biến tự động cần phải tồn tại trong khi hàm đang thực thi. Chúng cần duy trì hoạt động khi hàm thực hiện lời gọi đến các hàm khác. Nhưng khi một hàm được gọi trả về lời gọi nó, các biến tự động của hàm được gọi cần phải giải phóng. Mẫu tin kích hoạt của hàm được gọi là một nơi phù hợp để dự trữ bộ nhớ cho các biến tự động của hàm được gọi. Mẫu tin kích hoạt tồn tại đủ lâu theo thời gian các hàm hoạt động. Khi hàm trả về - và không cần các biến tự động cục bộ nữa - mẫu tin kích hoạt được lấy ra từ ngăn xếp, các biến tự động cục bộ không biết đến chương trình. Số lượng bộ nhớ trong một máy tính là hữu hạn, do đó, chỉ có một số lượng bộ nhớ nhất định được sử dụng để lưu trữ mẫu tin kích hoạt trên ngăn xếp lời gọi hàm. Khi nhiều lời gọi xảy ra có thể có mẫu tin kích hoạt của chúng được lưu trữ trên ngăn xếp lời gọi hàm và lỗi tràn ngăn xếp có thể xảy ra.

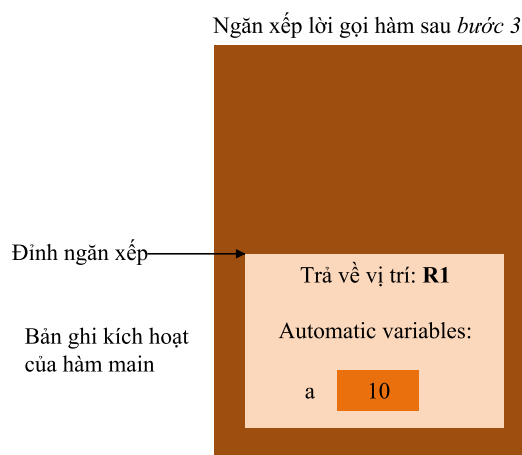
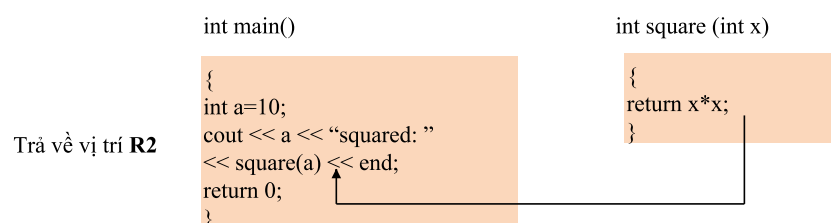
Ngăn xếp lời gọi hàm khi hoạt động

Step 2: main gọi hàm square thực hiện việc tính toán



Hình 4.18: Gọi stack sau khi hàm **main** gọi hàm **square** để thực hiện phép tính.

Step 3: square trả về kết quả của nó trong hàm main



Hình 4.19: Gọi stack sau hàm **square** trả về hàm **main**.

Như đã biết, ngăn xếp lời gọi hàm và mẫu tin kích hoạt hỗ trợ cơ chế gọi hàm/trả về hàm, khởi tạo và hủy các biến tự động. Theo dõi cách thức gọi ngăn xếp hỗ trợ hoạt động của hàm `square` được gọi bởi hàm `main` (dòng 6-11 hình 4.20). Trước tiên, hệ điều hành gọi hàm `main` – việc này đẩy một mẫu tin kích hoạt vào ngăn xếp (trong hình 4.17). Mẫu tin kích hoạt cho hàm `main` biết làm thế nào để trả về hệ điều hành (ví dụ chuyển đến địa chỉ trả về R1) và chứa không gian cho biến tự động của hàm `main` (nghĩa là khởi tạo đến 10). Hàm `main`-trước khi trở lại hệ điều hành - gọi hàm `square` dòng 10 hình 4.20. Điều này dẫn đến mẫu tin kích hoạt cho hàm `square` (dòng 14-17) để đẩy vào ngăn xếp lời gọi hàm (Hình 4.18). Mẫu tin kích hoạt này chứa địa chỉ trả về mà hàm `square` cần trả lại hàm `main` (R2) và bộ nhớ cho biến tự động của `square` (`x`).

Sau khi `square` tính bình phương của các tham số, nó cần trả về hàm `main` - và không cần nhiều bộ nhớ cho các biến tự động `x`. Vì vậy, ngăn xếp được lấy ra – đưa ra bình phương các giá trị trả về trong hàm `main` (R2) và các biến tự động của `square` mất đi. Hình 4.19 trình bày ngăn xếp lời gọi hàm sau khi mẫu tin kích hoạt của hàm `square` được lấy ra.

Hàm `main` lúc này hiển thị kết quả gọi là hàm `square` (dòng 13). Chạm tới dấu ngoặc móc bên phải của hàm `main` dẫn đến mẫu tin kích hoạt được lấy ra từ ngăn xếp và đưa tới địa chỉ trả về của hàm `main` trong hệ điều hành (R1 trong hình 4.20) và dẫn đến bộ nhớ cho biến tự động của hàm `main` (A) trở nên không có sẵn.

```

1 #include <iostream>
2 using namespace std;
3
4 int square( int ); // prototype for function square
5
6 int main()
7 {
8     int a = 10; // value to square (local automatic variable in main)
9
10    cout << a << " squared: " << square(a) << endl; // display a squared
11 } // end main
12
13 // returns the square of an integer
14 int square( int x ) // x is a local variable
15 {
16     return x * x; // calculate square and return result
17 } // end function square

```

Hình 4.20: Hàm `square` được dùng để minh họa gọi hàm `stack` và kích hoạt các mẫu tin

Output chương trình Hình 4.20:

```
10 squared: 100
```

Chúng ta đã thấy cấu trúc dữ liệu ngăn xếp có giá trị trong việc thực hiện một cơ chế then chốt, hỗ trợ thực thi chương trình. Cấu trúc dữ liệu này có nhiều ứng dụng quan trọng trong khoa học máy tính. Thảo luận về ngăn xếp, hàng đợi, danh sách, cây và các cấu trúc dữ liệu khác cũng được đề cập trong Chương 11, Lập trình với thư viện chuẩn (STL).

4.7 Chồng hàm và khuôn mẫu hàm

4.7.1 Chồng hàm (hàm trùng tên)

Hàm trùng tên hay còn gọi là hàm chồng (đề). Đây là một kỹ thuật cho phép sử dụng cùng một tên gọi cho các hàm giống nhau (cùng mục đích) nhưng xử lý trên các kiểu dữ liệu khác nhau hoặc trên số lượng dữ liệu khác nhau. Ví dụ hàm sau tìm số lớn nhất trong 2 số nguyên:

```
int max(int a, int b) { return (a > b) ? a : b ; }
```

Nếu đặt `c = max(3, 5)` ta sẽ có `c = 5`. Tuy nhiên cũng tương tự như vậy nếu đặt `c = max(3.0, 5.0)` chương trình sẽ bị lỗi vì các giá trị (`double` 3.0, 5.0) không phù hợp về kiểu (`int`) của đối trong hàm `max`. Trong trường hợp như vậy, chúng ta phải viết hàm mới để tính `max` của 2 số thực. Mục đích và cách hoạt động của hàm này hoàn toàn giống hàm trước, tuy nhiên trong C và các NNLT cổ điển khác chúng ta buộc phải sử dụng một tên mới cho hàm “mới” này. Ví dụ:

```
double fmax(double a, double b) { return (a > b) ? a : b ; }
// Tương tự để thuận tiện ta sẽ viết thêm các hàm
char cmax(char a, char b) { return (a > b) ? a : b ; }
long lmax(long a, long b) { return (a > b) ? a : b ; }
double dmax(double a, double b) { return (a > b) ? a : b ; }
```

Tóm lại, ta sẽ có 5 hàm: `max`, `cmax`, `fmax`, `lmax`, `dmax` để làm cùng một công việc, việc sử dụng tên như vậy sẽ gây bất lợi khi cần gọi hàm.

Để khắc phục, C++ cho phép ta có thể khai báo và định nghĩa cả 5 hàm trên với cùng 1 tên gọi ví dụ là `max` chẳng hạn. Khi đó, ta có 5 hàm:

```
1: int max(int a, int b) { return (a > b) ? a : b ; }
2: double max(double a, double b) { return (a > b) ? a : b ; }
3: char max(char a, char b) { return (a > b) ? a : b ; }
4: long max(long a, long b) { return (a > b) ? a : b ; }
5: double max(double a, double b) { return (a > b) ? a : b ; }
```

Và lời gọi hàm bất kỳ dạng nào như `max(3, 5)`, `max(3.0, 5)`, `max('0', 'K')` đều được đáp ứng. Chúng ta có thể đặt ra vấn đề: với cả 5 hàm cùng tên như vậy, chương trình sẽ gọi đến hàm nào. Vấn đề được giải quyết dễ dàng vì chương trình sẽ dựa vào kiểu của các đối khi gọi để quyết định chạy hàm nào. Ví dụ lời gọi `max(3, 5)` có 2 đối đều là kiểu nguyên nên chương trình sẽ gọi hàm 1, lời gọi `max(3.0, 5)` hướng đến hàm số 2 và tương tự chương trình sẽ chạy hàm số 3 khi gặp lời gọi `max('0', 'K')`. Như vậy một đặc điểm của hai hàm trùng tên đó là trong danh sách đối của chúng phải có ít nhất một cặp đối nào đó khác kiểu nhau. Một đặc trưng khác cũng để phân biệt hai hàm trùng tên đó là số lượng đối trong các hàm phải khác nhau (nếu kiểu của chúng là giống nhau).

Ví dụ việc vẽ các hình: thẳng, tam giác, vuông, chữ nhật trên màn hình là giống nhau, chúng chỉ phụ thuộc vào số lượng các điểm nối và tọa độ của các điểm. Do vậy, ta có thể khai báo và định nghĩa 4 hàm vẽ nói trên với cùng chung tên gọi. Chẳng hạn:

```
// vẽ đường thẳng AB
void draw(Point_Type A, Point_Type B) ;
// vẽ tam giác ABC
void draw(Point_Type A, Point_Type B, Point_Type C) ;
```

```
// vẽ tứ giác ABCD
void draw(Point_Type A, Point_Type B, Point_Type C, Point_Type D) ;
```

Trong ví dụ trên, ta giả thiết `Point_Type` là một kiểu dữ liệu lưu toạ độ của các điểm trên màn hình. Hàm `draw(Point_Type A, Point_Type B, Point_Type C, Point_Type D)` sẽ vẽ hình vuông, chữ nhật, thoi, bình hành hay hình thang phụ thuộc vào toạ độ của 4 điểm `A, B, C, D`, nói chung nó được sử dụng để vẽ một tứ giác bất kỳ.

Tóm lại, nhiều hàm có thể được định nghĩa chồng (với cùng tên gọi giống nhau) nếu chúng thoả các điều kiện sau:

- Số lượng các tham đối trong hàm là khác nhau.
- Kiểu của tham đối trong hàm là khác nhau.

(Kỹ thuật chồng tên này còn áp dụng cả cho các toán tử (phép toán) được trình bày trong các chương sau).

```
1 #include <iostream>
2 using namespace std;
3
4 // Ham tim UCLN cua 2 doi
5 int GCD(int m, int n)
6 {
7     while (m != n)
8     {
9         if (m > n) m -= n;
10        else n -= m;
11    }
12    return m;
13 }
14
15 // Ham tim UCLN cua 3 doi (trung ten voi ham UCLN 2 doi)
16 int GCD(int m, int n, int k)
17 {
18     return GCD(GCD(m, n), k);
19 }
20
21 int main()
22 {
23     int num1, num2, num3;
24     cout << "Enter three numbers: " ;
25     cin >> num1 >> num2 >> num3 ;
26     cout << "GCD of num1 and num2 is " << GCD(num1, num2) << endl;
27     cout << "GCD of num1 and num3 is " << GCD(num1, num3) << endl;
28     cout << "GCD of three numbers is " << GCD(num1, num2, num3) << endl;
29     return 0;
30 }
```

Hình 4.21: Nạp chồng hàm tính ước số chung lớn nhất.

Chú ý: Cần tránh tình trạng nhập nhằng giữa hàm với đối mặc định và hàm trùng tên. Xét ví dụ sau, cho hàm với đối mặc định :

```
1. int sum(int a, int b, int c = 0, int d = 0);
```

Và hàm trùng tên với hàm trên:

```
2. int sum(int a, int b);
```

- Các hàm được khai báo và cài đặt như trên là hợp lệ (tuy có cùng tên nhưng số lượng đối là khác nhau)
- Nếu gọi `sum(1, 2, 3, 4)` hoặc `sum(1, 2, 3)`, chương trình sẽ gọi hàm 1.
- Tuy nhiên, nếu gọi `sum(4, 7)`, chương trình sẽ không quyết định được nên gọi hàm 1 hay hàm 2, trường hợp này chương trình dịch sẽ báo lỗi.

4.7.2 Khuôn mẫu hàm

Việc cho phép các hàm được trùng tên tạo dễ dàng cho NSD khi gọi đến các hàm này với cùng chỉ một tên gọi. Tuy nhiên, khi lập trình vẫn cần phải viết ra tất cả các hàm như vậy, điều này vẫn dẫn đến lãng phí công sức nếu các hàm có chung cùng cách hoạt động. Ta xét lại ví dụ của mục trước khi cần viết hàm tìm số lớn nhất của hai số, để phủ đầy đủ các loại kiểu ta cần viết rất nhiều hàm có cùng tên **max** như sau:

```
1: int max(int a, int b) { return (a > b) ? a : b ; }
2: double max(double a, double b) { return (a > b) ? a : b ; }
3: char max(char a, char b) { return (a > b) ? a : b ; }
4: long max(long a, long b) { return (a > b) ? a : b ; }
5: double max(double a, double b) { return (a > b) ? a : b ; }
6: .....
```

Rõ ràng các hàm trên có cùng cách viết, cách hoạt động, tên gọi, điểm khác biệt chỉ ở chỗ kiểu đầu vào và đầu ra. Do vậy, có lẽ ta chỉ nên viết một hàm để “dùng chung” cho tất cả ? có nghĩa kiểu của đối và/hoặc kiểu của hàm cũng được xem như một tham đối ? Điều này là thực hiện được bằng kỹ thuật “khuôn mẫu hàm” do C++ cung cấp, tức cho phép viết chung một hàm “mẫu” nhưng dùng được với mọi kiểu dữ liệu khác nhau.

Chìa khóa cho việc dùng chung ở đây là khai báo một kiểu chung với tên gọi bất kỳ nào đó, bằng câu lệnh

```
template <typename identifier>          // identifier: tên kiểu chung
```

hoặc

```
template <class identifier>             // (các từ typename và class là tương đương)
```

và trong hàm chỗ nào liên quan đến kiểu ta sẽ dùng tên chung **identifier** thay cho tên kiểu cụ thể. Trong phần gọi hàm, C++ sẽ tự động thay kiểu cụ thể vào tên kiểu chung này và tiến hành thực hiện hàm một cách bình thường. Dưới đây là ví dụ minh họa cho cách viết hàm mẫu getMax dùng chung cho các hàm tìm max đã liệt kê ở trên.

```
1 #include <iostream>
2 using namespace std;
3
4 // GenType là tên kiểu chung đại diện cho tất cả các kiểu
5 template <typename Gen_Type>
6 Gen_Type getMax (Gen_Type a, Gen_Type b)
7 {
```

```

8     return (a > b ? a : b);
9 }
10
11 int main()
12 {
13     int m, n;
14     double x, y;
15     char c, d;
16     cout << "Enter integers m, n: " ; cin >> m >> n;
17     cout << "Maximun of two integars " << m << " and " << n << " is: " << getMax(m, n
18         ) << endl;
19     cout << "Enter doubles x, y: "; cin >> x >> y;
20     cout << "Maximun of two doubles " << x << " and " << y << " is: " << getMax(x, y)
21         << endl;
22     cout << "Enter characters c, d: "; cin >> c >> d;
23     cout << "Maximun of two characters " << c << " and " << d << " is: " << getMax(c,
24         d) << endl;
25     return 0;
26 }

```

Hình 4.22: Khuôn mẫu hàm `getMax()`.

Để rõ ràng hơn (đối với NSD), trong lời gọi hàm ta có thể viết:

```

getMax <int> (m, n);
getMax <double> (x, y); ...

```

Tuy nhiên, để ngắn gọn ta chỉ cần viết `getMax(m, n)`, `getMax(x, y)` ... như trong ví dụ, C++ sẽ tự động nhận biết kiểu của các tham đối để thực hiện.

Ngoài việc các tham đối có chung một kiểu mẫu như trong ví dụ trên, ta cũng có thể viết hàm với các tham đối khác kiểu nhau (ví dụ tìm `max` của một số thực và một số nguyên), ta cần khai báo thêm kiểu trong khai báo `template` tương ứng với từng tham đối.

Ví dụ sau minh họa hàm tìm giá trị lớn nhất của 2 giá trị khác kiểu nhau (chú ý: các giá trị này phải được phép đổi kiểu tự động khi cần thiết) và trả lại giá trị theo kiểu thứ nhất.

```

1 #include <iostream>
2 using namespace std;
3
4 template <typename T, typename U> // 2 kiểu mẫu cho 2 đối khác kiểu nhau
5 T GetMax (T a, U b) // giá trị trả lại là kiểu của đối thu 1 (T)
6 {
7     T res;
8     res = (a > b ? a : b);
9     return res;
10 }
11
12 int main()
13 {
14     double double_num;
15     int integer_num;
16     char character;
17     cout << "Enter the double, the integer, the character: " ; cin >> double_num >>
18         integer_num >> character;

```

```

18     cout << "Maximum of " << double_num << " and " << integer_num << " is: " <<
        GetMax(double_num, integer_num) << endl;    // thực - nguyên, trả lại thực
19     cout << "Maximum of " << integer_num << " and " << character << " is: " << GetMax
        (integer_num, character) << endl;    // nguyên - kí tu, trả lại nguyên
20     cout << "Maximum of " << integer_num << " and " << character << " is: " << GetMax
        (character, integer_num) << endl;    // nguyên - kí tu, trả lại kí tu
21     return 0;
22 }

```

Hình 4.23: Khuôn mẫu hàm `getMax()` với 2 kiểu đối số khác nhau.

Trong ví dụ trên, giả sử ta nhập 3 giá trị tương ứng với `double_num`, `integer_num`, `character` là 64.5, 65, 'B'. Khi đó, lời gọi `getMax(double_num, integer_num)` là được phép khi so sánh giá trị thực (64.5) và nguyên (65) và giá trị trả lại là số thực (65.0) (C/C++ tự động đổi giá trị nguyên sang thực). Tương tự, lời gọi `getMax(integer_num, character)` trả lại số nguyên 66, còn lời gọi `getMax(character, integer_num)` trả lại kí tự 'B'. Tuy nhiên, lời gọi `getMax(integer_num, double_num)` là không được phép vì giá trị trả lại là kiểu nguyên mà kiểu thực không thể chuyển kiểu tự động về được kiểu nguyên.

4.8 Lập trình với hàm đệ quy

4.8.1 Khái niệm đệ quy

Một hàm gọi đến hàm khác là bình thường, nhưng nếu hàm lại gọi đến chính nó thì ta gọi hàm là đệ quy. Như vậy, khi một hàm nào đó gọi đến hàm đệ quy thì hàm đệ quy này sẽ chạy nhiều lần, có vẻ như sẽ chạy đến vô hạn lần, tuy nhiên giống như không tồn tại động cơ vĩnh cửu, hàm đệ quy cũng chỉ chạy đến khi “hết nhiên liệu” sẽ dừng.

Để minh họa, ta hãy xét hàm tính n giai thừa. Để tính $n!$ ta có thể xây dựng hàm `factorial()` dùng phương pháp lặp như sau:

```

1 #include <iostream>
2 using namespace std;
3
4 long Factorial(int n)
5 {
6     long res;
7     res = 1;
8     for (int count = 1; count <= n; count++)
9         res *= count;
10    return res;
11 }
12
13 int main()
14 {
15     int n;
16     cout << "n = " ; cin >> n;    // nhập số cần tính giai thừa
17     cout << n << "! = " << Factorial(n) << endl;
18     return 0;
19 }

```

Hình 4.24: Tính giai thừa.

Mặt khác, $n!$ cũng được tính thông qua $(n-1)!$ bởi công thức truy hồi

$$n! = \begin{cases} 1 & \text{nếu } n = 0 \\ (n-1)! \times n & \text{nếu } n > 0 \end{cases}.$$

Để tính giai thừa (của n) ta lại thông qua tính giai thừa của số nhỏ hơn ($n-1$), do đó ta có thể xây dựng hàm `factorial()` (đệ qui) tính $n!$ như sau:

```

1 #include <iostream>
2 using namespace std;
3
4 long Factorial2(int n)
5 {
6     if (n == 0) return 1;
7     else return Factorial2(n - 1) * n;
8 }
9
10 int main()
11 {
12     int n;
13     cout << "n = " ; cin >> n;           // nhập số cần tính giai thừa
14     cout << n << "! = " << Factorial2(n) << endl;
15     return 0;
16 }
```

Hình 4.25: Tính giai thừa bằng hàm đệ qui.

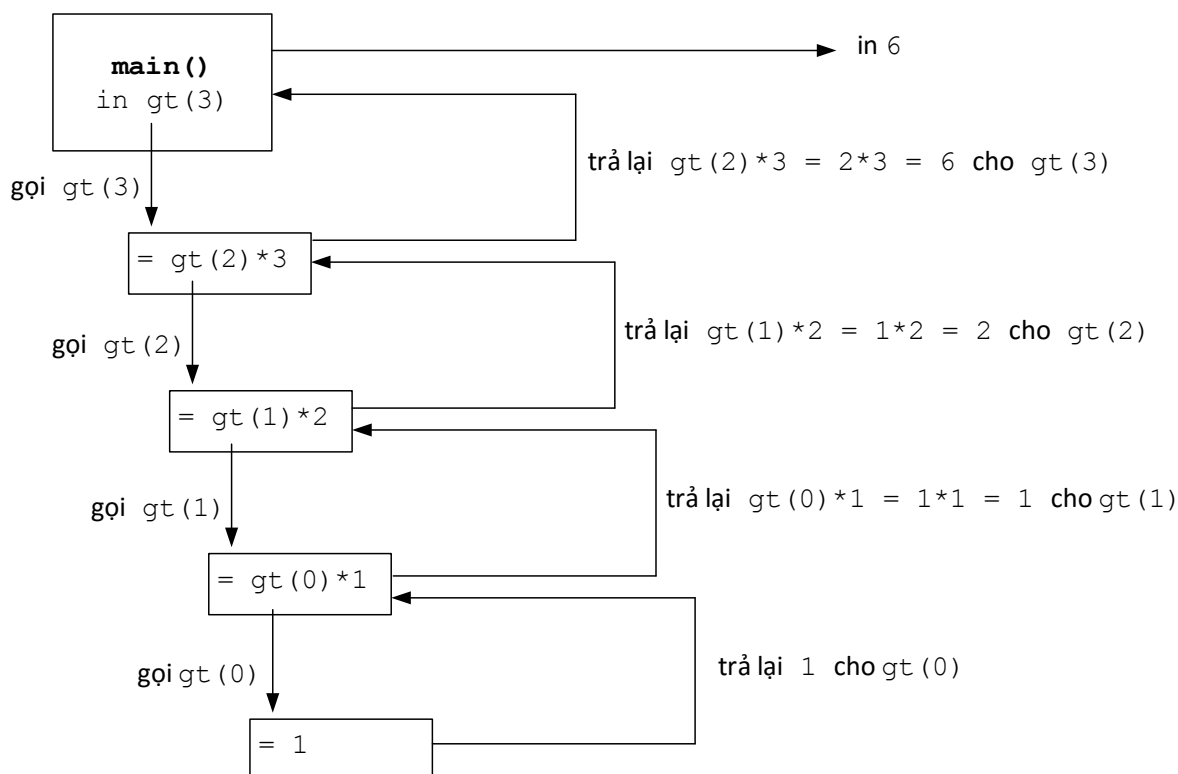
Hàm `factorial(n)` được cài đặt như trên là một hàm đệ qui vì hàm có gọi đến chính nó, cụ thể để tính `factorial(n)`, hàm đã gọi đến `factorial(n-1)` với điểm khác biệt là đối của lần gọi sau là $n-1$, bé hơn đối của lần gọi trước là n . Hiển nhiên, bằng cách này, để giải quyết `factorial(n-1)` hàm sẽ lại phải gọi đến `factorial(n-2)`, ... quá trình gọi đến chính nó cứ tiếp tục với đối của hàm mỗi lúc mỗi bé dần đến khi đối n giảm đến 0. Do câu lệnh `if (n == 0) return 1;` hàm sẽ trả lại giá trị 1 cho `factorial(0)` thay vì tiếp tục gọi đến chính nó. Từ đây, quá trình tính toán ngược được bắt đầu với `factorial(1) = factorial(0) * 1 = 1`, `factorial(2) = factorial(1) * 2 = 1 * 2 = 2`, ... cho đến `factorial(n)` được tính cuối cùng bằng `1.2.3. ... n`, tức `factorial(n)` bằng $n!$.

Ví dụ, trong hàm `main()` giả sử ta nhập 3 cho n , khi đó để thực hiện câu lệnh `cout << factorial(n)` để in $3!$, đầu tiên chương trình sẽ gọi chạy hàm `factorial(3)`. Do $3 > 0$ nên hàm `factorial(3)` sẽ trả lại giá trị `factorial(2) * 3`, tức lại gọi hàm `factorial` với tham đối thực sự ở bước này là $n = 2$. Tương tự, `factorial(2) = factorial(1) * 2` và `factorial(1) = factorial(0) * 1`. Khi thực hiện `factorial(0)` ta có đối $n = 0$ nên hàm trả lại giá trị 1, từ đó `factorial(1) = 1 * 1 = 1` và suy ngược trở lại ta có `factorial(2) = factorial(1) * 2 = 1 * 2 = 2`, `factorial(3) = factorial(2) * 3 = 2 * 3 = 6`. Chương trình in ra kết quả 6.

Có thể biểu thị cách hoạt động như trên bằng sơ đồ sau (để gọn, trong sơ đồ ta dùng tên gọi `gt` để thay cho `factorial`) :

So sánh hai cách viết về hàm `factorial`, ta thấy hàm đệ qui có đặc điểm:

- Hàm được viết rất gọn.



Hình 4.26: Quá trình gọi đệ quy của hàm **factorial**.

- Việc thực hiện gọi hàm nhiều lần gây mất thời gian.
- Mỗi lần gọi chương trình sẽ tạo nên một tập biến cục bộ mới trên ngăn xếp (các đối, các biến riêng khai báo trong hàm, địa chỉ của câu lệnh tiếp theo sau khi chạy xong hàm ...) độc lập với lần chạy trước, từ đó dễ gây tràn ngăn xếp (quá tải bộ nhớ).

Mặc dù chiếm nhiều thời gian (chạy chương trình) và không gian (bộ nhớ máy tính), đệ qui là cách viết rất gọn, dễ viết và đọc chương trình, mặt khác có nhiều bài toán hầu như tìm một thuật toán lặp cho nó là rất phức tạp trong khi viết theo thuật toán đệ qui thì lại rất dễ dàng.

Chú ý: Để dùng phương pháp đệ qui, ta phải viết riêng hàm cho mục đích này. Ví dụ để tính giai thừa bằng phương pháp lặp ta chỉ cần viết cả trong hàm **main()**. Để viết theo đệ qui, ta phải viết hàm **factorial** riêng, độc lập với hàm **main()**.

4.8.2 Lớp các bài toán giải được bằng đệ qui

Phương pháp đệ qui thường được dùng để giải các bài toán có đặc điểm:

- Giải quyết được dễ dàng trong các trường hợp riêng gọi là trường hợp suy biến hay cơ sở, trong trường hợp này hàm được tính bình thường mà không cần gọi lại chính nó (ví dụ trường hợp $n = 0$ trong bài toán tính $n!$).
- Đối với trường hợp tổng quát, bài toán có thể giải được bằng bài toán cùng dạng nhưng với tham số khác có kích thước nhỏ hơn tham số ban đầu. Và sau một số bước hữu hạn biến đổi cùng dạng, bài toán đưa được về trường hợp suy biến (ví dụ trường hợp $n > 0$ trong bài toán tính $n!$).

Trong trường hợp tính $n!$ nếu $n = 0$ hàm cho ngay giá trị 1 mà không cần phải gọi lại chính nó, đây chính là trường hợp suy biến. Trường hợp $n > 0$, hàm sẽ gọi lại chính nó nhưng với n giảm 1 đơn vị. Việc gọi này được lặp lại cho đến khi $n = 0$.

Một lớp rất rộng của bài toán dạng này là các bài toán có thể định nghĩa được dưới dạng đệ qui (còn gọi là truy hồi) như các bài toán lặp với số bước hữu hạn biết trước hay các bài toán UCLN, tháp Hà Nội, ... Việc đưa ra được định nghĩa của bài toán dưới dạng đệ qui sẽ làm dễ dàng cho công tác thiết kế thuật toán và từ đó viết hàm đệ qui cho bài toán.

Ví dụ: Gọi $S(n)$ là tổng của n số tự nhiên đầu tiên, tức $S(n) = 1 + 2 + 3 + \dots + n$. Tương tự bài toán tính tích n số tự nhiên đầu tiên (tức $n!$), $S(n)$ cũng được định nghĩa dưới dạng đệ qui như sau:

$$S(n) = \begin{cases} 1 & \text{nếu } n = 1 \text{ (trường hợp cơ bản hoặc suy biến)} \\ S(n-1) + n & \text{nếu } n > 1 \text{ (trường hợp tổng quát)} \end{cases}.$$

Từ đó, ta có chương trình:

```
1 #include <iostream>
2 using namespace std;
3
4 int Sum(int n)
5 {
6     if (n == 1) return 1;
7     else return Sum(n - 1) + n;
8 }
9
10 int main()
11 {
12     int n;
13     cout << "Enter n = " ; cin >> n;           // nhập số cần tính tổng
14     cout << "Sum of first n integer numbers = " << Sum(n) << endl;
15     return 0;
16 }
```

Hình 4.27: Tính giai thừa bằng hàm đệ qui.

4.8.3 Cấu trúc chung của hàm đệ qui

Dạng thức chung của một hàm đệ qui thường như sau:

```
if (tham đối suy biến)
    trình bày cách giải trực tiếp           // giả định đã có cách giải
else
    // tham đối chưa suy biến
    gọi lại hàm với tham đối "bé" hơn
```

Một số ví dụ

1. Tính $S(n) = \sqrt{3 + \sqrt{3 + \sqrt{3 + \dots + \sqrt{3}}}}$ với n dấu căn.

Có thể tính $S(n)$ bằng vòng lặp như các chương trước đã đề cập. Tuy nhiên, cũng có thể tính $S(n)$ bằng hàm đệ qui. Để làm được điều này, ta cần định nghĩa lại $S(n)$ dưới dạng đệ qui như

sau:

$$S(n) = \begin{cases} \sqrt{3} & \text{nếu } n = 1 \\ \sqrt{3 + S(n-1)} & \text{nếu } n > 1 \end{cases}.$$

Từ đó, ta có chương trình đệ qui:

```

1 #include <iostream>
2 #include <math.h>
3 using namespace std;
4
5 double S(int numSquare_roots)
6 {
7     double res;
8     if (numSquare_roots == 1) res = sqrt(3);
9     else res = sqrt(3 + S(numSquare_roots - 1));
10    return res;
11 }
12
13 int main()
14 {
15     int numSquare_roots;
16     cout << "Number of the Square roots = " ; cin >> numSquare_roots;
17     cout << "Result = " << S(numSquare_roots) << endl;
18     return 0;
19 }
```

Hình 4.28: Tính $S(n)$ với các dấu căn lồng nhau.

2. Tính $S(n) = \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots + \frac{1}{2}}}}$ với n dấu chia

Dạng định nghĩa đệ qui của $S(n)$ như sau:

$$S(n) = \begin{cases} 1/2 & \text{nếu } n = 1 \\ 1/(2 + S(n-1)) & \text{nếu } n > 1 \end{cases}.$$

Từ đó ta có chương trình đệ qui:

```

1 #include <iostream>
2 #include <math.h>
3 using namespace std;
4
5 double S(int numDivisions)
6 {
7     if (numDivisions == 1) return 1/2.0;           // chú ý 2 được viết 2.0
8     else return 1/(2 + S(numDivisions - 1));
9 }
10
11 int main()
12 {
13     int numDivisions;
14     cout << "Number of the Divisions = " ; cin >> numDivisions;
15     cout << "S = " << S(numDivisions) << endl;
16     return 0;
17 }
```

```
17 }
```

Hình 4.29: Tính $S(n)$ với n chẵn.

3. Dãy Fibonacci là dãy $f(n)$ được định nghĩa : hai số đầu tiên có giá trị $f(1) = f(2) = 1$, các số còn lại bằng tổng giá trị của 2 số kề trước nó, tức $f(n) = f(n-1) + f(n-2) (n > 2)$. Định nghĩa đệ quy của dãy số được viết:

$$f(n) = \begin{cases} 1 & \text{nếu } n = 1, 2 \\ f(n-1) + f(n-2) & \text{nếu } n > 2 \end{cases}$$

Từ đó, ta có chương trình đệ quy:

```
1 #include <iostream>
2 using namespace std;
3
4 long Fib(int n)
5 {
6     long res;
7     if (n==1 || n==2) res = 1;
8     else res = fib(n-1) + fib(n-2);
9     return res;
10 }
11
12 int main()
13 {
14     int n;
15     cout << "n = " ; cin >> n;
16     cout << "The first " << n << " fibonacci numbers\n";
17     for (int count = 1; count <= n; count++)
18         cout << Fib(count) << endl;
19     return 0;
20 }
```

Hình 4.30: Tính số Fibonacci.

4. Tìm UCLN của 2 số a, b . Có thể định nghĩa hàm `GCD(int m, int n)` dưới dạng đệ quy như sau:

- Nếu $a = b$ thì $\text{GCD}(a, b) = a$
- Nếu $a > b$ thì $\text{GCD}(a, b) = \text{GCD}(a-b, b)$
- Nếu $a < b$ thì $\text{GCD}(a, b) = \text{GCD}(a, b-a)$

Từ đó ta có chương trình đệ quy để tính UCLN của a và b :

```
1 #include <iostream>
2 using namespace std;
3
4 int GCD(int m, int n)
5 {
6     if (m == n) return m;
7     if (m > n) return GCD(m - n, n);
```

```

8     if (m < n) return GCD(m, n - m);
9 }
10
11 int main()
12 {
13     int m, n;
14     cout << "m = " ; cin >> m;
15     cout << "n = " ; cin >> n;
16     cout << "Greatest common divisor of " << m << " and " << n << " is " << GCD(
        m,n) << endl;
17     return 0;
18 }

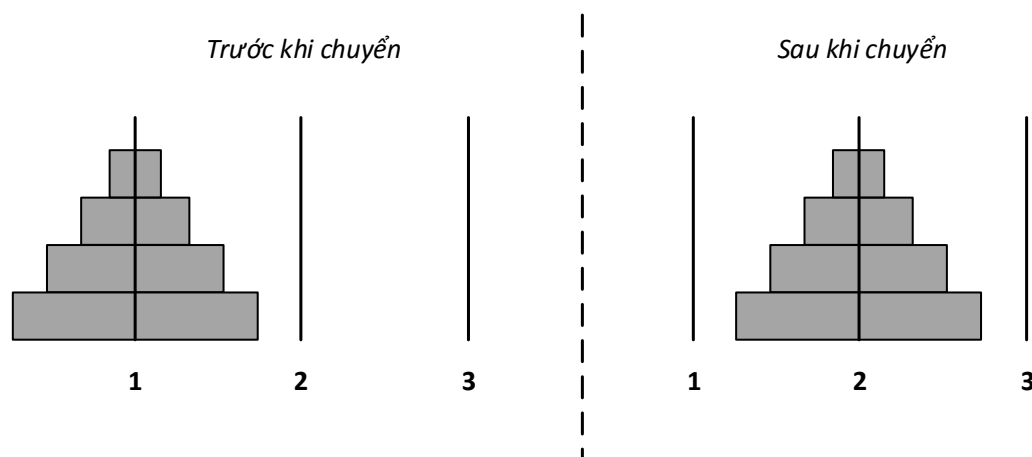
```

Hình 4.31: Tính UCLN bằng hàm đệ qui.

5. Chuyển tháp là bài toán cổ nổi tiếng, nội dung như sau: Cho một tháp n tầng, đang xếp tại vị trí 1. Yêu cầu bài toán là hãy chuyển toàn bộ tháp sang vị trí 2 (cho phép sử dụng vị trí trung gian 3) theo các điều kiện sau đây:

- Mỗi lần chỉ được chuyển một tầng trên cùng của tháp,
- Tại bất kỳ thời điểm, cả 3 vị trí, các tầng tháp lớn hơn phải nằm dưới các tầng tháp nhỏ hơn.

Bài toán chuyển tháp với 4 tầng được minh họa như trong hình vẽ bên dưới.



Hình 4.32: Bài toán Tháp Hà Nội.

Có thể tổng quát hóa bài toán như sau: chuyển n tầng tháp từ vị trí **source** đến vị trí **target**, trong đó **source**, **target** là các tham số có thể lấy giá trị là 1, 2, 3 thể hiện cho 3 vị trí. Đối với 2 vị trí **source** và **target**, dễ thấy vị trí trung gian **temp** (vị trí còn lại) sẽ là vị trí $6 - \text{source} - \text{target}$ (vì $\text{source} + \text{target} + \text{temp} = 1 + 2 + 3 = 6$). Từ đó để chuyển tháp từ vị trí **source** đến vị trí **target**, ta có thể xây dựng một cách chuyển đệ qui như sau:

- Chuyển $n-1$ tầng từ **source** sang **temp**,
- Chuyển 1 tầng còn lại từ **source** sang **target**,

- Chuyển trả **n-1** tầng từ **temp** về lại vị trí **target**

Hiển nhiên nếu số tầng là 1 thì ta chỉ phải thực hiện một phép chuyển từ **source** sang **target**.

Mỗi lần chuyển 1 tầng từ vị trí **i** đến **j** ta in **i -> j**. Chương trình sẽ nhập vào **input** là số tầng và in ra các bước chuyển theo kí hiệu trên. Như vậy, kết quả của chương trình là một dãy các bước chuyển dạng **i -> j**.

```

1 #include <iostream>
2 using namespace std;
3
4 void Transpose(int source, int target)
5 {
6     cout << source << " -> " << target << endl;
7     return;
8 }
9
10 void TransposeTower(int numfloors, int source, int target)
11 {
12     int temp;
13     if (numfloors == 1) Transpose(source, target);
14     else
15     {
16         temp = 6 - source - target;
17         TransposeTower(numfloors-1, source, temp);
18         Transpose(source, target);
19         TransposeTower(numfloors-1, temp, target);
20     }
21     return;
22 }
23
24 int main()
25 {
26     int num_floors;
27     cout << "Enter the number of floors: " ; cin >> num_floors;
28     TransposeTower(num_floors, 1, 2);
29     return 0;
30 }

```

Hình 4.33: Bài toán chuyển tháp.

Ví dụ nếu số tầng bằng 3 thì chương trình in ra kết quả là dãy các phép chuyển sau đây:

1 -> 2 , 1 -> 3 , 2 -> 3 , 1 -> 2 , 3 -> 1 , 3 -> 2 , 1 -> 2.

Ta có thể tính được số lần chuyển là $2^n - 1$ với n là số tầng.

Bài tập

1. Chọn câu sai trong các câu sau đây:

- (a) Hàm không trả lại giá trị thì không cần khai báo kiểu giá trị của hàm.
- (b) Các biến được khai báo trong hàm là cục bộ, tự xoá khi hàm thực hiện xong.
- (c) Hàm là đơn vị độc lập, không được khai báo hàm lồng nhau.
- (d) Được phép gọi hàm trong thân một hàm khác.

2. Chọn câu đúng nhất trong các câu sau đây:

- (a) Hàm phải được kết thúc với 1 câu lệnh `return`.
- (b) Phải có ít nhất 1 câu lệnh `return` cho hàm.
- (c) Các câu lệnh `return` được phép nằm ở vị trí bất kỳ trong thân hàm.
- (d) Không cần khai báo kiểu giá trị trả lại của hàm nếu hàm không có lệnh `return`.

3. Chọn câu sai trong các câu sau đây:

- (a) Các biến khai báo trong thân hàm không được phép trùng với tên đối.
- (b) Các biến cục bộ trong thân hàm được chương trình dịch cấp phát bộ nhớ.
- (c) Tất cả tham đối hình thức sẽ được cấp phát bộ nhớ tạm thời khi hàm được gọi.
- (d) Kiểu của tham đối thực sự phải giống (hoặc C++ có thể tự động chuyển kiểu được về) kiểu của tham đối hình thức tương ứng với nó trong lời gọi hàm.

4. Chọn câu sai trong các câu sau đây:

- (a) Số lượng các đối mặc định là tùy ý.
- (b) Được phép khai báo các tham đối mặc định ở bất kỳ vị trí nào trong danh sách đối.
- (c) Vị trí các đối mặc định phải liên tiếp nhau và ở cuối danh sách đối.
- (d) Với hàm không có biến mặc định thì số tham đối thực sự phải bằng số tham số hình thức trong lời gọi hàm.

5. Hàm dưới đây có 2 biến vào đại diện cho số byte và số bit (1 byte bằng 8 bit), hàm trả lại tổng số bit.

```
int total_bit(int bytes, int bits)
{
    int bits;
    bits = 8*bytes + bits;
    return bits;
}
```

Hãy cho biết hàm trên viết đúng hay sai. Vì sao ?

6. Viết hàm trả lại giá trị là nhiệt độ Celcius tính theo nhiệt độ Fareinheit ($C = (5/9)(F-32)$). Áp dụng hàm này in bảng gồm 2 cột nhiệt độ Fareinheit và Celcius tương ứng (nhiệt độ Fareinheit đi từ 32 độ đến 212 độ, mỗi dòng cách nhau 10 độ).

7. Viết 2 hàm tính diện tích hình vuông và diện tích hình tròn. Áp dụng các hàm này tính phần diện tích giới hạn bởi hình tròn bán kính r và hình vuông ngoại tiếp của nó.
8. Viết hàm gồm 3 đối x, y, z . Hàm trả lại 1 nếu $x \leq y \leq z$ và 0 nếu ngược lại. Áp dụng: Nhập 3 số bất kỳ từ bàn phím, in ra dãy số này theo thứ tự tăng dần.
9. Viết hàm in n dấu sao, bắt đầu tại cột thứ k . Áp dụng: in tam giác cân (gồm các dấu sao) có độ dài cạnh đáy nhập từ bàn phím.
10. Viết hàm tìm UCLN của 2 số. Áp dụng: tìm BCNN (bội chung nhỏ nhất) của 2 số. ($\text{BCNN}(m, n) = m * n / \text{UCLN}(m, n)$).
11. Viết hàm kiểm tra một số nguyên n có là số nguyên tố. Áp dụng:
 - (a) In ra 100 số nguyên tố đầu tiên.
 - (b) In ra các số nguyên tố bé hơn 1000.
 - (c) In các cặp số sinh đôi bé hơn 1000. (Các số “sinh đôi” là các số nguyên tố mà khoảng cách giữa chúng là 2).
12. Số hoàn chỉnh n là số bằng tổng mọi ước (trừ n) của nó (Ví dụ $6 = 1 + 2 + 3$). Hãy in ra mọi số hoàn chỉnh từ 1 đến 1000.
13. Nhập số tự nhiên chẵn $n > 2$. Hãy kiểm tra số này có thể biểu diễn được dưới dạng tổng của 2 số nguyên tố hay không ?.
14. Viết hàm tính tổng của 4 số nguyên dương. Nhập vào 4 số nguyên dương **a, b, c, d**. Sử dụng hàm để in kết quả của **a + b**, **a + b + c** và **a + b + c + d**.
15. Viết hàm input cho phép nhập giá trị cho 3 biến nguyên bất kỳ và hàm **sum()** trả lại tổng của 3 số nguyên. Viết chương trình sử dụng các hàm trên để nhập vào 3 số nguyên và in ra tổng của chúng.
16. Viết chương trình liệt kê nghiệm của họ phương trình: $ax^2 + bx + 1 = 0$, trong đó hệ số a lấy trên tập các số $\{1, -1, -2\}$ và b tương tự với $\{-2, 2, 1\}$.
 Hướng dẫn: Viết hàm giải phương trình bậc 2 với giá trị trả lại là số nghiệm của phương trình. Hàm có 5 đối **a, b, c** và **x1, x2**, trong đó **x1, x2** là đối tham chiếu dùng để lưu 2 nghiệm của phương trình. Khai báo hai mảng hằng **A[3]** và **B[3]** để lưu các hệ số đã cho trong đầu bài.
17. Nhập số nguyên dương N . Viết hàm đệ quy tính:
 - (a) $S_1 = \frac{1+2+\dots+N}{N}$
 - (b) $S_2 = \sqrt{1^2 + 2^2 + \dots + N^2}$
18. Viết hàm đệ quy tính $n!$. Áp dụng chương trình con này tính tổ hợp chập k theo công thức: $C(n, k) = n! / (k!(n - k)!)$.
19. Viết hàm đệ quy tính số Fibonacci thứ n . Dùng chương trình con này tính **f(2) + f(4) + f(6) + f(8) + f(10)**.

20. Thuật toán Euclide tìm UCLN của 2 số có thể phát biểu lại như sau: nếu số nhỏ hơn (hoặc bằng) là ước của số kia thì đó là UCLN của 2 số. Ngược lại, UCLN của 2 số là UCLN của số lớn và phần dư của số lớn với số còn lại. Hãy viết công thức (thuật toán) truy hồi để tìm UCLN của a, b . Từ đó viết hàm đệ qui để thực hiện thuật toán này.

Chương 5

Mảng

5.1 Lập trình và thao tác với mảng một chiều

5.1.1 Ý nghĩa của mảng

Khi cần lưu trữ một dãy n phần tử dữ liệu, chúng ta cần khai báo n biến tương ứng với n tên gọi khác nhau. Điều này sẽ rất khó khăn cho người lập trình để có thể nhớ và quản lý được hết tất cả các biến, đặc biệt khi n lớn. Trong thực tế, hiển nhiên chúng ta gặp rất nhiều dữ liệu có liên quan đến nhau về một mặt nào đó, ví dụ chúng có cùng kiểu và cùng thể hiện một đối tượng: như các tọa độ của một vectơ, các số hạng của một ma trận, các sinh viên của một lớp hoặc các dòng kí tự của một văn bản ... Lợi dụng đặc điểm này, toàn bộ dữ liệu (cùng kiểu và cùng mô tả một đối tượng) có thể chỉ cần chung một tên gọi để phân biệt với các đối tượng khác, và để phân biệt các dữ liệu trong cùng đối tượng ta sử dụng cách đánh số thứ tự cho chúng, từ đó việc quản lý biến sẽ dễ dàng hơn, chương trình sẽ gọn và có tính hệ thống hơn.

Giả sử ta có 2 vectơ trong không gian ba chiều, mỗi vectơ cần 3 biến để lưu 3 tọa độ, vì vậy để lưu tọa độ của 2 vectơ chúng ta phải dùng đến 6 biến, ví dụ x_1 , y_1 , z_1 cho vectơ thứ nhất và x_2 , y_2 , z_2 cho vectơ thứ hai. Một kiểu dữ liệu mới được gọi là mảng một chiều cho phép ta chỉ cần khai báo 2 biến v_1 và v_2 để chỉ 2 vectơ, trong đó mỗi v_1 hoặc v_2 sẽ chứa 3 dữ liệu được đánh số thứ tự từ 0 đến 2, trong đó ta có thể ngầm định thành phần 0 biểu diễn tọa độ x , thành phần 1 biểu diễn tọa độ y và thành phần có số thứ tự 2 sẽ biểu diễn tọa độ z .

Tóm lại, mảng là một dãy các thành phần có cùng kiểu được sắp xếp liên tiếp trong bộ nhớ. Tất cả các thành phần đều có cùng tên là tên của mảng. Để phân biệt các thành phần với nhau, các thành phần sẽ được đánh số thứ tự (còn gọi là chỉ số) từ 0 cho đến hết mảng. Như vậy, nếu mảng có n thành phần thì thành phần cuối cùng trong mảng sẽ được đánh số là $n-1$. Khi cần nói đến thành phần cụ thể nào của mảng ta sẽ dùng tên mảng và kèm theo chỉ số của thành phần đó nằm trong cặp dấu [].

Dưới đây là hình ảnh của một mảng có tên `score` gồm 5 số nguyên (mỗi số nguyên chiếm 2 bytes trong bộ nhớ) toàn bộ mảng chiếm 10 bytes liên nhau, giả sử từ byte có địa chỉ 200, khi đó byte cuối cùng mảng `score` chiếm dụng là 209. Các thành phần được đánh số từ 0 đến 4.

Bảng 5.1: Phân bố bộ nhớ của mảng một chiều

Địa chỉ	200	201	202	203	204	205	206	207	208	209	210	211	...
score	score[0]	score[1]	score[2]	score[3]	score[4]								
Chỉ số	0	1	2	3	4								

5.1.2 Thao tác với mảng một chiều

Khai báo

Để khai báo mảng một chiều ta có thể dùng cú pháp sau:

```
Type_Name array_name[SIZE];
```

Trong đó, **SIZE** là một hằng số biểu thị kích thước tức số thành phần (hoặc số phần tử) của mảng và **Type_Name** là kiểu của các thành phần.

Ví dụ, điểm **x** với tọa độ nguyên trong không gian 3 chiều có thể khai báo như một mảng gồm 3 thành phần:

```
int x[3];
```

Trong đó, các thành phần của **x** lần lượt là **x[0]**, **x[1]**, **x[2]**.

Tương tự, điểm số của 5 sinh viên có thể được khai báo như mảng 5 thành phần:

```
int score[5];
```

Với kích thước của mảng ta thường dùng tên hằng, ví dụ:

```
const int NUMBER_OF_STUDENTS = 5;
int score[NUMBER_OF_STUDENTS];
```

Khai báo này cũng tương đương với `int score[5]`, tuy nhiên, sử dụng tên hằng có nhiều thuận lợi về sau, ví dụ khi số sinh viên thay đổi ta chỉ cần điều chỉnh chỉ một dòng định nghĩa hằng **NUMBER_OF_STUDENTS** thay vì phải thay đổi nhiều nơi trong chương trình có liên quan đến số sinh viên.

Chú ý kích thước của mảng phải được biết trước khi chạy chương trình, vì vậy nó không thể là biến hoặc biểu thức liên quan đến biến. Ví dụ:

```
int size;
cout << "Enter size of array: ";
cin >> size;
int score[size];           // sai
```

Sử dụng mảng

- Truy cập thành phần mảng

Để truy cập đến thành phần của mảng, ta sử dụng tên mảng và chỉ số đặt trong cặp dấu ngoặc []. Ví dụ, **score[0]** để chỉ điểm của sinh viên thứ 0, **score[i]** là điểm của sinh viên thứ **i** (hiển nhiên **i** là một biến hoặc hằng và đã có giá trị cụ thể). Tổng quát, một chỉ số có thể là một biểu thức nguyên và có giá trị nằm trong khoảng từ 0 đến **SIZE-1**. Ví dụ:

```
i = 2;
cout << score[i];           // in điểm của sinh viên thứ 2
score[i + 1] = 8;          // gán điểm 8 cho sinh viên thứ 3
```

Mặc dù mảng biểu diễn một đối tượng nhưng chúng ta không thể áp dụng các thao tác lên toàn bộ mảng mà phải thực hiện thao tác thông qua từng thành phần của mảng. Ví dụ, chúng ta không thể nhập hoặc in dữ liệu cho cả mảng `score[5]` bằng câu lệnh:

```
cin >> score;           // sai
cout << score;          // sai
```

mà phải nhập, in cho từng phần tử từ `score[0]` đến `score[4]`. Dĩ nhiên trong trường hợp này, chúng ta phải cần đến lệnh lặp `for`:

```
int index ;
for (index = 0 ; index < NUMBER_OF_STUDENTS ; index++)
    cin >> score[index] ;
```

Tương tự, giả sử mỗi phân số (gồm tử và mẫu) được khai báo như mảng 2 thành phần, chúng ta cần cộng 2 phân số `a`, `b` và đặt kết quả vào `c`. Không thể viết:

```
c = a + b ;           // sai
```

mà cần phải tính từng phần tử của `c`:

```
c[0] = a[0] * b[1] + a[1] * b[0] ;           // tử số
c[1] = a[1] * b[1] ;                         // mẫu số
```

• Truy cập ra ngoài miền chỉ số

Một điểm cần lưu ý là truy nhập ra ngoài mảng tức thành phần nằm ngoài vùng chỉ số. Trường hợp này, chương trình vẫn chạy nhưng cho kết quả sai. Ví dụ với mảng `int score[5]`, giả sử byte đầu tiên của mảng đặt tại địa chỉ 200 (khi đó giá trị của phần tử cuối cùng `score[4]` đặt tại byte 208), nếu ta dùng câu lệnh:

```
int i = 3;
i = i + 2;
cout << score[i];
```

Chương trình sẽ tính ra `i = 5` và tìm đến địa chỉ 2 bytes tiếp sau `score[4]` là 210 và 211 để lấy giá trị in ra màn hình. Đây là một giá trị ngẫu nhiên nào đó có sẵn (thường gọi là rác) chứ không phải của `score`. Do vậy, chương trình vẫn chạy nhưng cho kết quả sai. Đây là lỗi nếu không chú ý từ đầu sẽ khó phát hiện trong quá trình gỡ lỗi.

Một số ví dụ

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int a[2], b[2], sum[2], pro[2] ;
8     cout << "Enter numerator of fraction a: " ; cin >> a[0] ;
9     cout << "Enter denominator of fraction a: " ; cin >> a[1] ;
10    cout << "Enter numerator of fraction b: " ; cin >> b[0] ;
11    cout << "Enter denominator of fraction b: " ; cin >> b[1] ;
12
13    sum[0] = a[0]*b[1] + a[1]*b[0] ;
```

```

14     sum[1] = a[1] * b[1] ;
15     pro[0] = a[0]*b[0];
16     pro[1] = a[1] * b[1] ;
17     cout << "Sum of two fractions = " << sum[0] << '/' << sum[1] << endl;
18     cout << "Product of two fractions = " << pro[0] << '/' << pro[1] << endl;
19
20     return 0;
21 }

```

Hình 5.2: Tìm tổng, tích 2 phân số.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      const int MAX = 100;
8      double SEQ[MAX];
9      int count, numElement;
10     int numPositive, numNegative, numZero;
11     cout << "Enter the number of elements of sequence: " ; cin >> numElement;
12     cout << "Enter elements of sequence: " ;
13     for (count = 0; count < numElement; count++)
14     {
15         cout << "SEQ[" << count << "] = " ;
16         cin >> SEQ[count];
17     }
18
19     numPositive = numNegative = numZero = 0;
20     for (count = 0; count < numElement; count++)
21     {
22         if (SEQ[count] > 0 ) numPositive++;
23         if (SEQ[count] < 0 ) numNegative++;
24         if (SEQ[count] == 0 ) numZero++;
25     }
26
27     cout << "The number of positives = " << numPositive << endl;
28     cout << "The number of negatives = " << numNegative << endl;
29     cout << "The number of zeros = " << numZero << endl;
30
31     return 0;
32 }

```

Hình 5.3: Nhập dãy số nguyên, tính số số hạng dương, âm, bằng không của dãy.

```

1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      const int MAX = 100;
8      double SEQ[MAX];

```

```

9      int count, numElement;
10     int minValue, minId;
11     cout << "Enter the number of elements of sequence: " ; cin >> numElement;
12     cout << "Enter elements of sequence: " << endl;
13     for (count = 0; count < numElement; count++)
14     {
15         cout << "SEQ[" << count << "] = " ;
16         cin >> SEQ[count];
17     }
18
19     minValue = SEQ[0];
20     minId = 0;
21     for (count = 1; count < numElement; count++)
22     {
23         if (SEQ[count] < minValue)
24         {
25             minValue = SEQ[count];
26             minId = count;
27         }
28     }
29
30     cout << "The minimum value of sequences is " << minValue << " at position " <<
        minId << endl;
31
32     return 0;
33 }

```

Hình 5.4: Tìm số bé nhất của một dãy số. In ra số này và vị trí của nó trong dãy. Chương trình sử dụng mảng SEQ để lưu dãy số, `used_size` là số phần tử thực sự trong dãy, `minValue` lưu số bé nhất tìm được và `minId` là vị trí của số này trong dãy. `minValue` được khởi tạo bằng giá trị đầu tiên (`SEQ[0]`), sau đó lần lượt so sánh với các số hạng còn lại, nếu gặp số hạng nhỏ hơn, `minValue` sẽ nhận giá trị của số hạng này. Quá trình so sánh tiếp tục cho đến hết dãy. Vì số số hạng của dãy là biết trước (`used_size`), nên số lần lặp cũng được biết trước (`used_size - 1` lần lặp), do vậy chúng ta sẽ sử dụng câu lệnh `for` cho ví dụ này.

Khởi tạo mảng

Cũng giống như các loại biến khác, trong quá trình khai báo ta cũng có thể kết hợp khởi tạo luôn giá trị cho mảng bằng các cú pháp sau:

```

Type_Name array_name[SIZE] = {value1, value2, ..., valuen};
Type_Name array_name[] = {value1, value2, ..., valuen};

```

- Dạng khai báo thứ 1 cho phép khởi tạo mảng bởi dãy giá trị trong cặp dấu {}, mỗi giá trị cách nhau bởi dấu phẩy (,), các giá trị này sẽ được gán lần lượt cho các phần tử của mảng bắt đầu từ phần tử thứ 0 cho đến hết dãy. Số giá trị có thể bé hơn số phần tử ($n \leq \text{SIZE}$). Các phần tử mảng chưa có giá trị sẽ không được xác định cho đến khi trong chương trình nó được gán một giá trị nào đó.
- Dạng khai báo thứ 2 cho phép vắng mặt số phần tử (`SIZE`), trường hợp này `SIZE` được xác định bởi số giá trị của dãy khởi tạo (`n`). Do đó, nếu vắng mặt cả hai là không được phép (chẳng hạn khai báo `int a[]`; là sai).

Ví dụ:

- Khai báo 3 phân số a, b, c; trong đó $a = 1/3$ và $b = 3/5$:

```
int a[2] = {1, 3} , b[2] = {3, 5} , c[2] ;
```

Ở đây, ta ngầm qui ước thành phần đầu tiên (chỉ số 0) là tử và thành phần thứ hai (chỉ số 1) là mẫu của phân số.

- Khai báo dãy data chứa được 5 số thực độ chính xác gấp đôi:

```
double data[] = { 0, 0, 0, 0, 0 }; // khởi tạo tạm thời bằng 0
```

Trong trường hợp này, mảng data sẽ có số thành phần cố định là 5.

5.1.3 Mảng và hàm

Đối của hàm là mảng

Hiển nhiên, một phần tử đơn lẻ của mảng cũng được xem là một giá trị thông thường nào đó nên nó cũng được phép xuất hiện trong lời gọi hàm (truyền cho hàm), ví dụ: giả sử ta có hàm so sánh 2 số double `getMax(double x, double y)` và mảng thực double `A[n]`; khi đó ta có thể so sánh số đầu tiên và số thứ 5 của mảng A bằng lời gọi `getMax(A[0], A[5])`; chẳng hạn:

```
cout << getMax(A[0], A[5]);
```

Tổng quát hơn, thường chúng ta hay xây dựng các hàm làm việc trên toàn bộ mảng như vectơ hay ma trận các phần tử. Khi đó, tham đối của hàm sẽ phải là các mảng dữ liệu này, ví dụ cần xây dựng hàm tìm phần tử lớn nhất của một mảng 10 phần tử, khi đó ta có thể khai báo hàm như sau:

```
int getMax(int A[10]);
```

Như vậy, nếu cơ quan với 10 người và có bảng lương `int salary_table[10]`, thì ta có thể gọi đến hàm để tìm lương cao nhất bằng lệnh gọi:

```
cout << getMax(salary_table);
```

Trong ví dụ trên, chỉ có các mảng với số phần tử là 10 mới được gọi đến hàm `getMax`. Cách xây dựng hàm với số phần tử cố định như trên (10) là thiếu tính linh hoạt, do vậy C++ cho phép xây dựng hàm không cần khai báo trước số phần tử, tuy nhiên để dễ làm việc, số phần tử này được tách ra như một tham đối thứ 2, và do vậy ta có khai báo hoàn chỉnh:

```
int getMax(int A[], int size);
```

Bây giờ để tìm lương cao nhất ta gọi:

```
cout << getMax(salary_table, 10);
```

và cũng được phép tìm thành phần lớn nhất trong vectơ `int vector[12]` nào đó bằng lệnh gọi:

```
cout << getMax(vector, 12);
```

có nghĩa tham đối `size` ở đây mang tính linh hoạt hơn, nó không nhất thiết để chỉ kích thước cố định của mảng mà còn để chỉ một số thành phần bất kỳ nào đó tùy theo cách cài đặt và mục đích sử dụng của hàm. Ví dụ dưới đây minh họa cho việc tìm lương cao nhất trong bảng lương 5 người bằng lời gọi `cout << getMax(salary_table, 5)`; hoặc cũng có thể chỉ tìm lương cao nhất của 3 người đầu tiên bằng lời gọi `cout << getMax(salary_table, 3)`. Trong ví dụ này, ta đưa thêm hàm `setup` để nhập dữ liệu vào bảng lương `salary_table`.


```

1 #include <iostream>
2
3 using namespace std;
4
5 void Setup(int A[], int used_size)
6 {
7     cout << "Enter " << used_size << " numbers:\n";
8     for (int index = 0; index < used_size; index++)
9         cin >> A[index];
10    return;
11 }
12
13 int GetMax(int A[], int size)
14 {
15     int res = A[0];
16     for (int index = 0; index < size; index++)
17         if (A[index] > res) res = A[index];
18     return res;
19 }
20
21 int main()
22 {
23     int salary_table[5] ;
24     Setup(salary_table, 5);
25     cout << "Highest of salary : " ; cout << GetMax(salary_table, 5) << endl;
26     cout << "Highest salary of first three persons: " ; cout << GetMax(salary_table,
27         3) << endl;
28
29     return 0;
30 }

```

Hình 5.5: Quản lý bảng lương.

Tương tự như lời gọi `getMax(salary_table, 5)` hoặc `getMax(salary_table, 3)`, ta cũng có thể gọi hàm `setup` để nhập dữ liệu cho mảng bất kỳ với kích thước bất kỳ. Ví dụ `setup(score, 3)` (nhập điểm cho 3 sinh viên đầu tiên).

Như đã chú ý trong chương trước, các tham đối của hàm nếu chỉ khai báo thì không cần thiết phải ghi cả tên đối, ví dụ có thể khai báo:

```
int getMax(int[], int);
```

Dưới đây là ví dụ tính tích vô hướng của 2 vectơ có cùng số thành phần (chỉ cần 1 tham đối để chỉ số thành phần chung cho cả 2 vectơ).

```

1 #include <iostream>
2
3 using namespace std;
4
5 void Setup(int[], int);
6 int Procdut(const int[], const int[], int);
7
8 int main()
9 {
10     int vector_A[100], vector_B[100] ;

```

```

11     int index, numElement;
12     cout << "Enter number of elements : " ; cin >> numElement;
13     cout << "Fill up vector A. " ;
14     Setup(vector_A, numElement);
15     cout << "Fill up vector B. " ;
16     Setup(vector_B, numElement);
17
18     cout << "Scalar product of A and B is " << Procdut(vector_A, vector_B,
19         numElement) << endl;
20
21     return 0;
22 }
23
24 void Setup(int A[], int size)
25 {
26     cout << "Enter " << size << " numbers:\n";
27     for (int index = 0; index < size; index++)
28         cin >> A[index];
29     return;
30 }
31
32 int Procdut(const int A[], const int B[], int size)
33 {
34     int res = 0;
35     for (int index = 0; index < size; index++)
36         res += A[index] * B[index];
37     return res;
38 }

```

Hình 5.6: Tích vô hướng.

Trường hợp hàm có nhiều tham đối là mảng với kích thước sử dụng khác nhau ta cần thêm cho hàm mỗi mảng một tham đối kích thước.

Ngăn ngừa việc thay đổi giá trị của mảng – từ khóa **const**

Trong chương 4, khi trình bày về hàm ta đã đề cập đến việc thay đổi giá trị của các biến ngoài nếu các đối của hàm là biến tham chiếu. Bản chất của việc tác động lên biến tham chiếu là tác động đến địa chỉ ô nhớ mà biến tham chiếu đang hướng tới. Một biến mảng cũng gần giống như biến tham chiếu vì tên mảng cũng chính là địa chỉ ô nhớ nơi mảng đó bắt đầu. Vì vậy, các thao tác trên đối mảng thực chất là được thực hiện trên chính mảng ngoài được truyền cho hàm trong lời gọi. Nói cách khác, mọi thay đổi đối với đối mảng cũng thực sự làm thay đổi mảng ngoài tương ứng. Do vậy, để các hàm không vô tình thay đổi các đầu vào (tham đối mảng), thì các tham đối này cần khai báo kèm với từ khóa **const**.

Ví dụ sau đây mô tả việc in giá trị của một dãy số lưu trong mảng `sample_array` tăng thêm 1 đơn vị so với lưu trữ gốc. Chương trình này “vô tình” cũng thay đổi giá trị của mảng `sample_array` dù ta vẫn muốn giữ nguyên các giá trị gốc để dùng về sau.

```

1 #include <iostream>
2
3 using namespace std;
4
5 void Display(int[], int);
6

```

```

7 int main()
8 {
9     int sample_array[5] = { 1, 2, 3, 4, 5 };
10    cout << "Sequence of elements are growed one unit : \n" ;
11    Display(sample_array, 5);
12    cout << "Origin Sequence : \n" ;
13    for (int index = 0; index < 5; index++)
14        cout << sample_array[index] << " ";        // In lại mảng sample_array
15    cout << endl;
16
17    return 0;
18 }
19
20 void Display(int A[], int size)
21 {
22     for (int index = 0; index < 5; index++)
23         cout << ++(A[index]) << " ";    // Tăng thêm 1 cho phần tử mảng và in
24     cout << endl;
25     return;
26 }

```

Hình 5.7: Mảng bị sửa trong hàm

Trong ví dụ này, `sample_array` được khởi tạo bởi 5 số nguyên 1, 2, 3, 4, 5 và sau khi in xong 5 giá trị này đã thay đổi thành 2, 3, 4, 5, 6. Lỗi trong chương trình nằm ở câu lệnh `++A[i]`, nó làm tăng giá trị phần tử thứ `i` của mảng ngoài (`sample_array[i]`) trước khi được in ra.

Để giữ nguyên giá trị cũ của mảng `sample_array`, ta cần thêm từ khóa `const` vào trước tham đối `int A[]`, khi đó chương trình sẽ không chạy (báo lỗi) vì trong hàm `display` ta vẫn sử dụng câu lệnh `cout << ++A[index]`; để khắc phục ta cần sửa lại câu lệnh này thành `cout << A[index] + 1`; như ví dụ dưới.

```

1 #include <iostream>
2
3 using namespace std;
4
5 void Display(const int[], int);
6
7 int main()
8 {
9     int sample_array[5] = { 1, 2, 3, 4, 5 };
10    cout << "Sequence of elements are growed one unit : \n" ;
11    Display(sample_array, 5);
12    cout << "Origin Sequence : \n" ;
13    for (int index = 0; index < 5; index++)
14        cout << sample_array[index] << " ";    // In lại mảng sample_array
15    cout << endl;
16
17    return 0;
18 }
19
20 void Display(const int A[], int size)
21 {
22     for (int index = 0; index < 5; index++)

```

```

23     cout << A[index] + 1 << " "; // Tang them 1 cho phan tu mang va in
24     cout << endl;
25     return;
26 }

```

Hình 5.8: Tham đối const

Kỹ thuật khai báo tham đối hằng (**const**) như trên cũng được sử dụng cho nhiều kiểu tham đối khác để ngăn ngừa việc vô tình làm thay đổi các giá trị gốc của các biến ngoài khi truyền cho các tham đối này.

Tính thiếu nhất quán khi dùng khai báo tham đối hằng (**const**)

Giả sử ta có hàm FUNC với khai báo tham đối mảng A[] là hằng (không cho phép thay đổi A), hàm này gọi đến hàm func để xử lý mảng A[], nhưng func lại không khai báo tham đối mảng hằng (cho phép thay đổi A), ví dụ:

```

void func(int A[], int size)
{
    ...
}
void FUNC(const int A[], int size)
{
    func(A, 10);
}

```

Khi đó hầu hết các chương trình dịch của C++ sẽ báo lỗi hoặc cảnh báo: “Invalid conversion const int* to int*”. Khi đó, nói chung tham đối mảng trong hàm func cũng nên khai báo hằng.

Giá trị trả lại của hàm là mảng

Một hàm không thể trả lại giá trị là một mảng như những giá trị thông thường khác (int, double, ...), ngoài việc kết quả trả lại của hàm (thông qua câu lệnh return) là một con trỏ trỏ đến dãy kết quả (ngầm hiểu như một mảng). Kỹ thuật sử dụng con trỏ sẽ được trình bày trong các chương sau của giáo trình.

Hiện tại, để lấy kết quả là một mảng, ta có thể khai báo thêm một đối mảng trong hàm để lưu giữ kết quả thay cho giá trị trả lại của hàm. Ví dụ cộng hai vec tơ dưới đây minh họa điều này.

```

1 #include <iostream>
2
3 using namespace std;
4
5 void SumVector(int A[], int B[], int C[], int n)
6 {
7     for (int index = 0; index < n; index++)
8         C[index] = A[index] + B[index];
9     return;
10 }
11
12 int main()
13 {
14     int a[3] = {1, 2, 3};
15     int b[3] = {0, 2, 4};
16     int c[3];
17     SumVector(a, b, c, 3);
18     cout << "Sum of two vectors a and b is vector: ( ";

```

```

19     for (int index = 0; index < 2; index++)
20         cout << c[index] << ", ";           // result is vector (1, 4, 7)
21     cout << c[2] << " )" << endl;
22     return 0;
23 }

```

Hình 5.9: Trả kết quả bằng tham đối mảng

Như vậy, ngoài hai đối mảng A, B gần như bắt buộc phải có (là input), hàm còn có một đối mảng C khác để lưu giữ vectơ tổng (output). Các mảng A, B được khai báo hằng (kèm từ khóa `const`), mảng C cần lưu kết quả (thay đổi giá trị) nên không kèm từ khóa này. Kiểu trả lại của hàm là `void` (vì thực chất hàm không trả lại giá trị, nó chỉ ghi kết quả vào mảng C).

5.1.4 Tìm kiếm và sắp xếp

Tìm kiếm

Cho một dãy phần tử (được lưu dưới dạng mảng `data_arr[]`) và một phần tử `target` cho trước. Bài toán đặt ra là: Hãy trả lời `target` có là phần tử của dãy hay không? Nếu có thì nó ở vị trí thứ mấy trong mảng?

Hàm `int search(int A[], int x)` trình bày dưới đây cho phép tìm kiếm `x` trong mảng A bằng cách so sánh lần lượt từng phần tử `A[i]` của mảng với `x`, nếu có phần tử `A[i]` trùng với `x`, thuật toán dừng và trả lại vị trí `i` của phần tử này, nếu không thuật toán sẽ trả lại `-1`. Hàm sử dụng biến `found_at` để lưu kết quả được khởi tạo trước bằng `-1` (ngầm định không tìm thấy), trong quá trình so sánh nếu tìm thấy, `found_at` sẽ được đặt lại là vị trí của phần tử này trong mảng, ngược lại nếu duyệt hết cả mảng mà vẫn không tìm thấy `target` thì mặc nhiên `found_at = -1`.

```

1  #include <iostream>
2
3  using namespace std;
4
5  const int MAX = 50;
6  void Setup(int A[], int used_size);
7  int Search(const int A[], int used_size, int target);
8
9  int main()
10 {
11     int data_arr[MAX], used_size;
12     int target, pos;
13
14     cout << "Enter number of elements: ";
15     cin >> used_size;
16     Setup(data_arr, used_size);
17     cout << "Enter a number to search for: ";
18     cin >> target;
19
20     pos = Search(data_arr, used_size, target);
21
22     if (pos)
23         cout << target << " is stored in sequence at position " << pos << endl
24         << "(Remember: The first position is 0.)\n";
25     else

```

```

26         cout << target << " is not on the sequence.\n";
27
28     return 0;
29 }
30
31 void Setup(int A[], int used_size)
32 {
33     cout << "Enter " << used_size << " numbers: ";
34     for (int index = 0; index < used_size; index++)
35         cin >> A[index];
36     return;
37 }
38
39 int Search(const int A[], int used_size, int target)
40 {
41     int found_at = -1;                // ngam định không tìm thấy target
42     for (int index = 0; index < used_size; index++)
43         if (A[index] == target)
44         {
45             found_at = index;
46             break;
47         }
48     return found_at;
49 }

```

Hình 5.10: Tìm kiếm trên mảng

Sắp xếp

Giả sử cần sắp xếp tăng dần các giá trị của dãy số `int A[n]`. Thuật toán chọn (Selection Sort) là thuật toán đơn giản, dễ hiểu để thực hiện công việc này. Thuật toán được tiến hành bằng cách chọn dần từng số hạng bé nhất, bé “thứ hai”, “thứ ba”, ... để lấp đúng vào vị trí của nó (thứ nhất, thứ hai, thứ ba ...) trong dãy.

Để thuận lợi cho trình bày chi tiết ta tạm quay lại cách đánh số phần tử bắt đầu từ 1 đến n và thực hiện trên dãy cụ thể gồm các số hạng: 30, 25, 8, 12, 4, 21.

Bảng 5.11: Dãy ban đầu

30	25	8	12	4	21
----	----	---	----	---	----

- Đầu tiên, thuật toán tìm chọn số bé nhất trong dãy (từ vị trí 1 đến n), giả sử là `A[k]` và đặt số này lên đầu dãy bằng cách trao đổi nó với `A[1]`. Như vậy, sau bước này ta đã có một số bé nhất đã nằm đúng vị trí (đã được sắp xếp) và với $n-1$ số còn lại chưa được sắp xếp.

Trong ví dụ trên, số bé nhất tìm được là 4 (`A[5]`), trao đổi với `A[1]` (30) ta được dãy mới

Bảng 5.12: Lần trao đổi đầu tiên

4	25	8	12	30	21
---	----	---	----	----	----

- Tiếp theo, ta lại tìm chọn số bé thứ hai để đảo lên vị trí tiếp theo (`A[2]`). Số bé “thứ hai” này chính là số bé nhất trong phần mảng còn lại (chưa được sắp xếp) (từ vị trí 2 đến n).

Trong ví dụ trên, số bé “thứ hai” được tìm là $A[3] = 8$, trao đổi với $A[2]$ ta được dãy mới với 2 vị trí đã sắp xếp

Bảng 5.13: Lần trao đổi thứ thứ hai

4	8	25	12	30	21
---	---	----	----	----	----

- Quá trình tiếp tục như vậy cho đến phần tử thứ $n-1$. Hiển nhiên phần tử thứ n không cần phải sắp xếp.

Áp dụng với ví dụ trên, các bước còn lại như sau:

Bảng 5.14: Quá trình sắp xếp

4	8	25	12	30	21	Chọn 12 đổi chỗ với 25
4	8	12	25	30	21	Chọn 21 đổi chỗ với 25
4	8	12	21	30	25	Chọn 25 đổi chỗ với 30
4	8	12	21	25	30	Sắp xếp xong

Tổng quát: Giả sử đã sắp được $i-1$ vị trí, ta sẽ tìm số bé nhất trong dãy còn lại (từ vị trí thứ i đến n) và trao đổi số này với số ở vị trí thứ i . Quá trình tiếp tục đến khi lấp đủ $n-1$ phần tử vào đúng vị trí (phần tử thứ n hiển nhiên tự vào đúng vị trí). Như vậy thuật toán sẽ cần $n-1$ bước tìm chọn và đổi chỗ.

Có thể mô tả thuật toán bằng lược đồ:

```
for (i = 1 to n - 1)           // chỉ cần sắp n-1 vị trí đầu tiên
{
    - Tìm số bé nhất trong đoạn từ i + 1 đến n
    - Đổi chỗ số vừa tìm được với số thứ i
}
```

Để tìm số bé nhất trong đoạn từ $i + 1$ đến n ta sẽ dùng vòng lặp for tương tự như trong hàm `getMax` (xem phần 5.1.3 – hàm tìm phần tử lớn nhất). Dưới đây là chương trình hoàn chỉnh cho hàm sắp xếp một mảng.

```
1 #include <iostream>
2
3 using namespace std;
4
5 const int MAX = 50;
6 void Setup(int A[], int n);           // ham nhap day so
7 void Print(const int A[], int n);     // ham in day so
8 int Sort(int A[], int n);            // ham sap xep
9
10 int main()
11 {
12     int data_arr[MAX];
13     int used_size, i, j;
14     int tmp;
15     cout << "This program sorts numbers from lowest to highest.\n";
```

```

16     cout << "Enter the number of elements of sequence: " ;
17     cin >> used_size;
18     Setup(data_arr, used_size);
19     Sort(data_arr, used_size);
20     cout << "In sorted order the numbers are:\n";
21     Print(data_arr, used_size);
22
23     return 0;
24 }
25
26 void Setup(int A[], int n)
27 {
28     cout << "Enter " << n << " numbers: ";
29     for (int index = 0; index < n; index++)
30         cin >> A[index];
31     return;
32 }
33
34 void Print(const int A[], int n)                // ham in day so
35 {
36     for (int index = 0; index < n; index++)
37         cout << A[index] << " ";
38     cout << endl;
39     return;
40 }
41
42 int Sort(int A[], int n)                        // ham sap xep
43 {
44     int tmp, min_value;
45     int id_min;
46     for (int id1 = 0; id1 < n-1; id1++)
47     {                                           // chon so be nhat
48         min_value = A[id1];
49         id_min = id1;
50         for (int id2 = id1+1; id2 < n; id2++)
51             if (A[id2] < min_value)
52             {
53                 min_value = A[id2];
54                 id_min = id2;
55             }
56         tmp = A[id1];                          // doi A[id1] voi A[id2]
57         A[id1] = A[id_min];
58         A[id_min] = tmp;
59     }
60 }

```

Hình 5.15: Sắp xếp chọn

Để che giấu bớt chi tiết, ta nên tách các đoạn chương trình chọn số bé nhất và trao đổi 2 số trong sort thành các hàm riêng swap(int &v1, int &v2) (xem chương 4) và get_index_of_smallest(const int A[], int start_index, int last_index) như trong phiên bản dưới của chương trình sắp xếp.

```

1 #include <iostream>

```



```
2
3 using namespace std;
4
5 const int MAX = 50;
6 void Setup(int A[], int n);           // ham nhap day so
7 void Print(const int A[], int n);     // ham in day so
8 void Swap(int &v1, int &v2);         // ham doi 2 gia tri
9 // ham tim chi so cua so be nhat trong day
10 int Get_Index_Of_Smallest(const int A[], int start_index, int last_index);
11 int Sort(int A[], int n);            // ham sap xep
12
13 int main()
14 {
15     int data_arr[MAX];
16     int used_size, i, j;
17     int tmp;
18     cout << "This program sorts numbers from lowest to highest.\n";
19     cout << "Enter the number of elements of sequence: " ;
20     cin >> used_size;
21     Setup(data_arr, used_size);
22     Sort(data_arr, used_size);
23     cout << "In sorted order the numbers are:\n";
24     Print(data_arr, used_size);
25
26     return 0;
27 }
28
29 void Setup(int A[], int n)
30 {
31     cout << "Enter " << n << " numbers: ";
32     for (int index = 0; index < n; index++)
33         cin >> A[index];
34     return;
35 }
36
37 void Print(const int A[], int n)      // ham in day so
38 {
39     for (int index = 0; index < n; index++)
40         cout << A[index] << " ";
41     cout << endl;
42     return;
43 }
44
45 void Swap(int &v1, int &v2)
46 {
47     int tmp;
48     tmp = v1;
49     v1 = v2;
50     v2 = tmp;
51 }
52
53 int Get_Index_Of_Smallest(const int A[], int start_index, int last_index)
54 {
55     int min_value = A[start_index];
```

```

56     int id_min = start_index;
57     for (int i = start_index+1; i <= last_index; i++)
58         if (A[i] < min_value)
59         {
60             min_value = A[i];
61             id_min = i;
62         }
63     return id_min;
64 }
65
66 int Sort(int A[], int n) // ham sap xep
67 {
68     int id_min;
69     for (int i = 0; i < n-1; i++)
70     {
71         id_min = Get_Index_Of_Smallest(A, i, n-1); // chon so be nhat
72         Swap(A[i], A[id_min]);
73     }
74 }

```

Hình 5.16: Sắp xếp chọn (bản cải tiến)

5.2 Lập trình và thao tác với mảng nhiều chiều

5.2.1 Mảng 2 chiều

Để thuận tiện trong việc biểu diễn các loại dữ liệu phức tạp như ma trận hoặc các bảng biểu có nhiều tiêu chí, C++ đưa ra kiểu dữ liệu mảng nhiều chiều. Tuy nhiên, việc sử dụng mảng với số chiều lớn hơn 2 khó lập trình và ít được sử dụng, vì vậy trong mục này chúng ta chỉ bàn đến mảng hai chiều.

Đối với mảng một chiều m phần tử, nếu mỗi thành phần của nó lại là mảng một chiều n phần tử (ví dụ màn hình máy tính là một mảng gồm 24 phần tử (dòng), mỗi phần tử lại là một mảng chứa 80 kí tự) thì ta gọi mảng là hai chiều với số phần tử (hay kích thước) mỗi chiều là m và n . Ma trận là một minh họa cho hình ảnh của mảng hai chiều, nó gồm m dòng và n cột, tức chứa $m \times n$ phần tử, và hiển nhiên các phần tử này có cùng kiểu. Tuy nhiên, về mặt bản chất mảng hai chiều không phải là một tập hợp với $m \times n$ phần tử cùng kiểu mà là tập hợp với m thành phần, trong đó mỗi thành phần là một mảng một chiều với n phần tử. Điểm nhấn mạnh này sẽ được giải thích cụ thể hơn trong các chương sau.

Bảng 5.17: Minh họa mảng hai chiều

	0	1	2	3
0	?	?	?	?
1	?	?	?	?
2	?	?	?	?

Hình trên minh họa hình thức một mảng hai chiều với 3 dòng, 4 cột, cũng giống mảng một

chiều, các chỉ số (dòng, cột) đều được tính từ 0. Thực chất trong bộ nhớ tất cả 12 phần tử của mảng được sắp liên tiếp theo từng dòng của mảng như minh hoạ trong hình dưới đây.

Bảng 5.18: Phân bố bộ nhớ của mảng hai chiều

?	?	?	?	?	?	?	?	?	?	?	?
dòng 0				dòng 1				dòng 2			

Và vì vậy, trông nó giống với (và bản chất là) mảng một chiều. Tuy nhiên, tại thời điểm này, để đơn giản, ta hình dung và sử dụng mảng 2 chiều như hình ảnh của một ma trận.

5.2.2 Thao tác với mảng hai chiều

Khai báo.

Để khai báo mảng hai chiều, ta có thể dùng cú pháp sau:

```
Typename Array_name[SIZE1][SIZE2];
```

Trong đó SIZE1, SIZE2 là các hằng số biểu thị 2 kích thước (hai chiều) của mảng, tức số thành phần (hoặc số phần tử) của mảng và Typename là kiểu của các thành phần. Ví dụ:

```
const int NUM_STUDENTS = 20;
const int NUM_COURSES = 3;
int mark[NUM_STUDENTS][NUM_COURSES];
```

Mảng mark được dùng để ghi điểm số của sinh viên, được hiểu như mark[i][j] là điểm của sinh viên thứ i đạt được đối với môn học thứ j.

Trong khai báo, như mảng một chiều, mảng hai chiều cũng có thể được khởi tạo bằng dãy các dòng giá trị, các dòng cách nhau bởi dấu phẩy, mỗi dòng được bao bởi cặp ngoặc {} và toàn bộ giá trị khởi tạo nằm trong cặp dấu {} hoặc đơn giản hơn có thể khởi tạo bằng một dãy liên tục các giá trị (như ví dụ bên dưới), chương trình tự động nhận biết và gán các giá trị này cho từng dòng của mảng. Ví dụ:

```
int mark[NUM_STUDENTS][NUM_COURSES] = { 1, 2, 3, 4, 5, 6, 7 };
```

Trong khởi tạo trên sinh viên đầu tiên của danh sách có số điểm lần lượt của 3 môn là 1, 2, 3 và sinh viên thứ hai có điểm 4, 5, 6. Điểm môn đầu tiên của sinh viên thứ ba là 7, các môn khác chưa xác định.

Đối với trường hợp khai báo có khởi tạo, có thể bỏ qua kích thước thứ nhất (số dòng) nhưng kích thước thứ hai (số cột) bắt buộc phải có, số dòng sẽ được xác định thông qua khởi tạo. Với ví dụ trên ta cũng được phép khai báo:

```
int mark[][NUM_COURSES] = { 1, 2, 3, 4, 5, 6, 7 };
```

trong khai báo này chương trình tự động xác định số dòng là 3.

Qua các thảo luận trên người đọc có thể tự rút ra kết luận tại sao kích thước thứ hai (số cột) của mảng là bắt buộc phải khai báo.

Trường hợp khởi tạo thiếu một số thành phần ta nên dùng thêm cặp {} một cách tường minh để tránh gây nhầm lẫn. Ví dụ:

```
int mark[NUM_STUDENTS][NUM_COURSES] = { {1, 2}, {3}, {4, 5, 6} };
```

Khởi tạo thiếu điểm môn 3 cho sinh viên 1, thiếu môn 2 và 3 cho sinh viên 2 còn điểm sinh viên 3 được khởi tạo đầy đủ. Các thành phần còn thiếu được mặc định là 0.

Sử dụng mảng hai chiều

- Tương tự mảng một chiều, các chiều trong mảng cũng được đánh số từ 0.
- Không sử dụng các thao tác trên toàn bộ mảng mà phải thực hiện thông qua từng phần tử của mảng.
- Để truy nhập phần tử của mảng, ta sử dụng tên mảng kèm theo 2 chỉ số chỉ vị trí dòng và cột của phần tử.

Ví dụ trong Hình 5.19 minh họa cho việc in ra màn hình điểm số của các sinh viên như được khai báo và khởi tạo ở trên. Trong chương trình, để truy cập đến mỗi thành phần của mảng ta cần dùng 2 chỉ số *i* và *j* cho dòng và cột.

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     const int NUM_STUDENTS = 20;
8     const int NUM_COURSES = 3;
9     int used_num_students;
10
11     used_num_students = 3;
12     int mark[NUM_STUDENTS][NUM_COURSES] = { 1, 2, 3, 4, 5, 6, 7 };
13     for (int i = 0; i < used_num_students; i++)
14     {
15         cout << "Marks of student #" << i+1 << " is: ";
16         for (int j = 0; j < NUM_COURSES; j++)
17             cout << mark[i][j] << " ";
18         cout << endl;
19     }
20     return 0;
21 }
```

Hình 5.19: in ra màn hình điểm số của các sinh viên.

Output hình 5.19

```

Marks of student #1 is 1 2 3
Marks of student #2 is 4 5 6
Marks of student #3 is 7 0 0
```

Với sinh viên thứ ba, điểm 2 môn thứ hai và thứ ba là 0, mặc dù điểm các môn này của sinh viên chưa được khởi tạo. Lý do là vì khi khai báo một biến nào đó, chương trình dịch sẽ bố trí biến này chiếm một số bytes trong bộ nhớ và tự động khởi tạo giá trị “rỗng” cho biến (0, NULL, ...).

Như đã chú ý đối với mảng một chiều, việc truy cập đến thậm chí một phần tử nào đó nằm ngoài vùng khai báo của mảng đều vẫn hợp lệ về mặt cú pháp, do đó nếu in điểm môn nào đó của

sinh viên thứ 21 (nằm ngoài vùng được khai báo của mảng mark) chương trình vẫn chấp nhận, tuy nhiên giá trị in ra sẽ là một giá trị ngẫu nhiên nằm tại ô nhớ này và thường gọi là “rác”.

Tham đối của hàm là mảng hai chiều

Tương tự với mảng một chiều, mảng hai chiều cũng có thể là đối của những hàm xử lý trên mảng. Ví dụ cần viết hàm nhập giá trị cho mảng, hàm in giá trị mảng ra màn hình, hàm in giá trị trung bình từng dòng, từng cột của mảng, ...

Trong mảng một chiều khi tham gia làm đối của hàm, đối này không cần thiết phải khai báo kích thước, thay vào đó ta sẽ bổ sung một tham đối để chỉ kích thước tối đa của mảng hoặc kích thước sử dụng thực tế của mảng tùy từng trường hợp sử dụng.

Đối với mảng hai chiều, tham đối mảng cũng không cần phải khai báo kích thước thứ nhất (số dòng), tuy nhiên bắt buộc phải khai báo kích thước thứ hai (số cột).

Các từ khóa như `const`, ... được sử dụng giống như đối với mảng một chiều. Ví dụ dưới đây minh họa cho các hàm nhập mảng, tính điểm trung bình của từng sinh viên và từng môn học, in ấn kết quả thành dạng bảng biểu trên màn hình. Trong chương trình ngoài mảng mark, ta cần khai báo thêm hai mảng `stu_av[NUM_STUDENTS]` và `course_av[NUM_COURSES]` để lưu trữ điểm trung bình của sinh viên và điểm trung bình của từng môn học.

```

1 #include <iostream>
2 #include <iomanip>
3
4 using namespace std;
5
6 const int NUM_STUDENTS = 20;           // so sinh vien toi da
7 const int NUM_COURSES = 5;            // so mon hoc toi da
8 double stu_av[NUM_STUDENTS];          // mang chua dtb cua tung sinh vien
9 double course_av[NUM_COURSES];        // mang chua dtb theo tung mon hoc
10
11 void Fillup(int mark[][NUM_COURSES], int num_students, int num_courses);
12 void ComputeStudentAve(const int mark[][NUM_COURSES], int num_students, int
    num_courses, double stu_av[]);
13 void ComputeCourseAve(const int mark[][NUM_COURSES], int num_students, int
    num_courses, double course_av[]);
14 void Display(const int mark[][NUM_COURSES], int num_students, int num_courses);
15
16 int main()
17 {
18     int mark[NUM_STUDENTS][NUM_COURSES];
19     int num_students = 4;
20     int num_courses = 3;
21     cout << "Enter real number of students: "; cin >> num_students;
22     cout << "Enter real number of courses: "; cin >> num_courses;
23     Fillup(mark, num_students, num_courses);
24     ComputeStudentAve(mark, num_students, num_courses, stu_av);
25     ComputeCourseAve(mark, num_students, num_courses, course_av);
26     Display(mark, num_students, num_courses);
27     return 0;
28 }
29
30 void Fillup(int mark[][NUM_COURSES], int num_students, int num_courses)
31 {

```

```

32     for (int st_id = 0; st_id < num_students; st_id++)
33     {
34         cout << "Enter " << num_courses << " marks of student #" << st_id + 1 << ": "
35             ;
36         for (int crs_id = 0; crs_id < num_courses; crs_id++)
37             cin >> mark[st_id][crs_id];
38     }
39     return;
40 }
41 void ComputeStudentAve(const int mark[][NUM_COURSES], int num_students, int
    num_courses, double stu_av[])
42 {
43     for (int st_id = 0; st_id < num_students; st_id++)
44     {
45         double sum = 0;
46         for (int crs_id = 0; crs_id < num_courses; crs_id++)
47             sum = sum + mark[st_id][crs_id];
48         stu_av[st_id] = sum/num_courses;
49     }
50     return;
51 }
52
53 void ComputeCourseAve(const int mark[][NUM_COURSES], int num_students, int
    num_courses, double course_av[])
54 {
55     for (int crs_id = 0; crs_id < num_courses; crs_id++)
56     {
57         double sum = 0;
58         for (int st_id = 0; st_id < num_students; st_id++)
59             sum = sum + mark[st_id][crs_id];
60         course_av[crs_id] = sum/num_students;
61     }
62     return;
63 }
64
65 void Display(const int mark[][NUM_COURSES], int num_students, int num_courses)
66 {
67     cout.setf(ios::fixed);
68     cout.setf(ios::showpoint);
69     cout.precision(1);
70     cout << "\nCourse marks and average mark of students\n";
71     cout << setw(10) << "Student";
72     for (int crs_id = 1; crs_id <= num_courses; crs_id++) cout << setw(9) << "Crs#"
73         << crs_id;
74     cout << setw(10) << "Ave" << endl;
75     for (int st_id = 0; st_id < num_students; st_id++)
76     {
77         cout << setw(10) << st_id + 1;
78         for (int crs_id = 0; crs_id < num_courses; crs_id++)
79             cout << setw(10) << mark[st_id][crs_id];
80         cout << setw(10) << stu_av[st_id] << endl;
81     }
82     cout << "Crs. Ave = ";

```

```

82     for (int crs_id = 0; crs_id < num_courses; crs_id++)
83         cout << setw(10) << course_av[crs_id];
84     cout << endl;
85     return;
86 }

```

Hình 5.20: Tính điểm trung bình.

Giá trị trả lại của hàm là mảng hai chiều

Như đã biết, tại thời điểm này ta chưa trình bày về con trỏ nên chưa có cách để hàm trả lại giá trị là một mảng. Tuy nhiên, có thể khai báo mảng kết quả là đối của hàm và hàm sẽ lưu lại kết quả vào đối này.

Ví dụ dưới đây minh họa hàm nhân 2 ma trận: Cho 2 ma trận A ($m \times n$) và B ($n \times p$). Tính ma trận $C = A \times B$, trong đó C có kích thước là $m \times p$. Các ma trận đầu vào (A, B) hiển nhiên là đối (const) của hàm. Ma trận kết quả C cũng sẽ được khai báo là đối thứ 3 (không const) của hàm và kết quả sẽ được lưu tại đây.

Để nhân được ma trận, số cột của A phải bằng số dòng của B, khi đó phần tử dòng i, cột j của ma trận kết quả ($C[i][j]$) được tính là tích vô hướng của dòng i của A và cột j của B.

```

1  #include <iostream>
2  #include <iomanip>
3
4  using namespace std;
5
6  int main()
7  {
8      double A[10][10], B[10][10], C[10][10] ;
9      int m, n, p ;                               // các kích thước của ma trận
10     int i, j, k ;                                // các chỉ số vòng lặp
11
12     cout << "Enter sizes of two matrixes: " ; cin >> m >> n >> p;
13
14     cout << "Enter values of matrix A(" << m << " x " << n << "):\n" ;
15     // Nhập ma trận A
16     for (i = 0; i < m; i++)
17         for (j = 0; j < n; j++)
18             cin >> A[i][j] ;
19
20     cout << "Enter values of matrix B(" << n << " x " << p << "):\n" ;
21     // Nhập ma trận B
22     for (i = 0; i < n; i++)
23         for (j = 0; j < p; j++)
24             cin >> B[i][j] ;
25
26     // Tính ma trận C = A x B
27     for (i = 0; i < m; i++)
28         for (j = 0; j < p; j++)
29         {
30             C[i][j] = 0;
31             for (k = 0; k < n; k++) C[i][j] += A[i][k]*B[k][j] ;
32         }
33

```

```

34 // Hien thi ket qua
35 cout << "Result Matrix C(" << m << " x " << p << "):\n" ;
36 cout << setiosflags(ios::showpoint) << setprecision(2) ;
37 for (i = 0; i < m; i++)
38 for (j = 0; j < p; j++)
39 {
40     if (j == 0) cout << endl;
41     cout << setw(6) << C[i][j] ;
42 }
43 cout << endl;
44
45 return 0;
46 }

```

Hình 5.21: Nhân ma trận.

Để chương trình gọn hơn, các thao tác nhập, nhân ma trận, in kết quả sẽ được viết thành hàm như Hình 5.22.

```

1 #include <iostream>
2 #include <iomanip>
3
4 using namespace std;
5
6 const int SIZE = 10;          // so dong, cot toi da cua cac ma tran A, B, C
7
8 void Fillup(double [][][SIZE], int, int);
9 void ProcdutMatrix(const double [][][SIZE], const double [][][SIZE], double [][][SIZE], int,
10 int, int);
11 void Display(const double [][][SIZE], int, int);
12
13 int main()
14 {
15     double A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE] ;
16     int used_size1, used_size2, used_size3 ; // cac kich thuoc cua ma tran
17
18     cout << "Enter sizes of two matrixes: " ; cin >> used_size1 >> used_size2 >>
19         used_size3;
20     cout << "Enter values of matrix A(" << used_size1 << " x " << used_size2 << "):\n"
21         " ;
22     Fillup(A, used_size1, used_size2);
23     cout << "Enter values of matrix B(" << used_size2 << " x " << used_size3 << "):\n"
24         " ;
25     Fillup(B, used_size2, used_size3);
26     ProcdutMatrix(A, B, C, used_size1, used_size2, used_size3);
27     cout << "Result Matrix C(" << used_size1 << " x " << used_size3 << "):\n" ;
28     Display(C, used_size1, used_size3);
29     cout << endl;
30
31     return 0;
32 }
33
34 void Fillup(double matrix[][SIZE], int num_row, int num_col)
35 {
36     for (int i = 0; i < num_row; i++)

```



```

33     for (int j = 0; j < num_col; j++)
34         cin >> matrix[i][j] ;
35 }
36
37 void ProductMatrix(const double A[][SIZE], const double B[][SIZE], double res[][SIZE]
38                   , int size1, int size2, int size3)
39 {
40     int i, j, k;
41     for (i = 0; i < size1; i++)
42     for (j = 0; j < size2; j++)
43     {
44         res[i][j] = 0;
45         for (k = 0; k < size3; k++) res[i][j] += A[i][k]*B[k][j] ;
46     }
47
48 void Display(const double matrix[][SIZE], int num_row, int num_col)
49 {
50     cout << setiosflags(ios::showpoint) << setprecision(2) ;
51     for (int i = 0; i < num_row; i++)
52     {
53         for (int j = 0; j < num_col; j++)
54             cout << setw(6) << matrix[i][j] ;
55         cout << endl;
56     }
57 }

```

Hình 5.22: Nhân ma trận.

5.3 Lập trình và thao tác với chuỗi ký tự

Một chuỗi ký tự là một dãy bất kỳ các ký tự (kể cả dấu cách), do vậy nó có thể được lưu bằng mảng ký tự. Tuy nhiên, để máy có thể nhận biết được mảng ký tự này là một chuỗi, cần thiết phải có ký tự kết thúc chuỗi, theo qui ước là ký tự có mã 0 (tức '\0') tại vị trí nào đó trong mảng. Khi đó, chuỗi là dãy ký tự bắt đầu từ phần tử đầu tiên (thứ 0) đến ký tự kết thúc chuỗi đầu tiên (không kể các ký tự còn lại trong mảng).

Dưới đây là hình ảnh minh họa cho 3 chuỗi s1, s2, s3.

Bảng 5.23: Chuỗi ký tự

	0	1	2	3	4	5	6	7	// Vị trí các phần tử của mảng
s1	H	E	L	L	O	\0	A	B	// s1 = "HELLO"
s2	H	E	L	L	\0	O	\0	A	// s2 = "HELL"
s3	\0	H	E	\0	L	L	O	\0	// s3 = ""

Hình vẽ trên minh họa 3 chuỗi, mỗi chuỗi được chứa trong mảng ký tự có độ dài tối đa là 8. Nội dung chuỗi thứ nhất là "Hello" có độ dài thực tế là 5 ký tự, chiếm 6 ô trong mảng (thêm ô chứa ký tự kết thúc '\0'). Chuỗi thứ hai có nội dung "Hell" với độ dài 4 (chiếm 5 ô) và chuỗi cuối cùng biểu

thì một xâu rỗng (chiếm 1 ô). Chú ý mảng kí tự được khai báo với độ dài 8 tuy nhiên các xâu có thể chỉ chiếm một số kí tự nào đó trong mảng này và tối đa là 7 kí tự.

Lưu ý, một hằng kí tự được viết giữa cặp dấu nháy đơn, còn hằng xâu được bao bởi cặp dấu nháy kép. Ví dụ 'A' thể hiện hằng kí tự A, nó chỉ chiếm 1 byte trong bộ nhớ, trong khi đó "A" thể hiện xâu kí tự A có độ dài 1 (kí tự) nhưng lại chiếm 2 bytes trong bộ nhớ (cần thêm byte '\0' để kết thúc xâu).

5.3.1 Khai báo

Một xâu kí tự thực chất là mảng kí tự, do vậy để khai báo xâu ta dùng mảng như sau:

```
char string_name[size] ;           // không khởi tạo
char string_name[size] = string ;  // có khởi tạo
char string_name[] = string ;      // có khởi tạo
```

- Size là kích thước mảng kí tự, với kích thước này sẽ đủ để chứa xâu dài nhất lên đến size - 1 kí tự (1 kí tự còn lại dành để lưu dấu kết thúc xâu). Độ dài thực sự của xâu được tính từ đầu mảng đến kí tự '\0' đầu tiên (không kể dấu này). Do vậy, để chứa một xâu có độ dài tối đa n cần phải khai báo mảng s với kích thước ít nhất là n + 1.
- Cách khai báo thứ hai có kèm theo khởi tạo xâu, đó là dãy kí tự đặt giữa cặp dấu nháy kép. Ví dụ:

```
char name[26] ;           // xâu họ tên chứa tối đa 25 kí tự
char course[31] = "Programming Language C++" ;
```

Xâu course chứa tối đa 30 kí tự, được khởi tạo với nội dung "Programming Language C++" với độ dài thực sự là 24 kí tự (chiếm 25 ô đầu tiên trong mảng char course[31]).

- Cách khai báo thứ 3 tự chương trình sẽ quyết định độ dài của mảng bởi xâu khởi tạo (bằng độ dài xâu + 1). Ví dụ:

```
char month[] = "December" ;           // độ dài mảng = 9
```

Cần phân biệt 2 cách khai báo sau:

```
char month_1[100] = "December" ;      // và
char month_2[100] = {'D', 'e', 'c', 'e', 'm', 'b', 'e', 'r'} ;
```

8 bytes đầu tiên của cả hai mảng đều lưu nội dung là dãy kí tự D-e-c-e-m-b-e-r, tuy nhiên với cách khai báo (và khởi tạo) thứ nhất chương trình sẽ tự động gán thêm kí tự '\0' vào bytes thứ 9 của month_1, còn cách thứ 2 thì không. Nói cách khác, month_2 chỉ là một mảng kí tự đơn thuần còn month_1 là một xâu kí tự.

5.3.2 Thao tác với xâu kí tự

Tương tự như các mảng dữ liệu khác, xâu kí tự có những đặc trưng như mảng, tuy nhiên chúng cũng có những điểm khác biệt. Dưới đây là các điểm giống và khác nhau đó.

- Truy cập một kí tự trong xâu: cú pháp giống như mảng. Ví dụ:

```
// chú ý ký tự ' phải được viết là \'
char str[50] = "I\'m a student" ;
cout << str[0] ;           // in ký tự đầu tiên, tức ký tự 'I'
str[1] = 'a' ;             // đặt lại ký tự thứ 2 là 'a'
```

- Không được thực hiện các phép toán trực tiếp trên chuỗi như:

```
// khai báo hai chuỗi str = "Hello" và t
char str[20] = "Hello", t[20] ;
t = "Hello" ;           // sai, không gán được mảng cho 1 hằng
t = str ;               // sai, không gán được hai mảng cho nhau
if (str < t)             // sai, không so sánh được hai mảng
```

- Toán tử nhập dữ liệu >> vẫn dùng được nhưng có nhiều hạn chế. Ví dụ

```
char str[60] ;
cin >> str ;
cout << str ;
```

Nếu chuỗi nhập vào là "Tin học hóa" chẳng hạn thì toán tử >> chỉ nhập "Tin" cho `str` (bỏ tất cả các ký tự đứng sau dấu trắng), vì vậy khi in ra trên màn hình chỉ có từ "Tin".

Vì các phép toán không dùng được trực tiếp trên chuỗi nên các chương trình dịch đã viết sẵn các hàm thư viện được khai báo trong file nguyên mẫu `cstring`. Các hàm này giải quyết được hầu hết các công việc cần thao tác trên chuỗi. Nó cung cấp cho NSD phương tiện để thao tác trên chuỗi như gán, so sánh, sao chép, tính độ dài chuỗi, ... Để sử dụng được các hàm này đầu chương trình cần khai báo `#include <cstring>`.

- Khi duyệt chuỗi, để nhận biết hết chuỗi có thể dùng một trong hai cách sau:
 - Ký tự hiện tại của chuỗi là `'\0'`
 - Ký tự đang đọc nằm tại vị trí `strlen(str)` (`strlen()` là hàm trả lại độ dài của chuỗi `str`). Đây cũng chính là vị trí của ký tự `'\0'`.

Ví dụ: Cần duyệt từ đầu đến cuối chuỗi `str`:

- `for (int index = 0; index < strlen(str); index++)` hoặc
- `for (int index = 0; str[index] != '\0'; index++)`

5.3.3 Phương thức nhập chuỗi (`#include <iostream>`)

Do toán tử nhập >> có hạn chế đối với chuỗi ký tự, nên C++ đưa ra hàm riêng (còn gọi là phương thức) `cin.getline(s, n)` để nhập chuỗi ký tự. Hàm có 2 đối với `s` là chuỗi cần nhập nội dung và `n-1` là số ký tự tối đa của chuỗi. Nếu chuỗi do NSD nhập vào nhiều hơn `n-1` ký tự hàm chỉ lấy `n-1` ký tự đầu tiên đã nhập để gán cho `s`.

Ví dụ: Nhập một ngày tháng dạng Mỹ (mm/dd/yy), đổi sang ngày tháng dạng Việt Nam (dd/mm/yy) rồi in ra màn hình.

```
1 #include <iostream>
2 #include <cstring>
3
```

```

4 using namespace std;
5
6 int main()
7 {
8     char us_date[9], vn_date[9] = "  /  /  ";
9     cout << "Enter the date in format mm/dd/yy: ";
10    cin.getline(us_date, 9);
11    vn_date[0] = us_date[3]; vn_date[1] = us_date[4] ;           // ngay
12    vn_date[3] = us_date[0]; vn_date[4] = us_date[1] ;           // thang
13    vn_date[6] = us_date[6]; vn_date[7] = us_date[7] ;           // nam
14    cout << "Date in American style: " << us_date << endl;
15    cout << "Date in Vietnamese style: " << vn_date << endl;
16
17    return 0;
18 }

```

Hình 5.24: Đổi ngày từ dạng Mỹ sang dạng Việt.

5.3.4 Một số hàm làm việc với chuỗi ký tự (#include <cstring>)

Sao chép chuỗi

- `strcpy(s, t)`: Gán nội dung của chuỗi `t` cho chuỗi `s` (thay cho phép gán = không được dùng). Hàm sẽ sao chép toàn bộ nội dung của chuỗi `t` (kể cả ký tự kết thúc chuỗi) vào chuỗi `s`. Để sử dụng hàm này, cần đảm bảo độ dài của mảng `s` ít nhất cũng bằng độ dài của mảng `t`. Trong trường hợp ngược lại, ký tự kết thúc chuỗi sẽ không được ghi vào `s` và điều này có thể gây treo máy khi chạy chương trình.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[10], str_t[10] ;
9     // str_t = "Face" ;           // không được
10    strcpy(str_t, "Face") ;       // gán "Face" cho t
11    // str_s = str_t ;             // không được
12    strcpy(str_s, str_t) ;         // sao chép t sang s
13    cout << str_s << " to " << str_t << endl; // in ra: Face to Face
14    return 0;
15 }

```

Hình 5.25: Ví dụ `strcpy`.

- `strncpy(s, t, n)`: Sao chép `n` ký tự đầu tiên của `t` vào `s`. Hàm này chỉ làm nhiệm vụ sao chép, không tự động gán ký tự kết thúc chuỗi cho `s`. Do vậy NSD phải thêm câu lệnh đặt ký tự `'\0'` vào cuối chuỗi `s` sau khi sao chép xong.

```

1 #include <iostream>
2 #include <cstring>

```

```

3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[10], str_t[10] = "Steven";
9     strncpy(str_s, str_t, 5) ;           // copy 5 kí tự "Steve" vào s
10    str_s[5] = '\0' ;                   // đặt dấu kết thúc chuỗi
11    cout << str_s << " is younger brother of " << str_t << endl;
12                                     // in : Steve is younger brother of Steven
13
14    return 0;
15 }

```

Hình 5.26: Ví dụ strncpy.

Dạng tổng quát của hàm trên là: `strncpy(s + a, t + b, n)` cho phép copy n ký tự bắt đầu từ vị trí (chỉ số) b của chuỗi t và đặt (ghi đè) vào chuỗi s bắt đầu từ vị trí a .

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char us_date[12] = "02/29/2015";
9     char vn_date[12] = "    /    /    ";
10    strncpy(vn_date + 0, us_date + 3, 2) ;           // ngày
11    strncpy(vn_date + 3, us_date + 0, 2) ;           // tháng
12    strncpy(vn_date + 6, us_date + 6, 4);           // năm
13    cout << "Date in American style: " << us_date << endl;
14    cout << "Date in Vietnamese style: " << vn_date << endl;
15
16    return 0;
17 }

```

Hình 5.27: Chuyển đổi ngày bằng strncpy.

Ghép hai chuỗi

- `strcat(s, t)`: Nối một bản sao của t vào sau s (thay cho phép $+$). Hiển nhiên, hàm sẽ loại bỏ ký tự kết thúc chuỗi s trước khi nối thêm t . Việc nối sẽ đảm bảo lấy cả ký tự kết thúc của chuỗi t vào cho s (nếu s đủ chỗ) vì vậy NSD không cần thêm ký tự này vào cuối chuỗi. Tuy nhiên, hàm không kiểm tra xem liệu độ dài của s có đủ chỗ để nối thêm nội dung, việc kiểm tra này phải do NSD đảm nhiệm.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {

```

```

8     char str_s[100], str_t[100] = "Steve" ;
9     strncpy(str_s, str_t, 3);
10    str_s[3] = '\\0';                // s = "Ste"
11    strcat(str_s, "p");              // s = "Step"
12    cout << str_t << " goes " << str_s << " by " << str_s << endl;
13                                   // Steve goes Step by Step
14
15    return 0;
16 }

```

Hình 5.28: Ví dụ strcat.

- **strncat(s, t, n)**: Nối bản sao **n** kí tự đầu tiên của chuỗi **t** vào sau chuỗi **s**. Hàm tự động đặt thêm dấu kết thúc chuỗi vào **s** sau khi nối xong (tương phản với **strncpy()**). Cũng giống **strcat**, hàm đòi hỏi độ dài của **s** phải đủ chứa kết quả.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[20] = "Do be do" ;
9     char str_t[20] = "Steve is going home" ;
10    strcat(str_s, " to ");
11    strncat(str_s, str_t, 5);
12    cout << str_s << endl;           // Do be do to Steve
13
14    return 0;
15 }

```

Hình 5.29: Ví dụ strncat.

Tương tự, **strncpy**, hàm **strncat** cũng có dạng mở rộng **strncat(s, t + b, n)**; tức nối **n** kí tự của **t** kể từ vị trí **b** vào cho **s**.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[20] = "Do be do to" ;
9     char str_t[20] = "How are you ?" ;
10    strncat(str_s, str_t + 7, 4);
11    cout << str_s << endl;           // Do be do to you
12
13    return 0;
14 }

```

Hình 5.30: Ví dụ strncat.

So sánh hai chuỗi

- `strcmp(s, t)`: Hàm so sánh 2 chuỗi `s` và `t` (thay cho các phép toán so sánh). Giá trị trả lại là hàm dấu của hiệu 2 ký tự khác nhau đầu tiên của `s` và `t`. Tức nếu hiệu là âm thì sẽ trả lại `-1` (tương đương `s1 < s2`), nếu hiệu dương trả lại giá trị `1` (`s1 > s2`), nếu so sánh đến hết chuỗi, tất cả hiệu đều bằng `0` (`s1 == s2`) thì sẽ trả lại `0`. Trong trường hợp chỉ quan tâm đến so sánh bằng, nếu hàm trả lại giá trị `0` là hai chuỗi bằng nhau và nếu giá trị trả lại khác `0` là hai chuỗi khác nhau (chú ý: chuỗi với độ dài ngắn hơn không nhất thiết là chuỗi bé hơn).

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[20] = "Ha Noi" ;
9     char str_t[20] = "Ha noi" ;
10    cout << "Result of comparison " << str_s << " and " << str_t << " is " <<
        strcmp(str_s, str_t) << endl;           // -1
11    if (strcmp(str_s, str_t) < 0)
12        cout << str_s << " < " << str_t;
13    else
14        cout << str_s << " > " << str_t;
15    cout << endl;
16    return 0;
17 }

```

Hình 5.31: Ví dụ `strcmp`.

- `strncmp(s, t, n)`: Giống hàm `strcmp(s, t)` nhưng chỉ so sánh tối đa `n` ký tự đầu tiên của hai chuỗi.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     char str_s[20] = "Ha Noi" ;
9     char str_t[20] = "Ha noi" ;
10    if (strncmp(str_s, str_t, 2) == 0)
11        cout << "The first two characters of " << str_s << " and " << str_t << "
            is the same" << endl;
12    else
13        cout << "The first two characters of " << str_s << " and " << str_t << "
            is not the same" << endl;
14    cout << endl;
15    return 0;
16 }

```

Hình 5.32: Ví dụ `strncmp`.

- `strcmpi(s, t)`: Như `strcmp(s, t)` nhưng không phân biệt chữ hoa, thường. Khi đó "HA NOI" và "Ha Noi" là giống nhau nếu so sánh bằng `strcmpi`.

Lấy độ dài chuỗi

- `strlen(s)`: Hàm trả giá trị là độ dài của chuỗi `s`. Ví dụ:

```
char str[10] = "Ha Noi" ;
cout << strlen(str) ;           // 6
```

5.3.5 Các hàm chuyển đổi chuỗi dạng số thành số (#include <cstdlib>)

- `atoi(str)` ; `atol(str)` ; `atof(str)` ; (A-TO-I: alphabet to integer, ...)

Đối `str` là chuỗi ký tự biểu thị cho giá trị số phù hợp. Các hàm trên lần lượt chuyển đổi `str` thành số nguyên (`int`), số nguyên dài (`long`) và số thực (`double`). Trường hợp, trong chuỗi có chứa cả ký tự khác với chữ số, hàm sẽ chỉ chuyển đổi giá trị là các chữ số đứng trước ký tự lạ đầu tiên trong chuỗi. Ví dụ:

```
int x = atoi("123");           // x = 123
double y = atof("12.345");     // y = 12.345
atoi("VND123") = 0; atoi("1$23") = 1; atoi("123%") = 123
atof("a12.345") = 0.0; atof("1a2.345") = 1.0;
atof("12.34$5") = 12.34;
```

Chú ý ký tự 'e' hoặc 'E' được hiểu là số viết dưới dạng dấu phẩy động, nên: `atof("12.34e5") = 1.234E6` tức 1234000;

Trong ví dụ dưới đây, chúng ta sử dụng hàm `atoi` để tạo phiên bản mới `new_atoi` có tính năng chấp nhận mọi chữ số có trong chuỗi để ghép thành số nguyên. Hàm trả lại 0 nếu chuỗi không chứa chữ số nào. Ví dụ: `"$123" = 123`, `"a12$5b" = 125`, `"abc" = 0`.

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 const int MAX_SIZE = 10;
7
8 int New_Atoi(char input_str[])
9 {
10     char output_str[MAX_SIZE];
11     int num_digit = 0;
12     for (int index = 0; index < strlen(input_str); index++)
13     {
14         if ('0' <= input_str[index] && input_str[index] <= '9')
15             output_str[num_digit++] = input_str[index] ;
16     }
17     output_str[num_digit] = '\0';
18     if (num_digit > 0)
19         return atoi(output_str);
20     else
21         return 0;
```



```

22 }
23
24
25 int main()
26 {
27     char str[MAX_SIZE];
28     int res;
29     cout << "Enter string (less than or equal to 9 characters): ";
30     cin.getline(str, MAX_SIZE);
31     res = New_Atoi(str);
32     cout << res << endl;
33     return 0;
34 }

```

Hình 5.33: Hàm new_atoi.

Cách khai báo chuỗi dưới dạng mảng còn nhiều hạn chế, khó lập trình. Chuỗi được khai báo dưới dạng con trỏ ký tự và đặc biệt C++ cung cấp một lớp riêng về chuỗi (lớp String) sẽ giúp người lập trình làm việc thuận lợi hơn.

5.3.6 Một số ví dụ làm việc với chuỗi

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 int main()
7 {
8     const int MAX = 80;
9     char input_str[MAX];
10    int num_char = 0;
11    cout << "Enter any string: ";
12    cin.getline(input_str, MAX);
13    for (int index = 0; index < strlen(input_str); index++)
14        if (input_str[index] == 'a') num_char++;
15    cout << "Number of characters 'a' that is belong in the string: " << num_char <<
        endl ;
16    return 0;
17 }

```

Hình 5.34: Thống kê số chữ 'a' xuất hiện trong chuỗi s.

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 const int MAX = 80;
7
8 int Str_Length(char str[])
9 {
10     int num_char = 0;

```

```

11     for (int index = 0; str[index] != '\0'; index++)
12         num_char++;
13     return num_char;
14 }
15
16 int main()
17 {
18     char input_str[MAX];
19     cout << "Enter a string: ";
20     cin.getline(input_str, MAX);
21     cout << "Length of the string = " << Str_Length(input_str) << endl ;
22     return 0;
23 }

```

Hình 5.35: Tính độ dài chuỗi bằng cách đếm từng kí tự (tương đương với hàm strlen()).

```

1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5  const int MAX = 80;
6  const char BLANK = ' ';
7
8  int Get_Num_Words(char str[])
9  {
10     int num_words = 0;
11     bool reading_blank = true;
12     int index = 0;
13     while (str[index] != '\0')
14     {
15         if (reading_blank)
16         {
17             if (str[index] != BLANK)
18             {
19                 num_words ++;
20                 reading_blank = false;
21             }
22         }
23         else
24         {
25             if (str[index] == BLANK) reading_blank = true;
26         }
27         index++;
28     }
29     return num_words;
30 }
31
32 int main()
33 {
34     char inp_str[MAX];
35     cout << "Enter a string: ";
36     cin.getline(inp_str, MAX);
37
38     cout << "Number of words = " << Get_Num_Words(inp_str) << endl;

```

```

39
40     return 0;
41 }

```

Hình 5.36: Đếm số từ (qui ước là dãy ký tự bất kỳ, liên tục) trong chuỗi. Chương trình đọc từ đầu đến cuối chuỗi, sử dụng biến logic `reading_blank` để chỉ trạng thái hiện tại đang đọc dấu cách hay ký tự khác. Nếu ở trạng thái đang đọc dấu cách và gặp ký tự khác thì tăng số từ lên 1 và đổi trạng thái. Nếu trạng thái đang đọc ký tự khác và gặp dấu cách thì đổi trạng thái, nếu gặp ký tự khác thì không làm gì.

```

1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5  const int MAX = 80;
6  const char BLANK = ' ';
7  const char END_OF_STRING = '\0';
8
9  int main()
10 {
11     char str[MAX];
12     int head_flag, tail_flag;
13
14     cout << "Enter a string: ";
15     cin.getline(str, MAX);
16
17     head_flag = 0;
18     while (str[head_flag] == BLANK) head_flag++;
19     tail_flag = strlen(str) - 1;
20     while (str[tail_flag] == BLANK) tail_flag--;
21
22     int new_len = tail_flag - head_flag - 1;
23     strncpy(str, str + head_flag, new_len);
24     str[new_len] = END_OF_STRING;
25
26     cout << "New String : \"" << str << "\"" << endl;
27
28     return 0;
29 }

```

Hình 5.37: Cắt dấu cách 2 đầu của chuỗi `s`. Chương trình sử dụng cờ `head_flag` và `tail_flag` chạy từ 2 đầu chuỗi đến vị trí đầu tiên có ký tự khác dấu trắng. Dùng hàm `strncpy` sao chép đoạn ký tự từ `head_flag` đến `tail_flag` vào lại đầu chuỗi, sau đó đặt dấu kết thúc.

```

1  #include <iostream>
2  #include <cstring>
3  #include <conio.h>
4
5  using namespace std;
6  const char END_OF_STRING = '\0';
7  const char key[11] = "HaNoi2000";
8
9  int main()

```

```

10 {
11     char password[11];
12     int num_times = 1; // cho phép nhập lần đầu
13     cout << "Enter password:\n";
14     do {
15         int index = 0;
16         while ((password[index] = getch()) != 13 && ++index <= 10) cout << "X" ;
17             // 13 = enter
18         cout << "\n" ;
19         password[index] = END_OF_STRING;
20         if ( !strcmp(password, key))
21         {
22             cout << "Correct Password. Continue, please.\n" ;
23             break;
24         }
25         else cout << "Incorrect Password. " ;
26         num_times ++;
27         if (num_times <= 3)
28             cout << "Reenter\n";
29         else
30             cout << "You don't be permitted to use this software\n";
31     } while (num_times <= 3);
32
33     return 0;
34 }

```

Hình 5.38: Nhập mật khẩu (không quá 10 kí tự). In ra "đúng" nếu là "HaNoi2000", "sai" nếu ngược lại. Chương trình cho phép nhập tối đa 3 lần. Nhập riêng rẽ từng kí tự (bằng hàm getch() (#include <conio.h>) cho mật khẩu. Hàm getch() không hiện kí tự NSD gõ vào, thay vào đó NSD hiện kí tự 'X' để che giấu mật khẩu. Sau khi NSD nhấn Enter, chương trình so sánh xâu vừa nhập với "HaNoi2000", nếu đúng chương trình tiếp tục, nếu sai tăng số lần nhập (cho phép không quá 3 lần).

Bài tập

- Hãy nhập vào 16 số nguyên. In ra thành 4 dòng, 4 cột.
- Nhập vào dãy n số thực. Tính tổng dãy, trung bình dãy, tổng các số âm, dương và tổng các số ở vị trí chẵn, vị trí lẻ trong dãy. Tìm phần tử gần số trung bình nhất của dãy.
- Nhập vào dãy n số. Hãy in ra số lớn nhất, bé nhất của dãy.
- Tìm và chỉ ra vị trí xuất hiện đầu tiên của phần tử x trong dãy.
- Nhập vào dãy số. In ra dãy đã được sắp xếp tăng dần, giảm dần.
- Cho dãy đã được sắp tăng dần. Chèn thêm vào dãy phần tử x sao cho dãy vẫn sắp xếp tăng dần.
- Cho 2 dãy số đã sắp xếp tăng dần. Không dùng mảng phụ, viết chương trình in ra dãy trộn của 2 dãy trên (thành 1 dãy) cũng theo thứ tự sắp xếp tăng dần.
- Cho hai dãy a , b . Kiểm tra a có là dãy con của b hay không ? (a là dãy con của b nếu ta bỏ bớt một số phần tử trong b thì thu được dãy a . Ví dụ 1, 3, 5 là dãy con của 3, 4, 1, 2, 3, 7, 8, 5, 0, 9.
- Câu nào trong các khẳng định sau là sai :
 - Các phần tử của mảng được sắp xếp liên tục trong bộ nhớ.
 - Các phần tử của mảng chiếm một số byte như nhau.
 - Với mảng `int x[3][4]` số nguyên cuối cùng là phần tử `x[3][4]`.
 - Cho phép khởi tạo giá trị của mảng ngay trong khai báo.
- Xét 2 khởi tạo :


```
int a[2][3] = {{1, 2}, {3, 4, 5}} ;
int b[2][3] = {1, 2, 3, 4, 5} ;
```

Câu nào sau đây sai ?

 - Cách khởi tạo mảng b là được phép.
 - $a[i][j] = b[i][j]$ ($i = 0,1$; $j = 0,1,2$).
 - Hai mảng a , b có cùng số phần tử.
 - Các phần tử chưa được khởi tạo trong a , b sẽ nhận giá trị 0.
- Cho một ma trận nguyên kích thước $m \times n$. Tính:
 - Tổng tất cả các phần tử của ma trận.
 - Tổng tất cả các phần tử dương của ma trận.
 - Tổng tất cả các phần tử âm của ma trận.
 - Tổng tất cả các phần tử chẵn của ma trận.
 - Tổng tất cả các phần tử lẻ của ma trận.

12. Cho một ma trận thực kích thước $m \times n$. Tìm:
- (a) Số nhỏ nhất, lớn nhất (kèm chỉ số) của ma trận.
 - (b) Số nhỏ nhất, lớn nhất (kèm chỉ số) của từng hàng của ma trận.
 - (c) Số nhỏ nhất, lớn nhất (kèm chỉ số) của từng cột của ma trận.
 - (d) Số nhỏ nhất, lớn nhất (kèm chỉ số) của đường chéo chính của ma trận.
 - (e) Số nhỏ nhất, lớn nhất (kèm chỉ số) của đường chéo phụ của ma trận.
13. Giả sử có mảng số thực $A(m, n)$. Một phần tử gọi là điểm yên ngựa nếu nó là phần tử nhỏ nhất trong hàng và lớn nhất trong cột. Viết chương trình nhập một ma trận, in ra phần tử yên ngựa và chỉ số của nó (nếu có).
14. Giả sử có khai báo `char str[12] = "abcdef"`. Câu nào sau đây sai:
- (a) `str[6] = '\0'`.
 - (b) Độ dài của `str` bằng 6.
 - (c) `str` là một mảng kí tự.
 - (d) `str` là một chuỗi kí tự.
15. Cho khai báo `char a[8];`. Để gán giá trị “tin hoc” cho `a`, 2 sinh viên viết theo 2 cách khác nhau: sinh viên 1: `a = "tin hoc"`; sinh viên 2: `strcpy(a, "tin hoc")`. Chọn câu đúng nhất trong các câu sau:
- (a) sinh viên 1 đúng.
 - (b) sinh viên 2 đúng.
 - (c) cả 2 sinh viên đều đúng.
 - (d) cả 2 sinh viên đều sai.
16. Xét 2 khởi tạo: `char x[] = {'C', 'N', 'T', 'T'}` và `char y[] = "CNTT"`. Chọn câu đúng nhất trong các khẳng định sau:
- (a) Cả hai khởi tạo trên là không hợp lệ.
 - (b) `x` và `y` là 2 chuỗi kí tự như nhau.
 - (c) Chỉ có khởi tạo của `x` là được phép.
 - (d) Số phần tử mảng được khởi tạo trong `x` và trong `y` là khác nhau.
17. Đọc chuỗi vào, in ra "dung" nếu chuỗi vào là "Ha Noi". Ngược lại in ra "sai".
18. Nhập chuỗi. Không phân biệt viết hoa hay viết thường, hãy in ra các kí tự có mặt trong chuỗi và số lần xuất hiện của nó (ví dụ chuỗi "Trach - Van - Doanh" có chữ 'a' xuất hiện 3 lần, c(1 lần), d(1), h(2), n(2), o(1), r(1), t(1), -(2), space(4)).
19. Nhập chuỗi. Tính số từ có trong chuỗi. In mỗi dòng một từ.

Chương 6

Các kiểu dữ liệu trừu tượng

6.1 Kiểu dữ liệu trừu tượng bằng cấu trúc (struct)

Trong chương trước, ta thấy để lưu trữ các giá trị gồm nhiều thành phần dữ liệu giống nhau ta có thể sử dụng kiểu mảng. Tuy nhiên, trong thực tế rất nhiều dữ liệu là tập các kiểu dữ liệu khác nhau tập hợp lại, ví dụ lý lịch của mỗi người gồm nhiều kiểu dữ liệu khác nhau như họ tên, tuổi, giới tính, mức lương ... để quản lý dữ liệu kiểu này C++ đưa ra kiểu dữ liệu cấu trúc.

Kiểu cấu trúc giống kiểu mảng ở chỗ cùng quản lý một tập hợp các dữ liệu chia thành các thành phần. Các thành phần trong kiểu mảng được truy cập thông qua chỉ số, còn mỗi thành phần trong kiểu cấu trúc (còn được gọi là trường) sẽ được truy cập thông qua tên gọi của thành phần đó. Điểm giống và khác nhau nữa giữa kiểu mảng và cấu trúc là các thành phần được lưu trữ liên tiếp nhau trong bộ nhớ, tuy nhiên số bytes của từng thành phần trong kiểu cấu trúc là khác nhau, khác với kiểu mảng độ dài của các thành phần này là giống nhau vì chúng có cùng kiểu. Ví dụ, trong chương trình quản lý điểm tốt nghiệp của sinh viên, mỗi sinh viên sẽ là một đối tượng mà nó có ít nhất 3 thành phần dữ liệu cần phải có là: họ tên, năm sinh, điểm tốt nghiệp. Để quản lý đối tượng sinh viên như trên ta có thể xây dựng kiểu cấu trúc như sau:

```
struct Student
{
    char name[30];
    int birth_year;
    double mark;
};
```

Lưu ý, ở đây `Student` được gọi là thẻ tên (`identifier_tag`) của kiểu cấu trúc chứ không phải tên biến. Để đơn giản ta có thể gọi là kiểu cấu trúc `Student` hay ngắn gọn là kiểu `Student` (như các kiểu chuẩn `int`, `double`, `bool`, ...). Trong kiểu `Student` có chứa 3 thành phần với kiểu khác nhau là: xâu ký tự, số nguyên và số thực tương ứng với các tên thành phần này là: `name`, `birth_year`, `mark`.

Thông thường, các kiểu cấu trúc hay được dùng chung cho các hàm nên phần lớn chúng được khai báo như kiểu toàn cục. Tóm lại, việc xây dựng một thẻ tên kiểu cấu trúc hay kiểu cấu trúc sẽ tuân theo cú pháp sau.

6.1.1 Khai báo, khởi tạo

```
struct identifier_tag
{
    list of members ;
} var_list ;
```

- Mỗi thành phần (member) giống như một biến riêng của kiểu, nó gồm kiểu và tên thành phần. Một thành phần cũng còn được gọi là trường (field).
- Phần tên (tag) của kiểu cấu trúc và phần danh sách biến có thể có hoặc không. Tuy nhiên trong khai báo kí tự kết thúc cuối cùng phải là dấu chấm phẩy (;).
- Các kiểu cấu trúc được phép khai báo lồng nhau, nghĩa là một thành phần của kiểu cấu trúc có thể lại là một trường có kiểu cấu trúc khác.
- Một biến có kiểu cấu trúc sẽ được phân bố bộ nhớ sao cho các thành phần của nó được sắp xếp nhau liên tục theo thứ tự xuất hiện trong khai báo.
- Khai báo biến kiểu cấu trúc cũng giống như khai báo các biến kiểu cơ sở dưới dạng:

```
struct identifier_tag list_of_var ; // kiểu cũ trong C
```

hoặc theo C++ có thể bỏ qua từ khóa struct:

```
identifier_tag list_of_var ; // trong C++
```

Các biến được khai báo cũng có thể đi kèm khởi tạo:

```
identifier_tag var1 = {created_value }, var2, ... ;
```

Ví dụ khai báo kiểu Student:

```
struct Student
{
    char name[30];
    int birth_year;
    double mark;
} monitor = { "Nguyen Van "Anh", 1992, 8.7}, x;
```

Trong khai báo trên, ta đã đồng thời khai báo 2 biến kiểu **Student** là **monitor** (lớp trưởng) và **x**, trong đó **x** chưa được khởi tạo và **monitor** được khởi tạo với họ tên là Nguyễn Văn Anh, sinh năm 1992 và điểm tốt nghiệp là 8.7. Khi cần khai báo thêm biến có kiểu **Student**, có thể theo cú pháp, ví dụ:

```
Student vice_monitor, K58[60], y;
```

Trong khai báo trên, **vice_monitor**, **y** là các biến đơn, **K58** là một mảng mà các thành phần của nó là các sinh viên, ví dụ dùng để biểu diễn dữ liệu của một lớp học.

Ưu điểm của kiểu cấu trúc là dùng để biểu diễn tập các giá trị khác kiểu, tuy nhiên với tập giá trị cùng kiểu hiển nhiên vẫn có thể được biểu diễn bằng kiểu cấu trúc và trong nhiều trường hợp ý nghĩa của đối tượng sẽ rõ ràng hơn so với khi biểu diễn bởi kiểu mảng.

Ví dụ, chương trước ta đã từng biểu diễn phân số bởi mảng 2 thành phần với ngầm định thành phần thứ nhất là tử và thành phần thứ hai là mẫu. Dữ liệu phân số này cũng có thể được biểu diễn bởi cấu trúc như sau:


```
struct Fraction
{
    int numerator ;
    int denominator ;
} ;
```

Với cách biểu diễn này, các thành phần tử và mẫu của phân số đều đã được đặt tên thay vì phải ngầm định như cách biểu diễn dạng mảng. Tương tự, một ngày tháng có thể được khai báo :

```
struct Date {
    int day ;
    int month ;
    int year ;
} holiday = { 1, 5, 2000 } ;
```

một biến `holiday` cũng được khai báo kèm cùng kiểu này và được khởi tạo bởi bộ số 1, 5, 2000. Các giá trị khởi tạo này lần lượt gán cho các thành phần theo đúng thứ tự trong khai báo, tức `day = 1`, `month = 5` và `year = 2000`.

Vì các thành phần `day`, `month`, `year` cùng kiểu `int` nên cũng giống khai báo các loại biến khác, chúng có thể được gộp trên một dòng (vẫn giữ đúng thứ tự):

```
struct Date
{
    int day, month, year ;
} holiday = { 1, 5, 2000 } ;
```

Kiểu cấu trúc cũng có thể chứa thành phần là kiểu cấu trúc. Ví dụ, trong kiểu sinh viên được khai báo ở trên, ta có thể thay trường năm sinh (`birth_year`) bởi trường có chứa cả ngày tháng năm sinh như:

```
struct Student
{
    char name[30];
    Date birthday;
    double mark;
} monitor = { "Nguyen Van Anh", {1, 1, 1992}, 8.7}, x;
```

thành phần `birthday` của `Student` có kiểu `Date` cũng là một cấu trúc, khi đó cách khởi tạo giá trị cho biến `monitor` cũng đã được thay đổi cho phù hợp từ việc chỉ khởi tạo 1992 cho thành phần `birth_year` trong khai báo cũ thành `{1, 1, 1992}` cho trường `birthday` trong khai báo mới.

Kiểu cấu trúc `Class` dưới đây dùng chứa thông tin về một lớp học gồm tên lớp, và danh sách sinh viên cũng là một ví dụ minh họa cho việc kết hợp các loại kiểu khác nhau trong cùng một kiểu.

```
struct Class {
    char name[10],           // xâu kí tự
    Student list[MAX];       // mảng cấu trúc
} ;
```

Tên các thành phần được phép trùng nhau trong các cấu trúc khác nhau, ví dụ `name` xuất hiện trong cả hai cấu trúc `Student` và `Class`.

Giống các biến mảng, để làm việc với một biến cấu trúc, trong một số thao tác chúng ta phải thực hiện trên từng thành phần của chúng. Ví dụ để vào/ra một biến cấu trúc ta phải viết câu lệnh vào/ra cho từng thành phần như trong ví dụ trên.

Tuy nhiên, may mắn hơn so với cách làm việc của mảng, đó là 2 cấu trúc được phép gán (=) giá trị cho nhau một cách trực tiếp, trong khi mảng chỉ có thể gán từng thành phần. Phép gán trực tiếp này cũng tương đương với việc gán từng thành phần của cấu trúc. Ví dụ:

```
Student monitor = { "NVA", { 1, 1, 1992 }, 5.0 };
Student good_stu;
good_stu = monitor;
```

Chú ý: không gán bộ giá trị cụ thể cho biến cấu trúc. Cách gán này chỉ thực hiện được khi khởi tạo. Ví dụ:

```
Student good_stu;
good_stu = { "NVA", { 1, 1, 1992 }, 5.0 };           // sai
```

```
1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int main()
6 {
7     const int NUM_OF_STUDENTS = 3;
8     struct Date
9     {
10         int day, month, year ;
11     };
12     struct Student
13     {
14         char name[30];
15         Date birthday;
16         double mark;
17     };
18     Student monitor = { "Bill Gate", { 1, 11, 2001 }, 5.0 };
19     Student other_student;
20     other_student = monitor;
21     other_student.birthday.year = 2002;           // Dat lai nam sinh
22
23     cout << "Name : " << other_student.name << endl;
24     cout << "Birthday: " << other_student.birthday.day << "/" << other_student.
25         birthday.month << "/" << other_student.birthday.year << endl;
26     cout << "Mark: " << other_student.mark << endl;
27
28     return 0;
29 }
```

Hình 6.1: Gán các biến cấu trúc.

Chú ý: Mặc dù hai cấu trúc có thể được gán cho nhau (=) nhưng chúng lại không so sánh (==) được với nhau, trừ phi ta phải so sánh từng thành phần của chúng.

6.1.2 Hàm và cấu trúc

Đối của hàm là cấu trúc

Một cấu trúc có thể được sử dụng để làm đối của hàm dưới các dạng sau đây:

- Là một biến cấu trúc, khi đó giá trị truyền là một cấu trúc.
- Là một tham chiếu cấu trúc, giá trị truyền là một cấu trúc.
- Là một con trỏ cấu trúc, giá trị truyền là địa chỉ của một cấu trúc.

Dạng truyền theo dẫn trỏ sẽ được trình bày trong chương 7 của giáo trình.

Nhìn chung, một cấu trúc là kiểu dữ liệu lớn, chiếm nhiều bộ nhớ và vì vậy mất nhiều thời gian sao chép nên dạng truyền theo tham chiếu được sử dụng thường xuyên hơn truyền theo giá trị. Để tránh thay đổi giá trị biến ngoài thường ta khai báo tham đối kiểu tham chiếu dưới dạng `const`.

Ví dụ sau đây cho phép tính chính xác khoảng cách của 2 ngày tháng bất kỳ cũng như thứ của một ngày tháng. Về mặt dữ liệu, kiểu ngày tháng (`Date`) sẽ được khai báo dạng toàn cục. Mảng hằng `int NUM_DAYS[13]` cung cấp số ngày cố định của các tháng (`NUM_DAYS[i]` là số ngày của tháng `i`), tháng 2 vẫn xem là 28 ngày, nếu gặp năm nhuận số ngày của tháng 2 được cộng thêm 1.

Chương trình gồm các hàm:

- `int bissextile_year(int year)`: Hàm trả lại 1 nếu đối `year` là năm nhuận và 0 nếu ngược lại. Năm nhuận là năm chia hết cho 4 nhưng không chia hết cho 100, tuy nhiên nếu chia hết cho 400 thì năm lại nhuận.
- `int num_days_of_month(int month, int year)`: Hàm trả lại số ngày của tháng (`month`) trong năm `year`, đơn giản hàm lấy dữ liệu từ mảng `NUM_DAYS` cho sẵn và nếu `month = 2` và `year` là năm nhuận thì cộng thêm 1.
- `long DtoN(Date date)` (Date to Numeric). Hàm chuyển tương đương một ngày tháng thành một số nguyên dài là số ngày tính từ 1/1/1 đến `date`. Về mặt thuật toán, hàm sẽ tính số ngày đã qua từ năm 1 cho đến năm `year - 1`. Mỗi năm được cộng thêm 365 hoặc 366 ngày. Tiếp theo cộng thêm số ngày từ tháng 1 đến tháng `month - 1` của năm hiện tại (số ngày của từng tháng được lấy thông qua hàm `num_days_of_month`) và cuối cùng cộng thêm số ngày hiện tại (`day`).
- `Date NtoD(long n)` (Numeric to Date). Hàm chuyển tương đương một số nguyên thành ngày tháng (quan niệm số nguyên là số ngày tính từ 1/1/1 đến số ngày cần chuyển). Về mặt thuật toán, hàm sẽ trừ dần 365 hoặc 366 ngày để tính tăng lên một năm, số ngày còn lại (bé hơn 365 hoặc 366) sẽ được chuyển sang tháng và ngày theo cách tương tự.
- `long Distance_Dates(Date date1, Date date2)`: Hàm trả lại khoảng cách giữa hai ngày tháng, đơn giản là: `DtoN(date1) - DtoN(date2)`;
- Để tính thứ của một `date`, hàm chọn một ngày đã biết thứ (ví dụ ngày 1/1/2000 đã biết là thứ bảy) và lấy khoảng cách với `date`. Do thứ được lặp lại theo chu kỳ 7 ngày nên nếu khoảng cách này chia hết cho 7 (phần dư là 0) thì thứ của `date` cũng chính là thứ của ngày đã biết, hoặc dựa trên phần dư của khoảng cách với 7 ta có thể suy đoán ra thứ của `date`. Trong hình là chương trình và output.

```
1 #include <iostream>
2 using namespace std;
3
```

```

4 // number of days of months
5 const int NUM_DAYS[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
6 struct Date
7 {
8     int day, month, year ;
9 };
10
11 //----- Func. returns 1 if year is bissextile_year and 0 if not
12 int Bissextile_year(int year)
13 {
14     return (year%4 == 0 && year%100 != 0 || year%400 == 0)? 1 : 0;
15 }
16
17 //----- Func. returns number of days of any month
18 //----- (plus 1 if year is bissextile)
19 int Num_Days_Of_Month(int month, int year)
20 {
21     return NUM_DAYS[month] + ((month == 2) ? Bissextile_year(year) : 0);
22 }
23
24 //----- Func. returns total number of days from 1/1/1 to the date
25 long DtoN(Date date) // DtoN: Date to Numeric
26 {
27     long res = 0;
28     for (int index = 1; index < date.year; index++)
29         res += 365 + Bissextile_year(index);
30     for (int index = 1; index < date.month; index++)
31         res += num_days_of_month(index, date.year);
32     res += date.day;
33     return res;
34 }
35
36 //----- Func. returns a date that corresponds to a numeric
37 Date NtoD(long num_days) // NtoD: Nummeric to Date
38 {
39     Date res;
40     res.year = 1;
41     while (num_days > 365 + Bissextile_year(res.year))
42     {
43         num_days -= 365 + Bissextile_year(res.year);
44         res.year++;
45     }
46     res.month = 1;
47     while (num_days > num_days_of_month(res.month, res.year))
48     {
49         num_days -= num_days_of_month(res.month, res.year);
50         res.month++;
51     }
52     res.day = num_days;
53     return res;
54 }
55
56 //----- Func. returns number of days from date1 to date2
57 long Distance_Dates(Date date1, Date date2)

```

```

58 {
59     return DtoN(date1) - DtoN(date2);
60 }
61
62 //----- Func. returns day of week of any date
63 void DoW(Date date, char dow[])          // DoW: Day of week
64 {
65     Date curdate = {1, 1, 2000};          // the date 1/1/2000 is Sartuday
66     long dist = Distance_Dates(date, curdate);
67     int odd = dist % 7;
68     if (odd < 0) odd += 7;
69     switch (odd) {
70         case 0: strcpy(dow, "Sartuday"); break;
71         case 1: strcpy(dow, "Sunday"); break;
72         case 2: strcpy(dow, "Monday"); break;
73         case 3: strcpy(dow, "Tuesday"); break;
74         case 4: strcpy(dow, "Wednesday"); break;
75         case 5: strcpy(dow, "Thursday"); break;
76         case 6: strcpy(dow, "Friday"); break;
77     }
78 }
79
80 ///////////////////////////////////////////////////////////////////
81 int main()
82 {
83     Date your_birthday, today;
84     char dow_birthday[10], dow_today[10];
85     cout << "What date is your birthday ? (dd/mm/yyyy) " ;
86     cin >> your_birthday.day >> your_birthday.month >> your_birthday.year ;
87     cout << "And today ? (dd/mm/yyyy) " ;
88     cin >> today.day >> today.month >> today.year ;
89     DoW(your_birthday, dow_birthday);
90     DoW(today, dow_today);
91     cout << "You were born on " << dow_birthday << ". Today is " << dow_today << endl
92         ;
93     cout << "You were born " << Distance_Dates(today, your_birthday) << " days ago."
94         << endl;
95     return 0;
96 }

```

Hình 6.2: Khoảng cách giữa 2 ngày.

Giá trị của hàm là cấu trúc

Khác với kiểu mảng, một hàm có thể trả lại giá trị là một cấu trúc. Từ đó, ta có thể viết lại chương trình tính cộng, trừ, nhân chia hai phân số, trong đó mỗi phép toán là một hàm với 2 đối cấu trúc là 2 phân số và kết quả trả lại của hàm là kết quả của phép toán cũng là một cấu trúc phân số như chương trình dưới đây.

```

1 #include <iostream>
2 using namespace std;
3
4 struct Fraction
5 {

```



```

60     Display(a, b, c, '+');
61     c = Sub(a, b);                      // Compute and display a - b
62     Display(a, b, c, '-');
63     c = Product(a, b);                  // Compute and display a * b
64     Display(a, b, c, '*');
65     c = Divide(a, b);                   // Compute and display a / b
66     Display(a, b, c, ':');
67
68     return 0;
69 }

```

Hình 6.3: Phân số.

Với những kiến thức được trang bị đến thời điểm này, người đọc đã có thể viết được những chương trình nhỏ tương đối hoàn chỉnh như bài toán quản lý sinh viên trong mục tiếp theo.

6.1.3 Bài toán Quản lý sinh viên (QLSV)

Bài toán QLSV được trình bày ở đây như một ví dụ nhỏ về việc tổng hợp các đặc trưng lập trình và kiến thức về NNLT C/C++ đến thời điểm này. Bài toán đặt ra việc quản lý một danh sách sinh viên với các chức năng chủ yếu khi làm việc với danh sách là: Tạo, Xem, Xóa, Sửa, Bổ sung, Chèn, Sắp xếp và Thống kê. Trong mục này, chương trình được xây dựng để minh họa dữ liệu có kiểu cấu trúc (sinh viên) và mảng cấu trúc (danh sách sinh viên). Các phiên bản với kiểu lớp được trình bày trong mục 6.2 và với danh sách liên kết sẽ được trình bày trong chương 7. Mã đầy đủ của các phiên bản này được cho trong các **phụ lục A.1, A.2, A.3**.

Cấu trúc dữ liệu

Thông tin về sinh viên được tổ chức dưới dạng cấu trúc và Danh sách sinh viên là một mảng cấu trúc như khai báo:

```

1  const int MAXSIZE_OF_LIST = 60;
2  struct Date { int day, month, year ; };
3  struct Student { char name[30]; Date birthday; int sex; double mark; };
4  Student List[MAXSIZE_OF_LIST] ;           // Danh sach sinh vien
5  int num_students;                         // So luong sinh vien

```

Hình 6.4: Khai báo cấu trúc sinh viên.

Vì danh sách sinh viên là dữ liệu dùng chung của các hàm nên các khai báo trên được đặt ra bên ngoài tất cả các hàm (toàn cục).

Chức năng

Mỗi chức năng được thực hiện thông qua một hàm của chương trình. Chương trình gồm các hàm sau:

```

1  /* Nhóm ham chinh, lam viec voi danh sach */
2  void Make_List();                      // Tao danh sach
3  void Display_List();                   // Hien danh sach
4  void Update_List();                    // Cap nhat danh sach
5  void Insert_List();                     // Chem them vao danh sach
6  void Append_List();                     // Bo sung vao cuoi danh sach
7  void Remove_List();                     // Xoa khoi danh sach
8  void Sort_List();                       // Sap xep danh sach

```

```

9 void Count_List(); // Thống kê danh sách
10 /* Nhóm hàm làm việc với một sinh viên */
11 Student New_Student(); // Tạo một sinh viên mới
12 void Display_Student(Student x); // Hiển thị một sinh viên
13 void Update_Student(Student &x); // Cập nhật sinh viên
14 /* Nhóm hàm phục vụ */
15 void Set_List(); // Dùng để test chương trình
16 void Swap(Student &x, Student &y);
17 void Get_Firstname(const char name[], char firstname[]); // Lấy tên của họ tên
18 void Sort_List_by_Name(); // Sắp xếp tăng theo họ tên
19 void Sort_List_by_Mark(); // Sắp xếp giảm theo điểm

```

Hình 6.5: Danh sách hàm của chương trình QLSV.

Do danh sách sinh viên được khai báo toàn cục nên hầu hết các hàm thao tác trực tiếp với danh sách, không có đối và không cần giá trị trả lại.

Giao diện

Hàm main() tạo một menu cho phép NSD chọn 1 trong 8 chức năng hoặc chọn 0 để chấm dứt chương trình.

```

1 int main()
2 {
3     Set_List();
4     int choice;
5     do {
6         system("CLS");
7         cout << "\n===== MAIN MENU (Struct ver.) =====\n\n";
8         cout << "1: Make List of students \n";
9         cout << "2: Display List of students \n";
10        cout << "3: Update a student of the List \n";
11        cout << "4: Insert a student to the List \n";
12        cout << "5: Append a student to the List \n";
13        cout << "6: Remove a student from the List \n";
14        cout << "7: Sort List of students \n";
15        cout << "8: Count number of Male/Female students \n";
16        cout << "0: Exit \n";
17        cout << "\n===== \n";
18        cout << "\nYour choice ? " ;
19        cin >> choice ; cin.ignore();
20        system("CLS");
21        switch (choice)
22        {
23            case 1: Make_List() ; break;
24            case 2: Display_List() ; break;
25            case 3: Update_List() ; break;
26            case 4: Insert_List() ; break;
27            case 5: Append_List() ; break;
28            case 6: Remove_List() ; break;
29            case 7: Sort_List() ; break;
30            case 8: Count_List() ; break;
31        }
32    } while (choice) ;
33    return 0 ;
34 }

```


Hình 6.6: Hàm main() của chương trình QLSV.

Nếu NSD chọn 1 trên MAIN MENU, hàm main() sẽ gọi hàm Make_List() để thực hiện chức năng tạo lập danh sách. Để nhập danh sách, hàm lập vòng lặp từ 1 đến số sinh viên (biến num_students), mỗi lần lặp hàm gọi đến hàm New_student() để nhập dữ liệu cho một sinh viên. New_student() có giá trị trả lại là cấu trúc sinh viên vừa nhập. Dưới đây là mã của hàm Make_List() và New_student().

```

1 void Make_List()
2 {
3     cout << "Enter the number of students: ";
4     cin >> num_students;
5     for (int index = 0; index < num_students; index++)
6     {
7         cout << "Enter data values of student #" << index+1 << ":\n";
8         List[index] = New_student() ;
9     }
10 }
11
12 Student New_Student()
13 {
14     Student x ;
15     fflush(stdin) ;
16     cout << "    Full name: "; cin.getline(x.name, 30) ;
17     cout << "    Birthday: "; cin >> x.birthday.day >> x.birthday.month >> x.birthday
18         .year ;
19     cout << "    Sex (0: Male, 1: Female): "; cin >> (x.sex) ;
20     cout << "    Mark: "; cin >> (x.mark) ;
21     return x;
22 }

```

Hình 6.7: Hàm Make_List() và New_student().

Chức năng xem danh sách được thực hiện bởi hàm Display_List() bằng cách duyệt từ 1 đến số sinh viên và mỗi lần lặp sẽ gọi hàm Display_student() để hiện thông tin về sinh viên này.

```

1 void Display_List()
2 {
3     int index;
4     cout << "\nLIST OF STUDENTS" << endl ;
5     for (int index = 0; index < num_students; index++)
6     {
7         cout << index+1 << ". " ;
8         Display_student(List[index]) ;
9     }
10 }
11
12 void Display_Student(Student x)
13 {
14     cout << x.name << "\t" ;
15     cout << setw(2) << x.birthday.day << "/" << setw(2) << x.birthday.month << "/" <<
16         setw(2) << x.birthday.year << "\t" ;
17     if (x.sex == 0)

```

```

17     cout << "Male" << "\t" ;
18     else
19         cout << "Female" << "\t" ;
20     cout << x.mark;
21     cout << endl;
22 }

```

Hình 6.8: Hàm Display_List() và Display_student().

Chức năng sửa Update_List() yêu cầu NSD nhập sinh viên cần sửa (thông qua số thứ tự) và sau đó gọi hàm Update_Student() để sửa thông tin cho sinh viên này. Do kích thước của một cấu trúc thường là lớn nên để tiết kiệm bộ nhớ và thời gian sao chép, đối của hàm Update_Student() được khai báo dưới dạng tham chiếu. Để đảm bảo an toàn, tránh nhầm lẫn, hàm chỉ cho NSD sửa trên bản sao của sinh viên, sau khi NSD xác nhận thông tin đã sửa là chính xác thì hàm mới cập nhật bản sao này vào danh sách.

```

1 void Update_List()
2 {
3     int order;
4     cout << "Select the order of student: " ;
5     cin >> order ; cin.ignore();
6     Update_Student(List[order-1]) ;
7 }
8
9 void Update_Student(Student &x)
10 {
11     Student copy = x;
12     Display_Student(copy);
13     int choice;
14     do {
15         cout << "1: Name" << endl ;
16         cout << "2: Birthday" << endl ;
17         cout << "3: Sex" << endl ;
18         cout << "4: Mark" << endl ;
19         cout << "0: Exit" << endl ;
20         cout << "Select member to update ? " ;
21         cin >> choice ; cin.ignore();
22         switch (choice)
23         {
24             case 1: cout << "Enter new name: " ; cin.getline(copy.name, 30) ; break;
25             case 2: cout << "Enter new birthday: " ; cin >> copy.birthday.day >> copy
                .birthday.month >> copy.birthday.year ; break;
26             case 3: cout << "Enter new sex: " ; cin >> copy.sex ; break;
27             case 4: cout << "Enter new mark: " ; cin >> copy.mark ; break;
28         }
29     } while (choice) ;
30     int sure ;
31     cout << "Are you sure for updating (1: sure, 0: not sure) ? " ; cin >> sure ;
32     if (sure)
33     {
34         x = copy ;
35         cout << "\nInformation of student was updated\n";
36     }
37     else cout << "\nUpdating cancelled\n";

```

```

38
39 }

```

Hình 6.9: Hàm Update_List() và Update_Student().

Hàm Insert_List() cho phép chèn một sinh viên vào danh sách tại vị trí pos bằng cách đẩy các sinh viên từ vị trí này tiến lên một ô để dành ô trống pos cho sinh viên mới vừa tạo (bởi hàm New_student()).

```

1 void Insert_List()
2 {
3     cout << "Enter data of new student:\n" ;
4     Student new_student = New_Student() ;
5     int pos;
6     cout << "Enter position of new student: " ; cin >> pos;
7     for (int index = num_students-1; index > pos; index--)
8         List[index + 1] = List[index];
9     List[pos-1] = new_student;
10    num_students++;
11    cout << "The student is inserted to the List\n" ;
12
13 }

```

Hình 6.10: Hàm Insert_List().

Để bổ sung vào cuối danh sách một sinh viên mới ta gọi hàm Append_List(), hàm này sẽ gọi đến hàm New_student() để tạo sinh viên mới trước khi ghép vào danh sách.

```

1 void Append_List()
2 {
3     cout << "Enter data of new student:\n" ;
4     List[num_students] = New_student() ;
5     num_students ++ ;
6     cout << "The student is appended to the List\n" ;
7
8 }

```

Hình 6.11: Hàm Append_List().

Chức năng xóa được thực hiện qua hàm Remove_List(), hàm này cho phép NSD chọn sinh viên cần xóa thông qua số thứ tự của sinh viên trong danh sách. Thực chất của việc xóa là dồn các sinh viên phía sau sinh viên cần xóa về phía trước một ô để lấp đầy vị trí của sinh viên bị xóa.

```

1 void Remove_List()
2 {
3     Display_List();
4     int order;
5     cout << "Select the order of student for removing: " ;
6     cin >> order ; cin.ignore();
7     for (int index = order; index < num_students; index++)
8         List[index - 1] = List[index];
9     num_students -- ;
10    cout << "The student is removed from the List\n" ;
11
12 }

```

Hình 6.12: Hàm Remove_List().

Hàm Sort_List() và hai hàm “con” của nó là Sort_List_by_Name() và Sort_List_by_Mark() thực hiện chức năng sắp xếp tăng dần theo họ tên và giảm dần theo điểm số của sinh viên bằng thuật toán sắp xếp chọn. Hàm cho phép NSD chọn 1 trong 2 cách sắp xếp trên. Để trao đổi 2 cấu trúc, hàm gọi đến hàm swap(). Để sắp theo tên, hàm gọi đến hàm phục vụ Get_firstname(const char name[], char firstname[]) hàm này trả lại xâu tên của sinh viên vào tham đối firstname của hàm. Thực chất, hàm sắp xếp theo tên và nếu 2 tên trùng nhau hàm sẽ sắp tiếp theo họ bằng cách so sánh tên + họ của 2 sinh viên với nhau. Trong chương trình ta ngầm định họ tên được viết dưới dạng chuẩn: không chứa dấu cách ở hai đầu họ tên.

```

1 void Sort_List()
2 {
3     int choice;
4     cout << endl;
5     cout << "-----\n" ;
6     cout << "1: Sort List increasing by name" << endl ;
7     cout << "2: Sort List decreasing by mark" << endl ;
8     cout << "-----\n" ;
9     cout << "Your choice ? " ;
10    cin >> choice ; cin.ignore();
11    switch (choice)
12    {
13        case 1: Sort_List_by_Name() ; break;
14        case 2: Sort_List_by_Mark() ; break;
15    }
16    if (choice == 1)
17        cout << "List of students is sorted increasing by name\n";
18    else if (choice == 2)
19        cout << "List of students is sorted decreasing by mark\n" ;
20    else
21        cout << "List of students is not sorted\n" ;
22
23 }
24
25 void Sort_List_by_Name()    // increasing on name
26 {
27     int i, j;
28     for (i = 0; i < num_students - 1; i++)
29         for (j = i+1; j < num_students; j++)
30         {
31             char fn1[40], fn2[40];
32             Get_Firstname(List[i].name, fn1) ; strcat(fn1, List[i].name) ;
33             Get_Firstname(List[j].name, fn2) ; strcat(fn2, List[j].name) ;
34             if (strcmp(fn1, fn2) > 0)
35                 Swap(List[i], List[j]) ;
36         }
37 }
38
39 void Sort_List_by_Mark()    // decreasing on mark
40 {
41     int i, j;

```

```

42     for (i = 0; i < num_students - 1; i++)
43         for (j = i+1; j < num_students; j++)
44             if (List[i].mark < List[j].mark)
45                 Swap(List[i], List[j]) ;
46 }
47
48 void Get_Firstname(const char name[], char firstname[])
49 {
50     int index;
51     for (index = strlen(name); name[index] != ' '; index -- ) ;
52     int len = strlen(name) - index ;
53     strncpy(firstname, name + index + 1, len);
54     firstname[len] = '\0' ;
55 }
56
57 void Swap(Student &x, Student &y)
58 {
59     Student tmp;
60     tmp = x; x = y; y = tmp;
61 }

```

Hình 6.13: Hàm Sort_List() và các hàm phụ trợ.

Hàm Count_List() cho một thống kê đơn giản là tính số lượng sinh viên nam và số lượng sinh viên nữ.

```

1 void Count_List()
2 {
3     int num_male, num_female;
4     num_male = num_female = 0;
5     for (int index = 0; index < num_students; index++)
6         if (List[index].sex == 0) num_male++;
7         else num_female++;
8     cout << endl;
9     cout << "Number of Male is: " << num_male << endl;
10    cout << "Number of Female is: " << num_female << endl;
11    cout << "Total is: " << num_students << endl;
12
13 }

```

Hình 6.14: Hàm Count_List().

Và cuối cùng, để tránh tình nhầm chán vì phải tạo lại danh sách mỗi lần chạy chương trình, ta có thể tự động nạp trước một danh sách cố định nào đó bằng cách gọi hàm Set_List() đầu tiên nhất trong main(). Điều này không ảnh hưởng đến hoạt động của chương trình vì nếu cần ta có thể tạo lại danh sách mới bằng Make_List(), danh sách này sẽ ghi đè lên thông tin cố định của Set_List().

```

1 void Set_List()
2 {
3     Student x = { "Tran Thanh Son", { 2, 3, 1985 }, 0, 8.4 }; List[0] = x ;
4     Student y = { "Nguyen Thi Hanh", { 12, 4, 1977 }, 1, 6.5 }; List[1] = y ;
5     Student z = { "Le Nguyen Hanh", { 6, 11, 1991 }, 0, 5.2 }; List[2] = z ;
6     Student t = { "Cao Thuy Linh", { 28, 8, 2001 }, 1, 9.5 }; List[3] = t ;
7     num_students = 4 ;

```

```

8 }
9
10 void Make_List()
11 {
12     cout << "Enter the number of students: ";
13     cin >> num_students;
14     for (int index = 0; index < num_students; index++)
15     {
16         cout << "Enter data values of student #" << index+1 << ":\n";
17         List[index] = New_student() ;
18     }
19 }

```

Hình 6.15: Hàm Set_List().

Với thứ tự của danh sách sinh viên được nhập tự động (từ hàm Set_List()), ví dụ chọn chức năng sắp xếp ta sẽ có kết quả như bảng dưới:

Bảng 6.16: Sắp xếp sinh viên theo tên và theo điểm

Thứ tự gốc		Tăng dần theo tên		Giảm dần theo điểm	
Tran Thanh Son	8.4	Le Nguyen Hanh	5.2	Cao Thuy Linh	9.5
Nguyen Thi Hanh	6.5	Nguyen Thi Hanh	6.5	Tran Thanh Son	8.4
Le Nguyen Hanh	5.2	Cao Thuy Linh	9.5	Nguyen Thi Hanh	6.5
Cao Thuy Linh	9.5	Tran Thanh Son	8.4	Le Nguyen Hanh	5.2

Mã nguồn của chương trình hoàn chỉnh được cho trong phụ lục A.1.

Trong thiết kế chương trình Quản lý sinh viên, ta đã cố gắng chia nhỏ chương trình càng nhỏ càng tốt theo triết lý của thiết kế top-down và chia-để-trị. Cụ thể, có thể đưa toàn bộ mã của hàm New_student() vào trong hàm Make_List(), tuy nhiên như vậy một lần nữa ta lại phải đưa mã này vào các hàm Append_List() và Insert_List() vì các hàm này cũng gọi đến New_student(). Đó là lý do để nhập thông tin cho một sinh viên được xây dựng thành một hàm riêng New_student(). Tương tự, hàm Display_student() cũng được tách riêng vì nó được gọi bởi Display_List() và Update_List(). Và mặc dù hàm Update_student() chỉ được gọi duy nhất trong Update_List() nhưng ta cũng viết riêng thành một hàm hoàn chỉnh, cách thiết kế này có lợi khi cần thay đổi, mở rộng cấu trúc Student ta chỉ cần thay đổi, bổ sung ở các hàm “con” này và nó độc lập hoàn toàn với những hàm gọi đến nó.

6.2 Kiểu dữ liệu trừu tượng bằng lớp (class)

Phần này sẽ trình bày về lớp là kiểu dữ liệu mở rộng của kiểu cấu trúc. Lớp đóng vai trò trung tâm trong kỹ thuật lập trình hướng đối tượng, một kỹ thuật lập trình phát triển so với kỹ thuật lập trình cấu trúc. Những đặc trưng đặc sắc của kỹ thuật này sẽ dần được trình bày đầy đủ hơn trong các chương sau của giáo trình.

Hãy quay lại kiểu dữ liệu cấu trúc ở tiết trước và xem 2 cấu trúc trong cùng một chương trình, ví dụ cấu trúc Date và cấu trúc Student. Trong chương trình trên, nếu cần in ngày tháng

ta sẽ viết hàm `void Display(Date date)` hoặc cần in thông tin về sinh viên ta viết hàm `void Display(Student student)`, như vậy trong chương trình có hai hàm cùng chung mục đích, cùng tên và cùng cách thức làm việc, chỉ khác nhau ở chỗ mỗi hàm làm việc trên tập các dữ liệu của riêng mình. Rõ ràng, không thể dùng hàm `Display(Date date)` để in nội dung của một sinh viên và ngược lại. Cũng tương tự, liên quan đến `Date` sẽ có hàm `Distance_Date()` để tính khoảng cách của 2 ngày tháng, còn đối với `Student` thì không. Tóm lại, mỗi tập hợp dữ liệu đều có một tập các hàm xử lý riêng của nó.

Từ nhận xét trên, dữ liệu nào đi theo hàm xử lý đó, sẽ được “đóng” chung vào một “gói” và gói này ta gọi là lớp (class). Có nghĩa hàm `Display(Date date)` sẽ đi cùng cấu trúc `Date` thành một lớp và hàm `Display(Student student)` sẽ đi cùng cấu trúc `Student` thành lớp khác.

Như vậy, lớp là một cấu trúc ngoài thành phần dữ liệu (được gọi là các biến thành viên - member variables) còn thêm thành phần là các hàm xử lý những dữ liệu của lớp này. Các hàm thành phần này còn được gọi là các hàm thành viên hay phương thức thành viên (member functions, member methods). Ý tưởng gắn chung 2 loại thành phần này vào cùng một kiểu dữ liệu (cùng trở thành thành viên của lớp) được gọi là đóng gói (encapsulation).

Từ các thảo luận trên, ta xây dựng kiểu dữ liệu mới với khai báo sau.

6.2.1 Khai báo lớp

```
class class_identifier
{
    member methods;
    member variables;
};
```

Cũng giống như cấu trúc, ta lưu ý (đừng quên) kết thúc của khai báo là dấu chấm phẩy. Ngoài ra, trước dấu chấm phẩy này ta cũng có thể khai báo kèm theo các biến và kể cả khởi tạo. Trong một lớp, thứ tự khai báo của các thành viên (variables và methods) là không quan trọng, tuy nhiên có nhiều lý do để ta ưu chuộng cách khai báo các phương thức (methods) trước sau đó mới đến khai báo biến (variables).

```
1 #include <iostream>
2 using namespace std;
3
4 class Date
5 {
6 public:
7     void Display();           // method
8     int day, month, year ;    // variable
9 };
10
11 class Student
12 {
13 public:
14     void Display();           // method
15 private:
16     char name[30];            // variable
17     Date birthday;            // variable
18     int sex;                  // variable
```

```

19  double mark;                // variable
20  };

```

Hình 6.17: Ví dụ đơn giản về khai báo 2 lớp Date và Student.

Chú ý đối với `struct Date`, hàm `void Display(Date date)` định nghĩa bên ngoài cấu trúc và có đối kèm theo để biết hàm cần hiển thị dữ liệu của biến nào khi được gọi. Còn ở đây, trong `class Date`, hàm `void Display()` là hàm không đối. Vậy khi hàm được gọi nó sẽ hiển thị dữ liệu lấy từ đâu? Điều này sẽ được giải thích dần về sau. Hiển nhiên, không phải tất cả các hàm trong lớp đều là không đối.

Trong các khai báo trên, phương thức `Display()` chưa được định nghĩa. Định nghĩa của các phương thức có thể đặt ngay bên trong lớp lúc khai báo hoặc có thể đặt bên ngoài lớp. Vì một lớp có thể có rất nhiều phương thức nên việc đặt tất cả các định nghĩa này vào bên trong một lớp sẽ làm cho mã của lớp rất dài, khó theo dõi. Do vậy, thông thường các phương thức chỉ được khai báo bên trong còn định nghĩa của chúng sẽ đặt ở bên ngoài.

Khi định nghĩa phương thức bên ngoài lớp, để tránh nhầm lẫn (vì các phương thức của các lớp khác nhau có thể trùng tên) ta cần chỉ định phương thức này thuộc về lớp nào bằng cách chỉ ra tên lớp và dấu `::` trước tên phương thức. Dấu `::` (hai dấu hai chấm liền nhau) là kí hiệu của phép toán chỉ định phạm vi (scope resolution operator). Ví dụ:

```

1  void Date :: Display()
2  {
3      cout << day << "/" << month << "/" << year ;
4  }
5
6  void Student :: Display()
7  {
8      cout << name << "\t" ;
9      if (sex == 0) cout << "Male" << "\t" ; else cout << "Female" << "\t" ;
10     cout << mark << endl;
11 }

```

Hình 6.18: Định nghĩa phương thức.

Trong các định nghĩa này ta có vài điểm lưu ý:

- Hai hàm `Display()` là khác nhau, một của lớp `Date` (với toán tử phạm vi `Date ::`) và một của `Student` (`Student ::`)
- Các hàm đều không có đối, dữ liệu được sử dụng trong hàm được lấy từ chính các thành viên của lớp đó.
- Việc khai báo biến thành viên là lớp khác (`Date` trong `Student`) là được phép (tức trong lớp có chứa lớp) tuy nhiên, tại thời điểm này ta chưa bàn đến cách sử dụng. Do vậy, trong phương thức `Student :: Display()` ở trên tạm thời ta sẽ không in dữ liệu của biến thành viên `birthday`.

6.2.2 Sử dụng lớp

Đối tượng

Cũng giống các kiểu khác để khai báo một biến thuộc lớp ta dùng cú pháp, ví dụ:


```
Date holiday, birthday ;
```

dùng để khai báo 2 biến `holiday` và `birthday` có kiểu lớp `Date`. Trong lập trình hướng đối tượng, các biến kiểu lớp được gọi là đối tượng (object) và từ giờ về sau, ta sẽ dùng từ này để nói về biến kiểu lớp.

Tương tự cấu trúc, để truy cập đến các thành viên của lớp ta sử dụng phép toán chấm (dot operator). Ví dụ:

```
holiday.day = 1 ;
holiday.month = 5 ;
holiday.year = 2015 ;
holiday.Display() ;
```

Ba dòng lệnh đầu dùng để gán giá trị ngày 1/5/2015 cho đối tượng `holiday`. Dòng lệnh thứ 4 được hiểu là `holiday` truy cập đến hoặc gọi đến phương thức `Display()`, khi đó phương thức `Display()` sẽ thực hiện và các dữ liệu liên quan đến các biến trong phương thức sẽ được lấy từ `holiday`. Từ đó câu lệnh `holiday.Display()` sẽ in nội dung của `holiday`, tức 1/5/2015 ra màn hình. Dưới đây là ví dụ tổng hợp các ý trên.

```
1 #include <iostream>
2 using namespace std;
3
4 class Date
5 {
6 public:
7     void Display();           // method
8     int day, month, year ;    // variable
9 };
10
11 class Student
12 {
13 public:
14     void Display();           // method
15 private:
16     char name[30];            // variable
17     Date birthday;            // variable
18     int sex;                   // variable
19     double mark;              // variable
20 };
21
22 void Date :: Display() { ... } // code trong vi du truoc
23 void Student :: Display() { ... } // code trong vi du truoc
24
25 int main()
26 {
27     Date holiday, birthday ;
28     holiday.day = 1 ;
29     holiday.month = 5 ;
30     holiday.year = 2015 ;
31     holiday.Display() ;
32     cout << endl;
33
34     return 0 ;
35 }
```

Hình 6.19: Truy xuất các thành viên của lớp.

Tính đóng gói và các từ khóa `public:`, `private:`

Lớp là một kiểu dữ liệu có vẻ giống cấu trúc, tuy nhiên bản chất của nó khác rất xa so với cấu trúc. Điểm giống nhau giữa hai loại, đó là cùng lưu trữ dữ liệu. Việc xử lý các dữ liệu này do các hàm đảm nhiệm. Các hàm này có thể xuất hiện bất kỳ đâu, trong bất kỳ chương trình nào, xử lý bất kỳ loại dữ liệu nào và do bất kỳ thành viên nào của nhóm lập trình viết ra. Với tính chất “rộng mở” như vậy, độ an toàn, tính nhất quán của các dữ liệu có vẻ không được chắc chắn lắm ! Do đó, cần phân định rõ hàm nào được phép làm việc với dữ liệu nào. Sau khi phân định xong, nên “đóng” kín tất cả các thành viên dữ liệu và hàm này vào cùng một “gói” cách ly với bên ngoài để bảo vệ. Các gói như vậy ta gọi là lớp. Mỗi lớp là một đặc tả, trừu tượng hóa một thực thể trong thực tế như lớp ngày tháng, lớp nhà cửa, lớp xe cộ, lớp các dấu tích tự ... Các thành viên của mỗi lớp đều được mặc định là cách ly với bên ngoài, có nghĩa nếu một hàm, một câu lệnh nằm bên ngoài lớp thì sẽ không được quyền truy xuất đến các thành viên của lớp (hiển nhiên, các thành viên trong cùng một lớp thì vẫn được quyền truy xuất lẫn nhau).

Tuy nhiên, việc giao tiếp với bên ngoài là không thể tránh khỏi (vật chất luôn luôn vận động), do vậy một số thành viên của lớp cần được cấp phép bằng cách khai báo từ khóa `public:` trước thành viên đó. Thông thường, dữ liệu cần được bảo vệ, nên sẽ không được “cấp phép” ngoại trừ trường hợp đặc biệt, còn lại các phương thức sẽ đại diện cho lớp để giao tiếp với bên ngoài nên thường được khai báo dạng `public`. Tuy mặc định các thành viên của lớp là khép kín (riêng biệt) nhưng để rõ ràng, lúc cần ta có thể đặt từ khóa `private:` trước các thành viên không được phép `public` để chỉ đây là thành viên riêng, bên ngoài không được quyền truy xuất, gọi đến nó.

- Trạng thái `public`: Các thành viên được khai báo ở trạng thái này có nghĩa bên ngoài lớp có thể sử dụng được, thường là các phương thức.
- Trạng thái `private`: Các thành viên chỉ được sử dụng bởi các thành viên khác bên trong lớp, thường là các biến.

Các từ khóa chỉ trạng thái chung (`public`) và riêng (`private`) không nhất thiết phải đặt trước mỗi thành viên mà từ điểm nó xuất hiện các thành viên phía sau đều sẽ có đặc tính đó cho đến khi gặp từ khóa ngược lại. Ví dụ về các lớp được khai báo ở trên, ta có 4 thành viên của `Date` đều là `public`, trong khi lớp `Student` chỉ có `Display()` là thành viên `public`, còn lại (các biến dữ liệu) đều là thành viên `private`.

Ta xem lại chương trình làm việc với lớp `Date` dưới đây.

```

1 #include <iostream>
2 using namespace std;
3
4 class Date
5 {
6 public:
7     void Display();
8     int day, month, year ;
9 };
10
11 void Date :: Display()
```

```

12 {
13     cout << day << "/" << month << "/" << year ;
14 }
15
16 int main()
17 {
18     Date holiday, birthday ;
19     holiday.day = 1 ;
20     holiday.month = 5 ;
21     holiday.year = 2015 ;
22     holiday.Display() ;
23     cout << endl;
24
25     return 0 ;
26 }

```

Hình 6.20: Lớp Date.

Trong chương trình, từ ngoài lớp Date (trong hàm main) ta có 4 câu lệnh gọi các thành viên của Date:

```

holiday.day = 1 ;
holiday.month = 5 ;
holiday.year = 2015 ;
holiday.Display() ;

```

Cả 4 câu lệnh này đều hợp lệ và chương trình chạy tốt vì cả 4 thành viên đều được khai báo dưới dạng public. Tuy nhiên, với ý tưởng cần che giấu, bảo vệ dữ liệu thì cách khai báo này là không tốt. Có nghĩa ta cần khai báo lại các biến này là private, khi đó lớp Date sẽ như sau:

```

class Date
{
public:
    void Display();
private:
    int day, month, year ;
};

```

Đây là cách khai báo tốt, tuy nhiên do day, month, year bây giờ đã là private nên 3 dòng lệnh gán của chương trình trên sẽ gây lỗi. Ba biến này không được phép truy cập từ bên ngoài (ở đây là trong hàm main), nó chỉ được truy cập bởi một phương thức bên trong Date. Do vậy, để khắc phục, ta xây dựng một phương thức mới là thành viên public của lớp. Phương thức này sẽ không chỉ gán giá trị cố định 1/5/2015 cho đối tượng mà tổng quát hơn, nó sẽ gán giá trị bất kỳ dd/mm/yy cho đối tượng, từ đó phương thức được xây dựng sẽ có 3 đối tượng ứng.

Ta xây dựng lại Date bằng cách cho các biến thành viên day, month, year là private và phương thức void Set(int dd, int mm, int yy); dùng để gán giá trị cho 3 biến này như chương trình bên dưới

```

1 #include <iostream>
2 using namespace std;
3
4 class Date
5 {
6 public:

```

```

7     void Display();
8     void Set(int dd, int mm, int yy);
9 private:
10    int day, month, year ;
11 };
12
13 void Date :: Display()
14 {
15     cout << day << "/" << month << "/" << year ;
16 }
17
18 void Date :: Set(int dd, int mm, int yy)
19 {
20     day = dd ;
21     month = mm ;
22     year = yy ;
23 }
24
25 int main()
26 {
27     Date holiday ;
28     holiday.Set(1, 5, 2015) ;
29     holiday.Display() ;
30     cout << endl;
31
32     return 0 ;
33 }

```

Hình 6.21: Lớp Date che giấu dữ liệu.

Cũng tương tự, giả sử bây giờ ta chỉ cần in ngày, tháng của ngày lễ, ta sẽ không thể viết như: `cout << holiday.day << ", " << holiday.month ;`. Rõ ràng, ta cần đưa thêm vào lớp các phương thức (public) cho phép truy xuất, trả lại giá trị của từng biến thành viên như:

```

int getDay() ;
int getMonth();
int getYear();

```

khi đó lớp Date được mở rộng thành:

```

1 #include <iostream>
2 using namespace std;
3
4 class Date
5 {
6 public:
7     void Display();
8     void Set(int dd, int mm, int yy);
9     int getDay() { return day; }
10    int getMonth() { return month; }
11    int getYear() { return year; }
12 private:
13    int day, month, year ;
14 };
15

```

```

16 void Date :: Display() { ... }           // code trong ví dụ trước
17 void Date :: Set(int dd, int mm, int yy) { ... } // code trong ví dụ trước
18
19 int main()
20 {
21     Date holiday ;
22     holiday.Set(1, 5, 2015) ;
23     cout << holiday.getDay() << "/" << holiday.getMonth() << " of every year is
        holiday" ;
24     cout << endl;
25
26     return 0 ;
27 }

```

Hình 6.22: Lớp Date với các hàm truy xuất dữ liệu.

Tóm lại :

```

cout << holiday.day ;           // không dùng được vì day là private
cout << holiday.getDay() ;      // dùng được vì getDay() là public

```

Thông thường nếu định nghĩa hàm không quá dài, ta cũng có thể đặt ngay trong khai báo lớp. Các hàm get trong lớp này là một ví dụ.

Với cơ chế đóng gói, dữ liệu muốn giao tiếp với bên ngoài (cũng có nghĩa bên ngoài muốn truy xuất đến dữ liệu của lớp) sẽ phải thông qua các phương thức của lớp đó (dĩ nhiên phải là **public**), điều này đảm bảo được tính an toàn và nhất quán của dữ liệu, giúp người lập trình tránh những sai sót khi mã chương trình. Điều này cũng tương tự việc hàng xóm muốn nói chuyện với con cái nhà bên cạnh thường phải thông qua Bố Mẹ hoặc để tiếp cận với những người nổi tiếng cần thông qua quản lý của họ. Những người nổi tiếng cũng giống như trẻ em hoặc dữ liệu của thực thể đều cần được bảo vệ.

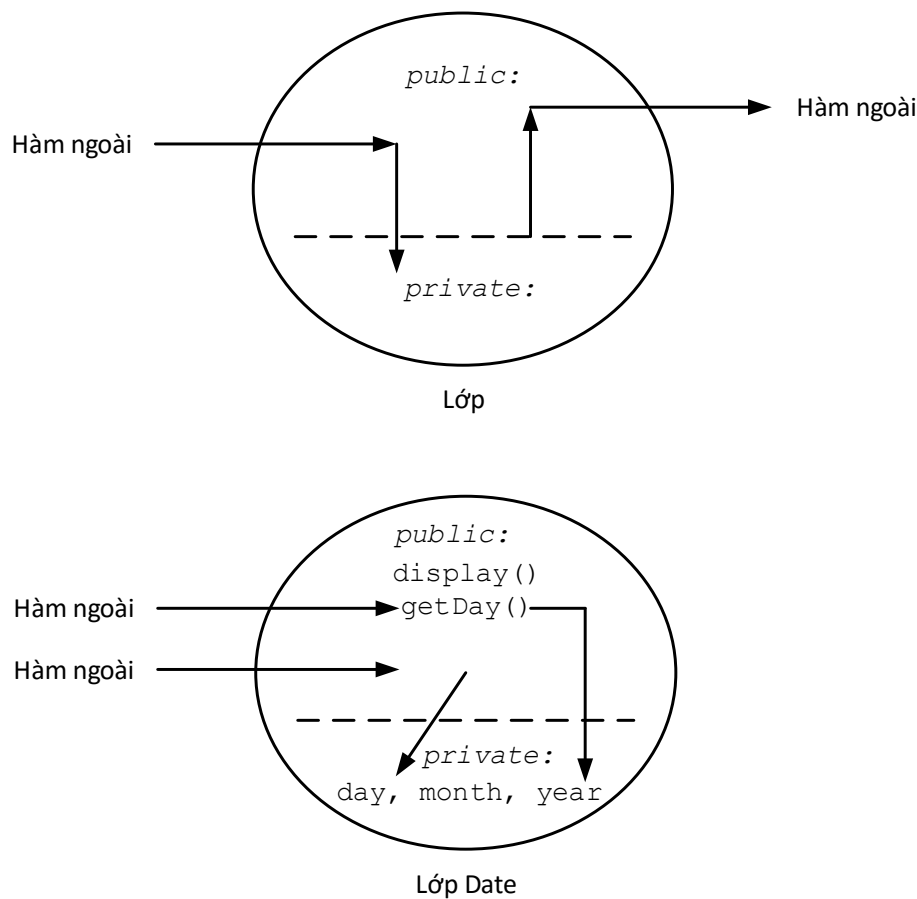
Ngoài các từ khóa **public**, **private** các thành viên của lớp còn có thể khai báo với từ khóa **protected**. Ảnh hưởng của từ khóa này giống như **private**, tuy nhiên nó cho phép mềm dẻo hơn đối với các lớp thừa kế. Phù hợp với mức độ, giáo trình này không trình bày về tính thừa kế, người đọc có thể tìm hiểu thêm trong các giáo trình C/C++.

Tóm lại, cơ chế đóng gói đảm bảo:

- Tính an toàn: Các thành viên **private** (thường là các biến chứa dữ liệu) là bất khả xâm phạm (không truy cập được từ bên ngoài), nó chỉ có thể được truy cập bởi các thành viên (phương thức) trong lớp đó.
- Tính nhất quán: Dữ liệu của lớp nào chỉ được xử lý bởi phương thức của lớp đó, bằng cách làm việc được qui định trước bởi lớp đó.

Cơ chế đóng gói còn có đặc trưng quan trọng nữa đó là nó cho phép ta tạo ra các lớp dữ liệu hoàn chỉnh, khép kín, độc lập với mọi chương trình. Điều này có nghĩa nếu ta thiết kế lớp đủ tổng quát và cài đặt hoàn chỉnh thì ta có thể sử dụng lại lớp này trong các chương trình khác mà không cần phải sửa mã cho phù hợp với chương trình mới.

Cơ chế đóng gói là ý tưởng quan trọng hình thành nên kỹ thuật lập trình hướng đối tượng, hiệu quả hơn so với các kỹ thuật lập trình trước đó. Người đọc có thể tìm hiểu thêm về chủ đề này trong các giáo trình về lập trình hướng đối tượng.



Hình 6.23: Minh họa cơ chế đóng gói.

Phép gán

Cũng giống như cấu trúc, hai đối tượng cùng lớp bất kỳ được phép gán giá trị cho nhau. Ví dụ:

```
Date holiday, birthday ;
holiday.Set(8, 3, 2015) ;
birthday = holiday ;
birthday.Display();           // 8/3/2015
```

Và cũng giống như cấu trúc, tuy gán được cho nhau nhưng hai đối tượng không thể so sánh (`==`) được với nhau, trừ phi so sánh từng biến thành viên. Tuy nhiên, trong phần sau chúng ta sẽ học cách làm thế nào để viết các phép toán cho lớp (ví dụ phép toán `==`).

Tóm tắt một số đặc trưng của lớp

Ngoại trừ những tính chất của một cấu trúc, lớp còn có những tính chất liên quan đến các phương thức như được tóm tắt dưới đây:

- Lớp gồm 2 loại thành viên: biến và phương thức (hoặc hàm).
- Định nghĩa của các phương thức có thể đặt bên trong hoặc ngoài lớp.
- Mỗi một thành viên có thể ở một trong hai trạng thái : **public** (chung) hoặc **private** (riêng).
- Các thành viên trong cùng một lớp có thể truy cập lẫn nhau. (Phương thức này có thể gọi đến phương thức khác hoặc biến thành viên).

- Bên ngoài có thể truy cập đến các thành viên `public` của lớp, nhưng không thể truy cập đến các thành viên `private` (Các thành viên `private` chỉ có thể được truy cập bởi các phương thức trong cùng một lớp).
- Thông thường các biến thành viên là `private`, các phương thức là `public`.
- Các phương thức thành viên được phép khai báo và định nghĩa giống như các hàm thông thường (đối mặc định, chồng (trùng tên), ...).
- Một lớp có thể sử dụng lớp khác để làm biến thành viên.
- Đối và giá trị trả lại của một phương thức được phép là đối tượng của lớp.
- Để chạy một phương thức cần có đối tượng gọi đến phương thức đó. Ví dụ: `holiday.Display()`.
- Khi chạy một phương thức, các biến thành viên được viết trong phương thức sẽ nhận dữ liệu truyền là dữ liệu của đối tượng gọi đến phương thức đó.

6.2.3 Bài toán Quản lý sinh viên

Trong phần này, ta tiếp tục minh họa cách viết và sử dụng lớp thông qua bài toán QLSV đã mô tả trong mục 6.1.3 (tạm gọi là phiên bản A1). Với phiên bản A2, các thực thể ngày tháng và sinh viên được tổ chức thành các lớp, các thay đổi giữa hai phiên bản sẽ được trình bày trong phần tiếp theo. Chương trình hoàn chỉnh cuối cùng được trình bày trong Phụ lục A2.

Chức năng của chương trình - hàm `main()`

Hoàn toàn giống với A1, do vậy hàm `main()` không thay đổi, chủ yếu tạo menu cho người dùng lựa chọn chức năng cần thực hiện (xem 6.1.3).

Khai báo lớp

```

1 //----- Khai bao lop ngay thang
2 class Date
3 {
4 public:
5     void Display();
6     void Set(int dd, int mm, int yy);
7     int getDay() { return day; }           // tra lai ngay
8     int getMonth() { return month; }       // tra lai thang
9     int getYear() { return year; }         // tra lai nam
10 private:
11     int day, month, year ;
12 };
13
14 //----- Khai bao lop sinh vien
15 class Student
16 {
17 public:
18     void Display();
19     void Set(char nme[], int dd, int mm, int yy, int sx, double mrk);
20     void New();
21     void Update();
22     void getName(char nme[]) { strcpy(nme, name); } // tra lai ho ten

```

```

23     Date getBirthday() { return birthday ; }           // tra lai ngay sinh
24     int getSex() { return sex; }                       // tra lai gioi tinh
25     double getMark() { return mark; }                 // tra lai diem
26 private:
27     char name[30];
28     Date birthday;
29     int sex;
30     double mark;
31 };

```

Hình 6.24: Khai báo các lớp của chương trình QLSV.

Hai cấu trúc `Date` và `Student` được chuyển thành 2 lớp. Các thành phần của cấu trúc cũ giờ là các biến thành viên và được đặt dưới từ khóa `private`. Ta bổ sung vào lớp các phương thức được sửa chữa từ các hàm của A1 và các phương thức mới theo nhu cầu. Một số phương thức thành viên được định nghĩa ngay trong phần thân của khai báo lớp (thường là những phương thức có định nghĩa khoảng vài dòng lệnh). Các phương thức còn lại được định nghĩa bên ngoài khai báo và được trình bày trong phần tiếp theo. `Student` có một biến thành viên là lớp `Date` (lớp trong lớp).

Về các phương thức, ta phân bố lại các hàm `displayDate()` và `displayStudent()` trong A1 về cho 2 lớp và cùng lấy tên là `Display()`. Các vị ngữ như `Date` hay `Student` là không cần thiết vì do tính đóng gói các hàm `Display()` này là độc lập nhau.

Một phát sinh so với A1 đó là các hàm bên ngoài sẽ không còn truy cập được trực tiếp vào các biến thành viên của 2 lớp, tuy nhiên, có hai nhóm thao tác ta cần dùng liên quan đến truy cập các biến thành viên là đặt lại giá trị và lấy giá trị của các biến này. Do vậy, trong mỗi lớp ta cần cài đặt thêm 2 nhóm phương thức được qui ước bắt đầu bằng từ `set` (đặt lại giá trị) và `get` (lấy giá trị).

Định nghĩa các phương thức

```

1  //----- Hien thi ngay thang
2  void Date::Display()
3  {
4      cout << day << "/" << month << "/" << year ;
5  }
6
7  //----- Dat gia tri cho ngay thang
8  void Date::Set(int dd, int mm, int yy)
9  {
10     day = dd ;
11     month = mm ;
12     year = yy ;
13 }
14
15 //----- Hien thi thong tin sinh vien
16 void Student::Display()
17 {
18     cout << name << "\t" ;
19     cout << setw(2) << birthday.getDay() << "/" << setw(2) << birthday.getMonth() <<
20         "/" << setw(2) << birthday.getYear() << "\t" ;
21     if (sex == 0) cout << "Male" << "\t" ; else cout << "Female" << "\t" ;
22     cout << mark << endl;
23 }
24 //----- Dat gia tri cho sinh vien

```



```

25 void Student::Set(char nme[], int dd, int mm, int yy, int sx, double mrk)
26 {
27     strcpy(name, nme) ;
28     birthday.Set(dd, mm, yy);           // gọi ham Set của Date
29     sex = sx;
30     mark = mrk;
31 }
32
33 //----- Tao moi mot sinh vien tu ban phim
34 void Student::New()
35 {
36     int mm, dd, yy ;
37     fflush(stdin) ;
38     cout << "    Full name: "; cin.getline(name, 30) ;
39     cout << "    Birthday: "; cin >> dd >> mm >> yy ; birthday.Set(dd, mm, yy) ;
40     cout << "    Sex (0: Male, 1: Female): "; cin >> (sex) ;
41     cout << "    Mark: "; cin >> (mark) ;
42 }
43
44 //----- Cap nhat thong tin mot sinh vien
45 void Student::Update()
46 {
47     Date brday;           // su dung lop khac trong phuong thuc
48     char nme[30]; int dd, mm, yy; int sx; double mrk;   // cac gia tri moi
49     // Lay noi dung của đối tượng ra các biến le ben ngoai
50     getName(nme); brday = getBirthday(); sx = getSex(); mrk = getMark();
51     dd = brday.getDay() ; mm = brday.getMonth() ; yy = brday.getYear() ;
52     Display();           // hien noi dung ban ghi can sua
53
54     int choice;
55     do {
56         cout << endl;
57         cout << "1: Name" << endl ;
58         cout << "2: Birthday" << endl ;
59         cout << "3: Sex" << endl ;
60         cout << "4: Mark" << endl ;
61         cout << "0: Exit" << endl ;
62         cout << "Select member to update ? " ;
63         cin >> choice ; cin.ignore();
64         switch (choice)
65         {
66             case 1: cout << "Enter new name: " ; cin.getline(nme, 30) ; break;
67             case 2: cout << "Enter new birthday: " ; cin >> dd >> mm >> yy ; break;
68             case 3: cout << "Enter new sex: " ; cin >> sx ; break;
69             case 4: cout << "Enter new mark: " ; cin >> mrk ; break;
70         }
71     } while (choice) ;
72     int sure ;
73     cout << "Are you sure for updating (1: sure, 0: not sure) ? " ; cin >> sure ;
74     if (sure)
75     {
76         Set(nme, dd, mm, yy, sx, mrk) ;
77         cout << "\nInformation of student was updated\n";
78     }

```

```

79     else cout << "\nUpdating cancelled\n";
80
81 }

```

Hình 6.25: Các phương thức của Date và Student.

Các phương thức đã được khai báo nhưng chưa cài đặt (bên trong phần khai báo) đã được cài đặt ở đây và luôn có toán tử chỉ định phạm vi `::` đi kèm cùng tên lớp, do vậy sự trùng tên của các phương thức là không thành vấn đề. Hầu hết các phương thức này được chuyển “ngang” từ bản A1 sang bản này, trừ một vài chú ý phải sửa chữa. Ví dụ trong phương thức `Student::Display()` ta cần hiển thị ngày sinh của sinh viên, ta không thể viết `cout << birthday.day` như trong bản A1 vì đây là thành viên `private` của lớp `Date` và `Student::Display()` không có quyền gọi đến. Để hiển thị được `day` ta cần phải thông qua một thành viên `public` của lớp `Date` đó là `getDay()`, từ đó câu lệnh trên phải được sửa thành: `cout << birthday.getDay()`. Tương tự để đặt lại giá trị ngày tháng năm sinh của sinh viên (hàm `Student::Set()`) ta phải cầu viện đến một thành viên của `Date` là `Date::Set()`, có nghĩa không thể viết `birthday.day = dd ...` mà phải là cả cụm: `birthday.Set(dd, mm, yy)`; (lưu ý `birthday` là một đối tượng của `Date`).

Hàm `Student::New()` cho phép tạo mới một đối tượng với dữ liệu nhập từ bàn phím (cũng là một dạng `Set`). Trừ các biến thành viên họ tên, giới tính, điểm mà `New` được quyền truy cập, các biến nếu thuộc lớp khác (`birthday` thuộc `Date`) ta phải nhập qua biến thường trung gian (`dd, mm, yy`) rồi sau đó gọi đến hàm `Set` của `Date` để gán giá trị cho `birthday`.

Cách làm việc của hàm `Student::Update()` cũng giống như bản A1. Để đảm bảo an toàn ta chỉ cập nhật trên bản sao của sinh viên cần sửa. Từ đó, ta cần đến nhóm hàm `get` trong `Student` để lấy thông tin của sinh viên này ra các biến lẻ bên ngoài và tiến hành sửa chữa giá trị của các biến này. Chỉ sau khi NSD đồng ý cập nhật ta mới dùng đến hàm `Student::Set()` để đặt lại các giá trị của sinh viên vừa được sửa chữa.

Các biến và hàm còn lại của chương trình

Các biến danh sách sinh viên, số sinh viên khai báo như A1. Các hàm phục vụ như: `getFirstname`, `swap` giống hoàn toàn A1. Trong `swap`, các đối tượng (kiểu lớp) cũng được truyền theo tham chiếu như khi nó là cấu trúc.

Các hàm chức năng chính của chương trình, chỉ cần sửa chữa nhỏ, chủ yếu tập trung vào các thay đổi từ kiểu cấu trúc sang lớp. Ví dụ để tạo sinh viên mới và gán vào ô `index` của `List`, trong A1 ta gọi chức năng bằng dòng lệnh: `List[index] = New_student()` ; còn ở đây nó được thay bằng dòng lệnh: `List[index].New()` ;

Chương trình hoàn chỉnh của phiên bản này, người đọc có thể tham khảo trong Phụ lục A2.

6.2.4 Khởi tạo (giá trị ban đầu) cho một đối tượng

Hàm tạo (Constructor function)

- *Hàm tạo mặc định.* Thông thường khi khai báo một đối tượng, cũng giống như các loại biến khác ta mong muốn khởi tạo trước cho toàn bộ hoặc một số biến thành viên của lớp những giá trị cho trước. Để làm điều này, ta phải viết một hàm đặc biệt với dạng được qui định sẵn được gọi là hàm tạo và hàm này sẽ tự động chạy mỗi khi đối tượng mới được khai báo, hàm sẽ khởi gán giá trị cho các thuộc tính (biến) của đối tượng và ngoài ra, còn có thể thực hiện một số công việc khác nhằm chuẩn bị cho đối tượng mới.

Như vậy, hàm tạo cũng là một phương thức của lớp (nhưng là phương thức đặc biệt) dùng để tạo dựng một đối tượng mới và được gọi chạy tự động. Khi gặp một khai báo, đầu tiên chương trình dịch sẽ cấp phát bộ nhớ cho đối tượng sau đó sẽ gọi đến hàm tạo.

Tuy nhiên, nếu trong lớp ta không viết hàm tạo thì chương trình vẫn cung cấp một hàm tạo được gọi là hàm tạo mặc định. Mỗi khi có đối tượng mới được khai báo, hàm tạo này sẽ được gọi, nhưng chỉ đơn giản là nó không làm gì. Các giá trị của đối tượng mới cũng là ngẫu nhiên (ta hay gọi là “rác”). Như vậy, mục đích của hàm tạo mặc định chỉ đơn giản là để phù hợp với cách hoạt động của C++.

Nói chung hầu hết các lớp đều cần có hàm tạo do NSD viết để phục vụ cho việc gán các giá trị ban đầu cho lớp.

- *Hàm tạo không đối.* Ví dụ sau mô tả một hàm tạo không đối cho lớp `Date`:

```
1 Date::Date()  
2 {  
3     day = 1 ;  
4     month = 1 ;  
5     year = 2000 ;  
6 }
```

Hình 6.26: Hàm tạo `Date()`.

Hàm tạo này gán giá trị ngày 1 tháng 1 năm 2000 cho bất kỳ đối tượng mới nào được khai báo.

- *Hàm tạo có đối.* Vẫn trên lớp `Date` ta có thể viết thêm hàm tạo có đối như sau:

```
1 Date::Date(int dd, int mm, int yy)  
2 {  
3     day = dd ;  
4     month = mm ;  
5     year = yy ;  
6 }
```

Hình 6.27: Hàm tạo có đối số.

Như vậy, một lớp có thể không có hàm tạo (sử dụng hàm tạo mặc định, giá trị các biến lúc đó là “rác”), có thể có một hàm tạo hoặc có thể có nhiều hàm tạo, có đối hoặc không có đối. Nói chung, để các đối tượng được khởi tạo một cách tường minh, hầu hết các lớp đều có ít nhất một hàm tạo không đối.

Trường hợp lớp có nhiều hàm tạo, hàm nào sẽ được gọi mỗi khi đối tượng được khai báo ? Để lựa chọn, chương trình sẽ so sánh số lượng giá trị kèm theo đối tượng được khai báo với số đối của hàm để quyết định chạy hàm nào.

Ví dụ quay lại hai hàm tạo trên, khai báo `Date holiday`; (không giá trị kèm theo) sẽ gọi chạy hàm thứ nhất (không đối), do vậy `holiday = 1/1/2000`. Nếu khai báo `Date holiday(20, 11, 2015)` hàm khởi tạo thứ hai (có đối) sẽ được gọi, khi đó các tham đối `dd`, `mm`, `yy` sẽ nhận các giá trị 20, 11, 2015 và gán cho các biến thành viên `day`, `month`, `year`, do đó `holiday` có giá trị ban đầu là 20/11/2015.

Ta cũng có thể viết hàm tạo thứ 3 với hai đối tự do và một đối mặc định như:

```
Date::Date(int dd, int mm, int yy = 2000)
```

Khi đó, khai báo `Date holiday(20, 11)` sẽ gọi hàm tạo thứ 3 và giá trị `holiday` sẽ là 20/11/2000.

Lưu ý các hàm tạo đều có dạng và chung mục đích như các hàm được chúng ta đặt tên bắt đầu bằng từ `Set` trong các ví dụ trước. Tuy nhiên, hàm tạo được gọi chạy tự động và một lần duy nhất ngay sau lúc đối tượng được khai báo, để khởi tạo (initialize) giá trị ban đầu cho đối tượng, còn hàm `Set` sẽ chạy mỗi lần có đối tượng gọi đến nó để “setup” lại giá trị của đối tượng này.

- *Tính nhập nhằng khi có hàm tạo có đối nhưng không có hàm tạo không đối.*

Khi trong lớp không có hàm tạo nào thì chương trình dịch sẽ cung cấp một hàm tạo ngầm định là hàm không làm gì cả. Khi lớp có ít nhất một hàm tạo thì chương trình dịch sẽ không phát sinh ra hàm tạo ngầm định này nữa. Do đó, nếu lớp có hàm tạo có đối nhưng không có hàm tạo không đối thì sẽ rất dễ gây ra lỗi. Cụ thể, trong trường hợp này nếu ta khai báo một đối tượng không giá trị kèm theo thì chương trình sẽ không thể khởi tạo cho đối tượng này (vì ta không xây dựng hàm tạo không đối và chương trình cũng không phát sinh ra hàm tạo ngầm định). Do vậy, cách viết chương trình tốt là trong lớp nên luôn luôn có hàm tạo không đối, hoặc một trong những hàm tạo có đối với tất cả các đối này là mặc định, khi đó hàm này cũng có thể phục vụ như hàm tạo không đối.

- *Cú pháp và một số đặc điểm của hàm tạo.*

- Tên của hàm tạo: Tên của hàm tạo bắt buộc phải trùng với tên của lớp (ví dụ `Date()`).
- Hàm tạo không có kết quả trả về.
- Không khai báo kiểu cho hàm tạo (kể cả `void`).
- Hàm tạo có thể được xây dựng bên trong hoặc bên ngoài định nghĩa lớp.
- Hàm tạo có thể có đối hoặc không đối, trong danh sách đối cũng có thể có những đối mặc định.
- Trong một lớp có thể có nhiều hàm tạo (cùng tên nhưng khác số đối).

- *Hàm tạo với phần khởi tạo trước.*

Ngoài hàm tạo mặc định, không đối, có đối ta còn một dạng hàm tạo với phần khởi tạo trước. Để đơn giản, ta minh họa bằng ví dụ hàm tạo dạng này:

```
Date::Date() : day(1), month(1), year(2015) // hàm không đối
{
    // không có dòng lệnh nào
}
```

Hàm tạo này tương đương với hàm tạo:

```
Date()
{
    day = 1;
    month = 1;
    year = 2015;
}
```

Hoặc xét hàm tạo:

```
// hàm này chỉ có hai đối
Date::Date(int dd, int mm) : yy(2015)
{
    day = dd;
    month = mm;
}
```

Hàm tạo này là tương đương với

```
Date(int dd, int mm)
{
    day = dd;
    month = mm;
    year = 2015;
}
```

Khai báo `Date holiday(8, 3);` sẽ tạo đối tượng `holiday` với giá trị 8/3/2015.

Như vậy, hàm tạo với phần khởi tạo (ngay trong dòng tiêu đề của hàm) sẽ thiết lập giá trị cụ thể (có thể là biểu thức) cho một hoặc nhiều các thuộc tính của đối tượng. Các thuộc tính còn lại sẽ được thiết lập giá trị bằng các câu lệnh bên trong hàm tạo. Thực tế, mọi hàm tạo với phần khởi tạo trước đều có thể thay thế một cách tương ứng bằng một hàm tạo có đối thông thường.

Danh sách các thuộc tính cần khởi tạo trước được viết nối tiếp theo tiêu đề của hàm với ngăn cách bởi dấu hai chấm (':'). Mỗi thuộc tính cần khởi tạo gồm tên biến thuộc tính và giá trị (có thể là biểu thức) nằm trong cặp dấu ngoặc tròn. Các thuộc tính cần khởi tạo trước cách nhau bởi dấu phẩy (',').

- *Gọi hàm tạo như một phương thức thông thường.*

Như trên, các hàm tạo được gọi một cách tự động mỗi khi có đối tượng mới được khai báo. Tuy nhiên, ta cũng có thể gọi hàm tạo một cách tường minh bằng câu lệnh:

```
Date(1, 1, 2015); // Date là tên lớp. Không có tên đối tượng
```

Câu lệnh này đầu tiên sẽ tạo ra một đối tượng mới vô danh (không có tên) và sau đó sẽ gọi đến hàm tạo với 3 đối (giả sử đã được cài đặt như trong các ví dụ trước) để gán giá trị cho đối tượng này. Như vậy, đối tượng vô danh có giá trị 1/1/2015. Mục đích sử dụng đối tượng vô danh này (trong một số giáo trình còn gọi là đối tượng hằng) là để gán giá trị cho các đối tượng khác. Ví dụ:

```
Date holiday; // gọi hàm khởi tạo không đối.
// holiday được gán giá trị "mới" 2/9/1945
holiday = Date(2, 9, 1945) ;
```

Như vậy, cách gán giá trị dạng này tương tự như các hàm Set trong các phần trên.

- *Bảng tóm tắt các dạng khởi tạo bằng hàm tạo.* Phần này ta giả thiết các loại dạng hàm tạo đã đề cập bên trên đều có cài đặt trong lớp `Date`. Giả sử ta cần tạo đối tượng `holiday` và gán giá trị 8/3/2015 bằng các câu lệnh sau:

```
Date holiday;
```

gọi hàm tạo không đổi, giả sử hàm này đã được viết để khởi tạo cho mọi đối tượng mới với giá trị không đổi là 8/3/2015. Nếu lớp không có hàm tạo nào thì câu lệnh này sẽ cho ra đối tượng holiday với giá trị chưa xác định. Nếu lớp có ít nhất một hàm tạo nhưng không có hàm tạo không đổi thì câu lệnh này sẽ bị chương trình dịch báo lỗi (nói chung, nên có hàm tạo không đổi)

```
Date holiday(8, 3, 2015);    // gọi hàm tạo 3 đối
// gọi hàm tạo 3 đối, trong đó có một đối mặc định là year = 2015.
Date holiday(8, 3);
```

Cũng có thể gọi hàm tạo hai đối trong đó có câu lệnh gán year = 2015. Hoặc gọi hàm tạo hai đối với phần khởi tạo trước year = 2015. Trường hợp lớp có cùng lúc hai trong ba hàm tạo dạng này thì chương trình dịch sẽ báo lỗi vì tính nhập nhằng (chỉ nên viết một).

```
Date holiday;
// gán giá trị bằng đối tượng hằng, hoặc
holiday = Date(8, 3, 2015);
// gán giá trị bằng phương thức Set (đã viết)
holiday.Set(8, 3, 2015);
```

Hàm tạo sao chép (Copy constructor)

Ngoài việc khởi tạo đối tượng từ những giá trị cụ thể (được hỗ trợ bởi các hàm tạo), ta cũng có thể khởi tạo đối tượng từ những giá trị của một đối tượng khác bằng các hàm tạo sao chép. Ví dụ xét các khai báo và khởi tạo cho 2 đối tượng holiday và birthday:

```
Date holiday(1, 5, 2015);    // từ giá trị cụ thể
Date birthday(holiday) ;    // từ đối tượng khác
```

Nếu lớp chưa có hàm tạo sao chép, thì câu lệnh này sẽ gọi tới một hàm tạo sao chép mặc định. Hàm này sẽ sao chép nội dung từng bit của holiday vào các bit tương ứng của birthday. Trong đa số các trường hợp, nếu lớp không có các thuộc tính kiểu con trỏ hay tham chiếu, thì việc dùng các hàm tạo sao chép mặc định (để tạo ra một đối tượng mới có nội dung như một đối tượng cho trước) là đủ và không cần xây dựng một hàm tạo sao chép mới. Ví dụ như trong trường hợp trên ta cũng có birthday = 1/5/2015.

Trong các trường hợp khác ta có thể xây dựng một hàm tạo sao chép theo mẫu:

```
class_name(const class_name &object)
{
    ...
}
```

Trong đó, đối tượng tham đối object được khai báo dưới dạng tham chiếu để tiết kiệm bộ nhớ và thời gian truyền, do vậy ta cũng thêm từ khóa const vì không muốn giá trị của object bị thay đổi sau khi hàm được gọi thực hiện.

Trong mẫu khai báo trên, ta lưu ý hàm tạo sao chép cũng không có kiểu đối trả lại và cũng vậy trong thân hàm cũng không có câu lệnh trả lại giá trị.

Ví dụ có thể xây dựng hàm tạo sao chép cho lớp Date như sau:

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
```

```

5  class Date
6  {
7  public:
8      Date();
9      Date(int dd, int mm, int yy);
10     Date(const Date &source);
11     void Display();
12     // ...
13 private:
14     int day, month, year ;
15 };
16
17 void Date :: Display()
18 {
19     cout << day << "/" << month << "/" << year ;
20 }
21
22 Date::Date(int dd, int mm, int yy)
23 {
24     day = dd ;
25     month = mm ;
26     year = yy ;
27 }
28
29 Date::Date(const Date &source)
30 {
31     day = source.day ;
32     month = source.month ;
33     year = source.year ;
34     cout << "The new object is initialized with values " << day << "/" << month << "/"
35          << year << endl;
36 }
37
38 int main()
39 {
40     Date d(2,3,4);
41     Date t(d);
42     t.Display();
43     cout << endl;
44
45     return 0;
46 }

```

Hình 6.28: Hàm tạo sao chép cho lớp Date.

Trong ví dụ trên, nếu không có câu lệnh cuối thì hàm tạo sao chép này hoàn toàn trùng với hàm tạo sao chép mặc định. Trong thực tế, câu lệnh cuối của hàm tạo trên hầu như không cần thiết, và vì vậy chỉ cần dùng hàm tạo sao chép mặc định là đủ. Tuy nhiên, khi lớp có các thuộc tính con trở hoặc tham chiếu thì cần phải viết hàm tạo sao chép để xử lý các vấn đề liên quan đến các thuộc tính này (sẽ đề cập đến trong các chương sau).

6.2.5 Hủy đối tượng

Cũng giống các biến đơn giản khác, biến được khai báo trong môđun (chương trình, khối lệnh, hàm ...) nào sau khi chạy xong môđun đó biến sẽ tự hủy, ở đây các đối tượng cũng tương tự. Tuy nhiên, một đối tượng trong quá trình khởi tạo và hoạt động có thể sinh ra một số vấn đề khác như xin cấp phát bộ nhớ, bộ nhớ này sẽ không tự hủy ..., do vậy cần viết một phương thức đặc biệt một cách tường minh để giải quyết các vấn đề này. Phương thức đặc biệt này được gọi là hàm hủy.

Hàm hủy (Deconstruction function)

Hàm hủy là một hàm thành viên của lớp (phương thức) có chức năng ngược với hàm tạo. Hàm hủy sẽ được chương trình gọi tự động trước khi một đối tượng tự hủy để thực hiện một số công việc có tính “dọn dẹp” liên quan đến đối tượng này.

- *Hàm hủy mặc định.* Nếu trong lớp không định nghĩa hàm hủy, thì một hàm hủy mặc định không làm gì cả được phát sinh. Đối với nhiều lớp thì hàm hủy mặc định là đủ, và không cần đưa vào một hàm hủy mới. Thông thường, hàm hủy chỉ cần khi phải giải phóng bộ nhớ do trong quá trình hoạt động đối tượng đã yêu cầu cấp phát.
- *Quy tắc viết hàm hủy.*
 - Tên của hàm hủy gồm một dấu ngã (đứng trước) và tên lớp:

```
~class_name()
```
 - Hàm hủy không có đối.
 - Là hàm không có kiểu, không có giá trị trả về.
 - Mỗi lớp chỉ có duy nhất một hàm hủy, là phương thức của lớp, có thể định nghĩa trong hoặc ngoài khai báo lớp. Ví dụ: `Date()`; là hàm hủy cho lớp `Date`.

6.2.6 Hàm bạn (friend function)

Ý nghĩa của hàm bạn

Như đã biết, các thuộc tính `private` của lớp là đóng kín đối với bên ngoài, để truy cập được chúng cần phải thông qua nhóm các hàm `get` (để lấy giá trị). Tuy nhiên, với các lớp có nhiều thuộc tính việc viết các hàm `get` này sẽ rất dài. Một kỹ thuật cho phép các hàm ngoài vẫn truy cập được vào các thuộc tính này đó là khai báo các hàm ngoài là bạn của lớp.

Cách khai báo hàm bạn

Để một hàm trở thành bạn của một lớp, ta cần khai báo tên hàm này vào bên trong định nghĩa của lớp kèm theo từ khóa `friend` đứng trước. Hàm được định nghĩa bên ngoài như các hàm thông thường khác (không có toán tử chỉ định phạm vi `::`). Lời gọi của hàm bạn giống như lời gọi của hàm thông thường.

Ví dụ: Ta xây dựng lớp số phức (`Complex`) gồm 2 thuộc tính phần thực (`real`), phần ảo (`image`) và hàm cộng (`Plus`) như sau :

```
class Complex
{
public:
    Complex Plus(Complex b)
```



```

    {
        Complex res;
        res.real = real + x.real ;
        res.image = image + x.image ;
        return res;
    }
private:
    double real;          // phần thực
    double image;         // phần ảo
};

```

Hàm Plus có một đối kiểu số phức và giá trị trả lại là một số phức. Khi một số phức x gọi đến hàm này với đối y , hàm sẽ thực hiện cộng x vào với y và trả lại số phức kết quả. Vì vậy ta có thể đặt z là tổng của x và y bằng câu lệnh:

```

Complex x, y, z;
// Gán giá trị cho x và y
z = x.Plus(y);

```

Cách viết này không giống với cách thông thường, ví dụ ta muốn viết $z = \text{Plus}(x, y)$, tuy nhiên hàm này có hai đối nên không thể là thành viên của lớp. Vì vậy, ta sẽ viết lại hàm Plus như một hàm thông thường bên ngoài lớp Complex như sau:

```

Complex Plus(Complex a , Complex b)
{
    Complex res;
    res.real = a.real + b.real ;
    res.image = a.image + b.image ;
    return res;
}

```

Tuy nhiên cách viết này bị lỗi vì hàm Plus (không phải thành viên của lớp) không được phép truy cập đến các thuộc tính private (real, image) của lớp. Để hàm hoạt động đúng, ta có thể giải quyết bằng cách thông qua các hàm getReal, getImage để lấy các giá trị này. Hoặc đơn giản hơn, ta chỉ cần đăng ký hàm là bạn của Complex bằng cách khai báo thêm tên hàm này vào bên trong lớp Complex cùng từ khóa friend, như chương trình hoàn chỉnh bên dưới.

```

1 #include <iostream>
2 using namespace std;
3
4 class Complex
5 {
6 public:
7     Complex();                               // ham tao khong doi
8     Complex(double r, double i);             // ham tao co doi
9     friend Complex Plus(Complex a, Complex b);
10    void Display();
11 private:
12    double real;                               // phan thuc
13    double image;                             // phan ao
14 };
15
16 Complex::Complex()                           // ham thanh vien cua lop
17 {
18     real = 0;

```

```

19     image = 0;
20 }
21
22 Complex::Complex(double r, double i)           // ham thanh vien cua lop
23 {
24     real = r;
25     image = i;
26 }
27
28 void Complex::Display()                       // ham thanh vien cua lop
29 {
30     cout << real << " + " << image << "i";
31 }
32
33 Complex Plus(Complex a, Complex b)           // ham ngoai lop
34 {
35     Complex c;
36     c.real = a.real + b.real ;
37     c.image = a.image + b.image ;
38     return c;
39 }
40
41 int main()
42 {
43     Complex comp1(1, 2);                     // 1 + 2i
44     Complex comp2(2, 3);                     // 2 + 3i
45     Complex comp3;
46     comp3 = Plus(comp1, comp2);
47     comp3.Display();
48     cout << endl;
49
50     return 0;
51 }

```

Hình 6.29: Hàm bạn của lớp Complex.

Hàm bạn của nhiều lớp

Trong khá nhiều chương trình, hai lớp cần truy cập lẫn nhau, tương tác với nhau về một khía cạnh nào đó. Tuy nhiên, phương thức của lớp này lại không thể truy cập đến thuộc tính của lớp kia và ngược lại. Khi đó, cách giải quyết tốt nhất là ta viết hàm bạn của cả hai lớp để nó có thể truy nhập được đến thuộc tính của cả hai.

Ví dụ nếu nhân hai ma trận ta có thể viết hàm nhân là thành viên trong lớp **Matrix** để làm điều này. Tuy nhiên, để nhân một ma trận (của lớp **Matrix**) với một vectơ (của lớp **Vector**) không thể hàm thành viên nào của **Matrix** có thể thực hiện được (vì không truy cập được đến **Vector**), cũng tương tự đối với các hàm thành viên của **Vector**. Do vậy, ta sẽ viết hàm bạn chung của cả hai lớp.

Chương trình dưới đây xây dựng 2 lớp **Vector**, **Matrix** và hàm bạn **Product(const Matrix &M, const Vector &v)** để nhân **M** với **v**. Nhắc lại, để nhân được **M** ($m \times n$) và **v**, số cột (**n**) của **M** phải bằng với số phần tử của **v** (**n**). Kết quả là một vectơ với số phần tử bằng số dòng (**m**) của **M**. Mỗi phần tử của vectơ kết quả là tích của vectơ dòng tương ứng của **M** và vectơ **v**.

```

1  #include <iostream>
2  using namespace std;
3  class Vector ;
4  class Matrix ;
5
6  class Vector
7  {
8  public:
9      void FillData();
10     void Display();
11     friend Vector Product(const Matrix &M, const Vector &v) ;
12 private:
13     int numElement;           // so phan tu cua vecto
14     double data[20];         // cac thanh phan cua vecto
15 };
16
17 class Matrix
18 {
19 public:
20     void FillData();
21     friend Vector Product(const Matrix &M, const Vector &v);
22 private:
23     int numRows, numCol;      // so dong, so cot
24     double data[20][20];     // cac phan tu cua ma tran
25 };
26
27 void Vector::FillData()
28 {
29     cout << "\nEnter number of vector elements: "; cin >> numElement ;
30     cout << "Enter elements of vector:\n";
31     for (int index = 1; index <= numElement; ++index)
32         cin >> data[index];
33 }
34
35 void Matrix::FillData()
36 {
37     cout << "\nEnter number of rows and columns of matrix: "; cin >> numRows >> numCol
38         ;
39     cout << "Enter elements of matrix:\n";
40     for (int index1 = 1; index1 <= numRows ; ++index1)
41     for (int index2 = 1; index2 <= numCol ; ++index2)
42         cin >> data[index1][index2];
43 }
44 void Vector::Display()
45 {
46     cout << "\nElements of vector is:\n";
47     for (int index = 1; index <= numElement; ++index)
48         cout << data[index] << " ";
49     cout << endl;
50 }
51
52 Vector Product(const Matrix &M, const Vector &v)
53 {

```

```

54     Vector res;
55     res.numElement = M.numRow; // so ptu cua vecto tich la so dong cua M
56     for (int index = 1; index <= M.numRow; ++index)
57     {
58         res.data[index] = 0;
59         for (int k = 1; k <= M.numCol; ++k)
60             res.data[index] += M.data[index][k] * v.data[k];
61     }
62     return res;
63 }
64
65 int main()
66 {
67     Matrix Mat;
68     Vector vec;
69     Vector result; // result = M * v
70     Mat.FillData();
71     vec.FillData();
72     result = Product(Mat, vec);
73     result.Display();
74
75     return 0;
76 }

```

Hình 6.30: Hàm bạn chung của Vector và Matrix.

Vì các kiểu Vector và Matrix xuất hiện trong định nghĩa (trong dòng tiêu đề của hàm Product) của cả hai lớp nên ta cần khai báo tên hai lớp này trước khi định nghĩa chúng. Để ngắn gọn, trong cài đặt 2 lớp ta tạm lược bớt các thành viên chưa cần thiết như các hàm tạo, hàm in ma trận ...

6.2.7 Tạo các phép toán cho lớp (hay tạo chồng phép toán - Operator Overloading)

Chúng ta hãy xem lại ví dụ về hàm cộng 2 số phức trong mục 6.2.6. Hàm có thể được định nghĩa như hàm thành viên với lời gọi $z = x.Plus(y)$ hoặc trực quan, quen thuộc hơn với lời gọi $z = Plus(x, y)$ nếu Plus được định nghĩa dưới dạng hàm bên ngoài và là bạn của lớp. Thậm chí sẽ là gần gũi nhất nếu ta có thể viết được $z = x + y$.

Vấn đề này được giải quyết một cách đơn giản bằng cách chỉ cần thay đổi tên hàm bạn (Plus) bằng tên operator+. Nói cách khác, các kí hiệu phép toán (+, -, *, /, %, =, ==, >, >=, <<, >> ...) đều có thể được sử dụng đi kèm cùng từ khóa operator để định nghĩa thành phép toán của lớp (thực tế ta đã thấy cùng một kí hiệu phép toán có thể dùng với những kiểu dữ liệu khác nhau, như phép - theo nghĩa hiệu của hai số hoặc đảo dấu của một số, phép >> là toán tử nhập dữ liệu hoặc đẩy bit sang phải ...). Nội dung của phép toán được định nghĩa là do các câu lệnh trong hàm quyết định (ví dụ dùng dấu + để tính “hiệu” của hai đối tượng !!!), tuy nhiên thông thường ta nên cài đặt nội dung phù hợp với ý nghĩa của kí hiệu đã quen dùng và cũng để phù hợp với các qui định mặc nhiên của C++ như tính ưu tiên của các phép toán chẳng hạn.

Các hàm toán tử này để thuận lợi thường được khai báo dưới dạng hàm bạn của lớp. Với phép toán một ngôi hàm có một đối và phép toán hai ngôi hàm có hai đối, đối thứ nhất ứng với toán hạng thứ nhất, đối thứ hai ứng với toán hạng thứ hai. Do vậy, với các phép toán không giao hoán (ví dụ

phép -) thì thứ tự đối là quan trọng.

Dưới đây chúng ta sẽ minh họa việc xây dựng lớp ngày tháng với tương đối đầy đủ một số phép toán quen dùng.

```

1 #include <iostream>
2 using namespace std;
3
4 // number of days of months
5 const int NUM_DAYS[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
6 class Date
7 {
8 public:
9     Date() {day = month = year = 1; }
10    Date(int new_day, int new_month, int new_year) {day = new_day; month = new_month;
        year = new_year; }
11    friend long DtoN(Date date);
12    friend Date NtoD(long num_days);
13    friend void DoW(Date date, char dow[]);
14    friend void Moon_Year(Date date, char mc[]);
15    friend long operator-(Date date1, Date date2);
16    friend Date operator-(Date date, long n);
17    friend Date operator+(Date date, long n);
18    friend bool operator==(Date date1, Date date2);
19    friend istream& operator>>(istream& is, Date& date);
20    friend ostream& operator<<(ostream& os, Date date);
21    friend Date operator++(Date& date);
22    friend Date operator--(Date& date);
23    friend Date operator++(Date& date, int n);
24    friend Date operator--(Date& date, int n);
25 private:
26     int day, month, year ;
27 };

```

Hình 6.31: Khai báo toán tử cho lớp Date.

Trong khai báo trên so với bản Date đã viết trước đây, ta đã lược bớt các hàm Set dùng để thiết lập giá trị cho Date và thay bằng hàm khởi tạo có đối Date(int, int, int). Tương tự hàm Display() cũng được thay thế bằng phép toán hiển thị << và hàm Distance_Date(Date date1, Date date2) thay bằng phép toán trừ.

Để thuận lợi trong cài đặt, tất cả các hàm và phép toán còn lại đều được khai báo là hàm bạn của Date.

Các hàm bissextile_year (kiểm tra năm nhuận) và num_days_of_month (tính số ngày của một tháng) được khai báo độc lập bên ngoài lớp như những hàm phục vụ thông thường, không phải là thành viên của lớp.

Các hàm DtoN, NtoD, DoW, Moon_Year được cài đặt như phiên bản cấu trúc đã được đề cập trong mục kiểu dữ liệu cấu trúc. Thuật toán cho hàm Moon_Year() (đổi năm dương lịch sang năm âm lịch) tương tự như thuật toán tìm thứ của ngày tháng (DoW).

Các phép toán còn lại gồm có:

- Phép trừ giữa 2 date cho lại khoảng cách của 2 date này.

- Phép trừ của một `date` với một số nguyên sẽ cho lại là một `date` mới.
- Phép cộng của một `date` với một số nguyên sẽ cho lại là một `date` mới. Thực chất hai phép toán cộng, trừ với một số nguyên có thể chỉ cần cài đặt một (khi gọi số nguyên có thể âm hoặc dương).
- Phép toán so sánh trả lại giá trị đúng khi cả 3 thành phần của 2 `date` cùng bằng nhau và sai khi ngược lại.
- Cần lưu ý các phép toán vào/ra

```

1 istream& operator>>(istream& is, Date& date)           // Phép toán nhập
2 {
3     int dd, mm, yy;
4     cin >> dd >> mm >> yy;
5     date = Date(dd, mm, yy);
6     return is;
7 }
8
9 ostream& operator<<(ostream& os, Date date)           // Phép toán hiển thị
10 {
11     os << date.day << "/" << date.month << "/" << date.year ;
12     return os;
13 }

```

Hình 6.32: Nhập - xuất cho lớp `Date`.

.

Các hàm này có 2 đối: Một đối để chỉ dòng nhập/xuất thuộc vào các lớp tương ứng (`istream` hoặc `ostream`), đối thứ 2 để chỉ `date` cần thao tác, trong hàm nhập đối này phải là đối tham chiếu. Tương tự, các đối dòng nhập xuất phải được khai báo tham chiếu. Các hàm trên có giá trị trả lại là các tham chiếu đến dòng nhập/xuất tương ứng. Việc trả lại tham chiếu này cho phép ta có thể gọi nối tiếp các phép toán, ví dụ: `cin >> a >> b >> c ...`

- Các phép toán tự tăng, giảm được viết thành 2 hàm, tương ứng với tăng (giảm) trước và sau.

```

1 Date operator++(Date& date)
2 {
3     long new_days = DtoN(date);
4     new_days++;
5     date = NtoD(new_days);
6     return date;
7 }
8
9 Date operator++(Date &date, int n)
10 {
11     Date old_date = date;
12     ++date;
13     return old_date;
14 }

```

Hình 6.33: Các phép toán tăng giảm cho lớp `Date`.

Trong cả 2 dạng `date` cần được khai báo tham chiếu vì giá trị sẽ thay đổi. Ngoài ra, cả hai hàm đều trả lại `date` kết quả (để tham gia vào các biểu thức). Với tăng trước giá trị trả lại là `date` đã tăng 1, với tăng sau giá trị trả lại vẫn là `date` cũ (chưa tăng). Ngoài ra trong danh sách đối, với tăng trước vẫn được viết bình thường, còn tăng sau để phân biệt trong danh sách đối có thêm một đối nguyên (không sử dụng).

Toàn bộ cài đặt lớp `Date` được cho trong phụ lục B1. Phụ lục B2 trình bày một ví dụ khác về cài đặt lớp đa thức với các phép toán $+$ (cộng), $-$ (trừ hoặc đảo dấu), $*$ (nhân), $>>$ (nhập), $<<$ (xuất) và tính giá trị các đa thức. Chương trình sẽ nhập 4 đa thức: P , Q , R , S . Sau đó tính đa thức: $F = -(P + Q) * (R - S)$. Cuối cùng tính giá trị $F(x)$, với x là một số thực nhập từ bàn phím.

Chú ý:

- Khi dùng các hàm toán tử như phép toán của C++, ta có thể kết hợp nhiều phép toán để viết các công thức phức tạp. Cũng cho phép dùng dấu ngoặc tròn để quy định thứ tự thực hiện các phép tính. Thứ tự ưu tiên của các phép tính vẫn tuân theo các quy tắc ban đầu của C++. Chẳng hạn các phép $*$ và $/$ có thứ tự ưu tiên cao hơn so với các phép $+$ và $-$.
- Việc định nghĩa các hàm toán tử (chồng phép toán) thực chất không chỉ sử dụng cho lớp, mà còn được sử dụng độc lập, ví dụ như cho các kiểu dữ liệu mảng, xâu, cấu trúc chẳng hạn ...

6.3 Dạng khuôn mẫu hàm và lớp

Thông thường một thuật toán được dùng để giải quyết cùng một vấn đề với nhiều kiểu dữ liệu khác nhau. Tuy nhiên, trong lập trình cổ điển với mỗi kiểu dữ liệu ta phải viết một hàm khác nhau (chỉ để khai báo kiểu đối hoặc kiểu giá trị trả lại của hàm) để thể hiện thuật toán. Điều này gây lãng phí công sức, do vậy C++ cho phép định nghĩa một tên gọi làm kiểu mẫu (template) và hàm được viết theo tên mẫu này. Khi gọi hàm các tên kiểu thực sự sẽ thay thế tên mẫu trước khi hàm thực hiện. Nói cách khác, lúc này kiểu cũng là tham đối của hàm mẫu. Tóm lại, ta có thể viết một hàm với kiểu mẫu để dùng chung cho các kiểu dữ liệu khác nhau. Các hàm viết như vậy được gọi là khuôn mẫu hàm (hoặc hàm mẫu) và đã được trình bày trong mục 4.6.2 của chương 4.

Cũng tương tự, thông thường các thành viên của lớp (dữ liệu và hàm) cũng được khai báo cố định trước về kiểu. Kỹ thuật khuôn mẫu nói trên cho phép ta xây dựng các lớp tổng quát hơn với kiểu cũng là kiểu mẫu và khi sử dụng lớp các kiểu mẫu này được thay thế bằng các kiểu cụ thể tùy theo đối tượng.

6.3.1 Khai báo một kiểu mẫu

Để dùng một tên mẫu thay cho kiểu cụ thể, ta cần định nghĩa trước bằng cú pháp:

```
template <class T> // T: tên kiểu mẫu
```

hoặc

```
template <typename T>
```

việc dùng từ khóa `typename` hoặc `class` là như nhau, việc dùng từ khóa nào tùy thuộc sở thích, thói quen của lập trình viên. Trong giáo trình, từ phần này trở đi, ta sẽ sử dụng từ khóa `class`. (Chú

ý: nghĩa của từ khóa `class` ở đây không liên quan gì đến lớp và các đối tượng (T không phải là một lớp), nó chỉ đơn thuần là danh từ chung để chỉ một loại, kiểu, lớp, họ, tập ...).

6.3.2 Sử dụng kiểu mẫu

Dưới đây là ví dụ về một lớp đặc tả các cặp giá trị (kiểu tổng quát T). Mỗi đối tượng của lớp có 2 giá trị cùng kiểu. Ngoài hàm tạo không đối, lớp có 4 phương thức :

- Phương thức `Display` (để hiển thị đối tượng ra màn hình) không đối, không giá trị trả lại. Được cài đặt trực tiếp bên trong khai báo lớp.
- Phương thức `Set` (để gán giá trị cho đối tượng) có hai đối kiểu T. Không giá trị trả lại. Được cài đặt trực tiếp bên trong khai báo lớp.
- Phương thức `getMax` (để lấy giá trị lớn nhất của đối tượng) không đối, có giá trị trả lại kiểu T. Được cài đặt bên ngoài khai báo lớp.
- Phương thức `Swap` (để trao đổi hai giá trị của đối tượng) không đối, không giá trị trả lại. Được cài đặt bên ngoài khai báo lớp.

Chương trình tạo 2 đối tượng nguyên và kí tự bằng các lệnh khai báo:

```
Pair<int> my_object; và Pair<char> your_object;
```

Sau khi khởi tạo, chương trình in giá trị lớn nhất của mỗi đối tượng và thứ tự trao đổi giữa chúng.

```
1 #include <iostream>
2 using namespace std;
3
4 template <class T>
5 class Pair
6 {
7 public:
8     Pair() { value1 = value2 = 0; }
9     void Set(T first, T second) {value1 = first; value2 = second; }
10    T getMax() ;
11    void Swap();
12    void Display() { cout << "(" << value1 << ", " << value2 << ")" ; }
13 private:
14    T value1, value2;
15 };
16
17 template <class T>
18 T Pair<T>::getMax()
19 {
20     T res;
21     res = (value1 > value2) ? value1 : value2;
22     return res;
23 }
24
25 template <class T>
26 void Pair<T>::Swap()
27 {
```



```

28     T temp;
29     temp = value1; value1 = value2; value2 = temp;
30 }
31
32 int main()
33 {
34     Pair<int> my_object; my_object.Set(3, 5);
35     cout << "My pair of integers is " ; my_object.Display() ; cout << " and " <<
        my_object.getMax() << " is greatest. ";
36     my_object.Swap();
37     cout << "They was swapped to " ; my_object.Display() ; cout << endl;
38
39     Pair<char> your_object; your_object.Set('B', 'D');
40     cout << "Your pair of characters is " ; your_object.Display() ; cout << " and "
        << your_object.getMax() << " is greatest. ";
41     your_object.Swap();
42     cout << "They was swapped to " ; your_object.Display() ; cout << endl;
43
44
45     return 0;
46 }

```

Hình 6.34: Mẫu lớp Pair.

Qua ví dụ trên, ta rút ra một số nguyên tắc:

- Cần khai báo tên kiểu mẫu (`template <class T>`) trước khi định nghĩa lớp.
- Kiểu mẫu này (T) thay thế cho các vị trí xuất hiện của các kiểu cụ thể mà lập trình viên muốn tổng quát hóa thành kiểu mẫu (trừ các biến cố định không liên quan như các biến đếm, biến chỉ số là kiểu nguyên, kiểu của số π là `double ...`). Ngoài ra, mọi yếu tố khác không bị thay đổi như khi viết với kiểu cụ thể.
- Bất kỳ phương thức nào được cài đặt bên ngoài định nghĩa lớp (cũng giống như hàm thông thường) có 2 bổ sung cần chú ý: Trước mỗi phương thức đều có khai báo mẫu (ví dụ: `template <class T>`) và tên lớp phải đi kèm kiểu mẫu <T> (ví dụ: `void Pair<T>::Swap()`).
- Từ tên lớp với kiểu mẫu `Pair<T>`, ta có thể “chuyên biệt hóa” tên lớp này thành nhiều lớp khác nhau với tên kiểu cụ thể như `Pair<int>`, `Pair<char>` để sử dụng vào các mục đích cụ thể, như khai báo một đối tượng (`Pair<int> my_object;` hoặc `Pair<char> your_object;`), khai báo một tham đối đối tượng như `int Sum(const Pair<int> &pair_obj)`.

Các kiểu (lớp) chuyên biệt hóa này cũng có thể được đặt thành tên kiểu mới để thuận lợi cho sử dụng như:

```

typedef Pair<char> Pair_of_Characters;    // đặt lại tên mới cho Pair<char>

```

Để tránh sai sót, nói chung ta nên viết trước chương trình với một kiểu cụ thể nào đó. Sau khi chương trình đã hoàn thiện ta tiến hành chuyển kiểu sang kiểu mẫu bằng cách áp dụng các nguyên tắc trên.

6.3.3 Một số dạng mở rộng của khai báo mẫu

Trong mỗi khai báo `template`, ta có thể nhận thấy sự tương tự về mặt hình thức giữa kiểu mẫu (T) và tham đối trong các khai báo hàm. Điểm khác biệt ở đây là đối của các hàm là các giá trị, còn “đối” của các `template` là các kiểu và kể cả là các hàm (T được xem là các tham đối hình thức còn `int`, `char` ... là các tham đối thực sự). Dựa trên liên tưởng này ta cũng thấy “danh sách đối” của `template` cũng đa dạng như danh sách đối của hàm, với các dạng mở rộng như:

```
template <class T> : đối là một kiểu
template <class T, class U> : đối gồm hai kiểu (xem thêm phần khuôn mẫu hàm, mục 4.6.3)
template <class T, int N> : một kiểu và một số
template <class T = char> : một kiểu được mặc định là char
template <double Tfunction(int)> : một hàm
...
```

Việc sử dụng chi tiết các khai báo mẫu trên đây, người đọc có thể tham khảo thêm trong các sách chuyên sâu về C++.

Bài tập

1. Trong các khởi tạo giá trị cho các cấu trúc sau, khởi tạo nào đúng:

```
struct S1 {  
    int day, month, year;  
} var1 = {2, 3} ;  
struct S2 {  
    char name[10];  
    S1 birthday;  
} var2 = {"LyLy", 1, 2, 3};  
struct S3 {  
    S2 student;  
    double mark;  
} var3 = {{{"Coccoc", {4,5,6}}, 7};
```

- (a) S1 và S2 đúng.
 - (b) S2 và S3 đúng.
 - (c) S3 và S1 đúng.
 - (d) Cả 3 cùng đúng.
2. Đối với kiểu cấu trúc, cách gán nào dưới đây là không được phép:
- (a) Gán hai biến cho nhau.
 - (b) Gán hai phần tử mảng (kiểu cấu trúc) cho nhau.
 - (c) Gán một phần tử mảng (kiểu cấu trúc) cho một biến (cấu trúc) và ngược lại.
 - (d) Gán hai mảng cấu trúc cùng số phần tử cho nhau.
3. Cho số phức dưới dạng cấu trúc gồm 2 thành phần là thực và ảo. Viết chương trình nhập 2 số phức và in ra tổng, tích, hiệu, thương của chúng.
4. Cho phân số dưới dạng cấu trúc gồm 2 thành phần là tử và mẫu. Viết chương trình nhập 2 phân số, in ra tổng, tích, hiệu, thương của chúng dưới dạng tối giản.
5. Tính số ngày đã qua kể từ đầu năm cho đến ngày hiện tại. Qui ước ngày được khai báo dưới dạng cấu trúc và để đơn giản một năm bất kỳ được tính 365 ngày và tháng bất kỳ có 30 ngày.
6. Nhập một ngày tháng năm dưới dạng cấu trúc. Tính chính xác (kể cả năm nhuận) số ngày đã qua kể từ ngày 1/1/1 cho đến ngày đó.
7. Tính khoảng cách giữa 2 ngày tháng bất kỳ.
8. Hiện thứ của một ngày bất kỳ nào đó, biết rằng ngày 1/1/1 là thứ hai.
9. Hiện thứ của một ngày bất kỳ nào đó, biết rằng ngày 1/1/2000 là thứ hai.
10. Viết chương trình nhập một mảng sinh viên, thông tin về mỗi sinh viên gồm họ tên và ngày sinh (kiểu cấu trúc). Sắp xếp mảng theo tuổi và in ra màn hình.

11. Hãy nêu ý kiến khi tất cả các thành viên của một lớp đều được khai báo public: ?, private: ?

Các bài tập dưới đây đều sử dụng chung các định nghĩa của lớp Bicycle và Automobile

12. Cho định nghĩa lớp Bicycle như sau:

```
class Bicycle
{
public:
    void set(double the_price, char the_color); // thiết lập các giá trị tương ứng
    cho đối tượng
    double get_price( ); // trả lại giá tiền của đối tượng
    char get_color( ); // trả lại màu của đối tượng
private:
    double price;
    char color;
};
```

Hãy:

- Bổ sung thêm khai báo hàm tạo không đối vào lớp trên.
 - Cài đặt các hàm tạo, set và get của lớp.
13. Cho lớp Bicycle như trong bài tập trên và khai báo các đối tượng:

```
Bicycle favorite, peugeot;
```

Các câu lệnh nào sau đây được phép / không được phép ?

```
favorite.color = 'G'
peugeot.set(3000, 'B')
cout << peugeot.price;
cout << get_color(favorite);
cout << peugeot.get_color();
union = favorite ;
```

14. Để thay câu lệnh `peugeot.set(3000, 'B')` bằng `Bicycle peugeot(3000, 'B')`, trong khai báo của Bicycle cần thêm hàm gì ? Hãy viết ra hàm đó.
15. Hãy cài đặt hàm (bên ngoài lớp) so sánh màu của 2 xe đạp với nhau (lớp Bicycle trong bài tập trên).
16. Giả sử lớp ô tô được khai báo

```
class Automobile
{
public:
    void set(double the_price, char the_color); // thiết lập các giá trị tương ứng
    cho đối tượng
    double get_price( ); // trả lại giá tiền của đối tượng
    char get_color( ); // trả lại màu của đối tượng
private:
    double price;
    char color;
} mercedes, kia_morning;
```

Hãy cài đặt hàm (bên ngoài lớp) so sánh giá tiền của xe đạp (lớp `Bicycle`) và xe ô tô bất kỳ (lớp `Automobile`).

17. Hãy cài đặt hàm bạn (và điều chỉnh các khai báo `Bicycle`, `Automobile` một cách thích hợp) để tính tỉ lệ giữa giá tiền của một chiếc xe đạp (lớp `Bicycle`) và một chiếc xe ô tô bất kỳ (lớp `Automobile`).

18. Các câu lệnh nào dưới đây được phép / không được phép ?

```
if (mercedes.get_price() == kia_morning.get_price())  
    cout << "Very good" ;  
if (mercedes == kia_morning)  
    cout << "Wow" ;
```

19. Hãy cài đặt phép toán so sánh (kí hiệu `==`) cho lớp `Automobile`.

20. Hãy cài đặt các phép toán vào/ra (kí hiệu `>>`, `<<`) cho lớp `Automobile`.

Chương 7

Con trỏ và bộ nhớ

Hiểu đúng và sử dụng đúng con trỏ là rất quan trọng trong lập trình trên C++. Có 3 lý do chính cho vấn đề này. Thứ nhất, con trỏ cung cấp cách thức hữu hiệu để truy cập vào bộ nhớ máy tính. Thứ hai, con trỏ hỗ trợ cấp phát bộ nhớ động. Và cuối cùng, con trỏ có thể cải thiện tính hiệu quả của chương trình. Con trỏ là một trong những đặc tính mạnh nhất nhưng cũng chứa nhiều rủi ro nếu sử dụng không đúng cách. Ví dụ việc không khởi tạo con trỏ có thể làm chương trình bị đổ vỡ. Nó cũng là nguyên nhân gây ra nhiều lỗi khác mà rất khó tìm ra. Trong chương này, chúng ta sẽ đề cập đến khái niệm và các vấn đề về con trỏ trong C++.

7.1 Khái niệm con trỏ

Con trỏ là một biến mà lưu giữ địa chỉ của bộ nhớ. Địa chỉ này là vị trí của một biến khác trong bộ nhớ. Nhớ lại rằng, bộ nhớ máy tính được chia thành các ô nhớ (theo bytes) được địa chỉ hóa và các biến được lưu giữ như dãy các ô nhớ liên tiếp tùy theo kích cỡ của chúng. Ví dụ nếu một biến chứa địa chỉ của một biến khác thì biến đầu tiên được xem như là trỏ đến biến thứ hai. Hình 7.1 minh họa vấn đề này.

Địa chỉ “trỏ” tới biến bởi vì nó xác định nơi chứa biến đó, thay vì nói tên của biến là gì. Một biến lưu tại vị trí có địa chỉ 1003 có thể được trỏ bởi biến giá trị bằng 0x1003 trong bộ nhớ.

7.2 Biến con trỏ

Con trỏ được lưu trong biến và gọi là biến con trỏ. Khai báo con trỏ bao gồm kiểu cơ sở, dấu * và tên biến. Dạng tổng quát như sau:

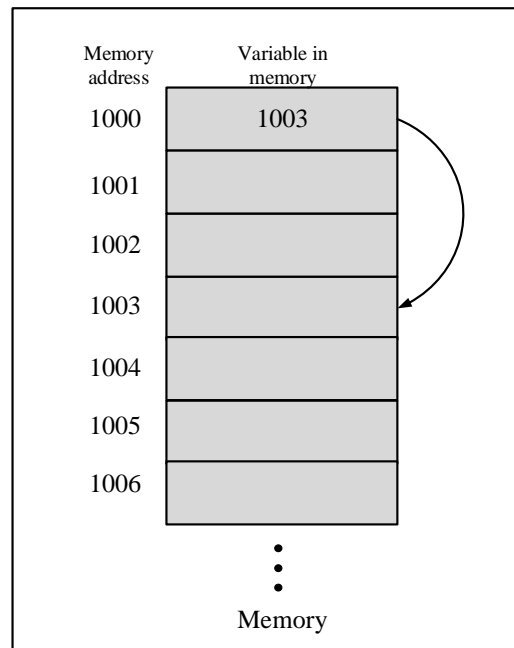
```
type_name *name1, * name2, ...
```

trong đó `type_name` là kiểu cơ sở của con trỏ và có thể là bất cứ kiểu nào đó mà hợp lệ. Tên của biến con trỏ được chỉ rõ bằng `name1`, `name2`. Dấu * phải được đặt trước tên mỗi biến con trỏ.

Ví dụ:

```
double *pointer1, *pointer2;
```

Biến `pointer1` và `pointer2` được khai báo là con trỏ để trỏ tới biến kiểu số `double` và như vậy nó không thể sử dụng như con trỏ để trỏ tới các biến kiểu khác như `int` hoặc `char`. Mỗi kiểu của biến



Hình 7.1: Biến trỏ đến biến khác trong bộ nhớ.

yêu cầu kiểu con trỏ khác nhau tương ứng.

```
int *p1, *p2, v1, v2, a[50];
```

Ở đây, dấu `*` phải đặt trước mỗi biến con trỏ. Trong khai báo trên, `p1`, `p2` là các con trỏ trỏ đến biến kiểu nguyên, còn `v1`, `v2` là các biến nguyên thông thường và `a` là mảng nguyên.

Sau khi khai báo con trỏ, để trỏ đến biến nào đó (ví dụ cho `p1` trỏ đến `v1`), ta dùng cú pháp:

```
p1 = &v1;
```

Ở đây, `&` là phép toán lấy địa chỉ. Câu lệnh trên lấy địa chỉ của `v1` và đặt vào `p1` (`p1` chứa địa chỉ của `v1`, tức `p1` trỏ đến `v1`). Lưu ý, `p1` phải là con trỏ cùng kiểu với `v1` (`int`).

Chúng ta có 2 cách để tham chiếu đến `v1`. Chúng ta có thể gọi `v1` hoặc gọi thông qua con trỏ của nó (`p1`) bằng `*p1`. Đây cũng là dấu `*` chúng ta sử dụng để khai báo `p1` nhưng ở đây nó có ý nghĩa khác. Khi dấu `*` sử dụng theo cách này, nó thường được gọi là toán tử tham chiếu lại (dereferencing). Việc sử dụng `v1` hay `*p1` là tương đương, tức có thể thay `v1` bằng `*p1` và ngược lại tại bất kỳ đâu trong biểu thức.

Trong C++, cách mà thông qua biến con trỏ `p1` thì giá trị `v1` chính là `*p1`. Đây cũng là dấu `*` chúng ta sử dụng để khai báo `p1` nhưng ở đây nó có ý nghĩa khác. Khi dấu `*` sử dụng theo cách này, nó thường được gọi là toán tử tham chiếu lại (dereferencing).

Ví dụ:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

Đoạn mã này sẽ cho kết quả như sau:

```
42
42
```


Chương trình trong hình 7.2 sẽ minh họa việc sử dụng và khai báo con trỏ. Trình bày cách in ra màn hình giá trị con trỏ và giá trị mà con trỏ trỏ tới.

```

1 //Chương trình minh họa việc thao tác với con trỏ
2 #include <iostream>
3 using namespace std;
4 int main(void)
5 {
6     int x = 100;
7     int *p1, *p2;
8     p1 = &x;
9     p2 = p1;
10    cout << p2 << endl; // in ra địa chỉ của x, không phải giá trị x
11    cout << *p2; // in ra giá trị x
12    return 0;
13 }
```

Hình 7.2: Sử dụng và khai báo con trỏ.

Output chương trình Hình 7.2:

```

0x22ff44
100
```

Con trỏ cũng có thể trỏ đến mảng bằng cách gán địa chỉ mảng (cũng là địa chỉ phần tử đầu tiên của mảng) cho con trỏ. Vì tên mảng là địa chỉ của mảng nên câu lệnh gán sẽ là:

```

p2 = a;        // không có dấu &
// hoặc
p2 = &a[0];    // có dấu &
```

Việc tham chiếu đến phần tử mảng a thông qua con trỏ được nói đến trong phần sau.

Phép toán đối với con trỏ

Các phép toán dưới đây với 2 con trỏ cùng tham gia sẽ chỉ làm việc được nếu 2 con trỏ có cùng kiểu cơ sở (để ngắn gọn từ đây trở đi ta gọi là hai con trỏ cùng kiểu).

Phép gán

Hai con trỏ cùng kiểu có thể gán được cho nhau bằng phép toán gán (=). Hiệu quả là hai con trỏ này cùng trỏ đến một nơi trong bộ nhớ.

Ví dụ:

```

double *p1, *p2, v;
p1 = &v;
p2 = p1;
v = 3.14;
*p1 = v * v;
cout << *p2;           // 9.8596
```

Phép toán số học

Chỉ có 2 phép toán số học đối với con trỏ là: cộng và trừ. Để hiểu phép toán số học thao tác với con trỏ, ký hiệu p1 là con trỏ kiểu short và giá trị hiện thời của nó là 2000. Giả sử kiểu short là 2 bytes, khi đó, câu lệnh sau:

```
p++
```

thì `p1` sẽ là 2002, không phải là 2001 vì `p1` tăng lên theo kích thước kiểu mà con trỏ trỏ đến. Nghĩa là `p1` sẽ trỏ tới số kiểu `short` tiếp theo. Điều này cũng tương tự như giảm (trừ) con trỏ. Ví dụ như trên, `p1` đang có giá trị là 2000 thì câu lệnh:

```
p1--
```

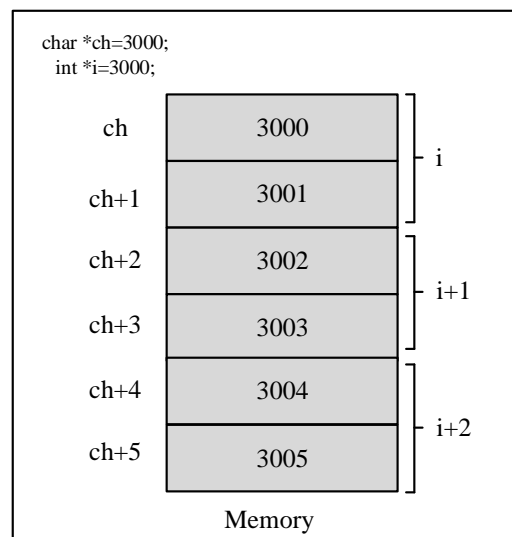
cho giá trị của `p1` là 1998, trỏ tới số kiểu `short` đứng kế trước số hiện tại.

Từ ví dụ trên, một cách tổng quát, các quy tắc sau đây áp dụng cho con trỏ số học. Mỗi khi một con trỏ được tăng lên, nó trỏ đến vị trí bộ nhớ của phần tử tiếp theo cùng kiểu với kiểu con trỏ. Mỗi lần nó được giảm đi, nó chỉ đến vị trí của các phần tử trước đó cùng kiểu với kiểu của con trỏ. Tất cả các con trỏ khác sẽ tăng hoặc giảm theo kích thước của các kiểu dữ liệu mà chúng trỏ đến. Cách tiếp cận này đảm bảo rằng một con trỏ luôn trỏ đến một phần tử thích hợp cùng kiểu dữ liệu với nó.

Hình 7.3 minh họa khái niệm này. Chúng ta không giới hạn cho các toán tử tăng và giảm. Ví dụ, chúng ta có thể thêm hoặc trừ đi các số nguyên hoặc từ con trỏ. Ví dụ, câu lệnh:

```
p1 = p1 + 12;
```

thì `p1` trỏ đến phần tử thứ 12 với kiểu trùng với kiểu của `p1`.



Hình 7.3: Phép toán số học với con trỏ liên quan đến kiểu của con trỏ.

Như vậy, phép toán tăng, giảm con trỏ cho phép làm việc thuận lợi trên mảng, vì mảng là dãy phần tử cùng kiểu sắp xếp liên tục trong bộ nhớ. Nếu con trỏ đang trỏ đến mảng (tức đang chứa địa chỉ đầu tiên của mảng), việc tăng con trỏ lên 1 đơn vị sẽ dịch chuyển con trỏ trỏ đến phần tử thứ hai hoặc nếu con trỏ `p` đang ở phần tử thứ `k` trong mảng thì `p + h` sẽ trỏ đến phần tử thứ `k + h` của mảng.

Từ đó, ta có thể cho con trỏ chạy từ đầu đến cuối mảng bằng cách tăng con trỏ lên từng đơn vị như trong câu lệnh `for` dưới đây chỉ sử dụng con trỏ để in các phần tử của mảng.

Ví dụ:

```
int a[5] = { 1, 2, 3, 4, 5 }, *p;
p = a;    // hoặc p = &a[0], cho p trỏ đến mảng a
for (int index = 0; index < 5; index++)
{
    cout << *(p+index) ; // in a
```

```
}
```

hoặc cho p chạy từ đầu đến cuối mảng

```
for (int index = 0; index < 10; index++)
{
    cout << *p ;
    p++;
}
```

Thông qua ví dụ trên, nếu p là con trỏ trỏ đến mảng a, ta có thể đồng nhất:

```
a[i] = p[i] = *(p + i) = *(a + i)
```

Việc cộng 2 con trỏ là vô nghĩa, tuy nhiên phép trừ của hai con trỏ p và q là được phép khi p và q là 2 con trỏ cùng trỏ đến các phần tử của một dãy dữ liệu nào đó trong bộ nhớ (ví dụ cùng trỏ đến 1 mảng dữ liệu). Khi đó, hiệu p - q là số thành phần giữa p và q (không phải là hiệu của 2 địa chỉ). Ví dụ nếu p = q + 12 thì p - q = 12. Nếu p và q là 2 con trỏ short, p có địa chỉ 200 và q có địa chỉ 208. Khi đó p - q = 4 và q - p = 4 (4 là số thành phần short từ địa chỉ 200 đến 208).

Phép toán số học

Các phép toán so sánh cũng được áp dụng đối với con trỏ, thực chất là so sánh giữa địa chỉ của hai nơi được trỏ bởi các con trỏ này. Thông thường, các phép so sánh <, <=, >, >= chỉ áp dụng cho hai con trỏ trỏ đến phần tử của cùng một mảng dữ liệu nào đó. Thực chất của phép so sánh này chính là so sánh chỉ số của 2 phần tử được trỏ bởi 2 con trỏ đó.

```
double a[10], *p, *q ;
p = a ;                // p trỏ đến mảng (tức p trỏ đến a[0])
q = &a[3] ;            // q trỏ đến phần tử thứ 3 (a[3])
cout << (p < q) ;      // 1
cout << (p + 3 == q) ; // 1
cout << (p > q - 1) ;   // 0
cout << (p >= q - 2) ;  // 0
for (p=a ; p <a+100; p++) cout << *p ; // in toàn bộ mảng a
```

Chú ý: Như các phần trước đã đề cập tên của một mảng (ví dụ mảng a ở trên) chính là địa chỉ của mảng đó (cũng là địa chỉ của phần tử đầu tiên a[0]). Do vậy: - a + k là địa chỉ của phần tử a[k], a + 100 là địa chỉ phần tử a[100] - *a là giá trị của phần tử a[0], *(a + k) là giá trị của phần tử a[k] - Nếu p là con trỏ trỏ đến a thì p và a chứa cùng giá trị (cùng là địa chỉ của phần tử a[0]), tuy nhiên p là một biến, có thể thay đổi (ví dụ tăng p thành p++ thì p sẽ trỏ đến phần tử a[1]) còn a là một hằng, a + 1 sẽ là địa chỉ của a[1] nhưng không thể tăng a như tăng p (ví dụ không thể viết a++) hoặc có thể đặt lại p = p + 100 nhưng không thể đặt lại a = a + 100.

7.3 Cấp phát bộ nhớ động

Khi tiến hành chạy chương trình, chương trình dịch sẽ bố trí các ô nhớ cụ thể cho các biến được khai báo trong chương trình. Vị trí cũng như số lượng các ô nhớ này tồn tại và cố định trong suốt thời gian chạy chương trình, chúng xem như đã bị chiếm dụng và sẽ không được sử dụng vào mục đích khác và chỉ được giải phóng sau khi chấm dứt chương trình. Việc phân bổ bộ nhớ như vậy được gọi là cấp phát tĩnh (cấp sẵn trước khi chạy chương trình và không thể thay đổi tăng, giảm kích thước hoặc vị trí trong suốt quá trình chạy chương trình). Ví dụ nếu ta khai báo một mảng

nguyên chứa 1000 số nguyên 4 bytes thì trong bộ nhớ sẽ có một vùng nhớ liên tục 4000 bytes để chứa dữ liệu của mảng này. Khi đó dù trong chương trình ta chỉ nhập vào mảng và làm việc với một vài số thì phần mảng rồi còn lại vẫn không được sử dụng vào mục đích khác. Đây là hạn chế thứ nhất của kiểu mảng. Ở một hướng khác, một lần nào đó chạy chương trình ta lại cần làm việc với hơn 1000 số nguyên. Khi đó, vùng nhớ mà chương trình dịch đã dành cho mảng là không đủ để sử dụng và chương trình sẽ bị lỗi. Đây chính là hạn chế thứ hai của mảng được khai báo trước.

Khắc phục các hạn chế trên của kiểu mảng, các ngôn ngữ lập trình cho phép chúng ta cấp phát động. Nghĩa là bộ nhớ sẽ được cấp phát đúng, đủ trong quá trình chạy chương trình theo yêu cầu của người sử dụng. Nhờ vậy, chúng ta có đủ số ô nhớ để làm việc mà vẫn tiết kiệm được bộ nhớ, và khi không dùng nữa ta có thể thu hồi (giải phóng) số ô nhớ này để chương trình sử dụng vào việc khác. Thông qua con trỏ ta có thể làm việc với bất kỳ địa chỉ nào của vùng được cấp phát. Cách thức bố trí bộ nhớ như vậy được gọi là cấp phát động.

Một vùng nhớ đặc biệt của bộ nhớ được gọi là heap là nơi dùng để cấp phát động. Bất cứ biến cấp phát động nào được tạo ra sẽ sử dụng vùng nhớ này. Khi vùng nhớ heap được sử dụng hết, việc yêu cầu cấp phát thêm sẽ bị lỗi.

Để cấp phát bộ nhớ cho biến trong vùng heap ta sử dụng toán tử `new`. Toán tử này tạo biến cấp phát động mới với kiểu được khai báo và trả về con trỏ trỏ đến biến mới này. Ví dụ, đoạn chương trình sau tạo biến cấp phát động mới với kiểu `MyType` và gán cho biến con trỏ `p` trỏ đến biến mới này:

```
MyType *p;
p = new MyType;
```

Trong C++ chuẩn, nếu không đủ bộ nhớ có sẵn trong vùng heap để tạo biến cấp phát động mới bằng toán tử `new` thì mặc định sẽ chấm dứt chương trình.

Kích cỡ của vùng nhớ heap thay đổi bởi máy tính và tùy thuộc vào sự thực hiện trong C++. Nó thường là lớn, và một chương trình khiêm tốn là không có khả năng sử dụng tất cả bộ nhớ trên heap. Tuy nhiên, thậm chí với những chương trình nhỏ, một thói quen tốt cho việc thực hành là nên giải phóng bộ nhớ đã được cấp phát động khi không còn dùng đến nó bằng toán tử `delete`. Lệnh dưới đây sẽ hủy vùng nhớ cấp phát động cho con trỏ `p` (trong câu lệnh `p = new MyType;` ở trên) và trả về vùng nhớ đã sử dụng bởi `p` cho vùng nhớ heap.

```
delete p;
```

Sau khi gọi toán tử `delete`, giá trị của `p` là không xác định và `p` được xem như biến chưa được khởi tạo.

7.4 Con trỏ và mảng động

7.4.1 Biến mảng và biến con trỏ

Trong phần trên, chúng ta đã thấy biến mảng thực sự là biến con trỏ. Ở đây, chúng ta sẽ thấy thêm tính hữu hiệu của công cụ con trỏ khi thao tác với mảng động. Mảng động là mảng mà kích cỡ của nó không cố định khi ta viết chương trình mà chỉ được xác định khi chương trình đang chạy.

Nhắc lại quan hệ về con trỏ và mảng. Cho hai biến `p` và `a` mô tả cùng kiểu biến :

```
int a[10];
int p;
```

Bởi vì a là con trỏ trỏ tới biến kiểu int nên giá trị của a có thể được gán cho biến con trỏ p như sau :

```
p = a ;
```

Sau phép gán này, p trỏ tới cùng bộ nhớ mà a trỏ, vì vậy p[0], p[1], ..., p[9] chính là các biến a[0], a[1], ..., a[9] tương ứng. Sử dụng ngoặc vuông và chỉ số với biến con trỏ hoàn toàn giống như biến mảng khi truy cập giá trị của mảng. Tuy nhiên, chúng ta không thể thay đổi giá trị của con trỏ mảng a :

```
int *p2 ;
a = p2; //Không hợp lệ
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main( )
5 {
6     int *p;
7     int a[10];
8
9     for (int i = 0; i < 10; i++)
10         a[i] = i;
11
12     p = a;
13
14     for (int index = 0; index < 10; index++)
15         cout << p[index] << " ";
16     cout << endl;
17
18     for (int index = 0; index < 10; index++)
19         p[index] = p[index] + 1;
20
21     for (int index = 0; index < 10; index++)
22         cout << a[index] << " ";
23     cout << endl;
24
25
26     return 0;
27 }
```

Hình 7.4: Sử dụng con trỏ như tên mảng.

Output chương trình Hình 7.5:

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

```
1 #include <iostream>
2 using namespace std;
3
4 int main( )
5 {
6     int *p;
7     int a[10];
8
```

```

9   for (int i = 0; i < 10; i++)
10      a[i] = i;
11
12   p = a;
13
14   for (int index = 0; index < 10; index++)
15      cout << p[index] << " ";
16   cout << endl;
17
18   for (int index = 0; index < 10; index++)
19      p[index] = p[index] + 1;
20
21   for (int index = 0; index < 10; index++)
22      cout << a[index] << " ";
23   cout << endl;
24
25
26   return 0;
27 }

```

Hình 7.5: Sử dụng con trỏ như tên mảng.

Output chương trình Hình 7.5:

```

0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10

```

7.4.2 Biến mảng động

Như đã thảo luận ở trên, đối với mảng được khai báo trước (tĩnh) chúng ta thường gặp phải vấn đề: viết chương trình nhưng không biết kích thước của mảng là bao nhiêu trước khi chương trình chạy. Ví dụ, mảng có thể là danh sách các sinh viên trong một lớp nhưng số sinh viên này có thể thay đổi trong thời gian chúng ta chạy chương trình. Chúng ta thường ước lượng kích thước lớn nhất có thể và hy vọng là đủ lớn để sử dụng. Vì vậy, nảy sinh hai vấn đề: thừa, lãng phí kích thước mảng quá lớn, không dùng hết hoặc thiếu kích thước mảng quá nhỏ, chương trình sẽ không thực hiện trong trường hợp số phần tử vượt quá kích thước đã khai báo trước của mảng.

Mảng động sẽ tránh được các vấn đề này. Khi chương trình sử dụng mảng động để lưu danh sách sinh viên thì số sinh viên trong lớp có thể là đầu vào của chương trình và mảng động sẽ được tạo chính xác với số sinh viên trên.

Tạo và sử dụng mảng động khá đơn giản.

Ví dụ, dòng lệnh sau sẽ tạo biến mảng động với 10 phần tử kiểu double:

```

double p;
p = new double [10];

```

Dĩ nhiên, mảng động không có tên mảng như đối với các mảng tĩnh thông thường. Tuy nhiên, ta có thể làm việc với mảng này thông qua con trỏ p trỏ đến nó.

Để có được mảng động với kiểu dữ liệu và/hoặc kích thước khác, chỉ đơn giản là thay thế kiểu double với kiểu dữ liệu mong muốn và/hoặc thay thế 10 với kích thước mong muốn.

Chú ý, khi chúng ta gọi toán tử `new`, kích thước của mảng động được cho trong ngoặc vuông theo sau kiểu dữ liệu, như ví dụ trên là kiểu dữ liệu `double`. Điều này thông báo cho máy tính kích thước bộ nhớ sẽ sử dụng cho mảng động. Nếu chúng ta quên dấu ngoặc vuông và 10 thì máy tính sẽ chỉ cấp phát 1 phần tử kiểu `double` thay vì cấp phát cho cả mảng 10 phần tử kiểu `double`.

Ví dụ chương trình trong hình 7.5, chúng ta có thể sử dụng biến mảng động kiểu `int` và kích thước là 10.

Chương trình trong hình 7.5 sẽ sắp xếp danh sách các số nguyên (`int`). Chương trình thực hiện với danh sách kích thước bất kỳ bởi vì nó sử dụng mảng động để lưu giữ. Kích thước của mảng được xác định khi chương trình chạy. Người sử dụng nhập vào bao nhiêu số cần sắp xếp thì toán tử `new` tạo ra mảng động với kích thước đó.

Chú ý toán tử `delete` sẽ xóa đi biến mảng động `a` trong chương trình hình 7.5. Nếu chương trình kết thúc thì chúng ta không cần toán tử `delete`, tuy nhiên nếu chương trình thực hiện việc khác với biến mảng động này thì chúng ta nên dùng `delete` để trả lại tự do cho phần bộ nhớ đã cấp phát cho `a`. Thao tác `delete` với mảng cấp phát động tương tự như với biến con trỏ chỉ khác là chúng ta phải thêm cặp ngoặc vuông đóng mở như sau:

```
delete [] a;
```

```
1 //Sorts a list of numbers entered at the keyboard.
2 #include <iostream>
3 #include <cstdlib>
4 #include <cstdint>
5
6 using namespace std;
7
8 void InputArray(int a[], int size);
9 //Input values from the keyboard.
10
11 void SortArray(int a[], int size);
12 //Output: The values of a[0] through a[size-1] have been rearranged
13 //so that a[0] <= a[1] <= ... <= a[size-1].
14
15 int main( )
16 {
17
18     cout << "This program sorts numbers from lowest to highest.\n";
19
20     int array_size;
21     cout << "How many numbers will be sorted? ";
22     cin >> array_size;
23
24     int *a;
25     a = new int[array_size];
26
27     InputArray(a, array_size);
28     SortArray(a, array_size);
29
30     cout << "In sorted order the numbers are:\n";
31     for (int index = 0; index < array_size; index++)
32         cout << a[index] << " ";
33     cout << endl;
```

```

34
35     delete [] a;
36
37     return 0;
38 }
39 //Uses the library
40 void InputArray(int a[], int size)
41 {
42
43     cout << "Enter " << size << " integers.\n";
44     for (int i = 0; i < size; i++)
45         cin >> a[i];
46 }
47
48 void SortArray(int a[], int size)
49 {
50     for (int i = 0; i < size-1; i++)
51         for (int j = i +1; j < size; j++)
52             if (a[i] > a[j]){
53                 int temp = a[i];
54                 a[i] = a[j];
55                 a[j] = temp;
56
57             }
58 }

```

Hình 7.6: Mảng cấp phát động.

Output chương trình Hình 7.6:

```

This program sorts numbers from lowest to highest.
How many numbers will be sorted? 5
Enter 5 integers.
6 7 8 3 4
In sorted order the numbers are:
3 4 6 7 8

```

7.5 Truyền tham số của hàm như con trỏ

Như đã đề cập, trong C++, có 3 cách truyền tham số cho hàm. Trong chương 4, chúng ta đã đề cập 2 cách truyền theo tham trị và truyền theo tham chiếu, trong phần này chúng ta sẽ đề cập truyền tham số theo con trỏ (tham biến). Như chúng ta đã đề cập trong chương 4, tham số có thể được truyền tới hàm sử dụng tham chiếu. Các tham số như vậy cũng có thể gọi hàm để thay đổi giá trị ban đầu của tham số được truyền vào trong hàm đó. Tham chiếu cũng có thể giúp chương trình truyền tham số là đối tượng dữ liệu lớn tới hàm và tránh được việc tạo bản sao đối tượng như cách truyền tham số kiểu giá trị. Con trỏ cũng như tham chiếu có thể sử dụng để thay đổi một hoặc nhiều biến được truyền vào hàm hoặc truyền con trỏ tới đối tượng dữ liệu lớn. Trong ngôn ngữ C++, chúng ta có thể sử dụng con trỏ như tham số đầu vào của hàm. Khi hàm được gọi thì địa chỉ của tham số được truyền vào. Chúng ta sẽ sử dụng lại ví dụ việc hoán đổi 2 số cho nhau trong hình 4.15 (truyền theo tham chiếu) của chương 4 để minh họa cho việc truyền tham số bằng con trỏ.


```

1 #include <iostream>
2 using namespace std;
3
4 void swap(int *x, int *y)
5 {
6     int temp ;
7     temp = *x ;
8     *x = *y ;
9     *y = temp ;
10 }
11
12 int main()
13 {
14     int a = 3, b = 5;
15     swap(&a, &b);
16     cout << "a = " << a << ", b = " << b << endl; // 5, 3 (a,b doi gia tri)
17     return 0;
18 }

```

Hình 7.7: Truyền tham số bằng con trỏ.

Output chương trình Hình 7.7:

```
a = 5, b = 3
```

7.6 Con trỏ hàm

Một tính năng đặc biệt khá khó hiểu và mạnh trong C++ là con trỏ hàm. Thậm chí mặc dù hàm không phải là biến nhưng nó vẫn có vị trí vật lý trong bộ nhớ mà có thể được gán cho con trỏ. Địa chỉ này là điểm nhập của hàm và nó là địa chỉ được sử dụng khi hàm được gọi. Khi con trỏ trỏ tới hàm thì hàm đó có thể gọi thông qua con trỏ. Các con trỏ hàm cũng cho phép các hàm được truyền như đối số tới các hàm khác.

Chúng ta có thể lấy địa chỉ của hàm bằng cách sử dụng chỉ tên của hàm không kèm theo bất cứ ngoặc hoặc các tham số của hàm đó (điều này tương tự cách lấy địa chỉ của mảng khi chúng ta chỉ dùng tên của mảng mà không có chỉ số kèm theo). Để xem chúng hoạt động thế nào, ta xem ví dụ trong hình 7.8.

```

1 #include <iostream>
2 #include <cstring>
3 #include <cstdlib>
4
5 using namespace std;
6
7 void Check(char *a, char *b, int (*cmp)(const char *, const char *));
8
9 int main()
10 {
11     char s1[80], s2[80];
12     int (*p)(const char *, const char *);
13     p = strcmp;

```

```

14
15     cout << "Input of s1 string: ";
16     cin.getline(s1,80);
17
18     cout << "Input of s2 string: ";
19     cin.getline(s2,80);
20
21     Check(s1, s2, p);
22
23
24     return 0;
25 }
26 void Check(char *a, char *b, int (*cmp)(const char *, const char *))
27 {
28     cout << "Testing for equality." << endl;
29     if(!(*cmp)(a, b)) cout << "Equal" << endl;
30     else cout << "Not Equal" << endl;
31     return;
32 }

```

Hình 7.8: Ví dụ về con trỏ hàm so sánh 2 xâu ký tự.

Output chương trình Hình 7.8

```

Input of s1 string: Hello
Input of s2 string: Hello John
Testing for equality.
Not Equal

```

Khi hàm `check()` trong hình 7.8 được gọi, hai con trỏ kiểu ký tự và một con trỏ hàm được truyền như tham số. Bên trong hàm `check()`, đối số được mô tả như con trỏ kiểu ký tự và con trỏ hàm. Dấu ngoặc bao quanh `*cmp` là cần thiết cho việc biên dịch nó một cách chính xác. Bên trong hàm `check()`, biểu thức

```
(*cmp)(a,b)
```

sẽ gọi `strcmp()`, hàm mà được trỏ bởi `cmp` với 2 đối số là `a` và `b`. Chú ý rằng, chúng ta có thể gọi `check()` bằng cách sử dụng trực tiếp `strcmp()` như sau:

```
check(s1,s2, strcmp);
```

Chúng ta có thể mở rộng ví dụ trong hình 7.8 thành ví dụ trong hình 7.9. Trong ví dụ này, nếu chúng ta nhập vào một chữ cái thì `strcmp()` được truyền qua hàm `check()`. Trái lại, `numcmp()` được truyền vào hàm `check()`. Bởi vì `check()` gọi hàm mà nó đã được truyền vào, nó có thể sử dụng hàm so sánh khác nhau trong các trường hợp khác nhau.

7.7 Lập trình với danh sách liên kết

Danh sách liên kết là danh sách được xây dựng dựa vào con trỏ. Danh sách liên kết không cố định kích cỡ và nó có thể mở rộng và giảm bớt trong khi chương trình đang chạy. Trong phần này sẽ giới thiệu về khái niệm và thao tác với danh sách liên kết.

```

1 //Chương trình minh họa về con trỏ hàm
2 //Con trỏ hàm có thể trỏ đến 2 hàm khác nhau NumCmp() và strcmp()
3 #include <iostream>
4 #include <cstring>
5 #include <cstdlib>
6
7 using namespace std;
8
9 void Check(char *a, char *b, int (*cmp)(const char *, const char *));
10
11 int NumCmp(const char *a, const char *b);
12
13 int main()
14 {
15     char s1[80], s2[80];
16     //int (*p)(const char *, const char *);
17     //p = strcmp;
18
19     cout << "Input of s1 string: ";
20     cin.getline(s1,80);
21
22     cout << "Input of s2 string: ";
23     cin.getline(s2,80);
24
25     if (isalpha(*s1))
26         Check(s1, s2, strcmp);
27     else
28         Check(s1,s2, NumCmp);
29
30
31     return 0;
32 }
33 //-----
34 void Check(char *a, char *b, int (*cmp)(const char *, const char *))
35 {
36     cout << "Testing for equality." << endl;
37     if(!(*cmp)(a, b)) cout << "Equal" << endl;
38     else cout << "Not Equal" << endl;
39     return;
40 }
41
42 //-----
43 int NumCmp(const char *a, const char *b)
44 {
45     if (atoi(a) == atoi(b)) return 0;
46     else return 1;
47 }

```

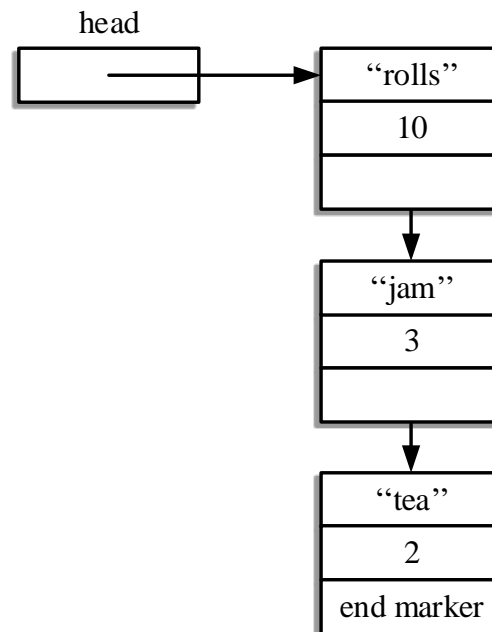
Hình 7.9: Ví dụ về con trỏ hàm mở rộng từ hình 7.8.

7.7.1 Nút và danh sách liên kết

Biến động khá phổ biến với một số kiểu dữ liệu phức tạp như dữ liệu kiểu mảng, kiểu cấu trúc hay kiểu lớp. Như chúng ta đã biết thì biến động của mảng rất hữu ích và đối với dữ liệu kiểu cấu trúc hay kiểu lớp nó cũng hữu ích như vậy nhưng theo cách khác. Biến động hoặc là kiểu cấu trúc hoặc là kiểu lớp thông thường có một hoặc nhiều biến thành viên là biến con trỏ kết nối chúng tới các biến động khác. Ví dụ, một trong những cấu trúc như vậy chứa danh sách mua sắm được biểu diễn như trong hình 7.10.

Nút

Một cấu trúc như hình 7.10 bao gồm các mục được vẽ như các hộp được nối với nhau bằng các mũi tên. Các hộp được gọi là các nút và các mũi tên thể hiện cho con trỏ. Mỗi nút trong hình 7.10 có 3 biến thành viên, một biến chứa dữ liệu kiểu chuỗi, một biến chứa dữ liệu kiểu số nguyên và biến còn lại chứa dữ liệu kiểu con trỏ trỏ đến một nút khác cùng kiểu trong danh sách. Các nút



Hình 7.10: Nút và con trỏ.

được thực hiện trong C++ có kiểu cấu trúc hoặc kiểu lớp. Ví dụ, định nghĩa kiểu cấu trúc cho một nút thể hiện như trong hình 7.10, cùng với đó là định nghĩa cho con trỏ trỏ tới một nút như sau:

```

struct    ListNode
{
    string item;
    int    count;
    ListNode *link;
};
typedef ListNode* ListNodePtr;
  
```

Thứ tự định nghĩa rất quan trọng. Định nghĩa **ListNode** phải thực hiện trước, sau đó nó được sử dụng để định nghĩa **ListNodePtr**.

Hộp có tên **head** trong hình 7.10 không phải là một nút, mà là một biến con trỏ có thể trỏ đến một nút. Biến con trỏ **head** được khai báo như sau:

```
ListNodePtr head;
```

Như minh họa, giả sử các khai báo như trên, trong tình huống được mô tả trong hình 7.10, nếu muốn thay đổi số nguyên trong nút đầu tiên từ 10 thành 12, ta có thể thực hiện như sau:

```
(*head).count = 12;
```

Trong biểu thức ở bên trái của toán tử gán, biến **head** là biến con trỏ, nên ***head** sẽ trỏ tới địa chỉ của một nút, cụ thể trong trường hợp này là nút chứa chuỗi **"rolls"** và số nguyên **10**. Nút này được tham chiếu bởi ***head** có kiểu cấu trúc; và biến thành viên của kiểu cấu trúc này chứa giá trị kiểu nguyên được gọi là **count**, nên **(*head).count** là tên của biến nguyên trong nút đầu tiên. Tuy nhiên, toán tử **"."** có độ ưu tiên cao hơn toán tử **"*"**, vì thế nếu không có cặp đóng mở ngoặc **"()"** thì toán tử **"."** sẽ được thực hiện đầu tiên (và có thể phát sinh lỗi). Phần tiếp theo mô tả một ký hiệu ngắn có thể giải quyết vấn đề trên.

Trong C++ có một toán tử có thể được sử dụng với con trỏ để đơn giản kí hiệu cho các biến thành viên của một cấu trúc hoặc một lớp. Toán tử **"->"** kết hợp hành động của toán tử tham chiếu **"*"** và toán tử **"."** tới thành viên của một cấu trúc hoặc một đối tượng cụ thể được trỏ bởi một con trỏ. Ví dụ, câu lệnh gán ở trên thay đổi số nguyên trong nút đầu tiên có thể được viết đơn giản hơn như sau:

```
head->count = 12;
```

Câu lệnh gán này và câu lệnh gán trước đó có ý nghĩa giống nhau, nhưng câu lệnh này thường được sử dụng hơn.

Chuỗi trong nút đầu tiên có thể được thay đổi từ **"rolls"** thành **"bagels"** với câu lệnh sau:

```
head->item = "bagels";
```

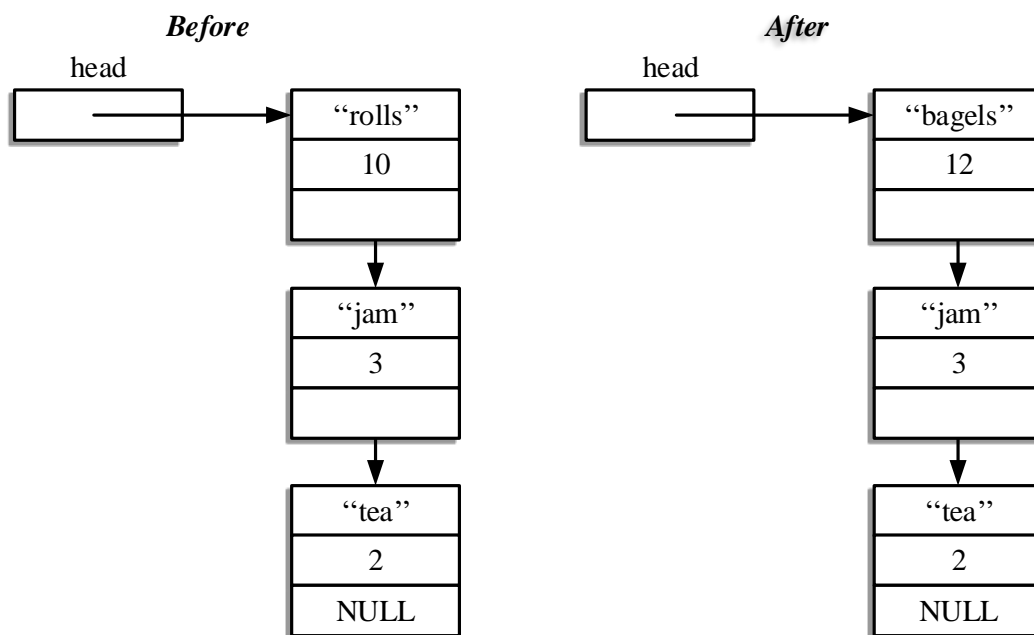
Kết quả của sự thay đổi này tới nút đầu tiên trong danh sách được mô tả như trong hình 7.11. Nhìn vào nút cuối cùng trong danh sách, ta thấy nút này có từ **NULL** ở vị trí lẽ ra là một con trỏ. Trong hình 7.10, vị trí này được điền với cụm từ "end marker", nhưng "end marker" không phải là một biểu thức trong C++. Trong các chương trình C++, chúng ta sử dụng hằng số **NULL** là một hằng số đặc biệt như dấu chấm hết báo hiệu kết thúc một danh sách liên kết.

Hằng số **NULL** được sử dụng cho hai mục đích khác nhau (nhưng thường trùng nhau). Thứ nhất, nó được sử dụng để cung cấp giá trị cho một biến con trỏ, nếu không thì biến đó sẽ không có bất cứ giá trị gì. Điều này sẽ ngăn cản một tham chiếu tới vùng nhớ, vì **NULL** không phải là địa chỉ của bất cứ một vùng nhớ nào. Thứ hai, nó được sử dụng như một điểm đánh dấu kết thúc. Một chương trình có thể duyệt qua danh sách các nút như trong hình 7.11 và khi chương trình đi tới nút có chứa **NULL** thì cho biết nó đã đi tới nút cuối cùng của danh sách.

Một con trỏ có thể được thiết lập giá trị **NULL** sử dụng toán tử gán như sau, trong đó khai báo một biến con trỏ **there** và khởi tạo nó với giá trị **NULL**:

```
double *there = NULL;
```

Hằng số **NULL** có thể được gán cho biến con trỏ của bất kì kiểu con trỏ nào. Thực tế thì hằng số **NULL** là số **0** dẫn đến một vấn đề không rõ ràng. Xem xét hàm nạp chồng sau:



Hình 7.11: Truy cập dữ liệu của một nút.

```
void func(int *p);
void func(int i);
```

Hàm nào sẽ được gọi nếu chúng ta gọi `func(NULL)` ? Vì `NULL` là số `0` nên cả hai hàm trên có giá trị ngang nhau. C++11 giải quyết vấn đề này bằng cách đưa thêm hằng số `nullptr`, `nullptr` không phải là số nguyên `0` nhưng là hằng số chữ được sử dụng để đại diện cho con trỏ `null`. Sử dụng `nullptr` ở bất cứ nơi nào chúng ta đã sử dụng `NULL` cho một con trỏ. Ví dụ, chúng ta có thể viết:

```
double *there = nullptr;
```

Danh sách liên kết

Trong phần trước, chúng ta đã sử dụng kiểu mảng để lưu trữ một danh sách các phần tử. Việc sử dụng mảng có điểm thuận lợi là các phần tử của nó được truy nhập trực tiếp, rất nhanh thông qua chỉ số (cách truy nhập này được gọi là truy nhập ngẫu nhiên (random access) ngược với cách truy nhập tuần tự (sequential access) mất nhiều thời gian hơn vì phải duyệt tuần tự qua hết các phần tử đứng trước mới định vị được phần tử cần truy nhập). Ưu điểm trên của kiểu mảng có được là do các phần tử của nó có kích thước bằng nhau (vì cùng kiểu) và được sắp xếp kề nhau liên tục, “thẳng hàng” trong bộ nhớ. Một số nhược điểm của mảng (tĩnh) như đã nêu cũng đã được khắc phục bằng mảng động. Tuy nhiên, việc sử dụng mảng động cũng còn có nhược điểm: ví dụ, cần tạo mảng động với 1000 phần tử số dạng `int` (4 bytes), khi đó chương trình sẽ tìm trong heap phần bộ nhớ đang rỗi một số lượng 4000 bytes “kề nhau liên tục” để cấp phát cho mảng. Đôi khi tổng lượng bytes rỗi của heap có thể lớn hơn 4000 bytes, nhưng chúng nằm “rải rác” chứ không kề nhau, vì vậy việc cấp phát thất bại.

Cần có một cấu trúc dữ liệu khác để giải quyết hạn chế trên. Cấu trúc (kiểu) dữ liệu mới này được gọi là danh sách liên kết.

Khái niệm danh sách liên kết

Cũng tương tự kiểu mảng, danh sách liên kết dùng để lưu trữ một danh sách, là một tập hợp các phần tử cùng kiểu được sắp xếp theo một thứ tự xác định. Tuy nhiên, danh sách liên kết khác với kiểu mảng ở các đặc điểm quan trọng sau:

- Không cần khai báo trước kích thước (tối đa) của danh sách. Trong quá trình hoạt động, nếu cần thêm phần tử vào danh sách, chương trình sẽ xin cấp phát thêm bộ nhớ vừa đủ để lưu các phần tử này. Ở chiều ngược lại, nếu một phần tử bị loại ra khỏi danh sách, chương trình sẽ đề nghị hệ điều hành thu hồi lại phần bộ nhớ này để sử dụng vào việc khác. Như vậy, tại thời điểm bất kỳ, số bộ nhớ bị chiếm dụng luôn luôn bằng đúng với số phần tử của danh sách, khác với kiểu mảng số bộ nhớ bị chiếm dụng luôn luôn là hằng số.

- Khi cần thêm phần tử vào danh sách chương trình sẽ xin cấp phát bộ nhớ, do vậy các phần tử của danh sách liên kết sẽ nằm ở những vị trí bất kỳ nào đó trong bộ nhớ chứ không nằm theo một trật tự kề nhau liên tiếp như trong kiểu mảng.

- Danh sách liên kết hầu như không bị xảy ra tình trạng đầy như mảng, nó chỉ lệ thuộc vào dung lượng bộ nhớ thực sự của máy tính.

Các phần tử của danh sách liên kết không nằm kề nhau liên tiếp trong bộ nhớ, vậy làm thế nào để chúng ta xác định được vị trí của các phần tử này và thứ tự giữa chúng. Để làm được điều này, chúng ta sẽ trang bị thêm cho mỗi phần tử của danh sách một con trỏ. Nhiệm vụ của các con trỏ này là tạo mối liên kết có thứ tự giữa các phần tử của danh sách bằng cách cho con trỏ của một phần tử bất kỳ trỏ đến (chứa địa chỉ của) phần tử đứng kế sau nó. Như vậy, xuất phát từ phần tử đầu tiên của danh sách (ví dụ là A), nhìn vào con trỏ của A ta sẽ tìm được phần tử đứng kế sau A (ví dụ là B), và thông qua con trỏ của B ta sẽ tìm được phần tử kế tiếp C ... cuối cùng thông qua sự dẫn đường của các con trỏ ta dễ dàng tìm thấy được toàn bộ các phần tử còn lại của danh sách theo đúng thứ tự như trên, đến khi gặp phần tử chứa con trỏ với giá trị **NULL** thì đó là phần tử cuối cùng.

Một danh sách với mỗi phần tử có một con trỏ trỏ đến phần tử đứng sau như vậy được gọi là danh sách liên kết (đơn) và được minh họa như hình 7.11. Một danh sách liên kết là một danh sách các nút, trong đó mỗi nút có một biến thành viên là một con trỏ trỏ tới nút kế tiếp trong danh sách. Nút đầu tiên trong danh sách liên kết được gọi là **head**, đó cũng là lý do vì sao biến con trỏ trỏ tới nút đầu tiên được đặt tên là **head**. Lưu ý rằng con trỏ **head** bản thân nó không đứng đầu danh sách mà nó chỉ trỏ tới nút đầu tiên của danh sách. Nút cuối cùng không có tên đặc biệt nào nhưng có tính chất đặc biệt, **NULL** như là giá trị của biến con trỏ thành viên của nó. Để kiểm tra xem một nút có phải là nút cuối cùng hay không, chúng ta chỉ cần kiểm tra xem biến con trỏ trong nút đó có bằng **NULL** hay không.

Mục tiêu của chúng ta trong phần này là xây dựng một số hàm cơ bản khi thao tác với danh sách liên kết. Để đơn giản các kí hiệu, ta sẽ sử dụng kiểu dữ liệu của một nút đơn giản hơn so với nút trong hình 7.11. Những nút này sẽ chỉ chứa một số nguyên và một con trỏ. Định nghĩa nút và con trỏ như sau:

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
```

Bây giờ, chúng ta sẽ xem xét việc bắt đầu xây dựng một danh sách liên kết với các nút có kiểu dữ liệu trên như thế nào. Đầu tiên, chúng ta khai báo một biến con trỏ có tên là `head`, biến này sẽ trỏ tới nút đầu tiên trong danh sách liên kết của chúng ta, và được khai báo như sau:

```
NodePtr head;
```

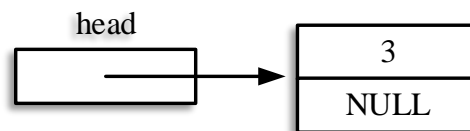
Để tạo nút đầu tiên, chúng ta sử dụng toán tử `new` để tạo một biến động mới, biến động này sẽ trở thành nút đầu tiên trong danh sách liên kết:

```
head = new Node;
```

Sau đó, ta gán giá trị tới các biến thành viên của nút mới này:

```
head->data = 3;
head->link = NULL;
```

Chú ý rằng, biến con trỏ của nút này phải được thiết lập bằng `NULL`, bởi vì hiện tại nút này cũng đang là nút cuối cùng trong danh sách (đồng thời cũng là nút đầu tiên trong danh sách). Ở giai đoạn này, danh sách liên kết của chúng ta như sau:



Chèn một nút vào vị trí đầu danh sách

Trong phần này, chúng ta giả sử danh sách liên kết đã chứa sẵn một hoặc nhiều nút, bây giờ ta sẽ xây dựng một hàm có chức năng chèn thêm một nút khác vào danh sách. Tham số đầu tiên cho hàm chèn là tham số gọi bởi tham chiếu cho biến con trỏ trỏ tới nút đầu của danh sách liên kết. Tham số còn lại sẽ đưa vào một số nguyên và lưu trữ trong nút mới. Khai báo hàm cho hàm chèn như sau:

```
void head_insert(NodePtr& head, int the_number);
```

Để chèn một nút mới vào danh sách liên kết, ta sử dụng toán tử `new` để tạo một nút mới. Dữ liệu sau đó được sao chép vào nút mới này và được chèn vào đầu danh sách. Khi chèn các nút theo cách này, nút mới sẽ trở thành nút đầu tiên của danh sách chứ không phải là nút cuối. Sau đó các biến động không có tên, chúng ta phải sử dụng biến con trỏ cục bộ để trỏ tới nút này. Nếu chúng ta gọi biến con trỏ cục bộ là `temp_ptr`, nút mới có thể được gọi là `*temp_ptr`. Các bước có thể được tổng hợp lại như sau:

Giải mã cho hàm `head_insert` :

1. Tạo một biến động được trỏ tới bởi `temp_ptr`. (Biến động này là một nút mới. Nút mới này có thể được gọi là `*temp_ptr`).
2. Gán dữ liệu cho nút mới này.
3. Thành viên liên kết của nút mới trỏ tới nút đầu của danh sách liên kết ban đầu.

4. Biến con trỏ tên head trỏ tới nút mới.

Hình 7.13 chứa lưu đồ của thuật toán. Bước 2 và 3 trong lưu đồ có thể được thể hiện bằng câu lệnh trong C++ như sau:

```
temp_ptr->link = head;
head = temp_ptr;
```

Hàm đầy đủ được định nghĩa trong hình 7.12.

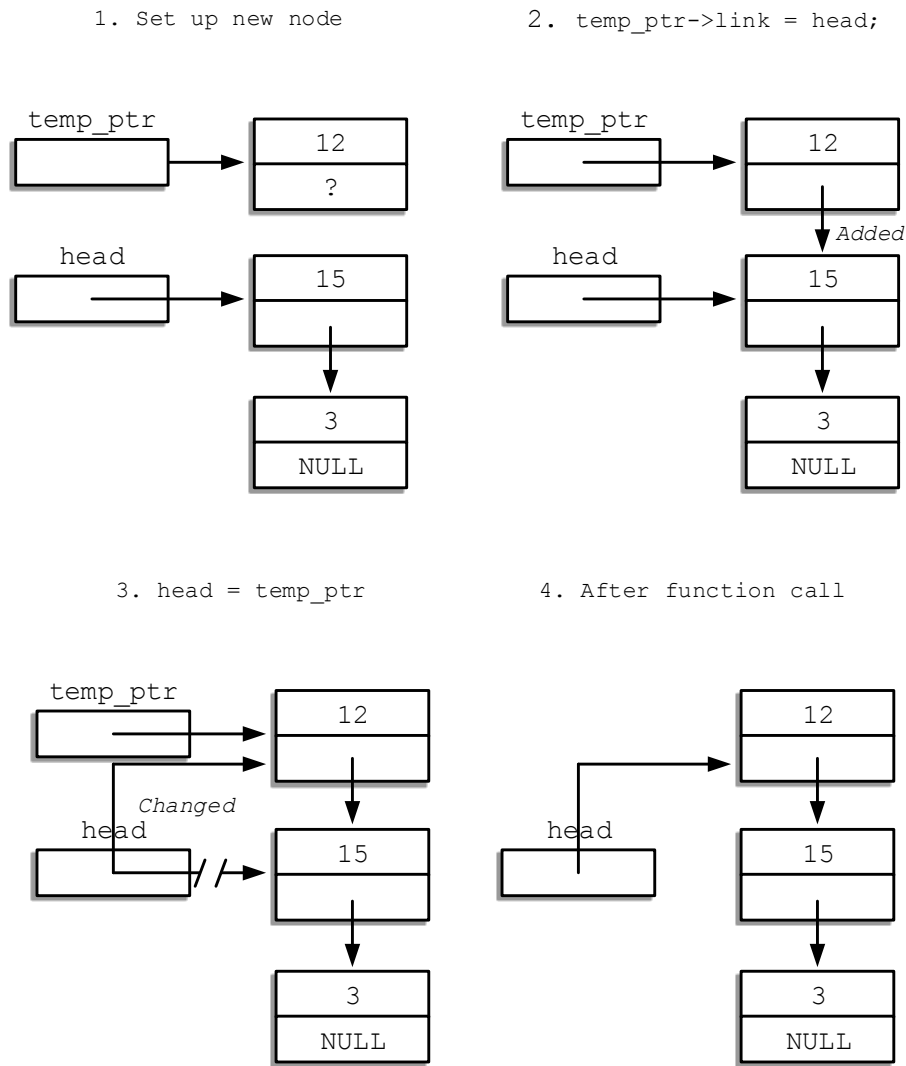
```
1 struct Node
2 {
3     int data;
4     Node *link;
5 };
6
7 typedef Node* NodePtr;
8
9 //Dau vao: Bien cho tro head tro toi nut dau tien cua danh sach lien ket.
10 //Dau ra: Mot nut moi chua the_number duoc them vao tai vi tri dau tien
11 //cua danh sach lien ket
12 void Head_Insert(NodePtr& head, int the_number);
13
14 void Head_Insert(NodePtr& head, int the_number)
15 {
16     NodePtr temp_ptr;
17     temp_ptr = new Node;
18
19     temp_ptr->data = the_number;
20
21     temp_ptr->link = head;
22     head = temp_ptr;
23 }
```

Hình 7.12: Hàm chèn thêm một nút vào đầu danh sách liên kết

Danh sách liên kết được đặt tên bởi tên của con trỏ trỏ tới nút đầu của danh sách, nhưng danh sách rỗng không có nút đầu. Để cụ thể một danh sách rỗng, ta sử dụng con trỏ **NULL**. Nếu biến con trỏ **head** được coi là trỏ tới nút đầu của danh sách liên kết và giờ muốn chỉ ra danh sách là rỗng, thì ta chỉ cần thiết lập giá trị của con trỏ head như sau:

```
head = NULL;
```

Khi xây dựng hàm thao tác với một danh sách liên kết, ta luôn phải kiểm tra để biết hàm có làm việc trên danh sách rỗng không, nếu không, ta có thể phải thêm trường hợp đặc biệt cho danh sách rỗng. Nếu ta xây dựng hàm không áp dụng cho danh sách rỗng, sau đó chương trình của chúng ta được xây dựng để xử lý các danh sách rỗng theo một số cách hoặc tránh trường hợp này. May mắn thay, danh sách rỗng thông thường có thể được xử lý như bất cứ một danh sách nào khác. Ví dụ, hàm **head_insert** trong hình 7.12 được thiết kế với các danh sách không rỗng như một mô hình, nhưng việc kiểm tra sẽ chỉ ra rằng nó cũng làm việc tốt đối với danh sách rỗng.



Hình 7.13: Thêm một nút vào danh sách liên kết.

Hiện tượng: Mất nút

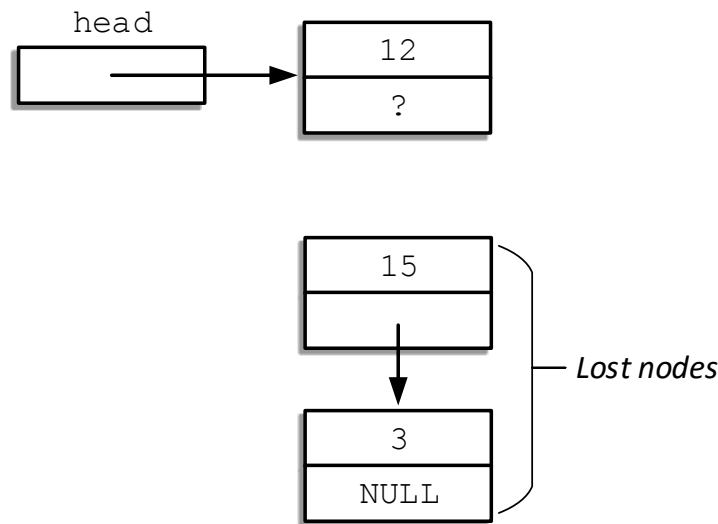
Ta có thể viết định nghĩa hàm `head_insert` (hình 7.12) sử dụng biến con trỏ `head` để xây dựng nút mới, thay vì sử dụng biến con trỏ cục bộ `temp_ptr`. Nếu đã thử việc đó, ta có thể bắt đầu hàm như sau:

```
head = new Node;
head->data = the_number;
```

Tại thời điểm nút mới được xây dựng, chứa dữ liệu đúng và nó được trỏ tới bởi con trỏ `head`. Tất cả những việc còn lại là gắn phần còn lại của danh sách tới nút này bằng việc thiết lập con trỏ thành viên được đưa ra bên dưới để nó trỏ sau trỏ tới nút trước đó là nút đầu tiên của danh sách như sau:

```
head->link
```

Hình 7.14 cho thấy trường hợp ta muốn chèn thêm giá trị dữ liệu mới là **12**, vào đầu danh sách. Nếu ta xử lý theo cách này, sẽ không có gì trỏ tới nút chứa giá trị **15**, vì không có con trỏ trỏ tới nó nên không có cách nào để chương trình (`head->link`) có thể tham chiếu tới đó, các nút bên dưới nút này cũng bị mất. Vì vậy, chương trình sẽ không có cách nào để truy xuất dữ liệu và làm bất cứ việc gì đối với những nút này.



Hình 7.14: Nút bị mất trong danh sách liên kết.

Một chương trình bị mất nút thì đôi khi được cho là "rò rỉ bộ nhớ", rò rỉ bộ nhớ có thể dẫn tới kết quả chương trình chạy ra khỏi bộ nhớ, là nguyên nhân khiến chương trình bị chấm dứt bất thường. Để tránh bị mất nút, chương trình phải luôn giữ một vài con trỏ trỏ tới đầu danh sách, thường là con trỏ trong biến con trỏ như **head**.

Tìm kiếm trên danh sách liên kết

Phần này, chúng ta sẽ xây dựng một hàm tìm kiếm trong một danh sách liên kết để xác định vị trí của một nút cụ thể. Chúng ta sử dụng các nút cùng kiểu như đã sử dụng trong mục trước, gọi là **Node** (định nghĩa về nút và kiểu con trỏ được đưa ra như hình 7.12). Hàm chúng ta xây dựng có hai đối số: một cho danh sách liên kết, và một số nguyên mà chúng ta muốn xác định vị trí của nút. Hàm sẽ trả về một con trỏ trỏ tới nút đầu tiên chứa số nguyên đó. Nếu không có nút nào thỏa mãn thì hàm trả về con trỏ **NULL**. Bằng cách này, chương trình của chúng ta có thể kiểm tra xem liệu một số nguyên có trong danh sách bằng kiểm tra để xem nếu hàm trả về một giá trị con trỏ khác **NULL** hay không?. Khai báo hàm và các chú thích cho hàm như sau:

```

NodePtr search(NodePtr head, int target);
//Điều kiện trước: Con trỏ head trỏ tới đầu của
//danh sách liên kết. Biến con trỏ ở nút cuối bằng NULL.
//Nếu danh sách rỗng, thì head bằng NULL. Trả về con
//trỏ trỏ tới nút đầu chứa target. Nếu không có nút nào
//thỏa mãn, hàm trả về NULL.

```

Chúng ta sẽ sử dụng một con trỏ cục bộ được gọi là **here** di chuyển qua danh sách để tìm kiếm **target**. Cách duy nhất để di chuyển quanh một danh sách liên kết, hay bất cứ cấu trúc dữ liệu khác gồm có các nút và con trỏ là dựa vào con trỏ. Vậy nên chúng ta sẽ bắt đầu với con trỏ **here** trỏ tới nút đầu tiên của danh sách và di chuyển con trỏ từ nút tới nút theo sau con trỏ ra ngoài của mỗi nút, kỹ thuật này được biểu diễn như trong hình 7.15 và thuật toán được mô tả như sau:

Giải mã cho hàm tìm kiếm

Cho biến con trỏ **here** trỏ tới nút **head** (là nút đầu tiên) của danh sách liên kết.

```
while (here không trở tới nút chứa target và here không trở tới nút cuối của danh sách)
{
    Thực hiện cho con trỏ here trở tới nút kế tiếp trong danh sách.
}
if (nút được trở tới bởi con trỏ here chứa target)
    return here;
else
    return NULL;
```

Để di chuyển con trỏ here tới nút kế tiếp, nút kế tiếp được trở bởi thành viên con trỏ của nút hiện thời được trở bởi **here**. Thành viên con trỏ của nút hiện thời được trở bởi here được cho bởi biểu thức

```
here->link
```

Di chuyển con trỏ here tới nút kế tiếp trong danh sách:

```
here = here->link;
```

Tổng hợp các phần trên lại, chúng ta có thuật toán theo giả mã sau:

Sơ bộ của mã nguồn cho hàm tìm kiếm:

```
here = head;
while (here->data != target && here->link != NULL)
    here = here->link;
if (here->data == target)
    return here;
else
    return NULL;
```

Chú ý biểu thức logic trong câu lệnh **while**. Chúng ta kiểm tra để xem nếu **here** không trở tới nút cuối cùng bằng việc kiểm tra xem biến thành viên **here->link** khác giá trị **NULL** hay không?

Nếu kiểm tra mã nguồn của hàm trên cho các trường hợp đặc biệt (ở đây là trường hợp danh sách rỗng), chúng ta sẽ thấy có vấn đề. Nếu danh sách rỗng, thì here bằng NULL và do đó các biểu thức sau là không xác định được:

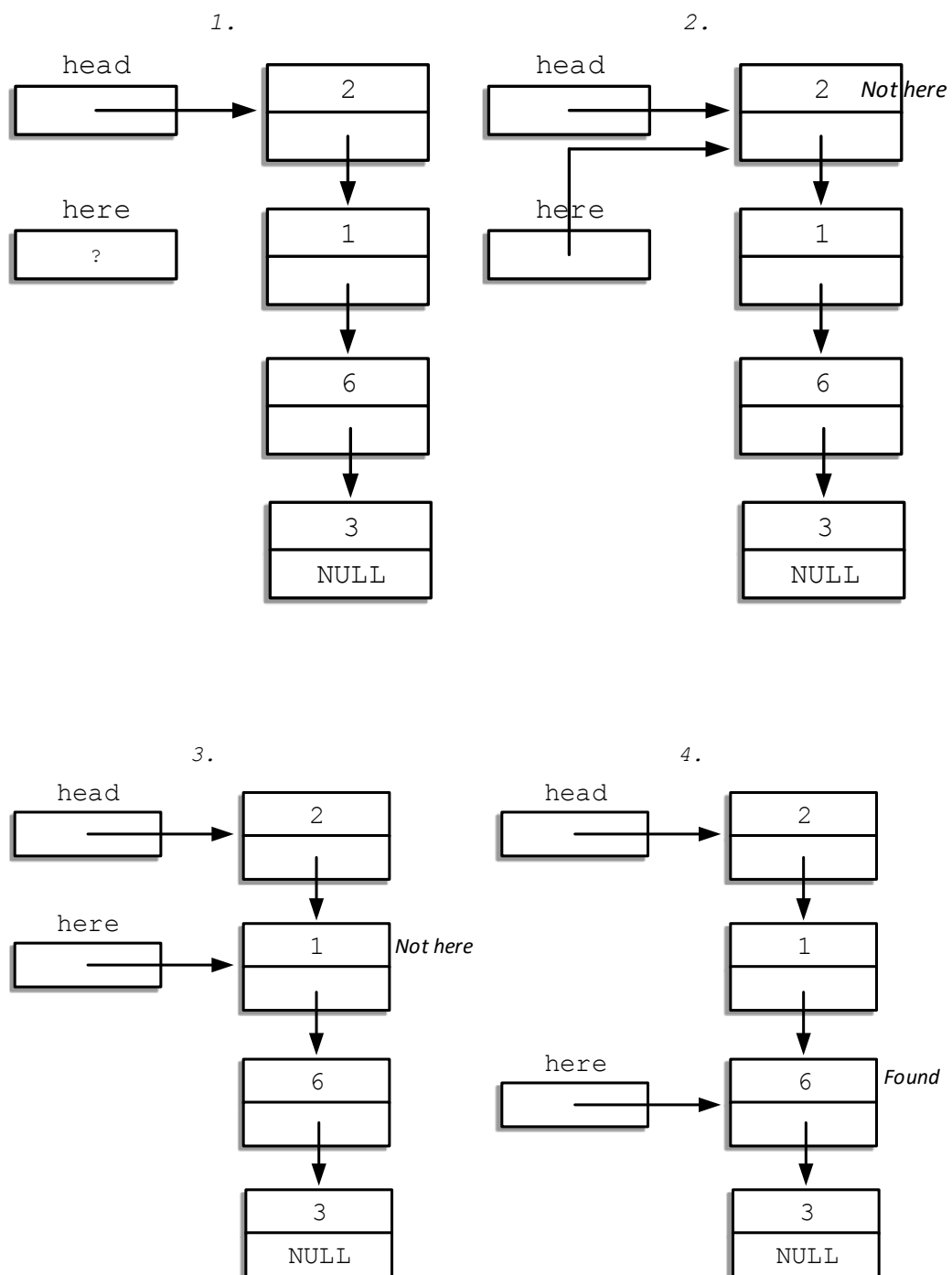
```
here->data
here->link
```

Khi con trỏ **here** là **NULL**, nó không trở tới bất kì nút nào, nên không có thành viên tên **data** nào cũng như không có thành viên tên **link** nào. Vì vậy, chúng tôi thực hiện một trường hợp đặc biệt của danh sách rỗng. Hàm được định nghĩa hoàn thiện như trong hình 7.16.

Khai báo hàm:

```
1 struct Node
2 {
3     int data;
4     Node *link;
5 };
6
7 typedef Node* NodePtr;
8
9 //Dau vao: Con tro head tro toi dau danh sach lien ket.
10 //Dau ra: Tra ve con tro tro toi nut dau tien ma chua gia tri target. Neu khong co
    nut
```

Target is 6



Hình 7.15: Tìm kiếm trong danh sách liên kết.

```

11 //chua gia tri target ham tra ve NULL.
12 NodePtr search(NodePtr head, int target);

```

Định nghĩa hàm:

```

1 NodePtr Search(NodePtr head, int target)
2 {
3     NodePtr here = head;

```

```

4
5  if (here == NULL)
6  {
7      return NULL;  Empty list case
8  }
9  else{
10     while (here->data != target &&
11            here->link != NULL)
12        here = here->link;
13
14     if (here->data == target)
15         return here;
16     else
17         return NULL;
18 }
19 }

```

Hình 7.16: Xác định vị trí nút trong danh sách liên kết

Con trỏ là biến iterator

Biến *iterator* là cấu trúc cho phép bạn duyệt toàn bộ các hạng mục dữ liệu được lưu trong cấu trúc dữ liệu để bạn có thể thực hiện bất cứ việc gì bạn muốn đối với mỗi hạng mục đó. Một iterator có thể là đối tượng của một lớp *integrator* hoặc đơn giản là chỉ số mảng hoặc là một con trỏ. Nó được khai báo như sau:

```

Node_Type *iter;
for (iter = head; iter != NULL; iter = iter->link)

```

Điều kiện *head* là một con trỏ tới nút đầu của danh sách liên kết và *link* là tên của biến thành viên của nút mà nút đó trỏ tới nút kế tiếp trong danh sách.

Ví dụ, để xuất dữ liệu trong tất cả các nút của danh sách liên kết đã thảo luận ở trên, có thể sử dụng:

```

NodePtr iter; //ươTng đương ởvi: Node *iter;
for (iter = head; iter != NULL; iter = iter->link)
    cout << (iter->data);

```

Định nghĩa *Node* và *NodePtr* như trong 7.16.

Chèn và gỡ bỏ một nút bên trong danh sách

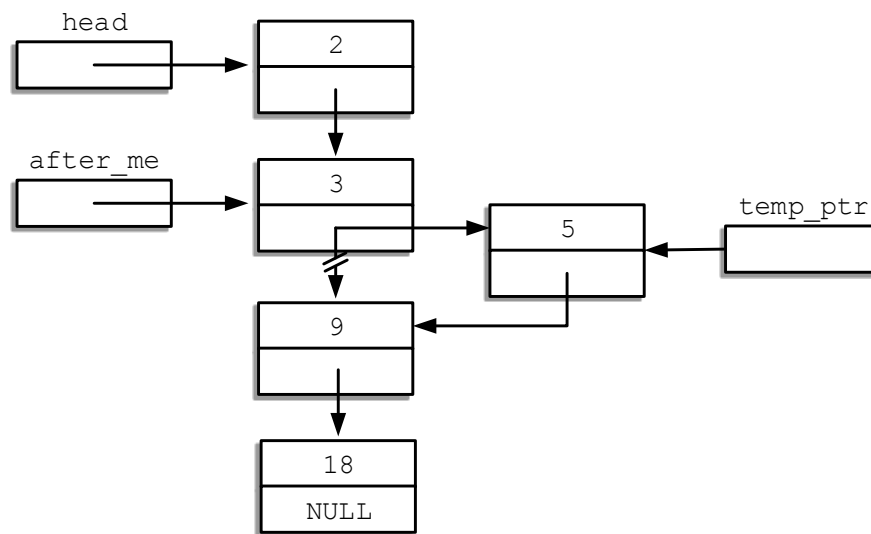
Tiếp theo, ta sẽ thiết kế hàm để chèn một nút tại vị trí cụ thể trong danh sách liên kết, nếu ta muốn các nút có thứ tự nhất định, như thứ tự số hoặc thứ tự chữ cái, ta sẽ không thể đơn giản là chèn một nút vào vị trí bắt đầu hay kết thúc của danh sách. Do đó ta sẽ xây dựng hàm để chèn một nút vào sau một nút được chỉ định của danh sách. Chúng ta giả sử rằng một số hàm khác hoặc một phần của chương trình đã đặt chính xác một con trỏ có tên **after_me** trỏ tới nút nào đó trong danh sách liên kết. Giờ ta muốn nút mới được đặt sau nút được trỏ tới bởi **after_me** như minh họa trong hình 7.17. Cách làm tương tự cho các nút với bất kì kiểu dữ liệu nào, nhưng để cụ thể, ta sẽ sử dụng các nút có cùng kiểu **Node** như trong các phần trước. Khai báo hàm như sau:

```

//Đầu vào: after_me trỏ tới một nút trong danh
//sách liên kết.

```

```
//Đầu ra: Nút mới chứa the_number được thêm vào
//sau khi nó được trả bởi after_me.
void insert(NodePtr after_me, int the_number);
```



Hình 7.17: Chèn phần tử vào giữa danh sách liên kết.

Một nút mới thiết lập theo cách như trong hàm **head_insert** trong hình 7.12. Sự khác nhau giữa các hàm này chính là việc giờ ta muốn chèn một nút mới không vào đầu danh sách, mà sau nút được trả bởi con trỏ **after_me**. Cách thức chèn được mô tả như hình 7.17 và được thể hiện trong C++ như sau:

```
//Thêm liên kết từ nút mới tới danh sách:
temp_ptr->link = after_me->link;
//Thêm liên kết từ danh sách tới nút mới:
after_me->link = temp_ptr;
```

Thứ tự của hai phép gán trên là rất quan trọng, nếu câu lệnh thứ hai được thực hiện trước thì **after_me->link** bị thay đổi trước khi nó được gán cho **temp_ptr->link**, hiện tượng mất nút sẽ xảy ra. Người đọc có thể kiểm tra lại điều này thông qua hình 7.17.

Bản hoàn thiện của hàm **insert** được đưa ra trong hình 7.18.

```
1 void Insert(NodePtr after_me, int the_number)
2 {
3     NodePtr temp_ptr;
4     temp_ptr = new Node;
5
6     temp_ptr->data = the_number;
7
8     temp_ptr->link = after_me->link;
9     after_me->link = temp_ptr;
10 }
```

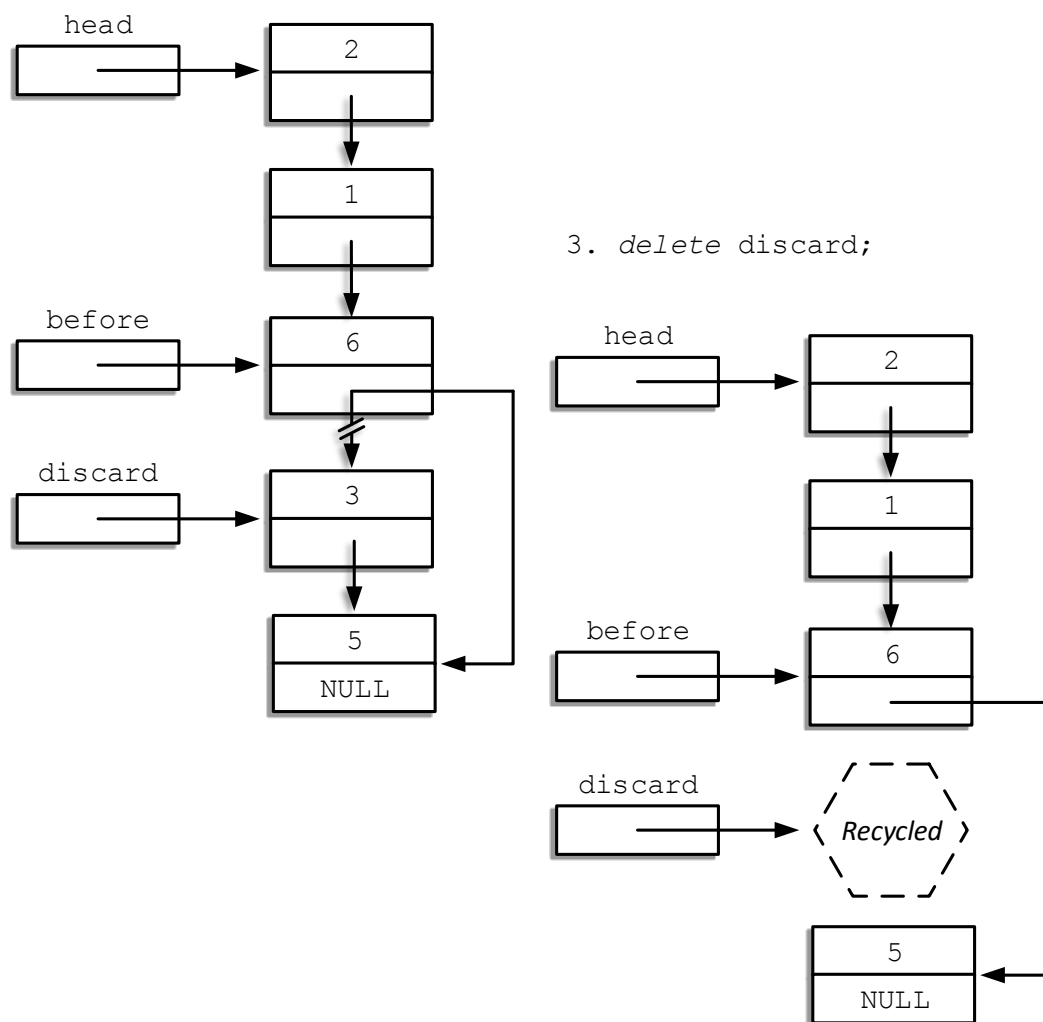
Hình 7.18: Hàm thêm một nút vào giữa danh sách liên kết

Để loại bỏ một nút từ danh sách liên kết cũng khá đơn giản. Hình 7.19 minh họa phương pháp này, tất cả những gì cần thiết để loại bỏ các nút theo lệnh sau:

```
before->link = discard->link;
```

Lệnh trên là đủ để loại bỏ một nút từ danh sách liên kết, nếu ta không sử dụng nút này nữa thì có thể hủy nó và trả lại vùng nhớ mà nó đã sử dụng, có thể thực hiện việc này với lời gọi **delete** như sau:

```
delete discard;
```



Hình 7.19: Xóa một nút trong danh sách liên kết.

7.7.2 Danh sách liên kết của lớp

Trong các ví dụ trước, ta đã tạo danh sách liên kết bằng việc sử dụng kiểu cấu trúc để tổ chức các nội dung của nút trong danh sách, nó cũng có thể tạo cấu trúc dữ liệu tương tự sử dụng kiểu lớp thay vì cấu trúc. Về mặt logic thì định nghĩa giống hệt nhau ngoại trừ cú pháp sử dụng và định nghĩa một lớp

Hình 7.20 và 7.21 minh họa làm thế nào để định nghĩa một lớp **Node** dưới dạng lớp. Các biến dữ liệu được khai báo **private** sử dụng nguyên tắc ẩn thông tin, và phương thức **public** được tạo ra để truy nhập giá trị dữ liệu và nút kế tiếp trong liên kết. Hình 7.22 tạo một danh sách ngắn gồm 5 nút bằng cách chèn các nút mới.


```

1 //Day la file header cho Node.h
2
3 namespace linkedlistofclasses
4 {
5     class Node
6     {
7     public:
8         Node( );
9         //Khoi tao mot nut
10        Node(int value, Node *next);
11        //Lay gia tri cua nut nay
12        int getData( ) const;
13        //Ham lay gia tri cua nut ke tiep
14        Node *getLink( ) const;
15        //Ham thay doi gia tri luu trong danh sach
16        void setData(int value);
17        //Ham thay doi tham chieu toi nut ke tiep
18        void setLink(Node *next);
19    private:
20        int data;
21        Node *link;
22    };
23    typedef Node* NodePtr;
24 }
25 //Danh sach lien ket cua lop
26 //Ket thuc Node.h

```

Hình 7.20: File giao diện của lớp danh sách liên kết.

```

1 //Day la file thuc hien Node_chuong_7.cpp
2 #include <iostream>
3 #include "Node_chuong_7.h"
4
5 namespace linkedlistofclasses
6 {
7     Node::Node( ) : data(0), link(NULL)
8     {
9         //Rong
10    }
11    Node::Node(int value, Node *next) : data(value), link(next)
12    {
13        //Rong
14    }
15    int Node::getData( ) const
16    {
17        return data;
18    }
19    Node* Node::getLink( ) const
20    {
21        return link;
22    }
23    void Node::setData(int value)
24    {
25        data = value;

```

```

26     }
27     void Node::setLink(Node *next)
28     {
29         link = next;
30     }
31 }
32 //linkedlistofclasses
33 //Ket thuc file Node_chuong_7.cpp

```

Hình 7.21: File thực hiện của lớp danh sách liên kết.

```

1 //Chương trình này trình bày cách tạo danh sách liên kết sử dụng Node class
2 #include <iostream>
3 #include "Node_chuong_7.h"
4
5 using namespace std;
6 using namespace linkedlistofclasses;
7
8 void Head_Insert(NodePtr& head, int the_number)
9 {
10     NodePtr temp_ptr;
11     //Thiết lập temp_ptr->link tới head và
12     //thiết lập giá trị dữ liệu tới the_number
13     temp_ptr = new Node(the_number, head);
14     head = temp_ptr;
15 }
16
17 int main()
18 {
19     NodePtr head, tmp;
20     //Tạo danh sách các nút 4 -> 3 -> 2 -> 1 -> 0
21     head = new Node(0, NULL);
22     for (int i = 0; i < 5; i++)
23     {
24         Head_Insert(head, i);
25     }
26     //Duyệt qua danh sách và hiển thị tất cả các giá trị
27     tmp = head;
28     while (tmp != NULL)
29     {
30         cout << tmp->getData() << endl;
31         tmp = tmp->getLink();
32     }
33     //Xóa tất cả các nút trong danh sách trước khi thoát chương trình
34     tmp = head;
35     while (tmp != NULL)
36     {
37         NodePtr nodeToDelete = tmp;
38         tmp = tmp->getLink();
39         delete nodeToDelete;
40     }
41     return 0;
42 }

```

Hình 7.22: Chương trình sử dụng lớp danh sách liên kết.

Output chương trình Hình 7.22

4
3
2
1
0
0

Bài tập

1. Giải thích khái niệm con trỏ trong C++.
2. Cái gì có thể hiểu lầm xảy ra trong khai báo dưới đây:

```
int* int_ptr1, int_ptr2;
```

3. Kết quả của đoạn mã sau sẽ in ra như thế nào?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

4. Hãy nêu các tình huống có thể xảy ra đối với đoạn chương trình sau:

```
int *p1 = new int;
int *p2;
*p1 = 5;
p2 = p1;
cout<< *p2;
delete p1;
cout<< *p2;
*p2 = 10;
```

5. Hãy nêu các tình huống có thể xảy ra đối với đoạn chương trình sau:

```
int *p1 = new int;
int *p2 = new int[10];
*p1 = 5;
p2 = p1;
cout<< *p2;
```

6. Hãy nêu các tình huống có thể xảy ra đối với đoạn chương trình sau:

```
int *a = new int[10];
for (int i = 0; i <= 10; i++) {
    print a[i];
    a[i] = i;
}
```

7. Nhập từ bàn phím vào một danh sách các số nguyên. Hãy sử dụng cấu trúc mảng động để lưu giữ dãy số nguyên này và tìm số lượng số nguyên chia hết cho số nguyên đầu tiên trong dãy. Hiện kết quả ra màn hình.
8. Sử dụng cấu trúc mảng động để lưu giữ một danh sách tên các sinh viên nhập vào từ bàn phím. Hãy sắp xếp danh sách tên sinh viên tăng dần theo độ dài của tên. Hiện ra màn hình danh sách sau khi đã sắp xếp.

Chương 8

Vào ra dữ liệu

Dữ liệu được nhập vào có thể từ bàn phím hoặc từ tệp (*file*). Tương tự, dữ liệu xuất ra sẽ được gửi tới màn hình hoặc file. Trong chương này của giáo trình sẽ trình bày và giải thích chúng ta có thể viết chương trình nhập dữ liệu từ file và xuất dữ liệu ra file như thế nào?

8.1 Dòng và vào ra file cơ bản

Chúng ta đã thực sự sử dụng file để lưu trữ chương trình của bạn. Chúng ta cũng có thể sử dụng file để lưu trữ đầu vào của chương trình hoặc nhận kết quả xuất ra từ chương trình. Dòng dữ liệu (*data stream*) cho phép chúng ta viết chương trình xử lý dữ liệu vào từ file và dữ liệu vào từ bàn phím theo một cách thống nhất và tương tự đối với xuất dữ liệu ra file và màn hình.

Dòng dữ liệu là dòng các ký tự (hoặc một loại khác của dữ liệu). Nếu dòng là đầu vào của chương trình, chúng ta gọi là dòng vào. Nếu dòng là đầu ra của chương trình, dòng được gọi là dòng ra. Nếu dòng vào từ bàn phím, thì chương trình của chúng ta sẽ là nhập dữ liệu từ bàn phím. Nếu dòng vào từ file thì chương trình của chúng ta sẽ là nhập dữ liệu từ file đó. Tương tự, dòng ra có thể xuất ra màn hình hoặc ra file.

Mặc dù, chúng ta có thể không nhận ra, nhưng thực sự đã từng sử dụng các dòng dữ liệu trong chương trình của chúng ta. Đối tượng `cin` mà chúng ta đã sử dụng là một dòng vào kết nối với bàn phím, và `cout` là một dòng ra kết nối với màn hình. Hai dòng này tự động có sẵn cho chương trình của bạn, miễn là nó có một chỉ thị biên dịch của tập tin khai báo `<iostream>`. Chúng ta có thể xác định dòng khác vào hoặc ra từ file; một khi đã xác định, chúng ta có thể sử dụng chúng trong chương trình của chúng ta như cách sử dụng các dòng vào `cin` và dòng ra `cout`.

Ví dụ, giả sử chương trình của chúng ta định nghĩa dòng dữ liệu vào `in_stream` từ một file nào đó (sẽ đề cập trong phần sau). Sau đó, chúng ta có thể đọc biến nguyên từ file vào biến nguyên mà chúng ta đã khai báo `i_number`.

```
int i_number;  
in_stream >> i_number;
```

Tương tự, nếu chương trình định nghĩa dòng dữ liệu ra `out_stream` với một file nào đó thì chúng ta có thể xuất giá trị của biến ra file đó.

```
out_stream << "the number is " << i_number << endl;
```

Khi dòng dữ liệu được kết nối với file mong muốn thì chương trình thao tác vào ra với file hoàn toàn giống như chương trình thao tác vào ra với bàn phím và màn hình.

8.2 Vào ra file

Để thực hiện vào ra file, chúng ta phải sử dụng `<fstream>` trong chương trình. File này định nghĩa một vài lớp về vào ra bao gồm `ifstream`, `ofstream` và `fstream`.

Trong C++, chúng ta mở file bằng cách kết nối nó với dòng dữ liệu. Trước khi bạn có thể mở file, trước tiên chúng ta phải có dòng dữ liệu. Có 3 kiểu của dòng dữ liệu là: vào, ra và vào/ra. Để tạo dòng vào, chúng ta phải mô tả dòng dữ liệu với lớp `ifstream` và đối với dòng dữ liệu ra, chúng ta phải mô tả nó với lớp `ofstream`. Dòng mà thực hiện cả vào/ra dữ liệu, chúng ta mô tả với lớp `fstream`. Ví dụ đoạn chương trình sau tạo một dòng vào, một dòng ra và một dòng có khả năng cả vào/ra dữ liệu:

```
ifstream in; // dòng vào
ofstream out; // dòng ra
fstream io; // dòng vào/ra
```

8.2.1 Mở file

Khi chúng ta tạo dòng dữ liệu, cách để liên kết nó với file là việc sử dụng hàm `open()`. Hàm này là hàm thành viên của 3 lớp dòng dữ liệu mà đề cập ở trên. Nguyên mẫu hàm như sau:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);
void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);
void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

Ở đây, `filename` là tên của file; nó có thể bao gồm đường dẫn chỉ rõ nơi chứa file. Giá trị của `mode` xác định file được mở như thế nào. Nó phải là một hoặc nhiều giá trị sau đây có kiểu `openmode` và liệt kê trong lớp `ios`:

`ios::app`: Mở để bổ sung tiếp vào cuối file. Giá trị này chỉ dùng cho dòng mở file để ghi.

`ios::ate`: Thiết lập đọc/ghi ở cuối file khi file được mở.

`ios::binary`: Mở file ở dạng nhị phân. Mặc định file được mở dạng văn bản.

`ios::in`: Mở để đọc từ file

`ios::out`: Mở để ghi vào file

`ios::trunc`: Nội dung file tồn tại sẽ bị xóa đi (file mở đã có từ trước) để ghi nội dung mới.

Chúng ta có thể kết hợp hai hoặc nhiều giá trị này bằng phép toán tử bit OR (`|`). Khi tạo dòng dữ liệu sử dụng `ofstream`, bất cứ file được mở mà tồn tại thì nội dung sẽ bị xóa bỏ.

Đoạn chương trình sau như cách chuẩn tắc cho việc mở file để ghi:

```
ofstream out;
out.open("test", ios::out);
```

Tuy nhiên, chúng ta sẽ hiếm khi sử dụng `open()` như trên bởi vì tham số `mode` cung cấp giá trị mặc định cho mỗi kiểu dòng dữ liệu. Như nguyên mẫu hàm đã đề cập ở trên thì `ofstream` thì `mode` mặc định là `ios::out|ios::trunc`. Vì vậy, đoạn chương trình có thể thường được viết như sau:

```
ofstream out;
out.open("test"); // defaults to output and normal file
```

Nếu `open()` bị thất bại thì dòng dữ liệu sẽ ước lượng là sai khi sử dụng biểu thức Boolean. Vì vậy trước khi sử dụng file, chúng ta nên kiểm tra chắc chắn thao tác mở file là thành công. Chúng ta có thể sử dụng lệnh sau:

```
if(!mystream) {
    cout<< "Cannot open file.\n";
    // handle error
}
```

Mặc dù, hoàn toàn thích hợp để mở file sử dụng hàm `open()` nhưng chúng ta thường sử dụng thích sử dụng hàm tạo tử của lớp `ifstream`, `ofstream` và `fstream` mà nó tự động thực hiện thao tác mở file. Hàm tạo tử này có cùng tham số và giá trị mặc định như hàm `open()`. Vì vậy, trong thực tế hầu hết chúng ta mở file theo cách sau:

```
ifstream mystream("myfile"); // open file for input
```

Chúng ta cũng có một cách khác kiểm tra việc mở file thành công hay không bằng cách sử dụng hàm `is_open()`. Đây là hàm thành viên của lớp `fstream`, `ifstream` và `ofstream`.

8.2.2 Đóng file

File nên được đóng khi chương trình chúng ta hoàn thành công việc đọc và ghi dữ liệu với file này. Đóng file sẽ ngắt kết nối giữa dòng dữ liệu với file. File được đóng sử dụng hàm thành viên `close()`. Ví dụ, để đóng file liên kết với dòng dữ liệu `mystream`, sử dụng lệnh sau:

```
mystream.close();
```

Chú ý là hàm `close` không có tham số. Nếu chương trình kết thúc bình thường nhưng không đóng file thì hệ thống sẽ tự động đóng file cho ta.

Tuy nhiên, thói quen tốt là nên phải đóng file sau khi hoàn thành đọc ghi file với một vài lý do sau (1): hệ thống sẽ chỉ đóng file cho chúng ta nếu chương trình kết thúc bình thường, như vậy nếu chương trình kết thúc bất thường nguyên nhân do lỗi thì file sẽ không được đóng và có thể gây ra hỏng file và mất mát dữ liệu trên file; (2): chúng ta có thể muốn chương trình gửi kết quả ra file và sau đó đọc dữ liệu đó vào trong chương trình. Để làm điều này, chương trình nên đóng file sau khi nó kết thúc ghi vào file và sau đó mở lại file với dòng dữ liệu vào.

8.3 Vào ra với file văn bản

Mặc định, tất cả các file được mở theo định dạng văn bản. Trong định dạng file văn bản một số ký tự có thể bị thay đổi, ví dụ như cặp ký tự xuống dòng và về đầu dòng là 13 (CR) và 10 (LF) được chuyển thành 1 ký tự là LF (với file dạng nhị phân thì 2 ký tự này vẫn giữ nguyên). Chúng ta dễ dàng đọc và ghi file văn bản. Đơn giản sử dụng toán tử `<<` và `>>` hoàn toàn giống như cách chúng ta thực hiện với vào ra dữ liệu từ màn hình và bàn phím với việc thay thế sử dụng `cin` và `cout` bằng dòng dữ liệu kết nối với file. Ví dụ, chương trình dưới đây tạo ra file `dulieu.txt` chứa một số sản phẩm và giá của nó.

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     ofstream out("dulieu.txt"); // output, normal file
8
9     if(!out) {
10         cout << "Cannot open INVENTORY file.\n";
11         return 1;
12     }
13
14     out << "Radios " << 39.95 << endl;
15     out << "Toasters " << 19.95 << endl;
16     out << "Mixers " << 24.80 << endl;
17
18     out.close();
19     return 0;
20 }
```

Hình 8.1: Thao tác ghi vào file văn bản.

Chương trình tiếp theo sẽ đọc dữ liệu từ file `dulieu.txt` được tạo trong chương trình 8.2 và xuất ra màn hình nội dung của file này.

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     ifstream in("dulieu.txt"); // input
8
9     if(!in) {
10         cout << "Cannot open INVENTORY file.\n";
11         return 1;
12     }
13
14     char item[20];
15     float cost;
16
17     in >> item >> cost;
18     cout << item << " " << cost << "\n";
19     in >> item >> cost;
20     cout << item << " " << cost << "\n";
21     in >> item >> cost;
22     cout << item << " " << cost << "\n";
23
24     in.close();
25     return 0;
26 }
```

Hình 8.2: Thao tác đọc dữ liệu từ file văn bản và xuất ra màn hình.

Output chương trình Hình 8.2:

```
Radios 39.95
Toasters 19.95
Mixers 24.8
```

8.4 Vào ra với file nhị phân

Trong khi đọc và ghi file dạng văn bản là khá dễ dàng, tuy nhiên nó không phải là cách hiệu quả nhất để xử lý file. Phần này, chúng ta đề cập đến cách đọc và ghi file dạng nhị phân. Khi thực hiện với file nhị phân, chắc chắn phải mở file sử dụng cờ `ios::binary`.

Để đọc và ghi dữ liệu dạng file nhị phân, C++ cung cấp cho chúng ta hàm `read()` và `write()` tương ứng. Dạng nguyên mẫu như sau:

```
istream&read(char *buf, streamsize num);
ostream&write(const char *buf, streamsize num);
```

Hàm `read()` đọc `num` ký tự từ dòng dữ liệu và đặt vào trong vùng nhớ đệm được trả bởi con trỏ `char buf`. Hàm `write()` viết `num` ký tự vào trong dòng dữ liệu từ vùng nhớ đệm được trả bởi `buf`. Kiểu `streamsize` là kiểu số nguyên được định nghĩa bởi thư viện C++.

Chương trình dưới đây minh họa viết một cấu trúc vào đĩa và sau đó đọc nó và in ra màn hình.

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstring>
4 using namespace std;
5
6 struct status {
7     char name[80];
8     double balance;
9     unsigned long account_num;
10 };
11
12 int main()
13 {
14     struct status acc;
15
16     strcpy(acc.name, "Nguyen Van A");
17     acc.balance = 1123.23;
18     acc.account_num = 34235678;
19
20     // write data
21     ofstream outbal("balance", ios::out | ios::binary);
22     if(!outbal) {
23         cout << "Cannot open file.\n";
24         return 1;
25     }
26
27     outbal.write((char *) &acc, sizeof(struct status));
28     outbal.close();
29
```

```

30 // now, read back;
31 ifstream inbal("balance", ios::in | ios::binary);
32 if(!inbal) {
33     cout << "Cannot open file.\n";
34     return 1;
35 }
36
37 inbal.read((char *) &acc, sizeof(struct status));
38
39 cout << acc.name << endl;
40 cout << "Account # " << acc.account_num;
41 cout.precision(2);
42 cout.setf(ios::fixed);
43 cout << endl << "Balance: $" << acc.balance;
44
45 inbal.close();
46
47 return 0;
48 }

```

Hình 8.3: Ghi và đọc dữ liệu từ tệp nhị phân.

Output chương trình Hình 8.3:

```

Nguyen Van A
Account # 34235678
Balance: $1123.23

```

Như chúng ta thấy chỉ cần một lần lời gọi `read()` hoặc `write()` để đọc hoặc ghi toàn bộ cả cấu trúc. Mỗi trường trong cấu trúc không cần phải đọc và ghi riêng biệt. Chương trình trên minh họa vùng đệm có thể là kiểu đối tượng bất kỳ.

8.5 Truy cập ngẫu nhiên

Trong hệ thống vào ra file của C++, chúng ta thực hiện truy cập ngẫu nhiên sử dụng hàm `seekg()` và `seekp()`.

```

istream&seekg(off_type offset, seekdir origin);
ostream&seekp(off_type offset, seekdir origin);

```

Ở đây, `off_type` là kiểu nguyên được định nghĩa trong lớp `ios`. Kiểu `seekdir` là kiểu liệt kê trong lớp `ios` mà xác định việc tìm kiếm sẽ xảy ra như thế nào.

Hệ thống vào ra của C++ quản lý hai con trỏ liên kết với file. Một con trỏ là `get` (con trỏ đọc) mà xác định vị trí trong file mà thao tác đọc tiếp theo được thực thi. Con trỏ còn lại là `put` (con trỏ ghi), nó xác định vị trí trong file mà thao tác ghi tiếp theo được thực thi. Mỗi khi thao tác đọc hoặc ghi được thực thi, con trỏ thích hợp sẽ tự động tăng lên. Tuy nhiên, sử dụng hàm `seekg()` và `seekp()` cho phép chúng ta truy cập file theo kiểu không tuần tự. Hàm `seekg()` sẽ dịch con trỏ `get` liên kết với file đi `offset` ký tự từ vị trí `origin`. Giá trị của `origin` như sau:

```

ios::beg Beginning-of-file
ios::cur Current location
ios::end End-of-file

```

Hàm `seekp()` sẽ dịch con trỏ put liên kết với file đi `offset` ký tự từ vị trí `origin`. Một cách tổng quát, truy cập vào ra ngẫu nhiên chỉ được thực hiện với những thao tác trên file nhị phân. Sự biến đổi ký tự xuất hiện trong file văn bản có thể là nguyên nhân mà vị trí dịch chuyển là không đồng bộ với nội dung thực sự trong file.

Chương trình tiếp theo trong hình 8.4 minh họa hàm `seekp()`. Nó cho phép thay đổi ký tự chỉ rõ trong file. Đặc biệt, tên file được đưa vào từ tham số dòng lệnh, tiếp theo là vị trí trong file mà chúng ta muốn thay đổi, cuối cùng là ký tự mới.

```

1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     if(argc!=4) {
9         cout << "Usage: CHANGE <filename> <character> <char>\n";
10        return 1;
11    }
12
13    fstream out(argv[1], ios::in | ios::out | ios::binary);
14    if(!out) {
15        cout << "Cannot open file.";
16        return 1;
17    }
18
19    out.seekp(atoi(argv[2]), ios::beg);
20
21    out.put(*argv[3]);
22    out.close();
23
24    return 0;
25 }
```

Hình 8.4: Chương trình minh họa sử dụng hàm `seekp()`.

Output chương trình Hình 8.4 khi chạy lệnh:change dulieu.txt 3 Z

```

RadZos 39.95
Toasters 19.95
Mixers 24.8
```

Chúng ta có thể xác định vị trí hiện thời của con trỏ file (con trỏ đọc hoặc ghi) bằng cách sử dụng hàm:

```

pos_type tellg();
pos_type tellp();
```

Ở đây, `pos_type` là kiểu nguyên được định nghĩa bởi `ios`. Chúng ta có thể sử dụng giá trị trả về của `tellg()` và `tellp()` như là tham số của hàm `seekg()` và `seekp()` tương ứng.

Bài tập

- Viết chương trình đếm số dòng của một file văn bản.
- Viết chương trình in nội dung file ra màn hình và cho biết tổng số chữ cái, tổng số chữ số đã xuất hiện trong file.
- Nhập vào số nguyên dương n . Tìm tất cả các số nguyên tố nhỏ hơn n và ghi ra file `nguyento.txt`
- Viết chương trình lưu 2000 số thực bất kỳ (sử dụng hàm ngẫu nhiên) vào file văn bản `sothuc.txt`. Mỗi dòng bao gồm 20 số. Sau đó hãy đọc các số thực trong file `sothuc.txt` trên và sắp xếp các số theo thứ tự tăng dần và lưu trong file `ketqua.txt`.
- Viết chương trình giả lập lệnh `type` để in nội dung file văn bản ra màn hình.
- Cho file văn bản `vidu1.txt` (đã chứa một số thông tin có sẵn). Hãy viết chương trình kiểm tra xem tập văn bản `vidu2.txt` có tồn tại hay không? Nếu tồn tại thì ghi tiếp vào cuối file nội dung của file `vidu1.txt`, ngược lại thì tạo mới và ghi nội dung của file `vidu1.txt`.
- Nhập bằng chương trình 2 ma trận số nguyên vào 2 file văn bản. Hãy tạo file văn bản thứ 3 chứa nội dung của ma trận tích của 2 ma trận trên.
- Viết chương trình tìm dãy con (là dãy bao gồm các phần tử liên tiếp trong dãy ban đầu) đơn điệu tăng và có tổng là lớn nhất trong dãy n số nguyên cho trước. Thông tin được lưu trong file `mang.txt`. Kết quả tìm được cũng được lưu trong file `ketqua.txt`.
File văn bản `mang.txt` có cấu trúc như sau:
 - Dòng đầu tiên ghi n (n là số nguyên dương).
 - Trong các dòng tiếp theo ghi n số nguyên ngẫu nhiên, mỗi dòng 10 số (các số cách nhau ít nhất một khoảng trống).File `ketqua.txt` ghi ra dãy con tìm được trên các dòng (mỗi dòng 10 số).
- Tổ chức quản lý file sinh viên dưới dạng nhị phân (Họ tên, ngày sinh, giới tính, điểm) với các chức năng: Nhập, xem, xóa, sửa, tính điểm trung bình chung.
- Thông tin về một nhân viên trong cơ quan bao gồm: Họ và tên, nghề nghiệp, số điện thoại, địa chỉ nhà riêng. Viết hàm nhập từ bàn phím thông tin của 7 nhân viên và ghi vào file `INPUT.DAT`. Viết hàm tìm trong file `INPUT.DAT` và in ra thông tin của 1 nhân viên theo số điện thoại được nhập từ bàn phím.

Chương 9

Xử lý ngoại lệ

Khi mới viết chương trình, ta thường viết mã với giả thiết rằng không có gì bất thường sẽ xảy ra với chương trình của chúng ta. Sau đó, khi chương trình đã chạy đúng, ta sẽ bắt đầu xem xét các trường hợp ngoại lệ có thể gây ra lỗi. Trong C++, bạn có thể sử dụng các tiện ích của ngôn ngữ để xử lý các trường hợp ngoại lệ như vậy.

Một lý do khác cho xử lý ngoại lệ là khi một hàm có thể được sử dụng bởi nhiều người hoặc nhiều chương trình khác nhau. Khi đó, mỗi chương trình sẽ có cách xử lý từng trường hợp ngoại lệ riêng. Như vậy, ta cần một cơ chế thông báo các trường hợp ngoại lệ từ một hàm ra bên ngoài. Từ đó, đoạn mã gọi hàm sẽ có khả năng xử lý ngoại lệ theo cách riêng của mình.

Trong C++, xử lý ngoại lệ diễn ra theo các bước sau: Đầu tiên, các hàm thư viện hoặc đoạn mã của bạn phát ra tín hiệu rằng có một trường hợp ngoại lệ đã xảy ra (được gọi là *ném ngoại lệ*). Sau đó, ở một chỗ khác trong chương trình sẽ có đoạn mã giải quyết trường hợp ngoại lệ này (gọi là *xử lý ngoại lệ* hay *bắt ngoại lệ*). Cách viết chương trình như vậy làm chương trình *sạch hơn*.

9.1 Các vấn đề cơ bản trong xử lý ngoại lệ

9.1.1 Ví dụ xử lý ngoại lệ

Chúng ta sẽ bắt đầu với một ví dụ đơn giản nhất trong xử lý ngoại lệ. Chương trình trong Hình 9.1 nhập vào tổng số tiền thưởng và số người được thưởng rồi in ra giá trị trung bình của tiền thưởng cho mỗi người. Rõ ràng, ta phải kiểm tra xem người dùng có nhập vào số lượng người lớn hơn 0 hay không và cho thông báo cho từng trường hợp. Chương trình trong Hình 9.1 thể hiện cách xử lý trường hợp bình thường không xử dụng ngoại lệ.

```
1 //Chương trình minh họa xử lý trường hợp đặc biệt
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int reward, number;
8     cout << "Enter total reward:\n";
9     cin >> reward;
10    cout << "Enter number of people:\n";
11    cin >> number;
```

```

12
13     if (number <= 0) {
14         cout << "Number of people should be positive." << endl;
15     } else {
16         double average = (double) reward / number;
17         cout << "Average reward is " << average << endl;
18     }
19 }

```

Hình 9.1: Xử lý trường hợp đặc biệt không dùng ngoại lệ.

Chương trình trong Hình 9.2 là chương trình trong Hình 9.1 được viết lại bằng cách xử lý ngoại lệ. Mặc dù các chương trình này còn đơn giản nhưng chúng cho thấy việc tách các đoạn mã xử lý chính (trường hợp bình thường) và đoạn mã xử lý trường hợp đặc biệt thành hai khối rời nhau.

```

1 //Chương trình minh họa xử lý ngoại lệ
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     try {
8         int reward, number;
9         cout << "Enter total reward:\n";
10        cin >> reward;
11        cout << "Enter number of people:\n";
12        cin >> number;
13
14        if (number <= 0)
15            throw number;
16
17        double average = (double) reward / number;
18        cout << "Average reward is " << average << endl;
19    }
20    catch (int e) {
21        cout << "Number of people = " << e << " (should be positive)." << endl;
22    }
23 }

```

Hình 9.2: Chương trình giống Hình 9.1 nhưng dùng ngoại lệ.

Trong Hình 9.2, đoạn mã **if-else** đã được thay bằng một lệnh **if** và lệnh **throw** như sau

```

if (number <= 0)
    throw number;

```

Lệnh này phát ra tín hiệu một trường hợp đặc biệt đã xảy ra. Như vậy, trong khối **try** là đoạn mã xử lý các trường hợp *bình thường* cùng với việc phát các tín hiệu ngoại lệ. Còn trong khối *catch* là đoạn mã xử lý các trường hợp ngoại lệ này.

Với C++, việc xử lý ngoại lệ có thể thực hiện bằng bộ ba từ khóa **try-catch-throw**. Cú pháp của khối **try** như sau:

```

try {
    <Đoạn mã cho trường hợp bình thường>
    <Phát tín hiệu ngoại lệ bằng throw>
}

```

```
<Các đoạn mã khác>
}
```

Cú pháp của lệnh **throw** như sau:

```
throw <Giá trị của một biểu thức>
```

Giá trị được ném (**throw**) được gọi là *ngoại lệ*. Việc thực hiện lệnh **throw** được gọi là *ném ngoại lệ*. Bạn có thể ném ngoại lệ có kiểu bất kì. Trong Hình 9.2, ngoại lệ được ném có giá trị kiểu **int**.

Khi lệnh **throw** được thực hiện, đoạn mã trong khối **try** dừng và chương trình chuyển đến đoạn mã trong một khối **catch**. Do đó, đoạn mã trong khối **catch** sẽ *bắt ngoại lệ* hoặc *xử lý ngoại lệ*. Khối **catch** nào được thực thi phụ thuộc vào ngoại lệ được ném đi và khai báo xử lý ngoại lệ của khối **catch**. Trong hình 9.2, khối **catch** như sau:

```
catch (int e) {
    cout << "Number of people = " << e
        << " (should be positive)." << endl;
}
```

Khối **catch** này khai báo rằng nó chỉ xử lý ngoại lệ có kiểu **int**. Do đó, nếu trong khối **try** phát tín hiệu ngoại lệ kiểu **int** (**throw** <Ngoại lệ kiểu **int**>), khối **catch** này sẽ được gọi để xử lý. Việc khai báo ngoại lệ của khối **catch** nhằm hai mục đích:

1. Khai báo kiểu ngoại lệ khối **catch** sẽ xử lý.
2. Đặt tên cho ngoại lệ để dùng trong khối **catch**.

Trong đoạn mã trên, biến **e** sẽ có giá trị bằng giá trị của **number** trong khối **try** khi khối này phát tín hiệu ngoại lệ. Rõ ràng, biến **e** luôn có giá trị nhỏ hơn hoặc bằng 0 vì khối **catch** chỉ được gọi khi **number** <= 0.

9.1.2 Định nghĩa lớp ngoại lệ

Lệnh **throw** có thể ném ngoại lệ có kiểu bất kì. Do đó, ta có thể định nghĩa một lớp đối tượng thể hiện các thông tin của ngoại lệ ta cần “ném” đi. Một lý do khác để định nghĩa lớp ngoại lệ riêng là ta muốn có các kiểu ngoại lệ khác nhau cho từng trường hợp ngoại lệ khác nhau.

Một lớp ngoại lệ chỉ là một lớp đối tượng bình thường. Chỉ khi nào ta dùng đối tượng của lớp này trong lệnh **throw** thì nó mới gọi là ngoại lệ. Chương trình trong Hình 9.3 thể hiện cách dùng lớp ngoại lệ.

```
1 //Chương trình minh họa định nghĩa lớp ngoại lệ
2 #include <iostream>
3 using namespace std;
4
5 class BadNumber {
6     int number;
7 public:
8     BadNumber(int number_)
9         : number(number_)
10    {}
11
12    int getNumber()
```

```

13     {
14         return number;
15     }
16 };
17
18 int main()
19 {
20     try {
21         int reward, number;
22         cout << "Enter total reward:\n";
23         cin >> reward;
24         cout << "Enter number of people:\n";
25         cin >> number;
26
27         if (number <= 0)
28             throw BadNumber(number);
29
30         double average = (double) reward / number;
31         cout << "Average reward is " << average << endl;
32     }
33     catch (BadNumber e) {
34         cout << "Number of people = " << e.getNumber()
35              << " (should be positive)." << endl;
36     }
37 }

```

Hình 9.3: Định nghĩa lớp ngoại lệ.

Lệnh **throw** của chương trình này

```
throw BadNumber(number);
```

khởi tạo một đối tượng **BadNumber** bằng cách gọi hàm khởi tạo và ném đối tượng (ngoại lệ) này đi.

9.1.3 Ném và bắt nhiều ngoại lệ

Khối **try** có thể có nhiều lệnh **throw** để ném các ngoại lệ với các kiểu khác nhau. Tất nhiên trong chương trình mỗi khối **try** chỉ ném được một ngoại lệ do lệnh **throw** sẽ khiến chương trình thoát khỏi khối **try**. Tương ứng với mỗi kiểu ngoại lệ sẽ có một khối **catch** tương ứng bắt lấy những ngoại lệ cùng kiểu. Các khối **catch** được viết liên tiếp sau khối **try** như ví dụ trong Hình 9.4.

```

1 //Chương trình minh họa việc ném nhiều kiểu ngoại lệ
2 #include <iostream>
3 using namespace std;
4
5 class BadNumber {
6     int number;
7 public:
8     BadNumber(int number_)
9         : number(number_)
10    {}
11
12     int getNumber()

```



```

13     {
14         return number;
15     }
16 };
17
18 class DivideByZero {};
19
20 int main()
21 {
22     try {
23         int reward, number;
24         cout << "Enter total reward:\n";
25         cin >> reward;
26         cout << "Enter number of people:\n";
27         cin >> number;
28
29         if (number < 0)
30             throw BadNumber(number);
31         else if (number == 0)
32             throw DivideByZero();
33
34         double average = (double) reward / number;
35         cout << "Average reward is " << average << endl;
36     }
37     catch (BadNumber e) {
38         cout << "Number of people = " << e.getNumber()
39             << " is negative." << endl;
40     }
41     catch (DivideByZero) {
42         cout << "Cannot divide by Zero" << endl;
43     }
44 }

```

Hình 9.4: Ném và bắt nhiều ngoại lệ.

Khi có nhiều khối `catch`, bạn nên viết các khối `catch` theo thứ tự tăng dần về tính tổng quát của kiểu ngoại lệ do việc lựa chọn các khối `catch` tuân theo thứ tự xuất hiện chúng. Ví dụ nếu lớp `SUV` thừa kế lớp `Car` thì bạn nên viết `catch (SUV)` trước `catch (Car)`. Ngoài ra, nếu bạn muốn bắt mọi loại ngoại lệ, hãy viết `catch (...)` với 3 dấu chấm trong ngoặc tròn. Tất nhiên, khối `catch (...)` nên được viết cuối cùng.

9.1.4 Ném ngoại lệ từ hàm

Khi ta viết một hàm, hàm này có thể phát tín hiệu ngoại lệ mà không cần phải xử lý ngoại lệ ngay trong hàm. Thay vào đó, đoạn mã gọi hàm này đặt lời gọi hàm trong một khối `try` và xử lý ngoại lệ phát ra từ hàm trong một khối `catch`. Cơ chế này giúp ta viết các hàm thư viện có thể ném ngoại lệ cho đoạn mã gọi hàm của người sử dụng thư viện. Đoạn mã gọi hàm thư viện có trách nhiệm xử lý ngoại lệ theo cách riêng của người sử dụng thư viện. Chương trình trong Hình 9.5 minh họa cách ném ngoại lệ từ hàm và xử lý ngoại lệ trong đoạn mã gọi hàm. Như bạn thấy, khối `try` giờ đã “trơ trọi” hơn rất nhiều so với phiên bản đầu tiên. Khối này hầu như chỉ xử lý các trường hợp “bình thường” không phát sinh ngoại lệ. Việc ném ngoại lệ được chuyển cho hàm còn việc xử lý

ngoại lệ nằm riêng ở các khối **catch**.

```

1 //Chương trình minh họa việc ném ngoại lệ từ hàm
2 #include <iostream>
3 using namespace std;
4
5 class BadNumber {
6     int number;
7 public:
8     BadNumber(int number_)
9         : number(number_)
10    {}
11
12    int getNumber()
13    {
14        return number;
15    }
16 };
17
18 class DivideByZero {};
19
20 double ComputeAverage(int reward, int number) throw (BadNumber, DivideByZero)
21 {
22     if (number < 0)
23         throw BadNumber(number);
24     else if (number == 0)
25         throw DivideByZero();
26     else
27         return (double) reward / number;
28 }
29
30 int main()
31 {
32     try {
33         int reward, number;
34         cout << "Enter total reward:\n";
35         cin >> reward;
36         cout << "Enter number of people:\n";
37         cin >> number;
38
39         double average = ComputeAverage(reward, number);
40         cout << "Average reward is " << average << endl;
41     }
42     catch (BadNumber e) {
43         cout << "Number of people = " << e.getNumber()
44             << " is negative." << endl;
45     }
46     catch (DivideByZero) {
47         cout << "Cannot divide by Zero" << endl;
48     }
49 }

```

Hình 9.5: Ném và xử lý ngoại lệ phát ra từ hàm.

9.1.5 Mô tả ngoại lệ

Nếu một hàm có thể phát sinh ngoại lệ mà không tự xử lý ngoại lệ này, bạn cần khai báo các kiểu ngoại lệ như vậy trong khai báo hàm. Trong Hình 9.5, đoạn mã khai báo hàm

```
double computeAverage(int reward, int number)
    throw (BadNumber, DivideByZero)
```

cho ta biết rằng hàm **computeAverage** có thể ném các ngoại lệ kiểu **BadNumber** và **DivideByZero**. Hàm này sẽ không tự xử lý các ngoại lệ thuộc hai lớp này mà đoạn mã gọi hàm có trách nhiệm xử lý chúng. Lưu ý khai báo hàm bao gồm cả việc khai báo ngoại lệ nên việc khai báo hàm và cài đặt hàm phải thống nhất cả về danh sách ngoại lệ này.

Trong C++, nếu bạn khai báo danh sách ngoại lệ trong khai báo hàm thì mọi ngoại lệ phát ra từ hàm mà không được xử lý trong hàm phải nằm trong danh sách ngoại lệ. Nếu ngoại lệ phát sinh không nằm trong danh sách ngoại lệ, chương trình sẽ dừng ngay lập tức. Tuy nhiên, nếu bạn không khai báo danh sách ngoại lệ của hàm, C++ coi như hàm có thể phát sinh ngoại lệ với kiểu bất kì mà không dừng chương trình. Trường hợp bạn khai báo danh sách ngoại lệ rỗng **throw ()**, hàm của bạn phải xử lý tất cả các ngoại lệ có thể phát sinh.

9.2 Kỹ thuật lập trình cho xử lý ngoại lệ

Ở mục trước, chúng ta đã sử dụng một ví dụ hết sức đơn giản và ngắn gọn để trình bày các cú pháp, cách sử dụng và xử lý ngoại lệ trong C++. Trong mục này, chúng ta sẽ xem xét các ví dụ phức tạp hơn nhằm trình bày các kỹ thuật lập trình cho việc xử lý ngoại lệ.

9.2.1 Ném ngoại lệ ở đâu

Thông thường, khi lập trình xử lý ngoại lệ, ta nên tách các đoạn mã ném ngoại lệ và xử lý ngoại lệ ra các hàm khác nhau như các đoạn mã sau:

```
void A() throw (MyException)
{
    ...
    throw MyException(<tham số khởi tạo>)
    ...
}

void B()
{
    try {
        ...
        A();
        ...
    }
    catch (MyException e) {
        <Xử lý ngoại lệ e>
    }
}
```

Như ta thấy, hàm `A()` khai báo rằng nó có thể ném ngoại lệ kiểu `MyException`. Còn hàm `B()` do gọi hàm `A()` nên nó cần xử lý ngoại lệ có thể xảy ra khi gọi `A()`. Việc tách các đoạn mã `throw` và `catch` ra các hàm khác nhau làm chương trình “trong sáng” hơn nhiều và hỗ trợ khả năng ném ngoại lệ từ các hàm thư viện. Chương trình sử dụng hàm thư viện chỉ việc viết đoạn mã `try` xử lý các trường hợp bình thường và viết riêng các đoạn mã `catch` xử lý từng trường hợp ngoại lệ có thể xảy ra khi gọi hàm thư viện.

9.2.2 Cây phả hệ ngoại lệ STL

C++ cung cấp một loạt kiểu ngoại lệ có thể phát sinh khi sử dụng thư viện chuẩn (Standard Template Library - STL) của C++. Kiểu ngoại lệ tổng quát nhất là kiểu `exception` được định nghĩa trong tiêu đề `<exception>`. Mọi ngoại lệ của STL đều thừa kế kiểu ngoại lệ này. Khai báo của kiểu `exception` như sau.

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
}
```

Trong đó, hàm `what()` cung cấp thông tin về ngoại lệ trong một chuỗi ký tự. Hình 9.6 cho thấy cách dùng thông báo ngoại lệ của C++.

```
1 //Chương trình minh họa ngoại lệ trong STL
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 int main()
7 {
8     try {
9         vector<int> v(10);
10        v.at(20) = 100; // ném ngoại lệ out_of_range
11    }
12    catch (exception& e) {
13        cout << "Error occurred what = " << e.what() << endl;
14    }
15 }
```

Hình 9.6: Sử dụng ngoại lệ của STL.

Output chương trình 9.6

```
Error occurred what = vector::_M_range_check
```

Lưu ý, ở đây ta dùng tham chiếu đến kiểu `exception`

```
catch (exception& e) {
    ...
}
```

để có thể gọi đến hàm `what()` của kiểu ngoại lệ `out_of_range` thừa kế kiểu `exception`. Bảng 9.7 mô tả cây phả hệ các kiểu ngoại lệ trong STL của C++. Trong đó, các ngoại lệ mà người mới

Bảng 9.7: Cây phả hệ ngoại lệ trong STL

Lớp ngoại lệ	Mô tả
<code>exception</code>	Lớp ngoại lệ tổng quát <code>#include <exception></code>
<code>bad_alloc</code>	Ngoại lệ khi không cấp phát được bộ nhớ
<code>bad_cast</code>	Ngoại lệ khi không thể <code>dynamic_cast</code>
<code>bad_function_call</code>	Ngoại lệ khi gọi hàm
<code>ios_base::failure</code>	Ngoại lệ khi nhập xuất
<code>logic_error</code>	Lỗi logic <code>#include <stdexcept></code>
<code>invalid_argument</code>	Tham số không hợp lệ
<code>length_error</code>	Kích thước không hợp lệ
<code>out_of_range</code>	Chỉ số mảng không hợp lệ
<code>runtime_error</code>	Lỗi run-time <code>#include <stdexcept></code>
<code>overflow_error</code>	Kết quả biểu thức quá lớn
<code>range_error</code>	Kết quả biểu thức vượt khỏi miền của kiểu dữ liệu
<code>underflow_error</code>	Kết quả biểu thức quá nhỏ

lập trình C++ hay gặp nhất là `bad_alloc`, `failure`, `out_of_range`.

9.2.3 Kiểm tra bộ nhớ

Khi bạn dùng toán tử `new` để xin cấp phát bộ nhớ, toán tử này có thể ném một ngoại lệ thuộc kiểu `bad_alloc` nếu nó không thể cấp phát bộ nhớ như được yêu cầu. Kiểu `bad_alloc` nằm trong cây phả hệ ngoại lệ của ngôn ngữ C++, bạn có thể dùng mà không cần định nghĩa lại. Ví dụ sau cho thấy cách dùng `bad_alloc`.

```
try {
    int* a = new int [1000000000000];
} catch (bad_alloc) {
    cout << "Cannot allocate the required memory" << endl;
}
```

Bài tập

1. Chương trình trong hình 9.8 cho thấy cách phát hiện một số thực quá nhỏ vượt qua khả năng thể hiện của máy tính. Bạn hãy sửa chương trình này để phát hiện xem với N bằng bao nhiêu thì máy tính không thể biểu diễn được xác suất tung đồng xu N lần đều được mặt ngửa. Giả sử 2 mặt sấp ngửa có xác suất như nhau.

```
1 #include <iostream>
2 #include <cfloat>
3 #include <stdexcept>
4 using namespace std;
5
6 int main()
7 {
8     try {
9         double d = DBL_MIN / 3.0;
10        cout << d << endl;
11        if (d < DBL_MIN)
12            throw underflow_error("Underflow occurred");
13    }
14    catch (exception& e) {
15        cout << "Error occurred what = " << e.what() << endl;
16    }
17 }
```

Hình 9.8: Phát hiện lỗi `underflow_error`.

Output chương trình 9.8

```
7.41691e-309
Error occurred what = Underflow occurred
```

2. Viết chương trình xin cấp phát bộ nhớ liên tục đến khi nhận được ngoại lệ `bad_alloc`. In ra kích thước bộ nhớ đã cấp phát được.
3. Viết chương trình **sao chép file** có xử lý ngoại lệ. Ví dụ: file không tồn tại, không thể tạo file, không thể đọc, không thể ghi. Thông báo giá trị `what()` của các ngoại lệ này. Tham số tên file nguồn và file đích nhập từ dòng lệnh.

Chương 10

Tiền xử lý và lập trình nhiều file

10.1 Các chỉ thị tiền xử lý

Như đã biết, trước khi chạy chương trình (từ văn bản chương trình tức chương trình nguồn) C++ sẽ dịch chương trình ra tệp mã máy còn gọi là chương trình đích. Thao tác dịch và chạy chương trình nói chung gồm có các phần:

- Xử lý sơ bộ chương trình (tiền xử lý).
- Kiểm tra lỗi cú pháp, dịch và tạo thành các mô đun chương trình (các file độc lập, chương trình con ...).
- Liên kết các mô đun chương trình vừa dịch và các mô đun có sẵn của C++ để tạo thành chương trình đích (mã máy) cuối cùng.
- Tải chương trình đích vào bộ nhớ và thực hiện.

Trong mục này, ta sẽ trình bày về bước 1, tức bước xử lý sơ bộ chương trình hay còn gọi là tiền xử lý, trong đó có các công việc liên quan đến các chỉ thị (thường) được đặt ở đầu tệp chương trình nguồn như `#include`, `#define`, `#ifndef` ...

Các khai báo dạng `#...` được gọi là chỉ thị tiền xử lý. Chú ý sau các khai báo này không có dấu chấm phẩy như sau các câu lệnh.

10.1.1 Chỉ thị bao hàm tệp `#include`

Yêu cầu ghép nội dung các tệp đã có vào chương trình trước khi dịch. Các tệp cần ghép thêm vào chương trình thường là các tệp chứa khai báo nguyên mẫu của các hằng, biến, hàm ... có sẵn trong C hoặc do lập trình viên tự viết. Có nghĩa, một chương trình có thể được viết trên nhiều file văn bản khác nhau, sau đó có thể ghép nối lại với nhau bằng các chỉ thị `#include`. Có hai dạng viết chỉ thị này.

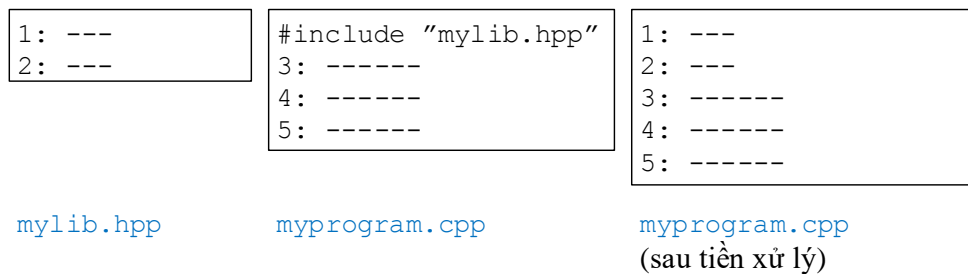
```
1: #include <file_name>
2: #include "path \ file_name"
```

Dạng khai báo 1 cho phép C++ ngầm định tìm tệp tại thư mục định sẵn (khai báo thông qua menu Options\Directories) thường là thư mục con `INCLUDE` của thư mục cài đặt C/C++ trên đĩa. Đây

là thư mục chứa các tệp nguyên mẫu của thư viện C/C++ (như `iostream`, `stdio.h`, `math.h`, `cstring` ...).

Dạng khai báo 2 cho phép tìm tệp theo đường dẫn (đến nơi khác với `INCLUDE`). Nếu chỉ có tên tệp (không có đường dẫn) sẽ ngầm định tìm trong thư mục hiện tại. Tệp thường là các thư viện được tạo bởi lập trình viên và được đặt trong cùng thư mục chứa chương trình. Cú pháp này cho phép lập trình viên chia một chương trình thành nhiều môđun đặt trên một số tệp khác nhau để dễ quản lý. Nó đặc biệt hữu ích khi lập trình viên muốn tạo các thư viện riêng cho mình. Dĩ nhiên, các thư viện riêng này cũng có thể đặt trong thư mục hệ thống `INCLUDE`, khi đó chỉ thị sẽ được khai báo theo dạng 1 (dùng `<>` thay cho `""`).

Khi gặp tên file đứng sau chỉ thị `#include`, chương trình dịch sẽ ghép nối toàn bộ văn bản của file này vào cùng chương trình ngay tại vị trí nó được khai báo. Có nghĩa chương trình trước khi dịch thực sự sẽ bao gồm cả 2 đoạn văn bản: của file được `#include` và của chương trình đang chứa `#include` này. Cụ thể như hình vẽ bên dưới:



Hình 10.1: Minh họa cơ chế `#include`.

10.1.2 Chỉ thị macro `#define`

Macro không đối

```
#define macro_name defined_macro
```

Trong khai báo trên, `macro_name` và `defined_macro` đều là các cụm từ. Trước khi dịch, bộ tiền xử lý sẽ tìm trong chương trình và thay thế bất kỳ vị trí xuất hiện nào của xâu `macro_name` thành xâu `defined_macro`.

Ta thường sử dụng macro để định nghĩa các hằng hoặc thay cụm từ này bằng cụm từ khác để nhớ hơn, ví dụ:

```
#define MAX 100      // thay MAX bằng 100
#define TRUE 1       // thay TRUE bằng 1
#define then         // thay then bằng xâu rỗng
#define begin {      // thay begin bằng dấu
#define end }        // thay end bằng dấu *)
```

Từ đó trong chương trình, ta có thể viết những đoạn lệnh như phần bên trái trong hình và sẽ được bộ tiền xử lý chuyển thành văn bản bên phải trước khi dịch:


```

if (i < MAX) then
begin
    Ok = TRUE;
    cout << i ;
end
    →
if (i < 100)
{
    Ok = 1;
    cout << i ;
}

```

Chú ý cách khai báo hằng: `const int MAX = 100;` và định nghĩa macro: `#define MAX 100` cho cùng kết quả lập trình nhưng chúng khác nhau về ý nghĩa và cách hoạt động.

Macro có đối

Ngoài việc dùng chỉ thị `#define` để định nghĩa hằng, nó còn được phép viết dưới dạng có đối để thay thế những hàm đơn giản. Ví dụ, để tìm số lớn nhất của 2 số (với kiểu bất kỳ) ta có thể viết một macro có đối đơn giản như sau:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Khi đó trong chương trình nếu có dòng `x = max(a, b)` thì nó sẽ được thay bởi: `x = ((a) > (b) ? (a) : (b))` và khi chạy x sẽ nhận được giá trị lớn nhất của a và b. Chú ý:

- Tên macro phải được viết liền với dấu ngoặc của danh sách đối. Ví dụ không viết `max (A, B)`.
- `#define square(X) (X*X)` viết sai về nội dung vì `square(5)` đúng (bằng `5*5`) nhưng `square(a + b)` sẽ thành `(a + b * a + b)` (tức `a + ba + b`).
- Cũng tương tự viết `#define max(A, B) (A > B ? A : B)` là sai (?) vì vậy luôn luôn bao các đối bởi dấu ngoặc.
- Kể cả `#define square(X) ((X)*(X))` viết đúng nhưng nếu giả sử lập trình viên muốn tính bình phương của 2 bằng đoạn lệnh sau:

```
int i = 1;
cout << square(++i);           // 9
```

thì kết quả in ra sẽ là 9 thay vì kết quả đúng là 4. Lí do là ở chỗ bộ tiền xử lý sẽ thay `square(++i)` bởi `((++i)*(++i))`, và với `i = 1` chương trình sẽ thực hiện như `3*3 = 9`. Do vậy cần cẩn thận khi sử dụng các phép toán tự tăng giảm trong các macro có đối.

Nói chung, nên hạn chế việc sử dụng các macro phức tạp, vì nó có thể gây nên những hiệu ứng phụ khó kiểm soát.

10.1.3 Các chỉ thị biên dịch có điều kiện `#if`, `#ifdef`, `#ifndef`

- Chỉ thị `#if`:

```

#if condition
    list_of_statements;
#endif
#if condition
    list_of_statements;
#else
    list_of_statements
#endif

```

Các chỉ thị này giống như câu lệnh `if`, mục đích của nó là báo cho chương trình dịch biết đoạn lệnh giữa `#if condition` và `#endif` chỉ được dịch nếu `condition` đúng. Ví dụ:

```
const int M = 1;
void main()
{
    int i = 5;
    #if M <= 1
        cout << i ;
    #endif
}
```

Trong chương trình trên, do $M \leq 1$ nên chương trình sẽ dịch và chạy câu `cout << i;` tức in ra màn hình giá trị `i = 5`.

- Chỉ thị `#ifdef name` và `#ifndef name`.

Chỉ thị này báo cho chương trình dịch biết đoạn lệnh có được dịch hay không khi một tên gọi (`name`) đã được định nghĩa hay chưa. `#ifdef` được hiểu là nếu tên đã được định nghĩa thì dịch, còn `#ifndef` được hiểu là nếu tên chưa được định nghĩa thì dịch. Để định nghĩa một tên gọi, ta dùng chỉ thị `#define name`.

Chỉ thị này đặc biệt có ích khi chèn (`#include`) các tệp thư viện vào file khác để sử dụng. Cụ thể, ví dụ tệp thư viện A được chèn vào file B và file C, và B cũng được chèn vào C. Như vậy, trong C xuất hiện 2 lần đoạn mã của A, khi dịch sẽ bị báo lỗi vì các đối tượng (trong A) bị khai báo trùng lặp hai lần. Để tránh việc này, ta cần sử dụng chỉ thị trên để báo cho chương trình dịch chỉ dịch đoạn mã A một lần (nếu đã dịch rồi thì sẽ không dịch nữa), như ví dụ minh họa sau:

Giả sử ta đã viết sẵn 2 tệp thư viện là `mylib.hpp` và `myfunc.hpp`, trong đó `mylib.hpp` chứa hàm `max(a, b)` tìm số lớn nhất giữa 2 số, `myfunc.hpp` chứa hàm `max(a, b, c)` tìm số lớn nhất giữa 3 số thông qua sử dụng hàm `max(a, b)`. Như vậy, `myfunc.hpp` phải có chỉ thị `#include mylib.hpp` để sử dụng được hàm `max(a, b)`. Dưới đây là cách tổ chức:

- Thư viện 1. tên tệp: `mylib.hpp`

```
int max(int a, int b)
{
    return (a > b ? a : b);
}
```

- Thư viện 2. tên tệp: `myfunc.hpp`

```
#include "mylib.hpp"
int max(int a, int b, int c)
{
    int tmp = max(a, b);
    return (tmp > c ? tmp : c);
}
```

Hàm `main` của chúng ta nhập 3 số, in ra `max` của từng cặp số và `max` của cả 3 số. Chương trình cần phải sử dụng cả 2 thư viện.

```

#include "mylib.hpp"
#include "myfunc.hpp"
main()
{
    int a, b, c;
    cout << "a, b, c = " ; cin >> a >> b >> c;
    cout << max(a, b) << max(b, c) << max(a, c) << max(a, b, c) ;
}

```

Trước khi dịch chương trình, bộ tiền xử lý sẽ chèn các thư viện vào trong tệp chính (chứa `main()`). Trong đó, `mylib.hpp` được chèn vào 2 lần (một lần của tệp chính và một lần được chèn vào theo cùng `myfunc.hpp`). Do vậy, khi dịch chương trình C++ sẽ báo lỗi (do hàm `int max(int a, int b)` được khai báo hai lần). Để khắc phục tình trạng này, trong `mylib.hpp` ta thêm chỉ thị mới như sau:

```

// tệp mylib.hpp
#ifndef MYLIB__      // nếu chưa định nghĩa tên gọi MYLIB__
#define MYLIB__      // thì định nghĩa nó
int max(int a, int b)    // và các hàm khác
{
    return (a>b? a: b);
}
#endif

```

Như vậy, khi chương trình dịch xử lý đoạn mã `mylib.hpp` (được chèn vào trong `main`) lần đầu do `MYLIB__` chưa định nghĩa nên máy sẽ định nghĩa từ này, và dịch đoạn chương trình tiếp theo cho đến `#endif`. Lần thứ hai, khi gặp lại đoạn lệnh này do `MYLIB__` đã được định nghĩa nên chương trình bỏ qua đoạn lệnh này không dịch.

Để cẩn thận trong cả `myfunc.hpp` (và tất cả các tệp thư viện khác), ta đều phải sử dụng cú pháp này, vì có thể trong một chương trình nào đó `myfunc.hpp` lại được chèn vào nhiều lần.

10.2 Lập trình trên nhiều file

10.2.1 Tổ chức chương trình

Một chương trình khi viết có thể tự bản thân nó được chia thành nhiều file (dễ quản lý), mỗi file mang một đặc trưng riêng, phục vụ cho chương trình (chưa kể chương trình còn sử dụng lại chất liệu trong những mô đun đã cài đặt khác, ví dụ mô đun chứa các hàm toán học (`math.h`), mô đun chứa khai báo và cài đặt các hàm vào/ra (`iostream`), các khai báo và hàm xử lý xâu (`string`) Các file hoặc mô đun có sẵn này sẽ được `#include` vào những chương trình cần đến nó.

Một chương trình thường được tổ chức ít nhất thành 3 file:

- Để che giấu chi tiết, các khai báo (hằng, biến, hàm, macro, ...) sẽ được ghi riêng vào các file được gọi là file tiêu đề (header) và thường lấy đuôi file là `*.hpp`;
- Phần cài đặt các hàm được tổ chức trong các file gọi là file cài đặt (implementation) với đuôi `*.cpp`;

- Chương trình chính cùng một vài hàm đặc thù phục vụ riêng được đặt trong file chính (main) với đuôi *.cpp.

Như vậy, các file header sẽ được include vào file main và các file implementation và hiển nhiên file implementation cũng cần được biết đến các khai báo trong file tiêu đề để sử dụng nên head cũng được include vào implementation (từ đó trong main, head sẽ được include 2 lần nhưng chỉ dịch một lần nhờ vào các chỉ thị `#ifndef`, `#define`, ... `#endif` như trên). Chú ý, các tên và đuôi file được đặt như trên là không bắt buộc.

Với cách tổ chức như vậy, việc đọc, hiểu, sửa chữa, bổ sung, làm việc nhóm ... để xây dựng một chương trình lớn sẽ dễ dàng hơn. Ví dụ, cần hiểu chức năng chương trình làm gì (output) ta chỉ cần đọc file có hàm `main()`. Cần hiểu cách thức làm việc của một chức năng nào đó trong chương trình (how) ta sẽ đọc các file implementation. Cần biết chương trình quản lý loại dữ liệu nào và được cung cấp những gì (input) ta sẽ đọc các file **header**.

Dĩ nhiên, chương trình có thể tổ chức thành nhiều file hơn nữa. Ví dụ các lớp Date, Student, List có thể tổ chức thành ba mô đun riêng biệt trong chương trình Quản lý sinh viên. Mỗi lớp sẽ gồm 1 file header và 1 file implementation, và vì vậy chương trình này sẽ được tổ chức thành 7 file (kể cả file chương trình chính).

10.2.2 Viết và kiểm tra các file include

- Các file include chỉ chứa chất liệu cho chương trình sử dụng, vì vậy trong các file này không có hàm `main()`.
- Cần mở đầu file bằng các chỉ thị `#ifndef name`, `#define name` và kết thúc file bởi `#endif`.
- Ban đầu có thể có hàm `main()` với mục đích kiểm tra các hàm đã cài đặt, đặc biệt đối các với file implementation. Sau khi đã kiểm tra tính đúng đắn của toàn bộ các hàm, ta có thể xóa hàm `main` ra khỏi file hoặc biến các hàm `main` này thành unit-test (hoặc đặt chú thích bao lấy hàm `main`).

Như vậy, các file header và các mô đun độc lập khác sẽ có thêm các chỉ thị tiền xử lý và không có hàm `main()`. Sau khi viết xong, ta cũng có thể dịch các file này để kiểm tra lỗi (về mặt văn phạm).

Chú ý: Về mặt thực hành, sau khi sửa xong file `#include` cần ghi (lên đĩa) trước khi chạy chương trình chính, vì mỗi lần dịch và chạy file chính, file include sẽ được gọi lại từ đĩa để chèn vào file chính.

10.2.3 Biên dịch chương trình có nhiều file

Khi chương trình bao gồm nhiều file header, implementation và main, công việc biên dịch gồm hai bước.

1. **Dịch** các file mã nguồn (*.cpp) ra mã đối tượng (*.obj). Đây là các file mã máy nhưng các chỉ lệnh đặt ở dạng tương đối (có thể di chuyển được).
2. **Liên kết** các file mã đối tượng (*.obj) thành file chạy (*.exe). Đây là file mã máy với các chỉ lệnh ở dạng tuyệt đối (có thể chạy được ngay).

Với các môi trường phát triển (IDE) hiện đại, các bước này thường được gói chung lại thành một chức năng Biên dịch (Build). Trong các môi trường phát triển này, người sử dụng chỉ cần chọn chức năng biên dịch trong menu hoặc ấn một phím tắt để biên dịch chương trình gồm nhiều file. Để sử dụng chức năng này, người sử dụng phải đưa toàn bộ các file mã nguồn vào một dự án (Project) để IDE biên dịch và liên kết tất cả các file trong dự án.

Bài tập

1. Để định nghĩa hằng ta có 2 cách viết:

```
const int MAX = 100, và
#define MAX 100
```

câu nào sau đây sai:

- (a) Kết quả của chương trình khi chạy là như nhau;
 - (b) Với cách 1 khi chạy chương trình MAX được thay thế bằng 100;
 - (c) Với cách 2 khi dịch chương trình MAX được thay thế bằng 100;
 - (d) Câu lệnh `MAX = MAX + 1;` không được phép đối với cách 1 nhưng được phép đối với cách 2.
2. Một sinh viên viết hàm tính tổng 2 số nguyên như sau:

```
Function integer sum(integer m, integer n)
Begin
    integer result;
    result = m + n;
    return result;
End
```

để hàm trên chạy được trong môi trường C/C++, bạn cần thêm các macro nào ?

3. Hãy chỉ ra điểm sai của hàm macro sau đây:

```
#define INCREASE(I) I++
```

4. Cho macro: `#define TIMES(A, B) A * B`
 Với `a = 2`, `b = 5`; hãy tính `TIMES(a, b + 1)` ?
5. Điều chỉnh lại macro `TIMES` trong bài trên để nó tính chính xác tích của 2 đối.
6. Viết macro trao đổi nội dung 2 biến (thay cho hàm `swap`). Áp dụng: Sắp xếp dãy số.
7. Xét chương trình:

```
const int M = 1;
int main()
{
    #if M <= 1
    cout << i ;
    #endif
    system("PAUSE");
    return 1;
}
```

điều gì sẽ xảy ra nếu `M = 1` và khi `M = 2` ?

8. Xét chương trình:

```
const int M = 1;
int main()
{
    int i = 5;
    #if M <= 1
        cout << i + i ;
    #else
        cout << i * i ;
    #endif
    system("PAUSE");
    return 1;
}
```

Và chương trình như trên nhưng bỏ đi tất cả các dấu #. Sự khác biệt giữa 2 chương trình đích (sau khi dịch) là gì ? có sự khác biệt về kết quả chạy của 2 chương trình trên hay không ?

9. Tổ chức lại chương trình QLSV (chương 6, 7, 8) thành nhiều file ví dụ:

- Date.hpp // Khai báo và cài đặt lớp **Date**
- Student.hpp // Khai báo và cài đặt lớp **Student**
- Node.hpp // Khai báo và cài đặt lớp **Node**
- Linked List.hpp // Khai báo và cài đặt lớp danh sách liên kết
- Và chương trình chính trong file: QLSV.cpp.

Chương 11

Lập trình với thư viện chuẩn STL

11.1 Giới thiệu thư viện chuẩn STL

Trong Khoa học máy tính, có rất nhiều cấu trúc dữ liệu hay được sử dụng khi viết chương trình. Các cấu trúc này được chuẩn hóa và cài đặt trên hầu hết các ngôn ngữ lập trình. Trong C++, **thư viện mẫu chuẩn** (*Standard Template Library - STL*) cài đặt các kiểu dữ liệu này. Trong STL, người ta đã cài đặt **ngăn xếp** (*stack*), **hàng đợi** (*queue*) và rất nhiều cấu trúc dữ liệu chuẩn khác. Người ta gọi các cấu trúc dữ liệu này là các **lớp chứa** (*containers*) bởi chúng được dùng để lưu trữ một tập hợp dữ liệu.

Trong chương này, chúng ta sẽ xem xét các lớp chứa cơ bản của STL. Chúng ta sẽ không đi vào các chi tiết sâu của STL bởi sự khổng lồ của nó. Thay vào đó, chúng ta sẽ tìm hiểu các cách sử dụng các lớp chứa phổ biến nhất của STL.

STL được phát triển bởi Alexander Stepanov và Meng Lee tại công ty Hewlett-Packard dựa trên nghiên cứu của Stepanov, Lee, và David Musser. STL bao gồm một loạt các thư viện viết bằng ngôn ngữ C++. Mặc dù STL không phải là phần lõi của C++ nhưng STL là một phần của các chuẩn chuẩn C++98/C++11 (chuẩn ANSI) nên mọi trình biên dịch C++ tuân thủ các chuẩn này đều phải cài đặt STL. Do đó với người lập trình bình thường, có thể coi STL là một phần của ngôn ngữ C++.

Một lớp chứa trong STL cần một **tham số kiểu** chỉ ra kiểu dữ liệu mà lớp chứa này chứa đựng. Các lớp chứa trong STL đều sử dụng khái niệm **con trỏ duyệt** (*iterator*), là lớp đối tượng cho phép người sử dụng duyệt qua các phần tử nằm trong đối tượng chứa. Chúng ta sẽ xem xét khái niệm con trỏ duyệt trong mục [11.2](#)

STL còn cài đặt rất nhiều **thuật toán tổng quát** (*generic algorithms*), chẳng hạn như tìm kiếm hoặc sắp xếp các phần tử trong đối tượng chứa. Chúng ta sẽ tìm hiểu một số thuật toán tổng quát trong mục [11.4](#).

11.2 Khái niệm con trỏ duyệt

Con trỏ duyệt (*iterator*) là khái niệm tổng quát hóa từ con trỏ (chương 7). Trong mục này, ta sẽ xem xét cách dùng con trỏ duyệt trong lớp mẫu **vector** của thư viện STL. Lớp mẫu **vector** thay thế cho khái niệm mảng của ngôn ngữ C. Khái niệm con trỏ duyệt còn được dùng trong rất nhiều lớp mẫu khác của thư viện STL. Hầu hết các thao tác đối với con trỏ duyệt của **vector** đều có

thể thực hiện trên con trỏ duyệt của các lớp mẫu khác trong STL. Ngữ nghĩa, cú pháp và cách đặt tên lớp, tên hàm giống nhau trong STL giúp việc ứng dụng các lớp mẫu của STL trở nên dễ dàng hơn.

Khai báo `using`

Trước tiên, do cấu trúc của thư viện STL gồm không gian tên `std` và các lớp lồng nhau (lớp trong lớp), để cho tiện khi viết chương trình, người sử dụng có thể khai báo

```
using my_space::my_function;
```

để sử dụng hàm `my_function` khi khai báo này có hiệu lực. Khi đó, lệnh `my_function(1,2)` hoàn toàn giống với lệnh `my_space::my_function(1,2)`. Như vậy, câu lệnh có thể viết gọn hơn. Khai báo `using` có thể dùng với lớp mẫu nằm trong một không gian tên. Ví dụ:

```
using std::vector;
vector<int>::iterator p;
```

Câu lệnh đầu tiên khai báo sử dụng lớp mẫu `vector`. Lớp này nằm trong không gian tên `std`. Câu lệnh thứ hai khai báo biến `p` là một con trỏ duyệt mà không cần chỉ ra toàn bộ đường dẫn `std::vector<int>::iterator`.

11.2.1 Các thao tác cơ bản với con trỏ duyệt

Con trỏ duyệt là khái niệm tổng quát hóa của con trỏ. Trên thực tế, nhiều khi con trỏ duyệt được cài đặt bằng con trỏ nhưng các chi tiết cài đặt đã được giấu đi. Qua đó, STL thống nhất cách người sử dụng truy xuất các lớp chứa (sẽ bàn trong phần sau). Mỗi lớp chứa đều có một loại con trỏ duyệt riêng nhưng cách dùng chúng tương tự như nhau.

Giống như con trỏ, con trỏ duyệt trỏ đến một phần tử trong lớp chứa. Qua các toán tử (được nạp chồng), ta có thể thao tác với con trỏ duyệt giống như thao tác với con trỏ:

- Dịch chuyển tiến bằng toán tử `++`.
- Dịch chuyển lùi bằng toán tử `--`.
- So sánh con trỏ bằng toán tử `==`, `!=`.
- Lấy giá trị con trỏ đang trỏ tới bằng toán tử `*`.

Không phải con trỏ duyệt nào cũng có đầy đủ các toán tử này, tuy nhiên ở mục này, con trỏ duyệt của lớp mẫu `vector` có tất cả các toán tử trên.

Một lớp mẫu của STL có các hàm khởi tạo con trỏ cũng như các hàm dùng để kiểm tra con trỏ có hợp lệ. Nếu `c` là một đối tượng chứa, `c` có các hàm sau:

- `c.begin()` trả về con trỏ duyệt tới phần tử đầu tiên của đối tượng chứa.
- `c.end()` trả về con trỏ duyệt chỉ ra con trỏ duyệt đã đi quá phần tử cuối cùng của dãy. Các con trỏ duyệt cần được so sánh với `c.end()` để kết thúc các vòng lặp hoặc quá trình duyệt đối tượng chứa.

Với các câu lệnh và toán tử trên, ta có thể viết đoạn mã duyệt đối tượng chứa cơ bản nhất như sau:

```
// p là con trỏ duyệt trên đối tượng chứa c
for (iterator p = c.begin(); p != c.end(); p++)
    my_process(p) // xử lý dữ liệu do p trỏ đến
```

Ví dụ trong Hình 11.1 cho thấy một số cách sử dụng cơ bản của lớp vector.

```
1 // Chương trình minh họa lớp chứa vector
2 #include <iostream>
3 #include <vector> // để sử dụng lớp chứa vector
4
5 int main()
6 {
7     using std::cout;    // đặt các câu lệnh using
8     using std::endl;    // ở đầu mỗi hàm
9     using std::vector;
10
11     vector<int> c;
12
13     for (int i = 0; i < 4; i++)
14         c.push_back(i); // đưa i vào cuối vật chứa c
15
16     cout << "Inside the container: ";
17     for (vector<int>::iterator p = c.begin(); p != c.end(); p++)
18         cout << *p << " ";
19     cout << endl;
20
21     cout << "Setting all entries to zeros ..." << endl;
22     for (vector<int>::iterator p = c.begin(); p != c.end(); p++)
23         *p = 0;
24
25     cout << "Inside the container: ";
26     for (vector<int>::iterator p = c.begin(); p != c.end(); p++)
27         cout << *p << " ";
28     cout << endl;
29
30     return 0;
31 }
```

Hình 11.1: Con trỏ duyệt trong lớp vector

Output chương trình 11.1

```
Inside the container: 0 1 2 3
Setting all entries to zeros ...
Inside the container: 0 0 0 0
```

Lớp chứa vector mô phỏng một mảng các giá trị có cùng kiểu. Lớp vector khác mảng bình thường ở chỗ nó có thể “co giãn” được. Người sử dụng có thể thêm, bớt phần tử hoặc xóa toàn bộ mảng một cách tùy ý. Chương trình trong hình 11.1 minh họa cách sử dụng con trỏ duyệt thao tác trên mảng. Chương trình này bắt đầu bằng câu lệnh

```
vector<int> c;
```

khai báo đối tượng chứa c có kiểu vector<int> là một vector các số nguyên. Đối tượng này có khả năng chứa các giá trị kiểu int dưới dạng một dãy số liên tiếp nhau. Như vậy, int là tham số kiểu

của lớp mẫu `vector` (xem chương 6). Sau vòng lặp đầu tiên

```
for (int i = 0; i < 4; i++)
    c.push_back(i); // day i vao cuoi vat chua c
```

hàm `push_back` lần lượt đẩy các giá trị `i = 0, 1, 2, 3` vào đối tượng `c`.

Với lớp `vector`, có 2 cách để truy xuất phần tử của nó. Cách thứ nhất là sử dụng chỉ số mảng như mảng bình thường. Ví dụ, `c[0]`, `c[1]`, v.v... Cách thứ hai là sử dụng con trỏ duyệt như ví dụ trên. Trong đó, `p` trỏ lần lượt đến trỏ đến các phần tử `c[0]`, `c[1]`, `c[2]`, `c[3]`. Sau khi xử lý mỗi phần tử, con trỏ `p` được dịch chuyển ra phía sau sang phần tử tiếp theo với toán tử `++`. Để bắt đầu duyệt, ta khởi tạo

```
vector<int>::iterator p = c.begin()
```

để `p` được khởi tạo trỏ đến phần tử đầu tiên của mảng. Để kiểm tra xem `p` đã duyệt hết mảng chưa, ta so sánh `p` với `c.end()` bằng toán tử `!=`. Nếu `p` khác `c.end()` nghĩa là `p` còn chưa duyệt hết mảng. Đoạn lệnh thứ hai

```
for (vector<int>::iterator p = c.begin(); p != c.end(); p++)
    cout << *p << " ";
```

duyet từ đầu đến cuối vật chứa `c` để in ra các số `0, 1, 2, 3`, cách nhau bởi dấu cách. Để ý rằng ta sử dụng toán tử `*` để truy xuất giá trị con trỏ duyệt `p` đang trỏ tới. Đoạn lệnh thứ ba gán tất cả các phần tử trong `c` bằng `0`:

```
for (vector<int>::iterator p = c.begin(); p != c.end(); p++)
    *p = 0;
```

cũng bằng toán tử `*`, tuy nhiên `*p` bây giờ là vế trái của biểu thức gán. Tương tự, ta cũng có thể dùng toán tử `->` cho các thành viên của biến con trỏ duyệt.

Từ khóa `auto`. Như ta thấy ở trên, để khai báo con trỏ duyệt, ta cần một lệnh khá dài

```
vector<int>::iterator p = c.begin()
```

và phải nhớ kiểu `vector<int>::iterator` của `p`. Trong chuẩn C++11 mới, người sử dụng có thể thay thế kiểu này bằng từ khóa `auto`. Trình biên dịch sẽ tự động tính toán kiểu thích hợp của `p`. Câu lệnh trên có thể thay thế bằng câu lệnh đơn giản hơn

```
auto p = c.begin()
```

Với câu lệnh này, trình biên dịch tính toán kiểu của vế phải (sau dấu bằng), tức là kiểu trả về của biểu thức `c.begin()` và tự động khai báo hộ người dùng kiểu của `p` là `vector<int>::iterator`.

11.2.2 Các loại con trỏ duyệt

Con trỏ duyệt `vector<int>::iterator p` ở mục trước có toán tử `++` để dịch chuyển tiến tới (sang phần tử kế tiếp). Các con trỏ duyệt có thể dịch chuyển tiến tới được gọi là **con trỏ duyệt tiến** (*forward iterator*). Ngoài khả năng dịch chuyển tiến tới, con trỏ duyệt `vector<int>::iterator` còn có thể dịch chuyển lùi lại bằng toán tử `--` sang phần tử phía trước. Các con trỏ có thể dịch chuyển theo cả hai hướng gọi là **con trỏ duyệt song hướng** (*bi-directional iterator*). Một loại con trỏ khác cho phép người dùng tiến tới hoặc lùi lại với số bước tùy ý gọi là **con trỏ duyệt ngẫu nhiên** (*random access iterator*). Con trỏ duyệt ngẫu nhiên dùng toán tử `+` và `-` với số nguyên là số bước dịch chuyển. Chương trình trong hình 11.2 minh họa cách dùng con trỏ duyệt ngẫu nhiên.

```

1 // Chương trình minh họa con trỏ duyệt ngẫu nhiên
2 #include <iostream>
3 #include <vector> // để sử dụng lớp chứa vector
4 #include <string> // lớp chuỗi kí tự
5
6 int main()
7 {
8     using std::cout;    // đặt các câu lệnh using
9     using std::endl;    // ở đầu mỗi hàm
10    using std::vector;   // để các hàm đọc lặp
11    using std::string;   // với nhau
12
13    vector<string> c;
14
15    c.push_back("Hello,");
16    c.push_back("my");
17    c.push_back("advanced");
18    c.push_back("programmer");
19
20    cout << "The container: ";
21    for (auto p = c.begin(); p != c.end(); p++)
22        cout << *p << " ";
23    cout << endl;
24
25    auto p = c.begin();
26    cout << "The third element is: " << *(p+2) << endl;
27
28    cout << "The reversed container: ";
29    for (vector<string>::reverse_iterator p = c.rbegin();
30         p != c.rend(); p++)
31        cout << *p << " ";
32    cout << endl;
33
34    return 0;
35 }

```

Hình 11.2: Con trỏ duyệt ngẫu nhiên và con trỏ duyệt ngược

Output chương trình 11.2

```

The container: Hello, my advanced programmer
The third element is: advanced
The reversed container: programmer advanced my Hello,

```

Câu lệnh

```
cout << "The third element is: " << *(p+2) << endl;
```

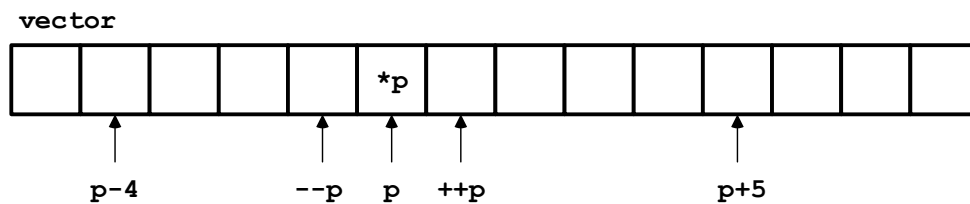
in ra phần tử nằm sau 2 bước nhảy so với phần tử hiện tại `p` đang trỏ đến.

Ngoài ra, chương trình này còn minh họa một loại con trỏ duyệt khác, **con trỏ duyệt ngược** (*reverse iterator*). Con trỏ duyệt này được khởi tạo bằng lệnh `c.rbegin()` trỏ tới phần tử cuối cùng trong `vector`. Toán tử `++` thực chất dịch chuyển loại con trỏ này về phía trước (phần tử ngay trước trong dãy). Để kết thúc duyệt, ta so sánh con trỏ duyệt với `c.rend()`. Chữ `r` ở đây là viết tắt của

từ `reverse`. Ở đây, ta thấy một triết lý xuyên suốt khi xây dựng thư viện chuẩn STL, đó là, *cách sử dụng tên toán tử, tên hàm rất giống nhau cho mọi tác vụ, mọi kiểu dữ liệu*.

Tổng kết lại, ta có các loại con trỏ duyệt trong STL như sau:

- **Con trỏ duyệt tiến tới:** có toán tử `++` để tiến tới;
- **Con trỏ duyệt song hướng:** có toán tử `++` để tiến tới và toán tử `--` để lui lại;
- **Con trỏ duyệt ngẫu nhiên:** có toán tử `++`, toán tử `--`, toán tử `+` và toán tử `-` với số nguyên là số bước nhảy;



Hình 11.3: Con trỏ duyệt.

Con trỏ duyệt hằng. Trong các ví dụ trên, con trỏ duyệt `p` có thể dùng để thay đổi giá trị nó trỏ tới. Với một số lớp chứa, ta không thể thay đổi giá trị do con trỏ duyệt của nó trỏ tới. Khi đó, ta phải dùng khái niệm con trỏ duyệt hằng (*const iterator*). Khai báo trong C++ như sau

```
vector<string>::const_iterator p = c.begin();
```

Khi đó, ta có thể đọc giá trị do `p` trỏ đến, ví dụ in nó ra màn hình:

```
cout << *p << endl;
```

Tuy nhiên, lệnh gán `*p = "Hello"`; lúc này sẽ bị chương trình dịch báo lỗi. Ngoài trường hợp bắt buộc phải sử dụng con trỏ duyệt hằng, nhiều lập trình viên sử dụng con trỏ duyệt hằng để hạn chế một đoạn mã không được phép sửa giá trị của các phần tử trong đối tượng chứa. Ví dụ, đoạn mã

```
for (vector<string>::const_iterator p = c.begin(); p != c.end(); p++)
    your_function(p);
```

gọi hàm `your_function` với từng phần tử của đối tượng chứa `c`. Tuy nhiên, `your_function` chắc chắn không có quyền thay đổi giá trị của phần tử do `p` trỏ đến vì `p` là con trỏ duyệt hằng. Việc này hạn chế các lỗi vô ý khi lập trình đồng thời cũng giúp lập trình viên bày tỏ rõ ý đồ thiết kế các hàm trong chương trình: Hàm nào chỉ truy xuất lấy giá trị, hàm nào có quyền sửa giá trị.

11.3 Khái niệm vật chứa

Khái niệm **vật chứa** (hay lớp chứa - *containers*) đại diện cho một loạt cấu trúc dữ liệu dùng để chứa đựng dữ liệu khác. Trong mục này, chúng tôi sẽ giới thiệu các cấu trúc dữ liệu **danh sách** (*list*), **hàng đợi** (*queue*), **ngăn xếp** (*stack*), **tập hợp** (*set*) và **ánh xạ** (*map*). Đây là các cấu trúc dữ liệu làm nền tảng cho các thuật toán của Khoa học máy tính. Chúng quan trọng đến nỗi thư

viện STL được chuẩn hóa (ANSI/ISO) để bao gồm các kiểu dữ liệu này. Kết quả là lập trình viên C++ nào cũng có thể sử dụng các cấu trúc dữ liệu này trong chương trình của mình.

Với các lớp chứa, người sử dụng có thể định nghĩa loại dữ liệu mà vật chứa chứa đựng. Ví dụ, bạn có thể khai báo một **vector** các **int** hoặc **string** như mục trước. Hoặc bạn có thể khai báo một danh sách các ký tự **list<char>**. Khi khai báo, kiểu dữ liệu cần chứa là tham số kiểu của lớp chứa.

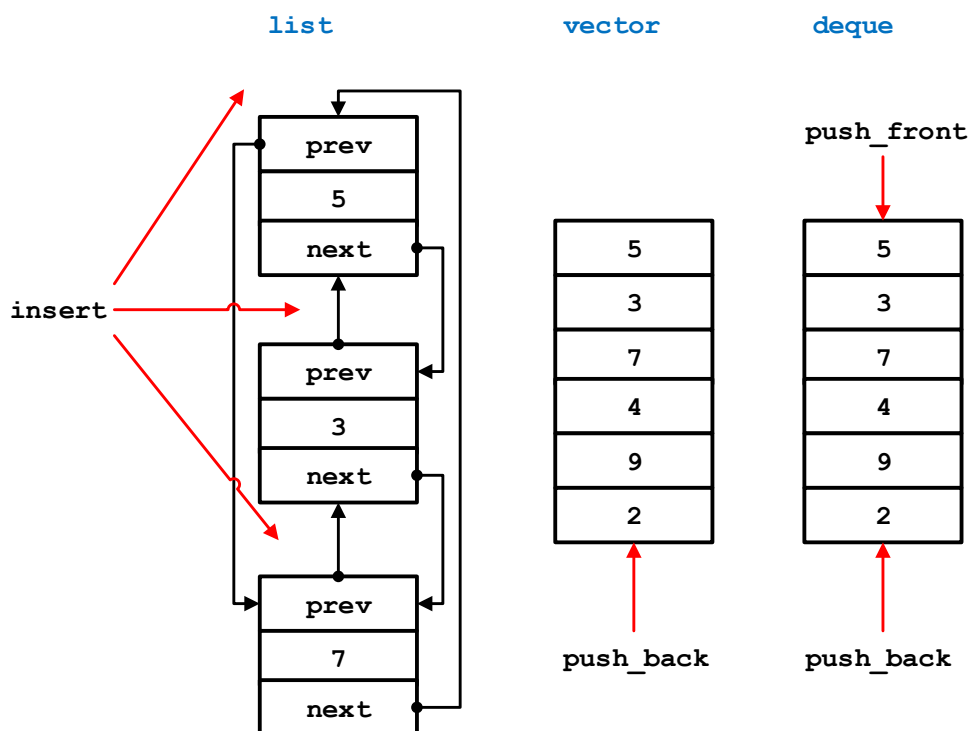
Tất cả các lớp chứa trong STL đều có con trỏ duyệt của nó. Khai báo con trỏ duyệt như sau

```
<tên lớp chứa>::iterator p = c.begin();
```

trong đó **c** là vật chứa có kiểu tương ứng. Các lớp chứa trong STL đều có hai hàm **begin()** và **end()** để người sử dụng duyệt qua vật chứa giống như cách ta duyệt qua **vector** ở mục trước.

11.3.1 Các vật chứa dạng dãy

Để mô tả một dãy dữ liệu, trong C++, ta có 3 loại cấu trúc dữ liệu **list** (danh sách), **vector** (véc-tơ) và **deque** (hàng đợi 2 đầu). Hình 11.4 mô tả các cấu trúc dữ liệu này. Chúng tôi sẽ không đi vào chi tiết cài đặt các cấu trúc dữ liệu mà chỉ nêu các thao tác khi làm việc với chúng.



Hình 11.4: Các lớp chứa dạng dãy.

Bảng 11.5 mô tả các thao tác và tốc độ của chúng trên các cấu trúc dữ liệu **list**, **vector**, **deque**.

Cấu trúc **list** cho phép chèn, xóa các phần tử ở vị trí bất kỳ rất nhanh. Tuy nhiên, **list** không có con trỏ duyệt ngẫu nhiên mà ta phải duyệt từ đầu đến cuối **list** để truy xuất một phần tử trong **list** (Bảng 11.6). Cấu trúc **vector** chỉ cho phép chèn và xóa nhanh ở cuối dãy. Còn việc chèn hoặc xóa phần tử ở giữa dãy chậm hơn nhiều. Đối lại, với **vector** ta có con trỏ duyệt ngẫu

Bảng 11.5: Các thao tác với list, vector và deque.

Kiểu	Thao tác	Lệnh	Tốc độ
list	Chèn	push_front, push_back, insert	Nhanh
	Xóa	pop_front, pop_back, erase	Nhanh
vector	Chèn vào cuối	push_back	Nhanh
	Xóa ở cuối	pop_back	Nhanh
	Chèn vào giữa	insert	Chậm
	Xóa ở giữa	erase	Chậm
deque	Chèn vào đầu	push_front	Nhanh
	Xóa ở đầu	pop_front	Nhanh
	Chèn vào cuối	push_back	Nhanh
	Xóa ở cuối	pop_back	Nhanh
	Chèn vào giữa	insert	Chậm
	Xóa ở giữa	erase	Chậm

Bảng 11.6: Con trỏ duyệt của list, vector và deque.

Kiểu	Con trỏ duyệt	Kiểu con trỏ	#include
list	list<T>::iterator, list<T>::const_iterator, list<T>::reverse_iterator, list<T>::const_reverse_iterator	song hướng	<list>
	vector<T>::iterator, vector<T>::const_iterator, vector<T>::reverse_iterator, vector<T>::const_reverse_iterator		
vector	deque<T>::iterator, deque<T>::const_iterator, deque<T>::reverse_iterator, deque<T>::const_reverse_iterator	ngẫu nhiên	<vector>
deque		ngẫu nhiên	<deque>

nhien. Do đó, việc truy xuất phần tử bất kì của dãy rất dễ dàng. Cấu trúc deque cho phép chèn và xóa nhanh ở cả hai đầu của dãy.

Các dãy kiểu list, vector và deque có một số cách khởi tạo nêu trong bảng 11.7. Ngoài ra, các kiểu này đều các hàm khởi tạo sao chép mặc định.

Ngoài các hàm thay đổi dãy như đã nêu, các vật chứa dạng dãy có thêm một số hàm khác như mô tả ở bảng 11.8. Để minh họa, trong mục này ta sẽ xây dựng một chương trình nhỏ đếm số lần xuất hiện của mỗi từ (một chuỗi liên tục ký tự chữ cái hoặc chữ số) trong file. Chương trình lấy tên file từ dòng lệnh. Để đếm số từ, ta dùng 2 vector. Một vector<string> chứa các từ, mỗi từ là một phần tử duy nhất trong vector. Một vector<int> lưu số đếm của từ tương ứng. Chi tiết xem ở chương trình trong hình 11.9. Để chạy chương trình, dùng lệnh

```
word_count word_count.cpp
```

```
1 // Dem tu bang vector
```


Bảng 11.7: Khởi tạo list, vector và deque.

Constructor	Ví dụ
()	Khởi tạo dãy rỗng. <code>list<int> a;</code> <code>vector<string> b;</code>
(n, val)	Khởi tạo dãy có n phần tử với giá trị val. <code>deque<int> a(10, 0); // 10 số 0</code> <code>vector<string> b(100, "Hello"); // 100 xâu Hello</code>
(p1, p2)	Khởi tạo dãy có các phần tử lấy từ khoảng [p1, p2) chỉ định bởi 2 con trỏ duyệt p1 và p2. <code>deque<int> a(10, 0); // 10 số 0</code> <code>vector<int> b(a.begin(), a.begin()+5); // Lấy 5 số đầu mảng a</code>

Bảng 11.8: Một số hàm của list, vector và deque.

Hàm thành viên	Ý nghĩa
<code>c.size()</code>	Trả về số phần tử của dãy
<code>c.begin()</code>	Trả về con trỏ duyệt tới phần tử đầu tiên
<code>c.end()</code>	Trả về con trỏ duyệt để kiểm tra đã hết dãy
<code>c.rbegin()</code>	Trả về con trỏ duyệt tới phần tử cuối cùng
<code>c.rend()</code>	Trả về con trỏ duyệt để kiểm tra (chiều nghịch)
<code>c.push_back(ele)</code>	Thêm phần tử <code>ele</code> vào cuối dãy
<code>c.push_front(ele)</code>	Thêm phần tử <code>ele</code> vào đầu dãy
<code>c.insert(p, ele)</code>	Thêm phần tử <code>ele</code> vào trước phần tử do <code>p</code> trỏ đến
<code>c.erase(p)</code>	Xóa phần tử do <code>p</code> trỏ đến
<code>c.clear()</code>	Xóa toàn bộ dãy (dãy trống)
<code>c.front()</code>	Phần tử đầu dãy, tương đương với <code>*(c.begin())</code>
<code>c.back()</code>	Phần tử cuối dãy, tương đương với <code>*(c.rbegin())</code>
<code>c1 == c2</code>	Trả về <code>true</code> nếu <code>c1.size() == c2.size()</code> và mỗi phần tử trong <code>c1</code> bằng với phần tử tương ứng trong <code>c2</code>
<code>c1 != c2</code>	Tương đương với <code>!(c1 == c2)</code>

```

2 #include <iostream>
3 #include <fstream>
4 #include <vector>
5
6 using namespace std;
7
8 void CountWords(istream& input, vector<string>& words, vector<int>& counts, bool
    shouldClear = true);
9
10 int main(int argc, char **argv)
11 {

```

```

12     if (argc < 2) {
13         cout << "Usage: word_count <filename>" << endl;
14         return 0;
15     }
16
17     ifstream input(argv[1]);
18     vector<string> words;
19     vector<int> counts;
20     CountWords(input, words, counts);
21
22     vector<int>::iterator pCount = counts.begin();
23     for (vector<string>::iterator pWord = words.begin(); pWord != words.end(); pWord
24         ++, pCount++)
25         cout << "(" << *pWord << "," << *pCount << ")";
26     return 0;
27 }
28
29 void CountWords(istream& input, vector<string>& words, vector<int>& counts, bool
30     shouldClear)
31 {
32     if (shouldClear) {
33         words.clear();
34         counts.clear();
35     }
36
37     string word;
38     input >> word;
39     while (input) {
40         bool found = false;
41         // tìm tu trong danh sách
42         vector<int>::iterator pCount = counts.begin();
43         for (vector<string>::iterator pWord = words.begin(); pWord != words.end();
44             pWord++, pCount++) {
45             if (word == *pWord) { // tìm thấy tu
46                 (*pCount)++;
47                 found = true;
48                 break;
49             }
50         }
51         if (!found) { // không tìm thấy, thêm vào danh sách
52             words.push_back(word);
53             counts.push_back(1);
54         }
55         input >> word;
56     }
57 }

```

Hình 11.9: Đếm từ bằng vector.

Output chương trình 11.9 trên chính nó

```

(//,4)(Dem,1)(tu,3)(bang,1)(vector,1)(#include,3)(<iostream>,1)(<
fstream>,1)(<vector>,1)(using,1)(namespace,1)(std;,1)(void,2)(
countWords(istream&,2)(input,,2)(vector<string>&,2)(words,,3)(vector<

```

```
int>&,2)(counts,,2)(bool,3)(shouldClear,1)(=,7)(true);,1)(int,1)(main(
int,1)(argc,,1)(char,1)(**argv),1)({,8)(if,4)((argc,1)(<,1)(2),1)(cout
,2)(<<,7)("Usage: ,1)(word_count,1)(<filename>",1)(endl;,1)(return,2)
(0;,2)({,8)(ifstream,1)(input(argv[1]);,1)(vector<string>,1)(words;,1)(
vector<int>,1)(counts;,1)(countWords(input,,1)(counts);,1)(vector<int
>::iterator,2)(pCount,2)(counts.begin();,2)(for,2)((vector<string>::
iterator,2)(pWord,4)(words.begin();,2)(!=,2)(words.end();,2)(pWord
++,2)(pCount++;,2)("(",1)(*pWord,1)(", ",1)(*pCount,1)(")";,1)(
shouldClear,1)((shouldClear,1)(words.clear();,1)(counts.clear();,1)(
string,1)(word;,3)(input,2)(>>,2)(while,1)((input,1)(found,2)(false
;,1)(tim,3)(trong,1)(danh,2)(sach,2)((word,1)(==,1)(*pWord),1)(thay,1)
((*pCount)++;,1)(true;,1)(break;,1)((!found,1)(khong,1)(thay,,1)(them
,1)(vao,1)(words.push_back(word);,1)(counts.push_back(1);,1)
```

Như đã thấy, chương trình đếm tất cả từ với định nghĩa từ là các đoạn kí tự liên nhau phân cách bởi khoảng trống (ký tự trống, ký tự tab, dấu xuống dòng). Để định nghĩa lại cách phân cách các từ bằng các ký tự khác (ví dụ, dấu đóng mở ngoặc, dấu chấm, dấu phẩy, ...) ta dùng gói locale và gói algorithm của C++.

```
#include <locale>
#include <algorithm>
```

Sau đó ta định nghĩa lớp my_ctype. Lớp này “đánh dấu” các ký tự mà ta coi là khoảng trống, tức là tất cả các ký tự không phải số và chữ cái trong bảng chữ cái Latin.

```
1 class my_ctype : public std::ctype<char>
2 {
3     mask my_table[table_size];
4 public:
5     my_ctype(size_t refs = 0)
6         : std::ctype<char>(&my_table[0], false, refs)
7     {
8         std::copy_n(classic_table(), table_size, my_table);
9         for (int c = 0; c < table_size; c++)
10             if (!isdigit(c) && !isalpha(c))
11                 my_table[c] = (mask)space;
12     }
13 };
```

Hình 11.10: Định nghĩa cách “đánh dấu” các ký tự khoảng trống.

Để sử dụng lớp này, ta thêm 2 câu lệnh vào sau khai báo biến input trong hàm main()

```
ifstream input(argv[1]);
std::locale newSpace(std::locale::classic(), new my_ctype);
input.imbue(newSpace);
```

Output chương trình 11.9 sau khi định nghĩa lại khoảng trống

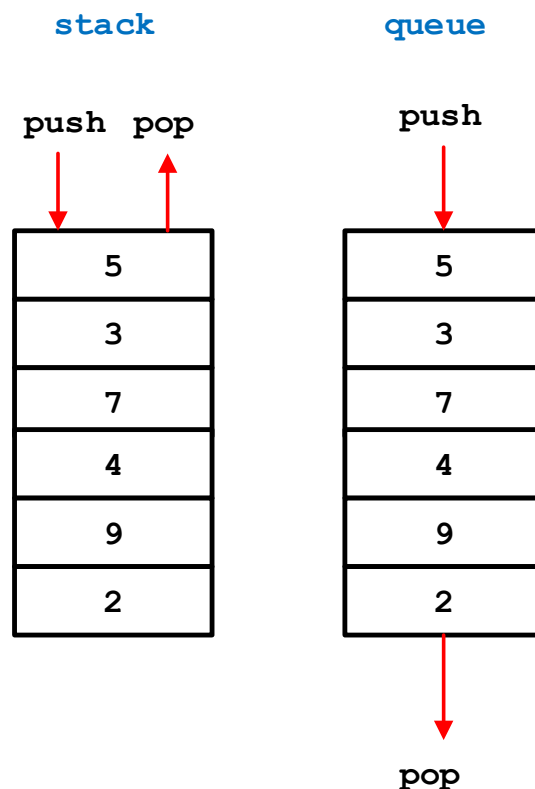
```
(Dem,1)(tu,3)(bang,1)(vector,12)(include,5)(iostream,1)(fstream,1)(
locale,3)(algorithm,1)(using,1)(namespace,1)(std,6)(class,1)(my,7)(
ctype,5)(public,2)(char,3)(mask,2)(table,8)(size,4)(t,1)(refs,2)(0,5)(
false,2)(copy,1)(n,1)(classic,2)(for,3)(int,8)(c,6)(if,5)(isdigit,1)(
```

```
isalpha,1)(space,1)(void,2)(countWords,3)(istream,2)(input,8)(string,6)
(words,10)(counts,8)(bool,3)(shouldClear,3)(true,2)(main,1)(argc,2)(
argv,2)(2,1)(cout,2)(Usage,1)(word,6)(count,1)(filename,1)(endl,1)(
return,2)(ifstream,1)(1,2)(newSpace,2)(new,1)(imbue,1)(iterator,4)(
pCount,6)(begin,4)(pWord,8)(end,2)(clear,2)(while,1)(found,3)(tim,3)(
trong,1)(danh,2)(sach,2)(thay,2)(break,1)(khong,1)(them,1)(vao,1)(push
,2)(back,2)
```

Bạn đọc có thể cải tiến thêm chương trình bằng cách đếm từ không phân biệt chữ hoa và chữ thường. Bạn đọc cũng có thể in ra các từ theo thứ tự từ điển hoặc thứ tự số đếm từ cao đến thấp.

11.3.2 Ngăn xếp và hàng đợi

Ngăn xếp (*stack*), hàng đợi (*queue*) và hàng đợi ưu tiên (*priority_queue*) là các cấu trúc dữ liệu vật chứa được xây dựng dựa trên các cấu trúc dữ liệu dạng dãy đã nêu ở mục trước. Vì thế ngăn xếp và hàng đợi được gọi là các **vật chứa chuyển tiếp** (*container adapter*). Người sử dụng có thể định nghĩa loại cấu trúc dữ liệu dạng dãy dùng để xây dựng ngăn xếp và hàng đợi. Tuy nhiên ở mục này, ta chỉ sử dụng các cấu trúc dữ liệu dạng dãy mặc định của chúng.



Hình 11.11: Ngăn xếp và hàng đợi.

Ngăn xếp và các hàng đợi định nghĩa rõ thứ tự thêm và lấy phần tử ra khỏi vật chứa. Hình 11.11 minh họa các thứ tự truy xuất này. Ngăn xếp *stack* là cấu trúc dữ liệu Vào Sau, Ra Trước (Last In First Out - LIFO). Nghĩa là việc chèn phần tử và lấy (xóa) phần tử đều thực hiện ở cuối dãy. Ngược lại, hàng đợi *queue* là cấu trúc dữ liệu Vào Trước, Ra Trước (First In First Out - FIFO).

Nghĩa là việc chèn thêm phần tử thực hiện ở cuối dãy còn việc lấy (xóa) phần tử sẽ thực hiện ở đầu dãy. Hàng đợi ưu tiên `priority_queue` cho phép lấy ra các phần tử lần lượt theo thứ tự ưu tiên từ lớn đến nhỏ. Nếu hai phần tử có độ ưu tiên như nhau thì phần tử chèn vào sớm hơn sẽ được lấy ra trước giống như hàng đợi bình thường.

Để sử dụng các lớp chứa này, ta cần thêm chỉ thị `#include<stack>`, `#include<queue>` vào đầu file mã nguồn. Các hàm thành viên của ngăn xếp và hàng đợi bao gồm:

- `push()`: Thêm phần tử mới;
- `pop()`: Lấy phần tử ra (theo thứ tự mô tả ở trên);
- `top()`: Lấy tham chiếu đến giá trị của phần tử đầu tiên (mà lệnh `pop()` lấy ra). Lớp `queue` không có hàm này mà thay đó vào bằng lệnh `front()` để lấy phần tử ở đầu `queue` và lệnh `back()` để lấy phần tử ở cuối `queue`;
- `size()`: Lấy số phần tử của vật chứa;
- `empty()`: Kiểm tra vật chứa có rỗng.

Bảng 11.12 mô tả một số ví dụ sử dụng các cấu trúc dữ liệu này. Chương trình trong hình 11.13 minh họa cách sử dụng `stack` để kiểm tra xem một chuỗi có các cặp dấu ngoặc `[]`, `()`, được viết hợp lệ. Ý tưởng của chương trình như sau: Khi gặp dấu mở ngoặc, ta đưa vào `stack<char>`, còn khi gặp dấu đóng ngoặc, kiểm tra xem đỉnh của `stack` có dấu mở ngoặc tương ứng hay không và gọi lệnh `pop()` để xóa dấu mở ngoặc đi. Khi đọc hết chuỗi, ngăn xếp phải rỗng nếu dấu ngoặc viết hợp lệ.

```

1 // Kiểm tra dấu ngoặc hợp lệ
2 #include <iostream>
3 #include <sstream>
4 #include <stack>
5 #include <algorithm>
6
7 using namespace std;
8
9 bool CheckParenTheses(istream& input);
10
11 int main()
12 {
13     string line;
14     cout << "Enter a string:";
15     getline(cin, line);
16
17     stringstream input(line);
18     cout << (CheckParenTheses(input) ? "Good" : "Bad")
19         << " parentheses" << endl;
20     return 0;
21 }
22
23 bool CheckParenTheses(istream& input)
24 {
25     stack<char> s;
26     char c;
```

Bảng 11.12: Các thao tác với stack, queue và priority_queue.

Lớp chứa	Ví dụ
stack	<pre>using std::stack; stack<int> c; c.push(1); c.push(2); c.push(3); cout << c.size() << endl; // 3 cout << c.top() << endl; c.pop(); // 3 cout << c.top() << endl; c.pop(); // 2 cout << c.top() << endl; c.pop(); // 1 if (c.empty()) cout << "empty" << endl; // empty</pre>
queue	<pre>using std::queue; queue<double> c; c.push(1.1); c.push(2.2); c.push(3.3); cout << c.size() << endl; // 3 cout << c.front() << endl; c.pop(); // 1.1 cout << c.front() << endl; c.pop(); // 2.2 cout << c.front() << endl; c.pop(); // 3.3 if (c.empty()) cout << "empty" << endl; // empty</pre>
priority_queue	<pre>using std::priority_queue; priority_queue<string> c; c.push("Long"); c.push("Vinh"); c.push("Thai"); cout << c.size() << endl; // 3 cout << c.top() << endl; c.pop(); // Vinh cout << c.top() << endl; c.pop(); // Thai cout << c.top() << endl; c.pop(); // Long if (c.empty()) cout << "empty" << endl; // empty</pre>

```

27
28 c = input.get();
29 while (input) {
30     if (c == '(' || c == '[' || c == '{') s.push(c);
31     else if (c == ')' || c == ']' || c == '}') {
32         if (s.empty()) return false; // không the rong
33         char top = s.top();
34         if (c == ')' && top != '(') return false; // dau ngoac khong dung
35         if (c == ']' && top != '[') return false;
36         if (c == '}' && top != '{') return false;
37         s.pop();
38     }

```

```

39     c = input.get();
40 }
41 return s.empty(); // phải rong
42 }

```

Hình 11.13: Kiểm tra cặp dấu ngoặc hợp lệ.

Output chương trình 11.13

```

Enter a string: (Good [ parentheses ] { bad parentheses } )
Good parentheses

```

11.3.3 Tập hợp và ánh xạ

Tập hợp **set** và ánh xạ **map** là hai cấu trúc dữ liệu vật chứa lưu trữ dữ liệu theo một thứ tự nhất định. Thứ tự này được định nghĩa bởi phép so sánh “nhỏ hơn”. Ngoài phép so sánh nhỏ hơn < mặc định do C++ cung cấp, người sử dụng có thể định nghĩa lại phép toán < cho kiểu dữ liệu của mình.

Tập hợp. Các phần tử trong **set** là duy nhất và có cùng kiểu. Kiểu của phần tử chính là tham số kiểu khi khai báo **set**. Việc chèn thêm một phần tử đã có sẵn trong tập hợp không làm thay đổi tập hợp đó. Nếu ta duyệt tập hợp bằng con trỏ duyệt, ta sẽ được các phần tử sắp sẵn theo thứ tự từ nhỏ đến lớn.

Bảng 11.14: Các thao tác với **set** và **unordered_set**.

Lệnh	Ý nghĩa
<code>insert(e)</code>	Thêm phần tử <code>e</code> vào tập hợp
<code>insert(p,e)</code>	Thêm phần tử <code>e</code> vào tập hợp có sử dụng gợi ý bằng con trỏ duyệt <code>p</code> để giúp quá trình thêm nhanh hơn (nếu <code>p</code> trỏ đến phần tử đứng trước <code>e</code>).
<code>insert(p1, p2)</code>	Thêm tất cả các phần tử nằm trong khoảng <code>[p1, p2)</code> vào tập hợp (không tính phần tử do <code>p2</code> trỏ đến).
<code>erase(p)</code>	Xóa phần tử do <code>p</code> trỏ đến.
<code>erase(e)</code>	Xóa phần tử có giá trị <code>e</code> .
<code>erase(p1, p2)</code>	Xóa tất cả các phần tử nằm trong khoảng <code>[p1, p2)</code> trong tập hợp.
<code>clear()</code>	Xóa tất cả các phần tử trong tập hợp.
<code>find(e)</code>	Trả về con trỏ duyệt đến phần tử có giá trị <code>e</code> . Nếu tập hợp không chứa <code>e</code> , trả về con trỏ duyệt <code>end()</code> .

Ánh xạ. Ánh xạ **map** là cấu trúc dữ liệu quản lý các cặp <khóa, giá trị> (`pair<key, value>`). Khi khai báo **map**, ta cần khai báo 2 tham số kiểu là kiểu của **khóa** và kiểu của **giá trị**. Có thể

hiểu khóa như là “chỉ số” dùng để truy xuất giá trị giống như chỉ số của mảng hoặc vector. Mỗi khóa trong map là duy nhất và cũng được sắp xếp theo một thứ tự giống như set. Vì vậy, nếu ta duyệt các phần tử của ánh xạ bằng con trỏ duyệt, ta sẽ được các phần tử có khóa từ nhỏ đến lớn. Người sử dụng có thể khai báo thứ tự này thông qua việc định nghĩa toán tử < của khóa. Đoạn mã

```
map<string, string> phoneNumber;
```

khai báo một ánh xạ từ kiểu string sang kiểu string dùng để tìm tên người bằng số điện thoại. Khi đó ta có thể thêm một số điện thoại mới vào phoneNumber bằng các lệnh:

```
phoneNumber["84982123456"] = "Long";
phoneNumber["84431234567"] = "Vinh";
phoneNumber["84982123456"] = "Thai"; // thay thế tên người
```

Kiểu phần tử của map là kiểu cặp std::pair<KeyType, ValueType>. Để tạo một phần tử kiểu cặp, C++ cung cấp hàm mẫu std::make_pair(key, value). Với ví dụ về số điện thoại ở trên, ta có thể dùng các câu lệnh tương đương như sau:

```
using std::make_pair;
phoneNumber.insert(make_pair("84982123456", "Long"));
phoneNumber.insert(make_pair("84431234567", "Vinh"));
phoneNumber.insert(make_pair("84982123456", "Thai"));
```

Một biến kiểu pair có hai thành viên first và second dùng để truy xuất đến phần tử thứ nhất và phần tử thứ hai trong cặp.

Để truy xuất giá trị bằng khóa, ta dùng khóa như là chỉ số của mảng. Đoạn mã sau in ra tên người dùng số điện thoại của họ:

```
cout << phoneNumber["84982123456"] << endl; // "Thai";
cout << phoneNumber["NEUNUMBER"] << endl; // "" xâu rỗng
```

Với khóa chưa có trong ánh xạ, việc sử dụng toán tử chỉ số [] tự động thêm một giá trị cho khóa này, khởi tạo bằng hàm constructor mặc định. Ở đây là xâu rỗng.

Chương trình trong hình 11.16 cải tiến chương trình đếm từ 11.9 (sau khi định nghĩa khoảng trống bằng đoạn mã 11.10) bằng cách sử dụng ánh xạ. Chúng tôi sử dụng ánh xạ mỗi từ đến số đếm của nó, tức là, ta sẽ dùng map<string,int>.

```
1 // Dem tu bang map
2 #include <iostream>
3 #include <fstream>
4 #include <map>
5 #include <locale>
6 #include <algorithm>
7
8 using namespace std;
9
10 class my_ctype : public std::ctype<char>
11 {
12     mask my_table[table_size];
13 public:
14     my_ctype(size_t refs = 0)
15         : std::ctype<char>(&my_table[0], false, refs)
16     {
17         std::copy_n(classic_table(), table_size, my_table);
```


Bảng 11.15: Các thao tác với map và unordered_map.

Lệnh	Ý nghĩa
insert(e)	Thêm phần tử e vào ánh xạ, trong đó e là một cặp pair<key, value>.
insert(p,e)	Thêm phần tử e vào ánh xạ có sử dụng gợi ý bằng con trỏ duyệt p để giúp quá trình thêm nhanh hơn (nếu p trỏ đến phần tử đứng trước ele).
insert(p1, p2)	Thêm tất cả các phần tử nằm trong khoảng [p1, p2) vào tập hợp (không tính phần tử do p2 trỏ đến).
erase(p)	Xóa phần tử do p trỏ đến.
erase(k)	Xóa phần tử có giá trị khóa bằng k.
erase(p1, p2)	Xóa tất cả các phần tử nằm trong khoảng [p1, p2) trong ánh xạ.
clear()	Xóa tất cả các phần tử trong tập hợp.
find(k)	Trả về con trỏ duyệt đến phần tử có giá trị khóa bằng k. Nếu ánh xạ không chứa khóa k, trả về con trỏ duyệt end().

```

18     for (int c = 0; c < table_size; c++)
19         if (!isdigit(c) && !isalpha(c))
20             my_table[c] = (mask)space;
21     }
22 };
23
24 void CountWords(istream& input, map<string, int>& wordCounts, bool shouldClear = true
25 );
26
27 int main(int argc, char **argv)
28 {
29     if (argc < 2) {
30         cout << "Usage: word_count <filename>" << endl;
31         return 0;
32     }
33
34     ifstream input(argv[1]);
35     std::locale newSpace(std::locale::classic(), new my_ctype);
36     input.imbue(newSpace);
37
38     map<string, int> wordCounts;
39     CountWords(input, wordCounts);
40
41     for (auto p = wordCounts.begin(); p != wordCounts.end(); p++)
42         cout << "(" << p->first << "," << p->second << ")";
43     return 0;
44 }

```

```

45 void CountWords(istream& input, map<string, int>& wordCounts, bool shouldClear)
46 {
47     if (shouldClear)
48         wordCounts.clear();
49
50     string word;
51     input >> word;
52     while (input) {
53         bool found = false;
54         // tìm tu trong danh sách
55         auto p = wordCounts.find(word);
56         if (p != wordCounts.end())
57             p->second++; // số đếm ở thành viên second
58         else
59             wordCounts[word] = 1; // không tìm thấy
60         input >> word;
61     }
62 }

```

Hình 11.16: Đếm từ bằng ánh xạ `map<string,int>`.

Output chương trình 11.16 trên chính nó

```

(0,5)(1,2)(2,1)(Dem,1)(Usage,1)(algorithm,1)(argc,2)(argv,2)(auto,2)(
bang,1)(begin,1)(bool,3)(c,6)(char,3)(class,1)(classic,2)(clear,1)(copy
,1)(count,1)(countWords,3)(cout,2)(ctype,5)(danh,1)(dem,1)(else,1)(end
,2)(endl,1)(false,2)(filename,1)(find,1)(first,1)(for,2)(found,1)(
fstream,1)(if,4)(ifstream,1)(imbue,1)(include,5)(input,8)(int,6)(
iostream,1)(isalpha,1)(isdigit,1)(istream,2)(khong,1)(locale,3)(main,1)
(map,5)(mask,2)(my,7)(n,1)(namespace,1)(new,1)(newSpace,2)(o,1)(p,8)(
public,2)(refs,2)(return,2)(sach,1)(second,3)(shouldClear,3)(size,4)(so
,1)(space,1)(std,6)(string,4)(t,1)(table,8)(thanh,1)(thay,1)(tim,2)(
trong,1)(true,1)(tu,2)(using,1)(vien,1)(void,2)(while,1)(word,6)(
wordCounts,10)

```

Như bạn đọc thấy, các từ tự động được sắp xếp theo thứ tự từ điển. Để ý rằng, mỗi phần tử của `map<string, int>` là một `pair<string, int>`. Do đó, trong vòng lặp duyệt ánh xạ để in các cặp, ta dùng thành viên `first` để lấy từ và thành viên `second` để lấy số từ:

```

for (auto p = wordCounts.begin(); p != wordCounts.end(); p++)
    cout << "(" << p->first << ", " << p->second << ")";

```

11.3.4 Hàm băm, tập hợp và ánh xạ không thứ tự (C++11)

Với chuẩn C++11, tập hợp không thứ tự `unordered_set` và ánh xạ không thứ tự `unordered_map` là hai cấu trúc dữ liệu có các thao tác chèn và xóa giống hệt `set` và `map` trong mục trước (Bảng 11.14 và Bảng 11.15). Tuy nhiên, các phần tử trong `unordered_set` và `unordered_map` không được sắp xếp thứ tự bằng việc so sánh giá trị hoặc khóa. Thay vào đó `unordered_set` và `unordered_map` sử dụng khái niệm **hàm băm** (*hash function*) tính toán vị trí của các phần tử trong vật chứa. Qua hàm băm, các thao tác chèn, xóa và sửa trên `unordered_set` và `unordered_map`

nhanh hơn set và map rất nhiều. Thông qua một số ví dụ, người dùng sẽ hiểu rõ hơn cách dùng và hiệu quả của các cấu trúc dữ liệu này.

```

1 // Chương trình minh họa tốc độ chèn của ánh xạ không thu tu (C++11)
2 #include <iostream>
3 #include <vector>
4 #include <map>
5 #include <unordered_map>
6 #include <cstdlib>
7 #include <chrono>
8
9 using namespace std;
10
11 int main()
12 {
13     using chrono::high_resolution_clock; // đồng hồ
14     using chrono::duration_cast;
15     using chrono::milliseconds;
16
17     const int N = 1000000, K = 10;
18     vector<string> keys(N, string(K, ' '));
19     map<string, string> m;
20     unordered_map<string, string> um;
21
22     srand(0);
23     for (int i = 0; i < N; i++) // khởi tạo N phần tử ngẫu nhiên
24         for (int j = 0; j < K; j++)
25             keys[i][j] = 'a' + (rand() % 26);
26
27     auto mapTime1 = high_resolution_clock::now();
28     for (int i = 0; i < N; i++) m[keys[i]] = keys[i];
29     auto mapTime2 = high_resolution_clock::now();
30     cout << "Map insertion time is "
31          << duration_cast<milliseconds>(mapTime2-mapTime1).count()
32          << " milliseconds" << endl;
33
34     auto umapTime1 = high_resolution_clock::now();
35     for (int i = 0; i < N; i++) um[keys[i]] = keys[i];
36     auto umapTime2 = high_resolution_clock::now();
37     cout << "Unordered map insertion time is "
38          << duration_cast<milliseconds>(umapTime2-umapTime1).count()
39          << " milliseconds" << endl;
40
41     return 0;
42 }

```

Hình 11.17: So sánh tốc độ chèn của map và unordered_map trên thao tác thêm 1.000.000 phần tử. Lưu ý, cần báo với chương trình dịch sử dụng chuẩn C++11 khi dịch chương trình.

Output chương trình 11.17

```

Map insertion time is 2546 milliseconds
Unordered map insertion time is 1033 milliseconds

```

11.4 Các thuật toán mẫu

Trong mục này, chúng tôi sẽ giới thiệu một số thuật toán đã được cài đặt sẵn trong STL để làm việc với vật chứa. Do khối lượng đồ sộ của STL, chúng tôi không thể liệt kê hết các thuật toán. Chúng tôi sẽ chỉ mô tả cách dùng một số thuật toán hay được sử dụng nhất đến bạn đọc nắm được tư tưởng thiết kế thuật toán của STL. Qua đó, bạn đọc có thể tìm hiểu thêm về STL từ các nguồn tư liệu khác dễ dàng hơn.

Trong STL, các thuật toán mẫu (hay hàm mẫu) được chuẩn hóa không những về cách gọi (tham số, kiểu trả về) mà còn đảm bảo tốc độ chạy của các thuật toán này. Nhờ đó, lập trình viên hoàn toàn kiểm soát được thời gian chạy chương trình của mình khi áp dụng thuật toán cung cấp bởi STL. Để hiểu hơn về hiệu suất của thuật toán, mục tiếp theo giới thiệu khái niệm “O-lớn” trừu tượng hóa cách đánh giá thời gian chạy.

11.4.1 Thời gian chạy và ký hiệu “O-lớn”

Khi nói về thời gian chạy của chương trình, một lập trình viên có thể nói: “Chương trình của tôi chạy mất 2 giây”. Tuy nhiên, ta cần biết 2 giây đó là thời gian chạy với bao nhiêu dữ liệu. Tất nhiên, ta không thể kỳ vọng một chương trình chạy mất 2 giây để sắp xếp 10 số thực cũng chỉ mất 2 giây để sắp xếp 1000 số thực. Do đó, ta cần một khái niệm quát mô tả thời gian chạy của chương trình phụ thuộc vào khối lượng dữ liệu nó xử lý.

Gọi $T(N)$ là thời gian chạy chương trình khi nó xử lý khối lượng dữ liệu là N . Rõ ràng, nếu đơn vị của $T(N)$ là thời gian (giờ, phút, giây) thì khi máy tính thay đổi, thời gian chạy của chương trình cũng thay đổi theo phụ thuộc vào tốc độ xử lý của CPU. Do đó, để thống nhất, người ta dùng số thao tác cơ bản để đánh giá $T(N)$. Thao tác cơ bản ở đây có thể là các phép gán, phép tính (cộng, trừ, nhân, chia, ...). Ví dụ, đoạn mã sau tính tổng của N số nguyên trong mảng a :

```
1: int sum = 0;
2: for (int i = 0; i < N; i++)
3:     sum += a[i];
4: cout << sum << endl;
```

Nếu coi các lệnh gán, so sánh, cộng và truy xuất mảng là các thao tác cơ bản thì các thao tác của đoạn mã trên là

- Dòng 1: có 1 thao tác;
- Dòng 2: có 3 thao tác, thực hiện N lần, tổng cộng $3N$ thao tác;
- Dòng 3: có 2 thao tác (truy xuất mảng và cộng), thực hiện N lần, tổng cộng $2N$ thao tác;
- Dòng 4: có 2 thao tác in.

Vậy tổng cộng đoạn mã trên có $5N + 3$ thao tác cơ bản, hay $T(N) = 5N + 3$. Do thời gian chạy một thao tác cơ bản phụ thuộc rất nhiều vào CPU, hằng số 5 đứng trước N trong biểu thức này không có ý nghĩa nhiều vì CPU có thể nhanh gấp 5 lần chẳng hạn. Hơn nữa, khi N lớn, hằng số 3 cũng không có nhiều ý nghĩa vì tốc độ chạy của chương trình phụ thuộc chủ yếu vào số N . Khi đó ta nói chương trình có **độ phức tạp thuật toán** là $O(N)$ (đọc là O-lớn của N).

Với cách tính gần tương tự như trên, thuật toán sắp xếp bằng 2 vòng lặp sau

```

1: for (int i = 0; i < N; i++)
2:     for (int j = i+1; j < N; j++)
3:         if (a[i] > a[j]) swap(a[i], a[j]);

```

có độ phức tạp thuật toán là $O(N^2)$. Ở đoạn mã này, ta có thể coi hàm `swap` là một thao tác cơ bản vì số phép toán cần để trao đổi hai biến là hằng số. Ở trường hợp này, công thức thật sự của $T(N)$ có thể phức tạp hơn nhiều nhưng với kí hiệu “ O -lớn”, ta chỉ quan tâm đến số hạng quan trọng nhất của $T(N)$ là N^2 .

Như vậy, với ký hiệu “ O -lớn”, ta có thể đánh giá thời gian chạy của các thuật toán không phụ thuộc nhiều vào kiến trúc tính toán (CPU). Đánh giá bằng “ O -lớn” cho ta công thức đơn giản và dễ hiểu hơn vì nó giản lược $T(N)$ bằng cách giữ lại số hạng quan trọng nhất.

11.4.2 Các thuật toán không thay đổi vật chứa

Trong mục này, chúng tôi giới thiệu một số thuật toán thao tác trên vật chứa nhưng không làm thay đổi vật chứa. Ví dụ tiêu biểu nhất cho thuật toán loại này là hàm `std::find`. Để sử dụng hàm này, bạn cần khai báo `#include <algorithm>`. Đây là tiêu đề chứa rất nhiều thuật toán trong STL. Hàm `find` có 3 tham số, người sử dụng dùng lời gọi `find(p1, p2, e)`. Trong đó `p1`, `p2` là hai con trỏ chỉ ra khoảng `[p1, p2)` (không tính phần tử do `p2` trỏ đến) ta cần tìm kiếm phần tử có giá trị `e` trong vật chứa. Nếu `find` tìm thấy `e`, nó sẽ trả về con trỏ duyệt trỏ đến `e`. Ngược lại, `find` trả về `p2`. Ví dụ trong hình 11.18 cho thấy cách sử dụng hàm `find` đối với lớp `vector<string>`.

```

1 // Chương trình minh họa thuật toán std::find
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5
6 int main()
7 {
8     using std::vector;
9     using std::find;
10    using std::cout;
11    using std::endl;
12    using std::string;
13
14    vector<string> names;
15    names.push_back("Long");
16    names.push_back("Thai");
17
18    auto pLong = find(names.begin(), names.end(), "Long");
19    cout << *pLong << endl; // "Long"
20
21    auto pVinh = find(names.begin(), names.end(), "Vinh");
22    if (pVinh == names.end())
23        cout << "Cannot find Vinh" << endl;
24
25    return 0;
26 }

```

Hình 11.18: Minh họa hàm `std::find`

Output chương trình 11.18

Long
Cannot find Vinh

Bảng 11.19 mô tả một số thuật toán truy xuất vật chứa khác. Lưu ý, các thuật toán này đều làm việc trên con trỏ duyệt của vật chứa tương ứng. Các thuật toán này đều có độ phức tạp $O(N)$ với N là kích thước dãy cần truy xuất (tức là $p2-p1$). Riêng thuật toán `binary_search` là trường hợp ngoại lệ, có độ phức tạp $O(\log N)$.

Bảng 11.19: Các thuật toán không thay đổi vật chứa.

Thuật toán	Ví dụ
<code>find(p1, p2, e)</code>	Tìm kiếm phần tử <code>e</code> trong khoảng <code>[p1, p2)</code>
<code>count(p1, p2, e)</code>	Đếm số phần tử có giá trị <code>e</code> trong khoảng <code>[p1, p2)</code>
<code>equal(p1, p2, p)</code>	Chỉ trả về <code>true</code> nếu các phần tử trong khoảng <code>[p1, p2)</code> giống hệt <code>p2-p1</code> phần tử tính từ phần tử <code>p</code> đang trở tới.
<code>search(p1, p2, t1, t2)</code>	Nếu dãy các phần tử trong <code>[t1, t2)</code> là dãy con của khoảng <code>[p1, p2)</code> , trả về con trỏ duyệt đến vị trí đầu tiên bắt đầu dãy con trong <code>[p1, p2)</code> . Ngược lại, nếu không tìm thấy trả về <code>p2</code> .
<code>binary_search(p1, p2, e)</code>	Tìm kiếm phần tử <code>e</code> trong khoảng <code>[p1, p2)</code> bằng phương pháp tìm kiếm nhị phân, chỉ trả về <code>true</code> nếu tìm thấy. Điều kiện của thuật toán này là các phần tử trong khoảng <code>[p1, p2)</code> đã được sắp xếp theo thứ tự tăng dần. Độ phức tạp của phương pháp tìm kiếm nhị phân là $O(\log N)$.
<code>foreach(p1, p2, func)</code>	Áp dụng hàm <code>func</code> cho tất cả các phần tử trong khoảng <code>[p1, p2)</code> . Đối số của hàm <code>func</code> là tham chiếu hằng đến kiểu của phần tử trong vật chứa.
<code>max(v1, v2)</code>	Trả về giá trị lớn nhất trong hai giá trị <code>v1, v2</code> .
<code>min(v1, v2)</code>	Trả về giá trị nhỏ nhất trong hai giá trị <code>v1, v2</code> .
<code>accumulate(p1, p2, e)</code>	Trả về giá trị bằng <code>e</code> cộng với tổng của tất cả các phần tử trong khoảng <code>[p1, p2)</code> .

Có lẽ, hàm mà lập trình viên C++ hay sử dụng nhất là hàm `for_each`. Hàm này chạy hàm `func` hoặc một đối tượng thuộc lớp có định nghĩa toán tử `()` (toán tử gọi hàm). Chương trình trong hình 11.20 tính tổng các phần tử trong một `vector` vào trong biến toàn cục `sum`.

```

1 // Tính tổng vector bằng foreach
2 #include <iostream>
3 #include <cstdlib>
4 #include <vector>
5 #include <algorithm>
6

```

```

7 using namespace std;
8
9 double sum = 0;
10
11 void Accumulate(double& v)
12 {
13     sum += v;
14 }
15
16 int main()
17 {
18     srand(0);
19     const int n = 100;
20     vector<double> v;
21     for (int i = 0; i < n; i++)
22         v.push_back( (double)(rand()) / RAND_MAX );
23
24     for_each(v.begin(), v.end(), Accumulate);
25     cout << sum << endl;
26     return 0;
27 }

```

Hình 11.20: Minh họa hàm `std::for_each` sử dụng hàm

Để không phải sử dụng biến toàn cục như trên, ta có thể khai báo một lớp với toán tử `()`. Các đối tượng thuộc lớp có toán tử `()` còn gọi là còn gọi là các **đối tượng hàm** (*function object*) (Xem chương trình trong hình 11.21).

```

1 // Tính tổng vector bằng foreach và doi tuong ham
2 #include <iostream>
3 #include <cstdlib>
4 #include <vector>
5 #include <algorithm>
6
7 using namespace std;
8
9 class Sum {
10     double sum;
11 public:
12     Sum() : sum(0) {}
13     void operator()(const double& v) { sum += v; }
14     double getSum() const { return sum; }
15 };
16
17 int main()
18 {
19     srand(0);
20     const int n = 100;
21     vector<double> v;
22     for (int i = 0; i < n; i++)
23         v.push_back( (double)(rand()) / RAND_MAX );
24
25     Sum sum = for_each(v.begin(), v.end(), Sum());
26     cout << sum.getSum() << endl;

```

```

27     return 0;
28 }

```

Hình 11.21: Minh họa hàm `std::for_each` sử dụng đối tượng hàm

Để ý rằng, lệnh `for_each` trả về một đối tượng hàm là kết quả thực hiện hàm. Ta cần lưu lại đối tượng này để lấy kết quả.

11.4.3 Các thuật toán thay đổi vật chứa

Có rất nhiều thuật toán thay đổi vật chứa trong `<algorithm>` như: hoán đổi `swap`, sao chép `copy`, xóa phần tử `remove`, đảo ngược dãy `reverse`, tráo đổi ngẫu nhiên `random_shuffle`, đặt giá trị phần tử `fill`, `iota`, biến đổi giá trị `transform`, sắp xếp `sort` v.v... và rất nhiều biến thể và thuật toán khác. Tất cả các lệnh này đều thao tác với các khoảng nằm giữa 2 con trỏ duyệt. Bảng 11.22 mô tả một số lệnh thông dụng của `<algorithm>`.

Bảng 11.22: Các thuật toán có thể thay đổi vật chứa, độ phức tạp $O(N)$.

Thuật toán	Ví dụ
<code>swap(v1, v2)</code>	Tráo đổi giá trị trỏ tới bởi hai tham chiếu <code>v1</code> và <code>v2</code> .
<code>copy(p1, p2, t1, t2)</code>	Sao chép các phần tử trong khoảng <code>[p1, p2)</code> sang khoảng <code>[t1, t2)</code> . Điều kiện là số phần tử bằng nhau (tức là <code>t2 - t1 == p2 - p1</code>).
<code>remove(p1, p2, e)</code>	Xóa các phần tử bằng giá trị <code>e</code> trong khoảng <code>[p1, p2)</code> .
<code>reverse(p1, p2)</code>	Đảo ngược thứ tự các phần tử trong khoảng <code>[p1, p2)</code> .
<code>random_shuffle(p1, p2)</code>	Tráo đổi ngẫu nhiên thứ tự các phần tử trong khoảng <code>[p1, p2)</code> .
<code>fill(p1, p2, e)</code>	Đặt giá trị các phần tử trong khoảng <code>[p1, p2)</code> bằng <code>e</code> .
<code>iota(p1, p2, e)</code>	Đặt giá trị các phần tử trong khoảng <code>[p1, p2)</code> tăng dần bắt đầu bằng <code>e</code> , tức là <code>e, ++e, ++e, ...</code> .
<code>transform(p1, p2, p, func)</code>	Biến đổi các phần tử trong khoảng <code>[p1, p2)</code> bằng hàm <code>func</code> và đặt vào các vị trí tương ứng bắt đầu từ con trỏ <code>p</code> .
<code>unique(p1, p2)</code>	Loại bỏ tất cả các giá trị trùng lặp trong khoảng <code>[p1, p2)</code> .
<code>sort(p1, p2)</code>	Sắp xếp các phần tử trong khoảng <code>[p1, p2)</code> tăng dần với độ phức tạp $O(N \log N)$.

```

1 // Các hàm thay đổi vật chứa
2 #include <iostream>
3 #include <cstdlib>
4 #include <vector>
5 #include <algorithm>
6
7 using namespace std;

```



```

8
9 class DoubleIt {
10 public:
11     int operator()(const int& v) { return v*2; }
12 };
13
14 template <typename IT>
15 void Print_Vec(string msg, IT p1, IT p2)
16 {
17     cout << msg;
18     for (; p1 != p2; p1++) cout << *p1 << " ";
19     cout << endl;
20 }
21
22 int main()
23 {
24     const int n = 10;
25     vector<int> v(n);
26
27     iota(v.begin(), v.end(), 1);
28     Print_Vec("Start: ", v.begin(), v.end());
29
30     random_shuffle(v.begin(), v.end());
31     Print_Vec("shuffled: ", v.begin(), v.end());
32
33     sort(v.begin(), v.end());
34     Print_Vec("sorted: ", v.begin(), v.end());
35
36     transform(v.begin(), v.end(), v.begin(), DoubleIt());
37     Print_Vec("doubled: ", v.begin(), v.end());
38     return 0;
39 }

```

Hình 11.23: Minh họa một số hàm thay đổi vật chứa.

Output chương trình 11.23

```

Start: 1 2 3 4 5 6 7 8 9 10
shuffled: 8 2 4 9 5 7 10 6 1 3
sorted: 1 2 3 4 5 6 7 8 9 10
doubled: 2 4 6 8 10 12 14 16 18 20

```

Chương trình bắt đầu bằng lệnh `iota` khởi tạo giá trị `vector v` bằng các số từ 1 đến 10, sau đó các giá trị này bị xáo trộn ngẫu nhiên bằng lệnh `random_shuffle`. Các giá trị bị xáo trộn được sắp xếp lại bằng lệnh `sort`. Cuối cùng, các giá trị được nhân với 2 bằng lệnh `transform`. Để cho tiện, chúng tôi viết hàm mẫu `print_vec` để in các giá trị nằm trong khoảng `[p1, p2)`. Lớp `DoubleIt` cài đặt toán tử hàm `()` tính giá trị gấp đôi của số nguyên. Đối tượng của lớp này là tham số của hàm `transform` sẽ thay các giá trị trong `vector` bằng giá trị gấp đôi.

Ta thấy trong chương trình 11.23, các lệnh `v.begin()`, `v.end()` lặp lại nhiều lần. Để cho gọn, nhiều lập trình viên viết một lệnh tiền xử lý như sau

```
#define ALL(c) (c).begin(), (c).end()
```

Khi đó, các câu lệnh trong chương trình 11.23 có thể thay bằng các câu lệnh rõ nghĩa hơn nhiều

```
iota(ALL(v), 1);
print_vec("Start: ", ALL(v));

random_shuffle(ALL(v));
print_vec("shuffled: ", ALL(v));

sort(ALL(v));
print_vec("sorted: ", ALL(v));

transform(ALL(v), v.begin(), DoubleIt());
print_vec("doubled: ", ALL(v));
```

Bảng 11.24: Một số chỉ lệnh tiện xử lý hay dùng với STL.

```
#define ALL(c) (c).begin(), (c).end()
```

Đại diện cho toàn bộ phần tử trong vật chứa `c`.

```
#define SIZE(c) ((int)(c).size())
```

Số phần tử trong vật chứa `c` (kiểu `int`).

```
#define TRACE(c,it) for(auto it = (c).begin(); it != (c).end(); it++)
```

Duyệt toàn bộ vật chứa `c` bằng vòng lặp `for` với con trỏ duyệt `it`.

```
#define PRESENT(c,ele) ((c).find(ele) != (c).end())
```

Biểu thức logic trả về `true` nếu phần tử `ele` có trong vật chứa `c`.

```
#define CPRESENT(c,ele) (find(ALL(c),ele) != (c).end())
```

Biểu thức logic trả về `true` nếu phần tử `ele` có trong vật chứa `c` (sử dụng thư viện `<algorithm>`).

```
#define pb push_back
```

Rút gọn lệnh `push_back`.

11.4.4 Các thuật toán tập hợp

Tập hợp là một khái niệm hết sức quan trọng trong khoa học máy tính và các thuật toán của nó. Trong C++, ngoài cấu trúc dữ liệu tập hợp, ta còn có thể biểu diễn tập hợp dưới dạng một dãy đã được sắp xếp (từ nhỏ đến lớn). Khi đó, C++ cung cấp các hàm là các phép toán trên tập hợp như hợp, giao, hiệu hai tập hợp và phép toán xác định tập hợp con. Bảng 11.25 mô tả các hàm thao tác với tập hợp mà thư viện C++ đã cung cấp sẵn. Bạn đọc có thể tự mình thử nghiệm với chúng.

11.5 Một số thư viện chuẩn khác trong STL

STL là một thư viện khá đồ sộ. Trong các chương trước và phần trước của chương này, chúng tôi đã giới thiệu một phần nhỏ của STL bao gồm các thư viện vào / ra, các vật chứa và các thuật toán làm việc trên vật chứa. Ngoài các thư viện trên, C++ còn cung cấp một loạt các thư viện hỗ trợ lập trình viên thao tác với xâu, bộ nhớ, thời gian, các luồng chương trình song song. Trong các

Bảng 11.25: Các thuật toán thao tác với tập hợp.

Thuật toán	Ví dụ
<code>merge(p1, p2, t1, t2, p)</code>	Trộn hai dãy đã được sắp xếp trong khoảng <code>[p1, p2)</code> và <code>[t1, t2)</code> thành dãy được sắp xếp bắt đầu bởi <code>p</code> .
<code>inplace_merge(p1, mid, p2)</code>	Trộn hai dãy đã được sắp xếp trong khoảng <code>[p1, mid)</code> và <code>[mid, p2)</code> thành dãy được sắp xếp trong khoảng <code>[p1, p2)</code> .
<code>set_difference(p1, p2, t1, t2, p)</code>	Thêm các phần tử trong khoảng <code>[p1, p2)</code> không nằm trong khoảng <code>[t1, t2)</code> vào dãy bắt đầu bởi <code>p</code> (phép hiệu tập hợp).
<code>set_intersection(p1, p2, t1, t2, p)</code>	Thêm các phần tử nằm trong cả hai khoảng <code>[p1, p2)</code> và <code>[t1, t2)</code> vào dãy bắt đầu bởi <code>p</code> (phép giao tập hợp).
<code>includes(p1, p2, t1, t2)</code>	Chỉ trả về <code>true</code> nếu mọi phần tử trong khoảng <code>[t1, t2)</code> là phần tử trong khoảng <code>[p1, p2)</code> (phép chứa tập hợp).

mục dưới đây, chúng tôi sẽ giới thiệu các thư viện này. Bạn đọc muốn tìm hiểu kĩ hơn các thư viện trong STL có thể đọc trên trang web <http://cplusplus.com>.

11.5.1 Xử lý chuỗi với `<string>`

Ta đã biết trong ngôn ngữ C, chuỗi ký tự được biểu diễn bằng một mảng ký tự `char[]`. Để thao tác với chuỗi ký tự dạng này, lập trình viên sử dụng thư viện `<string.h>` hoặc `<cstring>`. Các hàm thao tác với chuỗi ký tự trong `<string.h>` thường bắt đầu bằng tiền tố `str` như `strlen`, `strcat`, ... Các thao tác với chuỗi ký tự trong C không thuận tiện và đòi hỏi lập trình viên phải luôn để ý đến việc cấp phát đủ bộ nhớ cho chuỗi ký tự. Để giải quyết các khó khăn này, trong C++, lập trình viên có thể sử dụng lớp đối tượng `string` thuộc vào thư viện `<string>`. Các thao tác trên đối tượng `string` dễ dàng hơn nhiều so với ngôn ngữ C. Người sử dụng hầu như không phải quan tâm đến việc cấp phát bộ nhớ cho chuỗi ký tự của mình.

Khai báo chuỗi ký tự. Lập trình viên có thể khai báo chuỗi ký tự bằng nhiều cách do `<string>` đã xây dựng các hàm khởi tạo tương ứng. Ví dụ:

```
string s; // Khởi tạo chuỗi rỗng
string s1 = "Hello, world"; // chuỗi có nội dung "Hello, world"
string s2(3, 'a'); // chuỗi có 3 ký tự a
string s3("abc", 2); // 2 ký tự đầu "ab"
string s4(s1.begin(), s1.end()); // copy chuỗi s1
```

Chuỗi ký tự giống `vector<char>`. Chuỗi ký tự có thể coi như một `vector` các ký tự. Lớp `string` cài đặt các hàm lấy con trỏ duyệt như `begin()`, `end()`, `rbegin()`, `rend()`. Do đó, các thuật

toán trên vật chứa đã bàn ở mục trên đều có thể chạy trên **string**. Ngoài ra, **string** còn có các hàm như **size()**, **clear()**, **empty()**, **push_back()**, **insert()**, **erase()**, **find()** giống như một vật chứa bình thường.

Các thao tác riêng cho chuỗi ký tự.

- Toán tử cộng: Để nối 2 chuỗi, ta dùng toán tử **+** hoặc **+=**. Toán tử này có thể dùng để cộng các **string** với nhau hoặc với ký tự hoặc chuỗi ký tự kiểu C. Ví dụ:

```
string s5 = s1 + ' ' + s2 + " ok"; // chuỗi có nội dung "Hello, world aaa ok"
s3 += s2; // s3 trở thành abcaaa
```

- Lấy một đoạn con: Hàm thành viên **substr(pos, n)** cho chuỗi ký tự là chuỗi con bắt đầu từ vị trí **pos** và có **n** ký tự. Ví dụ:

```
string s6 = s1.substr(0, 5) // chuỗi có nội dung "Hello"
```

- So sánh 2 chuỗi: Hàm thành viên **compare(s)** so sánh chuỗi theo thứ tự từ điển với chuỗi **s** là tham số đầu vào. Kết quả trả về bằng 0 nếu 2 chuỗi giống hệt nhau. Kết quả *nhỏ hơn 0* nếu **s** đứng trước trong từ điển và *lớn hơn 0* nếu ngược lại. Ngoài ra, các **string** có thể so sánh bằng các toán tử so sánh như **==**, **!=**, **<**, **<=**, **>=**, **>**.
- Tìm kiếm chuỗi: Với **<string>**, ta có thể tìm kiếm một chuỗi con bằng các hàm **find_first_of(s,pos)** (tìm từ trái qua) và **find_last_of(s,pos)** (tìm từ phải qua). Hãy xem cách sử dụng các hàm này trong ví dụ 11.26.
- Đọc chuỗi từ luồng nhập: Để đọc chuỗi từ một luồng nhập **istream**, có thể dùng toán tử **>>** hoặc sử dụng hàm **std::getline()**. Hãy xem cách sử dụng hàm **std::getline()** trong ví dụ 11.26.

Ví dụ dưới đây đọc vào một file văn bản có định dạng CSV (comma separated values). Dữ liệu trong file CSV được phân cách bởi các dấu phẩy. Do đó khi đọc file để lấy dữ liệu, ta cần tìm vị trí các dấu phẩy trên từng dòng.

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <string>
5
6 using namespace std;
7
8 class CSVRow : public vector<string> {
9 public:
10     void print(ostream& s) {
11         for (unsigned int i = 0; i < size(); i++)
12             s << at(i) << " ";
13     }
14
15     void read(istream& s) {
16         string line;
17         clear();
18         getline(s, line); // read a line from the stream
```

```

19     size_type prev = 0, pos; // start from first position
20     while ( (pos = line.find_first_of(',', prev)) != string::npos) {
21         // found a comma
22         push_back(line.substr(prev, pos-prev));
23         prev = pos+1;
24     }
25     push_back(line.substr(prev));
26 }
27 };
28
29 istream & operator>>(istream& s, CSVRow& row)
30 {
31     row.read(s);
32     return s;
33 }
34
35 ostream & operator<<(ostream& s, CSVRow& row)
36 {
37     row.print(s);
38     return s;
39 }
40
41 int main(int argc, char** argv)
42 {
43     if (argc < 2) { // check CSV filename on command line
44         cout << "Please provide CSV file name." << endl;
45         return 0;
46     }
47
48     ifstream f(argv[1]);
49     CSVRow row;
50     while (f >> row) { // read a CSVrow
51         cout << row << endl; // print row without commas
52     }
53     return 0;
54 }

```

Hình 11.26: Minh họa thư viện `<string>`: Đọc file định dạng CSV.

Tạo một file văn bản `test.csv` có nội dung như sau

```

Hello, world,,,
Vinh,Thai
Long

```

Chạy chương trình trên với tham số dòng lệnh là `test.csv` ta được

```

Hello world
Vinh Thai
Long

```

11.5.2 Con trỏ thông minh và quản lý bộ nhớ với <memory> (C++11)

Có lẽ lập trình viên C hoặc C++ nào cũng từng phải đau đầu với lỗi *rỏ rỉ bộ nhớ* khi quên không giải phóng bộ nhớ đã cấp phát động bằng **malloc** hoặc **new** sử dụng con trỏ. Với C++11, vấn đề này phần nào được giải quyết với *con trỏ thông minh*. Khi cấp phát bộ nhớ, con trỏ thông minh không khác gì con trỏ bình thường. Do đó, gần như mọi đoạn mã sử dụng con trỏ bình thường đều có thể thay thế bằng con trỏ thông minh. Tuy nhiên, con trỏ thông minh *tự động giải phóng bộ nhớ* của đối tượng được cấp phát vào thời điểm “thích hợp”. Khi khai báo con trỏ thông minh, ta chỉ cần khai báo kiểu dữ liệu mà chúng trỏ đến dưới dạng tham số kiểu của **template**. Để sử dụng con trỏ thông minh, chỉ cần sử dụng câu lệnh **#include <memory>**.

Bộ nhớ dùng chung với `shared_ptr`. Khi nhiều con trỏ thường cùng trỏ đến một đối tượng được cấp phát động. Nếu ta dùng lệnh **delete** với một con trỏ, ngay lập tức, phần bộ nhớ do các con trỏ còn lại trỏ đến trở nên không hợp lệ. Với C++11, con trỏ thông minh **shared_ptr** giải quyết vấn đề này. Ta có thể có nhiều con trỏ trỏ đến cùng một đối tượng. Khi đó, các con trỏ này lưu giữ một biến đếm tăng khi có thêm một con trỏ và giảm khi một con trỏ bị hủy. Khi nào biến đếm bằng 0, bộ nhớ tương ứng sẽ được giải phóng.

Một điểm yếu của **shared_ptr** là khi các con trỏ này trỏ đến các đối tượng theo vòng tròn thì chúng sẽ không bao giờ bị giải phóng. Để giải quyết vấn đề này, C++11 sử dụng con trỏ thông minh **weak_ptr**. Chúng tôi không đi vào chi tiết ở đây nhưng con trỏ này cho phép quản lý bộ nhớ “lỏng lẻo” hơn nhờ đó vấn đề vòng tròn nêu trên không còn nữa. Đồng thời, với **weak_ptr**, ta luôn có thể kiểm tra xem bộ nhớ do các con trỏ trỏ đến có còn hợp lệ hay không.

Một số lưu ý khi dùng con trỏ thông minh. Các con trỏ thông minh bản chất là đối tượng trong C++. Để sử dụng chúng hiệu quả, cần tuân thủ một số quy tắc sau:

- Nên tạo **shared_ptr** ngay khi cấp phát bộ nhớ và chỉ sử dụng con trỏ này để tạo các con trỏ **share_ptr** và **weak_ptr** khác.
- Không sử dụng con trỏ thông thường trỏ đến cùng đối tượng với con trỏ thông minh.
- Không sử dụng **new** để cấp phát bộ nhớ, thay vào đó, dùng hàm mẫu **std::make_shared()**.

Ví dụ. Quan sát các thông báo trong ví dụ sau để thấy hoạt động của các con trỏ thông minh. Để dịch chương trình này, bạn cần khai báo sử dụng chuẩn C++11 với trình biên dịch. Nếu bạn dùng GNU C++ Compiler (**g++**), hãy thêm **-std=c++11** vào dòng lệnh biên dịch.

```

1 #include <memory>
2 #include <iostream>
3
4 using namespace std;
5
6 class Thing {
7     int value;
8 public:
9     Thing(int v = 0) : value(v) {
10         cout << "Created" << endl;
11     }
12     ~Thing() {
13         cout << "Deleted" << endl;

```

```

14     }
15
16     int get() const { return value; }
17     void set(int v) { value = v; }
18 };
19
20 int main()
21 {
22     shared_ptr<Thing> p1(make_shared<Thing>());
23     cout << p1->get() << endl; // p1->value = 0
24
25     shared_ptr<Thing> p2(p1); // p2 points to p1's object
26     p2->set(2);
27     cout << p1->get() << endl; // p1->value = 2
28 }

```

Hình 11.27: Minh họa thư viện `<memory>`.

Chạy chương trình trên, bạn sẽ nhận được các thông báo sau:

```

Created
0
2
Deleted

```

Lưu ý rằng, ta không cần phải tự giải phóng bộ nhớ bằng lời gọi hàm `delete`. Các con trỏ thông minh tự biết khi nào cần phải giải phóng bộ nhớ (khi hàm hủy của cả hai con trỏ `p1` và `p2` chạy). Để ý trong chương trình trên, đối tượng `Thing` chỉ được khởi tạo và giải phóng đúng một lần.

Con trỏ `unique_ptr`. Một loại con trỏ thông minh khác là `unique_ptr`. Con trỏ này không cho phép 2 con trỏ chỉ đến cùng một vùng nhớ cấp phát động, ngược lại với `shared_ptr`. Do đó con trỏ này khởi tạo và giải phóng bộ nhớ nhanh hơn.

```

unique_ptr<Thing> p1 (new Thing); // Khởi tạo unique_ptr trỏ đến đối tượng
unique_ptr<Thing> p2(p1); // Có lỗi - không thể có 2 con trỏ chỉ đến cùng đối tượng
unique_ptr<Thing> p3;
p3 = p1; // Có lỗi, không thể gán con trỏ kiểu này
unique_ptr<Thing> p4;
p4 = std::move(p1); // Chuyển sở hữu đối tượng sang p4
unique_ptr<Thing> p5(std::move(p4)); // Giờ p5 sở hữu đối tượng, p1 và p4 trắng tay

```

11.5.3 Tính toán thời gian với `<chrono>` (C++11)

Trong C++11, có một cách thống nhất để làm việc với thời gian là dùng thư viện `<chrono>`. Trước đây, khi làm việc với thời gian, lập trình viên thường phải có đoạn mã riêng cho từng hệ điều hành cụ thể như UNIX hoặc Windows. Với `chrono`, lập trình viên có một giao diện (API) chung để tính toán, đo đạc thời gian của chương trình. Thư viện này quản lý 2 loại đối tượng: **thời lượng** (duration) và **thời điểm** (time-point). Ngoài ra, thư viện còn cung cấp nhiều kiểu **đồng hồ** (clock) để đo đạc thời gian.

Thời lượng trong `chrono` được tính bằng giờ (hours), phút (minutes), giây (seconds), mili-giây (milliseconds), micro-giây (microseconds) và nano-giây (nanoseconds). Lập trình viên có thể dùng

các toán tử `+`, `-`, `+=`, `-=`, `<`, `>` để thao tác với thời điểm và thời lượng. Ví dụ: hiệu của hai thời điểm là thời lượng giữa hai thời điểm đó, tổng của thời điểm với thời lượng là một thời điểm khác, v.v... Ví dụ dưới đây minh họa cách tính toán thời điểm hiện tại và sử dụng các toán tử trên.

```

1 #include <iostream>
2 #include <chrono>
3
4 using namespace std;
5 using chrono::system_clock;
6
7 int main()
8 {
9     typedef chrono::duration<int> seconds_type;
10    typedef chrono::duration<int, std::milli> milliseconds_type;
11    typedef chrono::duration<int, std::micro> microseconds_type;
12
13    seconds_type s_oneday(60*60*24); // Số giây trong 1 ngày
14    milliseconds_type ms_oneday(s_oneday); // Số milli-giây trong 1 ngày
15
16    time_t tt;
17
18    system_clock::time_point today = system_clock::now(); // thời điểm hiện tại
19    tt = system_clock::to_time_t( today );
20    cout << "Today is " << ctime(&tt);
21
22    system_clock::time_point tomorrow = today + s_oneday; // ngày mai
23    tt = system_clock::to_time_t( tomorrow );
24    cout << "Tomorrow is " << ctime(&tt);
25
26    system_clock::time_point dayafter = tomorrow + ms_oneday; // ngày kia
27    tt = system_clock::to_time_t( dayafter );
28    cout << "The day after tomorrow is " << ctime(&tt);
29
30    // Số micro-giây giữa 2 thời điểm
31    microseconds_type mic_duration = chrono::duration_cast<microseconds_type>(
32        dayafter - today);
33    cout << "Duration is " << mic_duration.count() << " microseconds" << endl;
34 }
```

Hình 11.28: Minh họa thư viện `<chrono>`.

Output chương trình 11.28.

```

Today is Sun Mar 20 10:10:22 2016
Tomorrow is Mon Mar 21 10:10:22 2016
The day after tomorrow is Tue Mar 22 10:10:22 2016
Duration is 1001308160 microseconds
```

11.5.4 Lập trình song song với `<thread>` (C++11)

Trước khi có C++11, lập trình viên muốn tận dụng sức mạnh của hệ thống máy tính đa nhân hoặc đa luồng phải sử dụng các thư viện chuyên biệt cho từng hệ điều hành. Ví dụ, trên UNIX là

thư viện **pthread**s. Với C++11, lập trình viên có thể khai báo các luồng chương trình song song và thực hiện chúng một cách dễ dàng.

Đoạn chương trình ngắn sau cho thấy cách tạo và chạy nhiều luồng độc lập với hàm **main()**. Các luồng này in ra số thứ tự của chúng. Để ý rằng các thông báo không tuân theo một thứ tự cố định nào cả.

```

1 #include <iostream>
2 #include <thread>
3 #include <vector>
4
5 using namespace std;
6
7 void Thread_Hello(int tid) {
8     cout << "Hello, world from thread " << tid << endl;
9 }
10
11 int main()
12 {
13     vector<thread> t(10);
14     // Khởi động 10 luồng và gọi hàm thread_hello
15     for (unsigned int i = 0; i < t.size(); i++)
16         t[i] = thread(Thread_Hello, i);
17     // Đợi các luồng kết thúc
18     for (unsigned int i = 0; i < t.size(); i++)
19         t[i].join();
20     return 0;
21 }
```

Hình 11.29: Khởi động luồng với **<thread>**.

Output chương trình 11.29.

```

Hello, world from thread 1
Hello, world from thread 4
Hello, world from thread 3
Hello, world from thread 2
Hello, world from thread 0
Hello, world from thread 6
Hello, world from thread 7
Hello, world from thread 5
Hello, world from thread 8
Hello, world from thread 9
```

Đoạn chương trình sau sử dụng **T** luồng để tính tổng của phép toán $\log(1 + \exp(x))$ với x là các phần tử của mảng số thực gồm **N** phần tử. Các tham số **N**, **T** là tham số dòng lệnh. Ta sẽ quan sát thời gian chạy của chương trình khi **T** thay đổi.

```

1 #include <iostream>
2 #include <thread>
3 #include <memory>
4 #include <chrono>
5 #include <vector>
6 #include <cstdlib>
```

```

7  #include <cmath>
8
9  using namespace std;
10
11 typedef vector<double> vd_t;
12
13 // compute the sum of log(1+exp(a[i])) from position s to e
14 void Compute(const vector<double>& a, int s, int e, double& result) {
15     result = 0;
16     for (int i = s; i < e; i++)
17         result += log(1+exp(a[i]));
18 }
19
20 int main(int argc, char** argv)
21 {
22     typedef chrono::duration<int, std::micro> microseconds_type;
23     typedef chrono::system_clock::time_point time_point;
24     // record start time
25     time_point start = chrono::system_clock::now();
26     int N = atoi(argv[1]);
27     int T = atoi(argv[2]);
28
29     vd_t a(N, 1); // initialize N-element array
30
31     // compute partial sums
32     vector<thread> threads(T);
33     vd_t result(T);
34     for (int i = 0; i < T; i++) {
35         int s = i*(N/T), e = i < T-1 ? (i+1)*(N/T) : N;
36         threads[i] = thread(Compute, std::ref(a), s, e, std::ref(result[i]));
37     }
38     for (int i = 0; i < T; i++) threads[i].join();
39
40     // gather results
41     double ret = 0;
42     for (int i = 0; i < T; i++) ret += result[i];
43     cout << "Value is " << ret;
44
45     // compute runtime
46     time_point end = chrono::system_clock::now();
47     microseconds_type duration = chrono::duration_cast< microseconds_type >(end -
48         start);
49     cout << ", total runtime is " << duration.count() << " microseconds" << endl;
50     return 0;
51 }

```

Hình 11.30: Tính toán song song với `<thread>`.

Output chương trình 11.30 với $N = 100000000$ và $T = 1$.

Value is 1.31326e+08, total runtime is 7394521 microseconds

Output chương trình 11.30 với $N = 100000000$ và $T = 20$.

Value is 1.31326e+08, total runtime is 2616012 microseconds

Thông báo của chương trình trên cho thấy khi dùng nhiều luồng song song, thời gian chạy của chương trình giảm đi rõ rệt. Đây chính là lợi ích của chương trình được lập trình song song, tận dụng hết số nhân của bộ vi xử lý đa nhân (multi-core).

Bài tập

1. Viết chương trình nhập vào một dãy số nguyên sử dụng `vector<int>` và một giá trị nguyên `x`. Sử dụng thư viện `<algorithm>`
 - (a) Tìm giá trị nhỏ nhất bằng lệnh `min_element`.
 - (b) Tìm giá trị nhỏ nhất bằng lệnh `max_element`.
 - (c) Hỏi `x` có tồn tại trong mảng không?
 - (d) Liệt kê các vị trí trong mảng bằng giá trị `x`.
 - (e) Đếm số giá trị bằng `x` trong mảng bằng lệnh `count`.
 - (f) Thử các lệnh của thư viện `<algorithm>` trên mảng.
2. Viết chương trình nhập vào danh sách sinh viên với các thông tin: tên - `string`, tuổi - `int` sử dụng `vector`.
 - (a) Viết hàm in tên và tuổi của một sinh viên.
 - (b) Viết hàm sử dụng hàm in ở trên và hàm `foreach` của thư viện `<algorithm>` để in danh sách sinh viên.
 - (c) Viết hàm chuẩn hóa tên của một sinh viên (viết hoa đầu các từ, khoảng cách giữa các từ bằng 01 khoảng trắng, không có dấu cách ở đầu và cuối tên).
 - (d) Sử dụng hàm `transform` của thư viện `<algorithm>` để chuẩn hóa tất cả các tên trong danh sách sinh viên.
 - (e) Viết hàm so sánh 2 sinh viên bằng thứ tự tên (so sánh tên trước, họ và đệm sau).
 - (f) Sử dụng hàm so sánh vừa viết và hàm `sort` để sắp xếp danh sách sinh viên.
 - (g) Sử dụng hàm `unique` để loại bỏ tất cả các sinh viên trùng lặp trong danh sách.
3. **(Từ điển).** Trong bài tập này, bạn sẽ xây dựng công cụ từ điển Anh - Việt.
 - (a) Từ điển là một ánh xạ - `map` giữa xâu tiếng Anh và xâu tiếng Việt.
 - (b) Viết hàm nhập từ điển từ một tệp văn bản có định dạng như sau: mỗi dòng trong tệp là một cặp từ Anh - Việt cách nhau bởi dấu hai chấm (:), cần loại bỏ các khoảng trắng không cần thiết như ví dụ sau:


```
hello : xin chào
Dog : con chó
apple : quả táo
measurement : số đo
```
 - (c) Viết hàm tìm nghĩa tiếng Việt của một từ tiếng Anh trong từ điển. Nếu đầu vào không có trong từ điển, trả về chính từ đó. Nâng cao: cần tìm được nghĩa ngay cả khi đầu vào có các ký tự hoa, thường khác nhau.
 - (d) Viết hàm tách một đoạn văn bản tiếng Anh thành một chuỗi - `vector` - các từ tiếng Anh và các ký tự phân cách (dấu chấm, phẩy, dấu cách, ...).
 - (e) Viết hàm dịch một đoạn văn bản tiếng Anh bằng cách thay các từ tiếng Anh bằng nghĩa tiếng Việt tương ứng trong từ điển (sử dụng hàm ở trên). Nâng cao: cần giữ được cách viết hoa đầu câu khi chuyển ngữ.

Phụ lục A

Bảng từ khóa của ngôn ngữ C++

```
asm      auto
break
case     cdecl  char   class   const   continue
default delete  do      double
else     enum   extern
float    for friend
goto
huge
if        inline  int     interrupt
long
near      new
operator
pascal   private protected      public
register          return
short    signed  sizeof  static  struct    switch
template          this    typedef
union    unsigned
virtual  void     volatile
while
_cs      _ds      _es      _export far
_fastcall _loadds _saveregs      _seg      _ss
```


Phụ lục B

Thứ tự ưu tiên của phép toán

Thứ tự ưu tiên của các phép toán được quy định theo một số nguyên tắc:

- Phép toán một ngôi, Phép toán hai ngôi, phép gán;
- Cùng loại phép toán vẫn chia thành thứ tự ưu tiên khác nhau: ví dụ cùng các phép toán Logic thì $!$ có độ ưu tiên cao hơn (vì là một ngôi) $\&\&$, $||$ (hai ngôi). Cùng phép toán số học thì $*$, $/$ (còn gọi là lớp nhân) có độ ưu tiên cao hơn $+$, $-$ (còn gọi là lớp cộng)...;
- Khi trong biểu thức xuất hiện các phép toán cùng độ ưu tiên liên tiếp nhau, thì hầu hết các phép toán sẽ được tính từ trái sang phải, trừ các phép toán gán và điều kiện;
- Cặp dấu ngoặc tròn có độ ưu tiên cao nhất (sau toán tử phạm vi), do tính đối xứng nên với bất kỳ chiều kết hợp nào các biểu thức của cặp dấu ngoặc trong cùng cũng sẽ được tính đầu tiên. Khi viết biểu thức dài có tham gia nhiều phép toán, nói chung ta nên sử dụng cặp dấu ngoặc tròn để quy định thứ tự tính toán theo ý muốn.

Dưới đây là bảng quy định độ ưu tiên của các phép toán theo thứ tự từ cao xuống thấp

Thứ tự	Kí hiệu phép toán	Tên gọi	Hướng tính toán
1	::	Phạm vi	Trái
2	() [] -> . sizeof	Lấy thành phần, kích thước	Trái
3	++ --	Tự tăng/giảm	Trái
4		Đảo bit	Trái
5	!	Toán tử logic (phủ định)	Trái
6	& *	Toán tử con trỏ	Trái
7	(type)	Chuyển đổi kiểu	Trái
8	+ -	Đổi dấu	Trái
9	* / %	Toán tử số học (lớp nhân)	Trái
10	+ -	Toán tử số học (lớp cộng)	Trái
11	<< >>	Dịch bit	Trái
12	< <= > >=	Toán tử quan hệ	Trái
13	== !=	Toán tử quan hệ	Trái
14	& ↑	Toán tử thao tác bit	Trái
15	&&	Toán tử logic (và, hoặc)	Trái
16	?:	Toán tử điều kiện	Phải
17	= += -= *= /= %= >>= <<= &= ≐ =	Toán tử gán	Phải
18	,	Dấu phẩy	Trái

Phụ lục C

Phong cách lập trình

Mỗi lập trình viên có kinh nghiệm đều có phong cách lập trình riêng. Mỗi công ty phần mềm đều có hướng dẫn về phong cách lập trình của công ty cho lập trình viên của mình. Vì vậy, trong thời gian mới học, người học nên tập tuân thủ một phong cách lập trình nào đó để rèn tính kỷ luật cũng như làm cho các đoạn mã của mình trong sáng, dễ đọc, dễ bảo trì và có tính gợi nhớ về sau này. Chúng tôi giới thiệu dưới đây một số hướng dẫn về phong cách lập trình mà các tác giả của cuốn giáo trình này thấy cần thiết nhất đối với những người có ý định học tập chuyên sâu về lập trình trong ngôn ngữ C++.

Một số hướng dẫn chính. Dù theo phong cách lập trình nào, lập trình viên nên

- Viết mã dành cho người đọc mã chứ không phải cho người viết mã.
- Cố gắng dùng phong cách lập trình của đoạn mã sẵn có nếu phát triển thêm.
- Cố gắng dùng phong cách lập trình được nhiều lập trình viên khác dùng.
- Tránh sử dụng các cấu trúc quá lạ.
- Tránh sử dụng các cấu trúc mà một lập trình viên bình thường cảm thấy khó hiểu.
- Có thể không tuân thủ phong cách lập trình nếu cần tối ưu hóa đoạn mã.

Đặt tên.

- Khi đặt tên, dùng tên có đầy đủ ý nghĩa, tránh viết tắt. Chỉ sử dụng viết tắt khi từ đó đã quá quen thuộc với tất cả mọi người.
- Đặt tên file bằng chữ thường, có thể dùng dấu gạch dưới `_`.
- Đặt tên kiểu bằng cách viết hoa các chữ đầu từ. Ví dụ: **Student**, **Address**, **StringBuilder**.
- Đặt tên biến toàn bộ bằng chữ thường với dấu gạch dưới `_` ngăn cách các từ.
- Đặt tên hàm bằng cách viết hoa các chữ đầu từ. Trường hợp các hàm “nhỏ” có thể dùng chữ thường cùng với gạch dưới `_`.
- Đặt tên macro bằng cách viết hoa tất cả các ký tự.

Chú thích.

- Chú thích file: Bắt đầu một file với đoạn chú thích về tác giả, bản quyền và nội dung chính của file.
- Chú thích lớp: Trước khi khai báo lớp, viết một đoạn chú thích về mục đích của lớp và cách sử dụng các đối tượng của lớp như cách khai báo, một số hàm chính.
- Chú thích hàm: Trước mỗi khai báo hàm, viết một đoạn chú thích mô tả mục đích, đầu vào, đầu ra của hàm cũng như cách dùng hàm đó. Trong đoạn mã định nghĩa hàm, dùng chú thích để giải thích ý đồ, các bước khó hoặc lắt léo của thuật toán.
- Chú thích cho biến thành viên: Trước mỗi khai báo biến thành viên, viết một đoạn chú thích mô tả mục đích sử dụng của biến, các giá trị đặc biệt (có ý nghĩa) của biến đó.
- Chú thích cho dòng lệnh: tất cả các đoạn mã không rõ nghĩa cần được giải thích bằng chú thích trên dòng tương ứng.
- Sử dụng chú thích `// TODO` cho các đoạn mã đang chờ được viết.

Trình bày.

- Thụt dòng mỗi khối lệnh bằng 2 hoặc 4 kí tự trắng, các lệnh trong cùng khối viết thẳng hàng.
- Mỗi dòng dài không quá 80 kí tự.
- Các lệnh `if-else` liên nhau có thể viết như sau

```
if (<điều kiện>) { // không có khoảng trắng quanh điều kiện
    ... // thụt dòng 4 kí tự trắng
} else if (<điều kiện>) { // từ khóa else viết cùng dòng với dấu đóng ngoặc }
    ...
} else {
    ...
}
```

- Một số lệnh `if` ngắn có thể viết trên cùng dòng.
- Khi biểu thức logic quá dài, có thể xuống dòng sau các toán tử logic `&&` hoặc `||`.
- Sử dụng dòng trắng để làm rõ các đoạn mã hoặc tách các hàm rời ra cho dễ đọc.

Thứ tự `#include`. Để tránh các lỗi `#include`, sử dụng thứ tự sau khi khai báo thư viện

1. File tiêu đề tương ứng. Ví dụ, file `test.cpp` sẽ `#include "test.h"` đầu tiên.
2. File tiêu đề của C. Ví dụ, `#include <stdio>`, `#include <stdlib>`.
3. File tiêu đề của C++. Ví dụ, `#include <iostream>`, `#include <algorithm>`.
4. File tiêu đề của các thư viện khác.
5. File tiêu đề của dự án.

Biến cục bộ. Khi khai báo biến cục bộ, ngay lập tức hãy gán một giá trị khởi tạo cho nó. Ví dụ, không nên khai báo như sau

```
int i;  
i = some_function();
```

mà nên sử dụng lệnh gán (khởi tạo) ngay khi khai báo như sau

```
int i = some_function();
```

Biến toàn cục, biến tĩnh. Nói chung nên tránh sử dụng các loại biến này. Thay vào đó, thiết kế các hàm với danh sách tham đối phù hợp để trao đổi dữ liệu. Khi cần khai báo các biến tĩnh hoặc biến toàn cục, chỉ nên sử dụng các kiểu cơ bản như số nguyên, số thực, ... Tránh khai báo các biến loại này ở dạng đối tượng vì thứ tự gọi các hàm khởi tạo và hàm hủy của các biến này không xác định.

Hàm. Một số lưu ý khi viết hàm:

- Trong danh sách tham đối, khai báo tham đối đầu vào trước, tham đối đầu ra sau.
- Viết hàm ngắn, gọn trong 1 trang màn hình (40 dòng). Nếu hàm quá dài, tìm cách chia hàm thành các hàm nhỏ hơn.
- Sử dụng từ khóa **const** bất cứ khi nào có thể với hàm, tham đối của hàm.

Sử dụng macro. Hết sức cẩn thận khi sử dụng macro vì macro có tác dụng khắp nơi.

- Khi cần lặp lại một đoạn mã, hãy sử dụng hàm **inline**.
- Khi cần hằng số, hãy sử dụng **const**.
- Không định nghĩa macro trong file tiêu đề.

Phụ lục D

Hàm inline

Khi định nghĩa hàm thành viên là ngắn, chúng ta có thể định nghĩa hàm bên trong định nghĩa lớp. Chúng ta đơn giản thay thế việc mô tả hàm thành viên bên trong lớp bằng việc định nghĩa hàm thành viên. Tuy nhiên vì định nghĩa bên trong lớp nên không cần bao gồm tên lớp và toán tử truy cập phạm vi. Ví dụ, lớp `Pair` định nghĩa dưới đây có định nghĩa hàm `inline` cho 2 hàm khởi tạo và hàm thành viên `get_first` (khi định nghĩa bên trong lớp thì các hàm này tự động là `inline`, nếu định nghĩa bên ngoài lớp hoặc hàm riêng thì ta phải mô tả có từ khóa `inline` bắt đầu ngay khi khai báo hàm):

```
class Pair
{
public:
    Pair() {}
    Pair(char first_value, char second_value)
    : first(first_value), second(second_value) {}
    char get_first()
    {
        return first;
    }
    ...
private:
    char first;
    char second;
};
```

Chú ý ở đây không có dấu phẩy sau dấu ngoặc đóng trong định nghĩa hàm `inline`. Hàm `inline` sẽ được thực thi theo cách khác bởi trình biên dịch vì vậy chúng chạy hiệu quả hơn mặc dù chúng tiêu tốn nhiều dung lượng hơn. Với hàm `inline`, mỗi lời gọi hàm trong chương trình được thay bằng một phiên bản biên dịch của định nghĩa hàm này, vì vậy lời gọi tới hàm `inline` không mất thời gian như lời gọi hàm bình thường.

Tài liệu tham khảo

- [1] Lê Anh Cường and Phạm Bảo Sơn. *Lập trình căn bản với Java*. Nhà xuất bản ĐHQGHN, 2012.
- [2] Lê Anh Cường, Nguyễn Văn Vinh, Lê Sỹ Vinh, and Trần Thị Minh Châu. *Bài giảng Lập trình nâng cao*. Trường Đại học Công nghệ - ĐHQGHN, 2010.
- [3] Paul J Deitel and Harvey M Deitel. *C++ how to program*. PearsonPrentice Hall, 2008.
- [4] Phạm Hồng Thái and Vũ Bá Duy. *Giáo trình thực hành Tin học cơ sở*. Nhà xuất bản ĐHQGHN, 2011.
- [5] Brian W Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Professional, 1999.
- [6] Steve McConnell. *Code complete*. Pearson Education, 2004.
- [7] Walter Savitch. *Problem Solving with C++*. Pearson Education, 2015.
- [8] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.
- [9] Bjarne Stroustrup. *Programming: principles and practice using C++*. Pearson Education, 2014.
- [10] Trần Thị Minh Châu and Nguyễn Việt Hà. *Lập trình hướng đối tượng với Java*. Trường Đại học Công nghệ - ĐHQGHN, 2013.