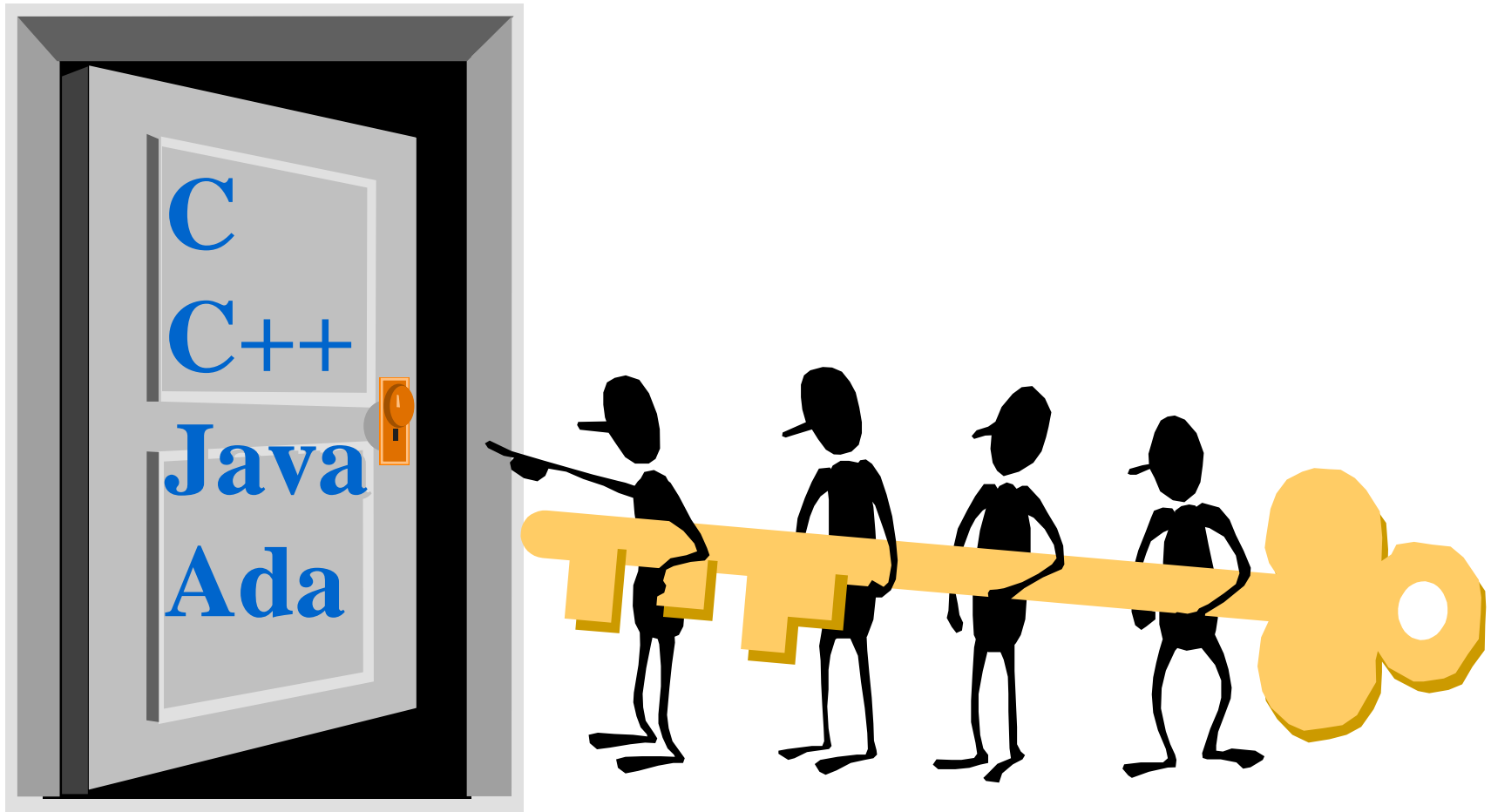


Principles and Fundamentals



Principles - Introduction

- ◆ This section introduces a set of **very important principles** and **concepts**.
- ◆ They are **necessary pre-requisites** for a proper understanding and use of C, C++ and Java.
- ◆ They provide a framework, a structure, that can be used to evaluate, learn and use not only C, C++ and Java but also other programming languages (e.g. Ada, Fortran, Smalltalk, etc...).
- ◆ This is the most important section of the course.

Principles - Syntax vrs. Semantics

- ◆ A **programming language** is a **formal notation** describing the **algorithms** (I.e. programs) executed by a “computer”.
- ◆ Like all other formal notations, a programming language consists of two major components:
 - the **syntax**;
 - the **semantics**.
- ◆ The syntax is a set of rules specifying the “**grammar**” of a language (I.e. how it is possible to describe programs by aggregating identifiers, keywords, constructs, etc...)

Principles - Syntax vrs. Semantics (cont)

- ◆ The semantics is a set of rules specifying the “**meaning**” of a program (I.e. what the “computer” has to do).
- ◆ Examples:
 - **inject** 1 milliliter **of** morphine **to** patient_A
(syntactically and semantically correct)
 - **iject** 1 milliler **of to** patient_A morphine
(syntactically wrong)
 - **inject** 1 liter **of** morphine **to** patient_A
(semantically wrong)

Principles - Syntax vrs. Semantics (cont)

- ◆ The **syntax is not very important** because:
 - it can be easily learned from the available literature;
 - it can always be checked by the compiler.
- ◆ The **semantics**, I.e. the meaning, on the contrary **is very important** because:
 - it is difficult to express and learn;
 - no compiler can check it.
- ◆ This **course focuses on the semantics** of C, C++ and Java languages.

Principles - Syntax vrs. Semantics (cont)

- ◆ The semantics of a programming language can be expressed in different ways, e.g.:
 - formal way \Rightarrow by mapping each single construct of the language into a domain whose semantics is known;
 - operational way \Rightarrow by specifying the **behaviour** of an **abstract executor** (a **virtual machine**) which executes syntactically correct programs written with the language.
- ◆ In this course the **operational approach** will be used.

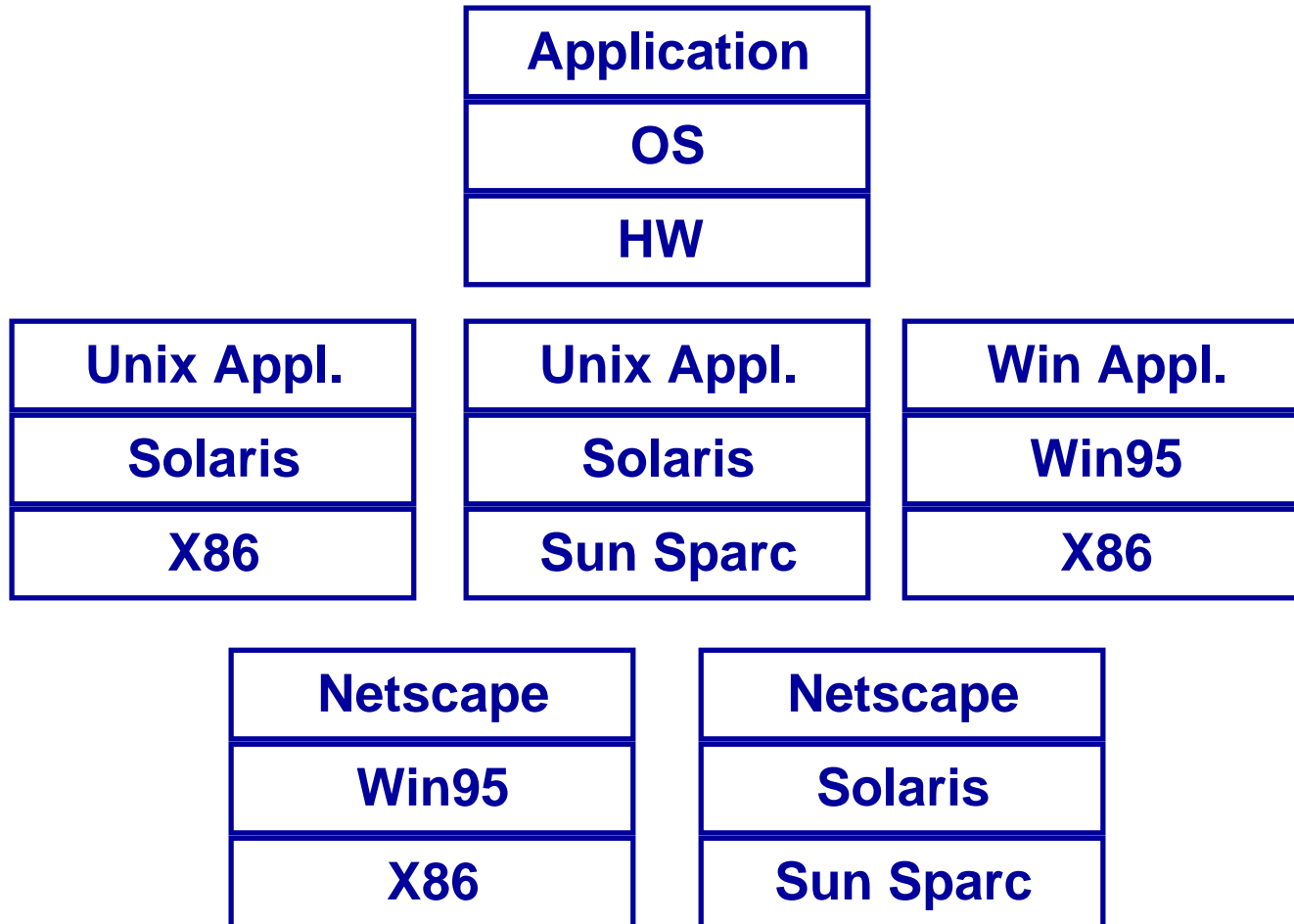
Principles - Virtual Machines



- ◆ A Virtual Machine is an **abstraction layer** imposed on a physical machine (or on another virtual machine) which **isolates, hides** the operations and behaviour of the machine(s) below it.

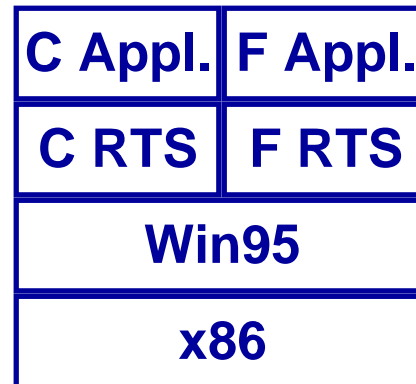
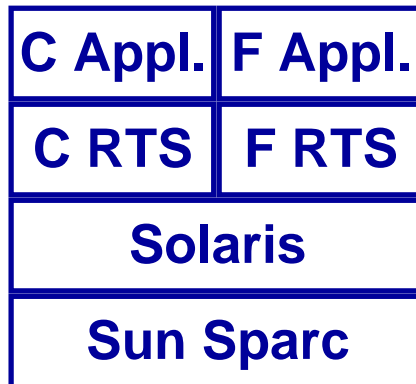
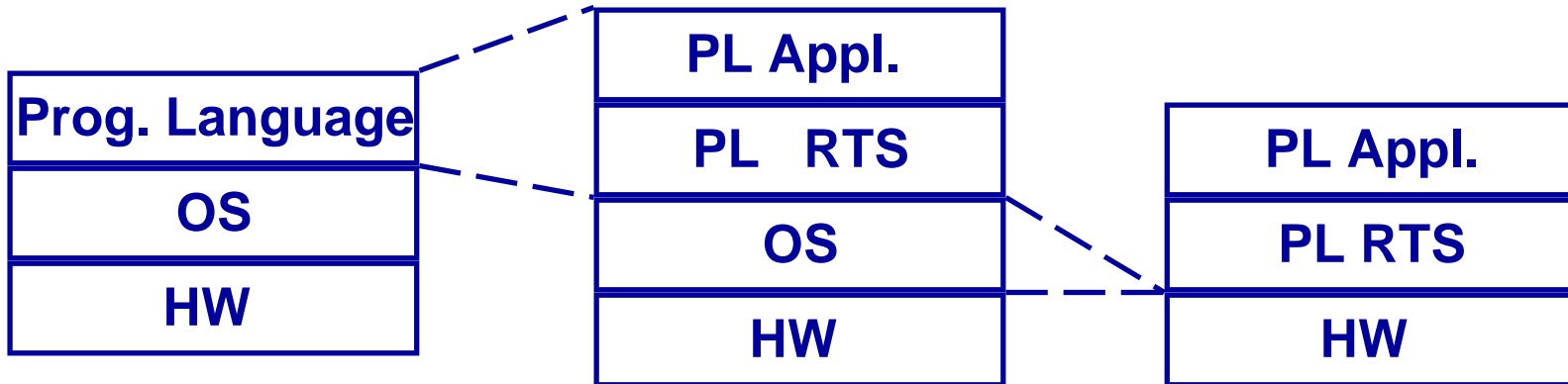
Principles - Virtual Machines

(cont)



Principles - Virtual Machines (cont)

(cont)



RTS =
Run Time
Support

Principles - Statements

- ◆ A program can be considered as a list of instructions, I.e. statements. There are different types of statements:
 - **Declaration Statements** - they generally produce no executable code; their effect is to inform the compiler about attributes (data structure, size, etc...) of names encountered in the program.
 - **Computation Statements** - these are statements that apply operators to operand to compute new values.
 - **Sequence-control Statements** - normally control flows automatically from a statement to the next one; these statements explicitly alter the flow of control.
 - **Structural Statements** - (structural constructs) they are used to group statements into structures.

Principles - Program Units

◆ Statements can be grouped in **blocks**.

– Ex.

```
begin                                {  
    statement 1;                       statement 1;  
    statement 2;                       statement 2;  
    ...  
    statement n                       statement n;  
end                                  }
```

Note: not all programming languages support blocks.

- ◆ Portions of a program can be decomposed into **subprograms**, I.e. **procedures** and **functions**.
- ◆ A **procedure** specifies a **sequence of actions** and it is invoked by a **procedure call statement**.
- ◆ A **function** specifies a sequence of actions and also **returns a value** called the result, and so a function call is an **expression**.

Note: the distinction between procedures and functions (valid in F77 and in Ada, in C/C++/Java sometimes is just a theoretical one).

- ◆ A **compilation unit** is a portion of a program which can be compiled in isolation. A compilation unit can contain more subprograms.
- ◆ A sound and sensible decomposition of a program into compilation units can improve the development and execution times.

Principles - l-value & r-value

- ◆ The assignment statement

`a = b;` (BTW: This is a C/C++/Java assignment statement!)

- ◆ actually means “put the **value** of **b** in the **location** denoted by **a**”. In other words on the **right** of the assignment operator we are interested in the **value**, on the **left** we are interested in the **location**.
- ◆ We refer to the value associated with a name as its **r-value** (right-value); and we refer to the location associated with a name as **l-value** (left-value).

Principles - Parameter Transmission

- ◆ Functions and procedures can receive information from the caller either via global variables or **parameters**.
- ◆ A distinction needs to be made between
 - **formal parameters** - the parameters used in the declaration of the function/procedure and
 - **actual parameters** - the parameters actually passed to the function/procedure at invocation time.
- ◆ There are three common methods of passing parameters:
 - **call-by-reference;**
 - **call-by-value;**
 - **call-by-name.**

Principles - Parameter Transmission (cont)

- ◆ **Call-by-reference:** the calling program passes to the called subprogram a pointer to the r-value (I.e. the l-value) of each actual parameter. The actual parameters can be modified by the called function/procedure.
- ◆ **Call-by-value:** the actual parameters are evaluated and their r-values are passed to the subprogram in well known locations (I.e. they are copied into these locations). The actual parameters cannot be modified by the called function/procedure.

Principles - Parameter Transmission (cont)

- ◆ Call-by-name: (macro) the subprogram itself is substituted for the call, with actual parameters literally substituted for formal parameters.
- ◆ In the case of functions, the return values can be passed back to the caller via three different types of mechanism:
 - **return-by-reference;**
 - **return-by-value;**
 - **return-by-name.**(BTW: Who's in control of the parameter transmission? The caller or the called subprogram?)

Principles - Parameter Transmission (cont)

Swap Example in F77 (swap.f)

```
C   Program Swap
      INTEGER A, B

C   Exec
      A = 1
      B = 2
      WRITE (6, 100) A, B
      CALL SWAP (A, B)
      WRITE (6, 110) A, B
      STOP

C   Formats
100 FORMAT ('Before swapping: A =
           ',I,', B = ',I,'.')
110 FORMAT ('After  swapping: A =
           ',I,', B = ',I,'.')
      END
```

Swap Example in C (swap.c)

```
/* Program Swap */
#include <stdio.h>

void swap(int i, int j);

void main(void) {
    int a, b;

    a = 1;
    b = 2;
    printf("Before swapping: A = %d, B =
           %d.\n", a, b);
    swap(a, b);
    printf("After  swapping: A = %d, B =
           %d.\n", a, b);
}
```

Principles - Parameter Transmission (cont)

```
C      Do the Swap
      SUBROUTINE SWAP (I, J)
      INTEGER I, J, TEMP

      TEMP = I
      I = J
      J = TEMP
      RETURN
      END
```

```
>Before swapping: A = 1, B = 2.
>After  swapping: A = 2, B = 1.
```

```
/* Do the Swap */
void swap(int i, int j) {
    int temp;

    temp = i;
    i = j;
    j = temp;
}
```

```
>Before swapping: A = 1, B = 2.
>After  swapping: A = 1, B = 2.
```

Principles - Binding Environments

- ◆ Given a statement like the following one:

$y = x + 1;$

- ◆ How is the value of **x** determined? **x** may occur in different places in the program. It may refer to a local or a global variable. It may be a formal or an actual parameter...

Principles - Binding Environments

(cont)

Binding Environments Example (benv.c)

```
#include <stdio.h>

/* Functions Prototypes */
int inc1(int x);
int inc2(void);

int x = 1; /* Static declaration */

int main(void) {
    int x = 2; /* Automatic declaration
               */

    /* Call to the first function */
    x = inc1(x);
    printf("X = %d.\n", x);

    /* Call to the second one */
    x = inc2();
    printf("X = %d.\n", x);

    return 0;
}

/* Increment Functions */
int inc1(int x) {
    return (x + 1);
}
int inc2(void) {
    return (x + 1);
}

>X = 3.
>X = 2.
```

Principles - Binding Environments (cont)

- ◆ To properly translate a program into machine code each mention of an **identifier** has to be **associated with** the actual **location** it represents.
- ◆ The association is a two-stage process:
 - **binding the identifier to its name**: where the name is unique in the program context (e.g. `x => _static_area_x` etc.);
 - **binding of the name to its location** (e.g. `_static_area_x => 0x1234abcd`)

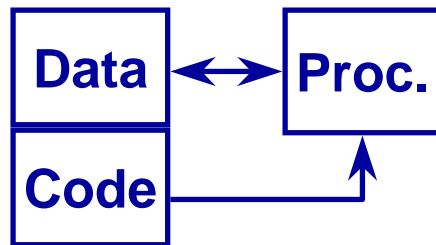
Principles - Binding Environments (cont)

- ◆ In many languages **the binding** of identifiers **depends** only **on their position** in the program (**static binding**).
- ◆ In other languages **the binding** of identifiers **can only be known at run time** (**dynamic binding**).
- ◆ The association between an identifier and its name is the **binding environment** of a statement, block, subprogram.
- ◆ The **scope** of a name is the portion of a program over which it may be used.

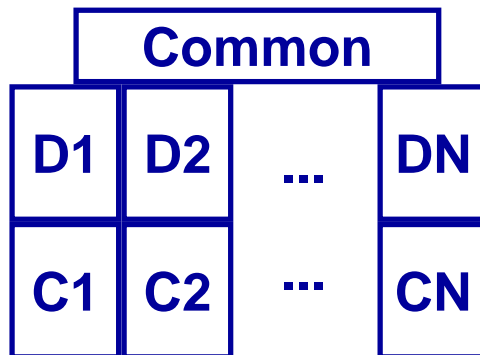
Principles - Storage Management

- ◆ In every program there are a number of elements to which storage must be allocated.
- ◆ Most obvious is the storage required for the code, the data structures, variables and constants...
- ◆ Less obvious is the the need for space for procedure-linkage information, the temporaries required for expression evaluation and parameter transmission, the space allocated for I/O buffers...
- ◆ Programming languages can be classified based on the way they manage the storage.

Principles - Static Languages VMs



Generic VM



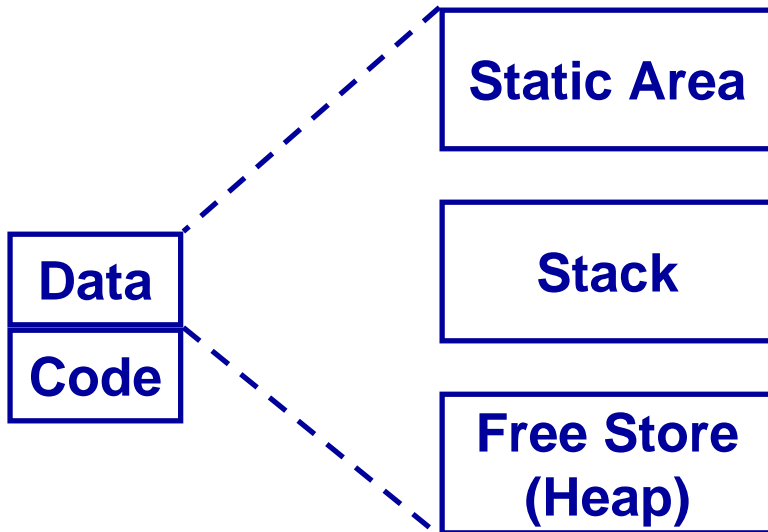
F77's VM

- ◆ If
 - (1) **the size of every data item can be determined by the compiler** (e.g. no variable length arrays...)
 - (2) **recursive procedure calls are not permitted**
- ◆ then the **space** for the code and data **can be allocated** at compile time, I.e. **statically**.

Principles - Static Languages VMs (cont)

- ◆ Each subprogram can be compiled separately.
- ◆ Each subprogram can store its return address in a private location.
- ◆ The total space needed by a program is the sum of
 - the space needed by the subprograms and their data;
 - the space needed for linkage information (I.e. return address).
- ◆ The **total space needed is known at compile time** and never changes.
- ◆ **Static allocation is easy to implement and requires no run-time support.**

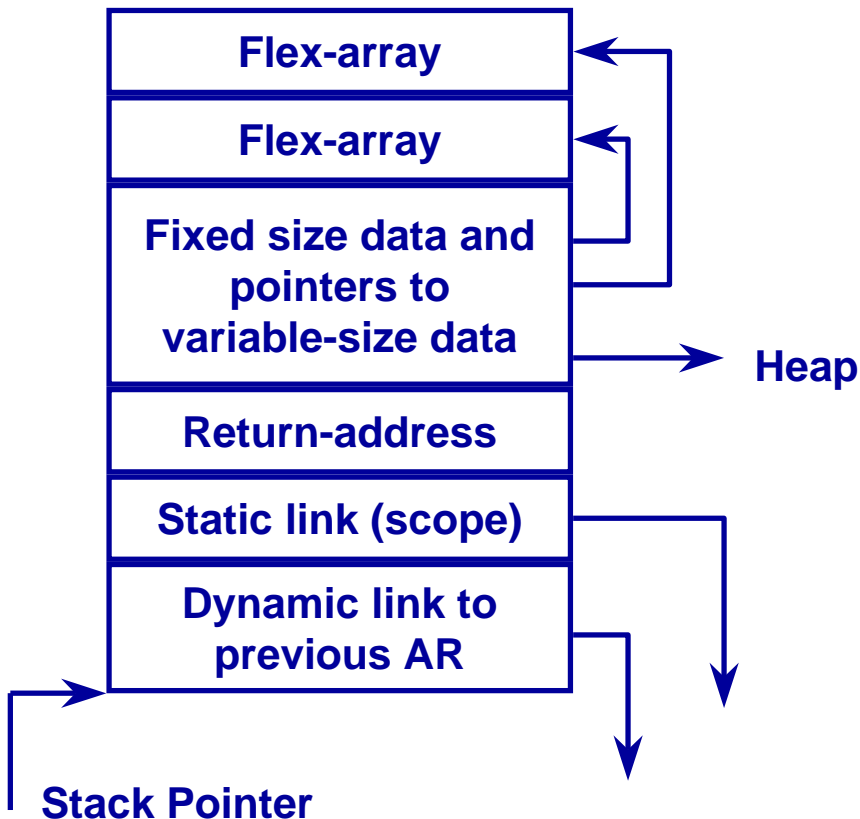
Principles - Dynamic Languages VMs



- ◆ In Dynamic Languages VMs data is divided into three areas:
 - the **Static Area** - for data items which are active along all the program's execution;
 - the **Stack** - for data items declared inside a subprogram and only active when the subprogram is executing;
 - the **Heap** - for data items explicitly allocated and deallocated during the execution of the program.

Principles - Dynamic Languages VMs (cont)

Activation Record



- ◆ The **activation record** of a subprogram in the stack contains:
 - **storage** for simple names and pointer to other data structure local to the procedure, pointers to the heap;
 - **temporaries** for expression evaluation and parameter passing;
 - the **return address**;
 - the **static link**, I.e. the pointer to the first AR in the chain of the ARs in scope;
 - the **dynamic link**, I.e. the pointer to the AR of the caller.
- ◆ Static links may be stored in another data structure, called **display**.

Principles - Dynamic Languages VMs (cont)

- ◆ The **stack permits** some **flexibility** in **managing** of **data structures**.
- ◆ The **stack permits** a programming language to have **recursive subprograms**.
- ◆ **Recursion** is very **expressive** and **elegant**.
- ◆ **Flexibility and recursion have a price** in terms of performances and memory occupation. **They require a Run-Time Support (RTS) system**.
- ◆ The **stack permits** an easy the **implementation of re-entrant code**.

Principles - Dynamic Languages VMs (cont)

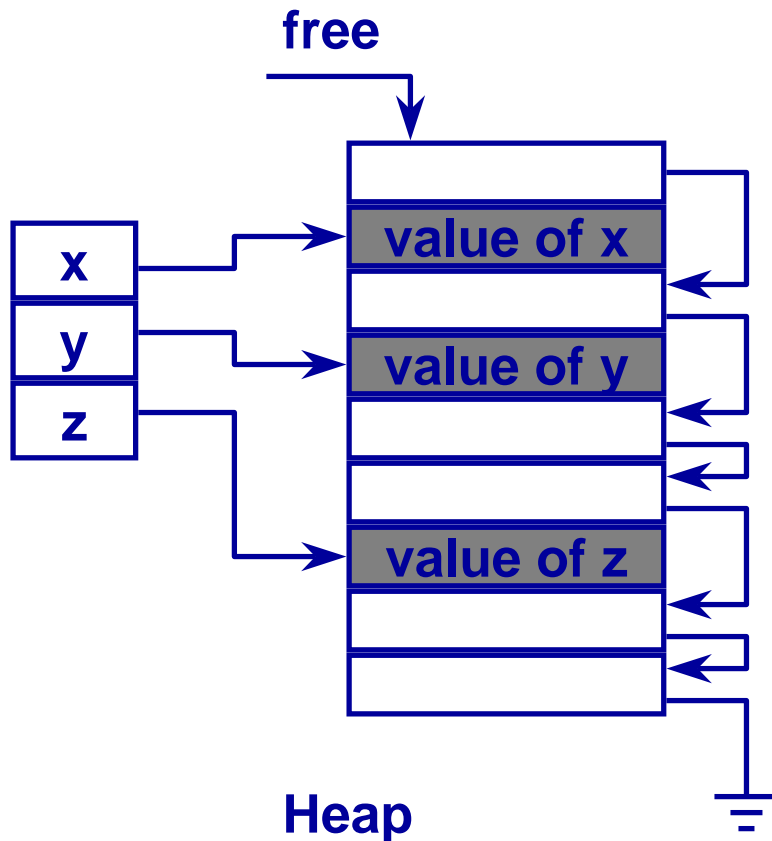
Recursive implementation of fact(n)
(fact.c)

```
int rec_fact(int n) {
    if (n < 0) {
        return -1;
    } else if (n == 0) {
        return 1;
    } else {
        return n * rec_fact(n - 1);
    }
}
```

Non recursive implementation of fact(n)
(fact.c)

```
int stat_fact(int n) {
    int temp, i;
    if (n < 0) {
        return -1;
    } else if (n == 0) {
        return 1;
    } else {
        for (i = 1, temp = 1; i <= n;
            i++) {
            temp *= i;
        }
        return temp;
    }
}
```

Principles - Dynamic Languages VMs (cont)



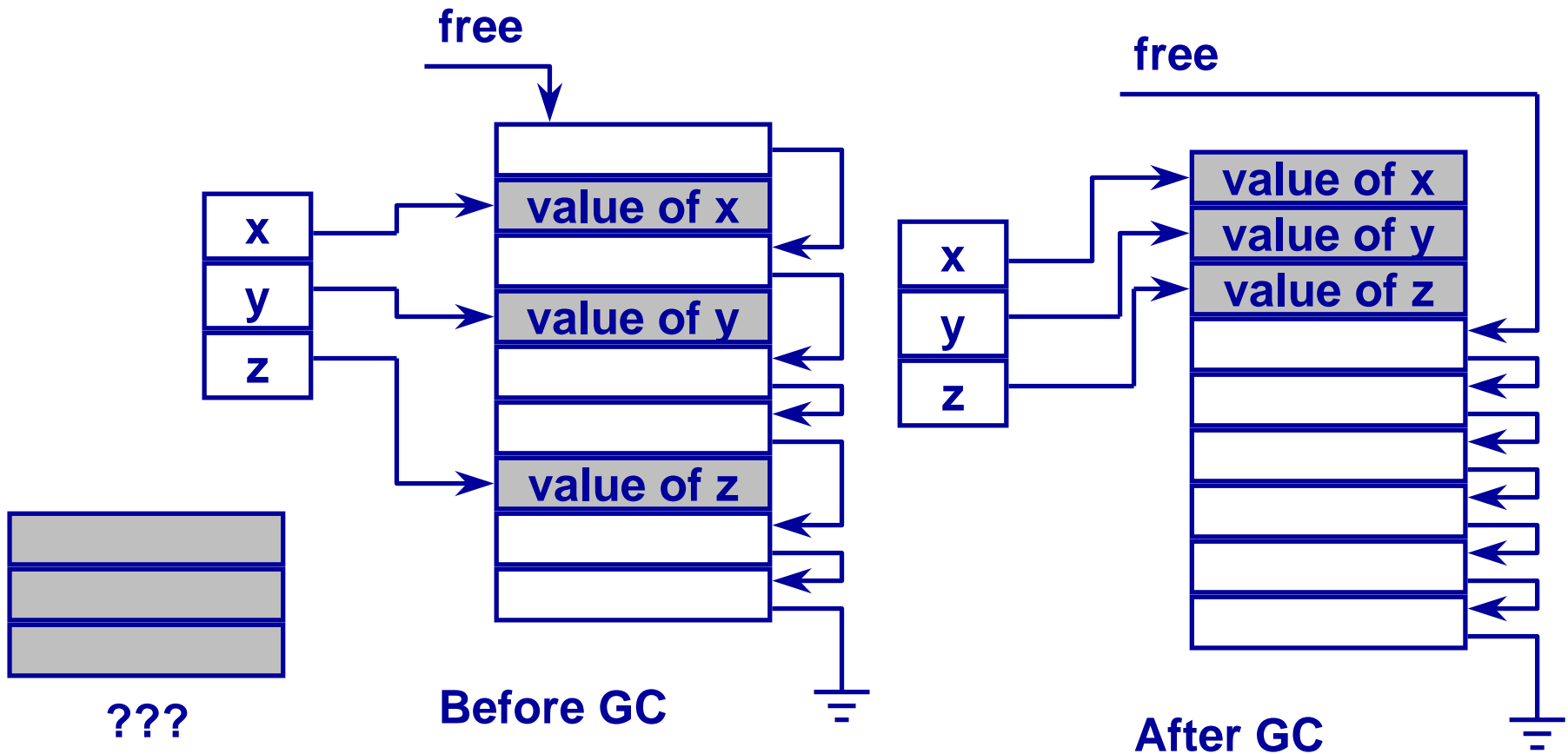
- ◆ The Heap (or Free Store) is a memory area where data items are explicitly allocated and deallocated during the execution of the program. E.g.

```
int x, *y, *z;
x = (int *) malloc(sizeof(int));
y = (int *) malloc(sizeof(int));
z = (int *) malloc(sizeof(int));
free(x);
free(z);
/* OOPS: where is free(y)? */
```

Principles - Dynamic Languages VMs (cont)

- ◆ **The heap provides a useful mechanism to handle data items which are only required in a particular moment and do not need to “stay alive” along all the program execution.**
- ◆ **Data items in the heap can be easily organised** in data structures like lists, trees, etc... very convenient for a relevant set of algorithms.
- ◆ **Data items in the heap require explicit de-allocation.**
- ◆ **A not very careful use of the heap may cause the fragmentation problem.**

Principles - Garbage Collection



- ◆ The **Garbage Collection** is the process of:
 - **(1) making available data items** which are **not referenced** any more (I.e. are not used);
 - **(2) compacting the available free data items** in one single contiguous area, I.e **removing the fragmentation.**
- ◆ When the **garbage collection** is performed automatically by the RTS, it imposes a **non-deterministic behaviour** in the running program.
- ◆ **C and C++ Run Time Supports do not perform any GC.**

Principles - Semi-Dynamic vrs. Dynamic PLs

- ◆ Up to now where we spoke about Dynamic Programming Languages we actually meant Semi-Dynamic Programming Languages.
- ◆ In a Dynamic Programming Language (e.g., LISP, SNOBOL, Java):
 - **all** (the majority of) the **data items** are **allocated in the heap**;
 - **no explicit deallocation** of the data items which are not used any more **is required**;
 - **the garbage collection is handled automatically** by the language RTS.

Principles - Object Orientation

- ◆ “An **object** is a software bundle of variables (attributes) and related methods (member functions). Software objects are often used to model real-world objects you find in everyday life.
- ◆ Software objects interact and communicate with each other using **messages** (I.e. invoking their methods).
- ◆ A **class** is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind.
- ◆ A class **inherits** state and behaviour from its superclass. Inheritance provides a powerful and natural mechanism for organising and structuring software programs.”

[From Javasoft Java tutorial]

Principles - Object Orientation

(cont)

- ◆ **“Encapsulation:** all the details about the composition, structure, and internal workings of an object are hidden from clients. The client's view is generically called the object's interface. The internals of the object are said to be hidden behind its interface. The interface of the computer keyboard with which I'm typing this book consists of labeled keys, in a specific layout, that I can press to generate a character in my word processor. The internal details of how a keystroke is translated into a character on the screen are encapsulated behind this interface. In the same manner, the internal implementation details of a Smalltalk string object are encapsulated behind the public member functions and variables of that Smalltalk object.”

From “Inside OLE”, by K. Brockscmidt, MS Press.

Principles - Object Orientation

(cont)

- ◆ **Polymorphism**: the ability to view two similar objects through a common interface, thereby eliminating the need to differentiate between the two objects. For example, consider the structure of most writing instruments (pens and pencils). Even though each instrument might have a different ink or lead, a different tip, and a different colour, they all share the common interface of how you hold and write with that instrument. All of these objects are polymorphic through that interface—any instrument can be used in the same way as any other instrument that also supports that interface, just as all Slinky toys, regardless of their size and material, act in many ways like any other Slinky. In computer terms, I might have an object that knows how to draw a square and another that knows how to draw a triangle. I can view both of them as having certain features of a "shape" in common, and through that "shape" interface, I can ask either object to draw itself.

From “Inside OLE”, by K. Brockscmidt, MS Press.

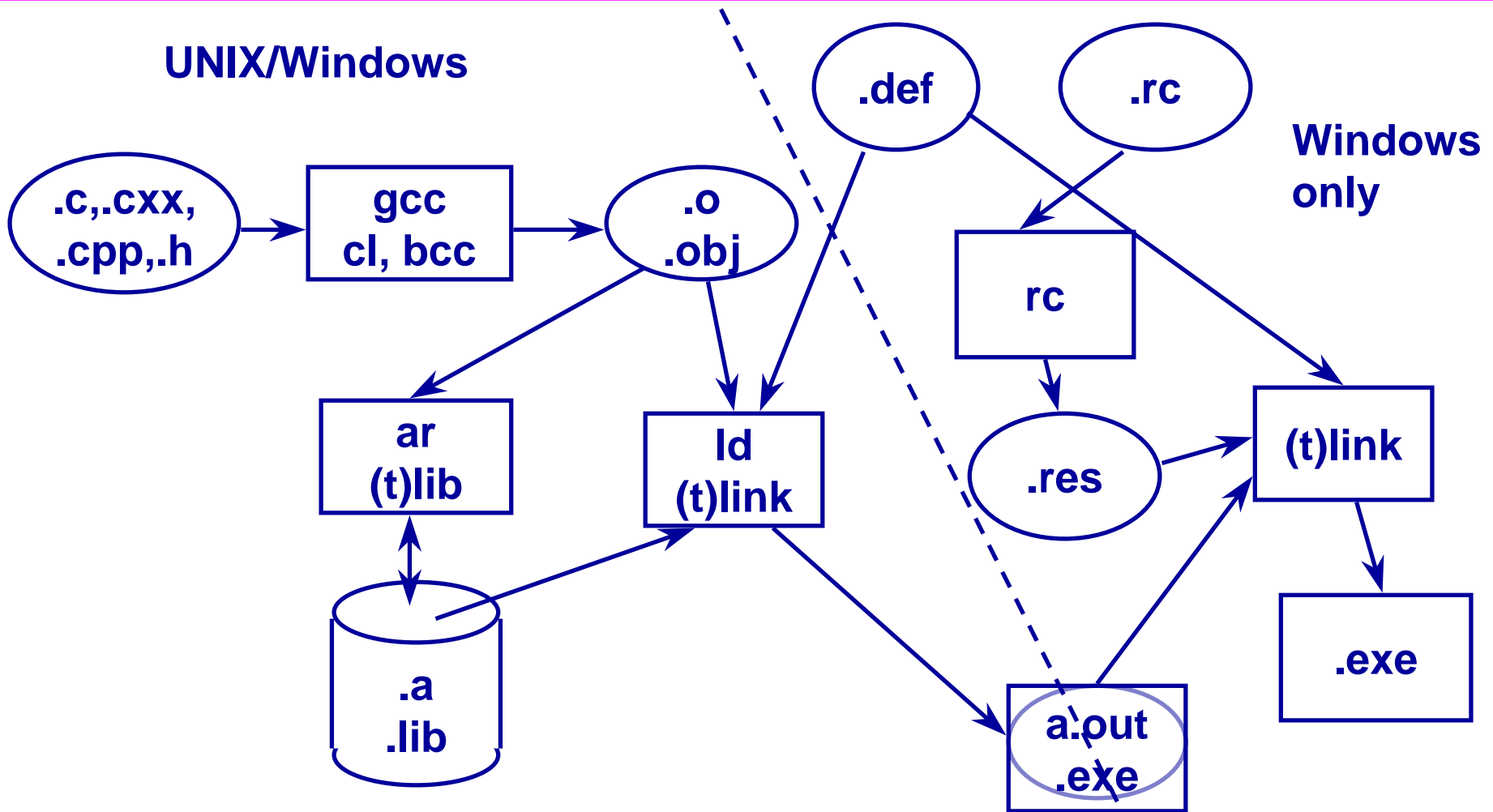
Principles - Object Orientation

(cont)

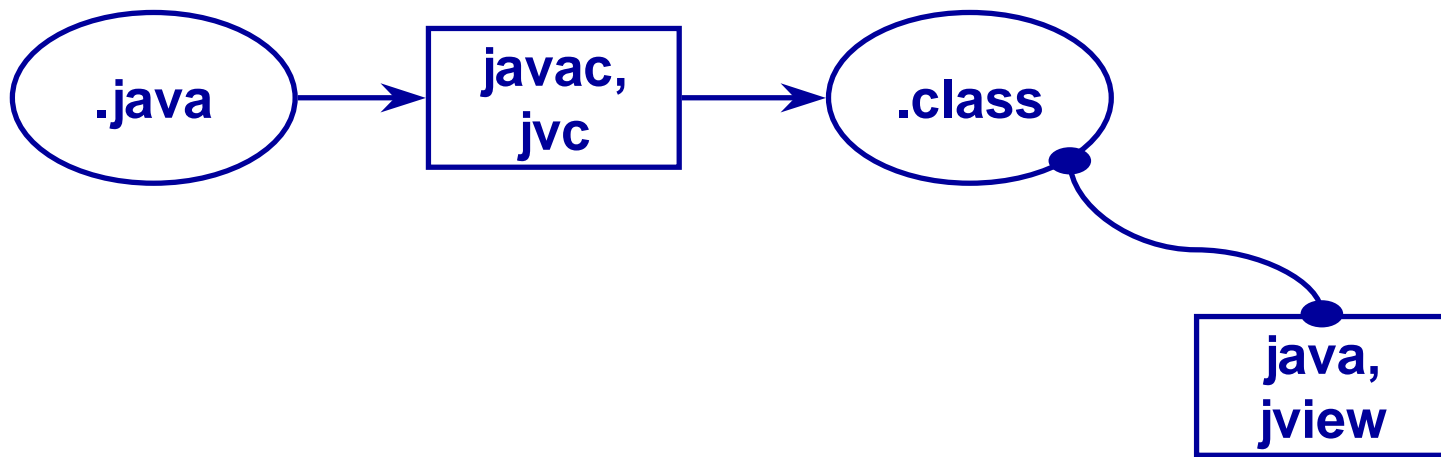
- ◆ **Inheritance**: a method to express the idea of polymorphism for which the similarities of different classes of objects are described by a common base class. The specifics of each object class are defined by a derived class—that is, a class derived from the base class. The derived class is said to inherit the properties and characteristics of the base class; thus, all classes derived from the same base class are polymorphic through that base class. For example, I might describe a base class called "Writing Instruments" and make derived classes of "Ball-point Pen," "Pencil," "Fountain Pen," and so forth. If I wanted to program different "shape" objects, I could define a base class "Shape" and then derive my "Square" and "Triangle" classes from that base class. I achieve polymorphism along with the convenience of being able to centralize all the base class code in one place, within the "Shape" class implementation.

From “Inside OLE”, by K. Brockscmidt, MS Press.

Principles - Application Generation



Principles - Application Generation (cont)



Principles - Summary

- ◆ Syntax vrs. Semantics
- ◆ Virtual machines
- ◆ Statements
- ◆ Program Units
- ◆ l-value & r-value
- ◆ Parameter Transmission
- ◆ Binding Environments
- ◆ Storage Management
 - Static Languages Virtual Machines
 - Dynamic Languages Virtual Machines
 - Garbage Collection
 - Semi-Dynamic vrs. Dynamic Programming Languages
- ◆ Object Orientation
- ◆ Application Generation

Principles - Further Reading

- ◆ A.V. Aho, R. Sethi, J.D. Ullman, “Compilers: Principles, Techniques and Tools”, Addison-Wesley Computer and Engineering Publishing Group, 1986, ISBN: 0-201-10088-6.
- ◆ C. Ghezzi, M. Jazayeri, “Programming Language Concepts”, John Wiley & Sons Inc., 1987, ISBN: 0-471-10426-4.
- ◆ <http://java.sun.com/docs/books/tutorial/java/concepts/index.html> (to get the JavaSoft point of view on object oriented concepts)
- ◆ K. Brodscmidt, “Inside OLE 2nd Edition”, Microsoft Press, ISBN:1-55615-843-2. (In chapter 1 some object oriented concepts are presented according to the Microsoft point of view).