# Informatica 3

## Presentazione del corso

Laurea in Ingegneria Informatica
Politecnico di Milano
Polo di Milano Leonardo

## Docente

- Prof. Maristella Matera
  Dipartimento di Elettronica e Informazione
  maristella.matera@polimi.it
  tel. 02 23993408

- Ricevimento
  Mercoledì, dalle 10:30 alle 12:00

## Obiettivi

- Concetti fondamentali dei linguaggi di programmazione
- Le tecniche fondamentali di progettazione di algoritmi e strutture di dati
- Strumenti di analisi e di valutazione delle prestazioni

- Distribuzione dell'attività didattica
  - 30 ore di lezione
  - 20 ore di esercitazione (a cura dell'Ing. Andrea Mocci)

## Programma del corso

- Parte I: Linguaggi di programmazione (Ghezzi & Jazayeri)
- Parte II: Algoritmi e strutture dati (Shaffer)

## Parte I: Linguaggi di programmazione

- Concetti generali, classificazioni fondamentali
- Sintassi e semantica
- La struttura dei dati
- La struttura di controllo
- Modularizzazione
- Linguaggi orientati agli oggetti
- Cenni ai linguaggi Interpretati

## Parte II: Algoritmi e strutture dati

- Introduzione alla complessità del calcolo
- La notazione “Theta-grande”
- La realizzazione di tipi astratti di dati in Java
- Strutture dati e algoritmi fondamentali
  - Algoritmi di ricerca e ordinamento
  - Alberi e loro gestione
  - Grafi, loro rappresentazione e gestione
  - Pile, code e tabelle hash
  - Applicazioni di elaborazioni numeriche e simboliche

## Modalita' di Esame

- Un'unica prova nei vari appelli

- Gli studenti sono ammessi agli appelli solo se regolarmente iscritti tramite Poliself nei tempi prescritti

## Materiale didattico

- Per la Parte I
  Ghezzi C., Jazayeri M.: Programming language concepts, 3rd edition, John Wiley & Sons, 2002.
- Per la Parte II
  C.A.Shaffer: A practical Introduction to data structures and algorithm analysis, Java Edition, Prentice Hall, 2002.
- Lucidi delle lezioni tratti dai testi scaricabili dal sito del corso su http://corsi.metid.polimi.it

# Informatica 3

Part I: Programming Languages
Introduction and overview

Laurea in Ingegneria Informatica
Politecnico di Milano
Polo di Milano Leonardo

## Introduction and overview

- Languages and
  the software development process
- Languages and
  software design methods
- Languages and
  computer architecture
- Programming Language Qualities
- Historical perspective

- Ghezzi & Jazayeri. Programming Language Concepts. Chapter 1.

## Programming Languages?

- Tools for writing software
- Different programming languages...
  - ...different approaches to programming
  - ...different pros, cons, etc.
- At least four decades of developments
- Weigh the merits,
  overview the achievements...

## Software development process

- Requirements analysis and specification
- Software design
- Implementation or coding
- Verification and validation
- Maintenance

- Where are programming languages used?
  - Implementation
  - Design phase to describe application decomposition

## Software development environment

- Integrated set of tools and techniques that aids in the development of software
  - Editors, compilers, linkers, libraries, debuggers
- Ideal scenario
  - All the steps of the development process are supported by one integrated environment
- Actual scenario
  - Coding phase is the best supported
  - CASE tools provide an approximation of the ideal scenario

## Software design methods

- Design methods are guidelines for producing a design
- The same set of requirements can yield to different designs
- Examples
  - Top-down design
  - Object-Oriented design
- Programming languages and design principles are strongly related
  - No loops, neither recursion in early FORTRAN...
  - No modules in FORTRAN and Pascal...

## Programming paradigms

- Programming languages may enforce certain programming styles (**programming language paradigms**)
  - Eiffel and Smalltalk are object-oriented, thus they enforce an object-oriented decomposition
  - LISP views programs as functions, thus it enforces a functional decomposition
- Paradigm-oriented languages
  - They enforce a specific paradigm
  - Es.: Eiffel enforces object-oriented design
- Pagadigm-neutral languages
  - They support different paradigms
  - Es.: C++ supports both procedural and object-oriented design

## Programming language paradigms

- Procedural programming
  - Programs decomposed into computation steps
  - Routines used as decomposition units
- Functional programming
  - Computation is viewed as application of functions
  - Functions are primary building blocks
  - Computation of values through expressions and functions
- Abstract data type programming
  - Abstract data types as the unit modularization

## Programming language paradigms

- Module-based programming
  - Modularization, i.e., group of entities (vars, procs, types, etc.) and an export interface
- Object-oriented programming
  - Modularization via class definitions
  - Classes are organized in hierarchies
  - Instances are created during program execution
- Generic programming
  - Definition of generic modules that are instantiated at compile or run time
- Declarative programming
  - declarative problem, not algorithm decomposition

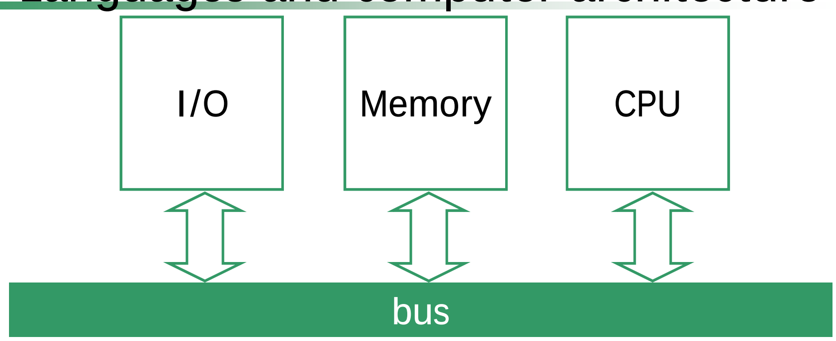
## Programming language paradigms

- Programming Language Paradigms
  - Von Neumann
    - procedural
    - state – based
    - statement-based
  - Non-Von Neumann
    - back to the mathematical notion of variable
    - no assignment
    - Functional
      - LISP
      - ML
    - Logic
      - Prolog

## In this course...

- The goal is ...
  - ...to create flexibility in learning new languages
  - ...to develop critical and comparative attitude
- "Von Neumann languages"
  - Main focus of this course
  - Few "sample languages" (C++, Java, Ada, ...)
- "non Von Neumann languages"
  - Only some hints
  - Other courses focus on this class of languages

## Languages and computer architecture




- Memory contains data and instructions
- Data are taken out of the memory, processed and put back into memory
- Memory roughly represents the state of the machine
- The execution of instructions results into a change of the state

## Languages and computer architecture

- Most current computers are similar to the original von Neumann architecture
- Languages have been constrained by the ideas of von Neumann
- Conventional programming languages are **abstractions** of an underlying von Neumann architecture
- They focus on the relevant aspects of computation, while letting irrelevant details go

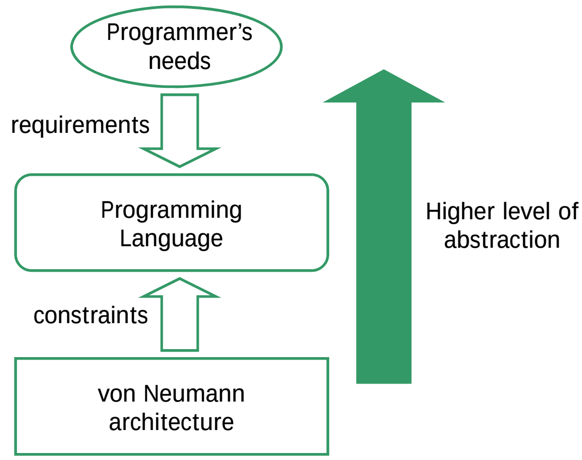
## Abstraction mechanisms

- Variables instead of memory cells
- Step-by-step execution instead of fetch/interpret/execute cycle

Higher level abstractions:

- Data types
- Procedure and functions
- Modularization
- Object-orientation
- Asbtract data types
- Concurrency

## Requirements and contraints




## Programming language qualities

- Programming languages are tools to write software
- The quality of the software is related to the quality of the language
- Software must
  - Be **reliable**
    writability, readability, simplicity, safety, robustness
  - Be **maintainable**
    factoring, locality
  - **Execute efficiently**
    but efficiency of programming is even more important

## Reliability

- Writability
  - Is it possible to express a program in a way that is natural for the problem?
- Readability
  - Is it possible to easily follow the logic of a program so as to discover the presence of errors?
- Simplicity
  - Does it allow to express algorithms easily?
- Safety
  - It should not be easy to write harmful programs!
- Robustness
  - Does it provide mechanisms to deal with undesired events?

## Maintainability

- Factoring
  - Does it allow programmers to factor related features into single units?
  - For example, with classes, modules, etc.
- Locality
  - The effect of a language feature should be restricted to a small local portion of the entire program
  - For example, in ADT programming, the change to a data structure has effect only in the class where it is defined
- Factoring promotes locality

## Efficiency

- Previously, the focus was on the achievement of speed and space
- Currently, the focus is on the productivity of the software development process
  - Reusability
  - Portability

## Historical perspective

- Early software process: simple programming
  - Late 50s - early 60s
  - FORTRAN: separate compilation, formulas
  - ALGOL60:block structure, recursion, data structure
  - COBOL: Files, I/O
- Early nonconventional languages
  - LISP: lists, simple operations, uniform code/data (in the form of list)
  - APL: arrays, lots of operations
  - SNOBOL4: strings, pattern-matching
  - LISP and SNOBOL4: symbolic computation (instead of numerical computation)

## Historical perspective

- Late 60s
  - Algol 68: orthogonality, formal specification
  - Simula 67: simulation, parallel execution, class
  - Pascal: Simplicity
  - BASIC: interactivity, interpretive programming
- 70s
  - Experimentation with paradigms
  - Object orientation: Smalltalk, Eiffel, ...
  - Logic: PROLOG
  - Concurrency and modularity: Mesa, Modula-2
  - Security: Euclid, Gypsy

## Historical perspective

- 80s
  - Object-orientation: C++, Ada, Eiffel
- 90s
  - Visual interfaces: Visual basic
  - Parallelism: FORTRAN 90
  - Scripting specifying activation patterns for existing tool fragments: Tcl/Tk, Perl, Python
  - Network programming: Java

# Informatica 3

Part I: Programming Languages
Syntax and semantics (A)

Laurea in Ingegneria Informatica
Politecnico di Milano
Polo di Milano Leonardo

## Syntax and semantics

- Language definition, syntax and semantics
- Language processing
- Routines
- Aliasing and overloading

- Ghezzi & Jazayeri. Programming Language Concepts. Chapter 2.

## Programming languages

- Tools for writing software
- Formal notations for describing algorithms for execution by computers
- Formal notations need
  - Syntax
    set of rules specifying how programs are written, by means of sentences obtained through combinations of *words* (keywords, identifiers, operators, numbers, etc.)
  - Semantics
    set of rules specifying "the meaning" of syntactically correct programs

## Language definition

- Syntax
  - Formal, based on some notation, e.g., extended Backus-Naur form (EBNF)
  - Informal, based on some verbal description, as done in the original FORTRAN description
- Semantics
  - Formal, based on the mapping onto mathematical domains
  - Operational, based on a high-level description of the expected program behavior

## Syntax

- Lexical rules
  - Define the “alphabet” of the language
  - E.g, set of chars, combinations
  - Example
    - Pascal does not distinguish between uppercase and lowercase chars, while C and Ada do
    - <> is a valid operator in Pascal, but denoted by != in C, and /= in Ada
- Syntactic rules
  - Define how to compose words to build syntactically correct programs
  - Helps programmer to write syntactically correct program
  - Allows to check whether a program is syntactically correct

## Extended Backus Naur Form (EBNF)

- Early descriptions were done using natural language
- ALGOL60 was described by John Backus using a context free grammar (BNF)

- EBNF is a metalanguage
- I.e., a language to describe other languages
- **Rules, entities, metasymbols**

## Syntax rules

```
<program>   ::= { <statement>* }          rule
<statement> ::= <assignment>   |
                <conditional> |
                <loop>
non-terminal                      metasymbols

<assignment>::= <identifier> = <expr> ;
<conditional>::=
  if <expr> { <statement>+ } |
  if <expr> {<statement>+} else {<statement>+}
terminals
<loop> ::= while <expr> {<statement>+}
<expr> ::= <identifier> | <number> |
           ( <expr> ) | <expr> <operator> <expr>
```

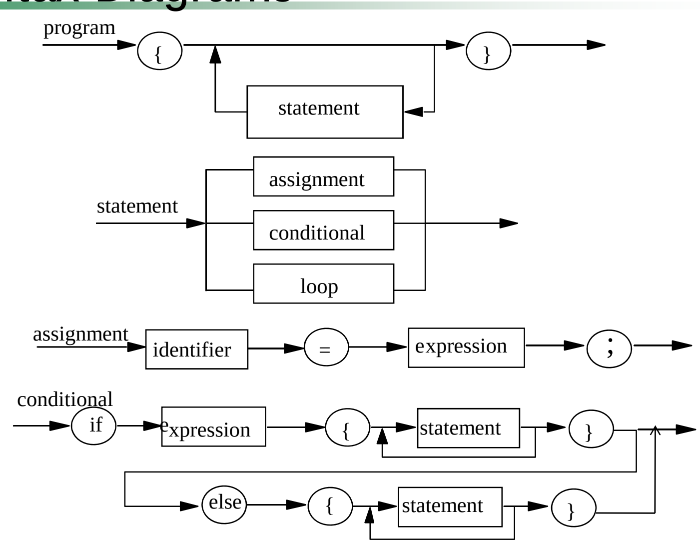
## Metasymbols

- **::=**
  "is defined as"
- **<…>**
  entities, nonterminals
- "**{**" "**;**" "**}**" "**if**" "**else**"
  terminals
- *
  "zero or more occurrences of the preceding element"
- +
  "one or more occurrences of the preceding statement"
- |
  "one of these choices"

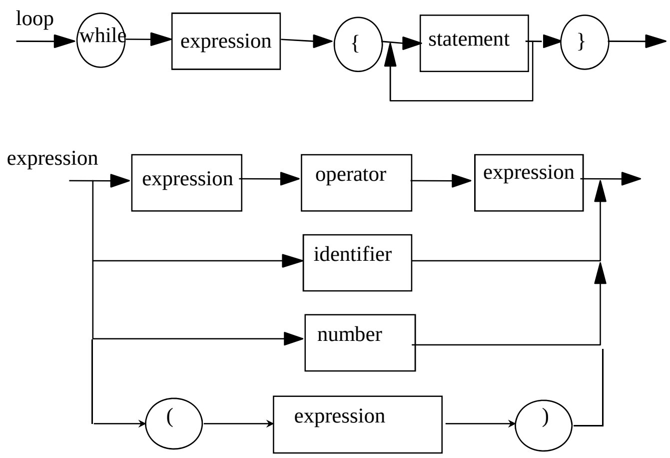
## Lexical rules

```
<operator> ::= + | - | * | / | = | /= |
               < | > | <= | >=

<identifier> ::= <letter> <ld>*

<ld> ::= <letter> | <digit>

<number> ::= <digit>+

<letter> ::= a | b | c | . . . | z

<digit> ::= 0 | 1 | . . . | 9
```

## Syntax Diagrams



## Syntax Diagrams




## Abstract syntax, concrete syntax

- C Language
```
while (x != y) {
. . .
};
```
- Pascal
```
while x <> y do
begin
...
end
```
- Same structure, they are expressed by the same syntactic rules
- Different at the lexical level
  - “{” and “}” instead of “begin” and “end”
  - “!=” instead of “<>”
  - In Pascal, “(” and “)” can be omitted
- They have the same abstract syntax but different concrete syntax
- Pragmatically they differ, “<>” is more readable than “!=”

## Pragmatics...

- Some languages allow brackets to be omitted in single statements

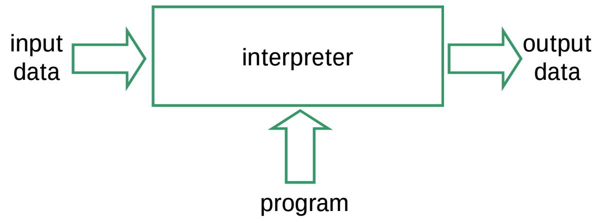
```
while (x != y) x = x + 1;
```

- Pragmatically, this can be error prone
- Modula-2 address this issue at a "concrete-syntax" level using the "end" keyword to end all the conditional/loop statements

## Language processing

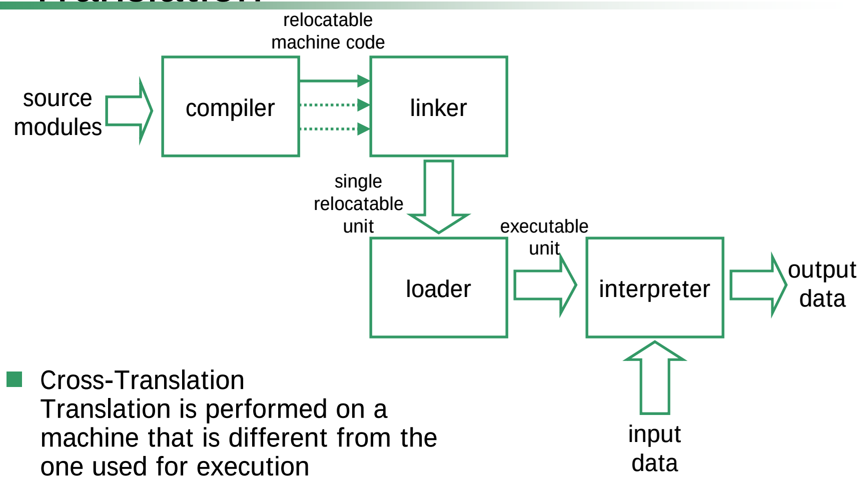
- Interpretation
  - Language statements are directly executed
- Translation
  - High-level language statements are translated into machine-level statements before being executed

## Interpretation



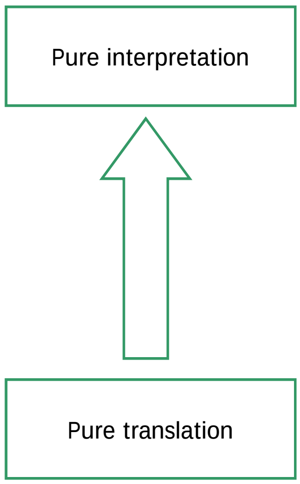

- main cycle
  1. Get the next statement
  2. Determine the actions to be executed
  3. Perform the actions
- Interpretation is a simulation on a host of a special purpose computer with a high-level machine language

## Translation




- Cross-Translation
  Translation is performed on a machine that is different from the one used for execution
  - Needed when the execution machine has a special purpose processor

## Interpretation vs. Translation




- Translation to intermediate code
- Interpretation of the intermediate code on a virtual machine
- Portability of executable code
- Examples
  - Java bytecode

## The concept of binding

- Programs deal with different types of entities
  - Variables
  - Routines
  - Statements
- Entities have different properties or attributes
- Variables
  - Name, type, storage area...
- Routines
  - Name, formal parameters, parameter-passing information...
- Statements
  - Actions to be taken
- Attribute information is stored in the descriptor
- Binding sets attributes for entities

## The concept of binding

- Different programming languages…
  - …have different number of entities
  - …have different number of attributes bound to entities
  - …allow different binding time
  - …have different binding stability
- Binding time
  - Determine when the binding occurs
- Stability
  - Determine whether certain bindings are fixed or modifiable
  - Static binding cannot be modified

## Binding time

- Language definition time
  - E.g., in most languages (FORTRAN, Ada, C++) the integer type is bound at language definition
- Language implementation time
  - E.g., then during implementation, the integer type is bound to a memory representation
- Compile-time (translation-time)
  - Pascal provides a way to redefine the integer type, the new representation is bound to the type when the program is compiled
- Execution-time (run-time)
  - Variables are usually bounded to a value during execution

## Static binding, dynamic binding

- Static Binding
  - Established before execution and cannot be changed thereafter
  - E.g., FORTRAN, Ada, C, C++ integer bound to a set of values at definition/implementation time
- Dynamic Binding
  - Established at run-time and usually modifiable during execution
  - E.g., variable values
- Exceptions
  - in Pascal, read-only constant are variables whose value is initialized at run-time but cannot be modified thereafter

## Variables

- Conventional language variables are memory abstractions
  - In main memory, cells are identified by an address
  - The contents of a cell are encoded representation of a value
- Variables
  - The variable name abstracts the address information
  - The assignment statement abstracts the cell modification

## Formally...

- A variable is a 5-tuple:

```
<name, scope, type, l_value, r_value>
```

- Name
  - String used in program statement to denote the variable
- Scope
  - Range of program instructions over which the name is defined
- Type
- L-value
  - Memory location associated to the variable
- R-value
  - Encoded value stored in the variable's location

## Name and scope

- The variable's name is usually introduced by a declaration statement
- The variable's scope usually extends from the declaration until a later closing point
  - The scope is the range of program statements over which the name is known
- A variable is visible inside its scope, invisible outside it

## Scope binding

- Static scope binding
  - The variable's scope is defined in terms of the syntactic structure of the program
  - The scope of a variable can be determined just examining the program text
  - Most of the programming languages adopt static scope binding
- Dynamic scope binding
  - The variable's scope is defined in terms of program execution
  - Each declaration extends its effect over all the instructions executed thereafter, until a new declaration for the same variable is encountered
  - APL, SNOBOL4, and LISP use dynamic scope binding

## Example: static scope binding

```
main()
{  int x,y;
   …
   {
       int temp;
       …
       temp = x-y;
   }

}
```

```
main()
{  int x,y;
   …
   {
       int temp;
       int x;
       …
       temp = x-y;
   }

}
```

## Example: dynamic scope binding

```
{
  /* block A */
  int x;
}
{
  /* block B */
  float x;
}
{
  /* block C */
  x = ...;
}
```

- Case 1: A then C
  - Variable x in block C refers to x declared in block A
- Case 2: B then C
  - Variable x in block C refers to x declared in block B

## Dynamic scope binding

- Rules for dynamic scope binding are simple and rather easy to implement
- The major disadvantages are in terms of
  - programming discipline
  - efficiency of implementation
- Programs are hard to read and difficult to debug

## Type

- Set of values, and ...
- Set of operations that can be legally performed on such values
- It protects variables from nonsensical operations
- A variable of a given type is also called an instance of the type
- A language can be
  - Typeless or untyped, e.g., assembly languages
  - Dynamically typed, e.g., LISP
  - Statically typed

## Typed Languages

- Built-in types
- Declarations of new types
  - Binding between the type name and implementation at translation time
- Abstract data types

## Definition of new types

```
typedef int vector[10];

main()
{
   vector a;
   a[0] = 3;
}
```

- **`vector`** is a new type
- The association between the type name and the type implementation is at translation time
- **`vector`** inherits all the operations of the represented data structure

## Definition of abstract data types (ADT)

- Associate new types with a set of legal operations
- Typical ADT declaration is structured as:

```
typedef new_type_name
{
  data structure for new_type_name;
  operations to manipulate data objects;
}
```

## Definition of Abstract Data Types

```
class StackOfChar {
private:
  int   size;
  char* top;
  char* s;
public:
  StackOfChar (int sz)
  {
    top = s = new char [size =sz];
  }
  ~StackOfChar ( ) {delete [ ] s;}
  void push (char c) {*top++ = c;}
  char pop ( ) {return *--top;}
  int length ( ) {return top - s;}
};
```

Data structure

Operations

## Type checking

- Statically typed language
  - Variables are bound to a type before runtime
- Dynamically typed language
  - The binding between a variable and its type is defined at run-time
  - Variables are polymorphic
- Type checking verifies correct use of variables
- Type checking can be performed statically
  - For statically typed languages
  - For certain categories of dynamically typed languages (we will see this case for OO languages)

## Question...

- Some languages allow implicit declaration, i.e., the first occurrence of the new variable name is also taken as an implicit declaration
- What kind of binding is this, static or dynamic?

## l-value

- The l-value of a variable is the storage area bound to the variable during execution
- The lifetime or extent of a variable is the period of time in which such a binding exists
- The l-value is used to hold the r-value
- Data object is the pair <l-value,r-value>
- The binding between the variable and the l-value is the memory allocation
- Memory allocation acquires storage and binds the variable
- Lifetime extends from allocation to deallocation

## Memory allocation

- Static allocation
  - Allocation is performed before run-time, deallocation is performed upon termination
- Dynamic allocation
  - Allocation and deallocation are at runt-time performed on explicit request (through creation statements) or automatically

## r-value

- The r-value of a variable is the encoded value stored in the location (l-value) of the variable
- The r-value is interpreted according to variable's type
- Instructions
  - Access variables through their l-value (lefthand side of assignments)
  - Modify their r-value (righthand side of assignments)
- The binding between variables and values is usually dynamic
- Generally, it is static only for constants

## Example

```
#define MAX = 10;

const int a = MAX;
int b = MAX;
int c;
```

- Both C and Pascal define constants as variables whose r-value cannot be changed once assigned
- In C and Ada such binding is established at run-time
- In Pascal it is established at compile-time

## Initialization

- What is the r-value bound to the variable as it is created?
- Different languages, different solutions
  - ML requires that the binding is established at creation
  - Other languages like C and Ada support such a binding but do not require it (`int I, J=0;`)
- What if no initialization is provided?
- Ignore
  - the bit string found in the storage area is considered the variable initial value
- System-defined initialization
  - int = zero, char = blank
- Special undefined value
  - The variable is considered initialized to an undefined value, and any read access to an undefined variable is trapped

## References and unnamed variables

- Some languages allow variables that can be accessed through the r-value of another variable
- Reference or pointers are variables whose r-value allow to access the content
  - of another variable
  - of an unnamed variable
- Access via a pointer is called dereferencing

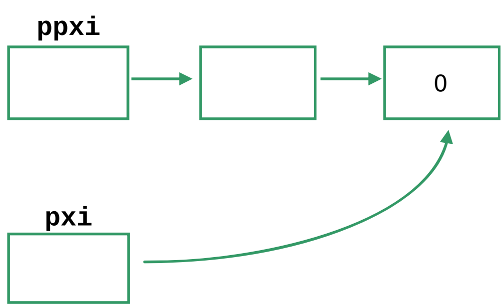
```
int  x = 5;
int* px;
px = &x;
```

The r-value of px allows to access x

```
int* px;
px = malloc(sizeof(int));
*px = 0;
```

```
p: ^integer;
new (p);
p^ = 0;
```

## Example: Pascal




```
type PI = ^integer;
type PPI = ^PI;
var pxi: PI;
var ppxi: PPI;
...
new(pxi);
pxi^ := 0;
...
new(ppxi);
ppxi^ := pxi;
```

## Routines

- Routines allow programs to be decomposed in a number of functional units
- Routines are general concepts found in most languages
  - Subprograms in assembly language
  - Subroutines in FORTRAN
  - Procedures and functions in Pascal and Ada
  - Functions in C
- Functions are routines that return a value
- Procedures are routines that do not return a value

## Routines

- Like variables, also routines have name, scope, type, l-value, and r-value
- Name
  - Introduced by a routine declaration
- Scope
  - The scope of the routine name extends from the declaration point to some closing point, either statically or dynamically determined

## Routines

- Activation
  - Routines activation is achieved through a routine invocation or routine call
  - The call statement must be in routine's scope
- Local scope
  - Routines also define a scope for the declarations that are nested in them
  - Such local declarations are only visible within the routine
  - Depending on the language, routines can also refer to nonlocal (i.e., global) items
- Header
  - Defines the "routine type", or signature, i.e., the types of the parameters and the return type
  - For instance, the signature for the function sum is

$$sum:int \rightarrow int$$

## Example

```
/* sum of the first n positive integers */

int sum (int n)
{
   int i, s;
   s = 0;
   for (i = 1; i <= n ; ++i)
       s+= i;
   return s;
}
```

header

body

## Routine calls

- Type correct
  - if the call conforms to the routine type

```
int sum(int n)
{ ... }
...
i = sum(10);
```

- Type incorrect
  - if the call does not conform to the routine type

```
int sum(int n)
{ ... }
...
i = sum(5.3);
```

## Routines

- l-value
  - The reference to the area where the routine body is stored
- r-value
  - The body that is currently bound to the routine
  - Usually this binding is statically determined at translation time
- Some languages allow variables of type “routine”, to which a routine value can be assigned

## Example

```
/* ps is a pointer to a function that
   takes an int and returns an int */
int   (*ps)(int);
int   i;

ps = &sum;         /* ps points to sum */

/* call the routine pointed by ps */
int i = (*ps)(5);
```

## Declaration and definition

- Some languages distinguish between routine declaration and definition
- Declaration
  - introduces the routine's header without specifying the body;
  - specifies scope
- Definition
  - specifies both the header and the body
- The distinction between declaration and definition support mutual recursion

## Example

```
/* declaration of A, A is visible from now on */
int A (int x, int y);

/* definition of B, B is visible from now on */
float B (int z)
{
   int w, u;
   w = A (z, u);     /* A is visible here */
   ...
};

int A(int x, int y)
{  ...
   float t = B(x);   /* B is visible here */
}
```

## Routine instance

- The representation of a routine during execution is called routine instance
- Code segment
  - Contains the instructions of the unit, the content is fixed
- Activation record (or frame)
  - Includes all the data objects associated with the local variables of a specific routine instance
  - Contains all the information needed to execute the routine
  - It is changeable
- Referencing Environment of the instance
  Given the instance U,
  the referencing environment for U, contains
  - U's local variable, i.e., its **local environment**
  - U's non local variables, i.e., its **non local environment**
  - Modifying a data object bound to a nonlocal variable is called **side-effect**

## Recursive activation

- All instances of the same unit composed of
  - same code segment
  - different activation records
- The binding between an activation record and its code segment is dynamic

## Parameters

- Formal parameters
  appear in the routine's definition
- Actual parameters
  appear in the routine's call

## Parameters binding

- Positional method

```
routine S(F1,F2...Fn);

call    S(A1,A2...An);
```

- Named Association

```
procedure Example (A:T1; B:T2:=W; C:T3);
Example(X,Y,Z); --purely positional
Example(X,C=>Z); --X to A, positionally
                 --Z to C, named ass.
```

## Example

- In some cases, the number of actual and formal parameters need not be the same.

```
int distance(int a=0, int b=0);

distance();    // eq. to distance(0,0)
distance(3);   // eq. to distance(3,0)
```

## Parameter passing

- Parameter passing supports inter-unit information flow
- In most cases, data entities may be passed
- When routine are first class objects, also routine may be passed
- Note
  - Similar effects could be achieved through global variables
  - But the use of parameters provides advantages in terms of readability and modifiability

## Generic Routines

```
int i,j;
float x,y;

swap(i,j); // swap the values of i & j

swap(x,y); // swap the values of x & y
```

## Generic swap routine

```
// the function is generic with respect to type T;
// a and b refer to the same locations as the
// actual parameters;

template <class T>
void swap (T& a , T& b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

the generic function is parameterized for class T

function definition is based on T

## Generic routines

- They are templates from which specific routines are generated through instantiation
- Instantiation binds generic parameters to actual parameters at compile-time
- Instantiation can be obtained through macroprocessing which generates a new instance for each parameter type
- Other implementation schemes are also possible

## Overloading

- It occurs when
  - ...more than one entity is bound to a name at a given point
  - ... and name occurrence provides enough information to disambiguate the binding
- Example

```
int i,j,k;
float a,b,c;
i = j + k;       // int addition
a = b + c;       // float addition
a = b + c + b();// b is overloaded
```

## Aliasing

- It is the opposite of overloading
- Two names, N1 and N2, are aliases if they denote the same entity at the same program point
- N1 and N2 share the same object in the same referencing environment
- Main issue: modifications under N1 have an effect visible under N2
- Aliasing may lead to error prone and difficult to read programs

```
int x =0;
int *i = &x;
int *j = &x;
```

## Example

```
int i;
int fun (int& a)
{ ...
  a = a + 1;
  print(i);
  ...
}
main()
{
  x = fun(i);
}
```