

NGÔN NGỮ

LẬP TRÌNH C#

Mục Lục

1. Microsoft .NET	10
Tình hình trước khi MS.NET ra đời	10
Nguồn gốc của .NET	12
Microsoft .NET	12
Tổng quan	12
Kiến trúc .NET Framework	13
Common Language Runtime	15
Thư viện .NET Framework	16
Phát triển ứng dụng client	16
Biên dịch và MSIL	17
Ngôn ngữ C#	18
2. Ngôn ngữ C#	20
Tại sao phải sử dụng ngôn ngữ C#	20
C# là ngôn ngữ đơn giản	20
C# là ngôn ngữ hiện đại	21
C# là ngôn ngữ hướng đối tượng	21
C# là ngôn ngữ mạnh mẽ	22
C# là ngôn ngữ ít từ khóa	22
C# là ngôn ngữ module hóa	22
C# sẽ là ngôn ngữ phổ biến	22
Ngôn ngữ C# với ngôn ngữ khác	23
Các bước chuẩn bị cho chương trình	24
Chương trình C# đơn giản	25
Phát triển chương trình minh họa	31
Câu hỏi & bài tập	35
3. Nền tảng ngôn ngữ C#	39
Kiểu dữ liệu	40
Kiểu dữ liệu xây dựng sẵn	41
Chọn kiểu dữ liệu	42
Chuyển đổi kiểu dữ liệu	43
Biến và hằng	44
Gán giá trị xác định cho biến	45
Hằng	46
Kiểu liệt kê	47

Kiểu chuỗi ký tự.....	50
Định danh.....	50
Biểu thức.....	50
Khoảng trắng.....	51
Câu lệnh.....	51
Phân nhánh không có điều kiện.....	52
Phân nhánh có điều kiện.....	53
Câu lệnh lặp.....	60
Toán tử.....	68
Namespace.....	76
Các chỉ dẫn biên dịch.....	80
Câu hỏi & bài tập.....	82
4. Xây dựng lớp - Đối tượng.....	87
Định nghĩa lớp.....	88
Thuộc tính truy cập.....	91
Tham số của phương thức.....	92
Tạo đối tượng.....	93
Bộ khởi dựng.....	93
Khởi tạo biến thành viên.....	96
Bộ khởi dựng sao chép.....	98
Từ khóa this.....	99
Sử dụng các thành viên static.....	100
Gọi phương thức static.....	101
Sử dụng bộ khởi dựng static.....	101
Sử dụng bộ khởi dựng private.....	102
Sử dụng thuộc tính static.....	102
Hủy đối tượng.....	104
Truyền tham số.....	107
Nạp chồng phương thức.....	112
Đóng gói dữ liệu với thuộc tính.....	116
Thuộc tính chỉ đọc.....	119
Câu hỏi & bài tập.....	121
5. Kế thừa – Đa hình.....	125
Đặc biệt hóa và tổng quát hóa.....	126
Sự kế thừa.....	129
Thực thi kế thừa.....	129
Gọi phương thức khởi dựng của lớp cơ sở.....	131
Gọi phương thức của lớp cơ sở.....	132

Điều khiển truy xuất.....	132
Đa hình.....	133
Kiểu đa hình.....	133
Phương thức đa hình.....	133
Từ khóa new và override.....	137
Lớp trừu tượng.....	139
Gốc của tất cả các lớp- lớp Object.....	142
Boxing và Unboxing dữ liệu.....	144
Boxing dữ liệu ngầm định.....	144
Unboxing phải thực hiện tường minh.....	145
Các lớp lồng nhau.....	147
Câu hỏi & bài tập.....	149
6. Nạp chồng toán tử.....	153
Sử dụng từ khóa operator.....	153
Hỗ trợ ngôn ngữ .NET khác.....	154
Sử dụng toán tử.....	154
Toán tử so sánh bằng.....	156
Toán tử chuyển đổi.....	157
Câu hỏi & bài tập.....	163
7. Cấu trúc.....	165
Định nghĩa một cấu trúc.....	165
Tạo cấu trúc.....	168
Cấu trúc là một kiểu giá trị.....	168
Gọi bộ khởi dựng mặc định.....	169
Tạo cấu trúc không gọi new.....	170
Câu hỏi & bài tập.....	172
8. Thực thi giao diện.....	176
Thực thi giao diện.....	177
Thực thi nhiều giao diện.....	180
Mở rộng giao diện.....	181
Kết hợp các giao diện.....	181
Truy cập phương thức giao diện.....	187
Gán đối tượng cho giao diện.....	187
Toán tử is.....	188
Toán tử as.....	190
Giao diện đối lập với trừu tượng.....	192
Thực thi phủ quyết giao diện.....	193
Thực thi giao diện tường minh.....	197

Lựa chọn thể hiện phương thức giao diện.....	200
Ăn thành viên.....	200
Câu hỏi & bài tập.....	207
9. Mảng, chỉ mục, và tập hợp.....	211
Mảng.....	212
Khai báo mảng.....	213
Giá trị mặc định.....	214
Truy cập các thành phần trong mảng.....	214
Khởi tạo thành phần trong mảng.....	216
Sử dụng từ khóa params.....	216
Câu lệnh foreach.....	218
Mảng đa chiều.....	220
Mảng đa chiều cùng kích thước.....	220
Mảng đa chiều có kích thước khác nhau.....	224
Chuyển đổi mảng.....	227
Bộ chỉ mục.....	232
Bộ chỉ mục và phép gán.....	236
Sử dụng kiểu chỉ số khác.....	237
Giao diện tập hợp.....	241
Giao diện IEnumerable.....	242
Giao diện ICollection.....	246
Danh sách mảng.....	247
Thực thi IComparable.....	251
Thực thi IComparer.....	254
Hàng đợi.....	259
Ngăn xếp.....	262
Kiểu từ điển.....	265
Hastables.....	266
Giao diện IDictionary.....	267
Tập khóa và tập giá trị.....	269
Giao diện IDictionaryEnumerator.....	270
Câu hỏi & bài tập.....	271
10. Xử lý chuỗi.....	275
Lớp đối tượng string.....	276
Tạo một chuỗi.....	276
Tạo một chuỗi dùng phương thức ToString.....	277
Thao tác trên chuỗi.....	278
Tìm một chuỗi con.....	285

Chia chuỗi.....	286
Thao tác trên chuỗi dùng StringBuilder.....	288
Các biểu thức quy tắc.....	290
Sử dụng biểu thức quy tắc qua lớp Regex.....	291
Sử dụng Regex để tìm tập hợp.....	294
Sử dụng Regex để gom nhóm.....	295
Sử dụng CaptureCollection.....	298
Câu hỏi & bài tập.....	301
11. Cơ chế ủy quyền và sự kiện.....	303
Ủy quyền.....	304
Sử dụng ủy quyền xác nhận phương thức lúc thực thi.....	304
Ủy quyền tĩnh.....	314
Dùng ủy quyền như thuộc tính.....	315
Thiết lập thứ tự thi hành với mảng ủy quyền.....	316
Multicasting.....	320
Sự kiện.....	324
Cơ chế publishing- subscribing.....	324
Sự kiện và ủy quyền.....	325
Câu hỏi & bài tập.....	333
12. Các lớp cơ sở .NET.....	335
Lớp đối tượng trong .NET Framework.....	335
Lớp Timer.....	337
Lớp về thư mục và hệ thống.....	340
Lớp Math.....	342
Lớp thao tác tập tin.....	345
Làm việc với tập tin dữ liệu.....	351
Câu hỏi & bài tập.....	362
13. Xử lý ngoại lệ.....	364
Phát sinh và bắt giữ ngoại lệ.....	365
Câu lệnh throw.....	365
Câu lệnh catch.....	367
Câu lệnh finally.....	373
Những đối tượng ngoại lệ.....	375
Tạo riêng các ngoại lệ.....	378
Phát sinh lại ngoại lệ.....	381
Câu hỏi & bài tập.....	385

Tham Khảo

Giáo trình “*Ngôn ngữ Lập trình C#*” được biên dịch và tổng hợp từ:

Programming C#, Jesse Liberty, O’Reilly.

C# in 21 Days, Bradley L.Jones, SAMS.

Windows Forms Programming with C#, Erik Brown, Manning.

MSDN Library – April 2002.

Quy ước

Giáo trình sử dụng một số quy ước như sau:

Các thuật ngữ được giới thiệu lần đầu tiên sẽ *in nghiêng*.

Mã nguồn của chương trình minh họa dùng font Verdana -10.

Các từ khóa của C# dùng font **Verdana-10, đậm** hoặc Verdana-10, bình thường.

Tên namespace, lớp, đối tượng, phương thức, thuộc tính, sự kiện... dùng font Verdana-10.

Kết quả của chương trình xuất ra màn hình console dùng font Courier New-10.

Chương 1

MICROSOFT .NET

- **Tình hình trước khi MS.NET ra đời**
 - **Nguồn gốc của .NET**
- **Microsoft .NET**
 - **Tổng quan**
 - **Kiến trúc .NET Framework**
 - **Common Language Runtime (CLR)**
 - **Thư viện .NET Framework**
 - **Phát triển ứng dụng client**
- **Biên dịch và MSIL**
- **Ngôn ngữ C#**

Tình hình trước khi MS.NET ra đời

Trong lĩnh vực công nghệ thông tin của thế giới ngày nay, với sự phát triển liên tục và đa dạng nhất là phần mềm, các hệ điều hành, các môi trường phát triển, các ứng dụng liên tục ra đời. Tuy nhiên, đôi khi việc phát triển không đồng nhất và nhất là do lợi ích khác nhau của các công ty phần mềm lớn làm ảnh hưởng đến những người xây dựng phần mềm.

Cách đây vài năm Java được Sun viết ra, đã có sức mạnh đáng kể, nó hướng tới việc chạy trên nhiều hệ điều hành khác nhau, độc lập với bộ xử lý (Intel, Risc,...). Đặc biệt là Java rất thích hợp cho việc viết các ứng dụng trên Internet. Tuy nhiên, Java lại có hạn chế về mặt tốc độ và trên thực tế vẫn chưa thịnh hành. Mặc dù Sun Corporation và IBM có đẩy mạnh Java, nhưng Microsoft đã dùng ASP để làm giảm khả năng ảnh hưởng của Java.

Để lập trình trên Web, lâu nay người ta vẫn dùng CGI-Perl và gần đây nhất là PHP, một ngôn ngữ giống như Perl nhưng tốc độ chạy nhanh hơn. Ta có thể triển khai Perl trên Unix/Linux hay MS Windows. Tuy nhiên có nhiều người không thích dùng do bản thân ngôn ngữ hay các qui ước khác thường và Perl không được phát triển thống nhất, các công cụ được xây dựng cho Perl tuy rất mạnh nhưng do nhiều nhóm phát triển và người ta không đảm bảo rằng tương lai của nó ngày càng tốt đẹp hơn.

Trong giới phát triển ứng dụng trên Windows ta có thể viết ứng dụng bằng Visual C++, Delphi hay Visual Basic, đây là một số công cụ phổ biến và mạnh. Trong đó Visual C++ là một ngôn ngữ rất mạnh và cũng rất khó sử dụng. Visual Basic thì đơn giản dễ học, dễ dùng nhất nên rất thông dụng. Lý do chính là Visual Basic giúp chúng ta có thể viết chương trình trên Windows dễ dàng mà không cần thiết phải biết nhiều về cách thức MS Windows hoạt động, ta chỉ cần biết một số kiến thức căn bản tối thiểu về MS Windows là có thể lập trình được. Do đó theo quan điểm của Visual Basic nên nó liên kết với Windows là điều tự nhiên và dễ hiểu, nhưng hạn chế là Visual Basic không phải ngôn ngữ hướng đối tượng (Object Oriented).

Delphi là hậu duệ của Turbo Pascal của Borland. Nó cũng giống và tương đối dễ dùng như Visual Basic. Delphi là một ngôn ngữ hướng đối tượng. Các điều khiển dùng trên Form của Delphi đều được tự động khởi tạo mã nguồn. Tuy nhiên, chức năng khởi động mã nguồn này của Delphi đôi khi gặp rắc rối khi có sự can thiệp của người dùng vào. Sau này khi công ty Borland bị bán và các chuyên gia xây dựng nên Delphi đã chạy qua bên Microsoft, và Delphi không còn được phát triển tốt nữa, người ta không dám đầu tư triển khai phần mềm vào Delphi. Công ty sau này đã phát triển dòng sản phẩm Jbuilder (dùng Java) không còn quan tâm đến Delphi.

Tuy Visual Basic bền hơn do không cần phải khởi tạo mã nguồn trong Form khi thiết kế nhưng Visual Basic cũng có nhiều khuyết điểm :

Không hỗ trợ thiết kế hướng đối tượng, nhất là khả năng thừa kế (inheritance).

Giới hạn về việc chạy nhiều tiểu trình trong một ứng dụng, ví dụ ta không thể dùng Visual Basic để viết một Service kiểu NT.

Khả năng xử lý lỗi rất yếu, không thích hợp trong môi trường Multi- tier

Khó dùng chung với ngôn ngữ khác như C++.

Không có User Interface thích hợp cho Internet.

Do Visual Basic không thích hợp cho viết các ứng Web Server nên Microsoft tạo ra ASP (Active Server Page). Các trang ASP này vừa có tag HTML vừa chứa các đoạn script (VBScript, JavaScript) nằm lẫn lộn nhau. Khi xử lý một trang ASP, nếu là tag HTML thì sẽ được gọi thẳng qua Browser, còn các script thì sẽ được chuyển thành các dòng HTML rồi gọi đi, ngoại trừ các function hay các sub trong ASP thì vị trí các script khác rất quan trọng.

Khi một số chức năng nào được viết tốt người ta dịch thành ActiveX và đưa nó vào Web Server. Tuy nhiên vì lý do bảo mật nên các ISP (Internet Service Provider) làm máy chủ cho Web site thường rất dè dặt khi cài ActiveX lạ trên máy của họ. Ngoài ra việc tháo gỡ các phiên bản của ActiveX này là công việc rất khó, thường xuyên làm cho Administrator nhức đầu. Những người đã từng quản lý các version của DLL trên Windows điều than phiền tại sao phải đăng ký các DLL và nhất là chỉ có thể đăng ký một phiên bản của DLL mà thôi. Và từ “DLL Hell” xuất hiện tức là địa ngục DLL...

Sau này để giúp cho việc lập trình ASP nhanh hơn thì công cụ Visual InterDev, một IDE (Integrated Development Environment) ra đời. Visual InterDev tạo ra các Design Time Controls cho việc thiết kế các điều khiển trên web,... Tiếc thay Visual InterDev không bền vững lắm nên sau một thời gian thì các nhà phát triển đã rời bỏ nó.

Tóm lại bản thân của ASP hãy còn một số khuyết điểm quan trọng, nhất là khi chạy trên Internet Information Server với Windows NT 4, ASP không đáng tin cậy lắm.

Tóm lại trong giới lập trình theo Microsoft thì việc lập trình trên desktop cho đến lập trình hệ phân tán hay trên web là không được nhịp nhàng cho lắm. Để chuyển được từ lập trình client hay desktop đến lập trình web là một chặng đường dài.

Nguồn gốc .NET

Đầu năm 1998, sau khi hoàn tất phiên bản Version 4 của Internet Information Server (IIS), các đội ngũ lập trình ở Microsoft nhận thấy họ còn rất nhiều sáng kiến để kiện toàn IIS. Họ bắt đầu xây dựng một kiến trúc mới trên nền tảng ý tưởng đó và đặt tên là Next Generation Windows Services (NGWS).

Sau khi Visual Basic được trình làng vào cuối 1998, dự án kế tiếp mang tên Visual Studio 7 được xác nhập vào NGWS. Đội ngũ COM+/MTS góp vào một universal runtime cho tất cả ngôn ngữ lập trình chung trong Visual Studio, và tham vọng của họ cung cấp cho các ngôn ngữ lập trình của các công ty khác dùng chung luôn. Công việc này được xúc tiến một cách hoàn toàn bí mật mãi cho đến hội nghị Professional Developers' Conference ở Orlando vào tháng 7/2000. Đến tháng 11/2000 thì Microsoft đã phát hành bản Beta 1 của .NET gồm 3 đĩa CD. Tính đến lúc này thì Microsoft đã làm việc với .NET gần 3 năm rồi, do đó bản Beta 1 này tương đối vững chắc.

.NET mang dáng dấp của những sáng kiến đã được áp dụng trước đây như p-code trong UCSD Pascal cho đến Java Virtual Machine. Có điều là Microsoft góp nhặt những sáng kiến của người khác, kết hợp với sáng kiến của chính mình để làm nên một sản phẩm hoàn chỉnh từ bên trong lẫn bên ngoài. Hiện tại Microsoft đã công bố phiên bản release của .NET.

Thật sự Microsoft đã đặt cược vào .NET vì theo thông tin của công ty, đã tập trung 80% sức mạnh của Microsoft để nghiên cứu và triển khai .NET (bao gồm nhân lực và tài chính ?), tất cả các sản phẩm của Microsoft sẽ được chuyển qua .NET.

Microsoft .NET

Tổng quan

Microsoft .NET gồm 2 phần chính : Framework và Integrated Development Environment (IDE). Framework cung cấp những gì cần thiết và căn bản, chữ Framework có nghĩa là khung hay khung cảnh trong đó ta dùng những hạ tầng cơ sở theo một qui ước nhất định để công việc được trôi chảy. IDE thì cung cấp một môi trường giúp chúng ta triển khai dễ dàng, và nhanh chóng các ứng dụng dựa trên nền tảng .NET. Nếu không có IDE chúng ta cũng có thể

dùng một trình soạn thảo ví như Notepad hay bất cứ trình soạn thảo văn bản nào và sử dụng command line để biên dịch và thực thi, tuy nhiên việc này mất nhiều thời gian. Tốt nhất là chúng ta dùng IDE phát triển các ứng dụng, và cũng là cách dễ sử dụng nhất.

Thành phần Framework là quan trọng nhất .NET là cốt lõi và tinh hoa của môi trường, còn IDE chỉ là công cụ để phát triển dựa trên nền tảng đó thôi. Trong .NET toàn bộ các ngôn ngữ C#, Visual C++ hay Visual Basic.NET đều dùng cùng một IDE.

Tóm lại Microsoft .NET là nền tảng cho việc xây dựng và thực thi các ứng dụng phân tán thể hệ kế tiếp. Bao gồm các ứng dụng từ client đến server và các dịch vụ khác. Một số tính năng của Microsoft .NET cho phép những nhà phát triển sử dụng như sau:

Một mô hình lập trình cho phép nhà phát triển xây dựng các ứng dụng dịch vụ web và ứng dụng client với Extensible Markup Language (XML).

Tập hợp dịch vụ XML Web, như Microsoft .NET My Services cho phép nhà phát triển đơn giản và tích hợp người dùng kinh nghiệm.

Cung cấp các server phục vụ bao gồm: Windows 2000, SQL Server, và BizTalk Server, tất cả điều tích hợp, hoạt động, và quản lý các dịch vụ XML Web và các ứng dụng.

Các phần mềm client như Windows XP và Windows CE giúp người phát triển phân phối sâu và thuyết phục người dùng kinh nghiệm thông qua các dòng thiết bị.

Nhiều công cụ hỗ trợ như Visual Studio .NET, để phát triển các dịch vụ Web XML, ứng dụng trên nền Windows hay nền web một cách dễ dàng và hiệu quả.

Kiến trúc .NET Framework

.NET Framework là một platform mới làm đơn giản việc phát triển ứng dụng trong môi trường phân tán của Internet. .NET Framework được thiết kế đầy đủ để đáp ứng theo quan điểm sau:

Để cung cấp một môi trường lập trình hướng đối tượng vững chắc, trong đó mã nguồn đối tượng được lưu trữ và thực thi một cách cục bộ. Thực thi cục bộ nhưng được phân tán trên Internet, hoặc thực thi từ xa.

Để cung cấp một môi trường thực thi mã nguồn mà tối thiểu được việc đóng gói phần mềm và sự tranh chấp về phiên bản.

Để cung cấp một môi trường thực thi mã nguồn mà đảm bảo việc thực thi an toàn mã nguồn, bao gồm cả việc mã nguồn được tạo bởi hãng thứ ba hay bất cứ hãng nào mà tuân thủ theo kiến trúc .NET.

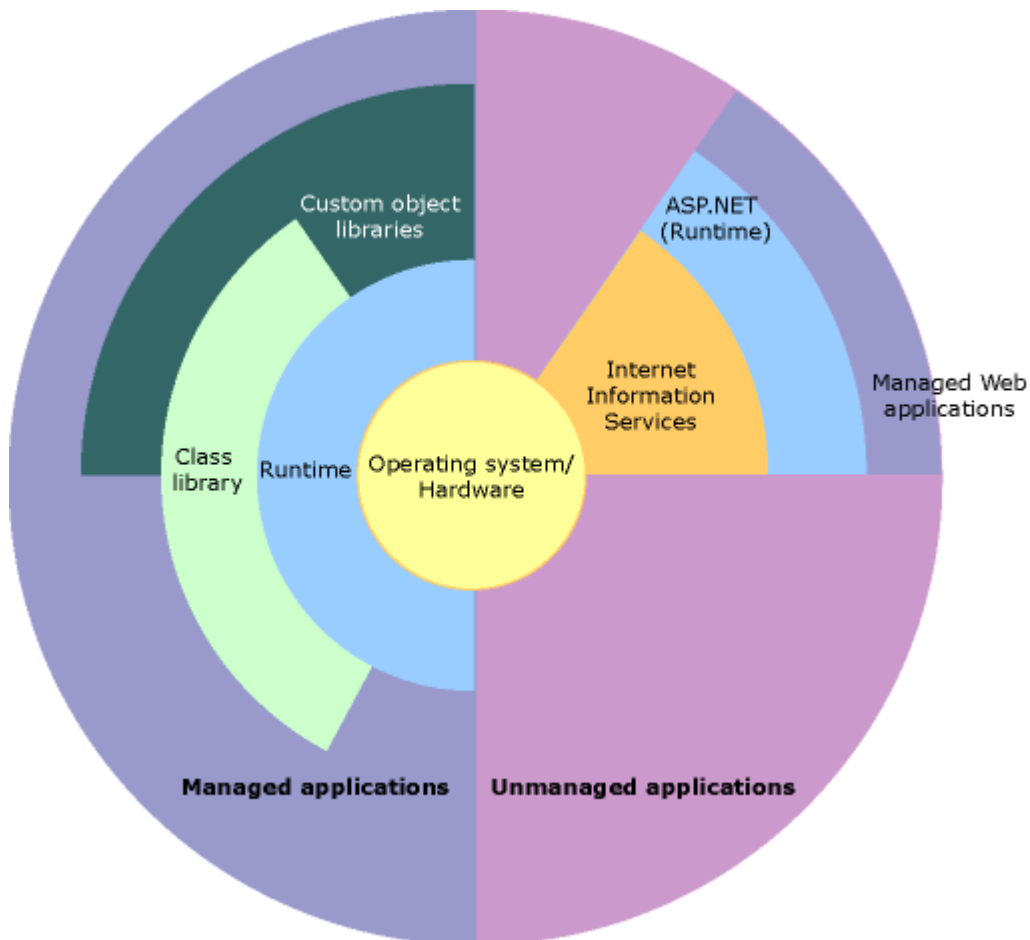
Để cung cấp một môi trường thực thi mã nguồn mà loại bỏ được những lỗi thực hiện các script hay môi trường thông dịch.

Để làm cho những người phát triển có kinh nghiệm vững chắc có thể nắm vững nhiều kiểu ứng dụng khác nhau. Như là từ những ứng dụng trên nền Windows đến những ứng dụng dựa trên web.

Để xây dựng tất cả các thông tin dựa trên tiêu chuẩn công nghiệp để đảm bảo rằng mã nguồn trên .NET có thể tích hợp với bất cứ mã nguồn khác.

.NET Framework có hai thành phần chính: Common Language Runtime (CLR) và thư viện lớp .NET Framework. CLR là nền tảng của .NET Framework. Chúng ta có thể hiểu runtime như là một agent quản lý mã nguồn khi nó được thực thi, cung cấp các dịch vụ cốt lõi như: quản lý bộ nhớ, quản lý tiến trình, và quản lý từ xa. Ngoài ra nó còn thúc đẩy việc sử dụng kiểu an toàn và các hình thức khác của việc chính xác mã nguồn, đảm bảo cho việc thực hiện được bảo mật và mạnh mẽ. Thật vậy, khái niệm quản lý mã nguồn là nguyên lý nền tảng của runtime. Mã nguồn mà đích tới runtime thì được biết như là mã nguồn được quản lý (managed code). Trong khi đó mã nguồn mà không có đích tới runtime thì được biết như mã nguồn không được quản lý (unmanaged code).

Thư viện lớp, một thành phần chính khác của .NET Framework là một tập hợp hướng đối tượng của các kiểu dữ liệu được dùng lại, nó cho phép chúng ta có thể phát triển những ứng dụng từ những ứng dụng truyền thống command-line hay những ứng dụng có giao diện đồ họa (GUI) đến những ứng dụng mới nhất được cung cấp bởi ASP.NET, như là Web Form và dịch vụ XML Web.



Hình 1.1: Mô tả các thành phần trong .NET Framework.

Common Language Runtime (CLR)

Như đã đề cập thì CLR thực hiện quản lý bộ nhớ, quản lý thực thi tiểu trình, thực thi mã nguồn, xác nhận mã nguồn an toàn, biên dịch và các dịch vụ hệ thống khác. Những đặc tính trên là nền tảng cơ bản cho những mã nguồn được quản lý chạy trên CLR.

Do chú trọng đến bảo mật, những thành phần được quản lý được cấp những mức độ quyền hạn khác nhau, phụ thuộc vào nhiều yếu tố nguyên thủy của chúng như: liên quan đến Internet, hệ thống mạng trong nhà máy, hay một máy tính cục bộ. Điều này có nghĩa rằng, một thành phần được quản lý có thể có hay không có quyền thực hiện một thao tác truy cập tập tin, thao tác truy cập registry, hay các chức năng nhạy cảm khác.

CLR thúc đẩy việc mã nguồn thực hiện việc truy cập được bảo mật. Ví dụ, người sử dụng giới hạn rằng việc thực thi nhúng vào trong một trang web có thể chạy được hoạt hình trên màn hình hay hát một bản nhạc, nhưng không thể truy cập được dữ liệu riêng tư, tập tin hệ thống, hay truy cập mạng. Do đó, đặc tính bảo mật của CLR cho phép những phần mềm đóng gói trên Internet có nhiều đặc tính mà không ảnh hưởng đến việc bảo mật hệ thống.

CLR còn thúc đẩy cho mã nguồn được thực thi mạnh mẽ hơn bằng việc thực thi mã nguồn chính xác và sự xác nhận mã nguồn. Nền tảng của việc thực hiện này là Common Type System (CTS). CTS đảm bảo rằng những mã nguồn được quản lý thì được tự mô tả (self-describing). Sự khác nhau giữa Microsoft và các trình biên dịch ngôn ngữ của hãng thứ ba là việc tạo ra các mã nguồn được quản lý có thể thích hợp với CTS. Điều này thì mã nguồn được quản lý có thể sử dụng những kiểu được quản lý khác và những thể hiện, trong khi thúc đẩy nghiêm ngặt việc sử dụng kiểu dữ liệu chính xác và an toàn.

Thêm vào đó, môi trường được quản lý của runtime sẽ thực hiện việc tự động xử lý layout của đối tượng và quản lý những tham chiếu đến đối tượng, giải phóng chúng khi chúng không còn được sử dụng nữa. Việc quản lý bộ nhớ tự động này còn giải quyết hai lỗi chung của ứng dụng: thiếu bộ nhớ và tham chiếu bộ nhớ không hợp lệ.

Trong khi runtime được thiết kế cho những phần mềm của tương lai, nó cũng hỗ trợ cho phần mềm ngày nay và trước đây. Khả năng hoạt động qua lại giữa mã nguồn được quản lý và mã nguồn không được quản lý cho phép người phát triển tiếp tục sử dụng những thành phần cần thiết của COM và DLL.

Runtime được thiết kế để cải tiến hiệu suất thực hiện. Mặc dù CLR cung cấp nhiều các tiêu chuẩn dịch vụ runtime, nhưng mã nguồn được quản lý không bao giờ được dịch. Có một đặc tính gọi là Just-in-Time (JIT) biên dịch tất cả những mã nguồn được quản lý vào trong ngôn ngữ máy của hệ thống vào lúc mà nó được thực thi. Khi đó, trình quản lý bộ nhớ xóa bỏ những phân mảnh bộ nhớ nếu có thể được và gia tăng tham chiếu bộ nhớ cục bộ, và kết quả gia tăng hiệu quả thực thi.

Thư viện lớp .NET Framework

Thư viện lớp .NET Framework là một tập hợp những kiểu dữ liệu được dùng lại và được kết hợp chặt chẽ với Common Language Runtime. Thư viện lớp là hướng đối tượng cung cấp những kiểu dữ liệu mà mã nguồn được quản lý của chúng ta có thể dẫn xuất. Điều này không chỉ làm cho những kiểu dữ liệu của .NET Framework dễ sử dụng mà còn làm giảm thời gian liên quan đến việc học đặc tính mới của .NET Framework. Thêm vào đó, các thành phần của các hãng thứ ba có thể tích hợp với những lớp trong .NET Framework.

Cũng như mong đợi của người phát triển với thư viện lớp hướng đối tượng, kiểu dữ liệu .NET Framework cho phép người phát triển thiết lập nhiều mức độ thông dụng của việc lập trình, bao gồm các nhiệm vụ như: quản lý chuỗi, thu thập hay chọn lọc dữ liệu, kết nối với cơ sở dữ liệu, và truy cập tập tin. Ngoài những nhiệm vụ thông dụng trên. Thư viện lớp còn đưa vào những kiểu dữ liệu để hỗ trợ cho những kịch bản phát triển chuyên biệt khác. Ví dụ người phát triển có thể sử dụng .NET Framework để phát triển những kiểu ứng dụng và dịch vụ như sau:

 Ứng dụng Console

 Ứng dụng giao diện GUI trên Windows (Windows Forms)

 Ứng dụng ASP.NET

 Dịch vụ XML Web

 Dịch vụ Windows

Trong đó những lớp Windows Forms cung cấp một tập hợp lớn các kiểu dữ liệu nhằm làm đơn giản việc phát triển các ứng dụng GUI chạy trên Windows. Còn nếu như viết các ứng dụng ASP.NET thì có thể sử dụng các lớp Web Forms trong thư viện .NET Framework.

Phát triển ứng dụng Client

Những ứng dụng client cũng gần với những ứng dụng kiểu truyền thống được lập trình dựa trên Windows. Đây là những kiểu ứng dụng hiển thị những cửa sổ hay những form trên desktop cho phép người dùng thực hiện một thao tác hay nhiệm vụ nào đó. Những ứng dụng client bao gồm những ứng dụng như xử lý văn bản, xử lý bảng tính, những ứng dụng trong lĩnh vực thương mại như công cụ nhập liệu, công cụ tạo báo cáo... Những ứng dụng client này thường sử dụng những cửa sổ, menu, toolbar, button hay các thành phần GUI khác, và chúng thường truy cập các tài nguyên cục bộ như là các tập tin hệ thống, các thiết bị ngoại vi như máy in.

Một loại ứng dụng client khác với ứng dụng truyền thống như trên là ActiveX control (hiện nay nó được thay thế bởi các Windows Form control) được nhúng vào các trang web trên Internet. Các ứng dụng này cũng giống như những ứng dụng client khác là có thể truy cập tài nguyên cục bộ.

Trong quá khứ, những nhà phát triển có thể tạo các ứng dụng sử dụng C/C++ thông qua kết nối với MFC hoặc sử dụng môi trường phát triển ứng dụng nhanh (RAD: Rapid

Application Development). .NET Framework tích hợp diện mạo của những sản phẩm thành một. Môi trường phát triển cố định làm đơn giản mạnh mẽ sự phát triển của ứng dụng client.

Những lớp .NET Framework chứa trong .NET Framework được thiết kế cho việc sử dụng phát triển các GUI. Điều này cho phép người phát triển nhanh chóng và dễ dàng tạo các cửa sổ, button, menu, toolbar, và các thành phần khác trong các ứng dụng được viết phục vụ cho lĩnh vực thương mại. Ví dụ như, .NET cung cấp những thuộc tính đơn giản để hiệu chỉnh các hiệu ứng visual liên quan đến form. Trong vài trường hợp hệ điều hành không hỗ trợ việc thay đổi những thuộc tính này một cách trực tiếp, và trong trường hợp này .NET tự động tạo lại form. Đây là một trong nhiều cách mà .NET tích hợp việc phát triển giao diện làm cho mã nguồn đơn giản và mạnh mẽ hơn.

Không giống như ActiveX control, Windows Form control có sự truy cập giới hạn đến máy của người sử dụng. Điều này có nghĩa rằng mã nguồn thực thi nhị phân có thể truy cập một vài tài nguyên trong máy của người sử dụng (như các thành phần đồ họa hay một số tập tin được giới hạn) mà không thể truy cập đến những tài nguyên khác. Nguyên nhân là sự bảo mật truy cập của mã nguồn. Lúc này các ứng dụng được cài đặt trên máy người dùng có thể an toàn để đưa lên Internet

Biên dịch và MSIL

Trong .NET Framework, chương trình không được biên dịch vào các tập tin thực thi mà thay vào đó chúng được biên dịch vào những tập tin trung gian gọi là Microsoft Intermediate Language (MSIL). Những tập tin MSIL được tạo ra từ C# cũng tương tự như các tập tin MSIL được tạo ra từ những ngôn ngữ khác của .NET, platform ở đây không cần biết ngôn ngữ của mã nguồn. Điều quan trọng chính yếu của CLR là chung (common), cùng một runtime hỗ trợ phát triển trong C# cũng như trong VB.NET.

Mã nguồn C# được biên dịch vào MSIL khi chúng ta build project. Mã MSIL này được lưu vào trong một tập tin trên đĩa. Khi chúng ta chạy chương trình, thì MSIL được biên dịch một lần nữa, sử dụng trình biên dịch Just-In-Time (JIT). Kết quả là mã máy được thực thi bởi bộ xử lý của máy.

Trình biên dịch JIT tiêu chuẩn thì thực hiện theo yêu cầu. Khi một phương thức được gọi, trình biên dịch JIT phân tích MSIL và tạo ra sản phẩm mã máy có hiệu quả cao, mã này có thể chạy rất nhanh. Trình biên dịch JIT đủ thông minh để nhận ra khi một mã đã được biên dịch, do vậy khi ứng dụng chạy thì việc biên dịch chỉ xảy ra khi cần thiết, tức là chỉ biên dịch mã MSIL chưa biên dịch ra mã máy. Khi đó một ứng dụng .NET thực hiện, chúng có xu hướng là chạy nhanh và nhanh hơn nữa, cũng như là những mã nguồn được biên dịch rồi thì được dùng lại.

Do tất cả các ngôn ngữ .NET Framework cùng tạo ra sản phẩm MSIL giống nhau, nên kết quả là một đối tượng được tạo ra từ ngôn ngữ này có thể được truy cập hay được dẫn xuất từ

một đối tượng của ngôn ngữ khác trong .NET. Ví dụ, người phát triển có thể tạo một lớp cơ sở trong VB.NET và sau đó dẫn xuất nó trong C# một cách dễ dàng.

Ngôn ngữ C#

Ngôn ngữ C# khá đơn giản, chỉ khoảng 80 từ khóa và hơn mười mấy kiểu dữ liệu được xây dựng sẵn. Tuy nhiên, ngôn ngữ C# có ý nghĩa cao khi nó thực thi những khái niệm lập trình hiện đại. C# bao gồm tất cả những hỗ trợ cho cấu trúc, thành phần component, lập trình hướng đối tượng. Những tính chất đó hiện diện trong một ngôn ngữ lập trình hiện đại. Và ngôn ngữ C# hội đủ những điều kiện như vậy, hơn nữa nó được xây dựng trên nền tảng của hai ngôn ngữ mạnh nhất là C++ và Java.

Ngôn ngữ C# được phát triển bởi đội ngũ kỹ sư của Microsoft, trong đó người dẫn đầu là Anders Hejlsberg và Scott Wiltamuth. Cả hai người này đều là những người nổi tiếng, trong đó Anders Hejlsberg được biết đến là tác giả của Turbo Pascal, một ngôn ngữ lập trình PC phổ biến. Và ông đứng đầu nhóm thiết kế Borland Delphi, một trong những thành công đầu tiên của việc xây dựng môi trường phát triển tích hợp (IDE) cho lập trình client/server.

Phần cốt lõi hay còn gọi là trái tim của bất cứ ngôn ngữ lập trình hướng đối tượng là sự hỗ trợ của nó cho việc định nghĩa và làm việc với những lớp. Những lớp thì định nghĩa những kiểu dữ liệu mới, cho phép người phát triển mở rộng ngôn ngữ để tạo mô hình tốt hơn để giải quyết vấn đề. Ngôn ngữ C# chứa những từ khóa cho việc khai báo những kiểu lớp đối tượng mới và những phương thức hay thuộc tính của lớp, và cho việc thực thi đóng gói, kế thừa, và đa hình, ba thuộc tính cơ bản của bất cứ ngôn ngữ lập trình hướng đối tượng.

Trong ngôn ngữ C# mọi thứ liên quan đến khai báo lớp đều được tìm thấy trong phần khai báo của nó. Định nghĩa một lớp trong ngôn ngữ C# không đòi hỏi phải chia ra tập tin header và tập tin nguồn giống như trong ngôn ngữ C++. Hơn thế nữa, ngôn ngữ C# hỗ trợ kiểu XML, cho phép chèn các tag XML để phát sinh tự động các document cho lớp.

C# cũng hỗ trợ giao diện interface, nó được xem như một cam kết với một lớp cho những dịch vụ mà giao diện quy định. Trong ngôn ngữ C#, một lớp chỉ có thể kế thừa từ duy nhất một lớp cha, tức là không cho đa kế thừa như trong ngôn ngữ C++, tuy nhiên một lớp có thể thực thi nhiều giao diện. Khi một lớp thực thi một giao diện thì nó sẽ hứa là nó sẽ cung cấp chức năng thực thi giao diện.

Trong ngôn ngữ C#, những cấu trúc cũng được hỗ trợ, nhưng khái niệm về ngữ nghĩa của nó thay đổi khác với C++. Trong C#, một cấu trúc được giới hạn, là kiểu dữ liệu nhỏ gọn, và khi tạo thể hiện thì nó yêu cầu ít hơn về hệ điều hành và bộ nhớ so với một lớp. Một cấu trúc thì không thể kế thừa từ một lớp hay được kế thừa nhưng một cấu trúc có thể thực thi một giao diện.

Ngôn ngữ C# cung cấp những đặc tính hướng thành phần (component-oriented), như là những thuộc tính, những sự kiện. Lập trình hướng thành phần được hỗ trợ bởi CLR cho phép lưu trữ metadata với mã nguồn cho một lớp. Metadata mô tả cho một lớp, bao gồm những

phương thức và những thuộc tính của nó, cũng như những sự bảo mật cần thiết và những thuộc tính khác. Mã nguồn chứa đựng những logic cần thiết để thực hiện những chức năng của nó. Do vậy, một lớp được biên dịch như là một khối self-contained, nên môi trường hosting biết được cách đọc metadata của một lớp và mã nguồn cần thiết mà không cần những thông tin khác để sử dụng nó.

Một lưu ý cuối cùng về ngôn ngữ C# là ngôn ngữ này cũng hỗ trợ việc truy cập bộ nhớ trực tiếp sử dụng kiểu con trỏ của C++ và từ khóa cho dấu ngoặc [] trong toán tử. Các mã nguồn này là không an toàn (unsafe). Và bộ giải phóng bộ nhớ tự động của CLR sẽ không thực hiện việc giải phóng những đối tượng được tham chiếu bằng sử dụng con trỏ cho đến khi chúng được giải phóng.

Chương 2

NGÔN NGỮ C#

- **Tại sao phải sử dụng ngôn ngữ C#**
 - C# là ngôn ngữ đơn giản
 - C# là ngôn ngữ hiện đại
 - C# là ngôn ngữ hướng đối tượng
 - C# là ngôn ngữ mạnh mẽ
 - C# là ngôn ngữ ít từ khóa
 - C# là ngôn ngữ module hóa
 - C# sẽ là ngôn ngữ phổ biến
- **Ngôn ngữ C# và những ngôn ngữ khác**
- **Các bước chuẩn bị cho chương trình**
- **Chương trình C# đơn giản**
- **Phát triển chương trình minh họa**
- **Câu hỏi & bài tập**

Tại sao phải sử dụng ngôn ngữ C#

Nhiều người tin rằng không cần thiết có một ngôn ngữ lập trình mới. Java, C++, Perl, Microsoft Visual Basic, và những ngôn ngữ khác được nghĩ rằng đã cung cấp tất cả những chức năng cần thiết.

Ngôn ngữ C# là một ngôn ngữ được dẫn xuất từ C và C++, nhưng nó được tạo từ nền tảng phát triển hơn. Microsoft bắt đầu với công việc trong C và C++ và thêm vào những đặc tính mới để làm cho ngôn ngữ này dễ sử dụng hơn. Nhiều trong số những đặc tính này khá giống với những đặc tính có trong ngôn ngữ Java. Không dừng lại ở đó, Microsoft đưa ra một số mục đích khi xây dựng ngôn ngữ này. Những mục đích này được tóm tắt như sau:

- C# là ngôn ngữ đơn giản
- C# là ngôn ngữ hiện đại
- C# là ngôn ngữ hướng đối tượng
- C# là ngôn ngữ mạnh mẽ và mềm dẻo

C# là ngôn ngữ có ít từ khóa
C# là ngôn ngữ hướng module
C# sẽ trở nên phổ biến

C# là ngôn ngữ đơn giản

C# loại bỏ một vài sự phức tạp và rối rắm của những ngôn ngữ như Java và c++, bao gồm việc loại bỏ những macro, những template, đa kế thừa, và lớp cơ sở ảo (virtual base class). Chúng là những nguyên nhân gây ra sự nhầm lẫn hay dẫn đến những vấn đề cho các người phát triển C++. Nếu chúng ta là người học ngôn ngữ này đầu tiên thì chắc chắn là ta sẽ không trải qua những thời gian để học nó! Nhưng khi đó ta sẽ không biết được hiệu quả của ngôn ngữ C# khi loại bỏ những vấn đề trên.

Ngôn ngữ C# đơn giản vì nó dựa trên nền tảng C và C++. Nếu chúng ta thân thiện với C và C++ hoặc thậm chí là Java, chúng ta sẽ thấy C# khá giống về diện mạo, cú pháp, biểu thức, toán tử và những chức năng khác được lấy trực tiếp từ ngôn ngữ C và C++, nhưng nó đã được cải tiến để làm cho ngôn ngữ đơn giản hơn. Một vài trong các sự cải tiến là loại bỏ các dư thừa, hay là thêm vào những cú pháp thay đổi. Ví dụ như, trong C++ có ba toán tử làm việc với các thành viên là ::, ., và ->. Để biết khi nào dùng ba toán tử này cũng phức tạp và dễ nhầm lẫn. Trong C#, chúng được thay thế với một toán tử duy nhất gọi là . (dot). Đối với người mới học thì điều này và những việc cải tiến khác làm bớt nhầm lẫn và đơn giản hơn.

Ghi chú: Nếu chúng ta đã sử dụng Java và tin rằng nó đơn giản, thì chúng ta cũng sẽ tìm thấy rằng C# cũng đơn giản. Hầu hết mọi người đều không tin rằng Java là ngôn ngữ đơn giản. Tuy nhiên, C# thì dễ hơn là Java và C++.

C# là ngôn ngữ hiện đại

Điều gì làm cho một ngôn ngữ hiện đại? Những đặc tính như là xử lý ngoại lệ, thu gom bộ nhớ tự động, những kiểu dữ liệu mở rộng, và bảo mật mã nguồn là những đặc tính được mong đợi trong một ngôn ngữ hiện đại. C# chứa tất cả những đặc tính trên. Nếu là người mới học lập trình có thể chúng ta sẽ cảm thấy những đặc tính trên phức tạp và khó hiểu. Tuy nhiên, cũng đừng lo lắng chúng ta sẽ dần dần được tìm hiểu những đặc tính qua các chương trong cuốn sách này.

Ghi chú: Con trỏ được tích hợp vào ngôn ngữ C++. Chúng cũng là nguyên nhân gây ra những rắc rối của ngôn ngữ này. C# loại bỏ những phức tạp và rắc rối phát sinh bởi con trỏ. Trong C#, bộ thu gom bộ nhớ tự động và kiểu dữ liệu an toàn được tích hợp vào ngôn ngữ, sẽ loại bỏ những vấn đề rắc rối của C++.

C# là ngôn ngữ hướng đối tượng

Những đặc điểm chính của ngôn ngữ hướng đối tượng (Object-oriented language) là sự đóng gói (encapsulation), sự kế thừa (inheritance), và đa hình (polymorphism). C# hỗ trợ tất

cả những đặc tính trên. Phần hướng đối tượng của C# sẽ được trình bày chi tiết trong một chương riêng ở phần sau.

C# là ngôn ngữ mạnh mẽ và cũng mềm dẻo

Như đã đề cập trước, với ngôn ngữ C# chúng ta chỉ bị giới hạn ở chính bởi bản thân hay là trí tưởng tượng của chúng ta. Ngôn ngữ này không đặt những ràng buộc lên những việc có thể làm. C# được sử dụng cho nhiều các dự án khác nhau như là tạo ra ứng dụng xử lý văn bản, ứng dụng đồ họa, bản tính, hay thậm chí những trình biên dịch cho các ngôn ngữ khác.

C# là ngôn ngữ ít từ khóa

C# là ngôn ngữ sử dụng giới hạn những từ khóa. Phần lớn các từ khóa được sử dụng để mô tả thông tin. Chúng ta có thể nghĩ rằng một ngôn ngữ có nhiều từ khóa thì sẽ mạnh hơn. Điều này không phải sự thật, ít nhất là trong trường hợp ngôn ngữ C#, chúng ta có thể tìm thấy rằng ngôn ngữ này có thể được sử dụng để làm bất cứ nhiệm vụ nào. Bảng sau liệt kê các từ khóa của ngôn ngữ C#.

abstract	<u>default</u>	<u>foreach</u>	<u>object</u>	<u>sizeof</u>	<u>unsafe</u>
as	<u>delegate</u>	<u>goto</u>	<u>operator</u>	<u>stackalloc</u>	<u>ushort</u>
base	<u>do</u>	<u>if</u>	<u>out</u>	<u>static</u>	<u>using</u>
bool	<u>double</u>	<u>implicit</u>	<u>override</u>	<u>string</u>	<u>virtual</u>
break	<u>else</u>	<u>in</u>	<u>params</u>	<u>struct</u>	<u>volatile</u>
byte	<u>enum</u>	<u>int</u>	<u>private</u>	<u>switch</u>	<u>void</u>
case	<u>event</u>	<u>interface</u>	<u>protected</u>	<u>this</u>	<u>while</u>
catch	<u>explicit</u>	<u>internal</u>	<u>public</u>	<u>throw</u>	
char	<u>extern</u>	<u>is</u>	<u>readonly</u>	<u>true</u>	
checked	<u>false</u>	<u>lock</u>	<u>ref</u>	<u>try</u>	
class	<u>finally</u>	<u>long</u>	<u>return</u>	<u>typeof</u>	
const	<u>fixed</u>	<u>namespace</u>	<u>sbyte</u>	<u>uint</u>	
continue	<u>float</u>	<u>new</u>	<u>sealed</u>	<u>ulong</u>	
decimal	<u>for</u>	<u>null</u>	<u>short</u>	<u>unchecked</u>	

Bảng 1.2: Từ khóa của ngôn ngữ C#.

C# là ngôn ngữ hướng module

Mã nguồn C# có thể được viết trong những phần được gọi là những lớp, những lớp này chứa các phương thức thành viên của nó. Những lớp và những phương thức có thể được sử dụng lại trong ứng dụng hay các chương trình khác. Bằng cách truyền các mẫu thông tin đến những lớp hay phương thức chúng ta có thể tạo ra những mã nguồn dùng lại có hiệu quả.

C# sẽ là một ngôn ngữ phổ biến

C# là một trong những ngôn ngữ lập trình mới nhất. Vào thời điểm cuốn sách này được viết, nó không được biết như là một ngôn ngữ phổ biến. Nhưng ngôn ngữ này có một số lý do để trở thành một ngôn ngữ phổ biến. Một trong những lý do chính là Microsoft và sự cam kết của .NET

Microsoft muốn ngôn ngữ C# trở nên phổ biến. Mặc dù một công ty không thể làm một sản phẩm trở nên phổ biến, nhưng nó có thể hỗ trợ. Cách đây không lâu, Microsoft đã gặp sự thất bại về hệ điều hành Microsoft Bob. Mặc dù Microsoft muốn Bob trở nên phổ biến nhưng thất bại. C# thay thế tốt hơn để đem đến thành công sơ với Bob. Thật sự là không biết khi nào mọi người trong công ty Microsoft sử dụng Bob trong công việc hằng ngày của họ. Tuy nhiên, với C# thì khác, nó được sử dụng bởi Microsoft. Nhiều sản phẩm của công ty này đã chuyển đổi và viết lại bằng C#. Bằng cách sử dụng ngôn ngữ này Microsoft đã xác nhận khả năng của C# cần thiết cho những người lập trình.

Microsoft .NET là một lý do khác để đem đến sự thành công của C#. .NET là một sự thay đổi trong cách tạo và thực thi những ứng dụng.

Ngoài hai lý do trên ngôn ngữ C# cũng sẽ trở nên phổ biến do những đặc tính của ngôn ngữ này được đề cập trong mục trước như: đơn giản, hướng đối tượng, mạnh mẽ... **Ngôn ngữ C# và những ngôn ngữ khác**

Chúng ta đã từng nghe đến những ngôn ngữ khác như Visual Basic, C++ và Java. Có lẽ chúng ta cũng tự hỏi sự khác nhau giữa ngôn ngữ C# và những ngôn ngữ đó. Và cũng tự hỏi tại sao lại chọn ngôn ngữ này để học mà không chọn một trong những ngôn ngữ kia. Có rất nhiều lý do và chúng ta hãy xem một số sự so sánh giữa ngôn ngữ C# với những ngôn ngữ khác giúp chúng ta phần nào trả lời được những thắc mắc.

Microsoft nói rằng C# mang đến sức mạnh của ngôn ngữ C++ với sự dễ dàng của ngôn ngữ Visual Basic. Có thể nó không dễ như Visual Basic, nhưng với phiên bản Visual Basic.NET (Version 7) thì ngang nhau. Bởi vì chúng được viết lại từ một nền tảng. Chúng ta có thể viết nhiều chương trình với ít mã nguồn hơn nếu dùng C#.

Mặc dù C# loại bỏ một vài các đặc tính của C++, nhưng bù lại nó tránh được những lỗi mà thường gặp trong ngôn ngữ C++. Điều này có thể tiết kiệm được hàng giờ hay thậm chí hàng ngày trong việc hoàn tất một chương trình. Chúng ta sẽ hiểu nhiều về điều này trong các chương của giáo trình.

Một điều quan trọng khác với C++ là mã nguồn C# không đòi hỏi phải có tập tin header. Tất cả mã nguồn được viết trong khai báo một lớp.

Như đã nói ở bên trên. .NET runtime trong C# thực hiện việc thu gom bộ nhớ tự động. Do điều này nên việc sử dụng con trỏ trong C# ít quan trọng hơn trong C++. Những con trỏ cũng có thể được sử dụng trong C#, khi đó những đoạn mã nguồn này sẽ được đánh dấu là không an toàn (unsafe code).

C# cũng từ bỏ ý tưởng đa kế thừa như trong C++. Và sự khác nhau khác là C# đưa thêm thuộc tính vào trong một lớp giống như trong Visual Basic. Và những thành viên của lớp được gọi duy nhất bằng toán tử “.” khác với C++ có nhiều cách gọi trong các tình huống khác nhau.

Một ngôn ngữ khác rất mạnh và phổ biến là Java, giống như C++ và C# được phát triển dựa trên C. Nếu chúng ta quyết định sẽ học Java sau này, chúng ta sẽ tìm được nhiều cái mà học từ C# có thể được áp dụng.

Điểm giống nhau C# và Java là cả hai cùng biên dịch ra mã trung gian: C# biên dịch ra MSIL còn Java biên dịch ra bytecode. Sau đó chúng được thực hiện bằng cách thông dịch hoặc biên dịch just-in-time trong từng máy ảo tương ứng. Tuy nhiên, trong ngôn ngữ C# nhiều hỗ trợ được đưa ra để biên dịch mã ngôn ngữ trung gian sang mã máy. C# chứa nhiều kiểu dữ liệu cơ bản hơn Java và cũng cho phép nhiều sự mở rộng với kiểu dữ liệu giá trị. Ví dụ, ngôn ngữ C# hỗ trợ kiểu liệt kê (enumerator), kiểu này được giới hạn đến một tập hằng được định nghĩa trước, và kiểu dữ liệu cấu trúc đây là kiểu dữ liệu giá trị do người dùng định nghĩa. Chúng ta sẽ được tìm hiểu kỹ hơn về kiểu dữ liệu tham chiếu và kiểu dữ liệu giá trị sẽ được trình bày trong phần sau

Tương tự như Java, C# cũng từ bỏ tính đa kế thừa trong một lớp, tuy nhiên mô hình kế thừa đơn này được mở rộng bởi tính đa kế thừa nhiều giao diện. ***Các bước chuẩn bị cho chương trình***

Thông thường, trong việc phát triển phần mềm, người phát triển phải tuân thủ theo quy trình phát triển phần mềm một cách nghiêm ngặt và quy trình này đã được chuẩn hóa. Tuy nhiên trong phạm vi của chúng ta là tìm hiểu một ngôn ngữ mới và viết những chương trình nhỏ thì không đòi hỏi khắt khe việc thực hiện theo quy trình. Nhưng để giải quyết được những vấn đề thì chúng ta cũng cần phải thực hiện đúng theo các bước sau. Đầu tiên là phải xác định vấn đề cần giải quyết. Nếu không biết rõ vấn đề thì ta không thể tìm được phương pháp giải quyết. Sau khi xác định được vấn đề, thì chúng ta có thể nghĩ ra các kế hoạch để thực hiện. Sau khi có một kế hoạch, thì có thể thực thi kế hoạch này. Sau khi kế hoạch được thực thi, chúng ta phải kiểm tra lại kết quả để xem vấn đề được giải quyết xong chưa. Logic này thường được áp dụng trong nhiều lĩnh vực khác nhau, trong đó có lập trình.

Khi tạo một chương trình trong C# hay bất cứ ngôn ngữ nào, chúng ta nên theo những bước tuần tự sau:

Xác định mục tiêu của chương trình.

Xác định những phương pháp giải quyết vấn đề.

Tạo một chương trình để giải quyết vấn đề.

Thực thi chương trình để xem kết quả.


Ví dụ mục tiêu để viết chương trình xử lý văn bản đơn giản, mục tiêu chính là xây dựng chương trình cho phép soạn thảo và lưu trữ những chuỗi ký tự hay văn bản. Nếu không có mục tiêu thì không thể viết được chương trình hiệu quả.

Bước thứ hai là quyết định đến phương pháp để viết chương trình. Bước này xác định những thông tin nào cần thiết được sử dụng trong chương trình, các hình thức nào được sử dụng. Từ những thông tin này chúng ta rút ra được phương pháp để giải quyết vấn đề.

Bước thứ ba là bước cài đặt, ở bước này có thể dùng các ngôn ngữ khác nhau để cài đặt, tuy nhiên, ngôn ngữ phù hợp để giải quyết vấn đề một cách tốt nhất sẽ được chọn. Trong phạm vi của sách này chúng ta mặc định là dùng C#, đơn giản là chúng ta đang tìm hiểu nó! Và bước cuối cùng là phần thực thi chương trình để xem kết quả.

Chương trình C# đơn giản

Để bắt đầu cho việc tìm hiểu ngôn ngữ C# và tạo tiền đề cho các chương sau, chương đầu tiên trình bày một chương trình C# đơn giản nhất.

 Ví dụ 2.1 : Chương trình C# đầu tiên.

```
class ChaoMung
{
    static void Main( )
    {
        // Xuất ra màn hình
        System.Console.WriteLine("Chao Mung");
    }
}
```

Kết quả:

Chao Mung

Sau khi viết xong chúng ta lưu dưới dạng tập tin có phần mở rộng *.cs (C sharp). Sau đó biên dịch và chạy chương trình. Kết quả là một chuỗi "Chao Mung" sẽ xuất hiện trong màn hình console.

Các mục sau sẽ giới thiệu xoay quanh ví dụ 2.1, còn phần chi tiết từng loại sẽ được trình bày trong các chương kế tiếp.

Lớp, đối tượng và kiểu dữ liệu (type)

Điều cốt lõi của lập trình hướng đối tượng là tạo ra các kiểu mới. Kiểu là một thứ được xem như trừu tượng. Nó có thể là một bảng dữ liệu, một tiểu trình, hay một nút lệnh trong một cửa sổ. Tóm lại kiểu được định nghĩa như một dạng vừa có thuộc tính chung (properties) và các hành vi ứng xử (behavior) của nó.

Nếu trong một ứng dụng trên Windows chúng ta tạo ra ba nút lệnh OK, Cancel, Help, thì thực chất là chúng ta đang dùng ba thể hiện của một kiểu nút lệnh trong Windows và các nút này cùng chia sẻ các thuộc tính và hành vi chung với nhau. Ví dụ, các nút có các thuộc tính như kích thước, vị trí, nhãn tên (label), tuy nhiên mỗi thuộc tính của một thể hiện không nhất thiết phải giống nhau, và thường thì chúng khác nhau, như nút OK có nhãn là “OK”, Cancel có nhãn là “Cancel”...Ngoài ra các nút này có các hành vi ứng xử chung như khả năng vẽ, kích hoạt, đáp ứng các thông điệp nhấn,...Tùy theo từng chức năng đặc biệt riêng của từng loại thì nội dung ứng xử khác nhau, nhưng tất cả chúng được xem như là cùng một kiểu.

Cũng như nhiều ngôn ngữ lập trình hướng đối tượng khác, kiểu trong C# được định nghĩa là một lớp (class), và các thể hiện riêng của từng lớp được gọi là đối tượng (object). Trong các chương kế tiếp sẽ trình bày các kiểu khác nhau ngoài kiểu lớp như kiểu liệt kê, cấu trúc và kiểu ủy quyền (delegates).

Quay lại chương trình ChaoMung trên, chương trình này chỉ có một kiểu đơn giản là lớp ChaoMung. Để định nghĩa một kiểu lớp trong C# chúng ta phải dùng từ khoá class, tiếp sau là tên lớp trong ví dụ trên tên lớp là ChaoMung. Sau đó định nghĩa các thuộc tính và hành động cho lớp. Thuộc tính và hành động phải nằm trong dấu { }.

Ghi chú: Khai báo lớp trong C# không có dấu ; sau ngoặc } cuối cùng của lớp. Và khác với lớp trong C/C++ là chia thành 2 phần header và phần định nghĩa. Trong C# , định nghĩa một lớp được gói gọn trong dấu { } sau tên lớp và trong cùng một tập tin.

Phương thức

Hai thành phần chính cấu thành một lớp là thuộc tính hay tính chất và phương thức hay còn gọi là hành động ứng xử của đối tượng. Trong C# hành vi được định nghĩa như một phương thức thành viên của lớp.

Phương thức chính là các hàm được định nghĩa trong lớp. Do đó, ta còn có thể gọi các phương thức thành viên là các hàm thành viên trong một lớp. Các phương thức này chỉ ra rằng các hành động mà lớp có thể làm được cùng với cách thức làm hành động đó. Thông thường, tên của phương thức thường được đặt theo tên hành động, ví dụ như DrawLine() hay GetString().

Tuy nhiên trong ví dụ 2.1 vừa trình bày, chúng ta có hàm thành viên là Main() hàm này là hàm đặc biệt, không mô tả hành động nào của lớp hết, nó được xác định là hàm đầu vào của lớp (entry point) và được CLR gọi đầu tiên khi thực thi.

Ghi chú: Trong C#, hàm Main() được viết ký tự hoa đầu, và có thể trả về giá trị void hay int

Khi chương trình thực thi, CLR gọi hàm Main() đầu tiên, hàm Main() là đầu vào của chương trình, và mỗi chương trình phải có một hàm Main(). Đôi khi chương trình có nhiều hàm Main() nhưng lúc này ta phải xác định các chỉ dẫn biên dịch để CLR biết đâu là hàm Main() đầu vào duy nhất trong chương trình.

Việc khai báo phương thức được xem như là một sự giao ước giữa người tạo ra lớp và người sử dụng lớp này. Người xây dựng các lớp cũng có thể là người dùng lớp đó, nhưng không hoàn toàn như vậy. Vì có thể các lớp này được xây dựng thành các thư viện chuẩn và cung cấp cho các nhóm phát triển khác... Do vậy việc tuân thủ theo các qui tắc là rất cần thiết.

Để khai báo một phương thức, phải xác định kiểu giá trị trả về, tên của phương thức, và cuối cùng là các tham số cần thiết cho phương thức thực hiện.

Chú thích


Một chương trình được viết tốt thì cần phải có chú thích các đoạn mã được viết. Các đoạn chú thích này sẽ không được biên dịch và cũng không tham gia vào chương trình. Mục đích chính là làm cho đoạn mã nguồn rõ ràng và dễ hiểu.

Trong ví dụ 2.1 có một dòng chú thích :

```
// Xuất ra màn hình.
```

Một chuỗi chú thích trên một dòng thì bắt đầu bằng ký tự “//”. Khi trình biên dịch gặp hai ký tự này thì sẽ bỏ qua dòng đó.

Ngoài ra C# còn cho phép kiểu chú thích cho một hay nhiều dòng, và ta phải khai báo “/*” ở phần đầu chú thích và kết thúc chú thích là ký tự “*/”.

 Ví dụ 2.2 : Minh họa dùng chú thích trên nhiều dòng.

```
class ChaoMung
{
    static void Main()
    {
        /* Xuất ra màn hình chuỗi 'chao mung'
        Su dung ham WriteLine cua lop System.Console
        */
        System.Console.WriteLine("Chao Mung");
    }
}
```

Kết quả:

Chao Mung

Ngoài hai kiểu chú thích trên giống trong C/C++ thì C# còn hỗ trợ thêm kiểu thứ ba cũng là kiểu cuối cùng, kiểu này chứa các định dạng XML nhằm xuất ra tập tin XML khi biên dịch để tạo sơ liệu cho mã nguồn. Chúng ta sẽ bàn kiểu này trong các chương trình ở các phần tiếp.

Ứng dụng Console

Ví dụ đơn giản trên được gọi là ứng dụng console, ứng dụng này giao tiếp với người dùng thông qua bàn phím và không có giao diện người dùng (UI), giống như các ứng dụng thường thấy trong Windows. Trong các chương xây dựng các ứng dụng nâng cao trên Windows hay Web thì ta mới dùng các giao diện đồ họa. Còn để tìm hiểu về ngôn ngữ C# thuần túy thì cách tốt nhất là ta viết các ứng dụng console.

Trong hai ứng dụng đơn giản trên ta đã dùng phương thức `WriteLine()` của lớp `Console`. Phương thức này sẽ xuất ra màn hình dòng lệnh hay màn hình DOS chuỗi tham số đưa vào, cụ thể là chuỗi “Chào Mừng”.

Namespace

Như chúng ta đã biết .NET cung cấp một thư viện các lớp đồ sộ và thư viện này có tên là FCL (Framework Class Library). Trong đó `Console` chỉ là một lớp nhỏ trong hàng ngàn lớp trong thư viện. Mỗi lớp có một tên riêng, vì vậy FCL có hàng ngàn tên như `ArrayList`, `Dictionary`, `FileSelector`,...

Điều này làm nảy sinh vấn đề, người lập trình không thể nào nhớ hết được tên của các lớp trong .NET Framework. Tệ hơn nữa là sau này có thể ta tạo lại một lớp trùng với lớp đã có chẳng hạn. Ví dụ trong quá trình phát triển một ứng dụng ta cần xây dựng một lớp từ điển và lấy tên là `Dictionary`, và điều này dẫn đến sự tranh chấp khi biên dịch vì C# chỉ cho phép một tên duy nhất.

Chắc chắn rằng khi đó chúng ta phải đổi tên của lớp từ điển mà ta vừa tạo thành một cái tên khác chẳng hạn như `myDictionary`. Khi đó sẽ làm cho việc phát triển các ứng dụng trở nên phức tạp, cồng kềnh. Đến một sự phát triển nhất định nào đó thì chính là cơn ác mộng cho nhà phát triển.

Giải pháp để giải quyết vấn đề này là việc tạo ra một namespace, namespace sẽ hạn chế phạm vi của một tên, làm cho tên này chỉ có ý nghĩa trong vùng đã định nghĩa.

Giả sử có một người nói Tùng là một kỹ sư, từ kỹ sư phải đi kèm với một lĩnh vực nhất định nào đó, vì nếu không thì chúng ta sẽ không biết được là anh ta là kỹ sư cầu đường, cơ khí hay phần mềm. Khi đó một lập trình viên C# sẽ bảo rằng Tùng là `CauDuong.KySu` phân biệt với `CoKhi.KySu` hay `PhanMem.KySu`. Namespace trong trường hợp này là `CauDuong`, `CoKhi`, `PhanMem` sẽ hạn chế phạm vi của những từ theo sau. Nó tạo ra một vùng không gian để tên sau đó có nghĩa.

Tương tự như vậy ta cứ tạo các namespace để phân thành các vùng cho các lớp trùng tên không tranh chấp với nhau.

Tương tự như vậy, .NET Framework có xây dựng một lớp `Dictionary` bên trong namespace `System.Collections`, và tương ứng ta có thể tạo một lớp `Dictionary` khác nằm trong namespace `ProgramCSharp.DataStructures`, điều này hoàn toàn không dẫn đến sự tranh chấp với nhau.

Trong ví dụ minh họa 1.2 đối tượng Console bị hạn chế bởi namespace bằng việc sử dụng mã lệnh:

```
System.Console.WriteLine();
```

Toán tử ‘.’

Trong ví dụ 2.2 trên dấu ‘.’ được sử dụng để truy cập đến phương thức hay dữ liệu trong một lớp (trong trường hợp này phương thức là WriteLine()), và ngăn cách giữa tên lớp đến một namespace xác nhận (namespace System và lớp là Console). Việc thực hiện này theo hướng từ trên xuống, trong đó mức đầu tiên namespace là System, tiếp theo là lớp Console, và cuối cùng là truy cập đến các phương thức hay thuộc tính của lớp.

Trong nhiều trường hợp namespace có thể được chia thành các namespace con gọi là subnamespace. Ví dụ trong namespace System có chứa một số các subnamespace như Configuration, Collections, Data, và còn rất nhiều nữa, hơn nữa trong namespace Collection còn chia thành nhiều namespace con nữa.

Namespace giúp chúng ta tổ chức và ngăn cách những kiểu. Khi chúng ta viết một chương trình C# phức tạp, chúng ta có thể phải tạo một kiến trúc namespace riêng cho mình, và không giới hạn chiều sâu của cây phân cấp namespace. Mục đích của namespace là giúp chúng ta chia để quản lý những kiến trúc đối tượng phức tạp.

Từ khóa using

Để làm cho chương trình gọn hơn, và không cần phải viết từng namespace cho từng đối tượng, C# cung cấp từ khóa là using, sau từ khóa này là một namespace hay subnamespace với mô tả đầy đủ trong cấu trúc phân cấp của nó.

Ta có thể dùng dòng lệnh :

```
using System;
```

ở đầu chương trình và khi đó trong chương trình nếu chúng ta có dùng đối tượng Console thì không cần phải viết đầy đủ : System.Console. mà chỉ cần viết Console. thôi.



Ví dụ 2.3: Dùng khóa using


```
using System;
class ChaoMung
{
    static void Main()
    {
        //Xuất ra màn hình chuỗi thông báo
        Console.WriteLine("Chao Mung");
    }
}
```

Kết quả:

Chao Mung

Lưu ý rằng phải đặt câu `using System` trước định nghĩa lớp `ChaoMung`.

Mặc dù chúng ta chỉ định rằng chúng ta sử dụng namespace `System`, và không giống như các ngôn ngữ khác, không thể chỉ định rằng chúng ta sử dụng đối tượng `System.Console`.

 Ví dụ 2.4: Không hợp lệ trong C#.

```
using System.Console;
class ChaoMung
{
    static void Main()
    {
        //Xuất ra màn hình chuỗi thông báo
        WriteLine("Chao Mung");
    }
}
```

Đoạn chương trình trên khi biên dịch sẽ được thông báo một lỗi như sau:

error CS0138: A using namespace directive can only be applied to namespace;
'System.Console' is a class not a namespace.

Cách biểu diễn namespace có thể làm giảm nhiều thao tác gõ bàn phím, nhưng nó có thể sẽ không đem lại lợi ích nào bởi vì nó có thể làm xáo trộn những namespace có tên không khác nhau. Giải pháp chung là chúng ta sử dụng từ khóa **using** với các namespace đã được xây dựng sẵn, các namespace do chúng ta tạo ra, những namespace này chúng ta đã nắm chắc sự liệu về nó. Còn đối với namespace do các hãng thứ ba cung cấp thì chúng ta không nên dùng từ khóa **using**.

Phân biệt chữ thường và chữ hoa

Cũng giống như C/C++, C# là ngôn ngữ phân biệt chữ thường với chữ hoa, điều này có nghĩa rằng hai câu lệnh `writeLine` thì khác với `WriteLine` và cũng khác với `WRITELINE`.

Đáng tiếc là C# không giống như VB, môi trường phát triển C# sẽ không tự sửa các lỗi này, nếu chúng ta viết hai chữ với cách khác nhau thì chúng ta có thể đưa vào chương trình gỡ rối tìm ra các lỗi này.

Để tránh việc lãng phí thời gian và công sức, người ta phát triển một số quy ước cho cách đặt tên biến, hằng, hàm, và nhiều định danh khác nữa. Quy ước trong giáo trình này dùng cú pháp lạc đà (camel notation) cho tên biến và cú pháp Pascal cho hàm, hằng, và thuộc tính.

Ví dụ :

Biến myDictionary theo cách đặt tên cú pháp lạc đà.

Hàm DrawLine, thuộc tính ColorBackground theo cách đặt tên cú pháp Pascal.

Từ khóa static

Hàm Main() trong ví dụ minh họa trên có nhiều hơn một cách thiết kế. Trong minh họa này hàm Main() được khai báo với kiểu trả về là void, tức là hàm này không trả về bất cứ giá trị nào cả. Đôi khi cần kiểm tra chương trình có thực hiện đúng hay không, người lập trình có thể khai báo hàm Main() trả về một giá trị nào đó để xác định kết quả thực hiện của chương trình. Trong khai báo của ví dụ trên có dùng từ khóa **static**:

```
static void Main()
{
    .....
}
```

Từ khóa này chỉ ra rằng hàm Main() có thể được gọi mà không cần phải tạo đối tượng ChaoMung. Những vấn đề liên quan đến khai báo lớp, phương thức, hay thuộc tính sẽ được trình bày chi tiết trong các chương tiếp theo.

Phát triển chương trình minh họa

Có tối thiểu là hai cách để soạn thảo, biên dịch và thực thi chương trình trong cuốn sách này:

Sử dụng môi trường phát triển tích hợp (IDE) Visual Studio .NET

Sử dụng chương trình soạn thảo văn bản bất kỳ như Notepad rồi dùng biên dịch dòng lệnh.

Mặc dù chúng ta có thể phát triển phần mềm bên ngoài Visual Studio .NET, IDE cung cấp nhiều các tiện ích hỗ trợ cho người phát triển như: hỗ trợ phần soạn thảo mã nguồn như canh lề, màu sắc, tích hợp các tập tin trợ giúp, các đặc tính intellisense,... Nhưng điều quan trọng nhất là IDE phải có công cụ debug mạnh và một số công cụ trợ giúp phát triển ứng dụng khác.

Trong cuốn sách này giả sử rằng người đọc đang sử dụng Visual Studio .NET. Phần trình này sẽ tập trung vào ngôn ngữ và platform hơn là công cụ phát triển. Chúng ta có thể sao chép tất cả những mã nguồn ví dụ vào trong một chương trình soạn thảo văn bản như Notepad hay Emacs, lưu chúng dưới dạng tập tin văn bản, và biên dịch chúng bằng trình biên dịch dòng lệnh C#, chương trình này được phân phối cùng .NET Framework SDK. Trong những chương cuối về xây dựng các ứng dụng trên Windows và Web, chúng ta sẽ sử dụng công cụ Visual Studio .NET để tạo ra các Windows Form và Web Form, tuy nhiên chúng ta cũng có thể viết bằng tay trong Notepad nếu chúng ta quyết định sử dụng cách làm bằng tay thay vì dùng công cụ thiết kế.

Sử dụng Notepad soạn thảo

Đầu tiên chúng ta sẽ mở chương trình Notepad rồi soạn thảo chương trình minh họa trên, lưu ý là ta có thể sử dụng bất cứ trình soạn thảo văn bản nào chứ không nhất thiết là Notepad. Sau khi soạn thảo xong thì lưu tập tin xuống đĩa và tập tin này có phần mở rộng là *.cs, trong ví dụ này là chaomung.cs. Bước tiếp theo là biên dịch tập tin nguồn vừa tạo ra. Để biên dịch ta dùng trình biên dịch dòng lệnh C# (csc.exe) chương trình này được chép vào máy trong quá trình cài .NET Framework. Để biết csc.exe nằm chính xác vị trí nào trong đĩa ta có thể dùng chức năng tìm kiếm của Windows.

Để thực hiện biên dịch chúng ta mở một cửa sổ dòng lệnh rồi đánh vào lệnh theo mẫu sau:

```
csc.exe [/out: <file thực thi>] <file nguồn>
```

Ví dụ: csc.exe /out:d:\chaomung.exe d:\chaomung.cs

Thường thì khi biên dịch ta chỉ cần hai phần là tên của trình biên dịch và tên tập tin nguồn mà thôi. Trong mẫu trên có dùng một trong nhiều tùy chọn khi biên dịch là /out, theo sau là tên của chương trình thực thi hay chính là kết quả biên dịch tập tin nguồn.

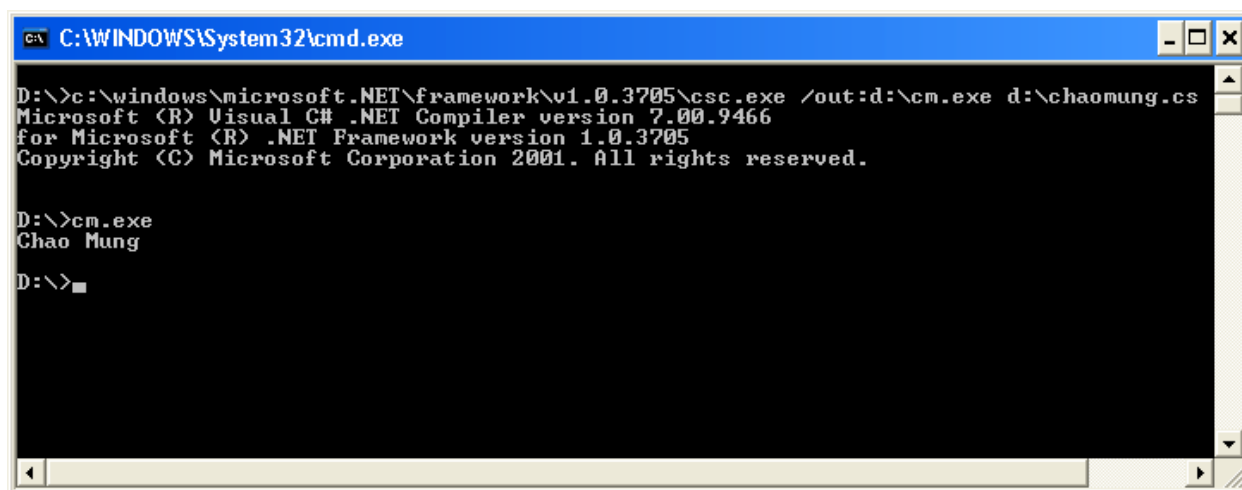
Các tham số tùy chọn có rất nhiều nếu muốn tìm hiểu chúng ta có thể dùng lệnh:

```
csc.exe /?
```

Lệnh này xuất ra màn hình toàn bộ các tùy chọn biên dịch và các hướng dẫn sử dụng.

Hai hình sau minh họa quá trình nhập mã nguồn chương trình C# bằng một trình soạn thảo văn bản đơn giản như Notepad trong Windows. Và sau đó biên dịch tập tin mã nguồn vừa tạo ra bằng chương trình csc.exe một trình biên dịch dòng lệnh của C#. Kết quả là một tập tin thực thi được tạo ra và ta sẽ chạy chương trình này.

Hình 2.2: Mã nguồn được soạn thảo trong Notepad.



```

C:\WINDOWS\System32\cmd.exe
D:\>c:\windows\microsoft.NET\Framework\v1.0.3705\csc.exe /out:d:\cm.exe d:\chaomung.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

D:\>cm.exe
Chao Mung

D:\>

```

Hình 2.3: Biên dịch và thực thi chương trình.

Sử dụng Visual Studio .NET để tạo chương trình

Để tạo chương trình chào mừng trong IDE, lựa chọn mục Visual Studio .NET trong menu Start hoặc icon của nó trên desktop, sau khi khởi động xong chương trình, chọn tiếp chức năng File New Project trong menu. Chức năng này sẽ gọi cửa sổ New Project (hình 2.4 bên dưới). Nếu như chương trình Visual Studio .NET được chạy lần đầu tiên, khi đó cửa sổ New Project sẽ xuất hiện tự động mà không cần phải kích hoạt.

Để tạo ứng dụng, ta lựa chọn mục Visual C# Projects trong cửa sổ Project Type bên trái. Lúc này chúng ta có thể nhập tên cho ứng dụng và lựa chọn thư mục nơi lưu trữ các tập tin này. Cuối cùng, kích vào OK khi mọi chuyện khởi tạo đã chấm dứt và một cửa sổ mới sẽ xuất hiện (hình 2.4 bên dưới), chúng ta có thể nhập mã nguồn vào đây.

Lưu ý rằng Visual Studio .NET tạo ra một namespace dựa trên tên của project mà ta vừa cung cấp (ChaoMung), và thêm vào chỉ dẫn sử dụng namespace System bằng lệnh **using**, bởi hầu như mọi chương trình mà chúng ta viết đều cần sử dụng các kiểu dữ liệu chứa trong namespace System.

Hình 2.4: Tạo ứng dụng C# console trong Visual Studio .NET.

Hình 2.5: Phân soạn thảo mã nguồn cho project.

Visual Studio .NET tạo một lớp tên là Class1, lớp này chúng ta có thể tùy ý đổi tên của chúng. Khi đổi tên của lớp, tốt nhất là đổi tên luôn tập tin chứa lớp đó (Class1.cs). Giả sử

trong ví dụ trên chúng ta đổi tên của lớp thành ChaoMung, và đổi tên tập tin Class1.cs (đổi tên tập tin trong cửa sổ Solution Explorer).

Cuối cùng, Visual Studio .NET tạo một khung sườn chương trình, và kết thúc với chú thích TODO là vị trí bắt đầu của chúng ta. Để tạo chương trình chào mừng trong minh họa trên, ta bỏ tham số string[] args của hàm Main() và xóa tất cả các chú thích bên trong của hàm. Sau đó nhập vào dòng lệnh sau bên trong thân của hàm Main()


```
// Xuất ra màn hình
```

```
System.Console.WriteLine("Chao Mung");
```

Sau tất cả công việc đó, tiếp theo là phần biên dịch chương trình từ Visual Studio .NET. Thông thường để thực hiện một công việc nào đó ta có thể chọn kích hoạt chức năng trong menu, hay các button trên thanh toolbar, và cách nhanh nhất là sử dụng các phím nóng hay các phím kết hợp để gọi nhanh một chức năng.

Trong ví dụ, để biên dịch chương trình nhấn *Ctrl-Shift-B* hoặc chọn chức năng:

Build Build Solution. Một cách khác nữa là dùng nút lệnh trên thanh toolbar: 

Để chạy chương trình vừa được tạo ra mà không sử dụng chế độ debug chúng ta có thể nhấn *Ctrl-F5* hay chọn Debug Start Without Debugging hoặc nút lệnh  trên thanh toolbar của Visual Studio .NET

Ghi chú: Tốt hơn hết là chúng ta nên bỏ ra nhiều thời gian để tìm hiểu hay khám phá môi trường phát triển Visual Studio .NET. Đây cũng là cách thức tốt mà những người phát triển ứng dụng và chúng ta nên thực hiện. Việc tìm hiểu Visual Studio .NET và thông thạo nó sẽ giúp cho chúng ta rất nhiều trong quá trình xây dựng và phát triển ứng dụng sau này. **Câu hỏi và trả lời**

Câu hỏi 1: Một chương trình C# có thể chạy trên bất cứ máy nào?

Trả lời 1: Không phải tất cả. Một chương trình C# chỉ chạy trên máy có Common Language Runtime (CLR) được cài đặt. Nếu chúng ta copy một chương trình exe của C# qua một máy không có CLR thì chúng ta sẽ nhận được một lỗi. Trong những phiên bản của Windows không có CLR chúng ta sẽ được báo rằng thiếu tập tin DLL.

Câu hỏi 2: Nếu muốn đưa chương trình mà ta viết cho một người bạn thì tập tin nào mà chúng ta cần đưa?

Trả lời 2: Thông thường cách tốt nhất là đưa chương trình đã biên dịch. Điều này có nghĩa rằng sau khi mã nguồn được biên dịch, chúng ta sẽ có một chương trình thực thi (tập tin có phần mở rộng *.exe). Như vậy, nếu chúng ta muốn đưa chương trình Chaomung cho tất cả những người bạn của chúng ta thì chỉ cần đưa tập tin Chaomung.exe. Không cần thiết phải đưa tập tin nguồn Chaomung.cs. Và những người bạn của chúng ta không cần thiết phải có trình biên dịch C#. Họ chỉ cần có C# runtime trên máy tính (như CLR của Microsoft) là có thể chạy được chương trình của chúng ta.

Câu hỏi 3: Sau khi tạo ra được tập tin thực thi .exe. Có cần thiết giữ lại tập tin nguồn không?

Trả lời 3: Nếu chúng ta từ bỏ tập tin mã nguồn thì sau này sẽ rất khó khăn cho việc mở rộng hay thay đổi chương trình, do đó cần thiết phải giữ lại các tập tin nguồn. Hầu hết các IDE tạo ra các tập tin nguồn (.cs) và các tập tin thực thi. Cũng như giữ các tập tin nguồn chúng ta cũng cần thiết phải giữ các tập tin khác như là các tài nguyên bên ngoài các icon, image, form.. Chúng ta sẽ lưu giữ những tập tin này trong trường hợp chúng ta cần thay đổi hay tạo lại tập tin thực thi.

Câu hỏi 4: Nếu trình biên dịch C# đưa ra một trình soạn thảo, có phải nhất thiết phải sử dụng nó?

Trả lời 4: Không hoàn toàn như vậy. Chúng ta có thể sử dụng bất cứ trình soạn thảo văn bản nào và lưu mã nguồn dưới dạng tập tin văn bản. Nếu trình biên dịch đưa ra một trình soạn thảo thì chúng ta nên sử dụng nó. Nếu chúng ta có một trình soạn thảo khác tốt hơn chúng ta có thể sử dụng nó. Một số các tiện ích soạn thảo mã nguồn có thể giúp cho ta dễ dàng tìm các lỗi cú pháp, giúp tạo một số mã nguồn tự động đơn giản...Nói chung là tùy theo chúng ta nhưng theo tôi thì Visual Studio .NET cũng khá tốt để sử dụng

Câu hỏi 5: Có thể không quan tâm đến những cảnh báo khi biên dịch mã nguồn

Trả lời 5: Một vài cảnh báo không ảnh hưởng đến chương trình khi chạy, nhưng một số khác có thể ảnh hưởng đến chương trình chạy. Nếu trình biên dịch đưa ra cảnh báo, tức là tín hiệu cho một thứ gì đó không đúng. Hầu hết các trình biên dịch cho phép chúng ta thiết lập mức độ cảnh báo. Bằng cách thiết lập mức độ cảnh báo chúng ta có thể chỉ quan tâm đến những cảnh báo nguy hiểm, hay nhận hết tất cả những cảnh báo. Nói chung cách tốt nhất là chúng ta nên xem tất cả những cảnh báo để sửa chữa chúng, một chương trình tạm gọi là đạt yêu cầu khi không có lỗi biên dịch và cũng không có cảnh báo (nhưng chưa chắc đã chạy đúng kết quả!).

Câu hỏi thêm

Câu hỏi 1: Hãy đưa ra 3 lý do tại sao ngôn ngữ C# là một ngôn ngữ lập trình tốt?

Câu hỏi 2: IL và CLR viết tắt cho từ nào và ý nghĩa của nó?

Câu hỏi 3: Đưa ra các bước cơ bản trong chu trình xây dựng chương trình?

Câu hỏi 4: Trong biên dịch dòng lệnh thì lệnh nào được sử dụng để biên dịch mã nguồn .cs và lệnh này gọi chương trình nào?

Câu hỏi 5: Phần mở rộng nào mà chúng ta nên sử dụng cho tập tin mã nguồn C#?

Câu hỏi 6: Một tập tin .txt chứa mã nguồn C# có phải là một tập tin mã nguồn C# hợp lệ hay không? Có thể biên dịch được hay không?

Câu hỏi 7: Ngôn ngữ máy là gì? Khi biên dịch mã nguồn C# ra tập tin .exe thì tập tin này là ngôn ngữ gì?

Câu hỏi 8: Nếu thực thi một chương trình đã biên dịch và nó không thực hiện đúng như mong đợi của chúng ta, thì điều gì chúng ta cần phải làm?

Câu hỏi 9: Một lỗi tương tự như bên dưới thường xuất hiện khi nào?

mycode.cs(15,5): error CS1010: NewLine in constan

Câu hỏi 10: Tại sao phải khai báo static cho hàm Main của lớp?

Câu hỏi 11: Một mã nguồn C# có phải chứa trong các lớp hay là có thể tồn tại bên ngoài lớp như C/C++?

Câu hỏi 12: So sánh sự khác nhau cơ bản giữa C# và C/C++, C# với Java, hay bất cứ ngôn ngữ cấp cao nào mà bạn đã biết?

Câu hỏi 13: Con trỏ có còn được sử dụng trong C# hay không? Nếu có thì nó được quản lý như thế nào?

Câu hỏi 14: Khái niệm và ý nghĩa của namespace trong C#? Điều gì xảy ra nếu như ngôn ngữ lập trình không hỗ trợ namespace?

Bài tập

Bài tập 1: Dùng trình soạn thảo văn bản mở chương trình exe mà ta đã biên dịch từ các chương trình nguồn trước và xem sự khác nhau giữa hai tập tin này, lưu ý sao khi đóng tập tin này ta không chọn lưu tập tin.

Bài tập 2: Nhập vào chương trình sau và biên dịch nó. Cho biết chương trình thực hiện điều gì?

```
using System;
class variables
{
    public static void Main()
    {
        int radius = 4;
        const double PI = 3.14159;
        double circum, area;
        area = PI * radius* radius;
        circum = 2 * PI * radius;
        // in kết quả
        Console.WriteLine("Ban kinh = {0}, PI = {1}", radius, PI);
        Console.WriteLine("Dien tich {0}", area);
        Console.WriteLine("Chu vi {0}", circum);
    }
}
```

Bài tập 3: Nhập vào chương trình sau và biên dịch. Cho biết chương trình thực hiện điều gì?

```
class AClass
```

```

{
    static void Main()
    {
        int x, y;
        for( x = 0; x < 10; x++, System.Console.Write("\n"));
        for( y = 0 ; y < 10; y++, System.Console.WriteLine("{0}",y));
    }
}

```

Bài tập 4: Chương trình sau có chứa lỗi. Nhập vào và sửa những lỗi đó

Bài tập 5: Sửa lỗi và biên dịch chương trình sau

```

class Test
{
    pubic static void Main()
    {

        Console.WriteLine("Xin chào");
        Consoile.WriteLine("Tam biet");

    }
}

```

Bài tập 6: Sửa lỗi và biên dịch chương trình sau

```

class Test
{
    pubic void Main()
    {

        Console.WriteLine('Xin chào');
        Consoile.WriteLine('Tam biet');

    }
}

```

Bài tập 7: Viết chương trình xuất ra bài thơ:

Rằm Tháng Giêng

Rằm xuân lồng lộng trăng soi,
Sông xuân nước lẫn màu trời thêm xuân.
Giữa dòng bàn bạc việc quân
Khuya về bát ngát trăng ngân đầy thuyền.

Hồ Chí Minh.

Chương 3

NỀN TẢNG NGÔN NGỮ C#

- **Kiểu dữ liệu**
 - Kiểu dữ liệu xây dựng sẵn
 - Chọn kiểu dữ liệu
 - Chuyển đổi các kiểu dữ liệu
- **Biến và hằng**
 - Gán giá trị xác định cho biến
 - Hằng
 - Kiểu liệt kê
 - Kiểu chuỗi ký tự
 - Định danh
- **Biểu thức**
- **Khoảng trắng**
- **Câu lệnh**
 - Phân nhánh không có điều kiện
 - Phân nhánh có điều kiện
 - Câu lệnh lặp
- **Toán tử**
- **Namespace**
- **Các chỉ dẫn biên dịch**
- **Câu hỏi & bài tập**

Trong chương trước chúng ta đã tìm hiểu một chương trình C# đơn giản nhất. Chương trình đó chưa đủ để diễn tả một chương trình viết bằng ngôn ngữ C#, có quá nhiều phần và chi tiết đã bỏ qua. Do vậy trong chương này chúng ta sẽ đi sâu vào tìm hiểu cấu trúc và cú pháp của ngôn ngữ C#.

Chương này sẽ thảo luận về hệ thống kiểu dữ liệu, phân biệt giữa kiểu dữ liệu xây dựng sẵn (như int, bool, string...) với kiểu dữ liệu do người dùng định nghĩa (lớp hay cấu trúc do người lập trình tạo ra...). Một số cơ bản khác về lập trình như tạo và sử dụng biến dữ liệu hay hằng cũng được đề cập cùng với cấu trúc liệt kê, chuỗi, định danh, biểu thức và câu lệnh.

Trong phần hai của chương hướng dẫn và minh họa việc sử dụng lệnh phân nhánh **if**, **switch**, **while**, **do...while**, **for**, và **foreach**. Và các toán tử như phép gán, phép toán logic, phép toán quan hệ, và toán học...

Như chúng ta đã biết C# là một ngôn ngữ hướng đối tượng rất mạnh, và công việc của người lập trình là kế thừa để tạo và khai thác các đối tượng. Do vậy để nắm vững và phát triển tốt người lập trình cần phải đi từ những bước đi đầu tiên tức là đi vào tìm hiểu những phần cơ bản và cốt lõi nhất của ngôn ngữ.

Kiểu dữ liệu

C# là ngôn ngữ lập trình mạnh về kiểu dữ liệu, một ngôn ngữ mạnh về kiểu dữ liệu là phải khai báo kiểu của mỗi đối tượng khi tạo (kiểu số nguyên, số thực, kiểu chuỗi, kiểu điều khiển...) và trình biên dịch sẽ giúp cho người lập trình không bị lỗi khi chỉ cho phép một loại kiểu dữ liệu có thể được gán cho các kiểu dữ liệu khác. Kiểu dữ liệu của một đối tượng là một tín hiệu để trình biên dịch nhận biết kích thước của một đối tượng (kiểu int có kích thước là 4 byte) và khả năng của nó (như một đối tượng button có thể vẽ, phản ứng khi nhấn,...).

Tương tự như C++ hay Java, C# chia thành hai tập hợp kiểu dữ liệu chính: Kiểu xây dựng sẵn (built-in) mà ngôn ngữ cung cấp cho người lập trình và kiểu được người dùng định nghĩa (user-defined) do người lập trình tạo ra.

C# phân tập hợp kiểu dữ liệu này thành hai loại: Kiểu dữ liệu giá trị (value) và kiểu dữ liệu tham chiếu (reference). Việc phân chi này do sự khác nhau khi lưu kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu trong bộ nhớ. Đối với một kiểu dữ liệu giá trị thì sẽ được lưu giữ kích thước thật trong bộ nhớ đã cấp phát là stack. Trong khi đó thì địa chỉ của kiểu dữ liệu tham chiếu thì được lưu trong stack nhưng đối tượng thật sự thì lưu trong bộ nhớ heap.

Nếu chúng ta có một đối tượng có kích thước rất lớn thì việc lưu giữ chúng trên bộ nhớ heap rất có ích, trong chương 4 sẽ trình bày những lợi ích và bất lợi khi làm việc với kiểu dữ liệu tham chiếu, còn trong chương này chỉ tập trung kiểu dữ liệu cơ bản hay kiểu xây dựng sẵn.

Ghi chú: Tất cả các kiểu dữ liệu xây dựng sẵn là kiểu dữ liệu giá trị ngoại trừ các đối tượng và chuỗi. Và tất cả các kiểu do người dùng định nghĩa ngoại trừ kiểu cấu trúc đều là kiểu dữ liệu tham chiếu.

Ngoài ra C# cũng hỗ trợ một kiểu con trỏ C++, nhưng hiếm khi được sử dụng, và chỉ khi nào làm việc với những đoạn mã lệnh không được quản lý (unmanaged code). Mã lệnh không được quản lý là các lệnh được viết bên ngoài nền .MS.NET, như là các đối tượng COM.

Kiểu dữ liệu xây dựng sẵn

Ngôn ngữ C# đưa ra các kiểu dữ liệu xây dựng sẵn rất hữu dụng, phù hợp với một ngôn ngữ lập trình hiện đại, mỗi kiểu dữ liệu được ánh xạ đến một kiểu dữ liệu được hỗ trợ bởi hệ thống xác nhận ngôn ngữ chung (Common Language Specification: CLS) trong MS.NET. Việc ánh xạ các kiểu dữ liệu nguyên thủy của C# đến các kiểu dữ liệu của .NET sẽ đảm bảo các đối tượng được tạo ra trong C# có thể được sử dụng đồng thời với các đối tượng được tạo bởi bất cứ ngôn ngữ khác được biên dịch bởi .NET, như VB.NET.

Mỗi kiểu dữ liệu có một sự xác nhận và kích thước không thay đổi, không giống như C++, int trong C# luôn có kích thước là 4 byte bởi vì nó được ánh xạ từ kiểu Int32 trong .NET.

Bảng 3.1 sau sẽ mô tả một số các kiểu dữ liệu được xây dựng sẵn

Kiểu C#	Số byte	Kiểu .NET	Mô tả
byte	1	Byte	Số nguyên dương không dấu từ 0-255
char	2	Char	Ký tự Unicode
bool	1	Boolean	Giá trị logic true/ false
sbyte	1	Sbyte	Số nguyên có dấu (từ -128 đến 127)
short	2	Int16	Số nguyên có dấu giá trị từ -32768 đến 32767.
ushort	2	UInt16	Số nguyên không dấu 0 – 65.535
int	4	Int32	Số nguyên có dấu –2.147.483.647 và 2.147.483.647
uint	4	UInt32	Số nguyên không dấu 0 – 4.294.967.295
float	4	Single	Kiểu dấu chấm động, giá trị xấp xỉ từ 3,4E-38 đến 3,4E+38, với 7 chữ số có nghĩa..
double	8	Double	Kiểu dấu chấm động có độ chính xác gấp đôi, giá trị xấp xỉ từ 1,7E-308 đến 1,7E+308, với 15,16 chữ số có nghĩa.
decimal	8	Decimal	Có độ chính xác đến 28 con số và giá trị thập phân, được dùng trong tính toán tài chính, kiểu này đòi hỏi phải có hậu tố “m” hay “M” theo sau giá trị.

long	8	Int64	Kiểu số nguyên có dấu có giá trị trong khoảng : -9.223.370.036.854.775.808 đến 9.223.372.036.854.775.807
ulong	8	UInt64	Số nguyên không dấu từ 0 đến 0xffffffffffffffff

Bảng 3.1 : Mô tả các kiểu dữ liệu xây dựng sẵn.

Ghi chú: Kiểu giá trị logic chỉ có thể nhận được giá trị là true hay false mà thôi. Một giá trị nguyên không thể gán vào một biến kiểu logic trong C# và không có bất cứ chuyển đổi ngầm định nào. Điều này khác với C/C++, cho phép biến logic được gán giá trị nguyên, khi đó giá trị nguyên 0 là false và các giá trị còn lại là true.

Chọn kiểu dữ liệu

Thông thường để chọn một kiểu dữ liệu nguyên để sử dụng như short, int hay long thường dựa vào độ lớn của giá trị muốn sử dụng. Ví dụ, một biến ushort có thể lưu giữ giá trị từ 0 đến 65.535, trong khi biến ulong có thể lưu giữ giá trị từ 0 đến 4.294.967.295, do đó tùy vào miền giá trị của phạm vi sử dụng biến mà chọn các kiểu dữ liệu thích hợp nhất. Kiểu dữ liệu int thường được sử dụng nhiều nhất trong lập trình vì với kích thước 4 byte của nó cũng đủ để lưu các giá trị nguyên cần thiết.

Kiểu số nguyên có dấu thường được lựa chọn sử dụng nhiều nhất trong kiểu số trừ khi có lý do chính đáng để sử dụng kiểu dữ liệu không dấu.

Stack và Heap

Stack là một cấu trúc dữ liệu lưu trữ thông tin dạng xếp chồng tức là vào sau ra trước (Last In First Out : LIFO), điều này giống như chúng ta có một chồng các đĩa, ta cứ xếp các đĩa vào chồng và khi lấy ra thì đĩa nào nằm trên cùng sẽ được lấy ra trước, tức là đĩa vào sau sẽ được lấy ra trước.

Trong C#, kiểu giá trị như kiểu số nguyên được cấp phát trên stack, đây là vùng nhớ được thiết lập để lưu các giá trị, và vùng nhớ này được tham chiếu bởi tên của biến.

Kiểu tham chiếu như các đối tượng thì được cấp phát trên heap. Khi một đối tượng được cấp phát trên heap thì địa chỉ của nó được trả về, và địa chỉ này được gán đến một tham chiếu.

Thình thoảng cơ chế thu gom sẽ hủy đối tượng trong stack sau khi một vùng trong stack được đánh dấu là kết thúc. Thông thường một vùng trong stack được định nghĩa bởi một hàm. Do đó, nếu chúng ta khai báo một biến cục bộ trong một hàm là một đối tượng thì đối tượng này sẽ đánh dấu để hủy khi kết thúc hàm.

Những đối tượng trên heap sẽ được thu gom sau khi một tham chiếu cuối cùng đến đối tượng đó được gọi.

Cách tốt nhất khi sử dụng biến không dấu là giá trị của biến luôn luôn dương, biến này thường thể hiện một thuộc tính nào đó có miền giá trị dương. Ví dụ khi cần khai báo một biến lưu giữ tuổi của một người thì ta dùng kiểu byte (số nguyên từ 0-255) vì tuổi của người không thể nào âm được.

Kiểu float, double, và decimal đưa ra nhiều mức độ khác nhau về kích thước cũng như độ chính xác. Với thao tác trên các phân số nhỏ thì kiểu float là thích hợp nhất. Tuy nhiên lưu ý rằng trình biên dịch luôn luôn hiểu bất cứ một số thực nào cũng là một số kiểu double trừ khi chúng ta khai báo rõ ràng. Để gán một số kiểu float thì số phải có ký tự f theo sau.

```
float soFloat = 24f;
```

Kiểu dữ liệu ký tự thể hiện các ký tự Unicode, bao gồm các ký tự đơn giản, ký tự theo mã Unicode và các ký tự thoát khác được bao trong những dấu nháy đơn. Ví dụ, A là một ký tự đơn giản trong khi \u0041 là một ký tự Unicode. Ký tự thoát là những ký tự đặc biệt bao gồm hai ký tự liên tiếp trong đó ký tự đầu tiên là dấu chéo '\'. Ví dụ, \t là dấu tab. Bảng 3.2 trình bày các ký tự đặc biệt.

Ký tự	Ý nghĩa
\'	Dấu nháy đơn
\"	Dấu nháy kép
\\	Dấu chéo
\0	Ký tự null
\a	Alert

\b	Backspace
\f	Sang trang form feed
\n	Dòng mới
\r	Đầu dòng
\t	Tab ngang
\v	Tab dọc

Bảng 3.2 : Các kiểu ký tự đặc biệt.

Chuyển đổi các kiểu dữ liệu

Những đối tượng của một kiểu dữ liệu này có thể được chuyển sang những đối tượng của một kiểu dữ liệu khác thông qua cơ chế chuyển đổi tường minh hay ngầm định. Chuyển đổi ngầm định được thực hiện một cách tự động, trình biên dịch sẽ thực hiện công việc này. Còn chuyển đổi tường minh diễn ra khi chúng ta gán ép một giá trị cho kiểu dữ liệu khác.

Việc chuyển đổi giá trị ngầm định được thực hiện một cách tự động và đảm bảo là không mất thông tin. Ví dụ, chúng ta có thể gán ngầm định một số kiểu short (2 byte) vào một số kiểu int (4 byte) một cách ngầm định. Sau khi gán hoàn toàn không mất dữ liệu vì bất cứ giá trị nào của short cũng thuộc về int:

```
short x = 10;
int y = x; // chuyển đổi ngầm định
```

Tuy nhiên, nếu chúng ta thực hiện chuyển đổi ngược lại, chắc chắn chúng ta sẽ bị mất thông tin. Nếu giá trị của số nguyên đó lớn hơn 32.767 thì nó sẽ bị cắt khi chuyển đổi. Trình biên dịch sẽ không thực hiện việc chuyển đổi ngầm định từ số kiểu int sang số kiểu short:

```
short x;
int y = 100;
x = y; // Không biên dịch, lỗi !!!
```


Để không bị lỗi chúng ta phải dùng lệnh gán tường minh, đoạn mã trên được viết lại như sau:

```
short x;
int y = 500;
x = (short) y; // Ép kiểu tường minh, trình biên dịch không báo lỗi
```

Biến và hằng

Một biến là một vùng lưu trữ với một kiểu dữ liệu. Trong ví dụ trước cả x, và y đều là biến. Biến có thể được gán giá trị và cũng có thể thay đổi giá trị khi thực hiện các lệnh trong chương trình.

Để tạo một biến chúng ta phải khai báo kiểu của biến và gán cho biến một tên duy nhất. Biến có thể được khởi tạo giá trị ngay khi được khai báo, hay nó cũng có thể được gán một giá trị mới vào bất cứ lúc nào trong chương trình. Ví dụ 3.1 sau minh họa sử dụng biến.

 Ví dụ 3.1: Khởi tạo và gán giá trị đến một biến.

```

class MinhHoaC3
{
    static void Main()
    {
        int bien1 = 9;
        System.Console.WriteLine("Sau khi khai tao: bien1 ={0}", bien1);
        bien1 = 15;
        System.Console.WriteLine("Sau khi gan: bien1 ={0}", bien1);
    }
}

```

Kết quả:

```

Sau khi khai tao: bien1 = 9
Sau khi gan: bien1 = 15

```


Ngay khi khai báo biến ta đã gán giá trị là 9 cho biến, khi xuất biến này thì biến có giá trị là 9. Thực hiện phép gán biến cho giá trị mới là 15 thì biến sẽ có giá trị là 15 và xuất kết quả là 15.

Gán giá trị xác định cho biến

C# đòi hỏi các biến phải được khởi tạo trước khi được sử dụng. Để kiểm tra luật này chúng ta thay đổi dòng lệnh khởi tạo biến bien1 trong ví dụ 3.1 như sau:

```
int bien1;
```

và giữ nguyên phần còn lại ta được ví dụ 3.2:

 *Ví dụ 3.2: Sử dụng một biến không khởi tạo.*

```

class MinhHoaC3
{
    static void Main()
    {
        int bien1;
        System.Console.WriteLine("Sau khi khai tao: bien1 ={0}", bien1);
        bien1 = 15;
        System.Console.WriteLine("Sau khi gan: bien1 ={0}", bien1);
    }
}

```


Khi biên dịch đoạn chương trình trên thì trình biên dịch C# sẽ thông báo một lỗi sau:

...error CS0165: Use of unassigned local variable 'bien1'

Việc sử dụng biến khi chưa được khởi tạo là không hợp lệ trong C#. Ví dụ 3.2 trên không hợp lệ.

Tuy nhiên không nhất thiết lúc nào chúng ta cũng phải khởi tạo biến. Nhưng để dùng được thì bắt buộc phải gán cho chúng một giá trị trước khi có một lệnh nào tham chiếu đến biến đó. Điều này được gọi là gán giá trị xác định cho biến và C# bắt buộc phải thực hiện điều này.

Ví dụ 3.3 minh họa một chương trình đúng.

 *Ví dụ 3.3: Biến không được khi tạo nhưng sau đó được gán giá trị.*

```
class MinhHoaC3
{
    static void Main()
    {
        int bien1;
        bien1 = 9;
        System.Console.WriteLine("Sau khi khai tạo: bien1 = {0}", bien1);
        bien1 = 15;
        System.Console.WriteLine("Sau khi gán: bien1 = {0}", bien1);
    }
}
```

Hằng

Hằng cũng là một biến nhưng giá trị của hằng không thay đổi. Biến là công cụ rất mạnh, tuy nhiên khi làm việc với một giá trị được định nghĩa là không thay đổi, ta phải đảm bảo giá trị của nó không được thay đổi trong suốt chương trình. Ví dụ, khi lập một chương trình thí nghiệm hóa học liên quan đến nhiệt độ sôi, hay nhiệt độ đông của nước, chương trình cần khai báo hai biến là DoSôi và DoDong, nhưng không cho phép giá trị của hai biến này bị thay đổi hay bị gán. Để ngăn ngừa việc gán giá trị khác, ta phải sử dụng biến kiểu hằng.

Hằng được phân thành ba loại: giá trị hằng (literal), biểu tượng hằng (symbolic constants), kiểu liệt kê (enumerations).

Giá trị hằng: ta có một câu lệnh gán như sau:

```
x = 100;
```

Giá trị 100 là giá trị hằng. Giá trị của 100 luôn là 100. Ta không thể gán giá trị khác cho 100 được.

Biểu tượng hằng: gán một tên cho một giá trị hằng, để tạo một biểu tượng hằng dùng từ khóa **const** và cú pháp sau:

```
<const> <type> <tên hằng> = <giá trị>;
```

Một biểu tượng hằng phải được khởi tạo khi khai báo, và chỉ khởi tạo duy nhất một lần trong suốt chương trình và không được thay đổi. Ví dụ:

```
const int DoSoi = 100;
```

Trong khai báo trên, 32 là một hằng số và DoSoi là một biểu tượng hằng có kiểu nguyên. Ví dụ 3.4 minh họa việc sử dụng những biểu tượng hằng.

 Ví dụ 3.4: Sử dụng biểu tượng hằng.

```
class MinhHoaC3
{
    static void Main()
    {
        const int DoSoi = 100; // Độ C
        const int DoDong = 0;  // Độ C
        System.Console.WriteLine( "Do dong cua nuoc {0}", DoDong );
        System.Console.WriteLine( "Do soi cua nuoc {0}", DoSoi );
    }
}
```

Kết quả:

```
Do dong cua nuoc 0
Do soi cua nuoc 100
```

Ví dụ 3.4 tạo ra hai biểu tượng hằng chứa giá trị nguyên: DoSoi và DoDong, theo qui tắc đặt tên hằng thì tên hằng thường được đặt theo cú pháp Pascal, nhưng điều này không đòi hỏi bởi ngôn ngữ nên ta có thể đặt tùy ý.

Việc dùng biểu thức hằng này sẽ làm cho chương trình được viết tăng thêm phần ý nghĩa cùng với sự dễ hiểu. Thật sự chúng ta có thể dùng hằng số là 0 và 100 thay thế cho hai biểu tượng hằng trên, nhưng khi đó chương trình không được dễ hiểu và không được tự nhiên lắm. Trình biên dịch không bao giờ chấp nhận một lệnh gán giá trị mới cho một biểu tượng hằng. Ví dụ 3.4 trên có thể được viết lại như sau

...

```
class MinhHoaC3
{
    static void Main()
    {
        const int DoSoi = 100; // Độ C
        const int DoDong = 0;  // Độ C
        System.Console.WriteLine( "Do dong cua nuoc {0}", DoDong );
    }
}
```

```

        System.Console.WriteLine( "Do soi cua nuoc {0}", DoSoi );
        DoSoi = 200;
    }
}

```

Khi đó trình biên dịch sẽ phát sinh một lỗi sau:

```

error CS0131: The left-hand side of an assignment must be a variable,
property or indexer.

```

Kiểu liệt kê

Kiểu liệt kê đơn giản là tập hợp các tên hằng có giá trị không thay đổi (thường được gọi là danh sách liệt kê).

Trong ví dụ 3.4, có hai biểu tượng hằng có quan hệ với nhau:

```

const int DoDong = 0;
const int DoSoi = 100;

```

Do mục đích mở rộng ta mong muốn thêm một số hằng số khác vào danh sách trên, như các hằng sau:

```

const int DoNong = 60;
const int DoAm = 40;
const int DoNguoi = 20;

```

Các biểu tượng hằng trên đều có ý nghĩa quan hệ với nhau, cùng nói về nhiệt độ của nước, khi khai báo từng hằng trên có vẻ cồng kềnh và không được liên kết chặt chẽ cho lắm. Thay vào đó C# cung cấp kiểu liệt kê để giải quyết vấn đề trên:

```

enum NhietDoNuoc
{
    DoDong = 0,
    DoNguoi = 20,
    DoAm = 40,
    DoNong = 60,
    DoSoi = 100,
}

```

Mỗi kiểu liệt kê có một kiểu dữ liệu cơ sở, kiểu dữ liệu có thể là bất cứ kiểu dữ liệu nguyên nào như int, short, long... tuy nhiên kiểu dữ liệu của liệt kê không chấp nhận kiểu ký tự. Để khai báo một kiểu liệt kê ta thực hiện theo cú pháp sau:

```

[thuộc tính] [bổ sung] enum <tên liệt kê> [:kiểu cơ sở] {danh sách các thành phần
liệt kê};

```

Thành phần thuộc tính và bổ sung là tự chọn sẽ được trình bày trong phần sau của sách. Trong phần này chúng ta sẽ tập trung vào phần còn lại của khai báo. Một kiểu liệt kê bắt đầu với từ khóa **enum**, tiếp sau là một định danh cho kiểu liệt kê:

```
enum NhietDoNuoc
```

Thành phần kiểu cơ sở chính là kiểu khai báo cho các mục trong kiểu liệt kê. Nếu bỏ qua thành phần này thì trình biên dịch sẽ gán giá trị mặc định là kiểu nguyên int, tuy nhiên chúng ta có thể sử dụng bất cứ kiểu nguyên nào như ushort hay long,...ngoại trừ kiểu ký tự. Đoạn ví dụ sau khai báo một kiểu liệt kê sử dụng kiểu cơ sở là số nguyên không dấu uint:

```
enum KichThuoc :uint
{
    Nho = 1,
    Vua = 2,
    Lon = 3,
}
```

Lưu ý là khai báo một kiểu liệt kê phải kết thúc bằng một danh sách liệt kê, danh sách liệt kê này phải có các hằng được gán, và mỗi thành phần phải phân cách nhau dấu phẩy.

Ta viết lại ví dụ minh họa 3-4 như sau.

 *Ví dụ 3.5: Sử dụng kiểu liệt kê để đơn giản chương trình.*

```
class MinhHoaC3
{
    // Khai báo kiểu liệt kê
    enum NhietDoNuoc
    {
        DoDong = 0,
        DoNguoi = 20,
        DoAm = 40,
        DoNong = 60,
        DoSoi = 100,
    }

    static void Main()
    {
        System.Console.WriteLine( "Nhiet do dong: {0}", NhietDoNuoc.DoDong);
        System.Console.WriteLine( "Nhiet do nguoi: {0}", NhietDoNuoc.DoNguoi);
        System.Console.WriteLine( "Nhiet do am: {0}", NhietDoNuoc.DoAm);
        System.Console.WriteLine( "Nhiet do nong: {0}", NhietDoNuoc.DoNong);
        System.Console.WriteLine( "Nhiet do soi: {0}", NhietDoNuoc.DoSoi);
    }
}
```

Kết quả:


```
Nhiệt độ đông: 0
Nhiệt độ người: 20
Nhiệt độ ấm: 40
Nhiệt độ nóng: 60
Nhiệt độ sôi: 100
```

Mỗi thành phần trong kiểu liệt kê tương ứng với một giá trị số, trong trường hợp này là một số nguyên. Nếu chúng ta không khởi tạo cho các thành phần này thì chúng sẽ nhận các giá trị tiếp theo với thành phần đầu tiên là 0.

Ta xem thử khai báo sau:

```
enum Thutu
{
    ThuNhat,
    ThuHai,
    ThuBa = 10,
    ThuTu
}
```

Khi đó giá trị của ThuNhat là 0, giá trị của ThuHai là 1, giá trị của ThuBa là 10 và giá trị của ThuTu là 11.

Kiểu liệt kê là một kiểu hình thức do đó bắt buộc phải thực hiện phép chuyển đổi tương minh với các kiểu giá trị nguyên:

```
int x = (int) ThuTu.ThuNhat;
```

Kiểu chuỗi ký tự

Kiểu dữ liệu chuỗi khá thân thiện với người lập trình trong bất cứ ngôn ngữ lập trình nào, kiểu dữ liệu chuỗi lưu giữ một mảng những ký tự.

Để khai báo một chuỗi chúng ta sử dụng từ khóa string tương tự như cách tạo một thể hiện của bất cứ đối tượng nào:

```
string chuoi;
```

Một hằng chuỗi được tạo bằng cách đặt các chuỗi trong dấu nháy đôi:

```
"Xin chào"
```

Đây là cách chung để khởi tạo một chuỗi ký tự với giá trị hằng:

```
string chuoi = "Xin chào"
```

Kiểu chuỗi sẽ được đề cập sâu trong chương 10.

Định danh

Định danh là tên mà người lập trình chỉ định cho các kiểu dữ liệu, các phương thức, biến, hằng, hay đối tượng.... Một định danh phải bắt đầu với một ký tự chữ cái hay dấu gạch dưới, các ký tự còn lại phải là ký tự chữ cái, chữ số, dấu gạch dưới.

Theo qui ước đặt tên của Microsoft thì đề nghị sử dụng *cú pháp lạc đà* (camel notation) bắt đầu bằng ký tự thường để đặt tên cho các biến là *cú pháp Pascal* (Pascal notation) với ký tự đầu tiên hoa cho cách đặt tên hàm và hầu hết các định danh còn lại. Hầu như Microsoft không còn dùng cú pháp Hungary như iSoNguyen hay dấu gạch dưới Bien_Nguyen để đặt các định danh.

Các định danh không được trùng với các từ khoá mà C# đưa ra, do đó chúng ta không thể tạo các biến có tên như class hay int được. Ngoài ra, C# cũng phân biệt các ký tự thường và ký tự hoa vì vậy C# xem hai biến bienNguyen và bienguyen là hoàn toàn khác nhau.

Biểu thức

Những câu lệnh mà thực hiện việc đánh giá một giá trị gọi là biểu thức. Một phép gán một giá trị cho một biến cũng là một biểu thức:

```
var1 = 24;
```

Trong câu lệnh trên phép đánh giá hay định lượng chính là phép gán có giá trị là 24 cho biến var1. Lưu ý là toán tử gán (=) không phải là toán tử so sánh. Do vậy khi sử dụng toán tử này thì biến bên trái sẽ nhận giá trị của phần bên phải. Các toán tử của ngôn ngữ C# như phép so sánh hay phép gán sẽ được trình bày chi tiết trong mục toán tử của chương này.

Do var1 = 24 là một biểu thức được định giá trị là 24 nên biểu thức này có thể được xem như phần bên phải của một biểu thức gán khác:

```
var2 = var1 = 24;
```

Lệnh này sẽ được thực hiện từ bên phải sang khi đó biến var1 sẽ nhận được giá trị là 24 và tiếp sau đó thì var2 cũng được nhận giá trị là 24. Do vậy cả hai biến đều cùng nhận một giá trị là 24. Có thể dùng lệnh trên để khởi tạo nhiều biến có cùng một giá trị như:

```
a = b = c = d = 24;
```

Khoảng trắng (whitespace)

Trong ngôn ngữ C#, những khoảng trắng, khoảng tab và các dòng được xem như là khoảng trắng (whitespace), giống như tên gọi vì chỉ xuất hiện những khoảng trắng để đại diện cho các ký tự đó. C# sẽ bỏ qua tất cả các khoảng trắng đó, do vậy chúng ta có thể viết như sau:

```
var1 = 24;
```

hay

```
var1  = 24 ;
```

và trình biên dịch C# sẽ xem hai câu lệnh trên là hoàn toàn giống nhau.

Tuy nhiên lưu ý là khoảng trắng trong một chuỗi sẽ không được bỏ qua. Nếu chúng ta viết:

```
System.WriteLine("Xin chào!");
```

mỗi khoảng trắng ở giữa hai chữ "Xin" và "chào" đều được đối xử bình thường như các ký tự khác trong chuỗi.

Hầu hết việc sử dụng khoảng trắng như một sự tùy ý của người lập trình. Điều cốt yếu là việc sử dụng khoảng trắng sẽ làm cho chương trình dễ nhìn dễ đọc hơn Cũng như khi ta viết một văn bản trong MS Word nếu không trình bày tốt thì sẽ khó đọc và gây mất cảm tình cho người xem. Còn đối với trình biên dịch thì việc dùng hay không dùng khoảng trắng là không khác nhau.

Tuy nhiên, cũng cần lưu ý khi sử dụng khoảng trắng như sau:

```
int x = 24;
```

tương tự như:

```
int x=24;
```

nhưng không giống như:

```
intx=24;
```

Trình biên dịch nhận biết được các khoảng trắng ở hai bên của phép gán là phụ và có thể bỏ qua, nhưng khoảng trắng giữa khai báo kiểu và tên biến thì không phải phụ hay thêm mà bắt buộc phải có tối thiểu một khoảng trắng. Điều này không có gì bất hợp lý, vì khoảng trắng cho phép trình biên dịch nhận biết được từ khoá `int` và không thể nào nhận được `intx`.

Tương tự như C/C++, trong C# câu lệnh được kết thúc với dấu chấm phẩy ';'. Do vậy có thể một câu lệnh trên nhiều dòng, và một dòng có thể nhiều câu lệnh nhưng nhất thiết là hai câu lệnh phải cách nhau một dấu chấm phẩy.

Câu lệnh (statement)

Trong C# một chỉ dẫn lập trình đầy đủ được gọi là câu lệnh. Chương trình bao gồm nhiều câu lệnh tuần tự với nhau. Mỗi câu lệnh phải kết thúc với một dấu chấm phẩy, ví dụ như:

```
int x; // một câu lệnh
```

```
x = 32; // câu lệnh khác
```

```
int y =x; // đây cũng là một câu lệnh
```

Những câu lệnh này sẽ được xử lý theo thứ tự. Đầu tiên trình biên dịch bắt đầu ở vị trí đầu của danh sách các câu lệnh và lần lượt đi từng câu lệnh cho đến lệnh cuối cùng, tuy nhiên chỉ đúng cho trường hợp các câu lệnh tuần tự không phân nhánh.

Có hai loại câu lệnh phân nhánh trong C# là : phân nhánh không có điều kiện (unconditional branching statement) và phân nhánh có điều kiện (conditional branching statement).

Ngoài ra còn có các câu lệnh làm cho một số đoạn chương trình được thực hiện nhiều lần, các câu lệnh này được gọi là câu lệnh lặp hay vòng lặp. Bao gồm các lệnh lặp **for**, **while**, **do**, **in**, và **each** sẽ được đề cập tới trong mục tiếp theo.


Sau đây chúng ta sẽ xem xét hai loại lệnh phân nhánh phổ biến nhất trong lập trình C#.

Phân nhánh không có điều kiện

Phân nhánh không có điều kiện có thể tạo ra bằng hai cách: gọi một hàm và dùng từ khoá phân nhánh không điều kiện.

Gọi hàm

Khi trình biên dịch xử lý đến tên của một hàm, thì sẽ ngưng thực hiện hàm hiện thời mà bắt đầu phân nhánh để tạo một gọi hàm mới. Sau khi hàm vừa tạo thực hiện xong và trả về một giá trị thì trình biên dịch sẽ tiếp tục thực hiện dòng lệnh tiếp sau của hàm ban đầu. ví dụ 3.6 minh họa cho việc phân nhánh khi gọi hàm.

 Ví dụ 3.6: Gọi một hàm.

```
using System;
class GoiHam
{
    static void Main()
    {
        Console.WriteLine( "Ham Main chuan bi gọi ham Func()..." );
        Func();
        Console.WriteLine( "Tro lai ham Main()" );
    }
    static void Func()
    {
        Console.WriteLine( "---->Toi la ham Func()..." );
    }
}
```

Kết quả:

```
Ham Main chuan bi gọi ham Func()...
---->Toi la ham Func()...
Tro lai ham Main()
```

Luồng chương trình thực hiện bắt đầu từ hàm Main xử lý đến dòng lệnh Func(), lệnh Func() thường được gọi là một lời gọi hàm. Tại điểm này luồng chương trình sẽ rẽ nhánh để thực hiện hàm vừa gọi. Sau khi thực hiện xong hàm Func, thì chương trình quay lại hàm Main và thực hiện câu lệnh ngay sau câu lệnh gọi hàm Func.

Từ khoá phân nhánh không điều kiện

Để thực hiện phân nhánh ta gọi một trong các từ khóa sau: **goto**, **break**, **continue**, **return**, **statementthrow**. Việc trình bày các từ khóa phân nhánh không điều kiện này sẽ được đề cập trong chương tiếp theo. Trong phần này chỉ đề cập chung không đi vào chi tiết.

Phân nhánh có điều kiện

Phân nhánh có điều kiện được tạo bởi các lệnh điều kiện. Các từ khóa của các lệnh này như : **if**, **else**, **switch**. Sự phân nhánh chỉ được thực hiện khi biểu thức điều kiện phân nhánh được xác định là đúng.

Câu lệnh *if...else*

Câu lệnh phân nhánh **if...else** dựa trên một điều kiện. Điều kiện là một biểu thức sẽ được kiểm tra giá trị ngay khi bắt đầu gặp câu lệnh đó. Nếu điều kiện được kiểm tra là đúng, thì câu lệnh hay một khối các câu lệnh bên trong thân của câu lệnh **if** được thực hiện.


Trong câu điều kiện **if...else** thì **else** là phần tùy chọn. Các câu lệnh bên trong thân của **else** chỉ được thực hiện khi điều kiện của **if** là sai. Do vậy khi câu lệnh đầy đủ **if...else** được dùng thì chỉ có một trong hai **if** hoặc **else** được thực hiện. Ta có cú pháp câu điều kiện **if... else** sau:

```
if (biểu thức điều kiện)
    <Khối lệnh thực hiện khi điều kiện đúng>
[else
    <Khối lệnh thực hiện khi điều kiện sai>]
```

Nếu các câu lệnh trong thân của **if** hay **else** mà lớn hơn một lệnh thì các lệnh này phải được bao trong một khối lệnh, tức là phải nằm trong dấu khối { }:

```
if (biểu thức điều kiện)
{
    <lệnh 1>
    <lệnh 2>
    ....
}
[else
{
    <lệnh 1>
    <lệnh 2>
    ...
}]
```

Như trình bày bên trên do **else** là phần tùy chọn nên được đặt trong dấu ngoặc vuông [...]. Minh họa 3.7 bên dưới cách sử dụng câu lệnh **if...else**.

 Ví dụ 3.7: Dùng câu lệnh điều kiện *if...else*.

```
using System;
class ExIfElse
{
    static void Main()
    {
        int var1 = 10;
        int var2 = 20;
        if ( var1 > var2)
```

```

{
    Console.WriteLine( "var1: {0} > var2:{1}", var1, var2);
}
else
{
    Console.WriteLine( "var2: {0} > var1:{1}", var2, var1);
}
var1 = 30;
if ( var1 > var2)
{
    var2 = var1++;
    Console.WriteLine( "Gan gia tri var1 cho var2");
    Console.WriteLine( "Tang bien var1 len mot ");
    Console.WriteLine( "Var1 = {0}, var2 = {1}", var1, var2);
}
else
{
    var1 = var2;
    Console.WriteLine( "Thiet lap gia tri var1 = var2" );
    Console.WriteLine( "var1 = {0}, var2 = {1}", var1, var2 );
}
}
}

```

Kết quả:

```

Gan gia tri var1 cho var2
Tang bien var1 len mot
Var1 = 31, var2 = 30

```

Trong ví dụ 3.7 trên, câu lệnh **if** đầu tiên sẽ kiểm tra xem giá trị của **var1** có lớn hơn giá trị của **var2** không. Biểu thức điều kiện này sử dụng toán tử quan hệ lớn hơn (>), các toán tử khác như nhỏ hơn (<), hay bằng (==). Các toán tử này thường xuyên được sử dụng trong lập trình và kết quả trả là giá trị đúng hay sai.

Việc kiểm tra xác định giá trị **var1** lớn hơn **var2** là sai (vì **var1** = 10 trong khi **var2** = 20), khi đó các lệnh trong **else** sẽ được thực hiện, và các lệnh này in ra màn hình:

```
var2: 20 > var1: 10
```

Tiếp theo đến câu lệnh **if** thứ hai, sau khi thực hiện lệnh gán giá trị của `var1 = 30`, lúc này điều kiện **if** đúng nên các câu lệnh trong khối **if** sẽ được thực hiện và kết quả là in ra ba dòng sau:

Gán giá trị `var1` cho `var2`

Tăng biến `var1` lên một


`Var1 = 31, var2 = 30`

*Câu lệnh **if** lồng nhau*

Các lệnh điều kiện **if** có thể lồng nhau để phục vụ cho việc xử lý các câu điều kiện phức tạp. Việc này cũng thường xuyên gặp khi lập trình. Giả sử chúng ta cần viết một chương trình có yêu cầu xác định tình trạng kết hôn của một công dân dựa vào các thông tin như tuổi, giới tính, và tình trạng hôn nhân, dựa trên một số thông tin như sau:

- Nếu công dân là nam thì độ tuổi có thể kết hôn là 20 với điều kiện là chưa có gia đình.
- Nếu công dân là nữ thì độ tuổi có thể kết hôn là 19 cũng với điều kiện là chưa có gia đình.
- Tất cả các công dân có tuổi nhỏ hơn 19 điều không được kết hôn.

Dựa trên các yêu cầu trên ta có thể dùng các lệnh **if** lồng nhau để thực hiện. Ví dụ 3.8 sau sẽ minh họa cho việc thực hiện các yêu cầu trên.

 *Ví dụ 3.8: Các lệnh **if** lồng nhau.*

```
using System;
class TinhTrangKetHon
{
    static void Main()
    {
        int tuoi;
        bool coGiaDinh; // 0: chưa có gia đình; 1: đã có gia đình
        bool gioiTinh; // 0: giới tính nữ; 1: giới tính nam
        tuoi = 24;
        coGiaDinh = false; // chưa có gia đình
        gioiTinh = true; // nam

        if ( tuoi >= 19)
        {
            if ( coGiaDinh == false)
            {
                if ( gioiTinh == false) // nu
                    Console.WriteLine(" Nu co the ket hon");
                else // nam
            }
        }
    }
}
```

```

        if (tuoi > 19) // phải lớn hơn 19 tuổi mới được kết hôn
            Console.WriteLine(" Nam co the ket hon");
    }
    else // da co gia dinh
        Console.WriteLine(" Khong the ket hon nua do da ket hon");
    }
    else // tuoi < 19
        Console.WriteLine(" Khong du tuoi ket hon" );
    }
}

```

Kết quả:

Nam co the ket hon

Theo trình tự kiểm tra thì câu lệnh **if** đầu tiên được thực hiện, biểu thức điều kiện đúng do tuổi có giá trị là 24 lớn hơn 19. Khi đó khối lệnh trong **if** sẽ được thực thi. Ở trong khối này lại xuất hiện một lệnh **if** khác để kiểm tra tình trạng xem người đó đã có gia đình chưa, kết quả điều kiện **if** là đúng vì `coGiaDinh = false` nên biểu thức so sánh `coGiaDinh == false` sẽ trả về giá trị đúng. Tiếp tục xét xem giới tính của người đó là nam hay nữ, vì chỉ có nam trên 19 tuổi mới được kết hôn. Kết quả kiểm tra là nam nên câu lệnh **if** thứ ba được thực hiện và xuất ra kết quả : “Nam co the ket hon”.

Câu lệnh switch

Khi có quá nhiều điều kiện để chọn thực hiện thì dùng câu lệnh **if** sẽ rất rối rắm và dài dòng, Các ngôn ngữ lập trình cấp cao đều cung cấp một dạng câu lệnh **switch** liệt kê các giá trị và chỉ thực hiện các giá trị thích hợp. C# cũng cung cấp câu lệnh nhảy **switch** có cú pháp sau:

```

switch (biểu thức điều kiện)
{
    case <giá trị>:
        <Các câu lệnh thực hiện>
        <lệnh nhảy>
    [default:
        <Các câu lệnh thực hiện mặc định>]
}

```

Cũng tương tự như câu lệnh **if**, biểu thức để so sánh được đặt sau từ khóa **switch**, tuy nhiên giá trị so sánh lại được đặt sau mỗi các từ khóa **case**. Giá trị sau từ khóa **case** là các giá trị hằng số nguyên như đã đề cập trong phần trước.

Nếu một câu lệnh **case** được thích hợp tức là giá trị sau **case** bằng với giá trị của biểu thức sau **switch** thì các câu lệnh liên quan đến câu lệnh **case** này sẽ được thực thi. Tuy nhiên phải có một câu lệnh nhảy như **break**, **goto** để điều khiển nhảy qua các **case** khác. Vì nếu không có các lệnh nhảy này thì khi đó chương trình sẽ thực hiện tất cả các **case** theo sau. Để dễ hiểu hơn ta sẽ xem xét ví dụ 3.9 dưới đây.

 Ví dụ 3.9: Câu lệnh switch.

```
using System;
class MinhHoaSwitch
{
    static void Main()
    {
        const int mauDo = 0;
        const int mauCam = 1;
        const int mauVang = 2;
        const int mauLuc = 3;
        const int mauLam = 4;
        const int mauCham = 5;
        const int mauTim = 6;
        int chonMau = mauLuc;

        switch ( chonMau )
        {
            case mauDo:
                Console.WriteLine( "Ban cho mau do" );
                break;
            case mauCam:
                Console.WriteLine( "Ban cho mau cam" );
                break;
            case mauVang:
                //Console.WriteLine( "Ban chon mau vang");
            case mauLuc:
                Console.WriteLine( "Ban chon mau luc");
                break;
            case mauLam:
                Console.WriteLine( "Ban chon mau lam");
                goto case mauCham;
            case mauCham:
```

```

        Console.WriteLine( "Ban cho mau cham");
        goto case mauTim;
    case mauTim:
        Console.WriteLine( "Ban chon mau tim");
        goto case mauLuc;
    default:
        Console.WriteLine( "Ban khong chon mau nao het");
        break;
    }
    Console.WriteLine( "Xin cam on!");
}
}

```

Trong ví dụ 3.9 trên liệt kê bảy loại màu và dùng câu lệnh **switch** để kiểm tra các trường hợp chọn màu. Ở đây chúng ta thử phân tích từng câu lệnh **case** mà không quan tâm đến giá trị biến chonMau.

Giá trị chonMau	Câu lệnh case thực hiện	Kết quả thực hiện
mauDo	case mauDo	Ban chon mau do
mauCam	case mauCam	Ban chon mau cam
mauVang	case mauVang case mauLuc	Ban chon mau luc
mauLuc	case mauLuc	Ban chon mau luc
mauLam	case mauLam case mauCham case mauTim case mauLuc	Ban chon mau lam Ban chon mau cham Ban chon mau tim Ban chon mau luc
mauCham	case mauCham	Ban chon mau cham
	case mauTim case mauLuc	Ban chon mau tim Ban chon mau luc
mauTim	case mauTim	Ban chon mau tim
	case mauLuc	Ban chon mau luc

Bảng 3.3: Mô tả các trường hợp thực hiện câu lệnh switch.

Trong đoạn ví dụ do giá trị của biến chonMau = mauLuc nên khi vào lệnh **switch** thì **case** mauLuc sẽ được thực hiện và kết quả như sau:

Kết quả ví dụ 3.9

Bạn chọn màu lúc

Xin cảm ơn!

Ghi chú: Đối với người lập trình C/C++, trong C# chúng ta không thể nhảy xuống một trường hợp **case** tiếp theo nếu câu lệnh **case** hiện tại không xong. Vì vậy chúng ta phải viết như sau:

```
case 1:    // nhảy xuống
case 2:
```

Như minh họa trên thì trường hợp xử lý **case 1** là xong, tuy nhiên chúng ta không thể viết như sau:

```
case 1:
    DoAnything();
    // Trường hợp này không thể nhảy xuống case 2
case 2:
```

trong đoạn chương trình thứ hai trường hợp **case 1** có một câu lệnh nên không thể nhảy xuống được. Nếu muốn trường hợp case1 nhảy qua **case 2** thì ta phải sử dụng câu lệnh **goto** một cách tường minh:

```
case 1:
    DoAnything();
    goto case 2;
case 2:
```

Do vậy khi thực hiện xong các câu lệnh của một trường hợp nếu muốn thực hiện một trường hợp **case** khác thì ta dùng câu lệnh nhảy **goto** với nhãn của trường hợp đó:

```
goto case <giá trị>
```

Khi gặp lệnh thoát **break** thì chương trình thoát khỏi **switch** và thực hiện lệnh tiếp sau khối **switch** đó.

Nếu không có trường hợp nào thích hợp và trong câu lệnh **switch** có dùng câu lệnh **default** thì các câu lệnh của trường hợp **default** sẽ được thực hiện. Ta có thể dùng **default** để cảnh báo một lỗi hay xử lý một trường hợp ngoài tất cả các trường hợp **case** trong **switch**.

Trong ví dụ minh họa câu lệnh **switch** trước thì giá trị để kiểm tra các trường hợp thích hợp là các hằng số nguyên. Tuy nhiên C# còn có khả năng cho phép chúng ta dùng câu lệnh **switch** với giá trị là một chuỗi, có thể viết như sau:

```
switch (chuoi1)
{
    case "mau do":
        ....
        break;
    case "mau cam":
```

```

        ...
        break;

        ...
    }

```

Câu lệnh lặp

C# cung cấp một bộ mở rộng các câu lệnh lặp, bao gồm các câu lệnh lặp **for**, **while** và **do... while**. Ngoài ra ngôn ngữ C# còn bổ sung thêm một câu lệnh lặp **foreach**, lệnh này mới đối với người lập trình C/C++ nhưng khá thân thiện với người lập trình VB. Cuối cùng là các câu lệnh nhảy như **goto**, **break**, **continue**, và **return**.

Câu lệnh nhảy goto

Lệnh nhảy **goto** là một lệnh nhảy đơn giản, cho phép chương trình nhảy vô điều kiện tới một vị trí trong chương trình thông qua tên nhãn. Tuy nhiên việc sử dụng lệnh **goto** thường làm mất đi tính cấu trúc thuật toán, việc lạm dụng sẽ dẫn đến một chương trình nguồn mà giới lập trình gọi là “*mì ăn liền*” rồi như mớ bòng bong vậy. Hầu hết các người lập trình có kinh nghiệm đều tránh dùng lệnh **goto**. Sau đây là cách sử dụng lệnh nhảy **goto**:

- ◆ Tạo một nhãn
- ◆ **goto** đến nhãn

Nhãn là một định danh theo sau bởi dấu hai chấm (:). Thường thường một lệnh **goto** gắn với một điều kiện nào đó, ví dụ 3.10 sau sẽ minh họa các sử dụng lệnh nhảy **goto** trong chương trình.



Ví dụ 3.10: Sử dụng goto.

```

using System;

public class UsingGoto
{
    public static int Main()
    {
        int i = 0;
        lap:    // nhãn
        Console.WriteLine("i:{0}",i);
        i++;
        if ( i < 10 )
            goto lap; // nhảy về nhãn lap
        return 0;
    }
}

```

Kết quả:

```
i:0
i:1
i:2
i:3
i:4
i:5
i:6
i:7
i:8
i:9
```

Nếu chúng ta vẽ lưu đồ của một chương trình có sử dụng nhiều lệnh **goto**, thì ta sẽ thấy kết quả rất nhiều đường chằng chéo lên nhau, giống như là các sợi mì vậy. Chính vì vậy nên những đoạn mã chương trình có dùng lệnh **goto** còn được gọi là “*spaghetti code*”.

Việc tránh dùng lệnh nhảy **goto** trong chương trình hoàn toàn thực hiện được, có thể dùng vòng lặp **while** để thay thế hoàn toàn các câu lệnh **goto**.

Vòng lặp while

Ý nghĩa của vòng lặp **while** là: “*Trong khi điều kiện đúng thì thực hiện các công việc này*”.

Cú pháp sử dụng vòng lặp **while** như sau:

while (Biểu thức)

<Câu lệnh thực hiện>

Biểu thức của vòng lặp **while** là điều kiện để các lệnh được thực hiện, biểu thức này bắt buộc phải trả về một giá trị kiểu bool là true/false. Nếu có nhiều câu lệnh cần được thực hiện trong vòng lặp **while** thì phải đặt các lệnh này trong khối lệnh. Ví dụ 3.11 minh họa việc sử dụng vòng lặp **while**.



Ví dụ 3.11: Sử dụng vòng lặp while.

```
using System;
public class UsingWhile
{
    public static int Main()
    {
        int i = 0;
        while ( i < 10 )
        {
            Console.WriteLine(" i: {0} ",i);
            i++;
        }
    }
}
```

```

    }
    return 0;
}
}

```

Kết quả:

```

i:0
i:1
i:2
i:3
i:4
i:5
i:6
i:7
i:8
i:9

```

Đoạn chương trình 3.11 cũng cho kết quả tương tự như chương trình minh họa 3.10 dùng lệnh **goto**. Tuy nhiên chương trình 3.11 rõ ràng hơn và có ý nghĩa tự nhiên hơn. Có thể diễn giải ngôn ngữ tự nhiên đoạn vòng lặp **while** như sau: “*Trong khi i nhỏ hơn 10, thì in ra giá trị của i và tăng i lên một đơn vị*”.

Lưu ý rằng vòng lặp **while** sẽ kiểm tra điều kiện trước khi thực hiện các lệnh bên trong, điều này đảm bảo nếu ngay từ đầu điều kiện sai thì vòng lặp sẽ không bao giờ thực hiện. do vậy nếu khởi tạo biến i có giá trị là 11, thì vòng lặp sẽ không được thực hiện.

Vòng lặp do...while

Đôi khi vòng lặp **while** không thỏa mãn yêu cầu trong tình huống sau, chúng ta muốn chuyển ngữ nghĩa của **while** là “*chạy trong khi điều kiện đúng*” thành ngữ nghĩa khác như “*làm điều này trong khi điều kiện vẫn còn đúng*”. Nói cách khác thực hiện một hành động, và sau khi hành động được hoàn thành thì kiểm tra điều kiện. Cú pháp sử dụng vòng lặp **do...while** như sau:

```

do
    <Câu lệnh thực hiện>
while ( điều kiện )

```

Ở đây có sự khác biệt quan trọng giữa vòng lặp **while** và vòng lặp **do...while** là khi dùng vòng lặp **do...while** thì tối thiểu sẽ có một lần các câu lệnh trong **do...while** được thực hiện. Điều này cũng dễ hiểu vì lần đầu tiên đi vào vòng lặp **do...while** thì điều kiện chưa được kiểm tra.



Ví dụ 3.12: Minh họa việc sử dụng vòng lặp do..while.

```
using System;
public class UsingDoWhile
{
    public static int Main( )
    {
        int i = 11;
        do
        {
            Console.WriteLine("i: {0}",i);
            i++;
        } while ( i < 10 )
        return 0;
    }
}
```

Kết quả:

i: 11

Do khởi tạo biến *i* giá trị là 11, nên điều kiện của **while** là sai, tuy nhiên vòng lặp **do...while** vẫn được thực hiện một lần.

Vòng lặp for


Vòng lặp **for** bao gồm ba phần chính:

- Khởi tạo biến đếm vòng lặp
- Kiểm tra điều kiện biến đếm, nếu đúng thì sẽ thực hiện các lệnh bên trong vòng **for**
- Thay đổi bước lặp.

Cú pháp sử dụng vòng lặp **for** như sau:

for ([phần khởi tạo] ; [biểu thức điều kiện]; [bước lặp])
<Câu lệnh thực hiện>

Vòng lặp **for** được minh họa trong ví dụ sau:

 *Ví dụ 3.13: Sử dụng vòng lặp for.*

```
using System;
public class UsingFor
{
    public static int Main()
    {
        for (int i = 0; i < 30; i++)
```

```

{
    if (i %10 ==0)
    {
        Console.WriteLine("{0} ",i);
    }
    else
    {
        Console.Write("{0} ",i);
    }
}
return 0;
}
}

```

Kết quả:

```

0
1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29

```

Trong đoạn chương trình trên có sử dụng toán tử chia lấy dư modulo, toán tử này sẽ được đề cập đến phần sau. Ý nghĩa lệnh `i%10 == 0` là kiểm tra xem `i` có phải là bội số của 10 không, nếu `i` là bội số của 10 thì sử dụng lệnh `WriteLine` để xuất giá trị `i` và sau đó đưa cursor về đầu dòng sau. Còn ngược lại chỉ cần xuất giá trị của `i` và không xuống dòng.

Đầu tiên biến `i` được khởi tạo giá trị ban đầu là 0, sau đó chương trình sẽ kiểm tra điều kiện, do 0 nhỏ hơn 30 nên điều kiện đúng, khi đó các câu lệnh bên trong vòng lặp **for** sẽ được thực hiện. Sau khi thực hiện xong thì biến `i` sẽ được tăng thêm một đơn vị (`i++`).

Có một điều lưu ý là biến `i` do khai báo bên trong vòng lặp **for** nên chỉ có phạm vi hoạt động bên trong vòng lặp. Ví dụ 3.14 sau sẽ không được biên dịch vì xuất hiện một lỗi.



Ví dụ 3.14: Phạm vi của biến khai báo trong vòng lặp.

```

using System;
public class UsingFor
{
    public static int Main()
    {
        for (int i = 0; i < 30; i++)
        {

```



```

        if (i % 10 == 0)
        {
            Console.WriteLine("{0} ", i);
        }
        else
        {
            Console.Write("{0} ", i);
        }
    }
    // Lệnh sau sai do biến i chỉ được khai báo bên trong vòng lặp
    Console.WriteLine(" Ket qua cuoi cung cua i:{0}", i);
    return 0;
}
}

```

Câu lệnh lặp foreach

Vòng lặp **foreach** cho phép tạo vòng lặp thông qua một tập hợp hay một mảng. Đây là một câu lệnh lặp mới không có trong ngôn ngữ C/C++. Câu lệnh **foreach** có cú pháp chung như sau:

```

foreach ( <kiểu tập hợp> <tên truy cập thành phần > in < tên tập hợp>)
    <Các câu lệnh thực hiện>

```

Do lặp dựa trên một mảng hay tập hợp nên toàn bộ vòng lặp sẽ duyệt qua tất cả các thành phần của tập hợp theo thứ tự được sắp. Khi duyệt đến phần tử cuối cùng trong tập hợp thì chương trình sẽ thoát ra khỏi vòng lặp **foreach**.

 Ví dụ 3.15 minh họa việc sử dụng vòng lặp *foreach*.

```

using System;
public class UsingForeach
{
    public static int Main()
    {
        int[] intArray = {1,2,3,4,5,6,7,8,9,10};
        foreach( int item in intArray)
        {
            Console.Write("{0} ", item);
        }
        return 0;
    }
}

```

```
}
-----
```

Kết quả:

```
1 2 3 4 5 6 7 8 9 10
-----
```

*Câu lệnh nhảy **break** và **continue***

Khi đang thực hiện các lệnh trong vòng lặp, có yêu cầu như sau: không thực hiện các lệnh còn lại nữa mà thoát khỏi vòng lặp, hay không thực hiện các công việc còn lại của vòng lặp hiện tại mà nhảy qua vòng lặp tiếp theo. Để đáp ứng yêu cầu trên C# cung cấp hai lệnh nhảy là **break** và **continue** để thoát khỏi vòng lặp.


Break khi được sử dụng sẽ đưa chương trình thoát khỏi vòng lặp và tiếp tục thực hiện các lệnh tiếp ngay sau vòng lặp.

Continue ngừng thực hiện các công việc còn lại của vòng lặp hiện thời và quay về đầu vòng lặp để thực hiện bước lặp tiếp theo

Hai lệnh **break** và **continue** tạo ra nhiều điểm thoát và làm cho chương trình khó hiểu cũng như là khó duy trì. Do vậy phải cẩn trọng khi sử dụng các lệnh nhảy này.

Ví dụ 3.16 sẽ được trình bày bên dưới minh họa cách sử dụng lệnh **continue** và **break**. Đoạn chương trình mô phỏng hệ thống xử lý tín hiệu giao thông đơn giản. Tín hiệu mô phỏng là các ký tự chữ hoa hay số được nhập vào từ bàn phím, sử dụng hàm ReadLine của lớp Console để đọc một chuỗi ký tự từ bàn phím.

Thuật toán của chương trình khá đơn giản: Khi nhận tín hiệu '0' có nghĩa là mọi việc bình thường, không cần phải làm bất cứ công việc gì cả, kể cả việc ghi lại các sự kiện. Trong chương trình này đơn giản nên các tín hiệu được nhập từ bàn phím, còn trong ứng dụng thật thì tín hiệu này sẽ được phát sinh theo các mẫu tin thời gian trong cơ sở dữ liệu. Khi nhận được tín hiệu thoát (mô phỏng bởi ký tự 'T') thì ghi lại tình trạng và kết thúc xử lý. Cuối cùng, bất cứ tín hiệu nào khác sẽ phát ra một thông báo, có thể là thông báo đến nhân viên cảnh sát chẳng hạn... Trường hợp tín hiệu là 'X' thì cũng sẽ phát ra một thông báo nhưng sau vòng lặp xử lý cũng kết thúc.

 *Ví dụ 3.16: Sử dụng **break** và **continue**.*

```
-----
using System;
public class TrafficSignal
{
    public static int Main()
    {
        string signal = "0"; // Khởi tạo tín hiệu
        // bắt đầu chu trình xử lý tín hiệu
        while ( signal != "X")
```

```

{
    //nhập tín hiệu
    Console.Write("Nhập vào một tín hiệu: ");
    signal = Console.ReadLine();
    // xuất tín hiệu hiện thời
    Console.WriteLine("Tín hiệu nhận được: {0}", signal);
    // phần xử lý tín hiệu
    if (signal == "T")
    {
        // Tín hiệu thoát được gửi
        // lưu lại sự kiện và thoát
        Console.WriteLine("Ngưng xử lý! Thoát\n");
        break;
    }
    if ( signal == "0")
    {
        // Tín hiệu nhận được bình thường
        // Lưu lại sự kiện và tiếp tục
        Console.WriteLine("Tất cả đều tốt!\n");
        continue;
    }
    // Thực hiện một số hành động nào đó
    // và tiếp tục
    Console.WriteLine("---bip bip bip\n");
}
return 0;
}
}

```

Kết quả: sau khi nhập tuần tự các tín hiệu : "0", "B", "T"

Nhập vào một tín hiệu: 0

Tín hiệu nhận được: 0

Tất cả đều tốt!

Nhập vào một tín hiệu: B

Tín hiệu nhận được: B

---bip bip bip

```

Nhap vao mot tin hieu: T
Tin hieu nhan duoc: T
Ngung xu ly! Thoat
    
```

Điểm chính yếu của đoạn chương trình trên là khi nhập vào tín hiệu “T” thì sau khi thực hiện một số hành động cần thiết chương trình sẽ thoát ra khỏi vòng lặp và không xuất ra câu thông báo bip bip bip. Ngược lại khi nhận được tín hiệu 0 thì sau khi xuất thông báo chương trình sẽ quay về đầu vòng lặp để thực hiện tiếp tục và cũng không xuất ra câu thông báo bip bip bip.

Toán tử

Toán tử được kí hiệu bằng một biểu tượng dùng để thực hiện một hành động. Các kiểu dữ liệu cơ bản của C# như kiểu nguyên hỗ trợ rất nhiều các toán tử như toán tử gán, toán tử toán học, logic....

Toán tử gán

Đến lúc này toán tử gán khá quen thuộc với chúng ta, hầu hết các chương trình minh họa từ đầu sách đều đã sử dụng phép gán. Toán tử gán hay phép gán làm cho toán hạng bên trái thay đổi giá trị bằng với giá trị của toán hạng bên phải. Toán tử gán là toán tử hai ngôi. Đây là toán tử đơn giản nhất thông dụng nhất và cũng dễ sử dụng nhất.

Toán tử toán học

Ngôn ngữ C# cung cấp năm toán tử toán học, bao gồm bốn toán tử đầu các phép toán cơ bản. Toán tử cuối cùng là toán tử chia nguyên lấy phần dư. Chúng ta sẽ tìm hiểu chi tiết các phép toán này trong phần tiếp sau.

*Các phép toán số học cơ bản (+, -, *, /)*

Các phép toán này không thể thiếu trong bất cứ ngôn ngữ lập trình nào, C# cũng không ngoại lệ, các phép toán số học đơn giản nhưng rất cần thiết bao gồm: phép cộng (+), phép trừ (-), phép nhân (*), phép chia (/) nguyên và không nguyên.

Khi chia hai số nguyên, thì C# sẽ bỏ phần phân số, hay bỏ phần dư, tức là nếu ta chia 8/3 thì sẽ được kết quả là 2 và sẽ bỏ phần dư là 2, do vậy để lấy được phần dư này thì C# cung cấp thêm toán tử lấy dư sẽ được trình bày trong phần kế tiếp.


Tuy nhiên, khi chia cho số thực có kiểu như float, double, hay decimal thì kết quả chia được trả về là một số thực.

Phép toán chia lấy dư

Để tìm phần dư của phép chia nguyên, chúng ta sử dụng toán tử chia lấy dư (%). Ví dụ, câu lệnh sau 8%3 thì kết quả trả về là 2 (đây là phần dư còn lại của phép chia nguyên).

Thật sự phép toán chia lấy dư rất hữu dụng cho người lập trình. Khi chúng ta thực hiện một phép chia dư n cho một số khác, nếu số này là bội số của n thì kết quả của phép chia dư là 0. Ví dụ 20 % 5 = 0 vì 20 là một bội số của 5. Điều này cho phép chúng ta ứng dụng trong

vòng lặp, khi muốn thực hiện một công việc nào đó cách khoảng n lần, ta chỉ cần kiểm tra phép chia dư n, nếu kết quả bằng 0 thì thực hiện công việc. Cách sử dụng này đã áp dụng trong ví dụ minh họa sử dụng vòng lặp **for** bên trên. Ví dụ 3.17 sau minh họa sử dụng các phép toán chia trên các số nguyên, thực...

 Ví dụ 3.17: Phép chia và phép chia lấy dư.

```
using System;
class Tester
{
    public static void Main()
    {
        int i1, i2;
        float f1, f2;
        double d1, d2;
        decimal dec1, dec2;

        i1 = 17;
        i2 = 4;
        f1 = 17f;
        f2 = 4f;
        d1 = 17;
        d2 = 4;
        dec1 = 17;
        dec2 = 4;
        Console.WriteLine("Integer: \t{0}", i1/i2);
        Console.WriteLine("Float: \t{0}", f1/f2);
        Console.WriteLine("Double: \t{0}", d1/d2);
        Console.WriteLine("Decimal: \t{0}", dec1/dec2);
        Console.WriteLine("\nModulus: : \t{0}", i1%i2);
    }
}
```

Kết quả:

```
Integer: 4
float:      4.25
double:     4.25
decimal: 4.25
```

Modulus: 1

Toán tử tăng và giảm

Khi sử dụng các biến số ta thường có thao tác là cộng một giá trị vào biến, trừ đi một giá trị từ biến đó, hay thực hiện các tính toán thay đổi giá trị của biến sau đó gán giá trị mới vừa tính toán cho chính biến đó.

Tính toán và gán trở lại

Giả sử chúng ta có một biến tên Luong lưu giá trị lương của một người, biến Luong này có giá trị hiện thời là 1.500.000, sau đó để tăng thêm 200.000 ta có thể viết như sau:

Luong = Luong + 200.000;

Trong câu lệnh trên phép cộng được thực hiện trước, khi đó kết quả của vế phải là 1.700.000 và kết quả này sẽ được gán lại cho biến Luong, cuối cùng Luong có giá trị là 1.700.000. Chúng ta có thể thực hiện việc thay đổi giá trị rồi gán lại cho biến với bất kỳ phép toán số học nào:

Luong = Luong * 2;

Luong = Luong - 100.000;

...

Do việc tăng hay giảm giá trị của một biến rất thường xảy ra trong khi tính toán nên C# cung cấp các phép toán tự gán (self- assignment). Bảng sau liệt kê các phép toán tự gán.

Toán tử	Ý nghĩa
+=	Cộng thêm giá trị toán hạng bên phải vào giá trị toán hạng bên trái
-=	Toán hạng bên trái được trừ bớt đi một lượng bằng giá trị của toán hạng bên phải
*=	Toán hạng bên trái được nhân với một lượng bằng giá trị của toán hạng bên phải.
/=	Toán hạng bên trái được chia với một lượng bằng giá trị của toán hạng bên phải.
%=	Toán hạng bên trái được chia lấy dư với một lượng bằng giá trị của toán hạng bên phải.

Bảng 3.4: Mô tả các phép toán tự gán.

Dựa trên các phép toán tự gán trong bảng ta có thể thay thế các lệnh tăng giảm lương như sau:

```
Luong += 200.000;
Luong *= 2;
Luong -= 100.000;
```

Kết quả của lệnh thứ nhất là giá trị của Luong sẽ tăng thêm 200.000, lệnh thứ hai sẽ làm cho giá trị Luong nhân đôi tức là tăng gấp 2 lần, và lệnh cuối cùng sẽ trừ bớt 100.000 của Luong. Do việc tăng hay giảm 1 rất phổ biến trong lập trình nên C# cung cấp hai toán tử đặc biệt là tăng một (++) hay giảm một (--).

Khi đó muốn tăng đi một giá trị của biến đếm trong vòng lặp ta có thể viết như sau:

```
bienDem++;
```

Toán tử tăng giảm tiền tố và tăng giảm hậu tố

Giả sử muốn kết hợp các phép toán như gia tăng giá trị của một biến và gán giá trị của biến cho biến thứ hai, ta viết như sau:

```
var1 = var2++;
```

Câu hỏi được đặt ra là gán giá trị trước khi cộng hay gán giá trị sau khi đã cộng. Hay nói cách khác giá trị ban đầu của biến var2 là 10, sau khi thực hiện ta muốn giá trị của var1 là 10, var2 là 11, hay var1 là 11, var2 cũng 11?

Để giải quyết yêu cầu trên C# cung cấp thứ tự thực hiện phép toán tăng/giảm với phép toán gán, thứ tự này được gọi là *tiền tố* (prefix) hay *hậu tố* (postfix). Do đó ta có thể viết:

```
var1 = var2++; // Hậu tố
```

Khi lệnh này được thực hiện thì phép gán sẽ được thực hiện trước tiên, sau đó mới đến phép toán tăng. Kết quả là var1 = 10 và var2 = 11. Còn đối với trường hợp tiền tố:

```
var1 = ++var2;
```

Khi đó phép tăng sẽ được thực hiện trước tức là giá trị của biến var2 sẽ là 11 và cuối cùng phép gán được thực hiện. Kết quả cả hai biến var1 và var2 đều có giá trị là 11.

Để hiểu rõ hơn về hai phép toán này chúng ta sẽ xem ví dụ minh họa 3.18 sau



Ví dụ 3.18: Minh họa sử dụng toán tử tăng trước và tăng sau khi gán.

```
using System;
class Tester
{
    static int Main()
    {
        int valueOne = 10;
        int valueTwo;
        valueTwo = valueOne++;
        Console.WriteLine("Thực hiện tăng sau: {0}, {1}",
            valueOne, valueTwo);
        valueOne = 20;
```

```

        valueTwo = ++valueOne;
        Console.WriteLine("Thuc hien tang truoc: {0}, {1}",
            valueOne, valueTwo);

        return 0;
    }
}

```

Kết quả:

Thuc hien tang sau: 11, 10

Thuc hien tang truoc: 21, 21

Toán tử quan hệ

Những toán tử quan hệ được dùng để so sánh giữa hai giá trị, và sau đó trả về kết quả là một giá trị logic kiểu bool (true hay false). Ví dụ toán tử so sánh lớn hơn (>) trả về giá trị là true nếu giá trị bên trái của toán tử lớn hơn giá trị bên phải của toán tử. Do vậy $5 > 2$ trả về một giá trị là true, trong khi $2 > 5$ trả về giá trị false.

Các toán tử quan hệ trong ngôn ngữ C# được trình bày ở bảng 3.4 bên dưới. Các toán tử trong bảng được minh họa với hai biến là value1 và value2, trong đó value1 có giá trị là 100 và value2 có giá trị là 50.

Tên toán tử	Kí hiệu	Biểu thức so sánh	Kết quả so sánh
So sánh bằng	==	value1 == 100 value1 == 50	true false
Không bằng	!=	value2 != 100 value2 != 90	false true
Lớn hơn	>	value1 > value2 value2 > value1	true false
Lớn hơn hay bằng	>=	value2 >= 50	true
Nhỏ hơn	<	value1 < value2 value2 < value1	false true
Nhỏ hơn hay bằng	<=	value1 <= value2	false

Bảng 3.4: Các toán tử so sánh (giả sử value1 = 100, và value2 = 50).

Như trong bảng 3.4 trên ta lưu ý toán tử so sánh bằng (==), toán tử này được ký hiệu bởi hai dấu bằng (=) liền nhau và cùng trên một hàng, không có bất kỳ khoảng trống nào xuất hiện giữa chúng. Trình biên dịch C# xem hai dấu này như một toán tử.

Toán tử logic

Trong câu lệnh **if** mà chúng ta đã tìm hiểu trong phần trước, thì khi điều kiện là true thì biểu thức bên trong **if** mới được thực hiện. Đôi khi chúng ta muốn kết hợp nhiều điều kiện với nhau như: bắt buộc cả hai hay nhiều điều kiện phải đúng hoặc chỉ cần một trong các điều kiện đúng là đủ hoặc không có điều kiện nào đúng...C# cung cấp một tập hợp các toán tử logic để phục vụ cho người lập trình.

Bảng 3.5 liệt kê ba phép toán logic, bảng này cũng sử dụng hai biến minh họa là x, và y trong đó x có giá trị là 5 và y có giá trị là 7.

Tên toán tử	Ký hiệu	Biểu thức logic	Giá trị	Logic
and	&&	(x == 3) && (y == 7)	false	Cả hai điều kiện phải đúng
or		(x == 3) (y == 7)	true	Chỉ cần một điều kiện đúng
not	!	!(x == 3)	true	Biểu thức trong ngoặc phải sai.

Bảng 3.5: Các toán tử logic (giả sử x = 5, y = 7).

Toán tử and sẽ kiểm tra cả hai điều kiện. Trong bảng 3.5 trên có minh họa biểu thức logic sử dụng toán tử and:

(x == 3) && (y == 7)

Toàn bộ biểu thức được xác định là sai vì có điều kiện (x == 3) là sai.

Với toán tử or, thì một hay cả hai điều kiện đúng thì đúng, biểu thức sẽ có giá trị là sai khi cả hai điều kiện sai. Do vậy ta xem biểu thức minh họa toán tử or:

(x == 3) || (y == 7)

Biểu thức này được xác định giá trị là đúng do có một điều kiện đúng là (y == 7) là đúng.

Đối với toán tử not, biểu thức sẽ có giá trị đúng khi điều kiện trong ngoặc là sai, và ngược lại, do đó biểu thức:

!(x == 3)

có giá trị là đúng vì điều kiện trong ngoặc tức là (x == 3) là sai.

Như chúng ta đã biết đối với phép toán logic and thì chỉ cần một điều kiện trong biểu thức sai là toàn bộ biểu thức là sai, do vậy thật là dư thừa khi kiểm tra các điều kiện còn lại một khi có một điều kiện đã sai. Giả sử ta có đoạn chương trình sau:

```
int x = 8;
```

```
if ((x == 5) && (y == 10))
```

Khi đó biểu thức **if** sẽ đúng khi cả hai biểu thức con là (x == 5) và (y == 10) đúng. Tuy nhiên khi xét biểu thức thứ nhất do giá trị x là 8 nên biểu thức (x == 5) là sai. Khi đó không cần thiết để xác định giá trị của biểu thức còn lại, tức là với bất kỳ giá trị nào của biểu thức (y == 10) thì toàn bộ biểu thức điều kiện **if** vẫn sai.

Tương tự với biểu thức logic or, khi xác định được một biểu thức con đúng thì không cần phải xác định các biểu thức con còn lại, vì toán tử logic or chỉ cần một điều kiện đúng là đủ:

```
int x =8;
```

```
if ( (x == 8) || (y == 10))
```

Khi kiểm tra biểu thức $(x == 8)$ có giá trị là đúng, thì không cần phải xác định giá trị của biểu thức $(y == 10)$ nữa.

Ngôn ngữ lập trình C# sử dụng logic như chúng ta đã thảo luận bên trên để loại bỏ các tính toán so sánh dư thừa và cũng không logic nữa!

Độ ưu tiên toán tử

Trình biên dịch phải xác định thứ tự thực hiện các toán tử trong trường hợp một biểu thức có nhiều phép toán, giả sử, có biểu thức sau:

```
var1 = 5+7*3;
```

Biểu thức trên có ba phép toán để thực hiện bao gồm $(=, +, *)$. Ta thử xét các phép toán theo thứ tự từ trái sang phải, đầu tiên là gán giá trị 5 cho biến `var1`, sau đó cộng 7 vào 5 là 12 cuối cùng là nhân với 3, kết quả trả về là 36, điều này thật sự có vấn đề, không đúng với mục đích yêu cầu của chúng ta. Do vậy việc xây dựng một trình tự xử lý các toán tử là hết sức cần thiết. Các luật về độ ưu tiên xử lý sẽ bảo trình biên dịch biết được toán tử nào được thực hiện trước trong biểu thức. Tương tự như trong phép toán đại số thì phép nhân có độ ưu tiên thực hiện trước phép toán cộng, do vậy $5+7*3$ cho kết quả là 26 đúng hơn kết quả 36. Và cả hai phép toán cộng và phép toán nhân đều có độ ưu tiên cao hơn phép gán. Như vậy trình biên dịch sẽ thực hiện các phép toán rồi sau đó thực hiện phép gán ở bước cuối cùng. Kết quả đúng của câu lệnh trên là biến `var1` sẽ nhận giá trị là 26.

Trong ngôn ngữ C#, dấu ngoặc được sử dụng để thay đổi thứ tự xử lý, điều này cũng giống trong tính toán đại số. Khi đó muốn kết quả 36 cho biến `var1` có thể viết:

```
var1 = (5+7) * 3;
```

Biểu thức trong ngoặc sẽ được xử lý trước và sau khi có kết quả là 12 thì phép nhân được thực hiện.

Bảng 3.6: Liệt kê thứ tự độ ưu tiên các phép toán trong C#.

STT	Loại toán tử	Toán tử	Thứ tự
1	Phép toán cơ bản	(x) $x.y$ $f(x)$ $a[x]$ $x++$ $x--$ new $typeof$ $sizeof$ $checked$ $unchecked$	Trái
2		$+$ $-$ $!$ \sim $++x$ $--x$ $(T)x$	Trái
3	Phép nhân	$*$ $/$ $\%$	Trái
4	Phép cộng	$+$ $-$	Trái
5	Dịch bit	$<<$ $>>$	Trái
6	Quan hệ	$<$ $>$ $<=$ $>=$ is	Trái

7	So sánh bằng	== !=	Phải
8	Phép toán logic AND	&	Trái
9	Phép toán logic XOR	^	Trái
10	Phép toán logic OR		Trái
11	Điều kiện AND	&&	Trái
12	Điều kiện OR		Trái
13	Điều kiện	?:	Phải
14	Phép gán	= *= /= %= += -= <=> >= &= ^= =	Phải

Bảng 3.6: Thứ tự ưu tiên các toán tử.


Các phép toán được liệt kê cùng loại sẽ có thứ tự theo mục thứ tự của bảng: thứ tự trái tức là độ ưu tiên của các phép toán từ bên trái sang, thứ tự phải thì các phép toán có độ ưu tiên từ bên phải qua trái. Các toán tử khác loại thì có độ ưu tiên từ trên xuống dưới, do vậy các toán tử loại cơ bản sẽ có độ ưu tiên cao nhất và phép toán gán sẽ có độ ưu tiên thấp nhất trong các toán tử.

Toán tử ba ngôi

Hầu hết các toán tử đòi hỏi có một toán hạng như toán tử (++ , --) hay hai toán hạng như (+, -, *, /, ...). Tuy nhiên, C# còn cung cấp thêm một toán tử có ba toán hạng (?). Toán tử này có cú pháp sử dụng như sau:

<Biểu thức điều kiện> ? <Biểu thức thứ 1> : <Biểu thức thứ 2>

Toán tử này sẽ xác định giá trị của một biểu thức điều kiện, và biểu thức điều kiện này phải trả về một giá trị kiểu bool. Khi điều kiện đúng thì <biểu thức thứ 1> sẽ được thực hiện, còn ngược lại điều kiện sai thì <biểu thức thứ 2> sẽ được thực hiện. Có thể diễn giải theo ngôn ngữ tự nhiên thì toán tử này có ý nghĩa : “*Nếu điều kiện đúng thì làm công việc thứ nhất, còn ngược lại điều kiện sai thì làm công việc thứ hai*”. Cách sử dụng toán tử ba ngôi này được minh họa trong ví dụ 3.19 sau.

 Ví dụ 3.19: Sử dụng toán tử ba ngôi.

```
using System;
class Tester
{
    public static int Main()
    {
        int value1;
```

```

        int value2;
        int maxValue;
        value1 = 10;
        value2 = 20;
        maxValue = value1 > value2 ? value1 : value2;
        Console.WriteLine("Gia tri thu nhat {0}, gia tri thu hai {1},
                           gia tri lon nhat {2}", value1, value2, maxValue);
        return 0;
    }
}

```

Kết quả:

Gia tri thu nhat 10, gia tri thu hai 20, gia tri lon nhat 20

Trong ví dụ minh họa trên toán tử ba ngôi được sử dụng để kiểm tra xem giá trị của value1 có lớn hơn giá trị của value2, nếu đúng thì trả về giá trị của value1, tức là gán giá trị value1 cho biến maxValue, còn ngược lại thì gán giá trị value2 cho biến maxValue.

Namespace

Chương 2 đã thảo luận việc sử dụng đặc tính *namespace* trong ngôn ngữ C#, nhằm tránh sự xung đột giữa việc sử dụng các thư viện khác nhau từ các nhà cung cấp. Ngoài ra, namespace được xem như là tập hợp các lớp đối tượng, và cung cấp duy nhất các định danh cho các kiểu dữ liệu và được đặt trong một cấu trúc phân cấp. Việc sử dụng namespace trong khi lập trình là một thói quen tốt, bởi vì công việc này chính là cách lưu các mã nguồn để sử dụng về sau. Ngoài thư viện namespace do MS.NET và các hãng thứ ba cung cấp, ta có thể tạo riêng cho mình các namespace. C# đưa ra từ khóa **using** để khai báo sử dụng namespace trong chương trình:

```
using < Tên namespace >
```

Để tạo một namespace dùng cú pháp sau:

```

namespace <Tên namespace>
{
    < Định nghĩa lớp A>
    < Định nghĩa lớp B >
    .....
}

```

Đoạn ví dụ 3.20 minh họa việc tạo một namespace.




Ví dụ 3.20: Tạo một namespace.

```

namespace MyLib
{
    using System;
    public class Tester
    {
        public static int Main()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine( "i: {0}", i);
            }
            return 0;
        }
    }
}

```

Ví dụ trên tạo ra một namespace có tên là MyLib, bên trong namespace này chứa một lớp có tên là Tester. C# cho phép trong một namespace có thể tạo một namespace khác lồng bên trong và không giới hạn mức độ phân cấp này, việc phân cấp này được minh họa trong ví dụ 3.21.

 *Ví dụ 3.21: Tạo các namespace lồng nhau.*

```

namespace MyLib
{
    namespace Demo
    {
        using System;
        public class Tester
        {
            public static int Main()
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine( "i: {0}", i);
                }
                return 0;
            }
        }
    }
}

```

```


    }
}

```

Lớp Tester trong ví dụ 3.21 được đặt trong namespace Demo do đó có thể tạo một lớp Tester khác bên ngoài namespace Demo hay bên ngoài namespace MyLib mà không có bất cứ sự tranh chấp hay xung đột nào. Để truy cập lớp Tester dùng cú pháp sau:

MyLib.Demo.Tester

Trong một namespace một lớp có thể gọi một lớp khác thuộc các cấp namespace khác nhau, ví dụ tiếp sau minh họa việc gọi một hàm thuộc một lớp trong namespace khác.

 Ví dụ 3.22: Gọi một namespace thành viên.

```

using System;
namespace MyLib
{
    namespace Demo1
    {
        class Example1
        {
            public static void Show1()
            {
                Console.WriteLine("Lop Example1");
            }
        }
    }
    namespace Demo2
    {
        public class Tester
        {
            public static int Main()
            {
                Demo1.Example1.Show1();
                Demo1.Example2.Show2();
                return 0;
            }
        }
    }
}
// Lớp Example2 có cùng namespace MyLib.Demo1 với

```

```
//lớp Example1 nhưng hai khai báo không cùng một khối.
namespace MyLib.Demo1
```

```
{
    class Example2
    {
        public static void Show2()
        {
            Console.WriteLine("Lop Example2");
        }
    }
}
```

Kết quả:

Lop Example1

Lop Example2

Ví dụ 3.22 trên có hai điểm cần lưu ý là cách gọi một namespace thành viên và cách khai báo các namespace. Như chúng ta thấy trong namespace MyLib có hai namespace con cùng cấp là Demo1 và Demo2, hàm Main của Demo2 sẽ được chương trình thực hiện, và trong hàm Main này có gọi hai hàm thành viên tĩnh của hai lớp Example1 và Example2 của namespace Demo1.

Ví dụ trên cũng đưa ra cách khai báo khác các lớp trong namespace. Hai lớp Example1 và Example2 đều cùng thuộc một namespace MyLib.Demo1, tuy nhiên Example2 được khai báo một khối riêng lẻ bằng cách sử dụng khai báo:

```
namespace MyLib.Demo1
```

```
{
    class Example2
    {
        ....
    }
}
```

Việc khai báo riêng lẻ này có thể cho phép trên nhiều tập tin nguồn khác nhau, miễn sao đảm bảo khai báo đúng tên namespace thì chúng vẫn thuộc về cùng một namespace. ***Các chỉ dẫn biên dịch***

Đối với các ví dụ minh họa trong các phần trước, khi biên dịch thì toàn bộ chương trình sẽ được biên dịch. Tuy nhiên, có yêu cầu thực tế là chúng ta chỉ muốn một phần trong

chương trình được biên dịch độc lập, ví dụ như khi debug chương trình hoặc xây dựng các ứng dụng...

Trước khi một mã nguồn được biên dịch, một chương trình khác được gọi là chương trình tiền xử lý sẽ thực hiện trước và chuẩn bị các đoạn mã nguồn để biên dịch. Chương trình tiền xử lý này sẽ tìm trong mã nguồn các kí hiệu chỉ dẫn biên dịch đặc biệt, tất cả các chỉ dẫn biên dịch này đều được bắt đầu với dấu rêu (#). Các chỉ dẫn cho phép chúng ta định nghĩa các định danh và kiểm tra các sự tồn tại của các định danh đó.

Định nghĩa định danh

Câu lệnh tiền xử lý sau:

```
#define DEBUG
```

Lệnh trên định nghĩa một định danh tiền xử lý có tên là DEBUG. Mặc dù những chỉ thị tiền xử lý khác có thể được đặt bất cứ ở đâu trong chương trình, nhưng với chỉ thị định nghĩa định danh thì phải đặt trước tất cả các lệnh khác, bao gồm cả câu lệnh using.

Để kiểm tra một định danh đã được định nghĩa thì ta dùng cú pháp `#if <định danh>`. Do đó ta có thể viết như sau:

```
#define DEBUG
//...Các đoạn mã nguồn bình thường, không bị tác động bởi trình tiền xử lý
...
#if DEBUG
    // Các đoạn mã nguồn trong khối if debug được biên dịch
#else
    // Các đoạn mã nguồn không định nghĩa debug và không được biên dịch
#endif
//...Các đoạn mã nguồn bình thường, không bị tác động bởi trình tiền xử lý
```

Khi chương trình tiền xử lý thực hiện, chúng sẽ tìm thấy câu lệnh `#define DEBUG` và lưu lại định danh DEBUG này. Tiếp theo trình tiền xử lý này sẽ bỏ qua tất cả các đoạn mã bình thường khác của C# và tìm các khối `#if`, `#else`, và `#endif`.

Câu lệnh `#if` sẽ kiểm tra định danh DEBUG, do định danh này đã được định nghĩa, nên đoạn mã nguồn giữa khối `#if` đến `#else` sẽ được biên dịch vào chương trình. Còn đoạn mã nguồn giữa `#else` và `#endif` sẽ không được biên dịch. Tức là đoạn mã nguồn này sẽ không được thực hiện hay xuất hiện bên trong mã hợp ngữ của chương trình.

Trường hợp câu lệnh `#if` sai tức là không có định nghĩa một định danh DEBUG trong chương trình, khi đó đoạn mã nguồn ở giữa khối `#if` và `#else` sẽ không được đưa vào chương trình để biên dịch mà ngược lại đoạn mã nguồn ở giữa khối `#else` và `#endif` sẽ được biên dịch.

Lưu ý: Tất cả các đoạn mã nguồn bên ngoài `#if` và `#endif` thì không bị tác động bởi trình tiền xử lý và tất cả các mã này đều được đưa vào để biên dịch.

Không định nghĩa định danh

Sử dụng chỉ thị tiền xử lý `#undef` để xác định trạng thái của một định danh là không được định nghĩa. Như chúng ta đã biết trình tiền xử lý sẽ thực hiện từ trên xuống dưới, do vậy một định danh đã được khai báo bên trên với chỉ thị `#define` sẽ có hiệu quả đến khi một gọi câu lệnh `#undef` định danh đó hay đến cuối chương trình:

```
#define DEBUG
#if DEBUG
    // Đoạn code này được biên dịch
#endif
....
#undef DEBUG
....
#if DEBUG
    // Đoạn code này không được biên dịch
#endif
.....
```

`#if` đầu tiên đúng do `DEBUG` được định nghĩa, còn `#if` thứ hai sai không được biên dịch vì `DEBUG` đã được định nghĩa lại là `#undef`.

Ngoài ra còn có chỉ thị `#elif` và `#else` cung cấp các chỉ dẫn phức tạp hơn. Chỉ dẫn `#elif` cho phép sử dụng logic “else-if”. Ta có thể diễn giải một chỉ dẫn như sau: “*Nếu `DEBUG` thì làm công việc 1, ngược lại nếu `TEST` thì làm công việc 2, nếu sai tất cả thì làm trường hợp 3*”:

```
....
#if DEBUG
    // Đoạn code này được biên dịch nếu DEBUG được định nghĩa
#elif TEST
    //Đoạn code này được biên dịch nếu DEBUG không được định nghĩa
    // và TEST được định nghĩa
#else
    //Đoạn code này được biên dịch nếu cả DEBUG và
    //TEST không được định nghĩa.
#endif
.....
```

Trong ví dụ trên thì chỉ thị tiền xử lý `#if` đầu tiên sẽ kiểm tra định danh `DEBUG`, nếu định danh `DEBUG` đã được định nghĩa thì đoạn mã nguồn ở giữa `#if` và `#elif` sẽ được biên dịch, và tất cả các phần còn lại cho đến chỉ thị `#endif` đều không được biên dịch. Nếu `DEBUG` không được định nghĩa thì `#elif` sẽ kiểm tra định danh `TEST`, đoạn mã ở giữa `#elif` và `#else` sẽ được

thực thi khi TEST được định nghĩa. Cuối cùng nếu cả hai DEBUG và TEST đều không được định nghĩa thì các đoạn mã nguồn giữa #else và #endif sẽ được biên dịch. **Câu hỏi và trả lời**

Câu hỏi 1: Sự khác nhau giữa dựa trên thành phần (Component-Based) và hướng đối tượng (Object- Oriented)?

Trả lời 1: Phát triển dựa trên thành phần có thể được xem như là mở rộng của lập trình hướng đối tượng. Một thành phần là một khối mã nguồn riêng có thể thực hiện một nhiệm vụ đặc biệt. Lập trình dựa trên thành phần bao gồm việc tạo nhiều các thành phần tự hoạt động có thể được dùng lại. Sau đó chúng ta có thể liên kết chúng lại để xây dựng các ứng dụng.

Câu hỏi 2: Những ngôn ngữ nào khác được xem như là hướng đối tượng?

Trả lời 2: Các ngôn ngữ như là C++, Java, SmallTalk, Visual Basic.NET cũng có thể được sử dụng cho lập trình hướng đối tượng. Còn rất nhiều những ngôn ngữ khác nhưng không được phổ biến lắm.

Câu hỏi 3: Tại sao trong kiểu số không nên khai báo kiểu dữ liệu lớn thay vì dùng kiểu dữ liệu nhỏ hơn?

Trả lời 3: Mặc dù điều có thể xem là khá hợp lý, nhưng thật sự không hiệu quả lắm. Chúng ta không nên sử dụng nhiều tài nguyên bộ nhớ hơn mức cần thiết. Khi đó vừa lãng phí bộ nhớ lại vừa hạn chế tốc độ của chương trình.

Câu hỏi 4: Chuyện gì xảy ra nếu ta gán giá trị âm vào biến kiểu không dấu?

Trả lời 4: Chúng ta sẽ nhận được lỗi của trình biên dịch nói rằng không thể gán giá trị âm cho biến không dấu trong trường hợp ta gán giá trị hằng âm. Còn nếu trong trường hợp kết quả là âm được tính trong biểu thức khi chạy chương trình thì chúng ta sẽ nhận được lỗi dữ liệu. Việc kiểm tra và xử lý lỗi dữ liệu sẽ được trình bày trong các phần sau.

Câu hỏi 5: Những ngôn ngữ nào khác hỗ trợ Common Type System (CTS) trong Common Language Runtime (CLR)?

Trả lời 5: Microsoft Visual Basic (Version 7), Visual C++.NET cũng hỗ trợ CTS. Thêm vào đó là một số phiên bản của ngôn ngữ khác cũng được chuyển vào CTS. Bao gồm Python, COBOL, Perl, Java. Chúng ta có thể xem trên trang web của Microsoft để biết thêm chi tiết.

Câu hỏi 6: Có phải còn những câu lệnh điều khiển khác?

Trả lời 6: Đúng, các câu lệnh này như sau: throw, try, catch và finally. Chúng ta sẽ được học trong chương xử lý ngoại lệ.

Câu hỏi 7: Có thể sử dụng chuỗi với câu lệnh switch?

Trả lời 7: Hoàn toàn được, chúng ta sử dụng biến giá trị chuỗi trong **switch** rồi sau đó dùng giá trị chuỗi trong câu lệnh case. Lưu ý là chuỗi là những ký tự đơn giản nằm giữa hai dấu ngoặc nháy.

Câu hỏi thêm

Câu hỏi 1: Có bao nhiêu cách khai báo comment trong ngôn ngữ C#, cho biết chi tiết?

Câu hỏi 2: Những từ theo sau từ nào là từ khóa trong C#: *field, cast, as, object, throw, football, do, get, set, basketball*.

Câu hỏi 3: Những khái niệm chính của ngôn ngữ lập trình hướng đối tượng?

Câu hỏi 4: Sự khác nhau giữa hai lệnh *Write* và *WriteLine*?

Câu hỏi 5: C# chia làm mấy kiểu dữ liệu chính? Nếu ta tạo một lớp tên *myClass* thì lớp này được xếp vào kiểu dữ liệu nào?

Câu hỏi 6: Kiểu chuỗi trong C# là kiểu dữ liệu nào?

Câu hỏi 7: Dữ liệu của biến kiểu dữ liệu tham chiếu được lưu ở đâu trong bộ nhớ?

Câu hỏi 8: Sự khác nhau giữa lớp và cấu trúc trong C#? Khi nào thì dùng cấu trúc tốt hơn là dùng *class*?

Câu hỏi 8: Sự khác nhau giữa kiểu *unsigned* và *signed* trong kiểu số nguyên?

Câu hỏi 9: Kiểu dữ liệu nào nhỏ nhất có thể lưu trữ được giá trị 45?

Câu hỏi 10: Số lớn nhất, và nhỏ nhất của kiểu *int* là số nào?

Câu hỏi 11: Có bao nhiêu bit trong một byte?

Câu hỏi 12: Kiểu dữ liệu nào trong .NET tương ứng với kiểu *int* trong C#?

Câu hỏi 13: Những từ khóa nào làm thay đổi luồng của chương trình?

Câu hỏi 14: Kết quả của $15\%4$ là bao nhiêu?

Câu hỏi 15: Sự khác nhau giữa chuyển đổi tường minh và chuyển đổi ngầm định?

Câu hỏi 16: Có thể chuyển từ một giá trị long sang giá trị *int* hay không?

Câu hỏi 17: Số lần tối thiểu các lệnh trong **while** được thực hiện?

Câu hỏi 18: Số lần tối thiểu các lệnh trong do **while** được thực hiện?

Câu hỏi 19: Lệnh nào dùng để thoát ra khỏi vòng lặp?

Câu hỏi 20: Lệnh nào dùng để qua vòng lặp kế tiếp?

Câu hỏi 21: Khi nào dùng biến và khi nào dùng hằng?

Câu hỏi 22: Cho biết giá trị *CanhCut* trong kiểu liệt kê sau:

```
enum LoaiChim
{
    HaiAu,
    BoiCa,
    DaiBang = 50,
    CanhCut
}
```

Câu hỏi 23: Cho biết các lệnh phân nhánh trong C#?

Bài tập

Bài tập 1: Nhập vào, biên dịch và chạy chương trình. Hãy cho biết chương trình làm điều gì?

```

class BaiTap3_1
{
    public static void Main()
    {
        int x = 0;
        for(x = 1; x < 10; x++)
        {
            System.Console.Write("{0:03}", x);
        }
    }
}

```

Bài tập 2: Tìm lỗi của chương trình sau? sửa lỗi và biên dịch chương trình.

```

class BaiTap3_2
{
    public static void Main()
    {
        for(int i=0; i < 10 ; i++)
            System.Console.WriteLine("so :{1}", i);
    }
}

```

Bài tập 3: Tìm lỗi của chương trình sau. Sửa lỗi và biên dịch lại chương trình.

```

using System;
class BaiTap3_3
{
    public static void Main()
    {
        double myDouble;
        decimal myDecimal;
        myDouble = 3.14;
        myDecimal = 3.14;
        Console.WriteLine("My Double: {0}", myDouble);
        Console.WriteLine("My Decimal: {0}", myDecimal);
    }
}

```

Bài tập 4: Tìm lỗi của chương trình sau. Sửa lỗi và biên dịch lại chương trình.

```
class BaiTap3_4
{
    static void Main()
    {
        int value;
        if (value > 100);
            System.Console.WriteLine("Number is greater than 100");
    }
}
```

Bài tập 5: Viết chương trình hiển thị ra màn hình 3 kiểu sau:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

a)

```
$ $ $ $ $ $
$ $ $ $ $
$ $ $ $
$ $ $
$ $
$ $
$
```

b)

```
          *
        * * *
      * * * * *
    * * * * * *
  * * * * * * *
* * * * * * * *
```

c)

Bài tập 6: Viết chương trình hiển thị ra trên màn hình.

1
 2 3 2
 3 4 5 4 3
 4 5 6 7 6 5 4
 5 6 7 8 9 8 7 6 5
 6 7 8 9 0 1 0 9 8 7 6
 7 8 9 0 1 2 3 2 1 0 9 8 7
 8 9 0 1 2 3 4 5 4 3 2 1 0 9 8
 9 0 1 2 3 4 5 6 7 6 5 4 3 2 1 0 9
 0 1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1 0

Bài tập 7: Viết chương trình in ký tự số (0..9) và ký tự chữ (a..z) với mã ký tự tương ứng của từng ký tự

Ví dụ:

'0' : 48

'1' : 49

....

Bài tập 8: Viết chương trình giải phương trình bậc nhất, cho phép người dùng nhập vào giá trị a, b.

Bài tập 9: Viết chương trình giải phương trình bậc hai, cho phép người dùng nhập vào giá trị a, b, c.

Bài tập 10: Viết chương trình tính chu vi và diện tích của các hình sau: đường tròn, hình chữ nhật, hình thang, tam giác.

Chương 4

XÂY DỰNG LỚP - ĐỐI TƯỢNG

- **Định nghĩa lớp**
 - Thuộc tính truy cập
 - Tham số của phương thức
- **Tạo đối tượng**
 - Bộ khởi dựng
 - Khởi tạo biến thành viên
 - Bộ khởi dựng sao chép
 - Từ khóa this
- **Sử dụng các thành viên static**
 - Gọi phương thức static
 - Sử dụng bộ khởi dựng static
 - Sử dụng bộ khởi dựng private
 - Sử dụng thuộc tính static
- **Hủy đối tượng**
- **Truyền tham số**
- **Nạp chồng phương thức**
- **Đóng gói dữ liệu với thành phần thuộc tính**
- **Thuộc tính chỉ đọc**
- **Câu hỏi & bài tập**

Chương 3 thảo luận rất nhiều kiểu dữ liệu cơ bản của ngôn ngữ C#, như int, long and char. Tuy nhiên trái tim và linh hồn của C# là khả năng tạo ra những kiểu dữ liệu mới, phức

tập. Người lập trình tạo ra các kiểu dữ liệu mới bằng cách xây dựng các lớp đối tượng và đó cũng chính là các vấn đề chúng ta cần thảo luận trong chương này.

Đây là khả năng để tạo ra những kiểu dữ liệu mới, một đặc tính quan trọng của ngôn ngữ lập trình hướng đối tượng. Chúng ta có thể xây dựng những kiểu dữ liệu mới trong ngôn ngữ C# bằng cách khai báo và định nghĩa những lớp. Ngoài ra ta cũng có thể định nghĩa các kiểu dữ liệu với những giao diện (interface) sẽ được bàn trong Chương 8 sau. Thể hiện của một lớp được gọi là những đối tượng (object). Những đối tượng này được tạo trong bộ nhớ khi chương trình được thực hiện.

Sự khác nhau giữa một lớp và một đối tượng cũng giống như sự khác nhau giữa khái niệm giữa loài mèo và một con mèo Mun đang nằm bên chân của ta. Chúng ta không thể đụng chạm hay đùa giỡn với khái niệm mèo nhưng có thể thực hiện điều đó được với mèo Mun, nó là một thực thể sống động, chứ không trừu tượng như khái niệm họ loài mèo.

Một họ mèo mô tả những con mèo có các đặc tính: có trọng lượng, có chiều cao, màu mắt, màu lông,...chúng cũng có hành động như là ăn ngủ, leo trèo,...một con mèo, ví dụ như mèo Mun chẳng hạn, nó cũng có trọng lượng xác định là 5 kg, chiều cao 15 cm, màu mắt đen, lông đen...Nó cũng có những khả năng như ăn ngủ leo trèo,...

Lợi ích to lớn của những lớp trong ngôn ngữ lập trình là khả năng đóng gói các thuộc tính và tính chất của một thực thể trong một khối đơn, tự có nghĩa, tự khả năng duy trì. Ví dụ khi chúng ta muốn sắp nội dung những thể hiện hay đối tượng của lớp điều khiển ListBox trên Windows, chỉ cần gọi các đối tượng này thì chúng sẽ tự sắp xếp, còn việc chúng làm ra sao thì ta không quan tâm, và cũng chỉ cần biết bấy nhiêu đó thôi.

Đóng gói cùng với đa hình (polymorphism) và kế thừa (inheritance) là các thuộc tính chính yếu của bất kỳ một ngôn ngữ lập trình hướng đối tượng nào.

Chương 4 này sẽ trình bày các đặc tính của ngôn ngữ lập trình C# để xây dựng các lớp đối tượng. Thành phần của một lớp, các hành vi và các thuộc tính, được xem như là thành viên của lớp (class member). Tiếp theo chương cũng trình bày khái niệm về phương thức (method) được dùng để định nghĩa hành vi của một lớp, và trạng thái của các biến thành viên hoạt động trong một lớp. Một đặc tính mới mà ngôn ngữ C# đưa ra để xây dựng lớp là khái niệm thuộc tính (property), thành phần thuộc tính này hoạt động giống như cách phương thức để tạo một lớp, nhưng bản chất của phương thức này là tạo một lớp giao diện cho bên ngoài tương tác với biến thành viên một cách gián tiếp, ta sẽ bàn sâu vấn đề này trong chương.

Định nghĩa lớp

Để định nghĩa một kiểu dữ liệu mới hay một lớp đầu tiên phải khai báo rồi sau đó mới định nghĩa các thuộc tính và phương thức của kiểu dữ liệu đó. Khai báo một lớp bằng cách sử dụng từ khóa **class**. Cú pháp đầy đủ của khai báo một lớp như sau:

```
[Thuộc tính] [Bổ sung truy cập] class <Định danh lớp> [: Lớp cơ sở]
{
```


<Phần thân của lớp: bao gồm định nghĩa các thuộc tính và phương thức hành động >

}

Thành phần thuộc tính của đối tượng sẽ được trình bày chi tiết trong chương sau, còn thành phần bổ sung truy cập cũng sẽ được trình bày tiếp ngay mục dưới. Định danh lớp chính là tên của lớp do người xây dựng chương trình tạo ra. Lớp cơ sở là lớp mà đối tượng sẽ kế thừa để phát triển ta sẽ bàn sau. Tất cả các thành viên của lớp được định nghĩa bên trong thân của lớp, phần thân này sẽ được bao bọc bởi hai dấu ({}).

Ghi chú: Trong ngôn ngữ C# phần kết thúc của lớp không có dấu chấm phẩy giống như khai báo lớp trong ngôn ngữ C/C++. Tuy nhiên nếu người lập trình thêm vào thì trình biên dịch C# vẫn chấp nhận mà không đưa ra cảnh báo lỗi.

Trong C#, mọi chuyện đều xảy ra trong một lớp. Như các ví dụ mà chúng ta đã tìm hiểu trong chương 3, các hàm điều được đưa vào trong một lớp, kể cả hàm đầu vào của chương trình (hàm Main()):

```
public class Tester
{

    public static int Main()
    {
        //....
    }
}
```

Điều cần nói ở đây là chúng ta chưa tạo bất cứ thể hiện nào của lớp, tức là tạo đối tượng cho lớp Tester. Điều gì khác nhau giữa một lớp và thể hiện của lớp? để trả lời cho câu hỏi này chúng ta bắt đầu xem xét sự khác nhau giữa kiểu dữ liệu int và một biến kiểu int . Ta có viết như sau:

```
int var1 = 10;
```

tuy nhiên ta không thể viết được

```
int = 10;
```

Ta không thể gán giá trị cho một kiểu dữ liệu, thay vào đó ta chỉ được gán dữ liệu cho một đối tượng của kiểu dữ liệu đó, trong trường hợp trên đối tượng là biến var1.

Khi chúng ta tạo một lớp mới, đó chính là việc định nghĩa các thuộc tính và hành vi của tất cả các đối tượng của lớp. Giả sử chúng ta đang lập trình để tạo các điều khiển trong các ứng dụng trên Windows, các điều khiển này giúp cho người dùng tương tác tốt với Windows, như là ListBox, TextBox, ComboBox,...Một trong những điều khiển thông dụng là ListBox, điều khiển này cung cấp một danh sách liệt kê các mục chọn và cho phép người dùng chọn các mục tin trong đó.


ListBox này cũng có các thuộc tính khác nhau như: chiều cao, bề dày, vị trí, và màu sắc thể hiện và các hành vi của chúng như: chúng có thể thêm bớt mục tin, sắp xếp,...

Ngôn ngữ lập trình hướng đối tượng cho phép chúng ta tạo kiểu dữ liệu mới là lớp ListBox, lớp này bao bọc các thuộc tính cũng như khả năng như: các thuộc tính height, width, location, color, các phương thức hay hành vi như Add(), Remove(), Sort(),...

Chúng ta không thể gán dữ liệu cho kiểu ListBox, thay vào đó đầu tiên ta phải tạo một đối tượng cho lớp đó:

```
ListBox myListBox;
```

Một khi chúng ta đã tạo một thể hiện của lớp ListBox thì ta có thể gán dữ liệu cho thể hiện đó. Tuy nhiên đoạn lệnh trên chưa thể tạo đối tượng trong bộ nhớ được, ta sẽ bàn tiếp. Bây giờ ta sẽ tìm hiểu cách tạo một lớp và tạo các thể hiện thông qua ví dụ minh họa 4.1. Ví dụ này tạo một lớp có chức năng hiển thị thời gian trong một ngày. Lớp này có hành vi thể hiện ngày, tháng, năm, giờ, phút, giây hiện hành. Để làm được điều trên thì lớp này có 6 thuộc tính hay còn gọi là biến thành viên, cùng với một phương thức như sau:

 *Ví dụ 4.1: Tạo một lớp ThoiGian đơn giản như sau.*

```
using System;

public class ThoiGian
{
    public void ThoiGianHienHanh()
    {
        Console.WriteLine("Hien thi thoi gian hien hanh");
    }

    // Các biến thành viên
    int Nam;
    int Thang;
    int Ngay;
    int Gio;
    int Phut;
    int Giay;
}

public class Tester
{
    static void Main()
    {
        ThoiGian t = new ThoiGian();
        t.ThoiGianHienHanh();
    }
}
```

Kết quả:

Hien thi thoi gian hien hanh

Lớp ThoiGian chỉ có một phương thức chính là hàm ThoiGianHienHanh(), phần thân của phương thức này được định nghĩa bên trong của lớp ThoiGian. Điều này khác với ngôn ngữ C++, C# không đòi hỏi phải khai báo trước khi định nghĩa một phương thức, và cũng không hỗ trợ việc khai báo phương thức trong một tập tin và sau đó định nghĩa ở một tập tin khác. C# không có các tập tin tiêu đề, do vậy tất cả các phương thức được định nghĩa hoàn toàn bên trong của lớp. Phần cuối của định nghĩa lớp là phần khai báo các biến thành viên: Nam, Thang, Ngay, Gio, Phut, va Giay.

Sau khi định nghĩa xong lớp ThoiGian, thì tiếp theo là phần định nghĩa lớp Tester, lớp này có chứa một hàm khá thân thiện với chúng ta là hàm Main(). Bên trong hàm Main có một thể hiện của lớp ThoiGian được tạo ra và gán giá trị cho đối tượng t. Bởi vì t là thể hiện của đối tượng ThoiGian, nên hàm Main() có thể sử dụng phương thức của t:

```
t.ThoiGianHienHanh();
```

Thuộc tính truy cập

Thuộc tính truy cập quyết định khả năng các phương thức của lớp bao gồm việc các phương thức của lớp khác có thể nhìn thấy và sử dụng các biến thành viên hay những phương thức bên trong lớp. Bảng 4.1 tóm tắt các thuộc tính truy cập của một lớp trong C#.

Thuộc tính	Giới hạn truy cập
public	Không hạn chế. Những thành viên được đánh dấu public có thể được dùng bởi bất kì các phương thức của lớp bao gồm những lớp khác.
private	Thành viên trong một lớp A được đánh dấu là private thì chỉ được truy cập bởi các phương thức của lớp A.
protected	Thành viên trong lớp A được đánh dấu là protected thì chỉ được các phương thức bên trong lớp A và những phương thức dẫn xuất từ lớp A truy cập.
internal	Thành viên trong lớp A được đánh dấu là internal thì được truy cập bởi những phương thức của bất cứ lớp nào trong cùng khối hợp ngữ với A.
protected internal	Thành viên trong lớp A được đánh dấu là protected internal được truy cập bởi các phương thức của lớp A, các phương thức của lớp dẫn xuất của A, và bất cứ lớp nào trong cùng khối hợp ngữ của A.

Bảng 4.1: Thuộc tính truy cập.

Mong muốn chung là thiết kế các biến thành viên của lớp ở thuộc tính **private**. Khi đó chỉ có phương thức thành viên của lớp truy cập được giá trị của biến. C# xem thuộc tính **private** là mặc định nên trong ví dụ 4.1 ta không khai báo thuộc tính truy cập cho 6 biến nên mặc định chúng là **private**:

```
// Các biến thành viên private
int Nam;
int Thang;
int Ngay;
int Gio;
int Phut;
int Giay;
```

Do lớp Tester và phương thức thành viên ThoiGianHienHanH của lớp ThoiGian được khai báo là public nên bất kỳ lớp nào cũng có thể truy cập được.

Ghi chú: Thói quen lập trình tốt là khai báo tường minh các thuộc tính truy cập của biến thành viên hay các phương thức trong một lớp. Mặc dù chúng ta biết chắc chắn rằng các thành viên của lớp là được khai báo **private** mặc định. Việc khai báo tường minh này sẽ làm cho chương trình dễ hiểu, rõ ràng và tự nhiên hơn.


Tham số của phương thức

Trong các ngôn ngữ lập trình thì tham số và đối mục được xem là như nhau, cũng tương tự khi đang nói về ngôn ngữ hướng đối tượng thì ta gọi một hàm là một phương thức hay hành vi. Tất cả các tên này đều tương đồng với nhau.

Một phương thức có thể lấy bất kỳ số lượng tham số nào, Các tham số này theo sau bởi tên của phương thức và được bao bọc bên trong dấu ngoặc tròn (). Mỗi tham số phải khai báo kèm với kiểu dữ liệu. ví dụ ta có một khai báo định nghĩa một phương thức có tên là Method, phương thức không trả về giá trị nào cả (khai báo giá trị trả về là void), và có hai tham số là một kiểu int và button:

```
void Method( int param1, button param2)
{
    //...
}
```

Bên trong thân của phương thức, các tham số này được xem như những biến cục bộ, giống như là ta khai báo biến bên trong phương thức và khởi tạo giá trị bằng giá trị của tham số truyền vào. Ví dụ 4.2 minh họa việc truyền tham số vào một phương thức, trong trường hợp này thì hai tham số của kiểu là int và float.

 *Ví dụ 4.2: Truyền tham số cho phương thức.*

```
using System;
public class Class1
{
    public void SomeMethod(int p1, float p2)
    {
        Console.WriteLine("Ham nhan duoc hai tham so: {0} va {1}",
            p1,p2);
    }
}
public class Tester
{
    static void Main()
    {
        int var1 = 5;
        float var2 = 10.5f;
        Class1 c = new Class1();
        c.SomeMethod( var1, var2 );
    }
}
```

Kết quả:

Ham nhan duoc hai tham so: 5 va 10.5

Phương thức SomeMethod sẽ lấy hai tham số int và float rồi hiển thị chúng ta màn hình bằng việc dùng hàm Console.WriteLine(). Những tham số này có tên là p1 và p2 được xem như là biến cục bộ bên trong của phương thức.

Trong phương thức gọi Main, có hai biến cục bộ được tạo ra là var1 và var2. Khi hai biến này được truyền cho phương thức SomeMethod thì chúng được ánh xạ thành hai tham số p1 và p2 theo thứ tự danh sách biến đưa vào.

Tạo đối tượng

Trong Chương 3 có đề cập đến sự khác nhau giữa kiểu dữ liệu giá trị và kiểu dữ liệu tham chiếu. Những kiểu dữ liệu chuẩn của C# như int, char, float,... là những kiểu dữ liệu giá trị, và các biến được tạo ra từ các kiểu dữ liệu này được lưu trên stack. Tuy nhiên, với các đối tượng kiểu dữ liệu tham chiếu thì được tạo ra trên heap, sử dụng từ khóa **new** để tạo một đối tượng:

```
ThoiGian t = new ThoiGian();
```

t thật sự không chứa giá trị của đối tượng ThoiGian, nó chỉ chứa địa chỉ của đối tượng được tạo ra trên heap, do vậy t chỉ chứa tham chiếu đến một đối tượng mà thôi.

Bộ khởi dựng

Thử xem lại ví dụ minh họa 4.1, câu lệnh tạo một đối tượng cho lớp ThoiGian tương tự như việc gọi thực hiện một phương thức:

```
ThoiGian t = new ThoiGian();
```

Đúng như vậy, một phương thức sẽ được gọi thực hiện khi chúng ta tạo một đối tượng. Phương thức này được gọi là bộ khởi dựng (constructor). Các phương thức này được định nghĩa khi xây dựng lớp, nếu ta không tạo ra thì CLR sẽ thay mặt chúng ta mà tạo phương thức khởi dựng một cách mặc định. Chức năng của bộ khởi dựng là tạo ra đối tượng được xác định bởi một lớp và đặt trạng thái này hợp lệ. Trước khi bộ khởi dựng được thực hiện thì đối tượng chưa được cấp phát trong bộ nhớ. Sau khi bộ khởi dựng thực hiện hoàn thành thì bộ nhớ sẽ lưu giữ một thể hiện hợp lệ của lớp vừa khai báo.


Lớp ThoiGian trong ví dụ 4.1 không định nghĩa bộ khởi dựng. Do không định nghĩa nên trình biên dịch sẽ cung cấp một bộ khởi dựng cho chúng ta. Phương thức khởi dựng mặc định được tạo ra cho một đối tượng sẽ không thực hiện bất cứ hành động nào, tức là bên trong thân của phương thức rỗng. Các biến thành viên được khởi tạo các giá trị tầm thường như thuộc tính nguyên có giá trị là 0 và chuỗi thì khởi tạo rỗng,..Bảng 4.2 sau tóm tắt các giá trị mặc định được gán cho các kiểu dữ liệu cơ bản.

Kiểu dữ liệu	Giá trị mặc định
int, long, byte,...	0
bool	false
char	'\0' (null)
enum	0
reference	null

Bảng 4.2: Giá trị mặc định của kiểu dữ liệu cơ bản.

Thường thường, khi muốn định nghĩa một phương thức khởi dựng riêng ta phải cung cấp các tham số để hàm khởi dựng có thể khởi tạo các giá trị khác ngoài giá trị mặc định cho các đối tượng. Quay lại ví dụ 4.1 giả sử ta muốn truyền thời gian hiện hành: năm, tháng, ngày,...để đối tượng có ý nghĩa hơn.

Để định nghĩa một bộ khởi dựng riêng ta phải khai báo một phương thức có tên giống như tên lớp đã khai báo. Phương thức khởi dựng không có giá trị trả về và được khai báo là public. Nếu phương thức khởi dựng này được truyền tham số thì phải khai báo danh sách tham số giống như khai báo với bất kỳ phương thức nào trong một lớp. Ví dụ 4.3 được viết lại từ ví dụ 4.1 và thêm một bộ khởi dựng riêng, phương thức khởi dựng này sẽ nhận một tham số là một đối tượng kiểu DateTime do C# cung cấp.

 Ví dụ 4.3: Định nghĩa một bộ khởi dựng.

```
using System;

public class ThoiGian
{
    public void ThoiGianHienHanh()
    {
        Console.WriteLine(" Thoi gian hien hanh la : {0}/{1}/{2}
                           {3}:{4}:{5}", Ngay, Thang, Nam, Gio, Phut, Giay);
    }
    // Hàm khởi dựng
    public ThoiGian( System.DateTime dt )
    {
        Nam = dt.Year;
        Thang = dt.Month;
        Ngay = dt.Day;
        Gio = dt.Hour;
        Phut = dt.Minute;
        Giay = dt.Second;

    }
    // Biến thành viên private
    int Nam;
    int Thang;
    int Ngay;
    int Gio;
    int Phut;
    int Giay;
}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        ThoiGian t = new ThoiGian( currentTime );
        t.ThoiGianHienHanh();

    }
}
```

Kết quả:

Thời gian hiện hành là: 5/6/2002 9:10:20

Trong ví dụ trên phương thức khởi dựng lấy một đối tượng DateTime và khởi tạo tất cả các biến thành viên dựa trên giá trị của đối tượng này. Khi phương thức này thực hiện xong, một đối tượng ThờiGian được tạo ra và các biến của đối tượng cũng đã được khởi tạo. Hàm ThờiGianHienHanH được gọi trong hàm Main() sẽ hiển thị giá trị thời gian lúc đối tượng được tạo ra.

Chúng ta thử bỏ một số lệnh khởi tạo trong phương thức khởi dựng và cho thực hiện chương trình lại thì các biến không được khởi tạo sẽ có giá trị mặc định là 0, do là biến nguyên. Một biến thành viên kiểu nguyên sẽ được thiết lập giá trị là 0 nếu chúng ta không gán nó trong phương thức khởi dựng. Chú ý rằng kiểu dữ liệu giá trị không thể không được khởi tạo, nếu ta không khởi tạo thì trình biên dịch sẽ cung cấp các giá trị mặc định theo bảng 4.2.

Ngoài ra trong chương trình 4.3 trên có sử dụng đối tượng của lớp DateTime, lớp DateTime này được cung cấp bởi thư viện System, lớp này cũng cung cấp các biến thành viên public như: Year, Month, Day, Hour, Minute, và Second tương tự như lớp ThờiGian của chúng ta. Thêm vào đó là lớp này có đưa ra một phương thức thành viên tĩnh tên là Now, phương thức Now sẽ trả về một tham chiếu đến một thể hiện của một đối tượng DateTime được khởi tạo với thời gian hiện hành.

Theo như trên khi lệnh :

```
System.DateTime currentTime = System.DateTime.Now();
```

được thực hiện thì phương thức tĩnh Now() sẽ tạo ra một đối tượng DateTime trên bộ nhớ heap và trả về một tham chiếu và tham chiếu này được gán cho biến đối tượng currentTime. Sau khi đối tượng currentTime được tạo thì câu lệnh tiếp theo sẽ thực hiện việc truyền đối tượng currentTime cho phương thức khởi dựng để tạo một đối tượng ThờiGian:

```
ThờiGian t = new ThờiGian( currentTime );
```

Bên trong phương thức khởi dựng này tham số dt sẽ tham chiếu đến đối tượng DateTime là đối tượng vừa tạo mà currentTime cũng tham chiếu. Nói cách khác lúc này tham số dt và currentTime cùng tham chiếu đến một đối tượng DateTime trong bộ nhớ. Nhờ vậy phương thức khởi dựng ThờiGian có thể truy cập được các biến thành viên public của đối tượng DateTime được tạo trong hàm Main().


Có một sự nhấn mạnh ở đây là đối tượng DateTime được truyền cho bộ dựng ThờiGian chính là đối tượng đã được tạo trong hàm Main và là kiểu dữ liệu tham chiếu. Do vậy khi thực hiện truyền tham số là một kiểu dữ liệu tham chiếu thì con trỏ được ánh xạ qua chứ hoàn toàn không có một đối tượng nào được sao chép lại.

Khởi tạo biến thành viên

Các biến thành viên có thể được khởi tạo trực tiếp khi khai báo trong quá trình khởi tạo, thay vì phải thực hiện việc khởi tạo các biến trong bộ khởi dựng. Để thực hiện việc khởi tạo này rất đơn giản là việc sử dụng phép gán giá trị cho một biến:

```
private int Giay = 30;    // Khởi tạo
```

Việc khởi tạo biến thành viên sẽ rất có ý nghĩa, vì khi xác định giá trị khởi tạo như vậy thì biến sẽ không nhận giá trị mặc định mà trình biên dịch cung cấp. Khi đó nếu các biến này không được gán lại trong các phương thức khởi dựng thì nó sẽ có giá trị mà ta đã khởi tạo. Ví dụ 4.4 minh họa việc khởi tạo biến thành viên khi khai báo. Trong ví dụ này sẽ có hai bộ dựng ngoài bộ dựng mặc định mà trình biên dịch cung cấp, một bộ dựng thực hiện việc gán giá trị cho tất cả các biến thành viên, còn bộ dựng thứ hai thì cũng tương tự nhưng sẽ không gán giá trị cho biến Giay.

 Ví dụ 4.4: Minh họa sử dụng khởi tạo biến thành viên.

```
public class ThoiGian
{
    public void ThoiGianHienHanh()
    {
        System.DateTime now = System.DateTime.Now;
        System.Console.WriteLine("\n Hien tai: \t {0}/{1}/{2} {3}:{4}:{5}",
            now.Day, now.Month, now.Year, now.Hour, now.Minute, now.Second);
        System.Console.WriteLine(" Thoi Gian:\t {0}/{1}/{2} {3}:{4}:{5}",
            Ngay, Thang, Nam, Gio, Phut, Giay);
    }
    public ThoiGian( System.DateTime dt)
    {
        Nam = dt.Year;
        Thang = dt.Month;
        Ngay = dt.Day;
        Gio = dt.Hour;
        Phut = dt.Minute;
        Giay = dt.Second;    // có gán cho biến thành viên Giay
    }
    public ThoiGian(int Year, int Month, int Date, int Hour, int Minute)
    {
        Nam = Year;
        Thang = Month;
        Ngay = Date;
    }
}
```

```

        Gio = Hour;
        Phut = Minute;

    }

    private int Nam;
    private int Thang;
    private int Ngay;
    private int Gio;
    private int Phut;

    private int Giay = 30 ; // biến được khởi tạo.

}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        ThoiGian t1 = new ThoiGian( currentTime );
        t1.ThoiGianHienHanh();

        ThoiGian t2 = new ThoiGian(2001,7,3,10,5);
        t2.ThoiGianHienHanh();

    }
}

```

Kết quả:

```

Hien tai:    5/6/2002    10:15:5
Thoi Gian:  5/6/2002    10:15:5

```

```

Hien tai:    5/6/2002    10:15:5
Thoi Gian:  3/7/2001    10:5:30

```

Nếu không khởi tạo giá trị của biến thành viên thì bộ khởi dựng mặc định sẽ khởi tạo giá trị là 0 mặc định cho biến thành viên có kiểu nguyên. Tuy nhiên, trong trường hợp này biến thành viên Giay được khởi tạo giá trị 30:

Giay = 30; // Khởi tạo

Trong trường hợp bộ khởi tạo thứ hai không truyền giá trị cho biến Giay nên biến này vẫn lấy giá trị mà ta đã khởi tạo ban đầu là 30:

```

ThoiGian t2 = new ThoiGian(2001, 7, 3, 10, 5);
t2.ThoiGianHienHanh();

```

Ngược lại, nếu một giá trị được gán cho biến `Giay` như trong bộ khởi tạo thứ nhất thì giá trị mới này sẽ được chồng lên giá trị khởi tạo.

Trong ví dụ trên lần đầu tiên tạo đối tượng `ThoiGian` do ta truyền vào đối tượng `DateTime` nên hàm khởi dựng thứ nhất được thực hiện, hàm này sẽ gán giá trị 5 cho biến `Giay`. Còn khi tạo đối tượng `ThoiGian` thứ hai, hàm khởi dựng thứ hai được thực hiện, hàm này không gán giá trị cho biến `Giay` nên biến này vẫn còn lưu giữ lại giá trị 30 khi khởi tạo ban đầu.

Bộ khởi dựng sao chép

Bộ khởi dựng sao chép thực hiện việc tạo một đối tượng mới bằng cách sao chép tất cả các biến từ một đối tượng đã có và cùng một kiểu dữ liệu. Ví dụ chúng ta muốn đưa một đối tượng `ThoiGian` vào bộ khởi dựng lớp `ThoiGian` để tạo một đối tượng `ThoiGian` mới có cùng giá trị với đối tượng `ThoiGian` cũ. Hai đối tượng này hoàn toàn khác nhau và chỉ giống nhau ở giá trị biến thành viên sao khi khởi dựng.

Ngôn ngữ C# không cung cấp bộ khởi dựng sao chép, do đó chúng ta phải tự tạo ra. Việc sao chép các thành phần từ một đối tượng ban đầu cho một đối tượng mới như sau:

```
public ThoiGian( ThoiGian tg)
{
    Nam = tg.Nam;
    Thang = tg.Thang;
    Ngay = tg.Ngay;
    Gio = tg.Gio;
    Phut = tg.Phut;
    Giay = tg.Giay;
}
```

Khi đó ta có thể sao chép từ một đối tượng `ThoiGian` đã hiện hữu như sau:

```
ThoiGian t2 = new ThoiGian( t1 );
```

Trong đó `t1` là đối tượng `ThoiGian` đã tồn tại, sau khi lệnh trên thực hiện xong thì đối tượng `t2` được tạo ra như bản sao của đối tượng `t1`.

Từ khóa this

Từ khóa **this** được dùng để tham chiếu đến thể hiện hiện hành của một đối tượng. Tham chiếu **this** này được xem là con trỏ ẩn đến tất cả các phương thức không có thuộc tính tĩnh trong một lớp. Mỗi phương thức có thể tham chiếu đến những phương thức khác và các biến thành viên thông qua tham chiếu **this** này.

Tham chiếu **this** này được sử dụng thường xuyên theo ba cách:

Sử dụng khi các biến thành viên bị che lấp bởi tham số đưa vào, như trường hợp sau:

```
public void SetYear( int Nam)
{
    this.Nam = Nam;
```

}

Như trong đoạn mã trên phương thức SetYear sẽ thiết lập giá trị của biến thành viên Nam, tuy nhiên do tham số đưa vào có tên là Nam, trùng với biến thành viên, nên ta phải dùng tham chiếu **this** để xác định rõ các biến thành viên và tham số được truyền vào. Khi đó this.Nam chỉ đến biến thành viên của đối tượng, trong khi Nam chỉ đến tham số.

Sử dụng tham chiếu **this** để truyền đối tượng hiện hành vào một tham số của một phương thức của đối tượng khác:

```
public void Method1( OtherClass otherObject )
{
    // Sử dụng tham chiếu this để truyền tham số là bản
    // thân đối tượng đang thực hiện.
    otherObject.SetObject( this );
}
```

Như trên cho thấy khi cần truyền một tham số là chính bản thân của đối tượng đang thực hiện thì ta bắt buộc phải dùng tham chiếu **this** để truyền.

Các thứ ba sử dụng tham chiếu **this** là *mảng chỉ mục* (indexer), phần này sẽ được trình bày chi tiết trong chương 9.

Sử dụng các thành viên tĩnh (static member)

Những thuộc tính và phương thức trong một lớp có thể là những thành viên thể hiện (instance members) hay những thành viên tĩnh (static members). Những thành viên thể hiện hay thành viên của đối tượng liên quan đến thể hiện của một kiểu dữ liệu. Trong khi thành viên tĩnh được xem như một phần của lớp. Chúng ta có thể truy cập đến thành viên tĩnh của một lớp thông qua tên lớp đã được khai báo. Ví dụ chúng ta có một lớp tên là Button và có hai thể hiện của lớp tên là btnUpdate và btnDelete. Và giả sử lớp Button này có một phương thức tĩnh là Show(). Để truy cập phương thức tĩnh này ta viết :

```
Button.Show();
```

Đúng hơn là viết:

```
btnUpdate.Show();
```

Ghi chú: Trong ngôn ngữ C# không cho phép truy cập đến các phương thức tĩnh và các biến thành viên tĩnh thông qua một thể hiện, nếu chúng ta cố làm điều đó thì trình biên dịch C# sẽ báo lỗi, điều này khác với ngôn ngữ C++.

Trong một số ngôn ngữ thì có sự phân chia giữa phương thức của lớp và các phương thức khác (toàn cục) tồn tại bên ngoài không phụ thuộc bất cứ một lớp nào. Tuy nhiên, điều này không cho phép trong C#, ngôn ngữ C# không cho phép tạo các phương thức bên ngoài của lớp, nhưng ta có thể tạo được các phương thức giống như vậy bằng cách tạo các phương thức tĩnh bên trong một lớp.