KS. TRỊNH QUỐC TIẾN

Visual C ++ 2008

HƯỚNG DẪN TỪNG BƯỚC
TỰ HỌC VÀ THỰC HÀNH

Visual C++ 2008

KÈM THEO CÁC BÀI TẬP ỨNG DỤNG

(TOÀN TẬP)

NHÀ XUẤT BẢN HỒNG ĐỨC

Hướng dẫn từng bước tự học và thực hành Visual C++ 2008

(Kèm theo các bài tập ứng dụng)

KS: TRỊNH QUỐC TIẾN
và nhóm tin học thực dụng

Hướng dẫn từng bước tự học và thực hành

Visual C++ 2008

(Kèm theo các bài tập ứng dụng)

(Toàn tập)

NHÀ XUẤT BẢN HỒNG ĐỨC

VISUAL C++ 2008

KS. TRỊNH QUỐC TIẾN

Chịu trách nhiệm xuất bản:
HOÀNG CHÍ DŨNG
Biên tập: NGUYỄN NAM
Bìa: LÊ THÀNH
Sửa bản in : HOÀNG SƠN

In 1.000 cuốn khổ (16x24)cm tại Công Ty Cổ Phần Văn Hóa Vạn Xuân.
Giấy đăng ký KHXB số 827-2007/CXB/50-18/HĐ cấp ngày 11/03/2008.
In xong và nộp lưu chiểu quý 2 năm 2008.

LỜI NÓI ĐẦU

C++ là một ngôn ngữ lập trình mạnh và linh hoạt với hàng trăm ngàn ứng dụng. Tuy nhiên, để biết cách tận dụng tối đa thế mạnh của ngôn ngữ C++ đòi hỏi phải có sự thực hành và trải nghiệm. Cuốn sách này sẽ là cầu nối đưa người học đến với thế giới lập trình đầy bí ẩn nhưng cũng lắm thú vị này, đặc biệt là trong môi trường phát triển phần mềm chuyên nghiệp Visual C++ 2008.

Thật vậy, "***Hướng dẫn từng bước tự học và thực hành Visual C++ 2008***" là một cuốn sách toàn tập hướng dẫn những người tự học Visual C++ 2008 các bước lập trình nhanh chóng từ lúc mới bắt đầu cho đến nâng cao. Sách gồm tám phần, được bố cục theo từng kỹ thuật thực hiện nhanh nhằm giúp tiết kiệm thời gian, đồng thời nâng cao kỹ năng lập trình trong Visual C++ 2008. Mỗi kỹ thuật có mã mẫu riêng mà người tự học có thể sử dụng trong các ứng dụng riêng của mình hoặc chỉnh sửa lại đôi chút cho phù hợp với mục đích mà mình mong muốn đạt được khi thiết kế và phát triển phần mềm, các bài tập ứng dụng được lồng ghép vào trong nội dung trình bày của mỗi kỹ thuật để vừa học vừa thực hành.

Ngoài ra, các kỹ thuật và mã được trình bày trong sách này không dành riêng cho hệ điều hành cụ thể nào mà áp dụng cho tất cả hệ điều hành có trình biên dịch hỗ trợ ngôn ngữ C++ chuẩn. Chính vì vậy, cuốn sách này sẽ hữu dụng cho cả những người lập trình Unix lẫn Microsoft Windows.

Hy vọng cuốn sách này sẽ giúp người học mở rộng kiến thức của mình về lập trình trong Visual C++ 2008, nắm vững các tính năng mạnh cũng như một số phương thức và thủ thuật hay để giải quyết nhanh chóng các vấn đề lập trình thường gặp hằng ngày, đáp ứng mục tiêu đặt ra.

Tác giả

Phần I

Hợp lý hóa phương tiện và cơ cấu của OOP

Kỹ thuật 1: Bảo vệ dữ liệu bằng Encapsulation

Tiết kiệm thời gian bằng cách

- Tìm hiểu encapsulation
- Tạo và thực thi một lớp đóng gói
- Thực hiện cập nhật đối với một lớp đóng gói

Từ điển định nghĩa encapsulation (sự đóng gói) là "đóng thùng hoặc như thể trong một bao" và đó chính xác là phương pháp mà C++ sử dụng. Một đối tượng (object) là một "bao" và thông tin và các thuật toán xử lý mà nó thực thi được che giấu khỏi người dùng. Tất cả những gì người dùng thấy là giao diện cấp chức năng cho phép sử dụng lớp (class) để làm công việc mà họ cần hoàn thành. Bằng cách đặt dữ liệu bên trong giao diện (interface), thay vì cho phép người dùng trực tiếp truy cập nó, dữ liệu được bảo vệ khỏi những giá trị không hợp lệ, các thay đổi sai hoặc sự chuyển đổi không đúng sang những kiểu dữ liệu mới.

Tạo và thực thi một lớp đóng gói

Listing 1.1 trình bày lớp StringCoding, một phương thức mã hóa đóng gói. Thực tế lợi ích của sự đóng gói (encapsulation) là: Nhà lập trình tận dụng lớp StringCoding không biết gì về thuật toán được sử dụng để mã hóa các chuỗi (string) - và không thật sự cần biết dữ liệu nào đã được sử dụng để mã hóa chuỗi lúc ban đầu. Tốt, nhưng tại sao làm như

vậy? Bạn có ba lý do tốt để "che giấu" sự thực thi của một thuật toán khỏi người dùng nó.

- Việc che giấu phần thực thi sẽ ngăn người ta vọc sửa dữ liệu nhập để làm cho thuật toán hoạt động khác. Những thay đổi này có thể nhằm mục đích làm cho thuật toán hoạt động chính xác, nhưng lại dễ dàng làm hỏng nó; theo một trong hai cách, việc can thiệp vào sẽ che giấu những lỗi có thể có khỏi những nhà phát triển.
- Việc che giấu thuật toán sẽ làm cho dễ dàng thay thế phần thực thi bằng một phần thực thi khác khả thi hơn nếu một phần thực thi được tìm thấy.
- Việc che giấu thuật toán làm cho người ta khó "crack" (bẻ khoá) mã và giải mã dữ liệu của bạn hơn.

Danh sách các bước sau đây hướng dẫn bạn cách tạo và thực thi phương thức đóng gói này:

1. Trong code editor mà bạn lựa chọn, tạo một file mới để chứa mã cho định nghĩa của file nguồn.

   Trong ví dụ này, file được đặt tên là ch01.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 1.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 1.1: Lớp StringCoding

```
#include <stdio.h>
#include <string>
class StringCoding
{
private:
    // The key to use in encrypting the string
    std::string sKey;
public:
    // The constructor, uses a preset key
    StringCoding( void )
    {
    sKey = "ATest";
    }
    // Main constructor, allows the user to specify a key
    StringCoding( const char *strKey )
    {
```

```
        if ( strKey )
            sKey = strKey;
        else
            sKey = "ATest";
    }
    // Copy constructor
    StringCoding( const StringCoding& aCopy )
    {
        sKey = aCopy.sKey;
    }
public:
    // Methods
    std::string Encode( const char *strIn );
    std::string Decode( const char *strIn );
private:
    std::string Xor( const char *strIn );
};
std::string StringCoding::Xor( const char *strIn )
{
    std::string sOut = "";
    int nIndex = 0;
    for ( int i=0; i<(int)strlen(strIn); ++i )
    {
        char c = (strIn[i] ^ sKey[nIndex]);
        sOut += c;
        nIndex ++;
        if ( nIndex == sKey.length() )
            nIndex = 0;
    }
    return sOut;
}
// For XOR encoding, the encode and decode methods are the same.
std::string StringCoding::Encode( const char *strIn )
{
    return Xor( strIn );
```

```
}
std::string StringCoding::Decode( const char *strIn )
{
    return Xor( strIn );
}
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        printf("Usage: ch1_1 inputstring1 [inputstring2...]\n");
        exit(1);
    }
    StringCoding key("XXX");
    for ( int i=1; i<argc; ++i )
    {
        std::string sEncode = key.Encode( argv[i] );
        printf("Input String : [%s]\n", argv[i] );
        printf("Encoded String: [%s]\n", sEncode.c_str() );
        std::string sDecode = key.Decode( sEncode.c_str() );
        printf("Decoded String: [%s]\n", sDecode.c_str() );
    }
    printf("%d strings encoded\n", argc-1);
    return 0;
}
```

3. Lưu mã nguồn dưới dạng một file trong ứng dụng code-editor và sau đó đóng code editor.
4. Biên dịch mã nguồn hoàn tất, sử dụng trình biên dịch (compiler) ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng mới trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh họa ở đây trong cửa sổ console của hệ điều hành:

```
$ ./ch1_1.exe "hello"
Input String : [hello]
Encoded String: [0=447]
Decoded String: [hello]
1 strings encoded
```

Chú ý rằng chuỗi đầu vào và chuỗi được giải mã thì như nhau - và chuỗi được mã hóa hoàn toàn không thể giải mã được (vì một chuỗi được mã hóa tốt thì khó có thể giải mã được). Và bất kỳ nhà lập trình sử dụng đối tượng sẽ không bao giờ thấy thuật toán đang quan tâm!

Thực hiện cập nhật đối với một lớp đóng gói

Một trong những lợi ích của sự đóng gói (encapsulation) là nó làm cho việc cập nhật dữ liệu ẩn trở nên đơn giản và tiện lợi. Với encapsulation, bạn có thể dễ dàng thay thế thuật toán mã hóa nền tảng trong Listing 1.1 bằng thuật toán khác nếu nhận thấy thuật toán đó làm việc tốt hơn. Trong thuật toán ban đầu, chúng ta đã thực hiện một "logic loại trừ or (hoặc)" để chuyển đổi một ký tự sang một ký tự khác. Trong ví dụ sau đây, giả sử chúng ta muốn sử dụng một phương pháp khác để mã hóa các chuỗi. Vì mục đích đơn giản, giả sử rằng thuật toán mới này mã hóa các chuỗi đơn giản bằng cách thay đổi mỗi ký tự trong chuỗi đầu vào sang vị trí mẫu tự kế tiếp trong bảng chữ cái: Một a trở thành một b, một c trở thành một d... Rõ ràng, thuật toán giải mã phải làm chính xác điều ngược lại, lấy chuỗi đầu vào trừ cho một vị trí mẫu tự để trả về một chuỗi đầu ra hợp lệ. Sau đó chúng ta có thể chỉnh sửa phương thức Encode trong Listing 1.1 để phản ánh sự thay đổi này. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Mở lại file nguồn trong code editor.

   Trong ví dụ này, chúng ta gọi file nguồn là ch01.cpp.

2. Chỉnh sửa mã như được minh họa trong Listing 1.2.

Listing 1.2: Cập nhật lớp StringCoding

```
std::string StringCoding::Encode( const char *strIn )
{
    std::string sOut = "";
    for ( int i=0; i<(int)strlen(strIn); ++i )
    {
        char c = strIn[i];
        c ++;
        sOut += c;
    }
    return sOut;
}
std::string StringCoding::Decode( const char *strIn )
```

```
{
    std::string sOut = "";
    for ( int i=0; i<(int)strlen(strIn); ++i )
    {
        char c = strIn[i];
        c --;
        sOut += c;
    }
    return sOut;
}
```

3. Lưu mã nguồn dưới dạng một file trong code editor và sau đó đóng code editor.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trên hệ điều hành ưa thích.

Bạn có thể nghĩ phương pháp này có một ảnh hưởng đối với các nhà phát triển nào đã sử dụng lớp. Thật ra, chúng ta có thể thực hiện những thay đổi này trong lớp (kiểm tra chương trình vừa có được trên Web site đi kèm của sách này với tên ch1_1a.cpp) và để yên phần còn lại của ứng dụng. Các nhà phát triển không cần bận tâm về điều này. Khi chúng ta biên dịch và chạy ứng dụng này, chúng ta có được kết quả sau đây:

```
$ ./ch1_1a.exe "hello"
Input String : [hello]
Encoded String: [ifmmp]
Decoded String: [hello]
1 strings encoded
```

Như bạn có thể thấy, thuật toán đã thay đổi, tuy nhiên việc mã hóa và giải mã vẫn diễn ra và mã ứng dụng đã không thay đổi gì cả. Vậy thì đây là sức mạnh thật sự của encapsulation: Nó là một hộp đen (black box). Những người dùng cuối không cần biết một điều gì đó làm việc như thế nào để sử dụng nó; họ chỉ cần biết những gì nó làm và cách làm cho nó thực hiện công việc của nó.

Kỹ thuật 2: Sử dụng Abstraction để mở rộng chức năng

Tiết kiệm thời gian bằng cách

- Tìm hiểu abstraction
- Sử dụng các phương thức ảo
- Tạo một ứng dụng danh sách thư tín
- Test các ứng dụng

Từ điển American Heritage Dictionary định nghĩa thuật ngữ abstraction (sự trừu tượng hóa) là "tiến trình không xem xét một hay nhiều thuộc tính của một đối tượng phức tạp để chăm sóc cho những đối tượng khác". Về cơ bản, điều này có nghĩa là chúng ta cần chọn lựa các phần của các đối tượng quan trọng đối với chúng ta. Để trừu tượng hóa (abstract) dữ liệu, chúng ta chọn đóng gói các phần đó của đối tượng chứa các loại chức năng cơ bản nhất định (các đối tượng cơ sở) - theo cách đó chúng ta có thể tái sử dụng chúng trong những đối tượng khác vốn định nghĩa lại chức năng đó. Các đối tượng cơ bản như vậy được gọi là các lớp cơ sở (base class). Các đối tượng được mở rộng được gọi là các lớp thừa kế (inherited class). Chúng cùng hình thành một nguyên lý cơ bản của C++. Sự trừu tượng hóa (abstraction) trong C++ được cung cấp thông qua phương thức ảo thuần túy. Phương thức ảo thuần túy (pure virtual method) là một phương thức trong một lớp cơ sở vốn phải được thực thi trong bất kỳ lớp dẫn xuất (derived class) để biên dịch và sử dụng lớp dẫn xuất đó.

Tạo một ứng dụng danh sách thư tín

Khái niệm này hơi trừu tượng, do đó sau đây là một ví dụ cụ thể để cho bạn thấy sự trừu tượng hóa thật sự diễn ra như thế nào: Giả sử bạn muốn thực thi một danh sách thư tín (mailing list) cho công ty của bạn. Danh sách thư tín này bao gồm các đối tượng (được gọi là các mailing-list entries) tượng trưng cho từng người mà bạn muốn liên lạc. Tuy nhiên, giả sử bạn phải tải dữ liệu từ một trong hai nguồn: từ một file chứa tất cả tên hoặc trực tiếp từ dòng lệnh của người dùng. Việc xem toàn bộ "dòng chảy" của ứng dụng này sẽ cho thấy rằng hai phía của hệ thống này có chung nhiều điểm: Để xử lý đầu vào từ một file, chúng ta cần một nơi nào đó để lưu trữ các tên, địa chỉ, thành phố, tiểu bang và mã zip từ một file. Để xử lý đầu vào từ dòng lệnh, chúng ta cần có khả năng tải chính xác cùng dữ liệu đó từ dòng lệnh và lưu trữ nó trong cùng một nơi. Sau đó, chúng ta cần khả năng in các mục danh sách thư tín hoặc trộn chúng vào một tài liệu khác. Sau

khi đầu vào được lưu trữ trong bộ nhớ, dĩ nhiên chúng ta thật sự không quan tâm nó đã đến đó như thế nào; chúng ta chỉ quan tâm làm thế nào chúng ta có thể truy cập dữ liệu trong các đối tượng. Hai đường dẫn khác nhau, dựa vào file và dựa vào dòng lệnh, chia sẻ cùng một thông tin cơ bản; thay vì thực thi thông tin hai lần, chúng ta có thể trừu tượng hóa nó thành một đối tượng chứa (container) cho dữ liệu danh sách thư tín. Sau đây là cách thực hiện điều đó:

1. Trong code editor mà bạn chọn lựa, tạo một file mới để chứa mã cho định nghĩa của lớp.

   Trong ví dụ này, file được đặt tên là ch02.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 2.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 2.1: Lớp BaseMailingListEntry

```
#include <string>
#include <iostream>
#include <stdio.h>
class BaseMailingListEntry
{
private:
    std::string sFirstName;
    std::string sLastName;
    std::string sAddressLine1;
    std::string sAddressLine2;
    std::string sCity;
    std::string sState;
    std::string sZipCode;
public:
    BaseMailingListEntry(void)
    {}
    BaseMailingListEntry( const BaseMailingListEntry& aCopy )
    {
        sFirstName = aCopy.sFirstName;
        sLastName = aCopy.sLastName;
        sAddressLine1 = aCopy.sAddressLine1;
        sAddressLine2 = aCopy.sAddressLine2;
```

```
            sCity = aCopy.sCity;
            sState = aCopy.sState;
            sZipCode = aCopy.sZipCode;
        }
        virtual bool First(void) = 0; // A pure virtual function
        virtual bool Next(void) = 0; // Another pure virtual function
        // Accessors
        std::string getFirstName() { return sFirstName; };
        std::string getLastName() { return sLastName; };
        std::string getAddress1() { return sAddressLine1; };
        std::string getAddress2() { return sAddressLine2; };
        std::string getCity() { return sCity; };
        std::string getState() { return sState; };
        std::string getZipCode() { return sZipCode; };
        void setFirstName(const char *strFirstName)
    { sFirstName = strFirstName; };
        void setLastName(const char *strLastName)
    { sLastName = strLastName; };
        void setAddress1( const char *strAddress1)
    { sAddressLine1 = strAddress1; };
        void setAddress2( const char *strAddress2)
    { sAddressLine2 = strAddress2; };
        void setCity(const char *strCity)
    { sCity = strCity; };
        void setState(const char *strState)
    { sState = strState; };
    void setZipCode( const char *strZipCode )
    { sZipCode = strZipCode; };
    };
```

Chú ý trong Listing 2.1, lớp cơ sở (lớp ??) chứa tất cả dữ liệu mà chúng ta sẽ sử dụng chung cho hai lớp dẫn xuất (các lớp File MailingListEntry và CommandLineMailing ListEntry), và thực thi hai phương thức - First và Next, vốn cho phép những lớp dẫn xuất đó ghi đè các tiến trình tải các thành phần của dữ liệu (cho dù từ một file hoặc dòng lệnh).

3. Lưu file trong bộ soạn thảo mã nguồn.
4. Sử dụng code editor ưa thích, thêm mã trong Listing 2.2.

Đôi khi bạn có thể tùy ý lưu mã này trong một file header riêng biệt và cũng đưa file header đó vào chương trình chính.

Listing 2.2: Lớp FileMailingListEntry Class

```
class FileMailingListEntry : public BaseMailingListEntry
{
    FILE *fpIn;
public:
    FileMailingListEntry( const char *strFileName )
    {
       fpIn = fopen(strFileName, "r");
    }
    virtual bool ReadEntry(void)
    {
       char szBuffer[ 256 ];
       fread( szBuffer, sizeof(char), 255, fpIn );
       if ( feof(fpIn) )
          return false;
       setFirstName( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setFirstName( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setAddress1( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setAddress2( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setCity( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setState( szBuffer );
       fread( szBuffer, sizeof(char), 255, fpIn );
       setZipCode( szBuffer );
       return true;
    }
    virtual bool First(void)
```

```
    {
        // Move to the beginning of the file, read in the pieces
        fseek( fpIn, 0L, SEEK_SET );
        return ReadEntry();
    }
    virtual bool Next(void)
    {
        // Just get the next one in the file
        return ReadEntry();
    }
};
```

5. Lưu file nguồn trong bộ soạn thảo mã nguồn.
6. Sử dụng code editor, thêm mã trong Listing 2.3 vào file mã nguồn.

   Bạn có thể tùy ý lưu mã này trong một file header riêng biệt và cũng đưa file header đó vào chương trình chính.
7. Lưu file nguồn trong bộ soạn thảo mã nguồn.

Listing 2.3: Lớp CommandLineMailingListEntry

```
class CommandLineMailingListEntry : public BaseMailingListEntry
{
private:
    bool GetALine( const char *prompt, char *szBuffer )
    {
        puts(prompt);
        gets(szBuffer);
        // Remove trailing carriage return
        szBuffer[strlen(szBuffer)-1] = 0;
        if ( strlen(szBuffer) )
            return true;
        return false;
    }
    bool GetAnEntry()
    {
        char szBuffer[ 80 ];
        if ( GetALine( "Enter the last name of the person: ",
```

```
szBuffer ) != true )
            return false;
        setLastName( szBuffer );
        GetALine("Enter the first name of the person: ",
szBuffer );
setFirstName( szBuffer );
        GetALine("Enter the first address line:", szBuffer );
        setAddress1(szBuffer);
        GetALine("Enter the second address line:", szBuffer );
        setAddress2(szBuffer);
        GetALine("Enter the city:", szBuffer );
        setCity(szBuffer);
        GetALine("Enter the state:", szBuffer);
        setState(szBuffer);
        GetALine("Enter the zip code:", szBuffer );
        setZipCode( szBuffer);
        return true;
    }
public:
    CommandLineMailingListEntry() {
}
virtual bool First(void)
{
    printf("Enter the first name for the mailing list:\n");
    return GetAnEntry();
}
virtual bool Next(void)
    {
        printf("Enter the next name for the mailing list:\n");
        return GetAnEntry();
    }
};
```

Test ứng dụng Mailing-List

Sau khi bạn tạo một lớp, điều quan trọng là phải tạo một driver thử nghiệm để không chỉ bảo đảm mã chính xác mà còn hướng dẫn mọi người cách sử dụng mã của bạn. Các bước sau đây trình bày cách thực hiện:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch02.cpp.

2. Gõ nhập mã từ Listing 2.4 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

   Một phương pháp hiệu quả hơn là sao chép mã từ file nguồn trên Web site đính kèm của sách này.

Listing 2.4: Chương trình thử nghiệm Mailing-List

```
void ProcessEntries( BaseMailingListEntry *pEntry )
{
    bool not_done = pEntry->First();
    while ( not_done )
    {
        // Do something with the entry here.
        // Get the next one
        not_done = pEntry->Next();
    }
}
int main(int argc, char **argv)
{
    int choice = 0;
    printf("Enter 1 to use a file-based mailing list\n");
    printf("Enter 2 to enter data from the command line\n");
    scanf("%d", &choice );
    if ( choice == 1 )
    {
        char szBuffer[ 256 ];
        printf("Enter the file name: ");
        gets(szBuffer);
```

```
            FileMailingListEntry fmle(szBuffer);
            ProcessEntries( &fmle );
        }
    else
        if ( choice == 2 )
        {
            CommandLineMailingListEntry cmle;
            ProcessEntries( &cmle );
        }
        else
            printf("Invalid option\n");
        return 0;
}
```

Chức năng chính của driver thật sự không nhiều cho lắm - tất cả những gì nó làm là tạo bất kỳ đối tượng mà bạn muốn sử dụng. Hàm ProcessEntries là hàm thú vị bởi vì nó là một hàm làm việc trên một kiểu lớp vốn không làm bất cứ điều gì - nó không biết loại đối tượng mailing-list entry nào mà nó đang xử lý. Thay vào đó, nó làm việc từ một pointer dẫn sang lớp cơ sở. Nếu chạy chương trình này, bạn sẽ thấy nó làm việc như đã quảng cáo, như bạn có thể thấy trong Listing 2.5.

Tương tự bạn có thể tạo một file chứa tất cả entry đã được gõ nhập vào các trường khác nhau ở trên để nhập các trường (field) đó vào hệ thống. Bạn có thể làm tất cả điều này mà không thay đổi một dòng của hàm ProcessEntries. Đây là sức mạnh của các hàm ảo thuần túy, và do đó là sức mạnh của sự trừu tượng hóa (abstraction).

Listing 2.5: Chương trình Mainling-List

```
Enter 1 to use a file-based mailing list
Enter 2 to enter data from the command line
2
Enter the first name for the mailing list:
Enter the last name of the person: Telles
Enter the first name of the person: Matt
Enter the first address line: 10 Main St
Enter the second address line:
Enter the city: Anytown
Enter the state: NY
```

```
Enter the zip code: 11518
Enter the next name for the mailing list:
Enter the last name of the person:
```

Kỹ thuật 3: Tùy biến một lớp với các hàm ảo

Tiết kiệm thời gian bằng cách

- Tìm hiểu polymorphism
- Ghi đè các phần được chọn của một lớp
- Tùy biến các lớp vào thời gian chạy
- Sử dụng các destructor với các hàm ảo

Polymorphism (tính đa hình) là những gì xảy ra khi bạn gán những ý nghĩa khác nhau vào một symbol hoặc toán tử trong những ngữ cảnh khác nhau.

Cứ cho là như vậy, hàm ảo thuần túy trong C++ (được thảo luận trong kỹ thuật 2) rất hữu dụng, nhưng C++ cho chúng ta thêm một khía cạnh: Nhà lập trình có thể ghi đè chỉ các phần được chọn của một lớp mà không buộc chúng ta ghi đè toàn bộ lớp. Mặc dù một hàm ảo thuần túy đòi hỏi nhà lập trình thực thi chức năng, nhưng một hàm ảo cho phép bạn ghi đè chức năng đó chỉ nếu bạn muốn, đây là một sự khác biệt quan trọng.

Những thay đổi nhỏ đối với lớp dẫn xuất được gọi là các hàm ảo (virtual function) - thật ra, chúng cho phép một lớp dẫn xuất ghi đè chức năng trong một lớp cơ sở mà không làm cho bạn chỉnh sửa lớp cơ sở. Bạn có thể sử dụng khả năng này để định nghĩa chức năng mặc định của một lớp nào đó, trong khi vẫn cho những người dùng cuối lớp tin chỉnh chức năng đó cho những mục đích riêng của họ. Phương pháp này có thể sử dụng để xử lý lỗi hoặc để thay đổi cách một lớp nào đó xử lý việc in hoặc hầu như bất cứ điều gì khác. Phần tiếp theo sẽ hướng dẫn bạn cách tùy biến một lớp, sử dụng các hàm ảo để thay đổi hành vi của một phương thức lớp cơ sở vào thời gian chạy.

Tùy biến một lớp với Polymorphism

Để hiểu các lớp cơ sở có thể được tùy biến như thế nào bằng cách sử dụng khả năng đa hình được cung cấp bởi các hàm ảo, hãy xem một ví dụ đơn giản về việc tùy biến một lớp cơ sở trong C++.

1. Trong code editor mà bạn chọn lựa, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

Trong ví dụ này, file được đặt tên là ch03.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 3.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 3.1: Mã nguồn lớp cơ sở hàm ảo

```
#include <string>
#include <stdio.h>
class Fruit
{
public:
    Fruit()
    {}
    virtual ~Fruit()
    {
        printf("Deleting a fruit\n");
    }
    virtual std::string Color()
    {
        return "Unknown";
    }
    void Print()
    {
        printf("My color is: %s\n",
        Color().c_str() );
    }
};
class Orange : public Fruit
{
public:
    Orange()
    {
    }
    virtual std::string Color()
    {
        return "Orange";
```

```
    }
};
class Apple : public Fruit
{
public:
    Apple()
    {
    }
    virtual std::string Color()
    {
        return "Reddish";
    }
};
class Grape : public Fruit
{
public:
    Grape()
    {
    }
    virtual std::string Color()
    {
        return "Red";
    }
};
class GreenGrape : public Grape
{
public:
    GreenGrape()
    {
    }
    virtual std::string Color()
    {
        return "Green";
    }
};
```

Test mã hàm ảo

Bây giờ bạn nên test mã. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Mở file nguồn ch03.cpp trong bộ soạn thảo mã nguồn ưa thích và thêm mã trong Listing 3.2 vào cuối file.

Listing 3.2: Driver chính cho mã hàm ảo

```
int main(int argc, char **argv)
{
    // Create some fruits
    Apple a;
    Grape g;
    GreenGrape gg;
    Orange o;
    // Show the colors.
    a.Print();
    g.Print()
    gg.Print();
    o.Print();
    // Now do it indirectly
    Fruit *f = NULL;
    f = new Apple();
    f->Print();                    →1
    delete f;
    f = new GreenGrape();
    f->Print();                    →2
    delete f;
}
```

2. Lưu mã nguồn trong bộ soạn thảo mã nguồn.

Có một vài điều thú vị cần ghi chú trong ví dụ này. Bạn có thể thấy lớp cơ sở gọi các phương thức được ghi đè như thế nào mà không cần phải "biết" về chúng (xem các dòng được đánh dấu 1 và 2). Những gì làm nên điều kỳ diệu này là một bảng dò tìm cho các hàm ảo (thường được gọi là v-table) chứa các pointer dẫn sang tất cả phương thức trong lớp. Bảng này không thể nhìn thấy được trong mã, nó được tự động tạo ra bởi trình biên

dịch C++ trong khi tạo mã máy cho ứng dụng. Khi linker tìm thấy một lệnh gọi đến một phương thức vốn được khai báo là virtual, nó sử dụng bảng dò tìm (lookup table) để phân giải phương thức đó vào thời gian chạy, thay vì vào thời gian biên dịch. Dĩ nhiên, đối với các phương thức không ảo, mã đơn giản hơn nhiều và có thể được quyết định vào thời gian biên dịch. Điều này có nghĩa rằng các hàm ảo có một hao phí nào đó (về các yêu cầu bộ nhớ và tốc độ mã) - do đó nếu bạn không sử dụng chúng trong mã, không khai báo mọi thứ là virtual. Có thể dường như phản trực giác khi định nghĩa các hàm ảo trong mã nếu bạn không sử dụng chúng, nhưng điều này thật sự không phải như vậy. Trong nhiều trường hợp, bạn có thể thấy những công dụng sau này cho lớp cơ sở vốn sẽ đòi hỏi bạn cho phép nhà phát triển tương lai ghi đè chức năng để thêm những tính năng mới vào lớp dẫn xuất.

3. Lưu mã nguồn dưới dạng một file trong code editor và sau đó đóng ứng dụng editor.

4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

5. Chạy chương trình trên console hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh họa bên dưới trong cửa sổ console.

```
The color of the fruit is: Apple
The color of the fruit is: Red
The color of the fruit is: Green
The color of the fruit is: Orange
The color of the fruit is: Apple
Deleting a fruit
The color of the fruit is: Green
Deleting a fruit
Deleting a fruit
Deleting a fruit
Deleting a fruit
Deleting a fruit
```

Như bạn có thể thấy, các lệnh gọi trực tiếp đến mã làm việc tốt. Ngoài ra, bạn còn thấy mã sử dụng một pointer dựa vào lớp cơ sở để truy cập chức năng trong các lớp dẫn xuất gọi các phương thức ảo được ghi đè phù hợp. Điều này để lại cho chúng ta một câu hỏi là các

phương thức hủy tạo (destructor) lớp dẫn xuất được gọi ra như thế nào và theo thứ tự nào. Hãy xem phương thức ảo cuối cùng đó - phương thức hủy tạo ảo (virtual destructor) trong lớp Fruit cơ sở.

Tại sao các Destructor làm việc?

Điều thú vị ở đây là destructor cho lớp cơ sở luôn được gọi. Bởi vì destructor được khai báo là virtual, destructor móc nối hướng lên qua các destructor cho những lớp khác vốn được dẫn xuất từ lớp cơ sở. Nếu chúng ta đã tạo các destructor cho mỗi lớp dẫn xuất và in ra kết quả thì ví dụ nếu chúng ta tạo một lớp PurpleGrape GreenGrape mới, vốn được dẫn xuất từ Grape, chúng ta thấy kết quả như sau:

```
PurpleGrape destructing
Grape destructing
Deleting a fruit
```

Cũng chú ý bảng ảo (virtual table) cho một lớp cơ sở có thể bị ảnh hưởng bởi mọi lớp được dẫn xuất từ nó (như chúng ta có thể thấy qua lớp GreenGrape). Khi gọi ra phương thức Print trên một đối tượng Fruit đã được tạo dưới dạng một lớp GreenGrape được dẫn xuất từ Grape, phương thức được gọi ra tại cấp lớp Grape. Điều này có nghĩa bạn có thể có bao nhiêu cấp thừa kế tùy thích. Như bạn có thể thấy, chức năng phương thức ảo (virtual-method) trong C++ cực kỳ hữu dụng.

Tóm lại, sau đây là hệ thống phân cấp để gọi đúng phương thức Print khi một đối tượng GreenGrape được chuyển đến một hàm vốn chấp nhận một đối tượng Fruit:

1. Fruit::Print được gọi ra.
2. Trình biên dịch xem bảng hàm ảo (v-table) và tìm entry (mục nhập) cho phương thức Print.
3. Phương thức được phân giải thành phương thức GreenGrape::Print.
4. Phương thức GreenGrape::Print được gọi.

Kỹ thuật 4: Thừa kế dữ liệu và chức năng

Tiết kiệm thời gian bằng cách

- Định nghĩa sự đa thừa kế
- Thực thi lớp configuration file
- Test lớp configuration file
- Trì hoãn việc tạo
- Xử lý lỗi bằng sự đa thừa kế

Nói chung, loại chức năng lớn nhất mà C++ phải cung cấp là sự thừa kế (inheritance) - chuyển các đặc tính từ một lớp cơ sở sang những lớp dẫn xuất của nó. Sự thừa kế là khả năng dẫn xuất một lớp mới từ một hoặc nhiều lớp cơ sở hiện có. Ngoài việc giảm bớt công sức viết mã, tính năng thừa kế trong C++ có nhiều công dụng tuyệt vời; bạn có thể mở rộng, tùy biến hoặc thậm chí giới hạn chức năng hiện có. Kỹ thuật này xem xét sự thừa kế và hướng dẫn bạn cách sử dụng sự đa thừa kế (multiple inheritance - một tính năng tiện lợi nhưng ít được biết đến) kết hợp lớp tốt nhất của một số lớp thành một lớp đơn cho người dùng cuối.

Để thật sự hiểu điều gì đang xảy ra, bạn phải hiểu một số điều về cách các trình biên dịch C++ thực thi sự thừa kế - và cách ngôn ngữ tận dụng phương pháp này như thế nào.

Mỗi lớp C++ chứa ba phần, mỗi phần có mục đích riêng của nó:

- **Sự lưu trữ cho dữ liệu vốn thuộc về lớp:** Mọi lớp cần dữ liệu để làm việc và phần này của lớp giữ cho dữ liệu có sẵn.
- **Các jump table (bảng nhảy):** Những bảng này lưu trữ các phương thức static của lớp để trình biên dịch có thể tạo các chỉ lệnh hiệu quả để gọi các phương thức trong của lớp.
- **Một v-table tùy chọn cho các phương thức ảo:** Nếu một lớp không cung cấp sự thừa kế, có thể có một v-table tùy chọn chứa các địa chỉ của bất kỳ phương thức ảo trong lớp. Sẽ không bao giờ có nhiều virtual table (bảng ảo) mỗi lớp, bởi vì bảng đó chứa các pointer dẫn sang tất cả phương thức ảo trong lớp.

Nhưng tại sao sự thừa kế làm việc? Bởi vì trình biên dịch tạo một "ngăn xếp" dữ liệu, theo sau là một "ngăn xếp" các phương thức, việc thực thi bất kỳ số cấp thừa kế không có vấn đề gì cả. Các cấp thừa kế xác định thứ tự của các "ngăn xếp". Nếu một lớp được dẫn xuất từ các lớp A, B, và C, bạn sẽ thấy ngăn xếp các phương thức cho A, theo sau là những ngăn xếp phương thức cho B, rồi đến các ngăn xếp phương thức cho C. Theo cách này, trình biên dịch có thể dễ dàng chuyển đổi lớp dẫn xuất thành bất kỳ lớp cơ sở của nó bằng cách chọn một điểm

trong ngăn xếp để bắt đầu. Ngoài ra, bởi vì bạn có thể thừa kế dữ liệu từ các lớp mà bản thân thừa kế từ những lớp khác, toàn bộ tiến trình tạo một tầng dữ liệu và phương thức. Đây là một điều tốt, bởi vì nó có nghĩa là cấu trúc lớp dễ dàng phù hợp với những chuyển đổi từ lớp cơ sở sang lớp dẫn xuất.

Thực thi một lớp ConfigurationFile

Đối với các mục đích của ví dụ này, giả sử bạn muốn thực thi một lớp file cấu hình (configuration file). Lớp này sẽ cho phép bạn lưu trữ thông tin cấu hình cho ứng dụng trong một file ngoài và truy cập nó một cách nhất quán qua suốt mã nguồn chương trình. Các file cấu hình có hai tập hợp chức năng cơ bản - một tập hợp thuộc tính (tượng trưng cho các cặp tên và giá trị) và một trình quản lý file (vốn đọc và ghi các cặp đó qua lại đĩa). Để tất cả điều này làm việc tốt, bạn phải thực thi chức năng cho lớp một cách chính xác theo cách đó: đầu tiên là các thuộc tính, sau đó đến sự quản lý chúng. Bạn nên có một lớp cơ sở thực thi sự quản lý thuộc tính và một lớp khác làm việc với chính file đĩa.

Do đó, sau đây là cách thực thi lớp.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file này được đặt tên là ch04.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 4.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 4.1: Mã nguồn Properties.

```
#include <string>
#include <stdio.h>
#include <vector>
class Properties
{
private:
    struct _Prop
    {
        public:
            std::string name;
            std::string value;
        public:
```

```
            _Prop operator=(const _Prop&
            aCopy )
            {
               name = aCopy.name;
               value = aCopy.value;
               return *this;
            }
      };
      std::vector< _Prop > sProps;
public:
      Properties(void)
      {
      }
virtual ~Properties()                          →1
      {
      }
      Properties( const Properties& aCopy )
      {
         std::vector< _Prop >::const_iterator
         iter;

         for ( iter = aCopy.sProps.begin();
         iter != aCopy.sProps.end(); ++iter )
            sProps.insert( sProps.end(),
            (*iter) );
      }
      int NumProperties( void )
      {
         return (int)sProps.size();
      }
      bool GetProperty( int idx, std::string&
      name, std::string& value )
      {
         if ( idx < 0 || idx >=
         NumProperties() )
```

```
            return false;
        name = sProps[idx].name;
        value = sProps[idx].value;
        return true;
    }
    void AddProperty( const std::string&
    name, const std::string& value )
    {
        _Prop p;
        p.name = name;
        p.value = value;
        sProps.insert( sProps.end(), p );
    }
};
```

Chú ý lớp này sử dụng Standard Template Library (STL) mà sẽ được trình bày thêm chi tiết trong phần V của sách này. Bây giờ, bạn chỉ việc giả định rằng lớp vector thực thi một mảng generic vốn có thể được mở rộng. Lớp vector không đòi hỏi số phần tử tối thiểu và có thể được mở rộng đến mức độ bộ nhớ cho phép.

Lớp thuộc tính sẽ hình thành cơ sở cho một loạt các kiểu thuộc tính, tất cả có thể xử lý các kiểu thuộc tính khác nhau. Ngoài ra, lớp này có thể được sử dụng làm cơ sở cho những lớp khác vốn cần khả năng lưu trữ thông tin thuộc tính.

Thật sự không có điều kỳ diệu ở đây; bạn có thể thấy rằng lớp đơn giản duy trì các tập hợp thuộc tính và có thể thêm chúng hoặc cung cấp chúng trở lại cho đối tượng gọi. Tuy nhiên, chú ý rằng bạn đã thực thi một destructor ảo (xem 1) cho lớp - mặc dù không có gì trong lớp chưa cần được hủy tạo. Thật sự không có cách nào để biết liệu điều này sẽ luôn đúng hay không, do đó bạn cũng có thể giả định rằng destructor sẽ cần thực hiện công việc làm sạch của nó vào một thời điểm nào đó. Bạn xây dựng lớp này một cách có chủ đích dưới dạng một lớp cơ sở cho sự thừa kế, do đó sẽ chỉ hợp lý nếu làm cho destructor trở nên ảo (virtual). Nếu destructor ảo, tất cả lớp dẫn xuất sẽ gọi destructor lớp cơ sở như là phần cuối cùng của tiến trình hủy tạo, bảo đảm rằng tất cả bộ nhớ cấp phát được giải phóng.

Bước kế tiếp là thực thi lớp vốn quản lý phần file của hệ thống. Đối với các mục đích của không gian, chỉ đoạn viết của lớp được minh họa trong Listing 4.2. Tuy nhiên, thực thi một phương thức ReadAPair vốn truy cập dữ liệu từ một file thì khá bình thường.

3. Sử dụng code editor, thêm mã từ Listing 4.2 vào file mã nguồn. Trong trường hợp này, file được gọi là ch04.cpp.

Listing 4.2: Lớp SavePairs

```
class SavePairs
{
    FILE *fpIn;
public:
    SavePairs( void )
    {
       fpIn = NULL;
    }
    SavePairs( const char *strName )
    {
       fpIn = fopen( strName, "w" );
    }
    virtual ~SavePairs()
    {
       if ( fpIn )
          fclose(fpIn);
    }
    void SaveAPair( std::string name,
    std::string value )
    {
       if ( fpIn )
          fprintf(fpIn, "%s=%s\n",
          name.c_str(), value.c_str());
    }
};
    ) )
                SaveAPair( name, value );
       }
       return true;
    }
};
```

Một lần nữa, bạn thực thi một virtual destructor (phương thức hủy tạo ảo) cho lớp bởi vì nó được ấn định là một lớp cơ sở cho sự thừa kế; cụ thể về những gì phải hủy chưa có ích lợi gì. Tuy nhiên, bạn có một công dụng thật sự của destructor, bởi vì file pointer vốn mở trong constructor (phương thức tạo) phải có một chỉ lệnh đóng (fclose) tương ứng để giải phóng bộ nhớ và xóa sạch file sang đĩa.

Với virtual destructor nằm ở đúng vị trí, việc duy nhất còn lại cần làm là kết hợp hai lớp khá hữu dụng này thành một lớp đơn vốn chứa chức năng của cả hai và cung cấp một giao diện cố kết cho người dùng cuối của lớp. Chúng ta sẽ gọi lớp kết hợp này là ConfigurationFile.

4. Sử dụng code editor, thêm mã trong Listing 4.3 vào file mã nguồn.

Listing 4.3: Lớp ConfugurationFile

```
class ConfigurationFile : public Properties,
    public SavePairs
{
public:
    ConfigurationFile(void)                          → 2
       : SavePairs()
    {
    }
    ConfigurationFile(const char
    *strFileName)                                    → 3
       : SavePairs(strFileName)
    {
    }
    virtual ~ConfigurationFile()
    {
       DoSave();
    }
 bool DoSave()                                       → 4
    {
       std::string name;
       std::string value;
       for (int i=0; i<NumProperties(); ++i)
```

```
{
    if ( GetProperty( i, name, value
```

5. Lưu mã nguồn trong code editor.

Test lớp ConfigurationFile

Sau khi bạn tạo một lớp, hãy tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã chính xác mà còn hướng dẫn người ta cách sử dụng mã.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch04.cpp. Dĩ nhiên, bạn có thể đặt cho chương trình này bất kỳ tên nào bạn muốn vì các tên file chỉ là các chuỗi mà con người đọc được. Trình biên dịch không quan tâm tên nào mà bạn đặt cho file.

2. Gõ nhập mã từ Listing 4.4 vào file.

   Hoặc tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 4.4: Chương trình thử nghiệm ConfigurationFile

```
int main(int argc, char **argv)
{
    ConfigurationFile cf("test.dat");
    cf.AddProperty( "Name", "Matt" );
    cf.AddProperty( "Address", "1000 Main
    St" );
}
```

3. Lưu mã trong file mã nguồn được tạo trong editor và sau đó đóng ứng dụng editor.
4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy chương trình trên console hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây từ chương trình trên cửa sổ console:

```
$ ./a.exe
$ cat test.dat
```

```
Name=Matt
Address=1000 Main St
```

Như bạn có thể thấy, file cấu hình đã được lưu đúng cách sang file xuất.

Trì hoãn việc tạo

Mặc dù constructor cho một lớp thì tuyệt vời và tốt, nhưng nó đưa ra một điểm thú vị. Phải làm gì nếu một trục trặc nào đó xảy ra trong tiến trình tạo và bạn cần báo hiệu cho người dùng? Bạn có hai cách để tiếp cận tình huống này; cả hai có những ưu điểm và khuyết điểm:

- Bạn có thể đưa ra một ngoại lệ (exception). Đưa ra các ngoại lệ là một tùy chọn được thảo luận trong kỹ thuật 53 - nhưng làm như vậy hiếm khi là một ý hay. Những người dùng của bạn thật sự không mong đợi một constructor đưa ra một ngoại lệ. Tệ hơn, một ngoại lệ có thể để đối tượng trong một trạng thái mơ hồ nào đó, nơi mà không biết rõ liệu constructor đã chạy xong hay không. Nếu bạn chọn lộ trình này, bạn cũng nên bảo đảm tất cả giá trị được in nghiêng trước khi bạn làm bất cứ điều gì vốn có thể tạo ra một ngoại lệ. (Ví dụ, điều gì xảy ra nếu bạn đưa ra một ngoại lệ trong một constructor lớp cơ sở? Lỗi sẽ được chuyển lên chương trình chính. Điều này rất dễ gây bối rối cho người dùng mà thậm chí không biết lỗi đến từ đâu).
- Bạn có thể trì hoãn bất kỳ công việc vốn có thể tạo ra một lỗi cho đến thời điểm sau trong việc xử lý đối tượng. Lựa chọn này thường có giá trị hơn và đáng được khai thác thêm.

Ví dụ, bạn sẽ mở m file trong constructor. Tiến trình mở file có thể chắc chắn thất bại vì bất kỳ số lý do. Một cách để xử lý lỗi này là kiểm tra nó, nhưng điều này có thể gây bối rối cho người dùng cuối, bởi vì họ sẽ không hiểu file đã được mở ở đâu lúc ban đầu và tại sao nó không mở được. Trong những trường hợp như vậy, thay vì một constructor trông như sau...

```
FileOpener::FileOpener( const char
    *strFileName)
{
    fpIn = fopen(strFileName, "r");
}
```

... bạn có thể chọn làm điều như sau:

```
FileOpener::FileOpener( const char
    *strFileName)
{
```

```
    // Hold onto the file name for later
    use.
    sFileName = strFileName;
    bIsOpen = false;
}
bool FileOpener::OpenFile()                          → 5
{
    if ( !bIsOpen )
    {
        fpIn = fopen(sFileName.c_str(),
        "r");
        if ( fpIn != NULL )
            bIsOpen = true;
    }
    return bIsOpen;
}
```

Bởi vì chúng ta không thể trả về trực tiếp một lỗi từ một constructor, chúng ta phân chia tiến trình thành hai phần. Phần đầu tiên gán các biến thành viên vào những giá trị mà người dùng đã chuyển vào. Không có cách nào một lỗi có thể xảy ra trong tiến trình này, do đó đối tượng sẽ được tạo đúng cách. Trong phương thức OpenFile (5 trong listing ở trên), chúng ta cố mở file và biểu thị trạng thái là giá trị trả về của phương thức.

Sau đó, khi bạn yêu cầu mã thật sự đọc từ file, bạn làm một điều gì đó như sau:

```
bool FileOpener::SomeMethod()
{
    if ( OpenFile() )
    {
        // Continue with processing
    }
    else
        // Generate an error
        return false;
}
```

Ưu điểm của phương pháp này là bạn có thể đợi cho đến khi bạn hoàn toàn trước khi bạn thật sự mở file mà lớp vận hành trên đó. Việc làm điều này có nghĩa là bạn không có hao phí file mỗi lần bạn tạo một do - và bạn không cần bận tâm về việc đóng thứ khó chịu đó nếu nó đã không bao giờ được mở. Ưu điểm của việc trì hoãn việc tạo là bạn có thể đợi cho đến khi dữ liệu thật sự được cần đến trước khi thực hiện hoạt động nhập và xuất đòi hỏi nhiều thời gian và bộ nhớ.

Xem kỹ lại lớp SavePairs (Listing 4.2), bạn có thể thấy một lỗi rất nghiêm trọng ẩn nấp ở đó.

Bạn có thấy nó hay không? Hãy tưởng tượng bạn có một đối tượng có kiểu SavePairs. Bây giờ bạn có thể tạo một bản sao của đối tượng đó bằng cách gán nó vào một đối tượng khác của lớp SavePairs, hoặc bằng cách chuyển nó theo giá trị vào một phương thức như sau:

```
DoSave(SavePairs obj);
```

Khi bạn thực hiện lệnh gọi hàm ở trên, bạn tạo một bản sao của đối tượng obj bằng cách gọi ra constructor copy cho lớp. Bây giờ, bởi vì bạn không tạo một constructor copy, bạn có một vấn đề nghiêm trọng. Tạo sao? Copy là một bản sao bitwise của tất cả phần tử trong lớp. Khi một bản sao được tạo của pointer FILE trong lớp, điều này có nghĩa bây giờ bạn có hai pointer trỏ sang cùng một khối bộ nhớ. Bởi vì bạn sẽ hủy bộ nhớ đó trong destructor cho lớp (bằng cách gọi fclose), mã giải phóng cùng một khối bộ nhớ hai lần. Đây là một vấn đề đặc trưng mà bạn cần giải quyết bất cứ khi nào bạn cấp phát bộ nhớ trong một lớp. Trong trường hợp này, bạn thật sự muốn có thể sao chép pointer mà không đóng nó trong bản sao. Do đó, những gì bạn thật sự cần làm là theo dõi việc pointer đang quan tâm có phải là một bản sao hoặc một bản gốc hay không. Để làm điều này, bạn có thể viết lại lớp như trong Listing 4.5:

Listing 4.5: Lớp SavePairs được sửa đổi

```
class SavePairs
{
    FILE *fpIn;
    bool bIsACopy;
public:
    SavePairs( void )
    {
        fpIn = NULL;
        bIsACopy = false;
    }
```

```
    SavePairs( const char *strName )
    {
        fpIn = fopen( strName, "w" );
        bIsACopy = false;
    }
    SavePairs( const SavePairs& aCopy )
    {
        fpIn = aCopy.fpIn;
        bIsACopy = true;
    }
    virtual ~SavePairs()
    {
        if ( fpIn && !bIsACopy )
        fclose(fpIn);
    }
    void SaveAPair( std::string name,
    std::string value )
    {
        if ( fpIn )
            fprintf(fpIn, "%s=%s\n",
            name.c_str(), value.c_str());
    }
};
```

Mã này trong Listing 4.5 có ưu điểm là làm việc chính xác cho dù nó được xử lý như thế nào. Nếu bạn chuyển một pointer vào file, mã sẽ tạo một bản sao của nó chứ không xóa nó. Nếu bạn sử dụng bản gốc của file pointer, nó sẽ được xóa đúng cách, không phải được sao chép.

Đây là một cải tiến. Nhưng mã này có thật sự sửa chữa tất cả vấn đề có thể xảy ra hay không? Dĩ nhiên, câu trả lời là không. Hãy tưởng tượng tình huống sau đây:

1. Tạo một đối tượng SavePairs.
2. Sao chép đối tượng bằng cách gọi constructor copy với một đối tượng mới.
3. Xóa đối tượng SavePairs gốc.
4. Gọi ra một phương thức trên bản sao nào sử dụng file pointer.

Điều gì xảy ra trong tình huống này? Không có gì tốt, bạn có thể bảo đảm. Vấn đề xảy ra khi đi đến bước cuối cùng và file pointer đã sao chép được sử dụng. Pointer gốc đã bị xóa, do đó bản sao trỏ vào những thứ linh tinh. Những điều tệ hại xảy ra - và chương trình có thể bị đổ vỡ.

Kỹ thuật 5: Tách biệt các quy tắc và dữ liệu với mã

Tiết kiệm thời gian bằng cách

- Sử dụng encapsulation để tách biệt các quy tắc và dữ liệu với mã
- Xây dựng một lớp hiệu lực hóa dữ liệu
- Test lớp hiệu lực hóa dữ liệu

Một trong những vấn đề lớn nhất trong thế giới phát triển phần mềm là bảo trì mã mà chúng ta đã không thiết kế hoặc thực thi lúc ban đầu. Điều thường khó làm nhất trong những trường hợp như vậy là biết chính xác mã được ấn định để làm việc như thế nào. Luôn có rất nhiều tài liệu cho bạn biết những gì mã thực hiện (hoặc những gì nhà lập trình ban đầu đã nghĩ sẽ xảy ra), nhưng hiếm khi nó cho bạn biết lý do tại sao.

Lý do là những quy tắc nghiệp vụ và dữ liệu vốn thực thi những quy tắc đó thường được nhúng ở nơi nào đó trong mã. Các ngày tháng, giá trị được mã hóa cứng - thậm chí các user name và password - có thể được ẩn giấu sâu bên trong cơ sở mã. Sẽ thật tuyệt vời hay không nếu có một cách nào đó để trích xuất tất cả dữ liệu và quy tắc nghiệp vụ đó và đặt chúng ở một nơi? Điều này thật sự nghe có vẻ như một trường hợp cho sự đóng gói (encapsulation) bây giờ phải không? Dĩ nhiên là đúng. Như được thảo luận trong kỹ thuật 1, encapsulation cho phép chúng ta cô lập người dùng khỏi việc thực thi mọi thứ. Câu nói đó mơ hồ và mang rất nhiều nghĩa, do đó để làm rõ ràng, hãy xem một vài ví dụ. Trước tiên, hãy xem xét trường hợp của quy tắc nghiệp vụ.

Khi bạn tạo mã cho một project phần mềm, bạn phải thường xem xét các quy tắc vốn áp dụng qua toàn bộ doanh nghiệp - chẳng hạn như số bộ phận trong một cơ sở dữ liệu kế toán, hoặc có lẽ một phép tính để quyết định số tiền tăng lương tối đa cho một nhân viên nào đó. Những quy tắc này xuất hiện dưới dạng các đoạn mã phân tán qua suốt toàn bộ project, thường trú trong các file và form khác nhau. Khi project kế tiếp xuất hiện, chúng thường được sao chép, được chỉnh

sửa hoặc được hủy bỏ. Vấn đề với phương pháp này là theo dõi các quy tắc là gì và ý nghĩa của chúng ngày càng trở nên khó hơn.

Bây giờ, giả sử bạn phải thực thi một số mã kiểm tra các ngày tháng trong hệ thống. Để chạy tiến trình kiểm tra, bạn có thể thử phân tán một số mã xung quanh toàn bộ hệ thống để kiểm tra tìm các năm nhuận, tính hiệu lực hóa dữ liệu... nhưng điều đó không hiệu quả và gây lãng phí. Sau đây là lý do tại sao giải pháp đó không có giải pháp gì cả:

Cách đây đã lâu, có một project được thực hiện tại một công ty rất lớn. Một cuộc kiểm toán phần mềm đã làm xuất hiện tối thiểu năm thường trình khác nhau (các hàm, macro và mã inline) vốn tính toán xem một năm nào đó có phải là một năm nhuần hay không. Điều này khá ngạc nhiên - nhưng thậm chí ngạc nhiên hơn là trong năm thường trình đó, ba thường trình thật sự sai. Nếu một lỗi đã xảy ra trong khi hệ thống đang tính toán xem năm hiện hành có phải là một năm nhuận hay không, nhà lập trình có biết nơi nào để tìm nhằm giải quyết vấn đề hay không? Dĩ nhiên là không.

Trong ví dụ này, bất kể rủi ro lỗi xảy ra, bạn vẫn phải quyết định xem một ngày tháng nào đó có hợp lệ hay không - và việc năm nào đó có phải là một năm nhuận hay không. Hai tác vụ cơ bản đầu tiên của bạn là xác lập các thiết lập mặc định thích hợp cho ngày tháng và bảo đảm rằng bạn có thể truy tìm tất cả thành phần của ngày tháng. Phương pháp này làm việc cho bất kỳ sự đóng gói quy tắc nghiệp vụ (business-rule). Đầu tiên, bạn phải biết các mảnh ghép của trò chơi chắp hình vốn đi vào các phép tính. Theo cách đó, bất cứ người nào xem mã sẽ biết chính xác những gì người này cần cung cấp. Sẽ không có dữ liệu "ẩn" trừ phi nó được truy tìm từ một nguồn bên ngoài. Mã nên plug-and-play (cắm vào là chạy); bạn nên có khả năng mang nó từ một project sang một project khác với những thay đổi tối thiểu.

Dĩ nhiên, thường hoàn toàn không thể loại bỏ mã ứng dụng ra khỏi những quy tắc nghiệp vụ. Nhưng đó thật sự không phải là mục đích của bạn khi bạn viết các đối tượng nghiệp vụ. Thay vào đó, bạn nên quan tâm đến cách những đối tượng đó sẽ được sử dụng như thế nào.

Đối tượng nên có tính mang chuyển (portable); nó sẽ được sử dụng trong nhiều project để hỗ trợ "quy tắc ngày tháng". Bạn muốn các ngày tháng hợp lệ và bạn muốn có khả năng trích xuất những thành phần của ngày tháng trong bất kỳ project vốn có thể cần dữ liệu đó. Đồng thời, bạn không muốn cho người ta nhiều hơn những gì họ cần, do đó bạn sẽ không bận tâm hỗ trợ phép toán ngày tháng, chẳng hạn như các phép tính để cộng các ngày hoặc năm với một ngày tháng nào đó.

Lớp cDate

Để đóng gói tốt nhất tất cả thông tin ngày tháng trong chương trình, cách dễ dàng nhất là tạo một lớp đơn quản lý việc lưu trữ, xử lý và xuất ngày tháng. Trong phần này, chúng ta tạo một lớp để thực hiện tất cả điều đó và gọi nó là cDate (dĩ nhiên là cho date class). Với một lớp ngày tháng (date class), chúng ta loại bỏ tất cả quy tắc và thuật toán để xử lý các ngày tháng chẳng hạn như các phép tính năm nhuận, phép toán ngày tháng và các phép tính ngày trong tuần và di chuyển chúng vào một nơi. Ngoài ra, chúng ta di chuyển việc lưu trữ ngày tháng, chẳng hạn như các thành phần ngày, tháng và năm được lưu trữ như thế nào vào một vùng mà người dùng không cần quan tâm đến.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file đó được đặt tên là ch05.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 5.1 vào file.

Listing 5.1: Lớp cDate

```
#include <string>
#include <stdio.h>
#include <time.h>
class cDate
{
private:
    int MonthNo;
    int DayOfMonth;
    int DayOfWeek;
    long YearNo;
protected:
    void GetTodaysDate()
    {
        // First, get the data
        time_t t;
        time(&t);
        struct tm *tmPtr = localtime(&t);
        // Now, store the pieces we care about
```

```
        MonthNo = tmPtr->tm_mon;
        YearNo = tmPtr->tm_year + 1900;
        DayOfMonth = tmPtr->tm_mday;
        DayOfWeek = tmPtr->tm_wday;
    }
    int ComputeDayOfTheWeek() // returns day of week
    {
        int sum_calc;
        int cent_off, year_off, month_off, day_off;
        int year_end;
        year_end = YearNo % 100; // year in century
// The following calculation calculates offsets for the
// century, year, month, and day to find the name of the
// weekday.
        cent_off = ((39 - (YearNo/100)) % 4 ) * 2;
        year_off = year_end + year_end/4;
    if (MonthNo == 1) // January
    {
        month_off = 0;
        if (((YearNo%4) == 0) && ((year_end !=0) ||
        ((YearNo%400) == 0)))
            year_off—; // leap year
    }
    else if (MonthNo == 2) // February
    {
        month_off = 3;
        if (((YearNo%4) == 0) && ((year_end !=0) ||
        ((YearNo%400) == 0)))
            year_off—; // leap year
    }
    else if ((MonthNo == 3) || (MonthNo == 11))
        month_off = 3;
    else if ((MonthNo == 4) || (MonthNo == 7))
        month_off = 6;
    else if (MonthNo == 5) // May
```

```
            month_off = 1;
        else if (MonthNo == 6) // June
            month_off = 4;
        else if (MonthNo == 8) // August
            month_off = 2;
        else if ((MonthNo == 9) || (MonthNo == 12))
            month_off = 5;
        else if (MonthNo == 10) // October
            month_off = 0;
        day_off = DayOfMonth % 7; // day offset
        sum_calc = (cent_off + year_off + month_off + day_off) % 7;
        // Using the calculated number, the remainder gives the day
        // of the week
        sum_calc %= 7;
        return sum_calc;
    }
    int MonthDays( int month, long year )
    {
        if ( month < 0 || month > 11 )
            return 0;
        int days[]={31,28,31,30,31,30,31,31,30,31,30,31 };
        int nDays = days[ month ];

            if ( IsLeapYear( year ) && month == 1)
                nDays ++;
            return nDays;
        }
    public:
        cDate(void)
        {
            // Get today's date
            GetTodaysDate();
        }
        cDate( int day, int month, long year )
        {
```

```
        if ( IsValidDate( day, month, year ) )
        {
            MonthNo = month;
            DayOfMonth = day;
            YearNo = year;
            DayOfWeek = ComputeDayOfTheWeek();
        }
    }
    cDate( const cDate& aCopy )
    {
        YearNo = aCopy.YearNo;
        MonthNo = aCopy.MonthNo;
        DayOfMonth = aCopy.DayOfMonth;
        DayOfWeek = aCopy.DayOfWeek;
    }
    // Accessors
    int Month() { return MonthNo; };
    long Year() { return YearNo; };
    int Day() { return DayOfMonth; };
    int DayOfTheWeek() { return DayOfWeek; };
    bool IsValidDate(int day, int month, long year);
    bool IsLeapYear( long year );
};
```

3. Trong code editor, thêm mã trong Listing 5.2 vào file mã nguồn cho ứng dụng. Hoặc, bạn có thể tạo một file mới có tên là date.cpp để lưu trữ riêng biệt tất cả thông tin này.

Đây là các phương thức không inline (nội dòng) cho lớp. Bạn có thể đặt chúng trong cùng một file với mã nguồn gốc hoặc tạo một file nguồn mới rồi thêm chúng vào nó.

Listing 5.2: Các phương thức không inline

```
bool cDate::IsValidDate( int day, int month, long year )
{
    // Is the month valid?
    if ( month < 0 || month > 11 )
        return false;
```

```
    // Is the year valid?
    if ( year < 0 || year > 9999 )
        return false;
    // Is the number of days valid for this month/year?
    if ( day < 0 || day > MonthDays(month, year) )
        return false;
    // Must be ok
    return true;
}
bool cDate::IsLeapYear( long year )
{
    int year_end = year % 100; // year in century
    if (((year%4) == 0) && ((year_end !=0) || ((year%400) == 0)))
        return true;
    return false;
}
```

Việc đặt mã này vào một đối tượng và chia sẻ mã đó giữa các project khác nhau vốn có thể cần chức năng này sẽ mang lại một số lợi ích rõ rệt:

- Ví dụ, nếu mã cần thay đổi để xử lý một lỗi nào đó trong phép tính năm nhuần, sự thay đổi đó có thể được hoàn tất ở một nơi.
- Quan trọng hơn, nếu các thay đổi được thực hiện để thực thi một cách mới hơn, nhanh hơn để tính năm nhuần hoặc ngày trong tuần, hoặc thậm chí để bổ sung chức năng, những thay đổi đó sẽ không ảnh hưởng đến các chương trình gọi. Chúng sẽ vẫn làm việc với giao diện như bây giờ.

Test lớp cDate

Sau khi bạn tạo một lớp, điều quan trọng là phải tạo một driver thử nghiệm - điều này không chỉ bảo đảm rằng mã chính xác mà nó còn hướng dẫn người khác cách sử dụng mã của bạn.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch1_5.cpp.

2. Gõ nhập mã từ Listing 5.3 vào file.

Listing 5.3: Chương trình thử nghiệm lớp cDate

```
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
    // Do some testing. First, a valid date
    cDate d1(31, 11, 2004);
    // Now, an invalid one.
    cDate d2(31, 12, 2004);
    // Finally, let's just create a blank one.
    cDate d3;
    // Print them out
    cout << "D1:" << "Month:" << d1.Month() << " Day:" << d1.Day() << "
Year:" << d1.Year()
    << endl;
    cout << "D2:" << "Month:" << d2.Month() << " Day:" << d2.Day() << "
Year:" << d2.Year()
    << endl;
    cout << "D3:" << "Month:" << d3.Month() << " Day:" << d3.Day() << "
Year:" << d3.Year()
    << endl;
    return 0;
}
```

3. Lưu mã nguồn dưới dạng một file trong code editor và đóng ứng dụng editor.
4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy chương trình trên console hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây từ chương trình trên cửa sổ console:

```
$ ./a.exe
D1: Month: 11 Day: 31 Year: 2004
D2: Month: 2011708128 Day: -1 Year:
    2011671585
D3: Month: 8 Day: 7 Year: 2004
```

Đây là lợi ích khi làm việc với lập trình hướng đối tượng trong C++: Bạn có thể thực hiện những thay đổi "trong hậu cảnh" mà không can thiệp vào công việc của những người khác. Bạn làm cho người ta có thể truy cập dữ liệu và các thuật toán mà không cần phải vật lộn với cách chúng được lưu trữ hoặc được thực thi như thế nào. Sau cùng, bạn có thể sửa chữa hoặc mở rộng những phần thực thi của các thuật toán mà không đòi hỏi những người dùng của bạn thay đổi tất cả những ứng dụng nào đang sử dụng các thuật toán đó.

Phần II
Làm việc với bộ tiền xử lý

Kỹ thuật 6: Xử lý nhiều hệ điều hành

Tiết kiệm thời gian bằng cách

- Định nghĩa một giải pháp đáp ứng nhiều hệ điều hành
- Tạo file header
- Test file header

Vấn đề với các file header C++ "chuẩn" là chúng là bất cứ thứ gì ngoại trừ chuẩn. Ví dụ, trên Microsoft Windows file header để chứa tất cả hàm xuất "chuẩn" là stdio.h - trong khi trên Unix, file header là unistd.h. Hãy tưởng tượng bạn biên dịch một chương trình vốn có thể sử dụng trên Unix hoặc Microsoft Windows. Mã trong tất cả file của bạn có thể trông như sau:

```
#ifdef WIN32
#include <stdio.h>
#else
#ifdef UNIX
#include <unistd.h>
#endif
#endif
```

Phương pháp viết mã này bề bộn và kém hiệu quả: Nếu bạn nhận được một trình biên dịch mới vốn thực thi các hằng cho hệ điều hành một cách khác biệt, bạn sẽ phải đi qua mỗi và mọi file để cập nhật mã. Hoặc, bạn có thể đơn giản bao hàm tất cả file trong một file header - nhưng điều đó buộc bạn bao hàm nhiều file header mà bạn thật sự không cần trong nhiều file chương trình, điều này sẽ tăng kích cỡ file

và có thể gây ra các vấn đề nếu bạn cần ghi đè một số tên hoặc kiểu hàm chuẩn. Rõ ràng sự bề bộn không phải là một giải pháp tốt cho lắm trong một trong hai cách.

Điều gì sẽ xảy ra nếu - thay vì bao hàm những thứ mà bạn không muốn và phải biên dịch có điều kiện xung quanh chúng - bạn bao hàm chỉ các file "thích hợp" cho một hệ điều hành cụ thể lúc ban đầu? Giải pháp đó chắc chắn gần như lý tưởng. Thật may thay, bộ tiền xử lý (pre-processor C++) cung cấp một cách hoàn hảo để giải quyết vấn đề này. Hãy tiếp tục đọc.

Tạo file header

Để có thể bao hàm một cách có điều kiện các phần mã mà chúng ta muốn sử dụng trong ứng dụng, chúng ta sẽ tạo một file header vốn tận dụng các giá trị được định nghĩa bởi bộ tiền biên dịch (pre-complier) để quyết định các file nào cần thiết. Các bước sau đây hướng dẫn bạn cách tạo một file như vậy:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là osdefines.h mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa thông tin header.

2. Gõ nhập mã từ Listing 6.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 6.1: File header

```
#ifndef _osdefines_h_
#define _osdefines_h_
// Remove the comment from the WIN32 define
    if you are
// developing on the Microsoft Windows plat
    form.Remove
// the comment on the UNIX define if you are
    developing
// on the UNIX platform
#define WIN32
// #define UNIX
// Now, define the header files for the
    Windows platform
```

```
#ifdef WIN32
#define standard_io_header <stdio.h>
#endif
#ifdef UNIX
#define standard_io_header <unistd.h>
#endif
// Make sure SOMETHING is defined
#ifndef standard_io_header
#error "You must define either WIN32 or
    UNIX"
#endif
#endif // _osdefines_h
```

3. Lưu mã nguồn dưới dạng một file trong code editor và đóng ứng dụng code editor.

Test file header

Sau khi tạo lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn.

Sau đây là cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và xem lớp được ấn định để sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch06.cpp.

2. Gõ nhập mã từ Listing 6.2 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Windows site đi kèm của sách này.

Listing 6.2: Chương trình chính

```
#include "osdefines.h"
#include standard_io_header
#define MAX_VALUES 100
#define STRING(A) #A
#define PASTE(A,B) (A##B)
#define MAKE_SAFE(s) (s==NULL? "NULL" : s )
```

```
int main(int argc. char **argv)
{
    int x = 100;
    // We can stringify a variable name
    printf("The value of %s is %d\n",
    STRING(x), x );
    int y = 200;
    int xy = 0;
    // We can use a macro to create a new
       variable.
    PASTE(x,y) = x*y;
    printf("The value of x = %d\n", x );
    printf("The value of y = %d\n", y );
    // The problem is that we can't
       stringify pastes.
    printf("The value of %s = %d\n",
    STRING(PASTE(x,y)), xy );
    char *s1 = NULL;
    char *s2 = "Something";
    printf("String1 = %s\n", MAKE_SAFE(s1));
    printf("String2 = %s\n", MAKE_SAFE(s2));
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng ứng dụng code editor.
4. Biên dịch file bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Để kiểm chứng rằng file header sẽ không làm việc trừ phi bạn định nghĩa hệ điều hành, chú giải các dòng WIN 32 và Unix trong file osdefines.h. Hãy thử biên dịch nó và bạn sẽ thấy một thông báo lỗi như sau:

```
$ gcc test.cpp
In file included from test.cpp:2:
osdefines.h:23:2: #error "You must define
either WIN32 or UNIX"
test.cpp:3:10: #include expects "FILENAME"
or <FILENAME>
```

Như bạn có thể thấy trình biên dịch rõ ràng biết rằng hệ điều hành không được định nghĩa. Bước kế tiếp là định nghĩa một trong hai hằng, phụ thuộc vào hệ điều hành mà bạn chọn. Có hai cách khác nhau để định nghĩa các hằng này. Bạn có thể đặt một câu lệnh #define ở đầu file header hoặc bạn có thể chuyển giá trị vào trình biên dịch với cờ biên dịch - D. Việc tái biên dịch chương trình sau hoạt động này sẽ không dẫn đến các lỗi - và nếu là như vậy, bạn biết file header thích hợp bây giờ được đưa vào.

Kỹ thuật số 7: Nắm vững các vấn đề của Assert

Tiết kiệm thời gian bằng cách

- Định nghĩa các vấn đề mà các assert có thể gây ra
- Biên dịch với các assert
- Xử lý các vấn đề assert

Khó nói về bộ tiền xử lý C++ mà không nói về macro assert. Macro đặc biệt này được sử dụng để test một điều kiện nào đó - và nếu điều kiện đó không true một cách lôgíc, để in ra một thông báo lỗi và thoát chương trình.

Đây là nơi bạn có thể quyết đoán với vấn đề kiểm tra để tìm các sự cố, do đó hãy xem qua các assert và tuân theo một kỹ thuật đơn giản để sử dụng chúng.

Vấn đề assert

Mục đích của việc sử dụng assert là kiểm tra để tìm một sự cố vào thời gian chạy. Các câu lệnh assert có giá trị trong quá trình gỡ rối và hiệu lực hoá mã ban đầu, nhưng chúng có giá trị giới hạn một khi chương trình thoát. Vì lý do này, bạn nên đưa vào đủ các câu lệnh assert để bảo đảm việc kiểm tra hệ thống sẽ làm lộ ra tất cả sự cố tiềm ẩn mà bạn nên kiểm tra và xử lý vào thời gian chạy. Hãy xem một ví dụ đơn giản về việc sử dụng một lệnh gọi assert trong chương trình.

1. Trong code editor mà bạn chọn lựa, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch07.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã nguồn cho ví dụ này.

2. Gõ nhập mã trong Listing 7.1 vào file, thay thế các tên riêng của bạn cho các hằng, biến và tên file in nghiêng.

Listing 7.1: Sử dụng assert

```
#include "stdio.h"
#include "assert.h"
int main(int argc, char **argv)
{
    assert( argc > 1 );
    printf("Argument 1 = %s\n", argv[1] );
    return 0;
}
```

3. Lưu file mã nguồn và đóng code editor

4. Biên dịch file nguồn, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

   Nếu bạn chạy chương trình này không có các đối số, bạn sẽ thấy rằng nó thoát một cách bất thường và hiển thị thông báo lỗi sau đây:

```
$ ./a.exe
assertion "argc > 1" failed: file
    "ch07a.cpp", line 6
Aborted (core dumped)
```

   Như bạn có thể thấy macro assert đã được kích khởi đúng cách và đã thoát chương trình, đây là hành vi mong đợi của hàm khi nó thất bại. Dĩ nhiên, đây không hẳn là điều bạn thường muốn chương trình thực hiện khi bạn không nhập một giá trị, nhưng nó minh hoạ nguồn gốc gây ra lỗi.

5. Biên dịch lại file nguồn bằng trình biên dịch ưa thích, sử dụng định nghĩa NDEBUG trên dòng lệnh.

   Không đơn giản là việc sử dụng một assert để thoát một chương trình thì xấu. Nhưng phần tệ nhất là nhiều môi trường biên dịch chỉ định nghĩa macro assert khi chương trình được biên dịch trong chế độ debugging (gỡ rối). Thật ra, macro assert chuyển vào trạng thái không hoạt động (off) khi chương trình được biên dịch để chạy tối ưu. Với trình biên dịch gcc, bạn tối ưu hoá mọi thứ bằng cách biên dịch bằng khoá chuyển đổi biên dịch -DNDEBUG. Nếu bạn biên dịch chương trình được cho ở đây với tập hợp khoá chuyển đổi đó, bạn có được kết quả rất khác biệt:

```
$ ./a.exe
Argument 1 = (null)
```

Ở trên là kết quả khi bạn chạy chương trình sau khi biên dịch với cờ -DNDEBUG cho trình biên dịch. Như bạn có thể thấy nó rất khác với trường hợp mà macro assert được bật.

Chú ý rằng không có đối số được cung cấp cho chương trình, do đó chúng ta thật sự bước qua bộ nhớ vào lúc này. Vì mảng pointer chứa dẫy các đối số dẫn đến ứng dụng, chúng ta giới hạn chỉ trong số đối số được chuyển vào. Nếu không có gì được chuyển vào chương trình, sẽ không có gì trong mảng đối số và các pointer trong mảng argv sẽ trỏ vào thông tin thừa. Thật may thay, chúng ta đã không cố làm bất cứ điều gì với pointer ngoại trừ in nó ra, nhưng có lẽ nó có thể dễ dàng khiến cho một chương trình tự đổ vỡ.

Hãy tưởng tượng nếu mã này đã được đưa vào một hệ thống sản xuất. Lần đầu tiên một build tối ưu hoá (thường được gọi là một "release") đã được tạo, chương trình sẽ bị đổ vỡ ngay khi người dùng chạy nó mà không cho chương trình bất kỳ đối số trên dòng lệnh. Rõ ràng, đây không phải là một giải pháp tối ưu khi bạn làm việc trong thực tế. Phần tiếp theo sẽ hướng dẫn bạn cách giải quyết vấn đề này.

Xử lý vấn đề Assert

Các macro Assert có giá trị - đặc biệt khi bạn theo dõi những vấn đề đặc biệt gây phiền phức. Bằng cách làm cho mã chứa đầy các assert, bạn có thể theo dõi những điều kiện mà bạn đã không mong đợi. Tuy nhiên, những assert đó sẽ không cho bạn đơn giản bỏ qua những điều kiện phiền phức mà bạn tìm thấy. Để giải quyết vấn đề này, phần liên quan của Listing 7.1 sẽ được viết lại. Bước sau đây hướng dẫn bạn cách thực hiện.

1. Chỉnh sửa mã nguồn cho ứng dụng thử nghiệm như trong Listing 7.2.

   Trong trường hợp này, file mã nguồn gốc được gọi là ch07.cpp.

Listing 7.2 Giải quyết vấn đề Asserts

```
#include "stdio.h"
#include "assert.h"
int main(int argc, char **argv)
{
    assert( argc > 1 );
    if ( argc > 1 )
        printf("Argument 1 = %s\n", argv[1]
```

```
    ;
    return 0;
}
```

Sự khác biệt ở đây là gì? Rõ ràng nếu bạn biên dịch chương trình trong chế độ debug (nghĩa là không được tối ưu hoá) và chạy nó không có các đối số, assert được kích khởi và chương trình thoát, đưa ra một thông báo lỗi như trước đó. Tuy nhiên, nếu bạn biên dịch trong chế độ optimized (tối ưu hoá), assert được bỏ qua và chương trình kiểm tra xem có đủ các đối số để xử lý hay không. Nếu không, câu lệnh gây lỗi vốn có thể làm cho chương trình đổ vỡ được bỏ qua. Khó và hơi buồn khi biết rằng bao nhiêu chương trình đã được đưa vào thế giới (hơn khoảng 20 năm qua) chứa các hàm như sau:

```
int func( char *s)
{
assert(s != NULL);
strcpy( myBuffer, s);
}
```

Hàm này có mục đích sao chép một chuỗi đầu vào vào một bộ đệm (buffer) vốn được cung cấp bởi nhà lập trình. Bộ đệm đó có một kích cỡ nhất định, nhưng chúng ta không kiểm tra tìm số ký tự tối đa có thể cho phép trong chuỗi đầu vào. Nếu số ký tự đi vào lớn hơn số ký tự trong mảng myBuffer, nó sẽ gây ra các vấn đề.

Như bạn có thể tưởng tượng, điều này gây ra nhiều vấn đề trong thực tế bởi vì bộ nhớ vốn không thuộc về ứng dụng được sử dụng và được gán các giá trị. Các Asserts rất hữu dụng cho việc định nghĩa các trường hợp test, bẫy các lỗi ngoại lệ mà bạn biết có thể xảy ra và tìm những vấn sự cố mà bạn thật sự không mong đợi xảy ra. Điều thú vị nhất về các assert là sau khi bạn tìm thấy sự cố mà nó xác định, bạn thường có thể sử dụng trình gỡ rối (debugger) để biết chính xác điều gì đã gây ra sự cố - và phương pháp tốt nhất là sử dụng cơ chế stack-trace. Kỹ thuật 62 sẽ hướng dẫn bạn cách đưa một cơ chế như vậy vào ứng dụng để bạn có thể tìm thấy các sự cố như sự cố này trong thời gian chạy.

Kỹ thuật 8: Sử dụng const thay vì #define

Tiết kiệm thời gian bằng cách

- So sánh các câu lệnh #define với các câu lệnh const
- Sử dụng câu lệnh const

- Tìm hiểu các lỗi do câu lệnh #define gây ra
- Giải quyết các lỗi

Qua suốt sách này, chúng ta sử dụng câu lệnh #define để tạo các macro và giá trị hằng để sử dụng trong các chương trình. Nó là một phương pháp hữu dụng - Tuy nhiên, nhóm tiêu chuẩn C++ đã chọn tạo một cách mới để định nghĩa các hằng trong ứng dụng: câu lệnh const. Câu lệnh const được sử dụng theo các sau đây:

```
const int MaxValues = 100;
```

Nếu câu lệnh này trông quen thuộc, nó giống nhiều như cách câu lệnh #define được sử dụng:

```
#define MAX_VALUES 100
```

Sự khác biệt là cấu trúc const được định nghĩa tại cấp trình biên dịch thay vì tại cấp bộ tiền xử lý. Với const, trình biên dịch có thể tối ưu hoá các giá trị tốt hơn và có thể thực hiện việc kiểm tra an toàn kiểu (type-safe).

Sau đây là một ví dụ. Trước tiên, đây là cách phương thức #define làm việc. Giả sử chúng ta viết một định nghĩa như sau:

```
#define NoValues 0
```

Và sau đó viết một câu lệnh C++ như sau:

```
char *sValues = NoValues;
```

Câu lệnh này sẽ biên dịch (mặc dù một số trình biên dịch có thể đưa ra một cảnh báo về một sự chuyển đổi không an toàn) bởi vì phần khai báo NoValues bằng với một giá trị chuỗi là NULL. Cho đến giờ rất tốt - nhưng hãy giả sử bây giờ chúng ta thay đổi giá trị đó bằng cách định nghĩa câu lệnh sau đây (chú ý rằng bất kỳ giá trị không rồng minh hoạ vấn đề này theo cùng một cách):

```
#define NoValues -99
```

Hành vi của phép gán sValues không thể dự đoán trước được. Một số trình biên dịch sẽ cho phép nó, gán một ký tự rất lạ (bất kỳ -99 nào trong tập hợp ký tự mà bạn đang sử dụng) vào chuỗi. Những trình biên dịch khác sẽ không cho phép nó và sẽ than phiền, đưa ra các lỗi khác lạ để hiểu và sửa chữa. Theo một trong hai cách, kết quả gây khó chịu.

Bây giờ đối với phương thức const. Nếu bạn viết

```
const char *NoValues = -99
```

thì bạn sẽ thấy ngay trình biên dịch phản ứng (bằng cách tạo ra một lỗi biên dịch) lúc bạn định nghĩa hằng như thế nào. Cấu trúc const an toàn kiểu (type-safe) - bạn cho nó một kiểu và bạn gán nó chỉ vào những thứ cùng kiểu hoặc có các kiểu tương thích; nó sẽ không chấp nhận bất cứ thứ gì khác, do đó tính nhất quán của nó không bị phá vỡ.

Sử dụng cấu trúc const

Chuẩn C++ cung cấp một phương thức để xử lý các sự cố gây ra bởi câu lệnh #define trong bộ tiền xử lý. Câu lệnh này là câu lệnh const được xử lý bởi trình biên dịch, không phải bộ tiền xử lý và do đó làm cho mã dễ hiểu và dễ gỡ rối hơn.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỳ thuật.

   Trong ví dụ này file được đặt tên là ch08.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã trong Listing 8.1 vào file.

Chú ý trong listing này, bạn có thể thấy hiệu ứng của phiên bản #define của câu lệnh và phiên bản const của câu lệnh. Trình biên dịch sẽ hiểu chúng một cách khác nhau như chúng ta sẽ thấy ngay sau đây.

Listing 8.1: Sử dụng các hằng

```
#include <stdio.h>
const int MaxValues = 100;
#define MAX_VALUES 100;                          → 1
int main(int argc, char **argv)
{
    int myArray[ MaxValues ];
    int myOtherArray[ MAX_VALUES ];
    for ( int i=0; i<MaxValues; ++i )
        myArray[i] = i;
    for ( int i=0; i<MAX_VALUES; ++i )
        myOtherArray[i] = i;
    return 0;
}
```

3. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Biên dịch chương trình này trông bình thường này, bạn sẽ nhận được các thông báo lỗi sau đây.

```
$ gcc ch08.cpp
ch08.cpp: In function 'int main(int,
    char**)':
ch08.cpp:9: error: syntax error before ';'
    token
ch08.cpp:13: error: syntax error before
    ';' token
ch08.cpp:14: error: 'myOtherArray' unde
    clared (first use this function)
ch08.cpp:14: error: (Each undeclared iden
    tifier is reported only once for each
      function it appears in.)
```

Phần kế tiếp mô tả cách sửa chữa những lỗi này.

Nhận dạng các lỗi

Xem các dòng mà các lỗi xuất hiện trên đó, hoàn toàn không biết rõ sự cố có thể là gì. Tham chiếu dòng đầu tiên được đánh dấu bằng ký hiệu → **1**.

Chắc chắn dòng này trông giống như một dòng mã hợp lệ - sự cố có thể là gì? Câu trả lời là không phải trình biên dịch mà là bộ tiền xử lý. Hãy nhớ bộ tiền xử lý lấy mọi thứ theo sau token mà bạn định nghĩa (trên dòng #define) và trung thành dán nó vào mã bất cứ nơi nào nó tìm thấy token sau đó. Trong trường hợp này sau khi việc dán (paste) xảy ra, bản thân dòng này được chuyển đổi thành

```
int myOtherArray[ 100; ];
```

Chú ý dấu chấm phẩy dư thừa ở giữa định nghĩa mảng. Điều đó không hợp lệ trong C++ và sẽ gây ra các lỗi. Nhưng thay vì trỏ vào dòng gây lỗi "thật sự" vốn là #define có một dấu chấm phẩy ở cuối, trình biên dịch đưa ra một lỗi gây bối rối về một dòng mà nó trông có vẻ tốt.

Định nghĩa #define có thể gây ra các lỗi, nhưng định nghĩa const của MaxValues không có một vấn đề như vậy. Những gì nó cung cấp đơn giản là một định nghĩa của một giá trị - và sau đó bạn có thể sử dụng giá trị đó bất cứ nơi nào trong chương trình mà một giá trị trực kiện hoặc hằng #define có thể được sử dụng.

Sửa chữa các lỗi

Làm thế nào chúng ta sửa chữa những sự cố đó trong trình biên dịch sao cho mã thực hiện những gì mà chúng ta muốn? Hãy xem một số cách chúng ta có thể làm cho mã biên dịch và thực hiện những gì chúng ta đã dự định thay vì để trình biên dịch đoán không đúng những gì chúng ta muốn.

1. Mở lại file nguồn trong code editor ưa thích.
2. Sau khi file mở ra, chỉnh sửa chương trình hiện có để sửa các lỗi biên dịch như sau.

   Chú ý mã này thay thế listing mã trước, nó không được thêm vào listing mã đó.

```
int main(int argc, char **argv)
{
    int xVal = 10;
    int myArray[ xVal ];
    for ( int i=0; i<xVal; ++i )
       myArray[i] = i;
    return 0;
}
```

Có một mối nguy hiểm với việc sử dụng phương pháp này. Xem xét dòng sau đây:

```
int xVal;

int myArray[ xVal ];
```

Hãy luôn khởi tạo tất cả biến trong mã thậm chí nếu bạn không nghĩ bạn sẽ cần đến chúng.

Trong phiên bản không được tối ưu hoá của mã, xVal được gán một giá trị là 0 - vốn cho phép bạn tạo một mảng 0 phần tử. Điều trục trặc bắt đầu khi bạn chạy phiên bản được tối ưu hoá: giá trị xVal không xác định và mã này có thể sẽ khiến cho một chương trình đổ vỡ. Cố đừng làm những điều như vậy. Cách tốt nhất để giải quyết mọi thứ như vậy là xác lập mức cảnh báo trình biên dịch sang mức cao nhất để phát hiện các biến không khởi tạo vốn được sử dụng.

Kỹ thuật 9: Các macro và tại sao không sử dụng chúng

Tiết kiệm thời gian bằng cách:

- Tìm hiểu những khuyết điểm của các macro
- Sử dụng các hàm thay vì các macro
- Tránh các vấn đề với các macro chuỗi
- Xác định các lỗi khi sử dụng các macro
- Sử dụng các macro một cách thích hợp

Bộ tiền xử lý và các macro là những thứ hữu dụng nhưng không quá nhiệt tình trong công dụng của chúng. Ngoài thẩm hoạ có thể có rõ ràng (bộ tiền xử lý trở nên điên dại và thay thế bất kỳ mã trong ứng dụng bằng bất cứ những gì nằm trong macro), các macro thường có những tác dụng phụ không rõ ràng khi chúng được gọi ra. không giống như các hàm - có những hệ thống phụ có thể được phát hiện trong trình gỡ rối - một macro không có chức năng gỡ rối (debug). Bởi vì bộ tiền xử lý "sao chép" mã của macro vào ứng dụng, trình gỡ rối (debugger) thật sự không hiểu macro hoạt động như thế nào. Và mặc dù các macro cho phép bạn tránh hao phí đẩy dữ liệu lên trên stack (ngăn xếp) và đẩy nó ra ngoài để gọi ra một hàm, macro tăng kích cỡ file bằng cách sao chép cùng một khối mã mỗi lần nó được sử dụng.

Dĩ nhiên, chính những lý do này không đủ để làm cho các nhà lập trình muốn tránh sử dụng các macro. Có những lý do tốt hơn nhiều. Ví dụ, xem xét macro min (thay thế cho "minimum") mà nhiều chương trình định nghĩa như sau:

```
#define mim(X, Y) ((X) < (Y) ? (X) : (Y))
```

Giả sử bạn sử dụng macro min với một đối số vốn thực hiện một điều khác nào đó - như sau -

```
next = min (x + y), func (z));
```

Macro mở rộng như sau:

```
next = ((x + y) < (func (z)) ? (x + y) : (func (z));)
```

Trong đó x + y thay thế x và func (z) thay thế y.

Bây giờ điều này có thể dường như không giống như một điều tệ hại. Nhưng nếu hàm func có một tác dụng phụ nào đó xảy ra khi nó được gọi nhiều lần thì phải làm sao? Tác dụng phụ sẽ không nhìn rõ ngay tức thì nếu đọc mã vốn cho thấy hàm được gọi hai lần. Nhưng

một nhà lập trình gỡ rối chương trình của bạn có thể bị bối rối nếu (ví dụ) một hàm func xuất hiện trông giống như vậy bởi vì nó có nghĩa là giá trị đầu vào đã được thay đổi không phải một lần như nó xuất hiện mà là hai lần:

```
int func(int &x)
{
    x *= 2;
    return x;
}
```

Rõ ràng, hàm này chấp nhận một đối số số nguyên đơn theo tham chiếu. Sau đó nó nhân đối số đó với hai và trả về nó. Vấn đề ở đây là gì? Bởi vì đối số được chuyển theo tham chiếu, đối số gốc được thay đổi. Kết quả này có thể là những gì đã được ấn định, nhưng nó cũng có thể gây ra các vấn đề như trong trường hợp của macro min. Thay vì làm cho hàm trả về hai lần giá trị và so sánh nó, chúng ta thật sự xem nó so với bốn lần đối số. Từ phiên bản mở rộng của macro, bạn có thể thấy rằng z sẽ được chuyển vào hàm hai lần và vì hàm lấy đối số của nó theo tham chiếu, nó sẽ được chỉnh sửa trong chương trình chính. Đây rất có thể không phải là những gì mà nhà lập trình đã muốn lúc đầu.

Khởi tạo một hàm với một macro chuỗi

Các vấn đề trở nên tinh vi hơn khi bạn cấp phát và sao chép các chuỗi (strings) - mà các nhà lập trình luôn thực hiện trong C++. Sau đây là tình huống thông thường: bạn có một chuỗi đầu vào, bạn muốn sao chép nó và lưu trữ kết quả trong một chuỗi khác. Một hàm thư viện được gọi là strdup thực hiện chính xác điều này. Nhưng giả sử bạn muốn sao chép chỉ một số byte nhất định của chuỗi gốc vào chuỗi mới. Bạn không thể sử dụng strdup bởi vì nó luôn nhân đôi chuỗi hoàn toàn. Hơn nữa giả sử để bảo toàn bộ nhớ, bạn muốn loại bỏ chuỗi gốc sau khi sao chép. Điều này có thể được thực hiện có lẽ để rút gọn một chuỗi hoặc để bảo đảm một thứ gì đó luôn nằm vừa trong một trường cơ sở dữ liệu cụ thể. Các bước sau đây hướng dẫn bạn cách tạo mã để thực hiện tác vụ tiện lợi này - thực thi nó dưới dạng một macro và sau đó dưới dạng một hàm để xem các vấn đề với mỗi đối tượng này có thể là gì.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này file được đặt tên là ch09.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã trong Listing 9.1 vào file.

Listing 9.1: File macro

```
#include <stdio.h>
#include <string.h>
// This will be our macro version
#define COPY_AND_TRUNC(ns, s) \
    if ( strlen(s) > 20 ) \
    { \
        ns = new char[ 20 ]; \
        memset( ns, 0, 20 ); \
        strncpy( ns, s, 20-1 ); \
    } \
    else \
    { \
        ns = new char[ strlen(s) ]; \
        memset( ns, 0, strlen(s) ); \
        strcpy( ns, s ); \
    } \
    delete s;
int main(int argc, char **argv )
{
    char *s = new char[80];
    strcpy( s, "This is a really long string
    to test something");
    char *ns = NULL;
    COPY_AND_TRUNC( ns, s );
printf("New string: [%s]\n", ns );
    char *s2 = new char[80];
    strcpy( s2, "This is a really long
    string to test something");
    COPY_AND_TRUNC( s2, s2 );
    printf("New string: [%s]\n", ns );
        return 0;
}
```

Chú ý bạn có thể tạo một macro đa dòng (multiple line) bằng cách sử dụng ký tự dấu gạch chéo ngược ("\") ở cuối dòng trước. Điều này sẽ mở rộng macro cho đến khi nó hầu như là một hàm hoàn chỉnh.

3. Biên dịch chương trình bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
4. Chạy chương trình trên hệ điều hành ưa thích.

   Nếu bạn đã làm tốt mọi thứ thì bạn sẽ thấy kết quả sau đây:

```
$ ./a.exe
New string: [This is a really lo]
New string: [(null)]
```

Sửa chữa những sự cố nào xảy ra với macro

Điều gì đã xảy ra ở đây? Kết quả của lệnh gọi hàm cuối cùng có lẽ nên giống với lệnh gọi hàm đầu tiên. Đây là một vấn đề nghiêm trọng được truy nguyên là tác dụng phụ của macro. Bởi vì thủ tục đã không kiểm tra xem đầu vào và đầu ra có như nhau hay không, bạn không thể an toàn xoá bộ đệm character-pointer mà bạn đã không cấp phát. Tuy nhiên, bằng cách làm theo các bước được trình bày ở đây, bạn có thể viết lại macro này dưới dạng một hàm tương đương nhưng an toàn hơn.

1. Mở lại file nguồn trong code editor.
2. Thực hiện các thay đổi đối với mã nguồn như được minh hoạ trong Listing 9.2. Chú ý rằng các dòng cần chỉnh sửa được minh hoạ tại →1 và →2. Các khối mã được minh hoạ ở đây sẽ được bổ sung.

Listing 9.2: File macro cập nhật

```
#include <stdio.h>
#include <string.h>
// This will be our macro version
#define COPY_AND_TRUNC(ns, s) \
    if ( strlen(s) > 20 ) \
    { \
       ns = new char[ 20 ]; \
       memset( ns, 0, 20 ); \
       strncpy( ns, s, 20-1 ); \
       } \
    else \
```

```
        { \
            ns = new char[ strlen(s) ]; \
            memset( ns, 0, strlen(s) ); \
            strcpy( ns, s ); \
        } \
        delete s; \
        s = NULL;
char *copy_and_truncate( char *& s )                    → 1
        {
        char *temp = NULL;
        if ( strlen(s) > 20 )
        {
            temp = new char[ 20 ];
            memset( temp, 0, 20 );
            strncpy( temp, s, 20-1 );
        }
        else
        {
            temp = new char[ strlen(s) ];
            memset( temp, 0, strlen(s) );
            strcpy( temp, s );
        }
        delete s;
        s = NULL;
        return temp;
}
int main(int argc, char **argv )
{
        char *s = new char[80];
        strcpy( s, "This is a really long string
        to test something");
        char *ns = NULL;
        COPY_AND_TRUNC( ns, s );
        printf("New string: [%s]\n", ns );
        char *s2 = new char[80];
```

```
    strcpy( s2, "This is a really long
    string to test something");
    COPY_AND_TRUNC( s2, s2 );
    printf("New string: [%s]\n", s2 );
    char *s3 = new char[80];
    strcpy( s3, "This is a really long
    string to test something");
    s3 = copy_and_truncate( s3 );                    → 2
    printf("New string: [%s]\n", s3 );
}
```

3. Lưu mã nguồn trong bộ soạn thảo mã nguồn và chọn ứng dụng soạn thảo mã nguồn.
4. Biên dịch chương trình bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

   Nếu bạn đã làm tốt mọi thứ, lần này bạn sẽ thấy kết quả sau đây trong cửa sổ console:

```
$ ./a.exe
New string: [This is a really lo]
New string: [(null)]
New string: [This is a really lo]
```

Chú ý rằng lần này hàm đã làm chính xác những gì bạn mong đợi. Bạn không chỉ không xoá pointer mà bạn còn không gây ra rò rỉ bộ nhớ mà phiên bản trước đã gây ra. Hãy tưởng tượng bạn phải can thiệp vào các macro như vậy lặp đi lặp lại nhiều lần để hoàn thành công việc của bạn. Để tránh sự phiền phức đó, bạn nên chọn các hàm cho bất cứ thứ gì ngoại trừ các macro đơn giản nhất. Hàm được trình bày trong mã được chỉnh sửa không gây ra các vấn đề trong khi macro trong listing ban đầu thì có. Điều này minh hoạ những vấn đề được gây ra vô ý bởi các macro.

Sử dụng các macro một cách thích hợp

Vậy thì các macro có lợi cho mục đích gì? Hãy nhớ một macro chỉ là đối tượng cú pháp thú vị. Nó dễ dàng với quá nhiều điều tốt. Sử dụng một đống macro có thể làm cho việc đọc mã của bạn dễ dàng hơn, nhưng bạn không bao giờ nên chỉnh sửa bản thân mã với một macro. Ví dụ, nếu bạn có một biểu thức đặc biệt phức tạp chẳng hạn như (*iter).C_Str() - vốn có thể xảy ra khi bạn sử dụng Standard Template Library (STL) trong C++, bạn có thể tạo một macro như sau:

```
#define PTR (x) (*x)
```

Sau đó, bạn có thể viết PTR (x).C_Str( ) và thường khi bạn viết nó, định nghĩa sẽ nhất quán. Đây không phải là một ví dụ phức tạp, nhưng nó cho bạn biết khi nào sử dụng và không nên sử dụng các macro trong những ứng dụng C++. Macro đơn giản, không có các tác dụng phụ và làm cho mã dễ đọc hơn sau này. Đây là tất cả lý do tốt để sử dụng một macro.

Kỹ thuật 10: Tìm hiểu sizeof

Tiết kiệm thời gian bằng cách

- Sử dụng hàm sizeof
- Khảo sát và tìm hiểu những kích cỡ byte của các kiểu khác nhau
- Sử dụng sizeof với các pointer

Toán tử sizeof về mặt kỹ thuật không phải là một phần của bộ tiền xử lý, nhưng nó nên được xem là một. Như với tên gọi, toán tử sizeof trả về kích cỡ tính bằng byte của một mẫu thông tin nào đó trong ứng dụng. Nó có thể được sử dụng trên các kiểu cơ bản chẳng hạn như int, long, float, double, hoặc char * và cũng trên các đối tượng, lớp và khối được cấp phát. Thực tế mọi thứ vốn là một kiểu hợp lệ có thể được chuyển đến hàm sizeof.

Hàm sizeof cực kỳ hữu dụng. Ví dụ, nếu bạn muốn cấp phát một khối bộ nhớ để chứa chính xác một kiểu dữ liệu riêng biệt, bạn có thể sử dụng hàm sizeof để quyết định bạn cần bao nhiêu byte để cấp phát như sau:

```
int bytes = sizeof(block);
char *newBlock = new char[bytes];
memcpy( newBlock, block, bytes );
```

Tính năng này cũng hữu dụng khi bạn lưu một đối tượng vào bộ nhớ trong khi thực thi một chức năng undo toàn cục. Bạn lưu trạng thái của đối tượng mỗi lần nó sẽ thay đổi và sau đó trả nó về trạng thái được lưu đó bằng cách đơn giản sao chép khối mã đè lên nó. Dĩ nhiên, có những cách khác để làm tác vụ này, nhưng tác vụ này đơn giản và rất mở rộng.

Các phần tiếp theo sẽ cho bạn thấy những gì sizeof có thể và không thể thực hiện.

Sử dụng hàm sizeof

Hàm sizeof có thể có giá trị trong việc xác định những cấu hình hệ thống, kích cỡ của các lớp và minh hoạ nhiều chi tiết bên trong của hệ thống C++. Các bước sau đây hướng dẫn bạn cách sử dụng hàm sizeof và nó có thể cho bạn biết một điều gì đó về những chi tiết bên trong của mã của chính bạn như thế nào:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch10.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 10.1 vào file.

Listing 10.1: Chương trình sizeof

```
#include <stdio.h>
#include <string>
class Foo
{
public:
    Foo() {};
    ~Foo() {};
};
class Bar
{
public:
    Bar() {};
    virtual ~Bar() {};
};
class Full
{
    int x;
    double y;
public:
    Full()
    {
    }
    virtual ~Full()
```

```
        {
        }
};
class Derived : public Foo
{
public:
        Derived() {};
        ~Derived() {};
};
int main()
{
        int x = 0;
        long y = 0;
        float z = 0;
        double d = 0.0;
        std::string s = "hello";
        // Basic types
        printf("size of char: %d\n",
        sizeof(char));
        printf("size of char *: %d\n",
        sizeof(char *));
        printf("size of int: %d\n", sizeof(x));
        printf("size of long: %d\n", sizeof(y));
        printf("size of float: %d\n",
        sizeof(z));
        printf("size of double: %d\n",
        sizeof(d));
        printf("size of string: %d\n", sizeof(s)
        );
        printf("size of Foo: %d\n",
        sizeof(Foo));
        printf("size of Bar: %d\n",
        sizeof(Bar));
        printf("size of Full: %d\n",
        sizeof(Full));
```

```
    printf("size of Derived: %d\n",
    sizeof(Derived));
}
```

3. Lưu mã nguồn dưới dạng một file trong code editor và sau đó đóng ứng dụng editor.
4. Biên dịch chương trình, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy chương trình.

   Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ console.

```
$ ./a.exe
size of char: 1
size of char *: 4
size of int: 4
size of long: 4
size of float: 4
size of double: 8
size of string: 4
size of Foo: 1
size of Bar: 4
size of Full: 16
size of Derived: 1
```

Từ kết quả, bạn có thể thấy rằng số byte mà mỗi phần tử in ra chiếm trong bộ nhớ cho chương trình này. Không có những điều ngạc nhiên thật sự ở đây ngoại trừ kích cỡ của các lớp. Hãy xem những kết quả này có ý nghĩa gì.

Đánh giá kết quả

Có một số kết luận thú vị cần đưa ra từ kết quả này. Ví dụ, một số kết quả không gây ngạc nhiên gì cả (ví dụ, kích cỡ của một trường ký tự là 1 byte), một số điều ngạc nhiên xuất hiện - ví dụ kích cỡ của một pointer ký tự giống như bất kỳ pointer khác vốn hoá ra là kích cỡ của một long. Điều đó có nghĩa số byte tối đa cho phép mà bạn có thể cấp phát sử dụng các pointer chuẩn có giá trị 4 byte - 32 bit (đó là lý do tại sao Microsoft Windows là một hệ điều hành 32 bit. Nhưng bạn đã biết điều đó).

Điều ngạc nhiên kế tiếp ẩn giấu giữa các đối tượng trong danh sách: kích cỡ của một chuỗi được thể hiện dưới dạng 4 byte vốn không

thể đúng - chuỗi mà nó lưu trữ dài hơn thế. Điều đó như thế nào? Câu trả lời là hàm sizeof trả về trực tiếp số byte được cấp phát bởi đối tượng - nghĩa là số byte được chiếm bởi các biến private và public trong đối tượng, với một vài byte cho các hàm ảo (virtual) (chẳng hạn như các hàm ảo Foo và Bar). Chú ý rằng mặc dù lớp bar không có các biến thành viên, nhưng nó vẫn chiếm 4 byte bởi vì nó cần bảng hàm ảo (hoặc v-table) được thảo luận trong kỹ thuật 2. Bây giờ tại sao lớp Foo chiếm 1 byte khi nó không có các phương thức ảo và không có các biến thành viên? Câu trả lời là hàm sizeof bắt buộc trả về 1 byte cho mọi lớp. Bảo đảm địa chỉ của một đối tượng sẽ không bao giờ giống như địa chỉ của một đối tượng khác. Nếu C++ cho phép các đối tượng có zero kích cỡ, trình biên dịch sẽ không bị phải buộc gán cho những đối tượng đó một địa chỉ mới trong bộ nhớ. Để minh hoạ, nếu chúng ta viết :

```
Foo f1;
Foo f2;
```

trình biên dịch sẽ tự do làm cho cả hai đối tượng này trỏ vào cùng một vị trí trong bộ nhớ. Điều này không được mong muốn thậm chí nếu cả hai đối tượng không được cấp phát bộ nhớ. Có hai đối tượng có cùng một vị trí sẽ phá vỡ quá nhiều hàm thư viện chuẩn. Bất kỳ hàm so sánh nguồn và đích (ví dụ) sẽ bị phá vỡ, thậm chí nếu sự phá vỡ đó không gây hại thật sự. Lý do cho điều này là so sánh việc thực hiện bằng cách xem các địa chỉ mà hai pointer chiếm trong bộ nhớ. Nếu hai địa chỉ giống nhau, điều giả định là những gì chúng trỏ vào giống nhau. Nếu hai đối tượng không có dữ liệu trong đó, nhưng chiếm cùng một vị trí trong bộ nhớ, chúng không giống nhau, thậm chí dường như vậy.

Lớp Bar cũng không chứa các thành viên, nhưng chứa một một hàm ảo, và do đó đẩy số byte được cấp phát sang 4. Cách làm việc đó gợi ý rằng có một điều gì đó mang tính rất vật lý về các hàm ảo và rằng bạn phải chịu một chi phí bộ nhớ để sử dụng tính năng đó.

Lớp Full chứa một số biến thành viên - một double chiếm 8 byte và một số nguyên (integer) chiếm 4 - và nó có 16 byte được cấp phát. 14 byte kia đến từ đâu? Bạn đã doán điều đó: từ bảng ảo (virtual table) vốn được tạo bởi virtual destructor đó. Điều này cho chúng ta biết gì? Thậm chí nếu bạn không có một phương thức ảo "bình thường", việc có một virtual destructor vẫn tạo ra một mục v-table - và điều đó có nghĩa là thêm 4 byte trong phần cấp phát.

Lớp Derived gây bối rối - dường như nó phải ngốn nhiều kích cỡ hơn bình thường. Tuy nhiên, khi bạn xem cẩn thận lớp này, bạn nhận ra rằng nó không chứa hàm ảo và lớp cơ sở mà nó dẫn xuất từ đó

cũng vậy. Do đó, một lần nữa đây là một ví dụ về một lớp trống vốn chiếm 1 byte.

Sử dụng sizeof với các pointer

Việc thảo luận hàm sizeof sẽ không hoàn chỉnh nếu không xem một sai sót mà các nhà lập trình C++ thường phạm phải khi chúng sử dụng hàm này. Xem xét chương trình nhỏ sau đây:

```
#include <stdio.h>
#include <stdlib.h>
const char arr[] = "hello";
const char *cp = arr;
main(){
    printf("Size of array %d\n",
                sizeof(arr));
    printf("Size of pointer %d\n",
                sizeof(cp));
    return(0);
}
```

Bởi vì một câu lệnh xuất kích cỡ của một mảng và câu lệnh khác xuất kích cỡ của một pointer sang mảng đó, bạn nghĩa rằng hai câu lệnh printf trong chương trình nhỏ này sẽ hiển thị các giá trị giống nhau. Nhưng chúng không làm như vậy. Thật ra, nếu bạn xem kết quả, nó trông như sau:

```
Sizeof array 6
Size of pointer 4
```

Kích cỡ của một mảng (array) được biết vào thời gian biên dịch và có thể được hiển thị bởi hàm sizeof. Mặt khác một pointer luôn là kích cỡ của một pointer bất kể nó trỏ vào những gì. Hơn nữa, nếu bạn cố trả về kích cỡ của một mảng bằng cách đưa vào câu lệnh sizeof (*cp) trong đó cp là mảng, bạn sẽ thấy câu trả lời (lần nữa không phải là 6 mà là 1. Tại sao như vậy? Bởi vì biểu thức *cp lượng giá sang một ký tự và kích cỡ của một ký tự đơn luôn là 1 byte. Hãy rất cẩn thận nếu bạn cố sử dụng hàm sizeof trên các pointer - đặc biệt nếu bạn muốn sử dụng kết quả để thể hiện kích cỡ của những gì đang được trỏ vào.

Phần III
Các kiểu dữ liệu

Kỹ thuật 11: Tạo các kiểu cơ bản riêng của bạn

Tiết kiệm thời gian bằng cách

- Loại bỏ mã được sao chép với các kiểu cơ bản tự tạo
- Kiểm tra các dãy (range) trong các giá trị số nguyên
- Test các kiểu cơ bản tự tạo

Trong C++, các kiểu (type) được phân chia thành hai vùng: cơ bản và do người dùng định nghĩa. Các kiểu cơ bản là các kiểu được định nghĩa bởi ngôn ngữ, thường được mô phỏng theo các kiểu được hỗ trợ trực tiếp bởi phần cứng máy tính. Những kiểu này bao gồm các số nguyên, số dấu động và ký tự. Các kiểu nâng cao, chẳng hạn như các chuỗi (string), cấu trúc (structure), và lớp (class) rơi vào vùng do người dùng định nghĩa. Kỹ thuật này sẽ xem xét vùng đầu tiên, kiểu cơ bản. Các kiểu nâng cao sẽ được đề cập trong kỹ thuật kế tiếp.

Bao nhiêu lần bạn đã viết một chương trình đòi hỏi một biến số nguyên cơ bản bị ràng buộc trong một dãy nào đó? Rốt cuộc bạn sao chép cùng một mã lặp đi lặp lại nhiều lần qua suốt ứng dụng, trong các khối như sau:

```
int value = get_a_value();
if ( value < 0 || value > 10 )
{
    printf("invalid input, try again\n");
    return false;
}
```

Dĩ nhiên sau khi bạn đã đưa tất cả khối này vào mã, ông chủ của bạn xuất hiện và cho bạn biết rằng những người ở trong bộ phận kế toán đã quyết định 10 không còn là số ma thuật nữa - bây giờ nó là 12. Do đó bạn chỉnh sửa tất cả mã, học một điều nào đó trong tiến trình - đó là tốt hơn bạn nên sử dụng các hằng trong tình huống này hơn là các biến. Các chỉnh sửa của bạn trông như sau:

```
const int maxValue = 12;
int value = get_a_value();
if ( value < 0 || value > maxValue )
{
    printf("invalid input, try again\n");
    return false;
}
```

Bạn kiểm tra mã trong nơi lưu trữ mã nguồn và đủ chắc chắn ông chủ đi vào lại văn phòng của bạn. Các kế toán đã yêu cầu một thay đổi khác. Trong khi giá trị tối đa có thể cho phép vẫn là 12. Các số 0 (zero) không được cho phép trong hệ thống kế toán. Giá trị nhỏ nhất mà bạn được cho phép nhập là 1. Càu nhàu, bạn viết lại mã một lần nữa (tận dụng những gì bạn đã học trong hai cuộc thử nghiệm đầu tiên) để tạo một thứ gì đó chung (hơn):

```
const int minValue = 1;
const int maxValue = 12;
int value = get_a_value();
if ( value < minValue || value > maxValue
    )
{
    printf("invalid input, try again\n");
    return false;
}
```

Thực thi lớp Range

Kỹ thuật này sẽ hướng dẫn bạn một cách khái quát hơn để giải quyết vấn đề này - bằng cách sử dụng một lớp C++. Ý tưởng là chỉ việc mở rộng kiểu cơ bản int để cho phép các giá trị tối thiểu và tối đa. Dĩ nhiên vấn đề là chúng ta vẫn muốn có khả năng sử dụng các kiểu khác (chẳng hạn như integer) cho các phép so sánh và phép gán và những thứ tương tự. Lớp được tạo trong các bước sau đây xử lý các giá trị tối thiểu và tối đa và giới hạn trong những giá trị đó.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch11.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 11.1 vào file của bạn.

Listing 11.1: Lớp Range

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
class IntRange
{
private:
    int iMin;
    int iMax;
    int iValue;
    virtual void SetValue( int value )
    {
       if ( value < GetMin() )
          value = GetMin();
       else
          if ( value > GetMax() )
             value = GetMax();
       iValue = value;
    }
public:
    IntRange(void)
    {
       iMin = 0;
       iMax = INT_MAX;
       iValue = iMin;
    }
    IntRange(int min, int max)
    {
       if ( min <= max )
       {
```

```
            iMin = min;
            iMax = max;
        }
        else
        {
            iMin = max;
            iMax = min;
        }
        iValue = iMin;
    }
    IntRange( int min, int max, int value )
    {
        if ( min <= max )
        {
            iMin = min;
            iMax = max;
    }
    else
    {
        iMin = max;
        iMax = min;
    }
    SetValue( value );
}
IntRange( const IntRange& aCopy )
{
    iMin = aCopy.iMin;
    iMax = aCopy.iMax;
    iValue = aCopy.iValue;
}
virtual ~IntRange()
{
}
virtual int GetMin(void)
{
```

```
        return iMin;
    }
    virtual int GetMax(void)
    {
        return iMax;
    }
    // Define a few operators
    IntRange& operator=(int value)
    {
        SetValue ( value );
        return *this;
    }
    IntRange& operator=(double value)
    {
        SetValue( (int)value );
        return *this;
    }
    virtual int GetValue(void) const
    {
        return iValue;
    };
```

Nếu bạn kiểm tra mã này bạn sẽ thấy rằng nó kiểm chứng giá trị của một biến số nguyên nằm trong một dãy nhất định. Bạn có thể định nghĩa các giá trị nhỏ nhất và lớn nhất riêng của bạn cho dãy và lớp sẽ bảo đảm rằng bất kỳ giá trị được gán vào biến đó nằm bên trong dãy đó.

Phần thú vị của lớp này là phần cuối cùng gồm các dòng bên dưới chú giải Define a few operators. Đây là nơi cho thấy khả năng mở rộng của C++. Chúng ta đã định nghĩa các toán tử gán để lớp của chúng ta có thể được sử dụng với các kiểu cài sẵn int và double. Rõ ràng, chúng ta có thể bổ sung thêm các kiểu ở đây kể cả các chuỗi và những thứ tương tự, nhưng bây giờ điều này thì đủ. Với khả năng này bây giờ chúng ta có thể sử dụng lớp InRange để lưu trữ dữ liệu, hãy chắc chắn rằng giá trị trong đối tượng sẽ luôn hợp lệ. Đó là nguồn an ủi và nó có nghĩa rằng không còn có bất kỳ lý do nào để viết mã như sau:

```
int x = get_a_value();
if ( x < min || x > max )
    do_some_error();
```

Thay vào đó, chúng ta có thể đơn giản viết

```
IntRange myRangeObj(min, max);
myRangeObj = val;
int x = myRangeObj.GetValue();
```

Chúng ta không cần phải kiểm tra giá trị trả về bởi vì mã đòi hỏi nó chính xác. Tuy nhiên, có một điều khác mà chúng ta có thể thực hiện với lớp này và đó là định nghĩa các toán tử ngoài cho nó. Khả năng định nghĩa một toán tử ngoài cực kỳ có lợi bởi vì nó cho phép những người dùng không truy cập mã nguồn của lớp tạo những cách mới để sử dụng lớp. Đây là điều hoàn toàn duy nhất đối với C++; không có ngôn ngữ nào trước đó có bất cứ điều gì như nó. Nếu không truy cập mã nguồn cho lớp này và chúng ta có thể ghi đè các phép toán cơ bản (chẳng hạn như nhỏ hơn, lớn hơn hoặc bằng với) trong mã riêng của chúng ta. Khả năng thêm các toán tử ngoài làm cho chúng ta có thể thêm những thứ mà nhà lập trình ban đầu đã không nghĩ đến cho các phép toán lớp.

3. Thêm mã từ Listing 11.2 vào file mã nguồn. Mã này có thể dễ dàng được thêm vào một ngày sau đó, trong một file riêng biệt bởi một nhà lập trình riêng biệt.

Listing 11.2: Các toán tử lớp Range

```
bool operator<(const IntRange& aRange, int
    aValue )
{
    return aRange.GetValue() < aValue;
}
bool operator==(const IntRange& aRange, int
    aValue )
{
    return aRange.GetValue() == aValue;
}
```

4. Lưu file mã nguồn và đóng code editor.

Test lớp Range

Sau khi tạo một lớp range, bạn nên tạo một driver thử nghiệm để không chỉ để bảo đảm mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn.

Sau đây là cách tạo một driver thử nghiệm vốn hiệu lực hoá các loại đầu vào khác nhau từ người dùng và minh hoạ lớp Range như được định nghĩa trong phần trước được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch11_1.cpp.

2. Gõ nhập mã từ Listing 11.3 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trong thư mục ch11 của Web site đi kèm của sách này.

Listing 11.3 Driver thử nghiệm lớp Range

```
int main(int argc, char **argv)
{
    IntRange i20(0,20);
    for (int i=1; i<argc; ++i )
    {
        i20 = atoi(argv[i]);
        printf("Setting value to %s, value
        is now %d\n", argv[i],
        i20.GetValue() );
    }
    i20 = 13;
    if ( i20 < 19 )
        printf("The value is under 19\n");
    else
        printf("The value is over 19\n");
    if ( i20 < 10 )
        printf("The value is under 10\n");
    else
        printf("The value is over 10\n");
    if ( i20 == 13 )
```

```
        printf("The value is 13\n");
    else
        printf("The value is NOT 13\n");
    return 0;
}
```

3. Biên dịch và chạy ứng dụng trong hệ điều hành mà bạn chọn .

   Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell trong hệ điều hành:

```
$ ./a.exe 1 2 -1 30
Setting value to 1, value is now 1
Setting value to 2, value is now 2
Setting value to -1, value is now 0
Setting value to 30, value is now 20
The value is under 19
The value is over 10
The value is 13
```

Như bạn có thể thấy lớp Range không cho phép các giá trị được gán bên ngoài các mục nhập hợp lệ mà chúng ta đã định nghĩa trong mã nguồn.

Chú ý lớp Range có thể được sử dụng như thế nào như thể nó là một kiểu cơ bản vốn là một phần của ngôn ngữ ngay từ đầu. Kỹ thuật mạnh mẽ đáng ngạc nhiên này cho bạn làm hầu như bất cứ điều gì mà bạn muốn (đó là bằng mã). Thậm chí nó cho phép chuyển đổi trực tiếp các kiểu dữ liệu riêng của bạn thành kiểu cơ sở mà bạn mở rộng để chuyển chúng trực tiếp đến các hàm vốn mong đợi kiểu cơ bản.

Kỹ thuật 12: Tạo các kiểu riêng của bạn

Tiết kiệm thời gian bằng cách

- Tạo các kiểu mà những người dùng sẽ muốn sử dụng]
- Tạo một lớp matrix
- Thêm các ma trận (matrix)
- Nhân một ma trận với một giá trị vô hướng
- Test lớp matrix

Dĩ nhiên, mặc dù tạo các phần mở rộng của các kiểu cài sẵn thì tốt, nhưng mục đích thực sự của lập trình trong C++ là tạo các kiểu

mới mà những nhà thiết kế ngôn ngữ không bao giờ nghĩ đến. Ví dụ, hãy tưởng tượng bạn cần làm việc với một ma trận (matrix) trong chương trình. Một ma trận (matrix) đơn giản là một mảng giá trị hai chiều. Kỹ thuật này sẽ hướng dẫn bạn tạo một kiểu mới, một lớp Matrix để sử dụng trong các phép tính đồ hoạ. Để làm điều này, bạn nên ghi nhớ rằng những người dùng không thật sự phải hiểu mọi thứ hoạt động như thế nào trong hậu cảnh; nếu họ có thể sử dụng những kiểu mới đó như thể chúng là một phần tự nhiên của ngôn ngữ, khả năng nhiều hơn người dùng sẽ sử dụng đối tượng và quay trở về nó lặp đi lặp lại nhiều lần. Đối với hệ thống lớp C++, điều đó có nghĩa là mục đích của chúng ta là làm cho đối tượng thành một thứ gì đó vốn có diện mạo và cách hoạt động như một kiểu cơ bản chẳng hạn như integer hoặc float.

Hãy bắt đầu với những điểm cơ bản về việc tạo một lớp Matrix. Lớp này sẽ cho phép bạn (cuối cùng) thực hiện đại số ma trận cơ bản - chẳng hạn như cộng hai ma trận, cộng hoặc nhân một hằng với một ma trận và những phép tính tương tự. Cú pháp tốt nhất cho lớp Matrix là một cú pháp mô phỏng gần giống với các ma trận thực tế - đó là một điều gì đó như sau:

```
Matrix m( 10, 10 );
M[5] [5] = 20 . 0;
```

Mã này định nghĩa một ma trận 10 x 10 và cấp phát không gian cho nó. Phần tử tại 5,5 sau đó được xác lập sang giá trị 20.0 và tất cả phần tử khác được xác lập sang giá trị 0.0 (vốn là mặc định). Ví dụ, chúng ta có thể tạo một lớp dẫn xuất vốn thực thi một ma trận đơn vị trong đó các giá trị đường chéo của ma trận từ trái sang phải được xác lập sang 1.0 và tất cả giá trị khác được xác lập sang 0.0.

Tuy nhiên, ngay lập tức chúng ta gặp phải một vấn đề rất nghiêm trọng. Mặc dù có thể - thật ra khá dễ dàng - ghi đè toán tử [ ] cho một lớp, nhưng không thể ghi đè toán tử [ ] [ ] (hoặc [ ] [ ] [ ], hay bất kỳ số cấp gián tiếp khác cho các mảng). Với giới hạn này, làm thế nào chúng ta tạo một lớp vốn "trông" giống như nó có một mảng hai chiều được đưa vào đó - và bạn có thể truy cập? Câu trả lời là một điều kỳ diệu nào đó của C++. Để xem điều này được thực hiện như thế nào, hãy bắt đầu xây dựng lớp Matrix bằng cách thực thi khả năng xem một mảng hai chiều như thể nó là một đối tượng đơn. Phần kế tiếp sẽ hướng dẫn bạn cách làm.

Tạo lớp Matrix

Lớp Matrix cho phép chúng ta xem một mảng hai chiều như thể nó là một đối tượng đơn. Đối với người dùng lớp này trông như thể nó là một mảng giá trị hai chiều, nhưng nó thực hiện việc kiểm tra lỗi và

cho phép người dùng truy vấn lớp để tìm những thuộc tính cơ bản, chẳng hạn như chiều rộng và chiều cao của ma trận. Để làm điều này, các bước sau đây hướng dẫn bạn cách tạo hai lớp riêng biệt, một lớp vốn đóng gói một hàng đơn của ma trận và một lớp vốn chứa các mảng của các hàng đó để thực thi ma trận hoàn chỉnh.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này file được đặt tên là ch12.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 12.1 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

3. Lưu file nguồn.

Listing 12.1 Lớp Matrix

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <vector>
class Row
{
private:
    std::vector< double > Columns;
public:
    Row( void )
    {
    }
    Row( int size )
    {
       // Initialize the column
       for ( int i=0; i<size; ++i )
          Columns.insert( Columns.end(),
    0.0 );
    }
    Row( const Row& aCopy )
    {
```

```
            std::vector< double >::const_iterator
            iter;
            for ( iter = aCopy.Columns.begin();
            iter !=
aCopy.Columns.end(); ++iter )
            {
               double d = (*iter);
               Columns.insert( Columns.end(),
      d);
      }
      }
      int size()
      {
         return Columns.size();
      }
      double& operator[](int index)
      {
         if ( index < 0 || index > Columns.
         size() )
            throw "Array Index out of
            Bounds";
            return Columns[ index ];
      }
};
class Matrix
{
private:
      std::vector< Row > Rows;
public:
      Matrix ( int rows, int cols )
      {
         for ( int i=0; i<rows; ++i )
         {
            Row r( cols );
            Rows.insert( Rows.end(), r );
```

```
        }
    }
    Row& operator[](int index)
    {
        if ( index < 0 || index > Rows.
        size() )
    throw "Array Index out of
    Bounds";
        return Rows[ index ];
        }
        void Print()
        {
           for ( int r=0; r<Rows.size(); ++r )
           {
              for ( int c=0; c<Rows[r].size();
              ++c )
                 printf(" %lf ", Rows[r][c]
                 );
              printf("\n");
           }
        }
        int RowCount()
        {
           return Rows.size();
        }
        int ColumnCount()
        {
           if ( Rows.size() )
              return Rows[0].size();
           return 0;
        }
};
```

Mã trong Listing 12.1 thực sự hầu như không làm gì cả - ngoại trừ cung cấp không gian lưu trữ cho ma trận và cung cấp các phương thức để đạt được không gian lưu trữ đó. Theo cách này, mã là một ví dụ hoàn hảo về thiết kế và lập trình hướng đối tượng. Bộ nhớ được cấp

phát, không gian được quản lý và dữ liệu được ẩn khỏi người dùng. Nếu không, chức năng thực sự của đối tượng nằm bên ngoài đối tượng (không phải là một ý hay). Chú ý việc sử dụng hai lớp riêng biệt để cho phép ảo giác về nhiều index ma trận.

Khi người dùng gọi ra toán tử [ ] trên lớp Matrix, nó thật sự trả về một đối tượng Row. Dĩ nhiên, tiến trình này trong suốt đối với người dùng vốn nghĩ rằng họ đơn giản thao tác trên một phần tử mảng nào đó trong ma trận. Sau khi bạn đạt được một phần tử Row, bạn sử dụng toán tử [ ] trên đối tượng đó để trả về các mục nhập Column riêng lẻ trong hàng.

Đối với người dùng dường như bạn đang sử dụng một mảng hai chiều. Kỹ thuật này có thể được sử dụng để thực thi bất kỳ số cấp mảng mà bạn muốn. Chỉ việc thay thế các mục nhập double trong lớp Column bằng các đối tượng Column bổ sung và bạn có một lớp mảng đa chiều vốn có thể xử lý hầu như bất kỳ số chiều.

Các phép toán ma trận

Sau khi chúng ta có lớp cơ bản ở đúng vị trí để thực thi việc lưu trữ dữ liệu cho ma trận, bước kế tiếp là thực thi chức năng để thao tác trên các ma trận. Đầu tiên, chúng ta cộng hai ma trận. Các bước sau đây hướng dẫn cách thực hiện:

1. Thêm mã từ Listing 12.2 vào file nguồn (hoặc lấy mã từ Web site đi kèm của sách này).

Những phép toán này nằm bên ngoài chức năng cơ bản của chính lớp, do đó chúng được trình bày dưới dạng các listing riêng biệt. Bạn có thể thêm chúng vào file gốc, tạo một file mới để chứa chúng, hoặc đặt chúng trong mã nguồn ứng dụng. Vì mục đích đơn giản, chúng ta chỉ thêm chúng vào file nguồn gốc.

Listing 12.2 Các toán tử ma trận

```
Matrix operator+( Matrix& m1, Matrix& m2 )
{
    // Here, we need to check that the rows
    // and columns of the two are the same.
    if ( m1.RowCount() != m2.RowCount() )
        throw "Adding Matrices: Invalid
        Rows";
    if ( m1.ColumnCount() !=
        m2.ColumnCount() )
        throw "Adding Matrices: Invalid
```

```
            Columns";
        Matrix m( m1.RowCount(),
                  m1.ColumnCount() );
        for ( int r=0; r<m1.RowCount(); ++ r )
        {
            for ( int c=0; c<m1.ColumnCount();
                  ++c )
                  m[r][c] = m1[r][c] + m2[r][c];
        }
        return m;
    }
```

2. Lưu file mã nguồn.

Ngoài chức năng thực sự của nó, đoạn mã này minh hoạ một số điểm quan trọng. Đầu tiên bởi vì toán tử được định nghĩa bên ngoài lớp, chúng ta chỉ có thể sử dụng các phương thức được định nghĩa là public trong lớp khi chúng ta làm việc trên dữ liệu. Thật may, như bạn có thể thấy qua mã, các phương thức đó là những gì mà chúng ta thật sự cần. Các "qui tắc" cho phép cộng ma trận khá đơn giản - bạn chỉ việc cộng các giá trị hàng và cột đó cho mỗi ma trận và đặt kết quả vào ma trận ra. Chú ý rằng bởi vì chúng ta trả về một đối tượng thay vì một tham chiếu dẫn sang một đối tượng, chúng ta cần quan tâm đến việc sao chép đối tượng. Nếu chúng ta trả về một đối tượng C++, nó sẽ tự động gọi ra constructor để tạo đối tượng ban đầu và sau đó copy constructor để trả về một bản sao của đối tượng. Thật may, copy constructor được định nghĩa cho lớp Matrix.

Một vấn đề với các toán tử là chúng không có cách thật sự để trả các lỗi trở về chương trình gọi. Ví dụ, khi bạn viết:

```
x + y + z
```

Thật sự không có cách nào để quyết định rằng một lỗi đã xuất hiện. Khi chúng ta cộng hai số nguyên, chúng ta có thể thật sự gây ra tất cả loại lỗi, chẳng hạn như underflow và overflows, nhưng những lỗi này hầu như được ẩn khỏi người dùng. Vì lý do này, cách duy nhất cho chúng ta có thể chỉ báo cho người dùng rằng đã có một sự cố là đưa ra một ngoại lệ. Đây là một quyết định rất khó thực hiện trong thiết kế bởi vì nó nẩy sinh khả năng xảy ra các ngoại lệ cho bất kỳ dòng nơi người dùng cuối viết mã - như trong ví dụ sau đây:

```
Matrix m3 = m1 + m2
```

Dòng này có thể đưa ra một ngoại lệ - trong trường hợp này bạn phải đặt nó trong một khối try / catch - nó có thể đẩy bạn thẳng ra khỏi ứng dụng. Rõ ràng cộng hai ma trận dường như không giống như loại điều mà bạn nên quan tâm về việc làm cho ứng dụng đổ vỡ. Chúng ta có thể đơn giản trả về một ma trận trống nếu các cận của cả hai ma trận không giống nhau; đó là một lựa chọn khác, nhưng là một lựa chọn phức tạp hơn. Bây giờ bạn nhận được một kết quả mà bạn đã không mong đợi từ một phép tính chuẩn. Đây là một lý do để không sử dụng các toán tử quá tải; chúng thường có các tác dụng phụ vốn không thật sự áp dụng cho các kiểu "chuẩn".

Nhân một ma trận với một giá trị vô hướng

Như với phép cộng hai ma trận, chúng ta cũng có thể nhân một ma trận với một giá trị vô hướng. Phép tính này thật sự dễ viết mã hơn phép cộng hai ma trận, nhưng có thêm một tác dụng phụ đáng bàn đến - trong một phút. Thứ tự công việc đầu tiên là viết mã cho phép tính. Để nhân một ma trận với một giá trị vô hướng, làm theo các bước sau đây:

1. Sử dụng code editor, mở lại file mã nguồn cho kỹ thuật này và xem nội dung của Listing 12.3.

Listing 12.3: Phép nhân vô hướng

```
Matrix operator*(Matrix& m1, double scalar)
{
    Matrix m( m1.RowCount(),
              m1.ColumnCount() );
    for ( int r=0; r<m1.RowCount(); ++ r )
    {
       for ( int c=0; c<m1.ColumnCount();
             ++c )
          m[r][c] = m1[r][c] * scalar;
    }
    return m;
}
```

2. Lưu file mã nguồn.

Sau đó bạn có thể sử dụng mã này trong chương trình - ví dụ bằng cách viết dòng sau đây:

```
Matrix m2 = m 1 * 4;
```

Lệnh này sẽ làm việc tốt, tạo một ma trận có kích cỡ thích hợp vốn là bội số vô hướng của ma trận gốc - và nó sẽ nhân tất cả phần tử với 4. Vấn đề xuất hiện khi bạn thử dòng sau đây:

```
Matrix m2 = 4 * m 1;
```

Bạn đã đảo ngược thứ tự của các toán hạng - và bất ngờ có một vấn đề với trình biên dịch. Bạn nhận được một lỗi như sau:

```
error: no match for 'operator*' in '4 * m4'
error: candidates are: Matrix
    operator*(Matrix&, double)
```

Lý do bạn nhận được lỗi này là trình biên dịch lấy lệnh 4 * m 1 và biên dịch nó thành một lệnh gọi đến operator * (int, Matrix& m 1). Bạn đã không định nghĩa phương thức này.

Đây là nơi điều kỳ diệu rõ rệt của C++ trở nên tinh vi hơn. C++ cho phép bạn định nghĩa các toán tử để cộng các lớp vốn sử dụng đơn giản như cộng hai số (như 1+2). Nó có thể xử lý những thứ đơn giản - ví dụ nhân số nguyên vô hướng - và nó hiểu rằng các số có thể được nhân theo bất kỳ thứ tự. Vấn đề xảy ra khi bạn cố áp dụng khái niệm đó vào các lớp riêng của bạn. Bạn phải áp dụng một vài thủ thuật riêng của bạn. Phần kế tiếp sẽ cho bạn biết cách thực hiện.

Nhân một ma trận với các giá trị vô hướng

Để giải quyết vấn đề này, chúng ta cần tạo một toán tử mới có hầu như cùng một mã và đặt các đội số theo thứ tự đối nghịch. Các bước sau đây hướng dẫn bạn cách làm:

1. Sử dụng code editor, mở lại file mã nguồn cho kỹ thuật này và chỉnh sửa mã như bạn thấy trong Listing 12.4.

Listing 12.4 Các hàm xử lý ma trận

```
Matrix scalar_multiplication( Matrix& m1,
    double scalar )
{
    Matrix m( m1.RowCount(),
            m1.ColumnCount() );
    for ( int r=0; r<m1.RowCount(); ++ r )
    {
       for ( int c=0; c<m1.ColumnCount();
          ++c )
          m[r][c] = m1[r][c] * scalar;
```

```
        }
        return m;
    }
    Matrix operator*(Matrix& m1, double scalar)
    {
        return scalar_multiplication( m1, scalar
        );
    }
    Matrix operator*(double scalar, Matrix& m1 )
    {
        return scalar_multiplication( m1, scalar
        );
    }
```

Chú ý mã này thay thế operator* hiện có đã được thực thi trước đó. Loại bỏ phần thực thi hiện có nếu không, bạn sẽ nhận được một lỗi định nghĩa trùng lặp từ trình biên dịch để định nghĩa hai phương thức đó.

Mã này cho phép chúng ta viết:

```
Matrix m2 = 4 * m 1;
```

hoặc

```
Matrix m2 = m 1 * 4
```

Bởi vì trình biên dịch có thể phân giải một trong hai thứ tự thành một phương thức hợp lệ, nó sẽ cho phép bạn thực hiện cả hai trong mã. Bởi vì hành động nhân thật sự giống nhau như một trong hai trường hợp, factor out, mã vốn thực hiện công việc "thực sự" và gọi nó từ cả hai phương thức. Refactoring này giảm tổng lượng mã và làm cho dễ theo dõi các vấn đề hơn.

2. Lưu file mã nguồn.

Test lớp ma trận

Sau khi tạo một lớp Matrix, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn.

Sau đây là thủ tục tạo một driver thử nghiệm vốn hiệu lực hoá các loại đầu vào khác nhau từ người dùng và minh hoạ cách sử dụng lớp Matrix như thế nào.

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này chương trình thử nghiệm được đặt tên là ch12.cpp.

2. Gõ nhập mã từ Listing 12.5 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 12.5: Driver thử nghiệm ma trận.

```
int main()
{
    Matrix m1(5,5);
    Matrix m2(5,5);
    m1[2][2] = 3;                    → 1
    m2[2][2] = 4;
    m2[3][3] = 5;
    Matrix m3 = m1 + m2;             → 2
    printf("Matrix 1:\n");
    m1.Print();
    printf("Matrix 3:\n");
    m3.Print();
    Matrix m4 = m3 * 5;              → 3
    Matrix m5 = 4 * m4;
    printf("Matrix 4:\n");
    m4.Print();
    printf("Matrix 5:\n");
    m5.Print();
}
```

3. Biên dịch và chạy ứng dụng trong hệ điều hành mà bạn chọn.

   Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả từ Listing 12.6 trong cửa sổ shell trên hệ thống.

Listing 12.6: Kết quả từ chương trình thử nghiệm ma trận.

```
$ ./a.exe
Matrix 1:
0.000000 0.000000 0.000000 0.000000
```

```
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 3.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
Matrix 3:
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 7.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 5.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
Matrix 4:
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 35.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 25.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
Matrix 5:
0.000000 0.000000 0.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
```

```
        0.000000
0.000000 0.000000 140.000000 0.000000
        0.000000
0.000000 0.000000 0.000000 100.000000
        0.000000
0.000000 0.000000 0.000000 0.000000
        0.000000
```

Như bạn có thể thấy từ kết quả ở trên, chúng ta hiển thị các đối tượng ma trận riêng lẻ vốn được tạo trong chương trình thử nghiệm. Kết quả cho thấy rằng ma trận đầu tiên (m1) hiển thị các giá trị dữ liệu được đặt vào nó trong dòng được đánh dấu → **1** trong mã driver thử nghiệm. Dòng được đánh dấu → **2** trình bày việc cộng hai ma trận mà sau đó được hiển thị dưới dạng Matrix 3. Tương tự mã được đánh dấu → **3** biểu thị việc nhân một ma trận với một giá trị vô hướng vốn được hiển thị trong kết quả dưới dạng Matrix 4. Nếu bạn làm phép toán, bạn sẽ thấy tất cả kết quả chính xác biểu thị rằng lớp Matrix và các phương thức xử lý của nó làm việc tốt.

Kỹ thuật 13: Sử dụng các kiểu liệt kê

Tiết kiệm thời gian bằng cách

- Định nghĩa các kiểu liệt kê
- Thực thi lớp Enumeration
- Test lớp Enumeration

Enumeration (kiểu liệt kê) là một kiểu ngôn ngữ được giới thiệu trong ngôn ngữ C đã di trú hầu như nguyên vẹn vào ngôn ngữ C++. Các kiểu liệt kê không phải là các kiểu thực, như các lớp. Bạn không thể định nghĩa các toán tử cho các kiểu liệt kê, bạn cũng không thể thay đổi cách làm việc của chúng. Như với các lệnh tiền xử lý, lệnh enumeration thật sự là một thứ mang tính cú pháp hơn, thay thế các giá trị hằng bằng các tên dễ đọc hơn. Nó cho bạn khả năng đọc hơi tốt hơn, nhưng nó không làm gì để thay đổi cách làm việc của mã. Các kiểu liệt kê cho phép bạn sử dụng những tên có ý nghĩa cho các giá trị và cho phép trình biên dịch thực hiện việc kiểm tra kiểu tốt hơn. Khuyết điểm của một kiểu liệt kê là bởi vì nó chỉ là một sự thay thế cú pháp cho một giá trị dữ liệu, bạn có thể dễ dàng đánh lừa trình biên dịch bằng cách gán các giá trị không hợp lệ vào kiểu liệt kê.

Dạng cơ bản của một kiểu liệt kê như sau:

```
enum <name> {
value1[=number],
value2,
value3,

valuen
} EnumerationTypeName;
```

Trong đó trường <name> là kiểu liệt kê mà chúng ta tạo, các tham số value là những giá trị riêng lẻ được định nghĩa trong kiểu liệt kê và number là điểm khởi đầu tuỳ chọn để bắt đầu đánh số các giá trị liệt kê. Ví dụ, hãy xem kiểu liệt kê sau đây:

```
enum color {
    Red = 1,
    White = 2,
    Blue
} ColorType;
```

Trong ví dụ này, mỗi lần trình biên dịch gặp phải Color Type::Red trong ứng dụng, nó hiểu giá trị là 1, White là 2 và Blue là 3 (bởi vì các số liên tục trừ phi bạn xác định theo cách khác).

Nếu các kiểu liệt kê thật sự không thay đổi lô gíc của mã, tại sao phải bận tâm đến chúng? Lý do chính cho các kiểu liệt kê là cải thiện khả năng đọc mã. Để minh hoạ điều này bạn sẽ xem một kỹ thuật đơn giản liên quan đến các kiểu liệt kê mà bạn có thể sử dụng để làm cho mã của bạn hơi sử dụng an toàn hơn và dễ hiểu hơn nhiều.

Bạn sẽ thấy các kiểu liệt kê là một dạng đơn giản hơn của lớp hiệu lực hoá Range được phát triển trong kỹ thuật 11. Các kiểu liệt kê được thi hành bởi trình biên dịch, không phải bởi mã của bạn và đòi hỏi ít công sức hơn nhiều để thực thi so với một lớp kiểm tra Range. Đồng thời chúng không mạnh bằng. Các kiểu liệt kê được sử dụng cho các mối quan tâm về khả năng đọc và sự bảo trì hơn là cho sự hiệu lực hoá.

Listing 13.1: Chương trình Enumeration

```
#include <stdio.h>
typedef enum
{
    Red = 0,
    Yellow,
```

```
    Green
} TrafficLightColor;
int ChangeLight( int color )
{
    switch ( color )
    {
        case 1: // Red
            printf("Changing light to RED. Stop!!\n");
            break;
        case 2: // Yellow
            printf("Changing light to YELLOW. Slow down\n");
            break;
        case 3: // Green
            printf("Changing light to GREEN. Go for it\n");
            break;
        default:
            printf("Invalid light state. Crashing\n");
            return -1;
    }
    return 0;
}
int ChangeLightEnum( TrafficLightColor color )
{
    switch ( color )
    {
        case Red: // Red
            printf("Changing light to RED. Stop!!\n");
            break;
        case Yellow: // Yellow
            printf("Changing light to YELLOW. Slow down\n");
            break;
        case Green: // Green
            printf("Changing light to GREEN. Go for it\n");
            break;
    }
```

```
    return 0;
}
```

Thực thi lớp Enumeration

Một kiểu liệt kê (enumeration) khi đối tượng thực tế mà nó mô phỏng có những giá trị rất đơn giản, rất riêng biệt. Ví dụ đầu tiên nảy ra trong ý nghĩ ngay tức thời là một đèn giao thông có ba trạng thái: đỏ, vàng và xanh lá cây. Trong các bước sau đây, hãy tạo một ví dụ đơn giản sử dụng phép ẩn dụ đèn giao thông để minh hoạ cách các kiểu liệt kê làm việc và có thể được sử dụng trong ứng dụng của bạn như thế nào.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là Ch13.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Lisdting 13.1 vào file.

   Tốt hơn, hãy sao chép mã mà bạn tìm thấy trên Web site đi kèm của sách này và thay đổi tên của các hằng và biến khi bạn chọn.

3. Lưu file mã nguồn.

Test lớp Enumeration

1. Thêm mã sau đây để test kiểu liệt kê và xác nhận rằng nó làm việc tốt.

Mã này có thể dễ dàng được di chuyển đến một file riêng biệt. Nó được đặt trong một file để tiện lợi. Mã minh hoạ tại sao các kiểu liệt kê an toàn kiểu (type-safe) hơn những kiểu số nguyên cơ bản và tại sao bạn có thể muốn sử dụng các kiểu liệt kê so với các kiểu cơ bản.

```
int main(int argc, char **argv)
{
    int clr = -1;
    ChangeLight( clr );
    TrafficLightColor c = Red;
    ChangeLightEnum( c );
    return 0;
}
```

2. Lưu file mã nguồn và đóng code editor.

3. Biên dịch và chạy ứng dụng bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

   Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ shell.

```
$ ./a.exe
Invalid light state. Crashing
Changing light to RED. Stop!!
```

Kỹ thuật 14: Tạo và sử dụng các cấu trúc

Tiết kiệm thời gian bằng cách:

- Định nghĩa các cấu trúc
- Tìm hiểu những ưu điểm của các cấu trúc so với các lớp
- Thực thi các cấu trúc
- Sử dụng các cấu trúc dẫn xuất
- Hiểu kết quả của lớp cấu trúc

Một trong những cấu trúc thú vị nhất được tạo cho ngôn ngữ lập trình C là structure (cấu trúc). Bởi vì nó đã cho phép nhà phát triển nhóm một cụm dữ liệu liên quan lại với nhau và chuyển dữ liệu đó đến các hàm khác nhau, cấu trúc C++ structure đã là sự khởi đầu của sự đóng gói (encapsulation) cho ngôn ngữ.

Khi ngôn ngữ C++ đã được thiết kế, cấu trúc là thành phần chính - thật ra các lớp C++ đơn giản là những phần mở rộng của cấu trúc. Các "trình biên dịch" C++ ban đầu thật sự đã là những chương trình biên dịch vốn lấy mã C++ và viết nó lại thành C, sau đó mã này được biên dịch bằng cách sử dụng trình biên dịch chuẩn cho ngôn ngữ đó. Điều này yêu cầu tất cả phần quản lý của các lớp phải được thực thi trong các cấu trúc. Cấu trúc C++ struct thật ra là một lớp mà trong đó tất cả thành viên là public.

Các cấu trúc vẫn hiện hữu trong C++. Thực tế, một cấu trúc chỉ là một lớp vốn làm cho tất cả thành viên dữ liệu của nó trở nên public theo mặc định. So sánh điều đó với một lớp chuẩn vốn có tất cả thành viên được chỉ định là private theo mặc định. Thật ra không có sự khác biệt giữa

```
struct foo {
    int x;
    int y;
};
```

và

```
class foo
{
public:
    int x;
    int y;
};
```

Ở đây C++ có một ưu điểm: bạn có thể làm nhiều với các cấu trúc trong C++ hơn so với bạn có thể làm trong C. Các cấu trúc có thể chứa những constructor (phương thức tạo) và accessor (phương thức truy cập). Thậm chí chúng có thể chứa những phương thức vốn không thao tác trên dữ liệu của chính lớp. Ngay cả bạn có thể có được những cấu trúc được dẫn xuất từ những cấu trúc khác. Mặc dù các trình biên dịch C++ ban đầu đã chuyển đổi các lớp thành những cấu trúc, nhưng các trình biên dịch mới hơn phân biệt giữa hai đối tượng này. Một lớp (class) vẫn là một struct, nhưng cách hiểu thì khác nhau. Một struct C++ không còn hoàn toàn tương thích ngược với C nữa.

Bạn không thể làm điều gì với các cấu trúc? Một mặt, bạn không thể có các phương thức ảo (virtual). Các cấu trúc không chứa các v-table ấn định sẵn, do đó chúng không thể chứa một hàm ảo. Rõ ràng, điều này có nghĩa rằng bạn không thể ghi đè các phương thức trong "cấu trúc cơ sở" cho một cấu trúc dẫn xuất. Điều này cũng có nghĩa là chúng không thể có các virtual destructor (phương thức huỷ tạo ảo).

Thực thi các cấu trúc

Phần này sẽ khai thác cách thực thi những cấu trúc trong C++ cũng như những gì bạn có thể thực hiện và không thể thực hiện với chúng. Kỹ thuật này sẽ xem xét cấu trúc kiểu C gốc, cấu trúc C++ cải tiến với sự khởi tạo và một cấu trúc C++ hoàn chỉnh hơn vốn chứa những phương thức.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch14.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã bên dưới vào file của bạn.

```
typedef struct classic_c_structure
{
    int x;
    int y;
```

```
} POINT;
```

Đây sẽ là cấu trúc "chuẩn" vì nó được thực thi trong C. Bây giờ hãy thử điều này trong C++ với một cải tiến nhỏ để tận dụng những gì mà ngôn ngữ mang lại.

3. Thêm định nghĩa cấu trúc sau đây vào file mã nguồn của bạn, sử dụng code editor ưa thích.

```
typedef struct c_plus_plus_structure
{
    int x;
    int y;
    c_plus_plus_structure()
    {
       x = 0;
       y = 0;
    }
} CPP_POINT;
```

Cấu trúc được liệt kê trong mã ở trên giống với cấu trúc trước, nhưng nó chứa một constructor tự động khởi tạo các giá trị trong cấu trúc, một sơ suất thường thấy trong số các nhà lập trình.

4. Thêm định nghĩa cấu trúc sau dây vào file mã nguồn, sử dụng code editor ưa thích:

```
typedef struct c_plus_plus_enhanced
{
    int x;
    int y;
    c_plus_plus_enhanced()
    {
       x = 0;
       y = 0;
    }
    void print()
    {
    printf("x = %d\n", x );
    printf("y = %d\n", y );
    }
} CPP_POINTE;
```

Trong trường hợp này, chúng ta đơn giản mở rộng cấu trúc C++ để có một phương thức khác cho phép kết xuất những giá trị dữ liệu của lớp.

5. Thêm một cấu trúc dẫn xuất vào file. Nhập mã sau đây vào code editor.

   Mã này sẽ cho bạn thấy việc dẫn xuất được xử lý như thế nào trong C++ cho các cấu trúc:

```
typedef struct new_struct : public
    CPP_POINTE
{
    int version;
    new_struct()                                    → 2
    {
       version = 1;
    }
    void print()
    {
    CPP_POINTE::print();
    printf("version = %d\n",
    version );
    }
} NEW_POINT;
```

6. Lưu file mã nguồn.

Hiểu kết quả

Sau khi bạn có mã thật sự cho cấu trúc được thực thi, bạn nên test để minh hoạ nó được sử dụng như thế nào và để xác thực nó làm việc tốt.

1. Thêm mã từ Listing 14.1 vào file mã nguồn ngay bên dưới các định nghĩa cấu trúc.

Listing 14.1: Test cấu trúc

```
void print_point( CPP_POINTE& p)
{
    p.print();
}
int main(int argc, char **argv)
```

```
{
    POINT p;
    CPP_POINT p1;
    CPP_POINTE p2;
    NEW_POINT p3;
    printf("POINT:\n");
    printf("X: %d\n", p.x);
    printf("Y: %d\n", p.y);
    printf("POINT 1:\n");
    printf("X: %d\n", p1.x);
    printf("Y: %d\n", p1.y);
    printf("POINT 2:\n");
    print_point(p2);
    printf("POINT 3:\n");
    print_point(p3);
}
```

Mã này đơn giản thi hành những cấu trúc khác nhau đã được định nghĩa trong file nguồn. Bạn sẽ có thể thấy điều gì xảy ra khi tiến trình khởi tạo hoàn tất và khi nó không hoàn tất.

2. Lưu file mã nguồn và đóng ceode editor.
3. Biên dịch và chạy chương trình, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ shell.

```
$ ./a.exe
POINT:
X: 2289768                                    → 1
Y: 1627507534
POINT 1:
X: 0
Y: 0
POINT 2:
x = 0
y = 0
POINT 3:
```

```
x = 0
y = 0
```

Có vài điều cần chú ý ở đây. Đầu tiên, bạn có thể thấy tại sao không khởi tạo các giá trị dữ liệu của một cấu trúc là một điều không nên làm (xem các dòng được biểu thị bởi → 1). Điểm đầu tiên, cấu trúc kiểu C đặc trưng chứa mã linh tinh vốn có thể dễ dàng gây ra những vấn đề nghiêm trọng trong một ứng dụng.

Hãy tưởng tượng mọi thứ sẽ tệ hại hơn như thế nào nếu cấu trúc chứa các pointer: các pointer vốn không được khởi tạo để trỏ vào một thứ nào đó sẽ trỏ vào một phần bộ nhớ không hợp lệ hoặc tệ hơn, trỏ vào một phần bộ nhớ mà không nên được chỉnh sửa.

Chú ý rằng khi một constructor được thêm vào cấu trúc, phiên bản cải tiến tự động gọi constructor (→ 2) cho lớp như bạn mong đợi. Theo cách đó, các phần tử cấu trúc đã được khởi tạo mà không đòi hỏi bất kỳ công việc từ người dùng. Hiển nhiên, bạn cũng có thể có các constructor đòi hỏi các đối số.

Thật tiện lợi khi có thể kết xuất các giá trị dữ liệu cho một cấu trúc một cách nhanh chóng và dễ dàng mà không làm bề bộn chương trình bằng các câu lệnh printf - như biến thể thứ ba của cấu trúc minh hoạ hàm thành viên print( ) của nó. Dĩ nhiên chúng ta có thể dễ dàng viết một hàm chấp nhận một cấu trúc có kiểu thích hợp để in nó ra (như chúng ta đã làm với kiểu thứ hai), nhưng sau đó nó phải xuất hiện ở mọi nơi cấu trúc đã được sử dụng.

Với cấu trúc dẫn xuất (new_struct), có một số điều thú vị cần lưu ý. Trước tiên, bởi vì chúng ta không thể ghi đè hàm print trong lớp cơ sở, cấu trúc không in ra dữ liệu vốn chỉ nằm trong lớp dẫn xuất. Đây là do giới hạn không có các virtual table (bảng ảo) trong các cấu trúc. Tuy nhiên, chúng ta có thể chuyển cấu trúc này đến bất cứ nơi nào vốn chấp nhận lớp cơ sở của nó - tương tự như chúng ta có thể với một lớp bình thường. Theo cách này, cấu trúc "có chức năng" như một lớp.

Kỹ thuật 15: Tìm hiểu các hằng

Tiết kiệm thời gian bằng cách:

- Khai thác những công dụng của các hằng
- Định nghĩa các hằng
- Thực thi các hằng
- Sử dụng từ khoá const

Bằng cách sử dụng cấu trúc hằng (hoặc câu lệnh constant) trong mã, các ứng dụng có thể làm cho an toàn hơn, dễ đọc hơn và hiệu quả hơn. Tuy nhiên, các nhà lập trình thường rất choáng ngợp trước số cách khác nhau đáng ngạc nhiên mà họ có thể sử dụng các hằng (constant) đến nỗi họ đơn giản tránh hoàn toàn cấu trúc, cho phép trình biên dịch chọn lựa những gì có thể thay đổi và không thể thay đổi trong ứng dụng. Một điều cần lưu ý: cho phép trình biên dịch thực hiện những lựa chọn cho bạn rất hiếm khi là một ý hay.

Kỹ thuật này sẽ khai thác những khả năng khác nhau để làm việc với các hằng trong C++ và ý nghĩa của chúng đối với bạn với tư cách là một nhà phát triển ứng dụng.

Định nghĩa các hằng

Để hiểu rõ nhất các hằng làm việc như thế nào và cách bạn có thể tận dụng chúng trong ứng dụng, các bước sau đây sẽ trình bày cho bạn một ví dụ đơn giản về việc định nghĩa các loại hằng khác nhau. Chúng ta sẽ tạo một file chứa các hằng để sử dụng làm các số nguyên, số dấu động và chuỗi ký tự. Bạn sẽ thấy cách một hằng có thể trực tiếp thay thế một giá trị #define cũng như cách trình biên dịch có thể được sử dụng để kiểm tra an toàn kiểu các phép gán hằng như thế nào.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch15.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 15.1 vào file của bạn.

Listing 15.1: Các hằng và định nghĩa của chúng

```
#include <stdio.h>
#include <stdlib.h>
// This is a constant value that can be used
// in place of a #define value
const int MaxValues = 100;
// Unlike the #define, you can use a const
// for typesafe constants
const char *StringName = "Matt Telles";
const double Cost = 100.35;
const long MaxLong = 1000000;
```

Bạn có thể định nghĩa các hằng hầu như có bất kỳ hình dạng hoặc kích cỡ. Các hằng có thể là các số, chuỗi, ký tự hoặc float. Quan trọng hơn trình biên dịch sẽ kiểm tra tính an toàn kiểu khi bạn gán một hằng vào một giá trị khác. Ví dụ, bạn được phép gán các giá trị chuỗi (string) vào các hằng chuỗi như sau:

```
string myString = StringName;
```

Tuy nhiên, bạn không được phép gán các giá trị MaxLong vào các số nguyên (integer) như sau:

```
int iVal = MaxLong; // The compiler will complain
```

3. Lưu file mã nguồn.

Dĩ nhiên tất cả những gì mà việc thêm các hằng đã thực sự thực hiện là cho chúng ta một cách khác để thay thế câu lệnh #define trong ứng dụng. Điểm chính thực sự của hằng C++ là cung cấp các chỉ lệnh biên dịch trong các hàm và lớp như bạn sẽ thấy ở phần kế tiếp.

Thực thi các biến hằng

Câu lệnh const cũng có thể được sử dụng để cho trình biên dịch biết một điều nào đó không được phép thay đổi như chúng ta sẽ thấy trong kỹ thuật đơn giản này vốn phụ thuộc vào một khía cạnh khác của từ khoá const. Làm theo các bước sau đây để thấy tất cả điều này như thế nào:

1. Thêm mã từ Listing 15.2 vào file mã nguồn của bạn.

Listing 15.2: Một hàm với một đối số bất khả biến.

```
// Functions can take constant arguments,
    allowing them
// to guarantee that values don't change.
int func( const int& x )
{
    // We can do this
    int z = x;
    // We cannot do this. Compile Error:
    //x = 10;                                    → 1
    return x;
}
```

Hàm này chấp nhận một đối số của tham chiếu kiểu const int. Modifier const cho trình biên dịch biết rằng đối số đầu vào không thể được chỉnh sửa. Nếu bạn không chú giải dòng được đánh dấu là →1,

bạn thấy trình biên dịch tạo một lỗi cho bạn biết rằng bạn không thể chỉnh sửa một tham chiếu hằng.

Ví dụ này xem xét một hàm vốn chấp nhận một đối số một số nguyên, hằng x. Chúng ta báo cho cả người dùng hàm và trình biên dịch biết rằng hàm này không bao giờ thay đổi giá trị của đối số đó thậm chí nếu nó được chuyển vào theo tham chiếu. Thông tin này rất quan trọng đối với trình biên dịch; với thông tin này nó có thể tối ưu hoá hàm để nó không cần phải đẩy trở lại giá trị ra khỏi ngăn xếp (stack) và bảo đảm rằng vị trí bộ nhớ mà nó sử dụng sẽ không thay đổi. Điều này có thể bởi vì dữ liệu có thể được loại bỏ sau khi hàm được hoàn tất. Sau cùng, có thể nó đã không thay đổi.

Hơn nữa, bởi vì chúng ta biết rằng đầu vào là một hằng, chúng ta có thể chuyển vào các giá trị vốn không đổi. Ví dụ, nếu tham chiếu đã không phải là một hằng, bạn phải viết dòng sau đây:

```
int x = 3;
int myVal = func( x );
```

bởi vì trình biên dịch không cho phép bạn viết dòng sau đây:

```
int myVal = func (3);
```

Tuy nhiên, khi bạn định nghĩa đối số đầu vào là một hằng, trình biên dịch bây giờ biết rằng vị trí bộ nhớ thật sự đang chứa giá trị của bạn (trong trường hợp này là → 3) sẽ không thay đổi và nó cho bạn chuyển vào giá trị số nguyên mà trước tiên không gán nó vào bất cứ giá trị nào. Cách sắp xếp này tiết kiệm một vài chu kỳ CPU và một số bộ nhớ - và bạn không cần phải viết một số mã ngớ ngẩn vốn không thật sự làm bất cứ điều gì.

2. Sử dụng code editor, thêm mã từ Listing 15.3 vào file mã nguồn của bạn.

Kỹ thuật này minh hoạ cách bạn có thể sử dụng từ khoá const trong một lớp sẽ hoàn thành những điều tương tự mà nó thực hiện bên ngoài một class listing.

Listing 15.3: Các hằng trong các lớp

```
// Classes can have constants in them
const int MaxEntries = 10;
class Foo
{
    int entries[ MaxEntries ];
public:
    // You can pass in constant references to arguments
```

```
Foo()
{
}
Foo( const Foo& aCopy )
{
    for ( int i=0; i<MaxEntries; ++i )
        entries[i] = aCopy.entries[i];
}
// You can return constant references from methods
const int * getEntries()
{
    return &entries[0];
}
// You can indicate that a method will NOT change
// anything in the object
int getEntry( int index ) const
{
    return entries[index];
    }
    // The two can be combined to say that the return value
    // cannot be changed and the object will not change
    const int& getAConstEntry( int index ) const
    {
        return entries[index];
    }
};
```

3. Lưu file mã nguồn và đóng code editor.

Như bạn có thể thấy cấu trúc const rất linh hoạt. Nó có thể được sử dụng để biểu thị một giá trị vốn được sử dụng để thay thế một số trong mã. Nó có thể được sử dụng để biểu thị một giá trị trả về vốn không thể được thay đổi. Nó có thể biểu thị một phương thức class chấp nhận một đối số mà nó không thay đổi, chẳng hạn như constructor Copy. Bạn có thể tưởng tượng viết một constructor Copy đã thay đổi đối tượng mà nó sao chép hay không? Giả sử bạn viết một dòng nào đó như sau:

```
Foo f1 = f;
```

Sau đó giả sử bạn làm cho đối tượng f thay đổi từ bên dưới bạn - nói về cơn ác mộng gỡ rối cơ bản. Vì lý do này, có thể tuỳ ý sử dụng một tham chiếu const biểu thị rằng bạn sẽ không thay đổi đối tượng đang được sao chép. Theo cùng một cách, chúng ta có thể chuyển vào các giá trị đến những phương thức và bảo đảm với người dùng rằng chúng sẽ không được sao chép (như trong hàm func trong Listing 15.2).

Dĩ nhiên, nếu bạn lấy một giá trị đầu vào và bảo đảm với người dùng rằng bạn sẽ không thay đổi nó thì bạn phải có khả năng cho một người nào đó trở lại một giá trị và bảo đảm họ không thay đổi nó. Đây được gọi là việc trả về một tham chiếu const. Ví dụ, nếu bạn có một biến trong, bạn có thể tạo một phương thức tham chiếu vốn cho nó trở lại, nhưng chỉ theo kiểu chỉ đọc (read-only). Đó là những gì mà phương thức getEntries thực hiện trong Listing 15.3. Nó trả về một pointer const vốn bảo đảm rằng người dùng không thay đổi bất cứ thứ gì trong chương trình gọi đối tượng.

Sau cùng, bạn có thể cho người dùng biết rằng phương thức mà bạn gọi sẽ không bao giờ thay đổi đối với đối tượng. Để làm điều này, bạn chỉ việc thêm từ khoá const ở cuối phương thức để cho phép phương thức của bạn gọi trên những đối tượng const. Điều này cũng sẽ cho phép trình biên dịch tránh phải tạo các bản sao của các đối tượng. nếu đối tượng không thể được thay đổi thông qua phương thức, không có lý do nào để bận tâm về việc làm cho vị trí bộ nhớ trở nên static.

Test ứng dụng Constant

Sau khi tạo lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn.

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch15.cpp.

   Bước kế tiếp là thêm một driver thử nghiệm đơn giản vào file nguồn để bạn có thể xem driver này hoạt động như thế nào.

2. Gõ nhập mã từ Listing 15.4 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đính kèm của sách này.

Listing 15.4: Driver thử nghiệm cho const.

```
int main(int argc, char **argv)
{
```

```
    Foo f;
    // Note that to get back the entries, we
    MUST define
    // our return type as constant
    const int *entries = f.getEntries();          → 2
    // You can't do this.
    // entries[2] = 2;
    return 0;
}
```

3. Lưu file mã nguồn và đóng code editor.
4. Biên dịch ứng dụng bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Chú ý chúng ta phải sử dụng một pointer constant (→ 2) để truy cập các mục nhập trong lớp và rằng chúng ta không thể chỉnh sửa các giá trị trong khối mục nhập (entry) đó được chuyển trở về. Trình biên dịch cũng cố cả hai điều kiện này vào thời gian biên dịch.

Do đó bạn có thể thấy const làm việc như thế nào trong thế giới C++ - và bạn có thể sử dụng nó như thế nào để thi hành những nhu cầu của ứng dụng trên người dùng cuối (dĩ nhiên theo một cách tốt).

Sử dụng từ khoá const

Có một công việc khác cho từ khoá const linh hoạt: bạn có thể sử dụng nó để phân biệt nhằm quyết định phương thức nào cần sử dụng; từ khoá const là một phần của chữ ký (signature) của một phương thức. Điều đó bạn có thể có hai phương thức được minh hoạ trong Listing 15.5 trong lớp của bạn cùng một lúc. Từ khoá const không được hiểu bởi người dùng mà là bởi trình biên dịch để quyết định phương thức nào đang đang được gọi. Nếu giá trị được chỉnh sửa, nó sẽ gọi phiên bản không phải const. Nếu giá trị không thể được chỉnh sửa, nó sẽ gọi phiên bản const.

Listing 15.5: Sử dụng const để phân biệt hai phương thức

```
// You can indicate that a method will
    NOT change
// anything in the object
int getEntry( int index ) const                  → 3
{
    return entries[index];
```

```
}
// Or not, depending on how you feel.
int getEntry( int index )                                    → 4
{
    return entries[ index ];
}
```

Phương thức đầu tiên trong những phương thức có thể được sử dụng từ bất kỳ đối tượng, constant (→ 3) hoặc (→ 4). Mặt khác, phương thức thứ hai chỉ được sử dụng từ một đối tượng không phải const. Nếu bạn chọn gọi phương thức thứ hai, bạn phải chấp nhận rủi ro là người dùng sẽ chỉnh sửa trực tiếp trong lớp của bạn.

Sau cùng cần chỉ ra rõ rằng const là thứ thật sự có thể được loại bỏ nếu người dùng chọn làm như vậy một cách rõ ràng. Do đó nếu bạn tạo phương thức getEntries (như trong Listing 15.3) - vốn đòi hỏi bạn sử dụng một pointer const int để gọi phương thức - nhà lập trình có thể khắc phục vấn đề nhỏ này bằng cách viết mã như mã trong Listing 15.6.

Listing 15.6 Loại bỏ tính const

```
// Now, you can break things this way
int *ent2 = (int *)f.getEntries();                           → 5
ent2[2] = 2;
```

C++ luôn giả định rằng nhà lập trình biết mình đang làm gì thậm chí nếu cho phép làm một số điều một cách không chính xác (vi phạm tinh thần của ngôn ngữ). Bằng cách gán rõ ràng (→ 5), bạn có thể "huỷ const" một pointer (nghĩa là làm cho một pointer const trở thành không phải const) và làm bất cứ điều gì bạn muốn. Trình biên dịch giả định rằng bởi vì bạn đã làm việc gán (cast), nên bạn biết chính xác kết quả cuối cùng là gì và hiểu mọi chi tiết phức tạp cho toàn bộ chương trình. Nếu pointer đã được định nghĩa là không đổi lúc ban đầu, có lẽ có một lý do tốt cho nó: trong việc phát triển ứng dụng riêng của bạn, bằng mọi giá hãy tránh một kỹ thuật như vậy.

Kỹ thuật 16: Định phạm vi các biến

Tiết kiệm thời gian bằng cách

- Định nghĩa phạm vi (scope)
- Khai thác bản chất của phạm vi

- Kết hợp các constructor và destructor để điều khiển và dự đoán phạm vi

Khái niệm về "scope" (phạm vi) khá độc đáo đối với những ngôn ngữ C và C++. Scope còn được gọi là thời gian sống (lifetime) là khoảng thời gian của ứng dụng mà một biến tồn tại trong khoảng thời gian đó. Một số biến có thể tồn tại trong toàn bộ chiều dài của chương trình (và thật không may như một số rò rỉ bộ nhớ nằm khá xa bên ngoài chiều dài đó) trong khi những biến khác có các khoảng thời gian rất ngắn - nghĩa là ít phạm vi hơn. Chiều dài thời gian sống của một c' hoặc biến thường tuỳ thuộc vào nhà phát triển, nhưng đôi khi bạn phải theo dõi kỹ khi nào một biến nằm bên ngoài phạm vị trong những ứng dụng riêng của chính bạn.

Khi thực thi các lớp và đối tượng, scope thường được xử lý tự động. Một đối tượng bắt đầu hiện hữu khi constructor của nó được gọi ra và ngừng hiện hữu khi destructor của nó được gọi ra. Ví dụ, giả sử một đối tượng của lớp Foo được tạo trên ngăn xếp bằng một lệnh như sau:

```
Foo f;
```

Đối tượng Foo sẽ được tự động huỷ khi scope mà nó hiện hữu trong đó bị huỷ. Xem xét ví dụ sau đây với ba instance khác nhau của lớp Foo được tạo trên heap:

```
Foo global_foo; //                          → 1
int main()
{
    Foo main_foo; //                        → 2
    for ( int i=0; i<10; ++i )
    {
        Foo loop_foo; //                    → 3
    } //
} //
//
```

Như bạn có thể thấy, ba đối tượng Foo khác nhau được tạo tại ba thời điểm khác nhau trong mã ứng dụng. Đối tượng thứ nhất, đối tượng global_foo (được biểu thị tại → 1) được tạo trước khi ứng dụng thậm chí khởi động, trong hàm main.

Đối tượng thứ hai main_foo, được biểu thị tại (→ 2) được tạo ở đầu hàm main và đối tượng thứ ba loop_foo được biểu thị tại (→ 3) hiện hữu trong vòng lặp for ở giữa hàm main. Đối tượng loop_foo được tạo mỗi lần chương trình duyệt qua vòng lặp for - tất cả 10 lần.

Nếu chúng ta thực thi lớp Foo để cho chúng ta biết khi nào đối tượng được tạo và mỗi đối tượng được huỷ trên dòng nào, chúng ta thực sự có thể xem thông tin này bằng mắt và có thể hiểu dòng chảy của tiến trình.

Phần tiếp theo sẽ trình bày một cách đơn giản để làm điều đó.

Minh hoạ Scope

Nhận dạng thời gian bắt đầu và dòng mã riêng biệt của một đối tượng trên màn hình là một cách để hiểu phạm vi (scope) của nó. Sau đây là cách làm cho điều đó xảy ra:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này file được đặt tên là ch16.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 16.1 vào file của bạn.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

   Mã này tạo một lớp rất đơn giản vốn tự ghi chép các tiến trình tạo và huỷ của nó. Sau đó, chúng ta sẽ sử dụng lớp này để xem thứ tự các đối tượng được tạo và được huỷ trong một xác lập chương trình thực tế.

Listing 16.1: Minh hoạ scope

```
#include <stdio.h>
#include <stdlib.h>
class Foo
{
private:
    long _Level;
public:
    Foo( long level )
    {
        printf("Creating foo object %ld\n",
        level );
        _Level = level;
    }
    ~Foo()
    {
```

```
        printf("Destroying foo object
        %ld\n", _Level );
    }
};
```

3. Thêm mã sau đây vào file mã nguồn, sử dụng code editor.

   Ở đây chúng ta chỉ việc tạo một driver thử nghiệm nhỏ để minh hoạ cách các đối tượng được tạo và được huỷ như thế nào. Bạn có thể dễ dàng đặt mã này trong một file riêng biệt, nhưng để đơn giản chúng ta chỉ việc thêm tất cả mã này vào cùng một file.

   Mã này minh hoạ các cấp phạm vi khác nhau global, local và loop mà một đối tượng có thể chiếm trong một ứng dụng.

```
Foo global_foo(0);
int main()
{
    Foo main_foo(1);
    for ( int i=0; i<3; ++i )
    {
        Foo loop_foo(2);
    }
}
```

4. Lưu file và đóng code editor.

5. Biên dịch và chạy ứng dụng trong hệ điều hành mà bạn chọn.

Hiểu kết quả

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell trên hệ thống:

```
$ ./a.exe
Creating foo object 0
Creating foo object 1
Creating foo object 2
Destroying foo object 2
Creating foo object 2
Destroying foo object 2
Creating foo object 2
Destroying foo object 2
```

```
Destroying foo object 1
Destroying foo object 0
```

Chú ý thứ tự các biến được huỷ chính xác đảo ngược với thứ tự chúng được tạo. Đây là do những vị trí của đối tượng trong file: đối tượng sau cùng được tạo là đối tượng đầu tiên được huỷ. Việc chú ý đến khi nào các đối tượng được tạo và được huỷ trong chương trình sẽ giúp bạn tối ưu hoá việc sử dụng bộ nhớ và cũng cho phép bạn kiểm soát khi nào các đối tượng có sẵn. Ví dụ, nếu một đối tượng được sử dụng để tạo một file đòi hỏi một tên file, việc khởi tạo đối tượng trước khi tên file có sẵn không hợp lý thậm chí nếu phương thức open của đối tượng được sử dụng sau đó. Bây giờ hãy xem xét đoạn mã sau đây của một chương trình:

```
int main()
{
    Foo *f = new Foo(10);
    // Some other code
}
```

Đối tượng Foo đã được tạo khi nào? Rõ ràng constructor được gọi trên dòng bằng lệnh gọi hàm mới. Tuy nhiên, không rõ ràng khi nào đối tượng bị huỷ. Nếu bạn chạy chương trình, bạn thấy kết quả sau đây:

```
Creating foo object 10
```

Sẽ không có bản in tương ứng nói rằng đối tượng đã bị huỷ - bởi vì đối tượng không bao giờ bị huỷ. Vì lý do này bạn luôn nên rất cẩn thận khi tạo các đối tượng trên heap (sử dụng constructor và không có lệnh gọi new) hoặc kết hợp các lệnh gọi với new và delete cho đối tượng (nói cách khác, nếu bạn cấp phát một đối tượng bằng new, huỷ cấp phát nó bằng delete). Nếu bạn không xoá tất cả đối tượng mà bạn tạo, bạn sẽ tạo ra các rò rỉ bộ nhớ trong ứng dụng vốn có thể khiến cho chương trình bị đổ vỡ, làm giảm tốc độ máy tính và vô số hành vi không thể dự đoán trước khác.

Một lựa chọn thay thế cho phương pháp này là lớp auto_ptr của Standard Template Library. Lớp này chứa một pointer dẫn sang một lớp được ấn định, nhưng nó làm như vậy bên trong một đối tượng được cấp phát trên heap. Khi đối tượng trên heap nằm bên ngoài phạm vi, đối tượng bị huỷ. Là một phần của tiến trình huỷ đó, pointer bị xoá. Đây thật sự là việc sử dụng thực tiễn nhất khái niệm scope trong C++.

Kỹ thuật 17: Sử dụng các Namespace

Tiết kiệm thời gian bằng cách:

- Giải quyết những xung đột tên với các namespace
- Tạo một ứng dụng namespace
- Test ứng dụng namespace

Trước đây đã có một ngôn ngữ được gọi là C. Nó là một ngôn ngữ thông dụng cho phép người ta tạo những thư viện mã tốt, có thể tái sử dụng mà sau đó họ đã bán trong thị trường với số tiền lớn. Sau này ngôn ngữ này đã được thay thế bởi một ngôn ngữ được gọi là C++ vốn cũng đã cho phép các nhà lập trình tạo mã tái sử dụng. Học từ những vấn đề của chuyện trước đây, C++, không giống như C cũng đã cho phép bạn đóng gói mã trong các lớp để giải quyết một trong những vấn đề rất khó của C: giải quyết các hàm có các xung đột tên. Bởi vì có các hàm khác nhau trong các thư viện khác nhau vốn làm những điều tương tự thì phổ biến. Điều không thể tránh khỏi là tên của những hàm đó tương tự. Bằng cách giới hạn các phương thức đó chỉ trong các tên lớp khác nhau, điều đó giải quyết được vấn đề này. Dĩ nhiên, điều đó đã không giúp ích khi các tên lớp giống nhau, nhưng vấn đề đó sẽ được thảo luận một chút.

Sau đây là cách vấn đề này đã làm việc như thế nào: ví dụ xem xét hai thư viện vốn hiện hữu trong một giao diện kiểu C: một thư viện thực hiện các chức năng tạo cửa sổ; thư viện kia quản lý các tài liệu. Trong thư viện Một, chúng ta có hàm sau đây:

```
void SetTitle( char *sTitle )
{
    // Set the title of the window to sTitle
    window->title = sTitle;
}
```

Trong thư viện Hai, thư viện tài liệu, chúng ta có hàm sau đây:

```
void SetTitle( char *sTitle )
{
    // Set the file name for the document
    _filename = sTitle;
}
```

Bây giờ cả hai hàm này có cùng một tên và cùng một chữ ký, nhưng chúng có những mục đích và mã rất khác nhau để thực thi chúng.

Ở đây chúng ta có hai thường trình thực hiện cùng một điều cơ bản: (xác lập một tiêu đề) với mã khác nhau - nhưng đó không phải là vấn đề. Vấn đề là linker trong C (và cũng trong C++) không thể giải quyết tình huống này; nó trở nên bất thường nếu hai hàm có cùng một tên và cùng một chữ ký. Nó có thể xử lý điều đó nếu bạn chỉ muốn một trong chúng, nhưng rõ ràng điều đó sẽ không có tác dụng. Do đó với sự xuất hiện của các lớp (class), bạn nghĩ tất cả điều này sẽ được giải quyết đúng không? Không. Như với tất cả mọi thứ khác trong cuộc sống, các vấn đề không thật sự biến mất, chúng chỉ biến thành một dạng khác. Dĩ nhiên, chúng ta có thể giải quyết vấn đề này, nó chỉ là việc hiểu vấn đề đã xuất phát từ đâu lúc ban đầu. Trong trường hợp này, câu trả lời là tạo các lớp khác nhau để bao bọc các hàm.

Hướng nhanh một chút đến thế giới C++. Các nhà lập trình vẫn phát triển các thư viện (library), nhưng bây giờ đây là những thư viện lớp thay vì thư viện các hàm đơn giản. Vấn đề cơ bản về xung đột tên đã không thay đổi - nhưng các thư viện mẫu của chúng ta (bây giờ được cập nhật trong C++) đã thay đổi. Bây giờ thư viện tạo cửa sổ (windowing library) thực thi các lớp để giải quyết khái niệm model-view-controller đều mới, đều khác. Khái niệm này cho phép bạn không chỉ làm việc với các tài liệu mà còn với các khung xem (view) của những tài liệu đó.

Khái niệm về một document (tài liệu) nghĩa là dữ liệu cho một cửa sổ hiển thị được đóng gói trong một lớp vốn có thể tải dữ liệu từ một file, lưu nó sang một file và xử lý nó cho những mục đích hiển thị.

Đồng thời, so với trong thư viện tài liệu, chúng ta có các lớp tài liệu lưu trữ text, thông tin định dạng và các preference (tuỳ chọn theo sở thích) cho thông tin. Trong thư viện tài liệu, lớp Document được định nghĩa như sau:

```
class Document
{
private:
    std::string filename;
    std::vector< std::string > lines;
public:
    Document()
    {
    }
    Document( const char *_filename)
    {
        filename = _filename;
```

```
    }
    boolean Read();
    Boolean Write();
};
```

Tuy nhiên, trong windowing library, lớp Document được định nghĩa hơi khác:

```
class Document
{
private:
    std::string title;
    std::vector< Window *> windows;
public:
    Document()
    {
    }
    Document( const char *_title );
    void CreateWindow();
};
```

Trong khi điều này không đơn giản như ví dụ hàm, vấn đề chính xác y như vậy. Linker sẽ cần giải quyết hai lớp khác nhau có cùng một tên. Điều này không thể được do đó nó sẽ than phiền về điều đó. Làm thế nào bạn giải quyết một xung đột như vậy? Câu trả lời nằm trong khái niệm về các namespace C++.

Các namespace đã được tạo trong C++ để giải quyết ngay vấn đề này. Trong khi bạn có thể có một lớp xung đột với một lớp khác, hoàn toàn không có khả năng toàn bộ thư viện lớp xung đột. Với tên thích hợp của một namespace, bạn có thể tránh toàn bộ vấn đề xung đột tên lớp.

Tạo một ứng dụng namespace

Sau đây là dạng cơ bản của việc định nghĩa một namespace:

```
namespace <name> {
    // some classes or definitions
};
```

Để minh hoạ tất cả điều này, hãy xem nhanh qua một số mã thật sự và sau đó khảo sát những tuỳ chọn mà bạn có thể sử dụng trong mã riêng của bạn để giải quyết những xung đột tên. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này file được đặt tên là ch17.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 17.1 vào file.

Listing 17.1 Sử dụng các namespace

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
using namespace std;
namespace foo_windowing
{
    class window
    {
    private:
        int window_id;
      string title;
    public:
      window(void)
      {
        window_id = 0;
        title = "";
      }
      window(int id, const char *t)
      {
        window_id = id;
        title = t;
      }
    };
    class document
    {
    private:
      int doc_id;
      string title;
    public:
```

```
        document(void)
        {
           doc_id = 0;
           title = "";
        }
        document(int id, const char *t)
        {
           doc_id = id;
           title = t;
        }
    };
}; // End of namespace foo_windowing
```

Trước khi đi đến bước kế tiếp, hãy xem những đặc điểm của mã này mà bạn có thể điều chỉnh theo các chương trình riêng của bạn.

Đầu tiên, chú ý câu lệnh using namespace ở phần trên cùng - bạn thường phải xác định đầy đủ tên của tất cả lớp trong bất kỳ namespace mà bạn sử dụng. Lớp string cho Standard Template Library ngẫu nhiên "sống" trong namespace std. Khi bạn sử dụng lệnh using namespace std, bạn cho trình biên dịch biết rằng bạn biết những gì bạn đang làm và rằng nó nên thử namespace std:: trên bất cứ những gì mà nó không thể nhận ra ngay tức thì. Dĩ nhiên, nếu bạn sử dụng tất cả namespace có sẵn, bạn tránh vấn đề nhập tiền tố trên namespace trên tất cả tên lớp - nhưng sau đó bạn cũng gặp phải lại vấn đề xung đột tên. Đó là lý do tại sao trình biên dịch yêu cầu bạn xác định tên bất cứ khi nào nghi ngờ.

Bước kế tiếp xem kỹ hơn namespace thứ hai mà chúng ta tạo (trong trường hợp này, cho các lớp tài liệu).

3. Chỉnh sửa mã nguồn trong code editor như được minh hoạ trong Listing 17.2.

Chỉ việc thêm mã này vào cuối listing mã nguồn hiện hành.

Listing 17.2: Tạo một Namespace mới

```
namespace bar_documents
{
    class document
    {
    private:
       string filename;
```

```
public:
        document(void)
        {
           filename = "";
        }
        document(const char *fn)
        {
           filename = fn;
        }
    };
        string name()
        {
           return filename;
        }
}; // End of namespace bar_documents
```

4. Lưu mã nguồn trong bộ soạn thảo mã nguồn.

Như bạn có thể thấy chúng ta có một xung đột tên rõ ràng nếu chúng ta sử dụng hai tập hợp lớp này cùng với nhau. Chúng đều định nghĩa một lớp được gọi là document. Tệ hơn, lớp document trong một tập hợp lớp rất khác với lớp document trong tập hợp lớp kia. Trong trường hợp đầu tiên, lớp document tượng trưng cho dữ liệu được hiển thị trong một cửa sổ. Trong trường hợp thứ hai, lớp document tượng trưng cho một file trên đĩa. Thật may thay, bằng cách đặt từng lớp trong một namespace khác, chúng ta có thể phân biệt giữa hai lớp một cách dễ dàng sao cho trình biên dịch biết những gì chúng ta đang nói đến. Một ứng dụng thực đưa ra một ví dụ trong phần kế tiếp.

Listing 17.3: Sử dụng các namespace trong một ứng dụng

```
// Let's default to the bar_documents namespace
using namespace bar_documents;
void print_doc( document& d )
{
    printf("Received file: %s\n", d.name().c_str() );
}
int main( int argc, char **argv )
{
    document d("file1.txt"); // Create a bar_documents document
```

```
    foo_windowing::document d1(1, "this is a title");
    // This is okay to do
    print_doc( d );
    // This would not be okay to do
    //print_doc( d1 );                                    → 1
}
```

Test ứng dụng namespace

Danh sách ở đây hướng dẫn các bước để tạo một driver thử nghiệm vốn hiệu lực hoá các loại đầu vào khác nhau từ người dùng và minh hoạ cách các namespace được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch17.cpp.

2. Gõ nhập mã từ Listing 17.3 vào file của bạn.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

```
$ ./a.exe
Received file: file1.txt
```

Ở đây chúng ta chèn một dòng để làm cho namespace bar_documents trở nên global - nghĩa là chúng ta không cần phải nói bar_documents::document ở mọi nơi chúng ta sử dụng lớp. Thực tế, (như bạn có thể thấy trong dòng đầu tiên của chương trình chính), chúng ta chỉ việc tạo một tài liệu thông qua một cách sử dụng chuẩn của lớp Documents. Tuy nhiên, khi chúng ta muốn sử dụng các lớp namespace foo_windowing, chúng ta vẫn phải xác định đầy đủ chúng như bạn có thể thấy trong dòng thứ hai của chương trình chính. Tương tự chúng ta có thể chuyển một tài liệu bar_documents đến hàm print_doc, nhưng chúng ta không thể làm điều tương tự này với một tài liệu foo_windowing. Nếu bạn không chú giải dòng vốn gọi print_doc với đối tượng tài liệu foo_windowing, bạn sẽ vẫn nhận được lỗi thời gian biên dịch được minh hoạ trong Listing 17.4.

3. Lưu file mã nguồn và đóng code editor.

4. Biên dịch file bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Listing 17.4: Cố sử dụng một lớp namespace khác.

```
ch3_5.cpp: In function 'int main(int, char**)':
ch3_5.cpp:74: error: no matching function for call to 'foo_windowing::document
    ::document(const char[16])'
ch3_5.cpp:28: error: candidates are: foo_windowing::document::document(const
    foo_windowing::document&)
ch3_5.cpp:39: error: foo_windowing::document::document(int,
    const char*)
ch3_5.cpp:34: error: foo_windowing::document::document()
```

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ shell:

```
$ ./a.exe
Received file: file 1.txt
```

Như bạn có thể thấy từ kết quả, hàm mà chúng ta mong đợi làm việc với một đối tượng tài liệu namespace bar_documents hoạt động tốt. Nếu chúng ta không chú giải dòng được đánh dấu bằng 1, chúng ta sẽ nhận được một lỗi biên dịch vì hàm đó không mong đợi nhận một đối tượng tài liệu từ một namespace khác. Điều này được minh hoạ trong Listing 17.4.

Kỹ thuật 18: Sửa chữa các đoạn gãy bằng các cast

Tiết kiệm thời gian bằng cách

- Định nghĩa các cast
- Tìm hiểu các vấn đề mà các cast có thể gây ra
- Giải quyết các vấn đề biên dịch liên quan đến cast
- Test mã cải tiến của bạn

Khi bạn gãy xương, bác sĩ đặt nó trong một khuôn (cast) để bảo đảm xương đặt vào đúng vị trí và giữ được đúng hình dạng. C++ có cùng một khái niệm về các cast - và chúng được sử dụng cho chính xác những lý do tương tự đó. Một cast trong C++ thay đổi một cách tường minh một biến hoặc một giá trị từ một kiểu này sang một kiểu khác. Theo cách này, chúng ta "sửa chữa" mọi thứ bằng cách làm cho chúng có đúng kiểu cho những gì mà chúng ta cần những giá trị đó.

Nhu cầu về các cast xuất phát từ bản chất cầu kỳ của trình biên dịch C++. Nếu bạn cho nó biết rằng bạn đang mong đợi một hàm nào đó để lấy một giá trị số nguyên, nó sẽ than phiền nếu nó được cho bất

cứ thứ gì ngoại trừ một giá trị số nguyên. Nó có thể than phiền (ví dụ) bằng cách cảnh báo cho bạn rằng thiếu tính chính xác như sau:

```
int func (int x);
long x = 100;
func (x);
```

Ở đây trình biên dịch phàn nàn về việc gọi ra lệnh gọi hàm này, nói rằng việc chuyển đổi một số nguyên long thành một số nguyên bình thường thì hợp lệ, nhưng bạn gặp rủi ro có thể mất đi tính chính xác. Trình biên dịch cũng có thể đúng. Ví dụ, giá trị của x là 50.000 thay vì 100. Số đó quá lớn để được lưu trữ trong một số nguyên bình thường, do đó nó overflow (chèn) và hàm được chuyển một số nguyên để sử dụng. Có tin điều đó hay không. Hãy thử đoạn lập trình nhỏ sau đây:

```
#include <stdio.h>
int func(short int x)
{
    printf("x = %d\n", x );
}
int main(int argc, char **argv)
{
    long x = 10;
    func(x);
    x = 50000;
    func(x);
    return 0;
}
```

Ở đây, bởi vì trình biên dịch gcc giả định các số nguyên int và long là cùng một thứ, chúng ta phải sử dụng một short int trong chính hàm cho gcc để minh hoạ vấn đề.

Khi bạn biên dịch và chạy chương trình này, bạn thấy kết quả như sau:

```
$ ./ a.exe
x = 10
x = -15536
```

Nếu bạn biết giá trị của x sẽ không bao giờ vượt quá giá trị tối đa của một số nguyên, bạn có thể an toàn gọi hàm thậm chí với một số nguyên long miễn là bạn cast nó một cách thích hợp:

```
func( (short int) x );
```

Lý do bạn có thể muốn làm một điều nào đó như vậy là bạn không muốn tạo một biến mới, gán giá trị cho nó và sau đó kiểm tra để bảo đảm nó đã không overflow dãy của kiểu biến mới đó. Cast làm tất cả điều này cho bạn.

Dĩ nhiên, việc bao bọc số nguyên trong một cast mạnh hơn đáng kể so với đơn giản làm cho một số nguyên trở thành một long hoặc double hoặc bất kỳ giá trị khác. Việc cast một kiểu sang một kiểu khác sẽ biến đổi biến gốc thành một biến mới mặc dù trong hậu cảnh. Điều này đúng cho dù chúng ta cast một biến trong ví dụ ở trên hoặc chỉnh sửa một biến bằng cách cast nó vào một kiểu khác.

Phần tiếp theo sẽ trình bày một ví dụ phức tạp hơn về một cast.

Sử dụng các cast

Giả sử bạn có hai lớp cơ sở, Base 1 và Base 2. Từ hai lớp này, chúng ta dẫn xuất một lớp thứ ba được gọi là Derived. Làm thế nào bạn có thể đạt được các hàm trong các lớp cơ sở vốn có cùng một tên trong cả hai lớp cơ sở? Câu trả lời là trong một cast như chúng ta sẽ thấy ở kỹ thuật ví dụ kế tiếp. Dường như chúng ta quay trở cuộc thảo luận về việc phân biệt tên và các namespace, nhưng đây khôngphải là trường hợp ở đây. Hãy xem xét sơ đồ sau đây:

Lớp 1: Phương thức A

Lớp 2: Phương thức A

Lớp 3: Dẫn xuất từ lớp 1 và 2

Khi chúng ta sử dụng lớp 3 và tham chiếu phương thức A, chúng ta thực sự muốn nói đến phương thức nào?

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch18.cpp, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 18.1 vào file.

Listing 18.1: Sử dụng các cast

```
#include <stdio.h>
#include <string>
using namespace std;
class Base1
{
private:
```

```
        string _name;
        long _value;
    public:
        Base1()
        {
            _name = "";
        }
        Base1( const char *n, long v)
        {
            _name = n;
            _value = v;
        }
        virtual ~Base1()
        {
        }
        string GetName()
        {
            return _name;
        }
        long GetValue()
        {
            return _value;
        }
        void SetName( const char *sName )
        {
            _name = sName;
        }
        void SetValue( long l )
        {
            _value = l;
        }
    };
    class Base2
    {
    private:
```

```
        string _fileName;
        long _fileLength;
    public:
        Base2()
        {
            _fileName = "";
            _fileLength = 0;
        }
        Base2( const char *n )
        {
            _fileName = n;
            _fileLength = 0;
        }
        string GetName()
        {
            return _fileName;
        }
        long GetLength()
        {
            return _fileLength;
        }
        void SetName( const char *sName )
        {
            _fileName = sName;
        }
        void SetFileLength( long l )
        {
            _fileLength = l;
        }
};
```

3. Lưu file mã nguồn.

Mã trong listing này đơn giản định nghĩa hai lớp vốn ngẫu nhiên chia sẻ chung một hoặc hai tên phương thức. Dĩ nhiên đây không phải là vấn đề bởi vì chúng được định nghĩa trong các phạm vi (scope) khác nhau - nghĩa là chúng có thể được điều hoà bởi trình biên dịch là

linker thành các entry khác nhau trong ứng dụng. Nói cách khác, không có vấn đề gì cả. Bây giờ hãy tạo một lớp.

4. Mở lại file mã nguồn bằng code editor và thêm mã từ Listing 18.2 vào file.

Listing 18.2: Lớp dẫn xuất

```
class Derived :public Base1, public Base2
{
public:
    Derived( void )
        : Base1( "AClass", 10 ),
        Base2( "Derived" )
    {
    }
    Derived( const char *name)
        : Base1( name, 0 ),
        Base2( "Derived" )
    {
    }
    void ADerivedMethod()
    {
        printf("In a derived method\n");
    }
};
```

5. Lưu file mã nguồn.

Mã này minh hoạ một lớp dẫn xuất vốn được tạo từ hai lớp cơ sở. Trong trường hợp này, cả hai lớp cơ sở chứa một phương thức có cùng một tên.

Chúng ta cần tìm một cách để đạt được phương thức riêng biệt trong lớp cơ sở mà chúng ta muốn, điều này chỉ có thể được thực hiện bằng các cast. Hãy xem điều đó bây giờ.

Nếu bạn biên dịch và liên kết chương trình này, nó làm việc tốt - ngoại trừ thiếu hàm chính của nó. Trình biên dịch sẽ không than phiền về các lớp cơ sở (Base), cũng không lớp dẫn xuất. Không có vấn đề gì với việc dẫn xuất từ hai lớp cơ sở vốn ngẫu nhiên chia sẻ chung một tên phương thức.

Vấn đề xảy ra khi chúng ta cố sử dụng một phương thức (method) từ các lớp cơ sở qua lớp dẫn xuất.

6. Thêm mã từ Listing 18.3 vào file để thực thi một driver thử nghiệm cho mã.

Listing 18.3 Driver thử nghiệm lớp dẫn xuất.

```
int main()
{
    Derived d;
    d.SetFileLength(100);
    d.SetName( "This is a name" );
    string s = d.GetName();
    return 0;
}
```

7. Lưu file mã nguồn và đóng code editor.
8. Biên dịch chương trình bằng cách sử dụng trình biên dịch mã nguồn ưa thích trên hệ điều hành ưa thích.

Bạn sẽ thấy kết quả từ trình biên dịch giống như kết quả của Listing 18.4.

Listing 18.4: Kết quả mẫu

```
$ gcc ch3_8.cpp -lstdc++
ch3_8.cpp: In function 'int main()':
ch3_8.cpp:102: error: request for member
    'SetName' is ambiguous
ch3_8.cpp:68: error: candidates are: void
    Base2::SetName(const char*)
ch3_8.cpp:34: error: void
    Base1::SetName(const char*)
ch3_8.cpp:103: error: request for member
    'GetName' is ambiguous
ch3_8.cpp:60: error: candidates are:
    std::string Base2::GetName()
ch3_8.cpp:26: error:
    std::string Base1::GetName()
```

Giải quyết các vấn đề trình biên dịch

Bây giờ chúng ta có một lỗi - vì vậy (dĩ nhiên) câu hỏi là làm thế nào bạn tiếp cận các phương thức lớp Base khi chúng xung đột? Thật ra, bạn có hai cách để làm điều này: định phạm vi (scope) một cách

tường minh hàm thành viên mà bạn muốn sử dụng, hoặc cast cụ thể đối tượng sao cho nó có kiểu mà bạn muốn nó thể hiện. Phần này sẽ hướng dẫn bạn cách thực hiện cả hai phương thức và thảo luận kết quả của mỗi phương thức.

1. Mở lại file mã nguồn từ ví dụ (nó được gọi là ch18.cpp) và thêm mã từ Listing 18.5.

Listing 18.5: Mã driver thử nghiệm được chỉnh sửa.

```
int main()
{
    Derived d;
    d.SetFileLength(100);
    /* 1 */ ((Base1)d).SetName( "This is a
       name" );
    /* 2 */ string s = d.Base1::GetName();
       return 0;
}
```

Hai lựa chọn được ghi nhãn bằng các chú giải giữa các dấu sao dưới dạng các khối 1 và 2. Dòng được ghi nhãn 1 là cast tường minh; dòng được ghi nhãn 2 là lệnh gọi phương thức được định phạm vi. Có lẽ bạn thấy rằng phương thức thứ hai hơi dễ đọc hơn, nhưng cả hai sẽ làm việc tốt. Sự khác biệt giữa chúng là các dòng được hiểu bởi trình biên dịch như thế nào. Khi bạn cast một đối tượng, đối với trình biên dịch, bạn đang thay đổi kiểu của nó. Khi bạn gọi phương thức SetName với cast của đối tượng, phương thức SetName được gọi với một đối tượng Base 1 thay vì một đối tượng Derived. Điều này có nghĩa gì?

Việc chỉnh sửa các lớp một chút sẽ đưa ra một minh hoạ tiện lợi, bắt đầu với bước kế tiếp.

2. Thay thế các định nghĩa gốc của Base 1 và Base 2 bằng mã sau đây được minh hoạ trong Listing 18.6.

Listing 18.6: Các listing lớp cơ sở mới.

```
class Base1
{
private:
    string _name;
    long _value;
    virtual void PrintNameChange()
    {
```

```
        printf("Changed name to %s\n"
        _name.c_str() );
    }
public:
    Base1()
    {
        _name = "";
    }
    Base1( const char *n, long v)
    {
        _name = n;
        _value = v;
    }
    virtual ~Base1()
    {
    }
    string GetName()
    {
        return _name;
    }
    long GetValue()
    {
        return _value;
    }
    void SetName( const char *sName )
    {
        _name = sName;
        PrintNameChange();
    }
    void SetValue( long l )
    {
        _value = l;
    }
};
class Derived :public Base1, public Base2
```

```
    {
        virtual void PrintNameChange()
        {
            printf("Derived: PrintNameChange
            called\n");
        }
    public:
        Derived( void )
            : Base1( "AClass", 10 ),
            Base2( "Derived" )
        {
        }
        Derived( const char *name)
            : Base1( name, 0 ),
            Base2( "Derived" )
        {
        }
        void ADerivedMethod()
        {
            printf("In a derived method\n");
        }
};
```

Test các thay đổi

Sau khi bạn đã thực hiện tất cả thay đổi trong các lớp Base cho những đối tượng mà bạn sử dụng, bước kế tiếp là test những thay đổi đó để bảo đảm trình biên dịch hài lòng và mã chạy tốt. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, mở lại file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch18.cpp.

2. Gõ nhập mã sau đây vào file như được minh hoạ trong Listing 18.7.

Listing 18.7: Driver thử nghiệm mới.

```
int main()
```

```
{
    Derived d;
    d.SetFileLength(100);
((Base1)d).SetName( "This is a name" );                    →1
    string s = d.Base1::GetName();
    d.Base1::SetName("This is another name");
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng ứng dụng editor.
4. Biên dịch và liên kết ứng dụng trên hệ điều hành ưa thích, sử dụng trình biên dịch mà bạn chọn.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây xuất hiện trên cửa sổ shell.

```
$ ./a.exe
Changes name to This is a name
Derived: PrintNameChange called
```

Nếu bạn truy vết qua mã, bạn sẽ thấy những gì đang xảy ra ở đây:

Trong trường hợp thứ nhất, trong đó chúng ta cast đối tượng d (được minh hoạ tại →1) vào một đối tượng Base 1, đối tượng không "biết" rằng nó thực sự là một đối tượng Derived khi phương thức ảo PrintNameChange được gọi. Kết quả phương thức lớp Base được sử dụng cho trường hợp cast (được minh hoạ tại →2).

Đối với trường hợp thứ hai, trong đó chúng ta định phạm vi phương thức. Tuy nhiên, đối tượng biết rõ nó là gì và sẽ gọi phương thức ảo thừa kế trong lớp Derived. Đây là một sự khác biệt rất quan trọng và có thể dẫn đến một số lỗi lô gíc rất tinh vi trong chương trình rất khó tìm ra và sửa chữa. Các cast là một kỹ thuật rất mạnh mẽ trong C++, nhưng chúng cũng là một cảnh báo nghiêm trọng rằng bạn làm một điều gì đó mà bạn không nên làm. Nếu những hành động của bạn có thể chấp nhận được trong ngôn ngữ, bạn sẽ không cần phải cho trình biên dịch biết rõ ràng rằng bạn đang thay đổi hành vi của mã thông qua một cast. Đây không phải là một điều tệ, nhưng bạn cần chú ý rằng bạn đang thay đổi hành vi. Hãy bảo đảm hiểu những tác dụng phụ của các hành động của bạn và những khả năng gây ra nhiều vấn đề hơn bạn giải quyết khi sử dụng một cast.

Kỹ thuật 19: Sử dụng các Pointer dẫn đến các hàm thành viên

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các pointer hàm thành viên
- Thực thi các pointer hàm thành viên
- Cập nhật mã với các pointer hàm thành viên
- Test mã

Các pointer dẫn đến các hàm thành viên mạnh mẽ không thể tin được và là một phần không thể thiếu được của ngôn ngữ C++. Nếu bạn sử dụng chúng, mã sẽ dễ hiểu và dễ mở rộng và việc bảo trì sẽ nhanh hơn nhiều. Ngoài ra, chức năng mới có thể được bổ sung mà không cần chỉnh sửa các hàm hiện có trong lớp. Các pointer dẫn sang các hàm thành viên nam giúp thay thế các câu lệnh switch phức tạp, các lookup table (bảng dò tìm) và nhiều cấu trúc phức tạp khác bằng một giải pháp đơn giản, dễ thực thi.

Tuy nhiên, bởi vì chúng gây bối rối khi thực thi và có cú pháp phức tạp, hầu như không ai sẵn lòng sử dụng nhưng thứ kém cỏi. Tuy nhiên, pointer dẫn sang một hàm thành viên là một công cụ thật sự hữu dụng trong kho vũ khí những kỹ thuật của bạn để giải quyết các vấn đề với ngôn ngữ lập trình C++. Thật ra vấn đề là hiểu chúng hoạt động như thế nào và cách làm cho trình biên dịch hiểu những gì bạn muốn thực hiện với chúng.

Điều đầu tiên cần hiểu là một pointer dẫn sang một hàm thành viên chính xác là gì? Trong những ngày trước đây của lập trình C, chúng ta có các pointer hàm cũ. Một pointer hàm đã là một pointer dẫn sang một hàm toàn cục trong khi một pointer dẫn sang một hàm thành viên làm việc chỉ với một hàm thành viên lớp; chúng là cùng một thứ. Về cơ bản các pointer hàm đã cho phép bạn làm điều sau đây:

```
typedef int (*error_handler) (char * )
```

Câu lệnh này định nghĩa một pointer dẫn sang một hàm vốn chấp nhận một đối số đơn có kiểu character pointer và trả về một giá trị số nguyên. Sau đó bạn có thể gán một hàm vào pointer này như sau:

```
int my_error_handler(char *s)
{
printf("Error: %s\n", s );
return 0;
```

```
}

error_handler TheErrorHandler = my_error_handler;
```

Trong một thư viện (ví dụ), đây là một cách rất hữu dụng để xử lý các lỗi. Bạn cho phép mỗi người dùng xác lập phương thức xử lý lỗi riêng của mình thay vì đơn giản in chúng ra hoặc mở ra một hộp thoại lỗi hoặc ghi chép những thứ khó chịu sang một file. Theo cách này, những người dùng cuối của thư viện có thể bẫy các lỗi và giải quyết chúng - bằng cách thay đổi phần mô tả, in ra thông tin gỡ rối bổ sung hoặc bất cứ điều gì khác đã làm việc cho họ.

Trong mã thư viện, bạn gọi ra phương thức xử lý lỗi (error handler) bằng cách viết:

```
(*TheErrorHandler) (theErrorString);
```

rõ ràng kiểm tra xem TheErrorHandler đã thực sự được gán vào bất cứ thứ gì trước tiên hoặc đã là NULL hay không.

Đó là cách các pointer hàm được sử dụng trong C. Khi C++ xuất hiện, cùng một loại chức năng này đã rất hữu dụng cho nhiều tác vụ khác nhau và đã tiếp tục làm việc tốt với các hàm toàn cục (global functions). Tuy nhiên, không có cách đơn giản nào để thực thi chức năng này vì một hàm thành viên đã đòi hỏi một đối tượng thao tác trên nó, do đó bạn không thể chỉ gán nó vào một pointer ngẫu nhiên. Bạn có thể lưu trữ một pointer hàm thành viên bất cứ nơi nào. Tuy nhiên, khi sử dụng nó, bạn cần một đối tượng của lớp của thành viên đó để sử dụng nó. Ví dụ:

```
class Foo
{
    // A member function
    void bar(int x );
    // This defines a type
    typedef void (*ptr_to_member_function)
    (int x) ;
public:
    ptr_to_member_function p1;
    Foo ()
    {
    // Assign the member function pointer
    p1 = bar;
    }
}
```

```
// Later in the code....
Foo::p1(0); // This won't work, since the
            member function requires a
            pointer.
    Foo x;
    x.p1(0); // This will work.
```

Với sự xuất hiện của các pointer hàm thành viên, bạn có thể gán một hàm thành viên vào một pointer ngẫu nhiên nếu bạn đã không quan tâm một chút cú pháp khác lạ. Bạn có thể định nghĩa những pointer này là:

```
typedef <return-type> (<classname>::*
PtrMemberFunc)( <args>) ;
```

Thực thi các pointer hàm thành viên

Hãy xem một kỹ thuật sử dụng các pointer hàm thành viên để thực thi một bộ xử lý lệnh đơn giản.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã nguồn cho kỹ thuật này.

   Trong kỹ thuật này, chương trình thử nghiệm được đặt tên là ch19.cpp.

2. Gõ nhập mã từ Listing 19.1 vào file.

Listing 19.1: Các pointer dẫn sang các hàm thành viên

```
#include <stdio.h>
#include <string>
#include <vector>
class Processor
{
    typedef bool
    (Processor::*PtrMemberFunc)( std::string
    ) ;                                          → 1
private:
    std::vector< PtrMemberFunc >
    _functionList;
protected:
    virtual bool ProcessHello(std::string s)
    {
```

```
        if ( s == "Hello" )
        {
            printf("Well, hello to you
            too!\n");
            return true;
        }
        return false;
    }
    virtual bool ProcessGoodbye( std::
    string s)
    {
        if ( s == "Goodbye" )
        {
            printf("Goodbye. Have a great
            day!\n");
            return true;
        }
        return false;
    }
    virtual bool ProcessOpen( std::string s)
    {
        if ( s == "Open" )
        {
            printf("The door is now open\n");
            return true;
        }
        return false;
    }
public:
    Processor()
    {
        _functionList.insert(
        _functionList.end(),
&Processor::ProcessHello );
        _functionList.insert(
```

```
            _functionList.end(),
    &Processor::ProcessGoodbye );
            _functionList.insert(
            _functionList.end(),
    &Processor::ProcessOpen );
        }
        virtual ~Processor()
        {
        }
        void ProcessCommand( const std::string&
        command )
        {
            std::vector< PtrMemberFunc >::iterator
            iter;
            for ( iter = _functionList.begin();
            iter !=
            _functionList.end(); ++iter )
            {
                PtrMemberFunc ptr = (*iter);
                if ( (this->*ptr)( command ) )
                    return;
            }
            printf("Unknown command %s\n",
            command.c_str() );
        }
};
```

3. Lưu mã nguồn trong editor

Chú ý sau khi bạn đã định nghĩa một pointer hàm thành viên (xem → 1), bạn có thể sử dụng nó giống như cách bạn sử dụng một pointer "bình thường". Trong trường hợp cụ thể này, chúng ta xây dựng một mảng pointer và đơn giản móc nối qua chúng để xem một chuỗi nào đó có được xử lý hay không. Mã như vậy có thể dễ dàng được sử dụng cho các bộ phân tích phương trình, bộ xử lý lệnh hoặc bất cứ điều gì khác vốn đòi hỏi một danh sách các mục khác cần được hiệu lực hoá và được phân tích. Listing 19.1 chắc chắn sạch hơn và dễ mở rộng hơn nhiều so với mã như sau:

```
switch ( command )
{
    case "Hello":
        // Do something
        break;
    // .. other cases
    default:
            printf("Invalid command: %s\n",
            command.c_str() );
            break
}
```

Chú ý trong trường hợp này, chúng ta cần thêm một switch case (trường hợp chuyển đổi) mới mỗi lần chúng ta muốn xử lý một lệnh mới. Với mảng pointer hàm, sau khi nó được định nghĩa và được thêm vào mã, hàm thành viên (member function) làm tất cả công việc.

Cập nhật mã với các pointer hàm thành viên

Listing 19.1 không chỉ sạch hơn và dễ mở rộng hơn mà nó còn dễ mở rộng hơn nhiều bởi vì bạn có thể ghi đè bất kỳ chức năng nào bạn muốn tại cấp hàm thành viên.

Đặc tính đó đáng được xem kỹ hơn để thấy được kỹ thuật này hữu dụng như thế nào trong ứng dụng. Hãy tưởng tượng bạn đã thực thi đối tượng Processor này và vui vẻ sử dụng nó để xử lý dữ liệu nhập từ người dùng. Bây giờ đột ngột người khác muốn sử dụng cùng một lớp - nhưng họ muốn thực thi một cách sử dụng hoàn toàn khác cho lệnh Open. Tất cả lệnh khác làm việc theo cùng một cách. Sẽ thú vị hay không khi sử dụng cùng một lớp để làm công việc - và chỉ ghi đè hàm mà bạn muốn thay đổi? Hoá ra bạn có thể. Hãy nhớ một pointer dẫn sang một hàm thành viên đơn giản là một pointer dẫn sang bất cứ thứ gì sẽ được gọi khi phương thức cụ thể đó được gọi ra trên đối tượng. Nếu chúng ta ghi đè một phương thức ảo trong một lớp dẫn xuất, nó sẽ tự động gọi phương thức đó khi chúng ta sử dụng bộ xử lý mới. Các bước sau đây thử điều đó:

1. Thêm mã từ Listing 19.2 vào file mã nguồn.

Listing 19.2: Lớp bộ xử lý lệnh (command processor)

```
class Processor2 : public Processor
{
protected:
```

```
    virtual bool ProcessOpen( std::string s)
    {
        if ( s == "Open" )
        {
            printf("Derived processing of
            Open\n");
            return true;
        }
        return false;
    }
    public:
    Processor2()
    {
    }
};
```

2. Lưu mã nguồn trong editor và đóng ứng dụng code editor.

Test mã pointer thành viên

Sau khi tạo một pointer dẫn sang một thành viên cho một lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn.

Các bước sau đây hướng dẫn cách tạo một driver thử nghiệm để cho thấy lớp được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch19.cpp.

2. Gõ nhập mã từ Listing 19.3 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 19.3: Driver thử nghiệm cho bộ xử lý lệnh.

```
int main(int argc, char **argv )
{
    Processor2 p;
    for ( int i=1; i<argc; ++i )
       p.ProcessCommand( argv[i] );
}
```

3. Đóng file mã nguồn trong editor và đóng ứng dụng editor.
4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích và sau đó chạy nó bằng lệnh sau đây:

```
./a.exe Open Goodbye
```

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây khi bạn chạy chương trình với những đối số này:

```
Derived processing of Open
Goodbye. Have a great day
```

Như bạn có thể thấy chúng ta không chỉ tạo một phương thức xử lý lệnh để xử lý lệnh Open mà chúng ta cũng cho phép dẫn xuất C++ chuẩn. Hơn nữa chúng ta có thể dễ dàng bổ sung thêm các phương thức xử lý (handler) nhằm xử lý cùng một lệnh, điều mà chúng ta không thể làm với câu lệnh switch.

Kỹ thuật 20: Định nghĩa các đối số mặc định cho các hàm và phương thức

Tiết kiệm thời gian bằng cách:

- Đơn giản hoá mã với những đối số mặc định
- Thực thi các đối số mặc định trong các hàm và phương thức
- Tuỳ biến các hàm và phương thức tự định nghĩa
- Tuỳ biến và các hàm và phương thức mà người khác đã viết
- Test mã

Trong C++ việc gọi không đúng các hàm và phương thức là một vấn đề thường thấy. Một giải pháp là kiểm tra tất cả dữ liệu đầu vào khi người dùng cuối gởi chúng vào - nhưng điều này có tác dụng phụ không may mắn và tạo thêm lỗi để nhà phát triển cuối xử lý. Vấn đề là lọc bỏ các giá trị kém khó hơn là chấp nhận chỉ những giá trị tốt.

Ví dụ, đối với một giá trị số nguyên phải nằm giữa 1 và 10, có 10 giá trị hợp lệ. Tuy nhiên, có một số giá trị số nguyên kém vô hạn vốn hữu bên ngoài dãy 1 đến 10. Sẽ tuyệt với hay không nếu bạn có thể cho nhà phát triển biết giá trị thích hợp nhất cho một số đối số hàm ít phổ biến hơn là gì? Sau đó nhà lập trình có thể bỏ qua những giá trị có lỗi bằng cách sử dụng những giá trị mặc định chấp nhận được. Ví dụ, xem xét hàm MessageBox, một hàm chuẩn được sử dụng bởi nhiều nhà lập trình Microsoft Windows. Hàm này vốn có chữ ký sau đây, hiển thị một thông báo để người dùng ứng dụng thấy và phản hồi lại.

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

Các đối số MessageBox:

- **hWind:** Một handle dẫn sang một cửa sổ Windows
- **lpText:** Text để hiển thị trong hộp thông báo
- **lpCaption:** Chú thích để đặt trong thanh tiêu đề của hộp thông báo
- **uType:** Các kiểu nút (OK, Cancel, About, Retry...) để hiển thị trong hộp thông báo

Đây là một hàm rất tiện lợi được sử dụng hầu như rất phổ biến bởi các nhà lập trình Windows để hiển thị các lỗi, cảnh báo và thông báo. Vấn đề là quá dễ quên thứ tự của các đối số. Ngoài ra các nhà lập trình hay sử dụng đối tượng nào đó lặp đi lặp lại nhiều lần bằng những cách như nhau. Điều này có nghĩa là cùng một mã được lặp đi lặp lại nhiều lần và các thay đổi phải được thực hiện khắp hệ thống khi mong muốn một cách mới để thực hiện mọi thứ. Do đó, khả năng tùy biến hàm MessageBox với một số giá trị mặc định sẽ tốt, do đó chúng ta chỉ việc gọi nó theo cách mà chúng ta muốn. Chúng ta chỉ xác định những giá trị vốn thay đổi nhằm giới hạn số đối số nhập vào, làm cho dễ nhớ thứ tự và dễ tránh các giá trị lỗi hơn.

Tuỳ biến các hàm mà người khác đã viết

Có lẽ một công dụng của hàm MessageBox là hiển thị thông báo lỗi. Bởi vì người dùng cuối có thể không làm điều gì với một lỗi như vậy, không có lý do nào để hiển thị nút Cancel trên hộp thông báo - mặc dù hầu hết ứng dụng làm ngay điều đó. Phần này hướng dẫn bạn cách tạo sự thay đổi riêng của bạn trên hàm MessageBox - hàm

ErrorBox - vốn khác với MessageBox chỉ ở điểm nó đặt từ "Error" ở đầu thanh tiêu đề hiển thị và nó hiển thị chỉ một nút OK có test. Không có lý do thật sự để tạo bất kỳ chức năng mới cho hàm này, bởi vì sau cùng, hàm MessageBox đã làm mọi thứ mà chúng ta muốn. Hàm của chúng ta trông giống như trong Listing 20.1.

Listing 20.1: Một hàm MessageBox được tuỳ biến

```
int ErrorBox( HWND hWnd, const char *text,
    const char *title = "Error", UINT type =
    MB_OK )
{
    MessageBox(hWnd, text, title, type );
}
```

Chúng ta đã không thêm nhiều giá trị ở đây, nhưng hãy xem hàm bây giờ được gọi trong một mã ứng dụng như thế nào:

```
// Display an error
ErrorBox( NULL, "You have to first enter a
    file name!");
```

Chắc chắn mã này dễ hiểu hơn lệnh gọi MessageBox hoàn chỉnh nhiều. Dĩ nhiên, là bạn không cần phải sử dụng phiên bản rút ngắn - bạn vẫn có thể sử dụng toàn bộ như sau:

```
ErrorBox(m_hWnd, "The system is very low
    on memory! Retry or Cancel?" "Critical
    Error", MB_RETRY | MB_CANCEL );
```

Phiên bản rút ngắn chắn chắn dễ đọc và nhất quán hơn và nó tiết kiệm cho bạn thời gian bởi vì nó đưa ra ít tham số hơn để cập nhật. Ví dụ, nếu ban quản lý quyết định rằng việc diễn đạt lỗi của bạn quá khó chịu và muốn thay thế nó bằng A problem has occurred thì phải làm sao? Nếu bạn sử dụng phiên bản dài của hàm này, cách để giải quyết vấn đề này là tìm tất cả lệnh gọi đến hàm trong mã của bạn. Tuy nhiên, với hàm bao bọc (wrapper function), chúng ta có thể loại bỏ lỗi trong chính lệnh gọi và đặt nó vào hàm bao bọc. Ví dụ, tất cả lỗi có thể bắt đầu với ""An error has occurred" và sau đó thêm text lỗi thực sự. Dĩ nhiên để làm cho mọi thứ thậm chí dễ dàng hơn, bạn có thể đi xa hơn một bước và cho phép hàm đọc các chuỗi dữ liệu từ một file bên ngoài - khả năng đó cũng cho phép sự quốc tế hoá.

Tuỳ biến các hàm mà bạn tự viết

Dĩ nhiên, việc đơn giản tuỳ biến mã của người khác không luôn luôn là phương thức tốt nhất khi thêm những đối số mặc định. Các đối

số mặc định là những đối số mà nhà phát triển hàm cung cấp. Nếu bạn không muốn xác định một giá trị ngoại trừ giá trị mặc định, hãy bỏ qua hoàn toàn đối số. Nếu bạn muốn thay đổi giá trị mặc định cho một sự gọi ra riêng biệt, bạn có thể làm như vậy. Điểm chính thực sự cho thói quen này là cung cấp những công dụng thích hợp nhất của một đối số nào đó trong khi bảo vệ khả năng người dùng thay đổi (hoặc tuỳ biến) các giá trị mặc định của bạn.

Kỹ thuật này tạo, trong một lớp, các hàm và phương thức vốn cho người dùng hoàn toàn kiểm soát đầu vào của mình - trong khi cung cấp những giá trị mặc định thích hợp nhằm cho phép sử dụng các phương thức và hàm của lớp để hoàn thành những thao tác thông thường.

Listing 20.2: Một lớp với các phương thức chứa những giá trị mặc định

```
#include <stdio.h>
#include <string>
class FileHandler
{
private:
    std::string m_fileName;
    FILE *m_fp;
    static const char *fileName()
    {
        return "log.txt";
    }
public:
    FileHandler( const char *fn = NULL )
    {
        if ( fn )
            m_fileName = fn;
        else
            m_fileName = fileName();
}
    int open( const char *name = fileName(), const char *mode = "rw" ) → 1
    {
        m_fileName = name;
        return open(mode);
```

```
    }
    int open( const std::string& mode ) → 2
    {
        m_fp = fopen( m_fileName.c_str(), mode.c_str() );
        if ( m_fp == NULL )
            return -1;
        return 0;
    }
};
```

Một ví dụ là một thao tác mà hầu hết các nhà lập trình thường làm: tạo một file. Các bước sau đây trình bày cách tạo một lớp File rất đơn giản vốn cho phép mở, đọc, ghi và đóng các file:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch20.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 20.2 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

3. Lưu file.

Trong constructor, chúng ta xác lập tên file mặc định dưới dạng một pointer NULL. Nếu người dùng ghi đè tên file, chúng ta sử dụng tên mà họ đề nghị. Nếu không, chúng ta chỉ việc sử dụng tên trong được trả về bởi phương thức fileName trong lớp.

Đối với phương thức oepn đầu tiên (1), chúng ta cũng cho phép người dùng ghi đè tên file - nhưng lần này chúng ta trực tiếp khởi tạo tên file từ phương thức trong (fileName( )). Giá trị mặc định này cho người dùng gọi hàm open đầu tiên không có các đối số nếu họ chọn.

Làm việc với các phương thức (không static) thông thường

Chú ý rằng cách gọi một phương thức dưới dạng một giá trị mặc định chỉ có tác dụng nếu phương thức đang đề cập là static. Bạn không thể làm điều này với một phương thức thông thường, bởi vì các phương thức thông thường đòi hỏi một đối tượng - và cách gọi này đặt bạn bên ngoài phạm vi của đối tượng mà bạn đang sử dụng.

Ví dụ, chúng ta không thể làm một điều gì đó như sau:

```
virtual const char *filename()
{
```

```
        return "log.txt";
    }
```

và sử dụng nó theo cách sau đây:

```
int open( const char *name = fileName( ),
const char *mode)
```

Trình biên dịch sẽ trở nên bực bội bởi vì phương thức filename đòi hỏi một pointer this để thao tác. Cấp độ mà chúng ta đang làm việc không có pointer this này. Tệ hơn bạn không thể viết một lệnh như sau:

```
int open( const char *name = this - >
filename, const char *mode)
```

Lý do bạn không thể thì đơn giản: pointer this không hiểu trong trường hợp này. Bạn không ở bên trong một đối tượng do đó bạn không thể yêu cầu thế giới bên ngoài tham chiếu đối tượng mà bạn đang ở trong đó. Đây là một sự phiền toái, nhưng đó là một trong những điều mà bạn phải quen. Nếu bạn thấy điều này quá phiền toái, hãy sử dụng cùng một kỹ thuật mà constructor sử dụng để gọi phương thức.

Sau cùng chúng ta có phương thức open thứ hai vốn có thể mở file - xác định chỉ chế độ mà chúng ta muốn. Chú ý rằng chúng ta không thể mặc định phương thức này. Tại sao? Nếu phương thức đã có một đối số mặc định, không có cách nào biết người dùng đã có muốn gọi phiên bản thứ nhất hoặc thứ hai của phương thức open hay không. Để minh hoạ, hãy xem xét chuỗi lệnh gọi sau đây trong một chương trình ứng dụng:

```
FileHandler fh; // Invokes the constructor
fh. open ( );
```

Bây giờ, nếu chúng ta cho biến thể thứ hai (# 2) của phương thức open một đối số mặc định, phiên bản nào của phương thức open sẽ được gọi? Nó có thể là:

```
fh. open (name, mode);
```

hoặc nó có thể là

```
fh. open (mode);
```

Trình biên dịch không có cách nào biết phương thức nào mà lúc đầu nhà phát triển đã ấn định, do đó nó không cho phép sử dụng kỹ thuật này vào thời gian biên dịch.

Dĩ nhiên, chúng ta cần thêm một phương thức để ghi sang file. Không có cách nào để mặc định một cách hợp lý các giá trị mà chúng

ta muốn ghi ra - điều đó hoàn toàn tuỳ thuộc vào người dùng cuối của lớp - do đó chúng ta không cần làm như vậy. Đối với một câu lệnh write, làm thế nào bạn biết dữ liệu nào mà người dùng cuối muốn ghi? Bạn không thể, cũng không có bất kỳ loại mẫu nào cho nó.

Trong code editor, chèn mã được minh hoạ trong Listing 20.3 vào mã nguồn của chương trình (nó nằm trước dấu ngoặc đóng của phần public của lớp).

Listing 20.3: Phương thức ghi file

```
bool write( const char *string )
{
    bool bRet = false;
    if ( m_fp )
    {
        fputs( string, m_fp )
        bRet = true;
    }
    return bRet;
}
```

Test mã mặc định

Bạn nên test lớp với các giá trị mặc định và không mặc định để xem những giả định của bạn có hợp lệ hay không.

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm để cho bạn thấy lớp được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch20.cpp.

2. Gõ nhập mã từ Listing 20.4 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 20.4: Driver thử nghiệm cho lớp FileHandler

```
int main(int argc, char **argv)
{
    FileHandler fh;                                    → 3
    if ( fh.open() == -1 )
```

```
        printf("Unable to open file. Errno
        %d\n", errno);
    fh.write("This is a test");
    FileHandler fh2("log2.txt");                    → 4
    fh.open("w");
    fh.write("This is another test");
}
```

3. Lưu mã nguồn và đóng bộ soạn thảo mã nguồn.
4. Biên dịch mã bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

   Nếu bạn đã làm đúng mọi thứ, chương trình sẽ tạo các file log.txt và log 2.txt. Những file này sẽ được tạo tại → 3 và → 4 trong mã driver. Các file sẽ chứa kết quả cho những đối tượng FileHandler. Tuy nhiên, bạn sẽ thấy nhanh rằng thực tế nó đã không tạo bất kỳ file nào. Phụ thuộc vào hệ điều hành và trình biên dịch, thậm chí bạn có thể làm cho một chương trình bị đổ vỡ. Vậy thì điều trục trặc gì đã xảy ra?

Giải quyết vấn đề

Nếu bạn đặt một số câu lệnh gỡ rối trong phương thức open với hai đối số, bạn có thể tìm thấy rất nhanh vấn đề: phương thức open được gọi một cách để qui cho đến khi ngăn xếp kiệt quệ và chương trình đổ vỡ. Tại sao? Do bản chất của các đối số mặc định, hàm oepn có các đối số mặc định gọi lại oepn trong chính nó sau khi đã gán các đối số. Sự kết hợp tốt nhất cho lệnh gọi open trong phương thức open ngẫu nhiên là chính phương thức. Điều này gây ra một vòng lặp đệ qui vô hạn. Do đó dĩ nhiên những điều tồi tệ xảy ra khi bạn tự gọi liên tục trong một phương thức. Hãy sửa chữa điều đó bằng cách chỉnh sửa phương thức open như trong Listing 20.5 (thay thế phương thức hiện có bằng mã listing mới).

Listing 20.5: Phương thức open được chỉnh sửa

```
int open( const char *name =
fileName(),const char *mode = "rw+")
{
    m_fileName = name;
```

```
        std::string s = mode;
        return open(s);
    }
```

Bây giờ nếu bạn biên dịch và chạy chương trình, bạn sẽ thấy nó chạy tốt và tạo hai file như bạn mong đợi.

Phần IV

Các lớp

Kỹ thuật 21: Tạo một lớp Complete

Tiết kiệm thời gian bằng cách:

- Định nghĩa một lớp Complete
- Tạo các template cho một lớp Complete
- Test lớp Complete

Khi bạn cố sử dụng hoặc tái sử dụng một lớp (class) trong C++, không có gì hoàn toàn thất vọng bằng thấy phương thức mà bạn cần không được thực thi trong lớp đó hoặc phương thức không tốt khi bạn cố sử dụng nó trong môi trường mà bạn đang sử dụng. Lý do cho điều này thường là nhà lập trình đã phát triển lớp không tạo một lớp - Complete - nhưng chính xác tạo một lớp như vậy có nghĩa gì? Đó là câu hỏi hay và kỹ thuật này sẽ cố trả lời câu hỏi đó.

Để làm công việc của nó, một lớp Complete phải tuân theo một qui tắc cụ thể. Sau đây là những qui tắc này theo đúng thứ tự:

1. Lớp phải thực thi một constructor void.
2. Lớp phải thực thi một constructor copy.
3. Lớp phải thực thi một virtual destructor (phương thức huỷ tạo ảo).
4. Lớp phải thực thi một phương thức get cho mỗi phần tử dữ liệu được định nghĩa trong lớp.
5. Lớp phải thực thi một phương thức set cho mỗi phần tử dữ liệu được định nghĩa trong lớp.
6. Lớp phải thực thi một phương thức clone để nó có thể tạo một bản sao của chính nó.

7. Lớp phải thực thi một toán tử gán.

Nếu bạn tạo một lớp vốn tuân theo tất cả qui tắc này, có thể bạn đã tạo một lớp Complete, lớp sẽ được tái sử dụng bởi các nhà lập trình lặp đi lặp lại nhiều lần. Điều này sẽ tiết kiệm cho bạn thời gian và công sức trong việc không cần phải bắt đầu lại từ đầu mỗi lần kiểu mà đó cần được sử dụng trong một project.

Tạo một khuôn mẫu lớp Complete

Các bước sau đây trình bày một ví dụ về lớp Complete mà bạn có thể sử dụng làm khuôn mẫu (template) cho tất cả lớp mà bạn thực thi sau này.

Listing 21.1: Lớp Complete

```
#include <stdio.h>
#include <string>
#include <string.h>
class Complete
{
private:
    bool dirty; // Keep track of the object state.          → 1
private:
    int x;
    double y;
    std::string s;
    char *p;
    // Create an initialization function that
    // can reset all values.
    void Init()
    {
       x = 0;
       y = 0;
       s = "";
       if ( p )
          delete p;
       p = NULL;
       dirty = false;
    }
```

```
        // Create a copy function that you can use to make
        // clones of an object.
        void Copy( const Complete& aCopy )
        {
            Init();
            x = aCopy.x;
            y = aCopy.y;
            s = aCopy.s;
            if ( p )
                delete p;
            p = NULL;
            if ( aCopy.p )
            {
                p = new char[ strlen(aCopy.p) ];
                strcpy( p, aCopy.p );
            }
            dirty = true;
        }
    public:
        // Always create a default constructor.
        Complete()
        {
            // We need to set the pointer first.
            p = NULL;
            // Now initialize.
            Init();
        }
            // Always create a copy constructor.
        Complete( const Complete& aCopy )
        {
            // We need to set the pointer first.
            p = NULL;
            // Now copy the object.
            Copy( aCopy );
        }
```

```
// Always create a full constructor with all accessible
// members defined.
Complete( int _x, double _y, std::string& _s, const char *_p )
{
   x = _x;
   y = _y;
   s = _s;
   if ( _p )
   {
      p = new char[ strlen(_p) ];
      strcpy ( p, _p );
   }
   else
      p = NULL;
   dirty = true;
}
// Always create a virtual destructor.
virtual ~Complete()
{
   if ( p )
      delete p;
}
// Next, define accessors for all data that can be public. If
// it's not intended to be public, make it a private accessor.
// First, define all the set methods. The "dirty" flag is not
// necessary for completeness, but it makes life a LOT easier.
void setX(const int& _x)
{
   if ( x != _x )
      dirty = true;
   x = _x;
}
void setY(const double& _y)
{
   if ( y != _y )
```

```
            dirty = true;
        y = _y;
    }
    // Note that for strings, we always use the easiest base type.
    void setS(const char *_s)
    {
        if ( s != _s )
            dirty = true;
        s = _s;
    }
    void setP( const char *_p)
    {
        if ( p != _p )
            dirty = true;
        // Always clear out a pointer before setting it.
    if ( p )
    {
        delete p;
        p = NULL;
    }
    if ( _p )
    {
        p = new char [ strlen(_p) ];
        strcpy( p, _p );
    }
    else
        p = NULL;
}
// Now the data get functions. Note that since they cannot modify
// the object, they are all marked as const.
    int getX(void) const
    {
        return x;
    }
    double getY(void) const
```

```
    {
        return y;
    }
    std::string getS(void) const
    {
        return s;
    }
    // Note: For internal pointers, always return a CONST pointer.
    const char * getP(void) const
    {
        return p;
    }
    // Implement a clone operator.
    Complete Clone(void)
    {
        Complete c;
        c.Copy( *this );
        return c;
    }
    // Implement an assignment operator.
    Complete operator=( const Complete& aCopy )
    {
        Copy( aCopy );
        return *this;
    }
};
```

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch21.cpp.

2. Gõ nhập mã từ Listing 21.1 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

3. Lưu file nguồn.

Test lớp Complete

Chỉ viết lớp thì chưa đủ. Bạn cũng cần test nó. Các trường hợp test cung cấp hai cách sử dụng khác nhau cho các nhà phát triển. Thứ nhất, chúng cung cấp một cách để bảo đảm bạn đã không làm cho mã hoạt động không tốt khi bạn thực hiện thay đổi. Thứ hai, quan trọng hơn, chúng đóng vai trò như một phần trợ giảng cho người dùng lớp của bạn. Bằng cách chạy driver thử nghiệm và xem thứ tự bạn thực hiện mọi thứ, nhà lập trình có thể khám phá cách sử dụng mã trong chương trình của chính mình cũng như những giá trị nào được mong đợi và giá trị nào là những giá trị lỗi. Nó hầu như là mã dễ hiểu.

Vậy thì bạn bắt đầu viết các phép test cho các lớp của bạn như thế nào? Trước tiên, bạn test tất cả điều kiện "bình thường". Một cách nhanh để minh hoạ là tạo một driver thử nghiệm để test các constructor cho lớp như sau:

1. Trong code editor mà bạn chọn, mở lại file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch21.cpp.

2. Gõ nhập mã từ Listing 21.2 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 21.2: Driver thử nghiệm lớp Complete

```
void DumpComplete( const Complete& anObj )
{
  printf("Object:\n");
  printf("X : %d\n", anObj.getX() );
  printf("Y : %lf\n", anObj.getY() );
  printf("S : %s\n", anObj.getS().c_str() );
  printf("P : %s\n", anObj.getP() ?
  anObj.getP() : "NULL" );
}
int main()
{
  // Test the void constructor.
```

```
Complete c1;
// Test the full constructor.
std::string s = "Three";
Complete c2(1, 2.0, s, "This is a test");
// Test the copy constructor.
Complete c3(c2);
// Test the = operator.
Complete c4=c1;
DumpComplete( c1 );
DumpComplete( c2 );
DumpComplete( c2 );
DumpComplete( c4 );
}
```

3. Lưu file mã nguồn trong editor và đóng ứng dụng code editor.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng. Bạn sẽ thấy kết quả từ Listing 21.3 xuất hiện trong cửa sổ console.

Listing 21.3: Kết quả

```
$ ./a.exe
Object:
X : 0
Y : 0.000000
S :
P : NULL
Object:
X : 1
Y : 2.000000
S : Three
P : This is a test
Object:
X : 1
Y : 2.000000
```

```
S   Three
P : This is a test
Object:
X : 0
Y : 0.000000
S
P   NULL
```

Như bạn có thể thấy, chúng ta có được kết quả mong đợi. Các giá trị được khởi tạo mặc định sang những gì chúng ta xác lập chúng trong các constructor và hàm copy khác nhau. Rất khó đánh giá quá mức tầm quan trọng của việc có các phép test đơn vị như vậy. Với các phép test đơn vị, bạn có sẵn một cách để kiểm chứng các thay đổi; bạn có các cách để bắt đầu test phần mềm; và bạn có tài liệu ngay trong mã. Test đơn vị là một phần quan trọng của những phương pháp phát triển chẳng hạn như eXtreme Programming (bây giờ được gọi là Agile Programming) và những phương pháp tương tự.

Một điều quan trọng khác cần ghi chú trong mã là cờ dirty (được minh hoạ tại → **1** trong Listing 21.1) được đưa vào mã. Cờ dirty đơn giản có thể dễ dàng được trích xuất đưa vào lớp nhỏ riêng của nó để quản lý các đối tượng dirty - và có thể được tái sử dụng qua nhiều lớp khác nhau như được minh hoạ trong Listing 21.4.

Listing 21.4: Một lớp ChangeManagement

```
class ChangeManagement
{
private:
    bool dirty;
public:
    ChangeManagement(void)
    {
        dirty = false;
    }
    ChangeManagement( bool flag )
    {
        setDirty(flag);
    }
    ChangeManagement( const ChangeManagement&
    aCopy )
```

```
    {
        setDirty(aCopy.isDirty());
    }
    virtual ~ChangeManagement()
    {
    }
    void setDirty( bool flag )
    {
        dirty = flag;
    }
    bool isDirty(void)
    {
        return dirty;
    }
    ChangeManagement& operator=(const
    ChangeManagement& aCopy)
    {
        setDirty( aCopy.isDirty() );
        return *this;
    }
};
```

Tại sao chúng ta có thể muốn tạo một cờ dirty? Vì một lý do, chúng ta có thể quản lý "dirtiness" của các đối tượng bên ngoài các đối tượng. Ví dụ, chúng ta có thể có một trình quản lý mà lớp ChangeManagement đã "giao tiếp" để báo cho nó biết khi nào các đối tượng nào đó trở nên dơ hoặc sạch (và do đó phải được ghi qua lại đĩa). Loại điều này rất hữu dụng cho một trình quản lý cache của các đối tượng, khi bộ nhớ tốn kém, nhưng không gian đĩa hoặc không gian lưu trữ khác có sẵn.

Kỹ thuật 22: Sử dụng sự thừa kế ảo

Tiết kiệm thời gian bằng cách

- Tìm hiểu sự thừa kế
- Định nghĩa sự thừa kế ảo (virtual inheritance)
- Thực thi sự thừa kế ảo
- Test và sửa mã

Không có gì ngẫu nhiên sách này dành một ít thời gian để tìm hiểu các lớp được tạo như thế nào - và cách bạn có thể thừa kế từ chúng như thế nào. Sự thừa kế (inheritance) cho phép bạn tiết kiệm thời gian và công sức bằng việc cung cấp một cơ sở mã sẵn sàng được tái sử dụng trong các lớp riêng của bạn. Tuy nhiên, khi bạn thiết kế lớp, có một số vấn đề sẽ gây cho bạn phiền toái và lo âu. Một vấn đề như vậy nằm trong sự đa thừa kế (multiple inheritance). Trong khi sự đa thừa kế có thể giải quyết nhiều vấn đề trong thiết kế lớp, nhưng nó cũng có thể gây nhiều vấn đề. Ví dụ, xem xét trường hợp của Listing 22.1.

Listing 22.1: Ví dụ về sự đa thừa kế

```
class Base
{
    char *name;
public:
    virtual const char *Name();
    void setName( const char *n );
}
class A : public Base
{
} class B
:
public Base
{
}
class C : public A, public B
{
}
```

Listing 22.1 trình bày một vấn đề với sự đa thừa kế mà bạn có lẽ không nghĩ có thể từng xảy ra - nhưng nó xảy ra gần như luôn luôn. Khi chúng ta có một lớp cơ sở mà tất cả các lớp trong hệ thống thừa kế thông tin từ đó - chẳng hạn như một lớp Object lưu trữ tên (đó là kiểu) của lớp - chúng ta gặp phải vấn đề thừa kế từ cùng một lớp cơ sở trong nhiều cách. Tình huống này thường được gọi là "tam giác chết" của lập trình hướng đối tượng. Nếu bạn thể hiện cụ thể một đối tượng có kiểu C như trong mã được minh hoạ trong Listing 22.1, trình biên dịch và linker (trình liên kết) sẽ không phản đối - mọi thứ dường như làm việc tốt. Tuy nhiên, vấn đề xuất hiện khi chúng ta cố sử dụng các phương thức trong lớp cơ sở Base. Nếu chúng ta viết

```
const char *name = c. Name ( );
```

thì trình biên dịch ngay tức thì đưa ra một lỗi cho dòng mã nguồn. Lý do rõ ràng: Chúng ta gọi phương thức name nào ở đây, có hai lớp Base trong cây thừa kế cho lớp C. Nó phải là lớp trong lớp cơ sở A, hoặc lớp trong lớp cơ sở B? Cả hai đều không rõ ràng mặc dù chúng ta có thể định phạm vi cho câu trả lời với một chút nỗ lực:

```
const char *name = c.B: :Name( );
```

Điều này sẽ giải quyết vấn đề bởi vì bây giờ trình biên dịch biết rằng chúng ta muốn nói đến phương thức Name vốn được thừa kế qua cây lớp B.

Thật không may, điều này không thật sự giải quyết vấn đề. Sau cùng, lớp C không phải là B, cũng không phải là A - nó là C và chỉ thế. Bạn có thể nghĩ một mẹo như vậy có thể "giải quyết" vấn đề:

```
class C : public A, public B
{
public:
   C()
      : Base("C")
   {
   }
```

Ở đây chúng ta cho trình biết cụ thể rằng đây là một đối tượng C và nên được sử dụng làm biên dịch đối tượng. Bởi vì constructor lớp cơ sở lấy một tên và tên đó được sử dụng trong phương thức Name, do đó nó phải gán giá trị tên vào C. Vấn đề là nếu bạn cố biên dịch sự bế bộn này, bạn nhận được lỗi được minh hoạ trong Listing 22.2:

. Điều này không giải quyết vấn đề gì cả, trình biên dịch than phiền bởi vì nó không thấy lớp cơ sở nào mà chúng ta đang cố sử dụng. Nó

là lớp cơ sở của A hoặc lớp cơ sở của B? Điều này không rõ ràng và do đó chúng ta không đơn giản gọi constructor lớp cơ sở.

Vậy thì làm thế nào chúng ta có thể thừa kế hai lớp cơ sở vốn "lần lượt" thừa kế từ một lớp cơ sở chung? Câu trả lời là trong một khái niệm C++ được gọi là sự thừa kế ảo (virtual inheritance). Sự thừa kế ảo kết hợp nhiều lớp cơ sở thành một lớp cơ sở đơn trong cây thừa kế sao cho không bao giờ có một sự mơ hồ nào.

Trong Listing 22.1, với hai lớp thừa kế từ một lớp cơ sở chung, vấn đề xảy ra bởi vì lớp cơ sở xảy ra trong cây thừa kế hai lần. Sự thừa kế ảo buộc trình biên dịch tạo chỉ một instance đơn của lớp cơ sở - và sử dụng dữ liệu trong instance đó bất cứ nơi nào dữ liệu đã được xác lập sau cùng. Điều này có nghĩa mã bỏ qua việc thể hiện cụ thể các lớp cơ sở, trong hai lớp cơ sở (A và B, trong ví dụ) và sử dụng chỉ phần thể hiện cụ thể trong cấp lớp sau cùng, C.

Listing 22.2: Kết quả trình biên dịch cho lỗi đa thừa kế.

```
ch3_11.cpp: In constructor 'C::C()':
ch3_11.cpp:65: error: 'Object' is an ambiguous base of 'C'
ch3_11.cpp:65: error: type 'class Object' is not a direct base of 'C'
ch3_11.cpp: In member function 'void C::Display()':
ch3_11.cpp:70: error: request for member 'Name' is ambiguous in multiple
    inheritance lattice
ch3_11.cpp:23: error: candidates are: virtual const char* Object::Name()
ch3_11.cpp:23: error: virtual const char* Object::Name()
```

Thực thi sự thừa kế ảo

Thực thi sự thừa kế ảo (virtual inheritance) trong các lớp cơ sở cho phép bạn tạo một cấu trúc thừa kế cho phép tất cả lớp khác thừa kế từ các lớp cơ sở của bạn làm việc tốt. Các bước sau đây xem xét cách chúng ta có thể tạo một cấu trúc thừa kế vốn thực thi sự thừa kế ảo:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch22.cpp mặc dù bạn có thể sử dụng bất cứ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 22.3 vào file.

   Hoặc tốt hơn, sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 22.3: Sự thừa kế lớp cơ sở.

```
#include <stdio.h>
#include <string>
class Object
{
private:
    char *name;
public:
    Object(void)
    {
        name=NULL;
    }
    Object(const char *n)
    {
        setName( n );
    }
    virtual ~Object()
    {
        if ( name )
            delete name;
            name = NULL;
    }
    virtual const char *Name()
    {
        return name;
    }
    virtual void setName(const char *n)
    {
        if ( name )
            delete name;
        name = NULL;
        if ( n )
        {
            name = new char[strlen(n)+1];
            strcpy( name, n );
```

```
        }
    }
};
class A : public virtual Object                    → 1
{
public:
    A()
        : Object("A")
    {
    }
    virtual ~A()
    {
    }
};
class B : public virtual Object                    → 2
{
public:
    B()
        : Object("B")
    {
    }
};
class C : public A, public B
{
public:
    C()
    {
    }
    void Display()
    {
        printf("Name = %s\n", Name() );
    }
};
int main(int argc, char **argv)
{
```

```
    C c;
    c.Display();
}
```

Các điểm cốt yếu dẫn đến mã ở trên nằm trong các dòng được đánh dấu bằng → 1 và → 2. Hai dòng này buộc trình biên dịch tạo chỉ một instance Object trong cây thừa kế của cả A và B.

3. Lưu mã dưới dạng một file trong code editor và đóng ứng dụng editor.

4. Biên dịch file mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích và sau đó chạy file thực thi vừa có được.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây:

```
$ ./a.exe
Name = (null)
```

Đây không phải là những gì chúng ta muốn thấy. Chúng ta mong đợi tên của lớp. Tên đó nên là "C". Phần kế tiếp sẽ giải quyết điều đó và cho chúng ta kiểu mà chúng ta thật sự muốn.

Sửa mã

Vấn đề trong ví dụ đơn giản xuất hiện bởi vì chúng ta đã giải định rằng mã hiển nhiên đi theo một trong hai đường dẫn qua cây thừa kế và gán một tên vào lớp. Với sự thừa kế ảo, điều như vậy không xảy ra. Trình biên dịch không biết lớp nào chúng ta muốn gán tên vào vì các giá trị thuộc về lớp C thay vì các lớp A và B. Chúng ta phải cho mã biết những gì cần làm. Hãy làm điều đó ở đây.

1. Mở lại file mã nguồn được tạo trước đó (được gọi là ch22.cpp) và biên tập nó trong code editor.

2. Chỉnh sửa constructor cho lớp C như sau:

```
C()
    : Object("C")
{
}
```

3. Biên dịch lại và chạy chương trình và bạn sẽ thấy kết quả chính xác sau đây từ ứng dụng:

```
$ ./a.exe
Name = C
```

Không luôn luôn chỉnh sửa các lớp cơ sở cho một đối tượng nào đó nhưng khi có thể, hãy sử dụng kỹ thuật này để tránh "kim cương khủng khiếp" (có một lớp được dẫn xuất từ hai lớp cơ sở vốn dẫn xuất

từ một lớp cơ sở chung) - và sử dụng các lớp vốn có các lớp cơ sở chung làm các lớp cơ sở riêng của bạn.

Kỹ thuật 23: Tạo các toán tử quá tải

Tiết kiệm thời gian bằng cách:

- Định nghĩa các toán tử quá tải
- Sử dụng các toán tử chuyển đổi
- Sử dụng các toán tử quá tải
- Test toán tử

Một trong những khả năng hấp dẫn nhất đã được thêm vào C++ là khả năng thật sự thay đổi cách trình biên dịch hiểu ngôn ngữ. Trước C++, nếu bạn có một lớp được gọi là Foo chẳng hạn và bạn muốn viết một phương thức hoặc hàm để thêm hai đối tượng Foo, bạn phải viết mã tương tự như sau:

```
Foo addTwoFoos( const Foo&f1, const
Foo& f2)
{
    Foo f3;
    // Do something to add the two foos (f1 and f2)
    return f3;
}
```

Sau đó bạn có thể gọi hàm trong mã ứng dụng như sau:

```
Foo f1(0);
Foo f2(1);
Foo f3;
f3 = addTwoFoos(f1,f2);
```

Các toán tử quá tải (Overloaded operator) cho phép bạn thay đổi cơ bản cú pháp của ngôn ngữ chẳng hạn như thay đổi cách toán tử cộng (+) được sử dụng. Tuy nhiên, với việc thêm các toán tử quá tải, bây giờ bạn có thể viết dòng mã như sau:

```
Foo operator+(const Foo& f1,
    const Foo& f2 )                    → 1
{
    Foo f3;
```

```
    // Do something to add them
    return f3;
}
```

Trong mã, bây giờ bạn có thể đưa vào các câu lệnh chẳng hạn như câu lệnh sau đây:

```
Foo = f 3 = f1 + f2
```

Nếu không có toán tử quá tải (→ **1**), dòng này sẽ tạo ra một lỗi biên dịch vì trình biên dịch không biết cách nào để thêm hai đối tượng có kiểu Foo.

Dĩ nhiên, khả năng này có một cái giá tương ứng. Khi bạn quá tải (overload) các toán tử như vậy, thậm chí câu lệnh trông đơn giản nhất có thể gây ra các vấn đề. Bởi vì bạn không còn có thể bảo đảm một dòng mã đơn giản dẫn đến một thao tác đơn, bạn phải bước vào mọi dòng mã trong trình gỡ rối để truy vết qua và xem điều gì thật sự xảy ra.

Ví dụ, xem câu lệnh trông đơn giản sau đây, sau đó chúng ta gán một đối tượng Foo vào một đối tượng khác:

```
Foo f1 = 12;
```

Câu lệnh này có thể được che giấu trong hàng trăm dòng mã. Nếu một lỗi xuất hiện trong mã xử lý câu lệnh gán đơn giản này, bạn phải đi sâu vào mọi một trong các dòng đó để tìm nó. Do đó, hãy xem xét: một toán tử quá tải có thể khó đánh bại về khả năng đọc. Nói A + B khi bạn có ý định cộng hai đối tượng thì trực quan hơn là viết add (A,B), nhưng nó là một cơn ác mộng gỡ rối. Hãy xem xét rất thận trọng nhu cầu thực sự về việc quá tải một toán tử cụ thể với sự phiền phức mà bạn có thể gây ra cho những người đang cố hiểu tại sao một tác dụng phụ trong mã của bạn khiến cho chương trình của họ không làm việc.

Sử dụng các toán tử chuyển đổi

Ngoài phép cộng, loại toán tử khác vốn thường được thực thi cho các lớp là một toán tử chuyển đổi (convertion operator). Với nó, bạn có thể chuyển đổi một lớp nào đó thành nhiều thứ. Ví dụ, bạn có thể chuyển đổi một chuỗi (string) thành một character pointer - hoặc một lớp thành một số nguyên (integer) để sử dụng trong các hàm khác. Trong những trường hợp như vậy, bạn sử dụng toán tử chuyển đổi như sau:

```
operator const char *( )
```

Phần const char * của toán tử định nghĩa những gì bạn chuyển đổi dữ liệu lớp thành. Từ khoá operator cho trình biên dịch biết rằng bạn đang thực thi từ khoá này dưới dạng một toán tử. Nếu bạn thực thi toán tử được cho ở đây trong mã, bạn có thể sử dụng lớp vốn thực thi mã ở bất cứ nơi nào bạn sử dụng một giá trị const char *. Bạn có thể làm điều này trực tiếp như trong các dòng mã sau đây:

```
printf ("The value as a string is; %s \ n",
(const char *)myOb);
```

Hoặc bạn có thể làm điều này một cách ngầm định bằng cách trước tiên đưa vào các dòng mã sau đây:

```
void print_a_string( const char *s )
{
    print("string: %s \n", s);
}
```

và sau đó tham chiếu các dòng đó với dòng sau đây:

```
print_a_string( myOb );
```

Trình biên dịch tự động gọi toán tử chuyển đổi "một cách lặng lẽ" khi đối tượng được chuyển đến hàm print_a_string. Sự chuyển đổi được áp dụng và pointer const char * được chuyển vào hàm thay vì đối tượng. Chú ý rằng, tiến trình này không đòi hỏi một số hao phí - và nếu sự chuyển đổi là sang một kiểu không cơ bản, một đối tượng tạm thời được tạo ra - điều này có thể gây ra các vấn đề nếu bạn đếm tham chiếu (theo dõi các phần cấp phát và huỷ cấp phát) các đối tượng. Bạn sẽ có một đối tượng mới được tạo bởi trình biên dịch vốn không xuất hiện trong mã của bạn.

Hãy luôn nhớ rằng việc bạn có thể sử dụng chức năng C++ chẳng hạn như các đối tượng quá tải không có nghĩa là bạn nên hoặc phải sử dụng khả năng đó. Hãy luôn luôn thực hiện những điều hợp lý nhất trong tình huống lập trình của bạn.

Sử dụng các toán tử quá tải

Các toán tử quá tải là những toán tử có cùng một tên nhưng có các số hoặc các loại đối số khác nhau. Hãy tạo một vài toán tử quá tải trong một lớp để minh hoạ kỹ thuật này trong C++.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch23 mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 23.1 vào file.

Listing 23.1: Các toán tử quá tải

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
class MyString
{
    char *buffer;
    int length;
private:
    void SetBuffer( const char *s )
    {
       if( buffer )
          delete buffer;
       buffer = NULL;
       length = 0;
       if ( s )
       {
          buffer = new char[ strlen(s)+1 ];
          strcpy( buffer, s );
          length = strlen(buffer);
       }
    }
public:
    MyString(void)                                    → 1
    {
       buffer = NULL;
       length = 0;
    }
    MyString( const char *s )
    {
       buffer = NULL;
       SetBuffer ( s );
```

```
{
// Create a string that is blank, of the
   length given.
MyString( int length )
{
   buffer = new char[ length+1 ];
   for ( int i=0; i<length; ++i )
      buffer[i] = ' ';
}
MyString( const MyString& aCopy )                    → 2
{
      buffer = NULL;
      SetBuffer ( aCopy.buffer );
}
virtual ~MyString()                                  → 3
{
   if ( buffer )
      delete buffer;
}
```

Mã này thực thi các constructor khác nhau và các phương thức trong mà chúng ta sẽ sử dụng trong lớp. Chú ý để trở thành một lớp hoàn chỉnh, chúng ta cung cấp constructor void (→ 1) và constructor copy (→ 2) cũng như một virtual destructor (→ 3). Ngoài ra, nhiều constructor khác nhau cho phép bạn thực hiện việc tạo hoàn chỉnh đối tượng bằng những cách khác nhau.

3. Thêm mã từ Listing 23.2.

   Trong trường hợp này, chúng ta chỉ có hai mẫu dữ liệu khác nhau trong lớp: chính bộ đệm (buffer) chứa chuỗi mà chúng ta đóng gói và chiều dài của bộ đệm (để theo dõi các index hợp lệ đi vào chuỗi). Thêm mã từ Listing 23.2 vào file nguồn để thực thi các phương thức truy cập (accessor).

Listing 23.2: Các phương thức accessor cho lớp MyString

```
// Accessor methods
    int Length() const
    {
       return length;
    }
    void setLength(int len)
    {
       if ( len != length )
       {
          char *temp = new char[ len+1 ];
          strncpy( temp, buffer, len );
          for ( int i=length; i<len; ++i )
             temp[i] = 0;
          delete buffer;
          buffer = temp;
       }
    }
    MyString& operator=(const MyString&
    aCopy )
    {
       SetBuffer( aCopy.buffer );
       return *this;
}
// We can overload the operator= as well
MyString& operator=(const char *str)
{
    SetBuffer(str);
    return *this;
}
```

4. Thêm mã từ Listing 23.3 vào file.

Điều này thêm các toán tử cho lớp. Đây là một bước tuỳ chọn mà bạn có thể muốn hoặc không muốn thêm vào các lớp riêng của bạn.

Listing 23.3: Các toán tử Class

```
// Be able to use the object "just as if" it were a string.
    operator const char*()
    {
       return buffer;
    }
    // Be able to iterate through the string using the [] construct.
    // Note that the users can change the string this way. Define a
    // const version of it too, so they cannot change the string.
    char& operator[]( int index )
    {
       // This is probably not the right thing to do, in reality,
       // but if they give us an invalid index, just give them the first byte.
       if ( index < 0 || index > length-1 )
          return buffer[0];
       return buffer[ index ];
    }
    const char& operator[]( int index ) const
    {
       // This is probably not the right thing to do, in reality,
       // but if they give us an invalid index, just give them the first byte.
       if ( index < 0 || index > length-1 )
          return buffer[0];
       return buffer[ index ];
    }
    // Now the fun stuff. Create an operator to return a sub-string of the
    // buffer.
    MyString operator()(int stIndex, int endIndex)
    {
       if ( stIndex < 0 || stIndex > length-1 )
          stIndex = 0;
       if ( endIndex < 0 || endIndex > length-1 )
          endIndex = length-1;
       if ( stIndex > endIndex )
       {
```

```
            int temp = stIndex;
            stIndex = endIndex;
            endIndex = temp;
        }
        // Okay, we have valid indices. Let's create the string of the right
        // size.
        MyString s( endIndex-stIndex+1 );
        // Copy the buffer into the string.
        for ( int i=stIndex; i<=endIndex; ++i )
            s[i-stIndex] = buffer[i];
        return s;
    }
    // Define some comparison operators, case-insensitive.
    bool operator==(const MyString& aString )
    {
        if ( Length() != aString.Length() )
        return false;
    for ( int i=0; i<Length(); ++i )
    {
        char c1 = (*this)[i];
        char c2 = aString[i];
        if ( toupper(c1) != toupper(c2 ) )
            return false;
    }
    return true;
}
    // Do the same for comparisons to literal strings.
    bool operator==(const char *str)
    {
        if ( Length() != strlen(str) )
            return false;
        for ( int i=0; i<Length(); ++i )
        {
                char c1 = (*this)[i];
```

```
            char c2 = str[i];
            if ( toupper(c1) != toupper(c2 ) )
                return false;
        }
        return true;
    }
};
```

Listing 23.3 thực thi một số toán tử cho lớp này (chúng ta sẽ thêm các toán tử so sánh, toán tử tạo index, một toán tử để trả về một chuỗi con của chuỗi và một số toán tử so sánh để có thể thấy tất cả được lắp ghép như thế nào).

Test lớp MyString

Sau khi tạo lớp MyString, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn.

Các bước sau đây trình bày cách tạo một driver thử nghiệm để minh hoạ lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch23.

2. Gõ nhập mã từ Listing 23.4 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Chú ý rằng chúng ta có thể sử dụng toán tử "[ ]" ở một trong hai phía của dấu bằng trong một biểu thức. Nếu bạn sử dụng [ ] làm giá trị l, bạn có thể thực sự gán trực tiếp các giá trị vào bộ đệm trong mã. Tuy nhiên, không giống như một mảng C++ "chuẩn", mã thực sự hiệu lực hoá để bảo đảm index mà bạn chuyển vào nằm trong dãy hợp lệ cho bộ đệm trong, do đó không còn chèn bộ đệm nữa và chương trình không còn bị đổ vỡ nữa.

3. Lưu mã nguồn trong code editor.

Listing 23.4: Driver thử nghiệm lớp Buffer.

```
void print_a_string( const char *s )
{
    printf("The string is: %s\n", s );
}
int main(int argc, char **argv)
{
    MyString s("This is a test");
    printf("The string is: [%s]\n", (const char *)s );
    s[4] = 'm';
    printf("The string is now: [%s]\n", (const char *)s );
    // Get a sub-string of the string.
    MyString sub = s(3,7);
    printf("The sub-string is: [%s]\n", (const char *)sub );
    // We can reset strings to be bigger or smaller.
    sub = "Hello world";
       printf("The sub-string is now: [%s]\n", (const char *)sub );
    if ( sub == "hEllO world" )
       printf("Strings compare\n");
    else
       printf("Strings do NOT compare\n");
    if ( sub == "Goodbye" )
       printf("Strings compare\n");
    else
       printf("Strings do NOT compare\n");
    MyString copy = sub;
    if ( sub == copy )
       printf("Strings compare\n");
    else
       printf("Strings do NOT compare\n");
    print_a_string( sub );
    return 0;
}
```

1. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

5. Chạy chương trình vừa có được trên hệ điều hành mà bạn chọn.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng trong cửa sổ console.

```
$ ./a.exe
The string is: [This is a test]
The string is now: [Thismis a test]
The sub-string is: [smis ]
The sub-string is now: [Hello world]
Strings compare
Strings do NOT compare
Strings compare
The string is: Hello world
```

Như bạn có thể thấy từ kết quả trong Listing 23.2, các hàm tạo index (toán tử [ ] và toán tử ( )) cho phép bạn truy tìm và chỉnh sửa các phần được chọn của chuỗi. Các hàm so sánh cũng làm việc cho thấy các toán tử quá tải làm việc tốt.

Ví dụ này thực sự cho thấy sức mạnh của các toán tử ghi đè và tạo các kiểu riêng của bạn trong C++. Bạn có thể bảo vệ người dùng cuối khỏi hầu như tất cả vấn đề vốn đã xảy ra trong những năm phát triển phần mềm.

Kỹ thuật 24: Định nghĩa các phương thức xử lý new và delete

Tiết kiệm thời gian bằng cách:

- Thực thi các phương thức xử lý (handler) new và delete
- Quá tải các phương thức xử lý new và delete
- Tạo một chương trình theo dõi cấp phát bộ nhớ
- Test chương trình

Một khối tạo cơ bản của ngôn ngữ C++ là một tập hợp từ khoá cốt lõi để cấp phát và giải phóng các khối bộ nhớ. Các từ khoá new và delete (ví dụ) đã được thêm vào ngôn ngữ chủ yếu để hỗ trợ việc thêm các đối tượng bằng các constructor và destructor - nhưng chúng cũng được sử dụng để cấp phát thêm các khối bộ nhớ "generic" chẳng hạn như các mảng ký tự.

Lý do chính để sử dụng toán tử new là nó tự động cấp phát khối bộ nhớ cần thiết và sau đó gọi constructor cho khối bộ nhớ để khởi tạo nó một cách phù hợp (các hàm alloc/malloc kiểu C cũ không thể làm điều đó).

Vấn đề với các toán tử new và delete không thực sự là cách chúng được sử dụng; đó là chúng không theo dõi những gì được cấp phát và những gì bị xoá. Có những vấn đề khác chẳng hạn như xử lý các bộ chứa đối tượng (cho phép bạn tái sử dụng các đối tượng mà không cần cấp phát lại những đối tượng mới) - nhưng hầu hết nhà lập trình đồng ý rằng vấn đề theo dõi việc cấp phát bộ nhớ quan trọng hơn. Nếu bạn có thể theo dõi chính xác khi nào bộ nhớ được cấp phát và được huỷ cấp phát trong ứng dụng, bạn sẽ tiết kiệm lượng thời gian đáng kể trong tiến trình gỡ rối nhằm cố tìm ra những rò rỉ bộ nhớ và các hoạt động ghi đè.

Hãy xem xét hàm sau đây được viết trong C++:

```
int func(int x)
{
    char *ptr = new char[200]; → 1
    if ( x < 0 || x > 100 )
        return -1; → 2
    // Do some processing of the ptr.
    // De-allocate memory.
    delete ptr;
}
```

Mã này chứa một rò rỉ bộ nhớ tinh vi nhưng quan trọng. Nếu bạn gọi hàm với một giá trị chẳng hạn 102 được chuyển cho x, bạn sẽ thấy vấn đề. Hàm cấp phát một khối bộ nhớ vốn dài 200 byte (tại → 1) và sau đó trả về một không huỷ cấp phát khối (tại → 2). Sau đó, bộ nhớ đó được tiêu thụ cho đến khi chương trình thoát và không còn có sẵn cho ứng dụng nữa. Điều này dường như không phải là một vấn đề lớn như vậy - trừ phi thường trình này được gọi một vài ngàn lần. Bất chợt khối 200 byte trở thành một rò rỉ bộ nhớ vài megabyte. Không phải là một kết quả tốt.

Thật may thay, những nhà thiết kế C++ đã xem xét rằng những vấn đề như vậy có thể dễ dàng xuất hiện. Mặc dù họ chọn không cài sẵn một hệ thống thu dọn thông tin thừa, như trong Java, nhưng họ cung cấp các khối tạo để xây dựng hệ thống cấp phát và huỷ cấp phát bộ nhớ riêng của bạn và theo dõi những thứ như vậy. Để theo dõi việc cấp phát bộ nhớ trong C++, chúng ta cần khả năng quá tải các phương thức xử lý (handler) new và delete trong ngôn ngữ. Bạn có thể nghĩ

rằng đây là một công việc phức tạp, nhưng như kỹ thuật 23 cho thấy, các toán tử quá tải (do đó chúng dường như là một phần cơ bản của ngôn ngữ) đơn giản trong C++. Các toán tử new và delete không là ngoại lệ đối với tiến trình quá tải mặc dù bạn phải bắt đầu với nó một cách hơi khác biệt. Trong kỹ thuật này, chúng ta sẽ xem cách quá tải các phương thức xử lý new và delete cho toàn bộ hệ thống mặc dù toàn bộ tiến trình có thể được hạ xuống cấp lớp.

Quá tải các phương thức xử lý new và delete

Làm thế nào chúng ta có thể quá tải các toán tử new và delete để theo dõi những gì được cấp phát trong một chương trình nào đó và báo cáo về những phần cấp phát nào đã không bao giờ được giải phóng? Hãy xem một ví dụ về điều đó ngay bây giờ:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch24.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 24.1 vào file.

Listing 24.1: Các phương thức xử lý new hoặc delete

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    long number;
    long address;
    long size;
    char file[64];
    long line;
} IALLOC_INFO;
IALLOC_INFO *allocations[ 100000 ];
int         nPos = 0;
void AddTrack(long addr, long asize)
{
    if ( asize == 2688 )
        printf("Found one!\n");
    IALLOC_INFO *info = (IALLOC_INFO *)malloc(sizeof(IALLOC_INFO));
    info->address = addr;
    info->size = asize;
```

```
    info->number = nPos;
    allocations[nPos] = info;
    nPos ++;
};
bool RemoveTrack(long addr)
{
    bool bFound = false;
    for(int i = 0; i != nPos; i++)
    {
        if(allocations[i]->address == addr)
        {
            // Okay, delete this one.
            free( allocations[i] );
            bFound = true;
            // And copy the rest down to it.
            for ( int j=i; j<nPos-1; ++j )
                allocations[j] = allocations[j+1];
            nPos --;
            break;
        }
    }
    if ( !bFound )
    printf("Unable to find allocation for delete [%ld]\n",addr);
    return bFound;
};
```

Mã này theo dõi tất cả phần cấp phát - và thêm hoặc loại bỏ chúng ra khỏi một mảng toàn cục khi chúng ta thực hiện việc xử lý. Theo cách này chúng ta có thể theo dõi tất cả lệnh gọi đến new hoặc delete trong ứng dụng - và báo cáo về các lệnh gọi đến new vốn không được kết hợp với một lệnh gọi đến delete. Dĩ nhiên, phương pháp này giới hạn bao nhiêu phần cấp phát mà chúng ta sẽ theo dõi (nó là một con số lớn nhưng không vô hạn). Chúng ta không thể cấp phát động mảng này mà không viết một cụm mã thú vị vốn nằm bên ngoài phạm vi của ví dụ này, do đó chúng ta sẽ sử dụng sự thoả hiệp này bây giờ.

3. Thêm mã từ Listing 24.2 vào file nguồn.

Mã này tạo một báo cáo về các khối nào hiện được cấp phát và chúng có kích cỡ như thế nào. Điều này hỗ trợ nhà phát triển theo dõi mã gây lỗi vốn đã tạo rò rỉ bộ nhớ lúc ban đầu. Dĩ nhiên, biết nơi sự rò rỉ đã xảy ra sẽ không giúp ích nếu sự rò rỉ xuất hiện trong một thư viện cấp thấp (bởi vì chúng ta không truy cập mã nguồn thư viện và không thể chỉnh sửa nó nếu chúng ta đã làm) nhưng ít ra kích cỡ sẽ giúp ích một phần. Bằng cách biết kích cỡ của phần cấp phát, chúng ta có thể ánh xạ kích cỡ đó sang một kích cỡ khối cụ thể trong chương trình hoặc kích cỡ của một đối tượng nào đó. Mã này có thể dễ dàng được di chuyển đến một file tiện ích riêng biệt, nhưng chúng ta sẽ đưa nó vào trong cùng một file vì mục đích đơn giản.

Mã này đơn giản bước qua danh sách các phần cấp phát mà chúng ta đã theo dõi, trong các thường trình add và remove và báo cáo về bất cứ điều gì một nó thấy chưa được giải phóng. Điều này không nhất thiết có nghĩa là phần cấp phát là một rò rỉ như chúng ta sẽ thấy. Điều này có nghĩa là vào thời điểm của snapshot (ảnh chụp nhanh) trạng thái bộ nhớ cụ thể này, các phần cấp phát trong danh sách đã không được giải phóng.

4. Thêm mã từ Listing 24.3 vào file nguồn bên dưới mã còn lại.

Mã này thực thi các phương thức new và delete thực sự. Một lần nữa chúng ta có thể dễ dàng di chuyển những phương thức này đến một file tiện ích riêng biệt, nhưng để nó trong một nơi thì dễ hơn. Chức năng này được bổ sung riêng biệt để xác định mạng sẽ thêm mã này vào một chương trình hiện có như thế nào.

Điều này sẽ thực thi sự quá tải thật sự của các toán tử new và delete. Để thực thi thao tác đó, thêm mã trong Listing 24.3 vào file nguồn.

Listing 24.2: Báo cáo cấp phát

```
void DumpUnfreed()
{
    long totalSize = 0;
    printf("———————————————— Allocations ————————————————————
    \n");
    for(int i = 0; i < nPos; i++)
    {
    IALLOC_INFO *pInfo = allocations[i];
    printf("(%ld) ADDRESS %x\t Size: %d unfreed\n",
       pInfo->number, pInfo->address, pInfo->size);
    totalSize += pInfo->size;
```

```
        }
        printf("-------------------------------------------------
        ----\n");
        printf("Total Unfreed: %d bytes\n\n\n", totalSize);
    };
```

Listing 24.3: Các phương thức xử lý new và delete quá tải

```
inline void * __cdecl operator new(unsigned int size)
{
    printf("Basic operator new called\n");
    void *ptr = (void *)malloc(size);
    AddTrack((long)ptr, size);
    return(ptr);
};
inline void * __cdecl operator new[](unsigned int size)
{
    printf("Array operator new called\n");
    void *ptr = (void *)malloc(size);
    AddTrack((long)ptr, size);
    return(ptr);
};
inline void __cdecl operator delete(void *p)
{
    printf("Basic operator delete called\n");
    if ( RemoveTrack((long)p) )
        free(p);
};
inline void __cdecl operator delete[](void *p)
{
    printf("Array operator delete called\n");
    if ( RemoveTrack((long)p) )
        free(p);
};
```

5. Lưu file mã nguồn.

Những phần thực thi này của mã không có gì đặc biệt. Chúng ta chỉ việc cấp phát bộ nhớ (sử dụng hàm C có sẵn (malloc) và huỷ cấp

phát bộ nhớ bằng cách sử dụng hàm free. Mã bao gồm một số câu lệnh printf gỡ rối vốn cho phép bạn hiển thị các hàm nào đang được gọi vào thời điểm nào. Trong mỗi toán tử cấp phát hoặc huỷ cấp phát, chúng ta gọi hàm theo dõi thích hợp để thêm hoặc loại bỏ phần cấp phát cụ thể này ra khỏi mảng toàn cục (global array). Một điều cần lưu ý là mã này thật sự tốt hơn phần thực thi C++ chuẩn bởi vì nó kiểm chứng rằng một pointer nào đó đã được cấp phát trước khi nó cho pointer đó được giải phóng. Bạn có thể gây ra một số rắc rối nếu bạn làm điều này:

```
char *c = new c[100];
// Do some stuff
delete c;
// Do some more stuff
delete c;
```

Hãy mong đợi những điều tồi tệ xảy ra trong một trường hợp như vậy: nó xoá cùng một pointer hai lần, điều này có khuynh hướng làm hỏng ngăn xếp và huỷ tất cả bộ nhớ trong hệ thống. Tuy nhiên, trong hệ thống, tiến trình này được đón bắt và một thông báo lỗi hiển thị. Hơn nữa, pointer thực sự không bị xoá lần thứ hai - do đó không có sự hư hỏng bộ nhớ trong hệ thống và chương trình không bị phá vỡ. Đó là lý do đủ tốt để sử dụng hệ thống như vậy trong mã sản xuất.

Test chương trình theo dõi sự cấp phát bộ nhớ

Để thấy mã theo dõi sự cấp phát làm việc như thế nào, cách dễ dàng nhất là tạo một driver thử nghiệm đơn giản để minh hoạ các phần khác nhau của hệ thống. hãy tạo một chương trình thử nghiệm đơn giản để sử dụng những phương thức new và delete đã được tạo. Các bước sau đây mà chúng ta đã tạo. Các bước sau đây hướng dẫn cách thực hiện:

1. Trong code editor mà bạn chọn, mở file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là CH24.

2. Gõ nhập mã từ Listing 24.4 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

3. Lưu mã nguồn và đóng code editor.

Listing 24.4: Driver thử nghiệm phương thức cấp phát bộ nhớ.

```
int main(int argc, char **argv)
{
    DumpUnfreed();
    char *c = new char[200];
    DumpUnfreed();
    char *c2 = new char[256];                    → 3
    DumpUnfreed();
    delete c;
    delete c;
    DumpUnfreed();
    int *x = new int[20];
    delete [] x;
    DumpUnfreed();
    Foo *f = new Foo();
    delete f;
    Foo *af = new Foo[5];
    delete [] af;
    Foo *af1 = new Foo[3];
    delete af1;
}
```

4. Biên dịch file nguồn, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn chạy file thực thi vừa có được, chương trình sẽ đưa ra kết quả được minh hoạ trong Listing 24.5.

Listing 24.5: Kết quả từ chương trình theo dõi bộ nhớ.

```
$ ./a.exe
---------------- Allocations ----------------
---------------------------------------------
Total Unfreed: 0 bytes
Array operator new called
---------------- Allocations ----------------
(0) ADDRESS a050648 Size: 200 unfreed
---------------------------------------------
```

```
Total Unfreed: 200 bytes
Array operator new called
---------- Allocations ----------
(0) ADDRESS a050648 Size: 200 unfreed
(1) ADDRESS a050770 Size: 256 unfreed
---------------------------------------

Total Unfreed: 456 bytes
Basic operator delete called
Basic operator delete called
Unable to find allocation for delete [168101448]
---------- Allocations ----------
(1) ADDRESS a050770 Size: 256 unfreed
---------------------------------------

Total Unfreed: 256 bytes
Array operator new called
Array operator delete called
---------- Allocations ----------
(1) ADDRESS a050770 Size: 256 unfreed
---------------------------------------

Total Unfreed: 256 bytes                    → 4
Basic operator new called
Foo Constructor called
Foo Destructor called
Basic operator delete called
Array operator new called
Foo Constructor called
Foo Constructor called
Foo Constructor called
Foo Constructor called
Foo Constructor called
Foo Destructor called
Foo Destructor called
Foo Destructor called
Foo Destructor called
```

```
Foo Destructor called
Array operator delete called
```

Có nhiều điều thú vị cần chú ý trong kỹ thuật này. Thứ nhất, nó cho bạn đánh giá tốt hơn những gì xảy ra trong hậu cảnh trong một chương trình C++. Thứ hai, bạn có thể thấy ngay các phần cấp phát và huỷ cấp phát được xử lý như thế nào và các rò rỉ xuất hiện ở đâu. Trong ví dụ, chúng ta có thể thấy ở cuối chương trình chúng ta có một rò rỉ bộ nhớ 256 byte (tại → 4 trong Listing 24.5). Chú ý rằng chúng ta in ra trạng thái hiện hành của chương trình một vài lần, do đó nó là sự hiển thị duy nhất chỉ báo sự rò rỉ ở cuối chương trình. Những thứ khác được để lại để minh hoạ chương trình cấp phát bộ nhớ như thế nào. Rõ ràng nếu xem mã chương trình, điều này xảy ra cho phần cấp phát c2 (xem → 3 trong Listing 24.4). Chúng ta chỉ cần thêm một lệnh gọi delete cho charcter pointer và mọi thứ sẽ tốt đẹp.

Điều thú vị khác cần chú ý trong kỹ thuật này là toán tử C++ new nào được gọi khi nào. Ví dụ, nếu bạn cấp phát một character pointer, nó gọi phiên bản mảng của toán tử new. Tình huống này phản trực giác - sau cùng bạn cấp phát một character pointer đơn - nhưng nó hợp lý nếu bạn thực sự nghĩ về nó. Nó là một mảng ký tự mà chúng ta ngầu nhiên xem là một chuỗi đơn vốn cho chúng ta phiên bản mảng (array) của toán tử new. Tương tự khi chúng ta cấp phát một đối tượng đơn, nó gọi toán tử cơ bản cho phần cấp phát new.

Hãy thử thêm mã sau đây vào cuối ứng dụng driver:

```
Foo * f1 = new Foo[3];
delete af 1;
```

Nếu bạn biên dịch đoạn mã này ở cuối chương trình driver, và sau đó chạy chương trình, bạn sẽ thấy kết quả sau đây:

```
Array operator new called
Foo Constructor called
Foo Constructor called
Foo Constructor called
Foo Destructor called
Basic operator delete called
Unable to find allocation for delete
    [168101480]
```

Xem kết quả, bạn sẽ nhận thấy constructor cho lớp đã được gọi ba lần, cho ba đối tượng mà chúng ta đã tạo. Tuy nhiên, destructor chỉ được gọi một lần. Tệ hơn, bởi vì khối bộ nhớ được cấp phát và thực sự là ba đối tượng riêng biệt, thường trình xoá của chúng ta không thể tìm thấy nó để xoá nó.

Kỹ thuật 25: Thực thi các thuộc tính

Tiết kiệm thời gian bằng cách

- Tìm hiểu các thuộc tính trong C++
- Thực thi một lớp Property
- Test lớp Property

Nếu bạn quen lập trình bằng những ngôn ngữ "mới" chẳng hạn như như Java hoặc C#, có lẽ bạn đã quen thuộc với khái niệm về những thuộc tính (property). Về cơ bản, những thuộc tính là các phần tử public của một lớp có những phương thức riêng của chúng để nhận và xác lập những giá trị của chúng. Tuy nhiên, không giống như những giá trị public truyền thống, một thuộc tính không thể được truy cập trực tiếp bởi nhà lập trình - mặc dù dường như có thể. Một thuộc tính có những hàm set và get vốn gọi ra khi bạn cố ghi hoạt động sang những giá trị thuộc tính. C++ không thực thi trực tiếp các thuộc tính trong ngôn ngữ. Điều này thật sự đáng tiếc bởi vì các thuộc tính tiết kiệm nhiều thời gian cho người dùng cuối bằng việc làm cho việc đọc và ghi các giá trị dữ liệu trong lớp trở nên dễ dàng hơn.

Ví dụ, bạn có một lớp có tên là Foo chứa một thuộc tính được gọi là age. Thuộc tính này có thể được xác lập bởi nhà phát triển ứng dụng, nhưng chỉ sang những giá trị trong dãy, ví dụ trong dãy từ 18 đến 80. Bây giờ, trong một ứng dụng C++ "chuẩn", bạn có thể định nghĩa lớp với một thành viên public chẳng hạn như sau:

```
class Foo
{
public:
    Foo()
    {
    }
    int age;
};
```

Nếu bạn có một lớp như vậy, bạn có thể viết mã ứng dụng để trực tiếp xác lập thuộc tính age như sau:

```
int main()
{
    Foo f;
    f.age = 22;
}
```

Vấn đề là bạn cũng có thể xác lập age sang một giá trị không hợp lệ. Giới hạn chỉ được thực thi bởi "qui tắc" rằng một tuổi không thể nằm bên ngoài dãy hợp lệ từ 18 đến 80. Mã của chúng ta không thi hành qui tắc này, điều này có thể gây ra các vấn đề cho các phép tính nào phụ thuộc vào qui tắc đang được tuân theo. Một phép gán không hợp lệ có thể trông như sau:

```
f.age = 10; // invalid
```

Giải pháp lý tưởng là cho phép người ta trực tiếp xác lập thuộc tính age trong lớp này, nhưng không cho phép họ xác lập nó sang những giá trị bên ngoài dãy hợp lệ. Ví dụ, nếu một người dùng đã làm như vậy và sau đó thêm câu lệnh sau đây:

```
f.age = 10;
```

Tuổi (age) sẽ không được xác lập và giữ lại giá trị cũ của nó. Việc cưỡng lại sự thay đổi không được phép này là ưu điểm của một thuộc tính thay vì cho giá trị thay đổi bất kể giá trị đầu vào nào được cho. Ngoài ra, chúng ta có thể tạo các thuộc tính chỉ đọc (read-only) vốn có thể được đọc, nhưng không được ghi. C++ không cung cấp trực tiếp khả năng này, nhưng nó cho phép chúng ta tự tạo một thứ như vậy. Một thuộc tính read-only hữu dụng cho những giá trị mà lập trình cần truy cập, nhưng không thể chỉnh sửa chẳng hạn như tổng bộ nhớ có sẵn trong hệ thống. Các thuộc tính như vậy tiết kiệm thời gian bằng việc làm cho mã dễ đọc hơn trong khi vẫn duy trì tính toàn vẹn dữ liệu.

Thực thi các thuộc tính

Tạo một lớp đơn giản thực thi những thuộc tính cho một kiểu riêng biệt - trong trường hợp này là các số nguyên (integer) có thể minh hoạ tính năng C++ này. Sau đây chúng ta có thể tuỳ biến lớp để cho phép chỉ các kiểu số nguyên hoặc các giá trị số nguyên riêng biệt.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này là file được đặt tên ch25.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 25.1 vào file.

Listing 25.1 Lớp Property.

```
#include <stdio.h>
#include <string>
class IntProperty
{
    int temp;
    int &iValue;
    bool bWrite;
public:
    void Init()
    {
        bWrite = false;
    }
    IntProperty(void)
        : iValue(temp)
    {
        Init();
    }
    virtual void set(int i)                    → 1
    {
        iValue = i;
    }
    virtual int get(void)                      → 2
    {
        return iValue;
    }
public:
    IntProperty(int& i)
        : iValue(i)
    {
        Init();
    }
    IntProperty( int i, bool read, bool write )
        : iValue(i)
```

```
{
    Init();
}
IntProperty( const IntProperty& aCopy )
    : iValue(aCopy.iValue)
{
    Init();
}
virtual ~IntProperty()
{
}
// Accessors
int getValue( void )
{
    return iValue;
}
    bool getWrite(void)
{
    return bWrite;
}
void setWrite(bool write)
{
    bWrite=write;
}
// Operators
IntProperty& operator=( int i )
    {
    if( bWrite )
        set(i);
    else
        printf("Trying to assign to a read-only property\n");
    return *this;
}
// Cast to int
operator int()
```

```
    {
        return get();
    }
};
```

Lớp này thực thi một thuộc tính theo các chuẩn C++ và làm việc như thể nó là một thuộc tính Java hoặc C#. Chúng ta sẽ có thể đọc và ghi sang giá trị dữ liệu mà không cần phải viết thêm mã , nhưng các giá trị dữ liệu sẽ được hiệu lực hoá cho dãy được phép. Để sử dụng nó, chúng ta cần nhúng nó trong một lớp mà người dùng cuối sẽ tương tác. Lớp này sẽ phơi bày thuộc tính intProperty cho người dùng cuối nhưng instance của lớp intProperty trong bất kỳ lớp khác sẽ làm việc với một biến trong của lớp đó. Lớp intProperty thật sự chỉ là một wrapper xung quanh một tham chiếu dẫn đến một biến, nhưng biến đó sẽ nằm bên ngoài phạm vi của lớp int Property.

Chú ý rằng các phương thức set (→1) và get (→2) của lớp nằm bên trong chính lớp, nhưng cũng được khai báo là virtual. Điều đó có nghĩa là thực thi một lớp dẫn xuất vốn lọc ra những giá trị dữ liệu nhất định thì bình thường như chúng ta sẽ thấy trong lớp Age Property sau đó trong kỹ thuật này.

Để dẫn xuất một lớp từ lớp int Property, chúng ta chỉ việc ghi đè các phương thức get và set bằng cách chúng ta muốn. Để giới hạn dãy giá trị số nguyên chẳng hạn, chúng ta chỉnh sửa phương thức set để chỉ cung cấp những giá trị mà chúng ta cho phép. Ngoài ra chúng ta phải ghi đè phương thức operator = bởi vì operator = không bao giờ được thừa kế bởi một lời dẫn xuất. Đó là do bạn có thể xác lập chỉ một phần của đối tượng - mà ngôn ngữ sẽ không cho bạn thực hiện, do đó bạn cũng phải ghi đè toán tử. Khi bạn tạo một lớp dẫn xuất, operator= được gọi cho lớp cơ sở. Điều này xác lập chỉ các biến thành viên trong lớp cơ sở và sẽ không xác lập các biến thành viên trong lớp dẫn xuất. Nếu không phần còn lại của lớp vẫn không đổi.

3. Thêm mã từ Listing 25.2 vào file nguồn.

Chúng ta có thể đơn giản tạo một file mới để lưu trữ lớp mới này nhưng chỉ việc kết hợp chúng cho mục đích này thì dễ dàng hơn.

Trong trường hợp này, chúng ta sẽ sử dụng intProperty trong một lớp khác.

Listing 25.2: Mở rộng lớp IntProperty

```
class AgeProperty : public IntProperty
{
private:
    virtual void set(int i)
    {
        if ( i >= 18 && i <= 80 )
            IntProperty::set(i);
    }
public:
    AgeProperty( int &var )
        : IntProperty(var)
    {
    }
    AgeProperty& operator=( int i )
    {
        IntProperty::operator=(i);
        return *this;
    }
};
```

Bây giờ để sử dụng lớp, chúng ta cần nhúng đối tượng dưới dạng một thành viên public của lớp đóng gói và cung cấp cho nó một thành viên dữ liệu mà nó có thể truy cập đến những giá trị set và get. Lớp thuộc tính chỉ là một wrapper bao quanh một giá trị bởi vì nó chứa một tham chiếu dẫn sang một giá trị dữ liệu bên ngoài lớp, nó có thể chỉnh sửa trực tiếp dữ liệu cho một lớp khác. Điều đó có nghĩa rằng bất kỳ thay đổi được thực hiện đối với tham chiếu trong lớp IntProperty sẽ được phản ánh ngay tức thì trong biến thành viên lớp nền tảng.

Để thấy được tất cả điều này được lắp ghép như thế nào, phần kế tiếp thêm một lớp vốn tận dụng lớp Age Property.

Test lớp Property

Sau khi đã định nghĩa lớp Property, chúng ta cần test nó. Các bước sau đây hướng dẫn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho chương trình thử nghiệm.

Trong ví dụ này chương trình thử nghiệm được đặt tên là ch25.cpp.

2. Đặt mã từ Listing 25.3 vào file driver thử nghiệm.

Listing 25.3: Test lớp IntValue

```
class TestIntValue
{
private:
    int myInt;
public:
    TestIntValue()
      : i(myInt)
    {
      myInt = 0;
    }
    void Print()
    {
      printf("myInt = %d\n", myInt );
    }
public:
    AgeProperty i;
};
```

Lớp này chứa một giá trị số nguyên đơn, thành viên dữ liệu duy nhất của nó. Thành viên dữ liệu được kết hợp với lớp Property trong cosntructor cho lớp, do đó bất kỳ thay đổi đối với biến thành viên sẽ được phản ánh ngay tức thì trong lớp Property và lớp TestIntValue cùng một lúc.

Bởi vì giá trị dữ liệu được sử dụng theo tham chiếu trong lớp Property, việc thay đổi thuộc tính tương đương việc thay đổi trực tiếp thành viên dữ liệu gốc. Chúng ta kiểm soát cách dữ liệu được thay đổi như thế nào trong khi cho phép trình biên dịch tạo mã thực hiện việc xử lý dữ liệu thật sự.

Lớp của chúng ta minh hoạ các giá trị dữ liệu thay đổi như thế nào và chúng được gán sang giá trị nào bất cứ lúc nào. Chúng ta sẽ sử dụng lớp này để thấy lớp thuộc tính làm việc như thế nào.

3. Thêm mã từ Listing 25.4 vào cuối file hiện có.

Listing 25.4: Mã driver thử nghiệm.

```
int main(int argc, char **argv)
{
    TestIntValue tiv;
    tiv.i = 23;                                    → 3
    printf("Value = %d\n", (int)tiv.i );
    tiv.Print();
    tiv.i.setWrite( true );                        → 5
    tiv.i = 23;
    printf("Value = %d\n", (int)tiv.i );
    int x = tiv.i;
    tiv.Print();
    printf("X = %d\n", x );
    tiv.i = 99;
    printf("Value = %d\n", (int)tiv.i );
}
```

4. Lưu file nguồn trong code editor và đóng ứng dụng editor.
5. Biên dịch file bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng:

```
$ ./a.exe
Trying to assign to a read-only property
Value = 0                                          → 4
myInt = 0
Value = 23                                         → 6
myInt = 23
X = 23
Value = 23
```

Kết quả ở trên minh hoạ rằng lớp thuộc tính làm việc tốt. Giá trị ban đầu của số nguyên là 0 như được xác định trong constructor. Bởi vì lớp đã mặc định sang read-only (setWrite đã chưa được gọi), một nỗ lực nhằm ghi sang biến (→ **3**) dẫn đến việc thay đổi đã không được thực hiện (→ **4**). Sau khi xác định cờ để cho phép các thay đổi (→ **5**), chúng ta có thể gán các giá trị vào biến và chỉnh sửa nó trong kết quả (→ **6**).

Kỹ thuật 26: Hiệu lực hoá dữ liệu với các lớp

Tiết kiệm thời gian bằng cách:

- Tìm hiểu sự hiệu lực hoá dữ liệu với các lớp
- Tạo một lớp hiệu lực hoá dữ liệu
- Test lớp

Hiệu lực hoá dữ liệu là một trong những chức năng cơ bản nhất và thông dụng nhất của một chương trình máy tính. Trước khi bạn có thể thao tác trên một mẫu dữ liệu nào đó, bạn cần biết liệu nó có hợp lệ hay không. Cho dù nó là một ngày tháng, một thời gian, một tuổi tác hoặc một số an sinh xã hội (Social Security) đi nữa, dữ liệu mà bạn chấp nhận đưa vào chương trình sẽ gây ra các vấn đề nếu nó có một định dạng không hợp lệ.

Hiệu lực hoá một kiểu dữ liệu là một dạng đóng gói hoàn hảo làm cho nó trở thành một tác vụ hoàn hảo để gán vào một lớp C++. Bởi vì chúng ta đóng gói dữ liệu và các qui tắc (rule) cho kiểu dữ liệu bên trong một lớp, chúng ta có thể di chuyển lớp đó từ project này sang project khác, bất cứ nơi nào kiểu dữ liệu được cần đến. Điều này tiết kiệm thời gian trong việc thực thi lớp cũng như thời gian và công sức trong việc hiệu lực hoá và test lớp.

Thực thi sự hiệu lực hoá dữ liệu với các lớp

Thực hiện các bước sau đây để tạo sự hiệu lực hoá riêng của bạn:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file header.

   Trong ví dụ này lớp được đặt tên là ch26.cpp.

2. Gõ nhập mã từ Listing 26.1 vào file.

Listing 26.1: Lớp hiệu lực hoá.

```
#include <string>
// Constants used in this validation
#define SSN_LENGTH 9                    → 1
#define SSN_DELIMITER '-'               → 2
// The validator class
class SSNValidator
{
```

```
    // Internal member variables
private:
    // This is the actual SSN.
    std::string _ssn;
    // This is the flag indicating validity.
    bool        _valid;
protected:
    bool IsValid(const char *strSSN);
public:
    // Constructors and destructor
    SSNValidator();
    SSNValidator( const char *ssn );
    SSNValidator( const std::string& ssn );
    SSNValidator( const SSNValidator& aCopy );
    virtual ~SSNValidator();
    // Accessors for this class
    bool Valid() { return _valid; };
    std::string SSN() { return _ssn; };
    void setSSN( const char *ssn);
    // Operators for this class
    SSNValidator operator=( const char *ssn );
    SSNValidator operator=( const std::string& ssn );
    SSNValidator operator=( const SSNValidator& aCopy );
    operator const char *();
};
```

3. Lưu mã trong code editor.

   Đây sẽ là định nghĩa cho đối tượng Validator. Sau đó lớp này có thể được đưa vào những module khác để hiệu lực hoá kiểu mà chúng ta định nghĩa. Trong ví dụ này chúng ta định nghĩa một số Social Security của Mỹ.

4. Gõ nhập mã từ Listing 26.2 vào file mới.

Listing 26.2: Social Security Number Validator

```
#include <ctype.h>
bool SSNValidator::IsValid(const char *strSSN)
{
    int i;
    // No NULL values allowed.
    if ( strSSN == NULL )
        return false;
    // Copy the result into a string, removing all delimiters.
    std::string sSSN;
    for ( i=0; i<(int)strlen(strSSN); ++i )
        if ( strSSN[i] != SSN_DELIMITER )
            sSSN += strSSN[i];
    // Must be 9 characters.
    if ( strlen(sSSN.c_str()) != SSN_LENGTH )
        return false;
    // Check to see whether all characters are numeric.
    for ( i=0; i<(int)strlen(sSSN.c_str()); ++i )
        if ( !isdigit( sSSN[i] ) )
            return false;
    // Must be okay.
    return true;
}
// Constructors and destructor
SSNValidator::SSNValidator()
{
    _ssn = "";
    _valid = false;
}
SSNValidator::SSNValidator( const char *ssn )
{
    // Only assign if valid.
    _valid = IsValid( ssn );
    if ( _valid )
        _ssn = ssn;
```

```
{
SSNValidator::SSNValidator( const std::string& ssn )
{
    // Only assign if valid.
    _valid = IsValid( ssn.c_str() );
    if ( _valid )
        _ssn = ssn;
}
SSNValidator::SSNValidator( const SSNValidator& aCopy )
{
    _ssn = aCopy._ssn;
    _valid = aCopy._valid;
}
SSNValidator::~SSNValidator()
{}
void SSNValidator::setSSN( const char *ssn)
{
    // Only assign if valid.
    if ( IsValid( ssn ) )
    {
        _valid = true;
        _ssn = ssn;
    }
}
// Operators for this class
SSNValidator SSNValidator::operator=( const char *ssn )
{
    // Only assign if valid.
    if ( IsValid( ssn ) )
    {
        _valid = true;
        _ssn = ssn;
    }
    return *this;
}
```

```
SSNValidator SSNValidator::operator=( const std::string& ssn )
{
    // Only assign if valid.
    if ( IsValid( ssn.c_str() ) )
    {
        _valid = true;
        _ssn = ssn;
    }
    return *this;
}
SSNValidator SSNValidator::operator=( const SSNValidator& aCopy )
{
    _valid = aCopy._valid;
    _ssn = aCopy._ssn;
    return *this;
}
SSNValidator::operator const char *()
{
    return _ssn.c_str();
}
```

5. Lưu mã và đóng code editor.

File mà chúng ta vừa định nghĩa sẽ là một kiểu thực sự mà bạn có thể sử dụng trong các ứng dụng riêng của bạn để lưu trữ và hiệu lực hoá Social Security Number (SSN). Bạn sẽ không bao giờ phải viết lại mã để kiểm tra chiều dài của một mục nhập (entry) hoặc nội dung của nó để xem nó có thể là một giá trị SSN hợp lệ hay không.

Chú ý rằng các hằng được cung cấp cho chiều dài của SSN và dấu tách của nó (xem các dòng được đánh dấu (→ **1** và → **2**). Điều này cho phép bạn dễ dàng chỉnh sửa mã nếu định dạng SSN thay đổi theo thời gian. Một ngày nào đó bạn có thể cần thay đổi SSN để sử dụng thêm chữ số hoặc để định dạng bằng một dấu tách khác. Chuẩn bị cho điều này bây giờ sẽ tiết kiệm cho bạn lượng thời gian lớn sau này.

Test lớp SSN Validator

Sau khi tạo lớp Validator, bạn nên tạo một driver thử nghiệm để bảo đảm mã của bạn chính xác và hướng dẫn mọi người cách sử dụng mã của bạn.

Tạo một driver thử nghiệm sẽ minh hoạ sự hiệu lực hoá các loại dầu vào khác nhau từ người dùng và sẽ cho thấy lớp Validator được sử dụng như thế nào. Driver sẽ chứa một số phéptest cơ bản của lớp cũng như chấp nhận các Social Security Number từ người dùng để thấy chúng có hợp lệ hay không.

Trong ví dụ này, chúng ta tạo một driver thử nghiệm vốn thực hiện hai điều. Thứ nhất, nó tạo một loạt các phép test chuẩn để minh hoạ các mục nhập tốt và xấu dự tính cho kiểu. Thứ hai, driver thử nghiệm cho phép nhà lập trình thử các kiểu mục nhập khác để thấy lớp có lưu trữ chúng hay không.

1. Trong code editor mà bạn chọn, mở lại file để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này chương trình thử nghiệm được đặt tên là ch26.cpp.

2. Thêm mã từ Listing 26.3 vào file driver thử nghiệm, thay thế các tên mà bạn đã sử dụng cho định nghĩa lớp SSN ở nơi thích hợp.

Listing 26.3: Mã driver thử nghiệm

```
const char *TrueOrFalse( bool value )
{
    if ( value )
        return "TRUE";
    return "FALSE";
}
void DoValidTest( const char *strName, SSNValidator& ssn, bool expected_result
)
{
    bool bValid = ssn.Valid();
    printf("%s: Result %s. Expected Result: %s. %s\n", strName,
        TrueOrFalse( bValid ), TrueOrFalse( expected_result ),
        ( bValid == expected_result ? "PASS" : "FAIL" ) );
}
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        printf("Usage: ch3_15 ssn1 [ssn2]...\n");
```

```
        exit(1);
    }
    for ( int i=1; i<argc; ++i )
    {
        SSNValidator ssn(argv[i]);
        if ( ssn.Valid() )
            printf("%s is a valid Social Security Number\n", ssn.SSN().c_str() );
        else
            printf("%s is NOT a valid Social Security Number\n", argv[i] );
    }
    // Do some generic testing.
    SSNValidator ssnNULL( NULL );
    DoValidTest( "NULL Test", ssnNULL, false );
    SSNValidator ssnGood("000-00-0000");
    DoValidTest( "Good Test", ssnGood, true );
    SSNValidator ssnBad("0000a0000");
    DoValidTest( "Bad Test", ssnBad, false );
    return 0;
}
```

3. Lưu file driver trong code editor và đóng chương trình code editor.

4. Biên dịch chương trình thử nghiệm bằng trình biên dịch mà bạn chọn và chạy nó trên hệ điều hành mà bạn chọn.

   Nhập các đối số dòng lệnh chẳng hạn như

   123456789 000-00-0000 0909 a 12345678

   012-03-3456

Đây đơn giản là các dạng Social Security Number, một số hợp lệ và một số không hợp lệ. Đối số thứ nhất chứa 9 chữ số và không có ký tự chữ - số nào sẽ hợp lệ. Đối số thứ ba không chứa 9 chữ số và do đó không hợp lệ. Dạng thứ tư chứa một ký tự không hợp lệ (a). Các mục nhập thứ hai và thứ ba trông hợp lệ, nhưng chúng ta không xử lý ký tự dấu gạch, do đó chúng sẽ được xem là không hợp lệ bởi chương trình.

Nếu chương trình làm việc tốt, bạn sẽ thấy kết quả từ driver thử nghiệm như được minh hoạ trong Listing 26.4.

Listing 26.4: Kết quả từ driver thử nghiệm

```
$ ./a 123456789 000-00-000 0909 a12345678
    01-02-2345
123456789 is a valid Social Security Number
000-00-000 is NOT a valid Social Security
    Number
0909 is NOT a valid Social Security Number
a12345678 is NOT a valid Social Security
    Number
01-02-2345 is NOT a valid Social Security
    Number
NULL Test: Result FALSE. Expected Result:
    FALSE. PASS
Good Test: Result TRUE. Expected Result:
    TRUE. PASS
Bad Test: Result FALSE. Expected Result:
    FALSE. PASS
```

Như bạn có thể thấy qua kết quả, đầu tiên chương trình kiểm tra các đối số đầu vào từ người dùng. Như chúng ta mong đợi, chỉ giá trị đầu vào đầu tiên hợp lệ. Tất cả mục nhập còn lại không hợp lệ. Các phép test còn lại đơn giản hiệu lực hoá rằng các điều kiện đã biết làm việc tốt.

Kỹ thuật 27: Xây dựng một lớp Date

Tiết kiệm thời gian bằng cách

- Tạo một lớp Date generic
- Thực thi chức năng ngày tháng vào lớp
- Test lớp

Một trong những tác vụ thông thường nhất mà bạn sẽ gặp phải khi một nhà lập trình làm việc với các ngày tháng. Bây giờ bạn tính toán khi nào một thứ gì đó được mua hoặc được bán hoặc hiệu lực hoá đầu vào từ người dùng, ứng dụng của bạn sẽ cần hỗ trợ các ngày tháng. Thư viện C chuẩn chứa các thường trình khác nhau để làm việc với các ngày tháng chẳng hạn như các hàm time và localtime. Tuy nhiên, vấn đề là những thường trình này không thực hiện việc hiệu lực hoá

thích hợp - và đối với vấn đề đó, chúng không dễ sử dụng. Do đó sẽ tuyệt vời nếu một lớp thực thi tất cả chức năng ngày tháng mà chúng ta muốn trong các ứng dụng. Bằng cách tạo một lớp vốn có thể dễ dàng được mang chuyển từ project này sang project khác, bạn sẽ tiết kiệm thời gian trong các giai đoạn phát triển, thiết kế và test của project.

Bởi vì các ngày tháng là một khối tạo cơ bản của các ứng dụng, sẽ hợp lý nếu tạo một lớp xử lý chúng và hiệu lực hoá chúng. Nếu bạn lập một danh sách tất cả chức năng cơ bản mà bạn muốn trong một lớp như vậy, có lẽ có một điều gì đó như sau:

- Hiệu lực hoá các ngày tháng
- Thực hiện các phép tính ngày tháng
- Tính ngày của tuần
- Trả về các tên ngày và tháng
- Chuyển đổi các ngày tháng dạng số thành các chuỗi

Có nhiều hàm khác hữu dụng, nhưng những hàm này quan trọng nhất trong bất kỳ ứng dụng. Trong kỹ thuật này, chúng ta sẽ xem những cách tận dụng một lớp Date trong các ứng dụng riêng của chúng ta và cách thực thi chức năng cần thiết để thực hiện mọi thứ trên danh sách đặc tính cho một lớp Date.

Tạo lớp Date

Làm theo các bước sau đây để tạo lớp Date cá nhân riêng của bạn:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho lớp Date.

   Trong ví dụ này, file được đặt tên là ch27.h mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 27.1 vào file.

Listing 27.1: Định nghĩa lớp Date

```
#ifndef CH27H_
#define CH27H_
#include <string>
const int MaxMonths = 12;
const int MaxYear = 9999;
typedef enum
{
    MMDDYYYY = 0,
    DDMMYYYY = 1,
```

```
        YYYYMMDD = 2
    } DateFormat;
    class Date
    {
    private:
        // Store dates in Julian format.
        long _julian;
        // The month of the year (0-11)
        int _month;
        // The day of the month (0-30)
        int _day_of_month;
        // The day of the week (0-6)
        int _day_of_week;
        // The year of the date (0-9999)
        int _year;
        // A string representation of the date
        std::string _string_date;
        // The format to use in the date
        DateFormat _format;
        // See whether a given date is valid.
        bool IsValid(int m, int d, int y);
        // Compute the day of the week.
        int DayOfWeek( int m, int d, int y );
        // Convert to Julian format.
        long ToJulian();
        // Convert from a Julian format.
        void FromJulian();
        // Initialize to defaults.
        void Init(void);
        // Make a copy of this date.
        void Copy( const Date& aCopy );
        // Convert to a string.
        const char *ToString();
    public:
        // Constructors and destructors
```

```
Date();
Date( int m, int d, int y );
Date( const Date& aCopy );
Date( long julian );
virtual ~Date();
// Operators.
// Assignment operator
Date operator=( const Date& date );
// Conversion to Julian date
operator long();
// Conversion to a string
operator const char *();
// Accessors
int Month() { return _month; };
int DayOfMonth() { return
_day_of_month; };
int DayOfWeek() { return
_day_of_week; };
int Year() { return _year; };
const char *AsString() { return
_string_date.c_str(); };
DateFormat Format() { return _format; };
void setMonth( int m );
void setDayOfMonth( int _day_of_month );
void setYear( int y );
void setFormat( const DateFormat& f );
// Operations
// Is a given year a leap year?
bool isLeapYear(int year) const;
// Is this date a leap year?
bool isLeapYear(void) const;
// Return the number of days in a given
month.
int numDaysInMonth( int month, int year
);
```

```
    // Return the number of days in the current
    month.
    int numDaysInMonth( void );
    // Some useful operators for manipulation
    Date operator+(int numDays);
    Date operator+=(int numDays);
    Date operator-(int numDays);
    Date operator-=(int numDays);
};
#endif
```

3. Lưu mã trong code editor và đóng file.

   File mà bạn vừa tạo là file header và giao diện cho lớp. Đây là những gì mà "công chúng" sẽ thấy khi họ muốn sử dụng lớp. Do đó, bước kế tiếp là thực thi chức năng của chính lớp.

4. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của lớp Date.

   Trong ví dụ này, file được đặt tên là ch27.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

5. Gõ nhập mã từ Listing 27.2 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này. Thay đổi tên của các hằng và biến khi bạn chọn.

   Đây là tất cả hằng và định nghĩa mà chúng ta sẽ sử dụng cho lớp. Bước kế tiếp là thêm phần thực thi thực sự.

Listing 27.2: File nguồn lớp Date

```
#include "ch27.h"
// Some information we need.
const char *MonthNames[] = {
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
```

```
        "September",
        "October",
        "November",
        "December"
};
int MonthDays[] =
{
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
char *DayNames[] = {
        "Sunday",
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday"
};
#define OCT5_1582 (2299160L)          // "really" 15-Oct-1582
#define OCT14_1582      (2299169L)          // "really" 4-Oct-1582
#define JAN1_1          (1721423L)
#define YEAR            (365)
#define FOUR_YEARS      (1461)
#define CENTURY         (36524L)
#define FOUR_CENTURIES (146097L)
static int DaysSoFar[][13] =
{
        {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365},
        {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}
        };
```

Thực thi chức năng Date

Sau khi đã định nghĩa lớp, đến lúc phải thực thi chức năng thực sự cho lớp đó. Trong trường hợp này chúng ta muốn thêm tất cả lôgíc để xử lý và định nghĩa các ngày tháng. Hãy làm điều đó bây giờ.

1. Mở lại file nguồn lớp Date (được đặt tên là ch27.cpp). Thêm mã từ Listing 27.3 vào file.

Listing 27.3: Chức năng Date.

```
void Date::Copy( const Date& aCopy )
{
    _julian = aCopy._julian;
    _month = aCopy._month;
    _day_of_month = aCopy._day_of_month;
    _day_of_week = aCopy._day_of_week;
    _year = aCopy._year;
    _format = aCopy._format;
    _string_date = aCopy._string_date;
}
void Date::Init()                                    → 1
{
    _julian = 0;
    _month = 1;
    _day_of_month = 1;
    _day_of_week = 0;
    _year = 2004;
    _format = MMDDYYYY;
    _string_date = AsString();
}
int Date::DayOfWeek( int m, int d, int y)
{
    _day_of_week = ((_julian + 2) % 7 + 1);
       return day_of_week;
}
bool Date::IsValid(int m, int d, int y)              → 2
{
    // Check the year.
    if ( y < 0 || y > MaxYear )
       return false;
    // Do the month.
```

```
    if ( m < 1 || m > MaxMonths )
        return false;
    // Finally, do the day of the month. First, the easy check...
    if ( d < 1 || d > 31 )
        return false;
    // Now, check the days per THIS month.
    int daysPerMonth = MonthDays[ m ];
    if ( isLeapYear( y ) )
        if ( m == 2 )
            daysPerMonth ++;
    if ( d > daysPerMonth )
        return false;
    // Looks good.
    return true;
}
long Date::ToJulian()                                    → 3
{
    int         a;
    int         work_year=_year;
    long        j;
    int         lp;
    // Correct for negative year (-1 = 1BC = year 0).
    if (work_year < 0)
        work_year++;
    lp = !(work_year & 3);                  // lp = 1 if this is a leap year.
    j =
        ((work_year-1) / 4)         +               // Plus ALL leap years...
        DaysSoFar[lp][_month-1] +
        _day_of_month           +
        (work_year * 365L)      +           // Days in years
        JAN1_1                  +
        -366; // adjustments
    // Deal with Gregorian calendar
    if (j >= OCT14_1582)
    {
```

```
        a = (int)(work_year/100);
        j = j+ 2 - a + a/4;                    // Skip days that didn't exist.
    }
    _julian = j;
    return _julian;
}
void Date::FromJulian()                                    → 4
{
    long      z,y;
    short     m,d;
    int       lp;
    z = _julian+1;
    if (z >= OCT5_1582)
    {
        z -= JAN1_1;
        z = z + (z/CENTURY) - (z/FOUR_CENTURIES) -2;
        z += JAN1_1;
    }
    z = z - ((z-YEAR) / FOUR_YEARS); // Remove leap years before the
    current year.
    y = z / YEAR;
    d = (short) (z - (y * YEAR));
    y = y - 4712;                  // This is our base year in 4713 B.C.
    if (y < 1)
        y--;
    lp = !(y & 3);                 // lp = 1 if this is a leap year.
    if (d==0)
    {
        y--;
        d = (short) (YEAR + lp);
    }
    m = (short) (d/30);            // This is a guess at the month.
    while (DaysSoFar[lp][m] >=d)
        m--;                       // Correct guess.
    d = (short) (d - DaysSoFar[lp][m]);
```

```
        _day_of_month = d;
        _month = (short) (m+1);
        if ( _month > 12 )
        {
            _month = 1;
            y ++;
        }
        _year = (short) y;
        _day_of_week = DayOfWeek( _month, _day_of_month, _year );
}
Date::Date()
{
        Init();
        ToString();
}
Date::Date( int m, int d, int y )
{
        Init();
        if ( IsValid( m, d, y ) )
        {
            _day_of_month = d;
            _month = m;
            _year = y;
            _julian = ToJulian();
            ToString();
            _day_of_week = DayOfWeek( _month, _day_of_month, _year );
        }
}
Date::Date( const Date& aCopy )
{
        Init();
        Copy( aCopy );
}
Date::Date( long julian )
{
```

```
    Init();
    _julian = julian;
    FromJulian();
    ToString();
}
Date::~Date()
{
    Init();
}
Date Date::operator=( const Date& date )
{
    Copy( date );
    return *this;
}
// Conversion to Julian date
Date::operator long()
{
    return _julian;
}
// Conversion to a string
Date::operator const char *()
{
    return _string_date.c_str();
}
void Date::setMonth( int m )
{
    if ( m < 0 || m > MaxMonths )
        return;
    _month = m;
}
void Date::setDayOfMonth( int d )
{
    if ( d < 1 || d > 31 )
        return;
    // Now check the days per THIS month.
```

```
    int daysPerMonth = MonthDays[ _month ];
    if ( isLeapYear( _year ) )
        if ( _month == 2 )
            daysPerMonth ++;
    if ( d > daysPerMonth )
        return;
    _day_of_month = d;
}
void Date::setYear( int y )
{
    if ( y < 0 || y > MaxYear )
        return;
    _year = y;
}
void Date::setFormat( const DateFormat& f )
{
    _format = f;
}
bool Date::isLeapYear( void ) const
{
    return ( (_year >= 1582) ?
            (_year % 4 == 0 && _year % 100 != 0 || _year % 400 == 0 ):
            (_year % 4 == 0) );
}
bool Date::isLeapYear( int year ) const
{
    return ( (year >= 1582) ?
            (year % 4 == 0 && year % 100 != 0 || year % 400 == 0 ):
            (year % 4 == 0) );
}
// Return the number of days in a given month.
int Date::numDaysInMonth( int m, int y )
{
    // Validate the input.
    // Check the year.
```

```
    if ( y < 0 || y > MaxYear )
        return -1;
    // Do the month.
    if ( m < 1 || m > MaxMonths )
        return -1;
    int daysPerMonth = MonthDays[ m ];
    if ( isLeapYear( y ) )
        if ( m == 2 )
            daysPerMonth ++;
    return daysPerMonth;
}
// Return the number of days in the current month.
int Date::numDaysInMonth( void )
{
    int daysPerMonth = MonthDays[ _month ];
    if ( isLeapYear( _year ) )
        if ( _month == 2 )
            daysPerMonth ++;
    return daysPerMonth;
}
Date Date::operator+(int numDays)
{
    long j = _julian;
    j += numDays;
    Date d(j);
    return d;
}
Date Date::operator+=(int numDays)
{
    _julian += numDays;
    FromJulian();
    ToString();
    return *this;
}
Date Date::operator-(int numDays)
```

```
{
    long j = _julian;
    j -= numDays;
    Date d(j);
    return d;
}
Date Date::operator-=(int numDays)
{
    _julian -= numDays;
    FromJulian();
    ToString();
    return *this;
}
const char *Date::ToString()
{
    char szBuffer[ 256 ];
    switch ( _format )
    {
       case MMDDYYYY:
    sprintf(szBuffer, "%02d/%02d/%02d", _month, _day_of_month, _year );
       break;
    case DDMMYYYY:
    sprintf(szBuffer, "%02d/%02d/%02d", _day_of_month, _month, _year );
       break;
    case YYYYMMDD:
    sprintf(szBuffer, "%02d/%02d/%02d", _year, _month, _day_of_month );
       break;
    default:
    sprintf(szBuffer, "%02d/%02d/%02d", _month, _day_of_month, _year );
       break;
    }
    _string_date = szBuffer;
    return _string_date.c_str();
}
```

Bây giờ có nhiều mã để xử lý. Đừng bận tâm - mã phân tích thành ba phần riêng biệt:

- Mã khởi tạo (được minh hoạ tại → 1) set hoặc get các biến thành viên riêng lẻ và khởi tạo chúng sang những giá trị mặc định hợp lý.
- Mã hiệu lực hoá (được minh hoạ tại → 2) kiểm tra xem dữ liệu đầu vào có hợp lý hay không với những qui tắc và các xác lập hiện hành.
- Mã thuật toán (được minh hoạ tại → 3 và → 4) thực hiện việc xử lý và tính toán ngày tháng thật sự.

2. Lưu file mã nguồn và đóng code editor.
3. Biên dịch mã thử nghiệm để bảo đảm tất cả mã được nhập đúng và chính xác.

Test lớp Date

Như với bất kỳ lớp tiện ích khác, sau khi bạn đã viết mã cho lớp, bạn phải có khả năng cung cấp một driver thử nghiệm cho lớp đó. Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ mã làm việc tốt và hướng dẫn những nhà lập trình khác cách sử dụng lớp trong các ứng dụng riêng của họ.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho driver thử nghiệm.

   Trong ví dụ này, file được đặt tên là ch27.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 27.4 vào file.

   Tốt hơn hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này. Đổi tên của các hằng và biến khi bạn chọn.

Listing 27.4: Mã driver thử nghiệm Date.

```
#include <stdio.h>
#include "date.h"
void DumpDate( Date& d )
{
    printf("Date:\n");
    printf("As String: %s\n", d.AsString() );
    printf("Month: %d\n", d.Month() );
    printf("Day : %d\n", d.DayOfMonth() );
    printf("Day of Week: %d\n", d.DayOfWeek() );
    printf("Year: %d\n", d.Year() );
```

```
    printf("Leap Year: %s\n", d.isLeapYear() ? "Yes" : "No" );
    printf("Number of days in this month: %d\n", d.numDaysInMonth() );
}
int main()
{
    // Initialized date to no values.
    Date d1;
    // Initialize to the end of the year to test edge cases.
    Date d2(12,31,2004);
    // Print out the dates as strings for testing.
    printf("D1 as string: %s\n", d1.AsString() );
    printf("D2 as string: %s\n", d2.AsString() );
    // Test year wrap and the operator +=.
    d2 += 1;
    printf("D2 as string: %s\n", d2.AsString() );          → 6
    // Test backward year wrap and the operator -=.
    d2 -= 1;
    printf("D2 as string: %s\n", d2.AsString() );
    // Test the assignment operator.
    Date d3 = d2;
    // Check to see whether the class works properly for
    // assigned objects.
    d3 -= 10;
    printf("D3 as string: %s\n", d3.AsString() );
    // Validate the day of the week.
    Date d4 (7,27,2004);
    printf("D4, day of week = %d\n", d4.DayOfWeek() );
    // Test the pieces of the date.
    Date d5;
    d5.setMonth( 11 );
    d5.setDayOfMonth( 31 );
    d5.setYear( 2004 );
    d5.setFormat( YYYYMMDD );
```

```
    DumpDate( d5 );
    return 0;
}
```

3. Lưu mã dưới dạng một file trong editor và đóng code editor.

4. Biên dịch và chạy ứng dụng.

Nếu bạn đã làm tốt mọi thứ và mã hoạt động chính xác, bạn sẽ thấy kết quả như sau:

```
$ ./a.exe
D1 as string: 01/01/2004                    → 5
D2 as string: 12/31/2004
D2 as string: 01/01/2005
D2 as string: 12/31/2004
D3 as string: 12/21/2004                    → 7
D4, day of week = 3
Date:
As String: 2004/12/31
Month: 12
Day : 31
Day of Week: 0
Year: 2004
Leap Year: Yes
Number of days in this month: 31
```

Có một số điều quan trọng cần chú ý từ kết quả này. Đầu tiên, hãy xem dòng được đánh dấu là → **5**trong listing. Dòng này được xuất cho đối tượng ngày tháng vốn được định nghĩa với constructor void. Như bạn có thể thấy đối tượng được khởi tạo phù hợp với một ngày tháng hợp lệ. Tiếp theo hãy xem dòng được đánh dấu là → **6**. Dòng này được xuất sau khi một ngày được cộng với ngày tháng 12/31/2004. Rõ ràng, điều này buộc ngày tháng bao bọc sang năm kế tiếp mà chúng ta có thể kiểm chứng bằng cách xem kết quả, thể hiện 01/01/2005. Bằng cách xem một lịch và chúng ta cũng có thể kiểm chứng rằng ngày tháng được thể hiện tại → **7** thực sự rơi vào một ngày thứ ba (3 trong kết quả). Sau cùng, chúng ta chạy một số phép test đơn giản để kiểm chứng số ngày trong tháng là chính xác cho tháng 12, các phần của ngày tháng được phân tích đúng và phép tính năm nhuần chính xác.

Tất cả dữ liệu xuất này cho phép chúng ta xác thực lớp làm việc tốt và chức năng có thể dễ dàng được di chuyển từ project này sang project khác. Điều này sẽ tiết kiệm cho chúng ta nhiều thời gian và cho phép chúng ta thiết kế các chương trình với chức năng ngày tháng đã tạo sẵn.

Kỹ thuật 28: Ghi đè chức năng với các phương thức ảo

Tiết kiệm thời gian bằng cách:

- Sử dụng các mẫu factory
- Xây dựng một lớp manager
- Test lớp manager

Một trong những "mẫu" phát triển phần mềm phổ biến nhất là mẫu factory. Nó là một phương pháp phát triển phần mềm làm việc như một factory (xí nghiệp): Bạn tạo các đối tượng từ một mô hình của một loại đối tượng cụ thể và mô hình đó định nghĩa những gì mà mô hình đó có thể thực hiện. Nói chung cách điều này thực thi là bạn tạo một lớp factory cấp phát, huỷ cấp phát và theo dõi một lớp cơ sở nhất định của các đối tượng. Lớp factory này thực sự chỉ hiểu cách quản lý kiểu đối tượng vốn hình thành một cơ sở cho tất cả đối tượng khác trong cây lớp. Tuy nhiên, thông qua điều kỳ diệu của các phương thức ảo (virtual methods), có thể quản lý tất cả đối tượng. Hãy xem điều này thực hiện như thế nào. Bằng cách tạo một factory, sử dụng các phương thức ảo xử lý nhiều kiểu đối tượng khác nhau, chúng ta sẽ tiết kiệm thời gian bằng việc không cần phải tái thực thi việc xử lý này mỗi lần chúng ta cần.

Đầu tiên chúng ta có một lớp quản lý một lớp cơ sở nào đó của các đối tượng - nó được gọi là một factory. Nó sử dụng các phương thức ảo để quản lý các đối tượng - nghĩa là để thêm các đối tượng mới, loại bỏ chúng, đưa chúng trở về người dùng và báo cáo về những đối tượng nào đang sử dụng và không đang sử dụng.

Tiếp theo, chúng ta có một tập hợp lớp dẫn xuất. Những lớp này ghi đè chức năng của lớp cơ sở bằng cách sử dụng các phương thức ảo để hoàn thành những tác vụ khác nhau. Ví dụ, hãy xem ý tưởng về các loại lớp khác nhau để đọc các kiểu file khác nhau. Chúng ta có một lớp cơ sở vốn được gọi là một lớp FileProcessor. Trình quản lý (manager) phải là một lớp FileProcessorManager. Trình quản lý tạo các

FileProcessor khác nhau dựa vào kiểu file được cần đến, tạo chúng nếu cần thiết hoặc trả về một FileProcessor vốn không đang được sử dụng.

Tạo một lớp factory

Bước đầu tiên để quản lý và xử lý các đối tượng là tạo một lớp factory làm việc với một lớp cơ sở generic. Các bước sau đây hướng dẫn bạn cách tạo một lớp như vậy vốn tận dụng những phương thức ảo để tạo, thêm và xoá các đối tượng. Trong trường hợp này, chúng ta tạo một lớp cơ sở được gọi là Object mà từ đó tất cả đối tượng được quản lý sẽ được dẫn xuất.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của mã factory.

   Trong ví dụ này, file được đặt tên là ch28.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 28.1 vào file.

Listing 28.1: Mã nguồn lớp cơ sở

```
#include <stdio.h>
#include <string>
#include <vector>
 lass Object
{
private:
    std::string _name;
    bool _inUse;
public:
    Object(void)
    {
        _name = "Object";
        _inUse = false;
    }
    Object( const char *name )
    {
        _name = name;
        _inUse = false;
    }
```

```
    Object( const Object& aCopy )
    {
        _name = aCopy._name;
        _inUse = aCopy._inUse;
    }
    virtual ~Object()
    {
    }
    virtual void MarkInUse( bool bFlag )
    {
        _inUse = bFlag;
    }
    virtual bool InUse( void )
    {
        return _inUse;
    }
    virtual const char *Name(void)
    {
        return _name.c_str();
    }
    virtual void Report() = 0;
};
class MyObject1 : public Object
{
public:
    MyObject1()
        : Object ("MyObject1")
    {
    }
    virtual void Report()
    {
        printf("I am a MyObject1 Object\n");
    }
};
class MyObject2 : public Object
```

```
{
public:
    MyObject2()
        : Object ("MyObject2")
    {
    }
    virtual void Report()
    {
        printf("I am a MyObject2 Object\n");
    }
};
class MyObject3 : public Object
{
public:
    MyObject3()
        : Object ("MyObject3")
    {
    }
    virtual void Report()
    {
        printf("I am a MyObject3 Object\n");
    }
};
class Factory
{
private:
    std::vector< Object *> _objects;
public:
    Factory()
    {
    }
    // Method to add an object to the pool
    virtual void Add( Object *obj )
    {
        obj->MarkInUse( true );
```

```
            _objects.insert( _objects.end(), obj );
        }
        // Method to retrieve an object not in use
        virtual Object *Get( void )
        {
            std::vector< Object *>::iterator iter;
            for ( iter = _objects.begin(); iter != _objects.end(); ++iter )
            {
                if ( (*iter)->InUse() == false )
                {
                    printf("Found one\n");
                    // Mark it in use
                    (*iter)->MarkInUse( true );
                    // And give it back
                    return (*iter);
                }
            }
            // Didn't find one.
            return NULL;
        }
        virtual void Remove( Object *obj )
        {
            std::vector< Object *>::iterator iter;
            for ( iter = _objects.begin(); iter != _objects.end(); ++iter )
            {
                if ( (*iter) == obj )
                {
                    (*iter)->MarkInUse( false );
                    break;
                }
        }
    }
    virtual void Report()                                    → 1
    {
        std::vector< Object *>::iterator iter;
```

```
        for ( iter = _objects.begin(); iter != _objects.end(); ++iter )
        {
            if ( (*iter)->InUse() == true )
            {
                printf("Object at %lx in use\n", (*iter) );
            }
            else
            {
                printf("Object at %lx NOT in use\n", (*iter) );
            }
            (*iter)->Report();
        }
    }
};
```

3. Lưu file sang đĩa và đóng code editor.
4. Biên dịch ứng dụng trên hệ điều hành mà bạn chọn, sử dụng trình biên dịch mà bạn chọn.

Test factory

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã. Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm đơn giản để minh hoạ lớp factory tương tác với những đối tượng dẫn xuất thông qua các phương thức ảo như thế nào.

1. Trong code editor mà bạn chọn, mở file nguồn để chứa mã cho driver thử nghiệm.

   Trong ví dụ này file được đặt tên là ch28.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 28.2 vào file.

Listing 28.2: Driver thử nghiệm cho đối tượng factory

```
int main()
{
    // Implement an object factory object
    Factory f;
    // Add some objects to the factory
```

```
        MyObject1 *obj1 = new MyObject1;
        MyObject2 *obj2 = new MyObject2;
        MyObject3 *obj3 = new MyObject3;
        f.Add( obj1 );
        f.Add( obj2 );
        f.Add( obj3 );
        // Remove one to simulate the destruction
        of an object
        f.Remove( obj1 );
        // Now try to get a new one back.
        Object *pObject = f.Get();
        printf("I got back a %s object\n",
    pObject->Name() );
        // Generate a report to see what is in
        use.
        f.Report();
}
```

3. Lưu file và đóng code editor.
4. Biên dịch toàn bộ chương trình và chạy nó trong hệ điều hành mà bạn chọn.

   Bạn sẽ thấy kết quả sau đây nếu bạn đã làm đúng mọi thứ. Chú ý rằng phụ thuộc vào hệ điều hành và phần cứng, các số thực sự thể hiện cho các địa chỉ sẽ khác nhau.

```
$ ./a.exe
Found one
I got back a MyObject1 object
Object at a050230 in use
I am a MyObject1 Object
Object at a050008 in use
I am a MyObject2 Object
Object at a050638 in use
I am a MyObject3 Object
```

Kết quả cho chúng ta thấy trình quản lý đang theo dõi các lớp dẫn xuất Object cơ sở khác và tạo chúng chỉ khi cần thiết. Như bạn có thể thấy, các phương thức ảo cho phép chúng ta tạo kiểu thích hợp cho lớp dẫn xuất cụ thể này và tạo chúng khi cần thiết.

Như bạn có thể thấy, factory manager có thể xử lý tất cả loại đối tượng khác nhau miễn là chúng được dẫn xuất từ một lớp cơ sở chung. Ngoài ra, các phương thức ảo có thể được sử dụng để phân biệt những đối tượng để cho các nhà lập trình khác biết những gì chúng ta có thể làm.

Kỹ thuật 29: Sử dụng các lớp mix-in

Tiết kiệm thời gian bằng cách

- Tìm hiểu các lớp mix-in
- Thực thi các lớp mix-in
- Test mã

Sự thừa kế (Inheritance) là một kỹ thuật cực kỳ mạnh trong C++. Tuy nhiên, vấn đề của sự thừa kế là bạn phải cho người dùng cuối truy cập tất cả phương thức public của một lớp hoặc ghi đè chúng một cách private (riêng) để "ẩn" chúng không được sử dụng bởi người dùng cuối. C++ tiến hành một phương pháp đòi hỏi nỗ lực tối đa để dẫn xuất các lớp bằng sự thừa kế. Phương pháp này hầu như không phải là một kỹ thuật tối ưu bởi vì việc loại bỏ chức năng không mong muốn ra khỏi một lớp vốn chứa nhiều phương thức đòi hỏi nhiều công việc hơn là tái tạo lớp ngay từ đầu. Ví dụ, nếu bạn thừa kế từ một lớp vốn chứa một phương thức print và bạn không muốn phương thức đó được phơi bày ra trước người dùng cuối, bạn phải làm ẩn phương thức bằng cách tạo một phiên bản private mới của nó. Điều này không quá khó khi chỉ có một phương thức như vậy và khi có hàng chục phương thức, tạo một lớp mới sẽ hợp lý hơn.

Thật may thay, C++ cung cấp một lựa chọn khác: lớp mix-in (hỗn hợp). Sau đây là cách làm việc của nó: Cách dễ nhất để giới hạn chức năng mà bạn cung cấp từ một lớp cơ sở là sử dụng lớp đó làm một thành viên dữ liệu của lớp thừa kế - và cho người dùng cuối chỉ truy cập các phương thức mà bạn muốn họ sử dụng thay vì cung cấp tất cả phương thức và loại bỏ những phương thức mà bạn không muốn sử dụng. Phương pháp này đặc biệt hữu dụng khi bạn có các lớp nhỏ mà bạn muốn khởi tạo và giới hạn (để chỉ bạn truy cập chúng) hoặc các lớp có toàn bộ chức năng nhiều hơn bạn cảm thấy cung cấp thoải mái (hoặc quá phức tạp để người dùng cuối xử lý). Lớp cơ sở nhúng là một lớp mix-in cho một lớp thừa kế.

Các lớp mix-in được thực thi dưới dạng các thành viên dữ liệu của lớp vốn cung cấp toàn bộ chức năng và được sử dụng để mở rộng chức năng đó. Những ưu điểm của kỹ thuật mix-in rõ ràng: nó cho người

dùng truy cập những tính năng mà bạn muốn sử dụng, bạn có thể giới hạn những gì người dùng có thể truy cập và bạn có thể đơn giản hoá những phương thức được cung cấp bằng việc cung cấp các wrapper riêng của bạn có những xác lập mặc định. Khi lớp mix-in được nhúng trong một lớp, người dùng có thể thể hiện cụ thể, bạn có thể điều khiển những phương thức nào trong lớp mix-in có sẵn. Để làm điều này, bạn có thể đơn giản viết các phương thức accessor vốn cho phép người dùng cuối truy cập những phương thức mà bạn muốn họ sử dụng trong lớp mix-in. Điều này có một số lợi ích. Đầu tiên bạn điều khiển sự truy cập nào mà người dùng có đối với chức năng. Thứ hai, nếu bạn thay đổi cách làm việc của lớp mix-in nhúng, người dùng cuối không bị ảnh hưởng. Sau cùng, bạn có thể điều chỉnh chức năng của lớp mix-in theo những nhu cầu riêng biệt của bạn, tuỳ biến hành vi của nó trong các phương thức wrapper. Bởi vì bạn không cần phải viết toàn bộ chức năng được cung cấp bởi lớp mix-in, bạn tiết kiệm nhiều thời gian và người dùng có được một hệ thống được gỡ rối đầy đủ, tiết kiệm cho họ thời gian.

Thực thi các lớp mix-in

Giả sử bạn muốn thêm khả năng lưu dữ liệu ở một trong các lớp. Bạn có thể thêm một lớp cơ sở được gọi là C vốn cho phép dữ liệu được ghi sang một file. Lớp này làm tất cả công việc quản lý file xuất, ghi sang nó và đóng nó. Sau đó, bạn có thể tạo một lớp mix-in để làm chức năng save và sau đó minh hoạ chức năng đó được sử dụng trong một lớp dẫn xuất như thế nào.

Để thực thi một lớp mix-in, bạn đơn giản làm các bước sau đây trong lớp hiện có riêng của bạn:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch29.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 29.1 vào file.

Listing 29.1: Lớp Mix-In

```
#include <stdio.h>
#include <string>
class Save
{
    FILE *fp;
public:
    Save( void )
```

```
    {
        fp = NULL;
    }
    Save( const char *strFileName )
    {
        fp = fopen( strFileName, "w" );                    → 1
    }
    virtual ~Save()
    {
        if ( fp )
            fclose(fp);                                    → 2
    }
    void Write( const char *strOut )                       → 3
    {
        if ( fp )
            fprintf(fp, "%s\n", strOut );
    }
    void Write( int i )
    {
        if ( fp )
            fprintf(fp, "%d\n", i );
    }
    void Write( double d )
    {
        if ( fp )
            fprintf(fp, "%ld\n", d);
    }
    FILE *getFilePointer()
    {
        return fp;
    }
};
class MyClass
{
```

```
private:
    Save s;
public:
    MyClass( void )
        : s("test.txt")                          → 4
    {
        s.Write("Start of MyClass");             → 5
    }
    MyClass( const char *strFileName )
        : s(strFileName)
    {
        s.Write("Start of MyClass");
    }
    virtual ~MyClass()
    {
        s.Write("End of My Class");              → 6
    }
    void Log( const char *strLog )
    {
        s.Write(strLog);
    }
};
int main(int argc, char **argv)
{
    MyClass mc;
    for ( int i=0; i<argc; ++i )
    mc.Log( argv[i] );
    return 0;
}
```

Trong listing ở trên, chức năng Save được thực thi trong một lớp mix-in vốn được sử dụng bởi lớp MyClass để cho người dùng cuối khả năng lưu dữ liệu từ các biến thành viên MyClass. Chú ý rằng người dùng cuối không truy cập trực tiếp chức năng Save, nhưng thay vào đó sử dụng nó thông qua phương thức Log vốn tận dụng các chức năng Save nhưng không trực tiếp phơi bày chúng.

3. Lưu file nguồn trong code editor và đóng code editor.

Biên dịch và test các lớp mix-in

Hãy kiểm chứng rằng mã làm việc như minh hoạ và cho phép bạn lưu dữ liệu trong những đối tượng MyClass. Để làm điều này, chúng ta sẽ biên dịch và chạy chương trình, sau đó xem kết quả. Các bước sau đây hướng dẫn bạn cách làm:

1. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

Chú ý rằng chúng ta đã thực thi tất cả chức năng xử lý file - các chức năng open được minh hoạ tại → **1**), close (được minh hoạ tại → **2**) và save của file trong lớp mix-in Save. Lớp này giải quyết tất cả công việc riêng biệt của hệ điều hành là xử lý các file pointer. Lớp chính trong ví dụ MyClass đơn giản làm việc với lớp mix-in và giả định rằng nó biết những gì cần làm cho các tổ hợp hệ điều hành và môi trường khác nhau.

2. Chạy chương trình trong shell hệ điều hành mà bạn chọn.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ không nhận được kết quả từ chính chương trình. Thay vào đó, bạn thấy một file được định nghĩa trong lớp MyClass tại → **4** (được gọi là test.txt) được tạo ra trong hệ thống file thường trú trong thư mục mà bạn đã chạy chương trình trong đó. File này chứa kết quả được minh hoạ trong Listing 29.2.

Listing 29.2: File kết quả test.txt

```
Start of MyClass → 7
./a
End of My Class → 8
```

Như bạn có thể thấy từ kết quả, chương trình ghi chép một số dữ liệu của chính nó, biểu thị sự bắt đầu và sự kết thúc của thời gian sống của lớp. Ngoài ra, nó cho phép người dùng xuất các đối số sang chương trình.

Bởi vì chúng ta đã không cung cấp bất kỳ đối số, nó đơn giản xuất tên của file thực thi, đây là đối số đầu tiên cho tất cả chương trình.

Chú ý lớp ghi chép các hành động riêng của nó (xem → **5** và → **6**, những hành động này được thể hiện trong file kết quả tại → **7** và →**8**), cũng như các hành động của lớp mà nó được gọi từ đó. Đặc tính tiện lợi này cho bạn một nhật ký gỡ rối thiết yếu mà bạn có thể xem cách vận hành của chương trình.

Phần V
Các array và template

Kỹ thuật 30: Tạo một lớp khuôn mẫu đơn giản

Tiết kiệm thời gian bằng cách:

- Làm cho các lớp có thể tái sử dụng bằng cách làm cho chúng trở thành generic
- So sánh các pointer rỗng với các lớp khuôn mẫu
- Thực thi các lớp khuôn mẫu
- Tìm hiểu kết quả

Bởi vì các lớp có thể được tái sử dụng lặp đi lặp lại nhiều lần trong C++ và chúng ta muốn tạo một lớp chung ở mức độ có thể để nó được sử dụng trong phạm vi ứng dụng rộng nhất. Ví dụ, việc tạo một lớp vốn có thể in ra thông tin bản quyền chỉ cho một công ty riêng biệt sẽ không hợp lý lắm, chẳng hạn như:

```
(c) Copyright 2004 MySoftwareCompany Inc.
```

Sẽ hợp lý hơn khi tạo một lớp in ra một ký hiệu bản quyền, thêm tên công ty từ một file khởi tạo và sau đó thêm năm từ các đối số được chuyển vào (hoặc từ năm hiện hành như được xác định bởi đồng hồ máy tính). Cho phép dữ liệu được chèn từ các nguồn bên ngoài làm cho lớp trở nên generic (chung) hơn, điều này lần lượt làm cho nó khả dụng hơn qua các ứng dụng khác nhau.

Đây là ngay trọng tâm của cấu trúc C++, được gọi là các template (khuôn mẫu). Như với tên gọi, template là thứ có thể được tuỳ biến cho các dạng dữ liệu riêng biệt. Ví dụ, xem xét một lớp xử lý các pointer dẫn sang các kiểu dữ liệu nào đó. Lớp có kiểu dữ liệu được tạo mã cứng vào nó và xử lý chỉ kiểu pointer cụ thể đó. Lớp này sẽ hữu dụng nếu nó đã xử lý một kiểu dữ liệu riêng biệt, chẳng hạn như một lớp có tên

là Foo. Tuy nhiên, lớp thậm chí hữu dụng hơn nếu nó đã xử lý tất cả kiểu dữ liệu khác nhau được gán vào các pointer. Trong C, chúng ta xử lý các mảng pointer không thuần nhất bằng cách sử dụng các pointer rỗng (void pointer) vốn là những pointer dẫn sang các khối bộ nhớ mà không biết loại cấu trúc, kiểu dữ liệu hoặc lớp nào mà khối phải thể hiện. Thật không may, với C++, các pointer rỗng không hiệu quả. Ví dụ, xem xét mã trong Listing 30.1:

Listing 30.1: Làm việc với các pointer rỗng.

```
#include <stdio.h>
#include <string.h>
void delete_func( void *obj )
{
     if ( obj )
        delete obj;
}
class Foo
{
     char *s;
public:
     Foo( const char *strTemp )
     {
        printf("Constructor for foo\n");
        s = new char[strlen(strTemp)+1];
        strcpy(s, strTemp);
     }
     virtual ~Foo()
     {
        printf("Destructor for foo\n");
        delete s;
     }
};
int main()
{
     Foo *f = new Foo("This is a test"); → 1
     func(f);
}
```

Listing ở trên minh hoạ cách một pointer có thể được xem là "generic" như thế nào - nghĩa là không có kiểu. Trong chương trình chính (được minh hoạ tại → 1), chúng ta tạo một đối tượng Foo mới sử dụng constructor chuẩn cho lớp. Sau đó, pointer này được chuyển đến hàm func vốn xoá pointer mà không biết nó là kiểu gì. Nếu destructor cho lớp đã được gọi, chúng ta thấy hai dòng kết quả, một dòng cho việc tạo lớp và một dòng cho việc huỷ lớp.

Nếu bạn biên dịch chương trình này và chạy nó, bạn thấy kết quả sau đây:

```
Constructor for foo
```

Không có lệnh gọi destructor tương ứng cho đối tượng. Tại sao? Bởi vì vào thời gian chạy, chương trình không "biết" obj là loại đối tượng nào trong hàm delete_func, và do đó không thể gọi destructor cho đối tượng. Để hàm "biết" nó nhận đối tượng gì, chúng ta phải chuyển nó theo kiểu. Nếu chúng ta viết một trình quản lý các pointer, chắc chắn sẽ hữu ích khi biết các kiểu dữ liệu là gì để chúng ta có thể huỷ chúng một cách phù hợp. Để tránh vấn đề các pointer rỗng, chúng ta đơn giản dẫn xuất tất cả đối tượng từ một kiểu cơ sở chung, chẳng hạn như một kiểu Object và sau đó gọi destructor đó cho tất cả đối tượng. Vấn đề với điều này là nó không chỉ tạo ra hao phí trong việc tạo các đối tượng và đòi hỏi thêm không gian cho lớp cơ sở, nó gây ra các vấn đề với sự đa thừa kế (để biết thêm chi tiết về sự đa thừa kế, xem kỹ thuật 22). Thật sự không có lý do để gây ra thêm những điều phức tạp khi có thể có những phương pháp đơn giản hơn và trong trường hợp này phương pháp đơn giản hơn là sử dụng các template C++. Các template sẽ tiết kiệm cho bạn thời gian và công sức bằng việc giảm lượng mã được viết và khái quát hoá các giải pháp được sử dụng qua nhiều project.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi cua file nguồn.

   Trong ví dụ này, file được đặt tên là ch30.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 30.2 vào file.

   Tốt hơn, hãy sao chép mã từ file nguồn trên Web site đi kèm của sách này.

Listing 30.2: Phương pháp template

```
#include <stdio.h>
#include <string.h>
#include <vector>
template <class A>                                    → 5
class Manager
{
    std::vector< A *> _objects;                       → 7
public:
    Manager()
    {
    }
    ~Manager()
    {
       Clean();
    }
    void AddInstance( A *pObj )
    {
       _objects.insert( _objects.end(), pObj
    );
    }
    void Clean()
    {
       std::vector< A *>::iterator iter;
       for ( iter = _objects.begin(); iter !=
    _objects.end(); ++iter )
       {
          delete (*iter);
       }
       _objects.clear();
    }
    A *NewInstance()
    {
       A *pObject = new A;
```

```
        AddInstance(pObject);
        return pObject;
    }
    void DeleteInstance(A *obj)
    {
        std::vector< A *>::iterator iter;
        for ( iter = _objects.begin(); iter !=
    _objects.end(); ++iter )
            if ( (*iter) == obj )
            _objects.erase(iter);
        delete obj;
    }
};
class Foo
{
    char *s;
public:
    Foo (void)
    {
        printf("Constructor for foo\n");
        const char *strTemp = "Hello world";
        s = new char[strlen(strTemp)+1];
        strcpy(s, strTemp);
    }
    Foo( const char *strTemp )
    {
        printf("Constructor for foo\n");
        s = new char[strlen(strTemp)+1];
        strcpy(s, strTemp);
    }
    Foo( const Foo& aCopy )
    {
        s = new char[strlen(aCopy.s)+1];
        strcpy( s, aCopy.s );
    }
```

```
    virtual ~Foo()
    {
        printf("Destructor for foo\n");
        delete s;
    }
    const char *String()
    {
        return s;
    }
    void setString( const char *str )
    {
        if ( s )
            delete [] s;
        s = new char[strlen(str)+1];
        strcpy( s, str );
    }
};
int main(void)
{
    Manager<Foo> manager;                    → 6
    Foo *f = manager.NewInstance();          → 2
    Foo *f1 = manager.NewInstance();
    Foo *f2 = manager.NewInstance();
    Foo *f3 = manager.NewInstance();
    manager.DeleteInstance( f );             → 4
    manager.Clean();
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

   Chú ý khi chương trình được chạy, nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell.

```
$ ./a.exe
Constructor for foo                    → 3
Constructor for foo
Constructor for foo
Constructor for foo
Destructor for foo
Destructor for foo
Destructor for foo
Destructor for foo
```

Kết quả cho thấy constructor được gọi từ NewInstance (được minh hoạ tại → **2**) và sau đó được biểu thị trong kết quả (tại → **3**), nhưng quan trọng hơn destructor được gọi ra đúng cách khi chúng ta gọi ra các phương thức DeleteInstance của trình quản lý (manager) (được minh hoạ tại → **4**).

Như bạn có thể thấy trình quản lý hiểu kiểu lớp Foo mặc dù thật sự chúng ta đã không sử dụng trên Foo ở bất cứ nơi nào trong định nghĩa manager. Chúng ta biết điều này bởi vì manager tạo và xoá các đối tượng một cách phù hợp. Nó làm điều này như thế này? về cơ bản từ khoá template (được minh hoạ tại → **5** trong code listing) làm tất cả công việc. Khi trình biên dịch gặp phải từ khoá template, nó xem toàn bộ khối (trong trường hợp này là toàn bộ định nghĩa lớp) như thể nó là một "macro" lớn. Như bạn có thể nhớ lại từ bộ tiền xử lý (C), các macro thay thế một chuỗi nào đó cho một từ khoá riêng biệt trong khối. Mọi nơi mục nhập xuất hiện trong template (A, trong ví dụ này) xuất hiện trong khối nó được thay thế bằng bất cứ những gì lớp khuôn mẫu được thể hiện (tại → **6**). Trong driver chính, bạn sẽ thấy dòng:

```
Manager<Foo> manager;
```

Dòng này được mở rộng bởi trình biên dịch để thay thế A bằng Foo mọi nơi nó xuất hiện trong định nghĩa template. Tuy nhiên, không giống như các macro, việc kiểm tra được thực hiện vào lúc sự thể hiện cụ thể được tạo để chắc chắn rằng mã được tạo là C++ hợp lệ. Ví dụ, nếu chúng ta bỏ qua constructor copy từ lớp Foo, nó sẽ tạo ra một lỗi trong việc sử dụng lớp Foo trong một lớp có tên STL bởi vì tất cả thao tác di chuyển trong vector được thực hiện bằng cách sao chép các đối tượng từ một nơi này sang nơi khác. Có lẽ bạn sẽ thấy một lỗi tại dòng được đánh dấu là → **7**. Lỗi có lẽ sẽ mới một điều gì đó về việc không tìm thấy một constructor copy cho lớp khi vector lớp

khuôn mẫu đã được mở rộng bởi trình biên dịch. Vì lý do này, trước tiên trình biên dịch thể hiện cụ thể toàn bộ lớp, sử dụng lớp được cung cấp, và sau đó biên dịch kết quả.

Kỹ thuật 31: Mở rộng một lớp khuôn mẫu

Tiết kiệm thời gian bằng cách:

- Sử dụng các lớp khuôn mẫu trong mã
- Test các lớp khuôn mẫu
- Sử dụng các đối số không phải khuôn mẫu lớp

Sau khi bạn đã tạo một lớp cơ sở vốn không được sử dụng làm khuôn mẫu (template), bạn có thể mở rộng lớp khuôn mẫu đó bằng cách tận dụng nó trong ứng dụng. Mở rộng một lớp khuôn mẫu cho phép tận dụng chức năng mà bạn đã định nghĩa trong template bằng nhiều cách. Thật sự có bốn cách để tập dụng một lớp khuôn mẫu trong mã riêng của bạn. Tất cả cách này sẽ tiết kiệm cho bạn thời gian bằng việc cho phép bạn tái sử dụng mã hiện có mà không cần phải viết lại nó và đạt được sự tinh thông của bộ ghi lớp khuôn mẫu gốc cho mã của bạn.

- Bạn có thể sử dụng lớp khuôn mẫu thật sự làm một đối tượng khuôn mẫu trong mã. Để làm điều này, chỉ việc sử dụng một lớp với một đối số khuôn mẫu mà bạn tự chọn. Đây là phương pháp khi làm việc với các lớp đối tượng chứa từ Standard Template Library chẳng hạn.
- Bạn có thể sử dụng lớp mà bạn đã nhận dạng là một khuôn mẫu làm một biến thành viên trong đối tượng riêng của bạn. Điều này có nghĩa là nhúng một instance của lớp khuôn mẫu với một đối số khuôn mẫu mà bạn tự chọn trong đối tượng.
- Để sử dụng lớp khuôn mẫu làm một lớp cơ sở cho đối tượng riêng của bạn, xác định trước đối số khuôn mẫu và sử dụng nó để nhận dạng lớp cơ sở.
- Bạn có thể sử dụng lớp khuôn mẫu làm một lớp cơ sở cho lớp thừa kế riêng của bạn (một lớp có khôn mẫu hoặc một lớp không có khuôn mẫu), cho phép người dùng cuối xác định một hoặc nhiều đối số khuôn mẫu cho lớp.

Kỹ thuật này xem xét tất cả tuỳ chọn này và khai thác tính linh hoạt và sức mạnh của mỗi tuỳ chọn.

Thực thi các lớp khuôn mẫu trong mã

Sẽ chẳng có ích lợi gì nếu chỉ thảo luận các cách khác nhau mà bạn có thể thực thi các lớp khuôn mẫu trong mã mà không có những ví dụ cụ thể. Hãy xem một vài cách khác nhau mà chúng ta có thể tận dụng một lớp cơ sở có khuôn mẫu trong các ứng dụng riêng của chúng ta. Sau đây là cách thực hiện:

1. Trong code editor mà bạn tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch31.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 31.1 vào file.

Listing 31.1 Sử dụng các lớp khuôn mẫu trong mã.

```
#include <stdio.h>
#include <string>
// The base template name
template < class A >
class Base
{
    std::string _name;
    A           *_pointer;
public:
    Base(void)
    {
       _name = "Nothing";
       _pointer = NULL;
    }
    Base(const char *strName, A *aPointer )
    {
       _name = strName;
       if ( aPointer )
          _pointer = new A(aPointer);
       else
             _pointer = NULL;
    }
    Base( const Base& aCopy )
```

```
{
    printf("Copy constructor called\n");
    _name = aCopy._name;
    _pointer = new A(aCopy._pointer);
}
virtual ~Base()
{
    delete _pointer;
}
A *Pointer()
{
    return _pointer;
}
std::string Name()
{
    return _name;
}
void setPointer( A *aPointer )
{
    if ( _pointer )
        delete _pointer;
    _pointer = new A(aPointer);
}
void setName( const char *strName )
{
    _name = strName;
}
void Print()
{
    printf("Base:\n");
    printf("Name = %s\n", _name.c_str());
    printf("Pointer = \n");
    if ( _pointer )
        _pointer->Print();
    else
```

```
                printf("Pointer is NULL\n");
        }
};
class Foo
{
private:
        int i;
public:
        Foo(void)
        {
            i = 0;
        }
        Foo ( int iNum )
        {
            i = iNum;
        }
        Foo( const Foo& aCopy )
        {
            i = aCopy.i;
        }
        Foo ( const Foo* aCopy )
        {
            i = aCopy->i;
        }
        virtual ~Foo()
        {
        }
        int getNumber( void )
        {
            return i;
        }
        void setNumber( int num )
        {
            i = num;
        }
```

```
    void Print(void)
    {
        printf("Foo: i = %d\n", i );
    }
};
// Case 1: Using base template as a member variable
class TemplateAsMember
{
    Base<Foo> _fooEntry;
public:
    TemplateAsMember(void)
    : _fooEntry("TemplateAsMember", NULL)
    {
    }
    TemplateAsMember( int intNum )
        : _fooEntry("TemplateAsMember", new Foo(intNum))
    {
    }
    void setNum(int iNum)
    {
        _fooEntry.Pointer()->setNumber ( iNum );
    }
    int getNum(void)
    {
        return _fooEntry.Pointer()->getNumber();
    }
    int multiply(int iMult)
    {
        _fooEntry.Pointer()->setNumber ( _fooEntry.Pointer()->getNumber()
        iMult );
    }
    void Print()
    {
        printf("TemplateAsMember\n");
        _fooEntry.Print();
```

```
    }
};
// Case 2: Using the base template as a base
    class
class TemplateAsBase : public Base<Foo>
{
public:
    TemplateAsBase(void)
        : Base<Foo>( "TemplateAsBase", NULL )
    {
    }
    TemplateAsBase(const char *name, Foo
    *pFoo)
        : Base<Foo>( name, pFoo )
    {
    }
    virtual ~TemplateAsBase(void)
    {
    }
    void Print()
    {
        printf("TemplateAsBase:\n");
        Base<Foo>::Print();
    }
};
// Case 3: Using the base template as a base
    class
// for another templated class
template < class A, class B >
class TemplateAsBaseTemplate : public Base<A>
{
private:
    B *_anotherPointer;
public:
    TemplateAsBaseTemplate( void )
```

```
        : Base<Foo>( "TemplateAsBaseTemplate",
            NULL )
    {
        _anotherPointer = NULL;
    }
    TemplateAsBaseTemplate( A* anA, B* aB )
        : Base<Foo>( "TemplateAsBaseTemplate",
        anA )
    {
        _anotherPointer = aB;
    }
    B* getBPointer()
    {
        return _anotherPointer;
    }
    void Print()
    {
        Base<A>::Print();
        if ( _anotherPointer )
            _anotherPointer->Print();
        else
            printf("Another pointer is NULL\n");
    }
};
class AnotherBase
{
private:
    int x;
public:
    AnotherBase()
    {
        x = 0;
    }
    AnotherBase( int i )
    {
```

```
        x = i;
    }
    virtual ~AnotherBase(void)
    {
    }
    void Print()
    {
        printf("AnotherBase: x = %d\n", x );
    }
};
```

Trong Listing 31.1, mã cho thấy mỗi trường hợp có thể có được xử lý và được sử dụng như thế nào. Chúng ta đã thực thi hai lớp "bình thường" Foo và AnotherBase vốn được sử dụng làm các đối số khuôn mẫu để chỉ định các lớp khuôn mẫu.

Test các lớp khuôn mẫu

Để kiểm tra xem mã có thực sự hoạt động hay không, chúng ta cần thực thi một driver thử nghiệm. Các bước sau đây thực hiện điều này cho mã trong Listing 31.1:

1. Trong code editor mà bạn chọn, mở lại file nguồn cho mã mà bạn vừa tạo.

   Trong ví dụ này file được đặt tên là ch31.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 31.2 vào file.

Listing 31.2: Driver thử nghiệm cho ví dụ khuôn mẫu

```
int main()
{
    printf("Creating base\n");
    Base<Foo> fooBase;
    printf("Creating template as member\n");
    TemplateAsMember tempMem;
    printf("Creating template as base\n");
    TemplateAsBase tempBase;
    printf("Creating template as base template\
    n");
```

```
    TemplateAsBaseTemplate<Foo,AnotherBase>
    tempAsBT;
    fooBase.Print();
    tempMem.Print();
    tempBase.Print();
    tempAsBT.Print();
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn, trên hệ điều hành mà bạn chọn.

Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

```
Creating base
    Void constructor called
    Creating template as member
    Creating base with name
    [TemplateAsMember]
    Creating template as base
    Creating base with name [TemplateAsBase]
    Creating template as base template
    Creating base with name
    [TemplateAsBaseTemplate]
    Base:
    Name = Nothing
    Pointer =
    Pointer is NULL                          → 1
    TemplateAsMember
    Base:
    Name = TemplateAsMember
    Pointer =
    Pointer is NULL
    TemplateAsBase:
    Base:
    Name = TemplateAsBase
```

```
Pointer =
Pointer is NULL
Base:
Name = TemplateAsBaseTemplate
Pointer =
Pointer is NULL
Another pointer is NULL
```

Kết quả từ chương trình này cho thấy từng phần thể hiện cụ thể template khác nhau làm việc. Như bạn có thể thấy trong mỗi trường hợp (ví dụ xem dòng được đánh dấu là → 1), constructor đã được gọi và các biến thành viên khác nhau đã gán những giá trị mặc định thích hợp. Xem các ví dụ, chúng ta sẽ thấy rõ rằng từng phương thức khác nhau đạt được cùng một kết luận.

Sử dụng các đối số không phải khuôn mẫu lớp

Xem bốn lựa chọn trong việc mở rộng các lớp cơ sở, có lẽ bạn sẽ chú ý một vài điều gọi ra những cách tiếp cận cụ thể đối với tiến trình.

Để tận dụng các lớp trong một lớp cơ sở được sử dụng làm một khuôn mẫu, bạn phải xác định phiên bản nào của lớp mà khuôn mẫu sẽ sử dụng. Lý do là bạn có thể có một lớp thừa kế từ nhiều của cùng một khuôn mẫu với các kiểu khác nhau được kết hợp với nó. Đây là điều tốt bởi vì nó cho phép bạn phân chia chức năng cụ thể thành các lớp riêng biệt.

Một điều khác cần chú ý là bạn có thể tạo một lớp khuôn mẫu vốn không đòi hỏi một lớp dưới dạng đối số của nó. Ví dụ, chúng ta có thể tạo một khuôn mẫu với một đối số dạng số, các bước sau đây hướng dẫn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới.

   Trong ví dụ này, file được đặt tên là ch31a.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 31.3 vào file.

   Listing 31.3: Một lớp khuôn mẫu với đối số không phải lớp

```
template <class A>
class LessThanTen
{
    A _element;
```

```
public:
    LessThanTen( A entry )
    {
       set ( entry );
    }
    LessThanTen( const LessThanTen& aCopy )
    {
       set( aCopy._element);
    }
    void set( A value )                                    → 3
    {
       if ( value > 0 && value < 10 ) → 2
          _element = value;
    }
    A get()
    {
       return _element;
    }
};
```

Mã này làm việc cho một đối số lớp, do đó miễn là lớp đó có thể được so sánh với một đối số. Nó cũng có thể được sử dụng cho một đối số không phải lớp chẳng hạn như một số nguyên, một số nguyên long, thậm chí một giá trị dấu động.

Với lớp này được minh hoạ trong Listing 31.3, không có lý do gì đối số template sẽ là một lớp. Thật ra, ngầm định rằng một phần tử số đã được sử dụng vì giá trị được chuyển vào phương thức set được so sánh với một giá trị số nguyên 10 (xem dòng được đánh dấu là → 2).

3. Thêm mã sau đây vào file nguồn để test lớp số nguyên. Mã này sẽ test lớp khuôn mẫu mà bạn vừa định nghĩa ở trên.

```
int main(void)
{
    LessThanTen<int> ten(3);                               → 5
    printf("The value is %d\n",
    ten.get() );
    ten.set( 23 );                                         → 4
    printf("The value is now %d\n",
```

```
    ten.get() );
    return 0;
}
```

4. Lưu file nguồn trong code editor và sau đó đóng code editor.
5. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn, trên hệ điều hành mà bạn chọn.

Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

Như bạn có thể thấy từ kết quả, chương trình tạo phù hợp một lớp mới vốn là một phiên bản template của lớp LessThanTen, sử dụng một số nguyên làm đối số template của nó. Lớp vừa có được chứa một phương thức được gọi là set lấy một đối số số nguyên (được minh hoạ tại → **3**) vốn phải nằm giữa 0 và 10. Vì giá trị (xem → **4**) không nằm trong dãy đó, nó không được gán và cho chúng ta thấy rằng câu lệnh print theo sau phép gán vẫn chứa giá trị 3.

Chú ý rằng mã làm việc với một đối số số nguyên (xem dòng được đánh dấu là → **5**) mặc dù template xác định một đối số lớp. Đối với C++, các số nguyên, số dấu động và những thứ tương tự có thể được xem là các đối số lớp đầu tiên (nghĩa là một kiểu cơ bản) cho mục đích của các template. Thực tế, (ít ra trong trường hợp này) bất kỳ đối số có thể được sử dụng miễn là mã chứa các toán tử so sánh lớn hơn (>) và nhỏ hơn (<) để bảo đảm rằng phương thức set làm việc tốt.

Khả năng sử dụng các lớp (class) hoặc kiểu cơ bản làm các đối số template làm cho cấu trúc template trở nên cực kỳ mạnh mẽ trong C++. Bởi vì bạn có thể viết một lớp quản lý số, các chuỗi ký tự hoặc giá trị lớp và làm cho nó làm việc không giới hạn, bạn tiết kiệm lượng thời gian rất lớn và giảm bớt phần công việc lặp lại.

Kỹ thuật 32: Tạo các template từ các hàm và phương thức

Tiết kiệm thời gian bằng cách:

- Tạo các khuôn mẫu hàm
- Tạo các khuôn mẫu phương thức
- Hiểu kết quả

Mặc dù tạo toàn bộ các lớp vốn là những khuôn mẫu (template) thì hữu dụng, nhưng đôi khi bạn chỉ muốn một hàm hoặc phương thức để

chấp nhận một đối số khuôn mẫu. Ví dụ, nếu bạn đã tạo một hàm so sánh sử dụng một khuôn mẫu cho tất cả kiểu, bạn có thể tạo (ví dụ) một hàm minimum so sánh hai kiểu dữ liệu bất kỳ kể cả các lớp miễn là bạn có thể biết kiểu nào có độ lớn nhỏ hơn kiểu kia. Kỹ thuật này hướng dẫn bạn tiết kiệm thời gian bằng việc khuôn mẫu hoá chỉ một hàm (và sau đó một phương thức) của một lớp.

Thực thi các khuôn mẫu hàm

Các khuôn mẫu hàm (function template) hoặc khuôn mẫu phương thức (method template) đơn giản là một hàm độc lập (bên trong một lớp trong trường hợp của một phương thức) có thể chấp nhận một hoặc nhiều đối số khuôn mẫu. Hãy xem bạn có thể thực thi các khuôn mẫu hàm trong mã riêng của bạn như thế nào bằng cách tạo một hàm generic tính giá trị nhỏ nhất của hai giá trị.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch32.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 32.1 vào file.

Listing 32.1: Hàm khuôn mẫu Min

```
#include <stdio.h>
#include <string>
template <class A>
A my_min( const A& val1, const A& val2)                    → 1
{
    if ( val1 == val2 )
        return val1;
    if ( val1 < val2 )
        return val1;
    return val2;
}
bool operator==(const std::string& s1, const std::string& s2)    → 5
{
    int len = s1.length();
    if ( len != s2.length() )
        return false;
    for ( int i=0; i<len; ++i )
```

```
        if ( s1[i] != s2[i] )
            return false;
    return true;
}
bool operator <(const std::string& s1, const std::string& s2          → 6
{
    int len = s1.length();
    if ( len > s2.length() )
        len = s2.length();
    for ( int i=0; i<len; ++i )
        if ( s1[i] > s2[i] )
            return false;
    return true;
}
int main(int argc, char **argv)
{
    // First, try it for integers
    int x1 = 100;
    int x2 = 30;
    int xmin = my_min(x1, x2);                                         → 2
    // Now, for floating-point numbers
    float f1 = 12.40;
    float f2 = 4.90;
    float fmin = my_min(f1, f2);
int xmin2 = my_min(x2, x1);                                            → 3
    printf("Xmin = %d\n", xmin);
    printf("Xmin2 = %d\n", xmin2 );
    printf("Fmin = %f\n", fmin );
    // Now that we have implemented the operators,
    // try it for strings.
    std::string s1 = "Hello world";
    std::string s2 = "Goodbye cruel world";
    if ( s1 == s2 )
        printf("Strings are equal\n");
```

```
    else
        if ( s1 < s2 )
            printf("string %s is less\n", s1.c_str() );
        else
            printf("String %s is less\n", s2.c_str() );
    std::string smin = my_min( s1, s2 );                    → 4
    printf("Min for strings returned: %s\n", smin.c_str() );
}
```

3. Lưu file nguồn trong code editor và đóng code editor.

   Chú ý chúng ta đã tạo một hàm khuôn mẫu có tên là my_min được minh hoạ tại → **1** (nó không được gọi là min, bởi vì tên đó xung đột với hàm Standard Template Library (STL) có cùng một tên) vốn có thể sử dụng với bất kỳ kiểu dữ liệu hỗ trợ các toán tử dấu bằng (=) và dấu nhỏ hơn (<).

   Trong trường hợp này mã nguồn cũng thực thi các phép tính nhỏ hơn và bằng cho lớp chuỗi chuẩn trong STL. Sau khi chúng ta thực thi những toán tử này, chúng ta có thể thể hiện cụ thể khuôn mẫu cho lớp chuỗi.

4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chạy.

Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

```
$ ./a.exe
Xmin = 30
Xmin2 = 30
Fmin = 4.900000
String Goodbye cruel world is less
Min for strings returned: Goodbye cruel
    world
```

Hãy xem cẩn thận kết quả này và thấy điều gì đang xảy ra. Chúng ta gọi hàm minimum my_min trong ba vị trí được thể hiện tại các dòng được đánh dấu → **2**, → **3**, và → **4**. Dòng thứ nhất tính min cho hai số nguyên, dòng thứ hai cho hai số dấu động và dòng thứ ba cho hai chuỗi.

Mặc dù các số nguyên và số dấu động có những toán tử so sánh riêng của chúng (nhỏ hơn, lớn hơn và ....) được đưa vào ngôn ngữ, nhưng các chuỗi (string) thì không. Do đó chúng ta thực thi các hàm

so sánh vốn sẽ được gọi bởi my_min tại các dòng được đánh dấu → 5 và → 6. Sau khi tất cả điều này được thực hiện, trình biên dịch tạo các phiên bản của những hàm này và bạn thấy kết quả các phép tính tối thiểu (minimum) được trình bày ở trên.

Tạo các khuôn mẫu phương thức

Tương tự, bạn cũng có thể tạo các phương thức của các lớp vốn chính là các khuôn mẫu, mặc dù toàn bộ lớp có thể không. Bạn có thể muốn làm điều này như là một cách để tránh viết nhiều mã giống nhau lặp đi lặp lại, chẳng hạn như tạo một phương thức gán cho các kiểu dữ liệu khác nhau.

Các bước sau đây hướng dẫn bạn cách tạo một khuôn mẫu phương thức:

1. Trong code editor mà bạn chọn, mở lại file nguồn cho mã mà bạn vừa tạo.

   Trong ví dụ này, file được đặt tên là ch32.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 32.2 vào file.

Listing 32.2: Một lớp có một phương thức khuôn mẫu.

```
class Foo
{
private:
    std::string   _name;
    int           _value;
public:
    Foo(void)
    {
        _name = "Nothing";
        _value = 0;
    }
    Foo(const char *strName, int iValue)
    {
        _name = strName;
        _value = iValue;
    }
    Foo( const Foo& aCopy )
```

```
    {
        _name = aCopy._name;
        _value = aCopy._value;
    }
    Foo operator=( const Foo& aCopy )
    {
        _name = aCopy._name;
        _value = aCopy._value;
        return *this;
    }
    // Templatized method to add values
    template < class A >
    void Add( const A& aValue )
    {
        _value += aValue;
    }
    // Templatized method to multiply values
    template < class A >
    void Multiply( const A& aValue )                    → 7
    {
        _value = _value * aValue;
    }
    // Method to dump the values
    void Print()
    {
        printf("Name: [%s]\n", _name.c_str()
    );
        printf("Value: %d\n", _value );
    }
};
int operator*( int iValue, std::string s )
{
    // Convert string to an integer
    int iMult = atoi( s.c_str() );
    // Do the multiplication
```

```
        int iResult = iValue * iMult;
        // Return it
    return iResult;
    }
```

Listing ở trên trình bày một lớp Foo vốn chứa một phương thức khuôn mẫu được gọi là Multiply (được minh hoạ tại → 7). Phương thức này sẽ cho phép bạn nhân nhiều kiểu dữ liệu khác nhau và gán kết quả vào một biến thành viên. Cũng chú ý hàm operator* được định nghĩa ngay cả để nhân giá trị với một chuỗi.

3. Thay đổi hàm chính của file nguồn như được minh hoạ trong Listing 32.3.

Listing 32.3: Driver thử nghiệm cho phương thức khuôn mẫu

```
int main(int argc, char **argv)
{
    // First, try it for integers
    int x1 = 100;
    int x2 = 30;
    int xmin = my_min(x1, x2);
    // Now, for floating point numbers
    float f1 = 12.40;
    float f2 = 4.90;
    float fmin = my_min(f1, f2);
    int xmin2 = my_min(x2, x1);
    printf("Xmin = %d\n", xmin);
    printf("Xmin2 = %d\n", xmin2 );
    printf("Fmin = %f\n", fmin );
    // Now that we have implemented the oper
    // ators, try it for strings.
    std::string s1 = "Hello world";
    std::string s2 = "Goodbye cruel world";
    if ( s1 == s2 )
       printf("Strings are equal\n");
    else
       if ( s1 < s2 )
          printf("string %s is less\n",
```

```
        s1.c_str() );
            else
                printf("String %s is less\n",
        s2.c_str() );
        std::string smin = my_min( s1, s2 );
        printf("Min for strings returned: %s\n",
        smin.c_str() );
        Foo f("MyFoo", 10);
        f.Add( 12.0 );
        f.Print();
        f.Multiply( -1 );
        f.Print();
        f.Multiply(std::string("12"));
        f.Print();
}
```

Chú ý rằng như với các hàm khuôn mẫu, nhà lập trình không tạo một instance của hàm thành viên khuôn mẫu bằng cách nào cả. Trình biên dịch sẽ tạo các instance khi cần thiết, bằng cách kết hợp những đối số khuôn mẫu có thể có với những template có sẵn. Hiển nhiên điều này sẽ chỉ có tác dụng nếu định nghĩa lớp khuôn mẫu có sẵn, do đó một lần nữa các phương thức khuôn mẫu sẽ là các phương thức được định nghĩa nội dòng (in-line) bởi vì không có cách "tự nhiên" để nhân một chuỗi với một số nguyên, chúng ta định nghĩa một toán tử toàn cục vốn chấp nhận một giá trị số nguyên và một chuỗi và trả về chuỗi được chuyển đổi được nhân với số nguyên. Sau khi toán tử này được định nghĩa, chúng ta có thể chuyển vào một chuỗi đến phương thức khuôn mẫu. (Nếu toán tử đã không được định nghĩa, bạn nhận được một lỗi biên dịch khi khuôn mẫu được mở rộng và phép nhân số nguyên với đối số đầu vào được thử).

4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

   Khi chương trình được chạy, nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

```
$ ./a.exe
Xmin = 30
Xmin2 = 30
Fmin = 4.900000
String Goodbye cruel world is less
```

```
Min for strings returned: Goodbye cruel
    world
Name: [MyFoo]                                    → 8
Value: 22
Name: [MyFoo]
Value: -22
Name: [MyFoo]
Value: -264
```

Phần đầu của kết quả là từ phần đầu tiên của kỹ thuật này và đã không thay đổi. Phần thứ hai với dòng được đánh dấu → 8 cho thấy kết quả của phương thức khuôn mẫu. Như bạn có thể thấy đầu tiên chúng ta gán cho biến thành viên value giá trị 10. Sau đó chúng ta cộng 12 với giá trị dẫn đến kết quả là 22. Bây giờ chúng ta bắt đầu sử dụng phương thức Multiple. Đầu tiên chúng ta nhân với một giá trị số nguyên -1 dẫn đến kết quả là -22. Sau đó, chúng ta nhân với một giá trị chuỗi 12 vốn được chuyển đổi thành một số nguyên bằng cách sử dụng hàm operator* mà chúng ta đã định nghĩa, điều này dẫn đến giá trị -264 - giá trị được in ra.

Chú ý chuỗi được lượng giá sang giá trị số nguyên của nó 12 và kết quả được nhân với giá trị hiện hành -22. Điều này dẫn đến tổng cộng -264 vốn là giá trị hiển thị trên console.

Kỹ thuật 33: Làm việc với các mảng (array)

Tiết kiệm thời gian bằng cách:

- Tìm hiểu lớp vector
- Thực thi các lớp mảng
- Làm việc với các thuật toán khác nhau với lớp vector

Standard Template Library (STL) cung cấp sự truy cập đến một số dạng lớp lưu trữ khác nhau. Một trong những dạng lớp này là lớp vector cho phép nhà lập trình lưu trữ các mảng đối tượng tuỳ ý. Thật không may, lớp vector cũng đòi hỏi bạn "kéo vào" (liên kết) toàn bộ STL để sử dụng nó. Điều này có thể gây ra hao phí đáng kể trong ứng dụng và có thể tiêu thụ lượng lớn bộ nhớ. Bởi vì lớp vector là một lớp khuôn mẫu, nó sao chép các cụm mã lớn mỗi lần nó được sử dụng. Mặc dù điều này thường không quan trọng cho các ứng dụng có qui mô lớn và xứng đáng với chi phí bộ nhớ và kích cỡ, nó có thể rất tốn kém khi các ứng dụng của bạn nhỏ hơn hoặc phải vận hành trong

những vùng bộ nhớ giới hạn hơn, như với các ứng dụng nhúng. Vì lý do này, bạn nên quen thuộc không chỉ với lớp vector mà còn với việc thực thi các lớp mảng (array) đơn giản vốn có thể sử dụng bộ nhớ giới hạn. Kỹ thuật này hướng dẫn bạn cách sử dụng các lớp vector và những kỹ thuật để làm việc với những thuật toán khác nhau với lớp vector. Kỹ thuật kế tiếp hướng dẫn bạn cách sử dụng các lớp mảng với hao phí giới hạn.

Sử dụng lớp vector

Lớp vector mạnh mẽ đáng kinh ngạc và sau khi bạn nắm vững nó, rất dễ sử dụng trong mã riêng của bạn. Hãy xem một ví dụ về việc làm việc với các vector. Chúng ta sẽ xem cách thêm dữ liệu vào một vector, bước qua (lặp lại) các giá trị trong mảng và in ra dữ liệu trong mảng. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này file được đặt tên là ch33.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 33.1 vào file.

Listing 33.1: Làm việc với các vector

```
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
using namespace std;
template < class A >
void print_array( vector<A> array )
{
    vector<A>::iterator iter;
    for ( iter = array.begin(); iter != array.end(); ++iter )
        cout << (*iter) << "\n";
}
void reverse_array( const vector<string>& arrayIn, vector<string>& arrayOut )
{
    vector<string>::const_iterator iter;
    for ( iter = arrayIn.begin(); iter != arrayIn.end(); ++iter )
    {
```

```
        arrayOut.insert( arrayOut.begin(), (*iter) );
    }
}
int main(int argc, char **argv)
{
    vector<string> stringArray;
    // First, add all of the arguments to the
    // array that were passed into the application.
    for ( int i=0; i<argc; ++i )                          → 1
        stringArray.insert( stringArray.end(), argv[i] );
    // Print them out using an iterator
    vector<string>::iterator iter;
    for (iter = stringArray.begin(); iter != stringArray.end();
            ++iter )
    {
        // This isn't necessary, but illustrates how to get
        // the actual item stored in an array element. Note that
        // the copy constructor will be invoked on this line.
        string s = (*iter);                               → 2
        cout << "Element: " << s << "\n";
    }
    // Now, we want to remove any element in the array which is the number
    // 3
    for ( iter = stringArray.begin(); iter != stringArray.end();
        iter ++ )
    {
        if ( (*iter) == "3" )
        {
            cout << "Erasing element " << (*iter) << "\n";
            stringArray.erase( iter );                    → 3
        }
    }
    // Display the results for the user
    printf("Array after removal\n");
```

```
    print_array( stringArray );
    // Next, reverse the array
    vector<string> outArray;
    reverse_array( stringArray, outArray );                    → 4
    printf("Array after reversal\n");
    print_array( outArray );
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng code editor.

   Mã nguồn này tận dụng sức mạnh và chức năng của Standard Template Library để làm những thao tác xử lý mảng (array) bao gồm thêm dữ liệu vào một mảng, đảo ngược các phần tử trong một mảng và lặp lại trên các phần tử trong một mảng

4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn, trên hệ điều hành mà bạn chọn.

Khi chương trình chạy, nếu bạn đã làm được tốt mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

```
$ ./a.exe 1 2 3 4
Element: ./a
Element: 1
Element: 2
Element: 3
Element: 4
Erasing element 3
Array after removal
./a
124
Array after reversal
421
./a
```

Như bạn có thể thấy, chương trình đọc các đối số từ dòng lệnh, đặt chúng vào mảng (được minh hoạ tại dòng được đánh dấu → **1**), in ra mảng bằng cách lặp lại trên mỗi phần tử và in ra dữ liệu chứa tại phần tử mảng đó (được minh hoạ tại → **2**). Tiếp theo, chương trình loại bỏ tất cả giá trị là 3 ra khỏi mảng, minh hoạ cách bạn có thể xoá khỏi một mảng (được minh hoạ tại → **3**) và để lại các phần tử còn lại theo thứ tự như thế nào.

Sau cùng, những giá trị trong mảng được đảo ngược bằng cách sao chép chúng sang một nảng mới theo thứ tự đảo ngược (được minh hoạ tại → 4) và kết quả được in ra nhiều lần.

Kỹ thuật 34: Thực thi lớp array

Tiết kiệm thời gian bằng cách:

- Thực thi các lớp array với hao phí giới hạn
- Sử dụng một lớp vector vốn lưu trữ các chuỗi
- Hiểu kết quả

Sau khi bạn đã học cách sử dụng lớp vector STL (xem kỹ thuật 33), việc xem cách bạn có thể tự thực thi cùng một loại lớp với hao phí giới hạn hơn sẽ mang lại rất nhiều thông tin. Hãy xem việc thực thi một lớp vector đơn giản vốn không chỉ lưu trữ các chuỗi mà còn chèn, xoá và lặp lại.

Tạo lớp mảng chuỗi

Như chúng ta đã thấy trong kỹ thuật 33, hao phí trong việc sử dụng Standard Template Library (STL) hơi cao khi bạn muốn thêm chỉ một mảng vào chương trình. Nếu bạn sử dụng nhiều lớp mảng, bạn cũng có thể sử dụng STL, bởi vì theo cách đó bạn chỉ phải trả cái giá (về bộ nhớ và kích cỡ ứng dụng) một lần. Mỗi lớp mảng vượt ngoài yếu tố đầu tiên sử dụng không gian không đáng kể trong ứng dụng. Hãy tạo một lớp mảng chuỗi đơn giản minh hoạ việc tạo các lớp mảng cho ứng dụng vốn làm việc với các kiểu dữ liệu riêng biệt một cách dễ dàng như thế nào. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch34.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 34.1 vào file.

Listing 34.1: Tạo lớp mảng chuỗi riêng của bạn

```
#include <stdio.h>
#include <string>
using namespace std;
class MyStringArray
{
```

```
    string *_strings;
    int _numstrings;
    int _chunksize;
    int _numused;
    void expand()
    {
        // Allocate a new block
        string *newBlock = new string[ _numstrings + _chunksize ];
        _numstrings += _chunksize;
        for ( int i=0; i<_numused; ++i )
            newBlock[i] = _strings[i];
        // Delete the old array
        delete [] _strings;
        // Re-assign the pointer
        _strings = newBlock;
    }
public:
    MyStringArray(void)
    {
        _chunksize = 10;
        _strings = new string[ _chunksize ];
        for ( int i=0; i<_chunksize; ++i )
            _strings[i] = "";
        _numstrings = _chunksize;
        _numused = 0;
    }
    MyStringArray( int nSize )
    {
        _chunksize = 10;
        if ( nSize <= _chunksize )
        {
            _strings = new string[ _chunksize
];
            _numstrings = _chunksize;
    }
```

```
    else
    {
        _strings = new string[ nSize ];
        _numstrings = nSize;
        }
        _numused = 0;
    }
    virtual ~MyStringArray(void)
    {
       delete [] _strings;
    }
    // Insert at start
    void insert_string( const string& s )
    {
       // See if it will fit.
       if ( _numused == _numstrings )
          expand();
       // It will now fit, move everything up
       for ( int i=_numused; i>=0; —i )
          _strings[i] = _strings[i-1];
       // Put in the new one
       _strings[0] = s;
       _numused ++;
    }
    void append_string( const string& s )
    {
       // See if it will fit.
       if ( _numused == _numstrings )
          expand();
       // Put in the new one
       _strings[_numused] = s;
       _numused ++;
    }
    string remove_at( int idx )
    {
```

```
        if ( idx < 0 || idx >= _numused )
            return string("");
        // Save this one
        string ret_string = _strings[idx];
        // And copy all the others after it
    back
    for ( int i=idx; i<_numused; ++i )
        _strings[i] = _strings[i+1];
    _numused--;
    return ret_string;
    }
    string get_at(int idx)
    {
        if ( idx < 0 || idx >= _numused )
            return string("");
        return _strings[idx];
    }
    int size()
    {
        return _numused;
    }
};
int main(int argc, char **argv)
{
    MyStringArray s(5);
    for ( int i=0; i<argc; ++i )
    {
        printf("Appending %s\n", argv[i] );
        s.append_string( argv[i] ); → 1
    }
    printf("Initial String Array:\n");
    for ( int j=0; j<s.size(); ++j )
        printf("String %d = [%s]\n", j,
    s.get_at(j).c_str());
    if ( s.size() > 5 )
```

```
    {
        string str = s.remove_at(5);                    → 4
        printf("Removed string %s\n",
    str.c_str());
    }
    printf("Final String Array:\n");
    for ( int i=0; i<s.size(); ++i )
        printf("String %d = [%s]\n", i,
    s.get_at(i).c_str());
}
```

Mã nguồn này sử dụng chức năng mảng đơn giản để thêm, loại bỏ và lặp lại trên các chuỗi trong một cấu trúc mảng không giới hạn.

Bây giờ mã không hoàn toàn tinh tế như lớp vector STL, nhưng nó cũng làm việc tốt.

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn thấy kết quả sau đây trong cửa sổ shell:

```
$ ./a.exe 1 2 3 4 5 6 7
Appending ./a                                           → 2
    Appending 1
    Appending 2
    Appending 3
    Appending 4
    Appending 5
    Appending 6
    Appending 7
    Initial String Array:
    String 0 = [./a]                                    → 3
    String 1 = [1]
    String 2 = [2]
    String 3 = [3]
    String 4 = [4]
    String 5 = [5]
```

```
String 6 = [6]
String 7 = [7]
Removed string 5
Final String Array:
String 0 = [./a]
String 1 = [1]
String 2 = [2]
String 3 = [3]
String 4 = [4]
String 5 = [6]
String 6 = [7]
```

Kết quả từ chương trình này như mọng đợi: chúng ta thêm từng đối số đầu vào vào mảng, theo dõi nó phát triển. Điều này được minh hoạ bắt đầu được đánh dấu → **1** trong mã và → **2** trong kết quả. Kế tiếp chúng ta in ra mảng, mong đợi thấy tất cả giá trị hiển thị (xem → **3** trong kết quả. Như mong đợi, chúng ta thấy tất cả tám giá trị dữ liệu. Tiếp theo, mã loại bỏ phần tử dữ liệu thứ năm như được minh hoạ tại → **4** trong code listing. Sau đó, chúng ta in ra mảng một lần nữa để cho thấy giá trị đã được loại bỏ và những giá trị dữ liệu khác vẫn còn.

Thực thi lớp mảng riêng của bạn có thể tiết kiệm cho bạn thời gian đáng kể trong việc cố giảm kích cỡ file và cho phép bạn sử dụng mã ở những nơi nó có thể không phù hợp do những ràng buộc bộ nhớ.

Như bạn có thể thấy từ kết quả này, chúng ta có thể tạo lớp mảng riêng của mình để thực thi phần lớn chức năng của lớp vectro STL với một phần nhỏ bộ nhớ và tốc độ cần thiết.

Kỹ thuật 35: Làm việc với các thuật toán vector

Tiết kiệm thời gian bằng cách:

- Sử dụng các thuật toán cài sẵn của STL
- Sử dụng các thuật toán vector
- Hiểu kết quả

Sức mạnh thật sự của Standard Template Library không phải là khả năng nó lưu trữ và truy tìm dữ liệu ở dạng khuôn mẫu. Thay vào đó, sức mạnh đó chính là các thuật toán cài sẵn mà bạn có thể sử

dụng trên những tiện ích lưu trữ khác nhau trong STL. Mặc dù các lớp đối tượng chứa làm một công việc hoàn hảo là duy trì dữ liệu mà bạn đặt vào chúng, nhưng các thuật toán khuôn mẫu cho phép bạn làm việc với dữ liệu đó, phân loại nó, tìm kiếm nó và chuyển đổi nó thành các dạng khác.

STL bao gồm những hàm thuật toán để lặp lại trên các tập hợp, tìm các mục nhập riêng biệt, loại bỏ các mục nhập riêng biệt và phân loại các mục nhập trong các tập hợp.... Kỹ thuật này sẽ xem xét những cách để xử lý dữ liệu một cách hiệu quả trong một tập hợp, sử dụng những thuật toán có sẵn cho các vector. Trong kỹ thuật này chúng ta sẽ xem xét những cách để phân loại, tìm kiếm và loại bỏ các phần tử trong một đối tượng chứa STL. Những thuật toán này có sẵn cho bất kỳ lớp đối tượng chứa STL, sử dụng các tên giống nhau trong mỗi trường hợp. Mặc dù chúng ta sẽ tập trung vào lớp vector trong kỹ thuật này, nhưng những thuật toán đó sẽ làm việc với các stack, map, queue và tất cả đối tượng chứa STL còn lại. Sử dụng thuật toán này sẽ tiết kiệm cho bạn thời gian trong việc thực hiện những tác vụ thích hợp nhất được yêu cầu cho các ứng dụng ngày nay.

Làm việc với các thuật toán vector

Nếu bạn được trình bày một mảng hoặc vectro dữ liệu, các chức năng nhất định hầu như sẽ luôn được yêu cầu bởi người dùng. Khả năng phân loại dữ liệu theo giá trị của nó là một trong những chức năng đó. Một yêu cầu khác là hầu như sẽ là khả năng tìm một giá trị dữ liệu nào đó trong vector. Sau cùng, chèn và loại bỏ dữ liệu là điều thiết yếu cho bất kỳ mảng (array). Hãy xem những kỹ thuật để thực hiện tất cả tác vụ này, sử dụng lớp vector STL và các thuật toán STL. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch35.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 35.1 vào file.

Listing 35.1: Sử dụng các thuật toán STL với lớp vector.

```
#include <stdio.h>
#include <string>
#include <vector>
#include <algorithm>
bool contains_c( const std::string& s )
{
```

```
    for ( int i=0; i<s.length(); ++i )
        if ( s[i] == 'c' )
            return true;
    return false;
}
int main(int argc, char **argv)
{
    // First, create a vector out of all
    // of the input strings
    std::vector< std::string > elements;
    for ( int i=0; i<argc; ++i )                                    → 1
        elements.insert( elements.end(), argv[i] );
    // Print out the elements.
    printf("Original list:\n");
    std::vector< std::string >::iterator iter;
    for ( iter = elements.begin(); iter != elements.end(); ++iter )
        printf("Element %s\n", (*iter).c_str() );
    // Now, sort the elements.
    std::sort( elements.begin(), elements.end() );                  → 2
    // Print them out again.
    printf("Sorted list:\n");
    for ( iter = elements.begin(); iter != elements.end(); ++iter )
        printf("Element %s\n", (*iter).c_str() );
    // Find a specific element, if it exists.
    std::vector< std::string >::iterator ptr_iter;
    ptr_iter = std::find( elements.begin(), elements.end(), "Hello"); → 3
    if ( ptr_iter != elements.end() )
        printf("Found the element %s\n", (*ptr_iter).c_str() );
    else
        printf("Didn't find the requested element\n");
    // If we found it, remove it from the list.
    if ( ptr_iter != elements.end() )
        elements.erase( ptr_iter );
    // And relist them.
    printf("Altered list:\n");
```

```
    for ( iter = elements.begin(); iter != elements.end(); ++iter )
        printf("Element %s\n", (*iter).c_str() );
    // See how many times "Why" is in the list.
    int cnt = std::count( elements.begin(), elements.end(), "Why" );
    printf("Found %d entries with the name \'Why\'\n", cnt );
    // Remove entries only if they contain the letter 'c'
    std::remove_if( elements.begin(), elements.end(), contains_c );
    printf("Final list:\n");
    for ( iter = elements.begin(); iter != elements.end(); ++iter )
        printf("Element %s\n", (*iter).c_str() );
    return 0;
}
```

3. Lưu file nguồn trong code editor và đóng code editor.

   Mã nguồn này tận dụng sức mạnh và chức năng của Standard Template Library để thực hiện những thao tác xử lý mảng. Thêm các phần tử, loại bỏ chúng, đếm số khớp với một chuỗi nào đó, tìm kiếm và phân loại mảng đều được minh hoạ.

4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

Khi chương trình chạy, nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 35.2 trong cửa sổ shell.

Listing 35.2: Kết quả từ chương trình thuật toán vector

```
$ ./a.exe Hello Goodbye Why What Iditarod
    Alpha Why Me Accord
Original list:
Element ./a
Element Hello
Element Goodbye
Element Why
Element What
Element Iditarod
Element Alpha
Element Why
Element Me
Element Accord
```

```
Sorted list:
Element ./a
Element Accord
Element Alpha
Element Goodbye
Element Hello
Element Iditarod
Element Me
Element What
Element Why
Element Why
Found the element Hello
Altered list:
Element ./a
Element Accord
Element Alpha
Element Goodbye
Element Iditarod
Element Me
Element What
Element Why
Element Why
Found 2 entries with the name 'Why'
Final list:
Element ./a
Element Alpha
Element Goodbye
Element Iditarod
Element Me
Element What
Element Why
Element Why
Element Why
```

Kết quả phân chia thành một vài bước khớp với các bước chương trình. Đầu tiên, chúng ta nhập dữ liệu từ dòng lệnh và lưu trữ nó vào

danh sách ban đầu như được đánh dấu trong kết quả. Điều này xảy ra tại dòng → **1** trong code listing. Tiếp theo, chúng ta phân loại danh sách, sử dụng thật toán STL sort (được minh hoạ tại → **2** trong code listing). Dòng mã này phân loại toàn bộ mảng, so sánh mỗi phần tử với phần tử kế tiếp và hoán đổi chúng vào đúng vị trí. Sau đó dữ liệu được xuất với danh sách phân loại header. Tác vụ kế tiếp là định vị một chuỗi riêng biệt, trong trường hợp này là "Hello" trong mảng (xem → **3**). Nếu nó được tìm thấy, phần tử đó bị loại bỏ và mảng được in ra nhiều lần, sử dụng danh sách thay đổi tiêu đề. Tác vụ kế tiếp là đếm số lần một chuỗi nào đó ("Why") xuất hiện trong danh sách và in ra giá trị đó. Sau cùng, chúng ta loại bỏ tất cả mục bắt đầu với chữ c và in danh sách. Tất cả điều này xảy ra chỉ trong một vài dòng mã, minh hoạ các thuật toán STL có thể mạnh mã và tiết kiệm thời gian như thế nào.

Như bạn có thể thấy, với mã tối thiểu, chúng ta đã hoàn thành một lượng chức năng khá lớn. Đối với riêng lý do này, bạn nên xem xét kỹ việc sử dụng lớp STL vector trong mã riêng của bạn.

Kỹ thuật 36: Xoá một mảng phần tử

Tiết kiệm thời gian bằng cách

- Sử dụng hàm delete với các mảng
- Xoá một phần tử ra khỏi một mảng
- Xoá toàn bộ một mảng
- Kết hợp những phương thức xoá với những phương thức cấp phát thích hợp
- Hiểu kết quả

Sử dụng hàm delete với các mảng (array) có thể là một trong những cấu trúc gây bối rối nhất trong C++. Khi nào bạn cần xoá một phần tử đơn trái với toàn bộ một mảng phần tử? Điều gì xảy ra nếu bạn không sử dụng phương thức xoá thích hợp với phương thức cấp phát thích hợp? Việc xoá một phần tử đơn khi bạn cấp phát một mảng phần tử sẽ dẫn đến việc không gọi được các destructor. Việc không xoá một phần tử cấp phát sẽ dẫn đến sự rò rỉ bộ nhớ và những đổ vỡ hệ thống tiềm ẩn. Nói chung, phương thức xoá chính xác phải được kết hợp với việc gọi ra thích hợp phương thức mới. Việc không thực hiện sự kết hợp này sẽ dẫn đến những rò rỉ bộ nhớ rất khó truy tìm - sau cùng điều này sẽ làm đổ vỡ ứng dụng của bạn. Nếu bạn thực hiện trước việc kết hợp những cấp phát và huỷ cấp phát chính xác, bạn sẽ tiết kiệm nhiều thời gian trong tiến trình gỡ rối và bảo trì.

Xem xét việc việc cấp các mảng và pointer

Cách thật sự duy nhất để hiểu chính xác những cấp phát và huỷ cấp phát diễn ra như thế nào là xem một ví dụ về việc cấp phát các đối tượng đơn có và không có các pointer trong các đối tượng đơn hoặc mảng đối tượng. Hãy làm điều này ngay bây giờ với một ví dụ mẫu ngắn.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên ch36.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gọ nhập mã từ Listing 36.1 vào file.

Listing 36.1: Cấp phát các đối tượng có và không có các pointer.

```
#include <stdio.h>
#include <vector>
class NoPointer
{
    int x;
public:
    NoPointer()
    {
       printf("NoPointer: Void Constructor
    called\n");
       x = 0;
    }
    NoPointer( int num )
    {
       printf("NoPointer: Full constructor
    called\n");
       x = num;
    }
    NoPointer( const NoPointer& aCopy )
    {
       printf("NoPointer: Copy constructor
    called\n");
       x = aCopy.x;
```

```
    }
    virtual ~NoPointer()
    {
        printf("NoPointer: Destructor
    called\n");
    }
    NoPointer operator=( const NoPointer&
    aCopy )
    {
        printf("NoPointer: operator=
    called\n");
        x = aCopy.x;
        return *this;
    }
    void setX( int num )
    {
        x = num;
    }
    int getX (void)
    {
        return x;
    }
};
class PointerClass
{
    char *ptr;
public:
    PointerClass()
    {
        printf("PointerClass: Void Constructor
        called\n");
        ptr = NULL;
    }
    PointerClass( const char *str )
    {
```

```
        printf("PointerClass: Full constructor
        called\n");
        ptr = new char[ strlen(str)+1 ];          → 1
        strcpy( ptr, str );
    }
    PointerClass( const PointerClass& aCopy )
    {
        printf("PointerClass: Copy constructor
        called\n");
        if ( aCopy.ptr )
        {
    ptr = new char[ strlen(aCopy.ptr)+1 ];
            strcpy( ptr, aCopy.ptr );
    }
    else
        ptr = NULL;
    }
    virtual ~PointerClass()
    {
        printf("PointerClass: Destructor
        called\n");
        delete [] ptr;                            → 2
}
PointerClass operator=( const
PointerClass& aCopy )
{
    printf("PointerClass: operator=
    called\n");
    if ( aCopy.ptr )
    {
    ptr = new char[ strlen(aCopy.ptr)+1 ];
        strcpy( ptr, aCopy.ptr );
    }
    else
            ptr = NULL;
```

```
        return *this;
    }
    void setPtr( const char *str )
    {
        if ( str )
        {
        str = new char[ strlen(str)+1 ];
            strcpy( ptr, str );
        }
        else
            ptr = NULL;
    }
    const char *getPtr (void)
    {
        return ptr;
    }
};
int main()
{
    // Just create one of each to see what
    happens.
    printf("Creating np\n");
    NoPointer *np = new NoPointer(5);
    printf("Creating pc\n");
    PointerClass *pc = new
    PointerClass("Hello");
    printf("Deleting np\n");
    delete np;
    printf("Deleting pc\n");
    delete pc;
    // Now, create an array of them.
    printf("Creating npa\n");
    NoPointer *npa = new NoPointer[5];          → 3
    printf("Creating pca\n");
    PointerClass *pca =
```

```
        new PointerClass[5];                                    → 4
        // Delete them.
        printf("Deleting npa\n");
        delete npa;
        printf("Deleting pca\n");
        delete pca;
        // Now, do it the right way.
        printf("Creating npa2\n");
        NoPointer *npa2 = new NoPointer[5];
        printf("Creating pca2\n");
        PointerClass *pca2 = new PointerClass[5];
        // Delete them.
        printf("Deleting npa2\n");
        delete [] npa2;
        printf("Deleting pca2\n");
        delete [] pca2;
        // See what happens with a vector.
        printf("Creating vector of
        PointerClass\n");
        std::vector< PointerClass > *pcv = new
        std::vector<PointerClass>;
        for ( int i=0; i<5; ++i )
        {
            PointerClass pc;
            pcv->insert(pcv->end(), pc );
        }
        printf("Deleting vector of
        PointerClass\n");
        delete pcv;
    }
```

Như bạn có thể thấy từ code listing ở trên, chúng ta đã tạo hai loại lớp khác nhau. Lớp thứ nhất, NoPointer là một lớp đơn giản chứa chỉ các biến thành viên cơ bản không có các pointer được lưu trữ trong dữ liệu thành viên của lớp. Lớp thứ hai, PointerClass chứa dữ liệu pointer được cấp phát khi một instance của lớp được tạo và được huỷ cấp

phát khi instance được giải phóng. Nếu bạn xem dòng → **1**, bạn sẽ thấy biến thành viên ptr được cấp phát như thế nào trong constructor cho PointerClass. Biến thành viên ptr đó sẽ được huỷ cấp phát tại dòng → **2** nếu mọi thứ diễn ra tốt đẹp trong lớp.

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

   Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 36.2 trong cửa sổ shell.

Listing 36.2: Kết quả từ ví dụ cấp phát

```
$ ./a.exe
Creating np
NoPointer: Full constructor called
Creating pc
PointerClass: Full constructor called
Deleting np
NoPointer: Destructor called
Deleting pc
PointerClass: Destructor called
Creating npa                                    → 7
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
Creating pca
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
Deleting npa                                    → 5
NoPointer: Destructor called
Deleting pca                                    → 6
```

```
PointerClass: Destructor called
Creating npa2
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
NoPointer: Void Constructor called
Creating pca2
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
PointerClass: Void Constructor called
Deleting npa2
NoPointer: Destructor called
NoPointer: Destructor called
NoPointer: Destructor called
NoPointer: Destructor called
NoPointer: Destructor called
Deleting pca2
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
Creating vector of PointerClass
PointerClass: Void Constructor called
PointerClass: Copy constructor called
PointerClass: Destructor called
PointerClass: Void Constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Void Constructor called
```

```
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Void Constructor called
PointerClass: Copy constructor called
PointerClass: Destructor called
PointerClass: Void Constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Copy constructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
Deleting vector of PointerClass
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
PointerClass: Destructor called
```

Phần quan trọng của kết quả này được minh hoạ trong các dòng được đánh dấu → **5** và → **6**. Như bạn có thể thấy chúng ta huỷ cấp phát các pointer mà chúng ta đã cấp phát trong mã tại các dòng được đánh dấu → **3** và → **4**. Trong cả hai trường hợp, chúng ta đã cấp phát một mảng đối tượng. Nếu bạn xem kết quả, bạn sẽ thấy rằng constructor cho lớp được gọi nhiều lần (xem các dòng bắt đầu với → **7**), nhưng destructor được gọi chỉ một lần. Điều này tạo ra một rò rỉ bộ nhớ tức thì trong ứng dụng.

Hãy luôn đặt các câu lệnh gỡ rối print trong các constructor và destructor để kiểm chứng rằng bạn đã gọi mỗi phương thức này với số lần thích hợp và đúng cách.

Kỹ thuật 37: Tạo các mảng đối tượng

Tiết kiệm thời gian bằng cách:

- Tìm hiểu những cách khác nhau để tạo các mảng đối tượng
- Khai báo các mảng trên ngăn xếp (stack)
- Tạo các đối tượng trên heap
- Sử dụng các lớp đối tượng chứa STL để tạo các mảng
- Hiểu kết quả

C++ cung cấp ba cách khác nhau để tạo các mảng đối tượng, sử dụng các cấu trúc ngôn ngữ và Standard Template Library (STL):

- **Khai báo các mảng trên ngăn xếp (stack):** Phương thức này, sử dụng cấu trúc [ ] tạo một mảng stactic (tĩnh) vốn hiện hữu từ thời điểm nó được cấp phát. Bạn tạo một mảng tĩnh bằng cách viết:

```
object - type array [ num-elements];
```

Trong đó object-type là loại đối tượng mà bạn muốn tạo một mảng của đối tượng đó và số phần tử trong mã được tượng trưng bởi tham số num-elements. Ưu điểm của phương pháp này là kích cỡ mảng được biết vào thời gian biên dịch và chương trình sẽ được tải lượng bộ nhớ thích hợp. Vấn đề với lớp này là mảng hiện hữu chỉ miễn là mảng nằm trong phạm vi (scope). Khi biến mảng nằm ngoài phạm vi out of scope, mảng bị huỷ.

- **Tạo các đối tượng trên heap:** Phương pháp này có ưu điểm là cho phép bạn điều khiển chính xác khi nào mảng được tạo hoặc được huỷ - nhưng một khuyết điểm là nó sẽ không tự động "làm sạch" (huỷ cấp phát). Nếu bạn không huỷ cấp phát một mảng (hoặc nếu bạn huỷ cấp phát nó không đúng cách), bạn tạo ra một rò rỉ bộ nhớ trong chương trình. Bạn tạo một đối tượng trên heap bằng cách viết

```
object-type object;
```

- **Sử dụng các lớp đối tượng chứa STL để tạo các mảng:** Ưu điểm của phương pháp này là nó không tạo những đối tượng cho đến khi chúng thực sự được chèn vào lớp đối tượng chứa - và nó tự động huỷ những đối tượng đó khi lớp đối tượng chứa nằm ngoài phạm vi. Các đối tượng chứa (container) có thể được tạo động trên ngăn xếp để bạn cũng có thể điều khiển phạm vi của các đối tượng trong đối tượng chứa. Hai khuyết điểm của phương pháp này: việc tăng hao phí đáng kể và các lớp đối tượng chứa đòi hỏi bạn thực thi một số phương thức trong các lớp - chẳng hạn như các constructor

copy, toán tử gán và phương thức huỷ tạo ảo (virtual destructor) - để tránh những rò rỉ bộ nhớ. Bạn tạo một lớp đối tượng chứa STL bằng cách viết:

```
vector<object - type array;
```

Kỹ thuật này xem những ưu điểm và khuyết điểm khác nhau của tạo các đối tượng bằng ba cách khác nhau này. Bằng cách hiểu bạn có thể hầu như dễ dàng tạo một mảng đối tượng trong mã như thế nào, bạn sẽ tiết kiệm thời gian khi viết, gỡ rối và tối ưu hoá.

Các bước sau đây dẫn dắt bạn qua ba kỹ thuật cấp phát mảng đối tượng trong một ví dụ:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch37.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 37.1 vào file.

Listing 37.1: Tạo một mảng đối tượng

```
#include <stdio.h>
#include <string>
#include <vector>
class Foo
{
    std::string _str;
public:
    Foo(void)
    {
       printf("Foo: Void constructor\n");
       _str = "";
    }
    Foo ( const char *s )
    {
       printf("Foo: Full constructor
    [%s]\n", s);
       _str = s;
    }
    Foo( const Foo& aCopy )
```

```
    {
        printf("Foo: Copy constructor\n");
        _str = aCopy._str;
    }
    virtual ~Foo()
    {
        printf("Foo: Destructor\n");
    }
    Foo operator=(const Foo& aCopy)
    {
        _str = aCopy._str;
        return *this;
    }
    std::string String()
    {
        return _str;
    }
    void setString( const char *str )
    {
        _str = str;
    }
};
int main()
{
    printf("Creating array via new\n");
    Foo *f = new Foo[2]("Hello");                    → 1
    printf("Creating array on heap\n");
    Foo f1[3];                                       → 2
    // Create a vector
    printf("Creating vector of foo\n");
    std::vector<Foo> fooVector;                      → 3
    printf("Adding objects to vector\n");
    Foo f2;
    Foo f3;
    Foo f4;
```

```
        fooVector.insert( fooVector.end(), f2 );
        fooVector.insert( fooVector.end(), f3 );
        fooVector.insert( fooVector.end(), f4 );
        printf("Deleting array on heap\n");
        delete [] f;
}
```

Xem code listing ở trên, bạn có thể thấy có ba cách khác nhau được định nghĩa trong chương trình. Tại → **1**, chúng ta cấp phát một mảng từ ngăn xếp, sử dụng toán tử new. Tại → **2**, chúng ta cấp phát một mảng trên heap, sử dụng cú pháp mảng chuẩn của C++. Sau cùng, tại → **3**, chúng ta thấy lớp vector Standard Template Library được sử dụng như thế nào để định nghĩa một mảng đối tượng.

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

   Khi chương trình được chạy, nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 37.2 trong cửa sổ shell.

Listing 37.2: Kết quả từ chương trình cấp phát mảng

```
$ ./a.exe
Creating array via new                    → 4
Foo: Full constructor [Hello]
Foo: Full constructor [Hello]
Creating array on heap                    → 5
Foo: Void constructor
Foo: Void constructor
Foo: Void constructor
Creating vector of foo                    → 6
Adding objects to vector
Foo: Void constructor
Foo: Void constructor
Foo: Void constructor
Foo: Copy constructor
Foo: Copy constructor
Foo: Copy constructor
```

```
Foo: Destructor
Foo: Copy constructor
Foo: Copy constructor
Foo: Copy constructor
Foo: Destructor
Foo: Destructor
Deleting array on heap                    → 7
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor
Foo: Destructor                           → 8
Foo: Destructor
Foo: Destructor
```

Như bạn có thể thấy mảng tĩnh được tạo tại thời điểm trình biên dịch tìm thấy mã cho nó. Mảng động (dynamic) được tạo tại thời điểm toán tử new được sử dụng để cấp phát nó và mảng vector không bắt đầu cấp phát không gian cho những đối tượng cho đến khi chúng được chèn vào vector. Hãy xem việc phân tích các dòng được minh hoạ trong Listing 37.2.

- → **4** Dòng này cho thấy thời điểm mảng được cấp phát bằng new. Như bạn có thể thấy hai lệnh gọi constructor được thực hiện, biểu thị hai đối tượng được tạo và được đưa vào mảng. Chú ý rằng bởi vì chúng ta đã cho toán tử new một đối số constructor, nó gọi constructor đầy đủ cho lớp.
- → **5** Dòng này cho thấy nơi chúng ta đã cấp phát một khối đối tượng trên heap, sử dụng cú pháp mảng C++. Chú ý constructor void được sử dụng cho những đối tượng này khởi tạo tất cả ba trong số đối tượng đó (một đối tượng cho mỗi phần tử mảng).
- → **6** Dòng này cho thấy nơi chúng ta đã sử dụng lớp vector Standard Template Library để lưu trữ các đối tượng. Các đối tượng đã được tạo trong lệnh gọi này. Sau đó chúng ta cấp phát một số đối tượng trên heap và thêm chúng vào vector. Chú ý các đối tượng được sao chép vào vector bằng cách sử dụng constructor copy để tạo các instance mới của lớp.

- → 7 Ở đây chúng ta xoá mảng mà chúng ta đã cấp phát bằng lệnh gọi new. Sẽ có hai đối tượng bị xoá trong tiến trình này và kết quả cho thấy thật ra có hai đối tượng bị huỷ.
- → 8 Dòng này cho thấy việc huỷ sau cùng các đối tượng đã được cấp phát trong mảng trên heap.

Nếu bạn đếm tổng số lệnh gọi destructor ở cuối lisiting kết quả, bạn sẽ thấy có 11 lệnh gọi này. Điều này có thể dường như không rõ ràng từ việc chúng ta đã cấp phát chỉ tám đối tượng trong chương trình chính (hai đối tượng trong mảng new, ba đối tượng trong mảng heap và ba đối tượng trong vector). Tuy nhiên, bởi vì constructor copy đã được gọi ra ba lần, thêm ba đối tượng nữa đã được tạo, tổng cộng là 11 đối tượng.

Một điều quan trọng cần chú ý trong mã là chúng ta luôn sử dụng phương thức delete [ ] để hủy cấp phát các mảng đối tượng. Nếu bạn thay thế phương thức delete [ ] bằng một lệnh gọi delete đơn giản, bạn sẽ thấy rằng mã không gọi destructor cho mỗi thành viên của mảng và rằng các rò rỉ bộ nhớ có thể dễ dàng xảy ra. Rủi ro rò rỉ bộ nhớ này đặc biệt quan trọng nếu bạn lưu trữ các pointer dẫn sang những đối tượng trong mảng và truy cập chúng thông qua bất kỳ loại lớp cơ sở.

Kỹ thuật 38: Làm việc với các mảng pointer đối tượng

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các mảng pointer đối tượng
- Thực thi các mảng pointer đối tượng
- Hiểu kết quả

Mặc dù các mảng đối tượng đơn giản dễ làm việc, nhưng các mảng pointer biểu thị những đối tượng thì xử lý hơi phức tạp hơn. Cú pháp để tạo và xoá những mảng này hơi khó hơn; các mảng pointer không thuần nhất vốn trỏ sang một đối tượng cơ sở chung đòi hỏi nhiều công việc hơn - và kỹ thuật này dẫn dắt bạn qua những gì phải được thực hiện. C++ cho phép bạn lưu trữ các pointer dẫn sang tất cả lớp liên quan - đó là những lớp được dẫn xuất từ một lớp cơ sở chung - trong một mảng đơn trong khi theo dõi các kiểu và kích cỡ của chúng. Điều này có thể tiết kiệm cho bạn nhiều thời gian bằng cách cho phép bạn đặt tất cả đối tượng liên quan trong một mảng trong khi xử

lý chúng một cách khác nhau bằng cách sử dụng các kiểu dẫn xuất của chúng.

Tạo một mảng đối tượng không thuần nhất

Nếu bạn đang làm việc với một lô lớp khác nhau, tất cả được dẫn xuất từ một lớp cơ sở đơn, việc lưu trữ tất cả chúng trong một nơi có thể có lợi. Một mặt bạn chỉ có một mảng để làm việc. Mặt khác, bởi vì các đối tượng đều liên quan, có thể bạn sẽ thực hiện cùng một công việc xử lý trên tất cả đối tượng đó cùng một lúc. Hãy xem một ví dụ về việc tạo một mảng không thuần nhất vốn lưu trữ nhiều lớp đối tượng.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch38.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 38.1 vào file.

Listing 38.1: Tạo một mảng pointer đối tượng

```
#include <stdio.h>
#include <string.h>
class Base
{
    char *ptr;
public:
    Base(void)
    {
       ptr = NULL;
    }
    Base( const char *str )
    {
       setString( str );
    }
    virtual ~Base()
    {
       printf("Base::~Base called\n");
       delete ptr;
    }
```

```
    void setString( const char *str )
    {
        ptr = new char[strlen(str)+1];
        strcpy( ptr, str );
    }
    const char *getString()
    {
        return ptr;
    }
};
class Derived : public Base
{
private:
    int _num;
public:
    Derived(void)
        : Base("DerivedVoid")
    {
        _num = 0;
    }
    Derived( int nVal )
        : Base("DerivedFull")
    {
        _num = nVal;
    }
    virtual ~Derived()
    {
        printf("Derived::~Derived called\n");
    }
        void setVal( int nVal )
        {
            _num = nVal;
    }
    int getVal ( void )
    {
```

```
        return _num;
    }
};
const int NumElements = 3;
int main(void)
{
    Base **bArray = new Base*[10];                    → 1
    for ( int i=0; i<NumElements; ++i )
        bArray[i] = new Derived(i);                   → 2
    // Print them out
    for ( int j=0; j<NumElements; ++j )
        printf("Object %s - %d\n", bArray[j]-
        >getString(), ((Derived *)bArray[j])-
        >getVal());
    // Delete them
    for ( int i=0; i<NumElements; ++i )
        delete bArray[i];
    delete [] bArray;
    return 0;
}
```

Code listing ở trên minh hoạ cách chúng ta tạo một mảng pointer và lưu trữ dữ liệu trong mảng đó như thế nào. Như bạn có thể thấy tại → **1**, cấp phát một mảng pointer không khác biệt so với cấp phát bất kỳ loại mảng khác trong C++. Sự khác biệt ở đây là trong khi không gian mảng được cấp phát, các đối tượng thật sự không được tạo ra. Đây là do chúng ta cấp phát không gian cho các pointer dẫn sang những đối tượng, không phải chính các đối tượng. Việc cấp phát thật sự các đối tượng và không gian mà chúng chiếm được minh hoạ tại dòng được đánh dấu là → **2**. Chú ý mặc dù chúng ta có một mảng pointer Base, nhưng chúng ta có thể tạo và lưu trữ các pointer Derived trong mảng vì chúng là một dạng dẫn xuất của Base.

3. Lưu file nguồn trong code editor và đóng code editor.
4. Biên dịch mã nguồn bằng trình biên dịch mà bạn chọn trên hệ điều hành mà bạn chọn.

Khi chương trình được chạy, nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ shell:

```
$ ./a.exe
Object DerivedFull - 0
Object DerivedFull - 1
Object DerivedFull - 2
Derived::~Derived called
Base::~Base called
Derived::~Derived called
Base::~Base called
Derived::~Derived called
Base::~Base called
```

Kết quả ở đây minh hoạ rằng mảng pointer được tạo như mong đợi. Chuỗi được lưu trữ trong lớp Base đã được tạo từ constructor Derived, đây là những gì mà chúng ta dự tính trước. Việc huỷ các đối tượng móc nối hướng lên để gọi các phương thức huỷ tạo lớp Base và lớp Derived. Tóm lại, mã này làm việc chính xác như quảng cáo.

Khả năng làm việc với một mảng pointer không thuần nhất rất mạnh mẽ trong C++ bởi vì nó có nghĩa rằng bạn không cần biết loại đối tượng nào mà bạn làm việc. Nếu chúng ta tạo các phương thức ảo để nhận và xác lập các giá trị trong lớp Base, thậm chí chúng ta không cần phải cast đối tượng trong printf trong hàm main.

Kỹ thuật 39: Thực thi một bảng tính

Tiết kiệm thời gian bằng cách:

- Tạo một phần thực thi bảng tính (spreadsheet) đơn giản
- Tạo lớp Column
- Tạo lớp Row
- Tạo lớp spreadsheet
- Test bảng tính

Một trong những ứng dụng nổi tiếng nhất (hoặc có lẽ khét tiếng xấu) để giúp làm cho máy tính cá nhân trở nên thông dụng là bảng tính (spreadsheet) - chỉ là một lưới các ô được sắp xếp trong các hàng và cột - nói cách khác là một mảng hai chiều. Các bảng tính có nhiều chức năng hơn các mảng đơn giản (ví dụ bạn có thể tạo một công thức nhưng về cơ bản một bảng tính là một mảng các hàng (row) và cột (column). Kỹ thuật này sử dụng Standard Template Library (STL) để thiết lập một shell bảng tính, kết quả có thể dễ dàng được sử dụng để tạo một phần thực thi bảng tính thật sự. Các bảng tính là những phần

tử thông thường của các ứng dụng ngày nay từ việc trình bày dữ liệu cho đến phân tích tình huống "điều gì xảy ra nếu". Bằng cách có một lớp bảng tính generic mà bạn có thể thả vào project kế tiếp, bạn sẽ thấy rằng bạn tiết kiệm nhiều thời gian trong giai đoạn thiết kế và thực thi của project.

Các điểm cơ bản của bảng tính là ba phần tử: cột (hoặc ô), hàng, và chính sheet. Mỗi cột chứa một mẩu dữ liệu và thông tin để định dạng mẩu dữ liệu đó để hiển thị. Cột chứa các phương thức để sao chép chính nó, xoá sạch chính nó, chỉnh sửa dữ liệu hoặc thông định dạng trong chính nó. Hàng đơn giản là một mảng cột vốn tạo nên một hàng đợi của bảng tính. Lớp Row cần có khả năng chỉnh sửa bất kỳ cột hiện có trong hàng cũng như thêm các cột mới và loại bỏ các cột. Một hàng nên có khả năng trả về nội dung của bất kỳ cột nào đó trong hàng đó sao cho người dùng cuối có thể chỉnh sửa trực tiếp nội dung.

Sau cùng, lớp Spreadsheet sẽ chứa một mảng hàng. Mảng này không biết gì về các cột riêng lẻ trong sheet, nó cũng không biết gì về định dạng hoặc dữ liệu. Sự đóng gói dữ liệu này nhất quán với mô hình hướng đối tượng nhưng cũng quan trọng trong khả năng chỉnh sửa dễ dàng các lớp cơ bản của hệ thống với sự thay đổi tối thiểu đối với các lớp phía trên.

Tạo lớp Column

Phần tử đầu tiên của bảng tính là cột. Hãy tạo một lớp đơn giản duy trì thông tin về cột và chứa các phương thức để làm việc với thông tin đó. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch39.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 39.1 vào file.

Listing 39.1: Lớp Column

```
#include <stdio.h>
#include <string>
#include <vector>
class Column
{
    std::string _format;
    std::string _value;
public:
```

```
    Column(void)
    {
        _format = "%s";
        _value = "";
    }
    Column( const char *format, const char
    *value )
    {
        _format = format;
        _value = value;
}
    Column( const Column& aCopy )
    {
        _format = aCopy._format;
        _value = aCopy._value;
    }
    virtual ~Column()
    {
    }
    Column operator=( const Column& aCopy )
    {
        _format = aCopy._format;
        _value = aCopy._value;
        return *this;
    }
    Column operator=(const char *value)
    {
        _value = value;
        return *this;
    }
    void setValue( const char *value )
    {
        _value = value;
    }
    std::string getValue( void )
```

```
    }
        return _value;
    }
    void setFormat( const char *format )
    {
        _format = format;
    }
    std::string getFormat( void )
    {
        return _format;
    }
    virtual std::string getFormattedString
    ( void ) const                                    → 1
    {
            char szBuffer[ 100 ];
            sprintf(szBuffer, _format.c_str(),
              value.c_str());
            std::string sRet = szBuffer;
            return sRet;
    }
};
```

Mã này thực thi phần tử cơ bản nhất của hệ thống bảng tính, cột (column). Như bạn có thể thấy, chúng ta thực thi một lớp hoàn chỉnh bằng cách thêm những phương thức cho các constructor, destructor, toán tử gán và phương thức accessor. Nó cũng thực thi một phương thức ảo (được minh hoạ tại → 1) để trả về nội dung của cột bằng một cách định dạng. Điều này sẽ cho phép những loại cột khác được định nghĩa sau đó ở dưới dòng nếu bạn muốn như vậy.

3. Lưu file trong code editor.

Bước kế tiếp là thực thi lớp Row vốn sẽ chứa một mảng cột.

Tạo lớp Row

Sau khi tạo một lớp Column, điều kế tiếp cần làm là tạo một lớp Row chứa các cột mà chúng ta muốn lưu trữ trong bảng tính. Bạn có thể xem lớp Column như là dữ liệu cho một ô đơn và lớp Row như là một danh sách các ô cho một hàng nào đó.

1. Thêm mã từ Listing 39.2 vào cuối file.

Listing 39.2: Lớp Row

```
class Row
{
    std::vector< Column > _columns;
    void Copy( const Row& aCopy )
    {
       std::vector< Column >::const_
          iterator iter;
       for ( iter = aCopy._columns.begin();
          iter != aCopy._columns.end();
          ++iter )
          _columns.insert( _columns.end(),
          (*iter) );
    }
public:
    Row(void)
    {
    }
    Row( unsigned int numColumns )
    {
       for ( int i=0; i<numColumns; ++i )
       {
          Column c;
          _columns.insert( _columns.end(),
    c );
       }
    }
    Row( const Row& aCopy )
    {
       Copy( aCopy );
    }
    Row operator=( const Row& aCopy )
    {
       Copy( aCopy );
```

```
    }
    Column& operator[]( int idx )                    → 3
    {
        if ( idx < 0 || idx > _columns.
        size()-1 )
            throw "Row: Index out
            of range";                               → 2
            return _columns[ idx ];
    }
    int NumColumns( void )
    {
        return _columns.size();
    }
    void Clear()
    {
        std::vector< Column >::iterator
        iter;
        for ( iter = _columns.begin();
        iter != _columns.end(); ++iter )
        (*iter).setValue( "" );
    }
    void Print() const
    {
        std::vector< Column >::const_
        iterator iter;
        for ( iter = _columns.begin();
        iter != _columns.end(); ++iter )
            printf("%s ",
     (*iter).getFormattedString().c_str() );
        printf("\n");
    }
};
```

Chú ý lớp Row không làm bất cứ điều gì với các cột ngoại trừ lưu trữ chúng và cho người dùng cuối truy cập các cột mà họ muốn. Cũng chú ý chúng ta sử dụng việc xử lý ngoại lệ (được minh hoạ tại → 2) để

giải quyết những tài liệu ngoại lệ của các index mảng nằm ngoài biên. Không có những xác lập mặc định tốt ở đây, do đó chúng ta chỉ giả định rằng đòi hỏi một số cột không hợp lệ là một lỗi nghiêm trọng.

Một điều có thể thay đổi ở đây là lớp Row không xử lý việc định lại kích cỡ. Thay vào đó, lớp Row đơn giản giả định rằng mảng cột luôn được thể hiện cụ thể ngay từ đầu. Để định lại kích cỡ một hàng một cách thích hợp, bạn cần tạo một mảng cột mới có đúng kích cỡ và sau đó sao chép kích cỡ hiện có vào hàng đó.

2. Lưu file.

Bước cuối cùng của tiến trình thực thi lớp là lắp ghép lớp Spreadsheet thật sự. Phần kế tiếp sẽ hướng dẫn cách thực hiện.

Tạo lớp Spreadsheet

Sau cùng chúng ta đi đến phần quan trọng cho người dùng cuối: chính lớp Spreadsheet. Dĩ nhiên, một bảng tính đơn giản là một danh sách các hàng tạo nên sheet vốn lần lượt là một danh sách các cột tạo nên mỗi hàng. Bảng tính của chúng ta sẽ luôn "vuông" - nghĩa là nó sẽ chứa số cột bằng nhau trong mỗi hàng.

1. Thêm mã từ Listing 39.3 vào cuối file.

Listing 39.3: Lớp Spreadsheet

```
class Spreadsheet
{
    int _cols;
    std::vector< Row > _rows;
    std::string _name;
    void _BuildSheet( int nRows, int nCols )
    {
       // If there is anything already
       here, remove it.
       _rows.erase(_rows.begin(),
       _rows.end());
       // Now, add in the rows.
       for ( int i=0; i<nRows; ++i )
       {
          Row row(nCols);
          _rows.insert( _rows.end(),
```

```
                row );
            }
        }
        void _InternalSetRows(`const unsigned
            int nRows )
        {
            _BuildSheet( nRows, _cols );
        }
        void _InternalSetCols( const unsigned
            int nCols )
        {
            // Save the number of rows, so we
            can rebuild it.
            int nRowCount = _rows.size();
            // Set the number of columns.
            _cols = nCols;
            // Now rebuild the rows.
            _BuildSheet( nRowCount, nCols );
        }
        void Copy( const Spreadsheet& aCopy )
        {
            _InternalSetCols( aCopy.
            NumColumns() );
            std::vector< Row >::const_iterator
            iter;
            for ( iter = aCopy._rows.begin();
            iter != aCopy._rows.end(); ++iter )
                _rows.insert( _rows.end(),
                    (*iter) );
            _name = aCopy._name;
        }
    public:
        Spreadsheet(void)
        {
        }
```

```
        Spreadsheet( const char *name )
        {
            _name = name;
        }
        Spreadsheet( const char *name, unsigned
          int nRows, unsigned int nCols )
        {
        _name = name;
        _InternalSetCols( nCols );
        _InternalSetRows( nRows );
    }
    Spreadsheet( const Spreadsheet& aCopy )
    {
        Copy( aCopy );
    }
    Spreadsheet operator=( const
    Spreadsheet& aCopy )
    {
        Copy( aCopy );
        return *this;
    }
    Row& operator[]( int idx )                    → 4
    {
        if ( idx < 0 || idx > _rows.
        size()-1 )
            throw "Spreadsheet: Index out of
            range";
        return _rows[idx];
    }
    Spreadsheet operator()(int r1, int c1,
      int r2, int c2)
    {
        Spreadsheet ret;
        // Assign the pieces.
        ret.setNumColumns( c2-c1+1 );
```

```
        ret.setNumRows( r2-r1+1 );
        // Now copy over the chunk they want.
        try
        {
            for ( int r = r1; r <= r2; ++r )
                for ( int c = c1; c <= c2;
                ++c )
                    ret[r-r1][c-c1] =
                    (*this)[r][c];
        }
        catch ( ... )
        {
            throw "Spreadsheet: Index out of
            range";
        }
        return ret;
    }
    void setNumColumns( int nCols )
    {
        _InternalSetCols( nCols );
    }
    void setNumRows( int nRows )
    {
        _InternalSetRows( nRows );
    }
    int NumColumns() const
    {
        return _cols;
    }
    int NumRows() const
    {
        return _rows.size();
    }
    void setName( const char *name )
    {
```

```
        _name = name;
    }
    std::string getName( void ) const
    {
        return _name;
    }
    void Print() const
    {
        std::vector< Row >::const_iterator
        iter;
        printf("Sheet: %s\n", _name.c_str()
        );
        for ( iter = _rows.begin(); iter !=
        _rows.end(); ++iter )
        {
            (*iter).Print();
        }
    }
    void Clear()
    {
        std::vector< Row >::iterator iter;
        for ( iter = _rows.begin(); iter !=
        _rows.end(); ++iter )
            (*iter).Clear();
        }
    };
```

Như đã để cập trước đó trong kỹ thuật này, bảng tính thực sự chỉ là nơi chứa các hàng vốn lần lượt là nơi chứa các cột. Lớp Column là một lớp duy nhất "hiểu" dữ liệu được lưu trữ có diện mạo như thế nào hoặc nó được định dạng hoặc sẽ được hiển thị như thế nào. Lớp Spreadsheet cho phép truy cập các hàng riêng lẻ trong sheet mà không cần biết các cột được lưu trữ trong mỗi hàng như thế nào.

2. Lưu file nguồn trong bộ soạn thảo mã nguồn và đóng ứng dụng soạn thảo.

Test bảng tính

Để thấy mã có thực sự hoạt động hay không, hãy thực thi một driver thử nghiệm cho mã. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, mở lại file nguồn cho mã mà bạn vừa tạo.

   Trong ví dụ này, file được đặt tên là ch39.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 39.4 vào cuối file.

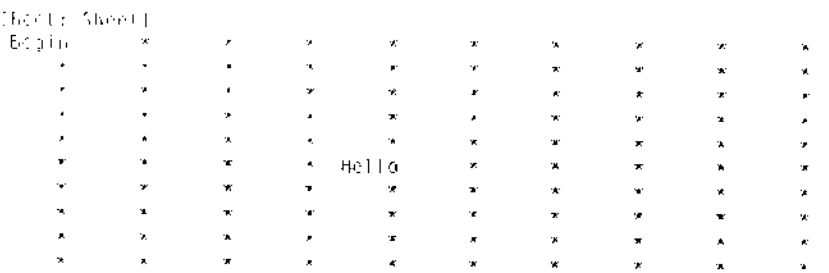
Listing 39.4: Driver thử nghiệm bảng tính

```
int main(int argc, char **argv)
{
    Spreadsheet s1("Sheet1", 10, 10 );
    // Initialize the spreadsheet.
    for ( int i=0; i<s1.NumRows(); ++i )
       for ( int j=0; j<s1.NumColumns();
       ++j )
       {
    s1[i][j] = "*";
    s1[i][j].setFormat("%6s");
    }
    // Set some values.
    s1[5][4] = "Hello";                          → 5
    s1[0][0] = "Begin";
    // Display it so that the user can see
    it.
    s1.Print();
    // Get a slice of the spreadsheet.
    Spreadsheet s2 = s1(0,0,3,3);
    s2.setName("Sheet 2");
    s2.Print();
    // Change a column, so we know that it
       works.
    s2[2][2] = "!";
```

```
    s2.Print();
    // Now, clear out the original sheet and
       display it.
    s1.Clear();
    s1.Print();
}
```

Khi chương trình chạy, nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả từ Listing 39.5 trong cửa sổ shell.

Listing 39.5: Kết quả từ ứng dụng driver thử nghiệm bảng tính



Kết quả cho thấy trạng thái của bảng tính vào thời điểm nó được hiển thị. Một dấu sao (*) hiển thị trong bất kỳ ô không chứa dữ liệu trong khi các ô chứa dữ liệu được thể hiện với giá trị dữ liệu. Ví dụ, bạn sẽ thấy chuỗi Hello ở giữa Sheet 1 đã được đặt ở đó tại → 5 trong code listing cho driver chính. Tương tự, góc trái phía trên của Sheet 1 chứa chuỗi Begin đã được đặt ở đó ở dòng tiếp theo trong chương trình driver.

Chúng ta có thể dễ dàng sử dụng lớp bảng tính này để lưu trữ dữ liệu, hiển thị nó cho người dùng hoặc xử lý dữ liệu chứa trong một định nghĩa hàng / cột.

Các dấu sao chỉ là những placeholder để cho thấy dữ liệu cột thật sự nằm ở đâu. Như bạn có thể thấy các giá trị dữ liệu được xác lập trong driver thử nghiệm xuất hiện nơi chúng phải hiện hữu.

Phần VI
Nhập và xuất

Kỹ thuật 40: Sử dụng các Stream chuẩn để định dạng dữ liệu

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các lớp Stream
- Định dạng dữ liệu với các lớp Stream
- Tìm hiểu kết quả

Nếu bạn đã lập trình trong C++ trong một thời gian dài, có lẽ bạn đã quen xuất dữ liệu bằng các hàm printf, fprintf, và sprintf có từ những ngày lập trình C. Bây giờ đến lúc nhảy vào việc sử dụng các thành phần stream của thư viện C++ chuẩn, bởi vì những thành phần này sẽ tiết kiệm cho bạn nhiều thời gian và tránh được nhiều phiền phức. Các thành phần stream hỗ trợ việc nhập, xuất và định dạng cho dữ liệu trong các ứng dụng C++. Như các hàm printf, fprintf, và sprintf, các stream (luồng) hiện hữu để ghi sang console, sang các file và để định dạng dữ liệu thành các chuỗi (string). Không giống như các hàm được đề cập trước, các stream an toàn kiểu và có thể mở rộng tiết kiệm cho bạn thời gian bằng cách giảm lượng mã bạn cần viết và lượng gỡ rối mà bạn cần thực hiện để tìm các vấn đề trong dữ liệu xuất.

Mặc dù hầu hết nhà lập trình biết bạn có thể nhập xuất dữ liệu qua các lớp stream, nhưng hầu hết không biết rằng các lớp stream có một nguồn chức năng định dạng phong phú cài sẵn.

Kỹ thuật này sẽ hướng dẫn bạn cách làm việc với chức năng định dạng của các lớp stream, cách xuất dữ liệu từ một stream và cách xuất các cột và thay đổi tính chính xác dấu động cho dữ liệu.

Làm việc với các stream

Để hiểu một thành phần stream có thể được sử dụng như thế nào trong ứng dụng để tiết kiệm thời gian và công sức, hãy xem một ví dụ đơn giản về một stream được sử dụng trong một ứng dụng. Trong trường hợp này chúng ta sẽ tạo một số dữ liệu trong chương trình và sau đó xuất dữ liệu đó sang người dùng. Ngoài ra, chúng ta sẽ xem cách mở rộng lớp stream bằng cách tạo control xuất riêng của chúng ta.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này file được đặt tên là ch40.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 40.1 vào file.

Listing 40.1: Sử dụng các stream

```
#include <stdio.h>
#include <iostream>
#include <sstream>
#include <vector>
using namespace std;
void PrintDoubleRow( int numElements, double
    *dArray, ostream& out )
{
    // First, set up some elements of the
       ostream.
    // Set the output floating point precision
       to 2 decimal places.
    out.precision(4);
    // Only show the decimal point if it is
       not a whole number.
    out << showpoint;
    for ( int i=0; i<numElements; ++i )
    {
       // Set each column to be 8 spaces.
       out.width(8);
       // Output the float.
       out << dArray[i];
```

```
    }
    out << endl;
}
template < class A > → 4
ostream& operator<<( ostream& out,
    vector< A >& dVector )
{
    vector<A>::iterator iter;
    for ( iter = dVector.begin(); iter !=
        dVector.end(); ++iter )
    {
        out.width(8);
        out << (*iter);
    }
    out << endl;
    return out;
}
int main(int argc, char **argv)
{
    double dArray[20];
    int nCount = 0;
    // See whether they gave us any on the
        command line.
    if ( argc > 2 )
    {
        for ( int i=1; i<argc; ++i )
        {
            stringstream str;
            double number;
            str.setf(ios::fixed,
            std::ios_base::floatfield);
            str.width(0);
            str.precision(4);
            str << argv[i];
            str >> number;
```

```
            dArray[nCount] = number;
            nCount++;
        }
    }
    else
    {
        // Prompt the user for input.
        bool bDone = false;
        while ( !bDone )
        {
            char szBuffer[80];
            cout << "Enter a number (or a
                dash (*) to quit): ";
            memset( szBuffer, 0, 80 );
            cin >> szBuffer;
            if ( szBuffer[0] == '*' )
                bDone = true;
            else
            {
            stringstream str;
            double number;
            str.setf(ios::fixed,
                std::ios_base::floatfield);
            str.width(0);
            str.precision(4);
            str << szBuffer;
            str >> number;
            dArray[nCount] = number;
            nCount++;
        }
    }
    } PrintDoubleRow( nCount, dArray,
    cout ); → 1
    // Now display it as a vector.
    vector< double > dVector; → 2
```

```
    for ( int i=0; i<nCount; ++i )
        dVector.insert( dVector.end(),
            dArray[i] );
    cout << "Vector: " << endl;
    cout << dVector << endl; → 3
}
```

Hãy xem điều gì đang xảy ra ở đây. Đầu tiên, chúng ta tạo một mảng giá trị double chuẩn và đặt dữ liệu từ người dùng vào mảng này. Sau đó, mảng đó được in ra bằng cách sử dụng lớp stream chuẩn (xem → **1**). Tiếp theo, chúng ta tạo một mảng "array" bằng cách sử dụng lớp Standard Template Library (xem → **2**). Sau đó, chúng ta in ra vector đó bằng cách sử dụng một stream được minh hoạ tại → **3**. Nhưng hãy đợi xem điều này diễn ra như thế nào? Các vector không nằm trong số các kiểu được hỗ trợ chuẩn cho các stream. Nếu bạn xem hàm khuôn mẫu được đánh dấu bằng → **4**, bạn sẽ thấy rằng chúng ta đã tạo một toán tử quá tải vốn lấy một đối tượng vector và xuất nó sang một stream. Trình biên dịch sẽ kết hợp toán tử quá tải cùng với việc tạo stream của vector và bảo đảm rằng tất cả làm việc tốt. Cũng chú ý việc sử dụng các phương thức width và precision của lớp stream để xác lập chiều rộng xuất của mỗi cột trong vector một cách thích hợp và chỉ xuất đúng số dấu thập phân.

3. Lưu file mã nguồn trong code editor và đóng ứng dụng biên soạn.
4. Biên dịch ứng dụng trong trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, khi bạn chạy ứng dụng bằng dữ liệu nhập dòng lệnh sau đây:

```
1 2 3 4
```

bạn thấy kết quả sau đây từ ứng dụng trên cửa sổ console:

```
$ ./a.exe 1 2 3 4
    1.000 2.000 3.000 4.000
Vector:
    1.000 2.000 3.000 4.000
```

Như bạn có thể thấy, kết quả giống nhau cho lớp mảng và lớp vector. Chúng ta có thể thấy chiều rộng của các cột cố định tại 8 ký tự như chúng ta đã xác định trong phương thức width của stream. Sau cùng, chú ý số dấu thập phân được cố định tại ba cho mỗi mục nhập, một lần nữa như được xác định trong phương thức precision.

Hoặc, bạn có thể nhập dữ liệu tại dòng nhắc. Để làm điều này, chạy chương trình không có các đối số đầu vào và sau đó nhập các giá trị khi được nhắc từ người dùng. Trong trường hợp này, bạn thấy kết quả sau đây từ chương trình trong cửa sổ console:

```
$ ./a.exe
Enter a number (or a dash (*) to quit): 1
Enter a number (or a dash (*) to quit): 2
Enter a number (or a dash (*) to quit): 3
Enter a number (or a dash (*) to quit): 4
Enter a number (or a dash (*) to quit): *
    1.000 2.000 3.000 4.000
Vector:
    1.000 2.000 3.000 4.000
```

Kết quả từ chương trình thì như nhau, sự khác biệt duy nhất là cách dữ liệu được đưa vào hệ thống. Một lần nữa, chú ý chiều rộng của các cột vẫn cố định và số dấu thập phân vẫn là những gì mà chúng ta đã xác định.

Kỹ thuật 41: Đọc và xử lý các file

Tiết kiệm thời gian bằng cách:

- Đọc các file với các lớp stream
- Xử lý các file với các lớp stream
- Tạo một file thử nghiệm
- Hiểu kết quả

Xử lý các file trong C++ thật sự giống như bất kỳ loại nhập hoặc xuất khác. Tuy nhiên, không giống như các hàm tương tự trong C, các hàm xử lý file trong C++ cho phép bạn sử dụng cùng một mã để xử lý dữ liệu - từ bàn phím hoặc một file. Tính khái quát này làm cho dễ viết mã vốn dễ test, chạy và bảo trì hơn đáng kể. Và dĩ nhiên khi mã viết nhanh hơn và dễ test hơn, nó tiết kiệm cho bạn thời gian trong project.

Kỹ thuật này hướng dẫn bạn cách sử dụng các lớp file stream để đọc và xử lý một file preferences đơn giản. Kỹ thuật này cũng cho bạn thấy phương pháp này so với kiểu thực hiện cũ như thế nào để bạn có thể dễ dàng đưa mã này vào bất cứ nơi nào bạn sử dụng các hàm kiểu C cũ hơn.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch41.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 41.1 vào file.

Listing 41.1: Lớp đọc file

```
#include <stdio.h>
#include <string.h>
#include <fstream>
#include <ios>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
// The old fashioned way
class Entry
{
private:
    char *strName;
    char *strValue;
    void Init()
    {
       strName = NULL;
       strValue = NULL;
    }
public:
    Entry()
    {
       Init();
    }
    Entry( const char *name, const char *value )
    {
       Init();
       setName( name );
```

```
    setValue ( value );
}
Entry( const Entry& aCopy )
{
    Init();
    setName( aCopy.strName );
    setValue( aCopy.strValue );
}
~Entry()
{
    if ( strName )
        delete [] strName;
    if ( strValue )
        delete [] strValue;
}
Entry operator=( const Entry& aCopy )
{
    setName( aCopy.strName );
    setValue( aCopy.strValue );
    return *this;
}
void setName(const char *name)
{
    if ( strName )
        delete [] strName;
    strName = new char[strlen(name)+1 ];
    strcpy( strName, name );
}
void setValue( const char *value )
{
    if ( strValue )
        delete [] strValue;
    strValue = new char[strlen
    (value)+1 ];
    strcpy( strValue, value );
```

```
    }
    const char *getName( void )
    {
        return strName;
    }
    const char *getValue( void )
    {
        return strValue;
    }
};
bool OpenFileAndReadOld( const char
    *strFileName, Entry* array, int
    nMaxEntries, int *numFound )
{
    FILE *fp = fopen ( strFileName, "r" );
    if ( fp == NULL )
        return false;
    int nPos = 0;
    while ( !feof(fp) )
    {
        char szBuffer[ 257 ];                    → 1
        memset( szBuffer, 0, 256 );
        if ( fgets( szBuffer, 256, fp ) ==
            NULL )
                break;
        // Look for the position of the '='
            sign.
        char *str = strstr(szBuffer, "=");
        if ( str )
        {
            // First, get the name.
            char szName[256];
            memset( szName, 0, 256 );
            strncpy(szName, szBuffer,
            strlen(szBuffer)-strlen(str) );
```

```
            // Now, get the value.
            char szValue[256];
            memset( szValue, 0, 256 );
            strncpy(szValue, str+1,
            strlen(str)-1 );
            if ( szValue[strlen(szValue)-1]
            == '\n' )
                  szValue[strlen(szValue)-1]
                  = 0;
            Entry e( szName, szValue );
            if ( nPos < nMaxEntries )
            {
               array[ nPos ] = e;
               nPos ++;
            }
         }
      }
      *numFound = nPos;
      fclose(fp);
      return true;
   }
```

Mã trong Listing 41.1 thực hiện mọi thứ bằng cách cũ (kiểu C), sử dụng các hàm dựa vào file. Việc thử nó với các stream sẽ tạo ra các toán tử có thể tái sử dụng.

3. Bây giờ thêm mã trong Listing 41.2 vào file nguồn bằng cách sử dụng bộ biên soạn mã nguồn mà bạn ưa thích.

Listing 41.2: Sử dụng các stream để đọc file

```
// Various operators used by the
    application.
ifstream& operator<<( string& sIn, ifstream&
    in )                                          → 3
{
    while ( !in.eof() )
    {
          char c;
```

```
            in.get(c);
            if ( in.fail() )
               return in;
            sIn += c;
            if ( c == '\n' )
               return in;
         }
      return in;
}
string operator-( string& sIn,
      char cIn )                                    → 4
{
      string sOut = "";
      for ( int i=0; i<sIn.length(); ++i )
         if ( sIn[i] != cIn )
            sOut += sIn[i];
      return sOut;
}
bool OpenFileAndReadNew( const char
      *szFileName, std::vector< Entry >&
      entries )
{
      ifstream in;
      in.open( szFileName );
      if ( in.fail() )
      {
         printf("Unable to open file %s\n",
         szFileName );
         return false;
      }
      // Process the file
      while ( !in.eof() )
      {
         // Get an input line
         string sLine = "";
```

```
            sLine << in; □ 2
            // Skip comments
            if ( sLine.length() && sLine[0]
               == '#' ) continue;
            // Remove all carriage returns and
            line feeds
            sLine = sLine - '\n';
            sLine = sLine - '\r';
            // Now, extract the pieces
            int ePos = sLine.find_first_of
               ('=', 0);
            if ( ePos != string::npos )
            {
               // Copy the name
               string name =
                  sLine.substr(0,ePos);
               string value =
                  sLine.substr(ePos+1);
                  Entry e(name.c_str(),
                     value.c_str());
                  entries.insert( entries.end(),
         e );
            }
         }
         in.close( );
         return true;
      }
```

Như bạn có thể thấy, mã dễ hiểu và dễ bảo trì hơn nhiều trong phiên bản stream. Khả năng đọc quan trọng trong việc viết mã bởi vì nhà lập trình bảo trì mất ít thời gian hơn để đọc và hiểu mục đích của bạn. Các phiên bản stream của mã từ phần mô tả riêng của chúng về những gì chúng ta đang cố thực hiện cải thiện hơn các hàm kiểu C gây bối rối. Quan trọng hơn nếu chúng ta muốn test các hàm từ bàn phím, việc chuyển vào đối tượng đầu vào chuẩn thay vì một file là điều bình thường. mã có thể giải quyết cả hai loại đầu vào.

Để hiểu phiên bản stream đơn giản như thế nào so với phiên bản cũ hơn, hãy xem hai đoạn tương tự của mã. Dòng được đánh dấu là → 1 trong listing gốc cho thấy chúng ta đọc một dòng từ nguồn đầu vào như thế nào. Dòng tương ứng trong phiên bản stream cập nhật được đánh dấu là → 2. Chú ý phiên bản stream không chỉ nhỏ hơn và dễ đọc hơn mà nó còn giải quyết các vấn đề mà mã gốc đã không thể làm được. Ví dụ, lớp string có thể xử lý số ký tự hầu như vô hạn trong khi bộ đệm (buffer) được sử dụng trong mã gốc có kích cỡ cố định. Tương tự, phiên bản stream tự động nhập số ký tự vào lớp string nhằm làm cho việc kiểm tra tìm các dòng trống trở nên đơn giản. Sau cùng, việc kiểm tra tìm các chuỗi con cho các lớp stream dễ dàng hơn đáng kể so với sử dụng hàm strstr cũ vốn đã đòi hỏi bạn kiểm tra tìm các giá trị trả về NULL và sự kết thúc của các phép so sánh chuỗi.

4. Lưu mã nguồn dưới dạng một file trong code editor.

Test mã đọc file

Sau khi đã tạo một chức năng đọc file, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác và còn hướng dẫn người ta cách sử dụng mã của bạn như thế nào.

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ hai phương thức nhập dữ liệu (OpenAndReadFile 01 và OpenAndReadFileNew) được sử dụng như thế nào - và kết quả của mỗi phương thức sẽ là gì.

1. Trong code editor mà bạn chọn, mở lại file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch 41.cpp.

2. thêm mã từ Listing 41.3 vào file.

Listing 41.3: Driver thử nghiệm đọc file

```
int main( int argc, char **argv )
{
    if ( argc < 2 )
    {
        printf("Usage ch5_3 filename\n");
        exit(1);
    }
    Entry entries1[50];
    int num = 0;
```

```
printf("Old Way:\n");
f ( OpenFileAndReadOld( argv[1],
  entries1, 50, &num ) == false )
{
   printf("Error processing file %s\n",
     argv[1] );
   exit(1);
}
for ( int i=0; i<num; ++i )
{
   cout << "Entry " << i << endl;
   cout << "Name: " <<
   entries1[i].getName() << endl;
   cout << "Value: " <<
   entries1[i].getValue() << endl;
}
printf("New Way:\n");
std::vector< Entry > entries2;
if ( OpenFileAndReadNew( argv[1],
   entries2 ) == false )
{
   printf("Error processing file %s\n",
   argv[1] );
   exit(1);
}
std::vector< Entry >::iterator iter;
int nPos = 0;
for ( iter = entries2.begin();
iter != entries2.end(); ++iter )
{
   cout << "Entry " << nPos << endl;
   cout << "Name: " <<
   (*iter).getName() << endl;
   cout << "Value: " <<
```

```
            (*iter).getValue() << endl;
            nPos ++;
        }
    }
```

Chương trình đơn giản này chỉ cho phép chúng ta đọc một file test bằng hai cách khác nhau để sử dụng các hàm C kiểu cũ và các hàm stream kiểu mới. Sau đó, chúng ta in ra các giá trị để so sánh chúng. Chú ý hàm kiểu cũ đòi hỏi chúng ta sử dụng một mảng có kích cỡ cố định, trong khi phiên bản stream sử dụng lớp vector để cung cấp số mục nhập hầu như vô hạn.

3. Lưu file mã nguồn và đóng ứng dụng biên tập mã nguồn.

Tạo file test

Để sử dụng driver thử nghiệm, chúng ta cần một số dữ liệu đầu vào để driver xử lý. Hãy tạo một file test đơn giản chứa dữ liệu mà chúng ta có thể đọc để so sánh các hàm nhập và xuất kiểu cũ và kiểu mới.

1. Trong code editor mà bạn chọn, tạo một file text để thử nghiệm ứng dụng.

   Trong trường hợp này, tên test.txt được sử dụng cho file test.

2. Gõ nhập text sau đây vào file test:

   ```
   Config1=A config string
   Config2=100
   Config3=Another line
   # This is a comment
   Config4=A.$5
   ```

3. Lưu file test và đóng ứng dụng code editor.

4. Biên dịch và chạy chương trình bằng trình biên dịch và hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây từ chương trình trên cửa sổ console.

```
$ ./a.exe test.txt
Old Way:                                    → 5
Entry 0
Name: Config1
Value: A config string
Entry 1
```

```
Name: Config2
Value: 100
Entry 2
Name: Config3
Value: Another line
Entry 3
Name: Config4
Value: A.$5
New Way:                                    → 6
Entry 0
Name: Config1
Value: A config string
Entry 1
Name: Config2
Value: 100
Entry 2
Name: Config3
Value: Another line
Entry 3
Name: Config4
Value: A.$5
```

Như bạn có thể thấy qua kết quả của chương trình, cả hai cách cũ và mới để xử lý dữ liệu thì có tác dụng như nhau. Tất cả mục nhập được thể hiện bên dưới dòng kết quả Old Way được đọc bằng cách sử dụng các hàm C kiểu cũ trong khi kết quả được thể hiện bên dưới dòng kết quả New Way được đọc bằng cách sử dụng các stream kiểu mới. Do đó bằng cách sử dụng chương trình đơn giản này, chúng ta có thể dễ dàng chuyển đổi các ứng dụng kiểu cũ thành các hàm kiểu mới một cách nhanh chóng và dễ dàng, tiết kiệm thời gian và công sức.

Không có thêm sự khác biệt về chức năng giữa các hàm C kiểu cũ và các stream C++ kiểu mới. Để thấy điều này, so sánh các dòng được minh hoạ tại → **5** và → **6**. Tuy nhiên, việc bỏ qua các toán tử bổ sung vốn có thể được tái sử dụng ở bất kỳ nơi nào làm cho mã kiểu mới vốn xử lý các stream nhỏ hơn đáng kể và dễ đọc hơn; nếu so sánh mã kiểu cũ vụng về và gây bối rối.

Kỹ thuật 42: Cách đọc các file Delimited

Tiết kiệm thời gian bằng cách

- Tìm hiểu các file delimited chuẩn
- Tạo các lớp generic để đọc hoặc tải các file delimited chuẩn vào ứng dụng
- Test kết quả

Có nhiều cách để lưu trữ dữ liệu trong các hệ thống file. Một trong những cách phổ biến nhất là một file delimited chuẩn, trong đó các trường riêng lẻ của các record được tách biệt bởi các dấu phân cách (delimiter) đã biết. Có rất nhiều ví dụ về điều này từ các file comma serated values (CSV) đến các file XML cho đến các record có kích cỡ cố định vốn chứa các byte rỗng.

Khả năng tải dữ liệu phân cách (delimited) vào ứng dụng rất hữu dụng và nó làm cho ứng dụng dễ sử dụng hơn rất nhiều.

Kỹ thuật này xây dựng một số lớp generic nhằm hỗ trợ tải và phân tích các file delimited. Kỹ thuật này đưa ra một vài giả định ở đây mặc dù từng giả định này khá dễ điều chỉnh nếu bạn cần thay đổi lôgíc. Các giả định như sau:

- Các file chỉ chứa các dấu phân cách vốn tách biệt các trường, nghĩa là các trường trong file không chứa các dấu phân cách.
- Các record hiện hữu một record mỗi dòng. Đây thực sự là điều tiện lợi. Kiểm tra một chữ ký kết thúc record ngoại trừ ký tự kết thúc dòng thì đủ dễ dàng.
- Các trường (field) có thể thay đổi kích cỡ.

Đọc các file phân cách

Do nhu cầu đọc các file phân cách là một vấn đề phổ biến như vậy trong thế giới phát triển phần mềm, chúng ta nên tạo một phương thức generic để đọc chúng. Bằng cách làm điều này, chúng ta có thể tiết kiệm nhiều thời gian bởi vì chúng ta có thể di chuyển mã để đọc các file từ project này đến project khác. Ví dụ, bạn có thể tạo một lớp generic để đọc bất kỳ loại file phân cách và test nó bằng nhiều đầu vào khác nhau.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch42.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 42.1 vào file.

Listing 42.1: Đọc một file phân cách

```
#include <stdio.h>
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
// Avoid having to type out std:: for all
// STL classes.
using namespace std;
// This class manages a list of delimiters.
class Delimiters                                    → 1
{
private:
    // Array of possible delimiters. Use a
    vector,
    // since the characters could include
    NULL byte
    // or other characters that string can't
    handle.
    vector< char > _delimiters;
protected:
    virtual void Copy ( const Delimiters&
        aCopy )
    {
        vector<char>::const_iterator iter;
        for ( iter =
            aCopy._delimiters.begin(); iter !=
            aCopy._delimiters.end(); ++iter )
                _delimiters.insert( _delimiters.
                    end(), (*iter) );
    }
public:
    Delimiters(void)
    {
```

```
    }
    Delimiters( const char *delimiterList )
    {
    for ( int i=0;
i<strlen(delimiterList); ++i)
    {
        _delimiters.insert( _delimiters.
        end(), delimiterList[i] );
    }
}
Delimiters( const Delimiters& aCopy )
{
    Copy ( aCopy );
}
Delimiters operator=( const Delimiters&
    aCopy )
{
    Copy( aCopy);
}
virtual ~Delimiters(void)
{
}
// Clear out the entire list.
virtual void Clear()
{
    _delimiters.erase( _delimiters.
        begin(), _delimiters.end() );
}
// Add a delimiter to the list.
virtual void Add( char c )
{
        _delimiters.insert( _delimiters.
            end(), c );
}
// See whether a given character is in
```

```
        the list.
    virtual bool Contains( char c )
    {
        vector<char>::const_iterator iter;
        for ( iter = _delimiters.begin();
        iter != _delimiters.end(); ++iter )
            if ( c == (*iter) )
                return true;
        return false;
    }
    // Remove a given delimiter.
    virtual bool Remove( char c )
    {
        vector<char>::iterator iter;
        for ( iter = _delimiters.begin();
        iter != _delimiters.end(); ++iter )
            if ( c == (*iter) )
            {
                _delimiters.erase( iter );
                return true;
            }
        return false;
    }
};
// This class manages the data for a given row of a
// delimited file.
class DelimitedRow                                    → 2
{
private:
    vector< string > _columns;
protected:
    virtual void Copy( const DelimitedRow& aCopy )
    {
        vector<string>::const_iterator iter;
     for ( iter = aCopy._columns.begin(); iter != aCopy._columns.end(); ++iter )
```

```
            _columns.insert( _columns.end(), (*iter) );
        }
    public:
        DelimitedRow(void)
        {
        }
        DelimitedRow( const char *col )
        {
            Add( col );
        }
        virtual ~DelimitedRow()
        {
        }
        virtual void Add( const char *col )
        {
            _columns.insert( _columns.end(), col );
        }
        int NumColumns( void )
        {
            return _columns.size();
        }
        string getColumn( int index )
        {
            if ( index < 0 || index > NumColumns()-1 )
                return string("");
            return _columns[index];
        }
};
// This class will handle a single delimited file.
class DelimitedFileParser                                    → 3
{
private:
        string _fileName;
        Delimiters _delim;
        ifstream _in;
```

```
        vector<DelimitedRow> _rows;
    protected:
        virtual void Copy( const DelimitedFileParser& aCopy )
        {
            _fileName = aCopy._fileName;
            _delim = aCopy._delim;
            vector<DelimitedRow>::const_iterator iter;
            for ( iter = aCopy._rows.begin(); iter != aCopy._rows.end(); ++iter )
                _rows.insert( _rows.end(), (*iter) );
        }
        virtual void _ParseLine( string sIn )
        {
            // Given a delimiter list, and an input string, parse through
            // the string.
            DelimitedRow row;
            string sCol = "";
            for ( int i=0; i<sIn.length(); ++i)
            {
                if ( _delim.Contains( sIn[i] ) )
                {
                    row.Add( sCol.c_str() );
                    sCol = "";
                }
                else
                    sCol += sIn[i];
            }
            row.Add( sCol.c_str() );
            _rows.insert( _rows.end(), row );
        }
    public:
        DelimitedFileParser(void)
        {
        }
        DelimitedFileParser( const char *fileName, const char *delimiters )
            : _delim( delimiters )
```

```
{
    Open( fileName );
}
DelimitedFileParser( const DelimitedFileParser& aCopy )
{
    Copy ( aCopy );
}
virtual ~DelimitedFileParser()
{
    _in.close();
}
virtual bool Open( const char
    *fileName )
{
    _fileName = fileName;
    _in.open( _fileName.c_str() );
    return !_in.fail();
}
virtual bool Parse()
{
    // Make sure the file is open.
    if ( _in.fail() )
    {
        return false;
    }
    while ( !_in.eof() )
    {
        string sIn = "";
        while ( !_in.eof() )
        {
            // Get an input line.
            char c;
        _in.get(c);
        if ( _in.fail() )
                break;
```

```
                if ( c != '\r' && c !=
                '\n' )
                sln += c;
                if ( c == '\n' )
                    break;
            }
            // Parse it.
            if ( sln.length() )
                _ParseLine( sln );
        }
        return true;
}
int NumRows()
{
        return _rows.size();
}
DelimitedRow getRow( int index )
{
    if ( index < 0 || index >=
    NumRows() )
            throw "getRow: index out of
                range";
        return _rows[index];
    }
};
```

Mã này thực thi một parser (bộ phân tích cú pháp) geeneric và đối tượng chứa của các file phân cách. Bằng cách xác định dấu phân cách giữa các trường, chỉ báo kết thúc record và tên file nguồn, bạn có thể sử dụng lớp này để đọc và truy tìm tất cả trường riêng lẻ trong file.

Như bạn có thể thấy từ mã, toàn bộ ứng dụng gồm ba lớp, từng lớp quản lý một phần riêng biệt của tiến trình. Tiến trình có ba phần: quản lý file (được minh hoạ tại → **3**), lưu trữ dữ liệu (được minh hoạ tại → **2**) và quản lý dấu phân cách (được minh hoạ tại → **1**).

3. Lưu file nguồn trong bộ soạn thảo mã nguồn và đóng ứng dụng soạn thảo.

Test mã

Sau khi đã tạo chức năng, bạn nên tạo một driver thử nghiệm để bảo đảm không chỉ mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ file parser được sử dụng như thế nào và kết quả là gì.

1. Trong code editor mà bạn chọn, mở lại file hiện có để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch42.cpp.

2. Thêm mã từ Listing 42.2 vào file.

Listing 42.2: Driver thử nghiệm dấu phân cách (delimiter)

```
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
       printf("Usage ch5_2
    <delimitedFile>\n");
       exit(1);
    }
    DelimitedFileParser fileParser( argv[1],
       ":" );
    fileParser.Parse();
    printf("%d Rows found\n",
       fileParser.NumRows() );
    for ( int i=0; i<fileParser.NumRows();
       ++i )
    {
       DelimitedRow row =
          fileParser.getRow(i);
       printf("Row: %d\n", i );
       for ( int j=0; j<row.NumColumns();
       ++j )
          printf("Column %d = [%s]\n", j,
```

```
                row.getColumn(j).c_str() );
        }
    }
```

3. Lưu file mã nguồn và đóng ứng dụng biên soạn mã.
4. Tạo một file test để thử nghiệm ứng dụng trong text editor mà bạn chọn.

   File này được sử dụng để nhập vào ứng dụng và chứa các record phân cách.

   Trong trường hợp này, tên test_delimited được sử dụng cho file test.
5. Gõ nhập text sau đây vào file test:

```
Line 1:Column 2:This is a
    test:100:200:300
Line 2:Column 2:This is another
    test:200:300:400
```

6. Lưu file test và đóng ứng dụng soạn thảo mã.
7. Biên dịch và chạy chương trình bằng trình biên dịch và hệ điều hành mà bạn ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau từ chương trình trên cửa sổ console:

```
$ ./a.exe test_delimited.txt
2 Rows found
Row: 0
Column 0 = [Line 1]
Column 1 = [Column 2]
Column 2 = [This is a test]
Column 3 = [100]
Column 4 = [200]
Column 5 = [300]
Row: 1
Column 0 = [Line 2]
Column 1 = [Column 2]
Column 2 = [This is another test]
Column 3 = [200]
```

```
Column 4 = [300]
Column 5 = [400]
```

Như bạn có thể thấy từ kết quả, parser quyết định đúng rằng có ba record trong file test mà chúng ta đã cho nó để nhập vào. Từng dòng đầu vào chứa năm cột dữ liệu được tách biệt bằng một ký tự dấu hai chấm (:). Bằng cách cho parser biết dấu phân cách là một dấu hai chấm, nó tách mỗi dòng thành các cột riêng lẻ và trả dữ liệu đó trở về người dùng dưới dạng một đối tượng DelimitedRow trong một vector của các đối tượng như vậy.

Bây giờ lớp này có thể được di chuyển từ project này đến project khác trong bất kỳ tình huống nơi chúng ta cần đọc một file phân cách và sử dụng các thành phần riêng lẻ của mỗi record (hoặc dòng) trong file. Điều này tiết kiệm cho chúng ta thời gian trong việc thực thi chức năng lặp đi lặp lại nhiều lần và tiết kiệm cho chúng ta công sức bởi vì mã sẽ được gỡ rối.

Kỹ thuật 43: Viết các đối tượng dưới dạng XML

Tiết kiệm thời gian bằng cách:

- So sánh mã XML với mã C++
- Sử dụng XML để lưu trữ và phục hồi dữ liệu qua lại các lớp C++
- Tạo một lớp XMLWriter
- Test lớp XMLWriter
- Hiểu kết quả

Thuật ngữ hiện hành của thế giới lập trình là XML và XML compatibility (khả năng tương thích XML). XML (eXtended Markup Language) thật sự chỉ là một biến thể của ngôn ngữ hiển thị SGML (mà HTML cũng bắt nguồn từ đó) đã được tối ưu hoá cho việc lưu trữ dữ liệu thay vì hiển thị.

Không có gì ngạc nhiên khi thấy khả năng xuất dữ liệu dưới dạng mã XML đã trở nên rất quan trọng trong thế giới lập trình. Bởi vì cấu trúc của XML rất giống như cấu trúc của C++ - xét về hiển thị phân cấp và các lớp và thuộc tính - bạn có thể dễ dàng sử dụng XML để lưu trữ và phục hồi dữ liệu qua lại các lớp C++.

Dạng chung của một cấu trúc XML như sau:

```
<xml>
    <structure-name>
```

```
        <element-name>
          value
        </element-name>
      </structure-name>
    </xml>
```

Như bạn có thể thấy, nó trông giống như một cấu trúc lớp C++ với một tên cấu trúc là tên của lớp và một tên phần tử là tên của mỗi mẫu dữ liệu thành viên của lớp đó. Sau đây là một ví dụ:

```
Class Foo
{
    int x; // We can think of the semicolon as </int>
    // And so forth
};
```

Trong định nghĩa lớp ở trên, chúng ta có một tên đối tượng (lớp Foo), một phần tử (int) và một giá trị cho phần tử đó (x). Điều này ánh xạ hoàn toàn trực tiếp vào schema chung XML. Định nghĩa ban đầu đã là một đối tượng XML thật sự trong khi định nghĩa này là một lớp C++ thật sự. Tuy nhiên, bạn có thể thấy cái này ánh xạ sang cái kia như thế nào. Chúng ta có thể viết lớp ở trên như sau.

```
<Foo>
    <int> x </int>
</Foo>
```

Và như bạn có thể thấy, hai đối tượng ánh xạ rất tốt. Bằng cách lưu trữ dữ liệu theo định dạng XML, chúng ta có thể đọc dữ liệu không chỉ vào các ứng dụng C++ mà còn vào bất kỳ ứng dụng khác vốn hiểu định dạng XML chẳng hạn như các cơ sở dữ liệu. Lưu trữ dữ liệu theo một định dạng chuẩn đã biết tiết kiệm thời gian bằng việc loại bỏ nhu cầu viết các trình biên dịch (translator) cho các định dạng dữ liệu và bằng cách cho phép bạn sử dụng các ứng dụng hiện có với dữ liệu của bạn. Trong ví dụ này, chúng ta sẽ xem cách viết một lớp C++ theo định dạng XML và sau đó xem cách đọc trở lại dữ liệu XML đó sang một lớp C++.

Tạo XMLWriter

Bước đầu tiên trong tiến trình là thêm khả năng ghi ra dữ liệu cho lớp theo định dạng XML. Chúng ta sẽ gọi phần tử thực hiện điều này là xử lý một đối tượng XMLWriter. Hãy xem một cách generic để tạo một XMLWrite vốn sẽ tiết kiệm cho chúng ta thời gian bằng việc cho phép chúng ta áp dụng chức năng này vào tất cả đối tượng trong hệ thống.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho định nghĩa của lớp.

   Trong ví dụ này, file được đặt tên là ch43.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá cần thiết.

2. Gõ nhập mã từ Listing 43.1 vào file.

Listing 43.1: Lớp XMLWriter

```
#include <stdio.h>
#include <string>
#include <vector>
#include <fstream>
using namespace std;
// Class to manage the storage of the XML
    data.
class XMLElement
{
private:
    string _name;
    string _value;
    vector< XMLElement > _subElements;
protected:
    virtual void Init()
    {
        _name = "";
        _value = "";
        _subElements.erase(
        _subElements.begin(), _subElements.end()
    );
    }
    virtual void Copy( const XMLElement&
    aCopy )
    {
        setName( aCopy.getName().c_str() );
        setValue ( aCopy.getValue().c_str()
    );
```

```
            vector< XMLElement >::const_
               iterator iter;
            for ( iter =
               aCopy._subElements.begin();
                  iter !=
               aCopy._subElements.end();
                        ++iter )
                  _subElements.insert
      ( _subElements.end(), (*iter) );
      }
public:
      XMLElement( void )
      {
         Init();
      }
      XMLElement( const char *name, const
      char *value )
      {
         setName( name );
         setValue( value );
      }
      XMLElement( const char *name, int
        value )
      {
         setName( name );
         char szBuffer[10];
         sprintf(szBuffer, "%d", value );
         setValue( szBuffer );
      }
      XMLElement( const char *name, double
      value )
      {
         setName( name );
         char szBuffer[10];
         sprintf(szBuffer, "%lf", value );
```

```
    setValue( szBuffer );
}
XMLElement( const XMLElement& aCopy )
{
    Copy( aCopy );
}
virtual ~XMLElement()
{
}
XMLElement operator=( const XMLElement&
  aCopy )
{
    Copy( aCopy );
    return *this;
}
// Accessors
void setName( const char *name )
{
    _name = name;
}
void setValue( const char *value )
{
    _value = value;
}
string getName( void ) const
{
    return _name;
}
string getValue( void ) const
{
    return _value;
}
// Sub-element maintenance.
void addSubElement( const XMLElement&
anElement )
```

```
        {
            _subElements.insert( _subElements.
            end(), anElement );
        }
        int numSubElements( void )
        {
            return _subElements.size();
        }
        XMLElement& getSubElement( int index )
        {
            if ( index < 0 || index >=
            numSubElements() )
            throw "getSubElement: index out
            of range";
            return _subElements[ index ];
        }
};
// Class to manage the output of XML data.
class XMLWriter
{
private:
    ofstream _out;
public:
    XMLWriter( void )
    {
    }
    XMLWriter( const char *fileName )
    {
        _out.open(fileName);
        if ( _out.fail() == false )
        {
            _out << "<xml>" << endl;
        }
    }
    XMLWriter( const XMLWriter& aCopy )
```

```
{
}
virtual ~XMLWriter()
{
    if ( _out.fail() == false )
    {
        _out << "</xml>" << endl;
    }
    _out.close();
}
void setFileName( const char *fileName )
{
    _out.open(fileName);
    if ( _out.fail() == false )
    {
    _out << "<xml>" << endl;
    }
}
virtual bool Write( XMLElement& aRoot )
{
    if ( _out.fail() )
        return false;
    // First, process the element.
    _out << "<" <<
aRoot.getName().c_str() << ">" << endl;
    // If there is a value, output it.
    if ( aRoot.getValue().length()
        != 0 )
            _out << aRoot.getValue().c_str()
            << endl;
    // Now, process all sub-elements.
    for ( int i=0;
i<aRoot.numSubElements(); ++i )
            Write( aRoot.getSubElement(i) );
    // Finally, close the element.
```

```
            _out << "</" <<
        aRoot.getName().c_str() << ">" << endl;
        }
};
```

Listing này minh hoạ những điểm cơ bản về chức năng ghi XML. Mỗi phần tử của một đối tượng XML sẽ được lưu trữ trong một đối tượng XMLElement. Sau đó writer (lớp XMLWriter) xử lý từng phần tử này để xuất chúng theo định dạng XML hợp lệ.

3. Lưu file mã nguồn và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng bằng trình biên dịch ưa thích, trên hệ điều hành ưa thích để xác nhận rằng bạn đã không phạm các lỗi.

Test XMLWriter

Sau khi tạo lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn như thế nào.

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ các kiểu phần tử dữ liệu khác nhau và sẽ minh hoạ lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn cho chương trình thử nghiệm.

   Trong ví dụ này chương trình thử nghiệm được đặt tên là ch43.cpp.

2. Gõ nhập mã từ Listing 43.2 vào file.

Listing 43.2: Mã thử nghiệm XMLWriter

```
class XmlTest
{
private:
    int iVal;
    string sVal;
    double dVal;
public:
    XmlTest()
    {
        iVal = 100;
        sVal = "Test";
```

```
        dVal = 123.45;
}
~XmlTest()
{
}
XMLElement getXML(void)
{
    XMLElement e("XmlTest", "");
    e.addSubElement(
    XMLElement("iVal", iVal) );
    e.addSubElement(
    XMLElement("sVal", sVal.
    c_str()) );
    e.addSubElement(
    XMLElement("dVal", dVal) );
    return e;
}
};
class XmlSuperClass                                    → 1
{
    XmlTest xt;                                        → 2
    int count;
public:
    XmlSuperClass()
    {
        count = 1;
    }
    ~XmlSuperClass()
{
}
XMLElement getXML()
{
        // First, do ourselves
        XMLElement e("XmlSuperClass", "");
        e.addSubElement(
```

```
            XMLElement("count", count) );
            // Now the sub-object
            e.addSubElement( xt.getXML() );
        return e;
    }
};
void TestWriter1(void)
{
        XMLElement ele1("Sub-Element1", "123");
        XMLElement ele2("Sub-Element2", "234");
        XMLElement subele1("Sub-Sub-Element1",
        "345");
        XMLElement subele2("Sub-Sub-Element2",
        "456");
        XMLElement root("Root", "");
        ele1.addSubElement( subele1 );
        ele2.addSubElement( subele2 );
        root.addSubElement( ele1 );
        root.addSubElement( ele2 );
        XMLWriter writer("test.xml");
        writer.Write( root );
}
```

3. Lưu file mã nguồn và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, việc chạy ứng dụng sẽ dẫn đến việc tạo ra hai file test.xml và test 2.xml. Nếu bạn xem nội dung của các file này, bạn sẽ thấy kết quả sau đây:

```
test.xml:
$ cat test.xml
<xml>
<Root>
<Sub-Element1>
123
<Sub-Sub-Element1>
```

```
345
</Sub-Sub-Element1>
</Sub-Element1>
<Sub-Element2>
234
<Sub-Sub-Element2>
456
</Sub-Sub-Element2>
</Sub-Element2>
</Root>
</xml>
test2.xml:
$ cat test2.xml
<xml>
<XmlSuperClass>                                    → 3
<count>
1
</count>
<XmlTest>                                          → 4
<iVal>                                             → 5
100
</iVal>
<sVal>
Test
</sVal>
<dVal>
123.450000
</dVal>
</XmlTest>
</XmlSuperClass>
</xml>
```

Nếu chúng ta xem hệ thống phân cấp lớp được thể hiện trong mã nguồn ứng dụng, chúng ta thấy rằng lớp chính, XmlSuperClass (được minh hoạ tại → 1), chứa các phần tử dữ liệu chuẩn (count, một số nguyên) và các đối tượng nhúng XmlTest, được minh hoạ tại → 2).

Trong kết quả XML, chúng ta thấy những phần tử này tại các dòng được đánh dấu → **3** và → **4**. Chú ý cách lớp nhúng chứa các phần tử riêng của nó (được minh hoạ tại → **5** trong danh sách kết quả) vốn là con của các lớp XmlTest và XmlSuperClass.

Mã cho thấy cả hai làm việc tốt - trường hợp đơn giản là sử dụng các lớp XmlElement và XmlWriter và trường hợp nhúng là xuất toàn bộ một lớp C++ với một đối tượng C++ nhúng.

Kỹ thuật 44: Loại bỏ khoảng trắng ra khỏi dữ liệu nhập

Tiết kiệm thời gian bằng cách:

- Loại bỏ các khoảng trắng đứng trước và đứng sau ra khỏi dữ liệu nhập
- Trả chuỗi được chỉnh sửa trở về ứng dụng
- Test mã

Mặc dù dường như không phải là một vấn đề quan trọng, nhưng việc xử lý khoảng trắng trong dữ liệu nhập từ các file hoặc console có thể là một phiền toái lớn đối với các nhà lập trình C++. Sau cùng khoảng trắng không rỗng; nó phải được xem xét. Ví dụ, khi bạn muốn lưu trữ một tên user trong cơ sở dữ liệu, bạn có thật sự muốn lưu trữ bất kỳ khoảng trống đứng trước và đứng sau, tab hoặc ký tự không in khác hay không? Nếu bạn làm như vậy, những người dùng sẽ phải nhớ gõ nhập lại những khoảng trống đó bất cứ khi nào họ đăng nhập (log in) vào ứng dụng của bạn. Trong khi đây có thể là một điều kiện an ninh hữu dụng, nhưng dường như bất kỳ người nào không thể nhớ thêm các khoảng trống đứng trước hoặc đứng sau vào một tên user hoặc password trong một ứng dụng.

Vì lý do này, nếu bạn cho mã khả năng loại bỏ các khoảng trắng đứng trước và đứng sau ra khỏi một chuỗi nào đó và trả chuỗi đó trả về ứng dụng gọi, bạn sẽ tiết kiệm được nhiều thời gian và tránh được nhiều phiền phức. Kỹ thuật này xem xét việc tạo chính xác khả năng đó. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch44.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 44.1 vào file.

Listing 44.1: Mã loại bỏ khoảng trắng

```
#include <string>
#include <ctype.h>
// Nobody wants to have to type std:: for
// all of the STL functions.
using namespace std;
string strip_leading( const string& sIn )
{
    string sOut;
    // Skip over all leading spaces.
    unsigned int nPos = 0;
    while ( nPos < sIn.length() )
    {
        if ( !isspace(sIn[nPos]) )
            break;
        nPos ++;
    }
    // Now we have the starting position of
    // the "real" string. Copy to the end...
    while ( nPos < sIn.length() )
    {
        sOut += sIn[nPos];
        nPos ++;
    }
    // ...and give back the new string,
    // without modifying the input string.
    return sOut;
}
string strip_trailing( const string& sIn )
{
    string sOut;                                    → 1
    // Skip over all trailing spaces.
    int nPos = sIn.length()-1;
    while ( nPos >= 0 )
```

```
        {
            if ( !isspace(sIn[nPos]) )
                break;
            nPos —;
        }
        // Now we have the ending position of
        // the "real" string. Copy from the
        // beginning to that position...
        for ( int i=0; i<=nPos; ++i )
            sOut += sIn[i];
        // ...and give back the new string,
        // without modifying the input string.
        return sOut;
}
int main(int argc, char **argv )
{
        if ( argc > 2 )
        {
            printf("Removing Leading Spaces\n");
            for ( int i = 1; i < argc; ++i )
            {
                printf("Input String: [%s]\n",
                argv[i] );
                string s = argv[i];
                s = strip_leading( s );
                printf("Result String: [%s]\n",
                    s.c_str() );
        }
        printf("Removing Trailing
        Spaces\n");
        for ( int i = 1; i < argc; ++i )
        {
            printf("Input String: [%s]\n",
            argv[i] );
            string s = argv[i];
```

```
            s = strip_trailing( s );
            printf("Result String: [%s]\n",
            s.c_str() );
        }
        printf("Removing both leading and
        trailing\n");
        for ( int i = 1; i < argc; ++i )
        {
            printf("Input String: [%s]\n",
            argv[i] );
            string s = argv[i];
            s = strip_trailing( strip_
            leading(s) );
            printf("Result String: [%s]\n",
            s.c_str() );
        }
    }
    else
    {
            bool bDone = false;
            while ( !bDone )
            {
                char szBuffer[ 80 ];
                printf("Enter string to fix: ");
                gets(szBuffer);
                printf("Input string: [%s]\n",
                    szBuffer );
                // Strip the trailing carriage
                    return.
                if ( strlen(szBuffer) )
                    szBuffer[strlen(szBuffer)-1]
                    = 0;
                if ( !strlen(szBuffer) )
                    bDone = true;
                else
```

```
            {
                string s = szBuffer;
                s = strip_leading( s );
                printf("After removing
                leading: %s\n", s.c_str() );
                s = strip_trailing( s );
                printf("After removing
                trailing: %s\n", s.c_str()
                );
            }
        }
    }
    return 0;
}
```

Bạn tìm thấy ký tự khoảng trắng cuối cùng và xén chuỗi tại điểm đó. Mặt khác loại bỏ khoảng trắng đứng đầu là một vấn đề phức tạp hơn. Như bạn có thể thấy tại dòng được đánh dấu là → **1** trong listing nguồn, bạn phải tạo một chuỗi riêng biệt để sử dụng cho giá trị trả về của hàm strip_leading. Tiếp theo, hàm này được tạo bằng cách tìm ký tự không trống đầu tiên trong chuỗi đầu vào và sao chép mọi thứ từ điểm đó đến cuối chuỗi vào chuỗi đầu ra. Sau đó, chuỗi đầu ra được trả về ứng dụng gọi không có khoảng trắng đứng trước.

3. Lưu file mã nguồn và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ và bạn chạy chương trình với những tuỳ chọn dòng lệnh sau đây, bạn sẽ thấy kết quả sau đây trong cửa sổ console:

```
$ ./a " this is a test " " hello "
" goodbye"
Removing Leading Spaces
Input String: [ this is a test ] → 2
Result String: [this is a test ] → 3
Input String: [ hello ]
Result String: [hello ]
Input String: [ goodbye]
```

```
Result String: [goodbye]
Removing Trailing Spaces
Input String: [ this is a test ]
Result String: [ this is a test] → 4
Input String: [ hello ]
Result String: [ hello]
Input String: [ goodbye]
Result String: [ goodbye]
Removing both leading and trailing
Input String: [ this is a test ]
Result String: [this is a test] → 5
Input String: [ hello ]
Result String: [hello]
Input String: [ goodbye]
Result String: [goodbye]
```

Trong kết quả, chúng ta thấy mỗi chuỗi được nhập vào hệ thống, sau đó các ký tự khoảng trắng khác nhau đứng trước và đứng sau chuỗi bị loại bỏ. Trong mỗi trường hợp, chuỗi được xuất sang người dùng để xem nó được chỉnh sửa như thế nào. Ví dụ, nếu chúng ta xem dòng nhập liệu tại → **2**, chúng ta thấy nó chứa các khoảng trống đứng trước và đứng sau. Khi hàm strip_leading được áp dụng, chúng ta có kết quả được minh hoạ tại → **3** vốn là cùng một chuỗi không có các khoảng trắng đứng trước. Khi hàm strip_trailing được áp dụng, chúng ta có được kết quả được minh hoạ tại → **4** vốn là cùng một chuỗi không có các khoảng trống đứng sau. Sau cùng, chúng ta áp dụng cả hai hàm cùng một lúc và nhận được kết quả được minh hoạ tại → **5** vốn không có các khoảng trắng đứng trước cũng không có các khoảng trắng đứng sau.

Bạn cũng có thể test ứng dụng bằng cách gõ nhập dữ liệu từ dòng nhắc bằng cách chạy ứng dụng không có các đối số đầu vào. Sau đây là một mẫu về diện mạo của ứng dụng test có dạng đó:

```
$ ./a.exe
Enter string to fix: this is a test
Input string: [ this is a test ]
After removing leading: this is a test
After removing trailing: this is a test
```

```
Enter string to fix:
Input string: []
```

Như bạn có thể thấy, dữ liệu nhập từ dòng lệnh hoặc từ các mục nhập người dùng (cho dù từ bàn phím hoặc một file) có thể chứa khoảng trắng. Khoảng trắng này phải được loại bỏ để xem các chuỗi "thực" trong nhiều trường hợp và những hàm này sẽ tiết kiệm cho bạn nhiều thời gian bằng việc thực hiện điều này một cách tự động.

Kỹ thuật 45: Tạo một file cấu hình

Tiết kiệm thời gian bằng cách:

- Tạo một giao diện chuẩn với một file cấu hình
- Tạo một lớp file cấu hình
- Tạo file đầu vào thử nghiệm
- Test lớp file cấu hình

Các file cấu hình là một phần cơ bản của bất kỳ ứng dụng nào cần được mang chuyển qua các hệ điều hành khác nhau. Do những điểm khác biệt trong các định dạng nhị phân (binary) và những mối quan tâm "endian" (vị trí của byte có nghĩa nhất), các file cấu hình thường được lưu trữ theo định dạng text. Điều này một phần còn phải bàn vì nó đòi hỏi ứng dụng có khả năng tải, phân tích cú pháp và làm việc với các mục nhập trong một file cấu hình trong khi hiểu dữ liệu được lưu trữ ở đó. Bởi vì định dạng là text, bạn phải quan tâm đến việc người dùng chỉnh sửa các file text, thay đổi chúng đến nỗi chúng không còn có một định dạng hợp lệ nữa và những điều tương tự. Do đó, sẽ hợp lý nếu có một giao diện chuẩn cho một file cấu hình và một định dạng chuẩn để sử dụng các file cấu hình dựa vào text. Điều này sẽ cho phép bạn sử dụng một định dạng chuẩn cho tất cả ứng dụng, tiết kiệm cho bạn thời gian và công sức.

Kỹ thuật này hướng dẫn bạn cách phát triển một phương thức để lưu trữ dữ liệu bằng cách đơn giản nhất có thể có trong một file cấu hình (dựa vào text) trong khi vẫn cho phép người dùng lưu trữ các kiểu dữ liệu mà họ cần. Một mục nhập điển hình trong một trong các file cấu hình tương tự như sau:

```
# This is a comment
Value = " This is a test"
```

Dòng đầu tiên của mục nhập là một trường comment - được bỏ qua bởi parser - vốn cho bất kỳ bộ đọc file cấu hình biết tại sao dữ liệu riêng biệt được lưu trữ trong key này (và nó có thể được hiểu hoặc

được chỉnh sửa như thế nào). Dòng thứ hai là chính giá trị gồm hai phần:

- Từ khoá mà chúng ta định nghĩa, trong trường hợp này là Value.
- Chuỗi hoàn chỉnh được gán vào giá trị này với các khoảng trống nhúng và khoảng trống có thể đứng đầu. Trong trường hợp này chuỗi giá trị là "This is a test". Chú ý rằng khi được đọc vào, chuỗi sẽ chứa các khoảng trống đứng đầu như người dùng mong đợi. Chú ý rằng lý do duy nhất chúng ta lưu trữ những khoảng trống này là chúng được chứa trong các dấu ngoặc kép biểu thị người dùng muốn giữ lại chúng. Nếu các khoảng trống đơn giản nằm trên các mép đứng đầu và đứng sau của các chuỗi trong mục nhập không có các dấu ngoặc kép, chúng ta sẽ loại bỏ chúng.

Tạo lớp file cấu hình

Lớp file cấu hình đóng gói việc đọc, phân tích cú pháp và lưu trữ dữ liệu trong file cấu hình dựa vào text. Các bước sau đây hướng dẫn bạn tạo một lớp độc lập vốn có thể đơn giản được di chuyển từ ứng dụng này đến ứng dụng khác, cho phép bạn tiết kiệm thời gian và có một giao diện nhất quán.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa định nghĩa cho lớp file cấu hình.

   Trong ví dụ này, file được đặt tên là ConfigurationFile.h mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn.

2. Gõ nhập mã từ Listing 45.1 vào file.

Listing 45.1: File header của file cấu hình

```
#ifndef _CONFIGURATIONFILE_H_
#define _CONFIGURATIONFILE_H_
#include <string>
#include <vector>
#include <fstream>
#include <map>
#include <list>
using namespace std;
class ConfigurationFile
{
public:
    ConfigurationFile(const char
    *strFileName);
```

```
        virtual ~ConfigurationFile(void);
        bool read(void);
        bool hasValue( const char *key );
        string getValue( const char *key );
        void setValue( const char *key, const
        char *value );
    protected:
        virtual void get_token_and_value();
        virtual char
        eat_white_and_comments(bool traverse_
        newlines=true);
        virtual bool
        advance_to_equal_sign_on_line();
        virtual void makeLower
        (string &instring);
    protected:
        fstream m_in;
        string m_token;
        string m_value;
        string m_sConfigFile;
        typedef pair <string, string>
        String_Pair;
        map<string, string> m_ConfigEntries;
    };
    #endif
```

File này chứa định nghĩa của lớp; nó không chứa mã để xử lý dữ liệu. File header có chức năng như giao diện cho những ứng dụng khác để sử dụng lớp như chúng ta sẽ thấy. Tốt nhất nên tách biệt mã thực thi thực sự ra khỏi định nghĩa, vì điều này giúp nhấn mạnh khái niệm đóng gói (encapsulation) của C++.

3. Lưu file mã nguồn.

4. Trong code editor mà bạn chọn, tạo một file mới để chứa định nghĩa cho lớp file cấu hình.

   Trong ví dụ này, file được đặt tên là ConfigurationFile.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

5. Gõ nhập mã từ Listing 45.2 vào file.

Listing 45.2: Mã nguồn file cấu hình

```
#include "ConfigurationFile.h"
#include <errno.h>
#include <algorithm>
#include <sstream>
#include <iostream>
#include <string>
template <class T>
bool from_string(T &t,
                 const std::string &s,
                 std::ios_base &
      (*f)(std::ios_base&))
{
      std::istringstream iss(s);
      return !(iss>>f>>t).fail();
}
class StringUtil                                    → 1
{
public:
      StringUtil() {}
      ~StringUtil() {}
      // Find the given string in the source
         string and replace it with the
      // "replace" string, everywhere
         instances of that string exist.
      static void findandreplace( string&
      source, const string& find, const string&
         replace )
{
      size_t j;
      for (;(j = source.find( find ))
      != string::npos;)
      {
         source.replace( j,
```

```
            find.length(), replace );
        }
    }
    // The following function returns a
        string with all-uppercase characters.
    static string makeUpper( const string&
        instring)
    {
        string temp=instring;
        transform( temp.begin(), temp.end(),
            temp.begin(), ::toupper );
        return temp;
    }
    // The following function returns a
    string with all-lowercase characters.
    static string makeLower( const string&
    instring)
    {
        string temp;
        transform( temp.begin(), temp.end(),
            temp.begin(), ::tolower );
        return temp;
    }
    static bool contains( const string&
    source, const char *find )
    {
        return ( 0!=strstr(source.
            c_str(),find) );
    }
    static string pad( const string&
    instring, char padchar, int length )
    {
    string outstring = instring;
    for ( int i=(int)outstring.length();
    i<length; ++i )
```

```
            outstring += padchar;
        return outstring;
}
// Trim the given characters from the
    beginning and end of a string.
// the default is to trim whitespace.
    If the string is empty or contains
// only the trim characters, an empty
    string is returned.
static string trim( const string
    &instring,
                        const string
&trimstring=string(" \t\n"))
{
    if (trimstring.size()==0) return
      instring;
    string temp="";
    string::size_type begpos=instring.find_first_not_of (trimstring);
    if (begpos==string::npos)
    {
          return temp;
    }
    else
    {
        string::size_type endpos=instring.find_last_not_of (trimstring);
        temp=instring.substr(begpos,
endpos-begpos+1);
    }
    return temp;
}
// Convert the string to an int. Note that a string exception is thrown if
// it is invalid.
static int toInt(const string & myInString)
{
    int i=0;
```

```
        string inString = trim(myInString);
        if( !from_string<int>(i, inString, std::dec) )
        {
            string exceptionText = "StringUtils::toInt() - Not an integer: " + inString;
            throw exceptionText;
        }
        // Time to run some more checks.
        for (unsigned int j=0; j < inString.length(); j++)
        {
            if ( !isNumeric(inString[j]) )
            {
                if (j==0 && inString[j] =='-')
                {
                    continue;
                }
                else
                {
                    string exceptionText = "StringUtils::toInt() - Not an integer: " +
                        inString;
                    throw exceptionText;
                }
        }
}
        return (i);
        }
        // Convert the string to an int. Note:
        A string exception is thrown if
        // it is invalid.
        static float toFloat(const string & myInString)
        {
            float f=0;
        string inString = trim(myInString);
        if( !from_string<float>(f, inString, std::dec) )
        {
            string exceptionText = "StringUtils::toFloat() - Not a float: " + inString;
```

```
            throw exceptionText;
        }
        // Now it runs some more checks.
        int dec_count=0;
        for (unsigned int j=0; j < inString.length(); j++)
        {
            if ( !isNumeric(inString[j]) )
            {
                if (j==0 && inString[j] =='-')
                {
                    continue;
                }
                else if (inString[j]=='.')
                {
                    dec_count++;
                    if (dec_count > 1)
                    {
                        string exceptionText = "StringUtils::toFloat() - Not a float: " +
                            inString;
                        throw exceptionText;
                    }
                    continue;
                }
                else
                {
                string exceptionText = "StringUtils::toFloat() - Not a float: " + inString;
                    throw exceptionText;
                }
            }
        }
        return (f);
}
// Returns true if the character is numeric.
static bool isNumeric(char c)
{
```

```
        return ('0' <= c && c <= '9');
}
// Replace environment variables in the string with their values.
// Note: environment variables must be of the form ${ENVVAR}.
static string substituteEnvVar( const string &myInString )
{
    string outString="";
    char variable[512];
    const char *s = myInString.c_str();
    while(*s!=0)
    {
       if (*s=='$' && *(s+1)=='{')
       {
          // When you've found beginning of variable, find the end.
          strcpy(variable,s+2);
          char *end = strchr (variable,'}');
          if (end)
          {
             *end='\0';
             char *cp = (char *)getenv(variable);
             if (cp)
             outString += (char *) getenv(variable);
             s = strchr(s,'}');
          }
          else
          {
             outString += *s;
          }
          }
          else
          {
             outString += *s;
          }
          s++;
          }
```

```
        return outString;
    }
};
ConfigurationFile::ConfigurationFile( const char *strConfigFile )        → 2
{
    if ( strConfigFile )
        m_sConfigFile = strConfigFile;
}
ConfigurationFile::~ConfigurationFile()
{
}
bool ConfigurationFile::read()                                           → 3
{
    m_in.open(m_sConfigFile.
    c_str(),ios::in);
    if (m_in.fail())
    {
        return false;
    }
    while (!m_in.eof())
    {
    //-----------------------------------------------
    // Get a token and value.
    // This gives values to member vars: m_token and m_value.
    //-----------------------------------------------
    get_token_and_value();
    if ( m_token.length() )
        m_ConfigEntries.insert( String_
Pair(m_token, m_value) );
}
    m_in.close();
    return true;
}
void ConfigurationFile::get_token_and_
    value(void)
```

```
{
    char token[1024];
    char ch;
    bool found_equal=false;
    int i=0;
    eat_white_and_comments();
    while(!(m_in.get(ch)).fail())
    {
        if ((ch != '\t'))
        {
        if ( (ch == '=') || (ch == ' ') || (ch == '\n') || (ch == '\r') ||
            (ch == '\t'))
        {
            if (ch == '=')found_equal=true;
            break;
        }
        token[i++]=ch;
    }
}
if (i==0)
{
    // It didn't find a token, in this case.
    m_token="";
    m_value="";
    return;
}
// Null-terminate the token that was found.
token[i++]='\0';
m_token = token;
makeLower(m_token);
// Advance to the equal sign, if need be.
if (!found_equal)
{
    if (!advance_to_equal_sign_on_line())
    {
```

```
            // The token had no value.
            m_token="";
            m_value="";
            return;
        }
    }
    // Get the token's value.
    i=0;
    char c = eat_white_and_comments(false);
    if ( c != '\n' )
    {
        i=0;
        while(!(m_in.get(ch)).fail())
        {
            if ((ch == '\t') || (ch == '\r') || (ch == '\n') || (ch == '#') )
            {
                while (ch!='\n')
                {
                    if (m_in.get(ch).fail()) break;
                }
                    break;
                }
                else
                {
                    token[i++]=ch;
                }
            }
        }
        if (i==0)
        {
            // This token had no value.
            m_value="";
        }
        else
        {
```

```
            token[i++]='\0';
            m_value=token;
            // Remove leading/trailing spaces.
            m_value = StringUtil::trim(m_value);
            // Strip leading and trailing quotes, if there are any.
            if ( m_value[0] == '"' )
                m_value = m_value.substr( 1 );
            if ( m_value[ m_value.length() -1 ] == '"' )
                m_value = m_value.substr( 0, m_value.length()-1 );
        }
    }
    bool
    ConfigurationFile::advance_to_equal_sign_on_line()
    {
        char ch;
        bool found_equal=false;
        while ( !(m_in.get(ch)).fail() )
        {
            if ((ch=='\r')||(ch=='\n')) break;
            if (ch == '=')
            {
                found_equal=true;
                break;
            }
        }
        return found_equal;
    }
    char
    ConfigurationFile::eat_white_and_comments
        (bool traverse_newlines)
    {
        char ch;
        bool in_comment;
        in_comment = false;
        while (!(m_in.get(ch)).fail())
```

```
    if (ch == '#')
        in_comment = true;
    else if (ch == '\n')
    {
        in_comment = false;
        if (!traverse_newlines)
        {
            return(ch); // Stop eating.
        }
    }
    else if ((!in_comment) && (ch != ' ') &&
    (ch != '\t') && (ch != '\r'))
        {
            m_in.putback(ch);
            return 0;
        }
    return 0;
}
void ConfigurationFile::makeLower
    (string &instring)
{
    for(unsigned i=0; i < instring.size();
    i++)
    {
        instring[i] = tolower(instring[i]);
    }
}
bool ConfigurationFile::hasValue( const char
    *key )                                          → 4
{
    bool bRet = false;
    std::string sKey = key;
    makeLower( sKey );
    if ( m_ConfigEntries.find( sKey.c_str()
    ) != m_ConfigEntries.end() )
```

```
            {
                bRet = true;
            }
            return bRet;
        }
    string ConfigurationFile::getValue( const
        char *key )
    {
        std::string sKey = key;
        makeLower( sKey );
        if ( m_ConfigEntries.find( sKey.
        c_str() ) != m_ConfigEntries.end() )
        {
            std::map<string, string>::iterator
                iter;
            iter =
        m_ConfigEntries.find(sKey.c_str());
            return (*iter).second;
        }
        return "";
    }
    void ConfigurationFile::setValue( const char
        *key, const char *value )
    {
        std::string sKey = key;
        makeLower( sKey );
        m_ConfigEntries[sKey] = value;
    }
```

Mã nguồn ở trên phân chia thành ba phần chung. Đầu tiên chúng ta tách biệt thường trình tiện ích vốn làm việc với các chuỗi và ký tự và đặt chúng trong lớp tiện ích StringUtil (được minh hoạ tại dòng được đánh dấu là → **1**. Tiếp theo, chúng ta có lớp file cấu hình thật sự được minh hoạ tại dòng được đánh dấu là → **2**. Lớp này quản lý việc lưu trữ và xử lý file. Việc xử lý được thực hiện trong hàm read được minh hoạ tại → **3** và các hàm lưu trữ bắt đầu với dòng được đánh dấu là → **4**. Như bạn có thể thấy, thường trình đơn giản đọc một dòng từ

file đầu vào và tách biệt nó thành hai phần, được phân chia bằng một dấu bằng. Các Comment (chú giải) vốn là các dòng trống hoặc được đánh dấu bằng một dấu pound ('#') được bỏ qua. Mọi thứ nằm bên trái dấu bằng được xem là "thẻ" trong khi mọi thứ nằm bên phải dấu bằng được xem là "giá trị". Các cặp thẻ (tag) và giá trị tạo nên dữ liệu cấu hình. Các thường trình truy tìm làm việc bằng cách cho phép người dùng xem một thẻ nào đó có được định nghĩa hay không và nếu có thì truy tìm giá trị của nó.

6. Lưu file nguồn trong bộ soạn thảo mã nguồn.

Thiết lập file test

Sau khi tạo bất kỳ lớp, bạn nên tạo một driver thử nghiệm để bảo đảm không chỉ mã của bạn chính xác mà còn hướng dẫn mọi người cách sử dụng mã của bạn như thế nào.

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ lớp sẽ được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch45.cpp.

2. Gõ nhập mà từ Listing 45.3 vào file.

Listing 45.3: Mã thử nghiệm file cấu hình.

```
#include <stdio.h>
#include "ConfigurationFile.h"
int main( int argc, char **argv )
{
    if ( argc < 3 )
    {
       printf("Usage: ch5_7 config-filename
    arg1 [arg2 .. ]\n");
       printf("Where: config-filename
    is the name of the configuration
    file\n");
       printf(" arg1 .. argn
    are the values to print out\n");
       return -1;
    }
```

```
    ConfigurationFile cf(argv[1]);
    if ( cf.read() == false )
    {
        printf("Unable to read configuration
    file\n");
        return -2;
    }
    for ( int i=2; i<argc; ++i )
    {
        if ( !cf.hasValue( argv[i] ) )
        {
            printf("Value %s NOT found in
    configuration file\n", argv[i] );
        }
        else
        {
            string s = cf.getValue
    ( argv[i] );
            printf("Key %s = [%s]\n",
    argv[i], s.c_str() );
        }
    }
    return 0;
}
```

3. Lưu file mã nguồn trong code editor.
4. Trong code editor mà bạn chọn, tạo một file test mới để chứa file test cấu hình thật sự cho chương trình thử nghiệm.

   Trong ví dụ này, file dữ liệu nhập test được đặt tên là input.cfg.
5. Gõ nhập text sau đây vào file:

```
Color=Blue                          → 5
Name=Matt                           → 6
Address=" "
City="Denver, CO"                   → 7
ZipCode=80232
#This is a comment
```

Test lớp file cấu hình

Bây giờ mọi thứ được thiết lập, các bước sau đây hướng dẫn bạn cách lắp ghép lại tất cả và chạy thử.

1. Biên dịch và chạy file mã nguồn (được gọi là ch45_7.cpp) cùng với file lớp cấu hình (được gọi là ConfigurationFile.cpp) trong trình biên dịch ưa thích, trên hệ điều hành ưa thích.
2. Chạy chương trình.

Nếu bạn đã làm tốt mọi thứ, bạn thấy kết quả sau đây trong cửa sổ console:

```
$ ./a.exe input.cfg Color Name City State
Key Color = [Blue]
Key Name = [Matt]
Key City = [Denver, CO]
Value State NOT found in configuration
file                                          → 8
```

Xem file đầu vào, bạn có thể thấy các giá trị của Color, Name và City đều là các mục nhập (nằm ở phía bên trái dấu bằng). Đối với các key đó, các giá trị là Blue, Matt và Denver, CO. Các mục này được minh hoạ tại các dòng được đánh dấu → **5**, → **6** và → **7**. Driver thử nghiệm đơn giản đọc file cấu hình bằng cách sử dụng lớp file cấu hình và sau đó hiển thị các cặp key (khoá) và giá trị khác nhau. Sau đó, driver thử nghiệm thi hành toàn bộ chức năng của mã truy tìm bằng cách tìm một giá trị (State) vốn không nằm trong file cấu hình và mã hiển thị một lỗi như được minh hoạ bởi dòng được đánh dấu là → **8**. Từ kết quả này và mã driver thử nghiệm, bạn có thể thấy chính xác lớp file cấu hình được sử dụng như thế nào, làm cho nó trở thành tài liệu hoàn hảo cho nhà phát triển. Từ listing, bạn cũng có thể thấy kết quả là những gì được mong đợi. Nói chung, nó cho thấy bạn có thể tiết kiệm thời gian và công sức như thế nào bằng việc sử dụng một định dạng cấu hình được chuẩn hoá và sử dụng lớp này để đọc nó.

Phần VII
Sử dụng chức năng cài sẵn

Kỹ thuật 46: Tạo một lớp Internationalization

Tiết kiệm thời gian bằng cách:

- Tìm hiểu sự quốc tế hoá (internationalization)
- Tạo các file ngôn ngữ
- Đọc một file quốc tế
- Tạo một bộ đọc chuỗi (string reader)
- Test mã

Trước đây nếu lập trình ở Mỹ, người ta tuỳ biến các ứng dụng chỉ dành cho những người Mỹ nói tiếng Anh. Mối quan tâm chính của bạn là mã thực thi những thuật toán nằm trong ứng dụng của bạn; nếu bạn có một thông báo lỗi để hiển thị, bạn viết một thông báo diễn đạt lỗi đó một cách mơ hồ và người dùng chỉ phải xử lý nó - bất kể người dùng này nói ngôn ngữ nào hoặc thông báo lỗi gây bối rối như thế nào. Thật may thay, những ngày mã bị thách thức bởi khả năng sử dụng đó đã qua rồi. Bây giờ các chuyên gia tạo các thông báo và bộ chỉ báo cho những ứng dụng để người dùng có thể hiểu rõ nhất những gì đang xảy ra. Bản thân các thông báo lỗi thường được tuỳ biến theo những nhóm khách hàng riêng biệt. Tuy nhiên, quan trọng nhất mã của chúng ta không còn chỉ hướng đến những người Mỹ nói tiếng Anh nữa. Các ứng dụng được phân phối khắp thế giới và cần làm việc bằng bất kỳ ngôn ngữ với bất kỳ bộ bảng chữ cái. Khả năng hiển thị các thông báo bằng bất kỳ ngôn ngữ được gọi là sự quốc tế hoá (internationalization).

Bạn có thể đơn giản thêm một tính năng quốc tế hoá vào chương trình của bạn - bạn phải thiết kế tính năng đó ngay từ đầu. Bạn không thể chỉ biên dịch nhanh các thông báo; bạn phải biết trước thông báo sẽ

thể hiện điều gì. Vì lý do này, tạo một hệ thống hỗ trợ sự quốc tế hoá là điều quan trọng.

Tiến trình quốc tế hoá thực sự gồm ba phần:

1. Nhận dạng tất cả text cần được hiển thị bằng những ngôn ngữ khác nhau. Đặt text này vào một file cùng với các định danh được sử dụng trong ứng dụng để hiển thị text.
2. Chuyển đổi text thành một định dạng vốn có thể được phân phối dễ dàng với ứng dụng.
3. Cung cấp một phương thức để truy cập tất cả nội dung này.

Chỉ bằng việc thực hiện tất cả điều này, chúng ta có thể tiết kiệm thời gian khi tạo các ứng dụng trong nhiều ngôn ngữ. Nếu bạn viết các ứng dụng với bất kỳ loại sức hấp dẫn khu vực quốc tế, cuối cùng bạn phải quốc tế hoá chúng. Bằng cách cài sẵn trước sự hỗ trợ này và cho ứng dụng khả năng tải các chuỗi trực kiện đó từ các nguồn bên ngoài, bạn không chỉ tiết kiệm thời gian sau này mà còn có thể tiết kiệm lượng không gian rất lớn trong bộ nhớ ứng dụng. Phương pháp này cũng cho phép bạn tuỳ biến các thông báo lỗi và hiển thị các dòng nhắc hướng đến các nhóm tuổi và khu vực khác nhau. Đây là thủ tục mà chúng ta sẽ sử dụng trong kỹ thuật này để minh hoạ cách tiết kiệm thời gian và công sức bằng việc tạo ra một cách quốc tế hoá các ứng dụng.

Xây dựng các file ngôn ngữ

Trước khi có thể hiển thị text, bạn cần có khả năng xây dựng các file chứa dữ liệu ngôn ngữ. Bạn không cần lưu trữ tất cả ngôn ngữ có thể có trong ứng dụng, bởi vì điều đó đòi hỏi các yêu cầu bộ nhớ. Do đó, chúng ta lưu trữ các ngôn ngữ theo định dạng text nén bằng cách ép ra các ký tự trả về và khoảng trống giữa các mục dữ liệu. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch46.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã từ Listing 46.1 vào file.

Listing 46.1: Lớp StringEntry

```
#include <stdio.h>
#include <string>
#include <vector>
```

```
#include <iostream>
#include <fstream>
using namespace std;
#define VERSION_STRING "Version 1.0.0"
class StringEntry
{
private:
unsigned long _id;
string _strEntry;
    unsigned long _offset;
    unsigned long _length;
protected:
    void Init()
    {
        setID ( 0 );
        setString( "" );
        setOffset( 0 );
        setLength( 0 );
    }
    void Copy( const StringEntry& aCopy )
    {
        setID ( aCopy.ID() );
        setString( aCopy.String() );
        setOffset( aCopy.Offset() );
        setLength ( aCopy.Length() );
    }
public:
    StringEntry(void)
    {
        Init();
    }
    StringEntry( unsigned long id, const char *strIn )
    {
        Init();
        setID( id );
        setString( strIn );
        // For now, assign the length to just be the length
        // of the string.
```

```
    setLength( strlen(strIn) );
}
StringEntry( const StringEntry& aCopy )
{
    Copy( aCopy );
}
StringEntry operator=( const StringEntry& aCopy )
{
    Copy( aCopy );
}
unsigned long ID() const
{
    return _id;
}
string String() const
{
    return _strEntry;
}
unsigned long Offset() const
{
    return _offset;
}
unsigned long Length() const
{
    return _length;
}
void setID( unsigned long id )
{
    _id = id;
}
void setString( const char *strIn )
{
    _strEntry = strIn;
}
void setString( const string& sIn )
{
    _strEntry = sIn;
}
```

```
        void setOffset( unsigned long offset )
        {
            _offset = offset;
        }
        void setLength( unsigned long length )
        {
            _length = length;
        }
        virtual void write( ofstream& out )
        {
            // Get the current output position.
            setOffset( out.tellp() );
            // Write out the string.
            const char *strOut = String().c_str();
            out << strOut;
        }
        virtual void dump( ostream& out )
        {
            out << "StringEntry:" << endl;
            out << "ID : " << ID() << endl;
            out << "String: [" << String().c_str() << "]" << endl;
            out << "Length: " << Length() << endl;
            out << "Offset: " << Offset() << endl;
        }
};
class StringWriter
{
private:
vector< StringEntry >       _entries;
    string                  _fileName;
    string                  _outputFileName;
    string get_line( ifstream& in )
    {
        string sOut = "";
        char cLastChar = 0;
        while ( !in.eof() )
        {
            // Read in a character at a time. If we hit end of line,
```

```
            // and the last character is NOT \, we are done.
            char c;
            in.get(c);
            if ( in.fail() )
            break;
            if ( c == '\n' )
            {
               // We found a return. See whether the last thing was a backslash.
               if ( cLastChar != '\\' )
               break;
               else
               {
                  // Remove the backslash.
                  sOut = sOut.substr(0, sOut.length()-1);
               }
            }
            sOut += c;
            cLastChar = c;
         }
         return sOut;
      }
      virtual bool ProcessLine( const string& sIn )
      {
         // There has to be a colon (:).
         int nColonPos = sIn.find_first_of( ':' );
         if ( nColonPos == string::npos )
               return false;
         // Get the pieces.
         string sNumber = sIn.substr(0, nColonPos);
         string sValue = sIn.substr( nColonPos+1 );
         // Add it to our list.
         StringEntry se( atol(sNumber.c_str()), sValue.c_str() );
         _entries.insert( _entries.end(), se );
         return false;
   }
      virtual bool Load()                                    → 1
      {
         // Try to open the input file.
```

```
            ifstream in(_fileName.c_str());
            if ( in.fail() )
               return false;
            // Read in the first line for version information.
            string sLine = get_line(in);
            if ( strcmp( sLine.c_str(), VERSION_STRING ) )
               return false;
            for ( int i=0; i<10; ++i )
            {
               if ( in.fail() )
                  break;
               sLine = get_line(in);
               // Ignore blank lines.
               if ( sLine.length() == 0 )
                  continue;
               // Ignore comments.
               if ( sLine[0] == '#' )
                  continue;
               if ( ProcessLine( sLine ) )
                  printf("Invalid input: %s\n", sLine.c_str () );
            }
         }
      public:
         StringWriter( void )
         {
         }
         StringWriter( const char *inputFileName, const char *outputFileName )
         {
            _fileName = inputFileName;
            _outputFileName = outputFileName;
            Load();
         }
         virtual bool Save( void )                                    → 2
         {
            // If there are no entries, abort.
            if ( _entries.size() == 0 )
               return false;
            // Try to open the output file.
```

```
        ofstream out( _outputFileName.c_str() );
        if ( out.fail() )
            return false;
        // Okay, process each of them.
        vector< StringEntry >::iterator iter;
        for ( iter = _entries.begin(); iter != _entries.end(); ++iter )
        {
            // Write out the entry.
            (*iter).write( out );
        }
        // Now, process the index file.
        string indexFileName = _outputFileName + ".idx";
        ofstream out2(indexFileName.c_str());
        if ( out2.fail() )
        {
            printf("Unable to open index file %s for output\n",
            indexFileName.c_str() );
            return false;
        }
        for ( iter = _entries.begin(); iter != _entries.end(); ++iter )
        {
            // Write out the entry.
            out2 << (*iter).Offset() << ", " << (*iter).Length() << ", " << (*iter).ID()
            << endl;
        }
        return true;
    }
};
int main(int argc, char **argv)
{
    if ( argc < 3 )
    {
        printf("Usage: StringEntry input-file output-file\n" );
        printf("Where: input-file is the file containing the string definitions\n");
        printf(" output-file is the final generated file name\n");
        return(-1);
    }
    StringWriter s(argv[1], argv[2]);
```

```
        if ( s.Save() == false )
            printf("Error generating file\n");
        return 0;
    }
```

Mã ở trên phân chia thành một lớp lưu trữ (StringEntry), một lớp ghi (StringWriter) và một driver thử nghiệm nhằm minh hoạ cách sử dụng mã. Driver thử nghiệm mong đợi hai đối số: Một file chứa các định nghĩa chuỗi và một đối số xác định tên của file để tạo cho một file xuất. File nhập đơn giản gồm các số ID (các giá trị số nguyên) theo sau là một dấu hai chấm và sau đó là chuỗi để mã hoá thành file xuất. Mỗi mục nhập (entry) trong file định nghĩa được đọc, được phân tích cú pháp và được đặt vào một đối tượng StringEntry. Tất cả điều này xảy ra trong hàm Load của lớp StringWriter được minh hoạ tại → **1**. Sau khi toàn bộ file nhập được phân tích cú pháp, nó được ghi sang định dạng xuất trong phương thức Save của lớp StringWriter, được minh hoạ tại → **2**. Kết quả của việc chạy chương trình này là một file ngôn ngữ mà sau đó được sử dụng bởi ứng dụng mang tính quốc tế.

3. Lưu mã nguồn trong code editor.

Tạo file text nhập liệu

Sau khi tạo ứng dụng để đọc file text và chuyển đổi nó thành một file ngôn ngữ quốc tế, bước kế tiếp là test ứng dụng bằng cách tạo một file text đơn giản chứa các chuỗi mà chúng ta sẽ sử dụng trong file ngôn ngữ quốc tế sau cùng. Các bước sau đây hướng dẫn bạn cách làm điều đó bằng cách tạo một file text rất nhỏ mà chúng ta sử dụng để test ứng dụng:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa text cho file ngôn ngữ test mà chúng ta sẽ sử dụng.

   Trong ví dụ này, file được đặt tên là test.in.eng, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa các chuỗi mà chúng ta muốn đặt trong file ngôn ngữ quốc tế đầu ra.

2. Gõ nhập text sau đây vào file, thay thế các giá trị riêng của bạn bất cứ nơi nào bạn chọn.

```
Version 1.0.0
# This is a comment
# This is another comment \
    but it is very very long
1:Hello                                    → 3
2:Goodbye
```

```
3:Why me?
4:English
5:French
```

3. Biên dịch file nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

   Trong ví dụ này, file nguồn đã được gọi là StringEntry.cpp. Tuy nhiên, bạn có thể gọi nó bằng bất kỳ tên nào bạn thích.

4. Chạy ứng dụng trên hệ điều hành mà bạn chọn, sử dụng file đầu vào dưới dạng một đối số cho ứng dụng.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trong cửa sổ console và hai file sẽ được tạo trong hệ thống file (có tên là my.eng và my.eng.idx):

```
$ ./a.exe test.in.eng my.eng
The my.eng file will look like this:
$ cat my.eng
HelloGoodbyeWhy me?EnglishFrench                    → 4
The my.eng.idx file will look like this:
$ cat my.eng.idx
0, 5                                                → 5
5, 7
12, 7
19, 7
26, 6
```

Như bạn có thể thấy từ hai kết quả được minh hoạ ở trên, sau khi writer hoàn tất file text đầu vào, nó không còn thật sự có định dạng đọc được nữa. Các chuỗi được ghép theo một định dạng xuất thành một định dạng xuất nhị phân (binary) với một file phụ chứa các index biểu thị nơi mỗi chuỗi bắt đầu và kết thúc. Ví dụ, file đầu vào chứa chuỗi được minh hoạ trong listing tại → **3**. Sau đó chuỗi này được ghi sang file xuất binary my.eng, tại → **4**. Index cho mục nhập cụ thể này được thể hiện trong file my.eng.idx tại → **5**. Mục nhập đầu tiên trong index biểu thị vị trí trong file (dựa vào 0). Như bạn có thể thấy chuỗi chúng ta đang xem bắt đầu tại vị trí đầu tiên của file đầu ra. Mục nhập thứ hai trong index biểu thị chiều dài của chuỗi. Trong trường hợp này là → **5**. Do đó, chúng ta đi đến vị trí zero, đến năm ký tự và nó sẽ là chuỗi đầu tiên. Như bạn có thể thấy thực tế đó là chuỗi Hello.

Đọc file quốc tế

Sau khi đã tạo file và file index cho nó, bước kế tiếp là tạo một reader đọc các chuỗi trong những ngôn ngữ khác nhau. Reader sử dụng một chương trình hai bước để đạt được điều này: Đầu tiên nó đọc file index để nó biết tất cả chuỗi nằm ở đâu trong file và sau đó nó tải một chuỗi cần đọc. Các bước sau đây hướng dẫn bạn cách thực hiện.

Listing 46.2: Lớp StringReader

```
#include <stdio.h>
#include <vector>
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
using namespace std;
class StringIndex
{
private:
    unsigned long _offset;
    unsigned long _length;
    unsigned long _id;
protected:
    virtual void Init()
    {
        setOffset(0);
        setLength(0);
        setID(0);
    }
public:
    StringIndex(void)
    {
        Init();
    }
    StringIndex( unsigned long offset, unsigned long length, unsigned id )
{
    Init();
    setOffset( offset );
```

```
        setLength( length );
        setID( id );
    }
    StringIndex( const StringIndex& aCopy )
    {
        setOffset( aCopy.getOffset() );
        setLength( aCopy.getLength() );
        setID ( aCopy.getID() );
    }
    virtual ~StringIndex()
    {
    }
    StringIndex operator=( const StringIndex& aCopy )
    {
        setOffset( aCopy.getOffset() );
        setLength( aCopy.getLength() );
        setID ( aCopy.getID() );
        return *this;
    }
    void setOffset( unsigned long offset )
    {
        _offset = offset;
    }
    void setLength( unsigned long length )
    {
        _length = length;
    }
    void setID( unsigned long id )
    {
        _id = id;
    }
    unsigned long getOffset( void ) const
    {
        return _offset;
    }
    unsigned long getLength( void ) const
    {
        return _length;
```

```
}
unsigned long getID( void ) const
{
    return _id;
}
    virtual void dump( void )
    {
        cout << "StringIndex: " << endl;
        cout << "Offset: " << getOffset() << endl;
        cout << "Length: " << getLength() << endl;
        cout << "ID : " << getID() << endl;
    }
};
class StringReader
{
private:
    string _fileName;
    vector< StringIndex > _indices;
protected:
    virtual bool Load(void)
    {
        string indexFileName = _fileName + ".idx";
        ifstream in(indexFileName.c_str());
        if ( in.fail() )
            return false;
        // Read in each line.
        while ( !in.eof() )
        {
            string sIn = "";
            while ( !in.eof() )
            {
                // Get an input line.
                char c;
                in.get(c);
                if ( in.fail() )
                    break;
                if ( c != '\r' && c != '\n' )
                    sIn += c;
```

```
            if ( c == '\n' )
                break;
            }
            if ( sIn.length() == 0 )
                break;
            // Okay, we have a line. Now, parse it.
            istringstream iss(sIn.c_str());
                char c;
                long lLength = 0;
                long lOffset = 0;
                long lID = 0;
                // Parse the line, eating the comma
                iss >> lOffset >> c >> lLength >> c >> lID;
                StringIndex si(lOffset, lLength, lID);
                si.dump();                                    → 6
                _indices.insert( _indices.end(), si );
            }
            return true;
        }
        virtual string _loadString( const StringIndex& si )
        {
            ifstream in(_fileName.c_str());
            in.seekg( si.getOffset() );
            string retStr;
            for ( int i=0; i<si.getLength(); ++i )
            {
                char c;
                in.get(c);
                if ( in.fail() )
                    break;
                retStr += c;
            }
            return retStr;
        }
    public:
        StringReader(void)
        {
        }
```

```
    StringReader( const char *fileName )
    {
        _fileName = fileName;
        Load();
    }
    StringReader( const StringReader& aCopy )
    {
        _fileName = aCopy._fileName;
        vector< StringIndex >::const_iterator iter;
        for (iter = aCopy._indices.begin(); iter != aCopy._indices.end(); ++iter )
            _indices.insert( _indices.end(), (*iter) );
    }
    string getString( long id )
    {
        // First, see if we have this id.
        vector< StringIndex >::const_iterator iter;
        for (iter = _indices.begin(); iter != _indices.end(); ++iter )
            if ( (*iter).getID() == id )
                return _loadString( (*iter) );
        return string("");
    }
};
```

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch46_1.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã từ Listing 46.2 vào file.

3. Lưu mã nguồn dưới dạng một file trong bộ soạn thảo và đóng ứng dụng soạn thảo.

Test StringReader

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm nhằm minh hoạ cách đọc một file ngôn ngữ và cho thấy lớp được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này chương trình thử nghiệm được đặt tên là ch46a.cpp.

2. Gõ nhập mã từ Listing 46.3 vào file.

Listing 46.3: Driver thử nghiệm String Reader

```
int main(int argc, char **argv)
{
    if ( argc < 3 )
    {
        printf("Usage: ch6_1 string-file-name id1 [id2, ..]\n");
        printf("Where: string-file-name is the name of the compressed string
        file to use\n");
        printf(" id1..etc are the ids to display\n");
        return -1;
    }
    StringReader sReader(argv[1]);
    for ( int i=2; i<argc; ++i )
        {
        int id = atoi(argv[i]);
        string s = sReader.getString( id );
        printf("String %d: [%s]\n", id, s.c_str() );
    }
    return 0;
}
```

3. Lưu file mã nguồn trong code editor.

4. Biên dịch và chạy file nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, việc chạy ứng dụng với các đối số được trình bày sẽ tạo ra kết quả sau đây trên cửa sổ console:

```
$ ./a.exe my.eng 2 3 5 9
String 2: [Goodbye]
String 3: [Why me?]
String 5: [French]
String 9: []
```

Bằng cách xem kết quả ở trên, chúng ta có thể thấy rằng khi chúng ta đọc dữ liệu từ file ngôn ngữ, sử dụng các index mà chúng ta

đã định nghĩa trong chương trình, chúng ta có được trở lại các chuỗi tương tự mà chúng ta đặt ở đó trong file text ngôn ngữ gốc. Điều này cho thấy chương trình làm việc tốt và rằng chúng ta nhận trực tiếp dữ liệu ngôn ngữ từ file chứ không phải từ các chuỗi được nhúng trong ứng dụng.

Kỹ thuật 47: Hash các bản dịch

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các hash table (các bảng băm)
- Tạo một ứng dụng biên dịch với các hash table
- Tạo một file text biên dịch
- Test ứng dụng

Hash table (bảng băm) là một trong những cấu trúc hữu dụng nhất trong Standard Template Library. Các hash table là những cấu trúc dữ liệu chứa hai loại giá trị thường là các cặp khoá (key) - giá trị. Một hash table được sử dụng bất cứ khi nào bạn muốn thay thế một giá trị bằng một giá trị khác hoặc ánh xạ một key nào đó vào một giá trị nào đó (như bạn có thể tưởng tượng, nó là một công cụ thường được mã hoá sử dụng).

Một trong những điều hữu dụng nhất mà bạn có thể thực hiện với một hash table là lưu trữ và dò tìm các giá trị hiện có và thay thế chúng bằng các giá trị mới. Ví dụ, trong một ứng dụng biên dịch, thật là tuyệt vời nếu có thể xây dựng một từ điển từ ngữ và những bản dịch vừa có được của chúng. Khả năng truy tìm thông tin dựa vào một cặp khoá - giá trị tiết kiệm nhiều thời gian trong một ứng dụng. Ví dụ, một tính năng tìm kiếm và thay thế có thể sử dụng chức năng này. Hoặc bạn có thể sử dụng chức năng này để thực thi một bộ phân tích lệnh (command parser) với một sự ánh xạ các chuỗi vào những giá trị số nguyên. Sử dụng các hash table có thể tiết kiệm cho bạn nhiều thời gian trong việc biên dịch một loại đầu vào thành một loại đầu vào khác trong các ứng dụng. Trong kỹ thuật này, chúng ta sẽ xem vấn đề chính xác đó và cách giải quyết nó trong mã ứng dụng.

Tạo một lớp translator

Chúng ta sẽ sử dụng chức năng hash table để tạo một lớp vốn "biên dịch" text bằng cách thay đổi các từ khoá nhất định trong text bằng những từ khác. Hãy tưởng tượng chúng ta muốn thay thế tất cả lần xuất hiện của từ computer bằng từ PC chẳng hạn. Loại chuyển đổi này thường được gọi là một filter (bộ lọc) bởi vì nó lấy dữ liệu nhập và lọc nó sang một định dạng xuất mới. Trong phần này, chúng ta sẽ

tạo lớp Translator lưu trữ text mà chúng ta muốn thay thế cùng với text mà chúng ta muốn chèn vào vị trí của text gốc.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch47, mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã sau đây từ Listing 47.1 vào file.

Listing 47.1: Lớp translator

```
#include <stdio.h>
#include <string>
#include <map>
#include <fstream>
#include <iostream>
using namespace std;
class Translator
{
private:
    string _fileName;
    map<string,string> _dictionary;                    → 2
protected:
    void Clear()
    {
       _dictionary.erase(
    _dictionary.begin(), _dictionary.end() );
    }
    bool Load( const char *fileName )                  → 1
    {
       ifstream in(fileName);
       if ( in.fail() )
          return false;
       // Just read in pairs of values.
       while ( !in.eof() )
       {
          string word;
          string replacement;
          in >> word;
```

```
            if ( word.length() )
            {
               in >> replacement;
               _dictionary[word] = replacement;
            }
         }
         return true;
         }
      public:
         Translator( void )
         {
         }
         Translator( const char *fileName )
         {
            Clear();
            setFileName( fileName );
         }
         Translator( const Translator& aCopy )
         {
            Clear();
            setFileName( aCopy.getFileName() );
         }
         Translator operator=( const Translator&
         aCopy )
         {
            Clear();
            setFileName( aCopy.getFileName() );
         }
         void setFileName( const string&
            sFileName )
         {
            _fileName = sFileName;
            Load(_fileName.c_str());
         }
         string getFileName( void ) const
         {
            return _fileName;
         }
```

```
    string replace( string in )                    → 3
    {
        map<string,string>::iterator iter;
        iter = _dictionary.find( in );
        if ( iter != _dictionary.end() )
        {
            return iter->second;
        }
        return in;
    }
};
```

Mã này quản lý những thao tác biên dịch. Mã gồm ba hàm cơ bản:

- Lớp quản lý một file đầu vào chứa những ánh xạ của text cũ sang text mới mà chúng ta muốn thay thế nó. (Xem → 1).

Phương thức này lấy một tên file, cố mở nó và đọc các cặp dữ liệu nếu thao tác mở đã thành công.

- Lớp quản lý việc lưu trữ các ánh xạ trong một hash table. (Xem → 2).

Điều này được thực hiện bởi hash table được tượng trưng bởi lớp ánh xạ Standard Template Library.

- Lớp thay thế một chuỗi nào đó bằng chuỗi mong muốn trong kết quả sau cùng. (Được minh hoạ tại → 3).

Chú ý rằng chúng ta chỉ việc chuyển vào mỗi chuỗi và thay thế nó nếu nó được tìm thấy trong hash table. Nếu không, chúng ta trả về chuỗi gốc. Điều này cho phép ứng dụng tiết kiệm thời gian bằng việc không cần phải kiểm tra xem chuỗi có cần được thay thế hay không.

3. Lưu mã nguồn dưới dạng một file trong code editor.

Từ điển (dictionary) sẽ làm công việc tải dữ liệu từ một file biên dịch và thay thế các từ riêng lẻ bằng các từ biên dịch đã xác định. Ví dụ, hãy xem xét ý tưởng thay thế tất cả lần xuất hiện của từ good bằng từ bad. Nếu chúng ta được cho chuỗi đầu vào good day, I am having a good time at this goodly party, chúng ta có thể biên dịch chuỗi này thành bad day, I am having a bad time at this goodly party. Chú ý chúng ta chỉ thay thế các mục tương hợp đầy đủ, không phải text xuất hiện ở bất cứ nơi nào trong chuỗi đầu vào. Bây giờ lớp hiện hữu, điều duy nhất chúng ta cần làm là sử dụng nó.

Test lớp translator

Sau khi tạo tự điển, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và cho thấy lớp sẽ được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch47.cpp.

2. Thêm mã từ Listing 47.2 vào file.

Listing 47.2: Driver thử nghiệm lớp translator

```
int main()
{
    Translator t("translate.txt"); → 4
    printf("Enter some text to translate:
      ");
    string in;
    bool bDone = false;
    while ( !bDone ) → 5
    {
       char c;
       cin.get(c);
       if ( c == '\n' )
          bDone = true;
       else
          in += c;
    }
    printf("Initial string: %s\n",
       in.c_str() );
    // Break it down into words.
    string word;
    for ( int i=0; i<in.length(); ++i )
    {
       if ( in[i] == ' ' )
       {
```

```
            if ( word.length() ) → 6
            {
               string sOut = t.replace(
               word );
               cout << sOut << " ";
            }
            word = "";
         }
         else
            word += in[i];
      }
      if ( word.length() ) → 7
      {
         string sOut = t.replace( word );
         cout << sOut << " ";
      }
   }
```

Mã ở trên đơn giản đọc dữ liệu nhập từ bàn phím, phân tích nó thành các từ và sau đó gọi translator (trình biên dịch) để cho thấy bất kỳ phần của chuỗi vốn cần được biên dịch từ file đầu vào. Translator thao tác trên một file được gọi là translate.txt, như được minh hoạ tại → **4**. Bạn có thể dễ dàng thay đổi tên file này hoặc thậm chí chuyển vào một tên file trên dòng lệnh nếu muốn. Mỗi dòng được đọc từ bàn phím, mỗi lần một ký tự cho đến khi gặp phải một ký tự xuống dòng (carriage return).

(Xem → **5**). Sau cùng, chúng ta phân tích cú pháp dòng dữ liệu nhập cho đến khi chúng ta gặp phải một khoảng trống (được minh hoạ tại → **6**) hoặc cuối chuỗi (được minh hoạ tại → **7**). Khi điều này xảy ra, chúng ta thay thế "từ" mà chúng ta đã phân tích cú pháp bằng cách gọi phương thức replace của translator.

3. Lưu file mã nguồn trong bộ soạn thảo và sau đó đóng ứng dụng soạn thảo.

4. Biên dịch file nguồn bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

5. Trong code editor, tạo một file text biên dịch.

   File này sẽ chứa các cặp từ cần được sử dụng trong file biên dịch; mỗi cặp gồm một từ và phiên bản biên dịch của từ đó. File này được đặt tên tên là translation.txt, nhưng bạn có thể đặt cho nó bất kỳ tên nào bạn thích.

6. Đặt text sau đây vào file translation.txt:

```
good bad
insult dis
talk jive
```

Bạn có thể đặt bất kỳ từ bạn muốn trong file. Từ thứ nhất sẽ được thay thế bởi từ thứ hai trong bất kỳ trình tự mà bạn nhập vào hệ thống.

7. Chạy chương trình trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây:

```
$ ./a.exe
        Enter some text to translate:
        Initial string: you are so good to
           talk and not insult me
        you are so bad to jive and not dis
           me
```

Kỹ thuật 48: Thực thi các file ảo

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các file ảo
- Tạo một lớp file ảo
- Test lớp

Những ngày này, dữ liệu ứng dụng có thể rất lớn và có thể chiếm nhiều bộ nhớ. Phụ thuộc vào footprint ứng dụng và hệ điều hành đích, tải toàn bộ file ứng dụng vào bộ nhớ cùng một lúc có thể không thực hiện được. Sự thiếu bộ nhớ phổ biến với các hệ thống nhúng, và với các thiết bị xách tay, nơi chỉ có sẵn một lượng bộ nhớ giới hạn để chia sẻ giữa rất nhiều ứng dụng. Có một số cách để giải quyết những tình huống như vậy từ việc đọc chỉ dữ liệu tuỳ mức độ có thể và không lưu trữ phần dư thừa cho đến giới hạn dữ liệu chỉ trong các cụm và buộc người dùng chọn cụm nào mà họ muốn. Tuy nhiên, những giải pháp này không hoàn toàn tinh tế đối với nhà phát triển hoặc người dùng cuối như file ảo (virtual file). File ảo là một cửa sổ đi vào file mà bạn cố truy cập. Đối với người dùng chúng như thể là toàn bộ file cùng một lúc - nhưng thật ra chúng thấy mỗi lần chỉ một phần nhỏ của nó. Nếu bạn đưa trước các view ảo của các file vào ứng dụng, bạn tiết kiệm thời gian bởi vì bạn sẽ không cần phải quay trở lại để tái thiết kế các ứng dụng khi các file trở nên lớn hơn bạn tưởng.

Tạo một lớp file ảo

Để quản lý các file ảo, chúng ta cần hai lớp khác nhau. Đầu tiên chúng ta sẽ cần một lớp vốn quản lý một cụm dữ liệu nào đó từ file. Lớp này sẽ quản lý dữ liệu trong cụm đó cũng như theo dõi nơi mẫu dữ liệu cụ thể đó đã được đọc từ file và nó có kích cỡ như thế nào. Sau đó, chúng ta cần có một trình quản lý (manager) theo dõi tất cả cụm dữ liệu đó, cấp phát các đối tượng mới để quản lý các phần riêng lẻ vốn được đọc vào và xoá các phần không còn được sử dụng nữa. Hãy tạo một vài lớp để thực hiện điều đó bây giờ.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch48.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho các đối tượng quản lý file ảo.

2. Gõ nhập mã được trình bày trong Listing 48.1 vào file.

Listing 48.1: Các lớp quản lý file ảo

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
using namespace std;
class FileChunk                                    → 1
{
private:
    string _chunk;
    long _offset;
    long _length;
    bool _inuse;
    long _accesses;
protected:
    void Clear()
    {
        _offset = -1;
        _length = -1;
        _chunk = "";
        _inuse = false;
        _accesses = 0;
```

```
    }
    void Copy( const FileChunk& aCopy )
    {
        _offset = aCopy._offset;
        _length = aCopy._length;
        _chunk = aCopy._chunk;
        _inuse = aCopy._inuse;
    }
    bool Read ( ifstream& in, long pos, long length )
    {
        _offset = pos;
        _chunk = "";
        _length = 0;
        // Seek to the position in the stream.
        in.seekg( pos );
        if ( in.fail() )
            return false;
        // Read up to the end of the file or the last of the length
        // bytes.
        for ( int i=0; i<length; ++i )
        {
            char c;
            in.get(c);
            if ( in.fail() )
                break;
            _length ++;
            _chunk += c;
        }
        _inuse = true;
    }
public:
    FileChunk( void )
    {
        Clear();
    }
    FileChunk( ifstream& in, long pos, long length )
    {
        Clear();
```

```
    Read( in, pos, length );
}
FileChunk( const FileChunk& aCopy )
{
    Clear();
    Copy( aCopy );
}
FileChunk operator=( const FileChunk& aCopy )
{
    Clear();
    Copy( aCopy );
    return *this;
}
// Accessors
long Offset()
{
    return _offset;
}
long Length()
{
    return _length;
}
string& Chunk()
{
    _accesses ++;
    return _chunk;
}
bool InUse(void)
{
    return _inuse;
}
void setOffset( long offset )
{
    _offset = _offset;
}
void setLength( long length )
{
    _length = length;
```

```
    }
    void setChunk( const string& chunk )
    {
        _chunk = chunk;
    }
    long AccessCount( void )
    {
        return _accesses;
    }
    bool Load( ifstream& in, long offset, long length )
    {
        Clear();
        return Read( in, offset, length );
    }
};
const int kChunkSize = 128;
class FileChunkManager                                    → 2
{
private:
    int _numChunks;
    FileChunk *_chunks;
    ifstream _in;
    string _fileName;
protected:
    FileChunk *findChunk( long theOffset )
    {
        for ( int i=0; i<_numChunks; ++i )
        {
            if ( _chunks[i].InUse() == true )
            {
                long offset = _chunks[i].Offset();
                long length = _chunks[i].Length();
                if ( theOffset >= offset && theOffset <= offset+length )
                    return &_chunks[i];
            }
        }
    return NULL;
}
```

```
FileChunk *addChunk( long theOffset )
{
    for ( int i=0; i<_numChunks; ++i )
    {
        if ( _chunks[i].InUse() == false )
        {
            if ( _chunks[i].Load( _in, theOffset, kChunkSize ) )
                return &_chunks[i];
        }
    }
    return NULL;
}
FileChunk *getLeastRecentlyAccessed()
{
    int idx = 0;
    long access = _chunks[0].AccessCount();
    for ( int i=0; i<_numChunks; ++i )
    {
        if ( _chunks[i].InUse() == true )
        {
            if ( _chunks[i].AccessCount() < access )
            {
                    idx = i;
                    access = _chunks[i].AccessCount();
                }
            }
        }
        return &_chunks[idx];
    }
public:
    FileChunkManager(void)
    {
        _numChunks = 0;
        _chunks = NULL;
}
FileChunkManager( const char *fileName, int nMaxChunks )
{
    _numChunks = nMaxChunks;
```

```
        _chunks = new FileChunk[ nMaxChunks ];
        _fileName = fileName;
        _in.open( fileName );
    }
    FileChunkManager( const FileChunkManager& aCopy )
    {
        _numChunks = aCopy._numChunks;
        _chunks = new FileChunk[ _numChunks ];
        for ( int i=0; i<_numChunks; ++i )
            _chunks[i] = aCopy._chunks[i];
        _fileName = aCopy._fileName;
        _in.open( _fileName.c_str() );
    }
        virtual ~FileChunkManager( void )
        {
            delete [] _chunks;
        }
        char operator[]( long offset )
        {
            // Find which chunk this offset is in.
            FileChunk *chunk = findChunk( offset );
            if ( chunk == NULL )
            {
                // There are none. See whether we can add one.
                chunk = addChunk( offset );
            }
            // If we have one, just get the data from it.
            // Otherwise, we have to go dump one.
            if ( !chunk )
            {
                chunk = getLeastRecentlyAccessed();
                chunk->Load( _in, offset, kChunkSize );
        }
            // Finally, extract the piece we need.
            int pos = offset - chunk->Offset();
            return chunk->Chunk()[pos];
        }
        // Dump the function to illustrate what is in the chunks.
```

```
    void Dump(void)
    {
        for ( int i=0; i<_numChunks; ++i )
        {
            printf("Chunk %d: %s\n", i, _chunks[i].InUse() ? "In Use" : "NOT
            Used" );
            printf("Offset: %ld\n", _chunks[i].Offset() );
            printf("Length: %ld\n", _chunks[i].Length() );
            if ( _chunks[i].InUse() )
                printf("String: [%s]\n", _chunks[i].Chunk().c_str() );
        }
    }
};
```

Mã ở trên trình bày hai lớp cơ bản: các lớp FileChunk và FileChunkmanager. Lớp FileChunk được minh hoạ tại → **1** quản lý một cụm dữ liệu đơn trong file. Dữ liệu này bao gồm offset trong file, chính đối tượng file và text từ vị trí đó trong file. Nó cũng lưu trữ chiều dài của cụm dữ liệu vốn đã được đọc vào thật sự vì một số cụm ở cuối file có thể nhỏ hơn khối nguyên cỡ. Lớp thứ hai, FileChunkManager được minh hoạ tại → **2** quản lý một mảng đối tượng FileChunk, theo dõi đối tượng nào đang sử dụng và offset của chúng là gì. Khi một khối của file được yêu cầu, đối tượng quản lý xem qua danh sách của nó để thấy có một khối có dữ liệu đó hay không và nếu có yêu cầu text cụ thể đó từ đối tượng cụ thể đó.

Lưu file nguồn trong code editor.

Test lớp file ảo

Sau khi tạo một lớp, bạn tạo một driver thử nghiệm để bảo đảm rằng không chỉ mã chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn như thế nào.

Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và cho thấy lớp sẽ được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch48.

2. Gõ nhập mã được minh hoạ trong Listing 48.2 vào file.

Listing 48.2: Driver thử nghiệm lớp file ảo

```
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        printf("Usage: ch6_3 filename\n");
        printf("Where: filename is the file
            to load\n");
    }
    FileChunkManager fcm( argv[1], 5 );
    for ( int i=0; i<4096; ++i )
    {
        char c = fcm[i];
    }
    fcm.Dump();
    return 0;
}
```

3. Lưu file mã nguồn trong code editor và đóng ứng dụng soạn thảo.
4. Biên dịch toàn bộ ứng dụng bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng.

Bạn sẽ cần chuyển vào một tên file cho chương trình để quản lý. Đối với kết quả mẫu này, file text thật sự tượng trưng cho chương trình ch48.cpp được sử dụng.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 48.3 khi bạn chạy ứng dụng trên một chương trình nào đó.

Listing 48.3: Kết quả từ driver thử nghiệm

```
Chunk 0: In Use
Offset: 3999
Length: 128
String: [
        int pos = offset - chunk->Offset();
        return chunk->Chunk()[pos];
    }
    // Dump function to illustrate what is
```

```
        in the chunks.
]
Chunk 1: In Use                                          → 3
Offset: 129
Length: 128
String: [ate:
    string _chunk;
    long _offset;
    long _length;
    bool _inuse;
    long _accesses;
protected:
    void Clear()
    {
        _offset = -1;]
Chunk 2: In Use
Offset: 258
Length: 128
String: [et = -1;
        _length = -1;
        _chunk = "";
        _inuse = false;
        _accesses = 0;
    }
    void Copy( const FileChunk& aCopy )
    {
        _offset ]
Chunk 3: In Use
Offset: 387
Length: 128
String: [offset = aCopy._offset;
        _length = aCopy._length;
        _chunk = aCopy._chunk;
        _inuse = aCopy._inuse;
    }
    bool Read( ifstream& in]
Chunk 4: In Use
Offset: 516
```

```
Length: 128
String: [& in, long pos, long length )
    {
        _offset = pos;
        _chunk = "";
        _length = 0;
        // Seek to the position in the
            stream
    in.se]
```

Kết quả ở trên xác định mỗi cụm của file được đọc vào như thế nào và theo thứ tự nào. Bạn có thể thấy các cụm riêng lẻ được đọc vào như thế nào, chẳng hạn như cụm được minh hoạ tại dòng được đánh dấu là → **3**. Bạn có thể thấy cụm gồm một khối text dài 128 byte bắt đầu tại vị trí 129. Cụm này được đánh dấu là "In Use" biểu thị rằng trình quản lý vẫn đang xử lý từ cụm đó.

Cụm gồm một khối text dài 128 byte.

Như bạn có thể thấy từ kết quả ở trên, file được đọc trong các cụm và các cụm đó không nằm theo thứ tự vị trí một cách cụ thể. Không bao giờ có hơn 512 byte đang sử dụng vào bất cứ thời điểm nào, nhưng đối với người dùng dường như toàn bộ file có sẵn để sử dụng.

Cải thiện lớp file ảo

Trong khi lớp file ảo chắc chắn hữu dụng, nhưng có một số cải tiến có thể được thực hiện cho nó chẳng hạn như:

- Kích cỡ của mỗi cụm có thể được cấu hình
- Thuật toán đơn giản vốn quyết định cụm nào cần loại bỏ khi tất cả cụm có sẵn được cải tiến.

Kỹ thuật 49: Sử dụng các Iterator cho các tập hợp

Tiết kiệm thời gian bằng cách:

- Sử dụng các tập tập (collection)
- Tìm hiểu các iterator (bộ lặp)
- Xử lý các tập hợp với các iterator
- Hiểu kết quả

Các tập hợp (collection) là trung tâm của Standard Template Library (STL). Chúng hình thành cơ sở cho khả năng tái sử dụng các lớp và cho phép bạn xử lý các nhóm lớn dữ liệu một cách có thứ tự và không thứ tự. Nếu không có các tập hợp, việc viết mã sẽ gây ra nhiều vấn đề hơn - chúng ta phải viết các mảng, danh sách liên kết riêng của chúng ta và những thứ tương tự. Chắc chắn bạn vẫn phải viết các lớp mảng riêng của bạn hoặc thậm chí sử dụng các mảng tĩnh, nhưng những lớp này sẽ không có những năm thử nghiệm và sử dụng mà các tập hợp trải qua. Điều này có nghĩa là có nhiều lỗi hơn, nhiều sự cố thiết kế hơn và cả một tiến trình phát triển chậm hơn. Bằng cách sử dụng các lớp trong STL cho các đối tượng chứa, bạn tiết kiệm thời gian bằng việc không cần phải phát triển các lớp riêng của bạn và bằng việc không cần phải gỡ rối mã mà bạn vừa viết để thực thi đối tượng chứa riêng của bạn.

Để giao diện với các tập hợp thuộc bất kỳ loại, STL cung cấp một công cụ chung chung được gọi là iterator (bộ lặp). Các iterator là những công cụ rất hữu dụng, rất đơn giản cho phép bạn đạt được bất kỳ mẫu của một tập hợp, xử lý nó và in ra những giá trị của các phần tử dữ liệu của đối tượng chứa. Khá tiện lợi, nhưng các iterator có nhiều khả năng hơn thế - như được minh hoạ trong kỹ thuật này. Như với tên gọi, một iterator cho phép bạn "lặp lại trên" (hoặc di chuyển qua) một tập hợp dữ liệu. Ví dụ, khi viết các mảng C++ chuẩn, chúng ta có thể tạo ra một số mã như sau:

```
int array[20];
for ( int i=0; i<20; ++i )
    printf("Array element %d = %d\n", i, array[i] );
```

Mã này làm việc cho một mảng bình thường bởi vì tất cả trong mảng được bảo đảm là nằm gần kề trong bộ nhớ. Với các đối tượng chứa STL sự bảo đảm đó không hiện hữu. Các đối tượng chứa STL quản lý các vùng nhớ dữ liệu vốn có thể được phân tán khắp nơi trong bộ nhớ. Tuy nhiên, với một lớp iterator, chúng ta có thể xem các tập hợp STL như thể chúng giống như mã ở trên.

Kỹ thuật này trình bày một ví dụ về việc sử dụng các tập hợp đối tượng chứa trong STL và cách sử dụng các iterator với những tập hợp đó. Kỹ thuật này khai thác một số loại tập hợp khác nhau từ các vector (mảng) đến các map (ánh xạ) và danh sách liên kết (linked list). Trong tất cả trường hợp chúng ta có thể lặp lại trên những trường hợp này bằng cách sử dụng các loại iterator tương tự nhau. Kỹ thuật này giải thích cách di chuyển tiến lùi qua các tập hợp. Cũng như cách chèn và loại bỏ những thứ ra khỏi các loại đối tượng chứa khác nhau. Sau cùng, kỹ thuật này xem xét một số điều thú vị hơn về các iterator chẳng hạn như hoán đổi các phần tử và sử dụng các iterator trên các file. Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch49.cpp mặc dù Bạn cũng có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá cần thiết.

2. Gõ nhập mã từ Listing 49.1 vào file.

Listing 49.1: Lặp lại trên các lớp tập hợp trong STL

```
#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <list>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <set>
using namespace std;
int main( int argc, char **argv )
{
    // Add a bunch of items to each type.
    char *names[] = {
        "Matt",
        "Sarah",
        "Rachel",
        "Jenny",
        "Lee",
        "Kim",
        NULL
    };
    // First, do the vector.
    vector< string > nameArray;
    for ( int i=0; names[i]; ++i )
    {
        nameArray.insert( nameArray.end(), names[i] );
    }
// Next, load the map.
map< char, string > nameMap;
```

```
for ( int i=0; names[i]; ++i )
    nameMap[ names[i][0] ] = names[i];
// The linked list is next.
list< string > nameList;
for ( int i=0; names[i]; ++i )
{
    nameList.insert( nameList.end(), names[i] );
}
// Sets are popular.
set<string, greater<string> > nameSet;
for ( int i=0; names[i]; ++i )
{
    // Try inserting them twice to see what happens.
    nameSet.insert( nameSet.end(), names[i] );
    nameSet.insert( nameSet.end(), names[i] );
}
// Now, iterate over them.
for ( int i=0; i<nameArray.size(); ++i )
    printf("Array[%d] = %s\n", i, nameArray[i].c_str() );
map<char, string>::iterator iter;
int idx = 0;
for ( iter = nameMap.begin(); iter != nameMap.end(); ++iter )        → 1
{
    printf("Map Entry[%d]:\n", idx);
    printf("Key : %c\n", (*iter).first );
    printf("Value : %s\n", (*iter).second.c_str() );
    idx ++;
}
printf("Set:\n");
set<string, greater<string> >::iterator set_iter;
for ( set_iter = nameSet.begin(); set_iter != nameSet.end(); ++set_iter )
{
    printf("Set Entry: %s\n", (*set_iter).c_str() );
}
printf("Original List:\n");
list<string>::iterator list_iter;
for ( list_iter = nameList.begin(); list_iter != nameList.end(); ++list_iter )
{
```

```
        printf("List Entry: %s\n", (*list_iter).c_str() );
    }
    // Iterators can be used to remove items.
    for ( list_iter = nameList.begin(); list_iter != nameList.end(); ++list_iter )
    {
        if ( (*list_iter) == "Matt" )
        {
            // Note, once we delete something, the iterator is no longer
            // valid.
            nameList.erase( list_iter );
            break;
        }
    }
    printf("Final List:\n");
    for ( list_iter = nameList.begin(); list_iter != nameList.end(); ++list_iter )
    {
        printf("List Entry: %s\n", (*list_iter).c_str() );
    }
    // You can also iterate in reverse.                    → 2
    printf("In reverse\n");
    list<string>::reverse_iterator riter;
    for ( riter = nameList.rbegin(); riter != nameList.rend(); ++riter )
    {
        printf("List Entry: %s\n", (*riter).c_str() );
    }
    // Iterators can be used to swap two elements.
    iter_swap (nameList.begin(), —nameList.end());
    printf("Swapped:\n");
    for ( list_iter = nameList.begin(); list_iter != nameList.end(); ++list_iter )
    {
        printf("List Entry: %s\n", (*list_iter).c_str() );
    }
    // Finally, you can iterate over streams.
    ifstream in("ch6_4.cpp");
    istream_iterator<string> cinPos(in);                    → 3
    for ( int i=0; i<10; ++i )
    {
        if (cinPos != istream_iterator<string>())
```

```
            {
                cout << *cinPos++;
            }
            cout << endl;
        }
        cout << endl;
        return 0;
    }
```

Mã ở trên trình bày những cách khác nhau mà chúng ta có thể lặp lại trên các lớp tập hợp trong STL. Ví dụ, tại → 1, bạn thấy sự lặp lại trên một tập hợp ánh xạ. Các ánh xạ (map) thường được truy cập bằng cách tham chiếu đến các phần tử nào đó trong chúng, nhưng như bạn có thể thấy ở đây, chúng cũng có thể được liệt kê theo thứ tự bằng cách sử dụng các iterator. Dòng → 2 cho thấy chúng ta có thể lặp lại trên một tập hợp theo thứ tự đảo ngược như thế nào - nghĩa là bắt đầu ở cuối và làm việc trở về ngay đầu - đơn giản bằng cách thay đổi loại iterator mà chúng ta sử dụng. Chú ý rằng mặc dù chúng ta đi ngược lại qua các đối tượng chứa, nhưng chúng ta vẫn gia tăng iterator. Điều này có nghĩa rằng chúng ta có thể viết một khối mã để lặp lại trên một tập hợp và sau đó chọn iterator mà chúng ta muốn sử dụng để định hướng mà không cần phải thay đổi bất kỳ mã.

3. Lưu file mã nguồn trong bộ soạn thảo và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy chương trình trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 49.2 trong cửa sổ console.

Listing 49.2: Kết quả từ cuộc thử nghiệm iterator.

```
Array[0] = Matt
Array[1] = Sarah
Array[2] = Rachel
Array[3] = Jenny
Array[4] = Lee
Array[5] = Kim
Map Entry[0]:
Key : J
Value : Jenny
```

```
Map Entry[1]:
Key : K
Value : Kim
Map Entry[2]:
Key : L
Value : Lee
Map Entry[3]:
Key : M
Value : Matt
Map Entry[4]:
Key : R
Value : Rachel
Map Entry[5]:
Key : S
Value : Sarah
Set:
Set Entry: Sarah
Set Entry: Rachel
Set Entry: Matt
Set Entry: Lee
Set Entry: Kim
Set Entry: Jenny
Original List:
List Entry: Matt
List Entry: Sarah
List Entry: Rachel
List Entry: Jenny
List Entry: Lee
List Entry: Kim
Final List:                                    → 4
List Entry: Sarah
List Entry: Rachel
List Entry: Jenny
List Entry: Lee
List Entry: Kim
In reverse
List Entry: Kim
```

```
List Entry: Lee
List Entry: Jenny
List Entry: Rachel
List Entry: Sarah
Swapped:                                    → 5
List Entry: Kim
List Entry: Rachel
List Entry: Jenny
List Entry: Lee
List Entry: Sarah
#include
<iostream>
#include
<string>
#include
<vector>
#include
<map>
#include
<list>
```

Kết quả được minh hoạ trong Listing 49.2 cho thấy chương trình thử nghiệm chạy tốt. Từ → **4**, chúng ta có thể thấy rằng mục nhập Matt đã bị loại bỏ khỏi danh sách. Tương tự từ → **5**, chúng ta có thể thấy các mục nhập Rachel và Lee trong danh sách đã được hoán đổi. Điều này biểu thị mã làm việc tốt.

Như bạn có thể thấy iterator là một công cụ cực kỳ mạnh. Nó không chỉ có thể được sử dụng trên tất cả loại tập hợp khác nhau mà nó còn cho phép bạn xem một stream đơn giản là một tập hợp dữ liệu (được minh hoạ trong Listing 49.1 tại → **3**). (Dĩ nhiên, khi bạn nghĩ về nó, điều đó cho thấy một stream chính xác là gì). Stream (luồng) được chuyển đến bộ lặp luồng (stream iterator) mà sau đó cho phép chúng ta bước qua file mỗi lần một ký tự nhảy đến bất kỳ vị trí nào mà chúng muốn trong file bằng cách tăng lượng hoặc giảm lượng iterator.

Kỹ thuật 50: Ghi đè allocator (bộ cấp phát) cho một lớp tập hợp

Tiết kiệm thời gian bằng cách:

- Khai thác bộ nhớ allocator
- Sử dụng các bộ cấp phát bộ nhớ
- Thực thi một bộ cấp phát bộ nhớ với một framework hiện có của các lớp
- Hiểu kết quả

Một trong những ưu thế thật sự của Standard Template Library (STL) là bạn có thể cấu hình các lớp để đáp ứng những nhu cầu cụ thể của bạn. Ví dụ, bạn có thể đặt bất kỳ lớp bạn cần trong một template nào đó miễn là lớp tuân theo những qui tắc cơ bản nhất định. Hoặc bạn có thể thay đổi cách bộ nhớ được cấp phát cho tập hợp - bạn có thể cấp phát bộ nhớ từ một heap tĩnh cho một tập hợp nào đó chẳng hạn, hoặc có lẽ làm cho bộ nhớ ổn định thông qua cơ chế cấp phát (nghĩa là không cho phép bộ nhớ di chuyển xung quanh trên hệ thống). Hai ví dụ cụ thể này cho phép bạn tận dụng STL trong một hệ thống nhúng. Bất cứ những gì bạn có thể mơ nghĩ đến cho hệ thống sử dụng, STL có thể giải quyết.

Mục đích của các bộ cấp phát bộ nhớ (memory allocator) là cung cấp các lớp đối tượng chứa với một cách độc lập với lớp là nhận được bộ nhớ được sử dụng cho việc lưu trữ các đối tượng. Ví dụ, khi định nghĩa một mảng trong C++ chuẩn chẳng hạn như:

```
int array [20];
```

Chúng ta yêu cầu 20 từ lưu trữ từ trình biên dịch. Tương tự chúng ta có thể cấp phát động một khối bộ nhớ như sau:

```
int *array = new int [20];
```

Trong trường hợp này, chúng ta đã yêu cầu động 20 từ lưu trữ đó, nhưng lần này từ chính hệ điều hành thay vì từ trình biên dịch. Tuy nhiên, trong STL, vấn đề phức tạp hơn. Làm thế nào chúng ta cấp phát các khối bộ nhớ khi chúng ta không biết chính xác những khối đó sẽ có kích cỡ bao nhiêu? Đó là những gì mà các bộ cấp phát (allocator) thực hiện cho chúng ta. Chúng cho phép chúng ta cấp phát bộ nhớ trong những đối tượng chứa STL.

Khả năng thay thế cách bộ nhớ được cấp phát có thể rất quan trọng nếu bạn làm việc trong một môi trường có bộ nhớ giới hạn hoặc môi trường mà bộ nhớ được cấp phát một cách đặc biệt. Khả năng

này giúp bạn khỏi phải thay đổi mã nền tảng để làm cho các lớp hoạt động. Bằng cách thực thi các bộ cấp phát tuỳ ý trong những ứng dụng, bạn có thể truy vết các phần cấp phát, tìm các lỗi trong mã và chuyển đổi nhanh mọi thứ nếu có cơ hội. Kỹ thuật này xem xét những điểm cơ bản về việc thực thi bộ cấp phát riêng của bạn cho STL - và sử dụng nó với framework hiện có của các lớp. Bởi vì chúng ta có thể thay đổi bộ cấp phát cho các đối tượng chứa STL, chúng ta có thể cấu hình chúng để làm việc trong những môi trường khác chẳng hạn như các hệ thống nhúng, các hệ điều hành khác nhau hoặc thậm chí các thiết bị xách tay vốn cấp phát bộ nhớ một cách hoàn toàn khác. Bằng việc hiểu cách thay đổi việc cấp phát bộ nhớ, bạn sẽ tiết kiệm thời gian trong việc gỡ rối các lỗi bộ nhớ phức tạp hoặc trong việc mang chuyển mã từ hệ thống này sang hệ thống khác.

Tạo một bộ cấp phát bộ nhớ tuỳ ý

Để khai thác cách cấp phát bộ nhớ trong STL, hãy tạo một khung sườn đơn giản cho phép ghi đè bộ cấp phát bộ nhớ mặc định cho một lớp đối tượng chứa. Điều này sẽ cho bạn thấy tiến trình cấp phát làm việc như thế nào và cho phép bạn có một khuôn mẫu mà từ đó bạn có thể tạo các bộ cấp phát riêng của bạn nếu có nhu cầu. Tìm hiểu tiến trình này sẽ tiết kiệm cho bạn thời gian đáng kể trong việc hiểu các bộ cấp phát bộ nhớ vốn không được ghi chép đầy đủ trong tài liệu STL.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file header của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch50.h mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho bộ cấp phát.

2. Gõ nhập mã từ Listing 50.1 vào file.

Listing 50.1: File định nghĩa lớp Allocator

```
#ifndef _ALLOCATOR_H_
#define _ALLOCATOR_H_
#include <limits>
#include <iostream>
using namespace std;
template <class T>
class MyAlloc {
    public:
        // Type the definitions.
        typedef T value_type;
```

```
        typedef T* pointer;
        typedef const T* const_pointer;
        typedef T& reference;
        typedef const T& const_reference;
        typedef std::size_t size_type;
        typedef std::ptrdiff_t difference_type;
    template <class U>
    struct rebind {
        typedef MyAlloc<U> other;
    };
// Return the address of values.
T* address (T& value) const {
    return &value;
}
/* Constructors and destructor
* have nothing to do because the allocator has no state.
*/
MyAlloc() throw()
{}
MyAlloc(const MyAlloc&) throw()
{}
template <class U>
MyAlloc (const MyAlloc<U>&) throw()
{}
~MyAlloc() throw() {
}
// Return maximum number of elements that can be allocated.
size_t max_size () const throw()
{
    return numeric_limits<std::size_t>::max() / sizeof(T);
}
// Allocate but don't initialize num elements of type T.
T* allocate (size_t num, const void* = 0)                    → 1
{
    // Print message and allocate memory with global new.
    cerr << "allocate " << num << " element(s)"
    << " of size " << sizeof(T) << std::endl;
```

```
    T* ret = (T*)(::operator new(num*sizeof(T)));
    cerr << " allocated at: " << (void*)ret << std::endl;
    return ret;
}
// Initialize elements of allocated storage p with value value.
void construct (T* p, const T& value)                    → 2
{
    // Initialize memory with placement new.
    new((void*)p)T(value);
}
// Destroy elements of initialized storage p.
void destroy (T* p)                                      → 3
{
    // Destroy objects by calling their destructor.
    p->~T();
}

    // Deallocate storage p of deleted elements.
    void deallocate (T* p, size_t num)                   → 4
    {
        // Print message and deallocate memory with global delete.
        cerr << "deallocate " << num << " element(s)"
                << " of size " << sizeof(T)
                << " at: " << (void*)p << std::endl;
        ::operator delete((void*)p);
    }
};
// Return that all specializations of this allocator are interchangeable.
template <class T1, class T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw() {
    return true;
}
template <class T1, class T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) throw()
{
```

```
        return false;
    }
    #endif
```

Listing ở trên không thật sự làm bất cứ điều gì; nó đơn giản thực thi định nghĩa lớp cho bộ cấp phát. Chúng ta đã ghi đè bốn phương thức vốn thật sự quan trọng ở đây:

- allocate, được minh hoạ tại → **1**. Phương thức này được gọi để cấp phát cụ thể một khối bộ nhớ. Trong trường hợp này chúng ta chỉ việc trả về một khối bộ nhớ trong khi in một số thông tin chẩn đoán.
- construct, được minh hoạ tại → **2**. Phương thức này được gọi để tạo một đối tượng mới với một khối bộ nhớ. Thật may, C++ cung cấp một phiên bản đặc biệt của toán tử new, được gọi là inplace new vốn cho phép bạn tạo một đối tượng trong một khối cấp phát sẵn. Điều này cho chúng ta sử dụng khối được cấp phát bởi phương thức allocator.
- destroy, được minh hoạ tại → **3**. Phương thức này không được ấn định để huỷ cấp phát khối bộ nhớ bởi vì nó có thể được tái sử dụng trong đối tượng chứa sau này. Thay vào đó, nó chỉ trực tiếp gọi ra destructor cho lớp.
- deallocate, được minh hoạ tại → **4**. Phương thức này giải phóng khối được cấp phát vốn đã được cấp phát trong phương thức allocate. Một lần nữa, chúng ta đã thêm một số thông tin chẩn đoán ở đây.

3. Lưu và đóng file nguồn trong bộ soạn thảo.
4. Trong code editor, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

Trong ví dụ này, file được đặt tên là ch50.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã thử nghiệm để sử dụng bộ cấp phát.

5. Gõ nhập mã từ Listing 50.2 vào file.

Listing 50.2: Driver thử nghiệm cho bộ cấp phát tuỳ ý

```
#include <vector>
#include "Allocator.h"
class MyBuffer
{
private:
    char *_buffer;
    int _length;
    virtual void Init()
    {
```

```
        setBuffer( NULL );
        setLength( 0 );
    }
    virtual void Message( const char *msg )
    {
        cout << "MyBuffer: " << msg << endl;
    }
public:
    MyBuffer( void )
    {
        Message("Void Constructor");
        Init();
    }
    MyBuffer( int length, const char
        *inBuffer )
    {
        Message("Full Constructor");
        setLength( length );
        setBuffer( inBuffer );
    }
    MyBuffer( const MyBuffer& aCopy )
    {
        Message("Copy Constructor");
        setLength ( aCopy.getLength() );
        setBuffer( aCopy.getBuffer() );
    }
    virtual ~MyBuffer( void )
    {
        Message("Destructor");
        if ( _buffer )
            delete [] _buffer;
    }
    MyBuffer operator=( const MyBuffer&
    aCopy )
    {
        Message("operator=");
        setLength ( aCopy.getLength() );
```

```
            setBuffer( aCopy.getBuffer() );
            return *this;
        }
        virtual void setLength( int length )
        {
            _length = length;
        }
        virtual void setBuffer( const char
        *buffer )
        {
            if ( buffer )
            {
                _buffer = new
                char[strlen(buffer)+1];
                memcpy( _buffer, buffer,
                strlen(buffer) );
            }
            else
                _buffer = NULL;
        }
        virtual int getLength( void ) const
        {
            return _length;
        }
        virtual const char *getBuffer( void )
            const
        {
            return _buffer;
        }
};
int main()
{
    std::vector<MyBuffer, MyAlloc<MyBuffer>
    > myVector;
    const char *s1 = "Hello world";
    const char *s2 = "Goodbye cruel world";
    const char *s3 = "Hello again world";
    MyBuffer m1(strlen(s1), s1);
```

```
        MyBuffer m2(strlen(s2), s2);
        MyBuffer m3(strlen(s3), s3);
        myVector.insert( myVector.end(), m1 );
        myVector.insert( myVector.end(), m2 );
        myVector.insert( myVector.end(), m3 );
}
```

Driver thử nghiệm đơn giản thi hành một số cấu trúc cơ bản hơn của một đối tượng chứa, tạo một vài đối tượng mới và thêm chúng vào một vector. Vector bị huỷ khi nó nằm ngoài phạm vi. Chúng ta đã thêm một số thông báo in chẩn đoán để minh hoạ khi nào các mẫu mã khác nhau trong lớp MyBuffer được gọi.

6. Lưu file nguồn trong bộ soạn thảo và đóng ứng dụng soạn thảo.
7. Biên dịch ứng dụng bằng trình biên dịch bạn chọn trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả được minh hoạ trong Listing 50.3 khi bạn chạy chương trình trong cửa sổ console.

Listing 50.3: Kết quả từ driver thử nghiệm

```
$ ./a.exe
MyBuffer: Full Constructor                    → 5
MyBuffer: Full Constructor
MyBuffer: Full Constructor
allocate 1 element(s) of size 12              → 6
allocated at: 0xa050768
MyBuffer: Copy Constructor
allocate 2 element(s) of size 12
allocated at: 0xa050788
MyBuffer: Copy Constructor
MyBuffer: Copy Constructor
MyBuffer: Destructor
deallocate 1 element(s) of size 12 at:
0xa050768
allocate 4 element(s) of size 12
allocated at: 0xa0507d0
MyBuffer: Copy Constructor                    → 7
MyBuffer: Copy Constructor
MyBuffer: Copy Constructor
MyBuffer: Destructor
```

```
MyBuffer: Destructor
deallocate 2 element(s) of size 12 at:
0xa050788                                    → 8
MyBuffer: Destructor
MyBuffer: Destructor
MyBuffer: Destructor
MyBuffer: Destructor
MyBuffer: Destructor
MyBuffer: Destructor
deallocate 4 element(s) of size 12 at:
0xa0507d0
```

Kết quả cho thấy bộ cấp phát làm công việc của nó, cho chúng ta biết khi nào mọi thứ được cấp phát và được huỷ. Các thông báo chúng ta đặt trong constructor được in ra (được minh hoạ tại → **5**) từ việc tạo trong chương trình chính. Việc cấp phát bộ nhớ của bộ cấp phát được minh hoạ tại → **6**. Điều này tương ứng với lệnh gọi chèn trong vector được gọi là myVector trong chương trình chính. Khi mỗi phần tử được đặt vào mảng, một đối tượng mới được cấp phát bởi phương thức allocate và sau đó đối tượng được sao chép vào khối đó như được minh hoạ tại → **7**. Sau cùng, vector nằm bên ngoài phạm vi và phương thức desallocate được gọi, như được minh hoạ tại → **8**. Điều này gọi các destructor cho lớp như được minh hoạ trong kết quả.

Hãy luôn tạo một phiên bản cấp phát gỡ rối (với các thông báo chẩn đoán in ra) cho các chương trình của bạn để bạn có thể theo dõi các rò rỉ bộ nhớ (và toàn bộ việc sử dụng bộ nhớ).

Kỹ thuật 51: Sử dụng lớp auto_ptr để tránh các rò rỉ bộ nhớ

Tiết kiệm thời gian bằng cách:

- Ngăn những rò rỉ bộ nhớ do các pointer bị ghi đè gây ra
- Sử dụng lớp auto_ptr
- Thực thi lớp auto_ptr
- Hiểu kết quả

Có vô số sản phẩm trên thị trường được phát hiện và giải quyết những rò rỉ bộ nhớ. Sách này cũng đưa vào một số kỹ thuật phát hiện các rò rỉ bộ nhớ. Tuy nhiên, cách tốt nhất để giải quyết những rò rỉ bộ là không có chúng lúc ban đầu. Những kỹ thuật lập trình phòng

thủ có thể tránh những vấn đề rò rỉ bộ nhớ và cuối cùng tiết kiệm cho bạn lượng thời gian đáng kể và tránh nhiều phiền toái.

Một trong những vấn đề rò rỉ bộ nhớ ngấm ngầm nhất và các pointer vốn bị ghi đè hoặc không huỷ cấp phát. Nếu một pointer không được huỷ cấp phát, một rò rỉ bộ nhớ sẽ xảy ra. Nếu đầy đủ các rò rỉ bộ nhớ xảy ra, chương trình không thể cấp phát bộ nhớ mới và có lẽ sẽ đổ vỡ. Với các pointer bị ghi đè, bộ nhớ mà chúng trỏ vào khôngphải là cùng bộ nhớ vốn đã được cấp phát. Kết quả bộ nhớ gốc không được huỷ cấp phát, điều này gây ra vấn đề rò rỉ bộ nhớ. Hoặc, pointer bị ghi đè có thể trỏ vào một thứ nào đó quan trọng trong bộ nhớ và khi nó được huỷ tham chiếu và được sử dụng để chỉnh sửa bộ nhớ đó, nó sẽ khiến cho chương trình đổ vỡ. STL cung cấp một công cụ tuyệt vời để tránh vấn đề đặc biệt này: lớp auto_ptr. Lớp này bảo đảm rằng một pointer luôn bị xoá khi công việc của nó hoàn tất, thậm chí nếu nó đã được sao chép đè, được bỏ qua hoặc được chuyển sang một đối tượng mồ côi. Kỹ thuật này khai thác cách sử dụng lớp auto_ptr để tránh các sự cố trong mã riêng của bạn.

Sử dụng lớp auto_ptr

Lớp auto_ptr làm cho mã sạch hơn bằng việc loại bỏ nhu cầu kiểm tra các cấp phát và huỷ cấp phát những đối tượng nằm ở khắp mã. Hãy xem các bước cần thiết để sử dụng lớp auto_ptr trong mã riêng của bạn. Về cơ bản, chỉ đòi hỏi một "bước" thật sự là bao bọc một pointer được cấp phát trong một đối tượng khuôn mẫu auto_ptr. Chúng ta sẽ xem đối tượng được cấp phát và sau đó được giải phóng như thế nào khi đối tượng khuôn mẫu auto_ptr nằm ngoài phạm vi.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch51.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này chứa mã nguồn cho các lớp.

2. Nhập mã từ Listing 51.1 vào file.

Listing 51.1: Sử dụng lớp auto_ptr với các hàm riêng của bạn.

```
#include <iostream>
#include <memory>
using namespace std;
class Tracker
{
private:
    static int _allocations;
```

```
    static int _frees;
public:
    Tracker( void )
{
    _allocations ++;
}
Tracker( const Tracker& aCopy )
{
    _allocations ++;
}
~Tracker()
{
    _frees ++;
}
static int Allocations()
{
    return _allocations;
}
static int Frees()
{
    return _frees;
}
static void reset()
{
    _allocations = 0;
    _frees = 0;
    }
    static void report()
    {
        cout << "Tracker Class:" << endl;
        cout << "Allocations: " << _allocations
            << endl;
        cout << "Frees: " << _frees << endl;
    }
};
int Tracker::_allocations = 0;
int Tracker::_frees = 0;
```

```
void func1()
{
    Tracker t1;
    Tracker *t2 = new Tracker();
    Tracker t3 = *t2;
    Tracker t4 = t1;
    t2 = new Tracker();                          → 1
}
void func2()                                     → 2
{
    Tracker t1;
    auto_ptr<Tracker> t2( new Tracker );
    Tracker t3 = *t2;
    Tracker t4 = t1;
    t2.reset( new Tracker );
}
void call_an_exception_function()
{
    throw 1;
}
void func3()                                     → 3
{
    Tracker *t = new Tracker;
    call_an_exception_function();
    delete t;
}
void func4()
{
    auto_ptr<Tracker> t(new Tracker);            → 4
    call_an_exception_function();
}
int main(void)
{
    cout << "Running function 1:" << endl;
    func1();
    Tracker::report();                           → 5
    Tracker::reset();                            → 6
```

```
        cout << endl;
        cout << "Running function 2:" << endl;
        func2();
        Tracker::report();
        Tracker::reset();
        cout << endl;
        cout << "Running function 3:" << endl;
        try
        {
        func3();
        }
        catch ( ... )
        {
        }
        Tracker::report();
        Tracker::reset();
        cout << endl;
        cout << "Running function 4:" << endl;
        try
        {
        func4();
        }
        catch ( ... )
        {
        }
        Tracker::report();
}
```

Mã thử nghiệm minh hoạ hai cách khác nhau mà bộ nhớ có thể bị rò rỉ:

- Bạn có thể quên huỷ cấp phát một pointer như được minh hoạ trong hàm func1 tại → 1. Trong trường hợp này, chúng ta chỉ việc cấp phát một đối tượng mới và không bao giờ giải phóng đối tượng vốn tạo một rò rỉ bộ nhớ. Hàm được gọi là func2 được ghi nhãn là → 2 trình bày cùng một mã sử dụng một lớp auto_ptr thay vì sự cấp phát đơn giản.
- Bạn có thể cấp phát và giải phóng một đối tượng, nhưng bởi vì hàm gọi một thứ gì đó vốn đưa ra một ngoại lệ, dòng huỷ cấp phát sẽ không bao giờ được chạy và một rò rỉ bộ nhớ sẽ xảy ra. Sự rò rỉ bộ nhớ tinh vi hơn này minh hoạ trong hàm func3 tại → 3. Hàm func4

trình bày cùng một mã cơ bản sử dụng một khuôn mẫu auto_ptr thay vào đó như được minh hoạ trong dòng được đánh dấu là → **4**.

3. Lưu mã nguồn trong bộ soạn thảo và đóng ứng dụng.
4. Biên dịch ứng dụng bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trong cửa sổ console.

Nếu bạn đã làm đúng mọi thứ, ứng dụng sẽ hiển thị kết quả được minh hoạ trong Listing 51.2.

Listing 51.2: Kết quả từ chương trình thử nghiệm auto_ptr

```
$ ./a.exe
Running function 1:
Tracker Class:
Allocations: 5
Frees: 3
Running function 2:
Tracker Class:
Allocations: 5
Frees: 5
Running function 3:
Tracker Class:
Allocations: 1
Frees: 0
Running function 4:
Tracker Class:
Allocations: 1
Frees: 1
```

Như bạn có thể thấy từ kết quả, lớp Tracker theo dõi các constructor khác nhau được gọi bao nhiêu lần và destructor được gọi bao nhiêu lần trong mỗi lần chạy. Báo cáo được thực hiện thông qua phương thức report của lớp Tracker như được trình bày trong Listing 51.1 tại → **5**. Chú ý rằng chúng ta xác lập lại số đếm mỗi lần, sử dụng hàm reset như được minh hoạ tại → **6**. Trong một tình huống lý tưởng không có các rò rỉ bộ nhớ, các số lần cấp phát và giải phóng như nhau. Đối với các hàm func1 và func3, các số cấp phát và giải phóng không như nhau biểu thị một sự rò rỉ bộ nhớ. Đối với các hàm func2 và func4, các trường hợp auto_ptr, các cấp phát và giải phóng khớp nhau biểu thị không có sự rò rỉ bộ nhớ.

Các hàm mà chúng ta gọi ra ở đây (func1, func2, func3, và func4) trình bày những cách khác nhau mà bộ nhớ có thể bị rò rỉ trong ứng dụng. Như bạn có thể thấy cách bình thường để làm mọi thứ dẫn đến vô số rò rỉ bộ nhớ tiềm tàng khó truy vết. So sánh các trường hợp auto_ptr mà thậm chí với những sự kiện ngoại lệ luôn giải phóng bộ nhớ của chúng.

Kỹ thuật 52: Tránh ghi đè bộ nhớ

Tiết kiệm thời gian bằng cách:

- Hiểu sự ghi đè bộ nhớ
- Tránh sự ghi đè bộ nhớ trong các mảng và khối được cấp phát
- Bảo vệ dữ liệu
- Hiểu kết quả

Sự ghi đè bộ nhớ là một điều rắc rối của nhà lập trình C++. Một sự ghi đè bộ nhớ (memory overwrite) xuất hiện khi bạn ghi sang một vùng bên ngoài một khối được cấp phát. Điều này tệ, vì bạn ghi sang một khối bộ nhớ mà bạn có thể hoặc có thể không sở hữu, nhưng chắc chắn đã không dự định chỉnh sửa. Ví dụ, nếu chúng ta có một câu lệnh C++ như sau:

```
char line[ 10 ];
```

và sau đó chúng ta viết một dòng nào đó như sau:

```
line[ 11 ] = 'a';
```

Chúng ta đã ghi đè một phần bộ nhớ hợp lệ và có thể gây ra các sự cố trong ứng dụng sau này. Thật không may, sự ghi đè bộ nhớ như vậy hơi khó tìm nếu không có những công cụ và phần mềm chuyên dụng. Hoặc, bạn có thể đơn giản tránh ghi bên ngoài các biên hợp lệ của một mảng hoặc khối được cấp phát, nhưng nói dễ hơn làm. Vấn đề thật sự ở đây là khi chúng ta ghi sang vị trí 11 của mảng 10 khối, chúng ta đã ghi đè một thứ gì đó trong bộ nhớ. Thứ gì đó này có thể là một biến khác, có thể là một địa chỉ trả về cho một hàm hoặc nó có thể là một pointer mà trước đó đã được cấp phát. Những điều này không phải là những điều tốt. Bạn cần vẫn duy trì phần cấp phát bộ nhớ mà bạn yêu cầu cho một khối bộ nhớ.

Kỹ thuật này xem xét một cách để sử dụng viết mã C++ để bảo vệ dữ liệu mà chúng ta làm việc không bị ghi đè. Đây là một khái niệm cơ bản về sự đóng gói (encapsulation) trong C++. Nếu bạn có dữ liệu, bạn cần bảo đảm rằng dữ liệu được sử dụng đúng cách. Điều đó có nghĩa là không gán giá trị bên ngoài một dãy hợp lệ, nó cũng có nghĩa là không ghi đè các biên đã được thiết lập.

Tạo một lớp bộ đệm an toàn bộ nhớ

Trường hợp ghi đè bộ nhớ phổ biến nhất khi các chuỗi được sao chép và được gán các giá trị. Lỗi này thường xuất hiện với việc sử dụng các hàm C strcpy và memory xảy ra bởi vì hàm không "biết" một chuỗi lớn như thế nào, do đó nó không thể bảo vệ ngăn ngừa các biên chuỗi được ghi đè. Để giải quyết vấn đề này, chúng ta sẽ tạo một lớp vốn hiểu các biên riêng của nó và chỉ cho phép đúng số ký tự được ghi hoặc được sao chép vào đối tượng. Theo cách này, chúng ta sẽ tạo một lớp bộ đệm (buffer) generic vốn có thể được sử dụng an toàn để lưu trữ các chuỗi trong tất cả ứng dụng, tiết kiệm thời gian trong việc thực thi lớp và trong việc gỡ rối các sự ghi đè bộ nhớ. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho kỹ thuật.

   Trong kỹ thuật này, file được đặt tên là ch52.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã nguồn cho các lớp.

2. Gõ nhập mã từ Listing 52.1 vào file.

Listing 52.1: Lớp Buffer

```
#include <iostream>
#include <fstream>
using namespace std;
class Buffer
{
private:
    char *_buffer;
    long _length;
    virtual void Init()
    {
        _buffer = NULL;
        _length = 0;
    }
    virtual void Clear()
    {
        if ( _buffer )
            delete [] _buffer;
        Init();
    }
```

```
public:
    Buffer(void)
    {
        Init();
    }
    Buffer( const char *buffer, int length )
    {
        Init();
        SetBuffer( buffer, length );
    }
    Buffer ( int length )
    {
        _buffer = new char[length];
        _length = length;
        set( 0 );
    }
    Buffer( const Buffer& aCopy )
{
    Init();
    SetBuffer( aCopy._buffer, aCopy._length );
}
virtual ~Buffer()
{
    if ( _buffer )
    delete [] _buffer;
}
Buffer operator=(const Buffer& aCopy )
{
    Clear();
    SetBuffer( aCopy._buffer, aCopy._length );
    return *this;
}
Buffer operator=(const char *buffer )
{
    Clear();
    SetBuffer( buffer, strlen(buffer) );
    return *this;
```

```
}
char& operator[]( int index )
{
    if ( index < 0 || index >= _length )
        throw "Buffer: Index out of range";                    → 4
    return _buffer[ index ];
}
Buffer operator()(int st, int end)
{
    // Validate the pieces.
    if ( st < 0 || st >= _length )
        throw "Buffer: Start index out of range";
    if ( end < 0 || end >= _length )
        throw "Buffer: End index out of range";
    Buffer b( _buffer+st, end-st+1 );
    return b;
}
void set( char c )
{
    for ( int i=0; i<_length; ++i )
        _buffer[i] = c;
}
virtual void SetBuffer( const char *buffer, int length )
{
    _buffer = new char[ length ];
    for ( int i=0; i<length; ++i )
        _buffer[i] = buffer[i];
    _length = length;
}
void empty()
    {
        set( 0 );
    }
    int Length() const
    {
        return _length;
    }
};
```

```
ostream& operator <<( ostream& out, Buffer &b )
{
    for ( int i=0; i<b.Length(); ++i )
        out << b[i];
    return out;
}
void func1()                                          → 1
{
    char *buffer = new char[10];
    strcpy( buffer, "This is a really long string");
    cout << "Func1: [1]" << buffer << endl;
    memset( buffer, 0, 11 );
    cout << "Func1: [2]" << buffer << endl;
    strcpy( buffer, "This is a short string");
    buffer[ 12] = 0;
    cout << "Func1: [3]" << buffer << endl;
}
void func2()                                          → 2
{
    Buffer b(10);
    try
    {
        b = "This is a really long string";
        cout << "Func2: [1]" << b << endl;
        b.set( 0 );
        cout << "Func2: [2]" << b << endl;
        b[12] = 0;
        cout << "Func2: [3]" << b << endl;
    }
    catch ( ... )                                     → 3
    {
        printf("Exception caught\n");
    }
}
int main()
{
    func1();
```

```
        func2();
    }
```

Hai hàm được trình bày ở trên minh hoạ vấn đề ghi đè bộ nhớ đầu tiên như được giải quyết bởi việc cấp phát mảng C++ chuẩn (func1, được minh hoạ tại → 1) và sau đó sử dụng lớp Bufferđể giải quyết vấn đề (func2, được minh hoạ tại → 2. Trong mỗi trường hợp, chúng ta cấp phát một bộ đệm ký tự gồm 10 ký tự. Trong trường hợp func1, chúng ta có một chuỗi dài hơn 10 ký tự nhiều vào bộ đệm. Điều này gây ra một sự ghi đè bộ nhớ và có thể làm cho chương trình đổ vỡ. Tuy nhiên, bởi vì C++ chuẩn không có cách nào để phát hiện điều này, bạn sẽ không thấy vấn đề này ngay tức thì. Trong trường hợp thứ hai, func2, chúng ta sử dụng lớp Buffer vốn phát hiện sự cố ngay khi bạn cố sao chép chuỗi lớn hơn vào bộ đệm được cấp phát nhỏ hơn và báo cáo nó.

3. Lưu mã nguồn trong code editor và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trong cửa sổ console.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng:

```
$ ./a.exe
Func1: [1]This is a re1
Func1: [2]
Func1: [3]This is a sh
Func2: [1]This is a really long string
Func2: [2]
Func2: [3]
```

Chú ý rằng bạn có thể thấy kết quả khác nhau trên các hệ điều hành khác nhau, phụ thuộc vào lỗi xảy ra như thế nào và nó có nhìn thấy được hay không. Đối với func1, chúng ta thấy rằng chuỗi không được xác lập sang những gì mà chúng ta mong đợi. Bạn có thể thấy chuỗi "This is a really long string" mà thậm chí tệ hơn. Tuy nhiên, trong trường hợp func2 chúng ta không bao giờ gán các chuỗi quá dài bởi vì một ngoại lệ được đưa ra và mã xử lý nó đúng cách.

Như bạn có thể thấy trong mã truy cập kiểu C "thuần tuý" (được minh hoạ tại → 1), không có các cuộc kiểm tra để xem nhà lập trình có ghi đè bộ đệm được cấp phát hay không. Rõ ràng trong mã riêng của bạn, tìm sự cố không hoàn toàn đơn giản. Dường như không có bất kỳ sự cố nào với mã ở đây. Mọi thứ in ra và tiếp tục xử lý phù hợp. Tuy nhiên, các cấu trúc trong bộ nhớ sẽ bị hư hại - và những hư

hại có thể hoặc có thể không xuất hiện khi chương trình đổ vỡ. Tệ hơn, bạn có thể cho người dùng dữ liệu xuất không chính xác mà họ có thể đưa ra những quyết định không hợp lệ trên đó.

Trong trường hợp thứ hai, kiểu C++, (được minh hoạ tại → **2**), bạn có thể thấy rằng mã bảo vệ ngăn ngừa các sự ghi đè và đưa ra một ngoại lệ tại dòng (được minh hoạ tại → **4** và được đón bắt tại dòng được đánh dấu là → **3**) khi một hoạt động ghi không hợp lệ xảy ra. Điều này cho phép nhà lập trình tìm thấy ngay tức thì sự cố đã xảy ra ở đâu và giải quyết nó một cách nhanh chóng và dễ dàng.

Kỹ thuật 53: Đưa ra, đón bắt, và đưa ra lại các ngoại lệ

Tiết kiệm thời gian bằng cách:

- Tìm hiểu xử lý ngoại lệ
- Đưa ra và ghi chép các ngoại lệ
- Giải quyết các ngoại lệ không được xử lý
- Đưa ra lại các ngoại lệ
- Hiểu việc xử lý ngoại lệ sáu cấu trúc

Tính năng xử lý ngoại lệ của C++ là một tính năng nhằm phân biệt nó với hầu như tất cả hệ thống lập trình cũ hơn. Như với Java, C# và những ngôn ngữ lập trình hiện đại khác, C++ cung cấp khả năng nhảy ra khỏi giữa một khối mã khi một sự kiện đặc biệt xảy ra.

Khái niệm về một sự xử lý ngoại lệ (exception handling) thật sự rất đơn giản. Trong các ngôn ngữ lập trình cũ hơn, khi các lỗi xảy ra, một mã lỗi được gởi đến ứng dụng gọi mà có thể - và thường đơn giản bỏ qua nó. Tính năng xử lý ngoại lệ thay đổi điều này. Nó buộc nhà phát triển ứng dụng xem xét trước những gì có thể bị trục trặc trong mã cấp thấp hơn và cung cấp các thường trình để giải quyết bất kỳ lỗi vốn xảy ra. Bây giờ khi một lỗi - đó là một ngoại lệ (exception) - xảy ra, chương trình chuyển quyền điều khiển đến thường trình được ấn định sẵn thích hợp. Tính năng xử lý lỗi bảo đảm các lỗi không bỏ qua. Chúng được giải quyết.

Kỹ thuật này xem xét kỹ hơn việc đưa ra và đón bắt các ngoại. Đầu tiên, kỹ thuật này xem xét việc đưa ra một ngoại lệ và ghi chép nó một cách chung chung. Ghi chép các lỗi thì quan trọng bởi vì nó cho phép bạn cung cấp một nhật ký gỡ rối hoàn chỉnh để thấy trục trặc gì đã xảy ra khi một sự cố xuất hiện. Điều này sẽ tiết kiệm cho bạn thời gian và công sức trong việc gỡ rối ứng dụng và tạo ra mã tốt hơn cho người dùng cuối.

Đưa ra và ghi chép các ngoại lệ

Để hiểu rõ nhất cách sử dụng tính năng xử lý ngoại lệ trong các ứng dụng riêng của bạn, hãy xem một ví dụ đơn giản về việc đưa ra các ngoại lệ mà trong đó những ngoại lệ đó được ghi chép vào một file lỗi đầu ra mà được sử dụng cho những mục đích gỡ rối. Để làm điều này chúng ta sẽ cần hai loại lớp khác nhau:

- Chúng ta cần một lớp để chứa thông tin về trục trặc gì đã xảy ra. Lớp sẽ chứa số dòng nơi lỗi đã xảy ra và thông tin và chi tiết bản chất của lỗi.
- Chúng ta cần một lớp vốn sẽ quản lý tiến trình đón bắt ngoại lệ và ghi chép thông tin vào một nhật ký (log) lỗi.

Các bước sau đây hướng dẫn bạn cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch53.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã nguồn cho các lớp.

2. Gõ nhập mã từ Listing 53.1 vào file.

Listing 53.1: Các lớp xử lý ngoại lệ

```
#include <iostream>
#include <string>
#include <fstream>
#include <stdio.h>
using namespace std;
class ExceptionClass □ 1
{
    string _message;
    string _file;
    long _line;
public:
    ExceptionClass(void)
{
    _message = "Unknown Exception";
}
ExceptionClass( const char *msg, const char *fileName, long lineNo )
{
    _message = msg;
    _file = fileName;
```

```
        _line = lineNo;
    }
    ExceptionClass( const ExceptionClass& aCopy )
    {
        _message = aCopy._message;
        _file = aCopy._file;
        _line = aCopy._line;
    }
    void setMessage(const char *msg, const char *fileName, long lineNo )
    {
        _message = msg;
        _file = fileName;
        _line = lineNo;
    }
        virtual string Report(void) const
        {
            string out;
            out = "Exception reported in file ";
            out += _file.c_str();
            out += " at line ";
            out += _line;
            return out;
        }
        virtual ostream& Report( ostream& out ) const
        {
            out << "Exception reported in file " << _file.c_str() << " at line " <<
            _line << endl;
            out << _message.c_str() << endl;
            return out;
        }
};
class ExceptionCatcher □ 2
{
private:
    string _message;
    ofstream _logFile;
    string _fileName;
public:
```

```
ExceptionCatcher( void )
{
   string msg = "Startup";
   LogMessage( msg );
}
ExceptionCatcher( const char *fileName )
   : _logFile( fileName )
{
   string msg = "Startup";
   msg += " [";
   msg += fileName;
   msg += "]";
   LogMessage( msg );
}
ExceptionCatcher( const ExceptionCatcher& aCopy )
   : _logFile ( aCopy._fileName.c_str() )
{
   _fileName = aCopy._fileName;
   _message = aCopy._message;
   string msg = "Startup";
   msg += " [";
   msg += _fileName;
   msg += "]";
   LogMessage( msg );
}
ExceptionCatcher( const ExceptionClass& exception )
{
   _message = exception.Report();
}
virtual ~ExceptionCatcher()
{
   string msg = "Shutdown";
   LogMessage( msg );
}
virtual void LogMessage( string msg )
{
   if ( !_logFile.fail() )
```

```
            _logFile << msg.c_str() << endl;
        }
        virtual void LogMessage( const ExceptionClass& exception )
        {
            if ( !_logFile.fail() )
            {
                exception.Report( _logFile );
            }
        }
};
void process_option( int x )
{
    if ( x < 2 || x > 8 )
        throw "Invalid Input to process_option";
    int z = 10 / x;
    cout << "Properly processed option " << x << endl;
}
int func1( int x)
throw( ExceptionClass )
{
    ExceptionClass ec;
    try
    {
        switch ( x )
        {
            case 0:
                cout << "You selected the first option" << endl;
                break;
            case 1:
                cout << "You selected the second option" << endl;
                break;
            case 2:
                process_option( x );
            default:
                ec.setMessage( "Invalid Option", __FILE__, __LINE__ );
                throw ec;
        }
}
```

```
    catch ( const char *msg )
    {
        string sErr = "Unknown Error: ";
        sErr += msg;
        ec.setMessage( sErr.c_str(), __FILE__, __LINE__ );
        throw ec;
    }
    return 0;
}
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        cout << "Usage: ch6_9 <inputs>" << endl;
        cout << "Where: inputs is a series of numbers" << endl;
        return -1;
    }
ExceptionCatcher catcher("errors.log");
    // Process the inputs.
    for ( int i=1; i<argc; ++i )
    {
        int iVal = atoi( argv[i] );
        try
        {
            func1(iVal);
        }
        catch ( ExceptionClass& ec )                    → 3
        {
            ec.Report( cout );
            catcher.LogMessage( ec );
        }
        catch ( ... )
        {
            cout << "Caught an exception" << endl;
        }
    }
    return 0;
}
```

Mục đích của mã này là minh hoạ cách xử lý một lỗi, ghi chép lỗi sang một file đầu ra và sau đó tận dụng thông tin trong file đó để xem trục trặc gì thật sự đã xảy ra. Driver thử nghiệm đơn giản cho phép người dùng nhập các tuỳ chọn từ dòng lệnh và sau đó chuyển chúng đến một hàm chọn vốn quyết định những gì cần làm dựa vào đầu vào đó. Nếu đầu vào nằm trong dãy, nó được xử lý. Nếu không một đối tượng ngoại lệ được tạo biểu thị sự cố nào đã xảy ra và ở đâu. Trong trường hợp này, đối tượng lỗi sẽ luôn xuất hiện trong đó người dùng đã nhập một giá trị bên ngoài dãy hợp lệ. Để làm điều này, chúng ta sử dụng lớp ExceptionClass, được minh hoạ tại → 1. Lớp này đơn giản chứa thông tin lỗi, và cho phép ứng dụng truy tìm nó. Nó cũng cung cấp một hàm báo cáo để định dạng thông tin bằng một cách mà người dùng đọc được và in nó ra. Lớp thứ hai ExceptionCatcher (được minh hoạ tại → 2) lấy thông tin từ đối tượng ExceptionClass và in nó sang file được xác định trong constructor của nó. Chú ý khi một lỗi xảy ra, nó được chuyển đến chương trình chính và được đón bắt tại → 3.

3. Lưu mã nguồn trong code editor và sau đó đóng ứng dụng biên soạn.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trong cửa sổ console.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng:

```
$ ./a.exe 1 2 3
You selected the second option
Properly processed option 2
Exception reported in file ch53.cpp at
line 138
Invalid Option
Exception reported in file ch53.cpp at
line 138
Invalid Option
```

Ngoài ra, bạn sẽ có một file trong hệ thống file được gọi là errors.log. File này chứa các mục nhập sau đây:

```
$ cat errors.log
Startup [errors.log]
Exception reported in file ch53.cpp at
line 138
```

```
Invalid Option
Exception reported in file ch53.cpp at
line 138
Invalid Option
Shutdown
```

Kết quả ở trên cho thấy đã có các lỗi được phát hiện trong chương trình, đây là điều mà chúng ta mong đợi bởi vì chúng ta đã đưa ra các giá trị đầu vào không hợp lệ. Đối với các giá trị đã được hiểu, thông báo Properly processed option theo sau là số tuỳ chọn hiển thị. Đối với tất cả giá trị khác, một ngoại lệ được tạo ra và lỗi Invalid Option hiển thị.

Giải quyết các ngoại lệ không được xử lý

Xử lý ngoại lệ là một điều tốt, nhưng đôi khi một loại ngoại lệ xuất hiện mà bạn không mong đợi. Điều này đặc biệt gây rắc rối khi bạn làm việc với các thư viện thứ ba vốn thay đổi theo thời gian hoặc ghi chép kém tài liệu các loại ngoại lệ mà chúng đưa ra. Rõ ràng bạn không thể làm gì về một loại ngoại lệ mà bạn không biết gì - nhưng ít ra bạn có thể ngăn chương trình hoạt động kém khi một ngoại lệ được đưa ra.

Các bước sau đây trình bày một ví dụ về một ngoại lệ không được xử lý đúng cách (một lỗi chia cho không) và cách bạn có thể sử dụng hàm set_terminate cài sẵn để giải quyết nó trước khi nó có thể dẫn đến các rò rỉ bộ nhớ và những điều tương tự trong ứng dụng. Hàm set_terminate định nghĩa một hàm do người dùng thực thi vốn sẽ được gọi trước khi chương trình thoát. Hàm này có thể được sử dụng để huỷ cấp phát bất kỳ khối bộ nhớ được cấp phát hoặc để đóng bất kỳ file đang mở hoặc để làm bất cứ công việc xử lý giây phút cuối khác vốn cần được hoàn tất để làm cho chương trình tắt sạch.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch53.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 53.2 vào file.

Listing 53.2: Sử dụng set_terminate trong ứng dụng

```
int *gAllocatedBuffer = NULL;
void term_func()
{
    cout << "term_func() was called by terminate().\n";
    // Do our global cleanup here.
```

```
        delete [] gAllocatedBuffer;
        // We MUST call exit, because the terminate routine will abort
        // otherwise.
        exit(-1);
}
int func2( void )
{
        set_terminate( term_func );
        try
        {
            int i = 10;
            int j = 0;
            int x = 0;
            if ( j != 0 )
                x = i / j;                                    → 4
            else
                throw "Error: Division by Zero!";
        }
        catch ( ExceptionClass& ec )
        {
            cout << "Exception Caught" << endl;
        }
}
```

Cũng nhớ thêm một lệnh gọi đến func2 trong hàm chính (main) để có thể xem kết quả của chương trình. Sau khi làm điều này, bạn sẽ thấy rằng khi hàm func2 được gọi ra, nó gây ra một lỗi chia không (được minh hoạ tại → **4** trong Listing 53.2), thường làm cho chương trình đổ vỡ mà không giải phóng khối bộ nhớ gAllocatedBuffer được cấp phát trở về hệ điều hành. Thay vào đó, chúng ta kiểm tra để tìm lỗi, đưa ra một ngoại lệ được đón bắt bởi mã do trình biên dịch tạo ra và sau đó gọi hàm kết thúc.

Chú ý rằng chúng ta đưa ra một ngoại lệ chứa một chuỗi ký tự, nhưng đón bắt chỉ các ngoại lệ có kiểu ExceptionClass. Hai điều này sẽ không khớp nhau - nghĩa là mã để đón bắt ngoại lệ sẽ không được bỏ qua.

3. Lưu mã nguồn trong code editor rồi đóng ứng dụng editor.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành mà bạn ưa thích.

5. Chạy ứng dụng trong cửa sổ console.

Nếu bạn làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng:

```
$ ./a.exe 1 2 3
You selected the second option
Properly processed option 2
Exception reported in file ch6_9.cpp at
line 138
Invalid Option
Exception reported in file ch6_9.cpp at
line 138
Invalid Option
term_func() was called by terminate().
```

Chú ý term_func đã được gọi bởi vì một ngoại lệ không xử lý đã được tạo bởi mã và không bao giờ được đón bắt bởi mã ứng dụng. Nếu chúng ta đã không cài đặt hàm kết thúc, chương trình đơn giản thoát và chúng ta không bao giờ thấy lệnh gọi làm term_func trong kết quả.

Đưa ra lại các ngoại lệ

Một trong những điều phiền toái nhất về việc xử lý lỗi truyền thống trong C và C++ là nó buộc bạn phải mất đi nhiều thông tin cấp thấp hơn. Ví dụ, nếu bạn gọi một hàm để đọc một record vốn lần lượt gọi một hàm để di chuyển đến record trong file mà lần lượt gọi một hàm để tìm kiếm đến offset trong file (vốn gây ra một lỗi), một số thông tin cấp thấp hơn có thể hữu dụng (việc offset không hợp lệ) bị mất tại cấp trên cùng. Tất cả những gì bạn biết là hàm để đọc record thất bại và có thể nó thất bại trong thường trình move. Bạn vẫn phải làm việc với một trình gỡ rối và bước vào chức năng cấp thấp nhất để xem điều gì thật sự đã xảy ra trong mã. Chúng ta có thể giải quyết điều này bằng cách móc nối các lỗi từ cấp thấp nhất đến cấp cao nhất của ứng dụng. Hiệu ứng móc nội này được hoàn thành bằng việc được đưa ra lại các ngoại lệ.

Tuy nhiên, với việc xử lý ngoại lệ, bạn có thể chuyển thông tin lên trên những dây chuyền (chain) sao cho hàm cấp cao nhất có thể báo cáo tất cả dữ liệu trong một lỗi nào đó từ cấp dưới cùng lên trên cùng.

Để chuyển thông tin từ một cấp thấp hơn của ứng dụng lên một cấp cao hơn, bạn phải đón bắt ngoại lệ, thêm vào chúng và đưa ra lại chúng. Danh sách sau đây cho bạn thấy chính xác cách làm điều đó từ việc tạo ngoại lệ ban đầu đến việc đón bắt nó và thêm vào nó để chuyển nó đến một cấp cao hơn.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch53.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 53.3 vào file.

Listing 53.3: Chuyển các ngoại lệ đến một cấp cao hơn.

```
int read_file( long offset )
    throw(string)
{
    if ( offset < 0 || offset >= 100 )
       throw string("read_file: Invalid offset");                    → 5
    return 0;
}
int read_record( long record )
       throw(string)
{
       if ( record < 0 || record > 10 )
          throw string("read_record: invalid record number");
       long offset = record * 10;
       try
       {
          read_file( offset );
       }
       catch ( string msg )
       {
          string sMsg = "record_record: unable to go to offset\n"; → 6
          sMsg += msg;
          throw sMsg;
       }
       return 0;
}
int func3(long recno)
{
    try
    {
       read_record( recno );
```

```
        }
        catch ( string s )
        {
            cout << "func 3: Error in read:" << endl;
            cout << s.c_str() << endl;                          → 7
        }
        catch ( ... )
        {
            cout << "func 3: Unknown error in read:" << endl;
        }
        cout << "End of func3\n";
    }
```

Trong ví dụ này, đầu tiên chúng ta đón bắt một lỗi tại cấp thấp nhất, hàm read_file và tạo một ngoại lệ vốn được đưa ra cho hàm read_record, như được minh hoạ tại → **5**. Sau đó lỗi này được đón bắt trong hàm read_record nhưng việc hàm read_record thất bại trong việc cố đọc dữ liệu được thêm vào và lỗi sau đó được đưa ra như được minh hoạ tại → **6**. Sau đó lỗi được đón bắt tại một cấp cao hơn, trong hàm func3 (như được minh hoạ tại → **7**) và kết quả hiển thị cho người dùng.

3. Trong hàm main, chỉnh sửa mã để gọi hàm new, thêm một lệnh gọi đến func3 bất cứ nơi nào bạn muốn trong mã như sau:

```
//func2();
cout << "Func3 [1]" << endl;
func3(2);
cout << "Func3 [2]" << endl;
```

Chú ý rằng lệnh gọi func2 đã được chú giải bởi vì nó thoát chương trình. Điều này dễ bỏ qua; nếu chương trình không bao giờ gặp phải các lệnh gọi func3, có lẽ là do bạn quên chú giải dòng này.

4. Lưu mã nguồn dưới dạng một file trong code editor và sau đó đóng ứng dụng biên soạn.

5. Biên dịch ứng dụng bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

6. Chạy ứng dụng trong cửa sổ console.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả từ ứng dụng như được trình bày trong Listing 53.4.

Listing 53.4: Kết quả cập nhật từ chương trình xử lý ngoại lệ.

```
$ ./a.exe 1 2 3
You selected the second option
Properly processed option 2
Exception reported in file ch6_9.cpp at line 138
Invalid Option
Exception reported in file ch6_9.cpp at line 138
Invalid Option
Func3 [1]
End of func3
Func3 [2]
func 3: Error in read:                                  → 8
record_record: unable to go to offset
read_file: Invalid offset
End of func3
Func3 [3]
func 3: Error in read:
read_record: invalid record number
End of func3
```

Như bạn có thể thấy từ dòng được đánh dấu là → **8** trong listing kết quả ở trên, hàm func3 tạo toàn bộ chuỗi lỗi thay vì một ký hiệu đơn giản cho biết rằng một lỗi đã xảy ra. Từ đây, chúng ta có thể thấy việc xem toàn bộ lịch sử của những sự cố gì đã xảy ra thay vì đơn giản rằng một lỗi đã xảy ra thì hữu dụng hơn nhiều như thế nào.

Kỹ thuật 54: Thi hành các mã trả về

Tiết kiệm thời gian bằng cách:

- Tìm hiểu những giới hạn của các mã trả về để xử lý các lỗi
- Các mã trả về với việc xử lý ngoại lệ để đón bắt các lỗi
- Hiểu kết quả

Xử lý các lỗi là lý do lớn nhất cho sự cố chương trình - đó là những gì tạo ra nhu cầu gỡ rối và bảo trì. Nếu bạn loại bỏ nguồn của các sự cố, bạn sẽ tự cho mình thêm thời gian để phát triển các lớp ứng dụng tốt hơn và những tính năng tốt hơn cho những người dùng của bạn. Kỹ thuật này sẽ hướng dẫn bạn cách kết hợp các mã trả về với việc xử lý ngoại lệ để tránh những sự cố này.

Trong C++, các phương thức và hàm có thể trả về một mã trạng thái (status code) vốn cho biết hàm đã thành công, thất bại hay được để lại trong một trạng thái ở giữa nào đó. Ví dụ, chúng ta có một hàm vốn trả về một mã trạng thái biểu thị phương thức nằm ở đâu trong việc xử lý dữ liệu. Mã trạng thái được trả về có thể được trông tương tự như sau:

```
int get_status(void)
{
    switch ( current_status )
    {
       case NotProcessing:
            return –1;
       case InProcessing:
            return 1;
       case ProcessingComplete:
            return 0;
    }
    return –99;
}
```

Bây giờ trong ví dụ này nếu hàm trả về một giá trị là -99, rõ ràng một điều gì đó rất tệ đang xảy ra - bởi vì chúng ta không biết trạng thái nào mà đối tượng có thể đạt đến. Nếu mã trạng thái là -99 thì chúng ta thường muốn ngừng xử lý ngay tức thì - và có thể thoát chương trình. Thật không may, không có cách nào để nhà phát triển hàm buộc người dùng sử dụng hàm kiểm tra mã trạng thái trả về để bảo đảm rằng họ biết một trục trặc nào đó đã xảy ra. Sẽ thú vị hay không nếu có một cách nhằm bảo đảm nhà lập trình đã xem mã trả về để thấy nó có không hợp lệ hay không.

Hoá ra bạn có thể bảo đảm rằng nhà phát triển đã kiểm tra các mã trả về. Bằng cách sử dụng các bước sau đây, bạn có thể buộc mã trả về được kiểm tra; nếu mã không được kiểm tra, bạn có thể đưa ra một ngoại lệ. Đây thật sự vẹn cả đôi đường. Xử lý ngoại lệ chỉ là một cách rất chắc chắn để buộc nhà phát triển xử lý các lỗi, nhưng nó cũng có một hao phí lớn trong tốc độ xử lý và việc sử dụng CPU. Mặt khác, các mã trả về có hao phí thấp, nhưng bạn không thật sự chắc chắn rằng nhà phát triển sẽ từng xem chúng. Bằng cách kết hợp hai phương pháp này, bạn có thể hoàn toàn chắc chắn rằng các lỗi được xử lý nhằm loại bỏ hầu hết các sự cố thời gian chạy (run-time) vốn xuất hiện đối với những người dùng cuối. Có một số hao phí liên quan ở đây do việc bổ sung khả năng xử lý ngoại lệ, nhưng hao phí đó được giảm đi bởi việc các ngoại lệ sẽ không được đưa ra nếu lỗi được kiểm tra đúng cách.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch54.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã nguồn cho các lớp.

2. Gõ nhập mã từ Listing 54.1 vào file.

Listing 54.1: Lớp mã trả về.

```
#include <iostream>
#include <string>
using namespace std;
template < class T >
class RetValue
{
    T _value;                                          → 1
    bool _checked;                                     → 2
public:
    RetValue( void )
    {
       _checked = false;
    }
    RetValue( const T& t )
    {
       _value = t;
       _checked = false;
    }
    RetValue( const RetValue& aCopy )
    {
       _value = aCopy._value;
       _checked = false;
    }
    virtual ~RetValue(void)
    {
       if ( !_checked )
          throw "Error: Return value not checked!!";   → 3
}
bool operator==( const T& t)                           → 7
{
```

```
        _checked = true;
        return t == _value;
    }
    bool operator!=( const T& t)
    {
        _checked = true;
        return t != _value;
    }
    bool operator <( const T& t)
    {
        _checked = true;
        return _value < t;
    }
    bool operator <=( const T& t)
    {
        _checked = true;
        return _value <= t;
    }
    bool operator >( const T& t)
    {
        _checked = true;
        return _value > t;
    }
    bool operator >=( const T& t)
    {
        _checked = true;
        return _value >= t;
    }
    operator T()
    {
        _checked = true;
        return _value;
    }
    bool operator!()
    {
        _checked = true;
        return !_value;
    }
```

```
T operator&( const T& t )
{
    _checked = true;
    return _value & t;
}
T operator|( const T& t )
{
    _checked = true;
    _value | t;
}
bool IsChecked()
    {
        return _checked;
    }
    T& Value()
    {
        return _value;
    }
};
RetValue<int> func( int iValue )
{
    if ( iValue == 34 )
        return RetValue<int>(1);
    if ( iValue == 35 )
        return RetValue<int>(2);
    return RetValue<int>(0);
}
RetValue<int> func2(int iValue)
{
    RetValue<int> ret = func(iValue);
    if ( ret )
        return ret;
    return RetValue<int>(0);
}
class MyReturnValue
{
    string _message;
public:
```

```
        MyReturnValue( void )
        {
            _message = "";
        }
        MyReturnValue( const char *msg )
        {
            _message = msg;
        }
        MyReturnValue( const MyReturnValue& aCopy )
        {
            _message = aCopy._message;
        }
        MyReturnValue operator=( const MyReturnValue& aCopy )
        {
            _message = aCopy._message;
            return *this;
        }
        string Message(void)
        {
            return _message;
        }
        string operator=( const string& msg )
        {
            _message = msg;
            return _message;
        }
        bool operator==( const MyReturnValue& aValue )
        {
            return _message == aValue._message;
        }
        bool operator<( const MyReturnValue& aValue )
        {
            return _message < aValue._message;
        }
};
int main()
{
    try
```

```
    {
        if ( !func( 34 ))                                    → 4
            printf("Success!!\n");
        if ( func(35) & 2 )                                  → 5
            printf("Error 35\n");
        RetValue<int> t1 = 5;
        int x = 5;
        int y = 3;
        printf("5 == 5? %s\n", t1 == x ? "Yes" : "No" );
        printf("5 == 3? %s\n", t1 == y ? "Yes" : "No" );
        int iVal = t1;
        printf("Calling func2\n");
        func2(34);                                           → 6
    }
    catch ( ... )
    {
        printf("Exception!\n");
    }
    try
    {
        RetValue<MyReturnValue> rv1 = MyReturnValue("This is a test");
        MyReturnValue rv = rv1;
        string s = rv.Message();
        printf("Return Value: %s\n", s.c_str() );
    }
    catch ( ... )
    {
        printf("Exception in MyReturnValue\n");
    }
    return 0;
}
```

Lớp cơ sở ở đây, RetValue thực thi một lớp mã trả về khuôn mẫu vốn có thể chấp nhận bất kỳ loại dữ liệu để sử dụng làm mã trả về "thực" cho các hàm và phương thức. Mã này được lưu trữ trong lớp dưới dạng một biến thành viên như được minh hoạ tại → **1**. Bên dưới là một biến thành viên khác được gọi là checked, được sử dụng để xem mã trả về đã từng được so sánh với bất cứ thứ gì hay không (Xem → **2**). Người dùng có thể kích khởi điều việc so sánh giá trị của mã

trả về sử dụng bất kỳ toán tử boolean chuẩn (chẳng hạn như bằng, không bằng, lớn hơn, nhỏ hơn và...). Sau khi tất cả toán tử này được sử dụng, mã trả về biết rằng nhà phát triển sử dụng mã trả về đã kiểm tra nó bằng một cách nào đó và cho phép nhà lập trình tiếp tục. Nếu đối tượng mã trả về nằm ngoài phạm vi mà lỗi không được kiểm tra, một ngoại lệ được đưa ra như được minh hoạ tại → **3**. Bởi vì lớp là lớp khuôn mẫu, bạn có thể lưu trữ bất cứ thứ gì bạn muốn trong dữ liệu thành viên lớp. Chúng ta minh hoạ điều này bằng cách tạo lớp riêng của chúng ta được gọi là MyReturnValue và trả nó về từ một hàm.

3. Lưu mã nguồn trong code editor và sau đó đóng ứng dụng biên soạn.
4. Biên dịch ứng dụng sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trong cửa sổ console.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây từ ứng dụng:

```
$ ./a.exe
Error 35
5 == 5? Yes
5 == 3? No
Calling func2
Exception!
Return Value: This is a test
```

Kết quả từ chương trình test nhỏ này cho thấy khi chúng ta kiểm tra một lỗi chẳng hạn như phép so sánh toán tử not (!) tại → **4** hoặc toán tử lôgíc and (&) tại → **5**, không có ngoại lệ được tạo ra bởi mã. Tuy nhiên, nếu một hàm vốn trả về một đối tượng khuôn mẫu RetValue chẳng hạn như func 2 được gọi như được minh hoạ tại → **6** và giá trị trả về không được kiểm tra, sẽ có một ngoại lệ được đưa ra.

Như bạn có thể thấy, nếu người dùng không chọn kiểm tra một giá trị trả về destructor cho lớp đưa ra một ngoại lệ khi đối tượng nằm ngoài phạm vi. Điều này buộc nhà phát triển ứng dụng xử lý vấn đề bằng cách này hoặc cách kia. Nếu nhà phát triển không xử lý ngoại lệ, nó kết thúc chương trình. Nếu nhà phát triển xử lý ngoại lệ, người này sẽ nhận ra ngay mã trả về đã không được xử lý ở đâu.

Điều đặc biệt tiện lợi về kỹ thuật này là nó cũng minh hoạ cách bạn có thể ghi đè hầu như mọi phép so sánh có thể có cho một đối tượng (chẳng hạn như operator== được minh hoạ tại → **7**). Bằng cách kiểm tra những phép tính khác nhau, chúng ta biết người dùng đã làm một điều gì đó như sau hay không:

```
if ( method_with_return_code ( ) ==
BadReturn)
{
}
```

thay vì một điều gì đó như sau:

```
int myRet = method_with_return_code ( );
```

Trong trường hợp thứ nhất, người dùng thật sự kiểm tra xem giá trị có bằng với một giá trị nào đó hay không. Trong trường hợp thứ hai, người dùng gán giá trị vào một biến khác vốn có thể không bao giờ được kiểm tra. Bằng cách xem cách người dùng đã truy cập giá trị trả về như thế nào, chúng ta có thể biết họ có thật sự kiểm tra mã trả về hay không. Đây là nơi các toán tử quá tải được áp dụng. Chúng ta xác lập cờ checked trong toán tử quá tải và do đó chúng ta biết kết quả có thể thật sự được xem hay không.

Ngoài ra, bạn phải quan tâm đến những thứ như chuyển các mã trả về lên chain (dây chuyền) từ các phương thức cấp thấp đến các phương thức cấp cao hơn. Nếu người dùng tạo một bản sao của đối tượng để thêm một kết quả hoặc kiểm tra kết quả hiện hành, chúng ta muốn biết về điều đó. Sau đó người dùng có thể chuyển một bản sao của đối tượng đến một thường trình gọi cấp cao hơn. Constructor copy cho lớp hơi khác với những constructor copy khác mà có thể ban đã thấy hoặc đã viết mã; nó đơn giản gán tất cả biến thành viên để giống như đối tượng mà nó sao chép. Thay vào đó nó sao chép giá trị của mã trả về và sau đó bảo đảm rằng cờ (flag) biểu thị rằng giá trị trả ề đã được kiểm tra được xác lập sang trạng thái unchecked (không kiểm tra), bởi vì nếu không người dùng có thể đơn giản sao chép đối tượng vào một mã trả về khác mà không bao giờ xem giá trị trạng thái "thực".

Kỹ thuật 55: Sử dụng các ký tự đại diện (wildcard)

Tiết kiệm thời gian bằng cách:

- Sử dụng các ký tự đại diện (wildcard)
- Thực thi một lớp vốn sử dụng các ký tự đại diện
- Test lớp wildcard

Nếu bạn đã bạn đã từng tìm kiếm các file trên một máy tính, có lẽ bạn đã sử dụng các ký tự đại diện (wildcard). Theo nghĩa này, wildcard là những ký tự được sử dụng trong các chuỗi tìm kiếm vốn không đại diện cho chính chúng, nhưng cho một dãy rộng các ký tự. Ý tưởng tìm

tất cả file tương hợp với một mẫu nào đó khá phổ biến trong thế giới máy tính. Ý tưởng tìm kiếm các file để tìm các chuỗi tương hợp các mẫu ký tự đại diện cũng vậy. Ví dụ, nếu bạn phải tìm một file nhưng không hoàn toàn có thể nhớ lại tên của file đó - tất cả những gì bạn nhớ là nó bắt đầu với từ convert hoặc conversion hoặc từ nào đó tương tự - sử dụng một ký tự đại diện là một giải pháp tuyệt vời. Tìm kiếm từ convert sẽ chỉ đưa ra các file vốn bắt đầu với từ cụ thể đó. Mặt khác tìm conv* sẽ cho bạn một sự lựa chọn các file rộng hơn nhiều. Ký tự đại diện dấu sao (*) tượng trưng cho bất kỳ nhóm zero hoặc nhiều ký tự. Điều này có nghĩa là danh sách vừa có được từ việc tìm kiếm bao gồm các file bắt đầu với conv và sau đó kết thúc trong bất kỳ nhóm zero hoặc nhiều ký tự chẳng hạn như

Convert
Conversion
Conversation

Bởi vì chúng tương hợp với zero hoặc nhiều ký tự, các dấu sao là những ký tự đại diện hữu dụng, nhưng chúng vẫn giới hạn. Sử dụng dấu sao mẫu A*B tương hợp AB, AbasdhB, và AbB. Nó không tương hợp ABC cũng không tương hợp AajhaBajksjB.

Các ký tự đại diện tượng trưng cho một khả năng mạnh mẽ nhằm tìm kiếm tất cả các từ tương hợp một gốc nào đó. Ngay cả tốt hơn, các ký tự đại diện cho phép bạn tương hợp các từ khi bạn không hoàn toàn chắc chắn về cách viết. Ví dụ, nếu bạn tìm từ conscious, nhưng không thể nhớ cách viết nó thì phải làm sao - nó có một s ở giữa hay không? Các ký tự đại diện cho phép bạn tìm kiếm từ này bằng mọi giá; bạn chỉ việc tìm con?cious. Ký tự đại diện dấu hỏi (?) tượng trưng cho bất kỳ ký tự đơn (hoặc không có gì cả); mẫu A?B tương hợp cả AB và AbB. Do đó, biểu thức con?cious tương hợp với từ conscious cho dù nó có chứa một s ở vị trí đó hay không.

Các ký tự dấu hỏi và dấu sao là những ký tự đại diện phổ biến - nhưng không phải là những ký tự đại diện duy nhất. Ví dụ, trong ngôn ngữ SQL, sử dụng một dấu phần trăm (%) thay vì một dấu sao để tương hợp nhiều ký tự. Vì lý do này, khi bạn thiết kế một lớp vốn cho phép các ký tự đại diện trong các chuỗi tìm kiếm, bạn nên cho thông tin đó có thể cấu hình. Mục đích của kỹ thuật này là hướng dẫn bạn cách tạo một lớp thực hiện việc tương hợp bằng các ký tự đại diện. Lớp này có thể được sử dụng để thêm chức năng tương hợp mẫu vào ứng dụng một cách nhanh chóng và dễ dàng nhằm tiết kiệm cho bạn thời gian và công sức trong việc phát triển phần mềm chất lượng mà người dùng thật sự muốn.

Listing 55.1: Lớp Match

```
#include <iostream>
#include <string>
using namespace std;
class Match
{
private:
    char _MatchMultiple;
    char _MatchSingle;
    string _pattern;
    string _candidate;
protected:
    bool match(const char *pat, const char *str)
    {
        if ( *pat == '\0' )
            return !*str;
        else
            if ( *pat == _MatchMultiple )
                return match(pat+1, str) || (*str && match(pat, str+1));
        else
            if ( *pat == _MatchSingle )
                return *str && (match(pat+1, str+1) || match(pat+1, str));
        return (*str == *pat) && match(pat+1, str+1);
    }
public:
Match(void)
{
    _MatchMultiple = '*';
    _MatchSingle = '?';
}
Match( const char *pat, const char *str )
{
    _MatchMultiple = '*';
    _MatchSingle = '?';
    _pattern = pat;
    _candidate = str;
}
```

```
Match( const Match& aCopy )
{
    _MatchMultiple = aCopy._MatchMultiple;
    _MatchSingle = aCopy._MatchSingle;
    _pattern = aCopy._pattern;
    _candidate = aCopy._candidate;
}
Match operator=( const Match& aCopy )
{
    _MatchMultiple = aCopy._MatchMultiple;
    _MatchSingle = aCopy._MatchSingle;
    _pattern = aCopy._pattern;
    _candidate = aCopy._candidate;
    return *this;
}
char Multiple(void)
{
    return _MatchMultiple;
}
char Single(void)
{
    return _MatchSingle;
    }
    void setMultiple( char mult )
    {
        _MatchMultiple = mult;
    }
    void setSingle( char single )
    {
        _MatchSingle = single;
    }
    void setPattern( const char *pattern )
    {
       _pattern = pattern;
    }
    void setCandidate( const char *candidate )
    {
       _candidate = candidate;
```

```
    }
    string getPattern( void )
    {
        return _pattern;
    }
    string getCandidate( void )
    {
        return _candidate;
    }
    bool matches()
    {
        return match( _pattern.c_str(), _candidate.c_str() );
    }
};
```

Tạo lớp tương hợp ký tự đại diện

Để tận dụng tốt nhất tính năng tương hợp ký tự đại diện trong ứng dụng, bạn nên đóng gói chức năng cho các chuỗi tương hợp thành một lớp đơn. Lớp đó sẽ xử lý cả hai công việc lưu trữ các ký tự tương hợp (chẳng hạn như một dấu sao hoặc dấu hỏi) và quyết định xem hai chuỗi có khớp với nhau hay không. Hãy phát triển một lớp như vậy và một driver thử nghiệm để minh hoạ nó được sử dụng như thế nào. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch55.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 55.1 vào file.

Mục đích của lớp này là xem hai chuỗi có khớp với nhau hay không kể cả các ký tự đại diện nếu cần thiết. Để làm điều này chúng ta cần:

- Một wildcard nhiều ký tự
- Một wildcard ký tự
- Một chuỗi mẫu nhập liệu
- Chuỗi tương hợp ứng cử

Ví dụ, nếu chúng ta muốn cho phép người dùng tương hợp chuỗi Colour cũng như Color sao cho chúng ta có thể kiểm tra tìm các cách viết của người Anh, chúng ta có thể sử dụng:

- **Wildcard nhiều ký tự:** Một dấu sao (*)
- **Wildcard một ký tự:** Một dấu hỏi (?)

- **Chuỗi tương hợp nhập liệu:** Colo*r
- **Chuỗi tương hợp ứng cử:** Color hoặc Colour

Kết quả của điều này là một sự tương hợp chắc chắn. Để làm điều này, chúng ta tạo một lớp chứa các biến thành viên cho các ký tự và chuỗi tương hợp và các thường trình để truy cập các phần tử tương hợp đó. Ngoài ra, lớp chứa một phương thức được gọi là matches, biểu thị xem các chuỗi nhập liệu và chuỗi ứng cử có khớp với nhau hay không.

3. Lưu mã nguồn trong code editor.

Test lớp tương hợp wildcard

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã như thế nào.

Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm để minh hoạ các loại đầu vào khác nhau từ người dùng và xem lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch6_12.cpp.

2. Gõ nhập mà từ Listing 55.2 vào file.

Listing 55.2: Driver thử nghiệm tương hợp wildcard.

```
string get_a_line( istream& in )
{
    string retStr;
    while ( !in.fail() )
    {
        char c;
        in.get(c);
        if ( in.fail() )
            break;
        if ( c != '\r' && c != '\n' )
            retStr += c;
        if ( c == '\n' )
            break;
    }
    return retStr;
```

```
}
int main(int argc, char **argv)
{
    char szPattern[ 80 ];
    char szString [ 80 ];
    bool done = false;
    while ( !done )
    {
        cout << "Enter the pattern: ";
        string sPattern = get_a_line( cin );
        if ( !sPattern.length() )
            done = true;
        else
        {
            cout << "Enter the string: ";
            string sString = get_a_line( cin
    );
            Match m(sPattern.c_str(),
            sString.c_str() );
            if ( m.matches() )
                printf("match\n");
            else
                printf("no match\n");
        }
    }
}
```

Driver thử nghiệm nhận được hai chuỗi từ người dùng và sử dụng việc tương hợp wildcard để xem chúng có khớp với nhau hay không. Chuỗi mẫu có thể chứa các wildcard tuỳ chọn mặc dù chuỗi cần tương hợp có thể không. Bằng cách tận dụng lớp Match mà chúng ta phát triển trong Listing 55.1, chúng ta kiểm tra xem hai chuỗi có phải là các ký tự wildcard tương hợp với nhau hay không.

3. Lưu mã nguồn trong bộ soạn thảo và đóng ứng dụng soạn thảo.
4. Biên dịch ứng dụng, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ console:

```
$ ./a.exe
Enter the pattern: A*B
Enter the string: AB
match
Enter the pattern: A*B
Enter the string: AajkjB
match
Enter the pattern: A*B
Enter the string: ABC
no match
Enter the pattern: A?B
Enter the string: AbaB
no match
Enter the pattern:
```

Như bạn có thể thấy, lớp tương hợp làm việc như đã nêu.

Phần VIII
Các tiện ích

Kỹ thuật 56: Mã hoá và giải mã dữ liệu cho Web

Tiết kiệm thời gian bằng cách:

- Tạo giao diện với Internet
- Mã hoá và giải mã các URL để sử dụng trên Internet
- Tạo một lớp URL Codec
- Test lớp đó

World Wide Web đã mang lại vô số cơ hội mới và vô số vấn đề mới. Hầu hết ứng dụng ngày nay cần có chức năng Web để hoạt động trực tiếp với các trình duyệt hoặc ứng dụng Web. Bất kể bạn phát triển loại ứng dụng nào, khả năng ứng dụng sẽ phải tương tác với Web hoặc với hệ thống từ xa sử dụng những giao thức Web.

Vấn đề lớn nhất trong việc tạo giao diện với Internet là mã hoá (encoding). Mã hoá là tiến trình biên dịch các ký tự vốn không thể sử dụng trực tiếp bởi một hệ thống thành các ký tự mà hệ thống có thể sử dụng được. Ví dụ, đối với World Wide Web, các ký tự chẳng hạn như dấu ampersand (&), dấu lớn hơn và nhỏ hơn (> và <) và những dấu khác không thể được sử dụng trực tiếp. Chúng ta cần thay đổi thành một dạng mà Web có thể sử dụng. Web nhận dạng các địa chỉ bằng một Uniform Resource Locator còn được gọi là URL. Một trong những qui tắc làm việc với các URL là chúng không chứa các ký tự chẳng hạn như các khoảng trống và dấu gạch chéo, bởi vì việc đưa chúng vào sẽ phá vỡ nhiều ứng dụng trình duyệt và hệ điều hành hiện có. Các trình duyệt (Browser) giả định rằng các khoảng trống và dấu gạch chéo biểu thị các ngắt trong một URL vốn là định dạng chuẩn cho các địa chỉ Web. Không có cách nào để trình duyệt, do đó chúng ta phải thay đổi chúng.

Vấn đề là thư viện C++ không cung cấp các chuẩn để mã hoá và giải mã các chuỗi URL. Kỹ thuật mã hoá và giải mã rất phổ biến nhưng nó đủ mới đến nỗi không được đưa vào STL hoặc thư viện C++ chuẩn. Vì lý do này, cuối cùng chúng ta phải tái thực thi mã trong mỗi và mọi ứng dụng mà chúng ta viết. Điều này trái với nguyên tắc của C++ là "viết một lần, tái sử dụng nhiều lần".

Tiết kiệm thời gian thường là nói về việc dự đoán trước những nhu cầu của ứng dụng và hoạch định trước cho chúng. Bằng việc hoạch định Web-enable cho mã - bất kể bạn có mong đợi ứng dụng hỗ trợ Web (tối thiểu ban đầu hay không) - cuối cùng bạn tiết kiệm nhiều thời gian. Sau đó, tạo một lớp đơn có thể tái sử dụng để làm công việc mã hoá và giải mã, một việc mà bạn có thể chèn khi cần thiết trong ứng dụng bạn phát triển sẽ là điều hợp lý. Đó là nội dung mà kỹ thuật này nói đến.

Tạo lớp URL Codec

Trong thế giới, "codec" là một compressor / decompressor (bộ nén / giải nén) thường được sử dụng để nén các định dạng audio hoặc video thành một kích cỡ nhỏ hơn. Tuy nhiên, khái niệm rất có thể áp dụng cho những gì chúng ta thực hiện với text bởi vì chúng ta làm việc với các luồng dữ liệu tương tự các định dạng video và audio. Đối với những mục đích của kỹ thuật này, chúng ta sẽ tạo một lớp đơn giản hiểu cách mã hoá một chuỗi sao cho nó có thể được sử dụng với các trình duyệt Web hiện có. Mỗi ký tự trong chuỗi sẽ được kiểm tra và nếu nó không có một định dạng hợp lệ Web, sẽ được mã hoá để sử dụng đúng cú pháp. Sau đây là cách thực hiện:

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch56.pp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã trong Listing 56.1 vào file.

Listing 56.1: Mã hoá và giải mã dữ liệu

```
#include <string>
#include <iostream>
using namespace std;
class URLCodec
{
    string _url;
protected:
    // Convert a hex string to an ASCII representation.
```

```
        char htoa (int number)
    {
        if ((number >= 0) && (number <= 9))
            return ('0' + number);
        else if ((number >= 10) && (number
        <= 15))
            return ('A' + number - 10);
        else
            return ('X');
    }
    // Convert an ASCII string into a hex
        digit.
    char atoh (unsigned char character)
    {
        if ((character >= '0') && (character
        <= '9'))
            return (character - '0');
        else if ((character >= 'A') &&
            (character <= 'F'))
                return (character - 'A' + 10);
        else if ((character >= 'a') &&
            (character <= 'f'))
                return (character - 'a' + 10);
        else
                return (0);
    }
public:
    URLCodec( void )
    {
        _url = "";
    }
    URLCodec( const char *strIn )
    {
        _url = strIn;
    }
    URLCodec( const URLCodec& aCopy )
    {
        _url = aCopy._url;
```

```
}
URLCodec operator=( const URLCodec&
aCopy )
{
    _url = aCopy._url;
    return *this;
}
void setURL ( const char *strIn )
{
    _url = strIn;
}
void setURL ( const string& sIn )
{
    _url = sIn;
}
string getURL ( void )
{
    return _url;
}
string encode()                                    → 1
{
    int index;
    string encoded;
    // Make a copy of the string.
    encoded = _url;
    // Scan the input string backward.
    index = encoded.length();
    while (index—)
    {
        // Check for special characters.
        if (!isalnum((unsigned
        char)encoded[index]))                      → 2
        {
            unsigned char special;
            char insert;
            special = (unsigned char)
                encoded[index];
            encoded.erase (index, 1);
```

```
                insert = htoa (special %
                    16);
                encoded.insert (index,
                    &insert, 1);
                insert = htoa (special /
                    16);
                encoded.insert (index,
                    &insert, 1);
                insert = '%';
                encoded.insert (index,
                    &insert, 1);
            }
        }
        return (encoded);
}
string encode_no_xml()                                    → 3
{
        int index;
        string encoded;

        // Make a copy of the string.
        encoded = _url;
        // Scan the input string backward.
        index = encoded.length();
        while (index—)
        {
            // Check for special characters.
            if ((!isalnum((unsigned
            char)encoded[index])) &&
                (encoded[index] != ' ') &&
                (encoded[index] != '<') &&
                (encoded[index] != '>') &&
                (encoded[index] != '_') &&
                (encoded[index] != '\n') &&
                (encoded[index] != '/') &&
                (encoded[index] != '"') &&
                (encoded[index] != '\'))
            {
```

```
            unsigned char special;
            char insert;
            special = (unsigned char)
               encoded[index];
            encoded.erase (index, 1);
            insert = htoa (special %
               16);
            encoded.insert (index,
               &insert, 1);
            insert = htoa (special /
               16);
            encoded.insert (index,
               &insert, 1);
            insert = '%';
            encoded.insert (index,
               &insert, 1);
         }
      }
      return (encoded);
}
string decode()                                    → 4
{
      int index;
      string decoded;
      // Make a copy of the string.
      decoded = _url;
         // Scan input string forwards
         index = 0;
         while (index < decoded.length())
         {
            // Check for encoded characters.
            if (decoded[index] == '%')
            {
               unsigned char special;
               special = (unsigned char)
               atoh(decoded[index+1]) *
               16;
               special += (unsigned char)
```

```
                    atoh(decoded[index+2]);
                    decoded.erase (index, 3);
                    decoded.insert (index, (char
                        *)&special, 1);
                }
                index++;
            }
            return (decoded);
        }
};
```

Lớp này sẽ xử lý việc mã hoá và giải mã các URL cũng như lưu trữ một chuỗi URL chung chung. Mỗi ký tự trong chuỗi được kiểm tra, bắt đầu ở phía sau chuỗi và làm việc ngược lại để chúng ta có thể hiểu đúng các ký tự khi cần thiết.

Có hai dạng phương thức encode được trình bày ở đây:

- Thứ nhất, được minh hoạ tại dòng được đánh dấu là → **1**, mã hoá tất cả ký tự trên chuỗi theo định dạng URL chuẩn. Điều này được thực hiện tại vòng lặp được minh hoạ bởi dòng được đánh dấu → **2**. Mỗi ký tự được kiểm tra để xem nó có thứ tự chữ - số hợp không và nếu không hợp lệ nó được thay thế bởi thứ tự thập lục phân tương đương.
- Thứ hai, được minh hoạ tại dòng được đánh dấu là → **3** thực hiện điều tương tự, nhưng nó không mã hoá các ký tự XML mà một số ứng dụng cho Web sẽ cần.

Nếu bạn làm việc với các URL chuẩn cho Web, sử dụng phiên bản mới nhất. Nếu bạn làm việc với các applet Java hoặc ứng dụng .Net chạy trên Web mong đợi các ký tự XML hợp lệ, hãy sử dụng phiên bản thứ hai. Trong bất kỳ trường hợp, bạn có thể sử dụng phương thức decode được minh hoạ tại → **4** để giải mã các ký tự thành một chuỗi mà con người đọc được.

3. Lưu mã nguồn trong code editor.

Test lớp URL Codec

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để không chỉ bảo đảm rằng mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Các bước sau đây trình bày cho bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và trình bày cách lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch56.cpp.

2. Gõ nhập mã từ Listing 56.2 vào file.

Listing 56.2: Driver thử nghiệm URL Codec

```
int main(int argc, char **argv)
{
    if ( argc < 2 )
    {
        cout << "Usage: ch7_1 url1 [url2
            url3]" << endl;
        cout << "Where: url[n] is the url
            you wish to see encoded/decoded"
            << endl;
        return -1;
    }
    for ( int i=1; i<argc; ++i )
    {
        URLCodec url( argv[i] );
        // First, try decoding it.
        string enc = url.encode();                    → 5
        string dec = url.decode();
        cout << "Input String:" << argv[i]
        << endl;
        cout << "Encoded:" << enc.c_str()
        << endl;
        cout << "Decoded:" << dec.c_str()
        << endl;
        // Now try decoding the result.
        URLCodec enc_url( enc.c_str() );
        string enc1 = url.encode();
        string dec1 = url.decode();
        cout << "Input String:" <<
        enc_url.getURL().c_str() << endl;
        cout << "Encoded:" << enc1.c_str()
            << endl;
```

```
            cout << "Decoded:" << dec1.c_str()
               << endl;
        }
        return 0;
    }
```

Mã driver thử nghiệm ở trên cho phép bạn test chức năng của các phương thức encode và decode của lớp URL Codec. Nếu bạn nhập một chuỗi từ dòng lệnh đến ứng dụng, nó sẽ in ra các phiên bản được mã hoá và được giải mã của chuỗi đó. Không có điều gì thật sự kỳ diệu về ứng dụng này. Như bạn có thể thấy từ listing, trước tiên mã cố mã hoá chuỗi mà bạn cho nó (được minh hoạ tại → **5**) và sau đó giải mã kết quả mã hoá đó để xem chúng có giống nhau hay không. Sau đó khối mã thứ hai mã hoá kết quả và giải mã nó để bảo đảm rằng mã làm việc tốt. Khi tất cả được hoàn tất, bạn sẽ thấy cùng một dữ liệu nhập và dữ liệu xuất trên console.

3. Lưu file mã nguồn trong bộ soạn thảo và đóng ứng dụng soạn thảo.

4. Biên dịch file nguồn bằng trình biên dịch ưa thích, trên hệ điều hành ưa thích.

5. Chạy ứng dụng trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả được trình bày trong

Listing 56.3 trên cửa sổ console.

Listing 56.3: Kết quả từ driver thử nghiệm.

```
$ ./a.exe "http://this is a bad url"
    "http://localhost/c:/x*.xml"
Input String: http://this is a bad url
Encoded: http%3A%2F%2Fthis%20is%
    20a%20bad%20url
Decoded: http://this is a bad url
Input String: http%3A%2F%2Fthis%20is%20a%
    20bad%20url
Encoded: http%3A%2F%2Fthis%20is%20
    a%20bad%20url
Decoded: http://this is a bad url
Input String: http://localhost/c:/x*.xml
Encoded: http%3A%2F%2Flocalhost%2Fc%3A%
    2Fx%2A%2Exml
```

```
Decoded: http://localhost/c:/x*.xml
Input String: http%3A%2F%2Flocalhost%2Fc%3A%
    2Fx%2A%2Exml
Encoded: http%3A%2F%2Flocalhost%2Fc%3A%
    2Fx%2A%2Exml
Decoded: http://localhost/c:/x*.xml
```

Chú ý các chuỗi đầu vào trên lệnh phải được đặt trong các dấu trích dẫn; nếu không chúng sẽ được phân tích thành các từ riêng biệt trên các ngắt khoảng trống.

Như bạn có thể thấy, dữ liệu nhập được chuyển đổi thích hợp thành phiên bản mã hoá của chuỗi URL vốn có thể được sử dụng bởi các trình duyệt Web hoặc Web server. Phiên bản giải mã là những gì mà bạn mong đợi có một dạng được sử dụng bởi bất kỳ ứng dụng.

Kỹ thuật 57: Mã hoá và giải mã các chuỗi

Tiết kiệm thời gian bằng cách:

- Bảo vệ dữ liệu bằng sự mã hoá
- Tìm hiểu và thực thi thuật toán Rot13
- Tìm hiểu và thực thi thuật toán XOR
- Hiểu kết quả

Sẽ thật tuyệt vời hay không nếu tất cả chúng ta có thể tin tưởng mọi người xung quanh chúng ta không xem hoặc truy cập thông tin riêng tư của chúng ta. Thật không may, không phải mọi người đều hoàn toàn đáng tin cậy như bạn nghĩ. Vấn đề là thông tin nhạy cảm chẳng hạn như các password, tên user và số thẻ tín dụng không nên được lưu trữ bằng một cách luôn đọc được. Nếu chúng ta không làm ẩn thông tin bằng một cách nào đó, chắc chắn thông tin sẽ tìm đường đi đến mọi cracker trên Internet và được sử dụng bằng mọi cách xấu và ngấm ngầm. Làm ẩn thông tin là một tác vụ thường được thực hiện bởi sự mã hoá (encryption) - chuyển đổi dữ liệu từ một định dạng mà con người đọc được thành một định dạng mà con người không thể đọc được. Hầu như có nhiều cách để lưu trữ dữ liệu bằng với số cách để tạo cho nó lúc ban đầu. Những phương thức quan trọng chẳng hạn như các thuật toán mã hoá RSS hoặc Blowflish rất phức tạp. Chúng đòi hỏi nhiều trang để giải thích (và cuối cùng chúng vẫn dường như khó hiểu).

Kỹ thuật này trình bày hai phương pháp rất đơn giản nhưng hiệu quả để mã hoá dữ liệu khỏi những cặp mắt tò mò: thuật toán Rot13 và thuật toán XOR (XOR là viết tắt của "Exclusive Or"). Cả hai phương thức này có thể đánh bại những kẻ rình mò, nhưng chúng không an

toàn. Bạn không nên sử dụng một trong hai phương thức này cho các ứng dụng có sức mạnh công nghiệp. Không khó nếu không phải không thể thêm chức năng mã hoá cho một ứng dụng sau khi nó đã được viết. Để tạo ra một hệ thống an toàn, sự mã hoá nên được đưa vào tiến trình càng sớm càng tốt. bằng cách thêm những thuật toán này vào giai đoạn thiết kế, bạn sẽ tiết kiệm thời gian và công sức và tạo ra một hệ thống an toàn hơn.

Thực thi thuật toán Rot13

Thuật toán Rot13 thực sự là một cách rất đơn giản để mã hoá dữ liệu nhằm làm cho dữ liệu đó khó đọc, nhưng hầu như dễ giải mã. Như với tên gọi, thuật toán này đơn giản xoay các ký tự 13 vị trí trong bảng chữ cái. Do đó, A trở thành N... Thuật toán bao bọc xung quanh, do đó bất cứ những gì đi qua Z quay trở về A. Các bước sau đây hướng dẫn bạn cách tạo một lớp đơn giản có thể mã hoá và giải mã các chuỗi Rot13. Lớp này chắc chắn không phải là một sự mã hoá có sức mạnh công nghiệp, nhưng nó sẽ làm cho người bình thường khó đọc các chuỗi của bạn.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch57.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã từ Listing 57.1 vào file.

Listing 57.1: Mã thuật toán Rot 13

```
#include <string>
#include <iostream>
using namespace std;
class Rot13Encryption
{
private:
    string _encrypt;
protected:
    string rot13(const string& strIn)
    {
        string sOut = "";
        for ( int i=0; i<(int)strIn.size();
            ++i )
        {
```

```
            char ch = strIn[i];
            // the following assumes that
            'a' + 25 == 'z' and
            'A' + 25 == 'Z', etc.                    → 1
            if( (ch >= 'N' && ch <= 'Z') ||
               (ch >= 'n' && ch <= 'z') )
               ch -= 13;
            else if( (ch >= 'A' && ch <=
               'M') || (ch >= 'a' && ch <=
               'm') )
                  ch += 13;
               sOut += ch;
            }
            return sOut;
         }
      public:
         Rot13Encryption(void)
         {
         } Rot13Encryption( const char *strIn )
         {
         if ( strIn )
         {
               _encrypt = rot13( strIn );
            }
         }
         Rot13Encryption( const Rot13Encryption&
         aCopy )
         {
            _encrypt = aCopy._encrypt;
         }
         Rot13Encryption operator=( const
           Rot13Encryption& aCopy )
         {
            _encrypt = aCopy._encrypt;
            return *this;
         }
         string operator=( const char *strIn )
         {
```

```
            if ( strIn )
            {
                _encrypt = rot13( strIn );
            }
            return _encrypt;
        }
        const char *operator<<( const char
        *strIn )
        {
            if ( strIn )
            {
                _encrypt = rot13( strIn );
            }
            return _encrypt.c_str();
        }
        string String(void) const
        {
            return _encrypt;
        }
};
ostream& operator<<( ostream& out, const
    Rot13Encryption& r13 )
{
    out << r13.String().c_str();
    return out;
}
}
```

Mã trong listing ở trên thực thi thuật toán Rot13 đơn giản. Phần lớn công việc được hoàn tất trong hàm Rot13 vốn đơn giản xoay các ký tự 13 vị trí trong bảng chữ cái. Nếu bạn xem mã tại → 1, bạn thấy điều này tiến hành như thế nào. Như chú giải trong hàm này xác định, nó giả định rằng bảng chữ cái gần kề cho tập hợp ký tự mà bạn làm việc. Điều này có nghĩa mã này sẽ không làm việc trên các hệ thống EBCDIC cũ hơn, nó cũng sẽ không làm việc với các tập hợp ký tự không phải tiếng Anh. Thật không may, điều này tương tự đối với hầu hết các thuật toán mã hoá dựa vào text. Các phương thức khác của lớp chẳng hạn như phương thức operator << là các hàm tiện ích được sử dụng để chuyển đổi chuỗi được mã hoá để xuất hoặc để stream nó sang một file xuất.

3. Lưu file nguồn trong text editor.

Lớp này sẽ xử lý thuật toán Rot13. Thuật toán này làm việc bằng cách đơn giản xoay dữ liệu xung quanh trong bảng chữ cái. Sau đó, chuỗi không thể đọc được bởi con người, đây là toàn bộ vấn đề của sự mã hoá.

Test thuật toán Rot13

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để bảo đảm không chỉ mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Danh sách sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và trình bày cách lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này chương trình thử nghiệm được đặt tên là ch57.cpp.

2. Gõ nhập mã từ Listing 57.2 vào file.

Listing 57.2: Test lớp Rot13 Encryption

```
int main( int argc, char **argv )
{
    Rot13Encryption r13("This is a test");
    cout << r13.String().c_str() << endl;
    cout <<
       Rot13Encryption(r13.String().c_str())
       << endl;
    return 0;
}
```

3. Biên dịch file nguồn, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

4. Chạy ứng dụng trong console.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ console:

```
Guvf vf n grfg
This is a test
```

Kết quả đầu tiên (được minh hoạ tại → 2) là phiên bản xoay của chuỗi. Nó vẫn có thể đọc được bởi con người ít ra lên đến một điểm bởi vì các ký tự được thay thế đều nằm trong bảng chữ cái) nhưng nó

chắc chắn không đưa ra bằng chứng về nội dung ngữ nghĩa của text mà nó mã hoá. Do đó, mục đích của mã hoá được giữ lại.

Thật không may, Rot13 là một trong những thuật toán đang được sử dụng phổ biến nhất; những hacker biết nó rất rõ. Chúng ta cần một phương pháp tốt hơn.

Thực thi thuật toán XOR

Thuật toán mã hoá kế tiếp mà chúng ta xem xét là thuật toán XOR. XOR là viết tắt của Exclusive OR, nghĩa là nó sử dụng toán tử "exclusive or" để chuyển đổi text. Một thuộc tính của phép toán or loại trừ (exclusive) là một ký tự vốn or loại trừ với một ký tự khác có thể được trả về trạng thái ban đầu của nó bằng cách or nó lần nữa với cùng một ký tự. Điều này có nghĩa một password mã hoá có thể được sử dụng để vừa mã hoá vừa giải mã một chuỗi bằng cách sử dụng XOR. Trong kỹ thuật này, chúng ta xây dựng một lớp đơn giản nhằm thực thi thuật toán XOR và cung cấp các phương thức để mã hoá và giải mã các chuỗi.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch57.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Thêm mã từ Listing 57.3 vào file của bạn.

Listing 57.3: Lớp XOREncryption

```
class XOREncryption
{
private:
   char *_encrypt;
   int _length;
   string _key;
protected:
   char *do_xor( const char *sIn, int
   length, const string& key)                    → 3
   {
      int idx = 0;
      char *strOut = new char[ length ];
      if ( !key.length() )
         return strOut;
      for ( int i=0; i<length; ++i )
      {
```

```
                char c = (sIn[i] ^ key[idx]);
                strOut[i] = c;
                idx ++;
                if ( idx >= key.length() )
                    idx = 0;
            }
            return strOut;
        }
    public:
        XOREncryption(void)
        {
            _encrypt = NULL;
        }
        XOREncryption( const char *strIn, int
        length, const char *keyIn )
        {
            if ( keyIn )
                _key = keyIn;
            if ( strIn )
            {
                _length = length;
                _encrypt = do_xor( strIn,
        length, _key );
            }
        }
        XOREncryption( const XOREncryption&
            aCopy )
        {
            _encrypt = new char [ aCopy._length
        ];
            memcpy ( _encrypt, aCopy._encrypt,
              aCopy._length );
            _key = aCopy._key;
            _length = aCopy._length;
        }
        XOREncryption operator=( const
          XOREncryption& aCopy )
        {
```

```
    _encrypt = new char [ aCopy._length
      ];
    memcpy ( _encrypt, aCopy._encrypt,
      aCopy._length );
    _key = aCopy._key;
    _length = aCopy._length;
    return *this;
}
~XOREncryption(void)
{
    delete _encrypt;
}
const char *operator=( const char *strIn
)
 {
    if ( _encrypt )
       delete _encrypt;
    if ( strIn )
    {
       _encrypt = do_xor( strIn,
          strlen(strIn), _key );
    }
    return _encrypt;
}
const char *operator<<( const char
*strIn )
{
    if ( strIn )
    {
       _encrypt = do_xor( strIn,
       strlen(strIn), _key );
    }
    return _encrypt;
}
const char *String(void) const
{
    return _encrypt;
}
```

```
    int Length(void) const
    {
        return _length;
    }
};
```

Mã trong lớp này thực thi một thuật toán XOR đơn giản. Chức năng chính của lớp được trình bày trong phương thức do_xor, được minh hoạ tại → **3**. Như bạn có thể thấy phương thức lấy khoá mã hoá đầu vào và XOR nó với chuỗi mà người dùng cung cấp. Lớp đòi hỏi hai chuỗi, một chuỗi vốn là một "key" (khoá) được sử dụng để mã hoá hoặc giải mã các chuỗi. Chuỗi thứ hai là chuỗi đầu vào cần được mã hoá hoặc giải mã. Việc chạy thuật toán với hai đầu vào giống nhau hai lần sẽ tạo ra chuỗi gốc.

3. Lưu file nguồn trong text editor.

Test thuật toán XOR

Các bước sau đây hướng dẫn bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và trình bày cách lớp được sử dụng như thế nào:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch57.cpp.

2. Gõ nhập mã từ Listing 57.4 vào file.

Listing 57.4: Test lớp XOREncryption.

```
int main( int argc, char **argv )
{
    Rot13Encryption r13("This is a test");
    cout << r13.String().c_str() << endl;
    cout <<
        Rot13Encryption(r13.String().c_str())
        << endl;
    XOREncryption x1("This is a test",
        strlen("This is a test"), "C++Test");
    cout << x1.String() << endl;
    XOREncryption x2(x1.String(),
        x1.Length(), "C++Test");                    → 4
```

```
    cout << x2.String() << endl;
    return 0;
}
```

3. Biên dịch file nguồn, sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

4. Chạy ứng dụng trong console.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ console:

```
$ ./a.exe
Guvf vf n grfg
This is a test
_CB'E_cJ_                                    → 5
This is a test                               → 6
```

Chú ý kết quả ở trên bao gồm sự mã hoá Rot13 mà chúng ta đã phát triển trước đó trong kỹ thuật này để so sánh. Các chuỗi đều được mã hoá cứng vào ứng dụng và kết quả của bạn có thể khác phụ thuộc vào font và loại thiết bị đầu cuối bạn sử dụng. Kết quả trên lớp XOREncryption được minh hoạ tại → **5** và → **6**. Chuỗi gốc là This is a test và hai dòng theo sau nó cho thấy dòng đó được mã hoá trước tiên và sau đó được giải mã bằng cách sử dụng cùng một khoá như thế nào. Trong trường hợp này, "khoá" là chuỗi C++Test.

Lớp XOREncryption không sử dụng một chuỗi để chứa phiên bản mã hoá của dữ liệu nhập (xem → **4** trong Listing 57.4), nó cũng không trả về giá trị dưới dạng một đối tượng chuỗi. Đây là do lớp string chứa chỉ dữ liệu chữ - số. Phép toán xor có thể tạo ra các giá trị không phải chữ - số và đôi khi khiến cho lớp string trả về chỉ một phần của chuỗi gốc.

Không bao giờ sử dụng một đối tượng string để lưu trữ các bộ đệm ký tự (character buffer) vốn có thể chứa các giá trị rỗng (NULL) hoặc ký tự điều khiển. Các lớp chuỗi giả định rằng ký tự rỗng kết thúc chuỗi và sẽ không lưu trữ bất kỳ ký tự ở phía sau giá trị rỗng.

Kỹ thuật 58: Chuyển đổi kiểu chữ của một chuỗi

Tiết kiệm thời gian bằng cách:

- Sử dụng các kỹ thuật hiện đại để chuyển đổi kiểu chữ của các chuỗi đầu vào

- Sử dụng hàm transform của Standard Template Library
- Hiểu kết quả

Trong những ngày lập trình C trước đây, chuyển đổi kiểu chữ (case) của một chuỗi đã là một vấn đề đơn giản. Bạn chỉ việc gọi hàm strupr và chuỗi ngay tức thì được chuyển đổi thành chữ hoa. Tương tự, nếu bạn gọi hàm strlwr, chuỗi được chuyển đổi thành chữ thường. Mọi thứ đã thật sự thay đổi nhiều như thế kể từ đó phải không? Trong một số cách mọi thứ đã thay đổi nhiều. Ví dụ, mã sau đây không thể cho phép:

```
string myString = "hello world"
strupr ( myString );
```

Mã này sẽ không biên dịch vì hàm strupr không chấp nhận một đối số chuỗi. Bạn cũng không thể viết mã sau đây và mong đợi nó làm việc:

```
strupr (myString.c_str( ));
```

Mã này cũng sẽ không biên dịch; hàm strupr không thể chấp nhận một pointer ký tự const - đây là những gì mà phương thức c_str trả về. Vậy thì làm thế nào viết mã để chuyển đổi các đối tượng chuỗi hiện đại thành chữ hoa và chữ thường? Bạn có thể sử dụng kỹ thuật brute-force (thúc ép thô bạo) như sau:

```
for ( int i=0; i<myString.length(); ++i )
myString[i] = toupper(myString[i]);
```

Mã này chắc chắn làm việc, nhưng nó không tinh tế cho lắm và nó không dự đoán trước tất cả trường hợp. Ví dụ, nó giả định rằng các chuỗi ký tự không nằm theo thứ tự gần kề trong bộ nhớ - đây là một giả định kém đưa ra về bất kỳ tập hợp Standard Template Library (STL). Toàn bộ mục đích của STL là cung cấp cho nhà phát triển một cách để truy cập các đối tượng chứa (các chuỗi chỉ là những đối tượng chứa của các đối tượng) mà không có bất kỳ giả định về cách chúng được tổ chức trong bộ nhớ như thế nào. Dĩ nhiên lựa chọn tốt hơn là sử dụng một iterator (xem kỹ thuật 49 để biết thêm chi tiết về các iterator). Tuy nhiên, STL cung cấp một phương pháp thậm chí tốt hơn cho toàn bộ mọi thứ vốn là hàm transform được tìm thấy trong các thuật toán của STL. Hàm transform cho phép bạn thao tác bằng một cách độc lập với đối tượng chứa để chỉnh sửa các phần tử riêng lẻ của một đối tượng chứa thông qua một hàm chuyển đổi. Điều này tiết kiệm thời gian bởi vì thuật toán đã được viết, được gỡ rối và được tối ưu hoá. Nó cũng tiết kiệm thời gian bởi vì bạn có thể dễ dàng mở rộng các hàm chuyển đổi mà không cần viết lại thuật toán cơ bản.

Hãy luôn chắc chắn rằng bạn sử dụng trước mã hiệu quả nhất cho ứng dụng của bạn. Thay vì cố thực thi các thuật toán riêng của bạn để

làm việc với Standard Template Library, chọn sử dụng các thuật toán trong file include <algorithm> vì chúng đã được tối ưu hoá để làm việc với các tập hợp STL.

Thực thi hàm transform để chuyển đổi các chuỗi

Thuật toán transform của Standard Template Library sử dụng một hàm được gọi bởi người dùng thuật toán để chuyển đổi mỗi phần tử của một đối tượng chứa (container) bằng một cách nào đó. Các bước sau đây hướng dẫn bạn cách tạo một hàm transform đơn giản để chuyển đổi kiểu chữ của một chuỗi thành chữ hoa hoặc chữ thường. Bằng cách xem kỹ thuật và cách mã được thực thi như thế nào, bạn sẽ có thể thấy cách mở rộng chức năng cho những sử dụng riêng của bạn trong tương lai. Để thực thi hàm này, chúng ta sẽ cần thực thi một lớp vốn làm công việc biến đổi. Thuật toán transform chấp nhận hai iterator và một đối tượng để làm công việc của nó. Hãy xem chính xác điều này được thực thi như thế nào trong mã riêng của bạn.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch58.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào mà bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã từ Listing 58.1 vào file.

Listing 58.1: Mã chuyển đổi cho lớp chuỗi STL.

```
#include <string>
#include <algorithm>
#include <iostream>
#include <vector>
#include <ctype.h>
using namespace std;
// A function for converting a string to
// lowercase.
string convert_to_lowercase( const string&
    sIn )
{
    // First, save a pointer to the
    function.
    int (*pf)(int)=tolower;
    // Next, convert the string.
    string sOut = sIn;
```

```
        transform(sOut.begin(), sOut.end(),
        sOut.begin(), pf);
        return sOut;
}
// A function for converting a string to
// uppercase.
string convert_to_uppercase( const string&
        sIn )
{
        // First, save a pointer to the
        function.
        int (*pf)(int)=toupper;
        // Next, convert the string.
        string sOut = sIn;
        transform(sOut.begin(), sOut.end(),
        sOut.begin(), pf);
        return sOut;
}
string strip_leading  ( const string& sIn )
{
        // Find the first non-white-space
           character.
        int iPos = 0;
        while ( iPos < sIn.length() && isspace
        ( sIn[iPos] ) )
           iPos ++;
        // Copy the rest of it.
        string sOut;
        for ( int i=iPos; i<sIn.length(); ++i )
           sOut += sIn[i];
        return sOut;
}
string strip_trailing ( const string& sIn )
{
        // Find the last non-white-space
        character.
        int iPos = sIn.length()-1;
        while ( iPos >= 0 && isspace( sIn[iPos]
```

```
        ) )
            iPos —;
        // Copy the rest of it.
        string sOut;
        for ( int i=0; i<=iPos; ++i )
            sOut += sIn[i];
        return sOut;
    }
    // This is a utility class that will convert
    // a string to lowercase.
    class StringConvertToLowerCase                    → 1
    {
    public:
        string operator()(string s)
        {
            return convert_to_lower_case(s);
        }
    };
    // This is a utility class that will strip
    // leading AND trailing white space.
    class StringStripSpace
    {
    public:
        string operator()(string s)
        {
            return strip_leading(strip_
                trailing(s));
        }
    };
```

Mã này thực thi tất cả phép biến đổi khác nhau có thể có cho lớp string và đưa vào một số phương thức để xử lý các chuỗi, (chẳng hạn như loại bỏ các ký tự đứng trước và đứng sau). Trong tất cả trường hợp, chúng ta sẽ sử dụng phương thức transform để thực sự chỉnh sửa các chuỗi hoặc mảng. Như driver trong kỹ thuật này minh hoạ, một chuỗi đơn không khó chuyển đổi hơn toàn bộ một mảng chuỗi. Chức năng quan trọng ở đây là lớp mà chúng ta sẽ sử dụng để chuyển đổi các chuỗi thành chữ thường vốn là lớp StringConvertToLowerCase được minh hoạ tại → 1. Hàm transform sử dụng lớp này để chuyển đổi các chuỗi. Như bạn có thể thấy, tất cả công việc được hoàn tất

trong một phương thức, phương thức operator( ). Phương thức này được gọi bởi một thuật toán transform để làm công việc của nó như chúng ta sẽ thấy ngay sau đây.

Test các chuyển đổi chuỗi

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để bảo đảm rằng không chỉ mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào.

Danh sách sau đây trình bày cho bạn cách tạo một driver thử nghiệm nhằm minh hoạ các loại đầu vào khác nhau từ người dùng và cho thấy lớp được sử dụng như thế nào.

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch58.cpp.

2. Gõ nhập mã từ Listing 58.2 vào file.

Listing 58.2: Driver thử nghiệm chuyển đổi chuỗi.

```
int main(int argc, char **argv)
{
    vector<string> stringArray;
    if ( argc < 2 )
    {
        cout << "Usage: ch7_3 string1
          [string2 string3...]" << endl;
        cout << "Where: string[n] is the
          string to convert" << endl;
        return -1;
    }
    // First, do them individually.
    cout << "Individual Conversions: " <<
       endl;
    for ( int i=1; i<argc; ++i )
    {
        cout << "Input String: " << argv[i]
           << endl;
        string sLower =
           convert_to_lower_case( argv[i] );
        cout << "Lowercase String: " <<
```

```
            sLower.c_str() << endl;
         string sUpper =
            convert_to_upper_case( argv[i] );
         cout << "Uppercase String: " <<
            sUpper.c_str() << endl;
         stringArray.insert(
            stringArray.end(), argv[i] );                    → 4
      }
      // Now do the whole array.
      transform(stringArray.begin(),
         stringArray.end(),
         stringArray.begin(),
         StringConvertToLowerCase() );
         // Print them out.
      cout << endl << "Array Conversions: " <<
         endl;
vector<string>::iterator iter;
for ( iter = stringArray.begin(); iter
      != stringArray.end(); ++iter )
         cout << "String: " <<
            (*iter).c_str() << endl;
      cout << endl << "Individual String
         Whitespace Strip Test: " << endl;
      string whiteSpace = " This is a test
         ";
      string sNoWhite = strip_leading(
         whiteSpace );
      cout << "Stripped String: [" <<
         sNoWhite.c_str() << "]" << endl;
      sNoWhite = strip_trailing( whiteSpace );
      cout << "Stripped String: [" <<
         sNoWhite.c_str() << "]" << endl;
      transform(stringArray.begin(),
         stringArray.end(),
        stringArray.begin(),
         StringStripSpace() );
      cout << endl << "Array of Strings
```

```
    Whitespace Strip Test: " << endl;
  for ( iter = stringArray.begin(); iter
    != stringArray.end(); ++iter )
      cout << "String: [" <<
        (*iter).c_str() << "]" << endl;
  return 0;
}
```

3. Lưu mã nguồn trong code editor và đóng ứng dụng biên soạn.
4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy chương trình trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả được trình bày trong Listing 58.3 trong cửa sổ console.

Listing 58.3: Kết quả từ cuộc thử nghiệm chuyển đổi chuỗi.

```
$ ./a.exe " This is a test" "This is
    another test " " Final Test "                    → 2
Individual Conversions:                              → 3
Input String: This is a test
Lowercase String: this is a test
Uppercase String: THIS IS A TEST
Input String: This is another test
Lowercase String: this is another test
Uppercase String: THIS IS ANOTHER TEST
Input String: Final Test
Lowercase String: final test
Uppercase String: FINAL TEST
Array Conversions:
String: this is a test                               → 5
String: this is another test
String: final test
Individual String Whitespace Strip Test:
Stripped String: [This is a test ]
Stripped String: [ This is a test]
Array of Strings Whitespace Strip Test:
String: [this is a test]
String: [this is another test]
String: [final test]
```

Dòng đầu tiên của kết quả đơn giản là tên file thực thi và các đối số cho chương trình (được minh hoạ tại → **2**). Như bạn có thể thấy chúng ta chuyển vào ba đối số. Các phép biến đổi khác nhau được chạy trên từng đối số này. Đầu tiên chúng ta sử dụng các hàm tiện ích (như được minh hoạ tại → **3**) để chuyển đổi từng chuỗi đầu vào. Điều này đơn giản cho thấy các hàm làm việc tốt. Tiếp theo, các chuỗi được thêm vào một mảng (được minh hoạ tại → **4** trong Listing 58.2). Sau đó, các hàm transform được áp dụng, chuyển đổi mỗi chuỗi thành chữ thường (được minh hoạ trong kết quả tại → **5**). Sau cùng, để thấy sự biến đổi có thể được áp dụng cho bất kỳ chuỗi như thế nào, chúng ta sử dụng các hàm loại bỏ khoảng trắng để thay đổi các chuỗi sao cho không có khoảng trắng đứng trước và đứng sau.

Kỹ thuật 59: Thực thi một giao diện chuỗi hoá

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các giao diện (interface)
- Tìm hiểu sự chuỗi hoá (serialization)
- Thực thi một giao diện chuỗi hoá
- Test giao diện

Thực thi các giao diện (interface) có thể tiết kiệm cho bạn vô số thời gian khi bạn làm cùng một công việc lặp đi lặp lại nhiều lần trong ứng dụng hoặc thậm chí qua các ứng dụng. Ngoài ra, nó thu thập tất cả mã cho một tác vụ nào đó ở một nơi làm cho việc thay đổi định dạng của các file xuất hoặc thuật toán được sử dụng cho chức năng của giao diện trở nên nhanh chóng và dễ dàng.

Nếu bạn đã từng sử dụng Java làm ngôn ngữ lập trình, có lẽ bạn đã quen thuộc với các giao diện. Đơn giản một giao diện (interface) là một lớp cơ sở mà từ đó bạn có thể thừa kế việc cung cấp một dịch vụ (service) nào đó. Ngôn ngữ C++ hỗ trợ các giao diện mặc dù chúng hơi khác về cú pháp. Không giống như một lớp cơ sở điển hình, các giao diện ít cụ thể hơn và thường không cho phép ứng dụng dựa vào chúng, nhưng thay vào đó sử dụng chúng để cung cấp một dịch vụ riêng biệt. Trong C++, giao diện là một lớp cơ sở ảo thuần tuý, chứa nhiều phương thức ảo thuần tuý. Phương thức ảo thuần tuý không giống như phương thức ảo thông thường phải được ghi đè bởi một lớp dẫn xuất. Ví dụ, một giao diện có thể cho phép lớp tự in ra hoặc tự lưu sang một file hoặc thậm chí cấp phát bộ nhớ của nó từ một dạng không gian phần cứng chuyên biệt nào đó. Kỹ thuật này hướng dẫn bạn cách thực thi một khái niệm quan trọng - chuỗi hoá (serialization) - dưới dạng một giao diện. Điều quan trọng nhất về các giao

diện và cách chúng sẽ tiết kiệm cho bạn phần lớn thời gian là nếu bạn thừa kế từ một giao diện, bạn có thể chuyển đối tượng đến bất kỳ hàm hoặc phương thức vốn làm việc với các đối tượng thực thi giao diện đó. Do đó, nếu bạn có một hàm được sử dụng để lưu tất cả loại đối tượng, bạn có thể chuyển đối tượng đến hàm miễn là nó thực thi giao diện Save.

Về cơ bản, serialization (chuỗi hoá) là khả năng một đối tượng tự ghi sang một dạng lưu trữ ổn định nào đó. Mã làm công việc này có khuynh hướng giống nhau từ lớp này đến lớp khác; dữ liệu được ghi là những gì thay đổi. Do đó, sự chuỗi hoá hoàn toàn thích hợp với tiến trình tạo một giao diện.

Có hai bước cơ bản để thực thi một giao diện trong lớp:

- Bạn phải tạo lớp giao diện thật sự và nhận dạng tất cả vùng mà bạn cần dữ liệu nhập từ lớp dẫn xuất. Tại giai đoạn này, bạn thực thi tất cả chức năng cho lớp giao diện - tạo nó như thể nó là một lớp chính cho ứng dụng.
- Sau khi đã tạo lớp giao diện, bạn thiết lập lớp dẫn xuất để thừa kế từ nó (và để thực thi bất kỳ hàm ảo mà giao diện đòi hỏi).

Thực thi giao diện serialization

Giao diện phổ biến nhất là giao diện serialization. Giao diện này được sử dụng bởi nhiều thư viện lớp phổ biến chẳng hạn như MFC cho phép một đối tượng được ghi sang bộ nhớ ổn định (chẳng hạn như một file) miễn là nó thực thi một giao diện nhất quán. Trong ví dụ này, chúng ta sẽ khai thác cách tạo các giao diện serialization và áp dụng giao diện đó vào một lớp hoặc các lớp nào đó.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho định nghĩa giao diện của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch59.h mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng serialization.

2. Gõ nhập mã từ Listing 59.1 vào file.

Listing 59.1: Mã giao diện serialization

```
#ifndef _SERIALIZE_H_
#define _SERIALIZE_H_
#include <string>
#include <vector>
#include <iostream>
#include <fstream>
using namespace std;
```

```
class SerializeEntry
{
    string _name;
    string _value;
public:
    SerializeEntry(void)
    {
    }
    SerializeEntry( const char *name, const
        char *value )
    {
        setName( name );
        setValue( value );
    }
    SerializeEntry( const char *name, int
     iValue )
    {
        setName( name );
        setValue( iValue );
    }
    SerializeEntry( const char *name, double
     dValue )
    {
        setName( name );
        setValue( dValue );
    }
    SerializeEntry( const SerializeEntry&
      aCopy )
    {
        setName( aCopy.getName().c_str() );
        setValue( aCopy.getValue().c_str()
    );
      }
    SerializeEntry operator=(const
      SerializeEntry& aCopy)
    {
        setName( aCopy.getName().c_str() );
        setValue( aCopy.getValue().c_str()
```

```
    );
        return *this;
    }
        void setName( const char *name )
        {
            if ( name )
                _name = name;
            else
                _name = "";
        }
        void setValue( const char *value )
        {
            if ( value )
                _value = value;
            else
                _value = "";
        }
        void setValue( int iValue )
        {
            char szBuffer[ 20 ];
            sprintf(szBuffer, "%d", iValue );
            setValue ( szBuffer );
        }
        void setValue( double dValue )
        {
            char szBuffer[ 20 ];
            sprintf(szBuffer, "%lf", dValue );
            setValue ( szBuffer );
        }
        string getName(void) const
        {
            return _name;
        }
        string getValue(void) const
        {
            return _value;
    }
    };
```

```
class Serialization
{
private:
    long _majorVersion;
    long _minorVersion;
protected:
    virtual bool getEntries( vector<
      SerializeEntry >& entries )
    {
      return true;
    }
    virtual string getClassName()
    {
      return "None";
    }
public:
    Serialization(void)
    {
      _majorVersion = 0;
      _minorVersion = 0;
    }
    Serialization( long major,
      long minor )                                    → 2
    {
      _majorVersion = major;
      _minorVersion = minor;
    }
    Serialization( const Serialization&
    aCopy )
    {
      _majorVersion = aCopy._majorVersion;
      _minorVersion = aCopy._minorVersion;
    }
    Serialization operator=( const
    Serialization& aCopy )
    {
      _majorVersion = aCopy._majorVersion;
      _minorVersion = aCopy._minorVersion;
```

```
    return *this;
}
// Accessors
void setMajorVersion( long major )
{
    _majorVersion = major;
}
long getMajorVersion( void )
{
    return _majorVersion;
}
void setMinorVersion( long minor )
{
    _minorVersion = minor;
}
long getMinorVersion( void )
{
    return _minorVersion;
}
// Functionality
bool write( const char *strFileName )                    → 1
{
// Note the call to the virtual
    method. This must
// be overridden in the derived
    class code.
vector< SerializeEntry > entries;
getEntries( entries );                                   → 3
if ( entries.size() == 0 )
    return false;
// Try opening the output file.
ofstream out( strFileName,
        ofstream::out | ofstream::app );
if ( out.fail() )
    return false;
// Write out the class name.
out << "<" << getClassName() << ">"
    << endl;
```

```
        // Write out the version information.
        out << "\t<VERSION>" <<
        _majorVersion << ":" <<
        _minorVersion << "</VERSION>" <<
        endl;
        vector< SerializeEntry >::iterator
            iter;
        for ( iter = entries.begin(); iter
        != entries.end(); ++iter )
        {
            out << "\t<" <<
                (*iter).getName().c_str() <<
            ">" << endl;
            out << "\t\t" <<
                (*iter).getValue().c_str() <<
            endl;
            out << "\t</" <<
                (*iter).getName().c_str() <<
                ">" << endl;
            }
            out << "</" << getClassName() << ">"
            << endl;
            return true;
        }
};
#endif
```

Mã này lưu một lớp được ấn định sang một stream (luồng) đầu ra theo định dạng XML chuẩn. Rõ ràng có thể đủ đơn giản chỉnh sửa lớp để xuất theo một định dạng khác nào đó, nhưng vì mục đích đơn giản, chúng ta sẽ sử dụng XML. Thành viên quan trọng của lớp là phương thức write được minh hoạ tại → 1. Phương thức này ghi ra các thành viên của một lớp theo định dạng XML sử dụng các biến thành viên của lớp để quyết định file xuất và thông tin phiên bản. Chú ý lớp theo dõi thông tin phiên bản riêng của nó để bạn có thể sử dụng nó nhằm quyết định xem một phiên bản ổn định của lớp có phải là phiên bản thích hợp cho ứng dụng hay không. Thông tin phiên bản được định nghĩa trong constructor như được minh hoạ tại → 2.

Hãy xem phương thức write; dường như mã làm mọi thứ mà bạn mong đợi một đối tượng serialization thực hiện. Phần quan trọng của

hàm được minh hoạ tại → 3 vốn là một lệnh gọi đến một phương thức ảo có tên là getEntries. Phương thức này vốn sẽ được ghi đè bởi bất kỳ lớp vốn sử dụng giao diện này trả về các biến thành viên riêng lẻ của lớp dưới dạng một mảng chuỗi. Sau khi phương thức này được tạo cho lớp, phần còn lại của chức năng của giao diện serialization sẽ làm việc bất kể nội dung của các lớp dẫn xuất là gì.

3. Lưu file nguồn và đóng file trong code editor.

Như bạn có thể thấy lớp serialization tự làm hầu hết công việc. Nó đòi hỏi thực thi chỉ hai phương thức ảo:

- Phương thức getElements trả về tất cả phần tử trong của lớp mà bạn muốn lưu.
- Phương thức getClassName trả về tên của lớp để sử dụng làm gốc của cây XML vốn nhận dữ liệu phần tử mà chúng ta ghi ra.

Ngoài ra, bạn có thể xác định các phiên bản chính và phụ cho file được chuỗi hoá sao cho nếu bạn muốn import các file được chuỗi hoá hiện có, sau này bạn sẽ có thể import dữ liệu được tạo ra bởi các phiên bản cũ hơn của mã nguồn và mặc định các giá trị bị thiếu một cách phù hợp nếu những giá trị mới đã được thêm vào thời điểm đó.

Test giao diện Serialization

Sau khi đã định nghĩa giao diện serialization, bước kế tiếp là thực thi giao diện đó cho một lớp thật sự. Điều này cho bạn thấy tiến trình serialization được sử dụng như thế nào và khái niệm interface sẽ tiết kiệm bao nhiêu thời gian trong các phần thực thi lớp sau này. Hãy xem việc tạo một lớp với các thành viên cần được lưu trữ một cách ổn định.

1. Trong code editor, tạo một file mới để chứa mã cho lớp thử nghiệm của kỹ thuật.

   Trong ví dụ này, file được đặt tên là ch59.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa mã thử nghiệm cho kỹ thuật này.

2. Gõ nhập mã từ Listing 59.2 vào file của bạn.

Listing 59.2: Test giao diện serialization

```
#include "serialize.h"
class MyClass : public Serialization
{
private:
    int _iValue;
    string _sValue;
```

```
        double _dValue;
    protected:
        virtual bool getEntries( vector
        < SerializeEntry >& entries )                    → 4
        {
            entries.insert( entries.end(),
            SerializeEntry( "iValue",
            _iValue ) );                                 → 5
            entries.insert( entries.end(),
            SerializeEntry( "sValue",
            _sValue.c_str() ) );
            entries.insert( entries.end(),
            SerializeEntry( "dValue", _dValue
            ) );
        }
        virtual string getClassName( void )
        {
            return "MyClass";
        }
    public:
        MyClass(void)
            : Serialization(1,0)                         → 6
        {
            _iValue = 0;
            _dValue = 0.0;
            _sValue = "";
        }
        MyClass(int iVal, double dVal, const
            char *sVal)
                : Serialization(1,0)
        {
            _iValue = iVal;
            _dValue = dVal;
            _sValue = sVal;
        }
        virtual ~MyClass()
        {
```

```
        Save();
    }
    virtual void Save()
    {
        write( "MyClass.xml" );
    }
};
int main()
{
    MyClass m1(1,2.0,"One");
    return 0;
}
```

Lớp này thực thi một tập hợp dữ liệu với một giá trị chuỗi, một giá trị số nguyên và một giá trị dấu động (double). Công việc quan trọng xảy ra trong hàm ảo getEntries vốn tạo một mảng phần tử để xuất để chuỗi hoá. Phương thức này được minh hoạ tại → **4** ghi đè phương thức giao diện serialization và sẽ được sử dụng để xuất. Sau khi phương thức này được ghi đè, phần mã còn lại làm việc như mong muốn trong giao diện. Chú ý việc sử dụng lớp Serialization (được minh hoạ tại → **5**) để chứa dữ liệu cho mỗi phần tử. Bởi vì lớp này có thể được dẫn xuất để tận dụng các kiểu dữ liệu khác, khái niệm interface làm việc thậm chí vào tương lai.

3. Lưu mã nguồn dưới dạng một file trong bộ soạn thảo và đóng ứng dụng soạn thảo.
4. Biên dịch mã nguồn bằng trình biên dịch ưa thích trên HTML ưa thích.
5. Chạy chương trình trên hệ điều hành ưa thích.

Nếu bạn đã làm đúng mọi thứ, chương trình sẽ tạo một file kết quả có tên là MyClass.xml trong hệ thống file của hệ điều hành. File này sẽ chứa kết quả sau đây sau khi ứng dụng được chạy.

```
$ cat MyClass.xml
<MyClass>
    <VERSION>1:0</VERSION>
    <iValue>
        1
    </iValue>
    <sValue>
        One
```

```
        </sValue>
        <dValue>
            2.000000
        </dValue>
    </MyClass>
```

Trong kết quả, bạn có thể thấy tất cả phần tử dữ liệu được xác định trong lớp MyClass đã được xuất thông qua giao diện serialization. Điều này cho thấy mã làm việc như đã xác định. Cũng chú ý thông tin phiên bản được xác định trong constructor cho lớp MyClass (được minh hoạ tại → **6**) được xuất trong file ổn định để chúng ta có thể tận dụng nó nếu chúng ta thêm các thành viên mới vào lớp MyClass.

Như bạn có thể thấy lớp serialization làm chính xác những gì nó phải làm. Ngoài ra, chú ý mã cho lớp serialization hầu như không có liên quan đến lớp mà nó được gọi từ đó. Việc thiếu sự chuyên dụng hoá này cho phép chúng ta dễ dàng tái sử dụng lớp làm một giao diện cho bao nhiêu lớp khác tuỳ thích - cho phép từng lớp chuỗi hoá dữ liệu một cách hiệu quả.

Cũng chú ý nếu bạn đột ngột được hướng dẫn thay đổi dữ liệu xuất sang một định dạng không phải XML, mã cần điều chỉnh ở chỉ một nơi.

Điều này tiết kiệm cho bạn thời gian, tránh những sai sót và theo kịp những ưu điểm tốt nhất của lập trình hướng đối tượng.

Kỹ thuật 60: Tạo một lớp Buffer chung

Tiết kiệm thời gian bằng cách:

- Tìm hiểu các lỗi tràn bộ đệm (buffer overflow)
- Ngăn các lỗi tràn bộ đệm không bị khai thác như là một rủi ro an ninh
- Tạo một lớp Buffer để xử lý các lỗi này
- Test lớp

Tràn bộ đệm xảy ra trong phần lớn các chương trình C hoặc C++. Bây giờ hãy tưởng tượng đọc dữ liệu từ một file nhập liệu. Mã C và C++ điển hình có thể trông như sau:

```
char szBuffer[80];
int nPos = 0;
while ( !eof(fp) )
{
```

```
        szBuffer[nPos] = fgetc(fp);
        if ( szBuffer[nPos] == '|' )
            break;
        nPos++;
    }
```

Thường trình này phải đọc một chuỗi từ file, đọc cho đến khi nó gặp phải một ký tự (|) hoặc cuối file. Nhưng điều gì xảy ra khi chuỗi dài hơn 80 ký tự? Trong trường hợp đó, thường trình tiếp tục đọc thông tin, ghi đè các vị trí bộ nhớ theo sau biến szBuffer. Chúng ta gọi vấn đề này là một sự tràn bộ đệm (buffer overflow). Thông tin nào được lưu trữ trong các vị trí bộ nhớ đó? Khó nói: Có thể không có gì quan trọng ở đây hoặc có thể một địa chỉ trả về quan trọng cho một hàm được ghi đè. Trong trường hợp trước, có thể bạn không bao giờ thấy một vấn đề. Trong trường hợp sau, chương trình có thể dễ dàng bị phá vỡ, phơi bày một điểm yếu nghiêm trọng ra trước thế giới bên ngoài.

Những vấn đề như vậy được gọi là sự tràn bộ đệm thật sự rất phổ biến trong thế giới phát triển phần mềm - nhưng trừ phi chúng gây ra những vấn đề lớn (hoặc thậm chí chưa được phát hiện), các nhà lập trình có khuynh hướng bỏ qua chúng. Thậm chí sự tràn bộ đệm được xem là vấn đề an ninh số 1 trong phần mềm ngày nay. Phụ thuộc vào ứng dụng, một sự tràn bộ đệm có thể làm đổ vỡ ứng dụng hoặc đơn giản huỷ một phần quan trọng nào đó của bức tường an ninh vốn ngăn một người dùng bên ngoài không chỉnh sửa dữ liệu trong một chương trình. Vậy thì tại sao chúng ta không làm một điều gì đó về điều này?

Các lời giải đáp đơn giản trông khá ngớ ngẩn: đây là cách mà chúng ta đã luôn viết mã, vấn đề không nghiêm trọng như vậy và chúng ta sẽ thay đổi bây giờ. Thực sự vấn đề tương đối dễ giải quyết - và không có lý do nào để không làm như vậy. Kỹ thuật này hướng dẫn bạn cách thực hiện như thế nào. Bằng cách loại bỏ những rủi ro an ninh, bạn sẽ cung cấp một ứng dụng an toàn hơn và giảm thiểu lượng thời gian bạn dành ra để cấp phát các bảng vá lỗi an ninh và giải quyết những cơn ác mộng hỗ trợ khách hàng, điều này sẽ tiết kiệm cho bạn nhiều thời gian.

Ví dụ, hãy tưởng tượng viết lại thường trình vừa được cho để nó trông như sau:

```
Buffer szBuffer(80); → 1
int nPos = 0;
try
{
    while ( !eof(fp) )
```

```
        {
            szBuffer[nPos] = fgetc(fp);
            if ( szBuffer[nPos] == '\' )
                break;
            nPos++;
        }
    }
    catch ( BufferException& exc )
    {
    printf("Buffer Overflow!");
    }
    return 0;
```

Bây giờ mã này không làm chương trình đổ vỡ. Không giống như mã trước đó, vốn đã sử dụng một mảng tĩnh, chương trình này sử dụng một lớp

Buffer (xem → 1) để quản lý bộ đệm. Lớp Buffer kiểm tra xem bộ đệm bên dưới đã bị ghi đè hay không và ngăn bộ nhớ trở nên quá tải. Nếu một trong hai điều kiện xảy ra, chương trình đưa ra một ngoại lệ - và phục hồi một cách tinh tế.

Giải pháp này cho phép các chương trình hoạt động theo cách lý tưởng đó là tự giải quyết các vấn đề. Có lẽ không thể phục hồi hoàn toàn từ một lỗi như vậy, nhưng ít ra chương trình có thể thoát một cách an toàn, tất tất cả nối kết và không để hệ điều hành bị tổn hại. Đây là những gì mà sự an ninh phần mềm nói đến. Vậy vấn đề là tạo một lớp thực thi bộ đệm chung và bảo vệ dữ liệu của bạn. Đó là mục đích của kỹ thuật này,

Tạo lớp Buffer

Giải pháp mà chúng ta sẽ thực thi cho sự tràn bộ đệm là tạo một lớp bảo vệ bộ đệm dữ liệu bằng cách chỉ cho phép truy cập đến các vùng hợp lệ của bộ đệm. Lớp này ghi đè các toán tử vốn cho phép truy cập đến các ký tự riêng lẻ và bảo đảm tất cả thao tác gán, sao chép, và xử lý làm việc chỉ trên các phần hợp lệ của bộ đệm. Bây giờ hãy tạo lớp đó bây giờ.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho file nguồn của kỹ thuật này.

   Trong ví dụ này, file được đặt tên là ch60.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn. File này sẽ chứa định nghĩa lớp cho đối tượng tự động hoá.

2. Gõ nhập mã trong Listing 60.1 vào file của bạn.

Listing 60.1: Lớp Buffer

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string>
using namespace std;
class BufferException                                    → 4
{
private:
    string _errMsg;
public:
    BufferException( void )                              → 5
    {
        _errMsg = "";
    }
    BufferException( const char *msg )
    {
        _errMsg = msg;
    }
    BufferException( const BufferException&
        aCopy )
    {
        _errMsg = aCopy._errMsg;
    }
    const char *Message()
    {
        return _errMsg.c_str();
    }
    void setMessage( const char *msg )
    {
        _errMsg = msg;
    }
};
class Buffer
{
private:
    // $Member: m_Buffer - This is the
```

```
        actual area of allocated memory.
        char *m_Buffer;
        // $Member: m_Size - This is the size of
        the allocated memory.
        int m_Size;
    protected:
        virtual void print_buffer( const char
            *strPrefix )
        {
            printf("%s: [", strPrefix );
            for ( int i=0; i<m_Size; ++i )
                  printf("%c", m_Buffer[i] );
               printf("]\n");
        }
        virtual void clear()
        {
            if ( m_Buffer )
               delete m_Buffer;
            m_Buffer = NULL;
            m_Size = 0;
        }
        virtual void copy( const Buffer& aBuffer
            )
        {
            m_Buffer = new char[ aBuffer.Size()
               ];
            memcpy( m_Buffer, aBuffer._Buffer(),
               aBuffer.Size() );
            m_Size = aBuffer.Size();
        }
        virtual void allocate( int nSize )
        {
            m_Buffer = new char[ nSize ];
            memset( m_Buffer, 0, nSize );
            m_Size = nSize;
        }
        const char *_Buffer(void) const
        {
```

```
        return m_Buffer;
    }
public:
    // Void constructor. No memory is allocated
    or available.
    Buffer(void)
        : m_Buffer(NULL), m_Size(0)
    {
        // Clear everything.
        clear();
    }
    // This is a standard constructor.
    Buffer( int nSize )
        : m_Buffer(NULL), m_Size(0)
    {
        clear();
        allocate( nSize );
    }
    // Copy the constructor.
    Buffer( const Buffer& aBuffer )
        : m_Buffer(NULL), m_Size(0)
    {
        clear();
        copy( aBuffer );
    }
    virtual ~Buffer()
    {
        clear();
    }
    // Operators
    Buffer &operator=(const Buffer& aBuffer)
    {
        clear();
        copy( aBuffer );
        return *this;
    }
    Buffer &operator=(const char *strBuffer)
    {
```

```
            // If they are assigning us NULL,
               just clear
            // everything out and get out of
               here.
        clear();
        if ( strBuffer == NULL )
            return *this;
        // Otherwise, we need to set up this
        object
        // as the passed-in string.
        allocate( (int)strlen(strBuffer)+1
            );
        memcpy( m_Buffer, strBuffer,
            strlen(strBuffer) );
            return *this;
        }
char& operator[](int nPos)                          → 2
{
    if ( nPos < 0 || nPos > m_Size-1 )
        throw BufferException( "Buffer:
            Array index out of range" );
    return m_Buffer[ nPos ];
}
operator const char*()                              → 3
{
    // Just give them back the entire
    buffer.
    return m_Buffer;
}
const char *c_str( void )
{
    return m_Buffer;
}
    // Here come the accessor functions.
    int Size() const
    {
        return m_Size;
    }
```

```
        // These are memory-based functions.
        void Set( char c )
        {
            for ( int i=0; i<m_Size-1; ++i )
                m_Buffer[i] = c;
        }
        void Clear( void )
        {
            Set( 0 );
        }
        Buffer operator()(int nStart,
            int nLength)                                    → 4
        {
            // Do some validation
            if ( nStart < 0 || nStart > m_Size-1
        )
            {
                throw BufferException("Buffer:
                    Array index out of range");
            }
            if ( nLength < 0 || nLength >
                m_Size-1 || nStart+nLength >
                m_Size-1 )
            {
                throw BufferException("Buffer:
                    Length out of range");
            }
            Buffer b(nLength+1);
                for ( int i=0; i<nLength; ++i )
                    b[i] = m_Buffer[i+nStart];
                return b;
        }
    };
```

Trong mã trong listing ở trên, các phần tử nhất định làm cho nó trở thành một bước quan trọng đi lên từ mảng ký tự C++ chuẩn. Trên hết chú ý cách các kỳ tự được truy tìm bằng cách sử dụng toán tử tạo index ([ ]). operator [ ], được minh hoạ tại → **2** cẩn thận kiểm tra xem index được yêu cầu có nằm trong một dãy hợp lệ hay không. Nếu

không nó sẽ đưa ra một ngoại lệ. Tương tự chuỗi con operator ( ) (được minh hoạ tại → **4**) kiểm tra để bảo đảm rằng tất cả ký tự nằm trong dãy hợp lệ. Không giống như mảng ký tự, lớp Buffer cho phép bạn trích xuất các đoạn nhỏ của chính nó, nhưng bảo vệ ngăn ngừa các đoạn không hợp lệ. Bởi vì giá trị trả về là một đối tượng Buffer, phương thức này an toàn khỏi sự tràn (overruns). Bạn có thể chú ý toán tử chuyển đổi được minh hoạ tại dòng được đánh dấu là → **3**; nó dường như cung cấp một cách để nhà lập trình huỷ chuỗi. Thật ra bởi vì nó trả về một pointer hằng dẫn đến bộ đệm, nhà lập trình không thể trực tiếp thay đổi nó mà không cast chuỗi, một việc mà trình biên dịch sẽ sẵn sàng chú ý. Toàn bộ ý tưởng là chúng ta ngăn nhà lập trình không làm một điều gì đó gây ra các sự cố mà không suy nghĩ kỹ về nó.

3. Lưu mã nguồn trong code editor.

Chú ý rằng chúng ta tạo lớp Exception riêng của chúng ta (được minh hoạ tại dòng được đánh dấu là → **5**) để trả về các lỗi từ lớp Buffer. Nên có các dạng ngoại lệ riêng của bạn thay vì sử dụng các kiểu cơ bản. Theo cách đó, nhà lập trình có thể xử lý các lỗi hệ thống bằng những cách khác với những giải pháp lập trình.

Test lớp Buffer

Sau khi tạo một lớp, bạn nên tạo một driver thử nghiệm để bảo đảm không chỉ mã của bạn chính xác mà còn hướng dẫn người ta cách sử dụng mã của bạn như thế nào. Các bước sau đây hướng dẫn cách thực hiện:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong ví dụ này, chương trình thử nghiệm được đặt tên là ch60.cpp.

2. Gõ nhập mã từ Listing 60.2 vào file.

Listing 60.2: Driver thử nghiệm cho lớp Buffer

```
int main(int argc, char* argv[])
{
    Buffer b1(20);
    Buffer b2;
    b2 = b1;
    b1 = "This is a test";                          → 6
    printf("The buffer is: %s\n", (const
      char *)b1);
```

```
    b1[2] = 'a';
    printf("The buffer is: %s\n", (const
       char *)b1);
    try
    {
       b1[-1] = 'a';                          → 7
    }
    catch ( BufferException& exc )
    {
       printf("Caught the error: %s\n",
          exc.Message() );
    }
    // Test the "sub-buffer" function.
    Buffer b3;
    b3 = b1(0,4);                             → 8
    printf("The new buffer is: [");
    for ( int i=0; i<b3.Size(); ++i )
       printf("%c", b3[i] );
    printf("]\n");
    return 0;
}
```

Mã ở trên thi hành một số chức năng quan trọng của lớp Buffer. Đầu tiên chúng ta kiểm tra xem các toán tử gán làm việc như mong đợi hay không. Điều này được minh hoạ tại → **6** trong Listing 60.2. Tiếp theo, chúng ta test xem lô gíc tạo index có làm việc tốt hay không như được minh hoạ tại → **7**. Nếu nó làm việc tốt, chúng ta muốn thấy ngoại lệ được đưa ra và được in ra trên console. Sau cùng, chúng ta test các hàm thường trình con bằng cách trích xuất một phần của bộ đệm và làm cho nó trở thành một đối tượng Buffer mới (được minh hoạ tại → **8**). Nếu toán tử làm việc chính xác, chúng ta sẽ thấy bộ đệm mới là bốn ký tự đầu tiên của chuỗi gốc, hãy test nó xem.

3. Lưu file nguồn trong code editor và đóng ứng dụng soạn thảo.
4. Biên dịch file nguồn bằng trình biên dịch ưa thích trên hệ điều hành ưa thích.
5. Chạy ứng dụng trên hệ điều hành ưa thích.

Nếu bạn đã làm tốt mọi thứ, bạn sẽ thấy kết quả sau đây trên cửa sổ console:

```
$ ./a.exe
The buffer is: This is a test
The buffer is: Thas is a test
Caught the error: Buffer: Array index out
of range
The new buffer is: [Thas ]
```

Như bạn có thể thấy kết quả chính là những gì chúng ta đã mong đợi. Lỗi đã được tạo ra và được đón bắt và thông tin ngoại lệ đã được in sang console. Chuỗi con đã là những ký tự mà chúng ta cũng mong đợi biểu thị rằng toán tử gán gốc và các toán tử chuỗi con chính xác. Bằng cách sử dụng lớp này, chúng ta có thể mong đợi tiết kiệm nhiều thời gian trong việc gỡ rối các ứng dụng và trong việc phát triển các ứng dụng bởi vì nhiều chức năng được phơi bày làm cho kiểm tra dữ liệu đầu vào nhanh hơn.

Như bạn có thể thấy, mã này mang lại tính an toàn cao hơn. Chắc chắn nó đánh bại việc cho phép các bộ đệm bị tràn và bộ nhớ bị quá tải.

Kỹ thuật 61: Mở một file bằng cách sử dụng nhiều đường dẫn

Tiết kiệm thời gian bằng cách:

- Thêm mã để cho phép người dùng xác định các đường dẫn tìm kiếm đi đến các file
- Tạo một lớp nhiều đường dẫn tìm kiếm
- Test lớp

Một điều thường xảy ra khi thiết kế phần mềm máy tính: bạn đề nghị người dùng xác định một file để mở trong ứng dụng của bạn và sau đó cho phép người này chọn file đó bằng cách "duyệt" đến đúng đường dẫn. Thật không may, không có cách nào tốt để tận dụng hệ điều hành nền tảng để tìm các file riêng biệt đặc biệt nếu bạn muốn ứng dụng có thể mang chuyển qua những hệ điều hành khác nhau. Do đó, sẽ hợp lý hơn nếu tạo một phương thức để thật sự nhìn qua tất cả đường dẫn tìm kiếm thật sự hiện hữu khi bạn mở một file một cách tự động mà không cần quan tâm nó nằm ở đâu hoặc cách truy cập chúng như thế nào.

Kỹ thuật này thực hiện công việc này bằng cách xây dựng một lớp tiện ích nhằm quản lý nhiều đường dẫn để tìm kiếm các file, sử dụng lớp đó để tìm và đọc bất kỳ file nào mà người dùng xác định. Không có điều kỳ diệu ở đây - chỉ là một cách tiện lợi để tận dụng chức năng

cài sẵn của các hàm C++ Standard Library để tránh các vấn đề sử dụng hệ điều hành để tìm các file vì tiến trình này khác nhau cho mỗi hệ điều hành. Không có điều gì kỳ diệu về việc tìm kiếm các đường dẫn tìm kiếm khác nhau và mở các file, nhưng việc kết hợp hai đối tượng thành một đối tượng đơn sẽ mang lại một số sức mạnh và khả năng mà bạn có thể tận dụng trong rất nhiều ứng dụng - với một chút nỗ lực của bạn. Dĩ nhiên, đó là toàn bộ vấn đề của việc tạo các lớp tiện ích trong C++.

Lớp tiện ích mới có trách nhiệm đối với ba công việc:

- Nó lưu trữ các đường dẫn khác nhau để tìm kiếm và bảo đảm rằng các đường dẫn đó đều hợp lệ
- Nó sử dụng các đường dẫn tìm kiếm đầu vào để tìm một file nào đó khi người dùng xác định một file
- Nó mở file đã chọn và trả một tham chiếu đối tượng stream trở về nhà lập trình để sử dụng trong việc xử lý file. Sau đó người dùng có thể tìm file này nằm ở đâu - và đọc file từ đối tượng stream khi cần thiết. Người dùng có thể không biết hoặc không quan tâm file mà họ đã yêu cầu thật sự được đặt ở đâu miễn là nó có thể được tìm thấy dọc theo một trong các đường dẫn tìm kiếm khác nhau.

Tạo lớp nhiều đường dẫn tìm kiếm

Một lớp tiện ích cho phép người dùng tìm một file bằng cách sử dụng nhiều đường dẫn tìm kiếm đơn giản hợp lý. Các bước sau đây tạo một lớp như vậy được gọi là MultiPathFile.

1. Trong code editor mà bạn chọn, tạo một file mới để chứa mã cho phần thực thi của file nguồn.

   Trong ví dụ này, file được đặt tên là ch61.cpp mặc dù bạn có thể sử dụng bất kỳ tên nào bạn chọn.

2. Gõ nhập mã từ Listing 61.1 vào file.

Listing 61.1: Lớp tiện ít nhiều đường dẫn tìm kiếm.

```
#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <sys/stat.h>
using namespace std;
class DirectoryList
{
private:
    vector< string > _entries;
```

```
public:
    DirectoryList(void)
    {
    }
    DirectoryList( const char *strDir )
    {
        _entries.insert( _entries.end(),
            strDir );
    }
    DirectoryList( const DirectoryList&
    aCopy )
    {
        vector<string>::const_iterator
            iter;
        for ( iter =
            aCopy._entries.begin(); iter !=
            aCopy._entries.end(); ++iter )
                _entries.insert(
                    _entries.end(), (*iter) );
    }
    bool is_dir( const char *strDir )                    → 1
    {
        struct stat st;
        if ( stat( strDir, &st )
            == 0 )                                       → 3
        {
            if ( st.st_mode & S_IFDIR )
                return true;
        }
        return false;
    }
    void addEntry( const char *strDir )
    {
        _entries.insert( _entries.end(),
            strDir );
    }
    void removeEntry( const char *strDir )
```

```
    {
        vector<string>::iterator iter;
        for ( iter = _entries.begin();
            iter != _entries.end(); ++iter )
        {
            if ( (*iter) == strDir )
            {
                _entries.erase( iter );
            }
        }
    }
    int count(void) const
    {
        return _entries.size();
    }
    DirectoryList operator+=( const char
        *strDir )
    {
        _entries.insert( _entries.end(),
            strDir );
        return *this;
    }
    DirectoryList operator-=( const char
    *strDir )
    {
        removeEntry( strDir );
        return *this;
    }
    string getEntry( int idx )
    {
        if ( idx < 0 || idx > count()-1 )
            throw "DirectoryList: Array
            Index out of range";
        return _entries[ idx ];
    }
};
class MultiPathFile
{
```

```
private:
    DirectoryList _pathList;
    ifstream _in;
    string _path;
public:
    MultiPathFile(void)
        : _path("")
    {
    }
    MultiPathFile( const DirectoryList&
        aPathList )
            : _pathList( aPathList ),
            _path("")
    {
    }
    MultiPathFile( const char *strDir )
        : _pathList( strDir ),
            _path("")
    {
    }
    void addPath( const char *strDir )
    {
        if ( _pathList.is_dir( strDir ) )
            _pathList.addEntry( strDir );
        else
            printf("Invalid path:
            [%s]\n", strDir );
    }
    void removePath( const char *strDir )
    {
        _pathList.removeEntry( strDir );
    }
    bool open( const char *strFileName )          → 2
    {
        for ( int i=0;
        i<_pathList.count(); ++i )
        {
            string sDir =
```

```
            _pathList.getEntry( i );
            string sFullPath = sDir +
            strFileName;
            _in.open( sFullPath.c_str(),
            ios::in );
            if ( !_in.fail() )
            {
               _path = sDir;
               return true;
            }
            _in.clear();
         }
            return false;
      }
      void close()
      {
         _in.close();
      }
      ifstream& file(void)
      {
         return _in;
      }
      string CurrentPath(void)
      {
         return _path;
      }
};
```

3. Lưu mã nguồn trong ứng dụng soạn thảo mã.

Mã ở trên phân chia thành hai lớp:

- **Lớp DirectoryList**: Lớp này duy trì một danh sách các thư mục khác nhau trong hệ thống và cho nhà lập trình có một cách nhất quán để truy cập các tên thư mục. Không có gì thực sự đáng ngạc nhiên trong lớp này. Nó chỉ là một wrapper xung quanh lớp vector Standard Template Library (STL) thực hiện việc kiểm tra xem một tên nào đó có phải là một thư mục hay không. (Xem is_dir, được minh hoạ tại → **1**).
- **Lớp MultiPathFile:** Đây thực sự là một phiên bản mở rộng của các file cơ bản được cung cấp bởi XML. Nó duy trì một danh sách thư mục cần tìm kiếm bằng cách sử dụng lớp DirectoryList để chứa các

thư mục khác nhau. Khi một yêu cầu open được nhận thông qua phương thức open như được minh hoạ tại → **2**, lớp lặp lại qua các thư mục khác nhau trong danh sách của nó và cố mở file trong mỗi thư mục. Nếu file được tìm thấy trong một thư mục đã ấn định, nó lưu trữ đường dẫn thư mục nơi file đã được tìm thấy và trả về một handle nhằm cho phép nhà lập trình truy cập file.

Hai lớp này (DirectoryList và MultiPathFile) làm tất cả công việc quản lý các đường dẫn tìm kiếm và sau đó tận dụng các đường dẫn tìm kiếm đó để mở và xử lý file. Chú ý việc sử dụng stat (một hàm C chuẩn) được minh hoạ tại dòng được đánh dấu là → **3** để kiểm tra xem một đường dẫn đầu vào có hợp lệ và có phải là một thư mục hay không.

Chú ý bạn sẽ cần thêm các đường dẫn tìm kiếm vào hệ thống bằng cách sử dụng dấu phân cách đường dẫn (đây là dấu gạch chéo tiến hoặc lùi phụ thuộc vào hệ điều hành bạn đang làm việc) được thêm vào cuối đường dẫn. Nghĩa là bạn phải nhập c:/windows/ thay vì chỉ là :/windows bởi vì hệ điều hành sẽ không thêm dấu phân cách cho bạn. Điều này có thể được giải quyết dễ dàng, nhưng đòi hỏi thêm mã trong những gì đã là một listing khá dài.

Test lớp nhiều đường dẫn tìm kiếm (Multiple-Search-Path)

Sau khi bạn tạo một lớp, bạn nên tạo một driver thử nghiệm để bảo đảm không chỉ mã của bạn chính xác mà còn hướng dẫn người khác cách sử dụng mã của bạn như thế nào. Các bước sau đây cho bạn biết cách thực hiện:

1. Trong code editor mà bạn chọn, mở lại file nguồn để chứa mã cho chương trình thử nghiệm.

   Trong v1i dụ này chương trình thử nghiệm được đặt tên là ch61.cpp.

2. Gõ nhập mã từ Liating 61.2 vào file.

   Listing 61.2: Chương trình thử nghiệm Multiple- Search - Path.

```
void display_file( ifstream& in )
{
    // Display the first 100 characters
    cout << endl;
    for ( int i=0; i<100; ++i )
    {
        char c;
        in.get(c);
        if ( !in.fail() )
```

```
            cout << c;
         else
            break;
      }
      cout << endl;
}
int main( int argc, char **argv )
{
      MultiPathFile paths;
      // First, add in all the paths
      for ( int i=1; i<argc; ++i )
      {
         paths.addPath( argv[i] );
      }
      // Now ask them what they want to do.
      bool bDone = false;
      while ( !bDone )
      {
         char szPath[ 256 ];
         char szFile[ 256 ];
         printf("Options:\n");
         printf("(1) Add a new search
            path\n");
         printf("(2) Open a file\n");
         printf("(3) Exit the
            program\n\n");
         printf("What do you want to do?
            ");
         int option = 0;
         scanf("%d", &option );
      getchar();
      switch ( option )
      {
         case 1:                                    → 4
            printf("Enter search
               path to add: ");
               memset( szPath, 0,
   sizeof(szPath) );
```

```
            gets(szPath);
            if ( strlen(szPath) )
               paths.addPath(
               szPath );
            break;
         case 2:                                          → 5
            printf("Enter file to
              open: ");
            memset( szFile, 0,
              sizeof(szFile) );
            gets(szFile);
            if ( strlen(szFile) )
               if ( !paths.open(
                  szFile ) )
                     printf("Error
                        finding file
                        %s\n",
                        szFile );
                  else
                  {
                     printf("File
                        found at:
                        [%s]\n",
                        paths.Curren
                        tPath().c_st
                        r() );
                     display_file(
                        paths.file()
                        );                                → 7
                  }
               break;
            case 3:                                       → 6
               bDone = true;
               break;
            default:
               printf("Invalid
                  Option\n");
```

```
                    break;
                }
            }
            return 0;
        }
```

Mã ở trên test nhiều đường dẫn tìm kiếm để mở các file. Khi được chạy chương trình cho phép người dùng thêm các đường dẫn tìm kiếm khác nhau vào danh sách thư mục của chúng để sử dụng, sau đó cho người dùng tìm kiếm file bằng cách sử dụng các đường dẫn đó. Nếu người dùng muốn thêm một đường dẫn tìm kiếm mới, người dùng nhập '1' tại dòng nhắc trong mã tại → **4**. Mã này nhận được một tên đường dẫn từ người dùng và thêm nó vào danh sách đường dẫn tìm kiếm. Nếu người dùng muốn tìm một file, người dùng nhập '2' tại dòng nhắc và chương trình nhắc về tên file cần tìm kiếm (xem → **5**). Nếu nó được tìm thấy, nó sẽ được in ra cùng với đường dẫn mà nó đã được tìm thấy trong đó. Sau cùng, việc nhập '3' tại dòng nhắc sẽ kết thúc vòng lặp và thoát chương trình (xem → **6**).

3. Lưu mã nguồn trong code editor và sau đó đóng ứng dụng biên soạn.

4. Biên dịch ứng dụng bằng cách sử dụng trình biên dịch ưa thích trên hệ điều hành ưa thích.

5. Chạy ứng dụng trong cửa sổ console của hệ điều hành ưa thích

Nếu bạn đã làm đúng mọi thứ, bạn sẽ thấy kết quả sau đây:

```
$ ./a.exe
Options:
(1) Add a new search path
(2) Open a file
(3) Exit the program
What do you want to do? 1
Enter search path to add: c:/matt/                    → 8
Options:
(1) Add a new search path
(2) Open a file
(3) Exit the program
What do you want to do? 1
Enter search path to add: c:/work/                    → 9
Options:
(1) Add a new search path
```

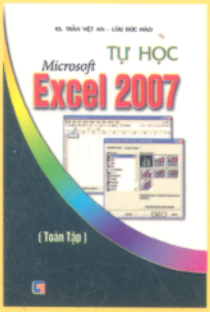
```
(2) Open a file
(3) Exit the program
What do you want to do? 1
Enter search path to add: c:/windows/                    → 10
Options:
(1) Add a new search path
(2) Open a file
(3) Exit the program
What do you want to do? 2
Enter file to open: DATECL.h                             → 11
File found at: [c:/work/]                                → 12
/*
*+-------------------------------------------------------
-----------------------
*I Header.....: DATE
Options:
(1) Add a new search path
(2) Open a file
(3) Exit the program
What do you want to do? 3
```

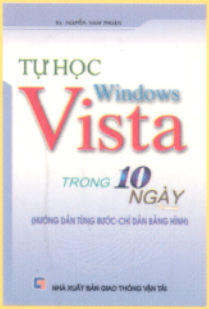
Như bạn có thể thấy, lớp tiện ích đã đi qua danh sách đường dẫn được cung cấp cho nó (được minh hoạ tại các dòng → **8**, → **9**, và → **10**), đã tìm thấy đường dẫn khớp với tên file được nhập (được minh hoạ tại → **11**), mở file và cho phép người dùng đọc nó. Ngoài ra, nó trả về đường dẫn đi đến nơi file đã được tìm thấy (được minh hoạ tại → **12**). Điều đó có thể được xuất cho người dùng ứng dụng. File được hiển thị thông qua lệnh gọi display_file trong Listing 61.2 tại dòng → **7**.

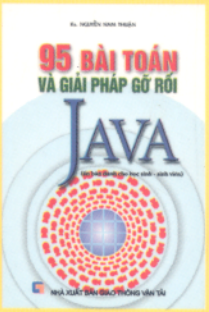
Mục lục

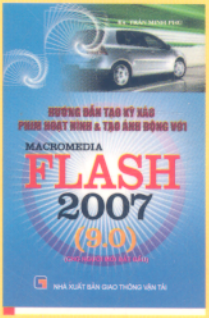
Visual C++ 2008

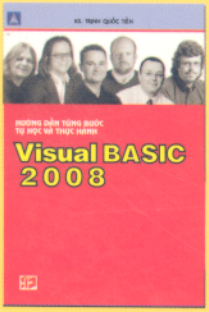
HƯỚNG DẪN TỪNG BƯỚC TỰ HỌC VÀ THỰC HÀNH

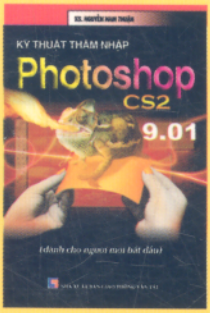


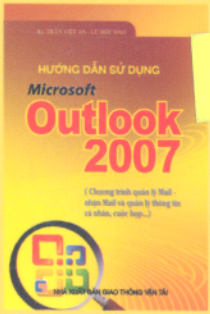


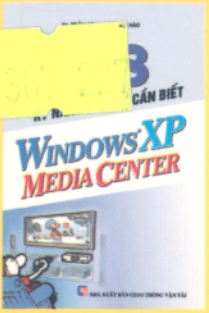












NVC
NHAN VAN CORP.
HỆ THỐNG NHÀ SÁCH NHÂN VĂN
Trân trọng tri thức Việt

TRUNG TÂM PHÁT HÀNH SÁCH:

* 875 CMT8 - P. 15- Q.10, TP. Hồ Chí Minh
Tel: 39770096 Fax: 9708161
* 486 Nguyễn thị Minh Khai , P.2, Q. 3, TP. Hồ Chí Minh
Tel/Fax: 8396733
* 01 Trường Chinh, P.11, Q. Tân Bình, TP. Hồ Chí Minh

www.nhanvan.com.vn

VISUAL C++ 2008
8 935072 829711
GIÁ: 86.000Đ