Chapter 6 Control Flow

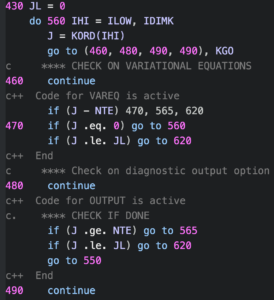
Focus on variable, expression, statement, and control structure…

Control Flow in machine/assembly

* Unstructured: jump/goto (conditional or unconditional branching)

Control flow in HL

Used to be unstructured: goto, break, continue,

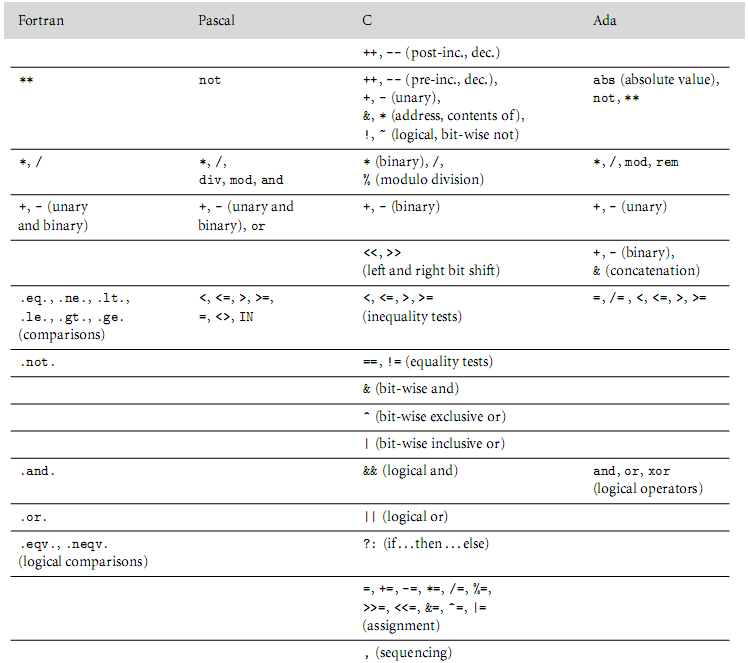


[Goto considered harmful](https://degenerateconic.com/goto-still-considered-harmful.html)

Structured:

* + Sequencing
  + Selection
  + Iteration
  + Procedural Abstraction
  + Recursion
  + Concurrency
  + Exception Handling and Speculation

How many operators are needed for flow control?

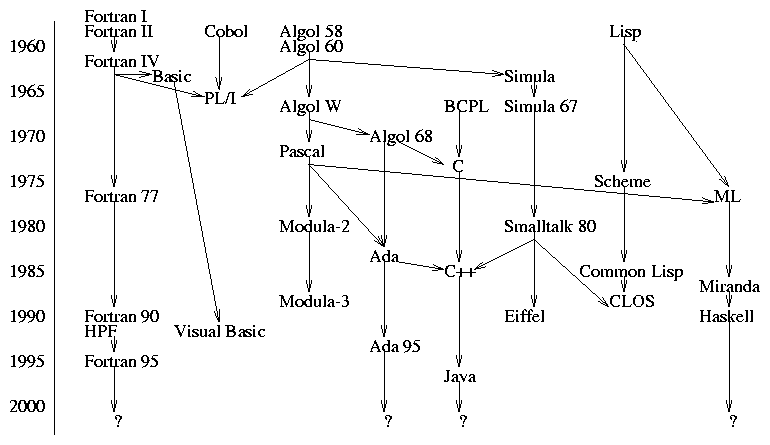


**Figure 6.1 – Operator precedence**

6.1 Expression Evaluation

Key issues

* Choice of expression notation
  + Infix, prefix, etc.
* Precedence & associativity rules and levels (see fig 6.1)
* C has 15 levels - too many to remember, could leads to confusion.,
* Pascal has 3 levels - too few?
* Fortran has 8
* Ada has 6, notice that a*nd, or* at same level
* When in doubt, use ()



---------------------------------- Beginning of brief History of C and C++ --------------------------------

* A descendent of ALGOL
* No significant new features,
* Used widely, especially in system programming

Historical Background of C and UNIX – some [background](https://www.youtube.com/watch?v=JoVQTPbD6UY)

1. In 1969, **Ken Thompson**, along with **Dennis Ritchie** and others, wrote the 1st UNIX OS for PDP-7 (with 8k memory), in assembly language. The OS has a primitive kernel, an assembler, an editor, and a simple shell.
2. In 1970, Ken Thompson implemented the 1st HL language on the UNIX system: B, based on BCPL (Martin Richard, 1967). B was constrained by memory limitation on PDP-7.

Characteristics of B:

Is an un-typed language

(memory cell is the only data type, pointers are integer to memory cells, etc.), causing complications.

1. In 1971, Dennis Ritchie extended B by creating int and char type. The new language is called NB.
2. In 1972, Dennis Ritchie created C, based on B and ALOGL 68. The only standard before 1989 is the [C book by Kernighan and Ritchie.](https://www.thriftbooks.com/w/the-c-programming-language-by-brian-w-kernighan-dennis-m-ritchie/248846/item/244520/?gclid=CjwKCAiAvK2bBhB8EiwAZUbP1ABqwSF23L46oWAQopourD4pdgePH2HGEp6HqVNJXBhOGGLghDK2OhoCsFwQAvD_BwE#idiq=244520&edition=3329143)
3. In 1974, UNIX V4 was rewritten in C, by Ken Thompson and Dennis Ritchie.
4. In 1976, UNIX V6 was made available to the public.
5. 1983, System V was released by Bell.
6. 1984, BSD 4.2 was released by UC Berkeley and used by many universities and corporations.
7. 1989, SVR4 was released by Bell.
8. In 1991, [Linus Torvalds](http://www.catb.org/~esr/faqs/linus/) released source code of Linux, a freeware kernel for PC.
9. 1994, Linux version 1.0 was released.
10. In 1980, [Bjarne Stroustrup](https://www.youtube.com/watch?v=86xWVb4XIyE&list=RDLV5An1sNznblQ&index=9) *created C with class.*

Design goal:

* + Facilitate simulation program.
  + As efficient as C.

Newfeatures:

* Type checking and conversion of function arguments
* Class concept borrowed SIMULA 67 and Smalltalk, with derived classes, friend functions, constructors, etc.

1. In 1984, extended *C with Class* to include virtual function, function/operator overloading, etc.
2. In 1985, 1st implementation of C++ (release 1.0) appeared, called Cfront.

Translate (by Cfront)

**C**

**Program**

##### C++ program

1. In 1989, Release 2.0 appeared, with multiple inheritance, etc.

…

\_\_\_\_\_\_\_\_\_------------------- end of C C++ Unix linux history --------------------------

How many features should a language provide?

Orthogonality & Algol 68 (p. 228)

[Orthogonality](http://en.wikipedia.org/wiki/Orthogonality)

* separation of specific features of a system
* Language features can be used in any combination, and they all make sense.
  + Features are not (heavily?) overlapping
* A main design principle of Algol

--------------------------A brief history of [Algol](https://www.youtube.com/watch?v=nUdqQ5gzzV8) ---------------------------------------------

The first step towards sophistication: [ALGOL 60 (ALGOrithmic Language)](http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html)

* Designed for scientific computing, but was also intended as a universal language
* Designed by a committee
* Syntax was formally described using the Backus-Naur Form (BNF, CFG)
* Considered a descendant of Fortran
* Generalize many Fortran features and added new features

New features in ALGOL 58 (not yet available in Fortran at the time):

* Explicit data type
* Compound statement

New features in ALGOL 60:

* Block structure
* Parameter passing by name and value
* Recursive procedure
* Stack-dynamic array

ALGOL evaluation



The Good:

* First machine-independent language
* Widely used to describe algorithms.
* Has many descendants, such as Pascal and C.
* BNF was adopted as a language description tool.

The bad:

* BNF was complex, making it hard to understand, at least initially
* Lack of I/O facility and portability

The ugly:

* No significant use in real applications in US.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_End of Algol History\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[Side Effect](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) – a [function](https://en.wikipedia.org/wiki/Subroutine) or [expression](https://en.wikipedia.org/wiki/Expression_(programming)) is said to have a **side effect** if it modifies the machine [state](https://en.wikipedia.org/wiki/State_(computer_science)) or has an *unexpected and observable* interaction with calling functions or the outside world.

* + For a function, it’s a *noticeable effect* of a function call *other than return value*, ex., changes value of global var, static var, parent var, caller var, reference argument, exception
  + Assignment statements provide the most typical example of side effects. But, assignments are EVERYWHERE, so, are side-effect everywhere?
    - “A programming construct (ex., assignment) is said to have side effect if it influences subsequent computation in any way other than by returning a value for uses in surrounding context.” (page 225)
  + It’s greatly reduced with the use of modules
  + In (pure) functional, logic, and dataflow languages, there are no such changes
    - These languages are called SINGLE-ASSIGNMENT languages
    - Scheme: [let](http://en.wikipedia.org/wiki/Scheme_(programming_language)) and do defines local variable with static (textual) scope, thus are NOT considered side effects
  + A language could outlaw side effects for functions, benefit:
    - easier to prove things about programs
    - closer to Mathematical intuition
    - easier to optimize
    - (often) easier to understand
  + Side effects are a particular problem if they affect state used in other parts of the expression in which a function call appears, ex., exp2.cpp
  + No side effort 🡪 order of execution irrelevant 🡪 better optimization
    - Compiler is better at optimization than us 🡪 make your program more abstract, less tricky
* Solutions to the side effect problem:

1. Disallows function side effect

Drawback:

1. Limit the programmer
2. Cannot be done easily
3. Operands in an expression must be evaluated in a particular order

Drawback: Limit the compiler

1. Function calls are legal only if the functions don’t change value of other operands in the same expression - Fortran’s solution, a good start
   * 1. Short-circuiting

**The result of an (logical) expression can be determined without evaluating all of its (AND/OR) components?**

Example:

*// The following C++ code reads and stores test scores in an array until either*

*// end of file or there is no more space in the array*

*int A[max], count = 0;*

*while (count<max && fin >> A[count++]); // does this work? yes*

If (p && p->val==target)

…

The idea: when count==max, cin >> A[count++] will not be evaluated, making use of the short circuit feature in C++.

* PASCAL and ADA does not support short circuit evaluations. As a result, the following code:

index := 1;

while ((index <=max) and (A[index] <> target) do

…

cause a “subscript out of range” error

Solutions to the problem:

* ADA allows the programmer to specify short circuit evaluation using *and then*, or *else*, for example:

index := 1;

while (index <=max) *and then* (A[index] /= target)

…

* In PASCAL, the while loop must be carefully modified:

Boolean found=false;

while ((index <=max) and (not found)) do

Begin

if (A[index]=target)

found := true;

else

…

end

* + 1. Variables as values vs. variables as references
  + value-oriented languages – a variable is a container for the value
    - C, Pascal, Ada
      * In C, each variable has an l-value (location) and an r-value (value)
      * LHS of an assignment could be an expression that yields an address

Ex., (f(a)+3) ->b[c] = 2;

* + reference-oriented languages – every variable is an l-value (address aka pointer)
    - When a value is expected from a variable (ex., appear on the RHS of an assignment), it must be *dereferenced*
    - most functional languages such as lisp and Scheme are reference-oriented, where dereferencing is automatic

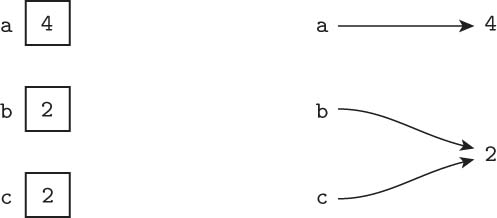
Ex.,

b := 2;

c := b;

a := b+c;

container reference



* + Java is deliberately in-between
    - built-in types are values
    - user-defined types are objects - references
* Expression-oriented vs. statement-oriented languages
  + expression-oriented – every construct is an expression, thus yields value
    - Algol-68

Ex.,

Begin

a := if b<c then d else e

a := begin f(b); g(c) end

g(d);

2+3;

End

* + - functional languages: Lisp, Scheme, ML

ex.,

(define f

(lambda …

exp)

)

* + statement-oriented:
    - most imperative languages
  + C halfway in-between
    - Mostly statement oriented

Ex.,

bool f ()

{

Exp // this exp is not returned or assigned to anything

// explicit return statement required

}

* + - Certain type of statements are expression-like, in that they returns value too, such as:
    - With the exception of some construct

Ex., a = (b>c)? : d : e; /\* expression-oriented \*/

a = b = c = 41;

6.1.2 Assignment statement

* General syntax:

<targetVariable> <assignmentOperator> <expression>

* l-valueOfVariable 🡨 valueOfExpression

Design issue 1: What symbol is used for the <assignmentOperator>?

1. Fortran, Basic, C, C++, PL/1, and Java use =

PL/1 and BASIC also uses = for equal comparison, for example, in PL/1

A = B = C // A gets the Boolean result of relational operation B = C, weird, isn’t it?

C, C++, Java uses = = for equal comparison

1. ALGOL 60, PASCAL, and ADA use :=
2. APL uses 🡨 for assignment, why? It is never confused with equal comparison

BTW, what does APL standard for?

Design issue 2: What (if any) alternative assignment statements are provided?

1. Multiple Targets - The same expression value is assigned to more than one location. For example,

PL/1:

sum, count=0

C, C++:

sum=count=0;

1. Conditional Target - The Same value is assigned to one of the variables based on a condition. For example, In C, C++, Java:

flag ? count1 : count2 = 0;

// is equivalent to

if (flag)

count1=0;

else

count2=0;

1. Compound Assignment Operators - Shorthand for commonly used assignment

For example,

C++ and JAVA:

sum += value;

1. Unary Assignment Operators - Abbreviated Assignments.

For example, in C++ and Java: ++ (increment) and -- (decrement).

// ++ and -- can either be prefix or postfix. For example,

sum += a[index++]; // postfix, it is equivalent to

sum = sum+a[index];

index = index + 1;

Unary operators are right associative (i.e., evaluated right to left). For example,

int m, n=5;

m = - ++n; // Is this valid in C++?

// what value, if any, will m get?

1. Assignment as an expression

In C, C++, and Java (yes), = is treated like a normal binary operator.

The value assigned to the variable and assignment statement returns a reference to the left side variable. This is convenient in many cases. For example,

int A= B= 4; // A and B both get 4

/ \* C example \*/

while ((ch=getchar()) != EOF) /\* read characters until End Of File \*/

{ … }

/\* C++ \*/

while (fin.get(ch)) // while (fin >> n) { … }

{ .. }

// Does this look familiar?

{

…

return s[x] = find (s[x]); // path compression in find

}

Drawback of assignment as expression: Loss of error detection during compilation.

Ex., if (A=B) // the condition always yields B’s value – a common mistake

Is expression-oriented nature of assignment (=) to blame for equality mistaken as assignment error?

Design issue 3: Are Mixed-Mode Assignment allowed?

1. Fortran, C, and C++ use [coercion](https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Coercion) (i.e., implicit type conversion)

For example,

char letterGrade = 65.9; // letterGrade gets \_\_\_’A’\_\_\_\_

Coercion takes place AFTER the right-side expression has been evaluated.

int sum=999, count=100;

float avg = sum/count; // avg gets \_\_\_\_\_\_\_\_\_

1. Java allows mixed-mode assignment only if coercion is widening. For example,

int a; float b;

b=a; // OK in Java

a=b; // illegal in Java

6.3 Sequencing

* Specifies a linear ordering on statements, one statement follows another
* Central to imperative programming

6.4 Selection – do one thing but not the other(s)

* Provides the means of choosing two or more paths in a program based on result(s) of its conditional part(s) (also called selector).
* It is either two-way or multi-way.

1. one-way selection statement
   * 1. All prolog conditional statements are one-way if statements

Ex., grandmother(X,Y) :- mother(X, Z), parent(Z, Y).

* + 1. Fortran IV had only using the form:

IF (Boolean Expression) statement.

Two-way selection can be simulated:

Ex.,

IF (Boolean Expression) Then

True Statements

Else

False Statements

…

Other statements

IF (Boolean Expression) Goto 20

False Statements

Goto 30

1. True Statements

30 Other Statements

Goto was unstructured and was considered [harmful](http://www.wordyard.com/2006/10/10/dijkstra-goto/), but was necessary in this situation.

1. Two-way Selection Statements

Design issue 1: What if only one branch is needed?

* Most languages treat it as a special case of the two-way selection. Ex.,

{ PASCAL Example }

if Boolean\_Expression Then

Begin

….

End

Design issue 2: How are *then* and *else* clauses specified

* Most languages allows for single statement, either simple or compound (C++)
* Some languages (e.g., perl) require all then and else clauses be compound statements.

Advantage: avoid ambiguity, see example below

// Consider the C++ statements:

int sum=0, count=1, result=100;

if (sum==0)

if (count==0)

result = 0;

else

result = 1;

// What is the value of the *result* at this point?

What semantic rule should we use?

Design issue 3: How should the meaning of nested selectors be specified?

* In some languages (e.g., C, C++, Java, Pascal), the rule states that the *else* clause will match with closest unmatched *if*. Alternatively, the users can make clear the desired meaning by using compound statement. For example:

if (sum==0) OR if (sum==0)

{ {

if (count==0) if (count==0) result = 0; result =0;

else }

result = 1; else

} //end if result = 1;

* Some languages prevent this problem ever occur
* Example, IF statements in ALGOL 60 cannot be placed directly in *THEN part of another IF*. A compound statement *must* be used. For example, two versions of algol

if sum = 0 then if sum = 0 then

begin begin

if count = 0 then if count := 0 then

result := 0 result := 0;

else end

result := 1 else

end result := 1;

* Some languages (e.g., ALGOL 68, Fortran 88 and 90, ADA, and Modula-2) require that each if statement ends with a special word such as *end if (ADA)* and *END* (Modula-2). For example,

-- The two versions of *nested if else* in ADA

if sum = 0 then if sum = 0 then

if count = 0 then if count := 0 then

result := 0; result := 0;

else end if;

result := 1; else

end if ; result :=1;

end if; else if;

Multiple Selection Constructs

* Allows the selection of one of *any number* of statements.
* Can be built from multiple two-way selections. For example,

if … else if … else if …. else …

* Fortran I has a special three-way selection which is efficient but may makes a program hard to read. It has the form:

IF (Arithmetic Expression) N1, N2, N3

IF (A-B) 10, 20, 30

1. … \* A-B > 0 or A>B

Goto 40

20 … \* A .EQ. B

Goto 40

1. … \* A < B
2. …

* Modern languages also have *case* (or *switch*) statement, which was first introduced in ALGOL in 1966.
* Most languages have a default/exceptional clause when the selector value is not in the list. For example:

{Pascal code that find the average of all even and odd numbers}

case num mod 2 of

0: begin

even := even+1;

evensum := evensum+num

end;

1: begin

odd := odd+1;

oddsum := oddsum+num

end;

else

writeln (‘The number ’, num, ‘ is out of range’);

end {of case }

* Case (PASCAL, ADA, Fortran 90) statements have implicit branches at the end of each code segment (i.e., control will transfer to the first statement after the case statement).
* Switch (C, C++) statements do NOT have implicit branches at the end. To branch out, an explicit break statement is required.

// store integers in their corresponding array based on number of digits

// Can we do this with a C++ switch?

if (c <10)

oned[bag1++]=c;

else if (c < 100)

twod[bag2++]=c;

else if (c < 1000)

threed[bag3++]=c;

else if (c <10000)

fourd[bag4++]=c;

int digit, cc;

for (digit=0, cc=c; cc>0;digit++,cc /=10); // count digits in the number c

switch (digit)

{

case 1: oned[bag1++]=c; break;

case 2: twod[bag2++]=c; break;

…

}

6.5 Iteration

* Cause statements to be repeated (0 or more times).
* Most commonly used in imperative and object oriented languages.
* In logic programming languages, recursion is usually used to repeat things.

For example,

fib(0,N) :- N=1.

fib(1,N) :- N=2.

fib(F, N) :- F>1, F1 is F-1, F2 is F-2,

fib(F1,N1), fib(F2,N2), N is N1+N2.

**Types of iterations**

1. Counter-Controlled Loops – repeat a predetermined number of times

* The counter is also called the Loop Control Variable (LCV).
* Each loop has an initial and terminal value of the LCV and a step-size indicating how the LCV is changed.

Design issues:

1. What is the data type of the LCV?
2. What value does the LCV have when loop ends?
3. Can LCV be changed in the loop body?

Example of iteration loop from selected languages

Fortran 95:

Do label LCV = initial, terminal [,stepSize]

…

label CONTINUE

* The default stepSize is 1.
* The loop is repeated until the LCV is greater than the terminal value.

Q. How many times does the following code repeats?

Do 10 J = 1, 10

J = 11;

SUM = J + SUM

1. CONTINUE

*IT STILL REPEATS 10 TIMES. The number of iteration is calculated at the beginning of the DO loop and stored in an internal count. The internal count is used to control the execution of the loop.*

* Fortran 95 Design choices:

1. LCV can be integer, real, or double-precision
2. The LCV retains its last value. LCV > terminalValue at the end of the loop. In the above example, J is 11 immediately after the loop.
3. LCV value can be changed in the body. However, this does not affect the number of iteration.

PASCAL:

for lcv := initialValue (to | downto) finalValue do

statement

* to is +1, downto is –1
* Design choices:

1. LCV is any ordinal data type.
2. LCV value is undefined if the loop terminates normally. Otherwise, it retains its last value.
3. LCV cannot be changed in the loop body.

ADA:

*for* LCV *in* [reverse] discreteRange *loop*

*…*

*end loop*

* The discreateRange is a subrange of an integer or enumeration type.
* The scope of the LCV is the loop. For example:

count : FLOAT := 1.35;

for count in 1..10 loop

sum := sum+count;

end loop

The float count is not affected by the FOR loop in which count is implicitly declared as integer.

* Design choices:

1. LCV is integer or enumeration type.
2. LCV is undefined at the end of the loop. It does not retain its value.
3. LCV cannot be assigned value in the loop body.

C, C++:

for ([expression1];[ expression2]; [expression3])

loop body

outside

*[ ] indicates optional*

* The loop body is a single statement (simple or compound)
* Expressions are regular expressions (e.g., j<10) or assignment statements (e.g., j=0).

Semantics:

expression1

loop:

if expression2 is 0 goto outside

loop body

expression3

goto loop

outside: …

* All of the expressions are optional. An absent expression2 is considered true. For example:

for (;;) // while(1)

{…// loop body } // when does this end? Whenever we break, or return, or exit

* In C++, LCV can be declared in expression1. For example:

for (int j=0;j<10;j++)

// scope j should be the loop itself

* For loop is more than a counter loop. It can be used to model any iteration structure. For example:

// traversing a linked list beginning at head

for (node \*p=head; p; p = p->next)

… // process the node pointed to by p

// what does the following statement do?

int v;

char ch;

for (v=0, ch='0'; isdigit(ch);v=v\*10+ch-'0',fin.get(ch));

* Design Choices

1. LCVs can be any data type.
2. Value of LCV that is declared within the loop is undefined (i.e.. compiler dependent) when loop ends.
3. LCV can be changed in the loop body.

Java:

for (expression1;boolean\_expression;expression3)

loop body;

design choices:

1. LCV can be any data type.
2. The scope of LCV declared within the loop is the FOR loop itself.
3. LCVs can be changed in the loop body.
4. Logically Controlled Loops – repeat unknown number of times

Design issues 1: pretest (0 or more times) vs. post-test (1 or more times)

* Some imperative languages (e.g., Pascal, C, C++, Java) have both.

For example:

// C, C++

while (expression) // repeat until expression is false

statement

do

statement

while (expression); // repeat until expression is false

{ Pascal }

while expression do {repeat until expression is false }

statement

repeat

statements

until expression {repeat until expression is true }

* Fortran IV has no logical loop.
* Java is the same as C++ except that the control expression must be Boolean.
* Ada and perl has pretest but no post-test,

Is that true and why?

Post-test loops are not necessary and somewhat dangerous.

1. User-located Loop Control Mechanisms

* Some languages allow the programs to choose a location for loop control, which is desirable in some situations

Design Issues:

1. Should the conditional mechanism be an integral part of the exit?
2. How many loop bodies should be exited?

Example

C++ allows the user to either *exit* a loop or skip the rest of the loop and continue with the next iteration using *continue*. For example:

// C, C++

for (;;)

{

cin >> v;

if (v < 0) *continue*; // goto next iteration, skip the rest

sum += v;

if (sum >1000) *break*; // exit the loop

}

// equivalent code

while(sum<=1000)

{

cin >> v;

if (v>=0)

sum += v;

}

Does C++ break from 1 loop or many?

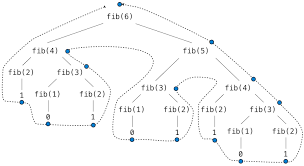
See break.cpp

6.6 Recursion – a sub that calls itself

* equally powerful as iteration
* mechanical transformations back and forth
* often more intuitive (sometimes less)
* no special syntax required
* fundamental to functional and logic languages
* *naïve* implementation is often inefficient.

**Example: fib sequence**

prolog: **fib.pl**



**Solution:**

Tail Recursion (kinda iteration)

* No computation done after the recursive call.

i.e., the recursive call is the LAST statement of the function.

* Function returns whatever its own recursive call returns.
* No dynamic allocation of recursive calls – compiler can reuse the space allocated to current function.
* Even for functions that are NOT naturally tail recursive, we can transform it.
* Ex., scheme (oct25.rkt), prolog (fib.pl)

[Memoization](https://www.youtube.com/watch?v=GtkyWJuW_Nc)

* Remember and recall results of previous calculations
* Goal: achieve total recall after linear times
* C++: fib.cpp

6.8 summary

* Evolution of control: structured control, elimination of goto
* Purpose of this evolution:
  + ease of programming
  + semantic elegance
  + runtime efficiency,
* New features are added to make program more elegant and efficient:

Ex., inline implementation of member functions,

exception