

Control Flow = ^{ลำดับ} Order ของ Execution

Ordering of what should be done in program execution

- Sequencing: implicit ordering from top to bottom
- Selection: choice is made among two or more statements
- Iteration: program fragment executed repeatedly
- Procedural abstraction: collection of control constructs encapsulated in a single unit
- Recursion: self-referential subroutines

↳ วิธีการ execute parameter ใน method

Expression

เลือกของ Execution
→ สิ่งที่เราจะได้ผลลัพธ์เป็นค่าใดค่าหนึ่ง

operator = ตัวดำเนินการ
(+, -, *, ÷)

Expression produces a value, i.e. literal constant, named variable, constant, or operator (or function) applied to operands (or arguments)

↪ เช่น $x = a + b$ ↪ expression มาวางกันด้วย operator

A language may specify the location of function name.

- Prefix: before arguments, e.g. $(+ 1 3)$ 2) in Lisp
- Infix: among arguments, e.g. $1 + 3$ in most imperative languages
- Postfix: after arguments, e.g. post-increment/decrement $(++ \text{ and } --)$ in C and its descendants

Most imperative languages use infix notation for binary operators, and prefix notation for many operators and other functions.

γ = priority (เครื่องหมายอะไรสำคัญกว่า) ถ้าไม่วงเล็บทำตามวงเล็บ

Precedence and Associativity

ทำจากที่ใดไปเกิดใหม่ (ปกติคือ ซ้ายไปขวา)

When operators are written in infix notation without parentheses, ambiguity arises as to what is an operand of what, e.g.

$a+b*c**d**e/f$ should be evaluated as

$((((a+b)*c)**d)**e)/f$ or

$a+(((b*c)**d)**(e/f))$ or

$a+((b*(c**(d**e)))/f)$?

Precedence says that certain operators, in the absence of parentheses, group more tightly than other operators, e.g.

$a-b*c$ is $a-(b*c)$

Associativity says sequences of operators of equal precedence, in the absence of parentheses, group to the left or to the right, e.g.

$a-b+c$ is $(a-b)+c$

Precedence

Operators at the top group most tightly. = *ทำงานก่อน*

In most languages , multiplication and division group more tightly than addition and subtraction.

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	*, / (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /=, <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		(bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		(logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %= >>=, <<=, &=, ^=, = (assignment)	
		, (sequencing)	

Associativity

Basic arithmetic operators almost always associate left-to-right.

9-3-2 is 4, not 8 because (9-3)-2

Exponentiation usually associates right-to-left.

4**3**2 is 4**(3**2)

Assignment associates right-to-left.

a = b = a+c is (a+c) assigned to b, then the same value assigned to a
become equal to

Exercise: Precedence and Associativity

- Given the precedence table and associativity rules in the previous slides,
- Apply parentheses to the expression to show how operands are grouped to operators and
 - Give the result of the expression
 - Where $a = 1, b = 2, c = 3, d = 2, e = 2, f = 3$

Fortran

$a + ((b * (c ** (d ** e))) / f)$

Pascal

$a < (b \text{ and } c) < d$

C

$(a < b) \&\& (c < d)$

←
ภาษา Fortran มี Association จากขวาไปซ้าย (สำหรับ **)

result is $a + (b * (c ** (d ** e)) / f$

เลขได้ทำก่อน

and มี priority สูงกว่า <, >

result is **Error** static semantic error

and มีที่สั้น

result is **False**

ซ้ายทำก่อนแล้วค่อยขวา

Evaluation Order within Expression (1)

Precedence and associativity **do not** specify the order in which the operands of a given operator are evaluated, e.g.

Precedence and associativity say $a - f(b) - c * d$ is $(a - f(b)) - (c * d)$

Which one is evaluated first, $(a - f(b))$ or $(c * d)$?

MM
อาจไม่ได้
รู้ก่อน

Similarly, in $f(a, g(b), h(c))$, what is the order in which the arguments will be evaluated?

But evaluation order is important.

- Impact on expression result via **side effect**

- What if $f(b)$ modifies c and/or d ?
- What if $g(b)$ modifies a and/or c ?

- Impact on code improvement

- In $a * b + f(c)$, for example, it might be desirable to call f first, because the product $a * b$ stored in a register would need to be saved (on stack) during the call to f (i.e. run time cost) as f might want to use all registers.

ถ้า $a * b$ ทำก่อน ต้องมี register เก็บค่าไว้ (ตัว $f(c)$ ก็ดันไป register อื่น!)
ถ้าทำ $f(c)$ ก่อน จะดีกว่า เพราะไม่ต้องเก็บอะไรใน bookkeeping

ข้อเสีย : คนเขียนอาจงง

Evaluation Order within Expression (2)

As for code improvement, most languages then leave the order of evaluation undefined, i.e. compiler can choose whatever order that results in faster code.

Be careful when writing expression in which side effect of evaluating one operand or argument can affect the value of another, e.g. use parentheses to impose ordering.

อาจไม่เร็วมาก แต่คนเขียนไม่งงแน่นอน

But Java and C# require left-to-right evaluation (i.e. cleaner semantics over run time cost).

Exercise: Precedence, Associativity, Evaluation Order

Given the precedence table and associativity rules in the previous slides, and evaluation order within expression is left to right, what is the result of this C program?

```
int give2() { printf("two\n"); return 2; }
```

```
int give3() { printf("three\n"); return 3; }
```

```
int give4() { printf("four\n"); return 4; }
```

```
int main() {
```

```
    printf("%d\n", give4() + (give2() * give3()) - (give4() / give2()));
```

```
    return 0;
```

```
}
```

บวก, ลบ จากซ้ายไปขวา

4 + 6 - 2 = 8



Assignment

เครื่องหมาย =
→ เปลี่ยนค่าที่ได้ในหัวแปร

In imperative language, assignment provides the means to make the changes to the values of variable in memory.

Assignment takes two arguments.

- A **value**
- A **reference to a variable** into which the value should be placed.

Assignment has a **side effect** i.e. it changes the value of a variable, thereby affecting the result of any later computation in which the variable appears.

```
//C
int max(int x, int y) {
    if (x > y) {return x;} else {return y;}
}
int main()
{
    int a, b;
    a = 1; b = 2;
    printf("max is %d\n", max(a, b)); //max is 2
    a = 3;
    printf("max is %d\n", max(a, b)); //max is 3
    return 0;
}
```

หมายเหตุ: การที่ตัวแปรเปลี่ยนค่า พอเรียกฟังก์ชันเดิมค่าเปลี่ยน

```
--Haskell has no assignment and no side effect
a, b :: Int
a = 1
b = 2
max a b
--2
max a b
--2
```

ค่าผ่านตลอดไป (ไม่เปลี่ยนแปลง)

Semantics of Assignment (1)

In **value model of variables**, a variable is a named container for a value (e.g. Pascal, C, Java's built-in type, PHP).

A variable has two interpretations when used with assignment.

- **l-value** refers to expression that denotes location.
- **r-value** refers to expression that denotes value.

```
//C
b = 2;           // l-value of b is used
c = b;           // r-value of b is used, l-value of c is used
a = b+c;         // r-values of b and c are used, l-value of a is used
```

a 4

b 2

c 2

$a = b$
(ค่าที่ส่งมาเก็บ)
คือถือว่าเป็น "value" ที่มาใส่ใน a

ใน address นั้นเก็บค่าเลข

แบบนี้จะกำหนดได้
ตัวแปรแต่ละตัวเป็น
Container ที่อิสระจากกัน

Semantics of Assignment (2)

$a = b$
 l value → a l value → b

In **reference model of variables**, a variable is a named reference to a value (e.g. Clu, Lisp, Haskell, Smalltalk, Java's user-defined type (class), Python, Ruby).

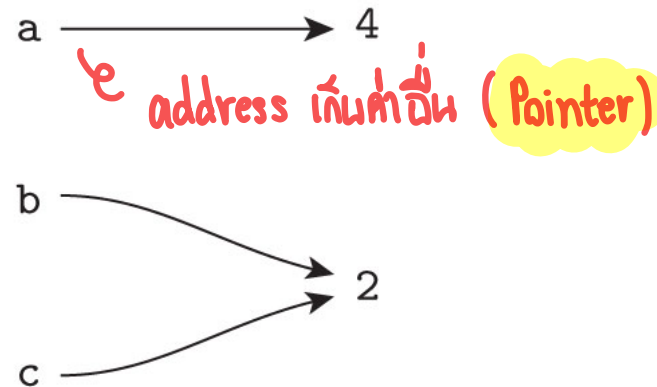
- Every variable is an l-value.
- When a variable appears where an **r-value** is expected, it must be dereferenced to obtain the value to which it refers (automatic in most languages).

คล้ายกับ *it
 ตาม pointer ไปหาค่า แล้วเอาค่าออกมา

```
%Clu
b := 2;           % l-value of b is used
c := b;           % l-value of b is dereferenced, l-value of c is used
a := b+c;         % l-values of b and c are dereferenced, l-value of a is used
                  % 2 and 4 are immutable values at some locations to which any variables can refer
```

dereference
 $a := 2 + 2 = 4$
 ไม่dereference จะ error
 โปรแกรมเมอร์ต้องรู้ว่าจะdereference

** เมื่อไหร่ก็ตามที่ตัวแปรอยู่ด้านขวา
 จะมีการdereference ก่อน



	value model	reference model
$a = 4$ $b = 4$ $b += 1$	$a \rightarrow [4]$ $b \rightarrow [4]$ (b's value is updated to 5)	$a \rightarrow [4]$ $b \rightarrow [5]$ (b's pointer is updated to point to a new heap location)

Stack เก็บ Heap เก็บ

Short-Circuit Evaluation

Consider these logical expressions,

$(a < b)$ and $(b < c)$

$(a > b)$ or $(b > c)$

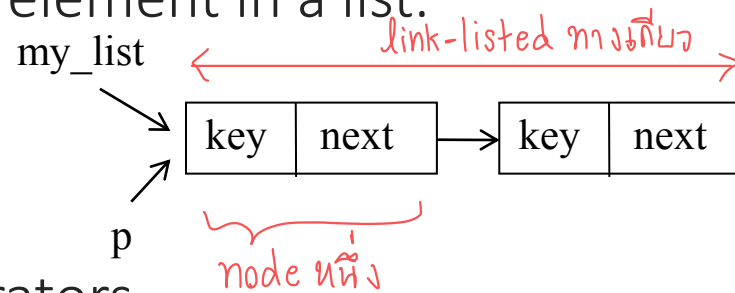
↓
เช่นถ้าเป็น $F \wedge \text{ไม่ต้องทำ}$ = F หมด
↑
ไม่ต้องทำ

When the overall value of these expressions can be determined from the first half of the computation, compiler will generate code that skips the second half.

This saves time.

Short-Circuit Changes Semantics of Boolean Expressions

An example of a search for an element in a list.



C short-circuits its logical operators.

```
//C
p = my_list;
while (p && p->key != val)
    p = p->next;
```

Handwritten red text: *ถ้า p เป็น null Error*
Handwritten orange text: *ไม่ผ่าน Test ถ้า p เป็น null*

Handwritten red text: *ไม่ Short-circuit*

Pascal does not short-circuit. Both `<>` will be evaluated before and, so run-time semantic error if `p` is nil (unsuccessful search).

```
(* Pascal *)
p := my_list;
while (p <> nil) and (p^.key <> val) do (*ouch!*)
    p := p^.next;
```

Handwritten red text: *error ถ้าเป็น null*

Exercise: Short-Circuit

How can we use short-circuit evaluation to make the following code safer?

```
const int MAX = 10;
```

```
int A[MAX];
```

```
...
```

```
if (A[i] > foo) ...
```

\downarrow
 $0 \leq i \ \&\& \ i \leq \text{MAX}-1 \ \&\&$

→ out of bound $\frac{1}{n}$

```
if (n/d < threshold) ...
```

\downarrow
 $(d \neq 0) \ \&\&$

Sequencing

ซ้ายไปขวา, บนลงล่าง

It is the principal means of controlling the order in which side effects occur.

When one statement follows another, the first statement executes before the second.

Sequence of statements can be enclosed as a compound statement (block), e.g.
begin...end or {...}.

Selection

Most languages employ variant of if ... then ... else ...

if *condition* then *statement*

else if *condition* then *statement*

else if *condition* then *statement*

...

else *statement*

เก็บ
ค่าผลลัพธ์ใน register

Short-Circuited Condition in Selection

In languages with short-circuit, compilers generate target code which evaluates the conditions for branching control to various locations without having to store any boolean values.

```
if ((A > B) and (C > D)) or (E ≠ F) then
    then_clause
else
    else_clause
```

```
    r1 := A
    r2 := B
    if r1 <= r2 goto L4
    r1 := C
    r2 := D
    if r1 > r2 goto L1
L4: r1 := E
    r2 := F
    if r1 = r2 goto L2
L1: then_clause
    goto L3
L2: else_clause
L3:
```

Nested If

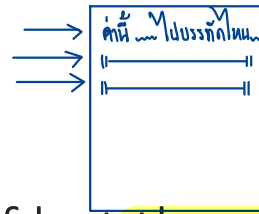
Compilers generate target code which tests each expression sequentially.

```
--Ada  
  
i := ... -- calculate tested expression  
if i = 1 then  
    clause_A  
elsif i = 2 or i = 7 then  
    clause_B  
elsif i in 3..5 then  
    clause_C  
elsif i = 10 then  
    clause_D  
else  
    clause_E  
end if;
```

```
r1 := ... -- calculate tested expression  
if r1 ≠ 1 goto L1 ถ้าไม่เท่ากับ 1 ก็ต้อง jump ไป L1  
    clause_A  
    goto L6  
L1: if r1 = 2 goto L2 ถ้าไม่ถูก  
    if r1 ≠ 7 goto L3 เงื่อนไข  
    L2: clause_B  
        goto L6  
    L3: if r1 < 3 goto L4  
        if r1 > 5 goto L4 ไม่ถูก ก็เงื่อนไข  
        clause_C  
        goto L6  
    L4: if r1 ≠ 10 goto L5 เงื่อนไขไม่ผ่าน Step  
        clause_D  
        goto L6  
    L5: clause_E  
    L6:
```

Case/Switch Statements

มีการสร้าง jump table



สามารถรู้เลยว่าต้องรันบรรทัดไหน
(ไม่ต้องผ่านการทดสอบบรรทัดๆ)

Less verbose syntactically than nested if but the principal motivation is to facilitate the generation of efficient target code.

-- General form

--Ada with case labels and arms

```
case ... --calculate tested expression
is
  when 1      => clause_A
  when 2 | 7  => clause_B
  when 3..5   => clause_C
  when 10     => clause_D
  when others => clause_E
end case;
```

```
goto L6 --jump to code to compute address
L1: clause_A
goto L7
L2: clause_B
goto L7
L3: clause_C
goto L7
...
L4: clause_D
goto L7
L5: clause_E
goto L7

L6: r1 := ... --computed target of branch
goto *r1
L7:
```

If-else ในเชิงทฤษฎี

Case/Switch Implementation Example

เปลี่ยนพ.ท.
แม้เวลาจริง ←

L6 checks boundary and fetches corresponding entry from the table and branches to it. So finding the correct arm is in constant time.

$$154 \quad r_1 = 10$$

execute & L4

$$r_1 = T[a]$$

The diagram illustrates a jump table and the execution flow of a program. At the top, a box labeled "Address at label L1" points to the first entry in the jump table. The jump table is a list of addresses and their corresponding indices:

Address	Index
&L1	0
&L2	1
&L3	2
&L3	3
&L3	4
&L5	5
&L2	6
&L5	7
&L5	8
&L4	9

A red bracket on the right side of the jump table is labeled "jump table". Below the jump table, the code execution flow is shown:

```

T:  -- tested expression = 1
    &L1  0
    &L2  1
    &L3  2
    &L3  3
    &L3  4
    &L5  5
    &L2  6
    &L5  7
    &L5  8
    &L4  9
L6: r1 := ...
    if r1 < 1 goto L5
    if r1 > 10 goto L5
    r1 -= 1
    r1 := T[r1]
    goto *r1
L7:
  
```

Annotations in the code include:

- "-- tested expression = 1" next to the first line of the jump table.
- "-- tested expression = 10" next to the entry for &L4 (index 9).
- "-- calculate tested expression" next to the line `r1 := ...`.
- "-- L5 is the 'else' arm" next to the `goto L5` lines.
- A red box around `r1 -= 1` with a red arrow pointing to it labeled `r1 -= 1`.
- A red arrow pointing from `r1 := T[r1]` to the jump table, labeled `r: map as index`.
- A red arrow pointing from `goto *r1` to the box labeled "content of r1".

Exercise: Case/Switch Implementation

What is the problem with jump table implementation in the previous slide?

ข้อผิดพลาดในกรณีนี้

What do you think a compiler should do about it?

If-else

Switch Statement with Fall-Through

โดยหาก case ใดๆ มีค่าตรงกับที่สนใจ ก็จะมีการ fall through ไปยังทุกเคส
ที่อยู่ถัดจาก case นี้ไปเรื่อย ๆ
จนกว่าจะมีการ break

Found in C and retained in C++, Java.

- Each possible value for tested expression must have its own label.
- A label with empty arm falls through into the code of subsequent label.
- To get out of a switch, a break statement must be used at the end of an arm, rather than falling through into the next.

```
switch (... /*tested expression */) {  
    case 1: clause_A;  
           break;  
  
    case 2:  
    case 7: clause_B;  
           break;  
  
    case 3:  
    case 4:  
    case 5: clause_C;  
           break;  
    case 10: clause_D;  
            break;  
    default: clause_E;  
            break; }  
        
```

break; จบจาก switch statement

Iteration

In most languages, iteration takes the form of loops.

An enumeration controlled loop executes over values in a given finite set.

A logically controlled loop executes until some boolean condition changes value.

An iterator iterates over elements of any well-defined set (collection).

while loop

Enumeration-Controlled Loop

การวนลูปแบบ enumeration controlled เป็นการวนลูปในจำนวนของค่าที่มีการจำกัดขอบเขตไว้ เช่น มีการกำหนดค่าเริ่มต้นและสิ้นสุด

การวนลูปแบบ logically controlled เป็นการวนลูปที่ขึ้นอยู่กับสถานะของค่าบูลีนค่าหนึ่งจนกว่าจะมีการเปลี่ยนแปลง

Boolean

Test for empty bounds first, i.e. test terminating condition before the first iteration.

```
(* Modula-2: enumeration-controlled *)
```

```
FOR i := first TO last BY step DO
```

```
...
```

```
END
```

```
/* C: combination of enumeration- and logically-controlled */
```

```
for (i = first; i <= last; i += step) {
```

```
...
```

```
}
```

pure (เพราะไม่มี boolean condition เลย)

เป็นแบบผสมกัน

```
r1 := first
```

```
r2 := step
```

```
r3 := last
```

```
goto L2
```

```
L1: ...
```

```
    r1 := r1 + r2
```

```
L2: if r1 <= r3 goto L1
```

-- loop body; use r1 for i

ตรวจสอบสถานะการวน "ก่อน" วนรอบใหม่

Logically Controlled Loop

Pre-test loop: Loop body may not be executed

```
while condition do statement
```

Post-test loop: Loop body is executed at least once.

๖ ทำอย่างน้อย 1 ครั้ง

๖
ตรวจสอบสถานะการวน "หลังจากประมวลผล
Block ที่วนซ้ำไปแล้ว 1 รอบ"

```
//C  
do {  
    line = read_line(stdin);  
} while line[0] != '$';
```

Mid-test loop: A special statement is nested inside a test for terminating condition.

๖
เช่น if เพื่อ
เช็คการจบ

```
//C  
for (;;) {  
    line = read_line(stdin);  
    if (all_blanks(line)) break;  
    consume_line(line);  
}
```

Iterator (1)

โดย true iterator หรือ iterator แท้ ก็คือตัว collection นั้นเป็น iterator ด้วยสามารถประมวลผลวนซ้ำแบบ iteration

ได้โดยตรง ซึ่งตัวอย่างการใช้ iterator แบบนี้ คือ ภาษา Python โดยตัวอย่างแรกเป็นดังนี้

True iterator (e.g. Clu, Python, Ruby, C#)

- Any container abstraction provides an iterator that enumerates its items.

```
#Python
#Iterator goes unseen as it is implicitly used
for i in [1, 2, 3]:
    print(i)
...
1
2
3

#range(first, last, step) is a built-in iterator.
#It yields integers in the range in increments of step, but not including last.
#It is a function but, when called each time, continues where it last left off, giving next integer.

my_list = ['one', 'two', 'three', 'four', 'five']
my_list_len = len(my_list)
for i in range(0, my_list_len, 2):
    print(my_list[i])
...
one
three
five
```

↪ เป็นการใช้ iterator เหมือนกัน (จับคู่เพียงครั้ง)

↪ เป็นการใช้ iterator แบบแบบๆ

↪ เป็น enumerate

True iterator

Iterator (2)

Iterator as an ordinary object (e.g. C++, Java, Ada, Python).

- Provides methods for initialization, generation of the next index value, and testing for completion.

for(iterator it = v.begin(); it != v.end; it++)
true iterator

```
//Java
ArrayList al = new ArrayList();
//add elements to the array list
al.add("C");
al.add("A");
al.add("E");

//use iterator to display contents of al
System.out.print("Contents of al: ");
Iterator itr = al.iterator();

while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
...
Contents of al: C A E
```

คนเขียนต้องรู้ที่ใช้ตรงๆ

Recursion

Stack frame เยอะ (เพราะตัวมันเองเรียกไม่เสร็จก็ต้องเรียกต่อ)

Functions calling themselves.

Functions calling other functions that call them back in turn.

- * Any iterative algorithm can be rewritten as a recursive algorithm and vice versa. Which to use in which circumstance is mainly a matter of taste.

๒ นกกลับ

Recursion and Iteration

$\text{gcd}(a, b)$ positive integers, a, b	{	a	if $a=b$
		$\text{gcd}(a-b, b)$	if $a>b$
		$\text{gcd}(a, b-a)$	if $b>a$

```
//C
//Iteration, assume a, b > 0
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a-b;
        else b = b-a;
    }
    return a;
}
```

```
//C
//Recursion, assume a, b > 0
int gcd(int a, int b) {
    if (a == b) return a;    //base case
    else if (a > b) return gcd (a-b, b);
    else return gcd(a, b-a);
}
```