Sequence Control

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

07, 2014

Outline

Expressions

- An expression is a syntactic entity whose evaluation either:
  - produces a value
  - fails to terminate $\rightarrow$ undefined
- Examples

  4 + 3 * 2

  (a + b) * (c - a)

  (b != 0) ? (a/b) : 0

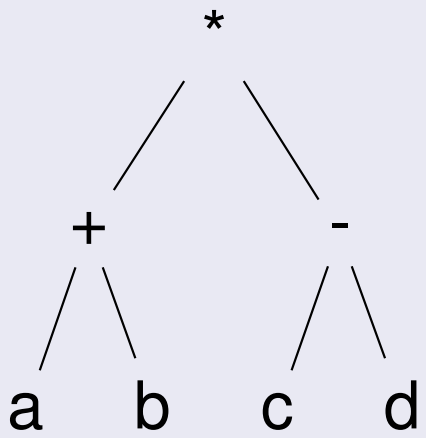
"expression" trong ngữ cảnh này được hiểu là một cấu trúc cú pháp mà khi được tính toán sẽ cho ra một giá trị, hoặc có thể dẫn đến việc không xác định (undefined) nếu việc tính toán đó không kết thúc được.

Expressions (cont'd)

Expression Evaluation Mechanism (Cơ chế tính toán của Expression): Các biểu thức có bản chất là kết hợp các hàm. Ví dụ, biểu thức (a + b) * (c - d) có thể được hiểu là việc thực hiện các phép cộng, trừ trước, sau đó mới đến phép nhân.

Expression Evaluation Mechanism

Expressions have functional composition nature



(a + b) * (c - d)

Expression Syntax

- Infix
- Prefix
- Postfix

Expression Syntax (Cú pháp của Expression): Có 3 dạng cú pháp chính:

Infix: Là dạng toán tử nằm giữa các toán hạng, ví dụ: a + b. Dạng này phổ biến trong các ngôn ngữ lập trình mệnh lệnh.
Prefix: Là dạng toán tử nằm trước các toán hạng, ví dụ: * + a b - c d.
Postfix: Là dạng toán tử nằm sau các toán hạng, ví dụ: a b + c d - *.

Mỗi dạng cú pháp sẽ có cách đọc và tính toán khác nhau. Infix thì dễ đọc dễ hiểu nhất, nhưng Prefix và Postfix thì lại có ưu điểm là không cần dùng dấu ngoặc đơn để xác định độ ưu tiên của các toán tử.

Infix Notation

"Infix Notation" là cách viết mà toán tử nằm giữa các toán hạng, ví dụ như (a + b) * (c - d). Cách này rất là tiện lợi khi dùng cho các toán tử hai ngôi (binary operators), tức là các toán tử cần hai toán hạng để thực hiện phép tính. Đó là lý do vì sao nó được sử dụng rộng rãi trong hầu hết các ngôn ngữ lập trình mệnh lệnh.

$$(a + b) * (c - d)$$

- Good for binary operators
- Used in most imperative programming language
- More than two operands?
  (b != 0) ? (a/b) : 0
- Smalltalk:
  myBox displayOn: myScreen at: 100@50

Tuy nhiên, khi mà biểu thức có nhiều hơn hai toán hạng thì sao? Ví dụ như cái biểu thức điều kiện (b != 0) ? (a/b) : 0 mà mình đã nói ở trên, hoặc một ví dụ khác trong Smalltalk là myBox displayOn: myScreen at: 100@50. Mày thấy đó, trong những trường hợp này, việc đọc và hiểu biểu thức có thể trở nên phức tạp hơn một chút.

Precedence

độ ưu tiên của các toán tử trong một biểu thức

$$3 + 4 * 5 = 23, \text{ not } 35$$

- Evaluation priorities in mathematics
- Programming languages define their own precedence levels based on mathematics
- A bit different precedence rules among languages can be confusing

Trong toán học, phép nhân (*) có độ ưu tiên cao hơn phép cộng (+). Cho nên, khi tính biểu thức này, mình phải tính 4 * 5 trước, rồi mới cộng với 3. Nếu không để ý tới độ ưu tiên này, mình sẽ ra kết quả sai là 35.

Các ngôn ngữ lập trình cũng định nghĩa độ ưu tiên của các toán tử dựa trên quy tắc toán học. Tuy nhiên, có một chút khác biệt nhỏ giữa các ngôn ngữ, và điều này đôi khi gây khó hiểu cho người lập trình, đặc biệt là khi chuyển đổi giữa các ngôn ngữ khác nhau.

Associativity

"Associativity" là tính chất kết hợp của các toán tử. Khi trong một biểu thức có nhiều toán tử cùng độ ưu tiên, thì mình cần phải có quy tắc để xác định xem toán tử nào được thực hiện trước. Đó chính là quy tắc kết hợp.

Trong hầu hết các ngôn ngữ lập trình, quy tắc kết hợp thường là từ trái sang phải. Ví dụ, biểu thức a - b - c sẽ được hiểu là (a - b) - c. Tuy nhiên, có một số trường hợp ngoại lệ, chẳng hạn như toán tử lũy thừa, thường có quy tắc kết hợp từ phải sang trái.

- If operators have the same level of precedence, then apply associativity rules
- Mostly left-to-right, except exponentiation operator
- An expression contains only one operator
  - Mathematics: associative
  - Computer: optimization but potential problems
  - $10^{20} * 10^{20} * 10^{-20}$

Trong toán học, các toán tử có tính chất kết hợp (associative), tức là thứ tự thực hiện các phép toán không ảnh hưởng đến kết quả. Ví dụ: (a + b) + c = a + (b + c). Tuy nhiên, trong máy tính, do vấn đề tối ưu hóa, thứ tự thực hiện có thể ảnh hưởng đến kết quả, đặc biệt là với các phép toán trên số thực, ví dụ 10^20 * 10^20 * 10^-20.

Parentheses

Dấu ngoặc đơn được dùng để thay đổi độ ưu tiên và tính kết hợp của các toán tử trong một biểu thức. Ví dụ, trong biểu thức (A + B) * C, dấu ngoặc đơn sẽ buộc phép cộng A + B phải được thực hiện trước phép nhân.

Thậm chí, một số ngôn ngữ lập trình có thể bỏ qua luôn các quy tắc về độ ưu tiên và tính kết hợp nếu sử dụng dấu ngoặc đơn một cách triệt để. APL là một ví dụ điển hình.

- Alter the precedence and associativity
  (A + B) * C
- Using parentheses, a language can even omit precedence and associativity rules
  - APL
- Advantage: simple
- Disadvantage: writability and readability

Việc sử dụng dấu ngoặc đơn có ưu điểm là đơn giản, dễ hiểu. Tuy nhiên, nó cũng có nhược điểm là làm cho code trở nên khó viết và khó đọc hơn, đặc biệt là khi biểu thức trở nên phức tạp.

Conditional Expressions

"Conditional Expressions" (biểu thức điều kiện) là một cách viết gọn của câu lệnh if...else

If statement

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

Thay vì viết 5 dòng cho câu lệnh if...else, mình chỉ cần viết 1 dòng với biểu thức điều kiện. Cú pháp của nó là (điều kiện) ? (giá trị nếu đúng) : (giá trị nếu sai). Nếu điều kiện đúng thì nó trả về giá trị thứ nhất, còn nếu sai thì nó trả về giá trị thứ hai.

Conditional Expression

average = (count == 0) ? 0 : sum / count;

- C-based languages, Perl, JavaScript, Ruby

Prefix Notation

Prefix Notation (Ký pháp tiền tố):
Toán tử đứng trước toán hạng. Ví dụ: * + a b - c d.
Có nhiều biến thể như Polish Prefix, Cambridge Polish Prefix, và Normal Prefix.

- Polish Prefix: * + a b - c d
- Cambridge Polish Prefix: (* (+ a b) (- c d))
- Normal Prefix: *(+(a,b),-(c,d))
  - Derived from mathematical function f(x,y)
  - Parentheses and precedence is no required, provided the -arity of operator is known
  - Mostly see in unary operators
  - LISP: (**append** a b c my_list)

Bắt nguồn từ hàm toán học f(x, y).
Không cần dấu ngoặc đơn hay quy tắc ưu tiên nếu biết rõ số lượng toán hạng của toán tử (arity).
Thường thấy ở các toán tử một ngôi (unary operators).
Ví dụ trong LISP: (append a b c my_list).

Postfix Notation

Postfix Notation (Ký pháp hậu tố):
Toán tử đứng sau toán hạng. Ví dụ: a b + c d - *.
Cũng có các biến thể như Polish Postfix, Cambridge Polish Postfix, và Normal Postfix.
Thường dùng cho toán tử giai thừa (ví dụ: 5!)

Được dùng trong mã trung gian của một số trình biên dịch.
Ví dụ trong PostScript: (Hello World!) show.

- Polish Postfix: a b + c d - *
- Cambridge Polish Postfix: ((a b +) (c d -) *)
- Normal Postfix: ((a,b)+,(c,d)-)*
  - Common usage: factorial operator (5!)
  - Used in intermediate code by some compilers
  - PostScript: (Hello World!) **show**

Operand Evaluation Order

Khi mà trong một biểu thức có nhiều toán hạng, thì thứ tự mà các toán hạng đó được tính toán có thể ảnh hưởng đến kết quả cuối cùng.

C program

```
int a = 5;
int fun1() {
    a = 17;
    return 3;
}
void main() {
    a = a + fun1();
}
```

Giá trị của a sau khi chạy đoạn code này có thể là 8 hoặc 20, tùy thuộc vào việc a và fun1() được tính toán theo thứ tự nào. Nếu a được lấy giá trị trước, rồi sau đó fun1() được gọi, thì a sẽ là 5 + 3 = 8. Nhưng nếu fun1() được gọi trước, nó sẽ thay đổi giá trị của a thành 17, rồi trả về 3, và khi đó a sẽ là 17 + 3 = 20.

What is the value of a? 8 20

Reason: Side-effect on the operand of the expression

Cái "side-effect" (tác dụng phụ) trong tài liệu nhắc tới là việc hàm fun1() làm thay đổi giá trị của biến a bên ngoài hàm đó. Chính cái tác dụng phụ này gây ra sự khác biệt trong kết quả.

Evaluation Mechanisms

Eager evaluation (Tính toán háo hức):
Tính toán tất cả các toán hạng trước.
Sau đó mới thực hiện các toán tử.
Ví dụ: trong biểu thức a == 0 ? b : b / a, nếu dùng eager evaluation, thì cả a, b, và b / a đều sẽ được tính toán trước, rồi sau đó mới xét điều kiện a == 0.

- Eager evaluation
  - First evaluate all operands
  - Then operators
  - How about a == 0 ? b : b / a
- Lazy evaluation
  - Pass the un-evaluated operands to the operator
  - Operator decide which operands are required
  - Much more expensive than eager
- Lazy for conditional, eager for the rest

nếu a mà bằng 0 thì b / a sẽ gây ra lỗi "chia cho 0". Nếu dùng eager evaluation thì lỗi này có thể xảy ra, nhưng nếu dùng lazy evaluation thì sẽ không, vì nếu a == 0 đúng thì b / a sẽ không được tính toán.

Lazy evaluation (Tính toán lười biếng):
Truyền các toán hạng chưa được tính toán cho toán tử.
Toán tử sẽ quyết định toán hạng nào cần thiết để tính toán.
Ví dụ: với biểu thức a == 0 ? b : b / a, nếu a == 0 là đúng, thì b / a sẽ không cần tính toán.
Lazy evaluation thường tốn kém hơn eager evaluation.

Thường dùng lazy evaluation cho các biểu thức điều kiện, và eager evaluation cho phần còn lại.

Short-Circuit Evaluation

"Short-Circuit Evaluation" (Tính toán đoản mạch) là một kỹ thuật tính toán biểu thức logic mà trong đó, việc tính toán có thể dừng lại giữa chừng nếu kết quả cuối cùng đã được xác định.

Ví dụ trong tài liệu: (a == 0) || (b/a > 2). Trong biểu thức này, nếu a == 0 là đúng, thì cả cái biểu thức (a == 0) || (b/a > 2) sẽ là đúng, không cần biết b/a > 2 là gì. Cho nên, nếu dùng "Short-Circuit Evaluation", thì khi a == 0 đúng, máy tính sẽ không cần tính b/a > 2 nữa, để tránh cái lỗi "chia cho 0".

$$(a == 0) \;||\; (b/a > 2)$$

- If the first operand is evaluated as true, the second will be short-circuited
- Otherwise, "divide by zero"
- How about (a > b) || (b++ / 3) ?
- Some languages provide two sets of boolean operators: short- and non short-circuit
  - Ada: "and", "or" versus "and then", "or else"

Ví dụ (a > b) || (b++ / 3). Ở đây, nếu a > b đúng, thì b++ / 3 sẽ không được tính, và giá trị của b sẽ không bị thay đổi.

Một số ngôn ngữ lập trình cung cấp cả hai loại toán tử logic: "short-circuit" và "non short-circuit". Ví dụ như Ada có "and", "or" (non short-circuit) và "and then", "or else" (short-circuit).

Statements

"statement" là một đơn vị cú pháp mà khi thực thi:

Không trả về giá trị.
Thay đổi trạng thái của hệ thống.

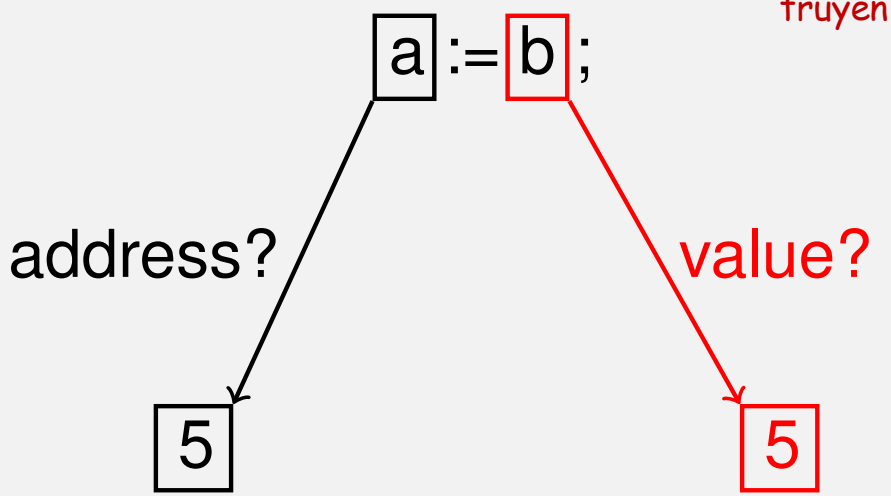
- A statement is a syntactic entity whose evaluation:
  - does not return a value, but
  - changes the state of the system
- Example,
  a = 5;
  print "pippo"
  begin ... **end**

Nói chung, "statement" là một chỉ thị để máy tính thực hiện một hành động nào đó, chứ không phải là một biểu thức để tính toán ra một giá trị.

Assignment Statements

Một câu lệnh gán thường có dạng leftExpr AssignOperator rightExpr, ví dụ như a := b;
Trong hình, bên trái dấu := là a (leftExpr), còn bên phải là b (rightExpr). Dấu := là toán tử gán (AssignOperator).

leftExpr AssignOperator rightExpr



Lấy giá trị của biến b, gán vào địa chỉ của biến a, khi đó giá trị của a sẽ bằng b (truyền tham trị).

Truyền tham trị (Pass by value): Khi mình gán a := b;, mình không hề đụng chạm gì đến biến b cả. Mình chỉ lấy cái giá trị hiện tại của b (là 5) và copy nó vào biến a. Sau khi gán xong, a và b là hai biến hoàn toàn độc lập, nằm ở hai vị trí khác nhau trong bộ nhớ. Nếu sau này mình có thay đổi giá trị của a thì cũng không ảnh hưởng gì đến b, và ngược lại.

- Evaluate left or right first is up to implementers

Thứ tự tính toán leftExpr và rightExpr là tùy thuộc vào cách mà người thiết kế ngôn ngữ lập trình quyết

Assignment Statements

Ngoài ra, các ngôn ngữ họ C còn coi phép gán là một biểu thức. Điều này có nghĩa là mình có thể dùng phép gán trong các biểu thức khác, ví dụ như trong vòng lặp while ((ch = getchar()) != EOF) {...}. Ở đây, phép gán ch = getchar() vừa gán giá trị cho ch, vừa trả về giá trị đã gán để so sánh với EOF.

- C-based languages consider assignment as an expression

```
while ((ch = getchar()) != EOF) {...}
```

- Introduce compound and unary assignment operators (+=, -=, ++, –)
  - Increasing code legibility
  - Avoiding unforeseen side effects

Các toán tử gán rút gọn và một ngôi như +=, -=, ++, --. Chúng giúp code dễ đọc hơn và tránh được những tác dụng phụ không mong muốn.

Control Structures

Cấu trúc điều khiển)

Control statements (Câu lệnh điều khiển): Dùng để chọn giữa các đường dẫn luồng điều khiển khác nhau, hoặc để lặp đi lặp lại việc thực thi một dãy các câu lệnh.
Control structure (Cấu trúc điều khiển): Là một câu lệnh điều khiển và tập hợp các câu lệnh mà nó điều khiển.
Nói một cách đơn giản, cấu trúc điều khiển là cách để mình điều khiển cái "dòng chảy" của chương trình, tức là mình muốn chương trình chạy theo hướng nào, có lặp lại cái gì hay không.

- Control statements
  - Selecting among alternative control flow paths
  - Causing the repeated execution of sequences of statements if, else, for, whie...
- Control structure is a control statement and the collection of its controlled statements

if, else, for, while... và code bên trong nó

Mày cứ hình dung chương trình của mình như là một cái dòng chảy vậy đó. Bình thường thì nó cứ chảy từ trên xuống dưới, từng câu lệnh một. Nhưng mà đôi khi mình muốn cái dòng chảy đó nó rẽ nhánh (chạy theo hướng này hoặc hướng kia tùy theo điều kiện), hoặc là mình muốn nó chảy lặp đi lặp lại một khúc nào đó.

Đó là lúc mình cần đến "cấu trúc điều khiển". Nó giống như là cái van, cái đập nước, hay là cái biển báo giao thông vậy đó. Mình dùng nó để điều khiển cái dòng chảy của chương trình theo ý mình muốn.

Ví dụ:

Câu lệnh điều khiển (Control statement): Là cái van, cái đập nước. Ví dụ như câu lệnh if...else (để rẽ nhánh), hoặc câu lệnh for, while (để lặp lại).
Cấu trúc điều khiển (Control structure): Là cả cái đoạn kênh mương mà mình điều khiển. Ví dụ như cả cái đoạn code if ...else (gồm câu lệnh if, câu lệnh else, và các câu lệnh bên trong), hoặc cả cái vòng lặp for (gồm câu lệnh for và các câu lệnh trong thân vòng lặp).

Two-way Selection

Two-way Selection (Rẽ nhánh hai hướng):
Chính là cái câu lệnh if...else.
Dạng tổng quát của nó là:
if control_expression
then clause
else clause
Cực kỳ quan trọng và không thể thiếu trong bất kỳ ngôn ngữ lập trình nào.

```
if control_expression
   then clause
   else clause
```

- Proved to be fundamental and essential parts of all programming languages

Dangling else

```
if (sum == 0)
    if (count == 0)
        result = 0;
    else
        result = 1;
```

"dangling else" (else treo lơ lửng), tức là khi có nhiều câu lệnh if lồng nhau mà chỉ có một câu lệnh else, thì không biết cái else đó thuộc về cái if nào.

Để giải quyết vấn đề này, người ta thường dùng block (dấu ngoặc nhọn {}) để xác định rõ ràng phạm vi của các câu lệnh if và else.

Một số ngôn ngữ khác như Fortran 95, Ada, Ruby thì dùng từ khóa đặc biệt để kết thúc câu lệnh if, còn Python thì dùng thụt lề để phân biệt.

- Solution: including block in every cases
- Not all languages have this problem
  - Fortran 95, Ada, Ruby: use a special word to end the statement
  - Python: indentation matters

Multiple-Selection

Multiple-Selection (Rẽ nhánh nhiều hướng):
Cái này cho phép mình chọn một trong nhiều đoạn code khác nhau để thực thi.
Perl và Python không có cái này.

switch...case

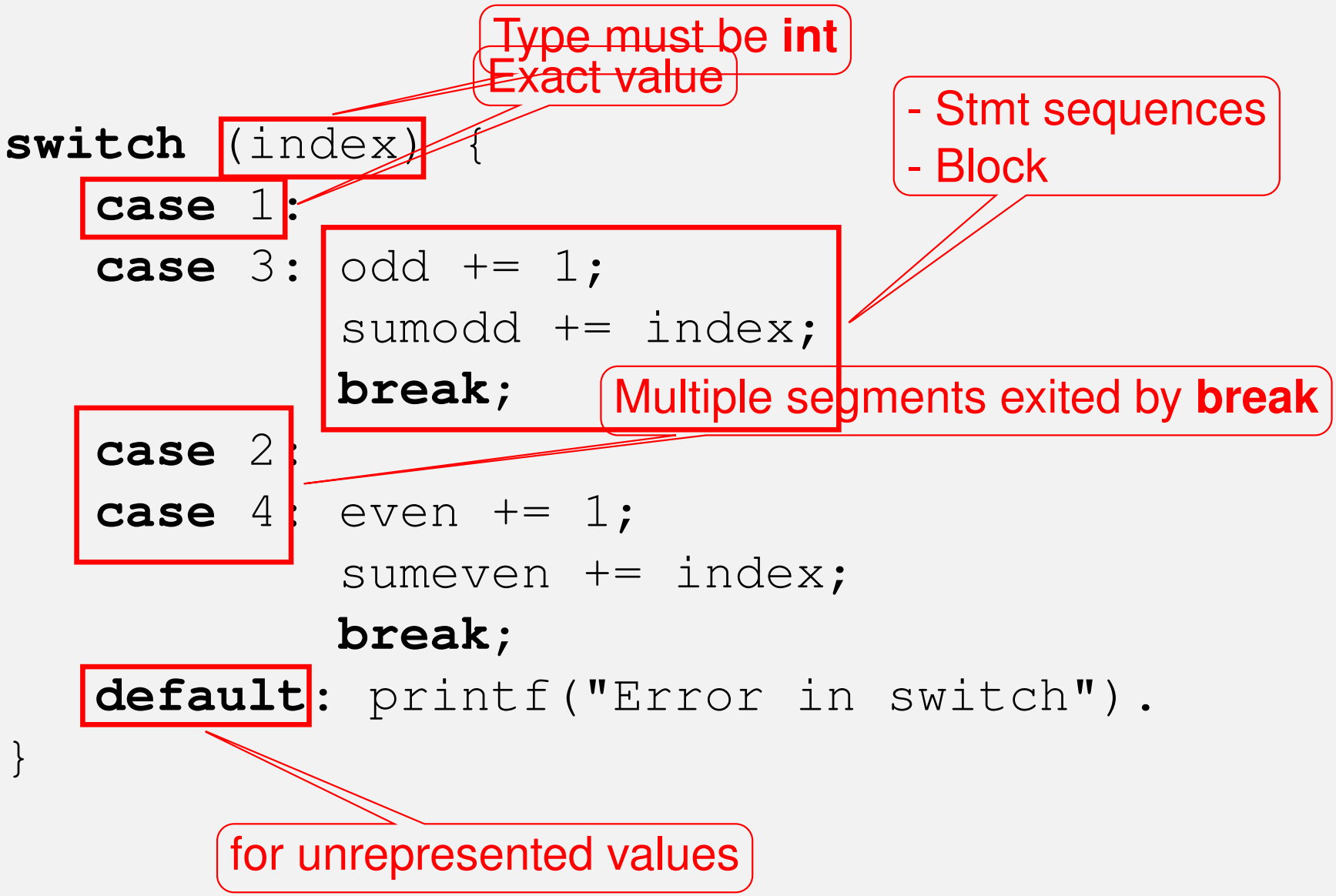
if, else if, else... không phải

Multiple-Selection thường ám chỉ những cấu trúc được thiết kế chuyên biệt để chọn một trong nhiều trường hợp dựa trên một giá trị cụ thể. Ví dụ điển hình nhất là câu lệnh switch...case. Ở đây, mình có một cái "selector expression" (ví dụ: một biến), và mình so sánh giá trị của nó với nhiều trường hợp khác nhau (case). Mỗi trường hợp tương ứng với một đoạn code riêng biệt.

- Allows the selection of one of any number of statements or statement groups
- Perl, Python: don't have this
- Issues:
  - Type of selector expression?
  - How are selectable segments specified?
  - Execute only one segment or multiple segments?
  - How are case values specified?
  - What if values fall out of selectable segments?

Một số vấn đề liên quan đến "Multiple-Selection", ví dụ như:
Kiểu dữ liệu của biểu thức điều kiện?
Cách xác định các đoạn code có thể chọn?
Chỉ thực thi một đoạn hay nhiều đoạn?
Cách chỉ định các giá trị trong mỗi trường hợp?
Xử lý thế nào nếu giá trị không thuộc trường hợp nào?

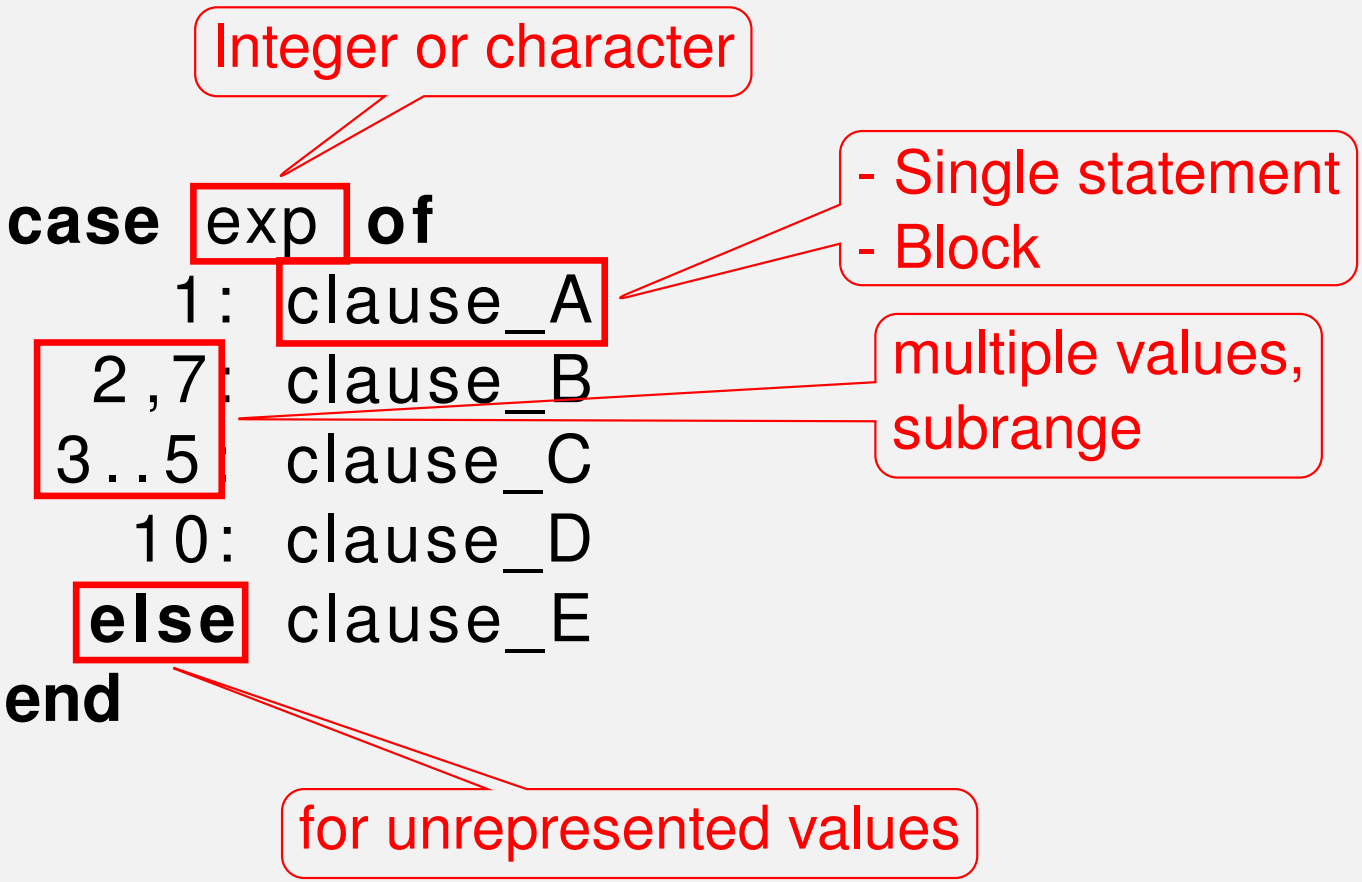
Case Study: C




```
switch (index) {
    case 1:
    case 3: odd += 1;
            sumodd += index;
            break;
    case 2:
    case 4: even += 1;
            sumeven += index;
            break;
    default: printf("Error in switch").
}
```

switch (index): Đây là bắt đầu của câu lệnh switch. Biến index ở đây là cái "selector expression " mà tài liệu nhắc tới. Giá trị của biến này sẽ được so sánh với các trường hợp case bên trong.
case 1: và case 3:: Đây là các trường hợp (case). Nếu giá trị của index bằng 1 hoặc 3, thì đoạn code sau dấu hai chấm (:) sẽ được thực hiện. Trong trường hợp này, nó sẽ thực hiện odd += 1; và sumodd += index;.
break;: Câu lệnh break này rất quan trọng trong switch...case của C. Nó sẽ khiến chương trình thoát ra khỏi câu lệnh switch ngay lập tức sau khi thực hiện xong đoạn code của case. Nếu không có break, chương trình sẽ tiếp tục chạy xuống các case tiếp theo, kể cả khi giá trị của index không khớp. Tài liệu có chú thích là "Multiple segments exited by break" để nhấn mạnh điều này.
case 2: và case 4:: Tương tự như case 1 và case 3, nếu index là 2 hoặc 4, thì even += 1; và sumeven += index; sẽ được thực hiện.
default:: Đây là trường hợp mặc định. Nếu giá trị của index không khớp với bất kỳ case nào ở trên, thì đoạn code sau default: sẽ được thực hiện. Trong ví dụ này, nó sẽ in ra "Error in switch ". Tài liệu có chú thích là "for unrepresented values" để chỉ ra ý nghĩa của default.
Type must be int và Exact value: Chú thích này nói rằng trong C, cái "selector expression" (ví dụ như index) thường phải là kiểu số nguyên (int), và các giá trị trong case phải là các giá trị cụ thể (exact value) chứ không phải là một khoảng giá trị (trừ khi mình dùng nhiều case liên tiếp như case 1: và case 3:).
- Stmt sequences và - Block: Chú thích này nói rằng sau mỗi case (và sau default), mình có thể viết một hoặc nhiều câu lệnh liên tiếp (statement sequences), hoặc mình có thể đặt các câu lệnh đó trong một block (dấu ngoặc nhọn {}). Tuy nhiên, trong ví dụ này thì không có block nào được dùng.

Case Study: Pascal



case exp of: Đây là cách bắt đầu cấu trúc case trong Pascal. exp ở đây là biểu thức điều kiện (tương tự như index trong ví dụ C).
1: clause_A: Nếu giá trị của exp là 1, thì đoạn code clause_A sẽ được thực hiện.
2, 7: clause_B: Nếu giá trị của exp là 2 hoặc 7, thì đoạn code clause_B sẽ được thực hiện. Mày thấy Pascal cho phép mình liệt kê nhiều giá trị cho một trường hợp bằng cách dùng dấu phẩy.
3..5: clause_C: Nếu giá trị của exp nằm trong khoảng từ 3 đến 5 (bao gồm cả 3 và 5), thì đoạn code clause_C sẽ được thực hiện. Pascal hỗ trợ việc chỉ định một khoảng giá trị như thế này.
10: clause_D: Nếu giá trị của exp là 10, thì đoạn code clause_D sẽ được thực hiện.
else clause_E: Tương tự như default trong C, nếu giá trị của exp không khớp với bất kỳ trường hợp nào ở trên, thì đoạn code clause_E sau else sẽ được thực hiện. Tài liệu có chú thích là "for unrepresented values" (cho các giá trị không được biểu diễn).
end: Đây là từ khóa để kết thúc cấu trúc case trong Pascal.
Integer or character: Chú thích này nói rằng trong Pascal, cái biểu thức điều kiện exp thường phải là kiểu số nguyên (Integer) hoặc kiểu ký tự (Character).
- Single statement và - Block: Tương tự như trong C, sau mỗi trường hợp (và sau else), mình có thể viết một câu lệnh đơn (single statement) hoặc một khối lệnh (Block) được bao bởi begin và end (trong Pascal). Trong ví dụ này, clause_A, clause_B, clause_C, clause_D, và clause_E có thể là một câu lệnh hoặc một khối lệnh.

Điểm khác biệt lớn nhất giữa switch...case trong C và case trong Pascal mà mày có thể thấy ở đây là Pascal hỗ trợ việc chỉ định nhiều giá trị hoặc một khoảng giá trị cho một trường hợp, và nó dùng từ khóa else và end thay vì default và dấu ngoặc nhọn. Quan trọng nữa là Pascal không cần dùng break vì sau khi thực hiện xong đoạn code của một trường hợp, nó sẽ tự động thoát ra khỏi cấu trúc case.

Iterative Statements

Câu lệnh lặp là cái mà nó khiến cho một hoặc nhiều câu lệnh được thực hiện lặp đi lặp lại, có thể là không lần nào, một lần, hoặc nhiều lần.

- Cause a statement or collection of statements to be executed zero, one or more times
- Essential for the power of the computer
  - Programs would be huge and inflexible
  - Large amounts of time to write
  - Mammoth amounts of memory to store
- Design questions:
  - How is iteration controlled?
    - Logic, counting
  - Where should the control appear in the loop?
    - Pretest and posttest

How is iteration controlled? (Vòng lặp được điều khiển như thế nào?)
Có thể dùng logic (ví dụ như lặp cho đến khi một điều kiện nào đó đúng), hoặc dùng biến đếm (ví dụ như lặp một số lần nhất định).
Where should the control appear in the loop? (Phần điều khiển nên xuất hiện ở đâu trong vòng lặp?)
Có thể là pretest (kiểm tra điều kiện trước khi thực hiện thân vòng lặp), hoặc posttest (kiểm tra điều kiện sau khi thực hiện thân vòng lặp).

câu lệnh lặp cực kỳ quan trọng, nó là cái làm cho máy tính trở nên mạnh mẽ. Chứ nếu không có lặp, thì chương trình sẽ rất là dài dòng, khó mà thay đổi, tốn nhiều thời gian để viết, và tốn rất nhiều bộ nhớ để lưu trữ.

Counter-Controlled Loops

- Counter-controlled loops must have:
  - Loop variable
  - Initial and terminal values
  - Stepsize

vòng lặp đếm là cái loại vòng lặp mà nó được điều khiển bởi một biến đếm. Để mà hoạt động được thì nó phải có:

Loop variable (Biến vòng lặp): Là cái biến mà nó sẽ thay đổi giá trị qua mỗi lần lặp.
Initial and terminal values (Giá trị đầu và cuối): Là cái giá trị bắt đầu và giá trị kết thúc của cái biến vòng lặp.
Stepsize (Bước nhảy): Là cái giá trị mà biến vòng lặp sẽ tăng hoặc giảm sau mỗi lần lặp.
Ví dụ, trong cái vòng lặp for i := 1 to 10 do ..., thì i là biến vòng lặp, 1 là giá trị đầu, 10 là giá trị cuối, và bước nhảy là 1 (vì nó tự động tăng lên 1 đơn vị sau mỗi lần lặp)

Case Study: Algol-based

một ví dụ về vòng lặp đếm trong một ngôn ngữ lập trình dựa trên Algol (một ngôn ngữ lập trình đời đầu). Nó minh họa cho cái phần "Counter-Controlled Loops"

General form

constant

```
for i := first to last by step
do
    loop body
end
```

Known number of iterations before executing

Semantic

```
[define end_save]
end_save := last
i = first
loop:
if i > end_save goto out
[loop body]
i := i + step
goto loop
out:
[undefine end_save]
```

for i := first to last by step do: Đây là dạng tổng quát của vòng lặp đếm.
i là loop variable (biến vòng lặp).
first là initial value (giá trị đầu).
last là terminal value (giá trị cuối).
step là stepsize (bước nhảy). Nếu by step bị bỏ qua, thì mặc định bước nhảy là 1.
do là bắt đầu phần thân vòng lặp.
loop body: Đây là các câu lệnh sẽ được thực hiện lặp đi lặp lại.
end: Đây là từ khóa để kết thúc vòng lặp.
Semantic: Phần này mô tả ý nghĩa của vòng lặp bằng một đoạn code tương đương:
[define end_save] và [undefine end_save] có thể là cách ngôn ngữ này quản lý một biến tạm để lưu giá trị cuối.
end_save := last gán giá trị cuối cho biến tạm.
i := first gán giá trị đầu cho biến vòng lặp i.
loop: là một nhãn để đánh dấu điểm bắt đầu của vòng lặp.
if i > end_save goto out là điều kiện dừng. Nếu i lớn hơn giá trị cuối, nó sẽ nhảy ra khỏi vòng lặp (goto out).
[loop body] là thực hiện các câu lệnh trong thân vòng lặp.
i := i + step tăng giá trị của biến vòng lặp theo bước nhảy.
goto loop quay lại đầu vòng lặp.
out: là nhãn đánh dấu điểm kết thúc của vòng lặp.
constant: Chú thích này có vẻ muốn nói rằng first, last, và step thường là các giá trị không thay đổi trong suốt quá trình thực hiện vòng lặp.
Known number of iterations before executing: Chú thích này nhấn mạnh rằng với vòng lặp đếm, số lần lặp thường được xác định trước khi vòng lặp bắt đầu.

Case study: C

General form

```
for (expr_1; expr_2; expr_3)
    loop body
```

Can be infinite loop

Semantic

```
        expr_1
      loop:
if expr_2 = 0 goto out
    [loop body]
        expr_3
     goto loop
     out: ...
```

for (expr_1; expr_2; expr_3): Đây là dạng tổng quát của vòng lặp for trong C. Nó có ba phần trong cặp dấu ngoặc đơn, cách nhau bởi dấu chấm phẩy:
expr_1: Thường là để khởi tạo biến vòng lặp (tương ứng với "initial value" trong tài liệu trước). Nó chỉ được thực hiện một lần duy nhất trước khi vòng lặp bắt đầu.
expr_2: Đây là điều kiện lặp (tương ứng với "terminal value" trong tài liệu trước, nhưng ở dạng điều kiện). Vòng lặp sẽ tiếp tục chạy miễn là điều kiện này còn đúng (khác 0 trong C).
expr_3: Thường là để cập nhật giá trị của biến vòng lặp (tương ứng với "stepsize" trong tài liệu trước). Nó được thực hiện sau mỗi lần lặp của thân vòng lặp.
loop body: Đây là các câu lệnh sẽ được thực hiện lặp đi lặp lại miễn là expr_2 còn đúng.
Semantic: Phần này mô tả ý nghĩa của vòng lặp for trong C bằng một đoạn code tương đương:
expr_1 được thực hiện đầu tiên.
loop: là một nhãn đánh dấu điểm bắt đầu của vòng lặp.
if expr_2 == 0 goto out là điều kiện dừng. Nếu expr_2 bằng 0 (tức là sai), nó sẽ nhảy ra khỏi vòng lặp (goto out).
[loop body] là thực hiện các câu lệnh trong thân vòng lặp.
expr_3 được thực hiện sau mỗi lần lặp.
goto loop quay lại đầu vòng lặp.
out: ... là nhãn đánh dấu điểm kết thúc của vòng lặp.
Can be infinite loop: Chú thích này nhắc nhở rằng vòng lặp for trong C có thể trở thành vòng lặp vô hạn nếu cái điều kiện expr_2 không bao giờ trở thành sai (ví dụ như for (;;) {}).

Logically Controlled Loops

vòng lặp điều khiển bằng logic là cái loại vòng lặp mà nó lặp đi lặp lại dựa trên một biểu thức Boolean (đúng/sai) chứ không phải dựa trên một biến đếm như vòng lặp đếm.

Tài liệu cũng nhấn mạnh là vòng lặp điều khiển bằng logic tổng quát hơn vòng lặp đếm. Vì với vòng lặp đếm, mình biết trước được số lần lặp (hoặc có thể tính toán được), còn với vòng lặp logic, mình không biết trước được, nó phụ thuộc vào cái điều kiện Boolean.

- Repeat based on Boolean expression rather than a counter
- Are more general than counter-controlled
- Design issues:
  - Should the control be pretest or posttest?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?

Hai câu hỏi thiết kế quan trọng:

Should the control be pretest or posttest? (Phần điều khiển nên là pretest hay posttest?)
Pretest: Kiểm tra điều kiện trước khi thực hiện thân vòng lặp. Ví dụ: vòng lặp while.
Posttest: Thực hiện thân vòng lặp trước, rồi mới kiểm tra điều kiện. Ví dụ: vòng lặp do...while.
Should the logically controlled loop be a special form of a counting loop or a separate statement? (Vòng lặp logic nên là một dạng đặc biệt của vòng lặp đếm hay là một câu lệnh riêng?)

Forms	Semantics
while (ctrl_expr) loop body	loop: if ctrl_expr is false goto out [loop body] goto loop out:...

---

do loop body **while** (ctrl_expr);	loop: [loop body] if ctrl_expr goto loop

User-Located Loop Control

Cái này là khi mà lập trình viên có thể tự do chọn cái vị trí để điều khiển vòng lặp, chứ không nhất thiết phải là ở đầu hoặc ở cuối vòng lặp.

Để làm được điều này, người ta thường thiết kế vòng lặp theo kiểu "vô hạn" (infinite loops), nhưng mà lại cho phép người dùng chèn thêm các câu lệnh để thoát ra khỏi vòng lặp ở bất kỳ chỗ nào mình muốn.

- Programmer can choose a location for loop control rather than top or bottom
- Simple design: infinite loops but include user-located loop exits
- Languages have exit statements: **break** and **continue**
- A need for restricted goto statement

Các ngôn ngữ lập trình thường cung cấp các câu lệnh như break và continue để thực hiện việc này.

Tài liệu cũng nhắc đến cái câu lệnh goto, nhưng mà theo kiểu "restricted goto" (goto bị hạn chế).

```
while (sum < 1000) {
    getnext(value);
    if (value < 0) break;
    sum += value;
}
```

- What if we replace **break** by **continue**?

thay break bằng continue trong đoạn code đó, thì nó sẽ getNextValue cho đến khi value lớn hơn hoặc bằng 0, rồi sau đó nó mới cộng vào sum, và quá trình này lặp lại cho đến khi sum lớn hơn hoặc bằng 1000, đúng không?

Iteration Based on Data Structures

Cái này là cái kiểu lặp mà nó được điều khiển bởi số lượng phần tử trong một cấu trúc dữ liệu.

Để làm được điều này, người ta thường dùng một cái gọi là "Iterator" (Bộ lặp).

- Controlled by the number of elements in a data structure
- Iterator:
  - Called at the beginning of each iteration
  - Returns an element each time it is called in some specific order
- Pre-defined or user-defined iterator

Iterator: Là một cái gì đó được gọi ở đầu mỗi lần lặp, và nó sẽ trả về một phần tử mỗi khi được gọi, theo thứ tự cụ thể nào đó.
Iterator có thể là "pre-defined" (được định nghĩa sẵn) hoặc "user-defined" (do người dùng tự định nghĩa).

Trong ví dụ này, cái vòng lặp foreach nó dùng một cái iterator để duyệt qua từng phần tử trong cái mảng strList. Mỗi lần lặp, nó lấy ra một cái tên (name) và in ra màn hình.

Nói chung, cái này là một cách lặp khác so với vòng lặp đếm hay vòng lặp logic. Thay vì lặp dựa trên biến đếm hay điều kiện, nó lặp dựa trên các phần tử trong một cấu trúc dữ liệu.

```
String[] strList = {"Bob","Carol","Ted"};
...
foreach (String name in strList)
        Console.WriteLine("Name:{0}", name);
```

Unconditional Branching

"unconditional branch" (nhánh không điều kiện), hay còn gọi là goto, là cái câu lệnh mạnh nhất để điều khiển luồng thực thi của các câu lệnh trong chương trình.
Tuy nhiên, nó cũng rất là nguy hiểm. Nó làm cho code trở nên khó đọc, dẫn đến việc khó hiểu, không đáng tin cậy và tốn kém để bảo trì.

- Unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements
- Dangerous: difficult to read, as the result, highly unreliable and costly to maintain
- Structured programming: say no to goto
- Java, Python, Ruby: no goto
- It still exists in form of loop exit, but they are severely restricted gotos.

Vì vậy, "structured programming" (lập trình có cấu trúc) khuyến cáo là nên nói không với goto.
Các ngôn ngữ như Java, Python, Ruby là không có goto.
Nó vẫn tồn tại dưới dạng các câu lệnh thoát khỏi vòng lặp (như break), nhưng chúng là những cái goto bị hạn chế rất nhiều.
Nói chung, goto là một câu lệnh rất mạnh, nhưng cũng rất nguy hiểm, nên tốt nhất là mình nên tránh dùng nó nếu có thể.

to be continued

Summary

Expressions (Biểu thức)
Operator precedence and associativity (Độ ưu tiên và tính kết hợp của toán tử)
Side effects (Tác dụng phụ)
Statements (Câu lệnh)
Assignment (Gán)
Selection Statement (Câu lệnh lựa chọn)
Loop structures (Cấu trúc lặp)

- Expressions
  - Operator precedence and associativity
  - Side effects
- Statements
  - Assignment
  - Selection Statement
  - Loop structures