

GIÁO TRÌNH

LẬP TRÌNH NÂNG CAO

bản nháp, bản nháp, bản nháp,

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

bản nháp, bản nháp,
bản nháp, bản nháp,

MỤC LỤC

| | |
|--|-----------|
| 1 Mở đầu | 1 |
| 1.1 Giải quyết bài toán bằng lập trình | 1 |
| 1.1.1 Thiết kế chương trình | 1 |
| 1.1.2 Chu kỳ phát triển phần mềm | 3 |
| 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++ | 6 |
| 1.6 Chương trình C++ đầu tiên | 7 |
| 1.7 Bài tập | 8 |
| 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 cout | 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 cin | 18 |
| 2.3 Kiểu dữ liệu và biểu thức | 20 |
| 2.3.1 Kiểu int và kiểu double | 20 |
| 2.3.2 Các kiểu số khác | 22 |
| 2.3.3 Kiểu C++11 | 23 |
| 2.3.4 Kiểu char | 23 |
| 2.3.5 Tương thích kiểu dữ liệu | 25 |
| 2.3.6 Toán tử số học và biểu thức | 26 |
| 2.4 Luồng điều khiển | 27 |
| 2.5 Phong cách lập trình | 31 |

| | |
|---|-----------|
| 2.5.1 Biên dịch chương trình với GNU/C++ | 31 |
| 3 Kiểm thử và gỡ rối chương trình | 35 |
| 3.1 Gỡ rối chương trình | 35 |
| 3.2 Kiểm thử chương trình | 35 |
| 3.2.1 Kiểm tra đơn vị chương trình | 35 |
| 3.2.2 Kiểm tra bậc cao hơn | 35 |
| 3.2.3 Sử dụng thư viện kiểm tra đơn vị chương trình | 35 |
| 4 Hàm | 37 |
| 4.1 Thiết kế từ trên xuống (top-down) | 37 |
| 4.2 Hàm | 38 |
| 4.2.1 Ý nghĩa của hàm | 38 |
| 4.2.2 Cấu trúc chung của hàm | 38 |
| 4.2.3 Khai báo hàm | 41 |
| 4.3 Cách sử dụng hàm | 43 |
| 4.3.1 Lời gọi hàm | 43 |
| 4.3.2 Hàm với đối mặc định | 44 |
| 4.4 Biến toàn cục và biến địa phương | 45 |
| 4.4.1 Biến địa phương (biến trong hàm, trong khối lệnh) | 45 |
| 4.4.2 Biến toàn cục (biến ngoài tất cả các hàm) | 46 |
| 4.4.3 Mức ưu tiên của biến toàn cục và địa phương | 47 |
| 4.5 Tham đổi và cơ chế truyền giá trị cho tham đổi | 50 |
| 4.5.1 Truyền theo tham trị | 50 |
| 4.5.2 Biến tham chiếu | 52 |
| 4.5.3 Truyền theo tham chiếu | 53 |
| 4.5.4 Hai cách truyền giá trị cho hàm và từ khóa const | 54 |
| 4.6 Chồng hàm và khuôn mẫu hàm | 55 |
| 4.6.1 Chồng hàm (hàm trùng tên) | 55 |
| 4.6.2 Khuôn mẫu hàm | 57 |
| 4.7 Lập trình với hàm đệ quy | 59 |
| 4.7.1 Khái niệm đệ qui | 59 |
| 4.7.2 Lớp các bài toán giải được bằng đệ qui | 61 |
| 4.7.3 Cấu trúc chung của hàm đệ qui | 62 |
| 4.8 Bài tập | 67 |
| 5 Mảng | 71 |
| 5.1 Lập trình và thao tác với mảng một chiều | 71 |
| 5.1.1 Ý nghĩa của mảng | 71 |

| | | |
|----------|--|------------|
| 5.1.2 | Thao tác với mảng một chiều | 72 |
| 5.1.3 | Mảng và hàm | 76 |
| 5.1.4 | Tìm kiếm và sắp xếp | 81 |
| 5.2 | Lập trình và thao tác với mảng nhiều chiều | 86 |
| 5.2.1 | Mảng 2 chiều | 86 |
| 5.2.2 | Thao tác với mảng hai chiều | 87 |
| 5.3 | Lập trình và thao tác với xâu kí tự | 94 |
| 5.3.1 | Khai báo | 94 |
| 5.3.2 | Thao tác với xâu kí tự | 95 |
| 5.3.3 | Phương thức nhập xâu (#include <iostream>) | 96 |
| 5.3.4 | Một số hàm làm việc với xâu kí tự (#include <cstring>) | 96 |
| 5.3.5 | Các hàm chuyển đổi xâu dạng số thành số (#include <cstdlib>) | 100 |
| 5.3.6 | Một số ví dụ làm việc với xâu | 102 |
| 5.4 | Bài tập | 105 |
| 6 | Các kiểu dữ liệu trừu tượng | 109 |
| 6.1 | Kiểu dữ liệu trừu tượng bằng cấu trúc (struct) | 109 |
| 6.1.1 | Khai báo, khởi tạo | 109 |
| 6.1.2 | Hàm và cấu trúc | 112 |
| 6.1.3 | Bài toán Quản lý sinh viên (QLSV) | 117 |
| 6.2 | Kiểu dữ liệu trừu tượng bằng lớp (class) | 125 |
| 6.2.1 | Khai báo lớp | 125 |
| 6.2.2 | Sử dụng lớp | 127 |
| 6.2.3 | Bài toán Quản lý sinh viên | 133 |
| 6.2.4 | Khởi tạo (giá trị ban đầu) cho một đối tượng | 137 |
| 6.2.5 | Hủy đối tượng | 142 |
| 6.2.6 | Hàm bạn (friend function) | 143 |
| 6.2.7 | Tạo các phép toán cho lớp (hay tạo chồng phép toán - Operator Overloading) | 147 |
| 6.3 | Dạng khuôn mẫu hàm và lớp | 150 |
| 6.3.1 | Khai báo một kiểu mẫu | 150 |
| 6.3.2 | Sử dụng kiểu mẫu | 150 |
| 6.3.3 | Một số dạng mở rộng của khai báo mẫu | 152 |
| 6.4 | Bài tập | 152 |
| 7 | Con trỏ và bộ nhớ | 157 |
| 7.1 | Quản lý bộ nhớ máy tính | 157 |
| 7.2 | Biến và địa chỉ của biến | 158 |
| 7.3 | Biến con trỏ | 158 |
| 7.4 | Mảng và con trỏ | 162 |

| | | |
|-----------|--|------------|
| 7.5 | Bộ nhớ động | 163 |
| 7.5.1 | Cấp phát bộ nhớ động | 163 |
| 7.5.2 | Giải phóng bộ nhớ động | 164 |
| 7.6 | Mảng động và con trỏ | 165 |
| 7.7 | Truyền tham số là con trỏ | 166 |
| 7.8 | Con trỏ hàm | 169 |
| 7.9 | Lập trình với danh sách liên kết | 171 |
| 7.9.1 | Nút và danh sách liên kết | 171 |
| 7.9.2 | Các thao tác với danh sách liên kết | 175 |
| 7.9.3 | Danh sách liên kết của lớp | 183 |
| 8 | Vào ra dữ liệu | 189 |
| 8.1 | Khái niệm dòng dữ liệu | 189 |
| 8.1.1 | Tệp văn bản và tệp nhị phân | 190 |
| 8.2 | Vào ra tệp | 190 |
| 8.2.1 | Mở tệp | 191 |
| 8.2.2 | Đóng tệp | 192 |
| 8.3 | Vào ra tệp văn bản và nhị phân | 192 |
| 8.3.1 | Vào ra tệp văn bản | 192 |
| 8.3.2 | Vào ra tệp nhị phân | 194 |
| 9 | Xử lý ngoại lệ | 199 |
| 9.1 | Các vấn đề cơ bản trong xử lý ngoại lệ | 199 |
| 9.1.1 | Ví dụ xử lý ngoại lệ | 199 |
| 9.1.2 | Định nghĩa lớp ngoại lệ | 201 |
| 9.1.3 | Ném và bắt nhiều ngoại lệ | 202 |
| 9.1.4 | Ném ngoại lệ từ hàm | 203 |
| 9.1.5 | Mô tả ngoại lệ | 205 |
| 9.2 | Kỹ thuật lập trình cho xử lý ngoại lệ | 205 |
| 9.2.1 | Ném ngoại lệ ở đâu | 205 |
| 9.2.2 | Cây phả hệ ngoại lệ STL | 206 |
| 9.2.3 | Kiểm tra bộ nhớ | 207 |
| 9.3 | Bài tập | 207 |
| 10 | Tiền xử lý và lập trình nhiều file | 209 |
| 10.1 | Các chỉ thị tiền xử lý | 209 |
| 10.1.1 | Chỉ thị bao hàm tệp <code>#include</code> | 209 |
| 10.1.2 | Chỉ thị macro <code>#define</code> | 210 |
| 10.1.3 | Các chỉ thị biên dịch có điều kiện <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> | 211 |

| | | |
|-----------|---|------------|
| 10.2 | Lập trình trên nhiều file | 213 |
| 10.2.1 | Tổ chức chương trình | 213 |
| 10.2.2 | Viết và kiểm tra các file include | 214 |
| 10.2.3 | Biên dịch chương trình có nhiều file | 214 |
| 10.3 | Bài tập | 215 |
| 11 | Lập trình với thư viện chuẩn STL | 217 |
| 11.1 | Giới thiệu thư viện chuẩn STL | 217 |
| 11.2 | Khái niệm con trỏ duyệt | 217 |
| 11.2.1 | Các thao tác cơ bản với con trỏ duyệt | 218 |
| 11.2.2 | Các loại con trỏ duyệt | 220 |
| 11.3 | Khái niệm vật chứa | 222 |
| 11.3.1 | Các vật chứa dạng dãy | 223 |
| 11.3.2 | Ngăn xếp và hàng đợi | 228 |
| 11.3.3 | Tập hợp và ánh xạ | 231 |
| 11.3.4 | Hàm băm, tập hợp và ánh xạ không thứ tự (C++11) | 234 |
| 11.4 | Các thuật toán mỗ | 236 |
| 11.4.1 | Thời gian chạy và ký hiệu “O-lớn” | 236 |
| 11.4.2 | Các thuật toán không thay đổi vật chứa | 237 |
| 11.4.3 | Các thuật toán thay đổi vật chứa | 240 |
| 11.4.4 | Các thuật toán tập hợp | 242 |
| 11.5 | Bài tập | 242 |

bản nháp, bản nháp,
bản nháp, bản nháp,

Chương 1

Mở đầu

Trong chương này, chúng tôi sẽ giới thiệu qua một số khái niệm cơ bản về: các kỹ thuật thiết kế và viết chương trình, lập trình, ngôn ngữ lập trình và 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++.

Bạn chắc chắn đã dùng qua nhiều chương trình khác nhau, ví dụ như chương trình soạn thảo văn bản “Microsoft Word”. Chương trình, hay phần mềm, được hiểu đơn giản là một tập các lệnh để máy tính thực hiện theo. Khi bạn đưa cho máy tính một chương trình và yêu cầu máy tính thực hiện theo các lệnh của chương trình, bạn đang chạy chương trình đó.

1.1.1 Thiết kế chương trình

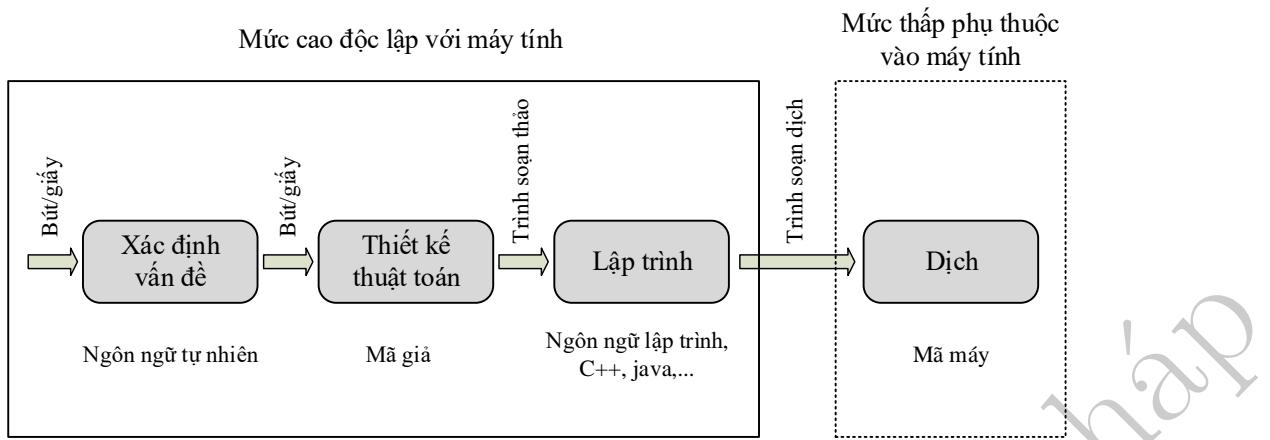
Thiết kế chương trình thường có 2 pha là pha giải quyết bài toán và pha thực hiện lập trình để xây dựng chương trình cho bài toán đó. Để tạo ra chương trình trong ngôn ngữ lập trình như C++, thuật toán được chuyển đổi biểu diễn trong ngôn ngữ lập trình C++. Lập trình là có thể hiểu đơn giản là quá trình viết ra các lệnh hướng dẫn máy tính thực hiện để giải quyết một bài toán cụ thể nào đó. Lập trình là một bước quan trọng trong quy trình thiết kế chương trình giải quyết một bài toán như mô tả ở Hình 1.1.

Quy trình trên có thể được chia ra thành hai mức: mức cao độc lập với máy tính (machine independent) và mức thấp phụ thuộc vào máy tính (machine specific).

Mức cao độc lập với máy tính.

Mức cao độc lập với máy tính thường được chia thành ba bước chính là: xác định vấn đề, thiết kế thuật toán và lập trình.

- Xác định vấn đề:** Bước này định nghĩa bài toán, xác định dữ liệu đầu vào, các ràng buộc, yêu cầu cần giải quyết và kết quả đầu ra. Bước này thường sử dụng bút/giấy và ngôn ngữ tự nhiên như tiếng Anh, tiếng Việt để mô tả và xác định vấn đề cần giải quyết.



Hình 1.1: Quy trình thiết kế chương trình giải quyết một bài toán

- **Thiết kế thuật toán:** Một thuật toán là một dãy các chỉ dẫn đúng nhằm giải quyết một bài toán. Các chỉ dẫn này cần được diễn đạt một cách hoàn chỉnh và chính xác sao cho mọi người có thể hiểu và tiến hành theo. Thuật toán thường được mô tả dưới dạng mã giả (pseudocode) hoặc bằng ngôn ngữ lập trình. Bước này có thể sử dụng giấy bút và thường không phụ thuộc vào ngôn ngữ lập trình. Ví dụ về thuật toán “tìm ước số chung lớn nhất (UCLN) của hai số x và y ” viết bằng ngôn ngữ tự nhiên:

- Bước 1: Nếu $x > y$ thì thay x bằng phần dư của phép chia x/y .
 - Bước 2: Nếu không, thay y bằng phần dư của phép chia y/x .
 - Bước 3: Nếu trong hai số x và y có một số bằng 0 thì kết luận UCLN là số còn lại.
 - Bước 4: Nếu không, quay lại Bước 1.

hoặc bằng mã giả:

```

repeat
    if x > y then x := x mod y
    else y := y mod x
until x = 0 or y = 0
if x = 0 then UCLN := y
else UCLN := x

```

- **Lập trình:** Bước chuyển đổi thuật toán sang một ngôn ngữ lập trình, phổ biến là các ngôn ngữ lập trình bậc cao, ví dụ như các ngôn ngữ C++, Java, Python. Bước này, lập trình viên sử dụng một chương trình soạn thảo văn bản để viết chương trình. Trong và sau quá trình lập trình, người ta phải tiến hành kiểm thử và sửa lỗi chương trình. Có ba loại lỗi thường gặp: lỗi cú pháp, lỗi trong thời gian chạy, và lỗi logic (xem chi tiết trong chương 3).

Mức thấp phụ thuộc vào máy tính Các ngôn ngữ lập trình bậc cao, ví dụ như C, C++, Java, Visual Basic, C#, được thiết kế để con người tương đối dễ hiểu và dễ sử dụng. Tuy nhiên, máy tính không hiểu được các ngôn ngữ bậc cao. Do đó, trước khi một chương trình viết bằng ngôn ngữ bậc cao có thể chạy được, nó phải được dịch sang ngôn ngữ máy, hay còn gọi là mã máy, mà máy tính có thể hiểu và thực hiện được. Việc dịch đó được thực hiện bởi một chương trình máy tính gọi là chương trình dịch.

1.1.2 Chu kỳ phát triển phần mềm

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

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

Như chúng ta thấy, quá trình giải quyết một bài toán thông qua các bước khác nhau để chuyển đổi từ ngôn ngữ tự nhiên mà con người hiểu được sang ngôn ngữ máy mà máy tính có thể hiểu và thực hiện được. Ngôn ngữ lập trình thường được chia ra thành hai loại: ngôn ngữ lập trình bậc thấp và ngôn ngữ lập trình bậc cao.

Ngôn ngữ lập trình bậc thấp như hợp ngữ (assembly language) hoặc mã máy là ngôn ngữ gần với ngôn ngữ máy mà máy tính có thể hiểu được. Đặc điểm chính của các ngôn ngữ này là chúng có liên quan chặt chẽ đến phần cứng của máy tính. Các họ máy tính khác nhau sử dụng các ngôn ngữ khác nhau. Chương trình viết bằng các ngôn ngữ này có thể chạy mà không cần qua chương trình dịch. Các ngôn ngữ bậc thấp có thể dùng để viết những chương trình cần tối ưu hóa về tốc độ. Tuy nhiên, chúng thường khó hiểu đối với con người và không thuận tiện cho việc lập trình.

Ngôn ngữ lập trình bậc cao như Pascal, Ada, C, C++, Java, Visual Basic, Python, ... là các ngôn ngữ có mức độ trừu tượng hóa cao, gần với ngôn ngữ tự nhiên của con người hơn. Việc sử dụng các ngôn ngữ này cho việc lập trình do đó dễ dàng hơn và nhanh hơn rất nhiều so với ngôn ngữ lập trình bậc thấp. Khác với ngôn ngữ bậc thấp, chương trình viết bằng các ngôn ngữ bậc cao nói chung có thể sử dụng được trên nhiều loại máy tính khác nhau.

Các chương trình viết bằng một ngôn ngữ bậc cao muốn chạy được thì phải được dịch sang ngôn ngữ máy bằng cách sử dụng chương trình dịch. Chương trình dịch có thể chia ra thành hai loại là trình biên dịch và trình thông dịch.

Một số ngôn ngữ bậc cao như C, C++ yêu cầu loại chương trình dịch được gọi là **trình biên dịch** (*compiler*). Trình biên dịch dịch mã nguồn thành mã máy – dạng có thể thực thi được. Kết quả của việc dịch là một chương trình thực thi được và có thể chạy nhiều lần mà không cần dịch lại. Ví dụ, với ngôn ngữ C++ một trình biên dịch rất phổ biến là gcc/g++ trong bộ GNU Compiler Collection (GCC) chạy trong các môi trường Unix/Linux cũng như Windows. Ngoài ra, Microsoft Visual C++ là trình biên dịch C++ phổ biến nhất trong môi trường Windows. Một số ngôn ngữ bậc cao khác như Perl, Python yêu cầu loại chương trình dịch gọi là **trình thông dịch** (*interpreter*). Khác với trình biên dịch, thay vì dịch toàn bộ chương trình một lần, trình thông dịch vừa dịch vừa chạy chương trình, dịch đến đâu chạy chương trình đến đó.

Trong môn học này, C++ được chọn làm ngôn ngữ thể hiện. Đây là một trong những ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi nhất trên thế giới. Trong phạm vi nhập môn của môn học này, C++ chỉ được giới thiệu ở mức rất cơ bản, rất nhiều tính năng mạnh của C++ sẽ không được nói đến hoặc chỉ được giới thiệu sơ qua. Người học nên tiếp tục tìm hiểu về ngôn ngữ C++, vượt ra ngoài giới hạn của cuốn sách này.

1.4 Môi trường lập trình bậc cao

Để lập trình giải quyết một bài toán bằng ngôn ngữ lập trình bậc cao, bạn cần có công cụ chính là: chương trình soạn thảo, chương trình dịch dành cho ngôn ngữ sử dụng, và các thư viện chuẩn của ngôn ngữ sử dụng (*standard library*), và chương trình tìm lỗi (*debugger*).

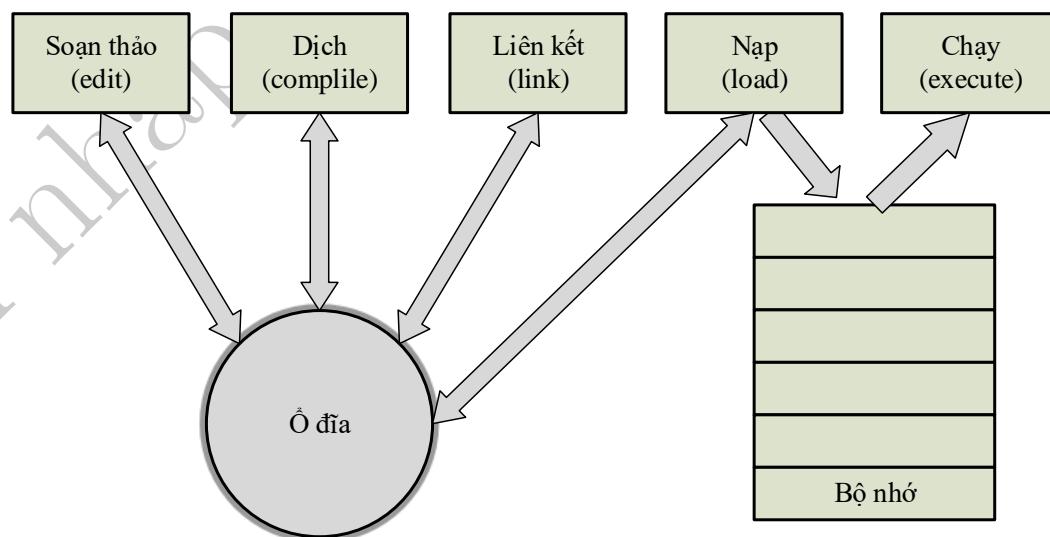
Các bước cơ bản để xây dựng và thực hiện một chương trình:

- **Soạn thảo:** Mã nguồn chương trình được viết bằng một phần mềm soạn thảo văn bản dạng

text và lưu trên ổ đĩa. Ta có thể dùng những phần mềm soạn thảo văn bản đơn giản nhất như Notepad (trong môi trường Windows) hay vi (trong môi trường Unix/Linux), hoặc các công cụ soạn thảo trong môi trường tích hợp để viết mã nguồn chương trình. Mã nguồn C++ thường đặt trong các tệp với tên có phần mở rộng là `.cpp`, `cxx`, `.cc`, hoặc `.C` (viết hoa).

- **Dịch:** Dùng trình biên dịch dịch mã nguồn chương trình ra thành các đoạn mã máy riêng lẻ (gọi là “object code”) lưu trên ổ đĩa. Các trình biên dịch phổ biến cho C++ là `vc.exe` trong bộ Microsoft Visual Studio hay `gcc` trong bộ GNU Compiler với các tham số thích hợp để dịch và liên kết để tạo ra tệp chạy được. Với C++, ngay trước khi dịch còn có giai đoạn tiền xử lý (*preprocessing*) khi các định hướng tiền xử lý được thực thi để làm các thao tác như bổ sung các tệp văn bản cần dịch hay thay thế một số chuỗi văn bản. Một số định hướng tiền xử lý quan trọng sẽ được giới thiệu dần trong cuốn sách này.
- **Liên kết:** Một tệp mã nguồn thường không chứa đầy đủ những phần cần thiết cho một chương trình hoàn chỉnh. Nó thường dùng đến dữ liệu hoặc hàm được định nghĩa trong các tệp khác hoặc trong thư viện chuẩn. Trình liên kết (*linker*) kết nối các đoạn mã máy riêng lẻ với nhau và với các thư viện có sẵn để tạo ra một chương trình mã máy hoàn chỉnh chạy được.
- **Nạp:** Trình nạp (*loader*) sẽ nạp chương trình dưới dạng mã máy vào bộ nhớ. Các thành phần bổ sung từ thư viện cũng được nạp vào bộ nhớ.
- **Chạy:** CPU nhận và thực hiện lần lượt các lệnh của chương trình, dữ liệu và kết quả thường được ghi ra màn hình hoặc ổ đĩa.

Thường thì không phải chương trình nào cũng chạy được và chạy đúng ngay ở lần chạy thử đầu tiên. Chương trình có thể có lỗi cú pháp nên không qua được bước dịch, hoặc chương trình dịch được nhưng gặp lỗi trong khi chạy. Trong những trường hợp đó, lập trình viên phải quay lại bước soạn thảo để sửa lỗi và thực hiện lại các bước sau đó.



Hình 1.2: Các bước cơ bản để xây dựng một chương trình.

Để thuận tiện cho việc lập trình, các công cụ soạn thảo, dịch, liên kết, chạy... nói trên được kết hợp lại trong một môi trường lập trình tích hợp (*IDE – integrated development environment*), trong

đó, tất cả các công đoạn đối với người dùng chỉ còn là việc chạy các tính năng trong một phần mềm duy nhất. IDE rất hữu ích cho các lập trình viên. Tuy nhiên, đối với những người mới học lập trình, thời gian đầu nên tự thực hiện các bước dịch và chạy chương trình thay vì thông qua các chức năng của IDE. Như vậy, người học sẽ có thể nắm được bản chất các bước của quá trình xây dựng chương trình, hiểu được bản chất và đặc điểm chung của các IDE, tránh tình trạng bị phụ thuộc vào một IDE cụ thể.

Ví dụ về các IDE phổ biến là Microsoft Visual Studio – môi trường lập trình thương mại cho môi trường Windows, và Eclipse – phần mềm miễn phí với các phiên bản cho cả môi trường Windows cũng như Unix/Linux, cả hai đều hỗ trợ nhiều ngôn ngữ lập trình.

Dành cho C++, một số môi trường lập trình tích hợp phổ biến là Microsoft Visual Studio, Dev-C++, Code::Blocks, KDevelop. Mỗi môi trường có thể hỗ trợ một hoặc nhiều trình biên dịch. Chẳng hạn Code::Blocks hỗ trợ cả GCC và MSVC Do C++ có các phiên bản khác nhau.

Có những bản cài đặt khác nhau của C++. Các bản ra đời trước chuẩn C++ 1998 (ISO/IEC 14882) có thể không hỗ trợ đầy đủ các tính năng được đặc tả trong chuẩn ANSI/ISO 1998. Bản C++ do Microsoft phát triển khác với bản C++ của GNU. Tuy nhiên, các trình biên dịch hiện đại hầu hết hỗ trợ C++ chuẩn, ta cũng nên chọn dùng các phần mềm này. Ngôn ngữ C++ được dùng trong cuốn sách này tuân theo chuẩn ISO/IEC 14882, còn gọi là “C++ thuần túy” (pure C++).

1.5 Lịch sử C và C++

Ngôn ngữ lập trình C được tạo ra bởi Dennis Ritchie (phòng thí nghiệm Bell) và được sử dụng để phát triển hệ điều hành UNIX. Một trong những đặc điểm nổi bật của C là độc lập với phần cứng (portable), tức là chương trình có thể chạy trên các loại máy tính và các hệ điều hành khác nhau. Năm 1983, ngôn ngữ C đã được chuẩn hóa và được gọi là ANSI C bởi Viện chuẩn hóa quốc gia Hoa Kỳ (American National Standards Institute). Hiện nay ANSI C vẫn là ngôn ngữ lập trình chuyên nghiệp và được sử dụng rộng rãi để phát triển các hệ thống tính toán hiệu năng cao.

Ngôn ngữ lập trình C++ do Bjarne Stroustrup (thuộc phòng thí nghiệm Bell) phát triển trên nền là ngôn ngữ lập trình C và cảm hứng chính từ ngôn ngữ lập trình Simula67. So với C, C++ là ngôn ngữ an toàn hơn, khả năng diễn đạt cao hơn, và ít đòi hỏi các kỹ thuật bậc thấp. Ngoài những thế mạnh thừa kế từ C, C++ hỗ trợ trừu tượng hóa dữ liệu, lập trình hướng đối tượng và lập trình tổng quát, C++ giúp xây dựng dễ dàng hơn những hệ thống lớn và phức tạp.

Bắt đầu từ phiên bản đầu tiên năm 1979 với cái tên “C with Classes” (C kèm lớp đối tượng) với các tính năng cơ bản của lập trình hướng đối tượng, C++ được phát triển dần theo thời gian. Năm 1983, cái tên ”C++” chính thức ra đời, các tính năng như hàm ảo (virtual function), hàm trùng tên và định nghĩa lại toán tử (overloading), hằng ... được bổ sung. Năm 1989, C++ có thêm lớp trừu tượng, đa thừa kế, hàm thành viên tĩnh, hằng hàm, và thành viên kiểu protected. Các bổ sung cho C++ trong thập kỷ sau đó là khuôn mẫu (template), không gian tên (namespace), ngoại lệ (exception), các toán tử đổi kiểu dữ liệu mới, và kiểu dữ liệu Boolean. Năm 1998, lần đầu tiên C++ được chính thức chuẩn hóa quốc tế bởi tổ chức ISO, kết quả là chuẩn ISO/IEC 14882 .

Đi kèm với sự phát triển của ngôn ngữ là sự phát triển của thư viện chuẩn C++. Bên cạnh việc tích hợp thư viện chuẩn truyền thống của C với các sửa đổi nhỏ cho phù hợp với C++, thư viện chuẩn C++ còn có thêm thư viện stream I/O phục vụ việc vào ra dữ liệu dạng dòng. Chuẩn C++ năm 1998 tích hợp thêm phần lớn thư viện STL (Standard Template Library – thư viện khuôn mẫu

chuẩn). Phần này cung cấp các cấu trúc dữ liệu rất hữu ích như vector, danh sách, và các thuật toán như sắp xếp và tìm kiếm.

Hiện nay, C++ là một trong các ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi nhất.

1.6 Chương trình C++ đầu tiên

Chương trình đơn giản trong Hình 1.3 sẽ hiện ra màn hình dòng chữ “Hello world!”. Trong chương trình có những đặc điểm quan trọng của C++. Ta sẽ xem xét từng dòng.

```

1 // The first program in C++
2 // Print "Hello world!" to the screen
3 #include <iostream>
4
5 using namespace std;
6
7 int main()
8 {
9     cout << "Hello world!";
10    return 0;
11 }
```

Hình 1.3: Chương trình C++ đầu tiên.

Hai dòng đầu tiên bắt đầu bằng chuỗi `//` là các dòng chú thích chương trình. Đó là kiểu chú thích dòng đơn. Các dòng chú thích không gây ra hoạt động gì của chương trình khi chạy, trình biên dịch bỏ qua các dòng này. Ngoài ra còn có dạng chú thích kiểu C dùng chuỗi `/*` và `*/` để đánh dấu điểm bắt đầu và kết thúc chú thích. Các lập trình viên dùng chú thích để giải thích và giới thiệu về nội dung chương trình.

Dòng thứ ba, `#include <iostream>` là một định hướng tiền xử lý (preprocessor directive) – chỉ dẫn về một công việc mà trình biên dịch cần thực hiện trước khi dịch chương trình. `#include` là khai báo về thư viện sẽ được sử dụng trong chương trình, trong trường hợp này là thư viện vào ra dữ liệu iostream trong thư viện chuẩn C++.

Tiếp theo là hàm `main()`, phần không thể thiếu của mỗi chương trình C++. Nó bắt đầu từ dòng khai báo header của hàm:

```
int main()
```

Mỗi chương trình C++ thường bao gồm một hoặc nhiều hàm, trong đó có đúng một hàm có tên `main`, đây là nơi chương trình bắt đầu thực hiện và kết thúc. Bên trái từ `main` là từ khóa `int`, nó có nghĩa là hàm `main` sẽ trả về một giá trị là số nguyên. Từ khóa là những từ đặc biệt mà C++ dành riêng cho những mục đích cụ thể. Chương 4 sẽ cung cấp thông tin chi tiết về khái niệm hàm và việc hàm trả về giá trị.

Thân hàm `main` được bắt đầu và kết thúc bởi cặp ngoặc , bên trong đó là chuỗi các lệnh mà khi chương trình chạy chúng sẽ được thực hiện tuần tự từ lệnh đầu tiên cho đến lệnh cuối cùng. Hàm `main` trong ví dụ đang xét có chứa hai lệnh. Mỗi lệnh đều kết thúc bằng một dấu chấm phẩy, các định hướng tiền xử lý thì không.

Lệnh thứ nhất gồm `cout`, toán tử `<<`, xâu kí tự `"Hello world!"`, và dấu chấm phẩy. Nó chỉ thị cho máy tính thực hiện một nhiệm vụ: in ra màn hình chuỗi kí tự nằm giữa hai dấu nháy kép

- "**Hello world!**" . Khi lệnh được thực thi, chuỗi kí tự **Hello world** sẽ được gửi cho **cout** – luồng dữ liệu ra chuẩn của C++, thường được nối với màn hình. Chi tiết về vào ra dữ liệu sẽ được nói đến trong Chương 8. Chuỗi kí tự nằm giữa hai dấu nháy kép được gọi là một xâu kí tự (**string**). Để ý dòng

```
using namespace std;
```

nằm ở gần đầu chương trình. Tất cả thành phần của thư viện chuẩn C++, trong đó có **cout** được dùng đến trong hàm **main**, được khai báo trong một không gian tên (**namespace**) có tên là **std**. Dòng trên thông báo với trình biên dịch rằng chương trình ví dụ của ta sẽ sử dụng đến một số thành phần nằm trong không gian tên **std**. Nếu không có khai báo trên, tiền tố **std::** sẽ phải đi kèm theo tên của tất cả các thành phần của thư viện chuẩn được dùng trong chương trình, chẳng hạn **cout** sẽ phải được viết thành **std::cout**. Chi tiết về không gian tên nằm ngoài phạm vi của cuốn sách này, người đọc có thể tìm hiểu tại các tài liệu [1] hoặc [2]. Nếu không có lưu ý đặc biệt thì tất cả các chương trình ví dụ trong cuốn sách này đều sử dụng khai báo sử dụng không gian tên **std** như ở trên.

Lệnh thứ hai nhảy ra khỏi hàm và trả về giá trị **0** làm kết quả của hàm. Đây là bước có tính chất quy trình do C++ quy định hàm **main** cần trả lại một giá trị là số nguyên cho biết trạng thái kết thúc của chương trình. Giá trị **0** được trả về ở cuối hàm main có nghĩa rằng hàm đã kết thúc thành công.

Để ý rằng tất các lệnh nằm bên trong cặp ngoặc của thân hàm đều được lùi đầu dòng một mức. Với C++, việc này không có ý nghĩa về cú pháp. Tuy nhiên, nó lại giúp cho cấu trúc chương trình dễ thấy hơn và chương trình dễ hiểu hơn đối với người lập trình. Đây là một trong các điểm quan trọng trong các quy ước về phong cách lập trình. Phụ lục A sẽ hướng dẫn chi tiết hơn về các quy ước này.

Đến đây ta có thể sửa chương trình trong Hình 1.3 để in ra lời chào "Hello world!" theo các cách khác nhau. Chẳng hạn, ta có thể in ra cùng một nội dung như cũ nhưng bằng hai lệnh gọi cout:

```
cout << "Hello "; cout << "world!";
```

hoặc in ra lời chào trên nhiều dòng bằng cách chèn vào giữa xâu kí tự các kí tự xuống dòng (kí tự đặc biệt được kí hiệu là **\n**):

```
cout << "Hello \n world!\n";
```

1.7 Bài tập

1. Nhập vào từ bàn phím một danh sách sinh viên. Mỗi sinh viên gồm có các thông tin sau đây: tên tuổi, ngày tháng năm sinh, nơi sinh, quê quán, lớp, học lực (từ 0 đến 9). Hãy ghi thông tin về danh sách sinh viên đó ra tệp văn bản **student.txt**
2. Sau khi thực hiện bài 1, hãy viết chương trình nhập danh sách sinh viên từ tệp văn bản **student.txt** rồi hiển thị ra màn hình:
 - Thông tin về tất cả các bạn tên là Vinh ra tệp văn bản vinh.txt
 - Thông tin tất cả các bạn quê ở Hà Nội ra tệp văn bản hanoi.txt

- Tổng số bạn có học lực kém (**<4**), học lực trung bình (**>=4** và **<8**), học lực giỏi (**8**) ra tệp văn bản **hocluc.txt** .
3. Sau khi thực hiện bài 2, hãy viết chương trình cho biết kích thước của tệp văn bản **student.txt**, **vinh.txt**. Kết quả ghi ra tệp văn bản **all.txt**.
4. Viết chương trình kiểm tra xem tệp văn bản **student.txt** có tồn tại hay không? Nếu tồn tại thì hiện ra màn hình các thông tin sau:
- Số lượng sinh viên trong tệp
 - Số lượng dòng trong tệp
 - Ghi vào cuối tệp văn bản dòng chữ “CHECKED”
 - Nếu không tồn tại, thì hiện ra màn hình dòng chữ “NOT EXISTED”.
5. Tệp văn bản **numbers.txt** gồm nhiều dòng, mỗi dòng chứa một danh sách các số nguyên hoặc thực. Hai số đứng liền nhau cách nhau ít nhất một dấu cách. Hãy viết chương trình tổng hợp các thông tin sau và ghi vào tệp văn bản **info.txt** những thông tin sau:
- Số lượng số trong tệp
 - Số lượng các số nguyên
 - Số lượng các số thực
- Lưu ý: Test chương trình với cả trường hợp tệp văn bản **number.txt** chứa một hay nhiều dòng trắng ở cuối tệp.
6. Trình bày sự khác nhau, ưu điểm, nhược điểm giữa tệp văn bản và tệp văn bản nhị phân. Khi nào thì nên dùng tệp văn bản nhị phân.
7. Cho file văn bản **numbers.txt** chứa các số nguyên hoặc thực. Hãy viết một chương trình đọc các số từ file **numbers.txt** và ghi ra file nhị phân **numbers.bin** các số nguyên nằm trong file **numbers.txt** .
8. Sau khi thực hiện bài 7, hãy viết một chương trình đọc và tính tổng của tất cả các số nguyên ở file nhị phân **numbers.bin** . Hiện ra màn hình kết quả thu được.
9. Sau khi thực hiện bài 7, hãy viết một chương trình đọc các số nguyên ở file nhị phân **numbers.bin** . Ghi các số nằm ở vị trí chẵn (số đầu tiên trong file được tính ở vị trí số 0) trong file nhị phân **numbers.bin** vào cuối file **numbers.txt** .
10. Cho hai file văn bản **num1.txt** và **num2.txt** , mỗi file chứa 1 dãy số đã được sắp không giảm. Số lượng số trong mỗi file không quá 109. Hãy viết chương trình đọc và ghi ra file văn bản **num12.txt** các số trong hai file **num1.txt** và **num2.txt** thỏa mãn điều kiện các số trong file **num12.txt** cũng được sắp xếp không giảm.
11. File văn bản **document.txt** chứa một văn bản tiếng anh. Các câu trong văn bản được phân cách nhau bởi dấu **‘.’** hoặc **‘!!’** . Hãy ghi ra file văn bản **sentences.txt** nội dung của văn bản **document.txt** , mỗi câu được viết trên một dòng. Ví dụ:

| | |
|--|--|
| document.txt | sentences.txt |
| this is a good house! However, too expensive. | this is a good house! However, too expensive. |

12. File văn bản **document.txt** chứa một văn bản có lõi cả các câu tiếng anh và các câu tiếng Việt. Các câu trong văn bản được phân cách nhau bởi dấu ‘.’ hoặc ‘!!’. Hãy ghi ra file văn bản **english.txt** (**viet.txt**) các câu tiếng Anh (Việt) trong văn bản **document.txt**.

Chương 2

Một số khái niệm cơ bản trong C++^{nhập}

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 //Chuong trinh minh hoa
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_of_bars;
26 cin >> one_weight;
27
28 total_weight = one_weight * number_of_bars;
29
30 cout << number_of_bars << " 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_of_bars`, `one_weight`, và `total_weight` là các biến. Chương trình được chạy với đầu vào thể hiện trong các đối thoại mẫu, `number_of_bars` đã thiết lập giá trị 11 trong câu lệnh.

```
cin >> number_of_bars;
```

giá trị của biến `number_of_bars` 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 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:

| | | | | | |
|-------------------|--------------------|--------------------|------------------------|-----------------------|------------------|
| <code>x</code> | <code>x1</code> | <code>x_1</code> | <code>_abc</code> | <code>ABC123z7</code> | <code>sum</code> |
| <code>RATE</code> | <code>count</code> | <code>data2</code> | <code>Big_Bonus</code> | | |

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_of_bars;
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_of_bars` là một biến kiểu `int`. Giá trị của `number_of_bars` 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_of_bars = 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_of_bars = number_of_bars + 3;
```

Giá trị thực là “Giá trị của `number_of_bars` bằng với giá trị của `number_of_bars` cộng với ba” hay “Giá trị mới `number_of_bars` bằng với giá trị cũ của `number_of_bars` 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_of_bars << " candy bars\n";
```

Câu lệnh này cho phép máy tính xuất ra màn hình hai mục: giá trị của biến `number_of_bars` 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_of_bars;
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 \ h 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 chưa 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 xâu). 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 bổ 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 [Enter]
12 [Enter]
5 [Enter]
```

Chương trình sẽ bỏ qua các dấu `Space`, dấu `Tab`, dấu `Enter` 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 `[Enter]`

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 `Enter`) 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 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 phần 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 **double**. Các số có độ chính xác đơn gọi là **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

3670000000000000.0

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 phần 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`.

Bảng 2.2: Một số kiểu dữ liệu số.

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

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 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;
```

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 kí tự đơn giống như các chữ cái, chữ số và các kí tự chấm câu.

Giá trị của kiểu dữ liệu này gọi là các kí 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 kí tự từ bàn phím. Ví dụ, biến `symbol` có thể lưu kí tự '`A`' hoặc một kí tự '`+`'. Chú ý các kí tự hoa và kí tự thường là hoàn toàn khác nhau.

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

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

Chú ý rằng giá trị xâu kí tự đặt trong dấu nháy kép, một kí 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 xâu các kí tự. Mặc dù xâu kí tự chỉ chứa một kí tự nhưng cũng không thể biến xâu "`A`" có giá trị kiểu `char`. Chú ý, với cả xâu kí tự và kí 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 kí tự khác khoảng trống và đọc kí 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 kí tự nhập vào và trường hợp không chứa khoảng trống dưới đây.

```
1 //Chuong trinh minh hoa kieu 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 kí tự 'Z' vào biến kiểu `int`. C++ coi các kí 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 kí 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 chẵn 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 trắng. Bạn có thể thêm các khoảng trắng 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 thử 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 2.

Khi bạn sử dụng phép chia / cho hai số nguyên, kết quả cũng là một số nguyên. Số 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 đường 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 pháp 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 //Chuong trinh minh hoa cau lenh 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 //Chuong trinh minh hoa lap 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 //Chuong trinh minh họa lop 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 thực 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 thực 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 thực 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 là mỗi hằng số phải được gán một giá trị ngay khi chúng được khai báo, giá trị này là không thể thay đổi.

2.5.1 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, and Ada ... Bạn có thể download 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 lẩn 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     system("PAUSE");
14     return 0;
15 }
```

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).

Chương 3

Kiểm thử và gỡ rối chương trình

3.1 Gỡ rối chương trình

3.2 Kiểm thử chương trình

3.2.1 Kiểm tra đơn vị chương trình

3.2.2 Kiểm tra bậc cao hơn

3.2.3 Sử dụng thư viện kiểm tra đơn vị chương trình

bản nháp, bản nháp,
bản nháp, bản nháp,

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 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 lại các hàm trong một chương trình thì chương trình này sẽ giải quyết tổng thể bài toán.

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: *n nhận hoặc không, một hoặc nhiều đầu vào là các tham đổi*, 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*. Một chương trình C/C++ là tập hợp của các hàm, trong đó phải 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 módun 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. Đôi 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 trả (đối là con trả).

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 trả, 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> <name of argument>**.
- 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**.

```

1 void printTenAsterisks()
2 {
3     for (int count = 1; count <= 10; count++)
4         cout << "*";
5     cout << endl;
6     return ;      // khong tra lai ket qua tinh toan nao, chi ket thuc
7 }
```

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

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**.

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 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 **printTenAsterik()**. Để in dãy 10 dấu **'*'** có thể gọi hàm **printTenAsterik()** hoặc **printAsterik(10)**, tuy nhiên để in dãy 12 dấu **'*'** hàm **printTenAsterik()** 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
```

```

3 using namespace std;
4
5 void printAsterik(int numAsteriks)
6 {
7     for (int count = 1; count <= numAsteriks; count++)
8         cout << "*" ;
9     cout << endl;
10    return ;
11 }
12
13 int main()
14 {
15     int edge_length;           // Chieu dai canh day cua tam giac
16     cout << "Enter the length of the base of an equilateral triangle: ";
17     cin >> edge_length;
18     int count1, count2;
19     for (count1 = 1; count1 <= edge_length; count1 += 2)
20     {
21         for (count2 = 1; count2 <= (edge_length - count1)/2; count2++)
22             cout << " ";
23         printAsterik(count1);
24     }
25
26     return 0;
27 }

```

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 luỹ thừa n (với n nguyên) của một số thực bất kỳ. Hàm này có hai đầu vào (đôi 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 ;           // bien luu ket qua
4     for (int count = 1; count <= n; count++)
5         res *= x ;
6     return res;                 // tra lai ket qua va ket thuc ham
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
3 using namespace std;
4
5 double power(double x, int n)
6 {

```

```

7     int i ;           // bien chi so
8     double res = 1.0 ;      // luu ket qua
9     for (i = 1; i <= n; i++)
10        res *= x ;
11    return res;          // tra lai ket qua va ket thuc ham
12 }
13
14 int main()
15 {
16     double x, res;
17     cout << "Enter value of x = "; cin >> x;
18     res = power(x, 4) + 2*power(x, 3) - 2*x + 1;
19     cout << "Value of x^4 + 2x^3 - 2x + 1 is " << res << endl;
20
21    return 0;
22 }
```

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
3 using namespace std;
4
5 double power(double, int);           // khai bao ham power (truoc)
6
7 int main()
8 {
9     cout << "Value of 5^2 is " << power(5, 2) << endl;      // main goi power
10    return 0;
11 }
12
13 double power(double x, int n)       // cai dat ham power (sau)
14 {
15     double res = 1.0 ;
16     for (int count = 1; count <= n; count++)
17         res *= x ;
18     return res;
19 }
```

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();           // khong doi, kieu ham la int
int square(int);         // mot doi kieu int, kieu ham la int
void alltrim(char[]);   // mot doi kieu xau, ham khong tra lai gia tri
int sum(int, int);       // hai doi kieu int, kieu ham la 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 đọc hiểu nội dung chính của chương trình. Vì vậy, hàm `main()` sẽ 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. Biến trong giá trị có thể trùng với tên đối. 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 đổi 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
3 using namespace std;
4
5 int GCD(int m, int n)      // Ham tim UCLN cua m va n
6 {
7     while (m != n)
8     {
9         if (m > n) m -= n;
10        else n -= m;
11    }
12    return m;
13 }
```

```

14
15 int main()
16 {
17     int numerator, denominator, gcd;
18     cout << "Enter the numerator = ";
19     cin >> numerator;
20     cout << "Enter the denominator = ";
21     cin >> denominator;
22     gcd = GCD(numerator, denominator);
23     cout << "Fraction in its lowest terms = "
24         << numerator/gcd << "/" << denominator/gcd << endl;
25
26     return 0;
27 }

```

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
3 using namespace std;
4
5 int sum(int m, int n, int k = 0, int h = 0)
6 {
7     return m + n + k + h;
8 }
9
10 int main()
11 {
12     cout << "5 + 2 + 3 + 7 equals to " << sum(5, 2, 3, 7) << endl; // 17
13     cout << "4 + 2 + 1 equals to " << sum(4, 2, 1) << endl; // 7
14     cout << "6 + 4 equals to " << sum(6, 4) << endl; // 10
15     system("PAUSE");
16     return 0;
17 }
```

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 măc định không liên tục
void sub(int x=0, int y); // sai vì đối măc 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: luỹ thừa, xoá màn hình và `main()`. Mục đích để minh họa biến cục bộ.

```

1 #include <iostream>
2
3 using namespace std;
4
5 double power(double x, int n)
6 {
7     int i;
8     double res = 1;
9     for (i = 1; i <= n; i++)
10         res *= x;
11     return res;
```

```

12 }
13
14 void clearScreen(int n) // xoa man hinh n lan
15 {
16     int i; // i, n la cuc bo cua clearScreen
17     for (i = 1; i <= n; i++)
18     {
19         cout << "Press any key to clear screen (" << i << "th time)";
20         cin.get(); // yeu cau doc vao phim bat ky
21         system("CLS");
22     }
23 }
24
25 int main()
26 {
27     double x; int n; // x, n la cuc bo cua main
28     cout << "Enter x = "; cin >> x;
29     cout << "Enter n = "; cin >> n;
30     cin.ignore(); // xoa ki tu con luu trong bo dem ban phim
31     clearScreen(3); // xoa man hinh 3 lan
32     cout << "Result = " << power(x, n) << endl; // in x ^ n
33     system("PAUSE");
34     return 0;
35 }

```

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.
- 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 `main()`.

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 tất cả các hàm khai báo sau này đều 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 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** (tổn 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 chẵn cho 5 thì người đó thắng cuộc hoặc trả 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 (tổn thể) để chứa kết quả này và 2 hàm cộng cho 2 người chơi.

```

1 #include <iostream>
2
3 using namespace std;
4
5 int x;
6
7 void playerOne()
8 {
9     int num;
10    cout << "First Player enters an integer : ";
11    cin >> num;
12    x += num;
13 }
14
15 void playerTwo()
16 {
17     int num;
18     cout << "Second Player enters an integer : ";
19     cin >> num;
20     x += num;
21 }
22
23 int main()
24 {
25     x = rand();
26     int time = 1;

```

```

27     while (time <= 3)
28     {
29         playerOne();
30         if (x % 5 == 0)
31         {
32             cout << "Your obtained number is " << x << ". You win" << endl;
33             break;
34         }
35         playerTwo();
36         if (x % 5 == 0)
37         {
38             cout << "Your obtained number is " << x << ". You win" << endl;
39             break;
40         }
41         time++;
42     }
43     if (time > 3) cout << "Game Over\n";
44     system("PAUSE");
45     return 0;
46 }
```

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
3 using namespace std;
4
5 int i;
6 double x;
7 double power(double x, int n);
8 void clearScreen(int n);
9
10 int main()
11 {
12     x = 2;
13     i = 5;
14     clearScreen(3);                                // xoa man hinh 3 lan
```

```

15     cout << x << " ^ " << i << " = " << power(x, i) << endl; // in 2 ^ 5
16     system("PAUSE");
17     return 0;
18 }
19
20 double power(double x, int n)
21 {
22     double res = 1;
23     for (i = 1; i <= n; i++)
24         res *= x;
25     return res;
26 }
27
28 void clearScreen(int n) // xoa man hinh n lan
29 {
30     for (i = 1; i <= n; i++)
31     {
32         cout << "Press any key to clear screen (" << i << "th time)";
33         cin.get();
34         system("CLS");
35     }
36 }

```

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 đến để xoá 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:

- 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 khi ở 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, xoá, 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 gấp 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).
- Tới 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 đổi 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 đổi) như trong ví dụ sau:

```

1 #include <iostream>
2
3 using namespace std;
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     system("PAUSE");
16     return 0;
17 }
```

Hình 4.12: Tính số bít.

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 đổi `bits` đã thay đổi giá trị (bằng 35), tuy nhiên đối `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.

```

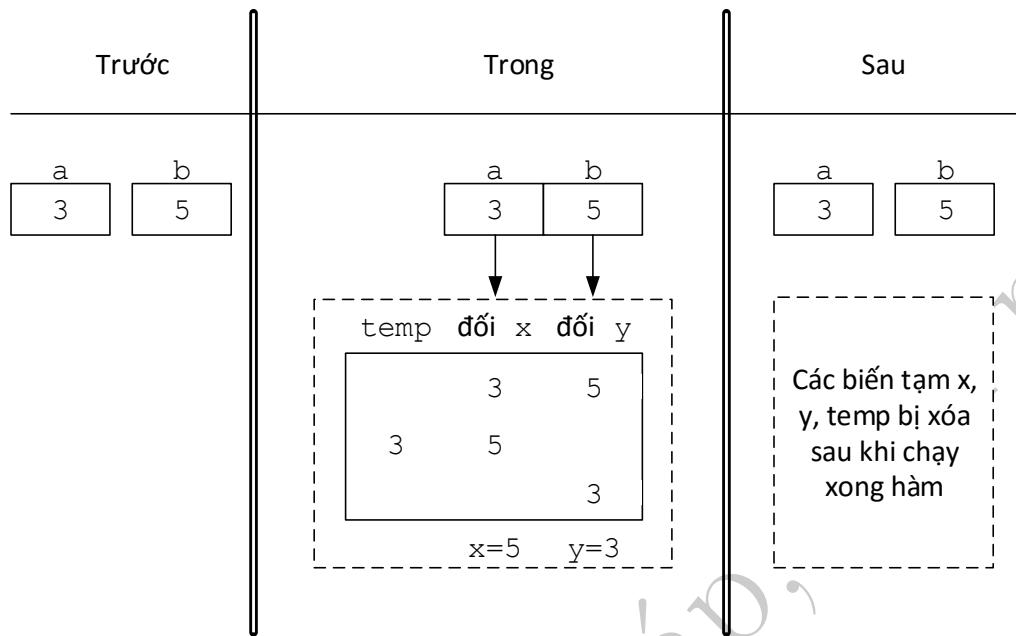
1 #include <iostream>
2
3 using namespace std;
4
5 void swap(int x, int y)
6 {
7     int temp ;
8     temp = x ;
9     x = y ;
10    y = temp ;
11    cout << "x = " << x << ", y = " << y << endl; // 5, 3 (x,y doi gia tri)
12 }
13
14 int main()
15 {
16     int a = 3, b = 5;
17     swap(a, b) ;
18     cout << "a = " << a << ", b = " << b << endl; // 3, 5 (a,b van nhu cu)
19     system("PAUSE");
20     return 0;
21 }
```

Hình 4.13: Tráo 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 đổi `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 dưới đây minh họa cách làm việc của hàm **swap**, trước, trong và sau khi gọi hàm.



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

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 số **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 thế 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.

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

Từ vị trí này trở đi việc sử dụng các tên **hung**, **tí** hoặc **dũng**, **teo** 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 đổi 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
3 using namespace std;
4
5 void swap(int &x, int &y)
6 {
7     int temp ;
8     temp = x ;
9     x = y ;
10    y = temp ;
11    cout << "x = " << x << ", y = " << y << endl; // 5, 3 (x,y doi gia tri)
12 }
13
14 int main()
15 {
16     int a = 3, b = 5;
17     swap(a, b) ;
```

```

18     cout << "a = " << a << ", b = " << b << endl; // 5, 3 (a,b doi gia tri)
19     system("PAUSE");
20     return 0;
21 }

```

Hình 4.15: Tráo đổi 2 số bằng tham chiếu.

Dưới đây là bảng chỉ ra sự khác biệt của 2 hàm swap. Sử dụng biến tham chiếu cho phép viết

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, b</code> không thay đổi | <code>a, b</code> có thay đổi |

được mọi chương trình một cách bình thường như ý muốn. Ví dụ viết hàm nhập giá trị cho một số biến. Các biến này không thể là đối bình thường của hàm vì sau khi gọi hàm 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 viết hàm dưới dạng đổi tham chiếu để lợi dụng tốc độ thực hiện. Tuy nhiên, cách viết này cũng có thể vô tình làm 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 đổi tham chiếu. Khi đó, nếu trong hàm có câu lệnh nào vô tình thay đổi giá trị của đổi, chương trình dịch sẽ báo lỗi.

4.6 Chồng hàm và khuôn mẫu hàm

4.6.1 Chồng hàm (hàm trùng tên)

Hàm trùng tên hay còn gọi là hàm chồng (đè). 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**) không phù hợp về kiểu (**int**) của đổi 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 đổi khi gọi để quyết định chạy hàm nào. Ví dụ lời gọi **max(3, 5)** có 2 đổi 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 gấp lời gọi **max('0', 'K')**. Như vậy một đặc điểm của hai hàm trùng tên đó là trong danh sách đổi của chúng phải có ít nhất một cặp đổi nào đó khác kiểu nhau. Một đặc trưng khác cũng để phân biệt hai hàm trùng tên nhưng khác nhau đó là số lượng đổi 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 toạ độ của chúng. 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 thỏa các điều kiện sau:

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

32 }

Hình 4.17: Nạp chồng hàm tính ước số chung lớn nhất.

Chú ý: Cần tránh tính 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(7, 4)`, chương trình sẽ không quyết định được nên gọi hàm 1 hay hàm 2, trường hợp này chương trình dịch sẽ báo lỗi.

4.6.2 Khuôn mẫu hàm

Việc cho phép các hàm trùng tên (chồng hàm) 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 còn dẫn đến lãng phí công sức. 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, tại sao ta không chỉ viết một hàm “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> // (class và typename là tương đương)
```

sau đó 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
3 using namespace std;
4
5 // GenType la ten kieu chung dai dien cho tat ca cac kieu
6 template <typename Gen_Type>
7 Gen_Type getMax (Gen_Type a, Gen_Type b)
8 {
9     return (a > b ? a : b);
10 }
11
12 int main()
13 {
14     int m, n;
15     double x, y;
16     char c, d;
17     cout << "Enter integers m, n: " ; cin >> m >> n;
18     cout << "Maximum of two integers " << m << " and " << n << " is: " << getMax(m, n)
19         ) << endl;
20     cout << "Enter doubles x, y: "; cin >> x >> y;
21     cout << "Maximum of two doubles " << x << " and " << y << " is: " << getMax(x, y)
22         << endl;
23     cout << "Enter characters c, d: "; cin >> c >> d;
24     cout << "Maximum of two characters " << c << " and " << d << " is: " << getMax(c,
25         d) << endl;
26     system("PAUSE");
27     return 0;
28 }
```

Hình 4.18: 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
3 using namespace std;
4
5 template <typename T, typename U> // 2 kieu mau cho 2 doi khac kieu nhau
6 T GetMax (T a, U b)           // gia tri tra lai la kieu cua doi thu 1 (T)
7 {
8     T res;
9     res = (a > b ? a : b);
```

```

10     return res;
11 }
12
13 int main()
14 {
15     double double_num;
16     int integer_num;
17     char character;
18     cout << "Enter the double, the integer, the character: " ; cin >> double_num >>
19         integer_num >> character;
20     cout << "Maximum of " << double_num << " and " << integer_num << " is: " <<
21         GetMax(double_num, integer_num) << endl; // thuc - nguyen, tra lai thuc
22     cout << "Maximum of " << integer_num << " and " << character << " is: " << GetMax
23         (integer_num, character) << endl; // nguyen - ki tu, tra lai nguyen
24     cout << "Maximum of " << integer_num << " and " << character << " is: " << GetMax
25         (character, integer_num) << endl; // nguyen - ki tu, tra lai ki tu
26     system("PAUSE");
27     return 0;
28 }
```

Hình 4.19: 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 nguyên và thực và giá trị trả lại là số thực (`65.0`), 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.

4.7 Lập trình với hàm đệ quy

4.7.1 Khái niệm đệ qui

Một hàm gọi đến hàm khác là bình thường, nhưng nếu hàm lại gọi đến chính nó thì ta gọi hàm là đệ qui. Như vậy, khi một hàm nào đó gọi đến hàm đệ qui thì hàm đệ qui 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 qui 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
3 using namespace std;
4
5 long factorial(int n)
6 {
7     long res;
8     res = 1;
9     for (int count = 1; count <= n; count++)
10        res *= count;
11    return res;
```

```

12 }
13
14 int main()
15 {
16     int n;
17     cout << "n = " ; cin >> n;           // nhap so can tinh giai thua
18     cout << n << "!" << factorial(n) << endl;
19     system("PAUSE");
20     return 0;
21 }
```

Hình 4.20: 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
3 using namespace std;
4
5 long factorial2(int n)
6 {
7     if (n == 0) return 1;
8     else return factorial2(n - 1) * n;
9 }
10
11 int main()
12 {
13     int n;
14     cout << "n = " ; cin >> n;           // nhap so can tinh giai thua
15     cout << n << "!" << factorial2(n) << endl;
16     system("PAUSE");
17     return 0;
18 }
```

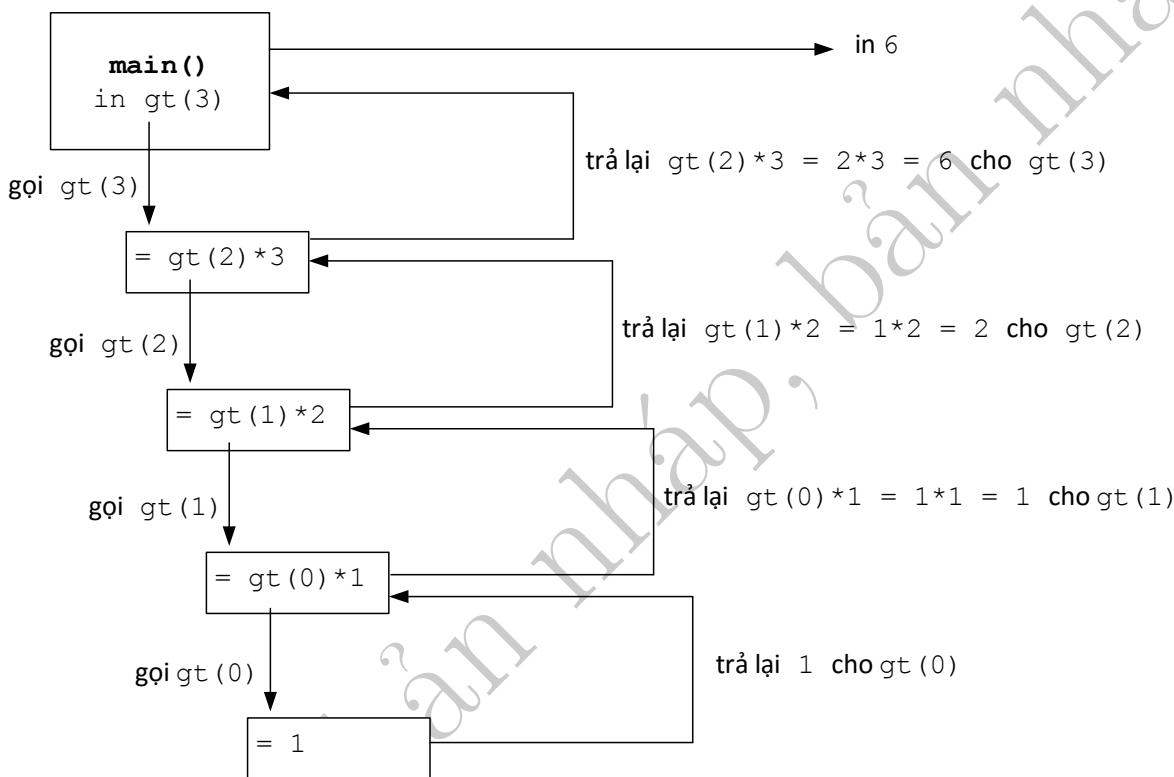
Hình 4.21: Tính giai thừa bằng hàm đệ qui.

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**, ..., 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 là gọi hàm `factorial` với tham số 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 điều $n = 0$ 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`):



Hình 4.22: Quá trình gọi đệ quy của hàm `factorial`.

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

- Hàm được viết rất gọn.
- 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.

4.7.2 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, 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
3 using namespace std;
4
5 int sum(int n)
6 {
7     if (n == 1) return 1;
8     else return sum(n - 1) + n;
9 }
10
11 int main()
12 {
13     int n;
14     cout << "Enter n = " ; cin >> n;           // nhap so can tinh tong
15     cout << "Sum of first n integer numbers = " << sum(n) << endl;
16     system("PAUSE");
17     return 0;
18 }
```

Hình 4.23: Tính giai thừa bằng hàm đệ qui.

4.7.3 Cấu trúc chung của hàm đệ qui

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

```

if (tham số suy biến)
    trình bày cách giải trực tiếp // giả định đã có cách giải
else
    // tham số chưa suy biến
    gọi lại hàm với tham số "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
4 using namespace std;
5
6 double S(int numSquare_roots)
7 {
8     double res;
9     if (numSquare_roots == 1) res = sqrt(3);
10    else res = sqrt(3 + S(numSquare_roots - 1));
11    return res;
12 }
13
14 int main()
15 {
16     int numSquare_roots;
17     cout << "Number of the Square roots = " ; cin >> numSquare_roots;
18     cout << "Result = " << S(numSquare_roots) << endl;
19     system("PAUSE");
20     return 0;
21 }

```

Hình 4.24: 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
4 using namespace std;
5
6 double S(int numDivisions)
7 {
8     if (numDivisions == 1) return 1/2.0;           // chia 2 được viết 2.0
9     else return 1/(2 + S(numDivisions - 1));
10}
11
12 int main()
13 {
14     int numDivisions;
15     cout << "Number of the Divisions = " ; cin >> numDivisions;
16     cout << "S = " << S(numDivisions) << endl;
17     system("PAUSE");
18     return 0;
19}

```

Hình 4.25: Tính $S(n)$ với n dấu chia.

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 kè trước nó, tức $f(n) = f(n-1) + f(n-2)$ ($n > 2$). Một định nghĩa đệ qui của dãy số được dễ dàng suy ra như sau:

Từ đó ta có chương trình đệ qui:

```

1 #include <iostream>
2
3 using namespace std;
4
5 long fib(int n)
6 {
7     long res;
8     if (n==1 || n==2) res = 1;
9     else res = fib(n-1) + fib(n-2);
10    return res;
11}
12
13 int main()
14 {
15     int n;
16     cout << "n = " ; cin >> n;
17     cout << "The first " << n << " fibonacci numbers\n";
18     for (int count = 1; count <= n; count++)
19         cout << fib(count) << endl;
20     system("PAUSE");
21     return 0;
22}

```

Hình 4.26: Tính số Fibonacci.

4. Tìm UCLN của 2 số a, b . Có thể định nghĩa hàm $\text{GCD}(\text{int } m, \text{ int } n)$ dưới dạng đệ qui 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 qui để tính UCLN của a và b:

```

1 #include <iostream>
2
3 using namespace std;
4
5 int GCD(int m, int n)
6 {
7     if (m == n) return m;
8     if (m > n) return GCD(m - n, n);
9     if (m < n) return GCD(m, n - m);
10 }
11
12 int main()
13 {
14     int m, n;
15     cout << "m = " ; cin >> m;
16     cout << "n = " ; cin >> n;
17     cout << "Greatest common divisor of " << m << " and " << n << " is " << GCD(
18         m,n) << endl;
19     system("PAUSE");
20 }
```

Hình 4.27: Tính UCLN bằng hàm đệ qui.

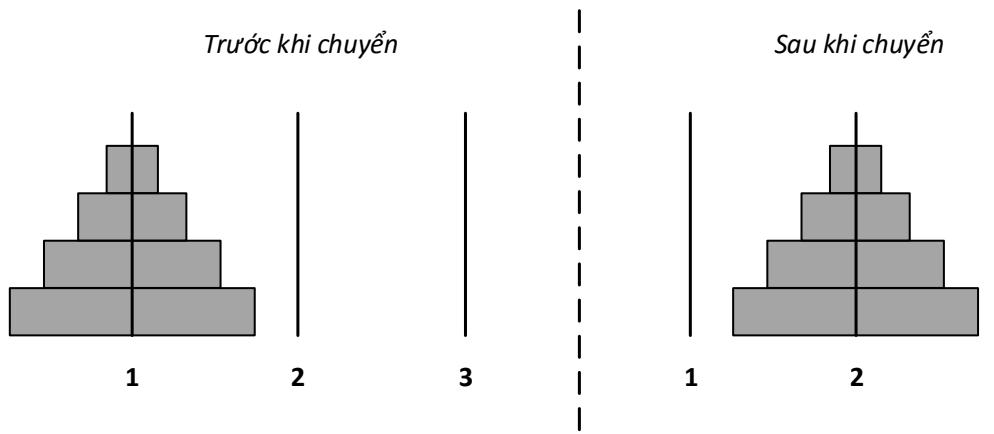
5. Chuyển tháp là bài toán cổ nổi tiếng, nội dung như sau: Cho một tháp n tầng, đang xếp tại vị trí 1. Yêu cầu bài toán là hãy chuyển toàn bộ tháp sang vị trí 2 (cho phép sử dụng vị trí trung gian 3) theo các điều kiện sau đây:

- 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 bởi hình vẽ dưới đây.

Bài toán có thể được đặt ra tổng quát hơn như sau: chuyển tháp 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-source-target** (vì **source + target + 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**



Hình 4.28: Bài toán Tháp Hà Nội.

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

```

```

31     return 0;
32 }
```

Hình 4.29: 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.

4.8 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) Các tham số hình thức sẽ được cấp phát bộ nhớ tạm thời khi hàm được gọi
- (d) Kiểu của tham số thực sự phải giống (hoặc C++ có thể tự động chuyển kiểu được về) kiểu của tham số hình thức tương ứng với nó trong lời gọi hàm.

4. Chọn câu sai trong các câu sau đây:

- (a) Số lượng các đối măc định là tùy ý.
- (b) Được phép khai báo các tham đối măc định ở bất kỳ vị trí nào trong danh sách đối.
- (c) Vị trí các đối măc định phải liên tiếp nhau và ở cuối danh sách đối
- (d) Với hàm không có biến măc định thì số tham số 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 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 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 số 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:
 - (a) Tìm UCLN của 3 số nhập từ bàn phím.
 - (b) Tìm phân số tối giản của một phân số
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 là số bằng tổng mọi ước của nó (Ví dụ $6 = 1 + 2 + 3$). Hãy in ra mọi số hoàn chỉnh từ 1 đến 100.
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 qui 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 qui 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 qui tính số Fibonacci thứ n . Dùng chương trình con này tính $f(2) + f(4) + f(6) + f(8)$.

20. Viết dưới dạng đệ qui các hàm

(a) UCLN

(b) Fibonacci

(c) Tháp Hà Nội

bản nháp, bản nháp,
bản nháp, bản nháp,

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 vec tơ 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 kề nhau liên tục 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 liền 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 bổ 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 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 gõ 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     system("PAUSE");
21     return 0;
22 }
```

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    system("PAUSE");
32    return 0;
33 }
```

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 " <<
31    minId << endl;
32
33    system("PAUSE");
34    return 0;
}

```

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 ($n \leq SIZE$). 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 }; // khai 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(slary_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 đổi `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     system("PAUSE");
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 vec tơ).

```

1 #include <iostream>
2
```

```

3 using namespace std;
4
5 void setup(int[], int);
6 int procduct(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 " << procduct(vector_A, vector_B,
19         numElement) << endl;
20
21     system("PAUSE");
22     return 0;
23 }
24
25 void setup(int A[], int size)
26 {
27     cout << "Enter " << size << " numbers:\n";
28     for (int index = 0; index < size; index++)
29         cin >> A[index];
30     return;
31 }
32
33 int procduct(const int A[], const int B[], int size)
34 {
35     int res = 0;
36     for (int index = 0; index < size; index++)
37         res += A[index] * B[index];
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 lai mang sample_array
15    cout << endl;
16
17    system("PAUSE");
18    return 0;
19 }
20
21 void display(int A[], int size)
22 {
23     for (int index = 0; index < 5; index++)
24         cout << ++A[index] << " "; // Tang them 1 cho phan tu mang va in
25     cout << endl;
26     return;
27 }
```

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 số `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 lai mang sample_array
15     cout << endl;
16
17     system("PAUSE");
18     return 0;
19 }
20
21 void display(const int A[], int size)
22 {
23     for (int index = 0; index < 5; index++)
24         cout << A[index] + 1 << " "; // Tang them 1 cho phan tu mang va in
25     cout << endl;
26     return;
27 }
```

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] << ", ";
21     cout << c[2] << " )" << endl;
22     system("PAUSE");
23     return 0;
24 }
```

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     system("PAUSE");
29     return 0;
30 }
31
32 void setup(int A[], int used_size)
33 {
34     cout << "Enter " << used_size << " numbers: ";
35     for (int index = 0; index < used_size; index++)
36         cin >> A[index];
37     return;
38 }
39
40 int search(const int A[], int used_size, int target)
41 {
42     int found_at = -1; // ngam dinh khong tim thay target
43     for (int index = 0; index < used_size; index++)
44         if (A[index] == target)
45         {
46             found_at = index;
47             break;
48         }
49     return found_at;
50 }

```

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 dãy 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 tráo 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]), tráo đổi với A[1] (30) ta được dãy mới

Bảng 5.12: Lần tráo 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, tráo đổi với A[2] ta được dãy mới với 2 vị trí đã sắp xếp

Bảng 5.13: Lần tráo đổi 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 lắp 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     system("PAUSE");
24     return 0;
25 }
26
27 void setup(int A[], int n)
28 {
29     cout << "Enter " << n << " numbers: ";
30     for (int index = 0; index < n; index++)
31         cin >> A[index];
32     return;
33 }
34
35 void print(const int A[], int n)          // ham in day so
36 {
37     for (int index = 0; index < n; index++)
38         cout << A[index] << " ";
39     cout << endl;
40     return;
41 }
42
43 int sort(int A[], int n)                  // ham sap xep
44 {
45     int tmp, min_value;
46     int id_min;
47     for (int id1 = 0; id1 < n-1; id1++)
48     {                                       // chon so be nhat
49         min_value = A[id1];
50         id_min = id1;
51         for (int id2 = id1+1; id2 < n; id2++)
52             if (A[id2] < min_value)
53             {
```

```

54         min_value = A[id2];
55         id_min = id2;
56     }
57     tmp = A[id1];                                // doi A[id1] voi A[id2]
58     A[id1] = A[id_min];
59     A[id_min] = tmp;
60 }
61 }
```

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     system("PAUSE");
27     return 0;
28 }
29
30 void setup(int A[], int n)
31 {
32     cout << "Enter " << n << " numbers: ";
33     for (int index = 0; index < n; index++)
34         cin >> A[index];
35     return;
36 }
37
38 void print(const int A[], int n)            // ham in day so
```

```

39 {
40     for (int index = 0; index < n; index++)
41         cout << A[index] << " ";
42     cout << endl;
43     return;
44 }

45

46 void swap(int &v1, int &v2)
47 {
48     int tmp;
49     tmp = v1;
50     v1 = v2;
51     v2 = tmp;
52 }

53

54 int get_index_of_smallest(const int A[], int start_index, int last_index)
55 {
56     int min_value = A[start_index];
57     int id_min = start_index;
58     for (int i = start_index+1; i <= last_index; i++)
59         if (A[i] < min_value)
60         {
61             min_value = A[i];
62             id_min = i;
63         }
64     return id_min;
65 }

66

67 int sort(int A[], int n) // ham sap xep
68 {
69     int id_min;
70     for (int i = 0; i < n-1; i++)
71     {
72         id_min = get_index_of_smallest(A, i, n-1); // chon so be nhat
73         swap(A[i], A[id_min]);
74     }
75 }

```

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 ký 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 họa trong hình dưới đây.

Bảng 5.18: Phân bổ 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 tương minh để tránh gây nhầm lẫn. Ví dụ:

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

Khai 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     system("PAUSE");
21     return 0;
22 }
```

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 bổ 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 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
13   num_courses, double stu_av[]);
14 void computeCourseAve(const int mark[][NUM_COURSES], int num_students, int
15   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     system("PAUSE");
28     return 0;
29 }
30
31 void fillup(int mark[][NUM_COURSES], int num_students, int num_courses)
32 {
33     for (int st_id = 0; st_id < num_students; st_id++)
34     {
35         cout << "Enter " << num_courses << " marks of student #" << st_id + 1 << ":" ;
36         for (int crs_id = 0; crs_id < num_courses; crs_id++)
37             cin >> mark[st_id][crs_id];
38     }
39     return;
40 }
41
42 void computeStudentAve(const int mark[][NUM_COURSES], int num_students, int
43 num_courses, double stu_av[])
44 {
45     for (int st_id = 0; st_id < num_students; st_id++)
46     {
47         double sum = 0;
48         for (int crs_id = 0; crs_id < num_courses; crs_id++)
49             sum = sum + mark[st_id][crs_id];
50         stu_av[st_id] = sum/num_courses;
51     }
52     return;
53 }
54
55 void computeCourseAve(const int mark[][NUM_COURSES], int num_students, int
56 num_courses, double course_av[])
57 {
58     for (int crs_id = 0; crs_id < num_courses; crs_id++)
59     {
60         double sum = 0;
61         for (int st_id = 0; st_id < num_students; st_id++)
62             sum = sum + mark[st_id][crs_id];
63         course_av[crs_id] = sum/num_students;
64     }
65 }
```

```

65
66 void display(const int mark[][NUM_COURSES], int num_students, int num_courses)
67 {
68     cout.setf(ios::fixed);
69     cout.setf(ios::showpoint);
70     cout.precision(1);
71     cout << "\nCourse marks and average mark of students\n";
72     cout << setw(10) << "Student";
73     for (int crs_id = 1; crs_id <= num_courses; crs_id++) cout << setw(9) << "Crs#"
74         << crs_id;
75     cout << setw(10) << "Ave" << endl;
76     for (int st_id = 0; st_id < num_students; st_id++)
77     {
78         cout << setw(10) << st_id + 1;
79         for (int crs_id = 0; crs_id < num_courses; crs_id++)
80             cout << setw(10) << mark[st_id][crs_id];
81         cout << setw(10) << stu_av[st_id] << endl;
82     }
83     cout << "Crs. Ave = ";
84     for (int crs_id = 0; crs_id < num_courses; crs_id++)
85         cout << setw(10) << course_av[crs_id];
86     cout << endl;
87     return;
}

```

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 trả về 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 ;                                // cac kich thuoc cua ma tran
10    int i, j, k ;                                // cac chi so vong lap
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 // Hiển thị kết quả
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 system("PAUSE");
46 return 0;
47 }
```

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;           // số dòng, cột tối đa của các ma trận A, B, C
7
8 void fillup(double [][]SIZE, int, int);
9 void productMatrix(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 ; // các kích thước của ma trận
```

```

16
17     cout << "Enter sizes of two matrixes: " ; cin >> used_size1 >> used_size2 >>
18         used_size3;
19     cout << "Enter values of matrix A(" << used_size1 << " x " << used_size2 << "):\n"
20         " ;
21     fillup(A, used_size1, used_size2);
22     cout << "Enter values of matrix B(" << used_size2 << " x " << used_size3 << "):\n"
23         " ;
24     fillup(B, used_size2, used_size3);
25     procdutMatrix(A, B, C, used_size1, used_size2, used_size3);
26     cout << "Result Matrix C(" << used_size1 << " x " << used_size3 << "):\n" ;
27     display(C, used_size1, used_size3);
28     cout << endl;
29
30
31 void fillup(double matrix[] [SIZE], int num_row, int num_col)
32 {
33     for (int i = 0; i < num_row; i++)
34         for (int j = 0; j < num_col; j++)
35             cin >> matrix[i][j] ;
36 }
37
38 void procdutMatrix(const double A[] [SIZE], const double B[] [SIZE], double res[] [SIZE]
39 , int size1, int size2, int size3)
40 {
41     int i, j, k;
42     for (i = 0; i < size1; i++)
43         for (j = 0; j < size2; j++)
44         {
45             res[i][j] = 0;
46             for (k = 0; k < size3; k++) res[i][j] += A[i][k]*B[k][j] ;
47         }
48 }
49
50 void display(const double matrix[] [SIZE], int num_row, int num_col)
51 {
52     cout << setiosflags(ios::showpoint) << setprecision(2) ;
53     for (int i = 0; i < num_row; i++)
54     {
55         for (int j = 0; j < num_col; j++)
56             cout << setw(6) << matrix[i][j] ;
57         cout << endl;
58 }

```

Hình 5.22: Nhân ma trận.

5.3 Lập trình và thao tác với xâu kí tự

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

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

Bảng 5.23: Xâu kí 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 xâu, mỗi xâu được chứa trong mảng kí tự có độ dài tối đa là 8. Nội dung xâu thứ nhất là "Hello" có độ dài thực tế là 5 kí tự, chiếm 6 ô trong mảng (thêm ô chứa kí tự kết thúc '\0'). Xâu thứ hai có nội dung "Hell" với độ dài 4 (chiếm 5 ô) và xâu cuối cùng biểu thi 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 kí tự ' phải được viết là \
char str[50] = "I'm a student" ;
cout << str[0] ;           // in kí tự đầu tiên, tức kí tự 'I'
str[1] = 'a' ;            // đặt lại kí tự thứ 2 là 'a'
```

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

```
// khai báo hai xâu 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 xâu nhập vào là "Tin học hóa" chẳng hạn thì toán tử >> chỉ nhập "Tin" cho str (bỏ tất cả các kí 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 xâu 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 xâu. Nó cung cấp cho NSD phương tiện để thao tác trên xâu như gán, so sánh, sao chép, tính độ dài xâu, ... Để sử dụng được các hàm này đầu chương trình cần khai báo `#include <cstring>`.

- Khi duyệt xâu, để nhận biết hết xâu có thể dùng một trong hai cách sau:

- Kí tự hiện tại của xâu là '\0'

- Kí tự đang đọc nằm tại vị trí `strlen(str)` (`strlen()` là hàm trả lại độ dài của xâu str). Đây cũng chính là vị trí của kí tự '\0'.

Ví dụ: cần duyệt từ đầu đến cuối xâu str:

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

5.3.3 Phương thức nhập xâu (#include <iostream>)

Do toán tử nhập `>>` có hạn chế đối với xâu kí tự, nên C++ đưa ra hàm riêng (còn gọi là phương thức) `cin.getline(s, n)` để nhập xâu kí tự. Hàm có 2 đối với s là xâu cần nhập nội dung và n-1 là số kí tự tối đa của xâu. Nếu xâu do NSD nhập vào nhiều hơn n-1 kí tự hàm chỉ lấy n-1 kí 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    system("PAUSE");
18    return 0;
19 }
```

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

5.3.4 Một số hàm làm việc với xâu kí tự (#include <cstring>)

Sao chép xâu

- `strcpy(s, t)`: Gán nội dung của xâu t cho xâu s (thay cho phép gán = không được dùng). Hàm sẽ sao chép toàn bộ nội dung của xâu t (kể cả kí tự kết thúc xâu) vào xâu 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 kí tự kết thúc xâu 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" ;                                // khong duoc
10    strcpy(str_t, "Face") ;                          // gan "Face" cho t
11    // str_s = str_t ;                                // khong duoc
12    strcpy(str_s, str_t) ;                            // sao chep t sang s
13    cout << str_s << " to " << str_t << endl;      // in ra: Face to Face
14    system("PAUSE");
15    return 0;
16 }
```

Hình 5.25: Ví dụ strcpy.

- **strncpy(s, t, n):** Sao chép n kí tự đầu tiên của t vào s. Hàm này chỉ làm nhiệm vụ sao chép, không tự động gắn kí tự kết thúc xâu cho s. Do vậy NSD phải thêm câu lệnh đặt kí tự '\0' vào cuối xâu 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 ki tu "Steve" vao s
10    str_s[5] = '\0' ;                                // dat dau ket thuc xau
11    cout << str_s << " is younger brother of " << str_t << endl;
12                                // in : Steve is younger brother of Steven
13
14    system("PAUSE");
15    return 0;
16 }
```

Hình 5.26: Ví dụ strncpy.

Dạng tổng quát của hàm trên là: **strncpy(s + a, t + b, n)** cho phép copy n kí tự bắt đầu từ vị trí (chỉ số) b của xâu t và đặt (ghi đè) vào xâu 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     system("PAUSE");
17     return 0;
18 }
```

Hình 5.27: Chuyển đổi ngày bằng `strncpy`.

Ghép hai xâu

- `strcat(s, t)`: Nối một bản sao của `t` vào sau `s` (thay cho phép `+`). Hiển nhiên hàm sẽ loại bỏ kí tự kết thúc xâu `s` trước khi nối thêm `t`. Việc nối sẽ đảm bảo lấy cả kí tự kết thúc của xâu `t` vào cho `s` (nếu `s` đủ chỗ) vì vậy NSD không cần thêm kí tự này vào cuối xâu. 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    system("PAUSE");
16    return 0;
17 }
```

Hình 5.28: Ví dụ `strcat`.

- `strncat(s, t, n)`: Nối bản sao `n` kí tự đầu tiên của xâu `t` vào sau xâu `s`. Hàm tự động đặt thêm dấu kết thúc xâu 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     strncat(str_s, " to ");
11     strncat(str_s, str_t, 5);
12     cout << str_s << endl;                                // Do be do to Steve
13
14     system("PAUSE");
15     return 0;
16 }
```

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    system("PAUSE");
14    return 0;
15 }
```

Hình 5.30: Ví dụ strncat.

So sánh hai xâu

- strcmp(s, t): Hàm so sánh 2 xâu s và t (thay cho các phép toán so sánh). Giá trị trả lại là hàm dấu của hiệu 2 kí tự khác nhau đầu tiên của s và t. Tức nếu hiệu là âm thì sẽ trả lại -1 (tương đương $s_1 < s_2$), nếu hiệu dương trả lại giá trị 1 ($s_1 > s_2$), nếu so sánh đến hết xâu, tất cả hiệu đều bằng 0 ($s_1 == s_2$) thì sẽ trả lại 0. Trong trường hợp chỉ quan tâm đến so sánh bằng, nếu hàm trả lại giá trị 0 là hai xâu bằng nhau và nếu giá trị trả lại khác 0 là hai xâu khác nhau (chú ý: xâu với độ dài ngắn hơn không nhất thiết là xâu 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 " <<
11        strcmp(str_s, str_t) << endl;                      // -1
12    if (strcmp(str_s, str_t) < 0)
13        cout << str_s << " < " << str_t;
```

```

13     else
14         cout << str_s << " > " << str_t;
15     cout << endl;
16     system("PAUSE");
17     return 0;
18 }
```

Hình 5.31: Ví dụ strcmp.

- `strncmp(s, t, n)`: Giống hàm `strcmp(s, t)` nhưng chỉ so sánh tối đa `n` kí tự đầu tiên của hai xâu.

```

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 <<
12            " is the same" << endl;
13    else
14        cout << "The first two characters of " << str_s << " and " << str_t <<
15            " is not the same" << endl;
16    cout << endl;
17    system("PAUSE");
18    return 0;
19 }
```

Hình 5.32: Ví dụ strncmp.

- `strcasecmp(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 `strcasecmp`.

Lấy độ dài xâu

- `strlen(s)`: Hàm trả giá trị là độ dài của xâu `s`. Ví dụ:

```

char str[10] = "Ha Noi" ;
cout << strlen(str) ; // 6
```

5.3.5 Các hàm chuyển đổi xâu dạng số thành số (`#include <cstdlib>`)

- `atoi(str)` ; `atol(str)` ; `atof(str)` ; (A-TO-I: alphabet to integer, ...)

Đối `str` là xâu kí 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 xâu có chứa cả kí tự khác với chữ số, hàm sẽ chỉ chuyển đổi giá trị là các chữ số đứng trước kí tự lạ đầu tiên trong xâu. 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 kí 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 xâu để ghép thành số nguyên. Hàm trả lại 0 nếu xâu 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     system("PAUSE");
34     return 0;
35 }

```

Hình 5.33: Hàm new_atoi.

Cách khai báo xâu dưới dạng mảng còn nhiều hạn chế, khó lập trình. Xâu được khai báo dưới dạng con trỏ kí tự và đặc biệt C++ cung cấp một lớp riêng về xâu (lớp String) sẽ giúp người lập trình làm việc thuận lợi hơn.

5.3.6 Một số ví dụ làm việc với xâu

```

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 <<
16        endl ;
17    system("PAUSE");
18    return 0;
}

```

Hình 5.34: Thống kê số chữ 'a' xuất hiện trong xâu 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    system("PAUSE");
23    return 0;
24 }

```

Hình 5.35: Tính độ dài xâu 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     system("PAUSE");
41     return 0;
42 }

```

Hình 5.36: Đếm số từ (qui ước là dãy kí tự bất kỳ, liên tục) trong xâu. Chương trình đọc từ đầu đến cuối xâu, sử dụng biến logic `reading_blank` để chỉ trạng thái hiện tại đang đọc dấu cách hay kí tự khác. Nếu ở trạng thái đang đọc dấu cách và gặp kí tự khác thì tăng số từ lên 1 và đổi trạng thái. Nếu trạng thái đang đọc kí tự khác và gặp dấu cách thì đổi trạng thái, nếu gặp kí 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     system("PAUSE");
29     return 0;
30 }

```

Hình 5.37: Cắt dấu cách 2 đầu của xâu s. Chương trình sử dụng cờ head_flag và tail_flag chạy từ 2 đầu xâu đến vị trí đầu tiên có kí tự khác dấu trắng. Dùng hàm strncpy sao chép đoạn kí tự từ head_flag đến tail_flag vào lại đầu xâu, 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     }
28 }

```

```

26     if (num_times <= 3)
27         cout << "Reenter\n";
28     else
29         cout << "You don't be permitted to use this software\n";
30     } while (num_times <= 3);
31
32     system("PAUSE");
33     return 0;
34 }

```

Hình 5.38: Nhập mật khẩu (không quá 10 kí tự). In ra "dung" 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).

5.4 Bài tập

1. Hãy nhập vào 16 số nguyên. In ra thành 4 dòng, 4 cột.
2. 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.
3. Nhập vào dãy n số. Hãy in ra số lớn nhất, bé nhất của dãy.
4. Tìm và chỉ ra vị trí xuất hiện đầu tiên của phần tử x trong dãy.
5. Nhập vào dãy số. In ra dãy đã được sắp xếp tăng dần, giảm dần.
6. Cho dãy đã được sắp tăng dần. Chèn thêm vào dãy phần tử x sao cho dãy vẫn sắp xếp tăng dần.
7. 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.
8. 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.
9. Câu nào trong các khẳng định sau là sai :
 - (a) Các phần tử của mảng được sắp xếp liên tục trong bộ nhớ
 - (b) Các phần tử của mảng chiếm một số byte như nhau
 - (c) Với mảng int x[3][4] số nguyên cuối cùng là phần tử x[3][4]
 - (d) Cho phép khởi tạo giá trị của mảng ngay trong khai báo
10. 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 ?

- (a) Cách khởi tạo mảng **b** là được phép.
- (b) $a[i][j] = b[i][j]$ ($i = 0, 1; j = 0, 1, 2$)
- (c) Hai mảng **a**, **b** có cùng số phần tử.
- (d) Các phần tử chưa được khởi tạo trong **a**, **b** sẽ nhận giá trị 0.

11. Cho một ma trận nguyên kích thước $m*n$. Tính:

- (a) Tổng tất cả các phần tử của ma trận.
- (b) Tổng tất cả các phần tử dương của ma trận.
- (c) Tổng tất cả các phần tử âm của ma trận.
- (d) Tổng tất cả các phần tử chẵn của ma trận.
- (e) Tổng tất cả các phần tử lẻ của ma trận.

12. Cho một ma trận thực kích thước $m*n$. 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 xâu 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 xâu 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 xâu vào, in ra "dung" nếu xâu vào là "Ha Noi". Ngược lại in ra "sai".
18. Nhập xâu. Không phân biệt viết hoa hay viết thường, hãy in ra các kí tự có mặt trong xâu và số lần xuất hiện của nó (ví dụ xâu "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 xâu. Tính số từ có trong xâu. In mỗi dòng một từ.

bản nháp, bản nháp,
bản nháp, bản nháp,

Chương 6

Các kiểu dữ liệu trùu tượng

6.1 Kiểu dữ liệu trùu 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 kí 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 kề 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 trường) 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
4 using namespace std;
5
6 int main()
7 {
8     const int NUM_OF_STUDENTS = 3;
9     struct Date
10    {
11         int day, month, year ;
12    };
13    struct Student
14    {
15        char name[30];
16        Date birthday;
17        double mark;
18    };
19    Student monitor = { "Bill Gate", { 1, 11, 2001 }, 5.0 };
20    Student other_student;
21    other_student = monitor;
22    other_student.birthday.year = 2002;                  // Dat lai nam sinh
23
24    cout << "Name : " << other_student.name << endl;
25    cout << "Birthday: " << other_student.birthday.day << "/" << other_student.
26        birthday.month << "/" << other_student.birthday.year << endl;
27    cout << "Mark: " << other_student.mark << endl;
28
29    system("PAUSE");
30}
```

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
3 using namespace std;
4
5 // number of days of months
6 const int NUM_DAYS[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
7 struct Date
8 {
9     int day, month, year ;
10 };
11
12 //----- Func. returns 1 if year is bissextile_year and 0 if not
13 int bissextile_year(int year)
14 {
15     return (year%4 == 0 && year%100 != 0 || year%400 == 0)? 1 : 0;
16 }
17
18 //----- Func. returns number of days of any month
19 //----- (plus 1 if year is bissextile)
20 int num_days_of_month(int month, int year)
21 {
22     return NUM_DAYS[month] + ((month == 2) ? bissextile_year(year) : 0);
23 }
24
25 //----- Func. returns total number of days from 1/1/1 to the date
26 long DtoN(Date date)                                // DtoN: Date to Numeric
27 {
28     long res = 0;
29     for (int index = 1; index < date.year; index++)
30         res += 365 + bissextile_year(index);
31     for (int index = 1; index < date.month; index++)
32         res += num_days_of_month(index, date.year);
33     res += date.day;
34     return res;
35 }
36
37 //----- Func. returns a date that corresponds to a numeric
38 Date NtoD(long num_days)                           // NtoD: Numeric to Date
39 {
40     Date res;
41     res.year = 1;
42     while (num_days > 365 + bissextile_year(res.year))
43     {
44         num_days -= 365 + bissextile_year(res.year);
45         res.year++;
46     }
47     res.month = 1;
48     while (num_days > num_days_of_month(res.month, res.year))
49     {
50         num_days -= num_days_of_month(res.month, res.year);
51         res.month++;
52     }
53     res.day = num_days;
```

```

54     return res;
55 }
56
57 //----- Func. returns number of days from date1 to date2
58 long Distance_Dates(Date date1, Date date2)
59 {
60     return DtoN(date1) - DtoN(date2);
61 }
62
63 //----- Func. returns day of week of any date
64 void DoW(Date date, char dow[])           // DoW: Day of week
65 {
66     Date curdate = {1, 1, 2000};           // the date 1/1/2000 is Saturday
67     long dist = Distance_Dates(date, curdate);
68     int odd = dist % 7;
69     if (odd < 0) odd += 7;
70     switch (odd) {
71         case 0: strcpy(dow, "Saturday"); break;
72         case 1: strcpy(dow, "Sunday"); break;
73         case 2: strcpy(dow, "Monday"); break;
74         case 3: strcpy(dow, "Tuesday"); break;
75         case 4: strcpy(dow, "Wednesday"); break;
76         case 5: strcpy(dow, "Thursday"); break;
77         case 6: strcpy(dow, "Friday"); break;
78     }
79 }
80
81 /////////////////////////////////
82 int main()
83 {
84     Date your_birthday, today;
85     char dow_birthday[10], dow_today[10];
86     cout << "What date is your birthday ? (dd/mm/yyyy) " ;
87     cin >> your_birthday.day >> your_birthday.month >> your_birthday.year ;
88     cout << "And today ? (dd/mm/yyyy) " ;
89     cin >> today.day >> today.month >> today.year ;
90     DoW(your_birthday, dow_birthday);
91     DoW(today, dow_today);
92     cout << "You were born on " << dow_birthday << ". Today is " << dow_today << endl
93     ;
94     cout << "You were born " << Distance_Dates(today, your_birthday) << " days ago."
95     << endl;
96     system("PAUSE");
97     return 0;
98 }
```

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 số 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
3 using namespace std;
4 struct Fraction
5 {
6     int numerator ;
7     int denominator ;
8 };
9
10 void display(Fraction a, Fraction b, Fraction c, char computing_mark)
11 {
12     cout << a.numerator << "/" << a.denominator; cout << " " << computing_mark << " "
13     ;
14     cout << b.numerator << "/" << b.denominator; cout << " = ";
15     cout << c.numerator << "/" << c.denominator << endl;
16 }
17
18 Fraction add(Fraction a, Fraction b)
19 {
20     Fraction c;
21     c.numerator = a.numerator*b.denominator + a.denominator*b.numerator;
22     c.denominator = a.denominator*b.denominator;
23     return c;
24 }
25
26 Fraction sub(Fraction a, Fraction b)
27 {
28     Fraction c;
29     c.numerator = a.numerator*b.denominator - a.denominator*b.numerator;
30     c.denominator = a.denominator*b.denominator;
31     return c;
32 }
33
34 Fraction product(Fraction a, Fraction b)
35 {
36     Fraction c;
37     c.numerator = a.numerator*b.numerator;
38     c.denominator = a.denominator*b.denominator;
39     return c;
40 }
41
42 Fraction divide(Fraction a, Fraction b)
43 {
44     Fraction c;
45     c.numerator = a.numerator*b.denominator;
46     c.denominator = a.denominator*b.numerator;
47     return c;
48 }
49
50 int main()
51 {
52     Fraction a, b, c;
53     cout << "Enter fraction #1: " << endl ;
```

```

53     cout << "\tnumerator: "; cin >> a.numerator;
54     cout << "\tdenominator: "; cin >> a.denominator;
55     cout << "Enter fraction #2: " << endl ;
56     cout << "\tnumerator: "; cin >> b.numerator;
57     cout << "\tdenominator: "; cin >> b.denominator;
58     cout << "Results:\n";
59     // Compute and display a + b
60     c = add(a, b);
61     display(a, b, c, '+');
62     // Compute and display a - b
63     c = sub(a, b);
64     display(a, b, c, '-');
65     // Compute and display a * b
66     c = product(a, b);
67     display(a, b, c, '*');
68     // Compute and display a / b
69     c = divide(a, b);
70     display(a, b, c, ':');
71     system("PAUSE");
72     return 0;
73 }

```

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 hàm chính, làm việc với danh sách
2 void Make_List();
3 void Display_List();
4 void Update_List();
5 void Insert_List();
6 void Append_List();
7 void Remove_List();
8 void Sort_List();
9 void Count_List();
10 // Nhóm hàm làm việc với một sinh viên
11 Student New_student();
12 void Display_student(Student x);
13 void Update_student(Student &x);
14 // Nhóm hàm phục vụ
15 void Set_List();           // only for testing funtions
16 void swap(Student &x, Student &y);
17 void Get_firstname(const char name[], char firstname[]);
18 void Sort_List_by_Name();  // increasing on name
19 void Sort_List_by_Mark();  // decreasing on mark

```

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)
18         cout << "Male" << "\t" ;
19     else
20         cout << "Female" << "\t" ;
21     cout << x.mark;
22     cout << endl;
23 }
```

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 câu 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
26                 .birthday.month >> copy.birthday.year ; break;
27             case 3: cout << "Enter new sex: " ; cin >> copy.sex ; break;
28             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 system("PAUSE");
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    system("PAUSE") ;
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     system("PAUSE") ;
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    system("PAUSE") ;
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 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 tráo 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    system("PAUSE") ;
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    system("PAUSE") ;
13 }
```

Hình 6.14: Hàm Count_List().

Và cuối cùng, để tránh tính 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 } ;
4     List[0] = x ;
5     Student y = { "Nguyen Thi Hanh", { 12, 4, 1977 }, 1, 6.5 } ;
6     List[1] = y ;
7     Student z = { "Le Nguyen Hanh", { 6, 11, 1991 }, 0, 5.2 } ;
8     List[2] = z ;
9     Student t = { "Cao Thuy Linh", { 28, 8, 2001 }, 1, 9.5 } ;
10    List[3] = t ;
11    num_students = 4 ;
12 }
13
14 void Make_List()
15 {
16     cout << "Enter the number of students: ";
17     cin >> num_students;
18     for (int index = 0; index < num_students; index++)
19     {
20         cout << "Enter data values of student #" << index+1 << ":\n";
21         List[index] = New_student() ;
22     }
23 }
```

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 |
| Nguyen Thi Hanh | 6.5 | Nguyen Thi Hanh | 6.5 | Tran Thanh Son |
| Le Nguyen Hanh | 5.2 | Cao Thuy Linh | 9.5 | Nguyen Thi Hanh |
| Cao Thuy Linh | 9.5 | Tran Thanh Son | 8.4 | Le Nguyen Hanh |

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 nhô càng tốt theo triết lý của thiết kế top-down và chia-dẻ-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 kĩ 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 kẽ 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 ưa chuộng cách khai báo các phương thức (methods) trước sau đó mới đến khai báo biến (variables).

¹ #include <iostream>

²

³ using namespace std;

⁴

```

5 class Date
6 {
7 public:
8     void Display();           // method
9     int day, month, year;    // variable
10 };
11
12 class Student
13 {
14 public:
15     void Display();           // method
16 private:
17     char name[30];           // variable
18     Date birthday;          // variable
19     int sex;                 // variable
20     double mark;             // variable
21 };

```

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. Hiện 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 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 số, 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
3 using namespace std;
4
5 class Date
6 {
7 public:
8     void Display();           // method
9     int day, month, year;    // variable
10};
11
12 class Student
13 {
14 public:
15     void Display();           // method
16 private:
17     char name[30];           // variable
18     Date birthday;          // variable
19     int sex;                 // variable
20     double mark;             // variable
```

```

21 };
22
23 void Date :: Display()
24 {
25     cout << day << "/" << month << "/" << year ;
26 }
27
28 void Student :: Display()
29 {
30     cout << name << "\t" ;
31     if (sex == 0) cout << "Male" << "\t" ; else cout << "Female" << "\t" ;
32     cout << mark << endl;
33 }
34
35 int main()
36 {
37     Date holiday, birthday ;
38     holiday.day = 1 ;
39     holiday.month = 5 ;
40     holiday.year = 2015 ;
41     holiday.Display() ;
42     cout << endl;
43     system("PAUSE");
44     return 0 ;
45 }

```

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 xâu ký 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
3 using namespace std;
4
5 class Date
6 {
7 public:
8     void Display();
9     int day, month, year ;
10 };
11
12 void Date :: Display()
13 {
14     cout << day << "/" << month << "/" << year ;
15 }
16
17 int main()
18 {
19     Date holiday, birthday ;
20     holiday.day = 1 ;
21     holiday.month = 5 ;
22     holiday.year = 2015 ;
23     holiday.Display() ;
24     cout << endl;
25     system("PAUSE");
26     return 0 ;
27 }
```

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 tương 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
3 using namespace std;
4
5 class Date
6 {
7 public:
8     void Display();
9     void Set(int dd, int mm, int yy);
10 private:
11     int day, month, year ;
12 };
13
14 void Date :: Display()
15 {
16     cout << day << "/" << month << "/" << year ;
17 }
18
19 void Date :: Set(int dd, int mm, int yy)
20 {
21     day = dd ;
22     month = mm ;
23     year = yy ;
24 }
25
26 int main()
27 {
28     Date holiday ;
29     holiday.Set(1, 5, 2015) ;
30     holiday.Display() ;
31     cout << endl;
32     system("PAUSE");
33     return 0 ;
34 }

```

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
3 using namespace std;
4
5 class Date
6 {
7 public:
8     void Display();
9     void Set(int dd, int mm, int yy);
10    int getDay() { return day; }
11    int getMonth() { return month; }
12    int getYear() { return year; }
13 private:
14     int day, month, year ;
15 };
16
17 void Date :: Display()           // code trong vi du truoc
18 void Date :: Set(int dd, int mm, int yy)   // code trong vi du truoc
19
20 int main()
21 {
22     Date holiday ;
23     holiday.Set(1, 5, 2015) ;
24     cout << holiday.getDay() << "/" << holiday.getMonth() << " of every year is
25         holiday" ;
26     cout << endl;
27     system("PAUSE");
28 }
```

Hình 6.22: Lớp Date với các hàm truy xuất dữ liệu.

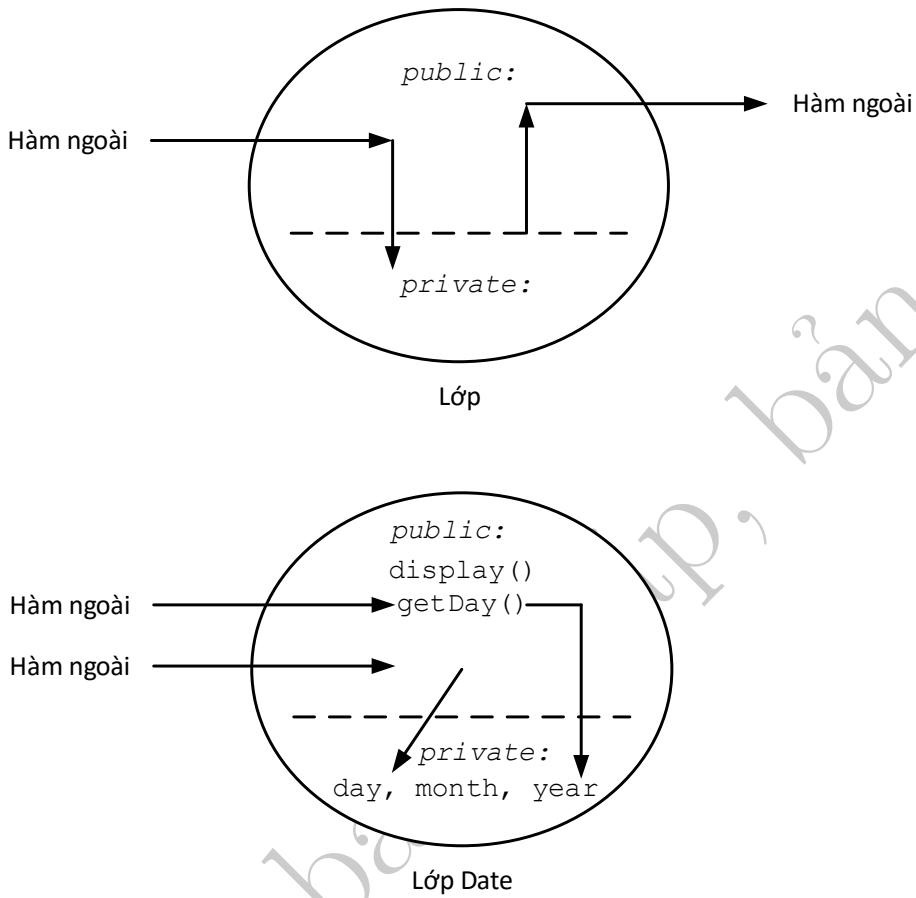
Tóm lại :

```
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 (đi 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.



Hình 6.23: Minh họa cơ chế đóng gói.

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 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.

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

Ngoài 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, chòng (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
25 //----- Dat gia tri cho sinh vien
26 void Student::Set(char nme[], int dd, int mm, int yy, int sx, double mrk)
27 {
28     strcpy(name, nme) ;
29     birthday.Set(dd, mm, yy);                                // goi ham Set cua Date
30     sex = sx;
31     mark = mrk;
32 }
33
34 //----- Tao moi mot sinh vien tu ban phim
35 void Student::New()
36 {
37     int mm, dd, yy ;
38     fflush(stdin) ;
39     cout << " Full name: "; cin.getline(name, 30) ;
40     cout << " Birthday: "; cin >> dd >> mm >> yy ; birthday.Set(dd, mm, yy) ;
41     cout << " Sex (0: Male, 1: Female): "; cin >> (sex) ;
42     cout << " Mark: "; cin >> (mark) ;
43 }
44
45 //----- Cap nhat thong tin mot sinh vien
46 void Student::Update()
47 {
48     Date brday;                                         // su dung lop khac trong phuong thuc
49     char nme[30]; int dd, mm, yy; int sx; double mrk;    // cac gia tri moi

```

```

49 // Lay noi dung cua doi tuong ra cac bien 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 system("PAUSE");
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 đối, 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 đối thì câu lệnh này sẽ bị chương trình dịch báo lỗi (nói chung, nên có hàm tạo không đối)

```
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 trả 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
4 using namespace std;
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 int main()
38 {
39
40     Date d(2,3,4);
41     Date t(d);
42     t.Display();
43     cout << endl;
44     system("PAUSE");
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 trỏ 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 huỷ.

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 số phức và giá trị trả lại là một số phức. Khi một số phức `x` gọi đến hàm này với đối số `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 = Plus(x, y)`, tuy nhiên hàm này có hai đối số 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
3 using namespace std;
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     system("PAUSE");
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 * 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
3 using namespace std;
4 class Vector ;
5 class Matrix ;
6
7 class Vector
8 {
9 public:
10     void fillData();
11     void Display();
12     friend Vector Product(const Matrix &M, const Vector &v) ;
13 private:
14     int numElement;           // so phan tu cua vecto
15     double data[20];          // cac thanh phan cua vecto
16 };
17
18 class Matrix
19 {
20 public:
21     void fillData();
22     friend Vector Product(const Matrix &M, const Vector &v);
23 private:
24     int numRow, numCol;        // so dong, so cot
25     double data[20][20];       // cac phan tu cua ma tran
26 };
```

```

28 void Vector::fillData()
29 {
30     cout << "\nEnter number of vector elements: "; cin >> numElement ;
31     cout << "Enter elements of vector:\n";
32     for (int index = 1; index <= numElement; ++index)
33         cin >> data[index];
34 }
35
36 void Matrix::fillData()
37 {
38     cout << "\nEnter number of rows and columns of matrix: "; cin >> numRows >> numCol
39         ;
40     cout << "Enter elements of matrix:\n";
41     for (int index1 = 1; index1 <= numRows ; ++index1)
42         for (int index2 = 1; index2 <= numCol ; ++index2)
43             cin >> data[index1][index2];
44 }
45 void Vector::Display()
46 {
47     cout << "\nElements of vector is:\n";
48     for (int index = 1; index <= numElement; ++index)
49         cout << data[index] << " ";
50     cout << endl;
51 }
52
53 Vector Product(const Matrix &M, const Vector &v)
54 {
55     Vector res;
56     res.numElement = M.numRow; // so phan tu cua vecto tich la so dong cua M
57     for (int index = 1; index <= M.numRow; ++index)
58     {
59         res.data[index] = 0;
60         for (int k = 1; k <= M.numCol; ++k)
61             res.data[index] += M.data[index][k] * v.data[k];
62     }
63     return res;
64 }
65
66 int main()
67 {
68     Matrix Mat;
69     Vector vec;
70     Vector result; // result = M * v
71     Mat.fillData();
72     vec.fillData();
73     result = Product(Mat, vec);
74     result.Display();
75     system("PAUSE");
76     return 0;
77 }
```

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 chẵng 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
3 using namespace std;
4
5 // number of days of months
6 const int NUM_DAYS[13] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
7 class Date
8 {
9 public:
10     Date() {day = month = year = 1; }
11     Date(int new_day, int new_month, int new_year) {day = new_day; month = new_month;
12         year = new_year; }
13     friend long DtoN(Date date);
14     friend Date NtoD(long num_days);
15     friend void DoW(Date date, char dow[]);
16     friend void Moon_Year(Date date, char mc[]);
17     friend long operator-(Date date1, Date date2);
18     friend Date operator-(Date date, long n);
19     friend Date operator+(Date date, long n);
20     friend bool operator==(Date date1, Date date2);
21     friend istream& operator>>(istream& is, Date& date);

```

```

21     friend ostream& operator<<(ostream& os, Date date);
22     friend Date operator++(Date& date);
23     friend Date operator--(Date& date);
24     friend Date operator++(Date& date, int n);
25     friend Date operator--(Date& date, int n);
26 private:
27     int day, month, year;
28 };

```

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 bỏ 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 // Phep toan nhap
2 istream& operator>>(istream& is, Date& date)
3 {
4     int dd, mm, yy;
5     cin >> dd >> mm >> yy;
6     date = Date(dd, mm, yy);
7     return is;
8 }
9
10 // Phep toan hien thi
11 ostream& operator<<(ostream& os, Date date)
12 {
13     os << date.day << "/" << date.month << "/" << date.year;

```

```

14     return os;
15 }
```

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 chằng 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 đổi, lớp có 4 phương thức :

- Phương thức **Display** (để hiển thị đối tượng ra màn hình) không đổi, 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 đổi, có giá trị trả lại kiểu T. Được cài đặt bên ngoài khai báo lớp.
- Phương thức **Swap** (để tráo đổi hai giá trị của đối tượng) không đổi, 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 tráo đổi giữa chúng.

```

1 #include <iostream>
2
3 using namespace std;
4
5 template <class T>
6 class Pair
7 {
8 public:
9     Pair() { value1 = value2 = 0; }
10    void Set(T first, T second) {value1 = first; value2 = second; }
11    T getMax() ;
12    void Swap();
13    void Display() { cout << "(" << value1 << ", " << value2 << ")" ; }
14 private:
15     T value1, value2;
16 };
17
18 template <class T>
19 T Pair<T>::getMax()
20 {
21     T res;
22     res = (value1 > value2) ? value1 : value2;
23     return res;
24 }
25
26 template <class T>
27 void Pair<T>::Swap()
28 {
29     T temp;
30     temp = value1; value1 = value2; value2 = temp;
31 }
32
33 int main()
34 {
35     Pair<int> my_object; my_object.Set(3, 5);
36     cout << "My pair of integers is " ; my_object.Display() ; cout << " and " <<
37         my_object.getMax() << " is greatest. ";
38     my_object.Swap();
39     cout << "They was swapped to " ; my_object.Display() ; cout << endl;
40
41     Pair<char> your_object; your_object.Set('B', 'D');
42     cout << "Your pair of characters is " ; your_object.Display() ; cout << " and "
43         << your_object.getMax() << " is greatest. ";
44     your_object.Swap();
45     cout << "They was swapped to " ; your_object.Display() ; cout << endl;
46
47     system("PAUSE");
48     return 0;
49 }
```

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++.

6.4 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:

- (a) Bổ sung thêm khai báo hàm tạo không đổi vào lớp trên
- (b) 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.

bản nháp, bản nháp,
bản nháp, bản nháp,

Chương 7

Con trỏ và bộ nhớ

Con trỏ là một công cụ mà một số ngôn ngữ lập trình bậc cao, đặc biệt là C và C++, cung cấp để cho phép chương trình quản lý và tương tác trực tiếp với bộ nhớ. Nó giúp chương trình linh động và hiệu quả. Tuy nhiên, sử dụng con trỏ cũng dễ dẫn đến sai sót và khó phát hiện ra lỗi.

7.1 Quản lý bộ nhớ máy tính

Để hiểu về con trỏ, trước tiên ta sẽ tìm hiểu về bộ nhớ máy tính. Ta có thể hình dung bộ nhớ máy tính là một loạt các ô nhớ nối tiếp nhau, mỗi ô nhớ có kích thước nhỏ nhất là một byte. Các ô nhớ đơn này được đánh số liên tục, ô nhớ sau có số thứ tự lớn hơn số thứ tự của ô nhớ liền trước là 1. Tức là ô nhớ 1505 sẽ đứng sau ô nhớ 1504 và đứng trước ô nhớ 1506 (xem Hình 7.1). Số thứ tự của ô nhớ có thể được gọi là địa chỉ của ô nhớ đó.

| Địa chỉ bộ nhớ | Nội dung ô nhớ cỡ 1 byte | Tên biến |
|----------------|--------------------------|----------|
| 1503 | | |
| 1504 | 'H' | greeting |
| 1505 | 'e' | |
| 1506 | 'l' | |
| 1507 | 'l' | |
| 1508 | 'o' | |
| 1509 | '\0' | |
| 1510 | ... | |

Hình 7.1: Biến greeting – xâu kí tự "Hello" – trong bộ nhớ.

7.2 Biến và địa chỉ của biến

Biến trong một chương trình là một vùng bộ nhớ được dùng để lưu trữ dữ liệu. Việc truy nhập dữ liệu tại các ô nhớ đó được thực hiện thông qua tên biến. Tức là, ta không phải quan tâm đến vị trí vật lý của ô bộ nhớ. Khi một biến được khai báo, phần bộ nhớ cần thiết sẽ được cấp phát cho nó tại một vị trí cụ thể trong bộ nhớ, vị trí đó được gọi là địa chỉ bộ nhớ của biến đó. Ví dụ, trong Hình 7.1, biến greeting có địa chỉ là 1504. Thông thường, hệ điều hành sẽ quyết định vị trí của biến trong bộ nhớ khi chương trình chạy. Để lấy địa chỉ của một biến, trong C++ ta sử dụng toán tử tham chiếu địa chỉ (`&`). Khi đặt toán tử tham chiếu trước tên của một biến, ta có một biểu thức có giá trị là địa chỉ của biến đó trong bộ nhớ. Ví dụ:

```
short apples = 9; // khai bao bien apples co gia tri 9
cout << apples; // hien ra gia tri cua bien apples bang 9
cout << &apples; // hien ra dia chi cua bien apples
```

Trong thực tế, trước khi chương trình chạy, chúng ta không thể biết được địa chỉ của biến apples.

7.3 Biến con trỏ

Biến con trỏ là một loại biến đặc biệt, được dùng để lưu giữ địa chỉ của các vùng nhớ. Tức là, giá trị của một biến con trỏ là địa chỉ của một ô nhớ trong bộ nhớ. Hay nói một cách hình tượng, biến con trỏ chỉ đến một ô nhớ trong bộ nhớ, nó được dùng để gián tiếp truy nhập đến ô nhớ đó.

Ví dụ, giả sử ta có biến apples kiểu int nằm tại địa chỉ 0x2a52ec, biến con trỏ ptrApples đang chỉ tới biến apples. Điều đó có nghĩa biến ptrApples đang có giá trị 0x2a52ec và điều quan trọng là có thể dùng ptrApples để gián tiếp truy nhập tới biến apples bằng cách dùng biểu thức (`*ptrApples`), xem Hình 7.2.



Hình 7.2: Con trỏ ptrApples chỉ tới biến apples, gián tiếp truy nhập ô nhớ apples.

Cũng như các loại biến khác, biến con trỏ cần được khai báo trước khi sử dụng. Với các biến apples và ptrApples nói trên, có thể khai báo bằng lệnh sau:

```
int *ptrApples, apples;
```

trong đó, biến ptrApples được khai báo thuộc kiểu `int*` (con trỏ tới một giá trị `int`), còn biến apples được khai báo thuộc kiểu `int`. Lưu ý là dấu `*` kí hiệu kiểu con trỏ chỉ áp dụng duy nhất cho biến nằm ngay sau nó chứ không áp dụng cho tất cả các biến nằm sau, mỗi biến con trỏ trong lệnh khai báo cần có một dấu `*` đặt trước nó. Ví dụ, lệnh sau khai báo hai biến con trỏ:

```
int *ptrApples, *ptrPineapples;
```

Con trỏ cần được khởi tạo bằng giá trị 0 (hay `NULL`) hoặc một địa chỉ ngay tại lệnh khai báo hoặc bằng một lệnh gán. Một con trỏ có giá trị 0 hay `NULL` (một hằng số tương đương với 0) được

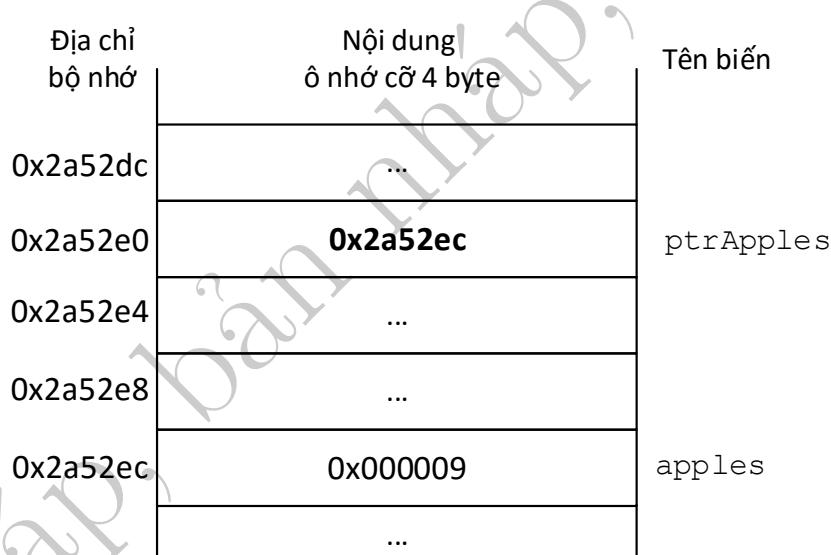
định nghĩa trong tệp thư viện `iostream`) không chỉ tới ô nhớ nào và được gọi là con trỏ `null`. Một con trỏ chưa được khởi tạo chỉ đến một vùng bộ nhớ không xác định hoặc chưa được khởi tạo, việc vô tình dùng con trỏ chưa được khởi tạo để truy nhập bộ nhớ là một lỗi lập trình khó phát hiện, có thể dẫn đến dữ liệu chương trình bị sửa làm chương trình chạy sai hoặc gây sự cố làm sập chương trình. Đây là một trong những đặc điểm không an toàn của ngôn ngữ C/C++. Do đó, lập trình viên nên chú ý khởi tạo con trỏ ngay sau khi khai báo.

Để truy nhập dữ liệu tại ô nhớ mà biến con trỏ chỉ đến ta sử dụng toán tử `*`. Ví dụ, ta dùng biểu thức `*ptrApples` để truy nhập biến `apples` mà `ptrApples` đang trỏ tới. Có thể dùng biểu thức `*ptrApples` ở bên trái cũng như bên phải của phép gán, nghĩa là để đọc cũng như để ghi giá trị vào ô nhớ `apples`. Nếu dùng phép toán truy nhập `*` cho con trỏ `null`, chương trình sẽ gặp lỗi run-time và dừng lập tức.

Hình 7.4 minh họa về việc khai báo và sử dụng biến con trỏ. Trong ví dụ, ta khai báo một biến con trỏ `ptrApples` dùng để lưu giữ địa chỉ ô nhớ của biến `apples`. Lệnh

```
ptrApples = &apples;
```

sẽ gán địa chỉ của biến `apples` cho biến con trỏ `ptrApples` và tạo hiệu ứng tương tự như trong Hình 7.2 và Hình 7.3.



Hình 7.3: Biến con trỏ `ptrApples` có giá trị là địa chỉ của biến `apples`.

Lưu ý, các tác động trên ô nhớ biến con trỏ `ptrApples` chỉ đến cũng chính là các tác động được thực hiện trên biến `apples` và ngược lại. Cụ thể là lệnh

```
apples++;
```

sẽ tăng giá trị của biến `apples` lên một, đồng nghĩa với việc tăng giá trị ở ô nhớ do biến con trỏ `ptrApples` chỉ đến lên 1. Tương tự, lệnh

```
*ptrApples += 1;
```

sẽ tăng giá trị tại ô nhớ mà biến con trỏ `ptrApples` thêm 1, đồng nghĩa với việc tăng giá trị của biến `apples` thêm 1.

Chương trình trong Hình 7.4 còn minh họa việc in giá trị của con trỏ ra màn hình. Có thể dùng `cout` để in giá trị của con trỏ như đối với nhiều kiểu dữ liệu khác. Tuy nhiên, định dạng của

giá trị in ra phụ thuộc vào từng trình biên dịch, một số hệ thống in ra các giá trị con trỏ ở dạng hệ cơ số 16 (như trong hình), các hệ thống khác sử dụng định dạng hệ thập phân.

```

1 //Chuong trinh minh hoa viec thao tac voi con tro
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     int apples = 9;
9     cout << "apples: " << apples << endl;
10    cout << "Address of apples: " << &apples << endl;
11
12    int *ptrApples;
13    ptrApples = &apples;
14    cout << "ptrApples: " << ptrApples << endl;
15    cout << "Value at ptrApples: " << *ptrApples << endl;
16
17    apples++;
18    cout << "ptrApples: " << ptrApples << endl;
19    cout << "Value at ptrApples: " << *ptrApples << endl;
20
21    *ptrApples +=1;
22    cout << "ptrApples: " << ptrApples << endl;
23    cout << "Value at ptrApples: " << *ptrApples << endl;
24
25    cin.get(); // Dung de xem man hinh ket qua
26
27    return 0;
28 }
```

Hình 7.4: Khai báo và sử dụng biến con trỏ.

Output hình 7.4

```

apples: 9
Address of apples: 0x24caf4
ptrApples: 0x24caf4
Value at ptrApples: 9
ptrApples: 0x24caf4
Value at ptrApples: 10
ptrApples: 0x24caf4
Value at ptrApples: 11
```

Một trong những ứng dụng thường của con trỏ là làm con chạy trên chuỗi các ô nhớ liên tiếp, chẳng hạn như mảng. Chương trình trong Hình 7.5 minh họa việc đó. Trong chương trình là một vòng lặp có nhiệm vụ duyệt và in ra toàn bộ các phần tử của mảng `score`. Con chạy trong vòng lặp là con trỏ `ptr` với giá trị ban đầu là địa chỉ của phần tử đầu tiên mảng `score`. Tại mỗi lần lặp, phần tử mảng hiện được `ptr` trỏ tới được đọc bằng biểu thức `*ptr` rồi in ra màn hình, sau đó `ptr` được tăng thêm 1 (`ptr++`) với hiệu quả là nó sẽ trỏ tới phần tử tiếp theo trong mảng. Vòng lặp dừng khi `ptr` bắt đầu trỏ vượt ra xa hơn vị trí phần tử cuối mảng (điều kiện lặp là `ptr <= &score[6]`).

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int score[7] = {1, 2, 3, 4, 5, 6, 7};
8
9     for (int *ptr = &score[0]; ptr <= &score[6]; ptr++)
10        cout << *ptr << " ";
11    cin.get();
12    return 0;
13 }

```

Hình 7.5: Dùng con trỏ làm biến lặp duyệt mảng.

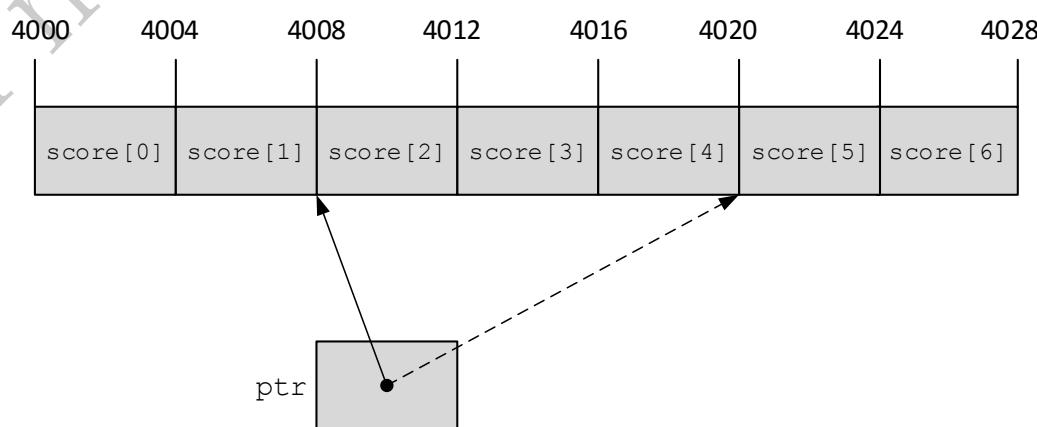
Output hình 7.5

1 2 3 4 5 6 7

Hình 7.5 còn minh họa các phép toán cộng và trừ con trỏ với một số nguyên và các phép so sánh con trỏ.

Có thể dùng các phép so sánh bằng/khác và lớn hơn/nhỏ hơn cho con trỏ. Các phép toán này so sánh các địa chỉ lưu trong các con trỏ. Một phép so sánh thường gấp là so sánh một con trỏ với 0 để kiểm tra xem đó có phải là con trỏ NULL hay không. Các phép so sánh lớn hơn/nhỏ hơn đối với các con trỏ là vô nghĩa trừ khi chúng đang trỏ tới các phần tử của cùng một mảng. Trong trường hợp đó, kết quả của phép so sánh cho thấy con trỏ này trỏ tới phần tử mảng có chỉ số cao hơn con trỏ kia. Trong Hình 7.5, con trỏ ptr được so sánh với địa chỉ của phần tử cuối cùng xem nó đã duyệt đến vị trí đó hay chưa.

Các phép toán cộng trừ đối với con trỏ không có nghĩa cộng hay trừ chính số nguyên đó vào giá trị lưu trong biến con trỏ. Thực tế, giá trị được cộng vào hay trừ đi là số nguyên đó nhân với kích thước của kiểu dữ liệu mà con trỏ trỏ tới. Ví dụ, giả sử con trỏ ptr kiểu int đang trỏ tới phần tử mảng score[2] và kiểu dữ liệu int có kích thước 4 byte, lệnh `ptr += 3;` sẽ gán cho ptr giá trị 4020 (nghĩa là $4008 + 3 * 4$) làm nó trỏ tới score[5]. Xem minh họa tại Hình 7.6.



Hình 7.6: Phép cộng 3 vào con trỏ ptr.

7.4 Mảng và con trỏ

Trong C++, mảng và con trỏ có quan hệ chặt chẽ với nhau và gần như có thể dùng thay thế cho nhau. Một cái tên mảng có thể được coi như một hằng con trỏ, có thể dùng biến con trỏ trong bất cứ cú pháp nào của mảng. Giả sử, khi khai báo một mảng

```
int score[7];
```

ta đã khai báo một vùng nhớ liên tiếp gồm 7 ô, mỗi ô chứa một số nguyên. Trong C++, mảng được cài đặt bằng con trỏ. Cụ thể là, tên mảng (`score`) có thể được coi là một hằng con trỏ kiểu `int` trỏ tới ô nhớ đầu tiên trong mảng (`score[0]`). Để truy cập đến phần tử có chỉ số `i` trong mảng `score`, ta có thể sử dụng `score[i]` hoặc `*(score + i)`. Ví dụ, `*score` và `score[0]` đều có ý nghĩa là phần tử đầu tiên trong mảng.

Cách dùng kiểu con trỏ để thao tác với mảng được minh họa trong Hình 7.7. Trong đó vòng lặp lần lượt in ra các phần tử trong mảng, với biểu thức `*(score + i)` được dùng để truy nhập phần tử thứ `i` trong mảng.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     int score[7] = {1, 2, 3, 4, 5, 6, 7};
8
9     for (int count = 0; count < 7; count++)
10        cout << *(score + count) << " ";
11    cin.get();
12    return 0;
13 }
```

Hình 7.7: Sử dụng tên mảng như con trỏ.

Ngược lại, nếu ta có `ptrScore` là một con trỏ kiểu `int`, lệnh sau gán giá trị của `score` cho `ptrScore` làm nó trỏ tới vị trí đầu tiên trong mảng `score`.

```
ptrScore = score;
```

Lệnh trên tương đương với

```
ptrScore = &score[0];
```

Sau đó ta có thể dùng biến con trỏ `ptrScore` để thao tác mảng `score` như thể `ptrScore` là một tên mảng.

Hình 7.8 minh họa việc đó, đây chỉ là sửa đổi nhỏ từ chương trình trong Hình 7.7 để thực hiện cùng một nhiệm vụ nhưng lại dùng `ptrScore` với cú pháp mảng thay vì dùng tên mảng `score` với cú pháp con trỏ.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
```

```

7   int score[7] = {1, 2, 3, 4, 5, 6, 7};
8   int *ptrScore = score;
9
10  for (int count = 0; count < 7; count++)
11      cout << ptrScore[count] << " ";
12  cin.get();
13  return 0;
14 }
```

Hình 7.8: Sử dụng con trỏ như tên mảng

7.5 Bộ nhớ động

Trong tất cả các chương trình ví dụ trước, ta mới chỉ dùng bộ nhớ trong phạm vi các biến được khai báo trong mã nguồn với kích thước được xác định trước khi chương trình chạy. Với C++, tất cả các biến được khai báo trong chương trình đều nằm trong bộ nhớ tĩnh và có kích thước xác định trước. Nếu chúng ta cần đến lượng bộ nhớ mà kích thước chỉ có thể biết được khi chương trình chạy thì sao? Chẳng hạn khi chương trình của ta cần làm việc với một mảng mà kích thước của nó không xác định được khi chương trình chưa chạy, và ta cũng không thể ước lượng độ dài tối đa mà cần đọc dữ liệu vào để xác định độ dài cần thiết.

Giải pháp là sử dụng bộ nhớ động. Không như các biến thông thường chỉ cần khai báo, các biến nằm trong vùng bộ nhớ động cần được cấp phát khi muốn sử dụng và giải phóng một cách tường minh khi không còn được dùng đến. Trong C++, con trỏ cùng với các toán tử `new` và `delete` là công cụ để xin cấp phát và giải phóng dữ liệu nằm trong bộ nhớ động.

7.5.1 Cấp phát bộ nhớ động

Khi một biến con trỏ được khai báo, ví dụ

```
int *ptrApples;
```

giá trị của biến con trỏ `ptrApples` chưa được xác định, nghĩa là nó chưa trỏ vào một vùng nhớ xác định trước nào. Chúng ta cũng có thể xin cấp phát một vùng nhớ động, và vùng nhớ này được quản lý (trở tới) bởi biến con trỏ `ptrApples` bằng cách sử dụng toán tử cấp phát bộ nhớ động `new` như sau:

```
ptrApples = new int;
```

Lệnh này sẽ cấp phát một vùng nhớ đủ cho một giá trị kiểu `int`, địa chỉ vùng nhớ này được ghi vào biến `ptrApples`. Quá trình cấp phát bộ nhớ này được thực hiện trong quá trình chạy chương trình và được gọi là cấp phát bộ nhớ động. Sau lệnh cấp phát trên, ta có thể tiến hành lưu giữ, cập nhật giá trị ở vùng nhớ này thông qua địa chỉ của nó (hiện lưu tại biến `ptrApples`). Nếu lệnh cấp phát không thành công (chẳng hạn vì không còn đủ bộ nhớ), `ptrApples` sẽ nhận giá trị 0. Lập trình viên cần chú ý đề phòng những tình huống này.

7.5.2 Giải phóng bộ nhớ động

Do lượng bộ nhớ động là có hạn, dữ liệu đặt trong bộ nhớ động nên được giải phóng khi dùng xong để dành bộ nhớ cho các yêu cầu cấp phát tiếp theo.

Việc giải phóng vùng nhớ được thực hiện bằng toán tử `delete`. Ví dụ, lệnh sau giải phóng ô nhớ kiểu `int` mà con trỏ `ptrApples` hiện đang trỏ tới.

```
delete ptrApples;
```

Sau khi được giải phóng, vùng nhớ đó có thể được tái sử dụng cho một lần cấp phát bộ nhớ động khác. Hình 7.9 minh họa về việc cấp phát, sử dụng, và giải phóng bộ nhớ động.

```
1 //Chuong trinh minh hoa viec cap phat va xoa bo nho cho con tro
2 #include <iostream>
3
4 using namespace std;
5
6 int main()
7 {
8     int *ptrApples;
9     ptrApples = new int;
10
11    *ptrApples = 5;
12
13    cout << "Value at ptrApples: " << *ptrApples << endl;
14
15    *ptrApples += 2;
16    cout << "Value at ptrApples: " << *ptrApples << endl;
17    delete ptrApples;
18
19    cin.get(); // Dung de xem man hinh ket qua
20    return 0;
21 }
```

Hình 7.9: Cấp phát sử dụng và giải phóng bộ nhớ động.

Output hình 7.9

```
Value at ptrApples: 5
Value at ptrApples: 7
```

Lưu ý: nếu vì sơ xuất của người lập trình hay vì lý do nào đó mà tại một thời điểm nào đó chương trình không còn lưu hoặc có cách nào tính ra địa chỉ của một vùng nhớ được cấp phát động, thì vùng bộ nhớ này được xem là bị "mất". Nghĩa là chương trình không có cách gì truy nhập hay giải phóng vùng bộ nhớ này nữa, sau khi chương trình kết thúc nó mới được hệ điều hành thu hồi. Ví dụ, đoạn chương trình sau làm thất thoát phần bộ nhớ được cấp phát do lệnh `new` thứ nhất, do con trỏ duy nhất giữ địa chỉ của ô nhớ đó đã bị gán giá trị mới, ta không còn có thể lấy lại được địa chỉ của vùng nhớ đó để sử dụng hoặc giải phóng.

```
ptr1 = new int;
ptr2 = new int;
ptr1 = ptr2;
```

7.6 Mảng động và con trỏ

Trong các chương trình ví dụ từ trước đến giờ, kích thước của mảng phải được xác định trước khi khai báo. Tức là, kích thước của mảng là một hằng số cho trước. Trong nhiều trường hợp, chúng ta khó có thể xác định trước được kích thước của mảng ngay từ khi lập trình, mà kích thước này phụ thuộc vào dữ liệu đầu vào cũng như quá trình chạy chương trình.

Để khắc phục tình trạng trên, C++ cho phép chúng ta sử dụng con trỏ để tạo mảng có kích thước động nằm trong bộ nhớ động. Những mảng được tạo theo cách đó được gọi là mảng động, bởi kích thước mảng và việc cấp phát bộ nhớ cho mảng được xác định và tiến hành trong quá trình chạy chương trình. Việc khai báo mảng động được tiến hành thành hai bước:

1. **Khai báo con trỏ mảng:** Trước tiên, khai báo một biến con trỏ:

```
data_type *array_name;
```

2. **Xin cấp phát bộ nhớ:** Sau khi khai báo, chúng ta xin cấp phát bộ nhớ cho biến con trỏ bằng toán tử `new []` theo cấu trúc sau:

```
array_name = new data_type [size];
```

hệ thống sẽ cấp cho chương trình một vùng nhớ liên tiếp có thể chứa được size phần tử có kiểu `data_type`. Địa chỉ của phần tử đầu tiên được chỉ đến bởi biến `array_name`. Lưu ý, `size` có thể là một hằng số, hoặc là một biến.

Sau khi đã cấp phát bộ nhớ thành công, ta có thể đối xử với biến con trỏ `array_name` như một mảng thông thường với `size` phần tử. Tức là, để truy cập đến phần tử thứ `i` trong mảng, ta có thể sử dụng `array_name[i]` hoặc `*(array_name + i)`.

3. **Giải phóng bộ nhớ:** Khi chúng ta không sử dụng đến mảng động nữa, chúng ta phải tiến hành giải phóng bộ nhớ đã xin cấp bằng cách sử dụng toán tử `delete []` như sau:

```
delete [] array_name;
```

Lưu ý khi giải phóng bộ nhớ động cho một mảng, nếu ta sử dụng lệnh

```
delete array_name;
```

thì chỉ có ô nhớ `array_name[0]` được giải phóng, còn các ô nhớ tiếp theo của mảng không được giải phóng.

Hình 7.10 minh họa việc sử dụng mảng động để tính tổng điểm của một danh sách các môn học của một sinh viên được nhập vào từ bàn phím. Số lượng danh sách môn học không được xác định trước mà được người dùng nhập vào từ bàn phím.

```

1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
7     int numberCourses;
8     cout << "Enter the number of courses:" ;
9     cin >> numberCourses;

```

```

10
11     int *arr_courses;
12     // Cap phat bo nho cho mang arr_courses
13     arr_courses = new int[numberCourses];
14
15     for (int count = 0; count < numberCourses; count++){
16         cout << "Enter the mark of course #" << count << ": ";
17         cin >> arr_courses[count];
18     }
19
20     int totalMark = 0;
21     for (int count = 0; count < numberCourses; count++)
22         totalMark += arr_courses[count];
23
24     cout << "Total Mark: " << totalMark << endl;
25
26     //Giai phong bo nho cho mang arr_courses
27     delete [] arr_courses;
28
29     //cin.ignore();
30     system("PAUSE");
31     //cin.get(); // Dung de xem man hinh ket qua
32     return 0;
33 }
```

Hình 7.10: Mảng động và con trỏ.

Output hình 7.10

```

Enter the number of courses:3
Enter the mark of course #0: 7
Enter the mark of course #1: 8
Enter the mark of course #2: 5
Total Mark: 20
```

7.7 Truyền tham số là con trỏ

Như đã được giới thiệu trong Chương ??, mục ??, C++ có hai phương pháp truyền dữ liệu vào trong hàm: truyền bằng giá trị và truyền bằng tham chiếu. Trong phương pháp truyền bằng tham chiếu, Chương ??, mục ?? đã nói về cách dùng đối số là tham chiếu. Trong mục này, chúng tôi sẽ giới thiệu kiểu truyền bằng tham chiếu còn lại: dùng đối số là con trỏ.

Còn nhớ rằng các đối số là biến tham chiếu cho phép hàm được gọi sửa giá trị dữ liệu của hàm gọi. Các đối số tham chiếu còn cho phép chương trình truyền lượng dữ liệu lớn vào hàm mà không phải chịu chi phí cho việc sao chép dữ liệu như khi truyền bằng giá trị. Cũng như vậy, các đối số là con trỏ có thể được sử dụng để sửa đổi các biến của hàm gọi hoặc để truyền các đối tượng dữ liệu lớn trong khi tránh được việc sao chép dữ liệu.

Trong C++, con trỏ được dùng kèm với toán tử * để thực hiện cơ chế truyền bằng tham chiếu. Khi gọi một hàm, các đối số cần sửa giá trị được truyền vào trong hàm bằng địa chỉ của chúng (sử dụng toán tử địa chỉ &).

Chương trình trong Hình 7.11 minh họa cách sử dụng con trỏ để truyền dữ liệu vào hàm qua tham chiếu. Hàm `swap` có nhiệm vụ tráo đổi giá trị của hai biến. Nó nhận các tham số `x` và `y` là con trỏ tới các biến cần đổi giá trị, rồi thông qua `x` và `y` truy nhập gián tiếp tới các biến đó để sửa giá trị. Còn tại hàm `main`, khi muốn đổi giá trị của hai biến `a` và `b`, địa chỉ của hai biến này được truyền vào khi gọi hàm `swap`.

```

1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 void swap(int *x, int *y)
6 {
7     int tmp = *x;
8     *x = *y;
9     *y = tmp;
10 }
11
12 int main()
13 {
14     int a, b;
15     cout << "Enter the numbers a and b: ";
16     cin >> a >> b;
17     cout << "Before swapping, a = " << a << ", b = " << b << endl;
18
19     swap(&a,&b);
20
21     cout << "After swapping, a = " << a << ", b = " << b << endl;
22
23 //cin.ignore();
24 system("PAUSE");
25 //cin.get(); // Dung de xem man hinh ket qua
26 return 0;
27 }
```

Hình 7.11: Tham số của hàm là con trỏ.

Output hình 7.11

```

Enter the numbers a and b: 5 7
Before swapping, a = 5, b = 7
After swapping, a = 7, b = 5

```

Trong Hình 7.12, biến con trỏ `courses` được truyền vào hàm `getMark` và `calculateMark`. Dẫn đến tham số hình thức của hàm `getMark` và `calculateMark` chính là con trỏ tới mảng `courses` trong hàm `main`.

```

1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 void GetMark(int *arr_courses, int numberCourses)
6 {
7     for (int count = 0; count < numberCourses; count++){
8         cout << "Enter the mark of course #" << count << ": ";
9         cin >> arr_courses[count];
10    }
11 }
12
13 int CalculateMark(const int *arr_courses, int numberCourses)
14 {
15     int totalMark = 0;
16     for (int count = 0; count < numberCourses; count++)
17         totalMark += arr_courses[count];
18     return totalMark;
19 }
20
21 int main()
22 {
23     int numberCourses;
24     cout << "Enter the number of courses:" ;
25     cin >> numberCourses;
26
27     int *arr_courses;
28     // Cap phat bo nho cho mang arr_courses
29     arr_courses = new int[numberCourses];
30
31     GetMark(arr_courses, numberCourses);
32     int totalMark = CalculateMark(arr_courses, numberCourses);
33
34     cout << "Total Mark: " << totalMark << endl;
35
36     //Giai phong bo nho cho mang arr_courses
37     delete [] arr_courses;
38
39     system("PAUSE");
40     return 0;
41 }

```

Hình 7.12: Tham số của hàm là con trỏ.

Trong những trường hợp ta muốn dùng con trỏ để truyền lượng lớn dữ liệu vào hàm nhưng lại không muốn cho hàm sửa đổi dữ liệu, ta có thể dùng từ khóa `const` để thiết lập quyền của hàm đối với tham số. Xem ví dụ hàm `calculateMark` ở Hình 7.12, trong đó, từ khóa `const` ở phía trước tham số hình thức `courses`

```
int calculateMark (const int *courses, int numberCourses)
```

quy định rằng hàm `calculateMark` phải coi dữ liệu `int` mà `courses` trả về là hằng và không được phép sửa đổi.

Về cách sử dụng từ khóa `const` khi khai báo một biến con trỏ, ta có 4 lựa chọn:

- Không quy định con trỏ hay dữ liệu được trỏ tới là hằng

```
float * ptr;
```

- Quy định dữ liệu được trỏ tới là hằng, con trỏ thì được sửa đổi.

```
const float * ptr;
```

- Quy định dữ liệu trỏ tới không phải là hằng, nhưng biến con trỏ thì là hằng

```
float * const ptr;
```

- Quy định cả con trỏ lẫn dữ liệu nó trỏ tới đều là hằng.

```
const float * const ptr;
```

Sử dụng `const` cho những hoàn cảnh thích hợp là một phần của phong cách lập trình tốt. Nó giúp giảm quyền hạn của hàm xuống tới mức vừa đủ, chẳng hạn một hàm chỉ có nhiệm vụ đọc dữ liệu và tính toán thì không nên có quyền sửa dữ liệu, từ đó giảm nguy cơ của hiệu ứng phụ không mong muốn. Đây chính là một trong các cách thi hành nguyên tắc quyền ưu tiên tối thiểu (the principle of least privilege), chỉ cấp cho hàm quyền truy nhập dữ liệu vừa đủ để thực hiện nhiệm vụ của mình.

Một điểm quan trọng khác cần lưu ý là tình trạng con trỏ có giá trị `null` hoặc không xác định do chưa gán bằng địa chỉ của biến nào. Nếu con trỏ `ptr` có giá trị `null` hoặc không xác định thì việc truy nhập `*ptr` sẽ dẫn đến lỗi run-time hoặc lỗi logic cho chương trình. Ta cần chú ý khởi tạo giá trị của các biến con trỏ và kiểm tra giá trị trước khi truy nhập. Ngoài ra, còn có một lời khuyên là nên sử dụng biến tham chiếu thay cho con trỏ bất cứ khi nào có thể để tránh trường hợp con trỏ `null` hoặc có giá trị không xác định.

7.8 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 7.13.

```

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     system("PAUSE");
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 7.13: Ví dụ về con trỏ hàm so sánh 2 xâu ký tự.

Output hình 7.13

```

Input of s1 string: Hello
Input of s2 string: Hello John
Testing for equality.
Not Equal
```

Khi hàm check() trong hình 7.13 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. Đầu 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 trả 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 7.13 thành ví dụ trong hình 7.14. 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.

7.9 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.

7.9.1 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 7.15.

Nút

Một cấu trúc như hình 7.15 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 7.15 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 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 7.15, 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 7.15 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 7.15, nếu muốn thay đổi số nguyên trong nút đầu tiên từ 10 thành 12, ta có thể thực hiện như sau:

```
(*head).count = 12;
```

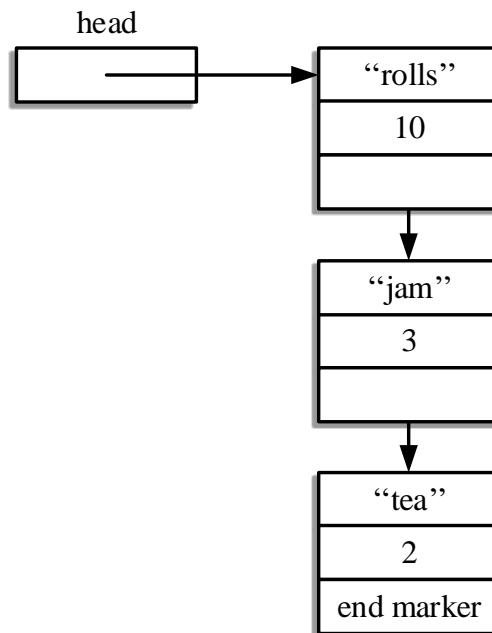
```

1 //Chuong trinh minh hoa ve con tro ham
2 //Con tro ham co the tro den 2 ham khac nhau NumCmp() va 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     system("PAUSE");
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 7.14: Lựa chọn hàm bằng con trỏ hàm.

Trong biểu thức ở bên trái của toán tử gán, biến **head** là biến con trỏ, nên ***head** sẽ trả tới địa chỉ của một nút, cụ thể trong trường hợp này là nút chứa chuỗi "**rolls**" và số nguyên **10**. Nút



Hình 7.15: Nút và con trỏ.

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 hóa việc truy cập 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 trả 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.

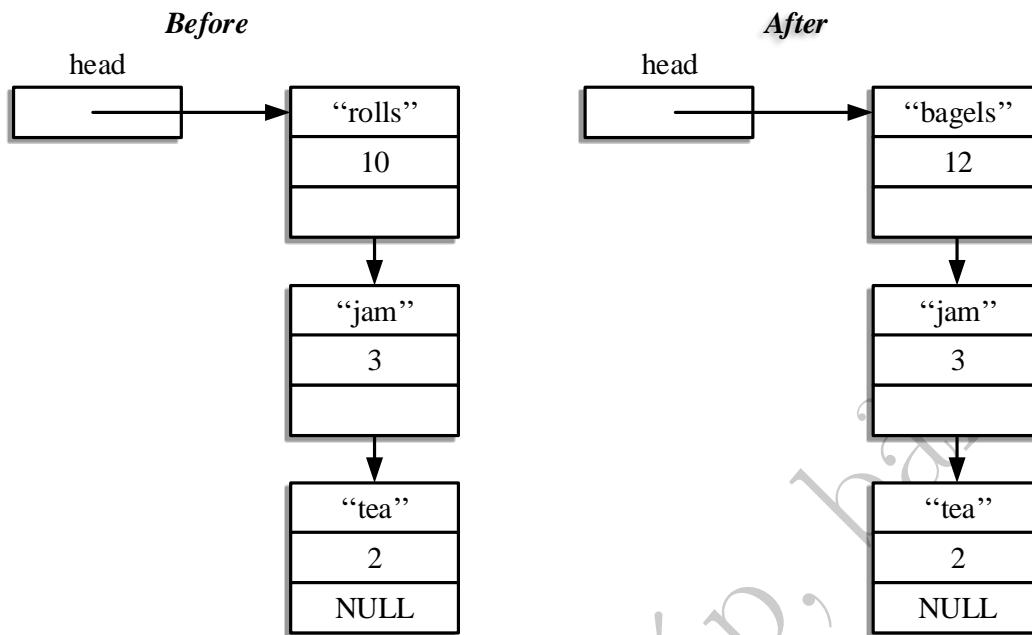
Chuỗi trong nút đầu tiên có thể được thay đổi từ `"rolls"` thành `"bagels"` với câu lệnh sau:

```
head->item = "bagels";
```

Kết quả của sự thay đổi này tới nút đầu tiên trong danh sách được mô tả như trong hình 7.16. Nhìn vào nút cuối cùng trong danh sách, ta thấy nút này có từ `NULL` ở vị trí lê ra là một con trỏ. Trong hình 7.15, 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 7.16 và khi chương trình đi tới nút có chứa `NULL` thì cho biết nó đã đi tới nút cuối cùng của danh sách.



Hình 7.16: Truy cập dữ liệu của một nút.

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:

```
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

Danh sách như hình 7.16 được gọi là danh sách liên kết. 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 trở 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 7.16. 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;
```

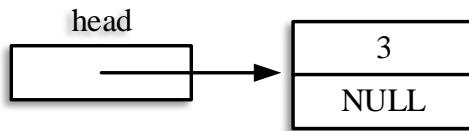
Để tạo nút đầu tiên, chúng ta sử dụng toán tử **new** để tạo một biến động mới, biến động này sẽ trở thành nút đầu tiên trong danh sách liên kết:

```
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:



7.9.2 Các thao tác với danh sách liên kết

Danh sách liên kết một nút của chúng ta đã được xây dựng như trên, để có danh sách liên kết lớn hơn, chương trình phải có khả năng chèn thêm các nút một cách hệ thống, phần tiếp theo sẽ mô tả một cách đơn giản để chèn một nút vào danh sách liên kết.

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 trả về 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 trả về 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:

Giả mã cho hàm `head_insert`:

1. Tạo một biến động được trả về 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 trả về nút đầu của danh sách liên kết ban đầu
4. Biến con trỏ tên `head` trả về nút mới

Hình 7.17 chia lưu đồ của thuật toán. Bước 2 và 3 trong lưu đồ có thể được thể hiện bằng câu lệnh trong C++ như sau:

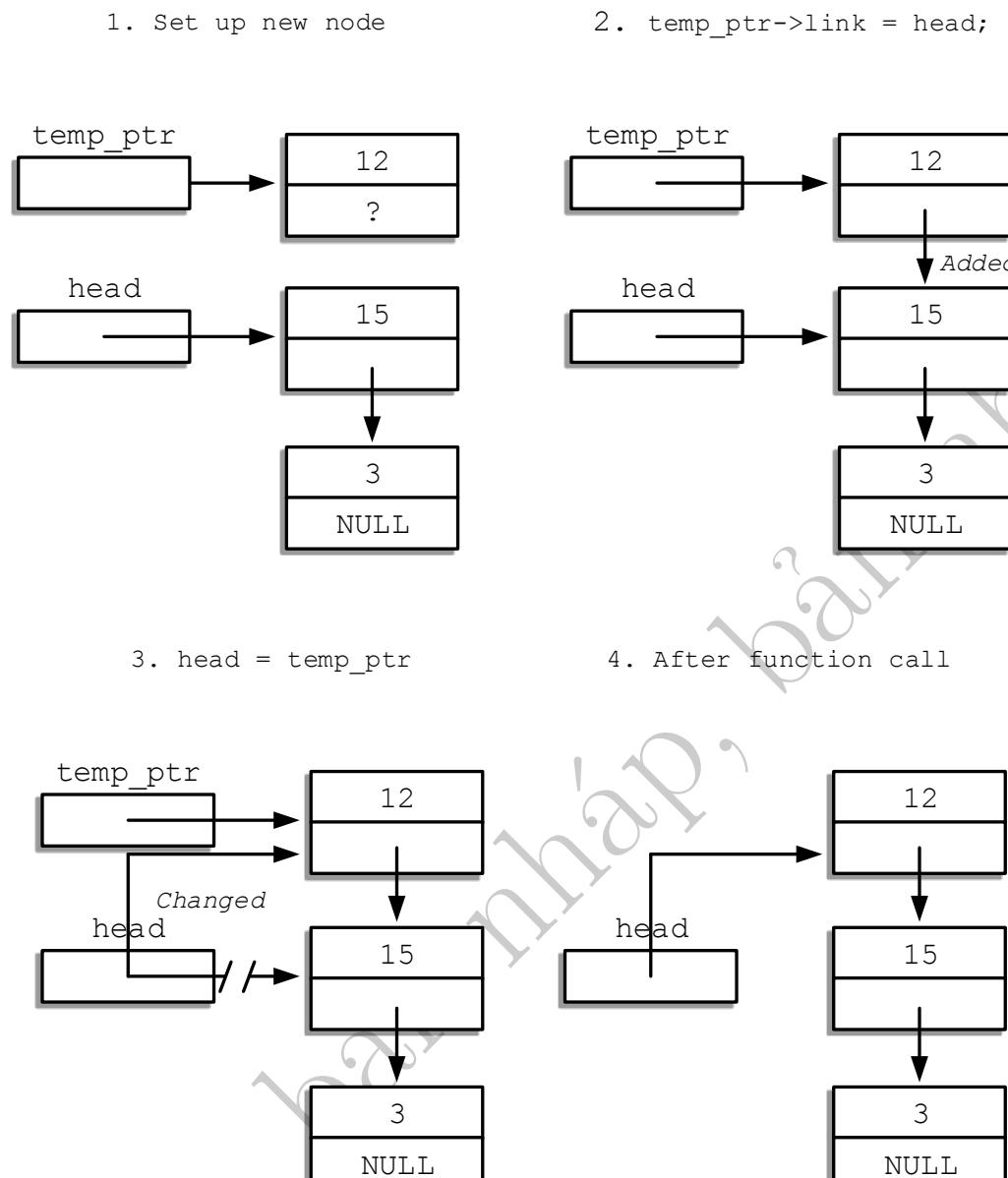
```
temp_ptr->link = head;
head = temp_ptr;
```

Hàm đầy đủ được định nghĩa trong hình 7.18.

```

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;

```



Hình 7.17: Thêm một nút vào danh sách liên kết.

23 }

Hình 7.18: Hàm chèn một nút mới vào đầu danh sách liên kết

Nếu muốn danh sách có cho phép khả năng không chứa một cái gì. Ví dụ, một danh sách mua sắm có thể không có gì trong đó bởi vì không có gì để mua trong tuần. Một danh sách không có gì trong đó gọi là danh sách rỗng. Danh sách liên kết được đặt tên bởi tên của con trỏ trả về 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 trả về nút đầu của danh sách liên kết và giờ muốn chỉ ra danh sách là rỗng, thì ta thiết lập giá trị của con trỏ `head` như sau:

```
head = NULL;
```

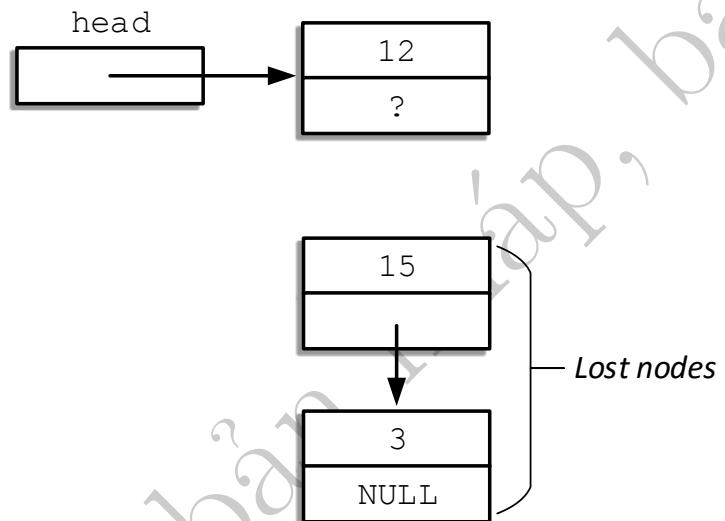
Khi xây dựng hàm thao tác với một danh sách liên kết, ta luôn phải kiểm tra để biết hàm có làm việc trên danh sách rỗng không, nếu không, ta có thể phải thêm trường hợp đặc biệt cho danh sách

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 bắt cứ một danh sách nào khác. Ví dụ, hàm `head_insert` trong hình 7.18 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.

Hiện tượng: Mất nút

Ta có thể viết định nghĩa hàm `head_insert` (hình 7.18) 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;
```



Hình 7.19: Nút bị mất trong danh sách liên kết.

Tại thời điểm nút mới được xây dựng, chứa dữ liệu đúng và nó được trả 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 trả sau trả tới nút trước đó là nút đầu tiên của danh sách như sau:

```
head->link
```

Hình 7.19 cho thấy trường hợp khi giá trị dữ liệu mới là **12**, minh họa đó cho thấy các vấn đề, nếu ta đã xử lý theo cách này, sẽ không có gì trả tới nút chứa giá trị **15**, vì không có con trỏ trả tới nó nên không có cách nào để chương trình có thể tham chiếu tới đó, các nút bên dưới nút này cũng bị mất. Một chương trình sẽ không có cách nào để một con trỏ trả tới những nút này cũng như nó không thể truy xuất dữ liệu và làm bắt cứ việc gì đối với những nút này bởi vì không có cách nào để có thể tham chiếu tới những nút đó.

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 trả tới đầu danh sách, thường là con trỏ trong biến con trỏ như `head`.

Tìm kiếm một danh sách liên kết

Tiếp theo chúng ta sẽ xây dựng một hàm tìm kiếm 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 7.18). 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**. 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, kĩ thuật này được biểu diễn như trong hình 7.20 và thuật toán được mô tả như sau: **Giả mã cho hàm tìm kiếm**

```
Thực hiện cho biến con trỏ here trỏ tới nút head (đó cũng là nút đầu tiên) của danh sách liên
kết.
while (here không trỏ tới nút chứa target và here không trỏ tới nút cuối của danh sách)
{
    Thực hiện cho con trỏ here trỏ tới nút kế tiếp trong danh sách.
}
if (nút được trỏ 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 trả bởi thành viên con trỏ của nút hiện thời được trả bởi **here**. Thành viên con trỏ của nút hiện thời được trả 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:

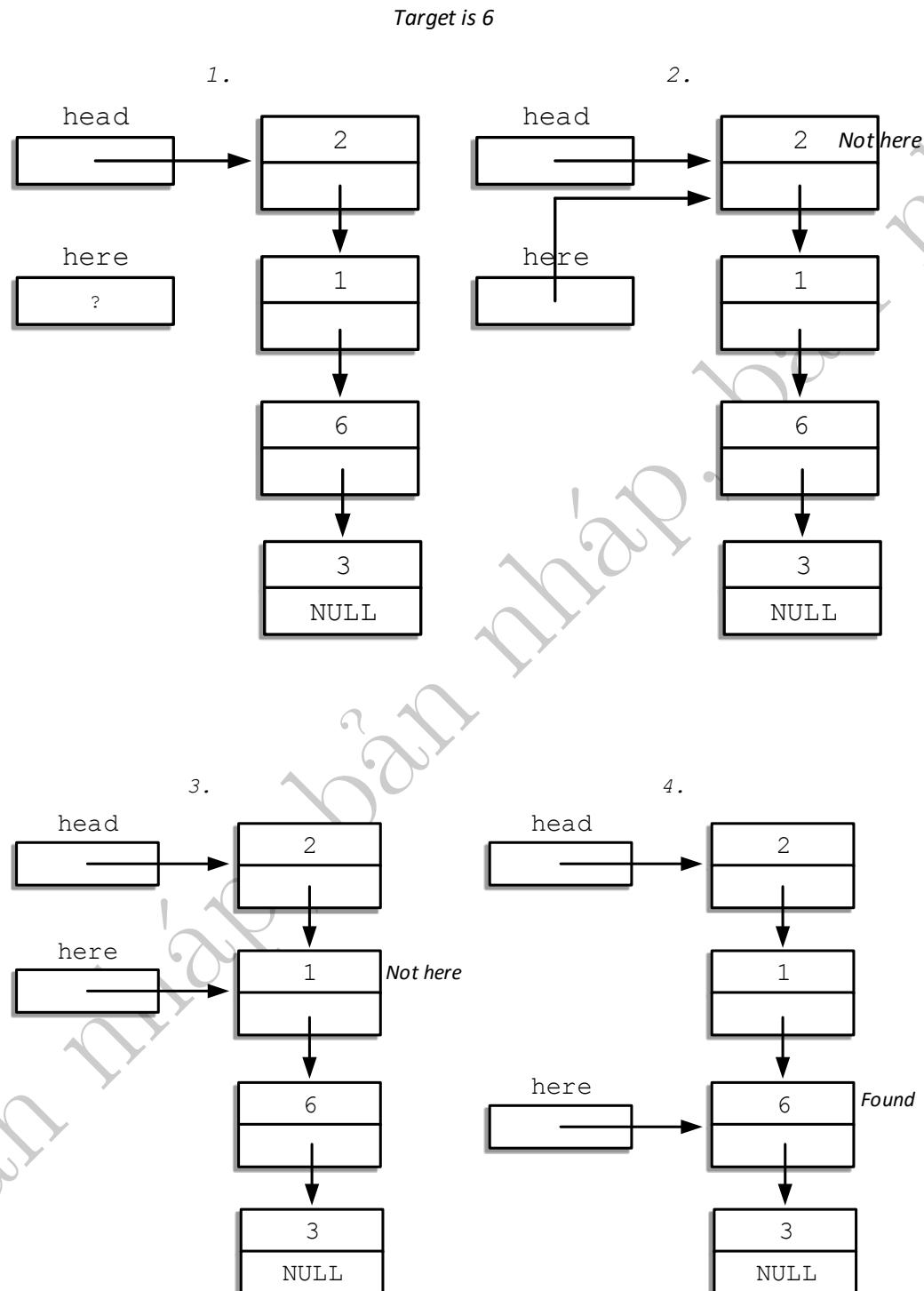
```
here = head;
while (here->data != target && here->link != NULL)
    here = here->link;
if (here->data == target)
    return here;
```

```

else
    return NULL;

```

Chú ý biểu thức logic trong câu lệnh **while**. Chúng ta kiểm tra để xem nếu **here** không trỏ tới nút cuối cùng bằng việc kiểm tra xem biến thành viên **here->link** khác giá trị **NULL**.



Hình 7.20: Tìm kiếm trong danh sách liên kết.

Nếu kiểm tra mã nguồn của trên, chúng ta sẽ thấy có vấn đề với danh sách rỗng. 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 trỏ 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 7.21.

```
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 7.21: Xác định vị trí nút trong danh sách liên kết

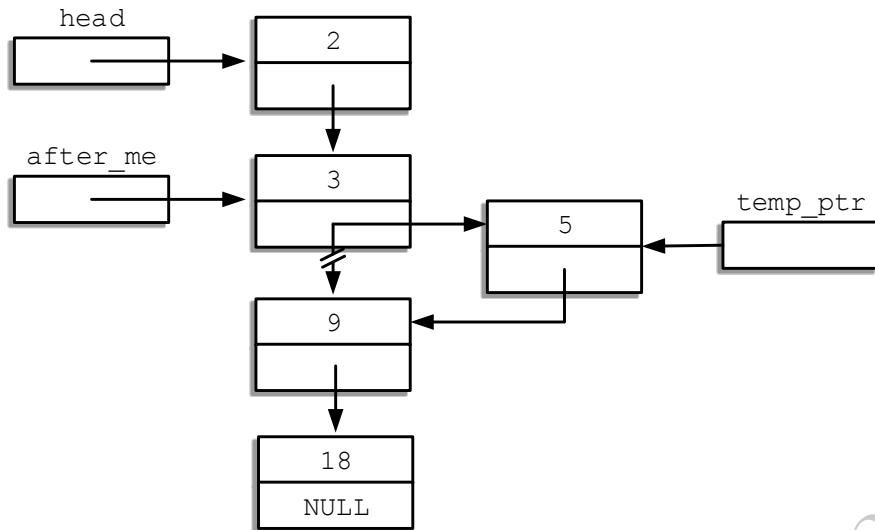
Chèn và gỡ bỏ một nút bên trong danh sách

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 7.22. 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 trỏ bởi after_me.
void insert(NodePtr after_me, int the_number);
```

Một nút mới thiết lập theo cách giống như vậy, đó là hàm `head_insert` trong hình 7.18. 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 trỏ bởi con trỏ `after_me`. Cách thức chèn được mô tả như hình 7.22 và được thể hiện trong C++ như sau:

```
// Thêm liên kết từ nút mới tới danh sách:
temp_ptr->link = after_me->link;
```



Hình 7.22: Chèn phần tử vào giữa danh sách liên kết.

```
//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, trong phép gán đầu tiên chúng ta muốn lấy giá trị con trỏ `after_me->link` trước khi nó bị thay đổi. Hàm hoàn thiện được đưa ra như 7.23.

Với hàm `insert`, ta sẽ thấy nó làm việc chính xác thậm chí nếu nút được trả bởi `after_me` là nút cuối cùng trong danh sách. Tuy nhiên, hàm `insert` sẽ không làm việc cho việc chèn một nút vào vị trí bắt đầu của danh sách, để thực hiện việc này ta có thể sử dụng hàm `head_insert` như 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 }
```

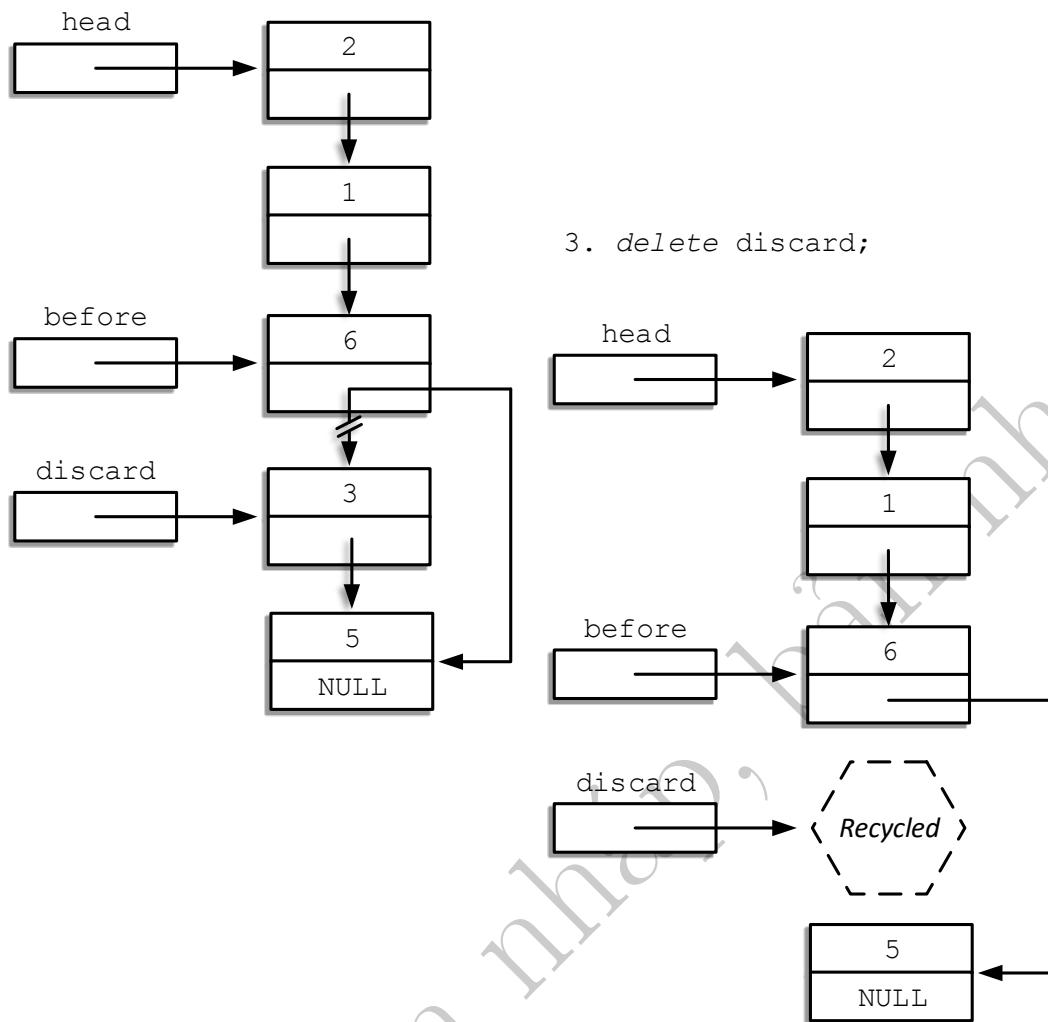
Hình 7.23: Hàm thêm một nút vào giữa danh sách liên kết

Để loại bỏ một nút từ danh sách liên kết cũng khá đơn giản. Hình 7.24 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 7.24: Xóa một nút trong danh sách liên kết.

7.9.3 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 7.25 và 7.26 minh họa làm thế nào để định nghĩa một lớp `Node`. 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 7.27 tạo một danh sách ngắn gồm 5 nút bằng cách chèn các nút mới.

```

1 //Đây là file header cho Node.h.
2 namespace linkedlistofclasses
3 {
4     class Node
5     {
6         public:
7             Node( );
8             //Khai tao mot nut
9             Node(int value, Node *next);

```

```

10     //Lay gia tri cua nut nay
11     int getData( ) const;
12
13     //Ham lay gia tri cua nut ke tiep
14     Node *getLink( ) const;
15
16     //Ham thay doi gia tri luu trong danh sach
17     void setData(int value);
18
19     //Ham thay doi tham chieu toi nut ke tiep
20     void setLink(Node *next);
21     private:
22         int data;
23         Node *link;
24     };
25     typedef Node* NodePtr;
26 } //Danh sach lien ket cua lop
27
28 //Ket thuc Node.h

```

Hình 7.25: File giao diện của lớp danh sách liên kết.

```

1 //Đây là file thực hiện 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
12    Node::Node(int value, Node *next) : data(value), link(next)
13    {
14        //Rong
15    }
16
17    int Node::getData() const
18    {
19        return data;
20    }
21
22    Node* Node::getLink() const
23    {
24        return link;
25    }
26
27    void Node::setData(int value)
28    {
29        data = value;
30    }
31
32    void Node::setLink(Node *next)

```

```

33     {
34         link = next;
35     }
36 } //linkedlistofclasses
37 //Ket thuc file Node_chuong_7.cpp

```

Hình 7.26: File thực hiện của lớp danh sách liên kết.

```

1 //Chuong trinh nay trinh bay cach tao danh sach lien ket su dung Node class
2 #include <iostream>
3 #include "Node_chuong_7.h"
4 using namespace std;
5 using namespace linkedlistofclasses;
6 void head_insert(NodePtr& head, int the_number)
7 {
8     NodePtr temp_ptr;
9     //Thiet lap temp_ptr->link toi head v
10    //thiet lap gia tri du lieu toi the_number
11    temp_ptr = new Node(the_number, head);
12    head = temp_ptr;
13 }
14
15 int main()
16 {
17     NodePtr head, tmp;
18
19     //Tao danh sach coint 4 -> 3 -> 2 -> 1 -> 0
20     head = new Node(0, NULL);
21     for (int i = 0; i < 5; i++)
22     {
23         head_insert(head, i);
24     }
25     //Duyet qua danh sach v hien thi tat ca cc gia tri
26     tmp = head;
27     while (tmp != NULL)
28     {
29         cout << tmp->getData() << endl;
30         tmp = tmp->getLink();
31     }
32
33     //Xoa tat ca cc nt trong danh sach truoc khi thoat chuong trinh
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 7.27: Chương trình sử dụng lớp danh sách liên kết.

Bài tập

- Trình bày sự khác biệt, ưu điểm, nhược điểm giữa biến tĩnh và biến động.
- Viết một chương trình với hai biến con trỏ x, y. Sử dụng hai biến x, y để lưu hai số thực được nhập từ bàn phím. Tính tổng của x và y và hiện kết quả ra màn hình.
- Tính giá trị của apples, *ptrApp, *ptrApp2 của đoạn mã sau:

```
int apples;
int *ptrApp = &apples;
int *ptrApp2 = ptrApp;
apples += 2;
*ptrApp--;
*ptrApp2 += 3;
```

- Xác định kết quả của đoạn mã sau:

```
int *p1 = new int;
int *p2;
*p1 = 5;
p2 = p1;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
*p2 = 10;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
p1 = new int;
*p1 = 20;
cout << "*p1 = " << *p1 << endl;
cout << "*p2 = " << *p2 << endl;
```

- 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;
```

- 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;
```

- 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;
}
```

8. 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.
9. Phân tích sự khác biệt, như được điểm, ưu điểm của việc sử dụng mảng động và mảng tĩnh. Những lưu ý khi khai báo, xin cấp phát và giải phóng bộ nhớ đối với mảng động.
10. 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.
11. Một bàn cờ có kích thước $m \times n$ ô vuông. Trạng thái trên mỗi ô vuông được biểu diễn bởi một kí tự in hoa từ 'A' đến 'Z'. Sử dụng mảng động hai chiều để lưu giữ trạng thái của bàn cờ nhập từ bàn phím. Tìm và hiện ra màn hình:
 - Các hàng thỏa mãn điều kiện tất cả các ô cùng một trạng thái;
 - Các cột thỏa mãn điều kiện tất cả các ô cùng một trạng thái;
 - Các đường chéo thỏa mãn điều kiện tất cả các ô cùng một trạng thái.

bản nháp, bản nháp,
bản nháp, bản nháp,

Chương 8

Vào ra dữ liệu

Tất cả các chương trình mà ta đã gặp trong giáo trình này đều lấy dữ liệu vào từ bàn phím và in ra màn hình. Nếu chỉ dùng bàn phím và màn hình là các thiết bị vào ra dữ liệu thì chương trình của ta khó có thể xử lý được khối lượng lớn dữ liệu, và kết quả chạy chương trình sẽ bị mất ngay khi chúng ta đóng cửa sổ màn hình in ra hoặc tắt máy. Để cải thiện tình trạng này, chúng ta có thể lưu dữ liệu tại các thiết bị lưu trữ thứ cấp mà thông dụng nhất thường là ổ đĩa cứng. Khi đó dữ liệu tạo bởi một chương trình có thể được lưu lại để sau này được sử dụng bởi chính nó hoặc các chương trình khác. Dữ liệu lưu trữ như vậy được đóng gói tại các thiết bị lưu trữ thành các cấu trúc dữ liệu gọi là **tệp** (*file*).

Chương này sẽ giới thiệu về cách viết các chương trình lấy dữ liệu vào (từ bàn phím hoặc từ một tệp) và ghi dữ liệu ra (ra màn hình hoặc một tệp).

8.1 Khái niệm dòng dữ liệu

Trong một số ngôn ngữ lập trình như C++ và Java, dữ liệu vào ra từ tệp, cũng như từ bàn phím và màn hình, đều được vận hành thông qua các **dòng dữ liệu** (*stream*). Ta có thể coi dòng dữ liệu là một kênh hoặc mạch dẫn mà dữ liệu được truyền qua đó để chuyển từ nơi gửi đến nơi nhận.

Dữ liệu được truyền từ chương trình ra ngoài theo một **dòng ra** (*output stream*). Đó có thể là dòng ra chuẩn nối và đưa dữ liệu ra màn hình, hoặc dòng ra nối với một tệp và đẩy dữ liệu ra tệp đó.

Chương trình nhận dữ liệu vào qua một **dòng vào** (*input stream*). Dòng vào có thể là dòng vào chuẩn nối và đưa dữ liệu vào từ màn hình, hoặc dòng vào nối với một tệp và nhận dữ liệu vào từ tệp đó.

Dữ liệu vào và ra có thể là các kí tự, số, hoặc các byte chứa các chữ số nhị phân.

Trong C++, các dòng vào ra được cài đặt bằng các đối tượng của các lớp dòng vào ra đặc biệt. Ví dụ, `cout` mà ta vẫn dùng để ghi ra màn hình chính là dòng ra chuẩn, còn `cin` là dòng vào chuẩn nối với bàn phím. Cả hai đều là các đối tượng dòng dữ liệu (khái niệm “đối tượng” này có liên quan đến tính năng hướng đối tượng của C++, khi nói về các dòng vào/ra của C++, ta sẽ phải đề cập nhiều đến tính năng này).

8.1.1 Tệp văn bản và tệp nhị phân

Về bản chất, tất cả dữ liệu trong các tệp đều được lưu trữ dưới dạng một chuỗi các bit nhị phân 0 và 1. Tuy nhiên, trong một số hoàn cảnh, ta không coi nội dung của một tệp là một chuỗi 0 và 1 mà coi tệp đó là một chuỗi các kí tự. Một số tệp được xem như là các chuỗi kí tự và được xử lý bằng các dòng và hàm cho phép chương trình và hệ soạn thảo văn bản của bạn nhìn các chuỗi nhị phân như là các chuỗi kí tự. Chúng được gọi là các **tệp văn bản** (*text file*). Những tệp không phải tệp văn bản là **tệp nhị phân** (*binary file*). Mỗi loại tệp được xử lý bởi các dòng và hàm riêng.

Chương trình C++ của bạn được lưu trữ trong tệp văn bản. Các tệp ảnh và nhạc là các tệp nhị phân. Do tệp văn bản là chuỗi kí tự, chúng thường trông giống nhau tại các máy khác nhau, nên ta có thể chép chúng từ máy này sang máy khác mà không gặp hoặc gặp phải rất ít rắc rối. Nội dung của các tệp nhị phân thường lấy cơ sở là các giá trị số, nên việc sao chép chúng giữa các máy có thể gặp rắc rối do các máy khác nhau có thể dùng các quy cách lưu trữ số không giống nhau. Cấu trúc của một số dạng tệp nhị phân đã được chuẩn hóa để chúng có thể được sử dụng thống nhất tại các hệ thống khác nhau. Nhiều dạng tệp ảnh và âm thanh thuộc diện này.

Mỗi kí tự trong một tệp văn bản được biểu diễn bằng 1 hoặc 2 byte, tùy theo đó là kí tự ASCII hay Unicode. Khi một chương trình viết một giá trị vào một tệp văn bản, các kí tự được ghi ra tệp giống hệt như khi chúng được ghi ra màn hình bằng cách sử dụng `cout`. Ví dụ, hành động viết số 1 vào một tệp sẽ dẫn đến kết quả là 1 kí tự được ghi vào tệp, còn với số 1039582 là 7 kí tự được ghi vào tệp.

Các tệp nhị phân lưu tất cả các giá trị thuộc một kiểu dữ liệu cơ bản theo cùng một cách, giống như cách dữ liệu được lưu trong bộ nhớ máy tính. Ví dụ, mỗi giá trị `int` bất kì, 1 hay 1039582 đều chiếm một chuỗi 4 byte.

8.2 Vào ra tệp

C++ cung cấp các lớp sau để thực hiện nhập và xuất dữ liệu đối với tệp:

- `ofstream`: lớp dành cho các dòng ghi dữ liệu ra tệp;
- `ifstream`: lớp dành cho các dòng đọc dữ liệu từ tệp;
- `fstream`: lớp dành cho các dòng vừa đọc vừa ghi dữ liệu ra tệp.

Đối tượng thuộc các lớp này do quan hệ thừa kế nên cách sử dụng chúng khá giống với `cin` và `cout` – các đối tượng thuộc lớp `istream` và `ostream` – mà chúng ta đã dùng. Khác biệt chỉ là ở chỗ ta phải nối các dòng đó với các tệp.

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     ofstream myfile; //mo ta doi tuong dong vao file output
8     myfile.open("hello.txt"); // mo file ra de ghi
9     myfile << "Hello!" << endl; // ghi chuoi "Hello" vao file va xuong dong
10    myfile.close(); // Dong file sau khi thao tac xong

```

```

11     return 0;
12
13 }

```

Hình 8.1: Các thao tác cơ bản với tệp văn bản.

Chương trình trong Hình 8.1 tạo một tệp có tên `hello.txt` và ghi vào đó một câu "Hello!" theo cách mà ta thường làm đối với `cout`, chỉ khác ở chỗ thay `cout` bằng đối tượng dòng `myfile` đã được nối với một tệp. Sau đây là các bước thao tác với tệp.

8.2.1 Mở tệp

Việc đầu tiên là nối đối tượng dòng với một tệp, hay nói cách khác là mở một tệp. Kết quả là đối tượng dòng sẽ đại diện cho tệp, bất kì hoạt động đọc và ghi đối với đối tượng đó sẽ được thực hiện đối với tệp mà nó đại diện. Để mở một tệp từ một đối tượng dòng, ta dùng hàm `open` của nó:

```
open(fileName, mode);
```

Trong đó, `fileName` là một xâu kí tự thuộc loại `const char *` với kết thúc là kí tự `null` (hằng xâu kí tự cũng thuộc dạng này), là tên của tệp cần mở, và `mode` là tham số không bắt buộc và là một tổ hợp của các cờ sau:

| | |
|--------------------------|---|
| <code>ios::in</code> | mở để đọc |
| <code>ios::out</code> | mở để ghi |
| <code>ios::binary</code> | mở ở dạng tệp nhị phân |
| <code>ios::ate</code> | đặt ví trí bắt đầu đọc/ghi tại cuối tệp. Nếu cờ này không được đặt giá trị gì, vị trí khởi đầu sẽ là đầu tệp. |
| <code>ios::app</code> | mở để ghi tiếp vào cuối tệp. Cờ này chỉ được dùng cho dòng mở tệp chỉ để ghi. |
| <code>ios::trunc</code> | nếu tệp được mở để ghi đã có từ trước, nội dung cũ sẽ bị xóa để ghi nội dung mới. |

Các cờ trên có thể được kết hợp với nhau bằng toán tử bit **OR** (`|`). Ví dụ, nếu ta muốn mở tệp `people.dat` theo dạng nhị phân để ghi bổ sung dữ liệu vào cuối tệp, ta dùng lời gọi hàm sau:

```
ofstream myfile;
myfile.open("people.dat", ios::out | ios::app | ios::binary);
```

Trong trường hợp lời gọi hàm `open` không cung cấp tham số `mode`, chẳng hạn Hình 8.1, chế độ mặc định cho dòng loại `ostream` là `ios::out`, cho dòng loại `istream` là `ios::in`, và cho dòng loại `fstream` là `ios::in | ios::out`.

Cách thứ hai để nối một dòng với một tệp là khai báo tên tệp và kiểu mở tệp ngay khi khai báo dòng, hàm `open` sẽ được gọi với các đối số tương ứng. Ví dụ:

```
ofstream myfile ("hello.txt", ios::out | ios::app | ios::binary);
```

Để kiểm tra xem một tệp có được mở thành công hay không, ta dùng hàm thành viên `is_open()`, hàm này không yêu cầu đối số và trả về một giá trị kiểu `bool` bằng `true` nếu thành công và bằng `false` nếu xảy ra trường hợp ngược lại.

```
if (myfile.is_open()) { /* file now open and ready */ }
```

8.2.2 Đóng tệp

Khi ta hoàn thành các công việc đọc dữ liệu và ghi kết quả, ta cần đóng tệp để tài nguyên của nó trở về trạng thái sẵn sàng được sử dụng. Hàm thành viên này không có tham số, công việc của nó là xả các vùng bộ nhớ có liên quan và đóng tệp:

```
myfile.close();
```

Sau khi tệp được đóng, ta lại có thể dùng dòng `myfile` để mở tệp khác, còn tệp vừa đóng lại có thể được mở bởi các tiến trình khác.

Hàm `close` cũng được gọi tự động khi một đối tượng dòng bị hủy trong khi nó đang nối với một tệp.

8.3 Vào ra tệp văn bản và nhị phân

8.3.1 Vào ra tệp văn bản

Chế độ dòng tệp văn bản được thiết lập nếu ta không dùng cờ `ios::binary` khi mở tệp. Các thao tác xuất và nhập dữ liệu đối với tệp văn bản được thực hiện tương tự như cách ta làm với `cout` và `cin`.

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     ofstream coursefile("courses.txt"); //mo ta doi tuong dong vao file output
8     if (coursefile.is_open()){
9         coursefile << "1. Introduction to Programming" << endl;
10        coursefile << "2. Mathematics for Computer Science" << endl;
11    }
12    else cout << "Error: Cannot open file";
13
14    return 0;
15
16 }
```

Hình 8.2: Ghi dữ liệu ra tệp văn bản.

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 //#include <cstdlib>
5
6 using namespace std;
7
8 int main()
9 {
10    ifstream infile("courses.txt"); //mo ta doi tuong file de doc
11    if (infile.is_open()){
12        while (!infile.eof())
```

```

13     {
14         string line;
15         getline(infile,line);
16         cout << line << endl;
17     }
18 }
19 else cout << "Error! Cannot open file!";
20 infile.close();
21 //system("PAUSE");
22 cin.get(); //Dung de xem man hinh output
23 return 0;
24
25 }

```

Hình 8.3: Đọc dữ liệu từ tệp văn bản.

Chương trình ví dụ trong Hình 8.2 ghi hai dòng văn bản vào một tệp. Chương trình trong Hình 8.3 đọc nội dung tệp đó và ghi ra màn hình. Để ý rằng trong chương trình thứ hai, ta dùng một vòng lặp để đọc cho đến cuối tệp. Trong đó, `infile.eof()` là hàm trả về giá trị `true` khi chạm đến cuối tệp, giá trị `true` mà `infile.eof()` trả về đã được dùng làm điều kiện kết thúc vòng lặp đọc tệp.

Kiểm tra trạng thái của dòng

Bên cạnh hàm `eof()` có nhiệm vụ kiểm tra cuối tệp, còn có các hàm thành viên khác dùng để kiểm tra trạng thái của dòng:

- `bad()` trả về `true` nếu một thao tác đọc hoặc ghi bị thất bại. Ví dụ khi ta cố viết vào một tệp không được mở để ghi hoặc khi thiết bị lưu trữ không còn chỗ trống để ghi.
- `fail()` trả về `true` trong những trường hợp `bad()` trả về `true` và khi có lỗi định dạng, chẳng hạn như khi ta đang định đọc một số nguyên nhưng lại gặp phải dữ liệu là các chữ cái.
- `eof()` trả về `true` nếu chạm đến cuối tệp
- `good()` trả về `false` nếu xảy tình huống mà một trong các hàm trên nếu được gọi thì sẽ trả về `true`.

Để đặt lại cờ trạng thái mà một hàm thành viên nào đó đã đánh dấu trước đó, ta dùng hàm thành viên `clear`.

Con trỏ get và put của dòng

Mỗi đối tượng dòng vào ra có ít nhất một con trỏ nội bộ. Con trỏ nội bộ của `ifstream` hay `istream` được gọi là **con trỏ get** hay **con trỏ đọc**. Nó chỉ tới vị trí mà thao tác đọc tiếp theo sẽ được thực hiện tại đó. Con trỏ nội bộ của `ofstream` hay `ostream` được gọi là **con trỏ put** hay **con trỏ ghi**. Nó chỉ tới vị trí mà thao tác ghi tiếp theo sẽ được thực hiện tại đó. Cuối cùng, `fstream` có cả con trỏ `get` và `con trỏ put`.

Để định vị vị trí hiện tại của các con trỏ `get` và `put`, ta có các hàm thành viên `tellg` và `tellp`. Các hàm này trả về một giá trị thuộc kiểu `pos_type`, là kiểu dữ liệu số nguyên biểu diễn vị trí hiện tại (tính từ đầu tệp) của con trỏ `get` của dòng (nếu gọi hàm `tellg`) hoặc con trỏ `put` của dòng (nếu gọi hàm `tellp`).

Để đặt lại vị trí của các con trỏ get và put, ta có các hàm `seekg(offset, direction)` và `seekp(offset, direction)` có công dụng di chuyển các con trỏ get và put tới vị trí `offset`. Trong đó, `offset` được tính từ đầu tệp nếu `direction` là `ios::beg` (giá trị mặc định của `direction`), từ cuối tệp nếu `direction` là `ios::end`, và từ vị trí hiện tại nếu `direction` là `ios::cur`.

Hình 8.4 minh họa cách sử dụng các con trỏ get và put để tính kích thước của một tệp văn bản.

```

1 #include <iostream>
2 #include <fstream>
3 // #include <cstdlib>
4
5 using namespace std;
6
7 int main()
8 {
9     char filename[100] = "courses.txt";
10    ifstream infile(filename); // mo ta doi tuong file de doc
11    long begin = infile.tellg();
12    infile.seekg(0, ios::end);
13    long end = infile.tellg();
14
15    cout << "The size of " << filename << " is " << (end - begin) << " bytes." << endl
16    ;
17
18 // system("PAUSE");
19 cin.get(); // Dung de xem man hinh output
20
21 return 0;
22 }
```

Hình 8.4: Dùng con trỏ dòng để xác định kích thước tệp.

8.3.2 Vào ra tệp nhị phân

Để đọc và ghi dữ liệu với tệp nhị phân, ta không thể dùng các toán tử `<<`, `>>` và các hàm như `getline` do dữ liệu có định dạng khác và các kí tự trắng không được dùng để tách giữa các phần tử dữ liệu. Thay vào đó, ta dùng hai hàm thành viên được thiết kế riêng cho việc đọc và ghi dữ liệu nhị phân một cách tuần tự là `write` và `read`. Cách dùng như sau:

```

output_stream.write(memory_block, size);
input_stream.read(memory_block, size);
```

Trong đó `memory_block` thuộc loại "con trỏ tới `char`" (`char*`), nó đại diện cho địa chỉ của một mảng `byte` lưu trữ các phần tử dữ liệu đọc được hoặc các phần tử cần được ghi ra dòng. Tham số `size` là một giá trị nguyên xác định số kí tự cần đọc hoặc ghi vào mảng đó.

Các chương trình trong Hình 8.5, Hình 8.6 và Hình 8.7 minh họa việc đọc và ghi dữ liệu kiểu người dùng tự định nghĩa từ tệp nhị phân. Chương trình trong hình 8.6 minh họa việc ghi dữ liệu kiểu cấu trúc `Student` (bao gồm 2 trường: tên là `name` và điểm là `score`) vào file nhị phân `scores.dat`. Chương trình trong hình 8.7 minh họa việc đọc dữ liệu kiểu cấu trúc `Student` từ file `scores.dat` (được tạo từ chương trình trong hình 8.6) và in ra màn hình. `<student.h>`

```

1 #ifndef STUDENT_H
2 #define STUDENT_H
3
4 #include <iostream>
5
6 using namespace std;
7 #define MAX_NAME_LENGTH 20
8
9 struct Student{
10     char name[MAX_NAME_LENGTH + 1];
11     float score;
12     Student()
13     {
14         name[0] = '\0';
15         score = 0;
16     }
17     Student(const char *, float);
18     void PrintInformation()
19     {
20         cout << name << "\t" << score << endl;
21     }
22 };
23
24 Student::Student(const char *n, float s)
25 {
26     int length = strlen(n);
27     if (length > MAX_NAME_LENGTH) length = MAX_NAME_LENGTH;
28     strncpy(name,n, length);
29     name[length] = '\0'; //Kết thúc một xâu bởi ký tự 0
30     score = s;
31
32 }
33 #endif

```

Hình 8.5: Kiểu bản ghi đơn giản Student.

```

1 #include <iostream>
2 #include <fstream>
3 #include <cstring>
4 //#include <cstdlib>
5 #include "student.h"
6
7 using namespace std;
8
9 int main()
10 {
11
12     ofstream outfile("scores.dat",ios::binary | ios::out);
13     if (outfile.is_open()){
14         Student anne("anne", 8.5);
15         Student julia("julia", 9.5);
16         Student bob("bob", 4.5);
17
18         outfile.write((char*)(&anne), sizeof(Student));

```

```

19     outfile.write((char*)(&julia), sizeof(Student));
20     outfile.write((char*)(&bob), sizeof(Student));
21
22     outfile.close();
23     cout << "Written data in file sucessfully!";
24 }
25 else cout << "Error: Cannot open file!";
26 //system("PAUSE");
27 cin.get(); //Dung de xem man hinh output
28 return 0;
29
30 }
```

Hình 8.6: Ghi dữ liệu ra tệp nhị phân scores.dat.

```

1 #include <iostream>
2 #include <fstream>
3 #include <cstring>
4 //#include <cstdlib>
5 #include "student.h"
6
7 using namespace std;
8
9 int main()
10 {
11
12     ifstream infile("scores.dat",ios::binary);
13     if (infile.is_open()){
14         while (!infile.eof()){
15
16             Student student;
17             infile.read((char*)(&student), sizeof(student));
18             if (infile.good()) student.PrintInformation();
19         }
20         infile.close();
21         cout << "Reading data in file sucessfully!";
22     }
23 }
24 else cout << "Error: Cannot open file!";
25 //system("PAUSE");
26 cin.get(); //Dung de xem man hinh output
27 return 0;
28
29 }
```

Hình 8.7: Đọc dữ liệu từ tệp nhị phân scores.dat.

Bài tập

- Nhập vào từ bàn phím một danh sách sinh viên. Mỗi sinh viên gồm có các thông tin sau đây: tên tuổi, ngày tháng năm sinh, nơi sinh, quê quán, lớp, học lực (từ 0 đến 9). Hãy ghi thông tin về danh sách sinh viên đó ra tệp văn bản student.txt.

2. Sau khi thực hiện bài 1, hãy viết chương trình nhập danh sách sinh viên từ tệp văn bản **student.txt** rồi hiển thị ra màn hình:
 - Thông tin về tất cả các bạn tên là Vinh ra tệp văn bản **vinh.txt**;
 - Thông tin tất cả các bạn quê ở Hà Nội ra tệp văn bản **hanoi.txt**;
 - Tổng số bạn có học lực kém (<4), học lực trung bình (4 và <8), học lực giỏi (8) ra tệp văn bản **hocluc.txt**.
 3. Sau khi thực hiện bài 2, hãy viết chương trình cho biết kích thước của tệp văn bản **student.txt**, **vinh.txt**, **hanoi.txt**, **hocluc.txt**. Kết quả ghi ra tệp văn bản **all.txt**.
 4. Viết chương trình kiểm tra xem tệp văn bản **student.txt** có tồn tại hay không? Nếu tồn tại thì hiện ra màn hình các thông tin sau:
 - Số lượng sinh viên trong tệp
 - Số lượng dòng trong tệp
 - Ghi vào cuối tệp văn bản dòng chữ “CHECKED”
 - Nếu không tồn tại, thì hiện ra màn hình dòng chữ “NOT EXISTED”.
 5. Tệp văn bản **numbers.txt** gồm nhiều dòng, mỗi dòng chứa một danh sách các số nguyên hoặc thực. Hai số đứng liền nhau cách nhau ít nhất một dấu cách. Hãy viết chương trình tổng hợp các thông tin sau và ghi vào tệp văn bản **info.txt** những thông tin sau:
 - Số lượng số trong tệp;
 - Số lượng các số nguyên;
 - Số lượng các số thực.
- Lưu ý: Test chương trình với cả trường hợp tệp văn bản **number.txt** chứa một hay nhiều dòng trắng ở cuối tệp.
6. Trình bày sự khác nhau, ưu điểm, nhược điểm giữa tệp văn bản và tệp văn bản nhị phân. Khi nào thì nên dùng tệp văn bản nhị phân.
 7. Cho file văn bản **numbers.txt** chứa các số nguyên hoặc thực. Hãy viết một chương trình đọc các số từ file **numbers.txt** và ghi ra file nhị phân **numbers.bin** các số nguyên nằm trong file **numbers.txt**.
 8. Sau khi thực hiện bài 7, hãy viết một chương trình đọc và tính tổng của tất cả các số nguyên ở file nhị phân **numbers.bin**. Hiện ra màn hình kết quả thu được.
 9. Sau khi thực hiện bài 7, hãy viết một chương trình đọc các số nguyên ở file nhị phân **numbers.bin**. Ghi các số nằm ở vị trí chẵn (số đầu tiên trong file được tính ở vị trí số 0) trong file nhị phân **numbers.bin** vào cuối file **numbers.txt**.
 10. Cho hai file văn bản **num1.txt** và **num2.txt**, mỗi file chứa 1 dãy số đã được sắp không giảm. Số lượng số trong mỗi file không quá 109. Hãy viết chương trình đọc và ghi ra file văn bản **num12.txt** các số trong hai file **num1.txt** và **num2.txt** thỏa mãn điều kiện các số trong file **num12.txt** cũng được sắp xếp không giảm.

11. File văn bản `document.txt` chứa một văn bản tiếng anh. Các câu trong văn bản được phân cách nhau bởi dấu `'.'` hoặc `'!'`. Hãy ghi ra file văn bản `sentences.txt` nội dung của văn bản `document.txt`, mỗi câu được viết trên một dòng. Ví dụ:

```
document.txt sentences.txt  
this is a good  
house! However, too expensive. this is a good house!  
However, too expensive.
```

12. File văn bản `document.txt` chứa một văn bản có lẫn cả các câu tiếng anh và các câu tiếng Việt. Các câu trong văn bản được phân cách nhau bởi dấu `'.'` hoặc `'!'`. Hãy ghi ra file văn bản `english.txt` (`viet.txt`) các câu tiếng Anh (Việt) trong văn bản `document.txt`.

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 //Chuong trinh minh hoa xu ly truong hop dac biet
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 //Chuong trinh minh hoa xu ly ngoai le
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 kỳ. 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 kỳ. 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 //Chuong trinh minh hoa viec nem nhieu kieu ngoai le
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 “trong sáng” 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 //Chuong trinh minh hoa viec nem ngoai le tu ham
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 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 //Chuong trinh minh hoa ngoai le 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; // nem ngoai le 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 [100000000000];
} catch (bad_alloc) {
    cout << "Cannot allocate the required memory" << endl;
}
```

9.3 Bài tập

- 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 bàng 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 định 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 module 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 gấp 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 đổi

```
#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`;
- và 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 módun 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 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 lây hàm main).

Như vậy các file header và các módun 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.

10.3 Bài tập

- Để định nghĩa hằng ta có 2 cách viết:

```
const int MAX = 100, và
#define MAX 100
```

câu nào sau đây sai:

- Kết quả của chương trình khi chạy là như nhau;
 - Với cách 1 khi chạy chương trình MAX được thay thế bằng 100;
 - Với cách 2 khi dịch chương trình MAX được thay thế bằng 100;
 - Câu lệnh MAX = MAX + 1; không được phép đổi với cách 1 nhưng được phép đổi với cách 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 ?

- Hãy chỉ ra điểm sai của hàm macro sau đây:

```
#define INCREASE(I) I++
```

- Cho macro: **#define** TIMES(A, B) A * B
Với a = 2, b = 5; hãy tính TIMES(a, b + 1) ?
- Điều chỉnh lại macro TIMES trong bài trên để nó tính chính xác tích của 2 số.
- Viết macro tráo đổi nội dung 2 biến (thay cho hàm swap). Áp dụng: Sắp xếp dãy số.

- 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 ?

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 trả 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 trả 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 trả duyệt

Con trả duyệt (*iterator*) là khái niệm tổng quát hóa từ con trả (chương 7). Trong mục này ta sẽ xem xét cách dùng con trả 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 trả duyệt còn được dùng trong rất nhiều lớp mẫu khác của thư viện STL. Hầu hết các thao tác đối với con trả 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 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 // Chuong trinh minh hoa lop chua vector
2 #include <iostream>
3 #include <vector> // de su dung lop chua vector
4
5 int main()
6 {
7     using std::cout;      // dat cac cau lenh using
8     using std::endl;      // o dau moi ham
9     using std::vector;
10
11     vector<int> c;
12
13     for (int i = 0; i < 4; i++)
14         c.push_back(i);   // day i vao cuoi vat chua 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 dãn” 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 << " ";
```

duyệt 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 vé 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 về 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 // Chuong trinh minh hoa con tro duyet ngau nhien
2 #include <iostream>
3 #include <vector> // de su dung lop chua vector
4 #include <string> // lop xau ki tu
5
6 int main()
7 {
8     using std::cout;      // dat cac cau lenh using
9     using std::endl;      // o dau moi ham
10    using std::vector;    // de cac ham doc lap
11    using std::string;    // voi 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;
```

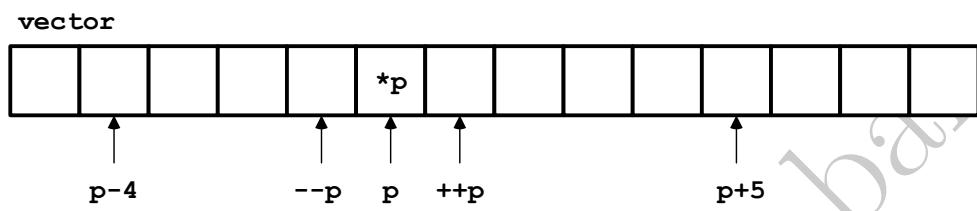
in ra phần tử nằm sau 2 bước nhảy so với phần tử hiện tại p đang trỏ đến.

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 chúa, 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 chúa. 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 chúa `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 chúa

Khái niệm **vật chúa** (hay lớp chúa - *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 chúa, 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 chúa.

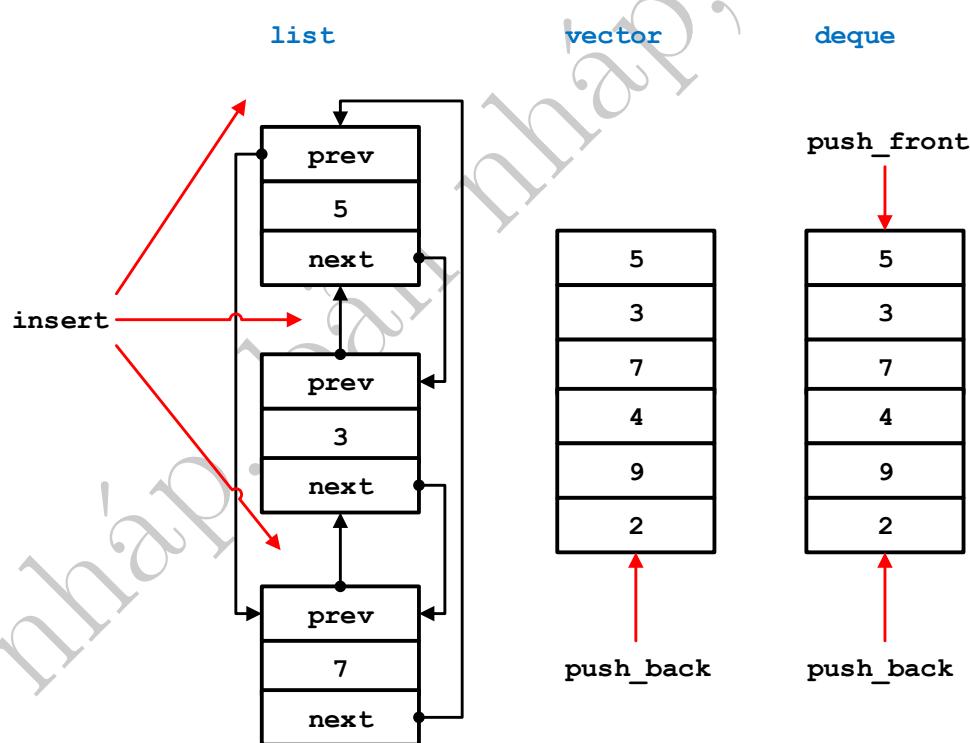
Tất cả các lớp chứa trong STL đều có con trỏ duyệt của nó. Khai báo con trỏ duyệt như sau

```
<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 dây

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 dang dây.

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 kì 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 dây. Còn việc chèn hoặc xóa phần tử ở giữa dây chậm hơn nhiều. Đổi 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, | song hướng | <list> |
| | list<T>::const_reverse_iterator | | |
| vector | vector<T>::iterator, | | |
| | vector<T>::const_iterator, | ngẫu nhiên | <vector> |
| | vector<T>::reverse_iterator, | | |
| | vector<T>::const_reverse_iterator | | |
| deque | deque<T>::iterator, | | |
| | deque<T>::const_iterator, | ngẫu nhiên | <deque> |
| | deque<T>::reverse_iterator, | | |
| | deque<T>::const_reverse_iterator | | |

nhiên. 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 có 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 |
|-------------|---|
| () | <p>Khởi tạo dãy rỗng.</p> <pre>list<int> a; vector<string> b;</pre> |
| (n, val) | <p>Khởi tạo dãy có n phần tử với giá trị val.</p> <pre>deque<int> a(10, 0); // 10 số 0 vector<string> b(100, "Hello"); // 100 xâu Hello</pre> |
| (p1, p2) | <p>Khởi tạo dãy có các phần tử lấy từ khoảng [p1, p2] chỉ định bởi 2 con trỏ duyệt p1 và p2.</p> <pre>deque<int> a(10, 0); // 10 số 0 vector<int> b(a.begin(), a.begin() + 5); // Lấy 5 số đầu mảng a</pre> |

Bảng 11.8: Một số hàm của list, vector và deque.

| Hàm thành viên | Ý nghĩa |
|-------------------|--|
| c.size() | Trả về số phần tử của dãy |
| c.begin() | Trả về con trỏ duyệt tới phần tử đầu tiên |
| c.end() | Trả về con trỏ duyệt để kiểm tra đã hết dãy |
| c.rbegin() | Trả về con trỏ duyệt tới phần tử cuối cùng |
| c.rend() | Trả về con trỏ duyệt để kiểm tra (chiều nghịch) |
| c.push_back(ele) | Thêm phần tử ele vào cuối dãy |
| c.push_front(ele) | Thêm phần tử ele vào đầu dãy |
| c.insert(p, ele) | Thêm phần tử ele vào trước phần tử do p trỏ đến |
| c.erase(p) | Xóa phần tử do p trỏ đến |
| c.clear() | Xóa toàn bộ dãy (dãy trống) |
| c.front() | Phần tử đầu dãy, tương đương với *(c.begin()) |
| c.back() | Phần tử cuối dãy, tương đương với *(c.rbegin()) |
| c1 == c2 | Trả về true nếu c1.size() == c2.size() và mỗi phần tử trong c1 bằng với phần tử tương ứng trong c2 |
| c1 != c2 | Tương đương với !(c1 == c2) |

```

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 void countWords(istream& input, vector<string>& words, vector<int>& counts, bool
29 shouldClear)
30 {
31     if (shouldClear) {
32         words.clear();
33         counts.clear();
34     }
35
36     string word;
37     input >> word;
38     while (input) {
39         bool found = false;
40         // tim tu trong danh sach
41         vector<int>::iterator pCount = counts.begin();
42         for (vector<string>::iterator pWord = words.begin(); pWord != words.end(); pWord++, pCount++) {
43             if (word == *pWord) { // tim thay tu
44                 (*pCount)++;
45                 found = true;
46                 break;
47             }
48         }
49         if (!found) { // khong tim thay, them vao danh sach
50             words.push_back(word);
51             counts.push_back(1);
52         }
53         input >> word;
54     }

```

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 liền nhau phân cách bởi khoảng trống (kí 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)(

```

```

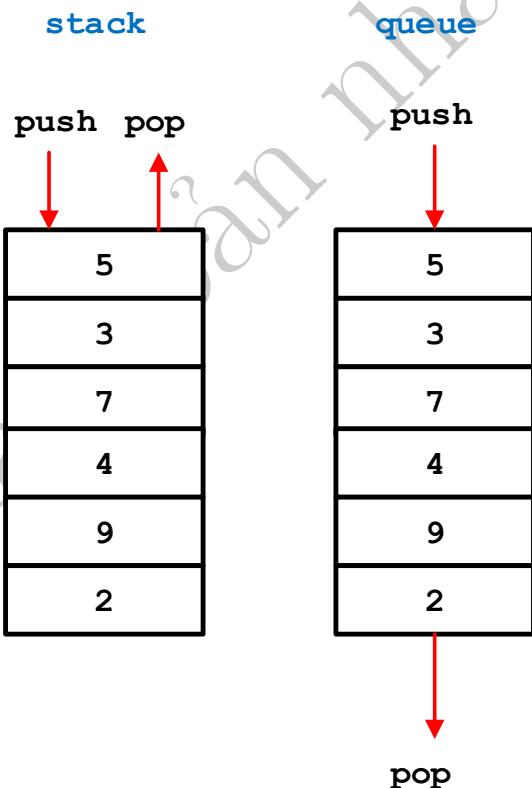
isalpha,1)(space,1)(void,2)(countWords,3)(istream,2)(input,8)(string,6)
(words,10)(counts,8)(bool,3)(shouldClear,3)(true,2)(main,1)(argc,2)(argv,2)(2,1)(cout,2)(Usage,1)(word,6)(count,1)(filename,1)(endl,1)(return,2)(ifstream,1)(1,2)(newSpace,2)(new,1)(imbue,1)(iterator,4)(pCount,6)(begin,4)(pWord,8)(end,2)(clear,2)(while,1)(found,3)(tim,3)(trong,1)(danh,2)(sach,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 xâu 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 xâu, 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; // khong 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.isEmpty(); // 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 e vào tập hợp |
| <code>insert(p, e)</code> | Thêm phần tử e vào tập hợp 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). |
| <code>insert(p1, p2)</code> | Thêm tất cả các phần tử nằm trong khoảng [p1, p2) vào tập hợp (không tính phần tử do p2 trỏ đến). |
| <code>erase(p)</code> | Xóa phần tử do p trỏ đến. |
| <code>erase(e)</code> | Xóa phần tử có giá trị e. |
| <code>erase(p1, p2)</code> | Xóa tất cả các phần tử nằm trong khoảng [p1, p2) 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 e. Nếu tập hợp không chứa e, 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["NEWNUMBER"] << 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 |
|-----------------------------|--|
| <code>insert(e)</code> | Thêm phần tử e vào ánh xạ, trong đó e là một cặp pair<key, value>. |
| <code>insert(p, e)</code> | 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). |
| <code>insert(p1, p2)</code> | Thêm tất cả các phần tử nằm trong khoảng [p1, p2) vào tập hợp (không tính phần tử do p2 trỏ đến). |
| <code>erase(p)</code> | Xóa phần tử do p trỏ đến. |
| <code>erase(k)</code> | Xóa phần tử có giá trị khóa bằng k. |
| <code>erase(p1, p2)</code> | Xóa tất cả các phần tử nằm trong khoảng [p1, p2) trong ánh xạ. |
| <code>clear()</code> | Xóa tất cả các phần tử trong tập hợp. |
| <code>find(k)</code> | 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         // tim tu trong danh sach
55         auto p = wordCounts.find(word);
56         if (p != wordCounts.end())
57             p->second++; // so dem o thanh vien second
58         else
59             wordCounts[word] = 1; // khong tim thay
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 anh xa 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. Chung 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 khát khao 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 gán để tráo 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 ký 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 // Chuong trinh minh hoa 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 $[p1, p2)$ |
| <code>count(p1, p2, e)</code> | Đếm số phần tử có giá trị <code>e</code> trong khoảng $[p1, p2)$ |
| <code>equal(p1, p2, p)</code> | Chỉ trả về <code>true</code> nếu các phần tử trong khoảng $[p1, p2)$ giống hệt $p2-p1$ phần tử tính từ phần tử <code>p</code> đang trả tới. |
| <code>search(p1, p2, t1, t2)</code> | Nếu dãy các phần tử trong $[t1, t2)$ là dãy con của khoảng $[p1, p2)$, trả về con trỏ duyệt đến vị trí đầu tiên bắt đầu dãy con trong $[p1, p2)$. 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 $[p1, p2)$ 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 $[p1, p2)$ 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 $[p1, p2)$. 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 $[p1, p2)$. |

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 // Tinh tong vector bang 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 // Tinh tong vector bang foreach va 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 `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 trả 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 $[p1, p2)$ sang khoảng $[t1, t2)$. Điều kiện là số phần tử bằng nhau (tức là $t2 - t1 == p2 - p1$). |
| <code>remove(p1, p2, e)</code> | Xóa các phần tử bằng giá trị <code>e</code> trong khoảng $[p1, p2)$. |
| <code>reverse(p1, p2)</code> | Đảo ngược thứ tự các phần tử trong khoảng $[p1, p2)$. |
| <code>random_shuffle(p1, p2)</code> | Tráo đổi ngẫu nhiên thứ tự các phần tử trong khoảng $[p1, p2)$. |
| <code>fill(p1, p2, e)</code> | Đặt giá trị các phần tử trong khoảng $[p1, p2)$ bằng <code>e</code> . |
| <code>iota(p1, p2, e)</code> | Đặt giá trị các phần tử trong khoảng $[p1, p2)$ 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 $[p1, p2)$ 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 $[p1, p2)$. |
| <code>sort(p1, p2)</code> | Sắp xếp các phần tử trong khoảng $[p1, p2)$ 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 tiền 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 Bài tập

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 $[p_1, p_2]$ và $[t_1, t_2]$ thành dãy được sắp xếp bắt đầu bởi p . |
| <code>inplace_merge(p1, mid, p2)</code> | Trộn hai dãy đã được sắp xếp trong khoảng $[p_1, mid]$ và $[mid, p_2]$ thành dãy được sắp xếp trong khoảng $[p_1, p_2]$. |
| <code>set_difference(p1, p2, t1, t2, p)</code> | Thêm các phần tử trong khoảng $[p_1, p_2]$ không nằm trong khoảng $[t_1, t_2]$ vào dãy bắt đầu bởi p (phép hiệu tập hợp). |
| <code>set_intersection(p1, p2, t1, t2, p)</code> | Thêm các phần tử nằm trong cả hai khoảng $[p_1, p_2]$ và $[t_1, t_2]$ vào dãy bắt đầu bởi p (phép giao tập hợp). |
| <code>includes(p1, p2, t1, t2)</code> | Chỉ trả về <code>true</code> nếu mọi phần tử trong khoảng $[t_1, t_2]$ là phần tử trong khoảng $[p_1, p_2]$ (phép chứa tập hợp). |

bản nháp, bản nháp,
bản nháp, bản nháp,