Procedural Programming

Contents

- Concepts of procedural programming
- Programming as an industrial process
- Language-independent programming
- Basic data types
- Basic control structures
- Comparison of languages
- The structure theorem (G. Jacopini)
- Program performance (timing experiments)
- Operators and expressions
- Data types and conversions
- Numeric accuracy (overflow, range, precision)

Classification of Programming Languages

- Main paradigms:
  - Procedural programming [Fortran, Pascal, C]
  - Logic programming [Prolog]
  - Functional programming [Lisp, Scheme]
  - Object-oriented programming [C++,Java,C#,Ruby]
- Multi-paradigm languages support multiple paradigms (e.g. Ruby = proc+ OO +functional)
- All languages have procedural components

BASICS OF PROCEDURAL PROGRAMMING

Why Procedural Programming?

- The basis of each programming effort is algorithmic thinking and procedural programming.
- Procedural programming requires the ability for complex combinatorial thinking.
- Procedural programming is a way to satisfy performance requirements of industrial software products.
- Object-oriented programming is a layer that comes on top of basic procedural programming.

Goals of procedural programming

- The basic goal of procedural programming is to achieve high performance of software products by developing procedures that efficiently use all capabilities of hardware.
- Procedural programming generally needs significant programming skills and effort.
- Procedural programs solve problems by specifying activities that manipulate data in memory locations in a way that eventually generates a problem solution.

Nonprocedural vs. procedural
What vs. How

- Nonprocedural programming is an approach that reduces procedural complexity and programming effort by making programs close to problem specification (as opposed to the specification of hardware activity)
- Nonprocedural programming approach usually generates programs with less programming effort than in the case of procedural programming. However, nonprocedural programs achieve lower performance than the equivalent procedural programs.

Components of Procedural Programming

- A set of basic data types
- A set of basic control structures
- Expression evaluation rules
- Care for program performance
- Care for accuracy of results

Language-independent Procedural Programming

- Language support for procedural programming can vary from language to language
- Union of features supported in various languages creates a General Pseudo Language (GPL)
- Program development in GPL is language-independent and has universal applicability

Programming: Science or Engineering?

- Theodore von Karman definition of difference between science and engineering:

  A scientist studies what is, whereas an engineer creates what never was.

  Programming = software engineering
  = engineering

Each program is an artifact

- Program is an artifact (an object made by human work)
- Two fundamental approaches:
  - Program as a mathematical object
  - Program as an engineering product

Program as a Mathematical Object

- Program is an algorithmic solution of a mathematical problem.
- It must be correct (it must generate correct results). Sometimes, this means to provide a formal proof of program correctness.
- It must generate solution after a finite number of steps.

Program as an Engineering Product

- Program as an engineering product is produced by software engineers in an industrial environment that usually assumes collaboration, outsourcing, and competition
- As industrial products, programs are made either for a specific buyer or with intention to be sold on a free market (as individual products, or as parts of other industrial products)
- Programs must satisfy conditions for engineering products (e.g. quality standards and warranty)

Engineering Conditions that Programs Must Satisfy

- Program must be built according to specifications
- It must be correct as a mathematical object, generate correct results
- It must be robust, tested, and satisfy QA standards
- It must be produced on time and within available budget
- It must be properly documented (installation + usage + maintenance)
- It must satisfy usability standards and have a quality user interface
- It must satisfy performance standards
- It must be designed to be robust, maintainable, and reusable

Large and Small Programs

- Large programs are typical engineering products
- Large programs are always designed as superposition and aggregation of small programs
- Therefore, even the smallest programs must satisfy same engineering criteria as large programs

Programming languages as tools for making software products

- We assume that programming is an engineering activity
- Under these assumptions programming languages are tools that affect the cost and performance of software products
- Programming languages must provide environment for efficient program design and development

Use of Programming Languages

- Each language is
  - a means for communication
  - a medium for expressing algorithmic ideas (computational procedures)
  - a tool for problem solving
- Programming languages support two kinds of communication:
  - Human-computer communication
  - Communication between humans (between software designer, and software maintenance engineer)

Features of Programming Languages

- Rule#1: All programming languages are good and deserve respect for their features and/or contribution to the development of the art of programming.
- Each language is oriented towards some specific application area.
- Each language has some unique good features.

A Simplistic Classification of Programming Languages

- Assembly language (symbolic machine language) is considered a low-level language (enables a direct control of hardware resources)
- High-Level Languages (HLL) are all languages above the level of assembly language (they provide less control of HW than assembly language)

Ratio of HLL and Assembly Language (AL)

- Rule of thumb:

  1 HLL instruction ≅ 5 AL instructions

- Consequences: compared to HLL, AL has:
  - 5 times more lines of code
  - 5 times more documentation (comment per line)
  - Less readable and more difficult for testing
  - 5 times more programming effort and development cost
  - Better performance (higher compactness and speed)

High-Level Languages

Language	**Primary application area**
FORTRAN	Engineering
COBOL	Business
Pascal	Education (teaching of programming)
BASIC	Fast learning (for beginners)
C	System software (procedural programming)
LISP/Scheme	List processing (functional programming)
PROLOG	Artificial intelligence (logic programming)
C++	General purpose (OO programming)
Ruby	OO programming & web applications
SQL	Database applications
Java	Internet applications (general purpose)

Level of Programming Languages

- Level of programming language is usually defined as the level of abstraction used in language constructs
- The level of abstraction denotes the distance from features and limitations of available hardware (primarily processor)
- Low level is NOT automatically bad
- High level is NOT automatically good

Abstraction and Performance

- Higher the level of abstraction, more comfortable programming for humans
- Each increase of the level of abstraction is paid by abstraction overhead and a corresponding reduction of performance
- It is important to properly evaluate the cost of abstraction and use a language that offers justifiable abstraction-performance tradeoff

High Level vs. Low Level

- Advantages of low-level languages (LLL):
  - – Possibilities to attain high performance
  - – Good control of HW (e.g. register variables in C)
- Advantages of high-level languages (HLL):
  - – High portability
  - – Abstraction – ignoring HW limitations

Example: Ruby's core interpreter is written in C
(C has some LLL properties and Ruby is a HLL)

Big Language vs. Small Language

- Small language features:
  - Small number of powerful components
  - Easy to learn and master (book<200pp; e.g. C)
  - The scope of language can be limited
- Big language features:
  - Large number of components
  - Difficult and/or time consuming to learn (book>1000 pp, e.g. C++)
  - Universal (designed for wide spectrum of problems)

Expressive Power of Language

- Expressive power of a language is defined as the ease of expressing algorithmic ideas in a given problem domain
- High expressive power means low programming effort (and low cost of software development effort)

Language-Independent Algorithmic Thinking and Procedural Programming

Language-independent approach to procedural programming

- Define a general pseudo language (GPL) that includes:
  - A complete collection of basic data types
  - A complete collection of basic control structures
- Use GPL as a mental framework for developing algorithms and programming procedures
- Interpret each programming language as a special case of GPL

Goals of language-independent programming

- Every programmer has a native programming language (usually the first programming language, or the language where programmer feels maximum level of confidence and security).
- The native language is a frame that creates habits that can permanently limit programmer's expressive power.
- Language-independent programming is an approach that avoids limitations of the native programming language by promoting thinking in GPL.
- Programming languages are tools for expressing algorithmic ideas. It is important to use multiple tools, and to solve each problem using the most suitable tool.

Defining GPL

- Create a table of all basic data types.
- Create a table of all basic control structures.
- These tables contain basic components of GPL.
- Compare real programming languages from the standpoint of their support to basic data types and basic control structures.

Ordinal Data Type (Pascal)

Ordinal data type is each data type containing ordered data that can be mapped onto ordinal numbers 0,1,2,… (exception: integers map onto themselves)

Example: **SUN < MON < TUE < WED < THU < FRI < SAT**

0 < 1 < 2 < 3 < 4 < 5 < 6

ord(x) returns the order of the ordinal expression x; e.g. ord('A') returns 65

pred(x) returns the predecessor of x; e.g. pred('B') returns 'A'

succ(x) returns the successor of x; e.g. succ('A') returns 'B'

Predecessor for loop is based on update index:=pred(index):

```
for i:=′Z′ downto ′A′ do begin … end
```

Successor for loop is based on update index:=succ(index):

```
for i:=′A′ to ′Z′ do begin … end
```

Classification of Data Types

GROUP / CLASS				DATA TYPE	EXAMPLE
BASIC DATA TYPES	STATIC	SCALAR	ORDINAL	LOGICAL	true, false
				CARDINAL	0,1,2,... ,m
				INTEGER	minint,... ,-1,0,1,... ,maxint
				CHARACTER	‘A’, ‘B’,... ‘0’, ‘1’, ...
				ENUMERATED	(JAN, FEB,... , DEC)
				SUBRANGE	0 .. 255 , MON .. FRI
			R&C	REAL	-12.34 , -0.1234E+2
				COMPLEX	(1.2 , 3.4) , (1.2 , -3.4)
		STRUCTURED		BIT STRING	‘10010110’ (length=const)
				C STRING (array of char)	“c_string” (length=const)
				ARRAY	a[100], M[200][10]
				SET	[1,2,3] [‘a’,’e’,’i’,’o’,’u’]
				RECORD	point{int x; int y;}; p.x , p.y
	DYNAMIC	VARIABLE SIZE		BIT STRING	‘1001011011’ (any length)
				CHARACTER STRING	“string” (any length)
				SET	[ ] , [1,2,3,...] (any length)
				ARRAY	{int a[n][k]; ... }, adr=new int[N];
				STACK	s : stack of integer
				QUEUE	q : queue of double
				SEQUENTIAL FILE	Sequential read, write, append
				RELATIVE FILE	Direct access using record #
				INDEXED FILE	Direct access using key
		VARIABLE STRUCTURE		LIST	Linear structures of linked nodes
				TREE	Tree structure of linked nodes
				POINTER	int n; int* address_of_n = &n;

Language Support for Data Types

GROUP / CLASS				DATA TYPE	PAS	BAS	FOR77	COB
BASIC DATA TYPES	STATIC	SCALAR	ORDINAL	LOGICAL	+	≈	+	-
				CARDINAL	-	-	-	-
				INTEGER	+	+	+	+
				CHARACTER	+	+	+	+
				ENUMERATED	+	-	-	-
				SUBRANGE	+	-	-	-
			R&C	REAL	+	+	+	+
				COMPLEX	-	-	+	-
		STRUCTURED		BIT STRING	-	-	-	-
				C STRING (array of char)	≈	+	+	+
				ARRAY	+	+	+	+
				SET	+	-	-	-
				RECORD	+	-	-	+
	DYNAMIC	VARIABLE SIZE		BIT STRING	-	-	-	-
				CHARACTER STRING	-	+	-	-
				SET	-	-	-	-
				ARRAY	-	-	-	-
				STACK	-	-	-	-
				QUEUE	-	-	-	-
				SEQUENTIAL FILE	+	≈	+	+
				RELATIVE FILE	-	≈	+	+
				INDEXED FILE	-	-	-	+
		VARIABLE STRUCTURE		LIST (various linking)	-	-	-	-
				TREE	-	-	-	-
				POINTER	+	-	-	-

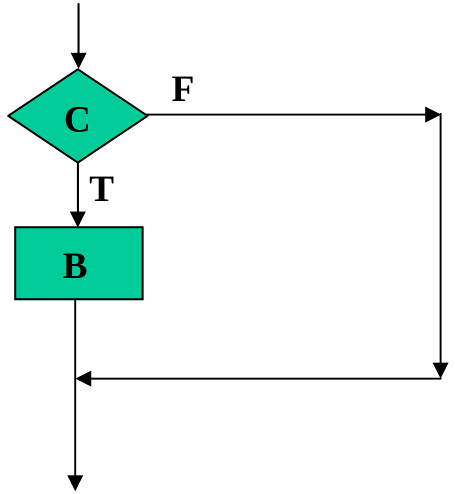
Classification of Control Structures

GROUP/CLASS		Sample control structure
SEQUENCE		begin-end
SELECTIONS		if-then-else
		if-then
		switch-case
		if-elsif-else
		if-thenif-else
LOOPS	LOOPS WITH CONDITIONAL EXIT	while-do
		do-while
		loop-exit-endloop
	INFINITE LOOP	loop-endloop
	LOOPS WITH COUNTER	perform-times (hidden counter)
		for-to-step
		for-to (succ)
		for-downto (pred)
		for (init;cond;update)
		foreach-in-do
	LOOPS WITH COUNTER AND CONDITION	for-to-step-where
		for-to-where (succ)
		for-downto-where (pred)
		foreach-in-where
JUMPS		break (exit from loop)
		stop (exit to the OS)
		return (from subprogram)
		go to (arbitrary destination)

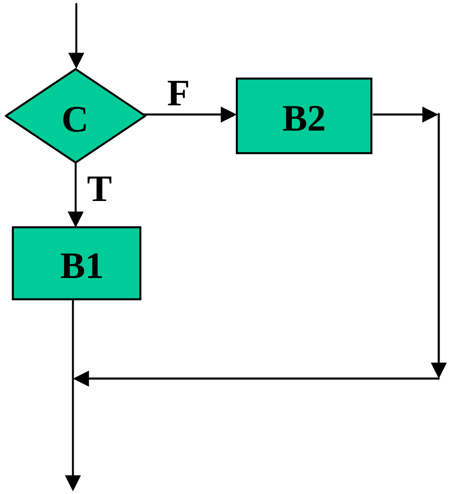
Language Support for Control Structures

GROUP/CLASS		Sample control structure	PAS	BAS	FOR77	COB
SEQUENCE		begin-end	+	-	-	+
SELECTIONS		if-then-else	+	≈	+	+
		if-then	+	+	+	+
		switch-case	+	≈	≈	≈
		if-elsif-else	-	-	+	+
		if-thenif-else	-	-	-	-
LOOPS	LOOPS WITH CONDITIONAL EXIT	while-do	+	-	-	+
		do-while	+	-	-	+
		loop-exit-endloop	-	-	-	-
	INFINITE LOOP	loop-endloop	≈	-	-	≈
	LOOPS WITH COUNTER	perform-times (hidden counter)	-	-	-	+
		for-to-step	-	+	+	+
		for-to (succ)	+	-	-	-
		for-downto (pred)	+	-	-	-
		for (init;cond;update)	-	-	-	-
		foreach-in-do	-	-	-	-
	LOOPS WITH COUNTER AND CONDITION	for-to-step-where	-	-	-	-
		for-to-where (succ)	-	-	-	-
		for-downto-where (pred)	-	-	-	-
		foreach-in-where	-	-	-	-
JUMPS		break (exit from loop)	-	-	-	-
		stop (exit to the OS)	-	+	+	+
		return (from subprogram)	+	+	+	+
		go to (arbitrary destination)	+	+	+	+

Skip and Branch



```
// Skip
if C
then  B
end if
```

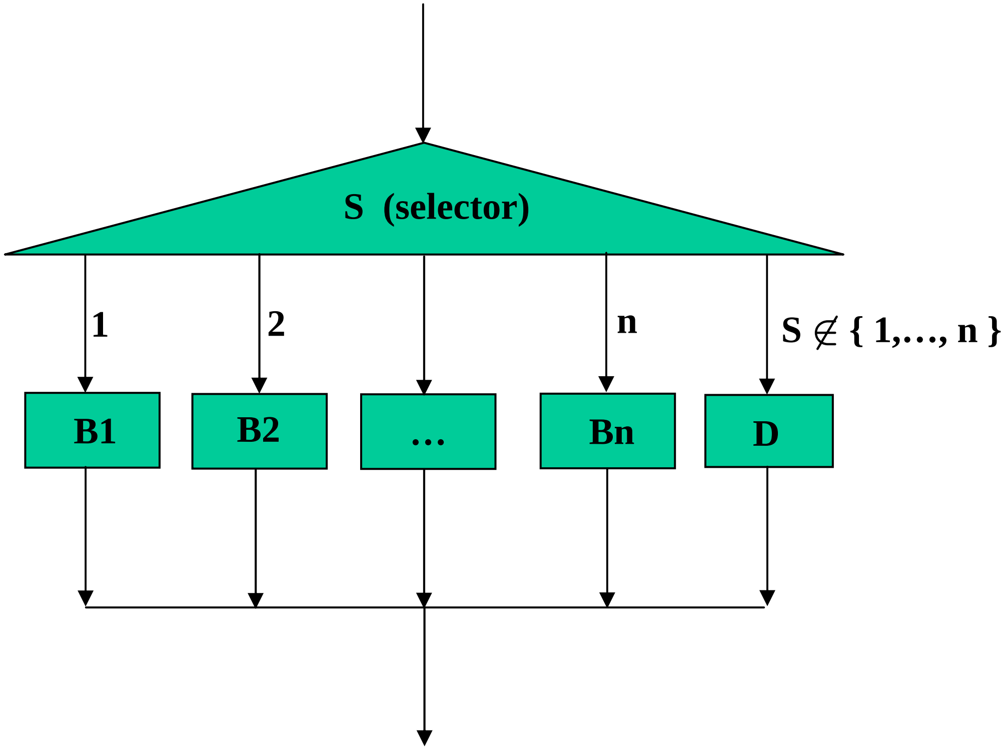


```
//Branch
if C
then B1
else B2
end if
```

Ruby

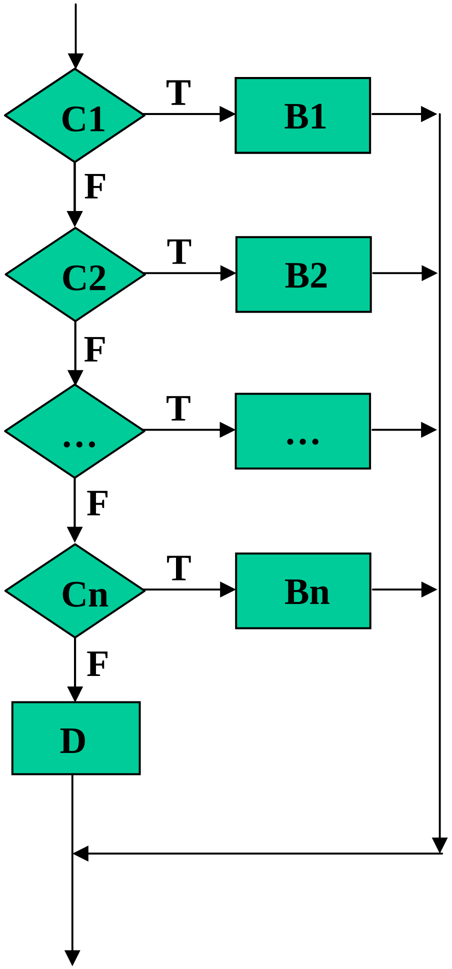
```
if C then B end

if C:  B end

if C

   B

end


B if C      (if as a modifier)


if C
   B1
else
   B2
end
```

Switch-case



```
case S of
   1:  B1;
   2:  B2;
   .........
   n:  Bn;
else D;
end case
```

If-elsif-else

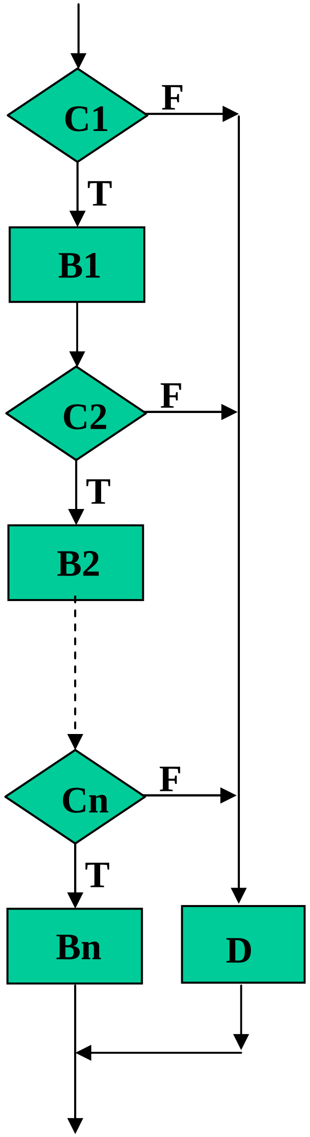


```
if    C1 then  B1
elsif C2 then B2
…………………….
elsif Cn then Bn
else D
endif
```

Ruby

```
if C1
   B1
elsif C2
   B2
elsif C3
   B3
else
   D
end

if C1then B1
  elsif C2 then B2
  elsif C3then B3
  else D
end
```

If-thenif-else



```
if     C1 then  B1
thenif C2 then B2
……………………
thenif Cn then Bn
else D
endif
```

Classification of Loops (1)

1. Loops with conditional exit

Loop with exit at the top	`while(condition) {BODY}`
Loop with exit at the bottom	`do {BODY} while(condition)`
Loop with exit in the middle	`while(1) {B1; if(ex) break; B2;}`
Loop without exit (infinite loop)	`do {BODY} while(1);`

2. Loops with counter

Loop with implicit counter	`PERFORM PAR n TIMES.`
Loop through set elements	`FOREACH element IN set DO body`
Loops with explicit counter	
Arithmetic loop	`for i=a to z step s :  BODY : next i`
Successor loop	`for i:=1 to n do begin BODY end`
Predecessor loop	`for i:=n downto 1 do begin BODY end`
Loop with general update	`for(i=0; i<n; i=update(i)) {BODY}`

Classification of Loops (2)

3. Loops with counter and condition

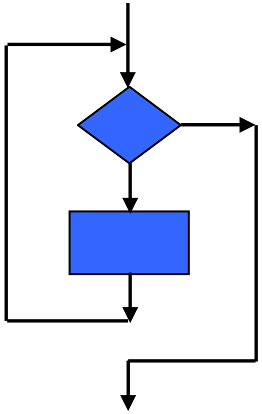
Loop through set elements `FOREACH element IN set WHERE cond DO body`

Loops with explicit counter

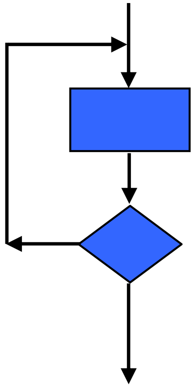
- **Arithmetic loop** `for i=a to z step s where cond: BODY: next i`
- **Successor loop** `for i:=1 to n where cond do begin BODY end`
- **Predecessor loop** `for i:=n downto 1 where cond do begin BODY end`

Loops with conditional exit

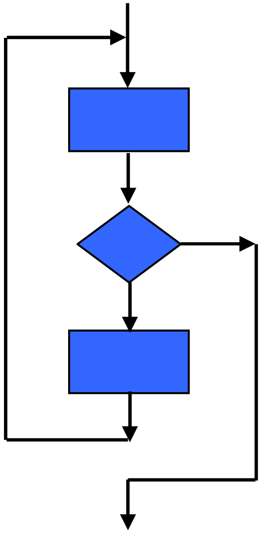
Exit at the top



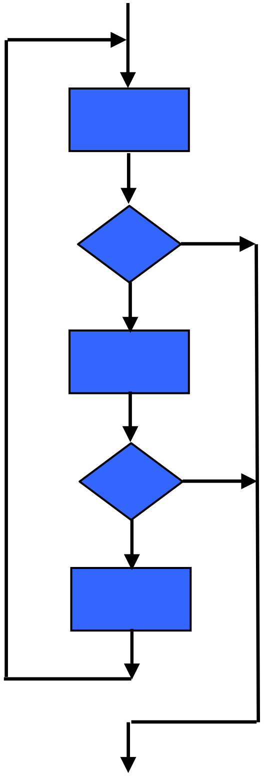
Exit at the bottom



Exit in the middle



Multiple exits

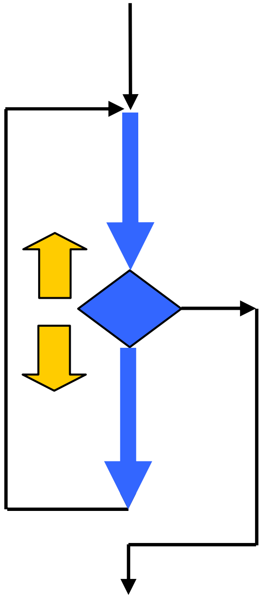


Condition = { Condition for staying in the loop

Condition for exit from the loop

Loop with a general conditional exit

```
loop
    .......
    exit when (condition) ;
    .......
end loop
```



Features: (1) exit at the top/bottom/middle
(2) multiple exits
(3) exit can be omitted (infinite loop)

Loop With a General Update

```
for( INIT ; COND ; UPDATE) BODY;

// This loop is implemented as follows:
INIT;
while (COND)       // If COND is false, we
{                  // skip BODY and UPDATE
   BODY;
   UPDATE;
}
```

Bodiless loop is possible, but not advised:

```
for( INIT ; COND; BODY , UPDATE);
```

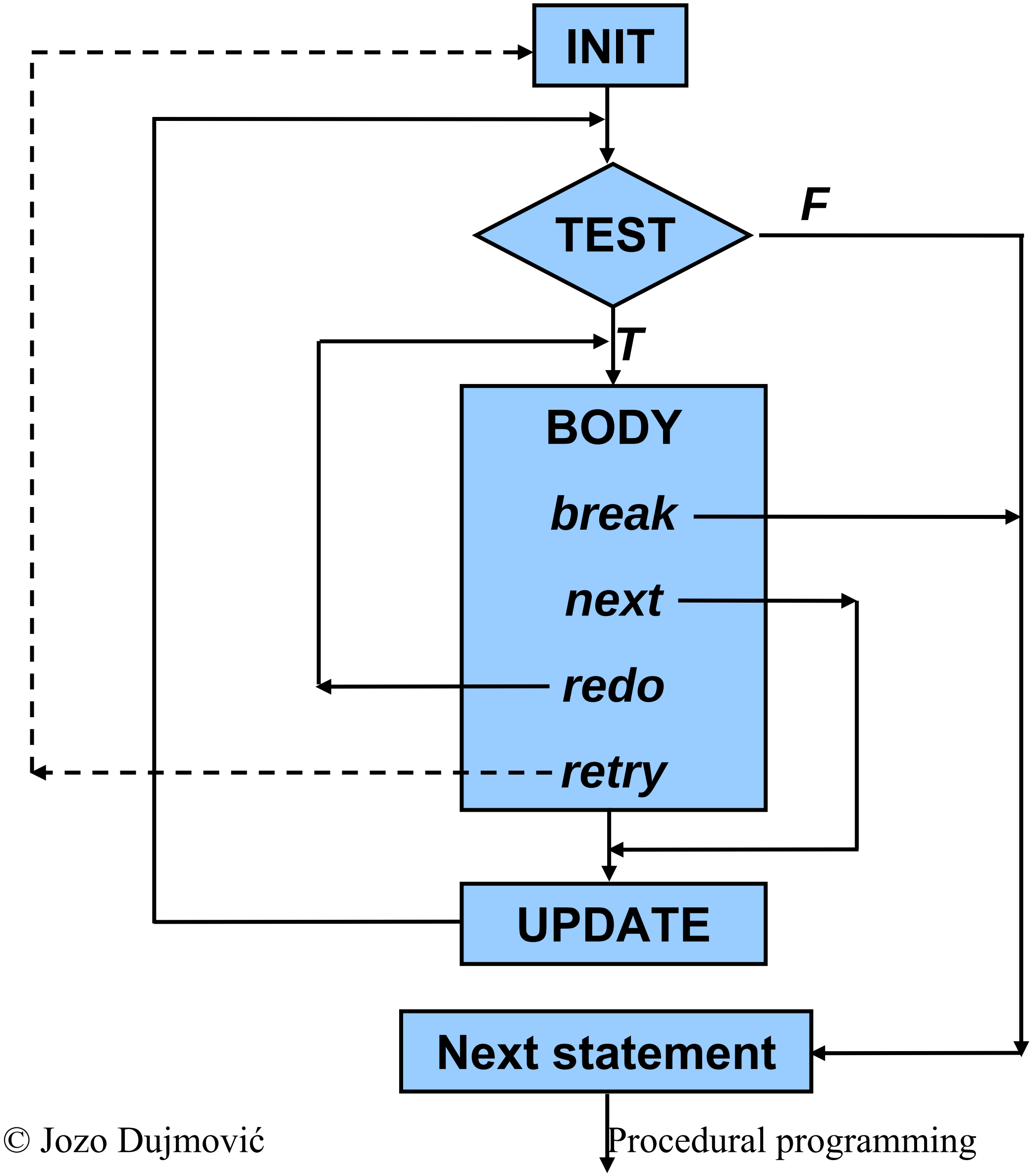
Altering control flow

- Return: causes a method to exit and return a value to its caller
- Break: exit from a loop or iterator
- Next: causes a loop (or iterator) to skip the rest of the current iteration and move on to the next iteration (similar to C++ *continue*)
- Redo: restarts a loop or iterator from the beginning
- Retry: restarts an iterator (useful for exception handling)
- Throw/catch: exception propagation and handling mechanism
- Raise/rescue: Ruby’s primary exception mechanism

Break, redo, next, and retry

- Break, redo, next, and retry alter the execution of the following loops: while, until, for, and iterator controlled loops
- Break terminates the immediately enclosing loop and transfers the control to the statement following the loop
- Next skips to the end of loop and starts the new iteration (it jumps to the point before the evaluation of the loop’s conditional)
- Redo jumps after a loop’s conditional (repeats the same loop iteration)
- Retry restarts the loop from the very beginning

for (INIT; TEST; UPDATE) BODY;



for i in 1..100

  …………..

  break if cond1

  …………..

  next if cond2

  …………..

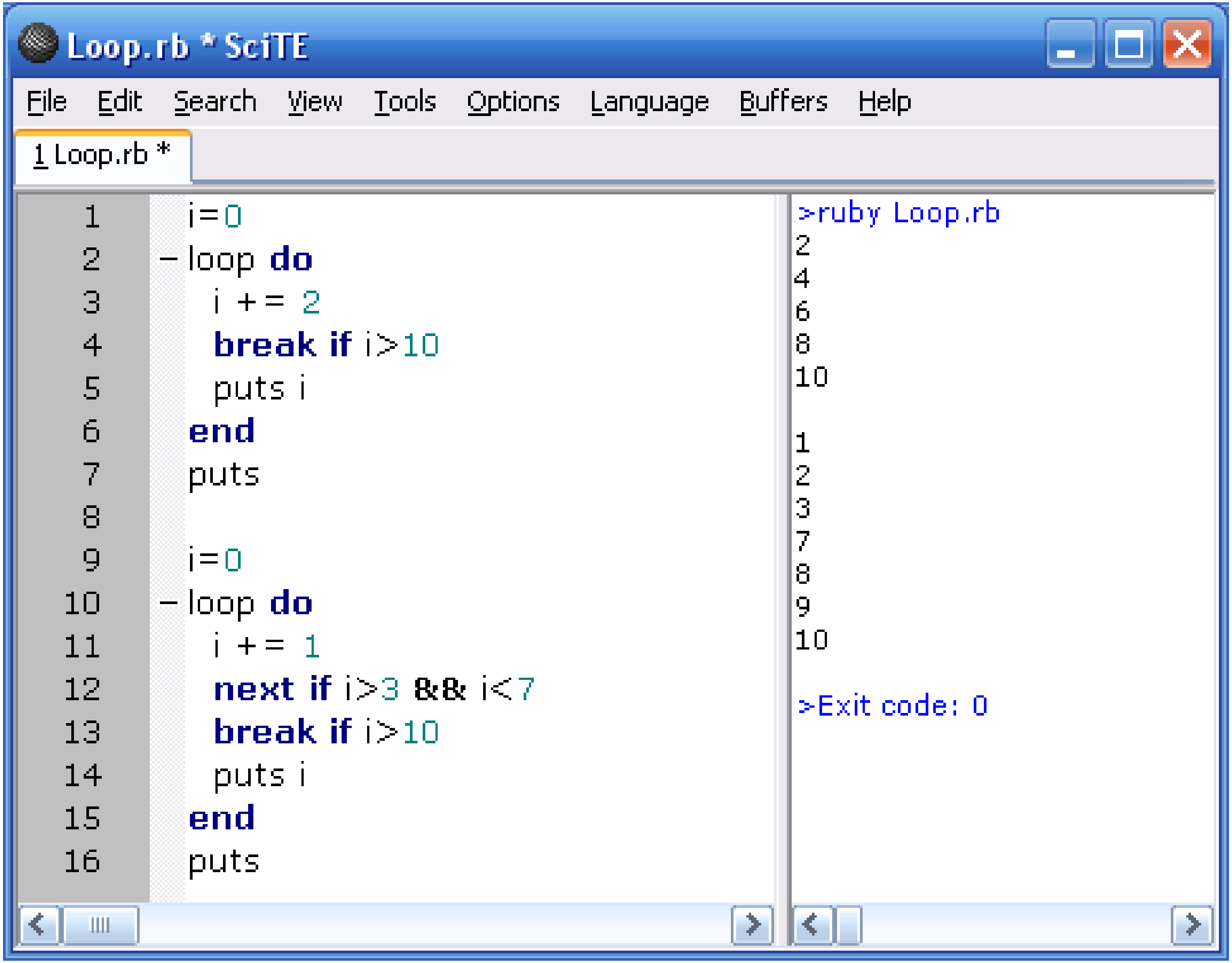
  redo if cond3

  …………..

  retry if cond4

  …………..

end



```
i=0
loop do
  i += 2
  break if i>10
  puts i
end
puts

i=0
loop do
  i += 1
  next if i>3 && i<7
  break if i>10
  puts i
end
puts
```

```
>ruby Loop.rb
2
4
6
8
10

1
2
3
7
8
9
10

>Exit code: 0
```

Loop With Implicit Counter

```
PERFORM { block } n TIMES;
```

Loop counter is not accessible; {block} is repeated n times.

COBOL version of the loop with implicit counter

```
     PERFORM  LOOP  N  TIMES.
     ------
     STOP RUN.
LOOP.
     ------
     ------    } loop body
     ------
```

A combination of loop and function call

Loops With Counter and Condition

Problem #1: Find the maximum component of an array a[1 .. n]

```
amax=a[1];  for i:=2 to n where a[i]>amax do amax=a[i];
```

C++ does not support this control structure. Therefore, this loop in C++ would be the implemented as follows:

```
      amax=a[0];
      for(i=1; i<n; i++)
            if(a[i] > amax)
                  amax=a[i];
```

Problem #2: Select sort of array a[1 .. n] (inefficient $O(n^2)$)

```
for i:=1 to n-1 do
    for j:=i+1 to n where a[j]<a[i] do SWAP(a[i], a[j]);
```

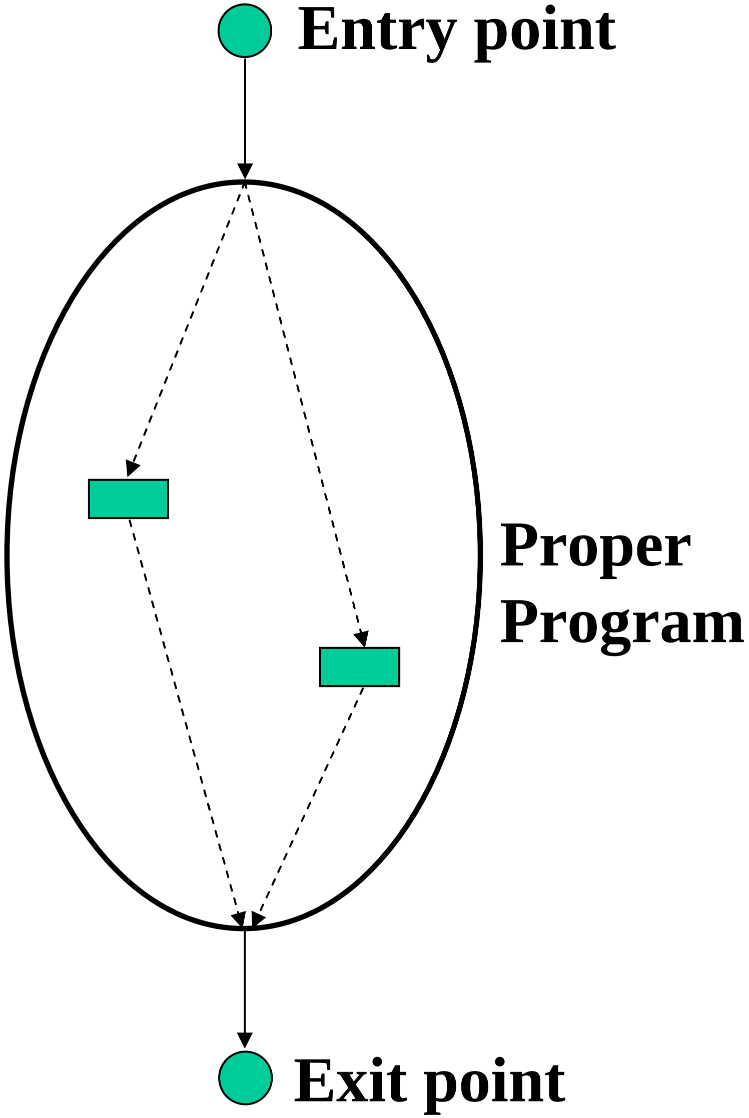
Some theoretical questions

- How to design a specific language?
- What data types to implement?
- What control structures to implement?
- More is better, or less is better? (big or small?)
- What are the components that are absolutely needed? (What is the minimum number of components?)

The Concept of Proper Program

Proper program (PP) is a program that satisfies the following three conditions:

- PP has only one entry point
- PP has only one exit point
- For each statement S in PP there is a path from the entry point to S, and there is a path from S to the exit point. (in other words, no infinite loops and no unreachable code)



Entry point

Proper
Program

Exit point

Giuseppe Jacopini [1966]: The Structure Theorem

- Each proper program can be composed using the following three components:
  - A loop with conditional exit (*while-do* or *do-while*)
  - Sequence (*begin .. end*)
  - Auxiliary variables (flags)
- Theoretically, **if-then-else** and other selections, as well as all other loops, are not necessary (they are only used to decrease the programming effort)

Charles S. Peirce Theorem

- *p* NAND *q* = NOT(*p* AND *q*)
- *p* NOR *q* = NOT(*p* OR *q*)
- **The logic operation NAND and the logic operation NOR are each individually sufficient to derive all other Boolean operations** [later nineteenth century]
- Consequence: all computer hardware could be build using only NOR or NAND gates

Examples of Peirce theorem

not $p = p$ NAND p

p and $q = (p$ NAND $q)$ NAND $(p$ NAND $q)$

p or $q =(p$ NAND $p)$ NAND $(q$ NAND $q)$

$p \rightarrow q =(p$ NAND $q)$ NAND p

not $p = p$ NOR p

p and $q = (p$ NOR $p)$ NOR $(q$ NOR $q)$

p or $q = (p$ NOR $q)$ NOR $(p$ NOR $q)$

$p \rightarrow q =((p$ NOR $q)$ NOR $q)$ NOR $((p$ NOR $q)$ NOR $q)$

Making if-then-else from while-do

```
// C++ version
// if(cond) ThenPart; else ElsePart;
C = cond;
NC = ! C;
while(C)  { ThenPart;  C  = 0; }
while(NC) { ElsePart;  NC = 0; }
```

Using while-do to make do-while

```
do {BODY} while(cond);


// Equivalent program
{BODY}
While(cond)
     {BODY}
```

Assumption: BODY can be a function call and its repetition does not cause problems

Making Loop With the Exit in the Middle

```
for( ; ; )
{   A;
    if(cond) break;
    B;
}
```

```
// Using do-while
do
{ A;
  flag = !cond;
  if(flag) B;
}while(flag);
```

```
// Using while-do
A;
while(!cond)
{ B;
  A;
}
```

Jacopini & Peirce
Building systems from components

- All software can be written using the while-do or do-while loops
- All hardware can be made using the NAND or NOR gates
- These theorems have profound theoretical significance
- In engineering practice, however, it is convenient to build systems using more than the minimum set of components

EXPRESSIONS

Basic concepts of expressions (1)

- Expressions are combinations of operands and operators that return a value
- Exceptionally, the returned value can be void (expression is executed for side effects)
- All language constructs can be interpreted as expressions
- A general concept of expression is very important – it explains almost everything in procedural programming

Basic concepts of expressions (2)

- All operands must have legal data types (3.14/”string” is illegal)
- Mixed mode expressions: simpler data types are automatically promoted to match more complex data types, and this consumes processor time.
- Expressions are expected to return a single value; all other activities are interpreted as side effects.

Examples of side effects

- If a void function displays results that activity is a side effect
- If a function modifies global variables, that activity is a side effect
- Arithmetic expressions can include side effects: x = (--n+1)*(2+k++)*(p=3); (modifications of n, k, and p)
- Side effects are not always safe; generally, they increase program complexity, and should be carefully controlled

Assignment as Expression

- In some languages (e.g. FORTRAN) an assignment is not an expression (i.e. it does not return a value). In such cases it is not legal to write x=3.14*(a=2.71*b+c)
- In C++ assignments are expressions. E.g., (n=3+4) returns 7 and it is legal to write

  cout << 3*(n=3+4); that assigns n=7 and inserts 21 in the output stream cout.
- Warning: if assignment is expression then both

  if(a==3) statement; and if(a=3) statement; are legal.
- Standalone expressions are also legal: 7; x; a+b;

All statements can be interpreted as expressions

```
(if x>y then z=x endif)
(if x>y then (if a<1 then z=x endif) endif)
(if x>y then
      (if z>x then y=a else y=b endif ) endif)
(if x>y then
      (if z>x then y=a endif) else y=b endif )
```

An example from Ruby

```
name = if            x == 1 then “one”
               elsif x == 2 then “two”
               elsif x == 3 then “three”
               elsif x == 4 then “four”
               else “many”
          end
```

An example from Ruby (cont.)

```
name = case
          when x == 1 then “one”
          when x == 2 then “two”
          when x == 3 then “three”
          when x == 4 then “four”
          else “many”
       end
```

Specification of Operators

- Arity (the number of operands)
- Type of operands (for all operands)
- Associativity (left to right or right to left)
- Parenthesis-Free Notation (suffix or prefix notation)
- Priority (order of execution in expressions without parentheses)

Arity

- **Unary**: <operator> <operand> or <operand> <operator>

  -x   !x   ~x

  ++x   --x   x++   x--

- **Binary**: <operand> <operator> <operand>

  x = y

  x + y ;   x – y ;   x * y ;   x / y ;   x % y ;

  x || y ;   x && y ;

  stream << y ;   stream >> y ;

  x & y ;   x | y ;   x ^ y ;

  unsigned >> ShiftCount ;   unsigned << ShiftCount;

- **Ternary**: <opnd> <operator1> <opnd> <operator2> <opnd>

  (x ? “nonzero” : “zero”)

Arithmetic Operators (+ , - , * , /)

- Both operands must have compatible data type
- Possible types: char, short, int, long, float, double
- Promotions: conversion of simpler data type to more complex to equalize the type of operands

  char→ short→ int→ long→ float→ double → long double

Type Conversion

- Type conversion from simple to more complex is done automatically to equalize data types in a mixed mode expression
- Type conversion from complex to simple usually generates warnings
- All type conversions consume processor time, and should be done only with justification; otherwise they show either negligence or ignorance

Sample Type Conversions

- Real = Int; // Automatic type conversion
- Real = Int + 0.; // Mixed mode expression, same as Real = double(Int);
- Int = Real; // Warning: loss of accuracy
- Int = int(Real); // No warning; compiler accepts int( ) as signal that the type conversion is intentional and justified

Types in Conditional Expression

```
( condition ?  expression_1 : expression_2 )
```

Same type of expressions (= type of result)

A typical error in the case of static typing:

```
cout << ( x < 0. ?  “\nERROR”  :  sqrt(x) )
```

String ≠ double

Static vs. Dynamic Typing

- Static typing: the type of object is constant from the moment of construction to the moment of destruction (type-checking can be performed at the compile-time)
- Dynamic typing: the type of object is determined at the time of assignment of value, i.e. at run-time. Types are associated with values not variables. E.g., what is the type of this result in the case of dynamic typing?

```
    result = ( x < 0. ?  “\nERROR”  :  sqrt(x) )
```

- Answer: either string or double depending on x.

Types of operands: stream extract/insert

Left operand of extract operator must be an input stream. Right operand must be an object that can be extracted from the stream (i.e. it cannot be an expression that combines multiple objects):

(<input stream> >> <object>)

Left operand of insert operator must be an output stream. Right operand must be an expression that is evaluated and the result is inserted in the stream:

(<output stream> << <expression>)

Associativity

- Left to right

  $a + b + c + d + e = ((((a + b) + c) + d) + e)$

  $a + b - c + d - e = ((((a + b) - c) + d) - e)$

  $a / b / c / d / e = ((((a / b) / c) / d) / e)$

  $a * b * c * d * e = ((((a * b) * c) * d) * e)$

  $a * b / c * d / e = ((((a * b) / c) * d) / e)$

- Right to left

  $(a = (b = (c = (d = e))))$

Insert and Extract Expressions

- << = insert operator (insert in stream)
- >> = extract operator (extract from stream)
- Insert expression:

  (<output stream> << <expression>)

  This expression returns <output stream>

  (cout << n) is an expression that returns cout
- Extract expression:

  (<input stream> >> <object>)

  (cin >> n) is an expression that returns cin

Associativity of insert/extract

Left to right:

```
((((cout << a)  << b) << c) << d;)
cout  << a << b << c << d;

((((cin >> a)  >> b)  >> c) >> d;)
cin >> a >> b >> c >> d;
```

Associativity and the order of expression evaluation

- Insert operator is associative from left to right but that does not mean that expressions are evaluated in order from left to right (1. expr1, 2. expr2., 3. expr3.): cout << expr1 << expr2 << expr3;
- Example: cout << f(n) << ++n << g(n) ; If the order of evaluating f(n), ++n, and g(n) is not specified then it is not clear what values of n are used for computing f(n) and g(n).

```
#include<iostream>
using namespace std;

int f(int n)
{
   return n;
}

int main(void)
{
   int n=2;
   cout << f(n) << f(n) << f(++n) << f(n) << f(--n) << endl;
   cout << f(n) << f(n) << f(++n) << f(n) << f(n) << endl;
   cout << f(++n) << f(++n) << f(++n) << f(++n) << f(n) << endl;
   return 0;
}

//  Functions are computed right to left, and the results are
//
//  22211
//  33322
//  76543
```

Function computation

Parenthesis-Free Notation

- Invented by Jan Lukasievicz (1929). Both the prefix and the suffix notation of operators can eliminate parentheses.
- Prefix notation (or “Polish notation”): (a+c)*c → *+abc
- Suffix notation(“Reverse Polish Notation”):

  x = (a+b)*c → x = ab+c*

  Suitable for programming:

```
push a
push b
add
push c
mul
pop x
```

Logic Operands and Operators

Two approaches to logic values:

- **(1) Special data type: false / true**
- **(2) Numeric values: nonzero / 0**

Advantage of special types: complete separation of numeric and logic values/expressions contributes to safety

Advantage of numeric values: combining arithmetic and logic data contributes to versatility and expressive power of language

Logic operators: not , and , or

! , && , ||

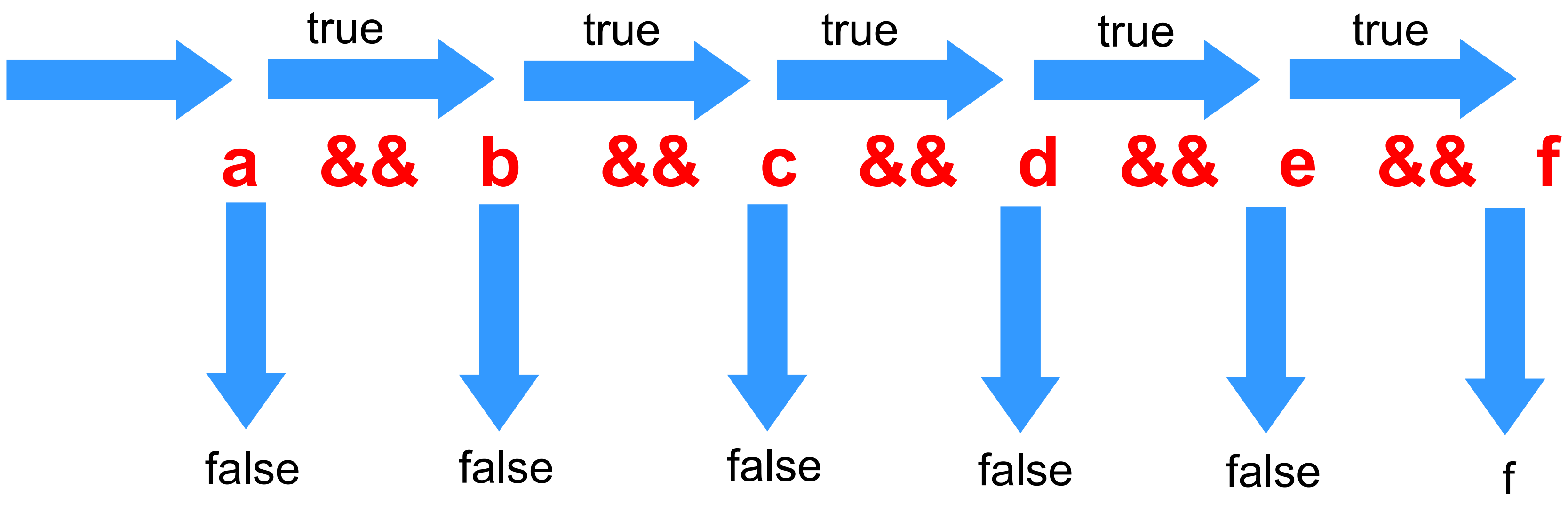
Complete vs.Short-Circuit Evaluation

Logic expression: (x>0) and (x<100)

If x=-3 then it is clear that x>0 is false and the evaluation of second factor (x<100) is not necessary (can be "short-circuited"). In Pascal the short-circuit evaluation is implementation-dependent, and consequently *both factors must be well-defined*. This can cause unsafe programming solutions and out-of-bound errors.

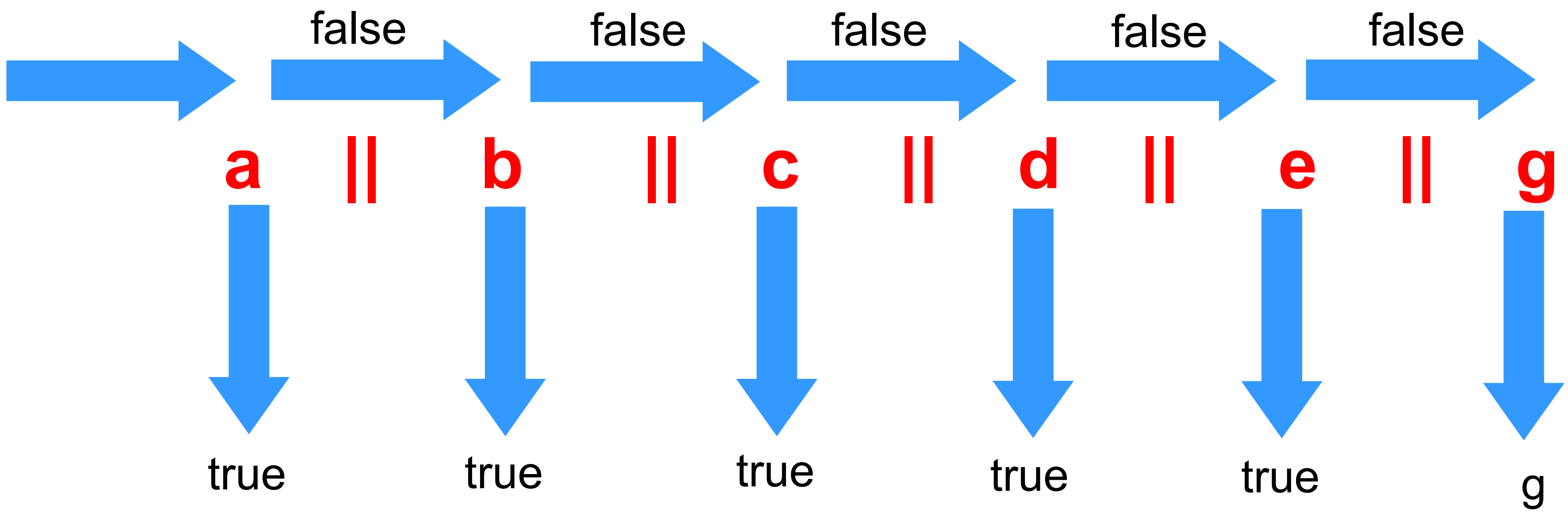
C and C++ use short-circuit evaluation.

Idea of short circuit evaluation: stop as soon as the result is known



The only conditions evaluated are those actually needed.

Idea of short circuit evaluation: stop as soon as the result is known



The only conditions evaluated are those actually needed.

Implementation of short-circuit evaluation

- a && b && c   a and b and c

```
if a==false then return a          // no eval of b,c
else if b==false then return b     // no eval of c
else return c
```

- a || b || c   a or b or c

```
if a==true then return a           // no eval of b,c
else if b==true then return b      // no eval of c
else return c
```

Examples of Unsafe Programs

Unsafe in Pascal (because of short-circuit evaluation of logic expression that is not mandatory):

limit = the largest allowable index into the list array

```
while (i <= limit) and (list[i] < > value) do i:=i+1;
i:=0;  repeat i:=i+1 until (i>limit) or (list[i]=value) ;
```

Safe in C:

```
while(i <= limit  &&  list[i] != value) ++i;
i=-1; do ++i; while(i<=limit  &&  list[i] != value);
```

Precedence of operators

- APL uses one level of precedence
- Pascal: 5 levels of precedence
- Fortran 77: 10 levels of precedence
- Fortran 90: 15 levels of precedence
- C++: 17 levels of precedence
- Too big or too small number of levels are inconvenient. A comfortable number of levels is somewhere between 7 and 10

A typical operator precedence (F77)

1. Power **
2. Numeric * or / .
3. Numeric Unary + or - .
4. Numeric Binary + or - .
5. Character/string concatenation //
6. Relational .EQ., .NE., .LT., .LE., .GT., .GE.
7. Logical .NOT.
8. Logical .AND.
9. Logical .OR.
10. Logical .XOR., .EQV., .NEQV.

PROGRAM PERFORMANCE

Program Performance Indicators

- For each program component: produce results with minimum use of resources
- Resources:
  - **Processor time** (high speed)
  - **Memory size** (compact data + compact code)
  - **Disk file accesses** (minimum access activity)
  - **Disk file size** (minimize the file size)

Performance Awareness

- Performance standards should be supported at each step of program life cycle: design, implementation, testing, maintenance
- The best way to satisfy performance standards is to promote performance awareness even for smallest programs

Program development cost

- Higher performance usually means more development effort and higher development cost
- Higher performance is usually achieved with more procedural details
- Performance awareness is inexpensive but it helps avoiding costly mistakes caused by brute force programming and ignorance of basic mechanisms of program execution

Balancing performance and cost

- Controlled reduction of performance caused by an organized effort to keep development time and cost within reasonable limits is an acceptable and frequent management strategy
- Reduction of performance due to insufficient performance awareness is neither professional nor acceptable

Algorithmic performance problems - order of complexity

- Algorithms can have different order of time complexity
- If a problem (e.g. sort) can be solved using several orders of complexity, then it is not acceptable to use higher order of complexity if there is a solution that uses a lower order of complexity
- Bubble sort: **$O(n^2)$**
- Quicksort: **$O(n \log n)$** and $n \log n < n^2$

Performance of Algorithms

Criteria:

- Maximum speed
- Minimum memory consumption
- Minimum complexity
- Readability
- Reliability (for wide range of input data)

Big-oh Notation

Performance of algorithms is primarily related to the consumption of processor time. The majority of algorithms have one or more parameters that can be interpreted as the problem size (or more precisely, the size of data structures that are related to the program run time). This parameter is usually denoted n, and the run time is $T(n)$. The notation

$$T(n) = O(f(n)) \qquad \text{(f is any function of n)}$$

means the following:

$$T(n) < c \cdot f(n)\ , \qquad c = \text{const.}, \qquad n >> 1$$

$O(f(n))$ = order of time complexity: the program run time is proportional to f(n). Functions T(n) and f(n) have the same growth rate for large values of n.

Example: Sum of an Array

```
for(sum=i=0; i<n; i++)
   sum += a[i];
```

n = problem size

Run time: $T(n) = c \cdot n$, c=const.

$T(n) = O(n)$, i.e. the run time is proportional to n

Classification of Algorithms
$T(n)=O(f(n))$

$f(n)$	Name of algorithm
Const.	Constant
$\log_2(n)$	Logarithmic
n	Linear
$n \log_2(n)$	Linear-logarithmic
n^2	Quadratic
n^k $(k\geq 2)$	Power function
k^n $(k>1)$	Exponential
$n!$	Factorial

Polynomial Algorithms

Non Polynomial Algorithms

Basic Goals of Developing Algorithms

Since $T(n) \approx c\, f(n)$, try to solve problems in minimum time. This means satisfying two goals:

- Develop an algorithm that has the lowest time complexity.
- At any order of complexity $f(n)$ try to develop an algorithm that has the minimum value of c.

Constant Algorithms: O(const)

```
void swap(int& x, int& y)
{ int t;
  t=x; x=y; y=t;
}
// Is it possible to swap without using t?
void swap(int& x, int& y)  // Is this program
{                          // Correct or
     x=x+y; y=x-y; x=x-y;  // incorrect?
}                          // Good or bad?
```

It Is a Bad Program! Reasons:

- It is tricky (needs some effort for understanding)
- It works only for data types that support addition and subtraction (int, char, float, double)
- It can cause rounding errors for float and double
- It is not safe: it causes overflow whenever the sum of two objects is greater than the maximum allowed value
- It does not work for arbitrary objects
- Avoiding t yields a negligible benefit

Constant Algorithms: O(const)

```
double sec(void)
{ return double(clock( ))/CLOCKS_PER_SEC; }
```

```
double urn(void)
{ return double(rand( ))/RAND_MAX; }
```

Computation of $x = \sqrt{A}$

$$x = \sqrt{A}$$

$$x^2 = A$$

$$x = A / x$$

$$2x = x + A / x$$

$$x = (x + A / x) / 2$$

Algorithm for Computing sqrt

Set an initial nonzero value of x. Then repeat the statement

$$x = \frac{1}{2}\left(x + \frac{A}{x} \right)$$

If this iteration generates a constant value of x, then this value must be

$$x = \sqrt{A}$$

This works well, and the number of iterations is rather small.

Constant Algorithms - sqrt: O(const)

```
Xnew = A;
do
{
     Xold = Xnew;
     Xnew = (Xold + A/Xold)/2;
} while (Xnew != Xold);
```

T(A) is approximately constant for any A

Computing sqrt

Iteration	Xold	Xnew
1	4	2.5
2	2.5	2.05
3	2.05	2.00061
4	2.00061	2

Implementation of x=sqrt(A) in the standard C library (math.h)

1. Normalization: A = a*2^e, a$\in$[0.5, 1)
2. Goal: compute x = sqrt(A)=sqrt(a) * $2^{e/2}$
3. Approximate y=sqrt(a) using a polynomial y=(c1*a+c2)*a+c3, c1,c2,c3 = constants
4. Improve the accuracy using 3 iterations of Newton’s (divide and average) method y=0.5*(y+a/y)

Linear Algorithms: O(n)
Array Initialization

```
void RandomArray(int a[ ], int n)
{  int i;
   randomize();
   for(i=0; i<n; i++)
      a[i] = rand( )%100;
}
```

$T(n) = O(n)$

Linear Algorithm: Maximum Value

```
int max(int a[], int n)
{
    int m = a[--n];
    while(--n >= 0)
        if(a[n] > m) m = a[n];
    return m;
}
```

$T(n) = O(n)$

Linear Algorithm: Mean Value

```
double Mean(int a[], int n)
{  int i; double s = 0.0;
   for(i=0; i<n; i++) s += a[i];
   return s/n;
}
```

T(n) = O(n)

Linear Algorithm: Factorial

```
int f(int n)
{   return (n<1) ? 1 : (n*f(n-1));
}
int F(int n)
{   int f=1, i;
    for(i=2; i<=n; i++) f *= i;
    return f ;
}
```

T(n) = O(n)   (both iterative and recursive)

Linear Algorithm: Two Variables

```
void merge (int a[ ], int na, int b[ ], int nb, int c[ ], int& nc)
{ int  i, j ;
   nc = i = j = 0 ;
  while (i<na && j<nb) c[nc++] = (a[i]<b[ j]) ? a[ i++] : b[ j++];
  while (i<na)         c[nc++] = a[ i++ ] ;
  while (j<nb)         c[nc++] = b[ j++ ] ;
}
```

Run time: T(na, nb) = c • nc = c(na+nb)

Logarithmic Algorithms: $O(\log_2(n))$

1. In each iteration half the size of data.
2. Continue halving until the size of data is 1.
3. Perform the final processing with the unit size data.

n = initial size of data

k = iteration ($k = 1, 2, 3, \ldots$).

$n/2^k$ = size of data in iteration k. At end $n/2^k = 1$.

From $n = 2^k$ it follows $k = \log_2(n)$ = total number of iterations

Run time $T(n) = ck = c \log_2(n) = O(\log_2(n))$; c = const.

Logarithmic Algorithm: Binary Search

```
int bsearch(int v[], int n, int x)
{ int low, high, mid ;
  low=0 ; high = n-1 ;
  while (low <= high)
  {  mid = (low + high) / 2 ;
     if      (x < v[mid])  high = mid-1 ;
     else if (x > v[mid])  low  = mid+1 ;
     else return mid ;
  }
  return -1 ;  /* no match */
}
```

Recursive Binary Search

```
int bsearch(int v[ ], int low, int high, int x)
{int mid = (low + high) / 2;
 if(low>high) return -1;
 if(x<v[mid]) return bsearch(v, low, mid-1, x);
 if(x>v[mid]) return bsearch(v, mid+1, high, x);
 return mid;
}
```

Linear-Logarithmic Algorithms $O(n \log_2(n))$

1. In each iteration half the size of data.
2. In each iteration process all n existing data items
3. Continue halving and processing until the size of data is 1.

n = initial size of data

k = iteration (k = 1, 2, 3, …).

$n/2^k$ = size of data in iteration k. At end $n/2^k = 1$.

From $n = 2^k$ it follows $k = \log_2(n)$ = total number of iterations

Run time $T(n) = c\, n\, k = c\, n \log_2(n) = O(n \log_2(n))$; c = const.

Linear-Logarithmic Algorithms: Quicksort

1. Decomposition of array: Select a middle component of an array and arrange the components of array so that those left of the middle component are less than or equal to the middle component, and those right of the middle component are greater than the middle component. This processes all n components and generates two subarrays.
2. Repeat the same decomposition process for each subarray.
3. Terminate this process when the size of all subarrays is 1. This occurs after $k = \log_2(n)$ iterations.
4. Run time $T(n) = c\ n\ k = c\ n \log_2(n) = O(n \log_2(n))$

Quicksort

```
void sort(int a[], int left, int right)
{ int i, mid;
  if(left >= right) return;
  for(mid=left, i=left+1; i <= right; i++)
    if(a[i] < a[left]) swap(a, ++mid, i);
  Swap(a, left, mid);
  sort(a, left, mid-1);
  sort(a, mid+1, right);
}
```

Quadratic Algorithms: $O(n^2)$

For each component of data set process all components of the data set:

```
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
      {Processing}
```

$T(n) = c \cdot n \cdot n = O(n^2)$;   c = const.

Quadratic Algorithm: Simple Select Sort

```
void sort( int a[], int n )
{ int i, j;
   for( i=0 ; i<n-1 ; i++ )
      for( j=i+1 ; j<n ; j++ )
         if( a[i] > a[j] ) swap(a[i], a[j]);
}
```

$T(n) \approx c(\text{number of executed if statements}) = c(1 + 2 + \ldots + n\text{-}1)$

$= cn(n-1)/2 = c(n^2-n)/2$

$T(n) = O(n^2)$

Quadratic Algorithm: Matrix Initialization

```
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
      a[i][j] = rand( );
```

$T(n) = c\ n^2 = O(n^2\ )$

Quadratic Algorithm: Matrix Transposition

```
for(i=0; i<n-1; i++)
   for(j=i+1; j<n; j++)
      swap(a[i][j] , a[j][i]);
```

$T(n) = c(1+2+\ldots+n-1) = c\,(n^2 - n) / 2 = O(n^2)$

Power Function Algorithm: Matrix Multiplication

```
for(i=0; i<n; i++)
     for(j=0; j<n; j++)
          for(c[i][j] = k = 0; k<n; k++)
               c[i][j] += a[i][k]*b[k][j];
```

$T(n) = c \bullet n \bullet n \bullet n = O(n^3)$;   c = const.

Power Function Algorithm: Nested Loops

```
for(i1=0; i1<n; i1++)
      for(i2=0; i2<n; i2++)
            . . . . . . . . . . .
            for(ik=0; ik<n; ik++)
                {Processing}
```

$T(n) = c \bullet n \bullet n \bullet \ldots \bullet n = O(n^k\ )$; c = const.

Exponential Algorithm: Fibonacci Numbers: 0, 1, 1, 2, 3, 5, 8, 13, …

```
int f(int n)
{
     return (n<2) ? n : f(n-1)+f(n-2) ;
}
```

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2)\,, \quad n > 1 \\
&= 1\,, \quad n = 1 \\
&= 0\,, \quad n = 0
\end{aligned}
$$

t denotes small time necessary for testing (n<2) and branching

$$T(n) = t + T(n-1) + T(n-2) \approx T(n-1) + T(n-2),\ n > 1$$

Solving T(n) = T(n-1) + T(n-2)

Suppose that $T(n) = cg^n, \quad g > 0$

$$cg^n = cg^{n-1} + cg^{n-2}$$

$$1 = g^{-1} + g^{-2}$$

$$g^2 - g - 1 = 0$$

$$g = \frac{1 \pm \sqrt{1+4}}{2}; \quad g = \frac{1+\sqrt{5}}{2} = 1.618$$

$$T(n) = c\,1.618^n = O(1.618^n)$$

Linear Fibonacci Numbers

```
int f(int n)
{ int i, zero=0, first=1, second=n;
  for(i=2; i<=n); i++)
  {  second = first + zero;
     zero = first;  first = second;
  }
  return second;
}
```

T(n) = c • n = O(n) ;   c = const.

Factorial Algorithms: O(n!)

Traveling salesman:

The salesman is in an initial city and has to visit n cities. Knowing the distances between each pair of cities find the shortest path connecting all n cities.

There are n! different paths, and each has a given distance. A trivial solution is to generate all n! path distances and select the minimum value.

Obviously $T(n) = c\ n! = O(n!)$

Such algorithm can be used only for very small values of n.

Probabilistic Cases: if-else

urn( ) = standard uniform random number $0 < \text{urn( )} < 1$

```
if(urn( ) < p)  {run time s}
else {run time t}
```

Average run time $T = ps + (1-p)t$

Probabilistic Cases: do-while

urn( ) = standard uniform random number $0 < urn( ) < 1$

```
do
   {run time t}  // n executions (average)
while(urn( ) < p)
```

The probability to stay in the loop = p

The probability of leaving the loop = 1-p

Exit condition: $n(1- p) = 1$ (there is only one exit)

The average run time $T = n\ t = t/(1 – p)$

Probabilistic Cases: while-do

urn( ) = standard uniform random number $0 < \text{urn}(\,) < 1$

```
while(urn( ) < p)
   {run time t}  // n executions (average)
```

The probability to stay in the loop = p

The probability of leaving the loop = $1-p$

Exit condition: $(n+1)(1-p) = 1$ (there is only one exit)

$n = 1/(1-p)-1 = p/(1-p)$

The average run time $T = n\,t = tp/(1 – p)$

The Brute Force Approach

Definition: The brute force approach denotes cases where the efficiency of algorithm is very low, but the extreme speed of computers still provides solutions in acceptable time.

Brute force approach is a kind of intellectual bankruptcy and it is not acceptable in professional software engineering products.

Brute force approach is acceptable as a temporary "quick and dirty" solution of some problems, or as an intermediate step towards the quality algorithmic solutions.

Brute Force and Unacceptable Performance

- Brute force solutions usually have a wrong (excessive) time complexity of solution.
- Consider a sequence of time complexities:

  O(const), O(log(n)), O(n), O(n log(n)), O(n^2),…
- E.g., it is not acceptable to use O(n^2) solution in cases where it is possible to use an O(n) solution
- The best way to understand brute force approach is to see examples of brute force algorithms.

A Typical Brute Force Example

Find all integers a, b, and c that are less than N and satisfy the condition $a^2+b^2 = c^2$.

Examples: $3^2 + 4^2 = 5^2$

$6^2 + 8^2 = 10^2$

Let us study the case N = 100.

Results

```
[ 1]  3  4  5  [ 2]  5 12 13  [ 3]  6  8 10  [ 4]  7 24 25  [ 5]  8 15 17
[ 6]  9 12 15  [ 7]  9 40 41  [ 8] 10 24 26  [ 9] 11 60 61  [10] 12 16 20
[11] 12 35 37  [12] 13 84 85  [13] 14 48 50  [14] 15 20 25  [15] 15 36 39
[16] 16 30 34  [17] 16 63 65  [18] 18 24 30  [19] 18 80 82  [20] 20 21 29
[21] 20 48 52  [22] 21 28 35  [23] 21 72 75  [24] 24 32 40  [25] 24 45 51
[26] 24 70 74  [27] 25 60 65  [28] 27 36 45  [29] 28 45 53  [30] 30 40 50
[31] 30 72 78  [32] 32 60 68  [33] 33 44 55  [34] 33 56 65  [35] 35 84 91
[36] 36 48 60  [37] 36 77 85  [38] 39 52 65  [39] 39 80 89  [40] 40 42 58
[41] 40 75 85  [42] 42 56 70  [43] 45 60 75  [44] 48 55 73  [45] 48 64 80
[46] 51 68 85  [47] 54 72 90  [48] 57 76 95  [49] 60 63 87  [50] 65 72 97
```

How many tests are necessary to generate these 50 results?

Brute Force Solution #1

```
long int a,b,c, tests=0;
for(a=1; a<N; a++)
   for(b=1; b<N; b++)
      for(c=1; c<N; c++)
      { ++tests;
        if(a*a+b*b == c*c)
          cout<< a <<‘ ‘<< b <<‘ ‘<< c <<‘\n’;
      }
cout<< "\nTotal number of tests = " << tests;
```

tests = 970299

Is this program correct? – Yes
Is this program good? No! Why?

- Test cases include both a<b and a>b; consequently, each result is generated twice.
- If $a^2 + b^2 = c^2$ then c>a and c>b. However, the program tests also the cases where c<a or c<b.
- In a triangle c < a+b. However, the program tests also the cases where c > a+b.
- The number of tests is unnecessarily too big.

Brute Force Solution #2
All Combinations Where a < b < c

```
for(a=1; a<N; a++)
    for(b=a+1; b<N; b++)
        for(c=b+1; c<N; c++)
            // tests
```

tests = 156849

Brute Force Solution #3
$a^2 + b^2 = c^2 < N^2; \;\; b^2 < N^2 - a^2$

```
for(a=1; a<N; a++)
    for(b=a+1; b < sqrt(N*N-a*a); b++)
        for(c=b+1; c<N; c++)
               // tests
```

tests = 148421

Brute Force Solution #4
a < b < c < a + b

```
for(a=1; a<N; a++)
    for(b=a+1; b < sqrt(N*N-a*a); b++)
        for(c=b+1; c<N && c<a+b; c++)
              // tests
```

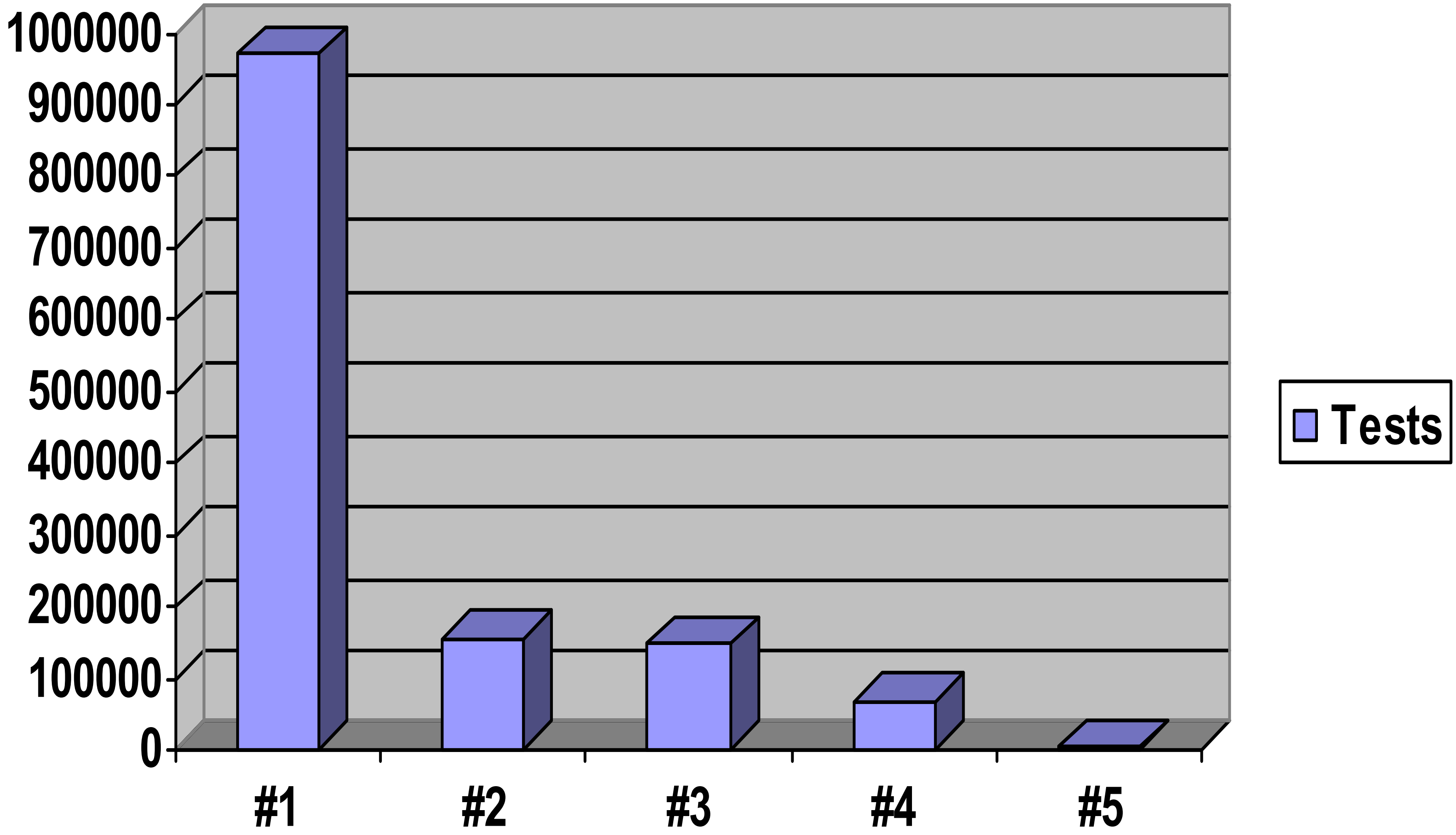
tests = 68796

Brute Force Solution #5
Only Combinations of a and b>a

```
tests = 0;
for(a=1; a<N; a++)
   for(b=a+1; b < sqrt(N*N-a*a); b++)
   { ++tests;
     c = sqrt(a*a + b*b); // c is integer
     if(a*a+b*b == c*c) cout << ..... ;
   }
```

tests = 3840

Reducing the Number of Tests



Find 3 Largest Elements of a[0..n-1]

- a[n] = array containing *positive* integers
- $n \geq 3$
- Problem: find 3 largest components in the array a[n] : first ≥ second ≥ third
- Example: a[8] = {3, 4, 5, 0, 5, 7, 1, 2}

  first = 7

  second = 5

  third = 5

Two approaches to Max3

- Wrong solution:

  Sort the whole array using bubble sort and then return the first three components

Complexity: O(n^2)

- Correct solution:

for(i=0; i<n; i++)

  Use a[i] to update the values of variables MAX, MID and MIN;

  (MIN ≤ MID ≤ MAX)

Complexity: O(n)

Max3: Unacceptable Solution O(n^2) (sort and take three largest values)

```
void Max3(int a[ ], int n,
          int& first, int& second, int& third)
{ int i, j;
  for( i=0 ; i<n-1 ; i++ )   // select sort
     for( j=i+1 ; j<n ; j++ )
        if( a[i] < a[j] ) swap(a[i], a[j]);
  first=a[0]; second=a[1]; third=a[2];
}
```

Max3 : Questionable Solution O(n)

```
void max3c(int a[ ], int n,
           int& first, int& second, int& third)
{ int i, j;
  for( i=0 ; i<3 ; i++ )    // truncated sort
     for( j=i+1 ; j<n ; j++ )
        if( a[i] < a[j] ) swap(a[i], a[j]);
  first=a[0]; second=a[1]; third=a[2];
}
```

a[ ] is modified. T(n) = c((n-1)+(n-2)+(n-3)) = c(3n-6) = O(n)

Max3: Correct Solution O(n)

```
void Max3(int a[ ], int n, int& first, int& second, int& third)
{ int i;
   first = second = third = 0;          //   first >= second >= third
   for(i=0; i<n; i++)
      if(a[i]>=first)              {third = second;  second = first;  first = a[i];}
      else if(a[i]>=second)  {third = second;  second = a[i];}
      else if(a[i]>third)        third = a[i] ;
}
```

This solution is faster than the previous O(n) solution. In addition, there is no modification of array a[ ]. If we remove "=" then we get three largest different values

Insert component x in sorted array a[ ], so that the array remains sorted

Unacceptable solution: expand array and then sort

```
void insert(int a[], int& n, int x)
{
  a[n++] = x;
  sort(a, n);
}
```

If **sort** is O(n^2) then **insert** is also O(n^2) .

Correct Insert Program: O(n)

```
void insert(int a[], int& n, int x)
{ int i = n-1;
  while(i>=0 && a[i]>x)
  {
     a[i+1] = a[i];
    i--;
  }
  a[i+1] = x; n++;
}
```

Computation of Power: 2^8

- Wrong solution:

```
p=1;
for(i=1; i<=8; i++)
   p *= 2;
```

Complexity: O(n)

- Correct solution:

```
p=2;
for(i=0; i<3; i++)
   p *= p;
```

Complexity $O(\log_2(n))$

Data conversion overhead problems

- Automatic conversion of data types (e.g. int to double) takes conversion time
- Conversion time is NOT negligible and can sometimes accumulate and cause significant performance reduction
- Unnecessary type conversions should be avoided, and performance effects should be experimentally verified

Numerical data types

- Mixed numerical expressions may contain char, short, int, long, float, double, long double
- In C and similar languages real constants (1.23) are by definition double
- Float constants must be written with suffix F (1.23F)
- Length of integers: short <= int <= long
- Length of real numbers: float < double

Data Conversion Rule

- If a numerical expression contains different data types (mix of simple and more complex) then the simple data types will first be converted to more complex data types, and then the operations will be performed with the more complex data.
- Automatic conversion consumes time and can be avoided by combining same data types

Order of conversion

char → short → int → long → float → double → long double

- Short = “simplest”
- Double = “most complex”

- Sometimes short integers (1 byte) can be stored in variables of type char (either signed char or unsigned char).

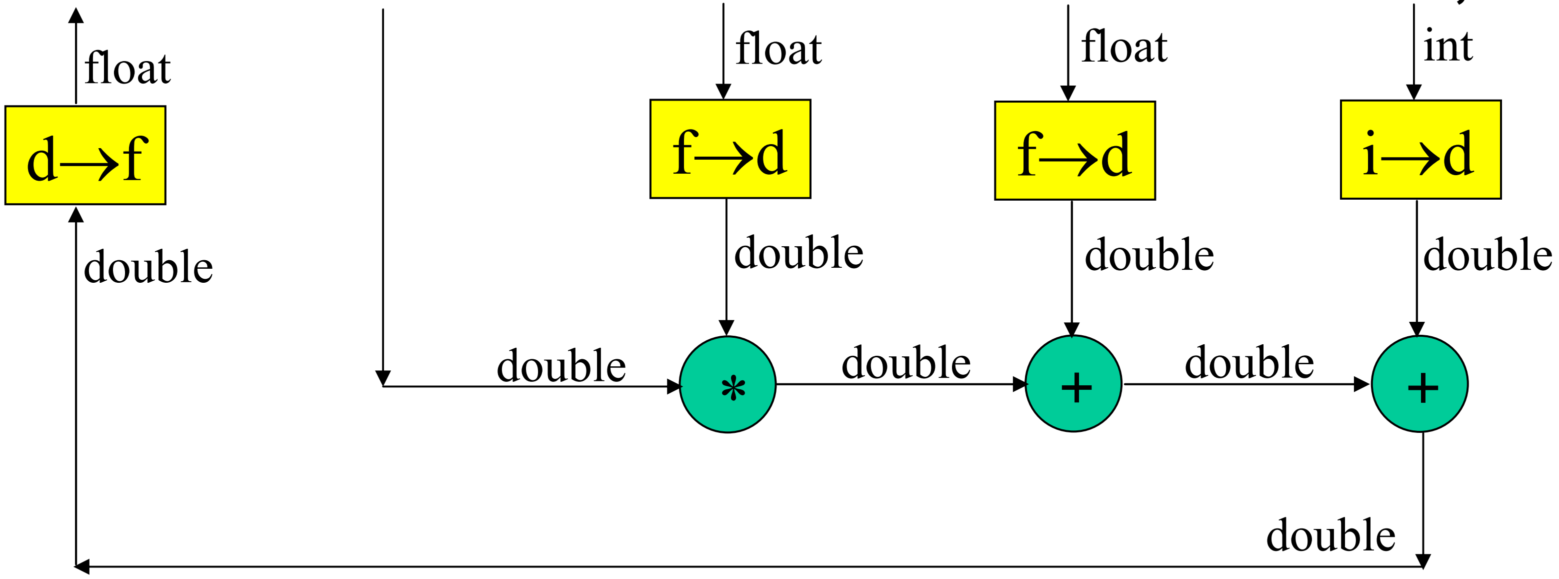
An example of conversion

- TwoPi = 2 * 3.1415926
- In this case the integer 2 will first be converted to double constant 2.0 and then the multiplication will be performed with two double precision numbers and will generate the double precision result.
- It is better to avoid conversion by writing

  TwoPi = 2. * 3.1415926

Type Conversion Overhead Example

```
float a,b,c;
a = 2. * b + c + 1 ;
```



Type conversions consume processor time!

a = 2.F * b + c + 1.F ; // No type conversions

Integer/Float/Double Conversions

- Conversions can go in two directions:
  - int →float →double →long double
  - long double → double → float →int
- Some conversions are fast, some are slow, and there is no rule that holds for all compilers
- A natural expectation is that the time for arithmetic operations satisfies t(int) < t(float) < t(double) < t(long double). Warning: there are cases where this is not true. Always make experiments to check performance!

TYPE CONVERSION OVERHEAD

Example #1

```
#include<iostream.h>
#include<time.h>

double msec(void) {return (1000.*clock())/CLOCKS_PER_SEC;}

void main(void)
{ cout << "THE COST OF TYPE CONVERSION IN VISUAL C++ (Release)";
  int    a=0, b=12345,  c=54321,   n = 10000000, k=n;
  double A=0, B=12345, C=54321,  t1,t2,t3;

  t1 = msec();
       while(n--) A += 2.* B + C + 1.; //  Computation without conversion
  t2 = msec();                         //  t2-t1 = time for computation only
       while(k--) a += 2.* b + c + 1;  //  Computation with type conversions
  t3 = msec();                         //  t3-t2 = conversion and computation

  cout << "\nRun time without type conversions = " << (t2-t1) << " msec"
       << "\nRun time with type conversions    = " << (t3-t2) << " msec"
       << "\nTime spent for type conversions   = " << (t3-2*t2+t1) << " msec"
       << "\n(Compute+convert)/Compute         = " << (t3-t2)/(t2-t1) << endl;
}
```

```
THE COST OF TYPE CONVERSION IN VISUAL C++ (Release)
Run time without type conversions = 31 msec
Run time with type conversions    = 359 msec
Time spent for type conversions   = 328 msec
(Compute+convert)/Compute         = 11.5806
```

RELEASE

TYPE CONVERSION OVERHEAD

Example #2

```
#include<iostream.h>
#include<time.h>

double msec(void) {return (1000.*clock())/CLOCKS_PER_SEC;}

void main(void)
{ cout << "THE COST OF TYPE CONVERSION IN VISUAL C++ (Debug)";
  int    a=0, b=12345,  c=54321,   n = 10000000, k=n;
  double A=0, B=12345, C=54321,  t1,t2,t3;

  t1 = msec();
       while(n--) A += 2.* B + C + 1.; //  Computation without conversion
  t2 = msec();                         //  t2-t1 = time for computation only
       while(k--) a += 2.* b + c + 1;  //  Computation with type conversions
  t3 = msec();                         //  t3-t2 = conversion and computation

  cout << "\nRun time without type conversions = " << (t2-t1) << " msec"
       << "\nRun time with type conversions    = " << (t3-t2) << " msec"
       << "\nTime spent for type conversions   = " << (t3-2*t2+t1) << " msec"
       << "\n(Compute+convert)/Compute         = " << (t3-t2)/(t2-t1) << endl;
}
```

```
THE COST OF TYPE CONVERSION IN VISUAL C++ (Debug)
Run time without type conversions = 187 msec
Run time with type conversions    = 328 msec
Time spent for type conversions   = 141 msec
(Compute+convert)/Compute         = 1.75401
```

DEBUG

TYPE CONVERSION OVERHEAD

Example #3

```
#include<iostream.h>
#include<time.h>

double msec(void) {return (1000.*clock())/CLOCKS_PER_SEC;}

void main(void)
{ cout << "THE COST OF TYPE CONVERSION IN VISUAL C++ (Debug)";
  int    a, b=12345,  c=54321,   n = 10000000, k=n;
  double A, B=1.2345, C=5432.1,  t1,t2,t3;

  t1 = msec();                            //  CONVERSION FROM INTEGER TO DOUBLE
       while(n--) A = 2.* B + C + 1.; //  Computation without conversion
  t2 = msec();                            //  t2-t1 = time for computation only
       while(k--) a = 2.* b + c + 1;  //  Computation with i->d type conversions
  t3 = msec();                            //  t3-t2 = conversion and computation

  cout << "\nRun time without type conversions = " << (t2-t1) << " msec"
       << "\nRun time with type conversions    = " << (t3-t2) << " msec"
       << "\nTime spent for type conversions   = " << (t3-2*t2+t1) << " msec"
       << "\nConversions/computation ratio     = " << (t3-2.*t2+t1)/(t2-t1) << endl;
}
```

```
THE COST OF TYPE CONVERSION IN VISUAL C++ (Debug)
Run time without type conversions = 46 msec
Run time with type conversions    = 360 msec
Time spent for type conversions   = 314 msec
Conversions/computation ratio     = 6.82609
```

DEBUG

Performance Issues Related to Matrix Operations

Storage of matrices

- According to language standard, matrices can be stored in memory in three ways:
  - Row-wise (e.g. C, C++, …)
  - Column-wise (e.g. Fortran, BLISS, …)
  - Unspecified (e.g. Pascal, Basic, …); in this case compiler designers select the storage method

Example of matrix storage method

$$Math \quad notation: \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

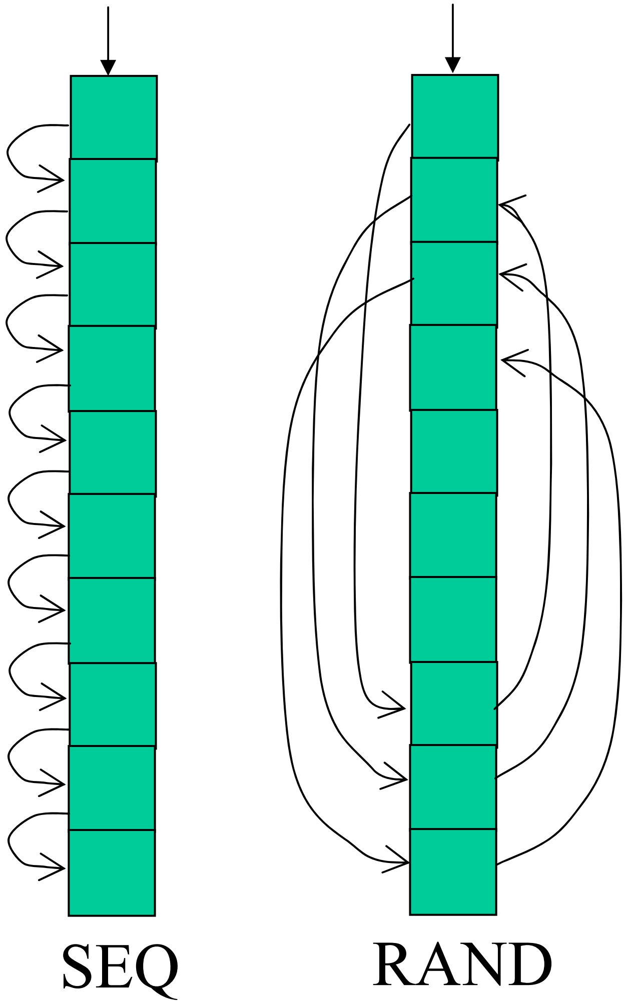
$$Row-wise\ storage: \quad [11 \quad 12 \quad 13 \quad 21 \quad 22 \quad 23]$$

$$Column-wise\ storage: [11 \quad 21 \quad 12 \quad 22 \quad 13 \quad 23]$$

Each matrix is stored in memory as a sequence of numbers

Matrix access/processing modes

- **Sequential** (processor accesses numbers in successive memory locations)
- **Random** (processor accesses numbers jumping from location to location)
- Sequential access is fast
- Random access is slow



Why is the random access slow?

- **Addressing mode**: inefficient indexed addressing (each access causes adding index [from memory] to the address of the first element)
- **Cashing**: high cache miss frequency
- **Paging**: high page fault frequency (for large matrices)

Why is the sequential access fast?

- **Addressing mode**: the most efficient register autoincrement addressing mode (address is kept in a register that is incremented simultaneously with the current operation)
- **Cashing**: high cache hit frequency
- **Paging**: minimum page fault frequency (for large matrices)

Example of matrix addition

```
double t1,t2,t3;
int M[N][N], sum=0;
int i,j,k,n;
- - - - - - - - - - - - - - -
t1 = sec();
    for(sum=0, k=0; k<K; k++)
     for(i=0; i<n; i++)
        for(j=0; j<n; j++)
           sum += M[i][j];  // Row-wise access
t2 = sec();
  for(sum=0, k=0; k<K; k++)
     for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            sum += M[j][i]; // Column-wise access
t3 = sec();
```

Matrix addition results for Windows

```
Visual C++ / Release version, Dell Dimension 8200
=================================================
Matrix   Row-wise    Column-wise    Column/Row
size       [sec]       [sec]        ratio
=================================================
  50       0.188       0.172        0.914894
 100       0.625       0.546          0.8736
 150       1.375       6.172         4.48873
 200       2.344      12.984         5.53925
 250      11.907      22.937         1.92635
 300      17.875          36         2.01399
 350      25.297      49.031         1.93821
 400      34.157      63.609         1.86225
 450      26.344      80.468         3.05451
 500      29.438      98.344         3.34072
```

Matrix addition results for Cygwin

```
Cygwin g++ -O3 , Dell Dimension 8200
=================================================
Matrix   Row-wise    Column-wise    Column/Row
size       [sec]       [sec]        ratio
=================================================
  50       0.125       0.313           2.504
 100       0.422        1.25         2.96209
 150        0.89       9.031         10.1472
 200       1.563      17.328         11.0864
 250       2.359      30.266           12.83
 300       3.391      47.922         14.1321
 350       4.562      65.329         14.3203
 400       5.953      85.374         14.3413
 450         7.5     107.719         14.3625
 500       9.219     136.156         14.7691
```

Matrix multiplication

- Program consists of three nested loops
- There are 3!=6 permutations of 3 loops and all of them generate the same result, but the time is not the same
- Permutations of loops cause different memory access patterns
- Fundamental question: What is the best permutation???

Example of matrix multiplication

```
double t1,t2, Tijk,Tikj,Tjik,Tjki,Tkij,Tkji, Tmin;
double Sijk,Sikj,Sjik,Sjki,Skij,Skji, Smin;
double a[N][N], b[N][N], c[N][N];
int i,j,k,n,m;

t1 = sec(); //////////////////////////////////////////  jki - version
     for(m=0; m<M; m++)for(j=0; j<n; j++)for(k=0; k<n; k++)for(i=0; i<n; i++)
          c[i][j] += a[i][k]*b[k][j];
t2 = sec(); Tjki = t2-t1; cout << setw(10) << t2-t1;

t1 = sec(); ////////////////////////////////////////// kji - version
     for(m=0; m<M; m++)for(k=0; k<n; k++)for(j=0; j<n; j++)for(i=0; i<n; i++)
          c[i][j] += a[i][k]*b[k][j];
t2 = sec(); Tkji = t2-t1; cout << setw(10) << t2-t1;

t1 = sec(); ////////////////////////////////////////// ikj - version
     for(m=0; m<M; m++)for(i=0; i<n; i++)for(k=0; k<n; k++)for(j=0; j<n; j++)
         c[i][j] += a[i][k]*b[k][j];
t2 = sec(); Tikj = t2-t1; cout << setw(10) << t2-t1;
```

Matrix multiplication results for Windows VCPP

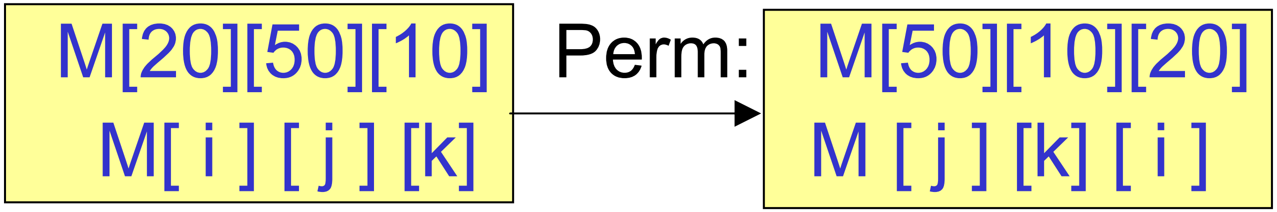
```
Run times and relative run times for various permutations of indices
=================================================================
 n      Tjki      Tkji      Tikj      Tkij      Tijk      Tjik
=================================================================
 50     0.094     0.109     0.047     0.063     0.047     0.046
          2.04      2.37      1.02      1.37      1.02         1
 75      0.36     0.344     0.171     0.172     0.141     0.141
          2.55      2.44      1.21      1.22         1         1
100      1.34      1.36      0.39     0.438     0.344      0.39
           3.9      3.95      1.13      1.27         1      1.13
125      2.67      2.69     0.781     0.812     0.672     0.813
          3.98         4      1.16      1.21         1      1.21
150      4.89      4.75      1.41      2.03      1.19      2.25
          4.12         4      1.19      1.71         1       1.9
175      8.39       8.8      3.13         5      4.09      6.06
          2.69      2.81         1       1.6      1.31      1.94
200      13.3      16.5      4.64      8.02      9.05      10.5
          2.86      3.55         1      1.73      1.95      2.26

Total run time and relative run time for all permutations
Run:       31      34.5      10.6      16.5      15.5      20.2
Rel:     2.94      3.27         1      1.57      1.47      1.91
```

Matrix multiplication results for Cygwin g++

```
Run times and relative run times for various permutations of indices
=================================================================
 n        Tjki      Tkji      Tikj      Tkij      Tijk      Tjik
=================================================================
 50      0.11     0.093     0.094     0.047     0.187     0.188
         2.34      1.98         2         1      3.98         4
 75     0.391     0.375     0.281     0.187     0.688     0.719
         2.09      2.01       1.5         1      3.68      3.84
100      1.44      1.39      0.64     0.453      1.64      1.69
         3.17      3.07      1.41         1      3.62      3.73
125      2.84      2.83      1.25     0.875      3.22      3.34
         3.25      3.23      1.43         1      3.68      3.82
150      5.25      5.12      2.28      2.45      5.67       6.2
          2.3      2.25         1      1.08      2.49      2.72
175      9.16      9.59      3.89      5.52      9.89      10.5
         2.35      2.47         1      1.42      2.54      2.71
200        14      16.6      5.66      8.67      15.6      15.5
         2.48      2.93         1      1.53      2.76      2.74

Total run time and relative run time for all permutations
Run:     33.2        36      14.1      18.2      36.9      38.2
Rel:     2.35      2.55         1      1.29      2.62      2.71
```

A general method for performance optimization of matrix operations

- Assume a matrix with **n** indices: M[i] [j]…[k]
- It is possible to make **n!** formal permutations of indices in all expressions that use the matrix. E.g.:
- Original:



M[20][50][10]

M[i] [j] [k]

Perm:

M[50][10][20]

M [j] [k] [i]

- Permutations change memory access patterns without changing the algorithm
- One of these n! permutations yields the best memory access pattern and the fastest program

Permutations for 3D matrices

- M[10] [20] [30] ←→ M[i] [j] [k]
- M[10] [30] [20] ←→ M[i] [k] [j]
- M[20] [10] [30] ←→ M[j] [i] [k]
- M[20] [30] [10] ←→ M[j] [k] [i]
- M[30] [10] [20] ←→ M[k] [i] [j]
- M[30] [20] [10] ←→ M[k] [j] [i]
- One of these permutations yields the fastest program

PERFORMANCE ISSUES RELATED TO RECURSION AND COMPILING

Factorial Computation

```
int f(int n)
{ return (n<2) ? 1 : n*f(n-1);
}
```

$T(n) = C \bullet n \;\; = O(n) \; ; \quad C = \text{const.}$

```
 int f(int n)
 { int i, p=1;
   for(i=2; i<=n; i++) p *= i;
   return p;
 }
```

$T(n) = c \bullet n \;\; = O(n) \; ; \;\; c = \text{const.};$

$c < C$ (Visual C++: $1.8 < C/c < 3.34$)

DEBUG VERSION

```
#include<iostream.h>
#include<time.h>
double msec(void) {return (1000.*clock())/CLOCKS_PER_SEC;}
int r(int n) { return (n<2) ? 1 : n*r(n-1);}
int f(int n)
{ int i, p=1;
  for(i=2; i<=n; i++) p *= i;
  return p;
}

void main(void)
{ cout << "THE COST OF RECURSION IN VISUAL C++ (Debug)";
  int    n = 10000000, k=n;
  double t1,t2,t3;
  t1 = msec();
       while(n--) r(8);   //  Recursive computation
  t2 = msec();            //  t2-t1 = time for recursive computation
       while(k--) f(8);   //  Iterative computation
  t3 = msec();            //  t3-t2 = time for iterative computation
  cout << "\nRecursive run time         = " << (t2-t1) << " msec"
       << "\nIterative run time         = " << (t3-t2) << " msec"
       << "\nRecursive/Iterative ratio  = " << (t2-t1)/(t3-t2)
       << "\nRecursive overhead (R-I)/I = " << 100.*((t2-t1)-(t3-t2))/(t3-t2) << " %"<<endl;
}
```

```
THE COST OF RECURSION IN VISUAL C++ (Debug)
Recursive run time         = 6593 msec
Iterative run time         = 1953 msec
Recursive/Iterative ratio  = 3.37583
Recursive overhead (R-I)/I = 237.583 %
```

RELEASE VERSION

```
#include<iostream.h>
#include<time.h>
double msec(void) {return (1000.*clock())/CLOCKS_PER_SEC;}
int r(int n) { return (n<2) ? 1 : n*r(n-1);}
int f(int n)
{ int i, p=1;
  for(i=2; i<=n; i++) p *= i;
  return p;
}

void main(void)
{ cout << "THE COST OF RECURSION IN VISUAL C++ (Release)";
  int    n = 10000000, k=n;
  double t1,t2,t3;
  t1 = msec();
       while(n--) r(8);   //  Recursive computation
  t2 = msec();            //  t2-t1 = time for recursive computation
       while(k--) f(8);   //  Iterative computation
  t3 = msec();            //  t3-t2 = time for iterative computation
  cout << "\nRecursive run time         = " << (t2-t1) << " msec"
       << "\nIterative run time         = " << (t3-t2) << " msec"
       << "\nRecursive/Iterative ratio  = " << (t2-t1)/(t3-t2)
       << "\nRecursive overhead (R-I)/I = " << 100.*((t2-t1)-(t3-t2))/(t3-t2) << " %"<<endl;
}
```

```
THE COST OF RECURSION IN VISUAL C++ (Release)
Recursive run time         = 687 msec
Iterative run time         = 375 msec
Recursive/Iterative ratio  = 1.832
Recursive overhead (R-I)/I = 83.2 %
```

TOTAL RUN TIME RATIO DEBUG / RELEASE = 8.05

Best use of languages and compilers

- For the same hardware, same OS, and the same algorithm, various languages can cause large differences in program performance
- For the same language, various compilers can yield large performance variations
- For the same compiler, the development version and the release version can cause large performance differences

NUMERICAL ACCURACY AS A PREPREQUISITE FOR PROGRAM QUALITY

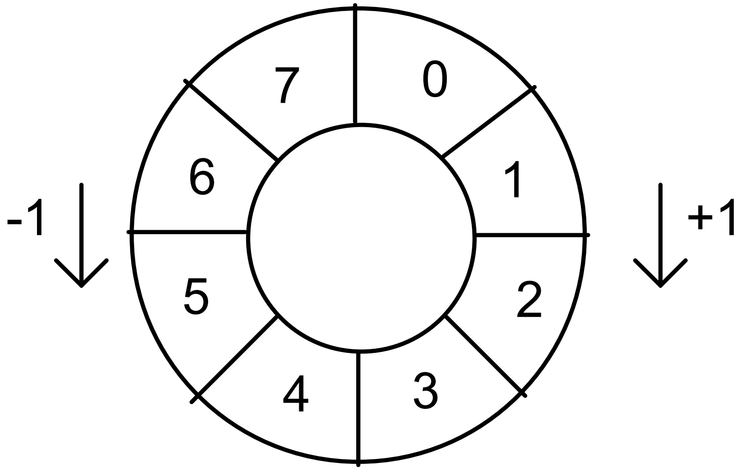
Basic Numeric Problems

- Integer overflow/underflow (range of integers)
- Real overflow/underflow (range of real numbers)
- Precision of real numbers (significant decimal digits)
- Problems of commutativity of addition

The Ring of Unsigned Integers

7+1=0

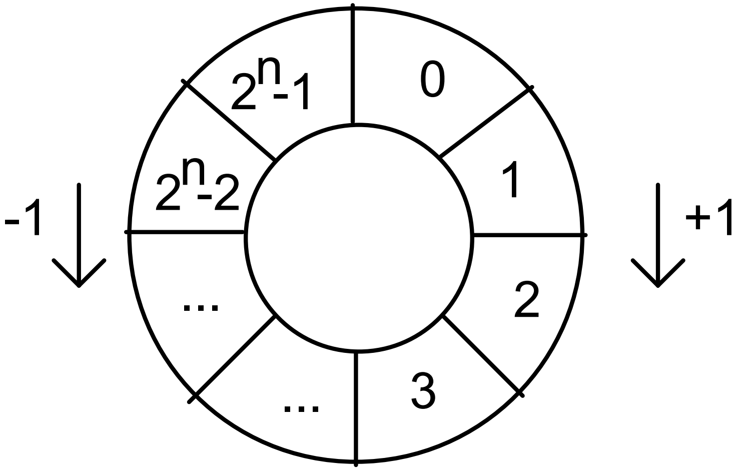
0-1=7



The case with 3 bits

$2^n-1+1=0$

$0-1=2^n-1$

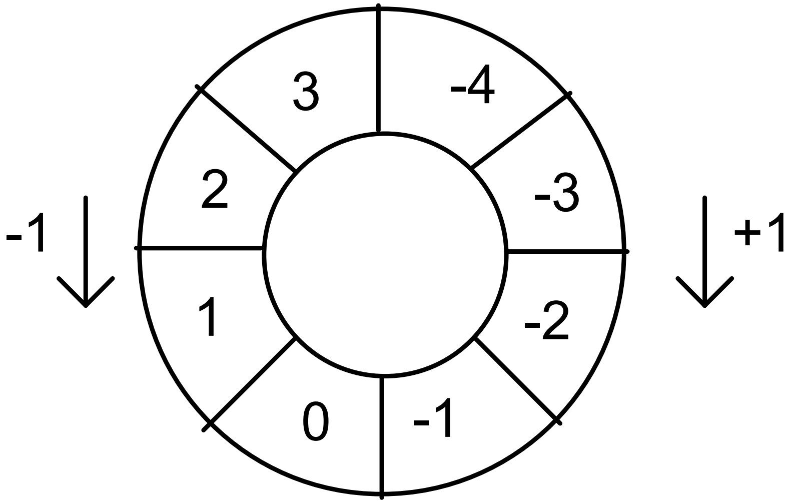


The general case with n bits

The Ring of Integers

$3+1=-4$

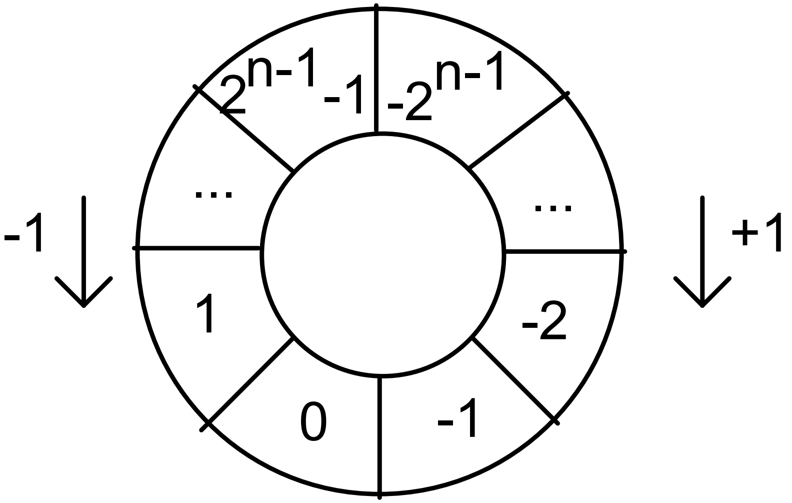
$-4-1=+3$



The case with 3 bits

$-2^{n-1} - 1 = 2^{n-1} - 1$

$2^{n-1} - 1 + 1 = -2^{n-1}$



The general case with n bits

Adding Single-byte (char) Integers

```
void main(void)
{
  { char add, sub, c=100, d=30;
    add = c+d;  sub = -c-d;
    cout << int(c)  << " + " << int(d) << " = " << int(add) << '\n';
    cout << int(-c) << " - " << int(d) << " = " << int(sub) << '\n';
  }
  { unsigned char add, sub, c=127, d=130;
    add = c+d;   sub = c-d;
    cout << unsigned(c) << " + " << unsigned(d) << " = " << unsigned(add) << '\n';
    cout << unsigned(c) << " - " << unsigned(d) << " = " << unsigned(sub) << '\n';
  }
}
```

```
Output for signed integers:
    100 + 30 = -126      MaxInt=2^7-1=127;       130: 127,-128,-127,-126
    -100 - 30 = 126      MinInt=-2^7=-128;       -130: -128, 127, 126
Output for unsigned integers:
    127 + 130 = 1        MaxInt=2^8-1=255;       257: 255, 0, 1
    127 - 130 = 253      MinInt=0;               -3: 0, 255, 254, 253
```

Adding Unsigned Integers

```
void main(void)
{unsigned short int p=65531, n=4, r, i;
 for(i=1; i<9; i++)
    cout << p << " + " << i << " =“ << setw(6)
         << (r = p+i)  << setw(15)
         << n << " - " << i << " =“ << setw(6)
         << (r = n-i)  << endl;
}
```

```
65531 + 1 = 65532              4 - 1 =     3
65531 + 2 = 65533              4 - 2 =     2
65531 + 3 = 65534              4 - 3 =     1
65531 + 4 = 65535              4 - 4 =     0
65531 + 5 =     0              4 - 5 = 65535
65531 + 6 =     1              4 - 6 = 65534
65531 + 7 =     2              4 - 7 = 65533
65531 + 8 =     3              4 - 8 = 65532
```

Adding Short Integers

```
void main(void)
{short int p=32763, n=-32764, r, i;
 for(i=1; i<9; i++)
  cout <<p<<" + "<<i<< " ="<<setw(7) << (r = p+i) <<setw(15)
       <<n<<" - "<<i<< " ="<<setw(7) << (r = n-i) <<endl;
}
```

```
32763 + 1 =  32764         -32764 - 1 = -32765
32763 + 2 =  32765         -32764 - 2 = -32766
32763 + 3 =  32766         -32764 - 3 = -32767
32763 + 4 =  32767         -32764 - 4 = -32768
32763 + 5 = -32768         -32764 - 5 =  32767
32763 + 6 = -32767         -32764 - 6 =  32766
32763 + 7 = -32766         -32764 - 7 =  32765
32763 + 8 = -32765         -32764 - 8 =  32764
```

Adding Integers

```
void main(void)
{ int p=2147483643, n=-2147483644, i;  // long int -> same results
  for(i=1; i<9; i++)
    cout << p << " + " << i << " ="<< setw(12) << (p+i) << setw(15)
         << n << " - " << i << " ="<< setw(12) << (n-i) << endl;
}
```

```
2147483643 + 1 =  2147483644     -2147483644 - 1 = -2147483645
2147483643 + 2 =  2147483645     -2147483644 - 2 = -2147483646
2147483643 + 3 =  2147483646     -2147483644 - 3 = -2147483647
2147483643 + 4 =  2147483647     -2147483644 - 4 = -2147483648
2147483643 + 5 = -2147483648     -2147483644 - 5 =  2147483647
2147483643 + 6 = -2147483647     -2147483644 - 6 =  2147483646
2147483643 + 7 = -2147483646     -2147483644 - 7 =  2147483645
2147483643 + 8 = -2147483645     -2147483644 - 8 =  2147483644
```

Computing Binomial Coefficients

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}; \quad \binom{n}{0} = \frac{n!}{0!n!} = 1$$

Avoid computing factorials: they can cause an integer overflow!

$$\binom{10}{5} = \frac{10\cdot 9\cdot 8\cdot 7\cdot 6}{1\cdot 2\cdot 3\cdot 4\cdot 5} = 10/1*9/2*8/3*7/4*6/5; \quad \binom{n}{k} = \binom{n}{n-k}$$

Alternate multiplications and divisions prevent overflow.

```
int num, den, n, k, C=1;
for(num=n, den=1;  den<=k ; num-- , den++) C = C*num / den;
```

The Size of Binary Numbers

$$\underbrace{\boxed{1010111000111111010110001011 1}}_{b} \qquad \underbrace{\boxed{123456789}}_{d}$$

The total number of possible codes with b binary digits is 2^b. The total number of possible codes with d decimal digits is 10^d. If binary and decimal notations are equivalent then

$$2^b = 10^d \qquad | \quad \log$$

$$b \cdot \log 2 = b \cdot 0.3 = d \cdot \log 10 = d$$

$$\boxed{\begin{array}{c} d = 0.3\ b \\ \\ b = 3.3\ d \end{array}}$$

Real Numbers: Mantissa and Exponent

The normalized decimal number can be written as follows:

$$0.mmmmmm \cdot 10^{eeee}$$

In this notation *mmmmmm* is called *mantissa*, and *eeee* is called *exponent*. Both mantissa and exponent are integers. Therefore, real numbers can be internally stored as two integers and a sign bit (s=0 for positive and s=1 for negative numbers) using the following format:

s	mmmmmmmmmmmmmmmmmmmm	eeeeeeee
sign	mantissa	exponent

Computers use this notation for real numbers. Of course, both mantissa and exponent are binary coded, and the base is usually 2:

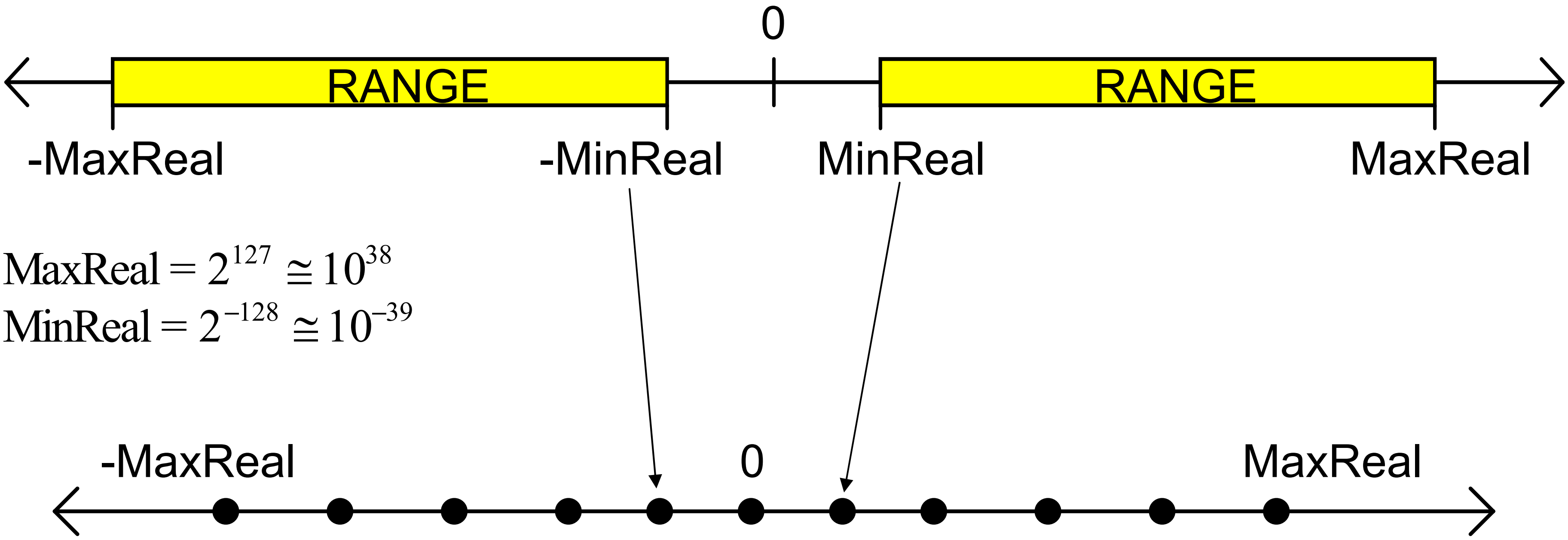
$$0.mmmmmm \cdot 2^{eeee}$$

Range and Precision

1. Using the notation with mantissa and exponent real numbers have standard uniform *format*
2. Exponent determines the *range* of real numbers (minimum and maximum values)
3. Mantissa determines the *precision* of real numbers (number of significant digits)

Range of Real Numbers

The *range* of numbers is determined by the smallest and the largest exponent. If the exponent field has 8 bits (the most frequent case) then -128 ≤ *eeee* ≤ 127. The corresponding range is



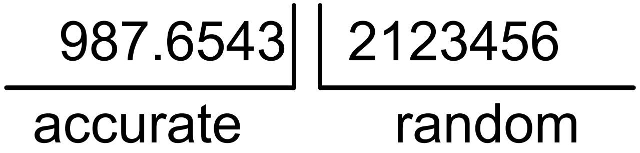
$$\mathrm{MaxReal} = 2^{127} \cong 10^{38}$$
$$\mathrm{MinReal} = 2^{-128} \cong 10^{-39}$$

Precision

Precision is measured as the number of significant decimal digits M, which depends on the size of the binary mantissa *m* :

$$M = m \log_{10} 2 = 0.30103\, m$$

In the case of the 24-bit mantissa the number of significant decimal digits is M=7.2. If the result of a computation is displayed as 987.65432123456 than we know that the first 7 digits can be accurate while the rest is a meaningless sequence of random digits generated during the binary to decimal conversion:



987.6543 | 2123456

accurate random

Imprecision of real numbers

- Real numbers are infrequently precise
- Powers of 2 (and their combinations) are correctly stored in memory. Examples of correct values: 0.5, 0.125
- Rational numbers with an infinite number of digits cannot be correctly represented in memory. For example, 1.0/7.0 stored in memory can only be
  - Less than 1/7, or
  - Greater than 1/7

An example of imprecision: 1./7. > 1/7 (1/7 denotes exact value)

```
$ cat realfor.f
        do x = 1./7. , 1.0, 1./7.
           write(*,*) x
        end do
        end

jozo@SFSU-DELL ~
$ ./a
  0.142857149
  0.285714298
  0.428571463
  0.571428597
  0.714285731
  0.857142866

jozo@SFSU-DELL ~
```

Significant Decimal Digits

```
double precision x, pi
parameter(pi = 3.14159)  ! insufficient precision
do x=0.D0, 1.001D0, 0.1D0   ! step = 0.1
   write(*,'(f5.2,e20.12)') x, dtan(x*pi/2.D0)
end do
end
```

```
0.00  0.000000000000E+00
0.10  0.158384310386E+00
0.20  0.324919415950E+00
0.30  0.509524970491E+00
0.40  0.726541753323E+00
0.50  0.999998732410E+00    (This should be 1)
0.60  0.137637971910E+01
0.70  0.196260620041E+01
0.80  0.307767291770E+01
0.90  0.631370489668E+01
1.00  0.788898123688E+06    (This should be infinity)
```

Commutativity of Addition

Truncation and rounding of real numbers causes that addition of three or more real numbers is not always commutative: a+b+c ≠ c+b+a. Generally, the addition of three or more real numbers should always start with the smallest number and end with the largest. Such result is more accurate than the result that is obtained if additions start with the largest number and end with the smallest number.

```
// Commutativity of addition: 1 + 1/2 + 1/3 + ... + 1/n is
// not the same as 1/n + … + 1/3 + 1/2 + 1

#include <iostream.h>
int main()
{  const int n=1000000;
   float sum1=0.F, sum2=0.F;
   for(int i=1; i<=n ; i++)
   {    sum1 += 1.F/i;
        sum2 += 1.F/(n-i+1.F);
   }
   cout <<"\n1 + 1/2 + 1/3 + ... + 1/" << n <<" = " << sum1;
   cout <<"\n1/" << n << " + ... + 1/3 + 1/2 + 1 = "<< sum2 << "\n\n";
   return 0;
}
```

```
1 + 1/2 + 1/3 + ... + 1/1000000 = 14.3574
1/1000000 + ... + 1/3 + 1/2 + 1 = 14.3927
```

Summary of Numerical Problems

1. The range of all numbers is limited: this holds for unsigned integers, signed integers, and real numbers of different sizes.
2. The sum of unsigned integers can be less than the numbers being added.
3. The sum of positive integers can be negative.
4. The sum of negative integers can be positive.
5. The range and the precision of real numbers are limited.

Summary of Numerical Problems (Cont.)

6. Addition of real numbers is restricted by truncation errors (1+x can yield 1)

7. The sum of real numbers depends on the order of addition (i.e. no commutativity)

8. Power function uses multiplication for integer powers, and logarithms for real powers; this can cause problems when computing powers of negative numbers (a^b = exp(b log a))

Conclusions

- Procedural programming is a fundamental component of all programming languages
- It is useful to see procedural programming from a language-independent angle
- Procedural programming affects performance of software products.
- Performance awareness is a fundamental component of professional software development
- Procedural programming should always be contrasted and complemented with nonprocedural programming.