

## Chương 1

# Ôn các kiến thức về cú pháp ngôn ngữ VC#

### 1.0 Dẫn nhập

### 1.1 Tổng quát về máy tính và ngôn ngữ VC#

### 1.2 Tập ký tự cơ bản của ngôn ngữ VC#

### 1.3 Extended Backus-Naur Form (EBNF) notation

### 1.4 Cú pháp định nghĩa tên nhận dạng (Name)

### 1.5 Cú pháp định nghĩa dấu ngăn (Seperator)

### 1.6 Cú pháp định nghĩa biểu thức

### 1.7 Quy trình tính biểu thức

### 1.8 Các lệnh định nghĩa các thành phần phần mềm

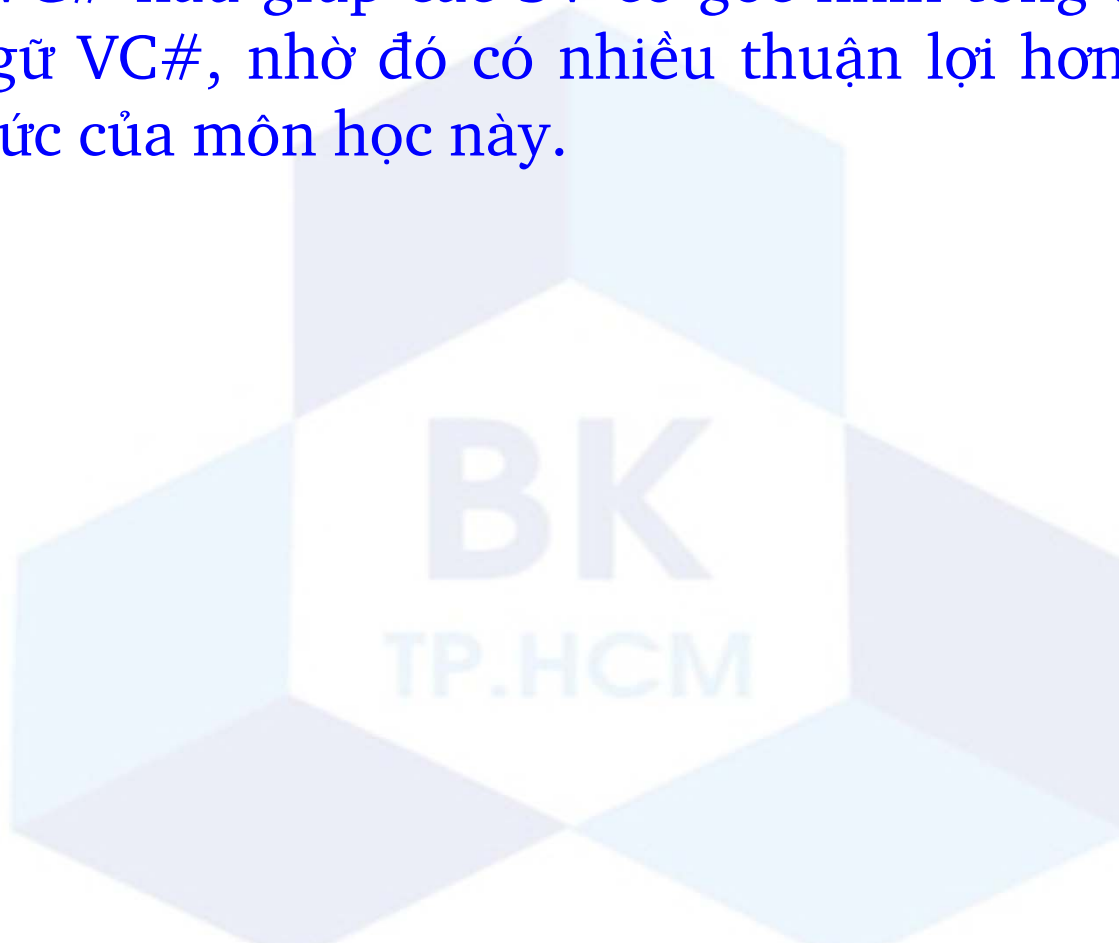
### 1.9 Các lệnh thực thi

### 1.10 Kết chương



# 1.0 Dẫn nhập

- ❑ Chương này sẽ tóm tắt lại 1 số kiến thức cơ bản về cú pháp của ngôn ngữ VC# hầu giúp các SV có góc nhìn tổng thể và hệ thống về ngôn ngữ VC#, nhờ đó có nhiều thuận lợi hơn trong việc học các kiến thức của môn học này.



## 1.1 Tổng quát về máy tính và ngôn ngữ VC#

- ❑ Máy tính số là thiết bị đặc biệt, nó là thiết bị tổng quát hóa, nghĩa là có thể thực hiện nhiều công việc khác nhau. Ta có thể nói máy tính số là thiết bị vạn năng.
- ❑ Vậy tại 1 thời điểm xác định, máy tính thực hiện công việc gì ? Nó không làm gì cả nếu con người không yêu cầu cụ thể nó.
- ❑ Làm sao để con người có thể yêu cầu máy tính thực hiện 1 công việc nào đó ? Ta phải viết chương trình giải quyết công việc tương ứng rồi đưa vào máy và nhờ máy chạy dùm.
- ❑ Viết chương trình là qui trình lớn và dài hạn gồm nhiều bước, trong đó các bước chính yếu là : xác định chính xác các chức năng của chương trình, phân tích cách giải quyết từng chức năng, tìm thuật giải chi tiết để giải quyết từng chức năng, đổi thuật giải chi tiết từ ngôn ngữ đời thường thành ngôn ngữ lập trình cho máy hiểu.



## 1.1 Tổng quát về máy tính và ngôn ngữ VC#

- ❑ Ngôn ngữ lập trình là ngôn ngữ giao tiếp giữa người và máy. Học ngôn ngữ lập trình cũng giống như học ngôn ngữ tự nhiên, nghĩa là học tuần tự các thành phần của ngôn ngữ từ thấp đến cao như :
  - Tập ký tự cơ bản
  - Cú pháp xây dựng từ (word). Từ được dùng để đặt tên nhận dạng cho từng phần tử cấu thành chương trình như hằng gọi nhớ, biến, hàm chức năng, class đối tượng,...
  - Cú pháp xây dựng biểu thức. Biểu thức (công thức toán học) miêu tả 1 quá trình tính toán tuần tự nhiều phép toán trên nhiều dữ liệu để tạo ra kết quả tính toán.
  - Cú pháp xây dựng từng câu lệnh : có 2 loại câu lệnh : lệnh định nghĩa và lệnh thực thi :



## 1.1 Tổng quát về máy tính và ngôn ngữ VC#

- Lệnh định nghĩa được dùng để định nghĩa và tạo mới phần tử cấu thành phần mềm. Thí dụ lệnh định nghĩa biến, định nghĩa hằng gọi nhớ, định nghĩa kiểu, định nghĩa hàm chức năng, ...
- Lệnh thực thi miêu tả 1 hành động cụ thể cần phải thực hiện. Thí dụ lệnh gán, lệnh kiểm tra điều kiện luận lý if, ...
- Cú pháp đặc tả 1 hàm chức năng
- Cú pháp đặc tả 1 class chức năng
- Cú pháp đặc tả 1 chương trình.



## 1.2 Tập ký tự cơ bản của ngôn ngữ VC#

- ❑ Ngôn ngữ VC# hiểu và dùng tập ký tự Unicode. Cụ thể trên Windows, mỗi ký tự Unicode dài 2 byte (16 bit) => có 65536 ký tự Unicode khác nhau trên Windows.
- ❑ Mặc dù vậy, VC# dùng chủ yếu các ký tự :
  - chữ tiếng Anh (a-zA-Z), '\_',
  - ký tự số (0-9),
  - khoảng trắng và các dấu ngăn như Tab (giống cột), CR (quay về đầu dòng), LF (xuống dòng).
  - các ký tự đặc biệt để miêu tả phép toán như +, -, \*, /, =, !, (, )
- ❑ Các ký tự khác, nhất là các ký tự có mã > 256 chỉ được dùng trong lệnh chú thích và chuỗi văn bản. Các ký tự có dấu tiếng Việt có mã từ 7840-7929.



## 1.3 Extended Backus-Naur Form (EBNF) notation

- Ta sẽ dùng qui ước EBNF để miêu tả cú pháp xây dựng các phần tử của ngôn ngữ VC#. Cụ thể ta sẽ dùng các qui ước EBNF sau đây :
  - $\#xN$ , trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự có mã thập lục phân tương ứng. Thí dụ ta viết  $\#x3e$  để miêu tả ký tự  $>$ .
  - $[a-zA-Z]$ ,  $[\#xN-\#xN]$ , trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự thuộc danh sách liệt kê các ký tự có mã liên tục. Thí dụ ta viết  $[0-9]$  để miêu tả 1 ký tự số thập phân từ 0 đến 9.
  - $[agn]$ ,  $[\wedge \#xN\#xN\#xN]$ , trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự thuộc danh sách liệt kê các ký tự rời rạc. Thí dụ ta viết  $[<@]$  để miêu tả 1 ký tự hoặc là  $<$  hay  $@$ .



## 1.3 Extended Backus-Naur Form (EBNF) notation

- $[\wedge a-zA-Z]$ ,  $[\wedge \#xN-\#xN]$ , trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự không thuộc danh sách liệt kê các ký tự có mã liên tục. Thí dụ ta viết  $[\wedge 0-9]$  để miêu tả 1 ký tự bất kỳ nhưng không phải là số thập phân từ 0 đến 9.
- $[\wedge agn]$ ,  $[\wedge \#xN\#xN\#xN]$ , trong đó N là chuỗi ký tự thập lục phân. Qui ước này miêu tả 1 ký tự không thuộc danh sách liệt kê các ký tự rời rạc. Thí dụ ta viết  $[\wedge <@]$  để miêu tả 1 ký tự bất kỳ nhưng không phải là < hay @.
- "string" hay 'string'. Qui ước này miêu tả chuỗi ký tự có nội dung nằm trong 2 dấu nháy kép hay nháy đơn. Thí dụ ta viết "DHBK" hoặc 'DHBK' để miêu tả chuỗi ký tự DHBK.





## 1.3 Extended Backus-Naur Form (EBNF) notation

- $A?$  miêu tả có từ 0 tới 1 lần A. Thí dụ  $S?$  miêu tả có từ 0 tới 1 phần tử S.
- $A^+$  miêu tả có từ 1 tới n lần A. Thí dụ  $S^+$  miêu tả có từ 1 tới n phần tử S.
- $A^*$  miêu tả có từ 0 tới n lần A. Thí dụ  $S^*$  miêu tả có từ 0 tới n phần tử S.
- $AB$  miêu tả phần tử A rồi tới phần tử B.
- $A \mid B$  miêu tả chọn lựa A hay B.
- $A - B$  miêu tả chuỗi thỏa A nhưng không thỏa B.
- **(expression)**. Qui ước này miêu tả kết quả của việc tính biểu thức. Thí dụ  $(\text{DefStatement} \mid \text{ExeStatement})$  để miêu tả sự tồn tại của phần tử DefStatement hay ExeStatement.
- $/* \dots */$  miêu tả chuỗi chú thích.



## 1.4 Cú pháp định nghĩa tên nhận dạng (Name)

- Mỗi phần tử trong chương trình đều được nhận dạng bởi 1 tên nhận dạng riêng biệt. Tên là 1 chuỗi từ 1 tới nhiều ký tự, ký tự đầu phải là ký tự chữ hay dấu `_`, các ký tự còn lại có thể là chữ, dấu `_` hay số. Độ dài maximum của tên là 255 ký tự. Cú pháp định nghĩa tên của VC# được viết theo EBNF bằng 3 luật sinh như sau :

`Name ::= NameStartChar (NameChar)*`

`NameStartChar ::= [a-zA-Z_]`

`NameChar ::= NameStartChar | [0-9]`

- Thí dụ `if`, `do`, `while`, `switch`, ... là những tên viết đúng cú pháp và được ngôn ngữ dùng để nhận dạng các lệnh thực thi. `intTuoi`, `dblX1`, `XuatKetqua`,... là những tên viết đúng cú pháp bởi người lập trình.



## 1.5 Cú pháp định nghĩa dấu ngăn (Seperator)

- Cú pháp miêu tả các phần tử lớn hơn thường có điểm chung là phần tử lớn gồm tuần tự nhiều phần tử nhỏ hợp lại theo 1 thứ tự xác định.
- Thường ta cần từ 1 tới n dấu ngăn nằm giữa các phần tử nhỏ kề nhau để ngăn chúng ra. Cú pháp miêu tả chuỗi từ 1 đến nhiều ký tự ngăn cách là :

$S ::= (\#x20 \mid \#x9 \mid \#xD \mid \#xA \mid \text{Comment}) +$

$\text{Comment} ::= \text{InLineComment} \mid \text{OutofLineComment}$

$\text{InLineComment} ::= "//" [\wedge \#xD\#xA]^*$

$\text{OutofLineComment} ::= "/*" (\text{Char}^* - (\text{Char}^* "*/" \text{Char}^*)) "*/"$

Thí dụ :

//đây là chú thích trên 1 dòng

/\* còn đây là

chú thích trên nhiều dòng \*/



## 1.6 Cú pháp định nghĩa biểu thức

- Ta đã biết trong toán học công thức là phương tiện miêu tả 1 qui trình tính toán nào đó trên các số.
- Trong VC# (hay ngôn ngữ lập trình khác), ta dùng biểu thức để miêu tả qui trình tính toán nào đó trên các dữ liệu  $\Rightarrow$  biểu thức cũng giống như công thức toán học, tuy nó tổng quát hơn (xử lý trên nhiều loại dữ liệu khác nhau) và phải tuân theo qui tắc cấu tạo chặt chẽ hơn công thức toán học.
- Để hiểu được biểu thức, ta cần hiểu được các thành phần của nó :
  - Các toán hạng : các biến, hằng dữ liệu,...
  - Các phép toán (toán tử) tham gia biểu thức :  $+$ ,  $-$ ,  $*$ ,  $/$ , ...
  - Qui tắc kết hợp toán tử và toán hạng để tạo biểu thức.
  - Qui trình mà máy dùng để tính trị của biểu thức.
  - Kiểu của biểu thức là kiểu của kết quả tính toán biểu thức.



## 1.6 Cú pháp định nghĩa biểu thức

- Biểu thức cơ bản (hay toán hạng) là phần tử nhỏ nhất cấu thành biểu thức bất kỳ. Một trong các phần tử sau được gọi là biểu thức cơ bản :
  - Biến, thuộc tính của đối tượng
  - Hằng gọi nhớ,
  - Giá trị dữ liệu cụ thể thuộc kiểu nào đó (nguyên, thực,...)
  - Lời gọi hàm hay gọi thông điệp,
  - 1 biểu thức được đóng trong 2 dấu ().
- Qui trình tạo biểu thức là qui trình lặp đệ qui : ta kết hợp từng toán tử với các toán hạng của nó, rồi đóng trong 2 dấu () để biến nó trở thành biểu thức cơ bản, rồi dùng nó như 1 toán hạng để xây dựng biểu thức lớn hơn và phức tạp hơn...



## 1.6 Cú pháp định nghĩa biểu thức

- Dựa theo số toán hạng tham gia, có 3 loại toán tử thường dùng nhất :
  - toán tử 1 ngôi : chỉ cần 1 toán hạng. Ví dụ toán tử '-' để tính phần âm của 1 đại lượng.
  - toán tử 2 ngôi : cần dùng 2 toán hạng. Ví dụ toán tử '\*' để tính tích của 2 đại lượng.
  - toán tử 3 ngôi : cần dùng 3 toán hạng. Ví dụ toán tử 'c?v1:v2' để kiểm tra điều kiện c hầu lấy kết quả v1 hay v2.
- VC# thường dùng các ký tự đặc biệt để miêu tả toán tử. Ví dụ :
  - toán tử '+' : cộng 2 đại lượng.
  - toán tử '-' : trừ đại lượng 2 ra khỏi đại lượng 1.
  - toán tử '\*' : nhân 2 đại lượng.
  - toán tử '/' : chia đại lượng 1 cho đại lượng 2...



## 1.6 Cú pháp định nghĩa biểu thức

- Trong vài trường hợp, VC# dùng cùng 1 ký tự đặc biệt để miêu tả nhiều toán tử khác nhau. Trong trường hợp này, ngữ cảnh sẽ được dùng để giải quyết nhầm lẫn.
- Ngữ cảnh thường là kiểu và số lượng các toán hạng tham gia.
- Thí dụ :
  - x // - là phép toán 1 ngôi
  - a-b // - là phép toán 2 ngôi
- Trong vài trường hợp khác, VC# dùng chuỗi nhiều ký tự để miêu tả 1 toán tử. Thí dụ :
  - a >= b // >= là toán tử so sánh lớn hơn hay bằng
  - a++ // ++ là toán tử tăng 1 đơn vị
  - a == b // == là toán tử so sánh bằng (không phải là toán tử gán)



## 1.6 Cú pháp định nghĩa biểu thức

- Giá trị luận lý : `true` | `false`
- Giá trị thập phân nguyên : `(+|-)? (decdigit)+` (Vd. 125, -548)
- Giá trị thập lục phân nguyên : `(+|-)? "0x" (hexdigit)+` (`0xFF`)
- Giá trị bát phân nguyên : `(+|-)? "0" (ocdigit)+` (`0577`)
- Giá trị nhị phân nguyên : `(+|-)? (bidigit)+ "b"` (`101110b`)
- Giá trị thập phân thực :  
`(+|-)? (decdigit)+ ("." (decdigit)*)? ("E" (+|-)? (decdigit)+)?`  
Vd : 3.14159, 0.31459e1, -83.1e-9,...
- Giá trị chuỗi : `"Nguyen Van A"`  
`"\"Nguyen Van A\""`
- Lưu ý dùng ký tự `\` để thực hiện cơ chế 'escape' dữ liệu hầu giải quyết nhầm lẫn.





## 1.7 Qui trình tính biểu thức

- Một biểu thức có thể chứa nhiều phép toán, qui trình tính toán biểu thức như sau : duyệt từ trái sang phải, mỗi lần gặp 1 phép toán (ta gọi là CurrentOp) thì phải nhìn trước phép toán đi ngay sau nó (SuccessorOp), so sánh độ ưu tiên của 2 phép toán và ra quyết định như sau :
  - nếu không có SuccessorOp thì tính ngay phép toán CurrentOp (trên 1, 2 hay 3 toán hạng của nó).
  - nếu phép toán CurrentOp có độ ưu tiên cao hơn phép toán SuccessorOp thì tính ngay phép toán CurrentOp (trên 1, 2 hay 3 toán hạng của nó).
  - nếu phép toán CurrentOp có độ ưu tiên bằng phép toán SuccessorOp và kết hợp trái thì tính ngay phép toán CurrentOp (trên 1, 2 hay 3 toán hạng của nó).



## 1.7 Qui trình tính biểu thức

- các trường hợp còn lại thì cố gắng thực hiện phép toán SuccessorOp trước. Việc cố gắng này cũng phải tuân theo các qui định trên,...
- Khi phép toán SussesorOp được thực hiện xong thì phép toán ngay sau SuccessorOp trở thành phép toán đi ngay sau CurrentOp việc kiểm tra xem CurrentOp có được thực hiện hay không sẽ được lặp lại.



## 1.7 Qui trình tính biểu thức

Bảng liệt kê độ ưu tiên của các toán tử từ trên xuống = từ cao xuống thấp :

Operator	Name or Meaning	Associativity
[ ]	Array subscript	Left to right
( )	Function call	Left to right
( )	Conversion	None
.	Member selection (object)	Left to right
++	Postfix increment	None
--	Postfix decrement	None
new	Allocate object	None
typeof	Type of	
checked		
unchecked		



## 1.7 Qui trình tính biểu thức

Operator	Name or Meaning	Associativity
++	Prefix increment	None
--	Prefix decrement	None
+	Unary plus	None
-	Arithmetic negation (unary)	None
!	Logical NOT	None
~	Bitwise complement	None
&	Address of	None
sizeof ( )	Size of type	None
typeid( )	type name	None
(type)	Type cast (conversion)	Right to left



## 1.7 Qui trình tính biểu thức

Operator	Name or Meaning	Associativity
*	Multiplication	Left to right
/	Division	Left to right
%	Remainder (modulus)	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
is		
as		



## 1.7 Qui trình tính biểu thức

Operator	Name or Meaning	Associativity
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
	Bitwise OR	Left to right
&&	Logical AND	Left to right
	Logical OR	Left to right
e1?e2:e3	Conditional	Right to left
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left



## 1.7 Qui trình tính biểu thức

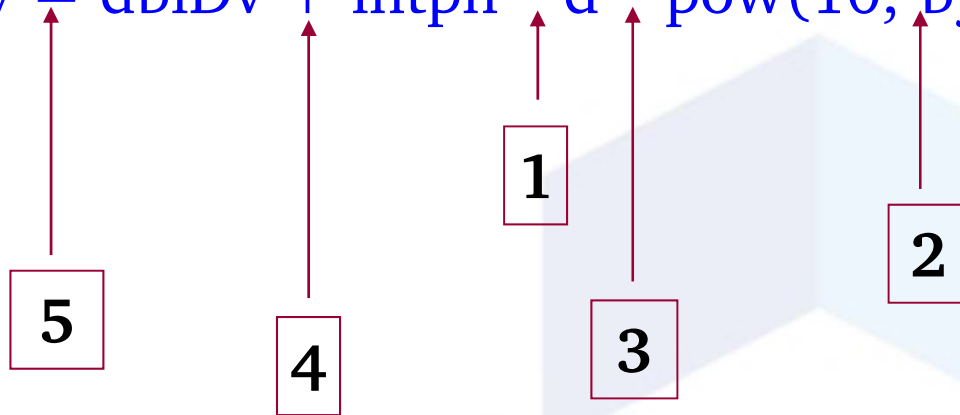
Operator	Name or Meaning	Associativity
<<=	Left-shift assignment	Right to left
>>=	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
??	null-coalescing	
,	Comma	Left to right



## 1.7 Qui trình tính biểu thức

- Thí dụ :

`dblDv = dblDv + intpn * d * pow(10,-bytPosDigit);`





## 1.8 Các lệnh định nghĩa thành phần phần mềm

### Định nghĩa hằng gọi nhớ

- Cú pháp định nghĩa hằng gọi nhớ cơ bản :

$\text{ConstDef} ::= \text{"const" S TName S Name S? "=" S? Expr S? ";"}$

Thí dụ :

`const double PI = 3.1416;`

### Định nghĩa biến cục bộ trong hàm

- Cú pháp định nghĩa biến cục bộ trong hàm :

$\text{VarDef} ::= \text{TName S Name (S? "=" S? Expr S?)? ";"}$

Thí dụ :

`double epsilon = 0.000001;`

**Định nghĩa kiểu người dùng** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)



## 1.8 Các lệnh định nghĩa thành phần phần mềm

**Định nghĩa hàm hay tác vụ chức năng** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)

**Định nghĩa chương trình** (học chi tiết trong môn Kỹ thuật lập trình và các chương sau của môn này)



## 1.9 Các lệnh thực thi

- Ta đã biết giải thuật để giải quyết 1 vấn đề nào đó là trình tự các công việc nhỏ hơn, nếu ta thực hiện đúng trình tự các công việc nhỏ hơn này thì sẽ giải quyết được vấn đề lớn.
- VC# (hay ngôn ngữ lập trình khác) cung cấp 1 tập các lệnh thực thi, mỗi lệnh thực thi được dùng để miêu tả 1 công việc nhỏ trong 1 giải thuật với ý tưởng chung như sau :
  - Nếu tồn tại lệnh thực thi miêu tả được công việc nhỏ của giải thuật thì ta dùng lệnh thực thi này để miêu tả nó.
  - Nếu công việc nhỏ của thuật giải vẫn còn quá phức tạp và không có lệnh thực thi nào miêu tả được thì ta dùng lệnh gọi hàm (function, method) trong đó hàm là trình tự các lệnh thực hiện công việc nhỏ này...



## 1.9 Các lệnh thực thi

- Hầu hết các lệnh thực thi đều có chứa biểu thức và dùng kết quả của biểu thức này để quyết định công việc kế tiếp cần được thực hiện  $\Rightarrow$  ta thường gọi các lệnh thực thi là các cấu trúc điều khiển.
- Để dễ học, dễ nhớ và dễ dùng, VC# (cũng như các ngôn ngữ khác) chỉ cung cấp 1 số lượng rất nhỏ các lệnh thực thi :
- **Nhóm lệnh không điều khiển :**
  - Lệnh gán dữ liệu vào 1 biến.
- **Nhóm lệnh tạo quyết định/rẽ nhánh :**
  - Lệnh kiểm tra điều kiện luận lý if ... else ...
  - Lệnh kiểm tra điều kiện số học switch
  - Lệnh goto



## 1.9 Các lệnh thực thi

- Nhóm lệnh lặp :
  - Lệnh lặp : while
  - Lệnh lặp : for, foreach
  - Lệnh lặp : do ... while
- Nhóm lệnh gọi hàm :
  - Lệnh gọi hàm
  - Lệnh thoát khỏi cấu trúc điều khiển : break, continue
  - Lệnh thoát khỏi hàm : return, throw
- Nhóm lệnh xử lý lỗi theo cơ chế ngoại lệ (Exception) :
  - Lệnh try
  - Mệnh đề catch của lệnh try
  - Mệnh đề finally của lệnh try



## 1.9 Các lệnh thực thi

**Lệnh gán** : là lệnh được dùng nhiều nhất trong chương trình, chức năng của lệnh này là gán giá trị dữ liệu vào 1 vùng nhớ để lưu trữ hầu sử dụng lại nó sau đó. Cú pháp :

**lvar S? "=" S? Expr S? ";"**

- biểu thức Expr bên phải sẽ được tính để tạo ra kết quả (1 giá trị cụ thể thuộc 1 kiểu cụ thể), giá trị này sẽ được gán vào ô nhớ do lvar qui định. Trước khi gán, VC# sẽ kiểm tra kiểu của 2 phần tử (qui tắc kiểm tra sẽ được trình bày sau).
- lvar có thể là biến đơn (intTuoi), phần tử của biến array (matran[2,3]), thuộc tính của đối tượng (rect.dorong).
- Ví dụ :

**x1 = (-b-Math.sqrt(delta))/2/a;**



## 1.9 Các lệnh thực thi

Lệnh kiểm tra điều kiện luận lý if ... else : cho phép dựa vào kết quả luận lý (tính được từ 1 biểu thức luận lý) để quyết định thi hành 1 trong 2 nhánh lệnh. Sau khi thực hiện 1 trong 2 nhánh lệnh, chương trình sẽ tiếp tục thi hành lệnh ngay sau lệnh IF. Cú pháp :

"if" S? "(" S? Expr S? ")" S? Statement S? ("else" S Statement)?

▫ Thí dụ :

if (delta < 0) //báo sai

System.Console.WriteLine ("Phương trình vô nghiệm");

else { //tính 2 nghiệm

x1 = (-b-sqrt(delta))/2/a;

x2 = (-b+sqrt(delta))/2/a;

}



## 1.9 Các lệnh thực thi

Lệnh kiểm tra điều kiện số học switch : cho phép dựa vào kết quả số học (tính được từ 1 biểu thức số học) để quyết định thi hành 1 trong n nhánh lệnh. Sau khi thực hiện 1 trong n nhánh lệnh, chương trình sẽ tiếp tục thi hành lệnh ngay sau lệnh switch. Cú pháp :

```
"switch" S? "(" Expr S? ")" S? "{" S?  
    "case" S expr1 S? ":" S? Statement*  
    "case" S expr2 S? ":" S? Statement*  
    ...  
    "case" S exprn S? ":" S? Statement*  
    ("default" S? ":" S? Statement*)? S?  
    "}"
```





## 1.9 Các lệnh thực thi

- Thí dụ :

```
switch (diem) {  
case 0 : case 1 : case 2 : case 3 : case 4 :  
    Console.WriteLine("Quá yếu"); break;  
case 5 : case 6 :  
    Console.WriteLine("Trung bình"); break;  
case 7 : case 8 :  
    Console.WriteLine("Khá"); break;  
case 9 : case 10 :  
    Console.WriteLine("Giỏi"); break;  
}
```



## 1.9 Các lệnh thực thi

Lệnh lặp do... while : cho phép lặp thực hiện 1 công việc nào đó từ 1 tới n lần theo 1 điều kiện kiểm soát. Cú pháp :

"do" S Statement S? "while" S? "(" S? Expr S? ")" S? ";"

▫ Thí dụ :

```
int i = 1;
```

```
long giaithua = 1;
```

```
do {
```

```
    i = i+1;
```

```
    giaithua = giaithua*i;
```

```
} while (i < n)
```

```
do
```

```
    giaithua *= (++i);
```

```
while (i < n);
```



## 1.9 Các lệnh thực thi

Lệnh lặp **while** : cho phép lặp thực hiện 1 công việc nào đó từ 0 tới n lần theo 1 điều kiện kiểm soát. Cú pháp :

"while" S? "(" S? Expr S? ")" S? Statement

▫ Thí dụ :

```
int i = 1;
```

```
long giaithua = 1;
```

```
while (i < n) {
```

```
    i = i+1;
```

```
    giaithua = giaithua*i;
```

```
}
```

```
while (i < n) giaithua *= (++i);
```



## 1.9 Các lệnh thực thi

Lệnh lặp for : cho phép lặp thực hiện 1 công việc nào đó từ 0 tới n lần theo 1 điều kiện kiểm soát số học. Cú pháp :

"for" S? "(" S? init-expr? S? ";" S? cond-expr? ";" S? loop-expr? S? ")"  
S? Statement

▫ Thí dụ :

```
int i;
```

```
long giaithua = 1;
```

```
for (i=2; i <=n; i++) {  
    giaithua = giaithua*i;  
}
```

```
for (i=2; i <=n; i++)  
    giaithua *= i;
```



## 1.9 Các lệnh thực thi

Lệnh lặp `foreach` : cho phép lặp thực hiện 1 công việc nào đó trên từng phần tử của 1 tập hợp xác định. Cú pháp :

`"foreach" "(" elemType elem "in" setVar ")" Statement`

Thí dụ :

```
Sinhvien[] dsSV;
```

```
...
```

```
foreach (Sinhvien sv in dsSV) {
```

```
    //xử lý sinh viên sv
```

```
}
```



## 1.9 Các lệnh thực thi

**Các lệnh lồng nhau :** Như ta đã thấy trong cú pháp của hầu hết các lệnh VC# đều có chứa thành phần Statement, đây là 1 lệnh thực thi VC# bất kỳ ta gọi cú pháp định nghĩa lệnh VC# là lặp đệ qui tạo ra các lệnh VC# lồng nhau. Ta gọi cấp ngoài cùng là cấp 1, các lệnh trong thân lệnh cấp 1 được gọi là lệnh cấp 2, các lệnh trong thân lệnh cấp 2 được gọi là lệnh cấp 3,... Để dễ đọc, các lệnh cấp thứ i nên giống cột nhờ i-1 ký tự Tab.

Ví dụ : đoạn chương trình tính ma trận tổng của 2 ma trận

```
const int N = 100;
```

```
double[,] a, b, c;
```

```
...
```

```
for (i = 0; i < N; i++)
```

```
    for (j = 0; j < N; j++)
```

```
        c[i, j] = a[i, j] + b[i, j];
```

//duyet theo hàng

//duyet theo cột



## 1.9 Các lệnh thực thi

Vấn đề thoát đột ngột khỏi cấp điều khiển : Trong cú pháp của hầu hết các lệnh VC# đều có chứa thành phần Statement mà đa số là phát biểu kép chứa nhiều lệnh khác. Theo trình tự thi hành thông thường, các lệnh bên trong phát biểu kép sẽ được thực thi tuần tự, hết lệnh này đến lệnh khác cho đến lệnh cuối, lúc này thì việc thi hành lệnh cha mới kết thúc. Tuy nhiên trong 1 vài trạng thái thi hành đặc biệt, ta muốn thoát ra khỏi lệnh cha đột ngột chứ không muốn thực thi hết các lệnh con trong danh sách. Để phục vụ yêu cầu này, VC# cung cấp lệnh break với cú pháp đơn giản sau đây :

**break;**

Lưu ý lệnh break chỉ cho phép thoát khỏi cấp trong cùng (lệnh chứa lệnh break. Để thoát trực tiếp ra nhiều cấp 1 cách tự do, ta dùng lệnh goto với cú pháp :

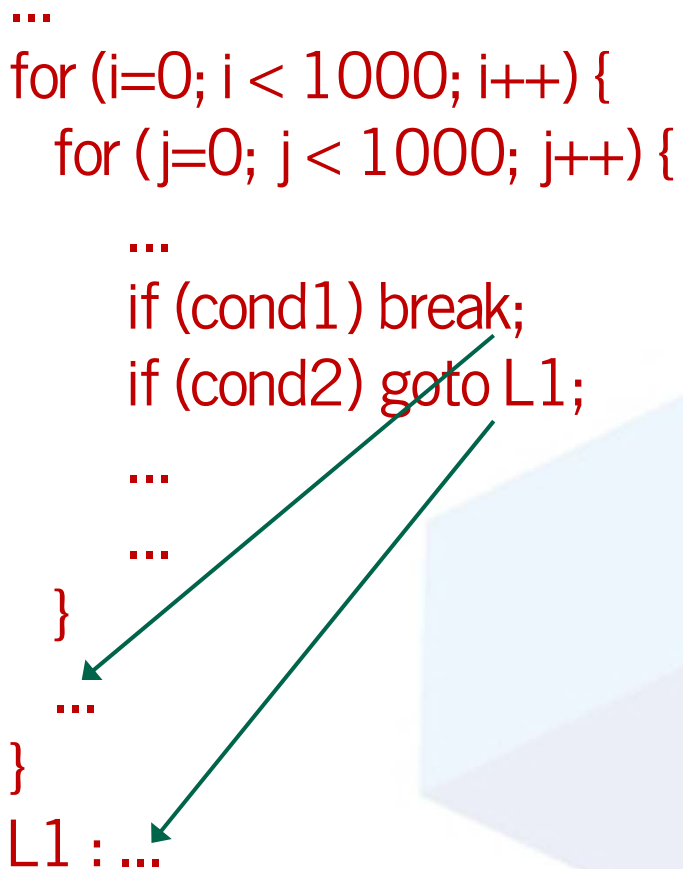
**goto stat\_label;** //trong đó stat\_label là nhãn của lệnh cần goto đến.



## 1.9 Các lệnh thực thi

Vấn đề thoát đột ngột khỏi cấp điều khiển : phân biệt lệnh break và goto

```
...  
for (i=0; i < 1000; i++) {  
    for (j=0; j < 1000; j++) {  
        ...  
        if (cond1) break;  
        if (cond2) goto L1;  
        ...  
    }  
    ...  
}  
L1 : ...
```

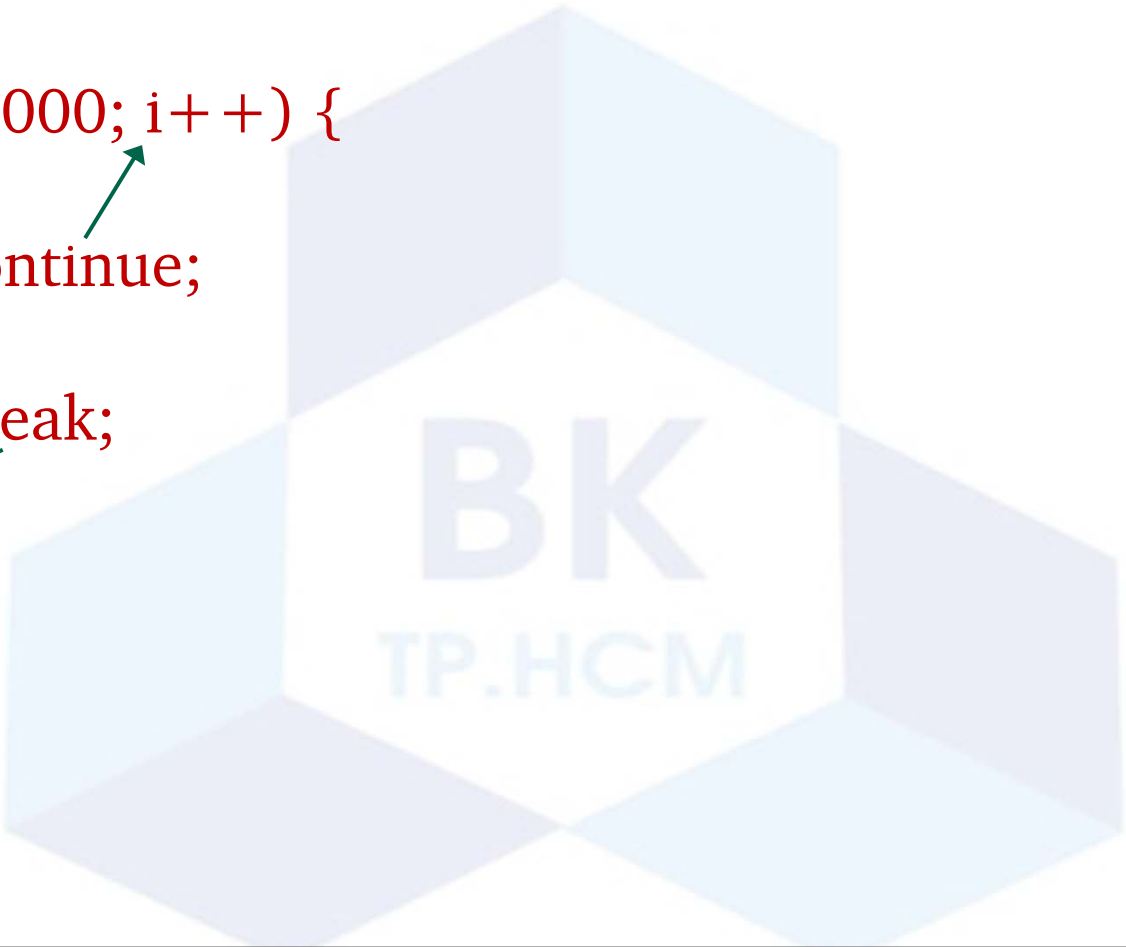




## 1.9 Các lệnh thực thi

Vấn đề thoát đột ngột khỏi cấp điều khiển : phân biệt lệnh break và continue

```
...  
for (i=0; i < 1000; i++) {  
    ...  
    if (cond1) continue;  
    ...  
    if (cond2) break;  
    ...  
}  
...
```



## 1.9 Các lệnh thực thi

**Vấn đề thoát đột ngột khỏi hàm :** Như ta đã biết hàm là danh sách các lệnh thực thi để thực hiện 1 chức năng nào đó. Thông thường thì danh sách lệnh này sẽ được thực hiện từ đầu đến cuối rồi điều khiển sẽ được trả về lệnh gọi hàm này, tuy nhiên ta có quyền trả điều khiển về lệnh gọi hàm bất cứ đâu trong danh sách lệnh của hàm. Cú pháp lệnh trả điều khiển như sau :

"return" S? ";" // nếu hàm có kiểu trả về là void

"return" S? "(" S? expr S? ")" S? ";" // nếu hàm có kiểu trả về void

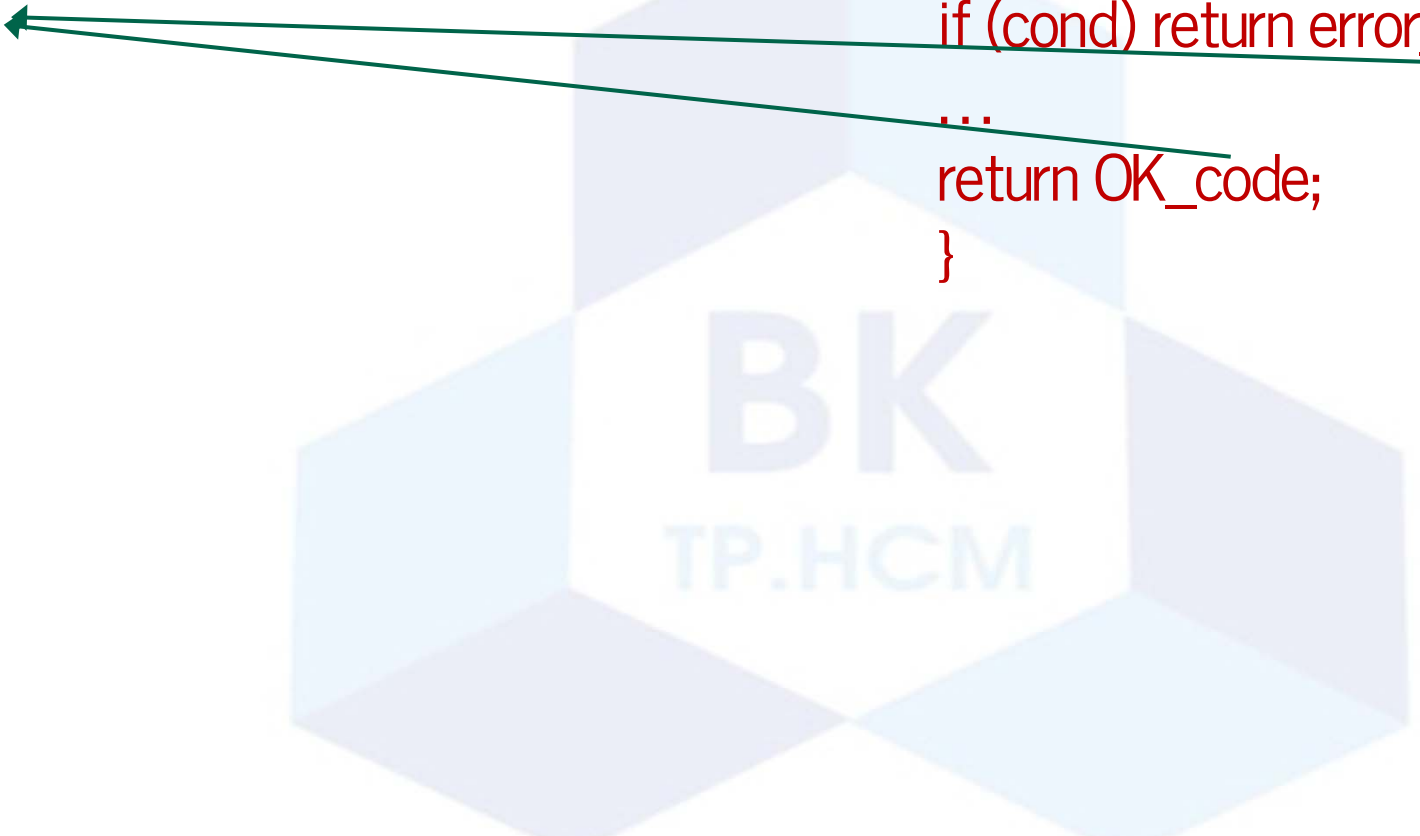


## 1.9 Các lệnh thực thi

Vấn đề thoát đột ngột khỏi hàm :

```
...  
kq = func1();  
...
```

```
int func1() {  
...  
if (cond) return error_code1;  
...  
return OK_code;  
}
```



## 1.9 Các lệnh thực thi

Vấn đề thoát đột ngột khỏi hàm :

Thuật giải gốc :

1. thực hiện chức năng 1
2. thực hiện chức năng 2
3. thực hiện chức năng 3

Dịch ra mã nguồn :

```
func1();  
func2();  
func3();
```

```
int func1() {  
    ...  
    if (cond1) return error_code1;  
    if (cond2) return error_code2;  
    ...  
    return OK_code;  
}
```



## 1.9 Các lệnh thực thi

Dịch ra mã nguồn :

```
int kq = func1();  
switch (kq) {  
case error_code1 :  
    //đoạn code giải quyết lỗi 1  
case error_code2 :  
    //đoạn code giải quyết lỗi 2  
}  
kq = func2();  
//kiểm tra và xử lý lỗi  
kq = func3();  
//kiểm tra và xử lý lỗi
```

```
int func1() {  
...  
if (cond1) return error_code1;  
if (cond2) return error_code2;  
...  
return OK_code;  
}
```

⇒ code mã nguồn quá phức tạp và khó hiểu so với giải thuật gốc



## 1.9 Các lệnh thực thi

Thoát khỏi hàm theo cơ chế Exception : đây là phương án mới để khắc phục việc thoát khỏi hàm bằng lệnh return.

Dịch ra mã nguồn :

```
try {  
    func1();  
    func2();  
    func3();  
}  
catch (Exception1 e1) {  
    //đoạn code giải quyết loại lỗi Exception1  
}  
catch (Exception2 e2) {  
    //đoạn code giải quyết loại lỗi Exception2  
}
```

```
void func1() {  
    ...  
    if (cond1) throw new Exception1("lỗi 1");  
    if (cond2) throw new Exception2("lỗi 2");  
    ...  
}
```



## 1.9 Các lệnh thực thi

Thoát khỏi hàm theo cơ chế Exception : đây là phương án mới để khắc phục việc thoát khỏi hàm bằng lệnh return.

Dịch ra mã nguồn :

```
void func() {  
    func1();  
    func2();  
    func3();  
}
```

```
void func1() {  
    ...  
    if (cond1) throw new Exception1("lỗi 1");  
    if (cond2) throw new Exception2("lỗi 2");  
    ...  
}
```

về lệnh try ... catch chứa lệnh gọi hàm func()... Cách xử lý này được lặp lại đến hàm Main(), nếu hàm Main() cũng không dùng lệnh try catch thì OS sẽ tự xử lý Exception dùm (trên Windows là 1 cửa sổ báo lỗi màu xanh rồi dừng luôn chương trình).



## 1.10 Kết chương

- ❑ Chương này đã tóm tắt lại 1 số kiến thức cơ bản về cú pháp của ngôn ngữ VC# hầu giúp các SV có góc nhìn tổng thể và hệ thống về ngôn ngữ VC#, nhờ đó có nhiều thuận lợi hơn trong việc học các kiến thức của môn học này.

