I. C# CƠ BẢN

Thông tin về Trung Tâm Lập Trình TP

Thông tin về khóa học

Đối tượng

Nền tảng cần có

C# là gì? Vì sao học c#?

1. C# là gì?

C# là một ngôn ngữ lập trình đơn giản, hiện đại, mục đích tổng quát, hướng đối tượng được phát triển bởi Microsoft và được phê chuẩn bởi European Computer Manufacturers Association (ECMA) và International Standards Organization (ISO).

C# được phát triển bởi Anders Hejlsberg và team của ông trong khi phát triển .Net Framework.

C# được thiết kế cho Common Language Infrastructure (CLI), mà gồm Executable Code và Runtime Environment, cho phép chúng ta sử dụng các ngôn ngữ high-level đa dạng trên các nền tảng và cấu trúc máy tính khác nhau.

1. Dưới đây là các lý do làm C# là ngôn ngữ lập trình chuyên nghiệp được sử dụng rộng rãi:

* C# là ngôn ngữ lập trình hiện đại, mục đích tổng quát.
* C# là ngôn ngữ hướng đối tượng tốt nhất(OOP).
* Dễ dàng để học.
* Là một ngôn ngữ được cấu trúc rõ ràng, có nhiều tài liệu hướng dẫn.
* Cung cấp môi trường để tạo ra các chương trình có hiệu quả.
* Có thể được biên dịch trên nhiều nền tảng máy tính khác nhau(Cross platforms).
* Là một phần của .Net Framework.

Các đặc điểm lập trình mạnh mẽ của C#

Cấu trúc C# khá gần với các ngôn ngữ high-level truyền thống, C và C++, và là một ngôn ngữ lập trình hướng đối tượng. Nó có sự giống nhau mạnh mẽ với Java, nó có nhiều đặc điểm lập trình mạnh mẽ mà làm cho nó trở nên ưa thích với các lập trình viên trên toàn thế giới.

Dưới đây là các đặc điểm quan trọng của C#:

* Điều kiện Boolean
* Tự động dọn rác bởi Garbage-Collector (GC)
* Thư viện chuẩn (Standard Library)
* Assembly
* Property và sự kiện (Event)
* Delegate Quản lý sự kiện
* Dễ dàng để sử dụng Generic
* Indexer
* Biên dịch có điều kiện (Conditional Compilation)
* Đa luồng dễ dàng (Multithreading)
* LINQ và Lambda Expression
* Tích hợp với Windows

Cài đặt môi trường (Enviroment)

**Giới thiệu .Net Framework**

.Net Framework là một nền tảng mang tính cách mạng giúp bạn viết các kiểu ứng dụng sau:

* Windows application
* Web application
* Dịch vụ Web

Các ứng dụng .Net Framework là các ứng dụng đa nền tảng. Nền tảng đã được thiết kế theo cách như vậy có thể được sử dụng từ bất kỳ ngôn ngữ lập trình nào: C#, C++, Visual Basic, Jscript, COBOL… Tất cả các ngôn ngữ này có thể truy cập tới nền tảng này cũng như giao tiếp với nhau.

.Net Framework gồm một thư viện code khổng lồ được sử dụng bởi các ngôn ngữ client như C#. Dưới đây là một số thành phần của .Net Framework.

* Common Language Runtime (CLR)
* .Net Framework Class Library
* Common Language Specification
* Common Type System
* Metadata và Assembly
* Windows Form
* ASP.Net và ASP.Net AJAX
* ADO.Net
* Windows Workflow Foundation (WF)
* Windows Presentation Foundation
* Windows Communication Foundation (WCF)
* LINQ

**Integrated Development Environment (IDE) cho C#**

Microsoft cung cấp các công cụ phát triển sau cho lập trình C#:

* Visual Studio 2015(Bảng đề nghị)
* Visual Studio 2013
* Visual Studio 2012

Cú pháp cơ bản (Basic Syntax)

C# là một ngôn ngữ lập trình hướng đối tượng. Trong phương pháp lập trình hướng đối tượng, một chương trình gồm các đối tượng đa dạng mà tương tác với nhau các cách thức của action. Các action mà một đối tượng có thể nhận được gọi là các phương thức. Các đối tượng cùng loại được xem như là có cùng kiểu hoặc, được xem như là trong cùng lớp.

Ví dụ, xét đối tượng Rectangle. Nó có các thuộc tính như length và width. Phụ thuộc vào thiết kế trên. Nó có thể cần những cách để cấp nhận các giá trị của thuộc tính này, tính toán diện tích và hiển thị chi tiết.

Chúng ta cùng theo dõi ví dụ sau triển khai một lớp Rectangle và cú pháp C# cơ bản của chương trình này. Để minh họa rõ ràng tính hướng đối tượng của C#, mình sẽ tạo hai class trong hai file riêng rẽ nhau.

Ví dụ lớp Rectangle: chứa các thuộc tính, phương thức chính

class Rectangle

{

// cac bien thanh vien

double \_length;

double \_width;

//phuong thuc

public void Acceptdetails()

{

\_length = 4.5;

\_width = 3.5;

}

//phuong thuc

public double GetArea()

{

return \_length \* \_width;

}

//phuong thuc

public void Display()

{

Console.WriteLine("Chieu dai: {0}", \_length);

Console.WriteLine("Chieu rong: {0}", \_width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

Lớp ExecuteRectangle: là lớp chứa phương thức main() để tiến hành thao tác trên đối tượng Rectangle của lớp Rectangle. Đến đây, có lẽ bạn chưa hiểu gì, nhưng không sao vì đây chỉ là chương giúp bạn làm quen với C# thôi.

using System;

namespace TPCenter

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Chuong trinh minh hoa tinh huong doi tuong ");

Console.WriteLine("--------------------------------------------------\n");

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//goi cac phuong thuc cua doi tuong nay

r.Acceptdetails();

r.Display();

Console.ReadLine();

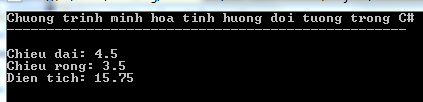
Console.ReadKey();

}

}

}

Nhấn phím **F5** để biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



**Từ khóa using**

Lệnh đầu tiên trong bất kỳ chương trình C# nào là:

using System;

Từ khóa using được sử dụng để bao namespace trong chương trình. Một chương trình C# có thể bao nhiều lệnh using.

**Từ khóa class**

Từ khóa class được sử dụng để khai báo một lớp .

**Comments**

Comment được sử dụng để khởi tạo code. Compiler bỏ qua các comment. Các comment đa dòng trong các chương trình C# bắt đầu với /\* và kết thúc với \*/ như sau:

/\* dong nay minh hoa comment nhieu dong .

Cu phap co ban C#

Ngon ngu lap trinh C# \*/

Comment đơn dòng được chỉ dẫn bởi ký hiệu '//'. Ví dụ:

// vi du comment don dong

**Biến thành viên**

Các biến là các thuộc tính hoặc thành viên dữ liệu của một lớp, được sử dụng để lưu giữ dữ liệu. Trong chương trình trước đó, lớp Rectangle có hai biến thành viên là length và width.

**Hàm thành viên**

Hàm là tập hợp các lệnh mà thực hiện một tác vụ cụ thể. Các hàm thành viên của một lớp được khai báo bên trong lớp đó. Lớp Rectangle chứa 3 hàm thành viên là: AcceptDetails, GetArea và Display.

**Thuyết minh một Class**

Trong chương trình trên, lớp ExecuteRectangle chứa phương thức Main() và khởi tạo lớp Rectangle.

**Định danh (Identifier)**

Một định danh là một tên được sử dụng để nhận diện một lớp, biến, hàm hoặc bất kỳ mục tự định nghĩa (user-defined).

Một tên phải bắt đầu với một chữ cái mà có thể được theo sau bởi một dãy các chữ cái, chữ số (0-9) hoặc dấu gạch dưới (\_). Ký tự đầu tiên của một định danh không thể là một chữ số.

Nó phải không chứa bất kỳ khoảng trống hoặc ký tự như ? - + ! @ # % ^ & \* ( ) [ ] { } . ; : " ' / và \. Tuy nhiên, dấu gạch dưới có thể được sử dụng.

Nó không nên là một từ khóa .

**Từ khóa**

Từ khóa là các từ dành riêng (Reserved Keyword) được định nghĩa trước cho C# compiler. Những từ khóa này không thể được sử dụng làm định danh. Tuy nhiên, nếu bạn muốn sử dụng các từ khóa này để làm định danh, bạn có thể đặt ký tự @ ở trước chúng.

, một số định danh có ý nghĩa đặc biệt trong ngữ cảnh của code, ví dụ như get và set được gọi là các contextual keyword (từ khóa thuộc ngữ cảnh).

|  |
| --- |
|  |
|  |
| **Reserved Keyword** | | | | | | |
| abstract | as | base | bool | break | byte | case |
| catch | char | checked | class | const | continue | decimal |
| default | delegate | do | double | else | enum | event |
| explicit | extern | false | finally | fixed | float | for |
| foreach | goto | if | implicit | in | in (generic modifier) | int |
| interface | internal | is | lock | long | namespace | new |
| null | object | operator | out | out (generic modifier) | override | params |
| private | protected | public | readonly | ref | return | sbyte |
| sealed | short | sizeof | stackalloc | static | string | struct |
| switch | this | throw | true | try | typeof | uint |
| ulong | unchecked | unsafe | ushort | using | virtual | void |
| volatile | while |  |  |  |  |  |
| **Contextual Keyword** | | | | | | |
| add | alias | ascending | descending | dynamic | from | get |
| global | group | into | join | let | orderby | partial (type) |
| partial (method) | remove | select | set |  |  |  |

Kiểu dữ liệu (Data Types)

Các biến được phân chia thành các kiểu sau:

* + Kiểu giá trị (Value type)
  + Kiểu tham chiếu (Reference type)
  + Kiểu con trỏ (Pointer type)

**Kiểu giá trị**

Các biến kiểu giá trị có thể được gán một giá trị một cách trực tiếp. Chúng được kế thừa từ lớp System.ValueType.

Kiểu giá trị trực tiếp chứa dữ liệu. Một số ví dụ là int, char, và float, tương ứng giữ số nguyên, chữ cái, và số thực. Khi bạn khai báo một kiểu int, hệ thống cấp phát bộ nhớ để lưu giá trị đó.

Bảng sau liệt kê các kiểu giá trị có sẵn 2010:

|  |  |  |  |
| --- | --- | --- | --- |
| **Kiểu** | **Biểu diễn** | **Dãy giá trị** | **Giá trị mặc định** |
| bool | Giá trị Boolean | True hoặc False | False |
| byte | Kiểu unsigned integer (8 bit) | 0 tới 255 | 0 |
| char | Kiểu Unicode character (16 bit) | U +0000 tới U +ffff | '\0' |
| decimal | Kiểu thập phân (128 bit) | (-7.9 x 1028 tới 7.9 x 1028) / 100 to 28 | 0.0M |
| double | Kiểu double (64 bit) | (+/-)5.0 x 10-324 tới (+/-)1.7 x 10308 | 0.0D |
| float | Kiểu float (32 bit) | -3.4 x 1038 tới + 3.4 x 1038 | 0.0F |
| int | Kiểu integer (32 bit) | -2,147,483,648 tới 2,147,483,647 | 0 |
| long | Kiểu signed integer (64 bit) | -9,223,372,036,854,775,808 tới 9,223,372,036,854,775,807 | 0L |
| sbyte | Kiểu signed integer (8 bit) | -128 tới 127 | 0 |
| short | Kiểu signed integer (16 bit) | -32,768 tới 32,767 | 0 |
| uint | Kiểu unsigned integer (32 bit) | 0 tới 4,294,967,295 | 0 |
| ulong | Kiểu unsigned integer (64 bit) | 0 tới 18,446,744,073,709,551,615 | 0 |
| ushort | Kiểu unsigned integer (16 bit) | 0 tới 65,535 | 0 |

**Từ khóa sizeof**

Để lấy kích cỡ chính xác của một kiểu hoặc một biến trên một nền tảng cụ thể, bạn có thể sử dụng phương thức **sizeof**. Biểu thức *sizeof(type)* hiển thị kích cỡ của đối tượng hoặc kiểu bằng giá trị byte. Ví dụ dưới đây để lấy kích cỡ của kiểu **int** trên bất kỳ máy tính:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Tu khoa sizeof ");

Console.WriteLine("-------------------------");

Console.WriteLine("Kich co cua kieu du lieu int la: {0}", sizeof(int));

Console.WriteLine("Kich co cua kieu du lieu float la: {0}", sizeof(float));

Console.WriteLine("Kich co cua kieu du lieu double la: {0}", sizeof(double));

Console.WriteLine("Kich co cua kieu du lieu char la: {0}", sizeof(char));

Console.ReadLine();

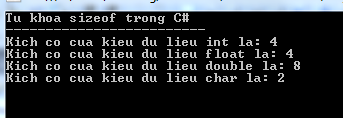
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Kiểu tham chiếu

Kiểu tham chiếu không chứa dữ liệu thực sự được lưu giữ trong một biến, nhưng chúng chứa một tham chiếu tới các biến.

Nói cách khác, chúng tham chiếu tới một vị trí bộ nhớ. Việc sử dụng nhiều biến, thì kiểu tham chiếu có thể tham chiếu tới tới một vị trí bộ nhớ. Nếu dữ liệu trong vị trí bộ nhớ bị thay đổi bởi một trong số các biến, thì biến khác tự động phản ánh sự thay đổi về giá trị này. Ví dụ các kiểu tham chiếu **có sẵn**  là: **object**, **dynamic,** và **string**.

Kiểu object

Kiểu **object** là lớp cơ sở cơ bản cho tất cả kiểu dữ liệu Common Type System (CTS). Object là một alias cho lớp System.Object. Các kiểu object có thể được gán giá trị của bất kỳ kiểu, kiểu giá trị, kiểu tham chiếu, kiểu tự định nghĩa (user-defined) khác. Tuy nhiên, trước khi gán các giá trị, nó cần một sự chuyển đổi kiểu.

Khi một kiểu giá trị được chuyển đổi thành kiểu object, nó được gọi là **boxing** và ngược lại, khi một kiểu object được chuyển đổi thành kiểu giá trị, nó được gọi là **unboxing**.

object obj;

obj = 100; // day la vi du boxing

Kiểu Dynamic

Bạn có thể lưu giữ bất kỳ kiểu giá trị nào trong biến kiểu dữ liệu dynamic. Việc kiểm tra các kiểu biến này diễn ra tại run time.

Cú pháp để khai báo một kiểu dynamic là:

dynamic <tên\_biến> = giá\_trị;

Ví dụ

dynamic d = 20;

Kiểu dynamic là tương tự với kiểu object, ngoại trừ việc kiểm tra cho các biến kiểu object diễn ra tại compile time, trong khi việc kiểm tra các biến kiểu dynamic diễn ra tại run time.

Kiểu string

Kiểu **string**  cho phép bạn gán bất kỳ giá trị chuỗi nào cho một biến. Kiểu string là một *alias* cho lớp **System.String**. Nó kế thừa từ kiểu object. Giá trị cho một kiểu string có thể được gán bởi sử dụng các hằng chuỗi trong hai mẫu: quoted và @quoted.

Ví dụ:

String str = "Hoc C# co ban va nang cao tai VietJack";

Và một hằng chuỗi @quoted trông như sau:

@"VietJack Team";

Các kiểu tự định nghĩa (user-defined) là: Class, Interface, hoặc Delegate. Chúng ta sẽ bàn về các kiểu này trong các chương sau.

## Kiểu con trỏ

Các biến kiểu con trỏ lưu giữ địa chỉ bộ nhớ của kiểu khác. Các con trỏ có khả năng như con trỏ trong C hoặc C++.

Cú pháp để khai báo một kiểu con trỏ là:

type\* identifier;

Ví dụ:

char\* cptr;

int\* iptr;

Chúng ta sẽ thảo luận về kiểu con trỏ ở chương: Unsafe Code .

**Chuyển đổi kiểu dữ liệu (Type Conversion)**

Chuyển đổi kiểu dữ liệu là biến đổi một kiểu dữ liệu này thành kiểu dữ liệu khác. Nó còn được gọi là Ép kiểu. , ép kiểu có hai mẫu sau:

* Chuyển đổi kiểu ngầm định (implicit) - Việc chuyển đổi này được thực hiện bởi C# theo một phương thức an toàn kiểu (type-safe). Ví dụ: việc chuyển đổi từ các lớp kế thừa thành các lớp cơ sở.
* Chuyển đổi kiểu tường minh (explicit) - Việc chuyển đổi này được thực hiện một cách rõ ràng bởi người dùng bằng việc sử dụng các hàm được định nghĩa trước. Các chuyển đổi kiểu tường minh cần một toán tử cast.

Ví dụ sau minh họa một sự chuyển đổi kiểu tường minh :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Chuyen doi kieu du lieu ");

Console.WriteLine("-------------------------------");

double d = 5678.74;

int i;

// cast kieu du lieu double thanh kieu du lieu int.

i = (int)d;

Console.WriteLine("Gia tri cua i = " +i);

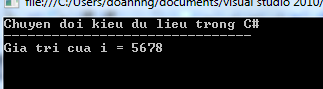
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Phương thức Chuyển đổi kiểu dữ liệu

C# cung cấp các phương thức chuyển đổi kiểu có sẵn được liệt kê trong bảng sau:

|  |  |
| --- | --- |
| **STT** | **Phương thức & Miêu tả** |
| 1 | **ToBoolean**  Chuyển đổi một kiểu thành một giá trị Boolean, nếu có thể |
| 2 | **ToByte**  Chuyển đổi một kiểu thành một byte |
| 3 | **ToChar**  Chuyển đổi một kiểu thành một Unicode character, nếu có thể |
| 4 | **ToDateTime**  Chuyển đổi một kiểu (kiểu integer hoặc string) thành các cấu trúc date-time |
| 5 | **ToDecimal**  Chuyển đổi một kiểu số thực hoặc số nguyên thành một kiểu thập phân |
| 6 | **ToDouble**  Chuyển đổi một kiểu thành một kiểu double |
| 7 | **ToInt16**  Chuyển đổi một kiểu thành một 16-bit integer |
| 8 | **ToInt32**  Chuyển đổi một kiểu thành một 32-bit integer |
| 9 | **ToInt64**  Chuyển đổi một kiểu thành một 64-bit integer |
| 10 | **ToSbyte**  Chuyển đổi một kiểu thành một kiểu signed byte |
| 11 | **ToSingle**  Chuyển đổi một kiểu thành một số small floating point |
| 12 | **ToString**  Chuyển đổi một kiểu thành một string |
| 13 | **ToType**  Chuyển đổi một kiểu thành một kiểu đã xác định |
| 14 | **ToUInt16**  Chuyển đổi một kiểu thành một kiểu unsigned int |
| 15 | **ToUInt32**  Chuyển đổi một kiểu thành một kiểu unsigned long |
| 16 | **ToUInt64**  Chuyển đổi một kiểu thành một unsigned big integer |

Ví dụ sau minh họa cách chuyển đổi các kiểu dữ liệu đa dạng thành kiểu dữ liệu string :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Chuyen doi kieu du lieu ");

Console.WriteLine("-------------------------------");

int i = 75;

float f = 53.005f;

double d = 2345.7652;

bool b = true;

//su dung phuong thuc ToString()

Console.WriteLine(i.ToString());

Console.WriteLine(f.ToString());

Console.WriteLine(d.ToString());

Console.WriteLine(b.ToString());

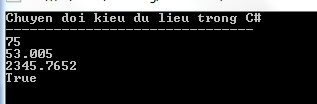
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Biến (Variables)

Một biến là không gì khác ngoài một tên được cung cấp cho khu vực lưu giữ mà chương trình có thể thao tác. Mỗi biến có một kiểu cụ thể, mà quyết định kích cỡ và cách bố trí bộ nhớ của biến đó, cách dãy giá trị có thể được lưu giữ trong bộ nhớ đó, và cách tập hợp các hoạt động có thể được áp dụng tới biến đó.

Các kiểu giá trị cơ bản có thể được phân chia thành:

|  |  |
| --- | --- |
| **Kiểu dữ liệu** | **Ví dụ** |
| Kiểu số nguyên | sbyte, byte, short, ushort, int, uint, long, ulong, và char |
| Kiểu số thực dấu chấm động | float và double |
| Kiểu thập phân | decimal |
| Kiểu Boolean | true hoặc false |
| Kiểu Nullable | Kiểu dữ liệu Nullable |

C# cũng cho phép định nghĩa các kiểu giá trị khác của biến như kiểu **enum** và các kiểu tham chiếu của biến như **class**, sẽ được bàn luận trong các chương tới.

Định nghĩa biến

Cú pháp để định nghĩa biến là:

<kiểu\_dữ\_liệu> <danh\_sách\_biến>;

Ở đây, **kiểu\_dữ\_liệu** phải là một kiểu dữ liệu hợp lệ , gồm: char, int, double hoặc bất kỳ kiểu dữ liệu tự định nghĩa (user-defined) nào và **danh\_sách\_biến** có thể chứa một hoặc nhiều tên định danh được phân biệt bởi dấu phảy.

Dưới đây là một số định nghĩa biến hợp lệ :

int i, j, k;

char c, ch;

float f, salary;

double d;

Bạn có thể khởi tạo một biến tại thời điểm định nghĩa, như sau:

int i = 100;

**Khởi tạo biến**

Biến được khởi tạo (được gán một giá trị) với một dấu bằng được theo sau bởi một biểu thức hằng. Form chung cho khởi tạo biến là:

tên\_biến = giá\_trị;

Các biến có thể được khởi tạo trong khai báo của chúng. Phần khởi tạo gồm một dấu bằng được theo sau bởi một biểu thức hằng, như sau:

<kiểu\_dữ\_liệu> <tên\_biến> = giá\_trị;

Một số ví dụ về khởi tạo biến là:

int d = 3, f = 5; /\* khai báo và khởi tạo biến d và f. \*/

byte z = 22; /\* khai báo và khởi tạo biến z. \*/

double pi = 3.14159; /\* khai báo và khởi tạo biến pi \*/

char x = 'x'; /\* khai báo và khởi tạo biến ký tự x. \*/

Việc khởi tạo biến một cách chính xác là một bài thực hành tốt cho bạn, nếu không thì chương trình có thể tạo ra kết quả không mong đợi.

Ví dụ sau sử dụng các kiểu biến đa dạng :

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Khai bao va khoi tao bien ");

Console.WriteLine("----------------------------------");

//khai bao bien

short a;

int b;

double c;

/\* khoi tao bien \*/

a = 10;

b = 20;

c = a + b;

Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);

Console.ReadLine();

Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:

Biến trong C#

**Nhận giá trị từ người dùng**

Lớp **Console** trong **System** namespace cung cấp hàm **ReadLine()** để nhận đầu vào từ người dùng (chẳng hạn nhập từ bàn phím) và lưu nó vào trong một biến.

Ví dụ:

int num;

num = Convert.ToInt32(Console.ReadLine());

Hàm **Convert.ToInt32()** chuyển đổi dữ liệu đã nhập bởi người dùng thành kiểu dữ liệu int, bởi vì **Console.ReadLine()** chấp nhận dữ liệu trong định dạng chuỗi.

**Biểu thức lvalue và rvalue**

Có hai loại biểu thức :

* **lvalue:** Một biểu thức mà là một lvalue có thể xuất hiện hoặc bên trái hoặc bên phải của phép gán .
* **rvalue:** Một biểu thức mà là một rvalue có thể xuất hiện bên phải nhưng không thể ở bên trái của một phép gán .

Các biến là lvalue và vì thế chúng có thể xuất hiện ở bên trái của một phép gán. Các hằng số là rvalue và vì thế chúng không thể được gán và không thể xuất hiện ở bên trái của một phép gán. Sau đây là một lệnh hợp lệ :

int g = 20;

Nhưng lệnh sau là không hợp lệ và sẽ tạo một Compile-time error :

10 = 20;

Hằng (Constant/Literal)

Constant liên quan tới các giá trị cố định mà chương trình không thể thay đổi trong khi thực thi. Những giá trị cố định này cũng được gọi là literal.

Constant là một kiểu dữ liệu thay thế cho Literal, còn Literal thể hiện chính nó. Trong ví dụ: const PI = 3.14 thì Constant ở đây là PI, còn Literal là 3.14.

Constant có thể là bất kỳ kiểu dữ liệu cơ bản nào , và có thể được phân chia thành giá trị hằng số nguyên, hằng số thực, hằng ký tự, hằng chuỗi và Boolean literal (tạm dịch: hằng logic). Ngoài ra, C# cũng có các kiểu hằng liệt kê.

Ngoài ra, constant được đối xử giống như các biến thông thường, ngoại trừ việc giá trị của chúng là không thể thay đổi sau khi định nghĩa.

**Hằng số nguyên**

**Hằng số nguyên** có thể là decimal (cơ số 10), octal (cơ số 8) hay hexadecimal (cơ số 16). Giá trị có tiền tố (prefix) là 0 cho octal, là 0x hay 0X cho hexadecimal và không có gì cho decimal.

Một hằng số nguyên cũng có các hậu tố (suffix) U hay L thể hiện kiểu unsigned hay long. Hậu tố có thể là chữ hoa hoặc chữ thường và có thể trong bất kỳ thứ tự nào.

Ví dụ về các hằng số nguyên:

212 /\* hợp lệ \*/

215u /\* hợp lệ \*/

0xFeeL /\* hợp lệ \*/

078 /\* không hợp lệ: 8 không phải là chữ số trong hệ cơ số 8 \*/

032UU /\* không hợp lệ: không thể lặp lại hậu tố \*/

Dưới đây là các kiểu hằng số nguyên đa dạng:

85 /\* thập phân \*/

0213 /\* hệ cơ số 8 \*/

0x4b /\* hệ cơ số 16 \*/

30 /\* int \*/

30u /\* unsigned int \*/

30l /\* long \*/

30ul /\* unsigned long \*/

**Hằng số thực**

Một **hằng số thực** bao gồm phần nguyên (integer part), dấu chấm thập phân (decimal point), phần lẻ (fraction part) và phần mũ (exponent part). Chúng ta có thể biểu diễn hằng số thực theo dạng decimal hay dạng mũ.

Xét các ví dụ sau:

3.14159 /\* hợp lệ \*/

314159E-5L /\* hợp lệ \*/

510E /\* không hợp lệ: phần exponent chưa hoàn thành \*/

210f /\* không hợp lệ: thiếu phần decimal hoặc exponent \*/

.e55 /\* không hợp lệ: thiếu phần integer hoặc fraction \*/

Khi thể hiện dạng decimal phải bao gồm dấu chấm thập phân, phần mũ hoặc cả hai. Khi thể hiện dạng mũ phải gồm phần nguyên, phần lẻ hoặc cả hai. Dạng mũ đi kèm với kí tự E hoặc e.

**Hằng ký tự**

Các **hằng kí tự**  mở đầu và kết thúc bởi dấu nháy đơn. Nếu hằng ký tự bắt đầu với L (ví dụ L'x') thì nó là kiểu wchar\_t. Nếu không thì, nó là hằng ký tự kiểu char, ví dụ như 'x'.

Hằng kí tự có thể là một kí tự (như 'X'), một escape sequence (như '\t') hay một kí tự mở rộng (như '\u02c0′).

Một số kí tự khi được đứng trước bởi dấu \ thì chúng sẽ mang một ý nghĩa đặc biệt như bắt đầu dòng mới '\n' hay tạo một tab '\t'. Chúng được biết như là escape sequence (dãy thoát). Bảng dưới đây thể hiện một số mã escape sequence phổ biến:

|  |  |
| --- | --- |
| **Dãy thoát** | **Ý nghĩa** |
| \\ | Ký tự \ |
| \' | Ký tự ' |
| \" | Ký tự " |
| \? | Ký tự ? |
| \a | Tiếng chuông |
| \b | Backspace |
| \f | Form feed |
| \n | Dòng mới |
| \r | Carriage return |
| \t | tab ngang |
| \v | tab dọc |
| \ooo | Số trong cơ số 8 của 1 đến 3 chữ số |
| \xhh . . . | Số thập lục phân của một hoặc nhiều chữ số |

Sau đây là ví dụ để chỉ một số ký tự dãy thoát:

using System;

namespace VietJackCsharp

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vi du ky tu tab va newline (dòng mới) ");

Console.WriteLine("--------------------------");

Console.WriteLine("Hello\tWorld\n\n");

Console.ReadLine();

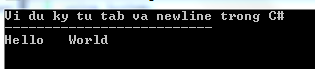
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



**Hằng chuỗi**

**Hằng chuỗi** chứa trong dấu nháy kép, ví dụ "abc". Một chuỗi sẽ chứa các kí tự tương tự hằng kí tự, gồm các ký tự thuần, escape sequence, và các ký tự mở rộng.

Có thể ngắt một dòng dài thành nhiều dòng bởi sử dụng hằng chuỗi và phân biệt chúng bởi sử dụng khoảng trắng (whitespace).

Xét ví dụ một hằng chuỗi thể hiện theo 3 cách khác nhau:

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

@"hello dear"

**Định nghĩa hằng**

Hằng được định nghĩa bởi sử dụng từ khóa **const**. Cú pháp để định nghĩa một hằng là:

const <kiểu\_dữ\_liệu> <tên\_hằng> = giá\_trị;

Chương trình sau minh họa cách định nghĩa và sử dụng một hằng trong chương trình C# của bạn:

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Hang so ");

Console.WriteLine("-------------------");

//khai bao hang so pi

const double pi = 3.14159;

double r;

Console.Write("Nhap ban kinh: ");

r = Convert.ToDouble(Console.ReadLine());

double dien\_tich = pi \* r \* r;

Console.WriteLine("\nBan kinh: {0}, Dien tich: {1}", r, dien\_tich);

Console.ReadLine();

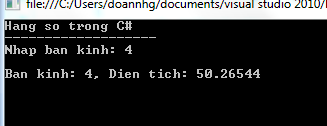
Console.ReadKey();

}

}

}

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Toán tử (Operators)

Một toán tử là một biểu tượng, mà nói cho compiler thực hiện các thao tác toán học và logic cụ thể. C# cung cấp nhiều toán tử có sẵn, đó là:

* Toán tử số học
* Toán tử quan hệ
* Toán tử logic
* Toán tử so sánh bit
* Toán tử gán
* Toán tử hỗn hợp

Toán tử số học trong C#

Bảng dưới liệt kê các toán tử số học được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A giữ giá trị 10, biến B giữ 20 thì:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_so_hoc_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| + | Thêm hai toán hạng | A + B sẽ cho kết quả là 30 |
| - | Trừ giá trị toán hạng hai từ toán hạng đầu | A - B sẽ cho kết quả là -10 |
| \* | Nhân hai toán hạng | A \* B sẽ cho kết quả là 200 |
| / | Chia lấy phần nguyên hai toán hạng | B / A sẽ cho kết quả là 2 |
| % | Chia lấy phần dư | B % A sẽ cho kết quả là 0 |
| ++ | Lượng gia giá trị toán hạng thêm 1 đơn vị | A++ sẽ cho kết quả là 11 |
| -- | Lượng giảm giá trị toán hạng một đơn vị | A-- sẽ cho kết quả là 9 |

Toán tử quan hệ trong C#

Bảng dưới đây liệt kê các toán tử quan hệ được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A giữ giá trị 10, biến B giữ 20 thì:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_quan_he_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| == | Kiểm tra nếu 2 toán hạng bằng nhau hay không. Nếu bằng thì điều kiện là true. | (A == B) là không đúng. |
| != | Kiểm tra 2 toán hạng có giá trị khác nhau hay không. Nếu không bằng thì điều kiện là true. | (A != B) là true. |
| > | Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn toán hạng bên phải hay không. Nếu lớn hơn thì điều kiện là true. | (A > B) là không đúng. |
| < | Kiểm tra nếu toán hạng bên trái nhỏ hơn toán hạng bên phải hay không. Nếu nhỏ hơn thì là true. | (A < B) là true. |
| >= | Kiểm tra nếu toán hạng bên trái có giá trị lớn hơn hoặc bằng giá trị của toán hạng bên phải hay không. Nếu đúng là true. | (A >= B) là không đúng. |
| <= | Kiểm tra nếu toán hạng bên trái có giá trị nhỏ hơn hoặc bằng toán hạng bên phải hay không. Nếu đúng là true. | (A <= B) là true. |

Toán tử logic trong C#

Bảng dưới đây chỉ rõ tất cả các toán tử logic được hỗ trợ bởi ngôn ngữ C#. Giả sử biến A có giá trị 1 và biến B có giá trị 0:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_logic_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| && | Được gọi là toán tử logic AND (và). Nếu cả hai toán tử đều có giá trị khác 0 thì điều kiện trở lên true. | (A && B) là false. |
| || | Được gọi là toán tử logic OR (hoặc). Nếu một trong hai toán tử khác 0, thì điều kiện là true. | (A || B) là true. |
| ! | Được gọi là toán tử NOT (phủ định). Sử dụng để đảo ngược lại trạng thái logic của toán hạng đó. Nếu điều kiện toán hạng là true thì phủ định nó sẽ là false. | !(A && B) là true. |

Toán tử so sánh bit trong C#

Toán tử so sánh bit làm việc trên đơn vị bit, tính toán biểu thức so sánh từng bit. Bảng dưới đây về &, |, và ^ như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Giả sử nếu A = 60; và B = 13; thì bây giờ trong định dạng nhị phân chúng sẽ là như sau:

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

Các toán tử so sánh bit được hỗ trợ bởi ngôn ngữ C# được liệt kê trong bảng dưới đây. Giá sử ta có biến A có giá tri 60 và biến B có giá trị 13, ta có:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_so_sanh_bit_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| & | Toán tử AND (và) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong cả hai toán hạng. | (A & B) sẽ cho kết quả là 12, tức là 0000 1100 |
| | | Toán tử OR (hoặc) nhị phân sao chép một bit tới kết quả nếu nó tồn tại trong một hoặc hai toán hạng. | (A | B) sẽ cho kết quả là 61, tức là 0011 1101 |
| ^ | Toán tử XOR nhị phân sao chép bit mà nó chỉ tồn tại trong một toán hạng mà không phải cả hai. | (A ^ B) sẽ cho kết quả là 49, tức là 0011 0001 |
| ~ | Toán tử đảo bit (đảo bit 1 thành bit 0 và ngược lại). | (~A ) sẽ cho kết quả là -61, tức là 1100 0011. |
| << | Toán tử dịch trái. Giá trị toán hạng trái được dịch chuyển sang trái bởi số các bit được xác định bởi toán hạng bên phải. | A << 2 sẽ cho kết quả 240, tức là 1111 0000 (dịch sang trái hai bit) |
| >> | Toán tử dịch phải. Giá trị toán hạng trái được dịch chuyển sang phải bởi số các bit được xác định bởi toán hạng bên phải. | A >> 2 sẽ cho kết quả là 15, tức là 0000 1111 (dịch sang phải hai bit) |

**Toán tử gán trong C#**

Đây là những toán tử gán được hỗ trợ bởi ngôn ngữ C#:

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_gan_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| = | Toán tử gán đơn giản. Gán giá trị toán hạng bên phải cho toán hạng trái. | C = A + B sẽ gán giá trị của A + B vào trong C |
| += | Thêm giá trị toán hạng phải tới toán hạng trái và gán giá trị đó cho toán hạng trái. | C += A tương đương với C = C + A |
| -= | Trừ đi giá trị toán hạng phải từ toán hạng trái và gán giá trị này cho toán hạng trái. | C -= A tương đương với C = C - A |
| \*= | Nhân giá trị toán hạng phải với toán hạng trái và gán giá trị này cho toán hạng trái. | C \*= A tương đương với C = C \* A |
| /= | Chia toán hạng trái cho toán hạng phải và gán giá trị này cho toán hạng trái. | C /= A tương đương với C = C / A |
| %= | Lấy phần dư của phép chia toán hạng trái cho toán hạng phải và gán cho toán hạng trái. | C %= A tương đương với C = C % A |
| <<= | Dịch trái toán hạng trái sang số vị trí là giá trị toán hạng phải. | C <<= 2 tương đương với C = C << 2 |
| >>= | Dịch phải toán hạng trái sang số vị trí là giá trị toán hạng phải. | C >>= 2 tương đương với C = C >> 2 |
| &= | Phép AND bit | C &= 2 tương đương với C = C & 2 |
| ^= | Phép OR loại trừ bit | C ^= 2 tương đương với C = C ^ 2 |
| |= | Phép OR bit. | C |= 2 tương đương với C = C | 2 |

**Các toán tử hỗn hợp trong C#**

Dưới đây là một số toán tử hỗn hợp quan trọng gồm **sizeof, typeof** và **? :** được hỗ trợ bởi ngôn ngữ C#.

[**Ví dụ**](http://vietjack.com/csharp/toan_tu_hon_hop_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Toán tử** | **Miêu tả** | **Ví dụ** |
| sizeof() | Trả về kích cỡ của một kiểu dữ liệu | sizeof(int), trả về 4 |
| typeof() | Trả về kiểu của một lớp | typeof(StreamReader); |
| & | Trả về địa chỉ của một biến | &a; trả về địa chỉ thực sự của biến |
| \* | Trỏ tới một biến | \*a; tạo con trỏ với tên là a tới một biến |
| ? : | Biểu thức điều kiện (Conditional Expression) | Nếu Condition là true ? Thì giá trị X : Nếu không thì Y |
| is | Xác định đối tượng là một kiểu cụ thể hay không | If( Ford is Car) // Kiểm tra nếu Ford là một đối tượng của lớp Car |
| as | Ép kiểu mà không tạo một exception nếu việc ép kiểu thất bại | Object obj = new StringReader("Hello");  StringReader r = obj as StringReader; |

**Thứ tự ưu tiên toán tử trong C#**

Thứ tự ưu tiên toán tử trong C# xác định cách biểu thức được tính toán. Ví dụ, toán tử nhân có quyền ưu tiên hơn toán tử cộng, và nó được thực hiện trước.

Ví dụ, x = 7 + 3 \* 2; ở đây, x được gán giá trị 13, chứ không phải 20 bởi vì toán tử \* có quyền ưu tiên cao hơn toán tử +, vì thế đầu tiên nó thực hiện phép nhân 3 \* 2 và sau đó thêm với 7.

Bảng dưới đây liệt kê thứ tự ưu tiên của các toán tử. Các toán tử với quyền ưu tiên cao nhất xuất hiện trên cùng của bảng, và các toán tử có quyền ưu tiên thấp nhất thì ở bên dưới cùng của bảng. Trong một biểu thức, các toán tử có quyền ưu tiên cao nhất được tính toán đầu tiên.

[**Ví dụ**](http://vietjack.com/csharp/thu_tu_uu_tien_toan_tu_trong_csharp.jsp)

|  |  |  |
| --- | --- | --- |
| **Loại** | **Toán tử** | **Thứ tự ưu tiên** |
| Postfix | () [] -> . ++ - - | Trái sang phải |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Phải sang trái |
| Tính nhân | \* / % | Trái sang phải |
| Tính cộng | + - | Trái sang phải |
| Dịch chuyển | << >> | Trái sang phải |
| Quan hệ | < <= > >= | Trái sang phải |
| Cân bằng | == != | Trái sang phải |
| Phép AND bit | & | Trái sang phải |
| Phép XOR bit | ^ | Trái sang phải |
| Phép OR bit | | | Trái sang phải |
| Phép AND logic | && | Trái sang phải |
| Phép OR logic | || | Trái sang phải |
| Điều kiện | ?: | Phải sang trái |
| Gán | = += -= \*= /= %=>>= <<= &= ^= |= | Phải sang trái |
| Dấu phảy | , | Trái sang phải |

Lệnh If, Else, Switch (Decision Making)

Các cấu trúc điều khiển luồng yêu cầu lập trình viên xác định một hoặc nhiều điều kiện để được đánh giá và kiểm tra bởi chương trình, cùng với các lệnh được thực hiện nếu điều kiện được xác định là đúng, hoặc các lệnh khác được thực hiện nếu điều kiện xác định là sai.

Dưới đây là mẫu chung của một cấu trúc điều khiển luồng hay gặp trong ngôn ngữ lập trình.



C# cung cấp các loại lệnh điều khiển luồng sau. Bạn nhấn vào link để thấy chi tiết.

|  |  |
| --- | --- |
| **Lệnh** | **Miêu tả** |
| [**Lệnh if trong C#**](http://vietjack.com/csharp/lenh_if_trong_csharp.jsp) | Một lệnh **if** bao gồm một biểu thức logic theo sau bởi một hoặc nhiều lệnh khác. |
| [**Lệnh if...else trong C#**](http://vietjack.com/csharp/lenh_if_else_trong_csharp.jsp) | Một lệnh **if** có thể theo sau bởi một lệnh **else** (tùy ý: có hoặc không), mà có thể được thực hiện khi biểu thức logic có giá trị false. |
| [**Lồng các lệnh if trong C#**](http://vietjack.com/csharp/long_cac_lenh_if_trong_csharp.jsp) | Bạn có thể sử dụng lệnh **if** hoặc lệnh **else if** bên trong lệnh **if** hoặc **else if** khác |
| [**Lệnh switch trong C#**](http://vietjack.com/csharp/lenh_switch_trong_csharp.jsp) | Lệnh **switch** cho phép kiểm tra điều kiện của một biến trước khi thực thi các lệnh |
| [**Lồng các lệnh switch trong C#**](http://vietjack.com/csharp/long_cac_lenh_switch_trong_csharp.jsp) | Bạn có thể sử dụng một lệnh **switch**bên trong một lệnh **switch**khác |

**Toán tử ? : trong C#**

Chúng ta đã bàn về **toán tử điều kiện ? :** trong chương trước mà có thể được dùng để thay thế cho lệnh **if...else**. Nó có mẫu chung như sau:

Exp1 ? Exp2 : Exp3;

Trong đó Exp1, Exp2 và Exp3 là các biểu thức. Chú ý việc sử dụng và đặt của dấu hai chấm.

Giá trị của biểu thức Exp1 trước dấu ? có giá trị **true**, Exp2 được thực hiện, và giá trị của nó là giá trị của biểu thức. Nếu Exp1 là **false** thì Exp3 được thực hiện và giá trị của nó là giá trị của biểu thức.

Vòng lặp (Loop)

Có một tình huống mà bạn cần phải thực hiện một đoạn code một vài lần. Nhìn chung, các câu lệnh được thực hiện một cách tuần tự. Câu lệnh đầu tiên của hàm được thực hiện trước, sau đó đến câu thứ 2 và tiếp tục.

Ngôn ngữ lập trình cung cấp cho chúng ta nhiều cấu trúc điều khiển và cho phép bạn thực hiện những phần phức tạp.

Vòng lặp cho phép thực hiện một lệnh và một nhóm lệnh nhiều lần , dưới đây là dạng tổng quát:



C# hỗ trợ những lệnh điều khiển sau đây. Click chuột vào link để xem chi tiết.

|  |  |
| --- | --- |
| **Kiểu vòng lặp** | **Miêu tả** |
| [**Vòng lặp while trong C#**](http://vietjack.com/csharp/vong_lap_while_trong_csharp.jsp) | Lặp lại một hoặc một nhóm các lệnh trong khi điều kiện đã cho là đúng. Nó kiểm tra điều kiện trước khi thực hiện thân vòng lặp. |
| [**Vòng lặp for trong C#**](http://vietjack.com/csharp/vong_lap_for_trong_csharp.jsp) | Thực thi một dãy các lệnh nhiều lần và tóm tắt đoạn code mà quản lý biến vòng lặp. |
| [**Vòng lặp do...while trong C#**](http://vietjack.com/csharp/vong_lap_do_while_trong_csharp.jsp) | Giống lệnh while, ngoại trừ ở điểm là nó kiểm tra điều kiện ở cuối thân vòng lặp. |
| [**Lồng các vòng lặp trong C#**](http://vietjack.com/csharp/long_vong_lap_trong_csharp.jsp) | Bạn có thể sử dụng một hoặc nhiều vòng lặp trong các vòng lặp while, for hoặc do..while khác. |

**Các lệnh điều khiển vòng lặp trong C#**

Các lệnh điều khiển vòng lặp thay đổi sự thực thi lệnh từ dãy thông thường của nó. Khi sự thực thi lệnh rời khỏi một phạm vi, tất cả các đối tượng tự động mà được tạo ra trong phạm vi đó bị hủy.

C# hỗ trợ các lệnh điều khiển vòng lặp sau đây. Click vào các đường link sau để biết thêm chi tiết.

|  |  |
| --- | --- |
| **Lệnh điều khiển** | **Miêu tả** |
| [**Lệnh break trong C#**](http://vietjack.com/csharp/lenh_break_trong_csharp.jsp) | Kết thúc **vòng lặp** hoặc lệnh **switch** và chuyển sang thực thi vòng lặp hoặc lệnh switch ngay sau nó. |
| [**Lệnh continue trong C#**](http://vietjack.com/csharp/lenh_continue_trong_csharp.jsp) | Khi gặp lệnh này thì chương trình sẽ bỏ qua các câu lệnh ở dưới nó (trong cùng một câu lệnh lặp) để thực hiện vòng lặp mới. |

**Vòng lặp vô hạn trong C#**

Một vòng lặp là vòng lặp vô hạn khi một điều kiện không bao giờ false. Vòng lặp **for** thường được sử dụng cho mục đích này. Khi bạn để ba biểu thức điều kiện trong vòng lặp for trống thì bạn sẽ tạo ra một vòng lặp vô hạn.

### **Ví dụ**

using System;

namespace TPCenter

{

class TestCsharp

{

static void Main(string[] args)

{

Console.WriteLine("Vong lap vo han trong C#");

Console.WriteLine("-------------------------------");

for (; ; )

{

Console.WriteLine("Dong nay se duoc in mai mai");

}

}

}

}

Khi biểu thức điều kiện vắng mặt, nó được giả sử là luôn đúng. Bạn có thể có một biểu thức khởi tạo và biểu thức lượng gia, nhưng các lập trình viên C# thường sử dụng for(;;) để biểu thị một vòng lặp vô hạn.

Tính đóng gói (Encapsulation)

**Encapsulation (Tính đóng gói)** được định nghĩa là "tiến trình đóng gói một hoặc nhiều mục bên trong một gói logic hoặc vật lý". Tính đóng gói, trong phương pháp lập trình hướng đối tượng, ngăn cản việc truy cập tới chi tiết của trình trình triển khai (Implementation Detail).

Tính trừu tượng và tính đóng gói là hai đặc điểm có liên quan với nhau trong lập trình hướng đối tượng. Tính trừu tượng cho phép tạo các thông tin liên quan có thể nhìn thấy và tính đóng gói cho lập trình viên khả năng *triển khai độ trừu tượng đã được kế thừa*.

Tính đóng gói được triển khai bởi sử dụng **Access Specifier**. Một Access Specifier định nghĩa phạm vi và tính nhìn thấy của một thành viên lớp. C# hỗ trợ các Access Specifier sau:

* Public
* Private
* Protected
* Internal
* Protected internal

**Public Access Specifier trong C#**

Public Access Specifier trong C# cho phép một lớp trưng bày các biến thành viên và các hàm thành viên của nó tới các hàm và đối tượng khác. Bất kỳ thành viên public nào trong C# có thể được truy cập từ bên ngoài lớp đó.

Ví dụ sau minh họa Public Access Specifier trong C#:

Để minh họa rõ ràng tính đóng gói trong C#, mình tạo hai lớp có tên lần lượt là: **Rectangle** và **ExecuteRectangle**.

Lớp **Rectangle**: chứa các thuộc tính, phương thức

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

public double length;

public double width;

//cac phuong thuc

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**: chứa phương thức **main()** để thao tác trên đối tượng **Rectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi trong C#");

Console.WriteLine("-------------------------------");

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Display();

Console.ReadLine();

Console.ReadKey();

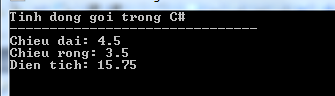
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Trong ví dụ, các biến thành viên length và width được khai báo là **public**, vì thế chúng có thể được truy cập từ hàm Main() bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

Hàm thành viên *Display()* và *GetArea()* cũng có thể truy cập các biến này một cách trực tiếp mà không cần sử dụng bất kỳ instance nào của lớp.

Hàm thành viên *Display()* cũng được khai báo là **public**, vì thế nó cũng có thể được truy cập từ hàm **Main()** bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

**Private Access Specifier trong C#**

Private Access Specifier trong C# cho phép một lớp ẩn các biến thành viên và các hàm thành viên của nó với các hàm và đối tượng khác. Chỉ có các hàm trong cùng lớp đó có thể truy cập tới các thành viên private. Ngay cả khi một Instance của một lớp cũng không thể truy cập các thành viên private của nó.

Ví dụ sau minh họa Private Access Specifier trong C#:

Lớp **Rectangle**

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

private double length;

private double width;

//cac phuong thuc

public void Acceptdetails()

{

Console.WriteLine("Nhap chieu dai: ");

length = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Nhap chieu rong: ");

width = Convert.ToDouble(Console.ReadLine());

}

public double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi trong C#");

Console.WriteLine("-------------------------------");

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Acceptdetails();

r.Display();

Console.ReadLine();

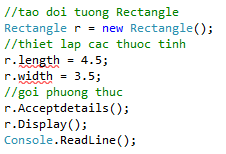
Console.ReadKey();

}

}

}

Trong ví dụ, các biến thành viên length và width được khai báo **private**, vì thế chúng không thể được truy cập từ hàm **Main()**. Trong Visual Studio 2010 sẽ báo hiệu một dấu gạch đỏ bên dưới hai biến này, giống như:



Các hàm thành viên *AcceptDetails()* và *Display()* có thể truy cập các biến này. Khi các hàm thành viên *AcceptDetails()* và *Display()* được khai báo **public**, chúng có thể được truy cập từ hàm **Main()** bởi sử dụng một Instance (một sự thể hiện) của lớp Rectangle, tên là **r**.

**Protected Access Specifier trong C#**

Protected Access Specifier trong C# cho phép một lớp con truy cập các biến thành viên và các hàm thành viên của lớp cơ sở của nó. Cách này giúp triển khai tính kế thừa. Chúng ta sẽ thảo luận chi tiết về tính kế thừa trong chương sau đó.

**Internal Access Specifier trong C#**

Internal Access Specifier trong C# cho phép một lớp trưng bày các biến thành viên và các hàm thành viên của nó tới các hàm và đối tượng khác trong *Assembly* hiện tại. Nói cách khác, bất kỳ thành viên nào với Internal Access Specifier trong C# có thể được truy cập từ bất kỳ lớp hoặc phương thức được định nghĩa bên trong ứng dụng mà thành viên đó được định nghĩa.

Ví dụ sau minh họa Internal Access Specifier trong C#:

Lớp **Rectangle**

using System;

namespace TPCenter

{

class Rectangle

{

//cac bien thanh vien

internal double length;

internal double width;

//cac phuong thuc

double GetArea()

{

return length \* width;

}

public void Display()

{

Console.WriteLine("Chieu dai: {0}", length);

Console.WriteLine("Chieu rong: {0}", width);

Console.WriteLine("Dien tich: {0}", GetArea());

}

}

}

Lớp **ExecuteRectangle**

using System;

namespace TPCenter

{

class ExecuteRectangle

{

static void Main(string[] args)

{

Console.WriteLine("Tinh dong goi trong C#");

Console.WriteLine("-------------------------------");

//tao doi tuong Rectangle

Rectangle r = new Rectangle();

//thiet lap cac thuoc tinh

r.length = 4.5;

r.width = 3.5;

//goi phuong thuc

r.Display();

Console.ReadLine();

Console.ReadKey();

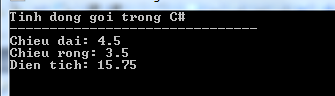
}

}

}

Nếu bạn không sử dụng lệnh **Console.ReadKey();** thì chương trình sẽ chạy và kết thúc luôn (nhanh quá đến nỗi bạn không kịp nhìn kết quả). Lệnh này cho phép chúng ta nhìn kết quả một cách rõ ràng hơn.

Biên dịch và chạy chương trình C# trên sẽ cho kết quả sau:



Trong ví dụ, bạn chú ý rằng hàm thành viên **GetArea()** không được khai báo với bất kỳ Access Specifier nào. Thì theo mặc định, Access Specifier của một thành viên lớp nếu chúng ta không khai báo là **private**.

**Protected Internal Access Specifier trong C#**

Protected Internal Access Specifier trong C# cho phép một lớp ẩn các biến thành viên và các hàm thành viên của nó với các hàm và đối tượng khác, ngoại trừ một lớp con bên trong cùng ứng dụng đó. Điều này cũng được sử dụng trong khi triển khai tính kế thừa trong C#.

Phương thức (Method)

Nullable

Mảng(Array)

Chuỗi(String)

Cấu trúc(Struct)

Enum

Lớp(Classes)

Tính kế thừa(Inheritance)

Tính đa hình(Polymorphism)

Nạp chồng toán tử(Operator Overloading)

Interface

Namespaces

Chỉ thị tiền xử lý(Preprocessor Directives)

Regular Expression

Exception Handling

File I/O

Attributes

Reflection

Properties

Indexer

Delegate

Event

Collection

Generic

Phương thức nặc danh(Anonymous Methods)

Unsafe Code

Multithreading

Tổng kết