



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

00 - Introduzione

INGEGNERIA INFORMATICA

Introduzione al Corso

Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

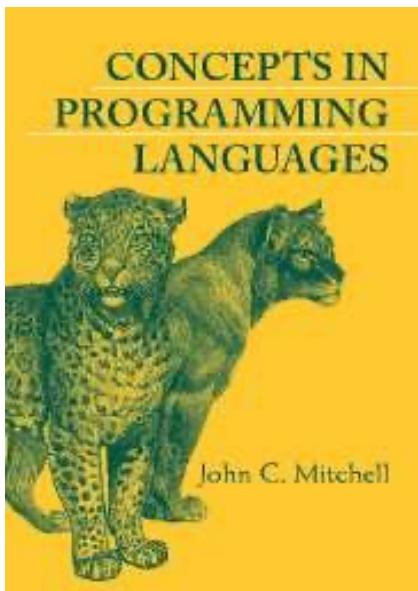
Ore di didattica frontale: 48

Prof. Claudio MENGHI

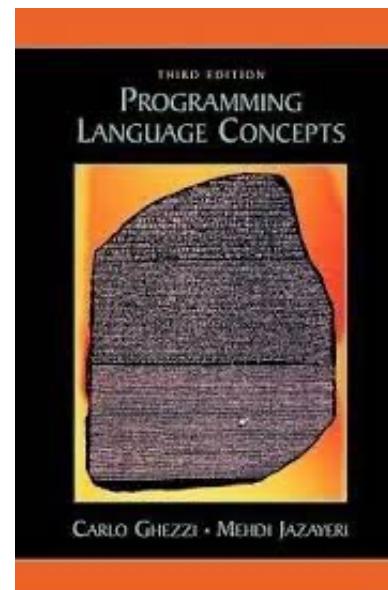
Dalmine

20 Settembre 2023

Introduzione



Chapter 1



Sezione 1.6



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Introduzione sui Linguaggi di Programmazione

- An ideal programming language will make it easy for programmers to write programs succinctly and clearly. Because programs are meant to be understood, modified, and maintained over their lifetime, a good programming language will help others read programs and understand how they work.
- Many software systems consist of interacting parts. These parts, or software components, may interact in complicated ways.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Introduzione sui Linguaggi di Programmazione

- A good language for large-scale programming will help programmers manage the interaction among software components effectively



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Introduzione sui Linguaggi di Programmazione

1970 Fortran

- no ricorsione: ricorsione era considerata inefficiente

1980 Object Oriented Programming

- object-oriented programming too inefficient and clumsy for real programming



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Introduzione sui Linguaggi di Programmazione

- **Computability:** Halting problem implies that programming language compilers and interpreters cannot do everything that we might wish they could do.
- **Static analysis:** Compile time vs run time.
- **Expressiveness versus efficiency:** memory management (Garbage collection)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

The history of modern programming languages begins around 1958-1960 with the development of Algol, Cobol, Fortran, and Lisp.

Fortan and Cobol

- Most programming was done with the native machine language of the underlying hardware.
- This was acceptable because programs were small and efficiency was extremely important.
- The two most important programming language developments of the 1950s were.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Fortan and Cobol

- Fortran (a contraction of formula translator) was that it became possible to use ordinary mathematical notation in expressions (Distributore)
- Cobol is a programming language designed for business applications. (Banks)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Language	Expressions	Functions	Heap storage	Exceptions	Modules	Objects	Threads
Lisp	x	x	x				
C	x	x	x				
Algol 60	x	x					
Algol 68	x	x	x				x
Pascal	x	x	x				
Modula-2	x	x	x		x		
Modula-3	x	x	x	x	x	x	
ML	x	x	x	x	x	x	
Simula	x	x	x			x	x
Smalltalk	x	x	x	x		x	x
C++	x	x	x	x	x	x	
Objective C	x	x	x			x	
Java	x	x	x	x	x	x	x



Java Versions

1996 - 1.0

1997 - 1.1 - RMI and serialization

1998 - 1.2 – Swing

2000 – 1.3 – Support for Windows 95

2002 – 1.4 – assert keyword

2004 – 1.5 – Generics, Autoboxing, Enumerations, Apple Mac OS X 10.5 (Leopard).

2006 – 1.6 – support for annotations

2011 – 1.7 - Strings in switch

2014 – 1.8 - lambda expressions



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

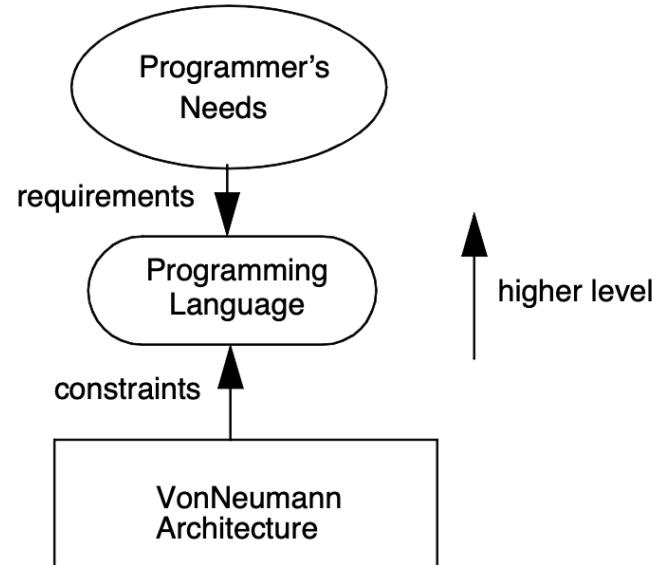


FIGURE 2. Requirements and constraints on a language

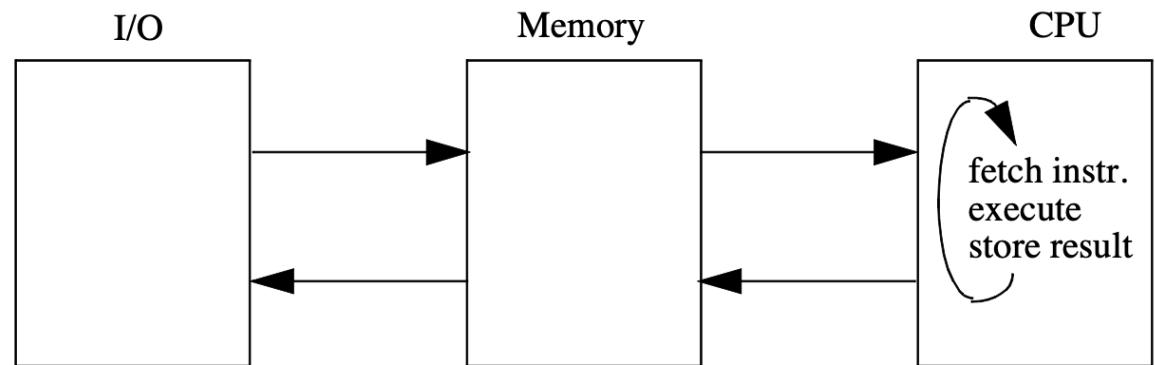


FIGURE 1. A Von Neumann computer architectur



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Paradigms

- **procedural:** they enforce the development of programs based on routines as the unit of modularization (FORTRAN and Pascal)
- **imperative languages:** consists of a sequential step-by-step execution of instructions which change the state of a computation by modifying a repository of values
- **functional programming:** The functional style of programming is rooted in the theory of mathematical functions. It emphasizes the use of expressions and functions.
- **object-oriented languages:** they enforce the development of programs based on object classes as the unit of modularization (Smalltalk and Eiffel)
- **abstract data type programming:** Abstract-data type (ADT) programming recognizes abstract data types as the unit of program modularity.
- **declarative programming:** This style emphasizes the declarative description of a problem, rather than the decomposition of the problem into an algorithmic implementation (PROLOG)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

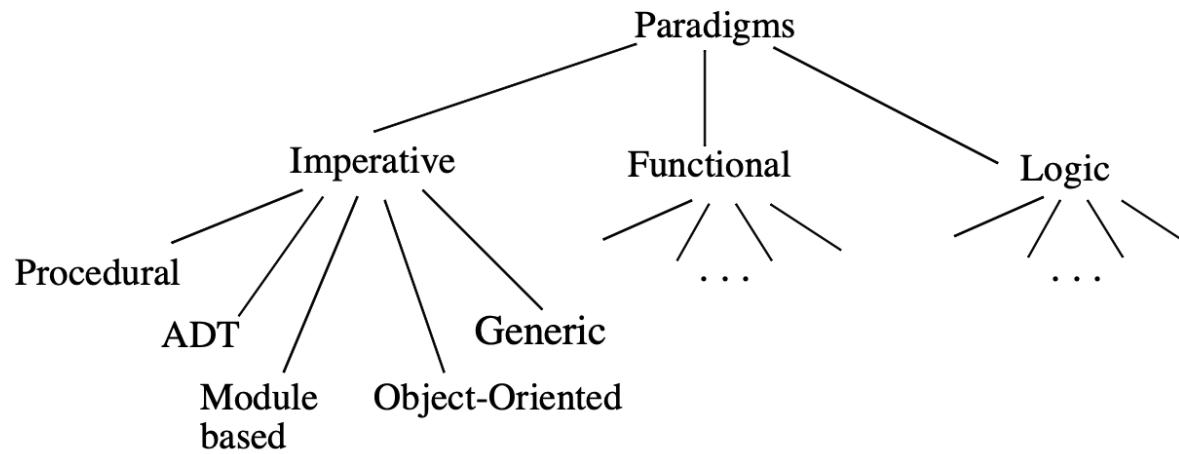


FIGURE 3. Hierarchy of paradigms



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Programming language qualities

- *Software must be reliable.* Users should be able to rely on the software.
- *Software must be maintainable.* Existing software must be modified to meet new requirements or to fix bugs
- *Software must execute efficiently.*



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Programming language qualities

- *Writability.* It refers to the possibility of expressing a program in a way that is natural for the problem
- *Readability.* It should be possible to follow the logic of the program and to discover the presence of errors by examining the program
- *Simplicity.* A simple language is easy to master and allows algorithms to be expressed easily, in a way that makes the programmer self-confident.
- *Safety.* The language should not provide features that make it possible to write harmful programs.
- *Robustness.* The language supports robustness whenever it provides the ability to deal with undesired events



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

01 - Computability and Fundamentals

INGEGNERIA INFORMATICA

Anno di corso: 1

Prof. Claudio MENGHI

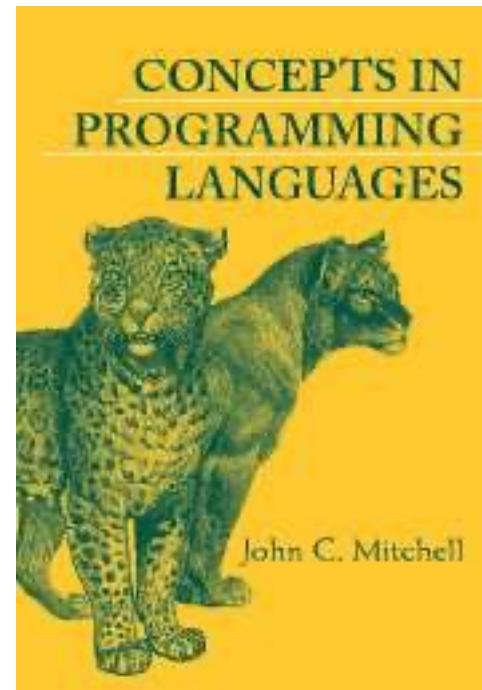
Anno accademico di offerta: 2023/2024

Crediti: 6

Dalmine

Ore di didattica frontale: 48

26 Settembre 2023

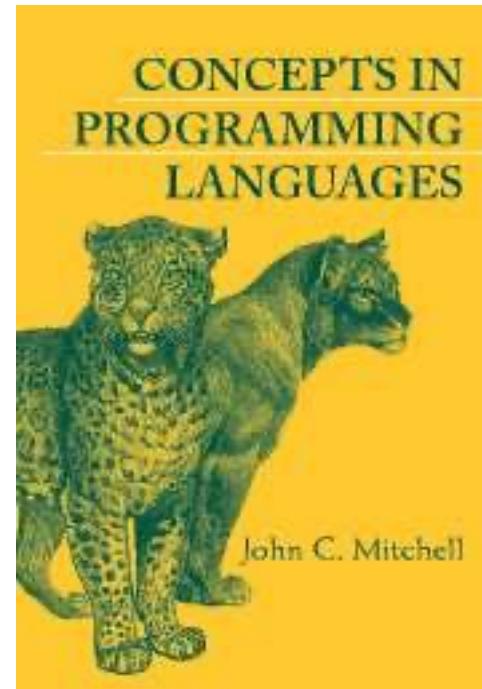


Capitolo 2: computability
Capitolo 4: fundamentals



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Capitolo 2: computability



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sommario

- funzioni “recursively defined” (definite ricorsivamente) e funzioni parziali
- funzioni computabili
- Turing completeness
- Indecidibilità e Halting problem



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

2.1 Funzioni Parziali e Computabilità

- Da un punto di vista matematico un programma è una funzione
- L'output di un programma dipende da (a) lo stato della macchina prima che il programma venga eseguito e (b) dall'input fornito al programma
- Tuttavia, un programma può implementare solo funzioni computabili



2.1.1 Espressioni, Errori e Non Terminazione

- In **matematica**, le espressioni possono avere un valore o meno
 - $3+2$ assume valore 5
 - $3/0$ non è definito



2.1.1 Espressioni, Errori e Non Terminazione

- Nella **computazione** eseguita per mezzo di calcolatori ci sono varie ragioni per cui un'espressione può non ritornare un valore
 - **Errore:** si verifica quando c'è un problema (per esempio la valutazione di un'espressione su due operandi non compatibili)
Esempio: divisione per zero
 - **Non terminazione:** si verifica quando la computazione procede all'infinito senza mai produrre un risultato
Esempio: $f(x:\text{int}) = \text{if } x = 0 \text{ then } 0 \text{ else } x + f(x-2)$
 $f(4)$ termina
 $f(5)$ non termina mai
[Hmmm davvero? 1pt bonus]



2.1.1 Espressioni, Errori e Non Terminazione

- [-2147483648 to 2147483647] 32 bits
 - -2147483643
 - -2147483645
 - -2147483647
 - 2147483647
 - 2147483645
 - 2147483643



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

2.1.1 Espressioni, Errori e Non Terminazione

- `java.lang.StackOverflowError`



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

2.1.2 Funzioni Parziali

- Una funzione partiziale è definita per certi argomenti ma non per altri ovvero può ritornare un risultato per qualche input ma non terminare per altri.
- Una funzione $f: A \rightarrow B$ da un insieme A a un insieme B è una *regola* che associa un unico valore $y=f(x)$ appartenente all'insieme B per ogni input x di A .
 - A è il dominio di f
 - B è il codominio di f



2.1.2 Funzioni Parziali

- Una funzione $f: A \rightarrow B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano le seguenti condizioni
 - Se $\langle x, y \rangle \in f$ e $\langle x, z \rangle \in f$, allora $y = z$
 - Per ogni $x \in A$, esiste un $y \in B$ con $\langle x, y \rangle \in f$
- Una funzione parziale $f: A \rightarrow B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano la seguente condizione
 - Se $\langle x, y \rangle \in f$ e $\langle x, z \rangle \in f$, allora $y = z$

Esempio: $f(x:\text{int}) = \text{if } x = 0 \text{ then } 0 \text{ else } x + f(x-2)$ è una funzione parziale [Termina solo se x è pari]



2.1.3 Computabilità

- Una funzione è computabile se c'è un programma che la computa, ovvero
- Una funzione $f: A \rightarrow B$ è computabile se esiste un algoritmo che, dato in input un qualsiasi input $x \in A$ termina e ritorna $y = f(x)$ come output
- È possibile che l'implementazione di tale algoritmo sia possibile in un linguaggio di programmazione ma non in un altro.



2.1.3 Computabilità

- La classe di funzioni sui numeri naturali che sono computabili in principio è la classe delle funzioni parziali ricorsive
 - La ricorsione è essenziale per la computazione
 - Le funzioni sono parziali in generale



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

2.1.3 Computabilità

- A function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.

Ci sono tre dimostrazioni

- Alonso Church
 - Lamda Calculus
- Alan Turing
- Tutti i linguaggi di programmazione sono Turing complete



2.1.3 Computabilità

- La macchina di Turing ha
 - Un nastro infinito sul quale è possibile leggere e scrivere e un controllore (a stati finiti)
 - Il nastro è diviso in un insieme di celle diviso
 - Il controllore può decidere se leggere o scrivere dal nastro o muoversi di una cella a sinistra o a destra



2.1.3 Computabilità

- Halting problem: dato un (generico) programma P che riceve una stringa x come input, determinare se il programma P termina quando riceve la stringa x
- Possiamo associare l'halting problem con una funzione f_{halt} tale che
 - $f_{\text{halt}}(P,x)=\text{halt}$ se il programma P termina per l'input x
 - $f_{\text{halt}}(P,x)=\text{not halt}$ se il programma P non termina per l'input x

L'halting problem è indecidibile: la funzione p non è computabile (in generale)



2.1.3 Computabilità

- Supponiamo che esista un programma Q che risolva l'halting problem
 - $$Q(P, x) = \begin{cases} \text{halt} & \text{se } P(x) \text{ termina} \\ \text{not halt} & \text{se } P(x) \text{ non termina} \end{cases}$$
- Utilizzando Q creiamo un programma D che a volte non termina
 - $D(P) = \begin{cases} \text{se } «Q(P, P)=\text{halt}» \text{ allora run forever} \\ \text{altrimenti halt} \end{cases}$



2.1.3 Computabilità

- Supponiamo che esista un programma Q che risolva l'halting problem
 - $D(P) = \begin{cases} \text{halt} & \text{se } P(P) \text{ non termina} \\ \text{not halt} & \text{se } P(P) \text{ termina.} \end{cases}$
- Consideriamo il comportamento di D(D)
 - $D(D) = \text{halt se } D(D) \text{ non termina}$
 - $D(D) = \text{not halt se } D(D) \text{ termina}$



2.1.3 Computabilità

- Halting problem: dato un (generico) programma P che riceve una stringa x come input, determinare se il programma P termina quando riceve la stringa x
- Possiamo associare l'halting problem con una funzione f_{halt} tale che
 - $f_{\text{halt}}(P,x)=\text{halt}$ se il programma P termina per l'input x
 - $f_{\text{halt}}(P,x)=\text{not halt}$ se il programma P non termina per l'input x

L'halting problem è indecidibile: la funzione p non è computabile (in generale)

Davvero??? [1pt bonus]



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

2.1.3 Computabilità

- Computabili “in principio”: sono le funzioni che sono computabili
- Computabili “in pratica”: alcune delle funzioni computabili in principio richiedono moltissimo tempo. Se una funzione non ritornerà un valore in un quantitativo di tempo pari alla durata della storia dell'universo all'ora non è computabile in pratica



2.1.3 Computabilità

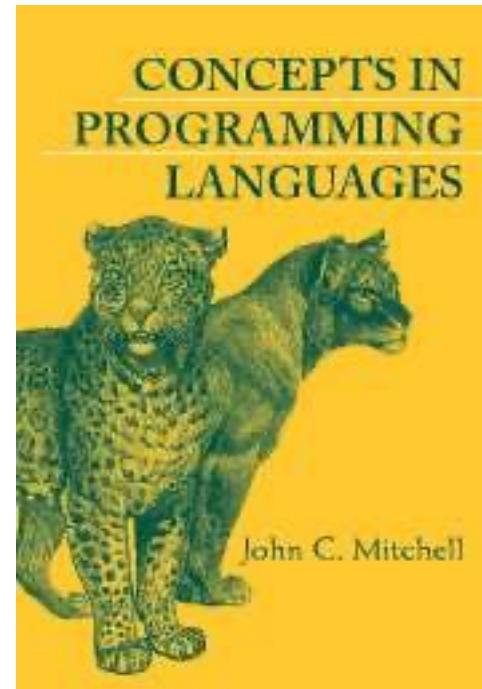
```
i=0;  
while(i!=f(i)) i=g(i);  
printf(...i.....);
```

Può il compilatore capire se il programma termina?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Capitolo 4: fundamentals



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Capitolo 4

- Descrizione di un compilatore e parser
- Lambda calculus
- Denotational semantics
- Linguaggi imperativi e funzionali



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.1.1 Structure of a Simple Compiler

- Sintassi: Il testo di un programma
- Semantica: quello che il programma significa [quello che fa]



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.1.1 Structure of a Simple Compiler

```
3 public class Test {  
4     public static int counter=0;  
5  
6     superpublic| static void main(String[] args) {  
7         check(-2147483641);  
8     }  
9  
10 }  
11  
12     public static int check(int arg) {  
13         if(counter<20) {  
14             System.out.println(arg);  
15             counter++;  
16         }  
17         if(arg==0) return 0;  
18         else return check(arg-2);  
19     }  
20  
21 }  
22 }
```

```
?  
3 public class Test {  
4     public static int counter=0;  
5  
6     public static void main(String[] args) {  
7         check(-2147483641);  
8     }  
9  
10 }  
11  
12     public static int check(int arg) {  
13         if(counter<20) {  
14             System.out.println(arg);  
15             counter++;  
16         }  
17         if(arg==0) return 0;  
18         else return check(arg-2);  
19     }  
20  
21 }  
22 }
```

Problems @ Javadoc Declaration Console X
Runned> Test (1) [Java Application] /Users/admin/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.6.v2023020
StackOverflowError
at Test/test.Test.check([Test.java:18](#))
at Test/test.Test.check([Test.java:18](#))



4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina
- **Interprete**: combina la traduzione con l'esecuzione del programma

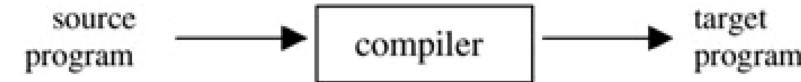


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

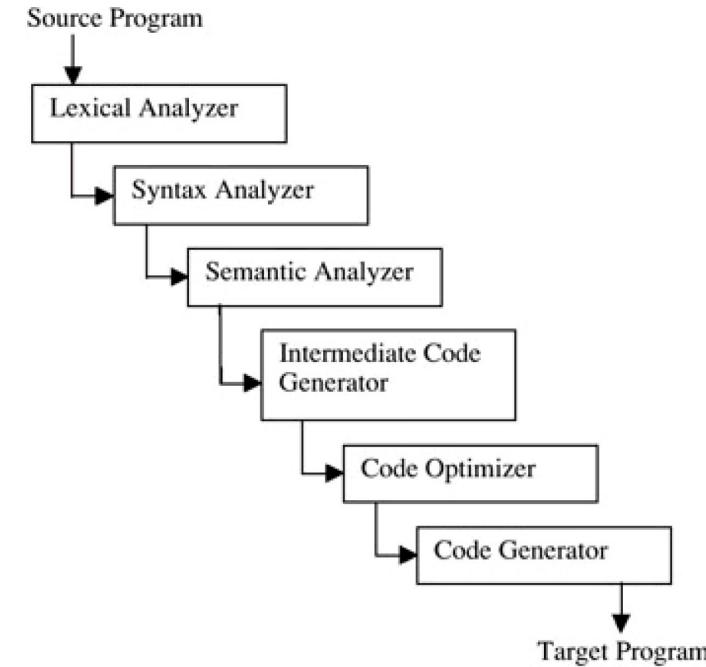
4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



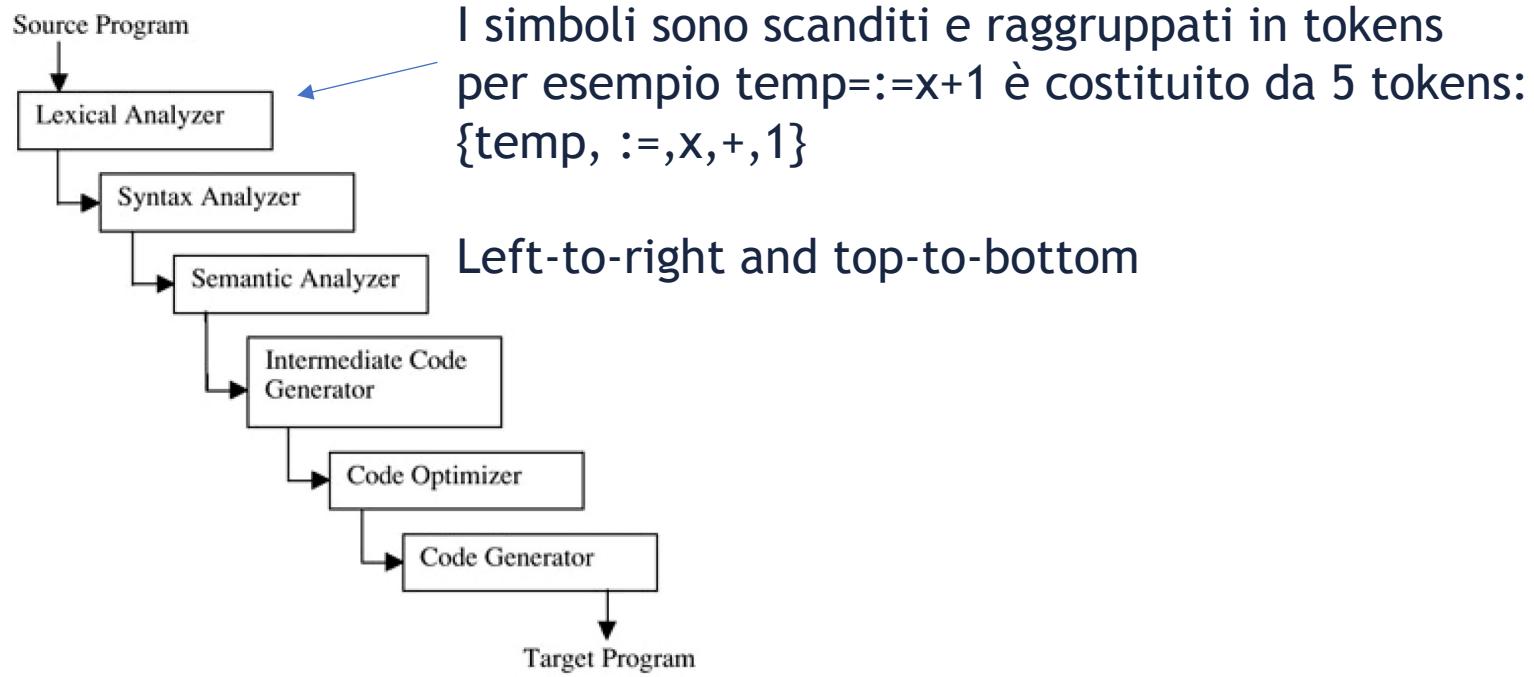
4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



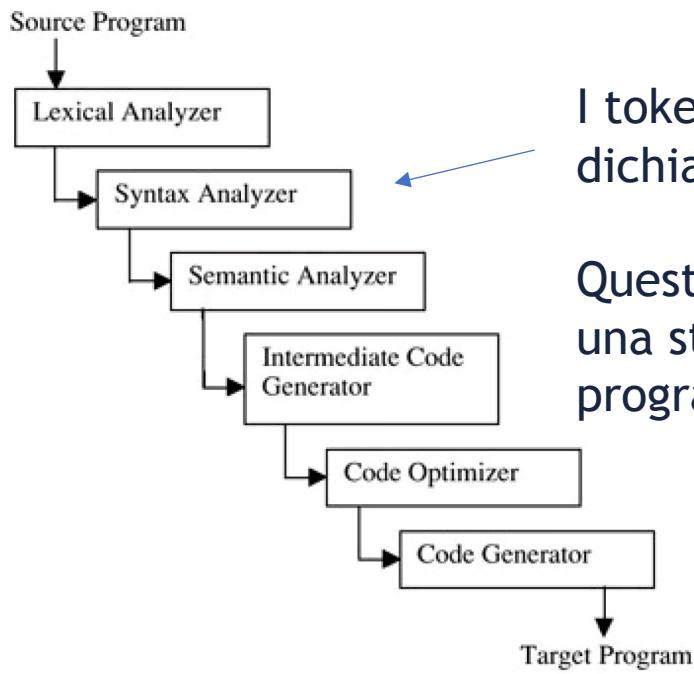
4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



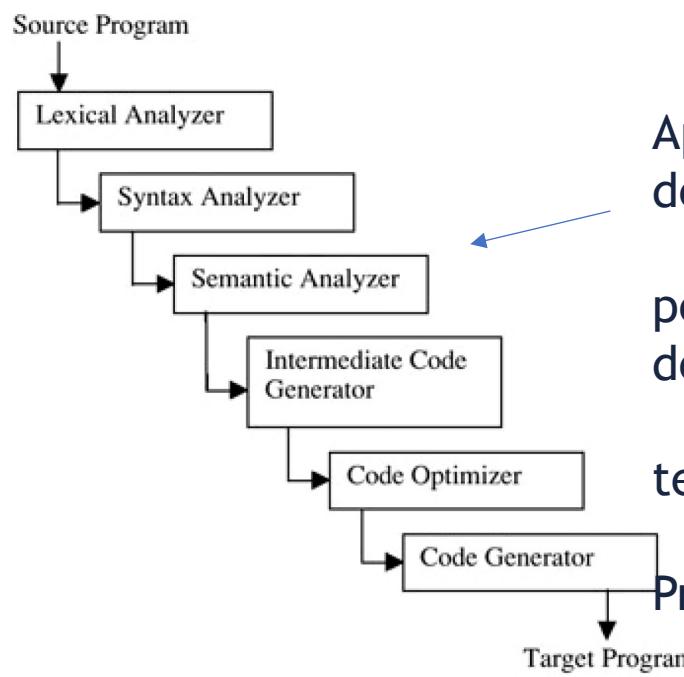
I token sono raggruppati in espressioni, statements e dichiarazioni in base alle regole della grammatica

Questa attività è eseguita dal parser. L'obiettivo è creare una struttura chiamata «parse tree» che rappresenta il programma



4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



Applica regole e procedure che dipendono dal contesto delle espressioni

per esempio controllare che i tipi all'interno dell'espressione

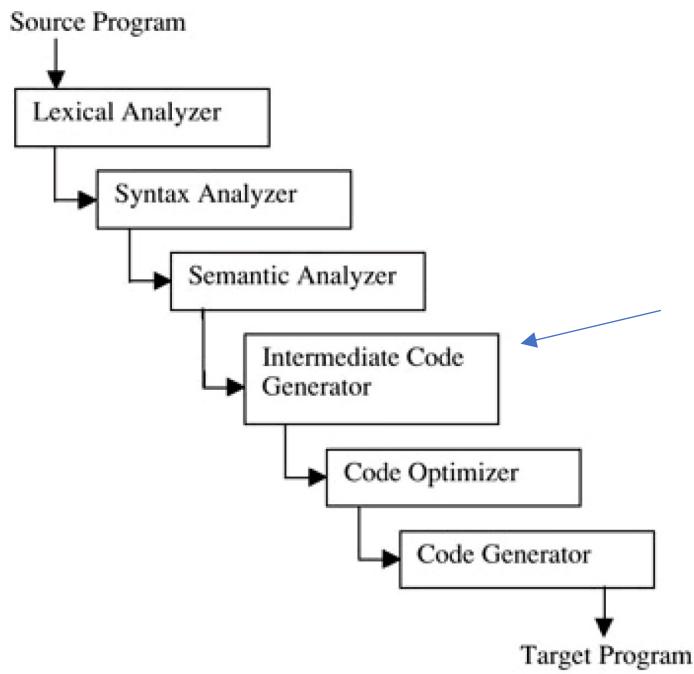
temp:=x+1 siano consistenti

Produce un «augmented parse-tree»



4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina

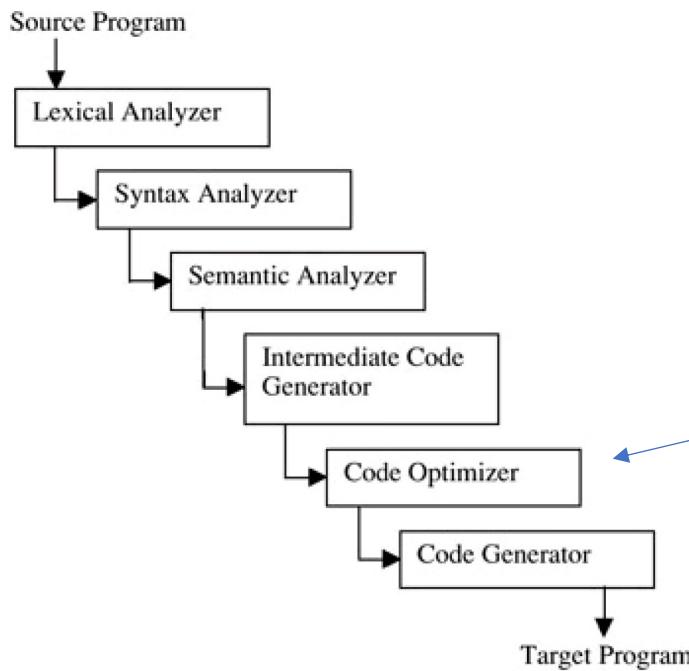


Producono una versione intermedia del codice per poi procedere a delle ottimizzazioni



4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



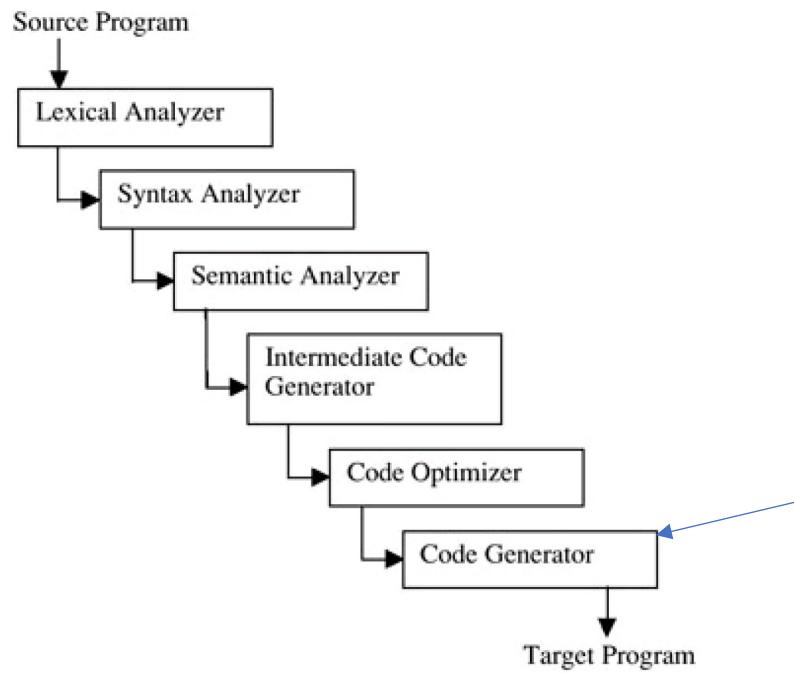
Applica un insieme di tecniche per ottimizzare il codice

- elimina sottoespressioni (se la stessa espressione è computata + di una volta)
- $x=y$ sostituisce y a x .
- elimina codice morto
- cerca di rimuovere istruzioni dai loop
- rimpiazza una funzione con il corrispettivo codice



4.1.1 Structure of a Simple Compiler

- **Compilatore:** traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



Converte il codice intermedio nel linguaggio del target program



4.1.2 Grammatiche e Parse Trees

- Grammatiche: forniscono un metodo per definire un insieme (infinito) di espressioni. Le grammatiche sono composte da
 - Un simbolo iniziale
 - Un insieme di non-terminali
 - Un insieme di terminali
 - Un insieme di regole di produzione

I non terminali sono simboli utilizzati per scrivere la grammatica. I terminali sono simboli che appariranno nel linguaggio



4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

Start symbol

$e ::= n \mid e+e \mid e-e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

Non terminali

↓

e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

$$\begin{aligned}e &::= n \mid e+e \mid e-e \\n &::= d \mid nd \\d &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\end{aligned}$$

Terminali



4.1.2 Grammatiche e Parse Trees

$e ::= n \mid e+e \mid e-e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Nel linguaggio

0, 1+3+5, 2+4 - 6 - 8

Non nel linguaggio

e, e+e, e+6 - e



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.1.2 Grammatiche e Parse Trees

$e ::= n \mid e+e \mid e-e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Derivazione: sequenza di step di “rimpiazzo” che porta a una stringa di terminali

$e \rightarrow n \rightarrow nd \rightarrow dd \rightarrow 2d \rightarrow 25$

$e \rightarrow e - e \rightarrow e - e+e \rightarrow \dots \rightarrow n-n+n \rightarrow \dots \dots 10-15+12$



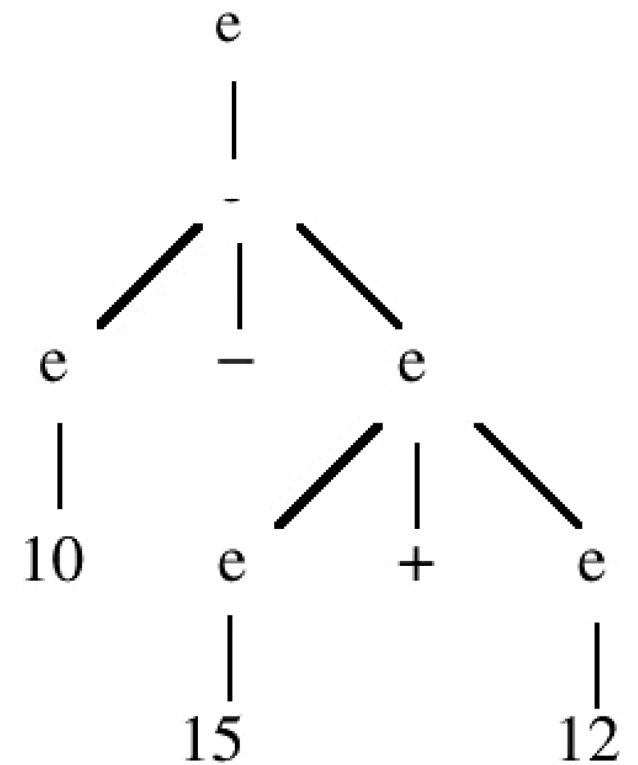
4.1.2 Grammatiche e Parse Trees - Ambiguità

10–15+12

$e ::= n \mid e+e \mid e-e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

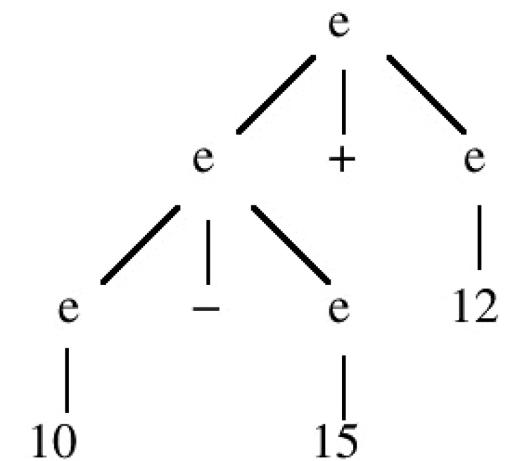
4.1.2 Grammatiche e Parse Trees - Ambiguità

10–15+12

$e ::= n \mid e + e \mid e - e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



4.1.2 Grammatiche e Parse Trees - Ambiguità

- Una grammatica è ambigua se la stessa espressione ha più di un «parse tree»



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.1.2 Grammatiche e Parse Trees - Precedenza

- Parsing: procedura di costruzione di un parse tree da una sequenza di simboli
- Parsing algorithm: un algoritmo che capisce quando una stringa appartiene a un linguaggio (e costruisce il corrispondente parse tree) è chiamato parsing algorithm.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.2 Lamda Calculus (λ)

4.2 Lamda Calculus (λ)

Notazione per descrivere la computazione

Composta da tre parti

- notazione per descrivere le funzioni
- meccanismo di prova per descrivere equazioni tra espressioni
- set di regole di calcolo chiamate riduzioni



4.2 Lamda Calculus (λ)

- una funzione è una regola per determinare un valore da degli argomenti

$$f(x) = x^2 + 3$$

- $h(x) = f(g(x))$

- la funzione h è definita dall'applicazione della funzione f alla funzione g .



4.2 Lamda Calculus (λ)

I due concetti principali del lambda calculus sono:

- lambda abstractions. If M è un espression, $\lambda x.M$ è la funzione che otteniamo trattando M come una funzione della variable x
 - per esempio $\lambda x.x$ è una astrazione che funzione la funzione di identità dato un x ritorna il suo valore oppure alternativamente
 - $I(x)=x$
- application. Per applicare una funzione ad un'altra, possiamo mettere l'espressione davanti all'altra
 - per esempio, possiamo applicare la funzione di identità all'espressione M scrivendo $(\lambda x.x)M$ (ovvero, dato un M ritorna se stesso)
 - $(\lambda x.x)M=??$
 - $(\lambda x.x)M=M$



4.2 Lamda Calculus (λ): Expression

Dato un insieme di variabili V con $x \in V$ una lambda expression è definita come:

λ term $\longrightarrow M ::= x \mid M \ M \mid \lambda x. M$

$M_1 \ M_2$ corrisponde all'applicazione di M_1 ad M_2

$\lambda x. M$ è la λ abstraction che dato un argomento x ritorna il valore M



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.2 Lamda Calculus (λ):

Linguaggio di programmazione = applied λ -calculus = pure λ -calculus+additional data types



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.2 Lamda Calculus (λ):

Variable binding

- free variable: variabile che non è “dichiarata nell’espressione” (opposto bounded variable)
 - esempio $x+3$
 - esempio $\lambda x.x+3$
 - esempio $\int f(x) dx$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell’Informazione e della Produzione

4.2 Lamda Calculus (λ):

Starting with **0** not applying the function at all, proceed with **1** applying the function once, **2** applying the function twice, **3** applying the function three times, etc.:

Number	Function definition	Lambda expression
0	$0 f x = x$	$0 = \lambda f. \lambda x. x$
1	$1 f x = f x$	$1 = \lambda f. \lambda x. f x$
2	$2 f x = f (f x)$	$2 = \lambda f. \lambda x. f (f x)$
3	$3 f x = f (f (f x))$	$3 = \lambda f. \lambda x. f (f (f x))$
\vdots	\vdots	\vdots
n	$n f x = f^n x$	$n = \lambda f. \lambda x. f^{\circ n} x$



4.2 C vs Lamda Calculus (λ):

- in C an assignment statement has side effects
- in lamda calculus gli assignment sono puramente funzionali



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.4 Functional and Imperative Languages

- natural language (linguaggi naturali): linguaggi utilizzati dagli umani
 - Ambiguità
 - frasi imperative “prendi il pesce”
 - frasi dichiarative “a Claudia piacciono le banane”
 - frasi interrogative e quesiti



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.4 Functional and Imperative Languages

```
{ int x=1;          /* declares new x */  
    x = x+1;        /* assignment to existing x */  
    { int y = x+1;  /* declares new y */  
        { int x = y+1; /* declares new x */  
    } } }
```

Imperative



4.3 Denotational Semantics

- Nella semantica denotazionale un programma è una funzione matematica da stato a stato.
- Lo stato è una funzione matematica che rappresenta i valori della memoria in un determinato stato dell'esecuzione di un programma
 - $x := 0; y := 0; \text{while } x \leq z \text{ do } y := y + x; x := x + 1$



4.3.2 Denotational Semantics of Binary Numbers

Grammatica

$e ::= n \mid e+e \mid e-e$

$n ::= b \mid nb$

$b ::= 0 \mid 1$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

4.3.2 Denotational Semantics of Binary Numbers

Grammatica

$e ::= n \mid e+e \mid e-e$

$n ::= b \mid nb$

$b ::= 0 \mid 1$

Semantica

$$E[[0]] = 0$$

$$E[[1]] = 1$$

$$E[[nb]] = E[[n]] * 2 + E[[b]]$$

$$E[[e_1+e_2]] = E[[e_1]] + E[[e_2]]$$

$$E[[e_1-e_2]] = E[[e_1]] - E[[e_2]]$$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

P1: Ho implementato una funzione f che genera tutte le possibili stringhe composte da “a e b” di lunghezza inferiore di ≤ 5 ?

Vero o Falso? Mi fido?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

P2: Ho implementato una funzione f che dato un programma p (passato come parametro) trova **tutte** le istanze di codice morto?

Vero o Falso? Mi fido?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

P3: Ho implementato una funzione f che genera tutte le combinazioni di stringhe composte dai caratteri “a” e “b” di lunghezza inferiore di 5?

Vero o Falso? Mi fido?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

02 - Memory Management

INGEGNERIA INFORMATICA

Programmazione Avanzata

Prof. Claudio MENGHI

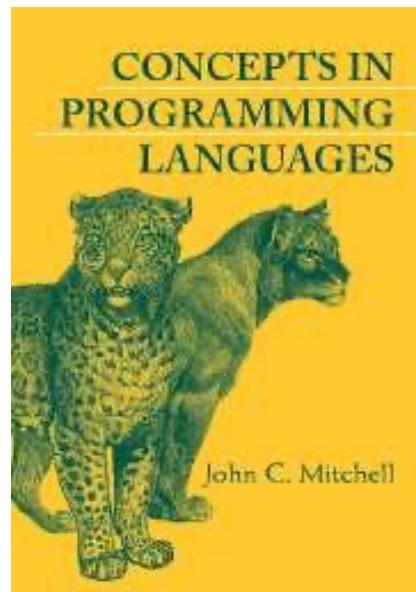
Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

Dalmine

27 Settembre 2023



Capitolo 7 - Scope, Functions, and Storage Management



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Introduzione

- Quando dichiariamo una variabile, il computer dove la memorizza?
- Quali sono le regole per accedere ad una variabile?
- Come vengono passate ai sottoprogrammi i dati?
- Principali feature:
 - Divisione di un programma in sottoprogrammi
 - Non come il BASIC
 - Non unica sequenza di istruzioni (con GOTO)
 - Non si sanno tutte le variabili prima dell'esecuzione e l'allocazione della memoria avviene dinamicamente
 - Non costringo al programmatore di dichiarare tutte le variabili fin dall'inizio
 - Uso della ricorsione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Scope, Functions, and Storage Management

- Scope: permette l'associazione di aree di programma ad aree di memoria
- Function calls: richiedono una nuova area di memoria dove salvare parametri e variabili
- Tail recursions (o tail calls)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Scope, Functions, and Storage Management

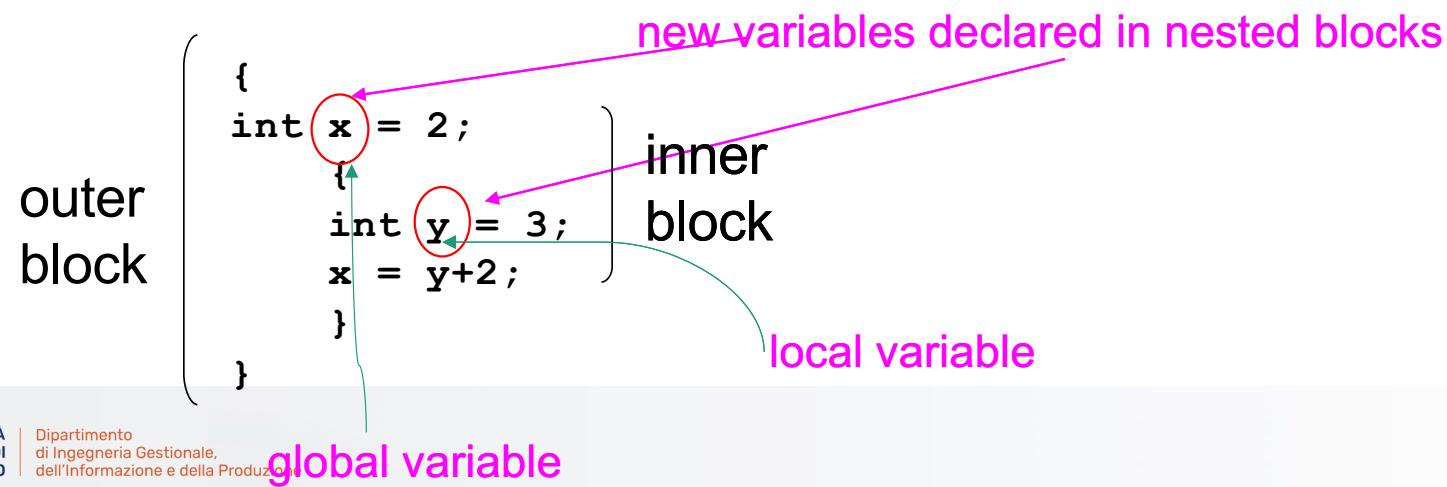
- Block-structured languages and stack storage
- In-line Blocks
 - activation records
 - storage for local, global variables
- First-order functions
 - parameter passing
 - tail recursion and iteration
- NO - Higher-order functions
 - deviations from stack discipline
 - language expressiveness => implementation complexity



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Block-Structured Languages

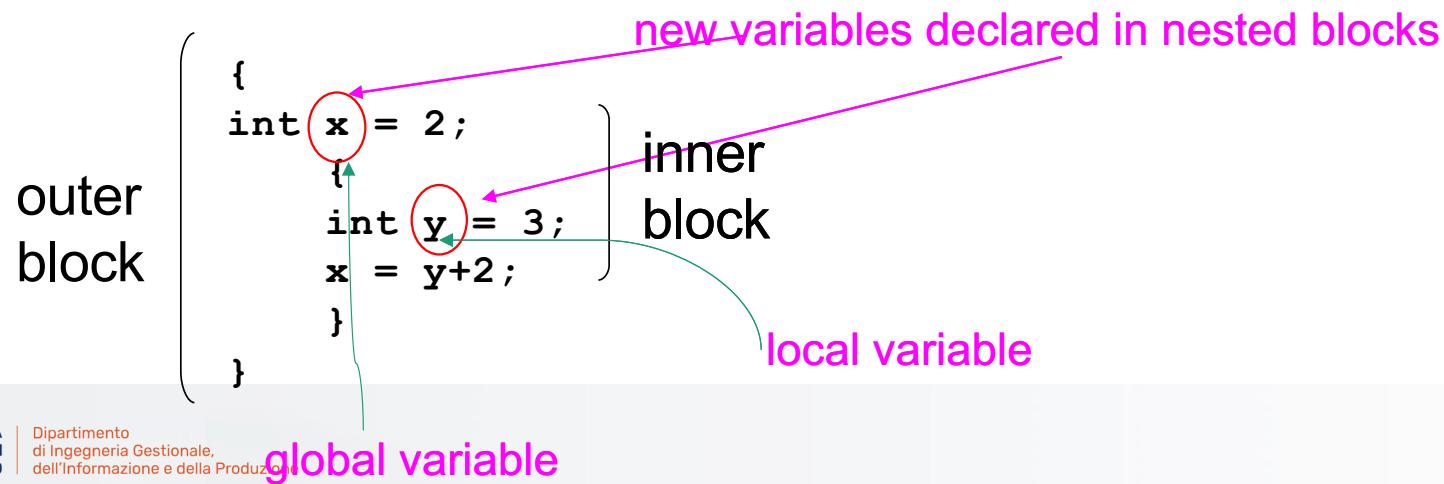


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Block-Structured Languages

- Storage management
 - Enter block: allocate space for variables
 - Exits block: some or all space may be deallocated
- Nested blocks, local variables



Examples

- Blocks in common languages
 - C/c++/Java { ... }
 - Algol begin ... end
 - ML let ... in ... end
- Two forms of blocks
 - In-line blocks
 - Blocks for control structure like if, for and so on.. similar to block inline
 - Blocks associated with functions or procedures
- Topic: block-based memory management, access to *local variables, parameters, global vars*
- It allows **recursive functions**



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Alcune note

Alcuni linguaggi (come Fortran) allocavano in modo fisso le variabili

Svantaggi ...

Ricorsione?

Block-structured languages:

New variables may be declared at various points in a program

Each declaration is visible within a block

When a program begins executing the instructions contained in a block, the memory is allocated

When a program exits, the memory is freed

An identifier that is not declared in the current block is considered global to the block



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Alcune note

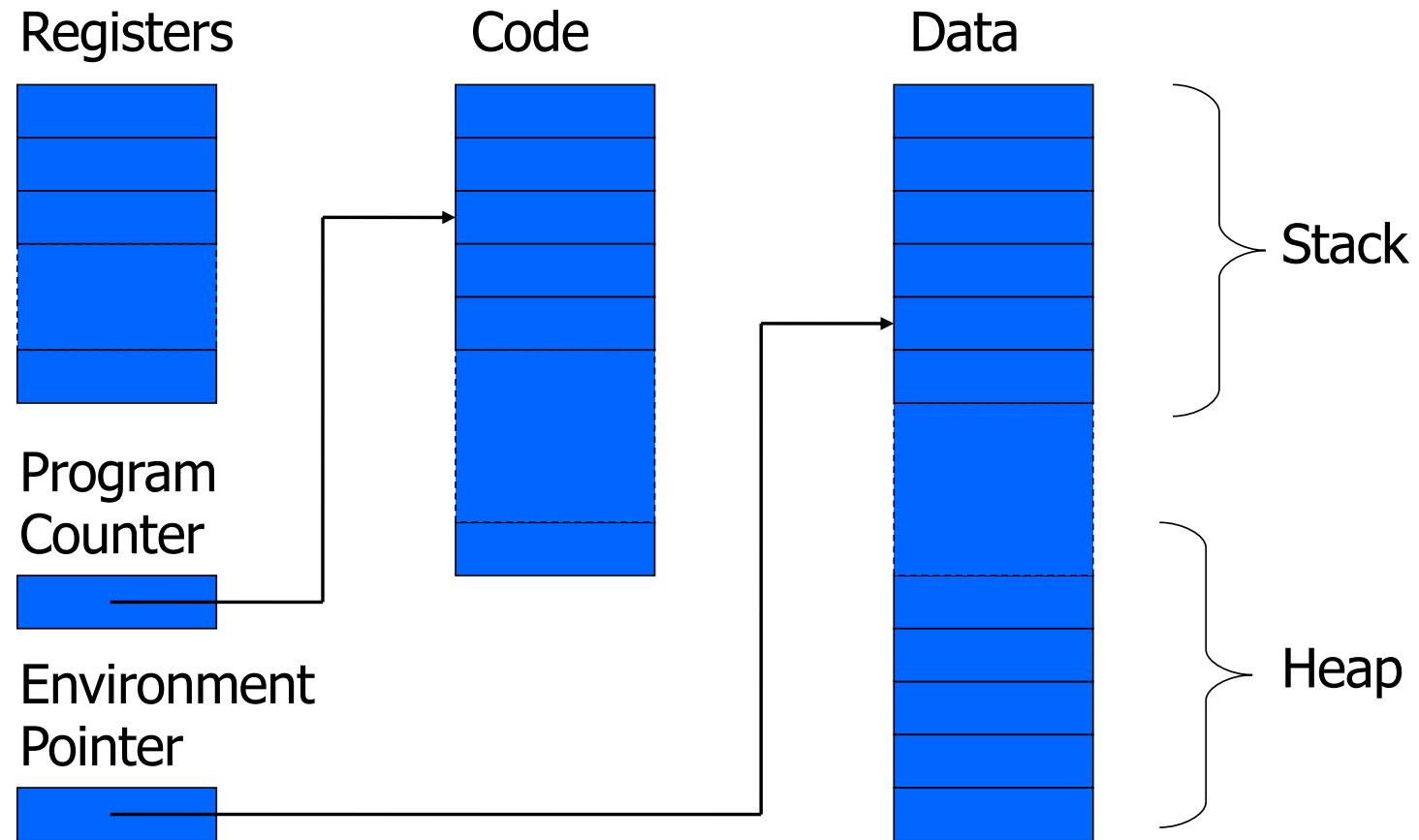
c e c++ non consentono la dichiarazione di funzioni locali all'interno di blocchi innestati.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Simplified Machine Model



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Interested in Memory Mgmt Only

- Registers, Code segment, Program counter
 - Ignore registers
 - Details of instruction set will not matter
- Data Segment
 - Stack contains data related to block entry/exit
 - Heap contains data of varying lifetime
 - **Environment pointer** points to current stack position
 - Block entry: add new activation record to stack
 - Block exit: remove most recent activation record



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In-line Blocks

- Sono blocchi che non sono il corpo di una funzione o di una procedura

```
{ int x=0;  
    int y=x+1;  
    { int z=(x+y)*(x-y);  
    }  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In-line Blocks

- Activation record
 - Data structure stored on run-time stack
 - Contains space for local variables (if any)
- Example

```
{ int x=0;  
    int y=x+1;  
    { int z=(x+y)*(x-y);  
    }  
}
```

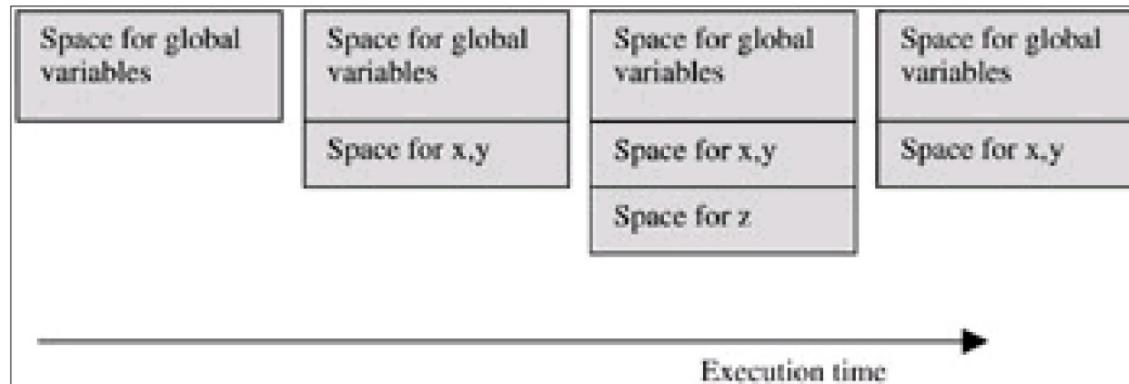
Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In-line Blocks



```
{ int x=0;  
    int y=x+1;  
    { int z=(x+y)*(x-y);  
    }  
}
```



Intermediate results on the stack

May need space for variables and intermediate results like $(x+y)$, $(x-y)$

Example:

```
int z = (x+y) *(x-y)
```

```
push x+y
```

```
push x-y
```

```
a1= pop a2 =pop push a1*a2
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Some basic concepts

- Scope
 - Region of program text where declaration is visible
- Lifetime
 - Period of time when location is allocated to program

```
{ int x = ... ;  
  { int y = ... ;  
    { int x = ... ;  
      ....  
    }  
  }  
}
```

- Inner declaration of x hides outer one.
- Called “hole in scope”
- Lifetime of outer x includes time when inner block is executed
- Lifetime ≠ scope
- Lines indicate “contour model” of scope.



Control Link

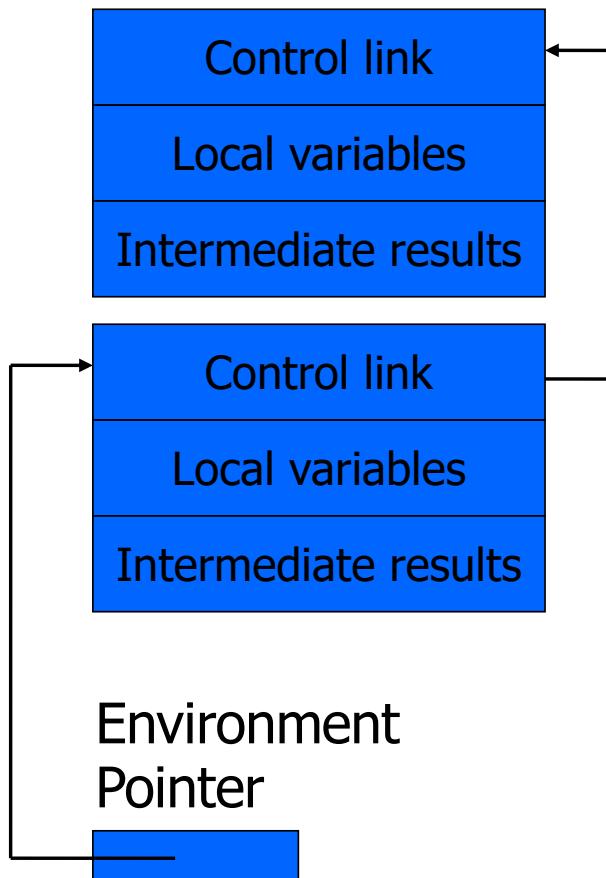
- Environment Pointer (EP) punta alla cima del record di attivazione corrente
- Record di attivazione ha dimensione variabile
- Come faccio a ripristinare EP quando faccio il pop del record di attivazione che non serve più?
- Uso il control link:
 - Puntatore alla cima del record di attivazione precedente
 - Viene salvato quando creo il record di attivazione
 - Viene ripristinato quando faccio il pop



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Activation record for in-line block



- Control link
 - pointer to previous record on stack
- Push record on stack:
 - Set new control link to point to old env ptr
 - Set env ptr to new record
- Pop record off stack
 - Follow control link of current record to reset environment pointer



Example

```
{ int x=0;  
    int y=x+1;  
    {  
        int z=(x+y)*(x-y);  
    }  
}
```

Push record with space for x, y (set control link = old env pointer, set env pointer)
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block (set env pointer to control link)
Pop record for outer block

Control link	
x	0
y	1

Control link	
z	-1
x+y	1
x-y	-1

Environment
Pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Scoping rules

◆ Global and local variables

- x, y are local to outer block
- z is local to inner block
- x, y are global to inner block

```
{ int x=0;  
  int y=x+1;  
  { int z=(x+y)*(x-y);  
  }  
}
```

◆ Static scope

- global refers to declaration in closest enclosing block

◆ Dynamic scope

- global refers to most recent activation record

These are same until we consider function calls.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizi

- Qualche esercizio sullo stack ...
- proviamo a stampare alcune info dello stack



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

03 - Memory Management

INGEGNERIA INFORMATICA

Programmazione Avanzata

Prof. Claudio MENGHI

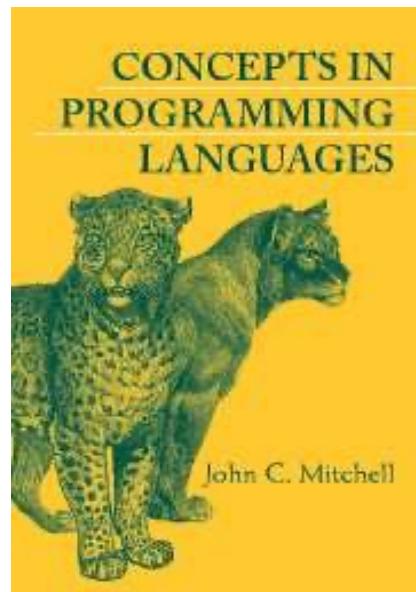
Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

Dalmine

27 Settembre 2023



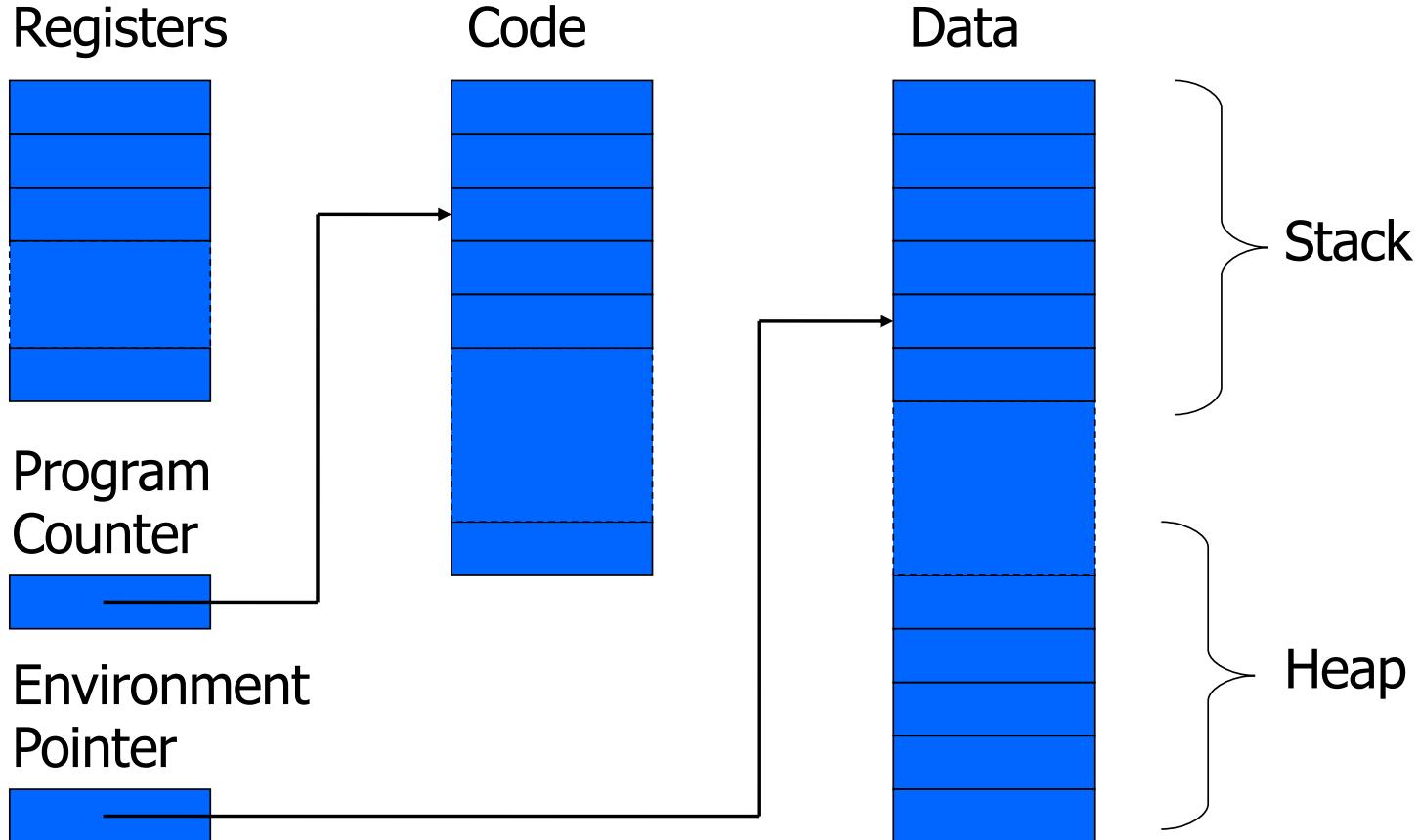
Capitolo 7 - Scope, Functions, and Storage Management



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Simplified Machine Model



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

EBP known as the frame pointer, (Base pointer)
ESP register is the stack pointer

Example

```
{ int x=0;  
    int y=x+1;  
    {  
        int z=(x+y)*(x-y);  
    }  
}
```

Push record with space for x, y (set control link = old env pointer, set env pointer)
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block (set env pointer to control link)
Pop record for outer block

Control link	
x	0
y	1

Control link	
z	-1
x+y	1
x-y	-1

Environment
Pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Per blocchi in-line di condizioni if e cicli for

- Del tutto simile come i blocchi inline
- Nel caso di cicli il RA viene messo solo una volta e usato per tutta la durata del ciclo (fino alla fine di tutti i cicli)
- Esempio
-

```
while(..){  
    int z;  
    z = ....  
}
```

Ra precedente

Control link

z

Risultati intermedi



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Se non ci sono var locali al blocco non c'è neanche RA

Promozione variabili globali

nota: non sempre il compilatore crea un record di attivazione di un blocco, la maggior parte delle volte nei compilatori moderni promuove la variabile a variabile globale al blocco.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Functions and procedures

- Syntax of procedures (Algol) and functions (C)

```
procedure P (<pars>)      <type> function f(<pars>)
begin                      {
  <local vars>          <local vars>
  <proc body>           <function body>
end;                      };
```

- Activation record must include space for

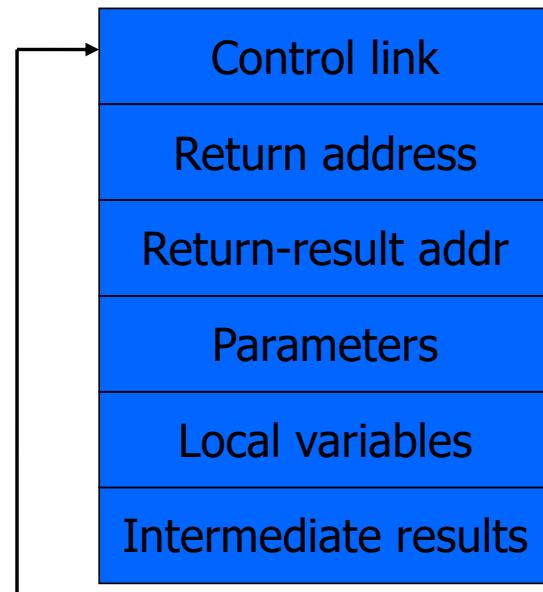
- parameters
- return address
- Local variables
(and intermediate result)
- location to put return value on function exit



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Activation record for function



- Return address
 - Location of code to execute on function return
- Return-result address
 - Address in activation record of calling block to receive return address
- Parameters
 - Locations to contain data from calling block

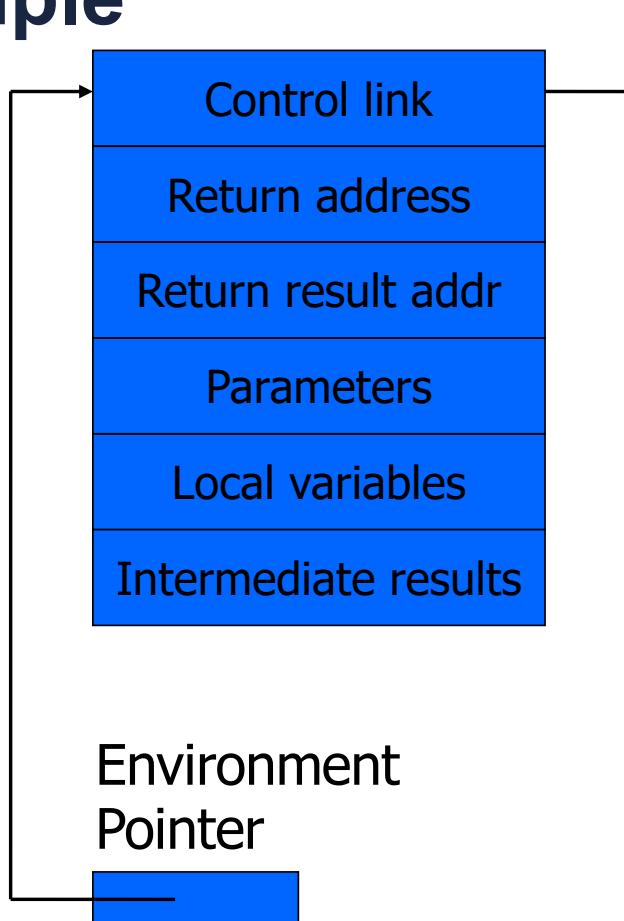
Environment
Pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example



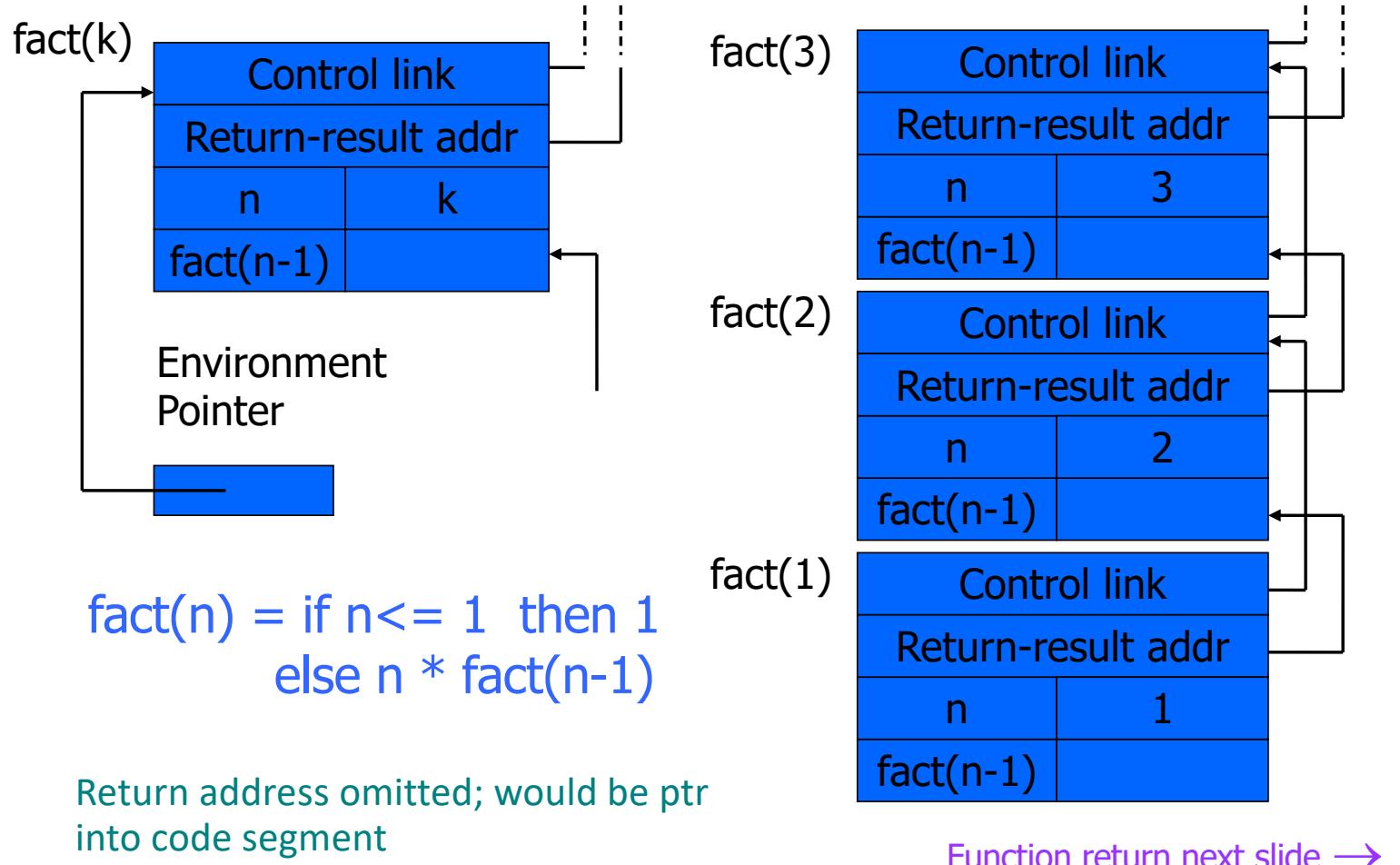
- Function
 $\text{fact}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n * \text{fact}(n-1) & \text{else} \end{cases}$
- Return result address
 - location to put $\text{fact}(n)$
- Parameter
 - set to value of n by calling sequence
- Intermediate result
 - locations to contain value of $\text{fact}(n-1)$



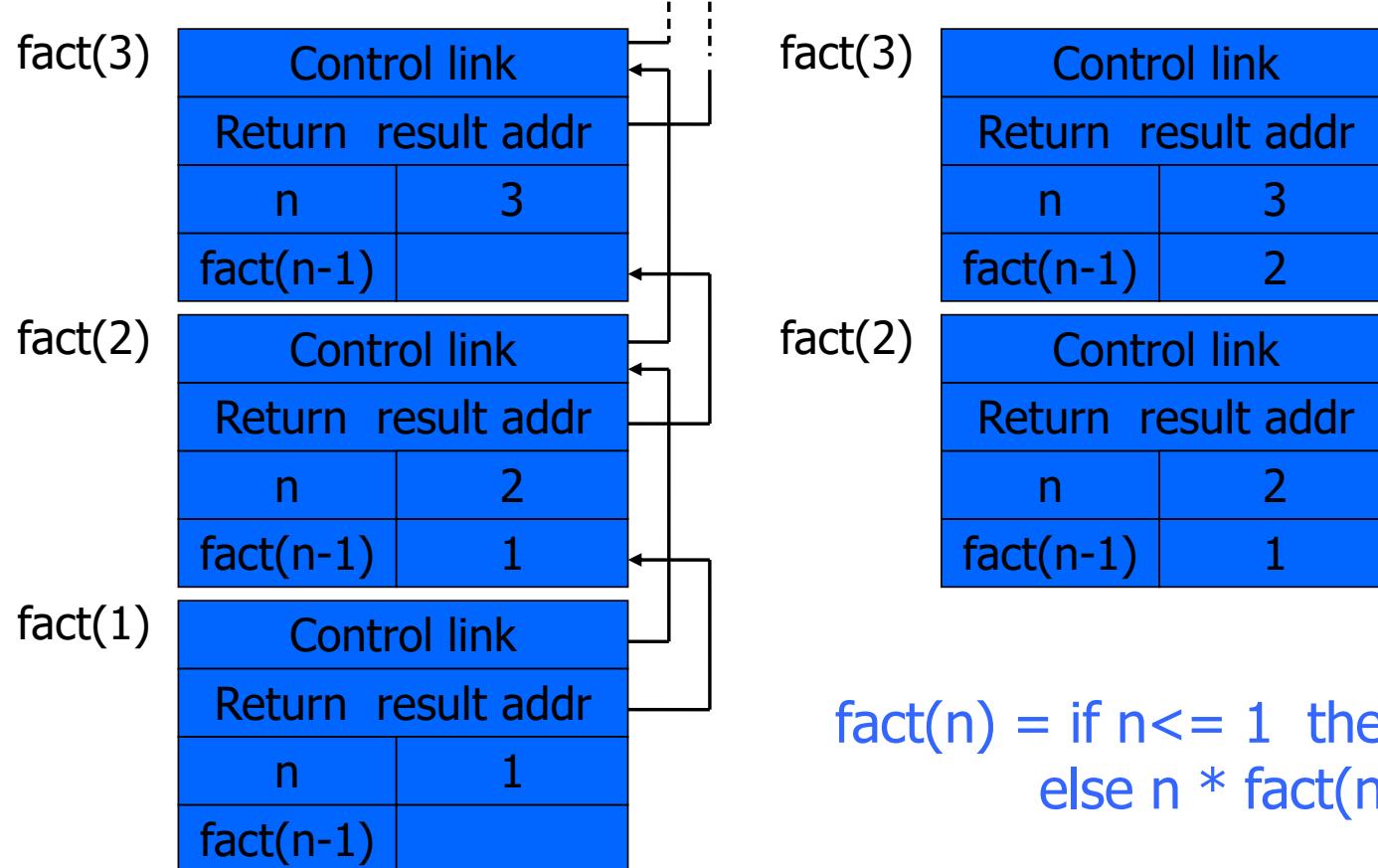
UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Function call

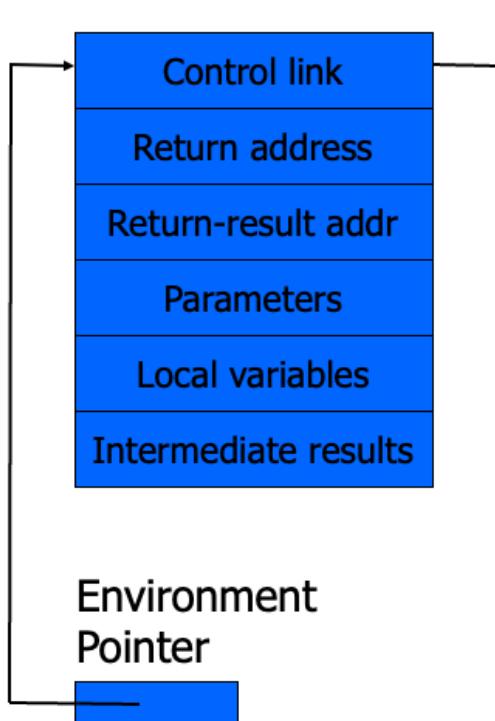


Function return

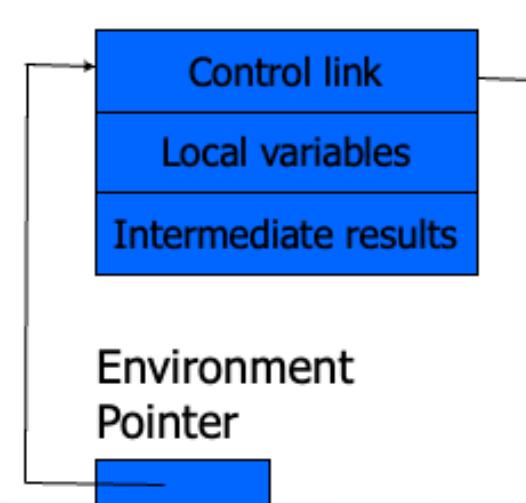


Activation record for function

- Function



- Inline block



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Topics for TODAY

- Parameter passing
 - use ML reference cells to describe pass-by-value, pass-by-reference
- Access to global variables
 - global variables are contained in an activation record higher “up” the stack
- Tail recursion
 - an optimization for certain recursive functions



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Parameter passing



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Formal parameters: il nome dei parametri utilizzati nella dichiarazione della funzione
- Actual parameters: le espressioni utilizzate per computare il valore dei parametri



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Formal parameters: x,y
- Actual parameters: z. 4^*z+1

```
proc p (int x, int y) {  
    if (x > y) then ... else ... ;  
    ...  
    x:= y*2 + 3;  
    ...  
}  
p (z, 4*z+1);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Differenze
 - Momento in cui i parametri sono valutati
 - Locazione utilizzata per memorizzare i parametri



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Differenze
 - Momento in cui i parametri sono valutati
 - Prima dell'esecuzione della funzione (caso tipico)
 - Tra questi abbiamo
 - Passaggio per reference
 - Passaggio per valore



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri: L-value and R-value

- Pensiamo alla differenza tra memory locations e il loro contenuto.
 - `int x;`
 - `int y;`
 - `x=y+3`

La locazione di una variabile è chiamata L-value

Il valore contenuto in una variabile è chiamato R-value



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Differenze
 - Momento in cui i parametri sono valutati
 - Prima dell'esecuzione della funzione (caso tipico)
 - Tra questi abbiamo
 - Passaggio per reference (passiamo il L-value)
 - Passaggio per valore (passiamo il R-value)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Passaggio per reference e per valore conseguenze
 - Side Effects (assegnamenti possono avere comportamenti diversi)
 - Aliasing quando due nomi si riferiscono allo stesso oggetto o locazione
 - Dangling pointers: do not resolve to a valid destination.
 - Efficienza: passaggio per valore può essere inefficienza se la struttura da essere copiata è molto grande



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di Parametri

- Pass-by-reference
 - Caller places L-value (address) of actual parameter in activation record
 - Function can assign to variable that is passed
 - In some language is also call by variable
 - PASCAL:
 - procedure Name (a,b : integer; VAR c,d: integer);
 - a and b are passed by value, c and d by reference
- Pass-by-value
 - Caller places R-value (contents) of actual parameter in activation record
 - Function cannot change value of caller's variable
 - Reduces aliasing (alias: two names refer to same loc)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

ML imperative features (review)

- General terminology: L-values and R-values
 - Assignment $y := x+3$
 - Identifier on left refers to **location**, called its L-value
 - Identifier on right refers to **contents**, called R-value
- ML reference cells and assignment (anche in C++)
 - Different types for location and contents
 - $x : \text{int}$ non-assignable integer value
 - $y : \text{int ref}$ location whose contents must be integer
 - $!y$ the contents
 - $\text{ref } x$ expression creating new cell initialized to x
 - ML form of assignment
 - $y := x+3$ place value of $x+3$ in location (cell) y
 - $y := !y + 3$ add 3 to contents of y and store in location y



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

(in C++)

- Anche in C++ esistono i riferimenti:

```
int y;  
int& x = y; (x is a reference to the variable y)
```

- Nota:
- References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined.
- Vedremo più avanti nel modulo C++



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example

pseudo-code

```
function f (x) =  
  { x := x+1; return x };  
var y : int = 0;  
print f(y)+y;
```

pass-by-ref

pass-by-value

Standard ML

```
fun f (x : int ref) =  
  ( x := !x+1; !x );  
y = ref 0 : int ref;  
f(y) + !y;
```

```
fun f (z : int) =  
  let x = ref z in  
    x := !x+1; !x  
  end;  
y = ref 0 : int ref;  
f(!y) + !y;
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example pseudo-code

```
function f (x) =  
    { x := x+1; return x };  
var y : int = 0;  
print f(y)+y;
```

pass-by-ref
→

pass-by-value
→

C++

```
int f (int & x) {  
    x = x+1;  
    return x;  
}  
int y = 0;  
cout<< f(y) + y;
```

```
int f (int x) {  
    x = x+1;  
    return x;  
}
```

```
int y = 0;  
cout<< f(y) + y;
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di puntatori

- Il passaggio di puntatori è un passaggio per valore, ma si usa (in C) per ottenere lo stesso effetto del passaggio per riferimento.
- Es.:

```
int f (int* x) {  
    *x = *x+1;  
    return *x;  
}
```

```
int y = 0;  
printf(f(&y) + y);
```

Se si vuole, si può evitare la modifica del parametro attuale mediante copia:

```
int f (int* x) {  
    int z = *x  
    return z+1;  
}
```

```
int y = 0;  
printf(f(&y) + y);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio degli array in C

- Come si passano gli array in C
- Si possono passare come array:
 - void foo(int arr[5])
 - ATTENZIONE: When an array is passed as a parameter, only the memory address of the array is passed (not all the values). An array as a parameter is declared similarly to an array as a variable, but no bounds are specified. The function doesn't know how much space is allocated for an array.
 - Ma arr è semplicemente un puntatore di interi, non c'è alcuna informazione sulla dimensione dell'array !!!
 - Attenzione quindi all'uso di **sizeof**
 - Vedi esempio !!!
- Soluzione:
 1. Passare anche la dimensione void foo(int arr[], int n)
 2. Mettere un terminatore (come le stringhe)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio stringhe

- Caso particolare le stringhe
 - Zero terminated (0 o '\0' NON '0')
 - Qualche problema (vedremo in futuro)
- Esercizio. Scrivi una funzione che prende una stringa in ingresso e restituisce quante 'a' ci sono
- Oppure palindroma (lab)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio struct

- Quando passo una struct passo l'intero record.
- Esempio:

```
struct student{  
    char firstname[30];  
    char surname[30];  
};
```

- void addStudent(struct student s) {}
- Tutta la struct è copiata sullo stack
- Lo stesso anche per il return value.
- Per questo si preferisce spesso usare il puntatore o riferimento
(occupo meno spazio sullo stack e non devo copiare tutti i dati)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passaggio di puntatori a puntatori

- Esercizio di passaggio di puntatore a puntatore
- Uso più frequente per modificare un puntatore.

PUNTATORE

```
int y = 10;
```

```
void styp(int* p) {  
    p = &y;  
}
```

```
int main(void) {  
    int m = 0;  
    int * q = &m;  
    styp(q);  
    printf("%d", *q);  
    return EXIT_SUCCESS;  
}  
>> 0  
- q punta ancora ad m
```

PUNTATORE a PUNTATORE

```
int y = 10;
```

```
void stypp(int** p) {  
    *p = &y;  
}
```

```
int main(void) {  
    int m = 0;  
    int * q = &m;  
    stypp(&q);  
    printf("%d", *q);  
    return EXIT_SUCCESS;  
}  
>> 10  
- q punta a y
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

- Una funzione che toglie il primo carattere da una stringa.
- Due alternative:
 - Usa stringhe
 - Usa puntatore a stringhe (puntatore a puntatore)
- Disegniamo lo stack



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Parameter passing & activation record

- pass by value: the value of the actual parameter is copied in the activation record as value of the formal parameter
 - Pass by pointer is a particular case
- pass by ref: the address of the actual parameter is copied in the activation record



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Osservazioni

- Il passaggio per riferimento ha alcuni vantaggi
 - Meno memoria (pensa ad un oggetto)
 - però alcuni svantaggi:
 - Indirezione ulteriore sullo stack
 - Side effect non desiderati – vedi esercizio sul libro
 - Vedi es 7.4
 - Come passare le costanti??
 - Ad esempio un numero
 - Solo L-values, non posso passare Rvalue
 - Posso inavvertitamente modificare il dato passato
 - Passaggio per nome: il nome del par. formale viene sostituito con il par. attuale
 - Vedi esercizio 5.2
- Esercizi 7.3, 7.5, 7.6. 7.7 7.8



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Passing parameters in Java

- Classically
 - Pass by value for primitive types
 - Pass “by reference” for reference types (Objects, arrays, ...)
 - NOT possible by value for Objects as in C++
- However
 - If you consider: Pass-by-reference
 - The formal parameter merely acts as an alias for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.
 - Java is not a real pass by reference but it is pass by value of the pointer



Swap in C++/Java

- Swap: cambio dei valori tra due variabili
 - Primitivi: semplice
 - Oggetti:???
- The Litmus Test
 - There's a simple "litmus test" for whether a language supports pass-by-reference semantics:
 - Can you write a traditional swap(a,b) method/function in the language?
- SWAP in C++
 - esercizio
- SWAP in JAVA????



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Access to global variables



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Access to global variables

- Two possible scoping conventions
 - Static scope: refer to closest enclosing block
 - Dynamic scope: most recent activation record on stack
- Example

```
int x=1;  
function g(z) = x+z;  
function f(y) = {  
    int x = y+1;  
    return g(y*x)  
};  
f(3);
```

outer block	x	1
-------------	---	---

f(3)	y	3
	x	4

g(12)	z	12
-------	---	----

Which x is used for expression $x+z$?



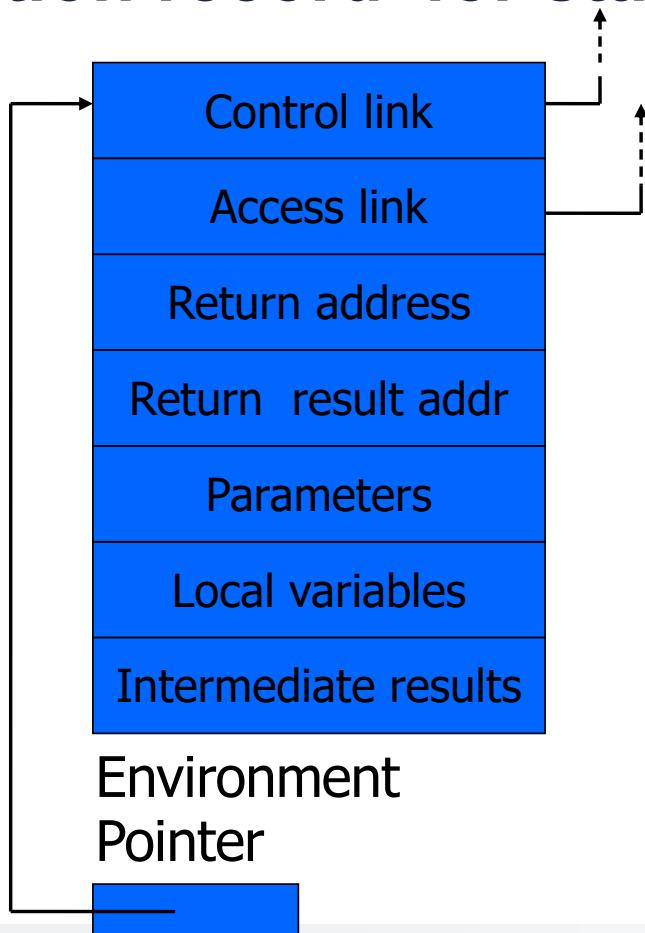
UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Static: prende $x = 1$ guardando il codice

Dinamico: prende il primo x sullo stack, $x = 4$

Dipartimento
di Ingegneria Gestionale
dell'Impresa e della Produzione

Activation record for static scope



- Control (dynamic) link
 - Link to activation record of previous (calling) block
- Access (static) link
 - Link to activation record of closest enclosing block in program text
- Difference
 - Control link depends on dynamic behavior of prog
 - Access link depends on static form of program text



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Access link

La maggior parte dei linguaggi usa il primo (statico)

Per tener traccia si usa lo static link mediante l'access link

Nei linguaggi che usiamo noi (tipo C) in cui ci sono solo due livelli (globale e locale) gli access link non sarebbero necessari – li segniamo per rendere chiaro dove è il record di attivazione globale

In C quindi l'access link punta sempre al RA delle variabili globali

VAR. GLOBALI

AL

AL

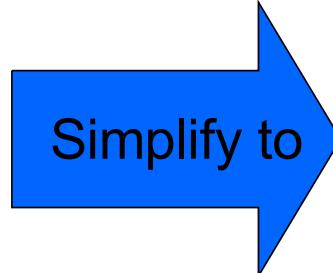


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Complex nesting structure

```
function m(...) {  
    int x=1;  
  
    ...  
    function n( ... ){  
        function g(z) = x+z;  
        ...  
    } ...  
    function f(y) {  
        int x = y+1;  
        return g(y*x) ;  
    } ...  
    f(3); ... }  
    ... n( ... ) ...}  
... m(...)
```



```
int x=1;  
function g(z) = x+z;  
function f(y) =  
{ int x = y+1;  
return g(y*x) };  
f(3);
```

Simplified code has same block nesting,
if we follow convention that each
declaration begins a new block.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

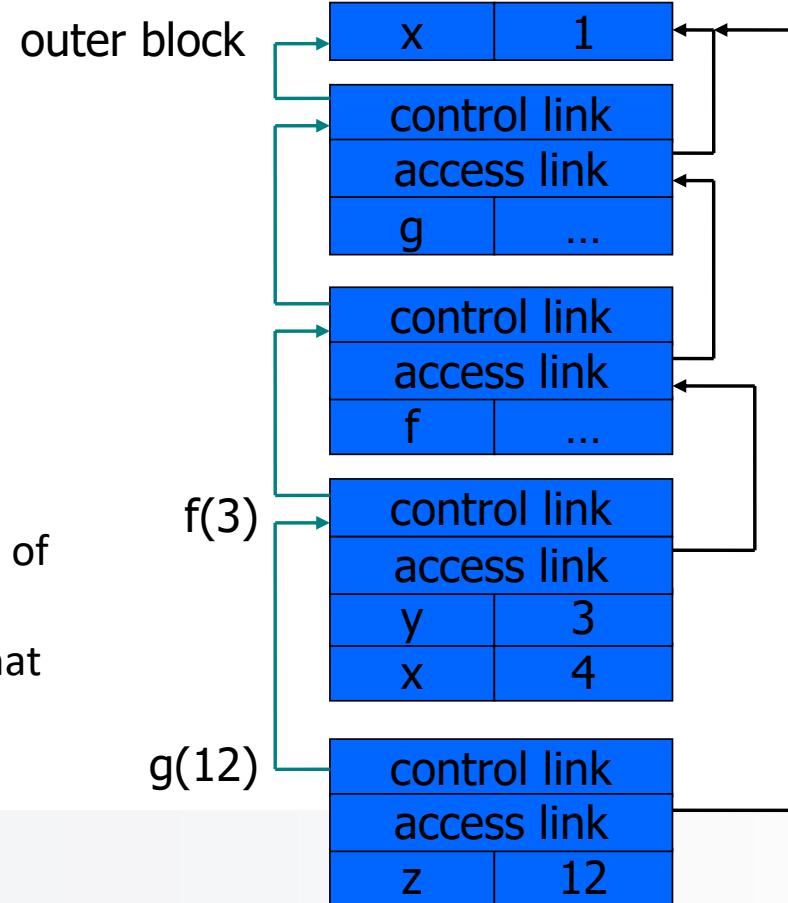
Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Static scope with access links

```
int x=1;
function g(z) = x+z;
function f(y) =
{ int x = y+1;
  return g(y*x) };
f(3);
```

Use access link to find global variable:

- Access link is always set to frame of closest enclosing lexical block
- For function body, this is block that contains function declaration



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Tail recursion



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Ricorsione

- Una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento (chiamata) a se stessa.
- Esempio: Funzione fattoriale su interi non negativi:
 $fatt(n) = n!$
- definita ricorsivamente come segue:
 - 1 se $n=0$
 - $fatt(n)= n*fatt(n-1)$ se $n>0$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempi di problemi ricorsivi:

1) Somma dei primi n numeri naturali:

- $\text{somma}(n)= 0$ se $n=0$
- $n+\text{somma}(n-1)$ altrimenti

2) Ricerca di un elemento el in una sequenza di interi:

- falso se sequenza terminata, altrimenti
- $\text{ricerca}(\text{el}, \text{sequenza})=\text{vero}$ se $\text{el}=\text{primo}(\text{sequenza})$, altrimenti
- $\text{ricerca}(\text{el}, \text{resto}(\text{sequenza}))=$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Programmi ricorsivi

- Molti linguaggi di programmazione offrono la possibilità di definire funzioni/procedure ricorsive.
- Calcolo del fattoriale di un numero:

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



Esempi (2)

- Alcune volte è necessario “complicare” la segnatura del metodo per renderelo ricorsivo:
- Ricerca di un elemento in un array (Java)

```
// cerca x in array a a partire dalla posizione pos  
boolean search(int x, int[] a, int pos)
```

```
boolean search(int x, int[] a, int pos) {  
    if (pos >= a.length)  
        return false;  
    if (a[pos] == x)  
        return true;  
    // non trovato nella posizione  
    // pos vai alla prossima  
    return search(x, a, pos + 1);  
}  
boolean search(int x, int[] a) {  
    return search(x, a, 0) {
```



Esempi (2 in C)

- In C spesso si passa anche la dimensione dell'array
- Ricerca di un elemento in un array (C)
- Array passato come puntatore

```
// cerca x in array a con lunghezza n  
bool search(int x, int* a, int n)
```

```
#include <stdbool.h>  
bool search(int x, int* a, int n) {  
    if (n == 0) return 0;  
    if (a[0] == x) return 1;  
    // non trovato nella posizione a[0]  
    //vai alla prossima  
    return search(x, a + 1, n - 1);  
}
```



Tail recursion (first-order case)

- Function g makes a *tail call* to function f if
 - Return value of function f is return value of g
- Example **tail call** **not a tail call**

```
fun g(x) = if x>0 then return f(x) else return f(x)*2
```
- Optimization
 - Can pop activation record on a tail call
 - Non al ritorno come teoricamente dovrebbe fare
 - Especially useful for recursive tail call
 - next activation record has exactly same form



Example Calculate least power of 2 greater than y

control	
return val	
x	1
y	3

```
fun f(x,y) = if x>y  
    then ret x  
    else ret f(2*x, y);
```

Chiamata:
 $f(1,3)$

control	
return val	
x	1
y	3

control	
return val	
x	2
y	3

control	
return val	
x	4
y	3

Optimization 1

- Set return value address to that of caller

Question

- Can we do the same with control link?

Optimization

- avoid return to caller



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Tail recursion elimination

f(1,3)

control	
return val	
x	1
y	3

f(2,3)

control	
return val	
x	2
y	3

f(4,3)

control	
return val	
x	4
y	3

```
fun f(x,y) = if x>y  
    then x  
    else f(2*x, y);  
f(1,3);
```

Optimization

- pop followed by push = reuse activation record in place

Conclusion

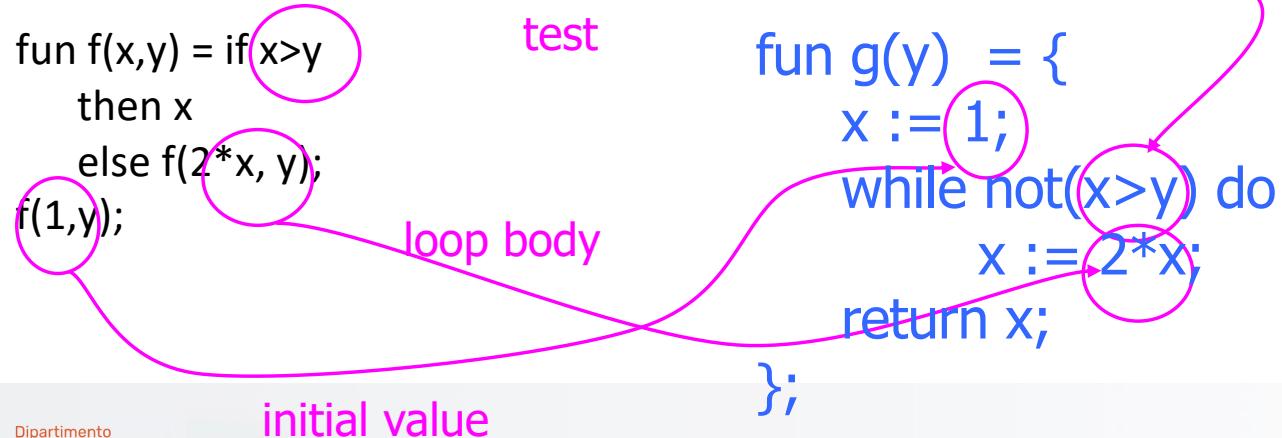
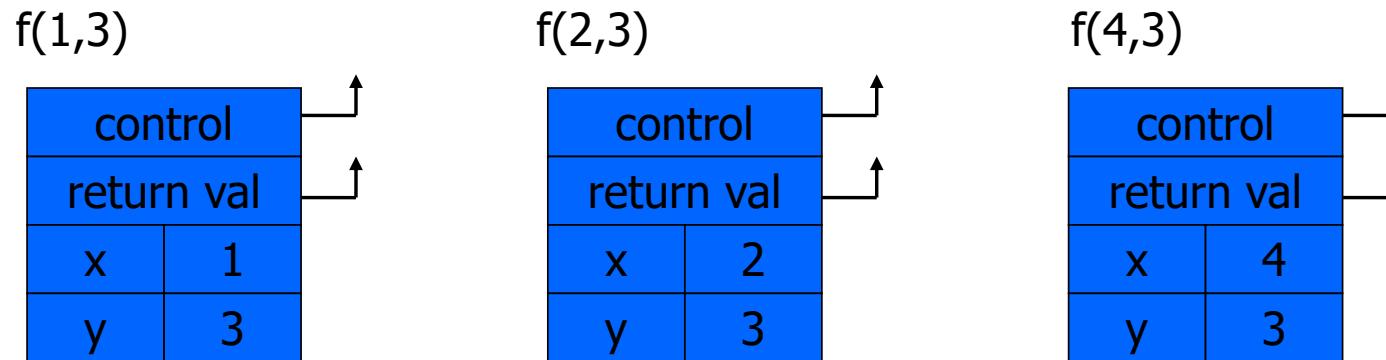
- Tail recursive function equiv to iterative loop



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Tail recursion and iteration



Higher-Order Functions

◆ Language features

- Functions passed as arguments
- Functions that return functions from nested blocks
- Need to maintain environment of function

◆ Simpler case

- Function passed as argument
- Need pointer to activation record “higher up” in stack

◆ More complicated second case

- Function returned as result of function call
- Need to keep activation record of returning function



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

05 – Type Systems and Type Inference

Programmazione Avanzata

Anno di corso: 1

Anno accademico di offerta: 2023/2024

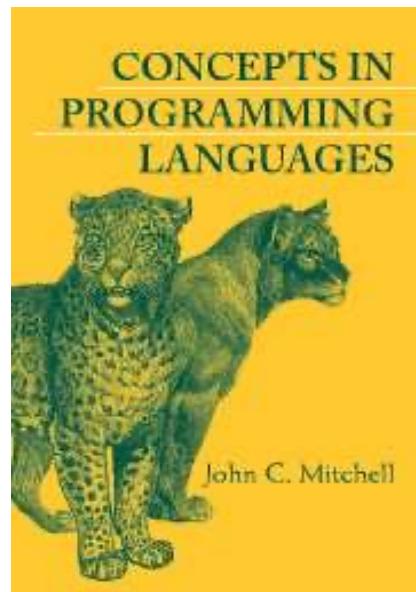
Crediti: 6

INGEGNERIA INFORMATICA

Prof. Claudio MENGHI

Dalmine

10 Ottobre 2023



Capitolo 6 - Type Systems and Type Inference



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

La sicurezza dei tipi nei linguaggi di programmazione

La sicurezza dei tipi in un programma è molto importante

- se l'esecutore (il PC o la macchina virtuale) non riesce a distinguere i tipi di un certo programma può facilmente causare errori
- molti attacchi sfruttano proprio debolezze nel controllo dei tipi di linguaggi diffusi come il C

"Well-typed programs never go wrong."

Robert Milner



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Tipo

Tipo: Insieme di valori omogenei + operazioni che si possono fare

Esempi:

- tipi semplici: Integers, String,
- tipi strutturati come classi, ...
- funzioni: int -> bool
 - Funzione che da un intero mi dà un boolean
 - Anche le funzioni e i metodi definisco un tipo

Esempi di non tipi:

- numeri dispari
- array contenenti String e Integer

Dipende però dal linguaggio di programmazione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

A cosa servono i tipi

Per organizzare e dare un nome ai concetti (documentazione)

- Spesso corrispondenti ai concetti nel dominio del problema che si vuole risolvere
- Indicare l'uso che si vorrà fare di certi identificatori (così il compilatore può controllare)

Per assicurarsi che sequenze di bit in memoria siano interpretate correttamente

- Per evitare errori come: 3 + true + "Angelo"

Per istruire il compilatore come rappresentare i dati

- Esempio short richiedono meno bit di int



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Errori di tipi a livello Hardware

Confondere dati con programmi

- Caricando quindi nei registri della CPU possibilmente codici non corretti

Esempio: cerco di eseguire un dato chiamando **x()** dove **x** non è una procedura ma un intero

Confondere tipi di dati semplici

Esempio: eseguo **float_add(3,4.5)** con 3 int
float_add: operazione della CPU che chiama una routine della FPU, se la CPU prende 3 come sequenza di bit float, potrebbe generare un errore hardware



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Errori semantici

Il programma fa qualcosa che non è quello che dovrebbe fare

- Esempio con tipi primitivi: `int_add(3, 4.5)`

In questo caso la sequenza di bit che rappresenta 4.5 può essere interpretato come int ma non sarà uguale come valore

- Esempio con oggetti ed ereditarietà in Java

Sia Quadrato sottoclasse di Figura:

```
class Quadrato extends Figura
```

Se non riesco a distinguere istanze di Qu. e Fig.:

`Figura a1 = new Quadrato()` **OK**

`Quadrato b1 = new Figura()` **NO**: Quadrato potrebbe avere dei metodi in più che potrei invocare ma non trovare perché b1 è una Figura



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Type safety: sicurezza dei tipi

Un linguaggio di programmazione L si dice **type safe** se non esiste programma scritto in L che possa violare la distinzione di tipi in L

Esempi di violazioni dei tipi:

- confondere interi e float
- chiamare una funzione attraverso un intero
- accedere ad una zona di memoria sbagliata (**non memory safe**)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sicurezza di alcuni linguaggi

Ecco una tabella che riporta la sicurezza di alcuni linguaggi di programmazione molto diffusi

Safety	Linguaggio	Motivo
Non safe	C e C++	Type cast, aritmetica dei puntatori
Quasi safe	Pascal	Deallocazione esplicita e dangling pointers
Safe	Java, Lisp,Python	Controllo completo dei tipi



Problemi del C/C++

Il C/C++ ha un sistema dei tipi non sicuro (posso facilmente violare la distinzione di tipi)

Alcuni tipi errori

- Type cast
- Dereferenziazione del null, ...
- Pointer arithmetic
- Accesso alla memoria non valida
 - **Violazione spaziale** come out of bound
 - **Violazione temporale** come dangling pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Quando si fa il type checking?

Tra i linguaggi **type safe** distinguiamo due categorie a seconda del **momento** in cui avviene il controllo dei tipi

run-time type checking

- Il controllo avviene durante l'esecuzione
- Esempio LISP: quando esegue l'istruzione `(car x)` - che applica `car` a `x` e `car` restituisce il primo elemento di una lista - controlla prima che `x` sia una lista

compile time type checking

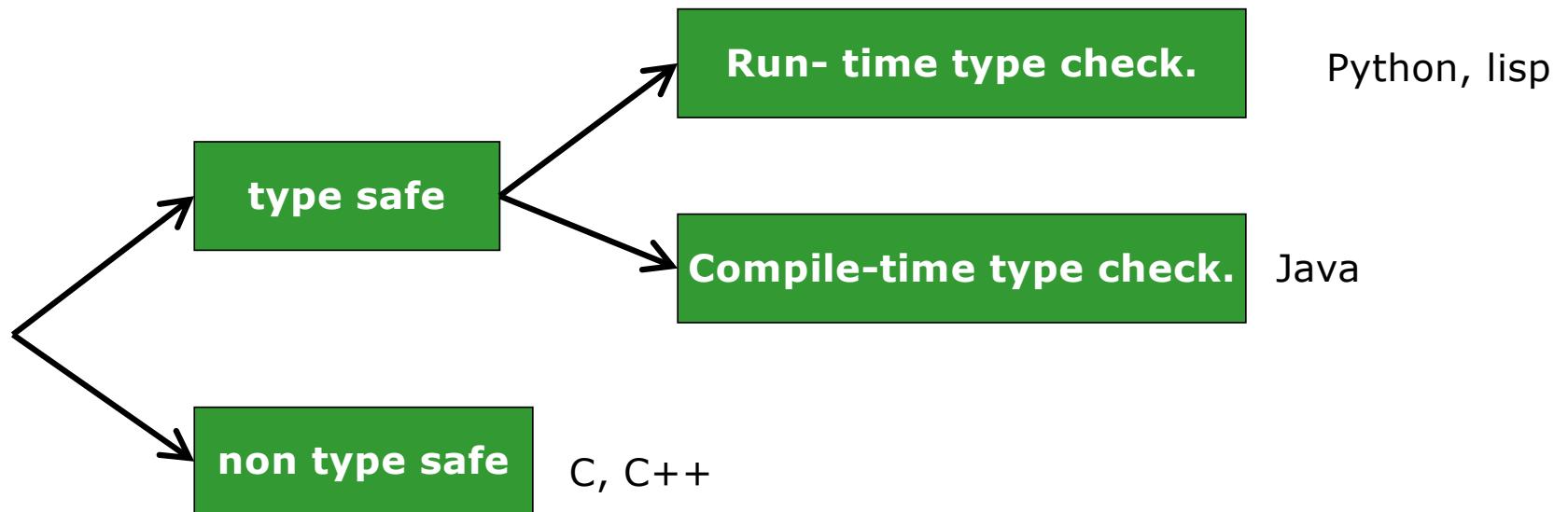
- Il controllo avviene durante la compilazione
- Esempio ML: se compila `f(x)` controlla che se `f` sia $A \rightarrow B$ e `x : A`



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Classificazione dei linguaggi



Vedi syllabus per approfondimento



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Java

Java usa **compile time**, però dove il compilatore non è sicuro della sicurezza dei tipi, introduce un controllo run-time (**conversioni dei tipi controllate**)

considera la seguente istruzione

```
Quadrato a = (Quadrato) b
```

- con b dichiarato di classe Figura (padre di Quad.)
- la conversione al sottotipo Quadrato è corretta solo se b è effettivamente una **istanza di** Quadrato (o di una sottoclassificazione)
- tale controllo non si può fare in compilazione
- il compilatore introduce un controllo da fare durante l'esecuzione che b sia convertibile a Quadrato



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- run-time checking rallenta l'esecuzione
 - **controlla le conversioni di tipo ogni volta**
- compile-time checking limita la flessibilità dei programmi
 - **tutte le istruzioni anche non eseguite devono essere corrette**
 - **Il controllo è conservativo**

alcuni programmi che non sono corretti compile time sono invece run time corretti



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- run-time checking rallenta l'esecuzione
 - **controlla le conversioni di tipo ogni volta**
- compile-time checking limita la flessibilità dei programmi
 - **tutte le istruzioni anche non eseguite devono essere corrette**
 - **Il controllo è conservativo**

alcuni programmi che non sono corretti compile time sono invece run time corretti
[Perché 1bpt]



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Dynamic Type checking in Python

In python ogni variabile ha un tipo e viene controllata (type safe) ma:

- Non è necessario dichiarare il tipo di una variabile

```
x = 5  
print(type(x))
```

- Posso comunque specificare il tipo (con alcune conversioni esplicite)

```
x = float(20)
```

```
x = bool(5)
```

```
z = float("3")
```

- Controlla comunque che un'operazione sia type safe:

```
y = x / "pippo" -> errore non esiste l'operazione (in C?)
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

problemi

I tipi si possono ridefinire

```
-x = int(5)
```

```
-print(type(x))
```

```
-x = "pippo"
```

```
-print(type(x))
```

Problemi

-Se il mio codice contiene un errore di tipo e non lo eseguo non me ne accorgo

```
x = int(5)
```

```
if x < 5:
```

```
    y = x / "pippo"
```

```
print(x)
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Per le funzioni anche peggio

```
# define a sum function (intended to be used for integers)
def sum(x,y):
    return x+y

print( sum(8,'hello'))
```

Puoi usare mypy:

```
def sum(x:int,y:int) -> int:
    return x+y

print( sum(8,4))
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Static typing e annotazioni

In alcuni linguaggi (es Rust, Xpand) non è necessario indicare il tipo di una variabile ma poi non si può cambiare

```
let mut sum = 5 + 10;  
println!("{}", sum);  
sum = "pippo" → ERRORE
```

Anche in java dalla 10 in poi **local variable type inference**,

```
var id=0;// At this moment, compiler interprets  
//variable id as integer.  
id="34"; // This will result in compilation error
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Flessibilità del run time chkng

In Lisp/python, possiamo scrivere

```
(cond ((< x 10) x) (else (car x))) OK
```

alcune volte ci sara' errore (catturato dal lisp stesso)

altre no - se x non è < 10 valuto car che si aspetta una lista

In Java, **non** posso scrivere

```
int x;  
if (0>-1) { x++; } else { x = "ciao"; } NO
```

perchè assegna ad x int una String

eppure questo programma è type safe, perchè nessuna esecuzione causa errori di tipo (0 è sempre > -1)



Type Inference

E' il processo di capire il tipo dei dati basandosi sul tipo delle espressioni nei quali appaiono

- type checking: il compilatore controlla che i tipi dichiarati dal programmatore sono in linea con le espressioni
- type inference: il tipo non e' specificato e un'inferenza logica e' richiesta per capire il tipo degli identificatory da usare



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Type Inference: Esempio 1

Esempio

```
- fun f1(x) = x+2;  
val f1 = fn : int → int
```

+ potrebbe essere applicato a vari tipi di parametri, tuttavia

Considerato che 2 è un intero, x deve essere a sua volta un intero

- ne segue che f1 va da interi ad interi



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Type Inference: Esempio 2

Esempio

```
- fun f2(g,h) = g(h(0));  
val f2 = fn : ('a → 'b) * (int → 'a) → 'b
```

f2 è parsato come $((\text{`a} \rightarrow \text{'b})^*(\text{int} \rightarrow \text{'a})) \rightarrow \text{'b}$

- considerate che h e' applicato a un intero, h va da interi ad un altro tipo (chiamiamolo 'a) h: int->a'
- g prende un 'a e lo trasforma in qualche cosa d'altro (chiamiamolo 'b) g: a->'b
- ne segue che f2 va da 'a ad 'b ('a->'b)
- f2 prende le due funzioni come argomento $(\text{a}' \rightarrow \text{'b})^*(\text{int} \rightarrow \text{'a}') \rightarrow \text{'b}'$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Type Inference Algorithm

- Step 1: Assegna un tipo ad ogni espressione e sotto espressione. Utilizza il tipo conosciuto per ogni variabile conosciuta (per esempio, a 3 assegna il tipo int)
- Step 2: Genera un insieme di vincoli sui tipi utilizzando l'albero sintattico dell'espressione. Per esempio, se una funzione e' applicata ad un argomento, il tipo dell'argomento deve corrispondere al tipo del dominio della funzione
- Step 3: Risovi questi vincoli per mezzo dell'unificazione (un metodo basato sulle sostituzioni per risolvere i sistemi di equazioni)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In sintesi

- Abbiamo visto:
 - l'importanza della sicurezza dei tipi
 - la definizione di linguaggio sicuro nei tipi
 - alcuni linguaggi sono safe altri no
- Ricordate che:
 - il C non è type safe – vedremo alcuni errori tipici
- Inoltre,
 - i linguaggi safe possono effettuare il controllo dei tipi o durante l'esecuzione (run-time) come il LISP o durante la compilazione (compile-time) come Java
 - i pro e contro dei due approcci sono: flessibilità (maggiore con runtime) e efficienza (maggiore con compile time)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

“C is not Safe”

Alcune caratteristiche del linguaggio C e C++ che **possono** dare errori:

1. dereferenziazione del null
2. type cast non controllato
3. pointer arithmetic
4. accesso alla memoria non valida
 - violazione **spaziale** come out of bound
 - violazione **temporale** come dangling pointers

Queste caratteristiche rendono il C molto **flessibile** e **veloce** a discapito della sua sicurezza

- è responsabilità del programmatore stare attento a non introdurre difetti



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Type Cast non Safe

Il C permette la conversione **non controllata** da un tipo ad un altro:

- da un tipo ad un sottotipo con possibile perdita di informazioni.
 - Esempio da double a int
- da intero ad una funzione per cercare di eseguire una certa locazione di memoria che potrebbe non essere un'istruzione corretta o fare qualcosa di non voluto

Programma corretto in C ma con type cast non safe:

```
double d;  
int i;  
...  
i = d; ➔ possibile perdita di informazioni
```

Dereferenziazione di null

- La dereferenziazione di un puntatore in C non viene controllata
- Se accedo ad una cella puntata da un puntatore nullo ho “segmentation fault”, cioè un errore del sistema operativo

Programma con accesso tramite puntatore null

```
int main() {
    int * ptr; ...
    ptr = NULL;
    *ptr = 2;
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Pointer arithmetic

Mediante l'aritmetica dei puntatori possiamo puntare a zone di memoria con tipi diversi

Esempio:

- se il puntatore p è definito di tipo A^*
- l'espressione $*(p+i)$ ha tipo A
- poiché il valore memorizzato a $p+i$ potrebbe avere qualsiasi tipo
- l'assegnamento $x = *(p+i)$ con x di tipo A , permette di memorizzare un valore di qualsiasi tipo in x



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

C non è memory safe

Inoltre mediante i puntatori si può facilmente accedere a memoria in modo scorretto

```
void f(int* p, int i, int v) {  
    p[i] = v;  
}
```

Accedo in questo modo all'indirizzo $p+i$ e $p+i$ potrebbe contenere dati importanti o altro codice

- Posso modificare il return address di una chiamata di una procedura ed eseguire altro codice, posso modificare dei diritti o leggere informazioni riservate
- Tipico “**buffer overflow**” /buffer overrun



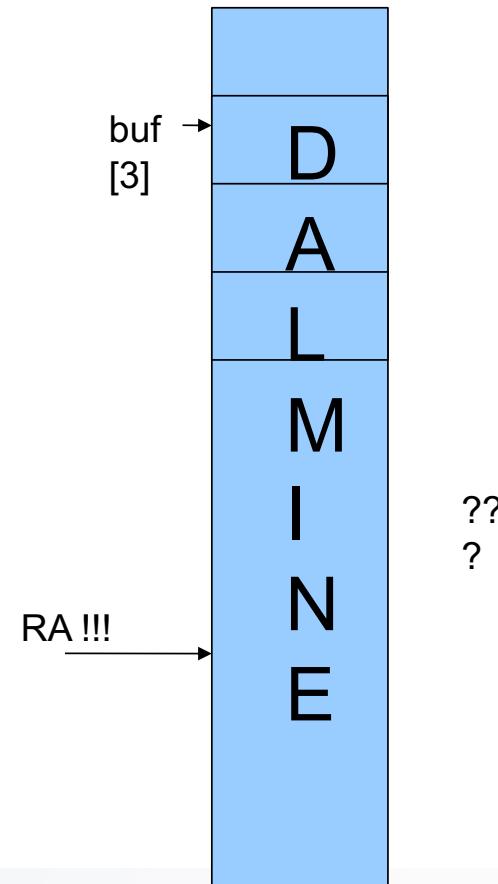
UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Buffer overrun

- A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer.
- For example copying the user input directly in the buffer using a `strcpy`,
- Variables declared on the stack are located next to the return address for the function's caller.
 - the result is that the return address for the function gets overwritten by an address chosen by the attacker.

```
strcpy(buf,"DALMINE");
```



<https://secgroup.dais.unive.it/teaching/security-course/overwriting-return-address/>



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio semplice

```
#include <string.h>

void foo (char *bar) {
    char c[12];
    strcpy(c, bar); // no bounds checking...
}

int main (int argc, char **argv) {
    foo(argv[1]);
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

type cast e violazione memoria

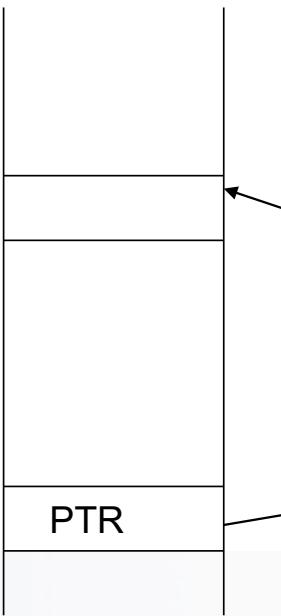
I puntatori in C sono assimilati a interi

Tramite cast di dati interi a puntatori, posso accedere ad una zona di memoria a piacere

*Programma (OK in compilazione)
con conversione da int a char**

```
int main() {
    char * PTR;
    PTR = 1000;
    *PTR = 'a';
}
```

1000



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Deallocazione esplicita e Dangling Pointers

In Pascal, C, ... una locazione puntata da un puntatore p può essere deallocata (liberata) dal programmatore: p è un “dangling pointer”

Ad esempio in C, faccio il **free** di un puntatore poi continuo ad usarlo

*Un puntatore è **dangling** se punta ad una zona di memoria che è stata liberata per essere riutilizzata*

- Il sistema operativo potrebbe allocare la stessa memoria nuovamente per memorizzare un altro tipo di valore
- Posso continuare ad usare p per accedere a questa memoria e rompere la type safety



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Uso di free

- Posso usare anche un puntatore dopo averne fatto il free
- Vedi esempio



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Memory leak

- In informatica, un **memory leak** ("perdita o fuoriuscita di memoria") è un particolare tipo di consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa, di variabili/dati non più utilizzati da parte dei processi. [wikipedia]



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio

Funzione che converte un intero in stringa corrispondente, restituendo il puntatore alla stringa ottenuta:

```
char * itoa(int i) {
    char buf[20];
    sprintf(buf,"%d",i);
    return buf;
}
```

A cosa punta buf ? buf viene restituito ma punta ad un array locale che viene deallocated
Ricorda sprintf



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Dangling Pointers sullo stack

Un esempio frequente di errore dovuto a dangling pointers è quando si usano puntatori a celle dello **stack**

Si verifica quando:

- si crea un puntatore p ad una zona A di memoria che è locale ad un metodo (ad esempio variabili locali)
- A è quindi allocata sullo stack
- A viene liberata all'uscita del metodo
- p è a questo punto un dangling pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio in C++

Esempio in C++:

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
    struct Point result = {x,y};
    return &result;
}
void bar() {
    struct Point *p = newPoint(1,2);
    p -> y = 1234;
}
```

newPoint restituisce un puntatore ad un oggetto (*result*) locale: in *bar* *p* è un dangling pointer



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Soluzione

Come si possono evitare dangling pointers?

1. Evitare di puntare zone di memoria sullo stack ed usare la **malloc**:

```
struct Point * result =  
    (struct Point*) malloc(sizeof(struct Point))
```

La malloc crea puntatori a zone sicure

Però attenzione che la sua gestione non è automatica come le variabili sullo stack

2. Uso del **garbage collector (gc)** invece che della deallocazione esplicita

- Il gcc marca lui le zone da liberare e che si possono riutilizzare
- Non usando free, il gc recura la memoria



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Cosa fare per avere evitare tali errori?

Se vogliamo scrivere codice safe cosa possiamo fare?

- Scrivere attentamente, progettare prima, documentare, etc.

Se vogliamo essere sicuri che il nostro codice è safe?

- Due soluzioni possibili
 - usare linguaggi type safe (Java, lisp;..) e linguaggi + astratti
 - usare linguaggi come C e dei tools che ci aiutano a rendere i programmi C safe



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In sintesi

- Abbiamo visto alcune fonti di violazioni di sicurezza del C:
 - dereferenziazione non controllata
 - typecast non controllato
 - aritmetica dei puntatori
 - Violazione “spaziale” della memoria, Buffer overflow, ...
 - deallocazione esplicita e dangling pointers
 - Violazione “temporale” della memoria, puntatori allo stack
- Le soluzioni proposte sono:
 - non usare C e passare a Java/C#, ...
 - usare C con tool e librerie che vedremo la prossima lezione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione





UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

07 – Programmi sicuri in C

INGEGNERIA INFORMATICA

Programmazione Avanzata

Prof. Claudio MENGHI

Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

Dalmine

17 Ottobre 2023

Svantaggi ad usare linguaggi astratti

C'è un prezzo da pagare se si vogliono usare linguaggi safe come Java, ...

- Prestazioni inferiori
 - per il controllo dei limiti nell'accesso agli array, garbage collection per evitare dangling pointers
- Impiego di maggiore memoria
 - per tenere informazione sui tipi, sulla dimensione degli array
- Annotazione dei tipi
 - maggiore verbosità nelle dichiarazioni
- Porting di codice già esistente in C
 - (per quanto Java abbia sintassi simile al C)



Vantaggi del C

Il C è tutt'oggi usato per molte applicazioni come il sistema operativo, i device drivers

- Ha prestazioni elevate
- Permette la gestione esplicita della memoria
- Permette il controllo della rappresentazione dei dati a basso livello
- Riuso del codice esistente già scritto in C



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Alcune violazioni di sicurezza del C

Errori “spaziali” di accesso alla memoria

- Out of bound access, buffer overflow, ...

Errori “temporali” di accesso alla memoria

- Dangling pointers,

Errori di cast

- Tra diversi tipi di puntatori, tra puntatori e dati interi, tipi unione

Memory leaks

- Programmi che non rilasciano la memoria anche quando non serve più



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come rendere il C safe?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come rendere il C safe?

- Restringere il linguaggio (non permettere all'utente di scrivere codice unsafe)
 - Vantaggi: codice safe per definizione
 - Svantaggi: limitiamo l'espressività



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come rendere il C safe?

- Restringere il linguaggio (non permettere all'utente di scrivere codice unsafe)
 - Vantaggi: codice safe per definizione
 - Svantaggi: limitiamo l'espressività
- Controlliamo mediante opportune procedure se abbiamo scritto codice unsafe
 - Vantaggi: non limitiamo l'espressività
 - Svantaggi: perdiamo delle violazioni (a meno che risolviamo l'halting problem)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come rendere il C safe?

- Restringere il linguaggio (non permettere all'utente di scrivere codice unsafe)
 - Vantaggi: codice safe per definizione
 - Svantaggi: limitiamo l'espressività
- Controlliamo mediante opportune procedure se abbiamo scritto codice unsafe
 - Vantaggi: non limitiamo l'espressività
 - Svantaggi: perdiamo delle violazioni (a meno che risolviamo l'halting problem)
 - Può essere fatto in:
 - Modo statico: compile time
 - Modo dinamico: durante l'esecuzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come rendere il C safe?

1. Tools per l'analisi statica e dinamica per trovare safety violations
 - esempio **Purify** della Rational/IBM è un tool per l'analisi **dinamica** per scoprire errori di accesso alla memoria, valgrind
 - Analisi statica: cppcheck, lint ...
 2. Librerie per rendere il programmi C safe
 3. Uso di un sottoinsieme «safe»
 4. Tools, e linguaggi per prevenire safety violation con due approcci distinti
 1. rendere sicuri programmi C : SafeC, CCured
 2. varianti safe del C: Cyclone, Vault



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Analisi statica con splint

- <http://www.splint.org/>
- Splint, short for Secure Programming Lint, is a programming tool for statically checking C programs for security vulnerabilities and coding mistakes. Formerly called LCLint, it is a modern version of the Unix lint tool.
- Splint has the ability to interpret special annotations to the source code, which gives it stronger checking than is possible just by looking at the source alone.
- Ultima versione 2007



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio di splint

```
#include <stdio.h>
int main()
{
    char c;
    while (c != 'x');
    {
        c = getchar();
        if (c = 'x')
            return 0;
        switch (c) {
        case '\n':
        case '\r':
            printf("Newline\n");
        default:
            printf("%c", c);
        }
    }
    return 0;
}
```

- Variable `c` used before definition
- Suspected infinite loop. No value used in loop test (`c`) is modified by test or loop body.
- Assignment of int to char: `c = getchar()`
- Test expression for if is assignment expression: `c = 'x'`
- Test expression for if not boolean, type char: `c = 'x'`
- Fall through case (no preceding break)



Analisi dinamica con Purify

approccio analisi dinamica

input programmi C/C++ (qualsiasi)

output eseguibili linkati con Purify

metodo inserimento di controlli per trovare durante l'esecuzione errori
di accesso alla memoria o memoria non rilasciata

pro si applica a codice già esistente

contro rallenta l'esecuzione e non garantisce la scoperta di ogni errore

info <http://www-306.ibm.com/software/awdtools/purify/>

valutazione: ★★★



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Analisi con valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- The Valgrind distribution currently includes six production-quality tools:
 - a memory error detector,
 - two thread error detectors,
 - a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a heap/stack/global array overrun detector, a second heap profiler that examines how heap blocks are used, and a SimPoint basic block vector generator.
 - It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, X86/Darwin and AMD64/Darwin (Mac OS X 10.5 and 10.6).
- Is open source (www.valgrind.org)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio con valgrind

```
#include <stdlib.h>
void f(void)
{
    int* x =
        malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

- Compile your program with -g to include debugging information so that Memcheck's error messages include exact line numbers.
- `valgrind --leak-check=yes myprog`
- Che errori trova ...
- Attenti non trova alcuni errori (overrun di variabili statiche)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio uso con <https://github.com/dynamorio/drmemory>

- Per windows potete usare drmemory
- Cpp check
- .. Tanti altri



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Librerie Safe per le stringhe

Scopo: evitare i buffer overflow e altri problemi tipici delle stringhe e dei buffer di char

1. Safe C String Library

<http://www.zork.org/safestr/>

Autori: Matt Messier e John Viega

2. ISO/IEC TR 24731

Meyers, Randy. Specification for Safer, More Secure C Library Functions, ISO/IEC TR 24731, June 6, 2004

www.open-std.org/jtc1/sc22/wg14/www/docs/n1172.pdf

Supportato da Microsoft



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

SafeC

approccio	traduttore da C a C (C fatto sicuro)
input	programmi C (qualsiasi)
output	programmi C sicuri

metodo garantisce la cattura delle violazioni di memoria e vari errori run time inserendo dei controlli e aggiungendo informazioni (ad esempio ai puntatori)

pro	si applica a C codice già esistente
contro	rallenta l'esecuzione e aumenta la memoria necessaria
info	http://www.eecs.umich.edu/~taustin/

valutazione: ★



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

<http://sourceforge.net/projects/safeclib/>

- The Safe C Library provides bound checking memory and string functions per ISO/IEC TR24731. These functions are alternative functions to the existing Standard C Library. The functions verify that output buffers are large enough for the intended result, and return a failure indicator if they are not. Optionally, failing functions call a "runtime-constraint handler" to report the error. Data is never written past the end of an array. All string results are null terminated. In addition, the functions in ISO/IEC TR 24731-1:2007 are re-entrant: they never return pointers to static objects owned by the function.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Uso di sottoinsieme del C

- Posso decidere di usare un sottoinsieme del C
- Senza aritmetica dei puntatori?
- Senza puntatori?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

MISRA C

IAR Systems implements the The Motor Industry Software Reliability Association's *Guidelines for the Use of the C Language in Vehicle Based Software*.

MISRA C describes a subset of C, suited for use when developing safety-critical systems.

MISRA C is a set of rules to be checked by the compiler. A message is generated for every deviation from a required or advisory rule, unless you have disabled it. Each message contains a reference to the MISRA C rule deviated from.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Categories of Rules

Environment rules

Character sets

Comments

Identifiers

Types

Constants

Declaration and definitions

Initialization

Operators

Conversions

Expressions

Control flow

Functions

Preprocessing directives

Pointers and arrays

Structures and unions

Standard Libraries



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example – Comments rule

Rule 9 (required) Comments shall not be nested.

How the rule is checked?

The compiler will generate an error, indicating a violation of this rule, if /* is used inside a comment.

Rule 10 (advisory) Sections of code should not be ‘commented out’.

How the rule is checked?

The compiler will generate an error, indicating a violation of this rule, whenever a comment ends with ;, {, or }.

Note: This rule is checked in such a manner that code samples inside comments are allowed and do not generate an error.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example – Identifier rule

Rule 12 (required)

No identifier in one namespace shall have the same spelling as an identifier in another namespace.

How the rule is checked?

The compiler will generate an error, indicating a violation of this rule, if a declaration or definition would hide an identifier if they were in the same namespace.

For example, fields of different structures will not generate an error.

Example of rule violations

```
struct an_ident { int an_ident; } an_ident;
```

Example of correct code

```
struct a_struct { int a_field; } a_variable;
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example – Types rule

Rule 13 (advisory) The basic types of char, int, short, long, float, and double should not be used,

but specific-length equivalents should be typedef'd for the specific compiler, and these type names used in the code.

How the rule is checked

The compiler will generate an error, indicating a violation of this rule, if any of the basic types given above is used in a declaration or definition that is not a typedef.

Example of rule violations

```
int x;
```

Example of correct code

```
typedef int SI_16  
SI_16 x;
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Example – Constant rule

Rule 18 (advisory) Numeric constants should be suffixed to indicate type, where an appropriate suffix is available.

How the rule is checked?

The compiler will generate an error, indicating a violation of this rule, for any integer constant whose type is not the same in any standard-conforming implementation.

Example of rule violations

100000

Examples of correct code

30000

100000L

100000UL



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sui puntatori:

MISRA rule that states the only pointer math allowed is the indexing operation.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

CCured

approccio traduttore da C a C (C fatto sicuro)

input programmi C con annotazioni particolari (opzionali)

output programmi C sicuri

metodo abbina analisi statica, controlli dinamici e garbage collector

pro si applica a C codice già esistente con minime modifiche

contro rallenta l'esecuzione

info <http://manju.cs.berkeley.edu/ccured/>

valutazione: ★★



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Cyclone

approccio	safe C-like language
input	programmi C modificati
output	programmi C sicuri
metodo	controlli dinamici solo dove necessario e garbage collector
pro	minimo overhead di tempo e di memoria
contro	richiede di modificare i programmi originali
info	http://cyclone.thelanguage.org/
valutazione:	★★★



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Vault

approccio safe C-like language

input programmi C modificati

output COM objects

metodo linguaggio astratto simile a Java/C#

pro minimo overhead di tempo e di memoria

contro richiede di riscrivere i programmi originali

info <http://research.microsoft.com/vault/>

valutazione: ★★



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Confronto sui linguaggi

Controllo sui dettagli a basso livello

- SafeC e CCured: pieno utilizzo del C, operazioni limitate sui puntatori
- Cyclone: più restrittivo del C
- Vault: meno efficiente, più astratto

Opzioni sulla gestione della memoria

- Cyclone: diverse opzioni
- Vault: oggetti “lineari” e regioni
- SafeC: malloc() e free() esplicite
- CCured: garbage collection



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Confronto dei costi

Prestazioni

Cyclone ≈ Vault < CCured << SafeC

Aumento memoria

Cyclone ≈ Vault < CCured << SafeC

Sforzo per annotare i tipi

SafeC < CCured < Cyclone ≈ Vault

Sforzo per portare codice esistente

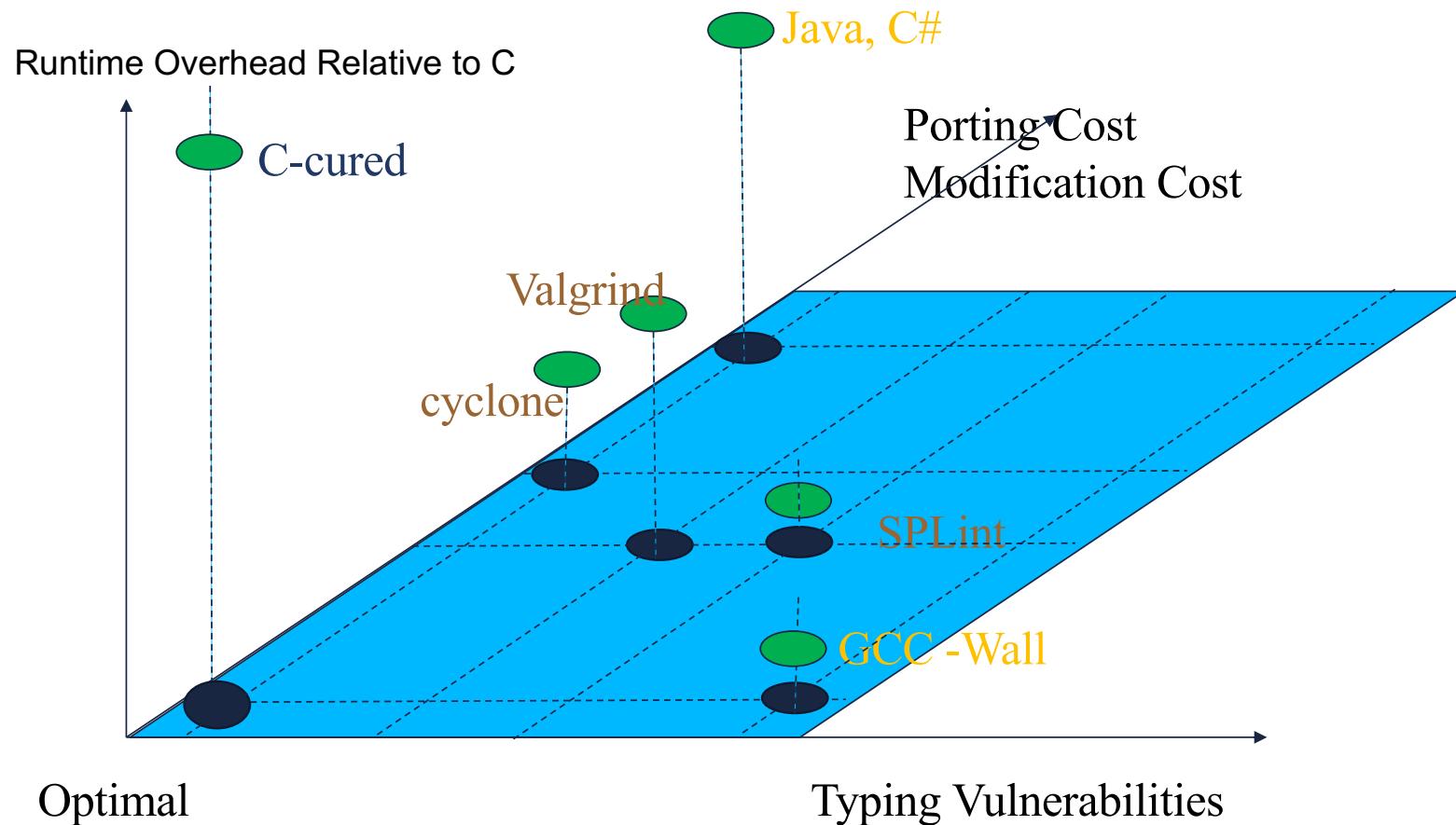
SafeC < CCured < Cyclone << Vault



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Conceptual Space



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In sintesi

- Abbiamo visto:
 - quali sono i pro e contro ad usare i linguaggi safe ad alto livello invece che il C
 - possibili modi di rendere il C safe
- Ricordate che:
 - è possibile effettuare l'analisi dinamica con tool come purify
- Per essere certi di avere programmi in C safe abbiamo visto e confrontato questi tool:
 - SafeC, CCured, Cyclone, Vault



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione





UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

08 – Introduzione all’Object- Orientation

Programmazione Avanzata

Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

INGEGNERIA INFORMATICA

Prof. Claudio MENGHI

Dalmine

17 Ottobre 2023

Schema della lezione

Cenni di progettazione Object-oriented

Concetti principali dell'object-orientation

1. Incapsulamento (abstraction)
2. Sottotipazione (subtyping)
3. ereditarietà (inheritance)
4. binding dinamico (dynamic binding/lookup)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Oggetti - Objects

- Un oggetto consiste in
 - dati nascosti
 - dati o variabili (di istanza)
 - anche possibili funzioni
 - operazioni pubbliche
 - metodi o funzioni membro
 - anche possibili variabili
- Sistemi Object-oriented
 - oggetti **mandano** messaggi **object → msg(arguments)** ad altri oggetti
 - (chiamate di funzioni/metodi) **object.method(arguments)**

OGGETTO	
dati nascosti	
msg ₁	method ₁
...	...
msg _n	method _n



Cosa c'è di interessante?

- Costrutto di encapsulamento generale
 - Strutture Dati
 - File system
 - Database
 - Window
 - Sistema Operativo ...



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Object-orientation

- Tutto è “Object-Oriented” ?
- Per noi è:
 - metodologia di progettazione/programmazione
 - organizzare concetti in oggetti e classi
 - costruire sistemi estensibili
 - utilizzando i seguenti concetti
 - dati e funzioni sono encapsulati in **oggetti**
 - la **sottotipazione** permette l'estensione dei tipi di dati
 - **l'ereditarietà** permette il riuso delle implementazioni



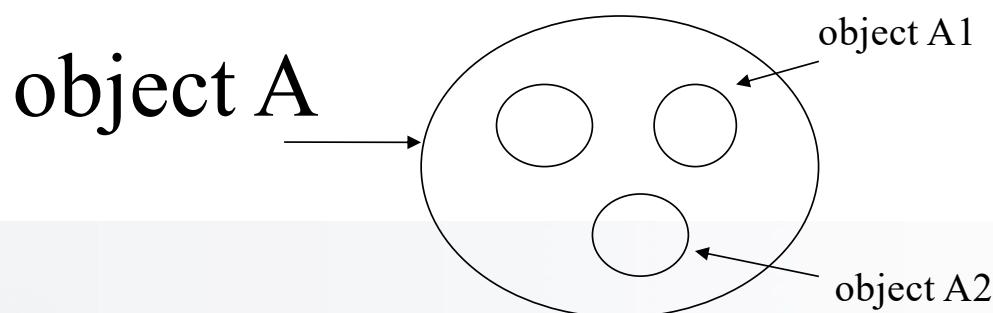
Progettazione Object-oriented [Booch]

- ◆ Quattro passi
 1. Identifica gli **oggetti** ad un certo livello d'astrazione
 2. Identifica la semantica (cioè il **comportamento** desiderato) degli oggetti
 3. Identifica **le relazioni** tra gli oggetti
 4. **Implementa** gli oggetti
- ◆ Processo iterativo
 - Implementa gli oggetti (punto 4) mediante i quattro passi
- ◆ Non necessariamente “top-down”
 - “livello d'astrazione” a qualsiasi livello



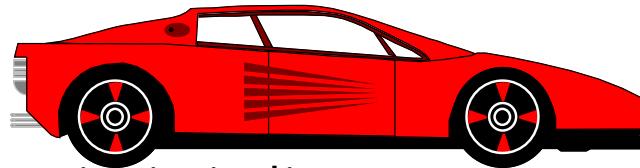
Progettazione OO

- Associa oggetti ai **componenti** o ai **concetti** di un sistema
- Perché iterativo (**raffinamento**)?
 - Un oggetto è tipicamente implementato usando un numero di oggetti che lo costituiscono
 - Si applica la stessa metodologia agli oggetti individuati (componenti o concetti)



Esempio: calcolo del peso di una automobile

- Oggetto “AUTO” :
 - Contiene una lista delle sue parti principali
 - telaio, motore, ruote,
 - Metodo per calcolare il peso
 - somma il peso dei componenti
- Oggetti componenti:
 - Ognuno può avere una lista delle sottocomponenti
 - Ognuno deve avere un metodo per il calcolo del peso



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Confronto con la progettazione top-down

- Somiglianza:
 - Un compito viene portato a termine completando un numero di sotto compiti più piccoli (*divide et impera*)
- Però:
 - si raffinano non solo le procedure ma anche la rappresentazione dei **dati**
 - **modellare** i concetti (dati e operazioni) del sistema
 - gli oggetti raggruppano dati e funzioni rendendo il raffinamento più naturale



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Concetti dell'Object-Orientation

- **incapsulamento - encapsulation**
- **sottotipazione - subtyping**
 - per estendere i concetti
- **ereditarietà - inheritance**
 - per riusare le implementazioni
- **binding dinamico - dynamic lookup**



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

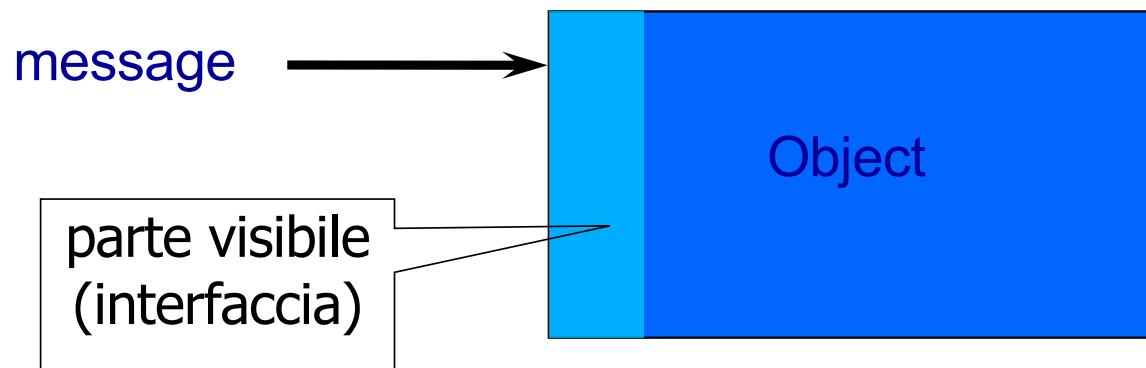
Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Incapsulamento

chi costruisce l'oggetto ha (deve avere) una vista dettagliata

chi usa un oggetto (utente o cliente) ha una vista astratta

L'incapsulamento è il meccanismo per separare queste due viste



Incapsulamento e ling. di programmazione

- Esistono diverse approcci all'incapsulamento:
- Anche linguaggi come il C offrono dei modi che vedremo
- Abstract Data Types

Principio dell' Information hiding: **segregation** of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).

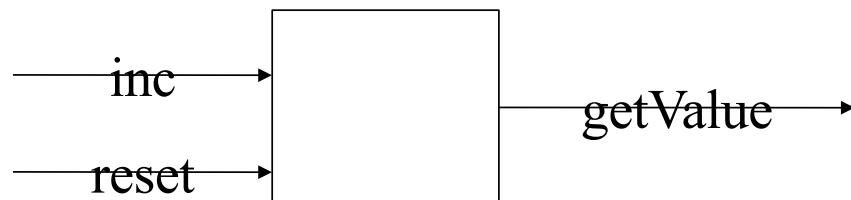


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Caso di studio con il C

- Vogliamo realizzare un contatore:
 1. un valore intero, variabile nel tempo
 2. tre operazioni (astrazione sulle operazioni)
 - reset() per impostare il contatore a zero
 - inc() per incrementare il valore attuale del contatore
 - getValue() per recuperare il valore attuale del contatore sotto forma di numero intero



.h e .c in C

- In C, il primo meccanismo per l'incapsulamento è la separazione tra file header e file source
 - File header .h viene importato dai client (con #include)
 - Il file .c contiene l'implementazione e viene unito agli altri moduli solo nel momento del linker
 - Se cambio il .c non devo cambiare nulla altro, se cambio il .h, i client che lo usano posso aver bisogno di cambiamenti
 - Esempio: cambio il nome ad un metodo.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sol. 1 Come nuovo tipo (int) di C

- Dichiarazione (in counter.h)

```
typedef int contatore;  
void reset(contatore*);  
void inc(contatore*);  
int getValue(contatore);
```

- Uso (in client.c)

```
#include "counter.h"  
int main(void) {  
    int v1, v2;  
    contatore c1, c2;  
    reset(&c1); reset(&c2);  
    inc(&c1); inc(&c1); inc(&c2);  
    v1 = getValue(c1);  
    v2 = getValue(c2);  
    return EXIT_SUCCESS;  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sol. 1 Come nuovo tipo (int) di C

- Dichiarazione (in counter.h)
typedef int contatore;
void reset(contatore*);
void inc(contatore*);
int getValue(contatore);
- Nota che
- reset e inc prendono come parametro formale un puntatore (**passaggio per riferimento**) perché devono modificare il contatore.
- getValue non modifica il contatore, allora passo il contatore **per valore**.
- Quando chiamo reset e inc devo usare l'operatore & :
- inc(&c2);



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Definizione di contatore e delle operazioni

- Poi devo definire cosa fanno i metodi (in counter.c)

```
#include "counter.h"

void reset(contatore* pc) {
    *pc = 0;
}

void inc(contatore* pc) {
    (*pc)++;
}

int getValue(contatore c) {
    return c;
}
```



Vantaggi e svantaggi del typedef

- Consente di separare interfaccia e implementazione
- Rende il cliente indipendente dalla struttura interna del tipo di dato (servitore)
 - **Esercizio:** proviamo a cambiare l'implementazione di contatore
 - Permette al cliente di definire tanti contatori quanti gliene occorrono
 - Ma non garantisce information hiding: tutti i clienti vedono la typedef, conoscono la struttura interna del contatore e possono violare il protocollo di accesso
 - **Esercizio:** proviamo a scrivere un cliente che incrementa di due



Sol2.- Contatore in C con un modulo

- Dichiarazione in un modulo (`mcounter.c`): contatore come singola risorsa protetta (int) dentro a un modulo

```
static int cont;
```

static → non va sullo stack

Ogni volta che importo il modulo ho una cella di memoria (static) per cont. Se non fosse static non sarebbe protetta, potrei importarla in altri moduli con **extern**

- con operazioni che agiscono implicitamente su essa, dichirate nel .h e implementate nel .c

```
void reset(void);
```

```
void inc(void);
```

```
int getValue(void);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Modulo

- mcounter.h

```
void reset(void);
```

```
void inc(void);
```

```
int getValue(void);
```

- mcounter.c

```
#include "mcounter.h"
```

```
static int count;
```

```
void reset(void) {
```

```
    count = 0;
```

```
}
```

```
void inc(void) {
```

```
    count ++;
```

```
}
```

```
int getValue(void) {
```

```
    return count;
```

```
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Uso del contatore modulo

- si importano solo le dichiarazioni delle funzioni (mcounter.h) e si usa il contatore definito nel modulo non si definisce un contatore nel main

```
#include "mcounter.h"

main() {
    int v;
    reset();
    inc();
    inc();
    v = getValue();
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Vantaggi e svantaggi del modulo

- Separa interfaccia e implementazione
- Rende il cliente indipendente dalla struttura interna del modulo (servitore)
- Garantisce l'incapsulamento
 - i clienti vedono solo le dichiarazioni delle operazioni: non conoscono la struttura interna della risorsa (privata) del modulo
- Offre al cliente una **singola** risorsa (da usare senza doverla definire): non è adatto se servono più risorse



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Obiettivo

- poter nascondere i dettagli dell'implementazione (come con l'uso del modulo)
- garantire information hiding e encapsulamento
- permettere modifiche all'implementazione
- poter definire e utilizzare più contatori (come con il typedef)
- poter introdurre tanti contatori e fare le operazioni su di essi



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Confronto con gli ADT

- Simile all'approccio tradizionale degli **abstract data types (ADT)**
- **Vedi:**
http://en.wikipedia.org/wiki/Abstract_data_type
- ADT:
 - Un tipo astratto +
 - Operazioni che posso fare (senza dire l'implementazione)
- Vantaggi degli ADT
 - si può separare l'interfaccia dall'implementazione
- Svantaggi
 - **vedi esempio**
 - due tipi di figura geometrica: Quadrato e Rettangolo



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Abstract Data Types

- is a specification of a set of data and the set of operations that can be performed on the data.
- it is independent of various concrete implementations
- The interface provides a constructor, which returns an abstract handle to new data, and
- several operations, which are functions accepting the abstract handle as an argument.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Implementazione degli ADT

- In computer science, an **opaque data type** is a data type that is incompletely defined in an interface, so that ordinary client programs can only manipulate data of that type by calling procedures that have access to the missing information.
- Some languages, such as C, allow the declaration of **opaque records** (structs), whose size and fields are hidden from the client.
- The only thing that the client can do with an object of such a type is to take its address, to produce an **opaque pointer**.

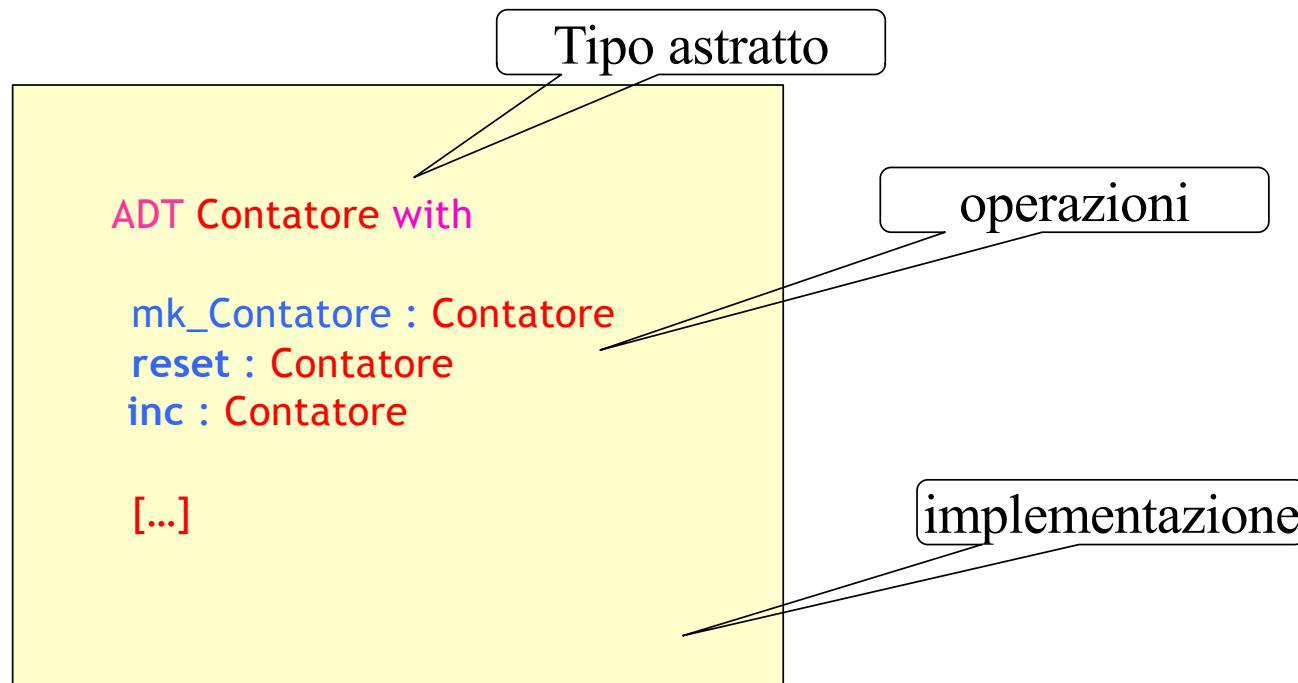


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Abstract data types: Contatore

Si possono definire anche in modo astratto (vedi wikipedia)



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Contatore come ADT in C

- Possiamo implementare un ADT in C con il **struct/pointer opaco**.
- Usando I puntatori il compilatore può finire anche senza sapere la dimensione reale del record.
- In .h

```
typedef struct counter *counterRef;  
counterRef make_counter(void);  
int getValue(counterRef c);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Implementazione

Nel .c metto l'implementazione, con la definizione dei tipi e dei metodi.

```
struct counter { /* counter is implemented as */
    unsigned long value; /* value */
};

/* Create new counter instance, initially null. */
counterRef make_counter(void){
    counterRef result = malloc(sizeof(struct counter));
    result->value = 0;
    return result;
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Nota

- Se cambio la struttura, aggiungo un nome di un cambio, cambio l'implementazione di un metodo, non devo cambiare i programmi che usano il contatore.h
- ATTENZIONE: in genere devo fare anche un “distruttore” altrimenti posso avere risorse non liberate.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Uso di void*

- Alcune volte si implementano i tipi opachi come void* (= puntatore generico)
- Nel .h

```
typedef void* counterRef;
```

```
counterRef create();
```

Nel .c – dovrò fare il cast quando richiesto

```
typedef struct{ int val;} counterStruct;  
counterRef create(){  
    counterStruct* pointer = malloc(sizeof(counterStruct));  
    pointer -> val = 0;  
    return (counterRef)pointer;  
}  
  
void print(counterRef c){  
    printf("%d",((counterStruct*)c)->val);  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Abstract data types: Quadrato

Si possono definire anche in modo astratto (vedi wikipedia)

ADT Quadrato with

mk_Quadrato : point * point -> Quadrato

area : Quadrato -> float

move : Quadrato * point -> Quadrato is

in

program

end



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Rettangolo, simile a Quadrato

ADT Rettangolo with

mk_Rettangolo : point * point -> Rettangolo

area : Rettangolo -> float

move : Rettangolo * point -> Rettangolo is ...

in

program

end



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Problemi con gli Abstract Data Types (C)

- Non posso mischiare **Quadrato** con **Rettangolo**
 - anche se le operazioni sono uguali
 - se dichiaro una variabile devo sapere se è di un tipo o di un altro
- “riuso” limitato
 - non posso riusare un codice scritto per un ADT per un altro ADT
- Data abstraction è una parte importante dell'OO ma viene proposta in modo **estensibile**
 - mediante i meccanismi di ereditarietà e sottotipazione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Concetti dell'Object-Orientation

- encapsulamento - encapsulation
- **sottotipazione - subtyping**
 - per estendere i concetti
- **ereditarietà - inheritance**
 - per riusare le implementazioni
- binding dinamico - dynamic lookup



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sottotipazione ed Ereditarietà

- Interfaccia
 - La vista **esterna** di un oggetto (del cliente)
- **Sottotipazione**
 - Relazione tra interfacce
 - I di sottotipo contiene la I del supertipo
- **Implementazione**
 - La rappresentazione **interna** di un oggetto
- **Ereditarietà**
 - Relazione tra implementazioni
 - Il codice della superclasse viene ereditato nella sottoclassse

I due concetti sono strettamente legati
ma distinti



Interfaccia di un oggetto

- Interfaccia
 - i **messaggi** che l'oggetto può ricevere
- Esempio: point
 - x-coord : returns x-coordinate of a point
 - y-coord : returns y-coordinate of a point
 - move : metodo per spostare un punto
- L'interfaccia di un oggetto è il suo **tipo**

QUESTIONI

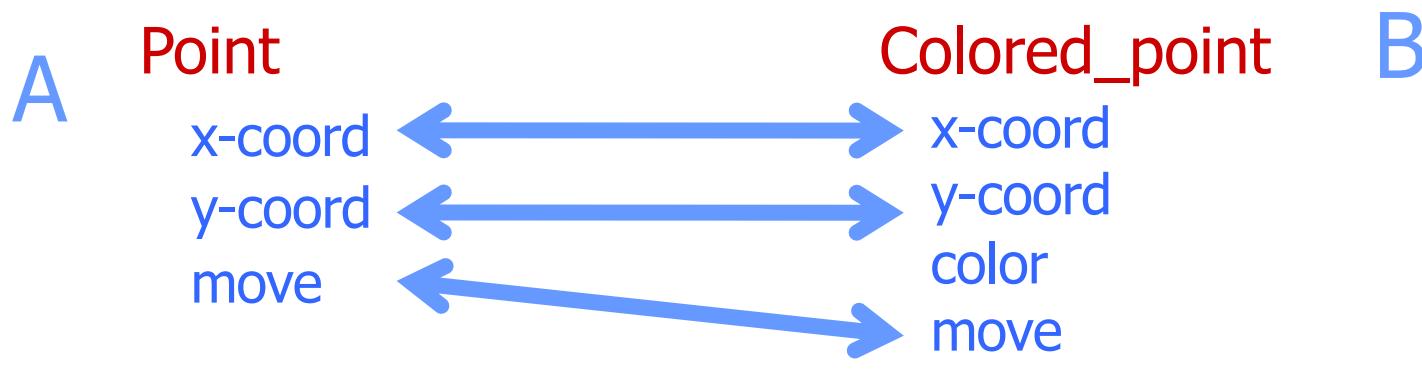


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sottotipi

- Se un interfaccia B **contiene** l'interfaccia A, allora un oggetto B può essere usato al posto di un oggetto A
- B è un **sottotipo** di A
 - **principio di sostituibilità**



- ◆ L'interfaccia di Colored_point contiene Point
 - Colored_point è un sottotipo di Point4



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Polimorfismo (di sottotipo)

- Se **B** è un **sottotipo** di **A**
dove c'è un termine di tipo **A** posso mettere un oggetto di tipo **B**
 - tutte le operazioni continueranno a funzionare
 - nella definizione di **variabili**
 - es. dichiaro var di tipo A: **A var;**
 - var potrebbe essere un oggetto di tipo B: **var = new B;**
 - es. dichiaro X di tipo Point: **Point X;**
 - X potrebbe essere un Colored_point
 - **X = new Colored_point;**
 - **variabili polimorfiche**



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

In java

```
class A{}  
class B extends A {}  
...  
Object o = new A(); // A è sottotipo di Object  
A h = new B();  
B j = new B();  
h = j;  
int x = 0;  
long l = x;  
B k = new A();  
B t = (B)h;
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Ereditarietà - Inheritance

- Nuovi oggetti possono essere definiti **riusando** (anche parzialmente) implementazioni di altri oggetti
- Meccanismo relativo alle **implementazioni**
- Ad esempio una classe **B** (figlio) può ereditare definizioni (codice) di una classe **A** (padre) evitando duplicazione di codice
 - B riusa codice di A



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Potenzialità dell'ereditarietà

`class A { int function (int x) ...}`

B eredità da A: `class B inherits A`

- B eredita il codice (membri: metodi e variabili) da A
 - A può nascondere qualcosa a B (**private**)
- B può introdurre nuovi membri

`class B { float foo (String x) ...}`

- B può **redifinire** alcuni membri di A

-in genere senza cambiare segnatura

`class B {float function (float x)}`

`class B {int function (int x) ...}`

non ridefinisce
fun di A

OK: ridefinisce **fun** di A

- B potrebbe **nascondere** alcuni membri di A



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esem

Sottotipazione e Ereditarietà sono diverse

EREDITARIETÀ NON È SOTTOTIPAZIONE

```
class Point
    private
        float x, y
    public
        point move (float dx, float dy);

class Colored_point
    private
        float x, y; color c
    public
        point move(float dx, float dy);
        point change_color(color newc);
```

◆ Subtyping

- Colored points possono essere usati al posto di points
- interessa il cliente

◆ Inheritance

- Colored points possono essere implementati usando l'implementazione di point
- Interessa l'implementatore



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio: ereditarietà != sottotipazione

[Snyder]

- Ho le seguenti tre strutture dati
 - Coda
 - posso inserire e rimuovere un elemento
 - il primo elemento che inserisco che è il primo che tolgo (FIFO)
 - Pila
 - posso inserire e rimuovere un elemento
 - il primo elemento che tolgo è l'ultimo inserito (LIFO)
 - Lista (già implementato)
 - posso inserire in testa: `insert_at_head`
 - posso inserire in coda: `insert_at_tail`
 - posso rimuovere dalla coda: `remove_at_tail`
- 
- 
- 



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio: ereditarietà != sottotipazione

- implemento (ad esempio in C++) **Coda** e **Pila** **riutilizzando** l'implementazione di **Lista**:
 - **Coda.insert = Lista. insert_at_head**
 - **Coda.remove = Lista. remove_at_tail**
 - **Pila.insert = Lista. insert_at_tail**
 - **Pila.remove = Lista. remove_at_tail**
- e nasconde in Pila e Coda le operazioni **insert_at_X** della Lista
- Coda e Pila ereditano da Lista però non sono sottotipi: non posso più usare Pila al posto di Lista
 - anzi concettualmente Lista è un sottotipo di Pila e di Coda perchè contiene l'interfaccia, cioè le operazioni di Pila e Coda
 - sotto alcune condizioni “forti” ereditarietà e sottotipazione coincidono



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Ereditarietà non è sottotipazione

- nei linguaggi OO **sottotipazione** e **ereditarietà** sono legate
 - in Java la sottotipazione è espressa mediante il meccanismo delle interfacce
 - interface A; B implements A: B è sottotipo di A ma non eredita nulla
 - in C++ subtyping ed ereditarietà pubblica coincidono
- Se si mettono vincoli sull'ereditarietà, possono coincidere
 - In Java posso ridefinire un metodo solo senza cambiare la segnatura -> sottoclasse è sottotipo
- sono però due concetti distinti
 - **sottotipazione** è riferito alle **interfacce**
 - **ereditarietà** è riferito alle **implementazioni**



Concetti dell'Object-Orientation

- **incapsulamento - encapsulation**
- **sottotipazione - subtyping**
 - per estendere i concetti
- **ereditarietà - inheritance**
 - per riusare le implementazioni
- **binding dinamico - dynamic lookup**
 - codice diverso per oggetti diversi



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Binding Dinamico

nell'approccio OO

- object -> message (arguments)
- il codice eseguito dipende da object e message
- il tipo di object può variare runtime (grazie al polimorfismo)

nei linguaggi di programmazione non OO (tipo C, Pascal), ma anche con gli ADT

- operation (operands)
- il significato è sempre lo stesso



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio

- in OO move di un punto x
x -> move (3,2)

non mi preoccupo che x sia Point o Colored_point: viene deciso runtime

- in Pascal o in C move (x,3,2)
 - so quando compilo quale move viene chiamata



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Overload e binding dinamico

- spesso si confonde binding dinamico con l'overload di un metodo, però
- **overload**: un metodo o operazione con lo stesso nome si applica a diversi tipi
 - **esempio: + va bene per interi e float**
- L'overloading viene risolto al tempo di **compilazione**
 - **esempio a + 2**
 - **2.0 +3.0 : viene utilizzato il + dei float**



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Single dispatch

`x ->message (y)`

il codice eseguito dipende runtime da `x` non da `y`

Si dice “single dispatch”

STATE ATTENTI, vedi esempio



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Single dispatch 2 - Java

Object definisce un metodo **equals** con par. **Object**
class **Object** { boolean **equals** (**Object** o) ...}

A eredita **Obj** e definisce **equals** con parametro **A**
class **A** extends **Object** { boolean **equals** (**A** a) }

A non ridefinisce il metodo **equals** di **Object** !!!

Creo due oggetti **A**

```
Object a1 = new A(); Object a2 = new A();  
a1.equals(a2); // quale equals è eseguito?
```

Se voglio essere sicuro di usare **equals** di **A** devo ridefinire
equals:

```
class A extends Object { boolean equals (Object a) }
```



Che cosa stampa?

```
public class ClassA {  
  
    public void stampa(ClassA p){  
        System.out.println("AAA");  
    }  
}
```

```
public class ClassB extends ClassA {  
  
    public void stampa(ClassB p){  
        System.out.println("BBB");  
    }  
  
    public void stampa(ClassA p){  
        System.out.println("AAA/BBB");  
    }  
}
```

```
public class ClassC extends ClassA {  
  
    public void stampa(ClassC p){  
        System.out.println("CCC");  
    }  
  
    public void stampa(ClassA p){  
        System.out.println("AAA/CCC");  
    }  
}
```

```
public static void main(String[] args) {  
    ClassA a1, a2;  
    ClassB b1;  
    ClassC c1;  
  
    a1 = new ClassB();  
    b1 = new ClassB();  
    c1 = new ClassC();  
    a2 = new ClassC();  
  
    b1.stampa(b1);  
    a1.stampa(b1);  
    b1.stampa(c1);  
    c1.stampa(c1);  
    c1.stampa(a1);  
    a2.stampa(c1);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

- Scrivi una classe A con un membro intero x.
- Con costruttore con un intero da assegnare a x
- Aggiungi il metodo boolean equals(A a) che restituisce true se a.x è uguale a this.x
- Cosa succede se fai

Object a1 = new A(1), a2 = new A(1)

A a3 = new A(1);

System.out.println(a1.equals(a1));

System.out.println(a1.equals(a2));

System.out.println(a3.equals(a1));

System.out.println(a1.equals(a3));



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Altro esercizio

```
class A { foo(A a){...}}  
class B extends A { foo(B b){...}}  
class C extends A { foo(A a){...}}
```

```
A x = new A();  
B y = new B(); A z = new B();  
C w = new C(); A v = new C();  
x.foo(x); x.foo(y); ...  
y.foo(x); y.foo(y); z.foo(x); z.foo(y);  
w.foo(x); w.foo(w); v.foo(x); v.foo(w);
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

OO in pratica

- Esercizio



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio: libreria geometrica

- Definisco il concetto generale Figura
- Implemento due forme: Cerchio, Rettangolo
- Implemento le seguenti funzioni
 - center, move, rotate, print, equals
- Come estendere la libreria?
 - Aggiungi Quadrato come estensione di Rettangolo
- Prova a implementarlo nel tuo linguaggio OO preferito !



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

OO Program Structure

- Group data and functions
- Class
 - Defines behavior of all objects that are instances of the class
- Subtyping
 - Place similar data in related classes
- Inheritance
 - Avoid reimplementing functions that are already defined



Code placed in classes



	center	move	rotate	print
Circle	c_center	c_move	c_rotate	c_print
Rectangle	r_center	r_move	r_rotate	r_print

- Dynamic lookup
 - circle → move(x,y) calls function c_move
- Conventional organization
 - Place c_move, r_move in move function



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Figura

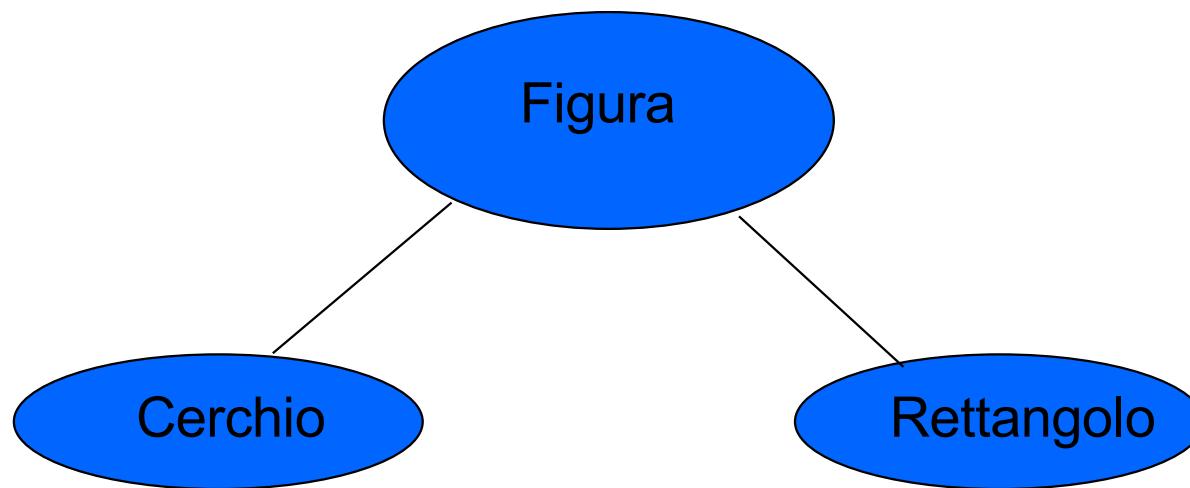
- L'interfaccia di ogni Figura include
`center, move, rotate, print, equals`
- Diversi tipi di Figura hanno implementazioni diverse
 - **Rettangolo:** i quattro vertici
 - **Cerchio:** centro e raggio



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Sottotipi



- L'interfaccia generale è definita in **Figura**
- Implementazioni sono definite in **Cerchio, Rettangolo**
- Si aggiungono facilmente nuove forme



Sommario

1. Cenni di progettazione Object-oriented
 2. Concetti principali dell'object-orientation
 - **incapsulamento**
 - **sottotipo**
 - **ereditarietà**
 - **binding dinamico**
- ◆ Prossime lezioni
- **Confronto tra i diversi linguaggi, come supportano l'OO: C++, Java, ...**



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

09 – Design Pattern

INGEGNERIA INFORMATICA

Programmazione Avanzata

Prof. Claudio MENGHI

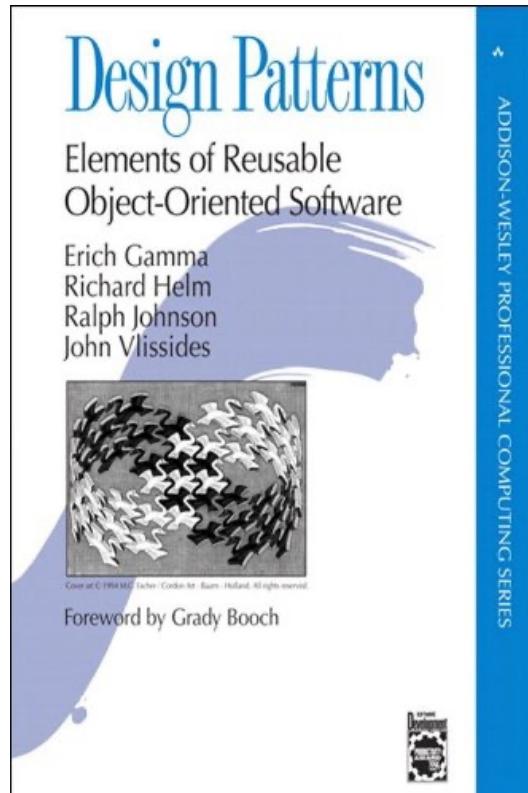
Anno di corso: 1

Anno accademico di offerta: 2023/2024

Crediti: 6

Dalmine

17 Ottobre 2023



1994



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

- Creational Patterns
 - **AbstractFactory**: Creates an instance of several families of classes
 - **Builder**: Separates object construction from its representation
 - **FactoryMethod**: Creates an instance of several derived classes
 - **Prototype**: A fully initialized instance to be copied or cloned
 - **Singleton**: A class of which only a single instance can exist



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

- Structural Patterns

- Adapter: Match interfaces of different classes
- Bridge: Separates an object's interface from its implementation
- Composite: A tree structure of simple and composite objects
- Decorator: Add responsibilities to objects dynamically
- **Facade**: A single class that represents an entire subsystem
- Flyweight: A fine-grained instance used for efficient sharing
- Proxy: An object representing another object



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

- Behavioral Patterns

- **ChainofResp.**: A way of passing a request between a chain of objects
- **Command**: Encapsulate a command request as an object
- **Interpreter**: A way to include language elements in a program
- **Iterator**: Sequentially access the elements of a collection
- **Mediator**: Defines simplified communication between classes
- **Memento**: Capture and restore an object's internal state
- **Observer**: A way of notifying change to a number of classes
- **State**: Alter an object's behavior when its state changes
- **Strategy**: Encapsulates an algorithm inside a class
- **TemplateMethod**: Defer the exact steps of an algorithm to a subclass
- **Visitor**: Defines a new operation to a class without change



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Design Pattern

M3 Object oriented programming su syllabus



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Cosa sono i design pattern

- Quando si e' cominciato a lavorare seriamente con i linguaggi ad oggetto, la gente si e' resa conto che si presentavano ad ogni programma, dei problemi ricorrenti.
- Un gruppo di 4 programmatore (la banda dei 4, GoF(Gang of Four) ha cercato di formulare una lista dei 23 **problemi piu ricorrenti (e delle loro soluzioni)** e cosi' nel 1994 sono nati i Design Patterns.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Idea degli architetti

- L'idea di design pattern si deve agli archietti (vedi libro di Alexander, 1977)
- Vedi libro



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Quali design pattern vedremo

- Singleton
- Facade
- Visitor
- MVC a informatica III B
- Materiale: su wikipedia o su bruce Eckel



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Singleton 10.4 (pag 291)

- Pattern creazionale
- A single instance of the class
- Quando ho bisogno di una unica istanza di una classe
-
- In Java
 - private constructor
 - static member

```
class A{  
    private A(){...}  
    public static A instance = new A();  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercizio

- Implementata Math come Singleton invece che usare metodi statici
- Diverse varianti:
 - Creato subito
 - Accesso all'istanza con metodo
 - Creazione quando richiesta



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Facade

- Structural pattern
- Raggruppo in un singolo oggetto più oggetti di classi distinte e fornisce un accesso a più alto livello agli oggetti sottostanti
- In genere poi passo le richieste agli oggetti sottostanti.
- Il client usa solo la facade



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esempio



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Visitor Pattern es. 10.3



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Synopsis

- Represent an **operation** to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
- You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento di Ingegneria dell'Informazione e della Produzione

Visitor Pattern

- Problem
 - ❑ Operations on collections of objects may not apply to all objects, or apply differently to different objects
- Context
 - ❑ Object interfaces are fixed and diverse
 - ❑ Need to allow new operations, without polluting” their classes with these operations.
- Solution
 - ❑ Represent operations to be performed as visitors, with the interface of every visitor representing the different kinds of objects



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Context

- You should use the Visitor pattern when:
 - Many distinct and unrelated operations need to be performed on an object structure, and you want to avoid “polluting” their classes with these operations.
 - The classes defining the object structure rarely change, but you often want to define new operations over the structure.
 - An object contains many classes of objects with differing interfaces.

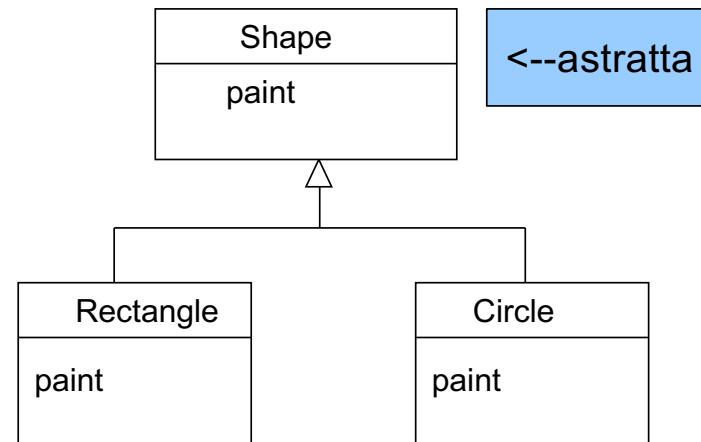


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

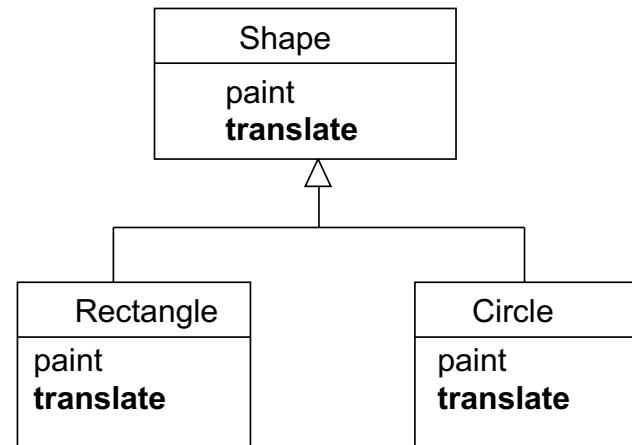
Example: problem

- Imagine you have a program that deals with geometric figures: rectangles and circles
- You want to add an **operation**, for example
 - You want to **translate** of shape



Possibile soluzione 1

- aggiungo l'operazione ad ogni classe:



CONTRO:

- Devo modificare le classi originali
- Se ho 10 operazioni devo modificare le classi
- Ho il codice sparso in tutte le classi
 - Posso mettere qualcosa nella superclasse, però...



Soluzione 2

- Creo una classe (singleton) che rappresenta l'operazione:

```
class Translate{  
    process(Shape s){...}  
    process(Rectangle e){...}  
    process(Circle e){...}  
}
```

- Devo stare attento al single dispatch. Esempio
- Shape e = new Circle()
- Translate.getInstance().process(e) ---
 - Va a prender process(shape) quale codice eseguo??
- Dovrei mettere tanti instanceof ...



Soluzione 3 - visitor

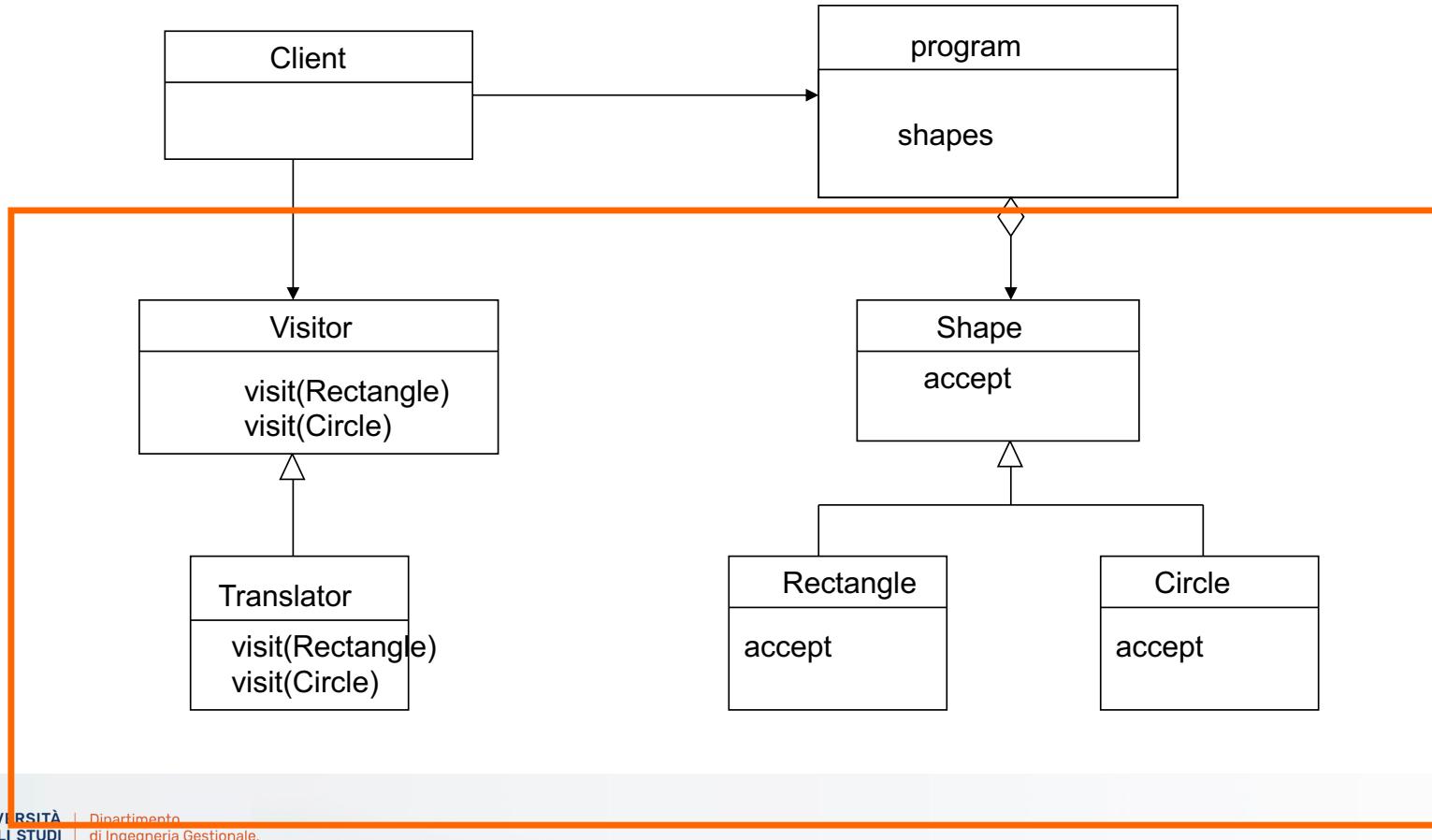
- The best way to do this is to have a (generico) visitor come in and performs the operation
- Ogni classe è in grado di accettare il visitor
- L'operatore/visitatore farà l'operazione translate quando visita l'oggetto



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Visitor Example



lato Visitable

Le classi della gerarchia devono essere visitabili:

```
interface visitable {  
    void accept(visitor v);  
}
```

- ogni classe originale deve implementare visitable (un metodo accept):

```
class Shape implements visitable{  
    abstract accept(visitor v);  
}  
class Rectangle extends Shape{  
    accept(visitor v){ v.visit(this);}  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

lato visitor

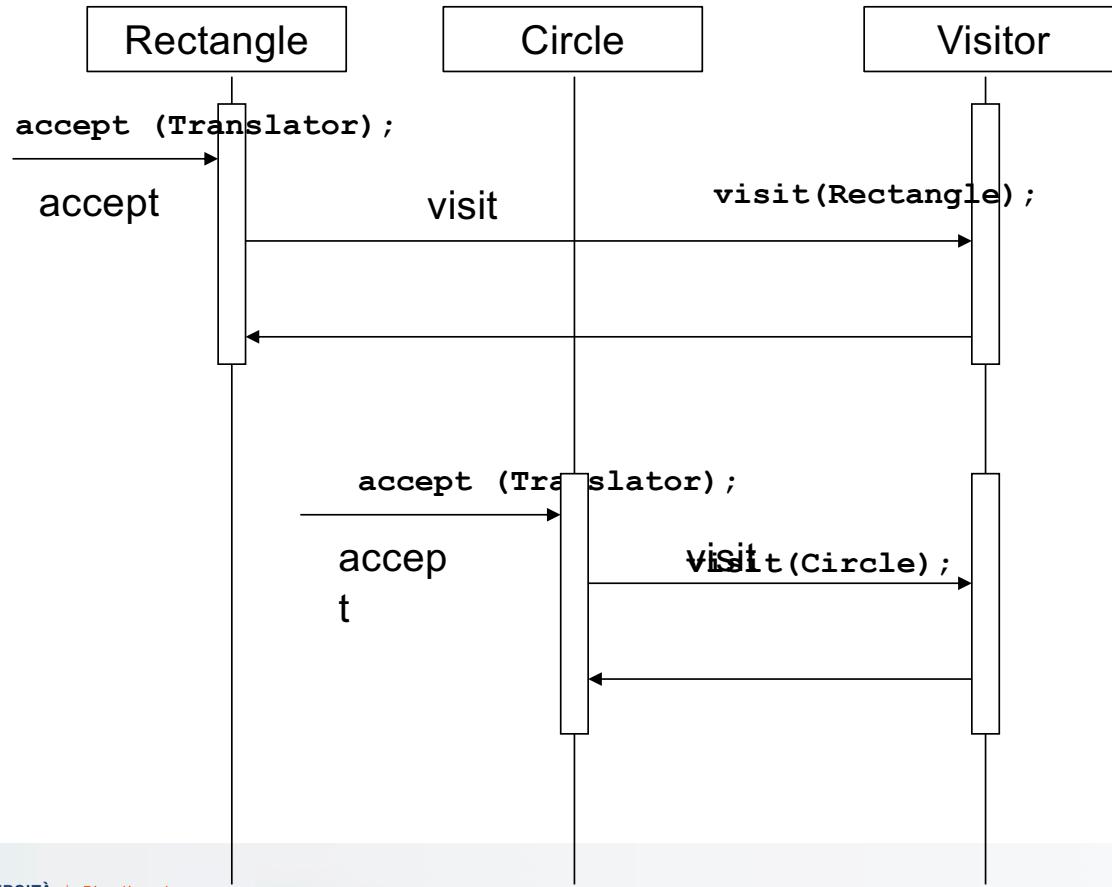
- Translator deve essere un visitor

```
public interface Visitor {  
    public void visit(Rectangle m);  
    public void visit(Circle m);  
}
```

```
public class Translator implements Visitor{  
    public void visit(Rectangle m){  
        System.out.println("translation of r");    }  
    public void visit(Circle m){  
        System.out.println("tarnslation of c");    }  
}
```



Visitor Interactions



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Cosa succede con il single dispatch

- Shape e = new Circle();
- Visitor v = new Translator();
- e.accept(v)

→ e.accept(v)

va a prendere l'accept di Circle !!! (dispatch su e)

→ l'accept quando fa v.visit va a prendere il visit di Translator
(dispatch su v)

Chiamato anche “double dispatch”

Risolve il problema del single dispatch



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Visitor Example

- Se volessi aggiungere una nuova operazione, come ad esempio la rotazione?
- Definisco un nuovo visitor senza modificare le classi figure

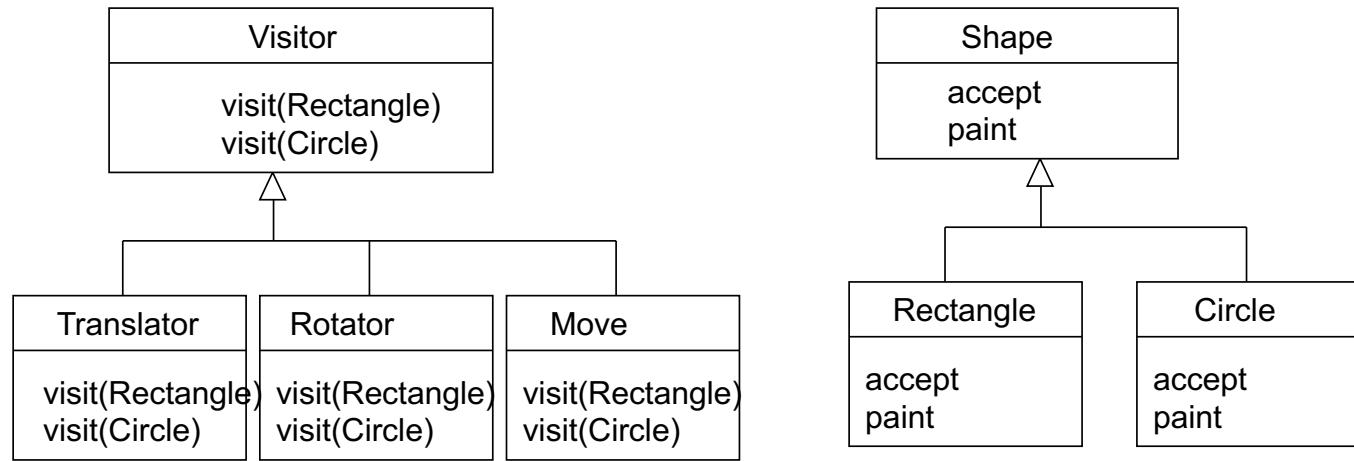
BASTA AGGIUNGERE UN ALTRO VISITOR



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Visitor Example



Consequences

- Positive
 - Visitor makes adding new operations easy, simply add a new visitor that implements that operation.
 - Visitor gathers related operations and separates unrelated ones.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Consequences

- Negative

- Visitor is not good for the situation where "visited" classes are not stable. Every time a new Composite hierarchy derived class is added, every Visitor derived class must be amended
- Often encapsulation is broken because the element class is forced to provide public operations that access internal state.
- If using an existing system changes will be required to existing code.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Related Patterns

- **Iterator**
 - The iterator pattern is an alternative to the Visitor pattern when the object structure to be navigated has a linear structure.
- **Composite**
 - The visitor pattern is often used with object structures that are organized according to the composite pattern.



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Reflection

- Nota che in Java si può usare la reflection per implementare il double dispatch invece che l'uso del pattern !!!



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Idee per il linguaggi diversi

- Multijava
 - <http://multijava.sourceforge.net/>
 - MultiJava is an extension to the Java programming language that adds open classes and symmetric multiple dispatch.
- Nice
 - <http://nice.sourceforge.net/index.html>



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Come fare restituire un valore da un visitor

Normalmente il visitor non restituisce niente. E se devo calcolare qualcosa: come fare? due alternative

1. modifica di visit

- ❑ definire i metodi visit che restitiscano un valore
- ❑ ad esempio restituiscano un Object
 - `Object visit(x...);`
- ❑ poi faccio il cast sapendo cosa effettivamente restituisce

2. aggiungere un campo e un metodo

- ❑ campo result, che viene settato alla fine della visita
- ❑ getResult che restituisce il risultato della visita



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Soluzione 3. visitor e generics

- L'alternativa è dichiarare il Visitor generico rispetto il tipo che restituisce:

```
public interface Visitor <T> {  
    public T visit(Rectangle m);  
    public T visit(Circle e);  
}  
// if the visitor returns a String  
public class GetInfo implements Visitor<String>{  
    public String visit(Rectangle m){  
        return "rettangolo");  }  
}
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

...

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Generic Visitable

- And a generic Visitable with a generic method

```
interface Visitable{  
    public <T> T accept(Visitor<T> ask);  
}  
class Circle implements Visitable{  
    public <T> T accept(Visitor<T> ask) {  
        return ask.visit(this);  
    }  
} ...
```



Java

Angelo Gargantini

Informatica III - 2022/2023

M4 java su syllabus

Outline

1 .Language Overview

- History and design goals

2. Classes and Inheritance

- Object features
- Encapsulation
- Inheritance

3. Types and Subtyping

- Primitive and ref types
- Interfaces; arrays
- Exception hierarchy
- Subtype polymorphism and generic programming
- Saltiamo il resto

Origins of the language

- James Gosling and others at Sun, 1990 - 95
- Oak language for “set-top box”
 - small networked device with television display
 - graphics
 - execution of simple programs
 - communication between local program and remote site
 - no “expert programmer” to deal with crash, etc.
- Internet application
 - simple language for writing programs that can be transmitted over network

Design Goals

- Portability
 - Internet-wide distribution: PC, Unix, Mac
- Reliability
 - Avoid program crashes and error messages
- Safety
 - Programmer may be malicious
- Simplicity and familiarity
 - Appeal to average programmer; less complex than C++
- Efficiency
 - Important but secondary

General design decisions

- Simplicity
 - Almost everything is an object
 - All objects on heap, accessed through pointers
 - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- Portability and network transfer
 - Bytecode interpreter on many platforms
- Reliability and Safety
 - Typed source and typed bytecode language
 - Run-time type and bounds checks
 - Garbage collection

Pro e contro di Java

	Portability	Safety	Simplicity	Efficiency
Interpreted	+	+		-
Type safe	+	+	+/-	+/-
Objects by means of pointers	+		+	-
Garbage collection	+	+	+	-
Concurrency support	+	+		

Java System

- The Java programming language
- Compiler and run-time system
 - Programmer compiles code
 - Compiled code transmitted on network
 - Receiver executes on interpreter (JVM)
 - Safety checks made before/during execution
- Library, including graphics, security, etc.
 - Large library made it easier for projects to adopt Java
 - Interoperability
 - Provision for “native” methods

Java Release History

- 1995 (1.0) – First public release
- 1997 (1.1) – Nested classes
- 2001 (1.4) – Assertions
- 2004 (1.5) – Tiger
 - Generics, foreach, Autoboxing/Unboxing, Typesafe Enums, Varargs, Static Import, Annotations, concurrency utility library
- 2006 (1.6) – Mustang
- 2011 (1.7) – Dolphin

Strings in switch Statement: It enabled using String type in Switch statements

Type Inference for Generic Instance Creation and Diamond Syntax
`List<Integer> list = new ArrayList<>();`
instead of `List<Integer> list = new ArrayList<Integer>();`

Improvements through Java Community Process

Da java 8

- 2014 (1.8) - Lambda Expressions, collections stream, security libraries, JavaFX

- Esempio: list.stream().range(1, 4).forEach(System.out::println);

- **Lambda:** list.forEach(**(n)->System.out.println(n);**);

- Java 9: Modularization

- Java 10: **Local-Variable Type Inference**

- **Esempio:**

```
for (var x : arr)  
    System.out.println(x + "\n");
```

- Java 11: **Running Java File with single command**

- **Java 12: Switch Expressions**

```
switch (x) {  
    case 1 -> System.out.println("Foo");  
    default -> System.out.println("Bar");  
}
```

```
String quarter = switch (month) { case JANUARY, FEBRUARY, MARCH -> "First Quarter"; //must be a single returning value  
case APRIL, MAY, JUNE -> "Second Quarter"; case JULY, AUGUST, SEPTEMBER -> "Third Quarter"; case OCTOBER, NOVEMBER, DECEMBER ->  
"Forth Quarter"; default -> "Unknown Quarter"; };
```

Da java 12

- Java 13 September 17, 2019
- Java 14 March 17, 2020
- Java 15 September 15, 2020
- Java 16 March 16, 2021
- Java 17 (LTS) September 14, 2021
- Java 18 March 22, 2022
- Java 19 September 20, 2022

```
public record Person (String name, String address) {}
```

```
public String checkShape(Shape shape) {  
    return switch (shape) {  
        case Triangle t && (t.getNumberOfSides() != 3) -> "This is a weird triangle";  
        case Circle c && (c.getNumberOfSides() != 0) -> "This is a weird circle";  
        default -> "Just a normal shape";  
    };  
}
```

Outline

- Objects in Java
 - Classes, encapsulation, inheritance
- Type system
 - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
 - Basics, wildcards, ...

Language Terminology

- Class, object -
- Field –
- Method -
- Static members -
- this -
- Package - set of classes in shared namespace
- Native method -

Java Classes and Objects (2)

- Syntax similar to C++
- Object
 - has fields and methods
 - is allocated on heap, not run-time stack
 - accessible through reference (only ptr assignment)
 - garbage collected
- Dynamic lookup
 - Similar in behavior to other languages
 - Static typing => more efficient than Smalltalk
 - Dynamic linking, interfaces => slower than C++

Point Class

```
class Point {  
    static public Point O = new Point(0);  
    private int x;  
    Point(int xval) {x = xval;}      // constructor  
    protected void setX (int y) {x = y;}  
    public int getX()   {return x;}  
}
```

- Visibility similar to C++, but not exactly (later slide)

Use of record instead of class

- As of JDK 14, we can replace our repetitious data classes with records.
Records are immutable data classes that require only the type and name of fields.
- The *equals*, *hashCode*, and *toString* methods, as well as the *private*, *final* fields and *public* constructor, are generated by the Java compiler.
- Ex: public record Person (String name, String address) {}
 - will create a class Person with the final fields name and address, the constructor, equals ...

Object initialization

- Java guarantees constructor call for each object
 - Memory allocated
 - Constructor called to initialize memory
 - Some interesting issues related to inheritance
 - We'll discuss later ...
- Cannot do this (would be bad C++ style anyway):
 - `Obj* obj = (Obj*)malloc(sizeof(Obj));`
- Static fields of class initialized at class load time
 - Talk about class loading later

Static fields and methods

- static field is one field for the entire class, instead of one per object.
- static method may be called without using an object of the class
 - static methods may be called before any objects of the class are created. Static methods can access only static fields and other static methods;
- Outside a class, a static member is usually accessed with the class name, as in `class_name.static_method(args)`,

static initialization block

```
class ... {  
    /* static variable with initial value */  
    static int x = initial_value;  
    /* --- static initialization block --- */  
    static {  
        /* code to be executed once, when class is loaded */  
    }  
}
```

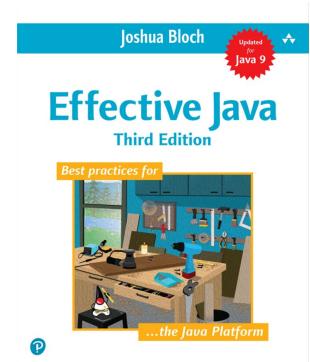
- the static initialization block of a class is executed once, when the class is loaded.

Garbage Collection and Finalize

- Objects are garbage collected
 - No explicit *free*
 - Avoids dangling pointers and resulting type errors
- Problem
 - What if object has opened file or holds lock?
- Solution
 - *finalize* method, **called by the garbage collector**
 - Before space is reclaimed, or when virtual machine exits
 - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
 - Important convention: call super.finalize
 - Don't design your Java programs such that correctness depends upon "timely" finalization.

Uso di finalize è sconsigliato

- Finalizers are unpredictable, often dangerous, and generally unnecessary.
- Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we'll cover later in this item, but as a rule, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. The Java 9 replacement for finalizers is cleaners. Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.



Packages and visibility

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier (friendly)	Y	Y	N	N
private	Y	N	N	N

Estensione delle classi (3)

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
 - Single inheritance only (but Java has interfaces)
- Some additional features
 - Conventions regarding *super* in constructor and *finalize* methods
 - Final classes and methods

Example subclass

```
class ColorPoint extends Point {  
    // Additional fields and methods  
    private Color c;  
    protected void setC (Color d) {c = d;}  
    public Color getC() {return c;}  
    // Define constructor  
    ColorPoint(int xval, Color cval) {  
        super(xval); // call Point constructor  
        c = cval; } // initialize ColorPoint field  
}
```

Class *Object*

- Every class extends another class
 - Superclass is *Object* if no other class named
- Methods of class *Object*
 - `getClass` – return the `Class` object representing class of the object
 - `toString` – returns string representation of object
 - `equals` – default object equality (not ptr equality)
 - `hashCode`
 - `clone` – makes a duplicate of an object
 - `wait`, `notify`, `notifyAll` – used with concurrency
 - `finalize`

Importance of hashCode

- Simply put, *hashCode()* returns an integer value, generated by a hashing algorithm.
- Objects that are equal (according to their *equals()*) must return the same hash code. **Different objects do not need to return different hash codes.**
 - If two objects are equal according to *equals()* method, then their hash code must be same.
 - If two objects are unequal according to *equals()* method, their hash code are not required to be different. Their hash code value may or may-not be equal.
- If hashCode is not correctly implemented, all the Hash* data structure won't work.
- Example

Constructors and Super

- Java guarantees constructor call for each object
- This must be preserved by inheritance
 - Subclass constructor must call super constructor
 - If first statement is not call to super, then call super() inserted automatically by compiler
 - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
 - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

```
ColorPoint () { this(0,blue); }
```

is compiled without inserting call to super
- Different conventions for finalize and super
 - Compiler does not force call to super finalize

Final classes and methods

- Restrict inheritance
 - Final classes and methods cannot be redefined
- Example

java.lang.String

- Reasons for this feature
 - Important for security
 - Programmer controls behavior of all subclasses
 - Critical because subclasses produce subtypes
 - Compare to C++ virtual/non-virtual
 - Method is “virtual” until it becomes final



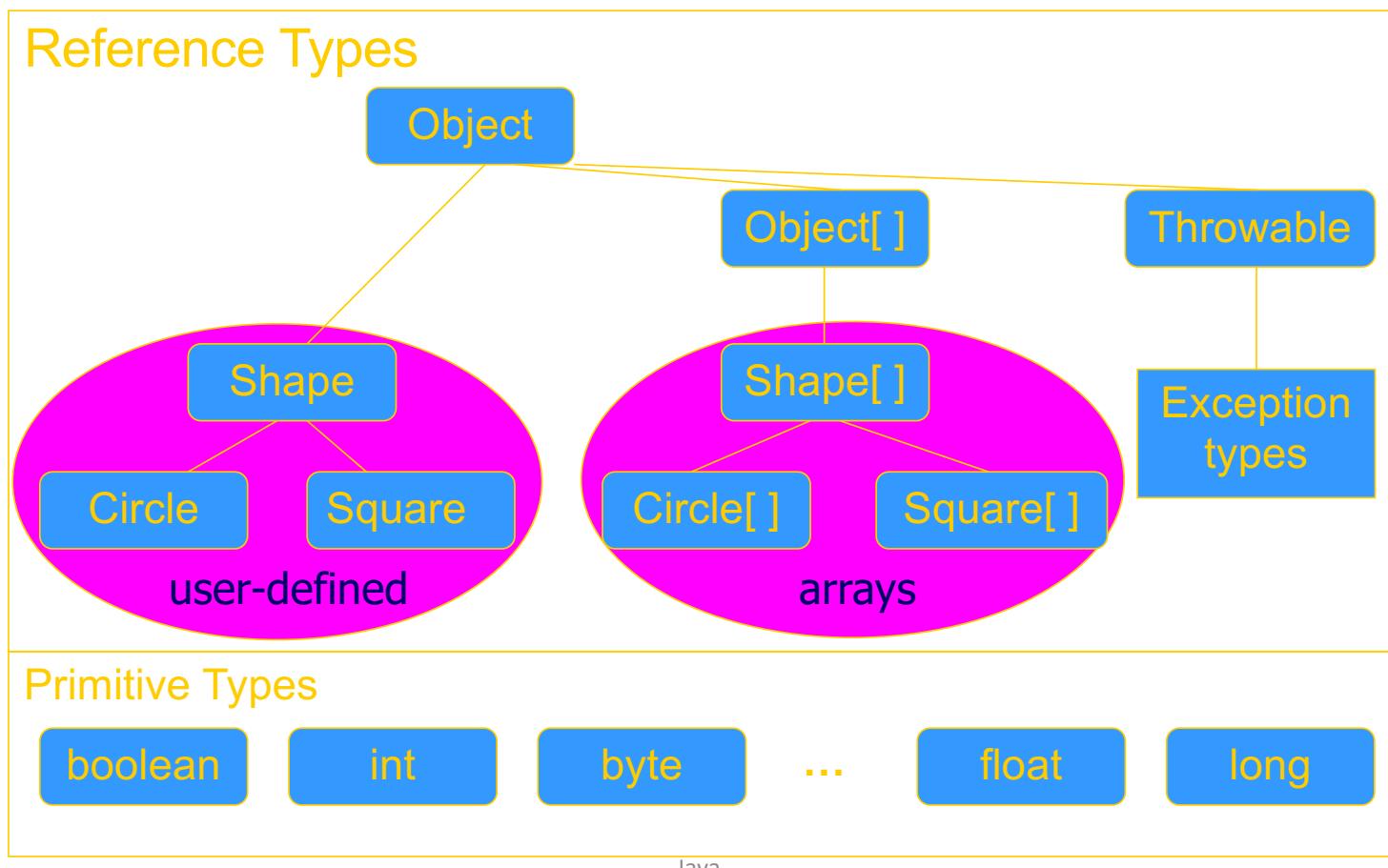
Altri argomenti

- Compatibilità di tipi e conversione
 - Sottoclassi e sottotipi
- Classi astratte e interfacce
- Ereditarietà e ridefinizione dei membri
- Binding dinamico

Java Types

- Two general kinds of types
 - Primitive types – *not* objects
 - Integers, Booleans, etc
 - Reference types
 - Classes, interfaces, arrays
 - No syntax distinguishing Object * from Object
- Static type checking
 - Every expression has type, determined from its parts
 - Some auto conversions, many casts are checked at run time
 - Example, assuming A <: B (A sottotipo di B)
 - Can use A x and type
 - If B x, then can try to cast x to A
 - Downcast checked at run-time, may raise exception

Classification of Java types



Subtyping

- Primitive types
 - Conversions: int -> long, double -> long, ...
- Class subtyping similar to C++
 - Subclass produces subtype
 - Single inheritance => subclasses form tree
- Interfaces
 - Completely abstract classes
 - no implementation
 - Multiple subtyping
 - Interface can have multiple subtypes (extends, implements)
- Arrays
 - Covariant subtyping – not consistent with semantic principles

Java class subtyping

- Signature Conformance
 - Subclass method signatures must conform to those of superclass
- Three ways signature could vary
 - Argument types
 - Return type
 - Exceptions
- How much conformance is needed in principle?
- Java rule
 - Java 1.1: Arguments and returns must have identical types, may remove exceptions
 - Java 1.5: covariant return type specialization

Covariance

- **Covariance** Definizione
- T si dice covariante (rispetto alla sottotipazione di Java) se ogni volta che A è sottotipo di B allora anche T di A è sottotipo di T B
 - T potrebbe essere il valore ritornato
 - ...

Covariance

- **Covariance** in Java 5
- I valori ritornati da un metodo ridefinito possono essere covarianti
- parameter types have to be exactly the same (invariant) for method overriding, otherwise the method is overloaded with a parallel definition instead.

```
class A {  
    public A whoAreYou() { ... }  
}  
  
class B extends A {  
    // override A.whoAreYou *and* narrow the return type.  
    public B whoAreYou() { ... }  
}
```

Java

35

Array types

- Automatically defined
 - Array type $T[]$ exists for each class, interface type T
 - Cannot extend array types (array types are final)
 - Multi-dimensional arrays as arrays of arrays: $T[][]$
- Treated as reference type
 - An array variable is a pointer to an array, can be null
 - Example: `Circle[] x = new Circle[array_size]`
 - Anonymous array expression: `new int[] {1,2,3, ... 10}`
- Every array type is a subtype of $Object[]$, $Object$
 - Length of array is not part of its static type

Array subtyping - covariance

- Covariance

- if $S <: T$ then $S[] <: T[]$
 - $S <: T$ means “ S is subtype of T ”

- Standard type error

```
class A {...}

class B extends A {...}

B[ ] bArray = new B[10]
A[ ] aArray = bArray // considered OK since B[] <: A[]
aArray[0] = new A() // compiles, but run-time error
                    // raises ArrayStoreException
// b/c aArray actually refers to an array of B objects
// so that assignment, aArray[0] = new A(); would violate the type of bArray
```

Interfacce (4)

- Java non ammette ereditarietà multipla
- Però posso definire delle interfacce
 - Lista di metodi che definiscono l'interfaccia
 - Ogni interfaccia identifica un tipo
 - Posso definire sottotipi di interface senza ereditare nulla

Interface subtyping: example

```
interface Shape {  
    public float center();  
    public void rotate(float degrees);  
}  
  
interface Drawable {  
    public void setColor(Color c);  
    public void draw();  
}  
  
class Circle implements Shape, Drawable {  
    // does not inherit any implementation  
    // but must define Shape, Drawable methods  
}
```

Java

39

Properties of interfaces

- Flexibility
 - Allows subtype graph instead of tree
 - Avoids problems with multiple inheritance of implementations (we will see C++ “diamond”)
- Cost
 - Offset in method lookup table not known at compile
 - Different bytecodes for method lookup
 - one when class is known
 - one when only interface is known
 - search for location of method
 - cache for use next time this call is made (from this line)

Tipi enumerativi (6)

Enumeration

- In prior releases, the standard way to represent an enumerated type was the int Enum pattern

```
// int Enum Pattern - has severe problems!  
public static final int SEASON_WINTER = 0;  
public static final int SEASON_SPRING = 1;  
public static final int SEASON_SUMMER = 2;  
public static final int SEASON_FALL = 3;
```

- Not typesafe
- No namespace - You must prefix constants of an int enum with a string (in this case SEASON_)
- Printed values are uninformative

In Java5

```
public enum Season {  
    WINTER, SPRING, SUMMER, FALL }
```

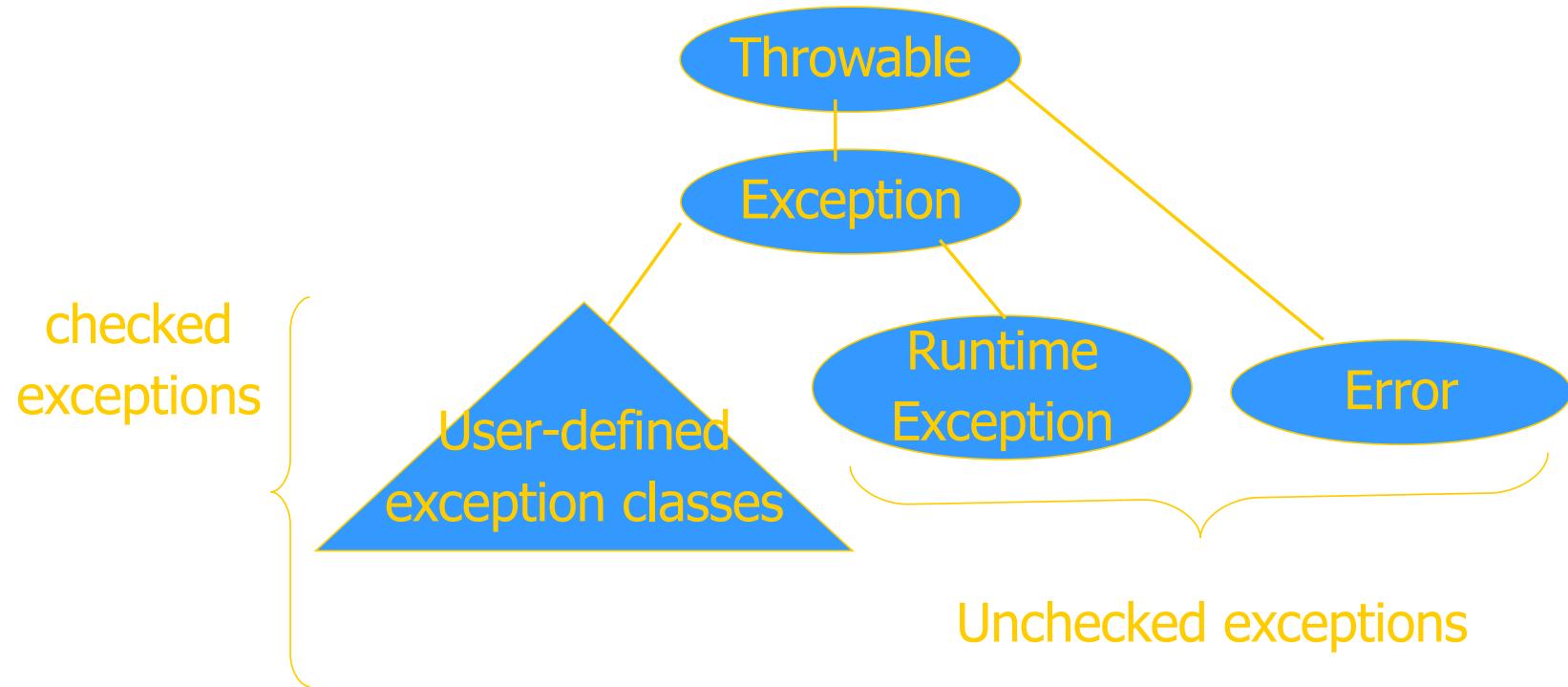
- Comparable
- `toString` which prints the name of the symbol
- static `values` method that returns an array containing all of the values of the enum type in the order they are declared
 - `for (Season s : Season.values()) ...`

Eccezioni e asserzioni (12)

Java Exceptions

- Similar basic functionality to ML, C++
 - Constructs to *throw* and *catch* exceptions
 - Dynamic scoping of handler
- Some differences
 - An exception is an object from an exception class
 - Subtyping between exception classes
 - Use subtyping to match type of exception or pass it on ...
 - Similar functionality to ML pattern matching in handler
 - Type of method includes exceptions it can throw
 - Actually, only subclasses of Exception (see next slide)

Exception Classes



- If a method may throw a checked exception, then this must be in the type of the method

Try/finally blocks

- Exceptions are caught in try blocks

```
try {  
    statements  
} catch (ex-type1 identifier1) {  
    statements  
} catch (ex-type2 identifier2) {  
    statements  
} finally {  
    statements  
}
```

- Implementation: finally compiled to jsr

Why define new exception types?

- Exception may contain data
 - Class Throwable includes a string field so that cause of exception can be described
 - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions

`catch <exception-type> <identifier> { ... }`

will catch any exception from any subtype of exception-type and bind object to identifier

REDEFINIZIONE DEI METODI CON ECCEZIONI

Binding Dinamico in Java

Overload vs Override

- Overload = più metodi o costruttori con lo stesso nome ma diversa segnatura
 - Segnatura: nome del metodo e lista dei tipi dei suoi argomenti
- L'overloading viene risolto in fase di compilazione
- Esempio

```
public static double valoreAssoluto(double x) {  
    if (x > 0) return x;  
    else return -x;  
}  
  
public static int valoreAssoluto(int x) {  
    return (int) valoreAssoluto((double) x);  
}
```

Compilazione: scelta segnatura

- In compilazione viene **scelta la segnatura del metodo da eseguire** in base:
 - (1) al **tipo del riferimento** utilizzato per invocare il metodo
 - (2) al **tipo degli argomenti** indicati nella chiamata

Esempio

- A r;...
- r.m(2)
- Il compilatore cerca fra tutte le segnature di metodi di nome **m** disponibili per il tipo **A** quella **“più adatta”** per gli argomenti specificati

Esempio

A r;

...

r.m(2)

- Se le segnature disponibili per il tipo A sono:

int m(byte b)

int m(long l)

int m(double d)

- il compilatore sceglie la seconda

- Ricordati che byte << short << int << long << float << double

Overriding

- Quando si riscrive in una sottoclasse un metodo della superclasse con la **stessa segnatura**.
- L'overriding viene risolto **in fase di esecuzione**
- **Compilazione:**
- scelta della segnatura: il compilatore stabilisce **la segnatura** del metodo da eseguire (early binding)
- **Esecuzione:**
- scelta del metodo: Il metodo da eseguire, tra quelli con la segnatura selezionata, viene scelto al momento dell'esecuzione, sulla base del **tipo dell'oggetto** (late binding)

Fase di compilazione

(1) Scelta delle segnature “candidate”

- Il compilatore individua le segnature che possono soddisfare la chiamata
 - (a) compatibile con gli argomenti utilizzati nella chiamata il numero dei parametri nella segnatura è uguale al numero degli argomenti utilizzati ogni argomento è di un tipo assegnabile al corrispondente parametro
 - (b) accessibile al codice chiamante
 - Se non esistono segnature candidate, il compilatore segnala un errore.
- ## (2) Scelta della segnatura “più specifica”

- Tra le segnature candidate, il compilatore seleziona quella che richiede il minor numero di promozioni

Esempio 1

A

assegna(x:long)

B eredita da A
e fa overloading

(stesso nome segnatura diversa)

B

assegna(x:int)

assegna(x:double)

C

assegna(x:int)

assegna(x:double)

C eredita da B
e fa overriding

(stesso nome e segnatura)

A alfa;

- alfa.assegna(2)

Una segnatura candidata:

assegna(long x)

- alfa.assegna(2.0)

Nessuna segnatura
candidata (**errore**)

Esempio 2

A

assegna(x:long)

B

assegna(x:int)

assegna(x:double)

C

assegna(x:int)

assegna(x:double)

B beta;

beta.assegna(2)

Tre segnature
candidate:

- **assegna(int x)**
- **assegna(double x)**
- **assegna(long x)**
- La più specifica è
assegna(int x)

Ambiguità

- Se per l'invocazione:
- `z(1, 2)`
- le segniture candidate sono:
 - `z(double x, int y)`
 - `z(int x, double y)`
- Il compilatore non `e in grado di individuare la segnatura pi`u specifica e segnala un messaggio di errore

Esecuzione: scelta del metodo

- La JVM sceglie il metodo da eseguire **sulla base della classe dell'oggetto** usato nell'invocazione
 - cerca un metodo con la segnatura selezionata in fase di esecuzione
 - risalendo la gerarchia delle classi a partire dalla classe dell'oggetto che deve eseguire il metodo

Esempio 1

```
A alpha = new B();  
alpha.assegna(2l)
```

EB: segnatura selezionata in A:
assegna(long x)

LB: Ricerca a partire da B un
metodo assegna(long)

Esegue il metodo di A

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

In questo caso metodo selezionato in EB
ed eseguito coincidono

Esempio 2

```
B beta = new C()  
beta.assegna(2)
```

EB: segnatura selezionata
di B: assegna(int x)

LB: Ricerca a partire da C
un metodo assegna(int)

Esegue il metodo di C

Come volevo,
poichè ho ridefinito il metodo

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

Esempio 3

A alfa = new C()
alfa.assegna(2)

EB Una segnatura
candidata: assegna(long
x)

LB: Ricerca a partire da C
un metodo
assegna(long)

Esegue il metodo di A
anche se 2 è int !!!

A
assegna(x:long)

B
assegna(x:int)
assegna(x:double)

C
assegna(x:int)
assegna(x:double)

E' dovuto al fatto che non ho
ridefinito il metodo di A

Java

61

Attenzione - Equals

- Quando si ridefiniscono i metodi in java bisogna usare la stessa segnatura !!
- Vedi il problema con equals

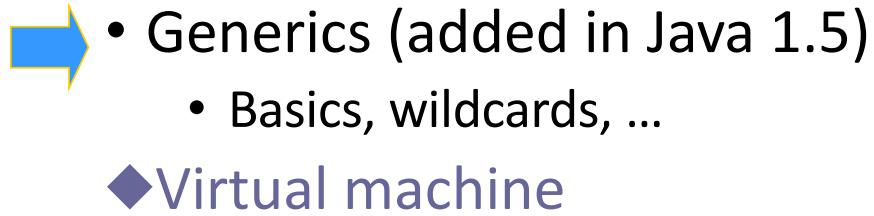
```
class A {  
    int x;  
    A(int y){x = y;}  
    public equals(A a){ return (x == a.x);}  
}  
  
Object a1 = new A(3);  
A a2 = new A(3);  
a1.equals(a2);
```

a2.equals(a1);

Esercizio, corretta implementazione di equals

Outline

- Objects in Java
 - Classes, encapsulation, inheritance
- Type system
 - Primitive types, interfaces, arrays, exceptions



◆ Virtual machine

- Loader, verifier, linker, interpreter
- Bytecodes for method lookup

◆ Security issues

Enhancements in JDK 5 (= Java 1.5)

- Enhanced for Loop
 - for iterating over collections and arrays
- Autoboxing/Unboxing
 - automatic conversion between primitive, wrapper types
- Typesafe Enums
 - enumerated types with arbitrary methods and fields
- Varargs
 - puts argument lists into an array; variable-length argument lists
- Static Import
 - avoid qualifying static members with class names
- Annotations (Metadata)
 - enables tools to generate code from annotations (JSR 175)
- Generics
 - polymorphism and compile-time type safety

varargs

- Varargs sono usati per dichiarare un metodo che possa prendere in ingresso un oggetto, n- oggetti o un array di oggetti.
- Esempio
- `print(String ... s)`
- Permette le seguenti chiamate:
- `print("pippo")`
- `print("pippo","pluto")`
- `print(new String[]{"a","b","c"})`
- Il tipo del parametro formale di un varargs è un array

Java Generic Programming

- Java has class Object
 - Supertype of all object types
 - This allows “subtype polymorphism”
 - Can apply operation on class T to any subclass S <: T
- Java 1.0 – 1.4 do not have templates
 - No parametric polymorphism
 - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
 - Can cast from supertype to subtype
 - Cast is checked at run time

Why no generics in early Java ?

- Many proposals
- Basic language goals seem clear
- Details take some effort to work out
 - Exact typing constraints
 - Implementation
 - Existing virtual machine?
 - Additional bytecodes?
 - Duplicate code for each instance?
 - Use same code (with casts) for all instances

Java Community proposal (JSR 14) incorporated into Java 1.5

Motivazione per l'introduzione dei generici

- **Programmazione generica**
- Se voglio realizzare programmi generici, cioè che vanno bene per diversi tipi, come posso fare?
- Posso usare scrivere gli algoritmi usando Object che a runtime potrà essere una qualsiasi sottoclasse
- Così era prima di 1.5
- Ad esempio una collezione generica

Esempio Lista di Object

(prima dei generici)

Ad esempio una lista

```
// creazione  
List myList = new LinkedList();  
  
// aggiungo  
myList.add(new Integer(0));  
  
// prendo il primo elemento  
Integer x = (Integer)  
    myList.iterator().next();
```

1. Il cast è necessario
2. posso inserire qualsiasi oggetto

Stack:

```
class Stack {  
    void push(Object o) {...}  
    Object pop() { ... }  
    ...}
```

```
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```

Come specializzare (senza generics)

- Posso specializzare mediante ereditarietà (senza usare i generics)
- Ad esempio se voglio una lista che prende solo gli interi

```
IntegerList extends ArrayList{  
    @Override  
    // devo mantenere la segnatura  
    boolean add(Object o){  
        // check o is Integer ...  
    }  
    @Override  
    //posso specializzare il return (cov.)  
    Integer get(int i){...}  
}
```

Sintassi dei generici

- Una versione generica della classe **Stack**
- Una classe generica è definita con il seguente formato:

```
class ClassName <T1 , T2 , ..., Tn > { ... }
```

- La sezione dei parametri di tipo, delimitati da parentesi angolari (<>), segue il nome della classe. Essa specifica i parametri di tipo (chiamati anche **variabili di tipo**) T1, T2, . . . e Tn.

Generics

Invece mediante i generici:

```
class Stack<A> {  
    void push(A a) { ... }  
    A pop() { ... }  
    ...  
}  
  
String s = "Hello";  
Stack<String> st = new Stack<String>();  
st.push(s);  
...  
s = st.pop();
```



Annoto con $\langle \rangle$ il TIPO Generico. A **non** è una classe

Declaring Generic classes

- For example a Coppia of two objects one of type E and the other of type F

```
class Coppia<E, F> {  
    E sinistro;  
    F destro;  
  
    Coppia(E a, F b) { ... }  
  
    E getSinistro() { return sinistro; }  
}
```

Metodi generici

- *Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors

```
public class Util {  
    public static <K,V> boolean compare(Pair<K,V> p1, Pair<K,V> p2)  
    {  
        return p1.getKey().equals(p2.getKey()) &&  
               p1.getValue().equals(p2.getValue());  
    }  
}
```

Bounded Type Parameters

Constraints on generic types

- One can introduce constraints over a type used as parameter in a generic class

< E **extends** T> : E must be a subtype of T

< E **super** T> : E must be a supertype of T

Esempio:

```
public <U extends Number> void inspect(U u){  
    System.out.println("U: " + u.getClass().getName());  
}
```

Metodi generici

- Analogamente a classi e interfacce generiche, in Java 5.0 è possibile definire metodi generici, ovvero parametrici rispetto ad uno o più tipi.

```
public class MaxGenerico {  
    public static <T extends Comparable<T>>  
        T max (Vector<T> elenco) {  
        ...  
    } }
```

- Nell'esempio:
 - la classe non ha parametri di tipo;
 - la dichiarazione di tipo è `<T extends Comparable<T>>`, immediatamente successiva ai modificatori;
 - il tipo del metodo è `T`;
 - la segnatura del metodo è `max(Vector<T>)`.

Java generics are type checked

- A generic class may use operations on objects of a parameter type
 - Example: `PriorityQueue<T> ... if x.less(y) then ...`
- Two possible solutions
 - C++: Link and see if all operations can be resolved
 - Java: Type check and compile generics w/o linking
 - This requires programmer to give information about type parameter
 - Example: `PriorityQueue<T extends ...> —`

Example: Hash Table

```
interface Hashable {  
    int hashCode();  
};  
  
class HashTable <Key extends Hashable, Value> {  
    void insert (Key k, Value v) {  
        int bucket = k.hashCode();  
        insertAt (bucket, k, v);  
    }  
    ...  
};
```

This expression must typecheck
Use “Key extends Hashable”

Interface Comparable<T>

- imposes a total ordering on the objects of each class that implements it (natural ordering)
- **int compareTo(T o):** comparison method
 - compares `this` object with `o` and returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`).
- Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

compareTo

- The natural ordering for a class C is said to be **consistent** with equals if and only if `(e1.compareTo((Object)e2) == 0)` has the same boolean value as `e1.equals((Object)e2)` for every e1 and e2 of class C.
- Altri vincoli:
 - $\text{sgn}(x.compareTo(y)) == -\text{sgn}(y.compareTo(x))$
 - the relation must be transitive:
 - $(x.compareTo(y)>0 \ \&\& \ y.compareTo(z)>0)$ implies $x.compareTo(z)>0$.
 - Finally, the implementer must ensure that $x.compareTo(y)==0$ implies that $\text{sgn}(x.compareTo(z)) == \text{sgn}(y.compareTo(z))$, for all z.

Example

Class MyClass implements

```
Comparable<MyClass>{  
    private int a;  
    ...  
    public int compareTo(MyClass  
other) {  
        return (this.a - other.a);  
    }  
}
```

Priority Queue Example

Generic types often requests the implementation of Comparable:

```
class PriorityQueue<T extends Comparable<T>> {  
    List<T> queue;    ...  
    void insert(T t) {  
        ... if (t.compareTo(queue.get(i))  
        ...  
    }  
    T remove() { ... }  
    ...  
}
```

No covarianza dei generics → conseguenze

- Nota che se S è sottotipo di T una classe P<S> non è sottotipo di P<T>
 - In questo modo non ho i problemi degli array
 - A è sottotipo di Object
 - Collection<A> non è Collection<Object>

Generics and Subtyping

- Questo è corretto?

```
1. List<String> ls = new ArrayList<String>();
```

```
2. List<Object> lo = ls;
```

- 1 sì (arrayList è un sottotipo di List).
- Ma 2? Una Lista di String è un sottotipo di una stringa di Object
- Attenzione, se fosse vero avrei ancora problemi simili a quelli degli array

```
lo.add(new Object()); // 3
```

```
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

- NON C'è covarianza dei generici
- A <: B non implica I<A> sottotipo di I !!

Generics e wildcard

- Vogliamo scrivere un metodo che prende una collezione e stampa tutti gli elementi:

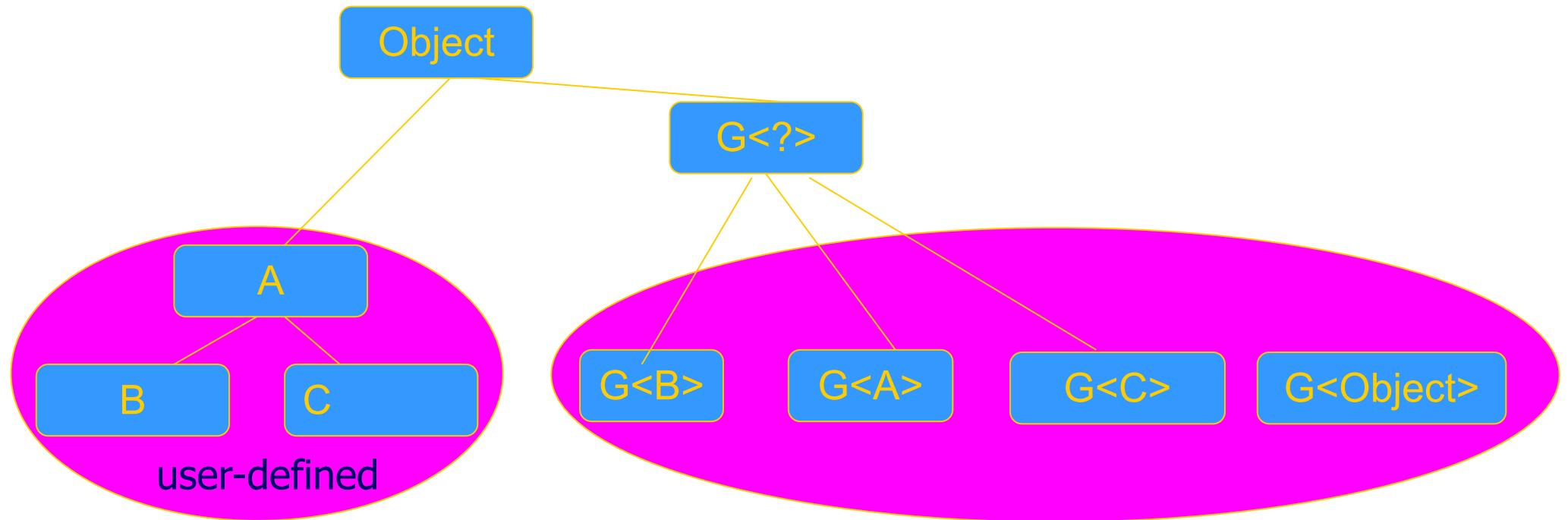
```
void printCollection(Collection c) {...}
```

- Con i generici???

```
void printCollection(Collection<Object> c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {  
        System.out.println(i.next().toString());  
    } }
```

- E se ho Collection<Student> non funziona !!!
- C'è un supertipo di Collection<Student>, Collection<...> ...??

Supertipo di generics



```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

Upper Bound su wildcard generics

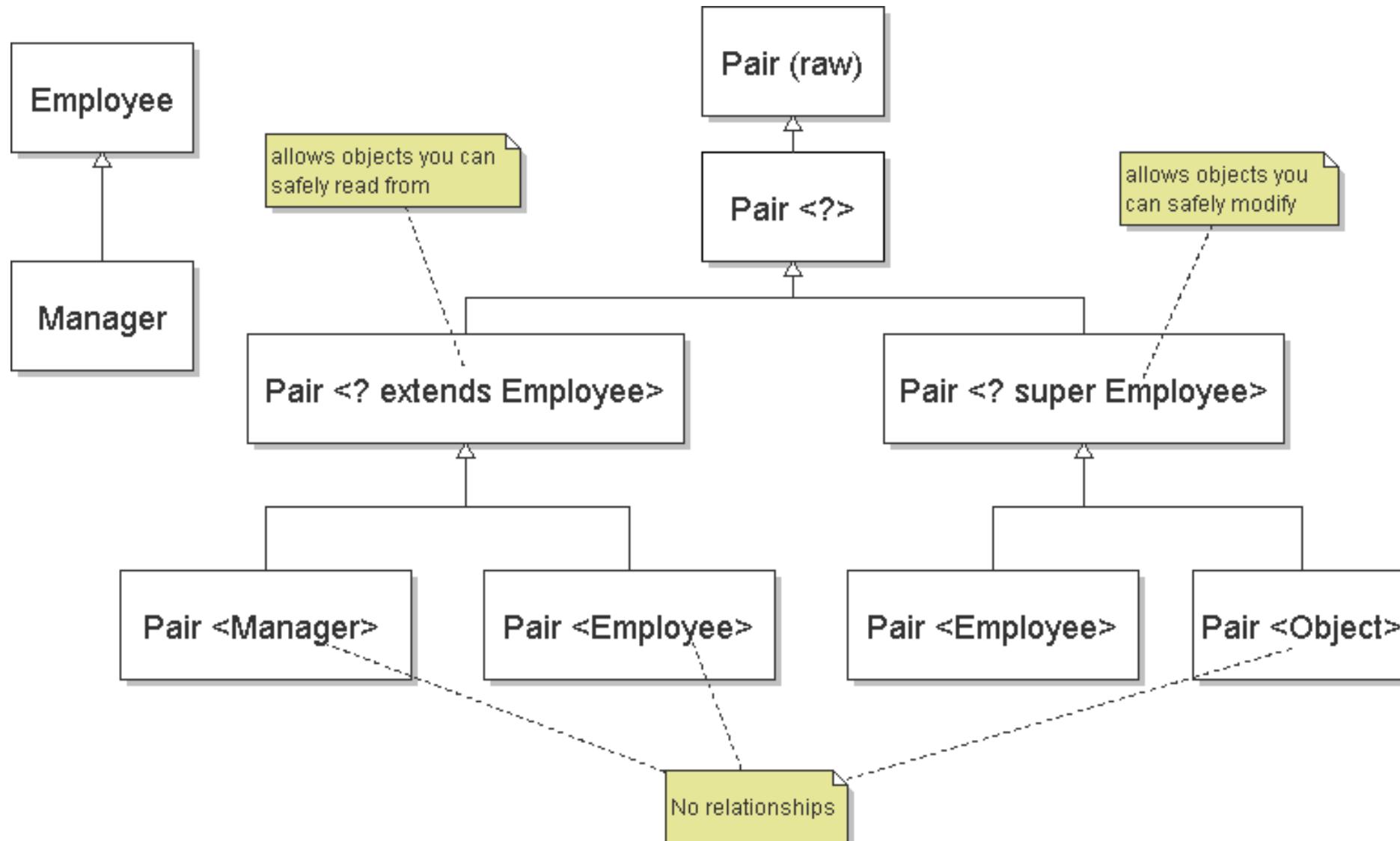
- Esempio:
Studente <: Persona, non ho che
List<Studente> <: List<Persona>

- Esempio:
 - stampaAnagrafica(List<Persona> p)
 - List<Studente> ls;
 - **stampAnagrafica(ls)** non compila
- Non posso però neanche definire:
 - stampaAnagrafica(List<?> p)
- Esiste un tipo intermedio: List<? extends Persona>

Wildcard e generics (Lower bound)

- Alcune volte non si vuole specificare esattamente il tipo ma si vuole essere più permissivi
- Persona **extends Comparable<Persona>**
- Studente **extends Persona**
- Studente non può essere sostituito a T in un generico che chiede **<T extends Comparable<T>**
 - Non potrei fare liste ordinate di studente
 - Però potrei utilizzare il compareTo di Persona, senza necessità di introdurne un altro compareTo nella sottoclassificazione
- Introduco: **<T extends Comparable<? super T>**

Inheritance rules for generic types



Comments on inheritance relations

- `Pair<Manager>` matches `Pair<? extends Employee>` => subtype relation (*covariant typing*)
- `Pair<Object>` matches `Pair<? super Employee>` => subtype relation (*contravariant typing*)
- `Pair<Employee>` can contain only *Employees*, but `Pair<Object>` may be *assigned* anything (*Numbers*) => *no subtype relation*
- also: `Pair<T> <= Pair<?> <= Pair (raw)`

```
List <String> sl = new LinkedList <String> ();
List x = sl;           // OK
x.add (new Integer (5)); // type safety warning
.
String str = sl.get (0); // throws ClassCast.
```

Implementing Generics

- Type erasure
 - Compile-time type checking uses generics
 - Compiler eliminates generics by erasing them
 - Compile List<T> to List, T to Object, insert casts
- “Generics are not templates”
 - Generic declarations are typechecked
 - Generics are compiled once and for all
 - No instantiation
 - No “code bloat”

More later when we talk about virtual machine ...

Esercizio

- Dichiara una classe A che ha come membro un intero
- Dichiara un classe B extends A che ha un metodo equals(B a)
- Dichiara una classe C extends A che ha un metodo equals(Object)
- Implementa i metodi `toString` in modo che stampino “A”, “B” e “C” e il valore dell'intero
- Dichiara una Lista di A usando i generici
- Inserisci qualche B e qualche C
- Stampa il contenuto della lista con un ciclo `for each`
- Domanda un intero x
 - `Scanner sc = new Scanner(System.in);`
 - `int x = sc.nextInt();`
- e cerca nella lista un elemento che sia `equals a new A(x)`
 - usa `for each` e `equals`
 - usa `contains` **QUALI PROBLEMI HAI???**

Auto boxing /unboxing

- Adds auto boxing/unboxing

User conversion

```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(new Integer(12));  
...  
int i = (st.pop()).intValue();
```

Automatic conversion

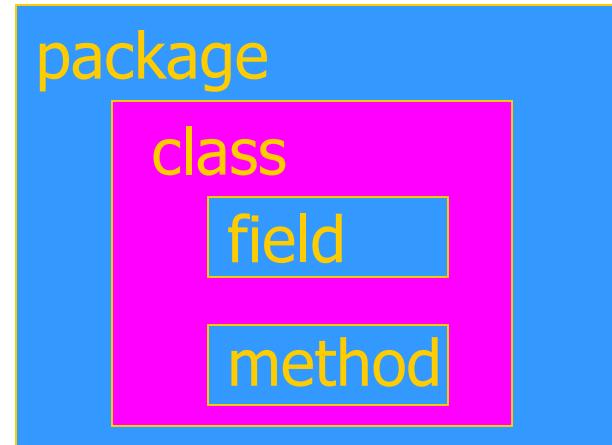
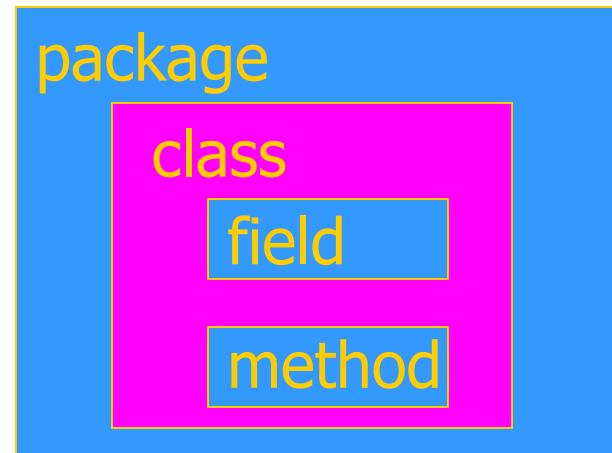
```
Stack<Integer> st =  
    new Stack<Integer>();  
st.push(12);  
...  
int i = st.pop();
```

Package e visibilità (18)

Packages and visibility

Encapsulation and packages

- Every field, method belongs to a class
- Every class is part of some package
 - Can be unnamed default package
 - File declares which package code belongs to



Visibility and access

- Four visibility distinctions
 - public, private, protected, package
- Method can refer to
 - private members of class it belongs to
 - non-private members of all classes in same package
 - protected members of superclasses (in diff package)
 - public members of classes in visible packages
Visibility determined by files system, etc. (outside language)
- Qualified names (or use import)
 - `java.lang.String.substring()`

Visibilità e overriding

- Quando si ridefinisce un metodo, questo non deve essere privato, altrimenti si fa overloading.

Esempio

```
class A {  
    private void m(String s)  
        /* ... */  
    }  
  
    void m(Object o) { /* ... */ }  
}  
  
class B extends A {  
    void m(String s) { /* ... */ }  
}
```

La classe B non ridefinisce m di A ma fa overloading:

```
Main in altra classe  
A a = new A();  
a.m("def"); ---  
m(object)  
A b = new B();  
b.m("def"); - idem
```

Overriding e visibilità

- Quando si ridefinisce, la visibilità può solo aumentare.
- Esempio:

```
public class A {  
    protected void m() { ... }  
    public static void main(String args[]) {  
        A a = new B();  
        a.m();  
    }  
}  
  
public class B extends A {  
    public void m() { ... }  
}
```

Overriding ed eccezioni

- Quando si esegue overriding di un metodo che dichiara di sollevare eccezioni C, il metodo ridefinito non può mai sollevare "più" tipi di eccezione (controllate) di quelli sollevati dall'originale. Può:
 - dichiarare a sua volta di sollevare eccezioni di classe C;
 - dichiarare di sollevare eccezioni di una sottoclasse di C;
 - dichiarare di non sollevare eccezioni.
- Non potrebbe, invece:
 - dichiarare di sollevare eccezioni di una superclasse di C o di una classe non legata a C da legami di ereditarietà.

Java Summary

- Objects
 - have fields and methods
 - alloc on heap, access by pointer, garbage collected
- Classes
 - Public, Private, Protected, Package (not exactly C++)
 - Can have static (class) members
 - Constructors and finalize methods
- Inheritance
 - Single inheritance
 - Final classes and methods

Java Summary (II)

- Subtyping
 - Determined from inheritance hierarchy
 - Class may implement multiple interfaces
- Virtual machine
 - Load bytecode for classes at run time
 - Verifier checks bytecode
 - Interpreter also makes run-time checks
 - type casts
 - array bounds
 - ...
 - Portability and security are main considerations

Some Highlights

- Dynamic lookup
 - Different bytecodes for by-class, by-interface
 - Search vtable + Bytecode rewriting or caching
- Subtyping
 - Interfaces instead of multiple inheritance
 - Awkward treatment of array subtyping (my opinion)
- Generics
 - Type checked, not instantiated, some limitations (<T>...new T)
- Bytecode-based JVM
 - Bytecode verifier
 - Security: security manager, stack inspection

Comparison with C++

- Almost everything is object + Simplicity - Efficiency
 - except for values from primitive types
- Type safe + Safety +/- Code complexity - Efficiency
 - Arrays are bounds checked
 - No pointer arithmetic, no unchecked type casts
 - Garbage collected
- Interpreted + Portability + Safety - Efficiency
 - Compiled to byte code: a generalized form of assembly language designed to interpret quickly.
 - Byte codes contain type information

Comparison

(cont'd)

- Objects accessed by ptr + Simplicity - Efficiency
 - No problems with direct manipulation of objects
- Garbage collection: + Safety + Simplicity - Efficiency
 - Needed to support type safety
- Built-in concurrency support + Portability
 - Used for concurrent garbage collection (avoid waiting?)
 - Concurrency control via synchronous methods
 - Part of network support: download data while executing
- Exceptions
 - As in C++, integral part of language design

Links

- **Enhancements in JDK 5**
 - <http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html>
- J2SE 5.0 in a Nutshell
 - <http://java.sun.com/developer/technicalArticles/releases/j2se15/>
- Generics
 - <http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm>

Introduction to C++

Informatica III – parte A
A. Gargantini

History

- C++ is an object-oriented extension of C [1985 from 1997 OO]
- C was designed by Dennis Ritchie at Bell Labs
 - used to write Unix
 - based on BCPL
- C++ designed by Bjarne Stroustrup at Bell Labs
 - His original interest at Bell was research on simulation
 - Early extensions to C are based primarily on Simula
 - Called “C with classes” in early 1980’s
 - Popularity increased in late 1980’s and early 1990’s
 - Features were added incrementally
- Classes, templates, exceptions, multiple inheritance, type tests...

Design Goals

- Provide object-oriented features in C-based language, without compromising efficiency
 - Backwards compatibility with C
 - Better static type checking
 - Data abstraction
 - Objects and classes
 - Prefer efficiency of compiled code where possible
- Important principle
 - If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature.

What is Data Abstraction?

- Abstract Data Types (ADTs)
 - type implementation & operations
 - hidden implementation
- types are central to problem solving
 - Not procedures like in C
- a weapon against complexity
- built-in and user-defined types are ADTs

How Well are ADTs Supported in C?

- Does C enforce the use of the ADTs interface and the hiding of its implementation?
- No

C++

- C++ is a superset of C, which has added features to support **object-oriented programming**
- C++ supports **classes**
 - things very like ADTs

How successful?

- Given the design goals and constraints,
 - this is a very well-designed language
- Many users -- tremendous popular success
- However, very complicated design
 - Many specific properties with complex behavior
 - Difficult to predict from basic principles
 - Most serious users chose subset of language
 - Full language is complex and unpredictable
 - Many implementation-dependent properties
 - Language for adventure game fans

Further evidence

- Many style guides for using C++ “safely”
- Every group has established some conventions and prohibitions among themselves.
 - don’t inherit implementation
 - SGI compiler group -- no virtual functions
 - Others

Significant constraints

- C has specific machine model
 - Access to underlying architecture
- No garbage collection
 - Consistent with goal of efficiency
 - Need to manage object memory explicitly
- Local variables stored in activation records
 - Objects treated as generalization of structs, so some objects may be allocated on stack
 - Stack/heap difference is visible to programmer

Overview of C++

- Additions and changes not related to objects
 - type bool
 - pass-by-reference & the Copy-Constructor
 - user-defined overloading
 - function template
 - exception handling
 - ...

OO Programming Languages

- Four main concepts:

1. Abstraction: implementation details hidden inside a program unit with a specific *interface*. The interface is a set of public functions (or methods) over hidden data.

2. Inheritance: reusing the definition of one kind of object to define another kind of object.

3. Dynamic lookup: a method is selected at run time, according to the *implementation* of the object, not some static property of the pointer/var used to name the object.

4. Subtyping is a relation on types that allows values (or objects) of one type to be used in place of values (or objects) of another.

Inheritance Is Not Subtyping!

*"Subtyping is a relation on interfaces,
inheritance is a relation on implementations."*

C++ Object System

- Object-oriented features
 - 1. Classes and Data Abstraction
 - 2. Encapsulation
 - 3. Inheritance
 - Single and multiple inheritance
 - Public and private base classes
 - 4. Objects, with dynamic lookup of virtual functions
 - 5. Subtyping
 - Tied to inheritance mechanism
 - A will be recognized by the compiler as a subtype of B only if B is a public base class of A

Objects in C++

Classes and Data Abstraction

Programmazione avanzata
Angelo Gargantini
AA 22/23



Classi vs strutture

Classi vs oggetti

C++ Object System

Object-oriented features

- Classes and Data Abstraction
- Encapsulation
- Inheritance
 - Single and multiple inheritance
 - Public and private base classes
- Objects, with dynamic lookup of virtual functions
- Subtyping
 - Tied to inheritance mechanism

C++ Object System

Object-oriented features

- **Classes and Data Abstraction**
- Encapsulation
- Inheritance
 - Single and multiple inheritance
 - Public and private base classes
- Objects, with dynamic lookup of virtual functions
- Subtyping
 - Tied to inheritance mechanism

Abstraction

- **Abstraction** means that implementation details are hidden inside a program unit with a *specific interface*.
- For objects, **the interface** consists of a set of public functions (or methods) that manipulate hidden data.
- Abstraction involves restricting access to a program component according to its specified interface.

C++: Classes and Data Abstraction

- C++ supports Object-Oriented Programming (OOP)
- OOP models real-world objects with software counterparts
- OOP encapsulates data (attributes) and functions (behavior) into packages called objects
- Objects have the property of information hiding

C++: Classes and Data Abstraction

Objects communicate with one another across **interfaces**

The interdependencies between the classes are identified

- makes use of
- a part of
- a specialisation of
- a generalisation of
- etc.

C and C++

- C programmers concentrate on writing functions
- C++ programmers concentrate on creating their own **user-defined types** called **classes**
- Classes in C++ are a natural evolution of the C notion of **struct**

Namespaces

A namespace is a declarative region that provides a scope to the identifiers inside it.

Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.

Namespaces

All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier, for example `std::vector<std::string> vec;`, or else by a using declaration for a single identifier (using `std::string`), or a using directive for all the identifiers in the namespace (using `namespace std;`).

Namespaces

The following example shows a namespace declaration

```
namespace ContosoData{  
    class ObjectManager { ....  
};  
    void func(ObjectManager) {}  
}
```

A User-Defined Type Time with a struct

```
// Create a structure, set its members, and print it
// structure definition

struct Time {

    int hour; // 0-23
    int minute; // 0-59
    int second; // 0-59

};

void printMilitary(const Time &); // prototype
void printStandard(const Time &); // prototype
```

```
main()
{
    Time dinnerTime; // variable of new type Time

    // set members to valid values
    dinnerTime.hour = 18;
    dinnerTime.minute = 30;
    dinnerTime.second = 0;

    cout << "Dinner will be held at";
    printMilitary(dinnerTime); // 18:30:00
    cout << " military time,\nwhich is ";
    printStandard(dinnerTime); // 6:30:00 PM
    cout << " standard time." << endl;
```



adds a newline ("\n") and flushes the buffer

```
// set members to invalid values
dinnerTime.hour = 29;
dinnerTime.minute = 73;
dinnerTime.second = 103;

cout << "\nTime with invalid values: ";
printMilitary(dinnerTime); // 29:73:103 bad values!
cout << endl;

return 0;
}// end main
```

```
// Print the time in military format
void printMilitary(const Time &t)
{
    cout << (t.hour < 10 ? "0" : "") << t.hour << ":"
    << (t.minute < 10 ? "0" : "") << t.minute << ":"
    << (t.second < 10 ? "0" : "") << t.second;
}

// Print the time in standard format
void printStandard(const Time &t)
{
    cout << ((t.hour == 0 || t.hour == 12) ? 12 : t.hour % 12)
        << ":" << (t.minute < 10 ? "0" : "") << t.minute
        << ":" << (t.second < 10 ? "0" : "") << t.second
        << (t.hour < 12 ? " AM" : " PM");
}
```

Comments

- Initialization is not required --> can cause problems
- A program can assign **bad** values to members of Time
- If the implementation of the **struct** is changed, all the programs that use the **struct** must be changed [No “interface”]

A Time Abstract Data Type with a Class

```
#include <iostream.h>
// Time abstract data type (ADT) definition
class Time {
public:
    Time() ;                                // default constructor
    void setTime(int, int, int);
    void printMilitary();
    void printStandard();
private:
    int hour;    // 0 - 23
    int minute;  // 0 - 59
    int second;  // 0 - 59
};
```

NOTA: può essere separato dalla dichiarazione di classe. In altro file .cpp

```
// Time constructor initializes each data member to zero.  
// No return value  
// Ensures all Time objects start in a consistent state.  
Time::Time () { hour = minute = second = 0; }  
  
// Set a new Time value using military time.  
// Perform validity checks on the data values.  
// Set invalid values to zero (consistent state)  
void Time::setTime(int h, int m, int s)  
{  
    hour = (h >= 0 && h < 24) ? h : 0;  
    minute = (m >= 0 && m < 60) ? m : 0;  
    second = (s >= 0 && s < 60) ? s : 0;  
}
```

```
// Print Time in military format
void Time::printMilitary()
{
    cout << (hour < 10 ? "0" : "") << hour << ":"
        << (minute < 10 ? "0" : "") << minute << ":"
        << (second < 10 ? "0" : "") << second;
}
```

```
// Print time in standard format
void Time::printStandard()
{
    cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second
        << (hour < 12 ? " AM" : " PM");
}
```

```
// Driver to test simple class Time  
  
main()  
{  
    Time t; // instantiate object t of class Time  
  
    cout << "The initial military time is ";  
    t.printMilitary(); // 00:00:00  
    cout << "\nThe initial standard time is ";  
    t.printStandard(); // 12:00:00 AM  
  
    t.setTime(13, 27, 6);  
    cout << "\n\nMilitary time after setTime is ";  
    t.printMilitary(); // 13:27:06  
    cout << "\nStandard time after setTime is ";  
    t.printStandard(); // 1:27:06 PM
```

```
t.setTime(99, 99, 99);
// attempt invalid settings
cout << "\n\nAfter attempting invalid settings:\n"
    << "Military time: ";
t.printMilitary(); // 00:00:00
cout << "\nStandard time: ";
t.printStandard(); // 12:00:00 AM
cout << endl;

return 0;
} // end main
```

Output

- The initial military time is 00:00:00
- The initial standard time is 12:00:00 AM
- Military time after setTime is 13:27:06
- Standard time after setTime is 1:27:06 PM
- After attempting invalid settings:
- Military time: 00:00:00
- Standard time: 12:00:00 AM

Comments

- `hour`, `minute`, and `second` are `private` data members. They are normally `not` accessible outside the class. [Information Hiding]
- Use a `constructor` to initialize the data members. This ensures that the object is in a consistent state when created.
- Outside functions set the values of data members by calling the `setTime` method, which provides `error checking`.

Classes as User-Defined Types

- Once the class has been defined, it can be used as a type in declarations as follows:

L'intero oggetto Time è allocato sullo stack

L'intero array di 5 Time è allocato sullo stack

```
Time sunset;           //object of type Time  
Time arrayOfTimes[5]; //array of Time objects  
Time *pointerToTime; //pointer to a Time object
```

Solo il puntatore è sullo stack – devo creare l'oggetto

Using Constructors

- Constructors can be overloaded, providing several methods to initialize a class.

Interface

```
Time();           // default constructor  
Time(int hr);  
Time(int hr, int min, int sec);
```

Implementation

```
Time::Time() { hour = minute = second = 0; }  
Time::Time(int hr) { setTime(hr, 0, 0); }  
Time::Time(int hr, int min, int sec)  
{ setTime(hr, min, sec); }
```

Using Constructors for «static» object (sullo stack)

```
Time t1; // Time() is invoked  
Time t1(); //ERROR, intended as a funct prototype  
  
Time t2(08); // class_name object_name(values)  
Time t2 = Time(08);  
Time t2 = 08;  
Time t2 = (Time) 08; // cast  
  
Time t3(08,15,04);  
Time t3 = Time(08,15,04);
```

Using Constructors and dynamic objects

```
Type_name * pointer_name;  
pointer_name = new Type_name;
```

where **Type** is a Class or a primitive type

```
int *ptr;  
ptr = new int;
```

```
Time *t;  
t = new Time;           // Time() is invoked  
t = new Time(08);       // Time(int) is invoked  
t = new Time(08,15,04); // Time(int, int, int)  
                      // is invoked
```

Using Constructors and array of objects

```
Time arrayOfTimes[5]; //Time() is invoked
```

Explicit array initialization:

```
//Only the first four elements are initialized  
//Time() (if any) is invoked for the other elements  
Time arrayOfTimes[8] = { 3, Time(05), Time(),  
Time(01,12,03)}
```

Using Constructors and dynamic arrays

```
Time *t = new Time[8];  
// Time() is invoked for each element
```

```
int i = 3;  
Time (*t) [20] = new Time[3*i] [20];  
// Multi-dimension array  
// Time() is invoked for each element
```

positive, can be variable positive, constant

The diagram consists of two green arrows. One arrow points from the text 'positive, can be variable' to the expression '3*i'. Another arrow points from the text 'positive, constant' to the expression '3*i'.

In both cases, explicit initialization is not allowed!

The constructor initializer list

- A list of “constructor calls” that appears only **in the definition of the constructor** – after the argument list
- The initialization in the list is executed before any of the main constructor code.
- This is the place to put all **const** initializations, primitive type variables and object variables, **except arrays**.

```
class Info
private:
    const int i;
    double m;
    Time t;
Public:
    Info(); // default constructor
};
```

```
Info::Info(int j, double n) : i(j), m(n), t(i) {}
```

Copy constructors

An object can be built starting from an existing one:

`S u (s); // Calls S's copy - constructor`

`S v = s; // Calls S's copy – constructor`

The copy constructor can be defined:

```
class S { public: S(const S&); };
```

In case it is not defined, the compiler will add a default that copies all the fields (even the pointers)

Destructors (1)

- To guarantee cleanup when using dynamic memory
- Destroy objects by
 - Calling the destructors of object member variables
 - Calling superclass destructors (if virtual)
- The destructor is called
 - At the end of object lifetime
 - Or during a call to delete
- Normally there is no need to call the destructor explicitly



Destructors (2)

- A public function member `~class_name` with no parameters and no return values

```
Class_name::~class_name() {  
    //delete operations  
    ...  
}
```

- Operator `delete`
- can be called only for an object created by `new`

```
delete ptr;  
delete [] ptr; //se è un array
```

Attenzione a non fare delete di puntatori a zone sullo stack

new() and delete() (1)

- For each new statement, you must provide exactly one corresponding delete statement
- Failing to do so causes memory and resource leaks and can cause undefined behavior ...

new() and delete() (2)

- Allocating memory

```
int* myInt = new int;
```

```
int* myIntArray = new int[10];
```

- Deallocating memory

```
delete myInt;
```

```
delete[] myIntArray;
```

Deleting zero pointers

**If the pointer you're deleting is zero,
nothing will happen.**

**For this reason, people often recommend setting
a pointer to zero immediately after you delete it,
to prevent deleting it twice.**

```
delete p;  
p = nullptr;
```

**Deleting an object more than once
is definitely a bad thing to do,
and will cause problems.**

Function Declaration

- A function is declared by

```
returnType funcName(  
    typename arg1, ...,  
    typename argN)
```

- Member function can include a const modifier in their signature

```
void helloWorld::sayHello(void) const
```

- A const method cannot modify class members
- private/protected/public modifier are not part of the function declaration

Function declaration: *Const* modifier

```
#include <iostream.h>
Class Car{
    private:
        int lenght;
        double weight;
    public:
        int fun_weight(double) const;
};

int Car::fun_weight(double new_weight) const
{
    // weight++; ERROR
    new_weight += weight;
    return (int) new_weight;
}
```

Function Declaration Examples

```
void output(const std::string& s);
```

```
double multiply(const double fac1,      pass-by-value  
                const double fac2);      pass-by-reference *
```

```
void addHeader(void* buf, const Date& date);
```

```
int main(int argc, char* argv[]);      pass-by-reference &  
int main(int argc, char** argv);
```

```
void doSomething(SomeBigObject bo);
```

```
void doSomething(SomeBigObject* bo);
```

```
void doSomething(SomeBigObject& bo);
```

Call by value

- Called function has its own local copy of the data
- Changes to the data are local and
- Will be discarded as soon as the namespace is left
- (highly) inefficient with large objects

Call by reference

- Passes memory address of variables to the function (word-size variable)
 - Very efficient
 - Allows variable modification avoiding double copy
- Two possible realizations in C++
 - `void doSomething(Data* data);`
 - Pointer based
 - Advantages and drawbacks of pointer approach
 - `void doSomething(Data& data);`
 - Reference based
 - No null-check necessary

pass-by-reference &

Object Variable Classification (like in C)

■ Extern variables **double x**

- global variables, the prefix *extern* when declared by other files

■ Static global variables **static double x**

- global variables, but can't be used by other files
- are zero-initialized by default

■ Automatic internal variables

- defined within a function/block

■ Static internal variables

- like static external variables,
- but defined within a function/block
- retains its state between calls to that function

```
int count_calls()
{ static int
calls=0;
//local static
return ++calls; }
```

Extern variables

- file1.c: declares an external global var

```
int GlobalVariable;  
// implicit definition  
  
void SomeFunction();  
// function prototype (decl.)  
  
int main() {  
    GlobalVariable = 1;  
    SomeFunction();  
    return 0;  
}
```

- file2.c uses the variable

```
extern int GlobalVariable;  
// explicit declaration
```

```
void SomeFunction() {  
// function header (definition)  
    ++GlobalVariable;  
}
```

Static member variables

- A *static variable*, member of a class, is a variable **shared by all objects** created from the class

```
Class Car{  
    private:  
        static int num_cars;  
    public:  
        ...  
};  
//Outside initialized, like an external variable,  
//even if private!  
int Car::num_cars = 22;
```

Static member functions (1)

- Executed in the same manner for all objects of the given class, e.g., to open a file or to set *static variables*.
- They can't:
 - access to non static variables,
 - invoke non static functions,
 - use the pointer *this*
 - be declared *virtual*
 - Constructors and destructors can't be *static*

Static member functions (2)

```
#include <iostream.h>
class Car{
    private:
        static int num_cars;
    public:
        Car(); // default constructor
        static void n_car();
};
```

Static member functions (3)

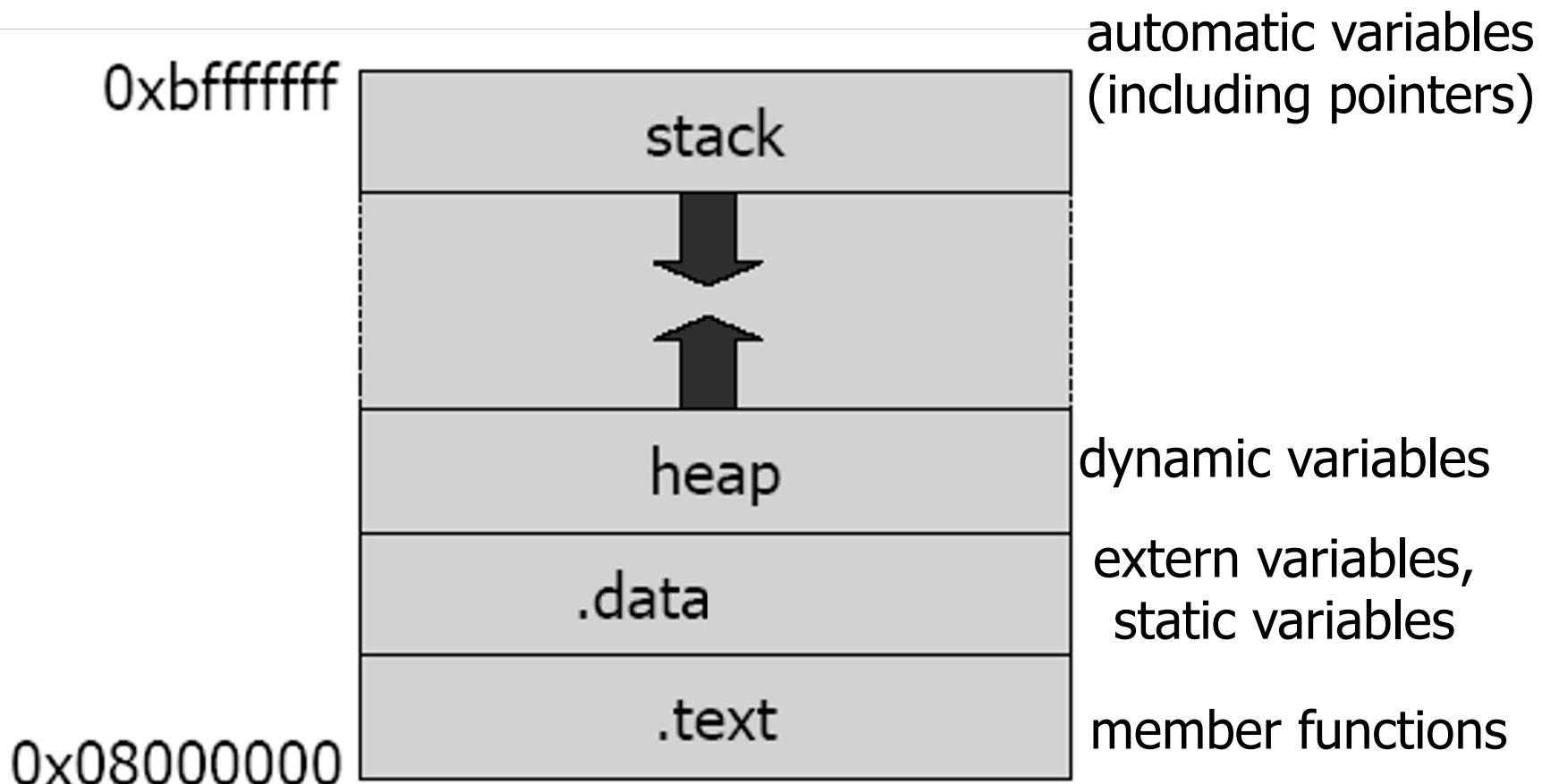
```
Car::Car() { num_cars++; }

void Car::n_car(){cout << num_cars << '\n';}

// Access to the static private variable is allowed!
int Car::num_cars = 0;

int main(int argc, char *argv[])
{
//cout << Car::num_cars;  ERROR Access to a
//private variable!
Car a;
Car::n_car(); // or a.n_car() bad style!
return 0;
}
```

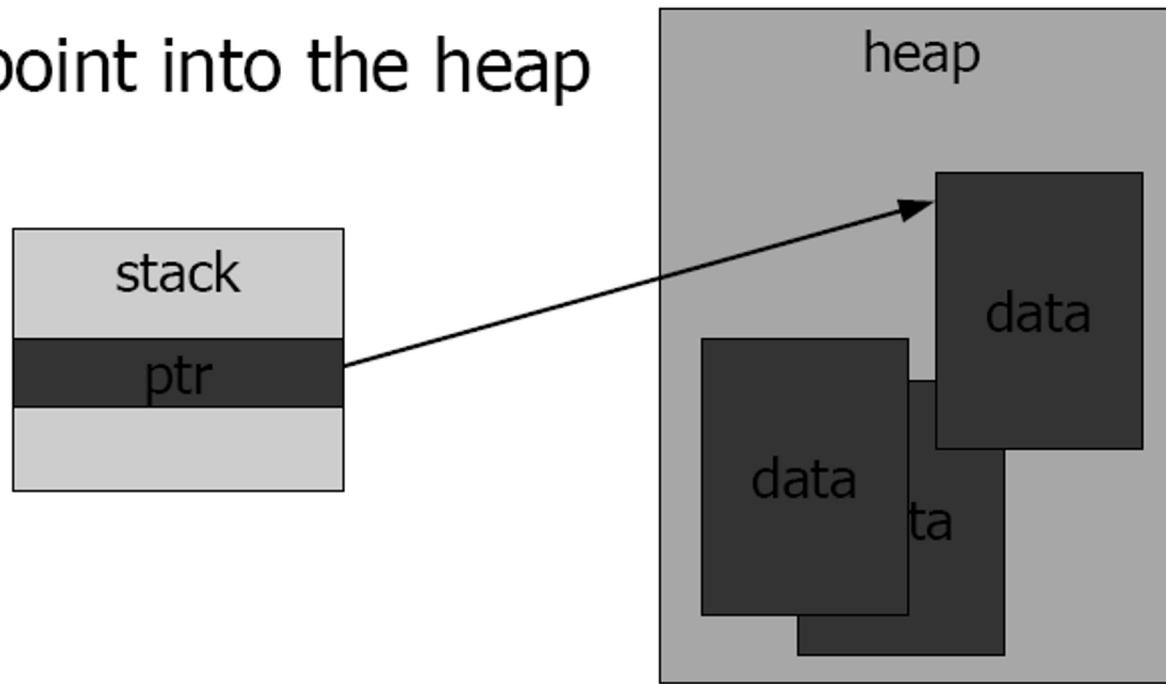
Memory layout (1)



Memory layout (2)

Pointer have a constant size of 1 word (16, 32, 64 bit)

- reside on the stack
- point into the heap



Inline functions

- Any function defined within a class body is automatically inline, but you can also make a non-class function inline by preceding it with the **inline** keyword.

```
inline int plusOne(int x) { return ++x; }  
inline int plusOne(int x); //has no effect
```

- Any behavior you expect from an ordinary function, you get from an inline function.
- The only difference is that an inline function is **expanded in place**, like a preprocessor macro in C, so the overhead of the function call is eliminated.

Inline + ricorsione

- Anche le funzioni ricorsive possono essere dichiarate inline, in quel caso verranno sviluppate solo fino ad un certo grado impostabile come opzione in gnu cpp ad esempio
- Le funzioni definite nel .h sono inline (e spesso viceversa)

Default arguments

- When functions have long argument lists, it is tedious to write (and confusing to read) the function calls
 - when most of the arguments are the same for all the calls.
-
- A commonly used feature in C++ is called *default arguments*.
 - A *default argument* is one the compiler inserts if it isn't specified in the function call.

```
void f(int size, int initQuantity = 0);  
void g(int x, int = 0, float = 1.1);  
void h(int = 0, int x, float = 1.1); //ERROR
```

Function overloading

```
void f(int size, int initQuantity);  
void f(int size, double initQuantity);  
int f(int size, int initQuantity);//ERROR
```

- The compiler resolves the correct version of an overloaded function based on the number/type of arguments in each call
- Functions differing only in their return type cannot be overloaded.
- Since the returned value may be implicitly converted, the compiler cannot resolve which version is intended to use
- An immediately useful place for overloading is in constructors.
- In C invece non si può!!!

Objects in C++

Objects, with dynamic lookup of virtual functions

C++ Object System

- Object-oriented features

- 1. Classes and Data Abstraction

- 2. Encapsulation

- 3. Inheritance

- 1. Single and multiple inheritance

- 2. Public and private base classes

- 4. Objects, with dynamic lookup of virtual functions

- 5. Subtyping

- 1. Tied to inheritance mechanism

Polymorphism in C++

- Runtime polymorphism
- Virtual functions
- Compile-time polymorfism
 - (parametric polymorfism)
- Generic programming
- templates

Run-time Polymorphism

- **Run-time polymorphism:** implemented with **dynamic lookup of virtual functions**
- ***Dynamic lookup:*** a method is selected dynamically, at run time, according to the implementation of the object that receives a message
 - not some static property of the pointer or variable used to name the object
- The important property of dynamic lookup is that **different objects may implement the same operation differently**

Virtual functions

- Member functions are either
 - Virtual, if explicitly declared or inherited as virtual
 - Non-virtual otherwise
- Non-virtual functions
 - Are called in the usual way. *Just ordinary functions.*
 - May be redefined in derived classes (overloading through *redefining*)
- Pay overhead only if you use virtual functions

Virtual members

- Must be explicitly declared as “virtual”
- May be *overridden* in derived (sub) classes
- Dynamic binding is activated
- Are accessed by indirection through **ptr** in object
- Explicitly as pointers or using references

```
class A { public: virtual void vi(){...}};  
class B : public A{ public: virtual void vi(){ ...}};  
int main() {  
    A* pa = new A; a -> vi(); // VIRTUAL CALL  
    A& ra = b; ra.vi(); // VIRTUAL CALL  
    A a = b; a.vi(); // NON VIRTUAL CALL  
}
```

Sample class: one-dimen. points

```
class Pt {  
public:  
    Pt(int xv);  
    Pt(Pt* pv);  
    int getX();  
    virtual void move(int dx);  
protected:  
    void setX(int xv);  
private:  
    int x;  
};
```

}

Overloaded constructor

Public read access to private data

Virtual function

Protected write access

Private member data

Sample derived class

```
class ColorPt: public Pt {  
public:  
    ColorPt(int xv,int cv);  
    ColorPt(Pt* pv,int cv);  
    ColorPt(ColorPt* cp);  
    int getColor();  
    virtual void move(int dx);  
    virtual void darken(int tint);  
protected:  
    void setColor(int cv);  
private:  
    int color;  
};
```

Overloaded constructor

Non-virtual function

Virtual functions

Protected write access

Private member data

Sample derived class

```
/* ----Definitions of Member Functions -----*/
```

```
void ColorPt::darker(int tint) { color += tint; }
```

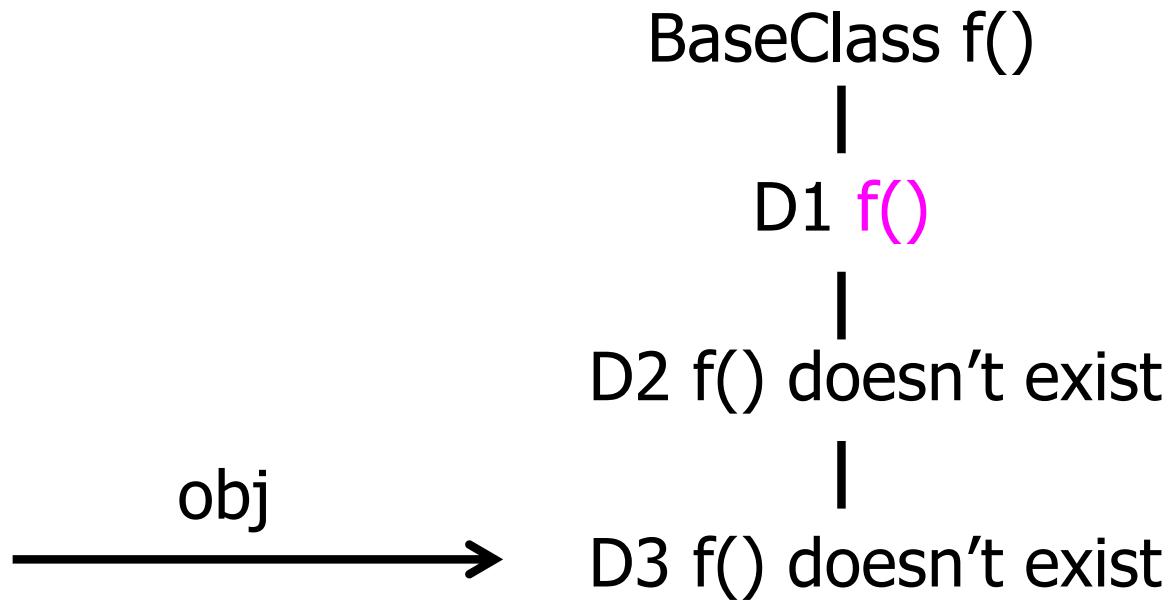
```
void ColorPt::move(int dx) {  
    Pt::move(dx); this->darker(1);  
}
```

Virtual functions and *indirection* (1)

- C++ allows a base class pointer to point to a (public) derived class object
- Upon method invocation, the method of the derived object is called (**dynamic binding**)
- This leads to generic algorithms **using base class pointers**

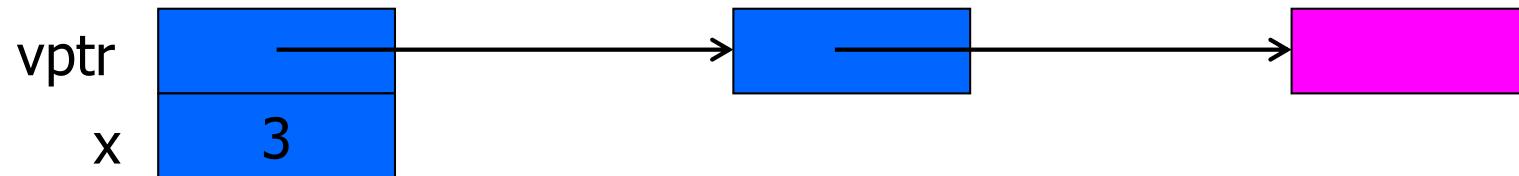
```
Pt* ptr = new ColorPt;  
ptr->move();
```

Virtual functions and *indirection* (2)

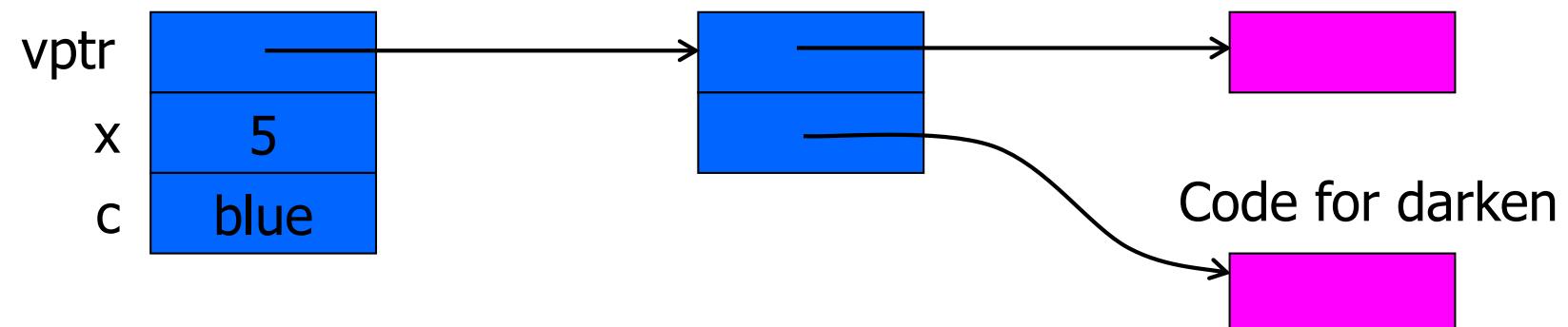


Run-time representation

Point object Point vtable Code for move



ColorPoint object ColorPoint vtable Code for move



Virtual pointers

Virtual tables

Function code

“this” pointer

- Code is compiled so that member function takes “object itself” as first argument

Code

```
int A::f(int x) { ... g(i) ...; }
```

compiled as

```
int A::f(A *this, int x) { ... this->g(i) ...; }
```

- “this” pointer may be used in member function
- Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Constructors/destructors and inheritance (2)

- destructors
 - always make destructors virtual in base classes
 - there might be cleanup work to be done in derived classes

```
class Employee {  
    //...  
public:  
    //...  
    virtual ~Employee() {}  
};
```

Non-virtual functions

- How is code for non-virtual function found?
- Same way as ordinary “non-member” functions:
- Compiler generates function code and assigns address
- Address of code is placed in **symbol table**
- At call site, address is taken from symbol table and placed in compiled code
- *But* some special scoping rules for classes
- Overloading
 - Remember: overloading is resolved at compile time
 - This is different from run-time lookup of virtual function

Overload

- An **overloaded** function is a function that shares its name with one or more other functions, but which has a different parameter list. The compiler chooses which function is desired based upon the arguments used.

Overridden

- An **overridden** function is a method in a descendant class that has a different definition than a **virtual** function in an ancestor class. The compiler chooses which function is desired based upon the type of the object being used to call the function.
 - Regardless the access modifier (private and so on) of the function
 - Si può fare overriding anche di metodi private (se virtual)
 - Not like Java
 - Vediamo un esempio

•**redefined**

- A **redefined** function is a method in a descendant class that has a different definition than a non-virtual function in an ancestor class. Don't do this. Since the method is not virtual, the compiler chooses which function to call based upon the static type of the object reference rather than the actual type of the object.

Virtual vs redefined Functions

```
class parent { public:  
    void printclass() {printf("p ");}
    virtual void printvirtual() {printf("p ")};  };  
  
class child : public parent { public:  
    void printclass() {printf("c ");}
    virtual void printvirtual() {printf("c ")};  };  
  
main() {
    parent p;  child c; parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}  
}
```

Output: p p c c p p p c

Esercizio

- Definiamo una classe A con un metodo virtual che ridefiniamo (overriding) in una sottoclasse B.
- Proviamo a chiamare quel metodo in diversi casi

Function call binding

- Early binding (C,C++)
 - At compile time
- Late binding (C++)
 - At runtime
- Mighty. But less efficient
- 1 more assembler statement per call
- Slight memory consumption due to the VPTRs

Objects in C++

Subtyping

C++ Object System

- Object-oriented features

- 1. **Classes and Data Abstraction**

- 2. **Encapsulation**

- 3. **Inheritance**

- Single and multiple inheritance

- Public and private base classes

- 4. **Objects, with dynamic lookup of virtual functions**

- 5. **Subtyping**

- Tied to inheritance mechanism

Subtyping (1)

- **Subtyping** is a relation on types that allows values of one type to be used in place of values of another.
- If some object **a** has all of the functionality of another object **b**, then we may use **a** in any context expecting **b**.
- **Inheritance Is Not Subtyping**
- "*Subtyping is a relation on interfaces, inheritance is a relation on implementations.*"
- **A typical example is C++**, in which
- A class A will be recognized by the compiler as a **subtype of B** only if B is a public base class of A

Subtyping (2)

- (A<:B = A subtype of B)
- Subtyping in principle
- A <: B if every A object can be used without type error whenever a B object is required

Pt:	int getX(); void move(int);	Public members
ColorPt:	int getX(); int getColor(); void move(int); void darken(int tint);	

- C++: A <: B if class A has public base class B

Sample public derived class

```
class ColorPt: public Pt {  
public:  
    ColorPt(int xv,int cv);  
    ColorPt(Pt* pv,int cv);  
    ColorPt(ColorPt* cp);  
    int getColor();  
    virtual void move(int dx);  
    virtual void darken(int tint);  
protected:  
    void setColor(int cv);  
private:  
    int color;  
};
```

In C++: public base class gives supertype!

} Overloaded constructor

Non-virtual function

} Virtual functions

Protected write access

Private member data

Public inheritance and subtyping

```
class ColorPt: public Pt {  
    ...  
};
```

ColorPt is a subtype of Pt.

I can write

```
Pt * p = new ColorPt;
```

```
// not so good  
ColorPt cpt;  
Pt p = cpt;
```

private derived class are not subtypes

```
class ColorPt: private Pt {  
    ....  
};  
ColorPt is not a subtype of Pt.  
} }  
I cannot write
```

```
Pt * p = new ColorPt;
```

```
ColorPt cpt;  
Pt p = cpt;
```

Independent classes not subtypes

```
class Point {  
public:  
    int getX();  
    void move(int);  
    ...  
};
```

```
class ColorPoint {  
public:  
    int getX();  
    void move(int);  
    int getColor();  
    void darken(int);  
    ...  
};
```

- C++ does not treat `ColorPoint <: Point` as written
- Need public inheritance `ColorPoint : public Pt`
- Subtyping based on inheritance:
 - An efficiency issue
 - An encapsulation issue: preservation under modifications to base class
... inheritance breaks encapsulation
 - *We will see "duck subtyping"*

Why C++ design?

- Client code depends only on public interface
 - In principle, if ColorPt interface contains Pt interface, then any client could use ColorPt in place of point
- However -- offset in virtual function table may differ
- Lose implementation efficiency
- Without link to inheritance
 - subtyping leads to loss of implementation efficiency
- Also encapsulation issue:
 - Subtyping based on inheritance is preserved under modifications to base class ...

Sottotipazione e covarianza. Riassunto critico

Rivediamo il concetto di sottotipazione per funzioni (metodi)

http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

Sottotipazione in generale

- $A <: B$: A sottotipo di B
- Subtyping principle
- $A <: B$ se un'espressione A può essere usata in modo sicuro in ogni contesto in cui sarebbe richiesto un B
- In questo modo vale il principio di sostituibilità
- Per variabili (istanze di classe):
 - `B b = new A()` (java)
 - `B* B = new A;` (C++)
- Dove mi aspetto un B posso passare un A.

Sottotipazione per funzioni

- Per le funzioni?
- Posso estendere e dire che una funzione è sottotipo di un'altra se può essere usata al suo posto.
- In teoria potrei ammettere l'overriding e il binding dinamico di tutte le funzioni che siano sottotipo
- Quando una funzione $f: W \rightarrow Z$ è sottotipo di un'altra?
- $f:W \rightarrow Z$ vuol dire che prende un W e restituisce un Z

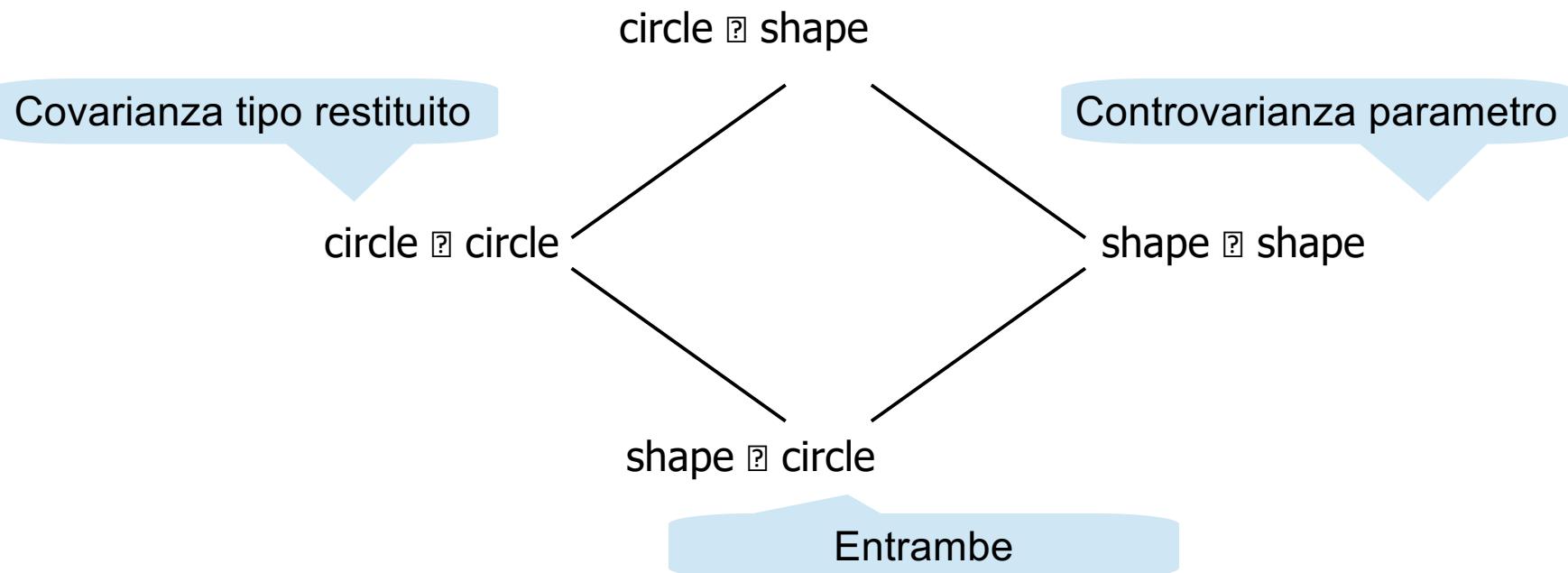
Covarianza tipo restituito

Rispetto ai parametri?

- Rispetto ai suoi argomenti (controvarianza)
- If $A <: B$, then $fb:B \rightarrow C <: fa:A \rightarrow C$
- Cioè fb può essere usata al posto di fa se prende come argomento un supertipo (B invece che A)
- Uso
- $fa(x)$ [x di tipo A] può essere sostituita da $fb(x)$
- Una sottoclasse quindi potrebbe “sostituire” fa con fb
- Terminology
- Contravariance: $A <: B$ implies $F(B) <: F(A)$

Esempio

- Se $\text{circle} \ll \text{shape}$, then



Nei linguaggi di programmazione

- In pratica i linguaggi di programmazione permettono la “sostituzione”- overriding solo in alcuni casi:
- C++: covarianza tipo ritornato in virtual functions
- Java: covarianza tipo ritornato (da Java 5)
- I parametri devono avere lo stesso tipo (invarianza) altimenti ho overloading
- ... altri linguaggi ...

In C++ - from 1998

- C++ supports the covariance of return types
- Only virtual
- Only pointers
- Example

```
class A{
public:
    virtual A * create() ...
};

class B : public A{
public:
    virtual B * create() ... // overriding
};
```

Subtyping with functions

```
class Point {  
    public:  
        int getX();  
        virtual Point *move(int);  
    protected: ...  
    private: ...  
};
```

```
class ColorPoint: public Point {  
    public:           Inherited, but repeated  
        int getX();   here for clarity  
        int getColor();  
        ColorPoint * move(int);  
    void darken(int);  
    protected: ...  
    private: ...  
};
```

- In principle: can have $\text{ColorPoint} <: \text{Point}$
- In practice: some compilers allow, others have not
This is covariant case; contravariance is another story

In Java

- Covarianza del tipo restituito, già visto

Slicing - attenzione

```
class A {  
    int foo;  
};  
class B : public A  
{
```

```
    int bar;  
};
```

- So an object of type B has two data members, foo and bar

- Polimorfism does not work without pointers, but copy constructor:
 - B b;
 - A a = b
- a will have only the foo attribute ! The member bar of b is lost

Details, details

- This is legal

```
class Point { ...  
    virtual Point * move(int);  
... }  
  
class ColorPoint: public Point { ...  
    virtual ColorPoint * move(int);  
... }
```

- But not legal if '*'s are removed

```
class Point { ... virtual Point move(int); ... }  
class ColorPoint: public Point { ...virtual ColorPoint move(int);... }
```

Related to subtyping distinctions for object L-values and object R-values
(Non-pointer return type is treated like an L-value for some reason)

Abstract Classes

- Abstract class:

- A class that has at least one *pure virtual member function*, i.e a function with an empty implementation

- Declare by: **virtual function_decl = 0;**

- A class without complete implementation

- Useful because it can have derived classes

Since subtyping follows inheritance in C++, use abstract classes to build subtype hierarchies.

- Establishes layout of virtual function table (vtable)

- Example

- Geometry classes

- Shape is abstract supertype of circle, rectangle, ...

C++ Summary

- Objects
- Created by classes
- Contain member data and pointer to class
- Encapsulation
 - member can be declared public, private, protected
 - object initialization partly enforced
- Classes: virtual function table
- Inheritance
 - Public and private base classes, multiple inheritance
 - Subtyping: Occurs with public base classes only

Duck typing

- deriva dal detto "Se cammina come un'anatra e fa starnazzare come un'anatra, allora deve essere un'anatra".
- Il duck typing è legato alla tipizzazione dinamica
- il tipo o la classe di un oggetto è meno importante dei metodi che definisce.
- Python usa il duck typing, non controlla i tipi (che non sono dichiarati).
- Invece, controlla la presenza di un determinato metodo o attributo.

Esempio



GENERIC ABSTRACTIONS in C++

- C++ Templates
- STL (Standard Template Library)

9.4 Programming Languages Concepts

by John Mitchell

Overview

- Motivation
- Template review
 - Function template
 - Class template
- What is the STL?
 - Containers
 - Iterators
 - Algorithms
- Glossed-over stuff

Motivation

- Abstract data types such as stacks or queues are useful for storing many kinds of data
- It is time consuming to write different versions of stacks for different types of elements
- Most typed languages support some form of **type parameterization**
- The **C++ template** is the most familiar type-parameterization mechanism
- The **C++ STL** is a large program library of parameterized abstract data types

C++ Function Template (1)

- A simple swap function:

```
void swap(int& x, int& y) {  
    int tmp=x; x=y; y=tmp; }
```

- A function template with a type variable **T** in place of **int**:

```
template<class T>  
void swap(T& x, T& y) {  
    T tmp=x; x=y; y= tmp; }
```

nota:

template<typename T> = **template<class T>**

C++ Function Template (2)

- Function templates are instantiated automatically by the program linker using the types of the function arguments

```
int i,j;  
...  
swap(i,j); // Use swap with T replaced by int  
string s,t;  
...  
swap(s,t); // Use swap with T replaced by String  
float a;  
...  
swap(i,a); // ERROR
```

C++ Function Template (3)

- For each type variable, at least one function argument must depend on the type variable
 - `template<class T> T f(T &); //OK`
 - `template<class T> T f(double); //ERROR`
 - `template<class T> T f(double, T&); //OK`
 - `template<class T, class S> T f(T &, S &); //OK`
 - `template<class T, class S> T f(S &); //ERROR`

C++ Function Template (4)

- Operations on Type Parameters limit the variability of the parameters
- A generic sort function:

```
template <class T>
void sort( int count, T * A[count] ) {
    for (int i=0; i< count-1; i++)
        for (int j=i+1; j< count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- If A is an array of type **T**, then **sort(n, A)** will work only if operator **<** (possibly *overloaded*) is defined on type **T**

Esercizio

- Definiamo la funzione max tra due elementi generici.

C++ Class Template

```
template <class T> class Complex {  
    private:  
        T re,im;  
    public:  
        Complex (const T& r, const T& I)  
            :re(r),im(i) {}  
        T getRe() {return re;}  
        T getIm() {return im;}  
}
```

- Type variables are fixed explicitly when the object is initialized
 - `Complex <double> x(1.0,2.0) // T = double`
 - `Complex <int> j(3,4) // T = int`
 - `Complex <char*> str("1.0","6") // T = char *`

C++ Class Template

- Type variables can be constant

```
template <class T, int dim> class Message{  
private:  
    T mess[dim];  
    ...  
public:  
    Message (T *str, int n) {  
        int end = min(n,dim);  
        for(int i=0; i<end; i++)  
            mess[i]= str[i];  
    }  
    Message <char, 80> m ("Message 1", 8);  
    // T = char, dim = 80
```

What is the STL?

- “Standard Template Library” by Alex Stepanov in 1976
- Basic motivation:
 - N data types, M containers, and K algorithms
 - Possibly $N * M * K$ implementations
 - CountIntegerInList(IntList il, int toFind),
CountIntegerInSet, CountDoubleInList, etc.
 - STL (with C++ templates): $N + M + K$ implementations
 - algorithms operate over containers of types
 - `set<int> mySet;`
 - `count(mySet.begin(), mySet.end(), 4);`
 - `list<double> myList;`
 - `count(myList.begin(), myList.end(), 3.14);`

Platforms

- STL is part of Standard C++
- Ported in all the major compilers
- Stlport.org
 - Free std C++ implementation (including iostreams), some nice features/performance

STL overview

- Fundamentally, the STL defines *algorithms* that operate over a *range* in a *container*
- Our order:
 - **Containers**: a collection of typed objects
 - **Iterators** (ranges): generalization of pointer or address to some position in a container
 - **Algorithms**

Containers

- **Lists**
 - vector, list, deque
- **Adaptors**
 - queue, priority_queue, stack
- **Associative**
 - map, multimap, set, multiset
 - hash_{above}

vector<T>

- #include <vector>
- A dynamic array: random-access, grows
- Array-indexing syntax: operator[] (`dim_type n`)

```
vector<int> v(10) ; v[0] = 4 ;
```

Defining a Vector

- Basic definition

```
vector<T> name;
```

Container's object name
Base element type

- The type can be any type or class!
- Must have: `#include <vector>`
- **Must have:** `using namespace std;`
- Creates an empty vector
- Example

```
vector<int> A;           // 0 ints
vector<double> B;        // 0 doubles
vector<string> C;        // 0 strings
```

Modifying a vector object

- Add a new element at the end of the vector
 - `push_back(const T &val)`
 - Inserts a copy of `val` after the last element of the vector
- Remove one element at the end of the vector
 - `pop_back()`
 - Removes the last element of the vector

How many elements?

- **size_type size()**
 - Returns the number of elements in the vector
`cout << A.size();`
 - Note: size_type is an “alias” name for an unsigned int
- **bool empty()**
 - Returns true if there are no elements in the vector; otherwise, it returns false

```
if (A.empty()) {  
    // ...
```

Example vector 1

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> A;
    if ( A.empty() ) cout << "A has size zero. ";
    A.push_back(3); // A: 3
    A.push_back(-25); // A: 3 -25
    cout << "Size of A: " << A.size(); // size 2
    A.pop_back(); // A: 3
    cout << "Size of A: " << A.size(); // size 1
}
```

Removing All Elements

- Two member function calls to remove all elements
 - Sometimes we need to “clear out” an existing vector
- **void `resize(size_type s)`**
 - The number of elements in the vector is now **s**.
 - Use with zero to remove all elements
 - If you “grow” a vector, default value/constructor used for new items
- **void `clear()`**
 - Removes all elements

```
vector<int> A;  
// assume we add elements to A here  
A.resize(0);           // A is now empty  
A.clear();            // same effect as above
```

Accessing Just One Element

- What if we want to retrieve or change one element?
 - Index value: from 0 to `size() - 1`
 - Pass index to the `at()` member function
- Example:

```
vector<int> A;  
// assume we add two or more elements to A  
A.at(0) = A.at(1) + 1;
```

- Note: can be used on left-hand side of assignment!
 - E.g. this changes the element stored at index 0
- Example: set last element to value of 1st element

```
A.at( A.size() - 1 ) = A.at(0);
```

What's Allowed on the Element?

- When you access one single element using `at()`, what are you allowed to do with that element?
 - Anything you could normally do with one variable of that type!
- Example: if `A` is a vector of `int`'s, and the element at index `i` exists
 - Element `A.at(i)` is an `int` just like any other `int` variable
 - We can print it, add to it, take its `sqrt`, pass it as a parameter to a function expecting an `int`
- Example: if `S` is a vector of strings, and `S.at(i)` exists
 - Element `S.at(i)` is one `string` object
 - We can print it, concatenate to it, call `size` or `substr` on it, pass it as a parameter to a function expecting a `string`

Vector Bounds Errors

- Elements only exist from index 0 to size()-1
 - Very common error to refer to `A.at(i)` where `i==A.size()`
 - If there are 10 items, the last one is at index 9
- What if you make such a *vector-bounds error?*
- The `at()` member function checks its parameter
 - If not in bounds, throws a run-time exception
 - Your program halts
 - (Heard of arrays? They don't do this check.)

Example 2

```
#include <vector>
#include <string>

int main() {
    int i;
    vector<string> A;
    A.push_back("I") ; A.push_back("am") ;
    A.push_back("me") ;

    for (i = 0; i < A.size(); ++i) // why not <= ?
        cout << A.at(i) << " ";
    cout << endl;
```

Example 2 continued

```
// swap 1st and last elements
string Temp = A.at(0);
A.at(0) = A.at( A.size()-1 ); // NOTE!!!
A.at( A.size()-1 ) = Temp;

A.at( A.size()-1 ) += "!";
// add ! to end

for (i = 0; i < A.size(); ++i)
    cout << A.at(i) << " ";
cout << endl;

return 0;
}
```

Operating on the Whole Vector

- We can do some things on the entire vector
 - Assignment: If two vectors are defined to hold the same kinds of elements
 - Example:
 - `vector<int> A, B;`
 - `// assume we add some elements A`
 - `B = A; // B's old contents gone, now == A`
- Logical equality operators `==` and `!=` work too
 - `if (B == A) { // same size, same (==) elements ?`

Function Examples: Input

```
void GetIntList(vector<int> &A) {  
    A.resize(0);  
    int Val;  
    while (cin >> Val) {  
        A.push_back(Val);  
    }  
}  
  
vector<int> List;  
cout << "Enter numbers: " ;  
GetIntList(List);
```

Function Example: Output

```
void PutIntList(const vector<int> &A) {  
    for (int i = 0; i < A.size(); ++i) {  
        cout << A.at(i) << endl;  
    }  
}  
  
vector<int> myList;  
// somehow values get into myList  
cout << "Your numbers: "  
PutIntList(myList)
```

- Question: Why is formal parameter const reference?

Other Useful Functions

- Often we need to search a vector for an item:
 - `int find (const vector<T> &vect, T target);`
 - Loops through the elements in the vector, searching for an element equal to `target`
 - Returns index of `target` if it's found.
 - If not found, return either -1 or `vect.size()`
 -
- Defined functions only allow us to add/remove at vector's end
 - By using `push_back()` and `pop_back()`
 - Could we write functions that take an index value and use it to tell us where to insert or remove an element?

Other Useful Functions (cont'd)

- **void deleteAt (vector<T> &vect, int idx);**
 - Remove the element at index **idx** (if it exists)
 - How? Must use loop to “shift down” elements, then call **pop_back ()** to remove unneeded element at the end
- **void insertAfter (vector<T> &vect, T newItem, int idx);**
 - Add **newItem** after element with index **idx**
 - How?
 - Must **push_back ()** to get one more “space”
 - Must use loop to “shift up” elements
 - Finally do: **vect.at(idx+1) = newItem;**

vector<T>

- Time:
 - constant time insertion and removal of elements at the end
 - linear time insertion and removal of elements at the beginning or in the middle.
- The “standard” container

Forward reference: Iterators

- v.begin() and v.end() return iterators
- Like pointers: arithmetic (++, --) and dereferencing (*)

```
for (vector<int>::iterator i =  
      v.begin() ; i != v.end() ; ++i)  
    cout << *i;
```

list<T>

- Bidirectional, linear list
- Sequential access only (not L[52])
- Constructors
 - ❑ `list<T>()`
 - ❑ `list<T>(size_t num_elements)`
 - ❑ `list<T>(size_t num, T init)`
- Properties
 - ❑ `l.empty() // true if l has 0 elements`
 - ❑ `l.size() // number of elements`

list<T>

- Adding/deleting elements
 - ❑ `l.push_back(43);`
 - ❑ `l.push_front(31);`
 - ❑ `l.insert(iterator,4) // insert 4 before the position “iterator”`
 - ❑ etc..
- Accessing elements
 - ❑ `l.front() // T &`
 - ❑ `l.back() // T &`
 - ❑ `l.begin() // list<T>::iterator`
 - ❑ `l.end() // list<T>::iterator`

list<T>

- Removing elements
 - ❑ `l.pop_back()` // returns nothing
 - ❑ `l.pop_front()` // returns nothing
 - ❑ `l.erase(iterator i)`
 - ❑ `l.erase(iter start, iter end)` // delete a *range*
- Time
 - ❑ Amortized constant time insertion and removal of elements at the beginning or the end, or in the middle [because you pass an iterator]

list<T>

- Other operations
 - ❑ `l.sort(), l.sort(CompFn)` // sorts in place
 - ❑ `l.splice(iter b, list<T>& grab_from)`

list<T>

- Example:

```
list<char> l;
for (int i = 0; i < 4; ++i)
{
    l.push_front(i + 'A');
    l.push_back(i + 'A');
}
for (list<char>::iterator i = l.begin();
     i != l.end(); ++i)
    cout << *i; // DCBAABCD
```

Other data structures

- Hashtables / Map
- Queue
- Stack
- Set
- ...
- algorithms ...

[hash_]map, [hash_]multimap

- A map is an “associative container”
- Given one value, will find another
 - `map<string, int>` is a map from strings to int's
 - maps are 1:1, multimap are 1:n
- map, multimap are **logarithmic** when inserting/deleting
 - Needs to maintain sortedness
- `hash_map`, `hash_multimap` are amortized **constant time**
 - Not sorted (“hashed”)

Map functions

- `m.insert(make_pair(key, value));` // inserts
- `m.count(key);` // times occurs (0, 1)
- `m.erase(key);` // removes it
- `m[key] = value;` // inserts it into the table
- `m[key]` //retrieves or creates a “default” for it
- `i=m.begin(), i=m.end()` // iterators
- `i->first, i->second` // per accedere a chiave e valore della coppia puntata da i

Hash_{...}

- There are `hash_map`, `hash_multimap`, `hash_set`, `hash_multiset`
- Basically, these are constant time insert/delete instead of log time
 - They don't maintain sortedness
 - Me: reduced running time from 10 min to 5 min

Hash performance

- Fill with 100,000 random elements
- Lookup 200,000 random elements
 - Same random seed
- map: fill 0.59967s
- map: lookups 1.57483s
- hash_map: fill 0.615407s
- hash_map: lookups 0.872557s
- So, if you don't need order, go with hash_map

Summary

- map: 1:1, sorted, $m[k] = v$
- multimap: 1:n, sorted,
`mm.insert(make_pair(k,v))`
- set: unique elements, sorted
- multiset: multiple keys allowed, sorted
- hash_: faster but **not sorted**

Iterators

- Touched on earlier
- An iterator is like a pointer
- You can increment to it to go to the “next” element
- You can [sometimes] subtract or add N
- You can dereference it
- Different kinds of iterators
- Most useful when combined with algorithms

Iterators

- `c.begin()` = start
- `c.end()` = 1 past the last element
 - Never dereference end! (`*c.end()` is bad!)
- Why? Makes loops simpler.
- Prefer `++i` because `i++` makes a temporary object and returns it, incrementing later.

Different kinds

- Technically:
 - ❑ random access ($i += 3; --i; ++i$)
 - ❑ bidirectional ($++i, --i$), store/retrieve
 - ❑ forward ($++i$), store/retrieve
 - ❑ input ($++i$) retrieve
 - ❑ output ($++o$) store
- But, writing code directly using iterators hurts a lot

Practical iterators

- **iterator**
 - “Standard”, goes from beginning to end
 - `c.begin()`, `c.end()`
- **const_iterator**
 - Like `iterator`, but changes can't be made (prefer!)
 - `c.begin()` and `c.end()` are overloaded so you can use them to assign their result to a `const_iterator`
- **reverse_iterator**
 - Goes from the end to the beginning with same semantics as iterator
 - Generally, `c.rbegin()` and `c.rend()`
 - `list`, `vector`, `deque`, `map`, `multimap`, `set`, `multiset`, `hash_`, `string`

Iterator example

```
vector<int> v;
for (int k = 0; k < 7; ++k) v.push_back(k);
display(v); // 0 1 2 3 4 5 6

for(vector<int>::iterator i = v.begin(); i != v.end();
    ++i)
    *i = *i + 3; // add three to content
display(v); // 3 4 5 6 7 8 9

for(vector<int>::const_iterator ci = v.begin();
    ci != v.end(); ++ci)
    cout << *ci << ' ';// *ci = *ci - 3; won't compile
cout << endl;//      3 4 5 6 7 8 9

for (vector<int>::reverse_iterator ri = v.rbegin();
    ri != v.rend(); ++ri)
{ *ri = *ri - 3;
    cout << *ri << ' ';}
cout << endl; //6 5 4 3 2 1 0
```

Sort Functions

- Just a touch!

```
vector<int> v;  
// fill v with 3 7 5 4 2 6  
sort (v.begin(), v.end() );
```

Polymorphic STL containers and iterators

- **STL and inheritance do not mix well together**
- A STL container expects to contain its objects directly, so:

```
ellipse e(rect1);  
rectangle r(rect2);  
list < shape > shapeList;  
shapeList.push_back(e);  
shapeList.push_back(r);
```

- this code will compile but will do a slicing to shapes.

(2)

- It becomes apparent that one level of indirection is needed to solve the problem. An obvious solution is to change the list of shapes to a list of pointers to shapes:
- `list < shape* > shapeList;`
- and list would be populated:
- `shapeList.push_back(&e);`
- `shapeList.push_back(&r);`
- Other problems arise...

Esempio

- Lista di studenti con due classi derivate LS e IL
 - ...
- Implementa un metodo (virtual?) calcolo media
- Fai un vector di Studenti
- Inserisci tre studenti
- Chiama per tutti in metodo calcolo media

Conclusion

- The STL has everything
- Let the compiler do the work for you
- Saves time and lines of code
- **Run-time efficiency** of the code that is generated
- Next steps:
 - ❑ Buy a good book on STL
 - Schildt's STL Programming from the Ground Up
 - ❑ Use it on your homeworks/personal projects
 - ❑ Learn about function objects
 - Didn't have time to cover them; another talk??

Resources

- Books
 - Schildt – “STL Programming from the Ground Up” ***
 - Schildt – “C/C++ Programmers Reference”
- URLs
 - <http://www.stlport.org/resources/StepanovUSA.html>
 - http://www.usenix.org/publications/library/proceedings/coots97/full_papers/sundaresan/sundaresan_html/node2.html
 - MSDN
 - Google: sgi stl <container or algorithm>

Sottotipazione e covarianza.

Riassunto critico

Rivediamo il concetto di sottotipazione per funzioni (metodi)

http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

Sottotipazione in generale

- $A <: B$: A sottotipo di B
- Subtyping principle
- $A <: B$ se un'espressione A può essere usata in modo sicuro in ogni contesto in cui sarebbe richiesto un B
- In questo modo vale il principio di sostituibilità
- Per variabili (istanze di classe):
 - `B b = new A()` (java)
 - `B* B = new A;` (C++)
- Dove mi aspetto un B posso passare un A.

Sottotipazione per funzioni

- Per le funzioni?
- Posso estendere e dire che una funzione è sottotipo di un'altra se può essere usata al suo posto.
- In teoria potrei ammettere l'overriding e il binding dinamico di tutte le funzioni che siano sottotipo
- Quando una funzione $f: W \rightarrow Z$ è sottotipo di un'altra?
- $f:W \rightarrow Z$ vuol dire che prende un W e restituisce un Z

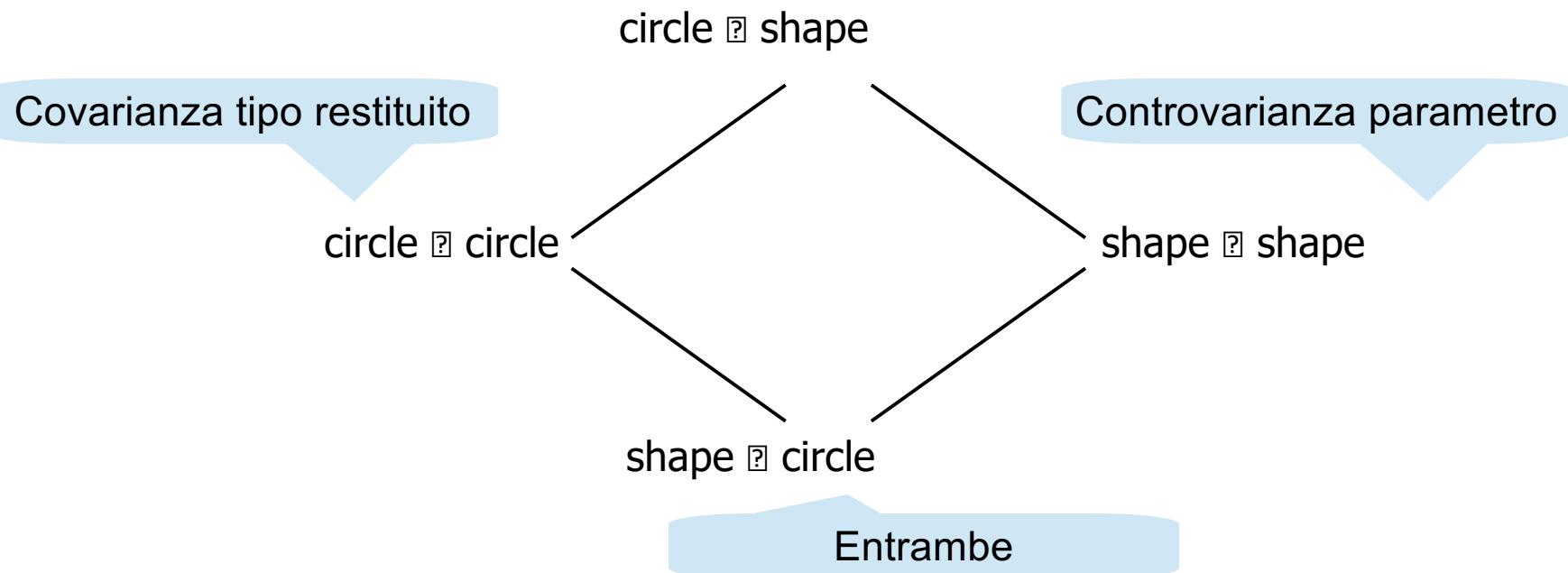
Covarianza tipo restituito

Rispetto ai parametri?

- Rispetto ai suoi argomenti (controvarianza)
- If $A <: B$, then $fb:B \rightarrow C <: fa:A \rightarrow C$
- Cioè fb può essere usata al posto di fa se prende come argomento un supertipo (B invece che A)
- Uso
- $fa(x)$ [x di tipo A] può essere sostituita da $fb(x)$
- Una sottoclasse quindi potrebbe “sostituire” fa con fb
- Terminology
- Contravariance: $A <: B$ implies $F(B) <: F(A)$

Esempio

- Se $\text{circle} \ll \text{shape}$, then



Nei linguaggi di programmazione

- In pratica i linguaggi di programmazione permettono la “sostituzione”- overriding solo in alcuni casi:
- C++: covarianza tipo ritornato in virtual functions
- Java: covarianza tipo ritornato (da Java 5)
- I parametri devono avere lo stesso tipo (invarianza) altimenti ho overloading
- ... altri linguaggi ...

C++ Smart pointers

- ANGELO GARGANTINI
- PROGRAMMAZIONE AVANZATA AA 22/23

Cosa sono gli smart pointer

Gli smart pointer cercano di risolvere il problema della gestione della memoria (memory allocation e deallocation)

- Evitare dangling pointers, memory leak etc.

Il principio è RAI^I o Resource Acquisition Is Initialization.

Obiettivo: quello di dare la proprietà di qualsiasi risorsa allocata nell'heap a un oggetto allocato nello stack il cui distruttore contiene il codice per eliminare o liberare la risorsa.

SMART POINTER → STACK + HEAP

- DEALLOCATO → STACK + PULIZIA DELLO HEAP

Come si usano i puntatori intelligenti

```
void UseRawPointer() {
// Utilizzo di un puntatore raw - non
consigliato .
Song *pSong = new Song(" Nothing on You");
// Usa pSong ...
// Non dimenticare di eliminare !
delete pSong;
}
```

I puntatori raw sono passati ad un puntatore intelligente che lo gestirà

```
void UseSmartPointer() {
// Dichiara un puntatore intelligente sullo stack e
// gli passa il puntatore raw.
unique_ptr<Song> song2(new Song(" Nothing on You"));
// Usa song2 ...
//wstring s = song2 -> duration_;
// ...
} // song2 viene cancellata automaticamente qui.
```

Distruzione di un puntatore intelligente

- Il distruttore del puntatore intelligente contiene la chiamata per eliminare anche il puntatore raw
 - poiché il puntatore intelligente è dichiarato nello stack, il suo distruttore viene richiamato quando il puntatore intelligente esce dallo scope
- Sono sicuro che il puntatore verrà cancellato (delete)
- Importante mai utilizzare l'espressione new o malloc sul puntatore intelligente stesso.

Come usare un puntatore intelligente

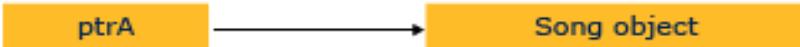
- Per accedere al puntatore encapsulato si utilizzano i soliti operatori del puntatore, -> e *, che la classe del puntatore intelligente sovraccarica per restituire il puntatore grezzo encapsulato.
- L'accesso al puntatore encapsulato utilizzando gli operatori smart pointer sovraccaricati * e -> non è significativamente più lento rispetto all'accesso diretto ai puntatori raw.
- I puntatori intelligenti di solito forniscono un modo per accedere direttamente al loro puntatore grezzo (get) – da usare con molta attenzione

Tipi di puntatori

- `unique_ptr`: Consente esattamente un proprietario del puntatore sottostante. Se lo distruggo per uno viene distrutto per tutti
- `shared_ptr` Puntatore intelligente con conteggio dei riferimenti. Utilizzare quando si desidera assegnare un puntatore raw a più proprietari, ad esempio, quando si restituisce una copia di un puntatore da un contenitore ma si desidera mantenere l'originale.
- `weak_ptr` Puntatore intelligente per casi speciali da utilizzare insieme a `shared_ptr`.

Unique ptr

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

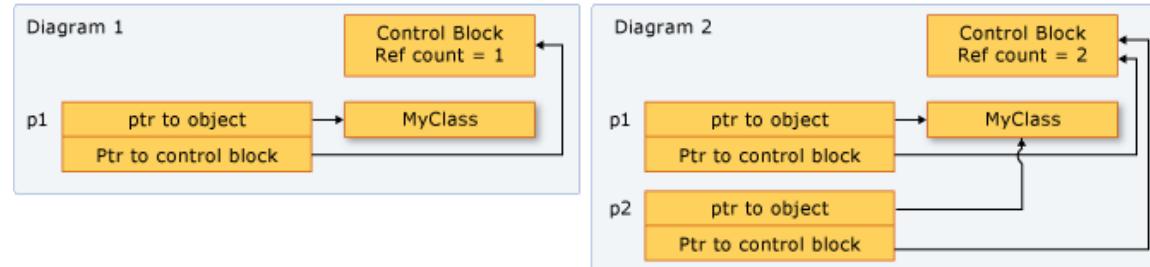


```
auto ptrB = std::move(ptrA);
```



- Un unique_ptr non condivide il puntatore. Non può essere copiato in un altro , passato per valore a una funzione o usato in qualsiasi algoritmo della libreria standard C++

Shared pointer



- Il tipo `shared_ptr` è progettato per scenari in cui più proprietari potrebbero dover gestire la durata dell'oggetto in memoria.
- Dopo avere inizializzato un oggetto `shared_ptr`, è possibile copiarlo, passarlo come valore negli argomenti di funzione e assegnarlo ad altre istanze di `shared_ptr`.
- Tutte le istanze puntano allo stesso oggetto e condividono l'accesso a un "blocco di controllo" che incrementa e decrementa il conteggio dei riferimenti ogni qualvolta un nuovo oggetto `shared_ptr` viene aggiunto, esce dall'ambito o viene reimpostato. Quando il conteggio dei riferimenti arriva a zero, il blocco di controllo elimina la risorsa di memoria e se stesso.

C++

Additions not related to objects

Overview

- Additions and changes not related to objects
 - type bool
 - pass-by-reference & the Copy-Constructor
 - user-defined overloading
 - function template and class template
 - exception handling
 - ...

Type bool

- Represents boolean-values
- Conversion rules with the `int` type

```
bool b1, b2, b3;  
int j, k;  
b1 = 3*5; // b1 = true  
b2 = 0; // b2 = false  
j = b1; // j = 1  
j = b1 || b2; // j = 1  
j = b1 && b2; // j = 0  
b1 = j == 0; // b1 = true
```

Reference variables (1)

```
int a, *ptr_a;  
int &ref_a = a;  
// ref_a is an address, a reference variable  
ref_a = 5;      // or a = 5  
ptr_a = &ref_a; // or ptr_a = &a;
```

Reference variables (2)

A reference variable is similar to a ***const* pointer**

```
int a, *ptr_a;  
int &ref_a = a;  
ref_a = 5;  
ptr_a = &ref_a;
```



```
int a, *ptr_a;  
int * const ptr_a = &a;  
*ptr_a = 5;  
ptr_a = ptr_a;
```

This implies:

- a reference variable must be initialized when defined
- must refer always to the same variable, reassignment is not allowed

Call-by-reference (1)

```
int f(int& t_in) {  
    t_in = 99;  
...  
}
```

A good style

```
int f(const int& t_in) {  
    t_in = 99; // ERROR  
...  
}
```

Call-by-reference (2)

- Two possible realizations in C++
 - `void doSomething(Data * data);`
 - pointer-based
 - Advantages and drawbacks of pointer approach
 - `void doSomething(Data & data);`
 - Reference based
 - Non null-checking necessary

Return-by reference

```
int & g(int &x) {  
    return x  
}
```

A C++ program can be made easier to read and maintain by using references rather than pointers. A C++ function can return a reference in a similar way as it returns a pointer. When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement (unless const).

```
g(j) = 99 . . . j = 99  
const int & h(int x);  
  
...  
h(j) = 99; // ERROR
```

Summary

- Variables hold values
 - `int v = 5`
- Pointers hold addresses of variables
 - `int* p = new(int);`
 - `*p = 5;`
- References refer to contents of another variable
 - `int& r = a;`

The Copy Constructor

- `ChewingGum(const ChewingGum& rhs);`
- Default copy constructor
 - Automatically generated if not present
 - Produces a complete **shallow** copy of the passed object (e.g. pointers are copied equal)
- User-defined copy constructor
 - Can take arbitrary measures to provide a copy of the rhs object (e.g. deep copies)

The Assignment Operator

ChewingGum& operator

```
= (const ChewingGum& rhs);
```

- Sets an object to be a copy of a passed object
- Default behavior: shallow copies
- Example
 - ChewingGum g1;
 - ChewingGum g2 = g1;

The Assignment Operator & Inheritance

- When assigning to a base class, the = is used.
- Example

```
class A{}; class B: public A{}
```

```
A a;
```

```
B b;
```

a = b --> assignment of a is used.

- Note that fields of B that are not in A are not copied (slicing)

The Copy Constructor and the Assignment Operator

- copy constructors and assignment operators
 - automatically generated in each class
 - no inheritance

```
class Employee {  
    //...  
    Employee(const Employee&);  
    Employee& operator=(const Employee&);  
};
```

```
Manager m("Homer", 3);  
Employee e = m;      // Slicing!!
```

The Copy Constructor and dynamic memory

- **always** declare a user-defined **copy constructor** for classes with dynamically allocated memory
 - the default implementation leads to
 - undefined behaviour (probably an access violation)
-

An example (1)

```
#include <iostream.h>
#include <string.h>
class Message {
    char *subject;
    char *message;
    //A function to initialize data members
    init_message(const char *,const char *) ;
public:
    //A constructor
    Message(const char *, const char * = "") ;
    //The copy-constructor
    Message(const Message & m) ;
    //Overloading of the assignment operator =
    const Message& operator=(const Message &) ;
    //The destructor
    ~Message();
};
```

An example (2)

```
//A function to initiale data members
Message::init_message(const char *s, const char *m) {
    subject = new char [strlen(s)+1];
    strcpy(subject,s);
    message = new char [strlen(m)+1];
    strcpy(message,m);
}

//A constructor
Message(const char *s, const char * m) {
    init_message(s,m);
}

//The destructor
~Message() {
    delete subject;
    delete message;
}
```

An example (3)

```
//The copy constructor
Message::Message(const Message & m) {
    init_message(m.subject,m.message);
}

//Overloading of the assignment operator =
const Message& Message::operator=(const Message & m) {
    // always check for self-assignment
    if (this == &m) return *this;
    // clean current object
    delete subject;
    delete message;
    init_message(m.subject,m.message);
    return *this; //the left element is returned
}
```

Assignment sequences

- C++ allows for

```
int a, b, c, d;  
a = b = c = d = 5;
```

- Objects should allow this as well
- assignment operator needs to return a reference to (**this*)

References (1)

- C++ as a language and programming guidelines
 - Stroustrup, B. (1999). *The C++ Programming Language*, Addison-Wesley.
 - Meyers, S. (1998). *Effective C++*, Addison-Wesley
 - Meyers, S. (1995). *More Effective C++*, Addison-Wesley
 - Meyers, S. (2000). *Effective STL*, Addison-Wesley
 - Alexandrescu, A. (2002). *Modern C++ Design*, Addison-Wesley

References (2)

- Process memory layout, etc.
 - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*, Chapters 3 and 6
 - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, Chapter 2
 - Santa Cruz Operation, Inc. (1997), *System V Application Binary Interface Intel386(tm) Architecture Processor Supplement*, Fourth Edition

String in C++

- C++ provides a simple, safe alternative to using `char*`s to handle strings. The C++ string class, part of the `std` namespace, allows you to manipulate strings safely.
- Declaring a string is easy:

```
using namespace std;  
string my_string;
```

or

```
std::string my_string;
```

- Vedi syllabus

References (3)

- **C++ Applications** by the creator of C++
- [Bjarne Stroustrup](#)
 - <http://www.research.att.com/~bs/applications.html>



A Brief Intro to **Scala**

Informatica III

AA 19/20

Origin

- Started at 2001 by Martin Odersky at EPFL Lausanne, Switzerland
- Scala 2.0 released in 2006
- Current version 2.12.4
- IDE (eclipse based) 4.7.0
- Twitter backend runs on Scala
 - **LinkedIn, Siemens, Sony, ...**

Scala

- Statically Typed
- Runs on JVM, full inter-op with Java
- Object Oriented
- Functional
- Dynamic Features

Scala is Practical

- Can be used as drop-in replacement for Java
 - Mixed Scala/Java projects
- Use existing Java libraries
- Use existing Java tools (Ant, Maven, JUnit, etc...)
- Decent IDE Support (NetBeans, IntelliJ, Eclipse)

Java

- What's wrong with Java?
 - Verbose
 - Too much of `Thing thing = new Thing();`
 - Too much “boilerplate,” for example, getters and setters
 - ...
- What's right with Java?
 - Very popular
 - Object oriented (mostly), which is important for large projects
 - Strong typing (more on this later)
 - The fine large library of classes
 - **The JVM!** Platform independent, highly optimized

Scala is like Java, except when it isn't

- Java is a *good* language, and Scala is a lot like it
- For each difference, there is a *reason*--none of the changes are “just to be different”
- Scala and Java are (almost) completely interoperable
 - Call Java from Scala? No problem!
 - Call Scala from Java? Some restrictions, but mostly OK.
 - Scala compiles to **.class** files (a *lot* of them!), and can be run with either the **scala** command or the **java** command
- To understand Scala, it helps to understand the reasons for the changes, and what it is Scala is trying to accomplish

Consistency is good

- In Java, every value is an object--unless it's a primitive
 - Numbers and booleans are primitives for reasons of efficiency, so we have to treat them differently (you can't "talk" to a primitive)
- In Scala, all values are objects. Period.
 - The compiler turns them into primitives, so no efficiency is lost (behind the scenes, there are objects like **RichInt**)
- Java has *operators* (`+`, `<`, ...) and *methods*, with different syntax
- In Scala, operators are just methods, and in many cases you can use either syntax

Differences with Java - basic

- Scala does not require semicolons to end statements.
- Value types are capitalized: Int, Double, Boolean instead of int, double, boolean.
- Parameter and return types follow, rather than precede as in C.
- Methods must be preceded by def.
- Local or class variables must be preceded by val (indicates an immutable variable) or var (indicates a mutable variable).
- The return operator is unnecessary in a function (although allowed); the value of the last executed statement or expression is normally the function's value.
- Instead of the Java cast operator (Type) foo, Scala uses foo.asInstanceOf[Type], or a specialized function such as toDouble or.toInt.
- Instead of Java's import foo.*;, Scala uses import foo._.
- Function or method foo() can also be called as just foo;

Scala is Concise

Type Inference

“capisce” il tipo di una espressione.

Statically typed lo stesso !

```
val sum = 1 + 2 + 3  
val nums = List(1, 2, 3)  
val map = Map("abc" -> List(1,2,3))
```

Nota: we use scala
worksheet

Explicit Types

```
val sum: Int = 1 + 2 + 3
```

```
val nums: List[Int] = List(1, 2, 3)
```

```
val map: Map[String, List[Int]] = ...
```

Higher Level

```
// Java - Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
}
}
```

Higher Level

Posso passare come argomento di una funzione una funzione

// Scala

```
val hasUpperCase = name.exists(_.isUpperCase)
```

Funzione f

Argomento di f

Less Boilerplate

```
// Java
public class Person {
    private String name;
    private int age;
    public Person(String name, Int age) { // constructor
        this.name = name;
        this.age = age;
    }
    public String getName() { // name getter
        return name;
    }
    public int getAge() { // age getter
        return age;
    }
    public void setName(String name) { // name setter
        this.name = name;
    }
    public void setAge(int age) { // age setter
        this.age = age;
    }
}
```

Less Boilerplate

// Scala

```
class Person(var name: String, var age: Int)
```

var: implicit
declaration
of field

Local or class variables must be preceded by **val** (indicates an immutable variable) or **var** (indicates a mutable variable).

Less Boilerplate

```
// Scala
class Person(var name: String, private var _age: Int) {
    def age = _age // Getter for age
    def age_=(newAge:Int) { // Setter for age
        println("Changing age to: "+newAge)
        _age = newAge
    }
}
```

Variables and Values

// variable

I-value vs r-value

var foo = "foo"

foo = "bar" // okay

// value

val bar = "bar"

~~bar = "foo"~~ // nope

null

- “I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

--Tony Hoare

null in Scala

- In Java, any method that is *supposed* to return an object *could* return **null**
 - Here are your options:
 - Always check for **null**
 - Always put your method calls inside a **try...catch**
 - Make sure the method can't possibly return **null**
- Yes, Scala has **null**--but only so that it can talk to Java
- In Scala, if a method *could* return “nothing,” write it to return an **Option** object, which is either **Some(theObject)** or **None**
 - This forces you to use a **match** statement--but only when one is really needed!

Esempio uso None

```
def.toInt(in: String): Option[Int] = {
  try{
    Some(Integer.parseInt(in.trim))
  } catch{
    case e: NumberFormatException => None
  }
}
```

Referential transparency

- In Scala, variables are really functions
 - Huh?
- In Java, if `age` is a public field of `Person`, you can say:
`david.age = david.age + 1;`
but if `age` is accessed via methods, you would say:
`david.setAge(david.getAge() + 1);`
- In Scala, if `age` is a public field of `Person`, you can say:
`david.age = david.age + 1;`
but if `Person` defines methods `age` and `age_=`, you would say (the same!)
`david.age = david.age + 1;`
- In other words, if you want to access a piece of data in Scala, you don't have to know whether it is computed by a method or held in a simple variable
 - This is the **principle of uniform access**
 - Scala *won't let you* use parentheses when you call a function with no parameters

Scala is Object Oriented
Scala is Functional

What is Multiparadigm Programming?

- Definition:
- A multiparadigm programming language provides “a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms.” [Tim Budd]

- Programming paradigms:
- imperative versus declarative (e.g., functional, logic)
- other dimensions – object-oriented, component-oriented, concurrency-oriented, etc.

Why Learn Multiparadigm Programming?

- Tim Budd:

“Research results from the psychology of programming indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is the number of years of experience in programming.”
- The “goal of multiparadigm computing is to provide ... a number of different problem-solving styles” so that a programmer can “select a solution technique that best matches the characteristics of the problem”.

Why Teach Multiparadigm Programming?

- Contemporary imperative and object-oriented languages increasingly have functional programming features, e.g.,
 - higher order functions (closures)
 - list comprehensions
- New explicitly multiparadigm (object-oriented/functional) languages are appearing, e.g.,
 - Scala on the Java platform (and .Net in future)
 - F# on the .Net platform

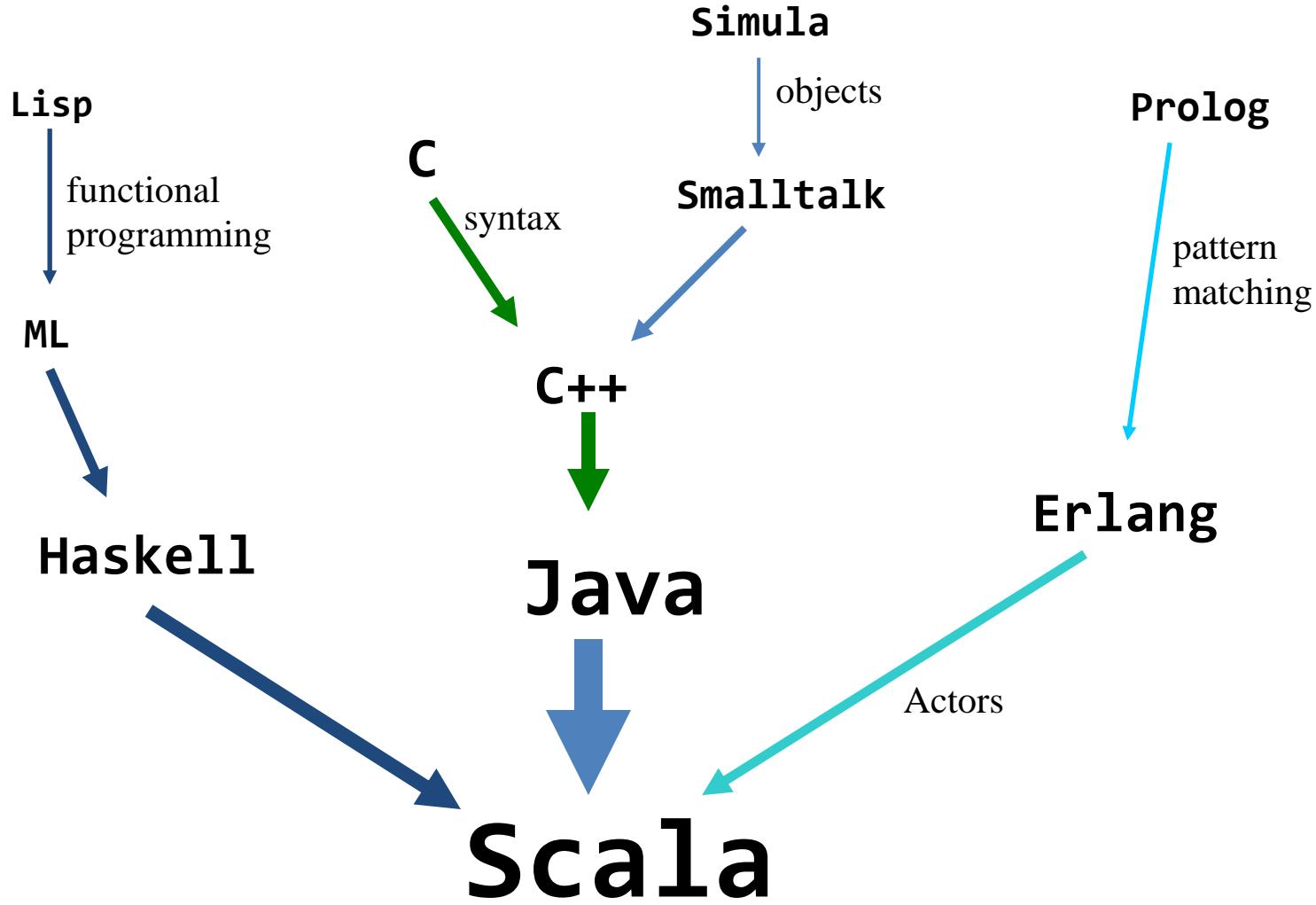
Functional languages

- The best-known functional languages are ML, OCaml, and Haskell
- Functional languages are regarded as:
 - “Ivory tower languages,” used only by academics (mostly but not entirely true)
 - Difficult to learn (mostly true)
 - The solution to all concurrent programming problems everywhere (exaggerated, but not entirely wrong)
- Scala is an “impure” functional language--you can program functionally, but it isn’t forced upon you

Scala as a functional language

- The hope--*my* hope, anyway--is that Scala will let people “sneak up” on functional programming (FP), and gradually learn to use it
 - This is how C++ introduced Object-Oriented programming
- Even a little bit of functional programming makes some things a lot easier
- Meanwhile, Scala has plenty of other attractions
- FP really is a different way of thinking about programming, and not easy to master...
- ...but...
- Most people that master it, never want to go back

Genealogy



Scala is Dynamic

(Okay not really, but it has lots of
features typically only found in
Dynamic languages)

Read-Eval-Print Loop

```
bash$ scala
```

```
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM,  
Java 1.6.0_22).
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala> class Foo { def bar = "baz" }
```

```
defined class Foo
```

```
scala> val f = new Foo
```

```
f: Foo = Foo@51707653
```

```
scala> f.bar
```

```
res2: java.lang.String = baz
```

Noi useremo però ScalalDE

Structural Typing

```
// Type safe Duck Typing
def doTalk(any:{def talk:String}) {
    println(any.talk)
}
```

```
class Duck { def talk = "Quack" }
class Dog   { def talk = "Bark" }
```

```
doTalk(new Duck) → "Quack"
doTalk(new Dog)  → "Bark"
```

tipizzazione dinamica dove la semantica di un oggetto è determinata dall'insieme corrente dei suoi metodi e delle sue proprietà anziché dal fatto di estendere una particolare classe o implementare una specifica interfaccia

il duck typing permette il polimorfismo (sottotipazione) senza ereditarietà

Scala has tons of other cool stuff

Default Parameter Values

```
def hello(foo:Int = 0, bar:Int = 0) {  
    println("foo: "+foo+" bar: "+bar)  
}
```

hello()	→	foo: 0	bar: 0
hello(1)	→	foo: 1	bar: 0
hello(1,2)	→	foo: 1	bar: 2

Named Parameters

```
def hello(foo:Int = 0, bar:Int = 0) {  
    println("foo: "+foo+" bar: "+bar)  
}
```

hello(bar=6)	→	foo: 0	bar: 6
hello(foo=7)	→	foo: 7	bar: 0
hello(foo=8,bar=9)	→	foo: 8	bar: 9

Everything Returns a Value

```
val a = if(true) "yes" else "no"
```

Non esiste il void

```
val b = try{
    "foo"
} catch {
    case _ => "error"
}
```

```
val c = {
    println("hello")
    "foo"
}
```

Lazy Vals

```
// initialized on first access
lazy val foo = {
    println("init")
    "bar"
}
```

```
foo → init
foo →
foo →
```

Nested Functions

```
// Can nest multiple levels of functions
def outer() {
    var msg = "foo"
    def one() {
        def two() {
            def three() {
                println(msg)
            }
            three()
        }
        two()
    }
    one()
}
```

By-Name Parameters

- ... argument is not evaluated at the point of function application, but instead is evaluated at each use within the function.
- Sintassi
 - Pass by value

```
def f (x:Int, y:Int) = x;
```
 - Pass by name

```
def f (x: => Int, y: => Int) = x;
```

By-Name Parameters

- Ci sia una funzione che ha side effect e che restituisce qualcosa

```
def something() = {  
    println("callingsomething")  
    1  
}
```

Poi due funzioni che stampano due volte l'argomento che passo:

```
def callByValue(x: Int) = {  
    println("x1=" + x)  
    println("x2=" + x)  
}  
  
def callByName(x: => Int) = {  
    println("x1=" + x)  
    println("x2=" + x)  
}
```

- Se chiamo:

```
callByName(something())
```

Come al solito ho:

- Valuto `something()` e passo il suo valore.

```
calling something
```

```
x1=1
```

```
x2=1
```

- Se chiamo:

```
callByName(something())
```

Valuto `something()` dentro in `callByName`:

```
calling something
```

```
x1=1
```

```
calling something
```

```
x2=1
```

Which is faster?

```
def test (x:Int, y:Int)= x*x
```

```
def test (x: =>Int, y:>Int)= x*x
```

We want to examine the evaluation strategy and determine which one is faster (less steps) in these conditions:

- test (2,3)
- call by value: test(2,3) -> 2*2 -> 4
- call by name: test(2,3) -> 2*2 -> 4
- test (3+4,8)
- call by value: test (7,8) -> 7*7 -> 49
- call by name: (3+4) (3+4) -> 7(3+4)-> 7*7 ->49
- Here call by value is faster.

- test (7,2*4)
- call by value: test(7,14) -> 7*7 -> 49
- call by name: 7 * 7 -> 49
- Here call by name is faster
- test (3+4, 2*4)
- call by value: test(7,2*4) -> test(7, 8) -> 7*7 -> 49
- call by name: (3+4)(3+4) -> 7(3+4) -> 7*7 -> 49
- The result is reached within the same steps.

Foreach

```
val list = List("mff", "cuni", "cz")
```

- Following 3 calls are equivalent

```
list.foreach((s : String) => println(s))
```

```
list.foreach(s => println(s))
```

```
list.foreach(println)
```

Many More Features

- **Actors**
- **Annotations** → @foo def hello = "world"
- **Case Classes** → case class Foo(bar:String)
- **Currying** → def foo(a:Int,b:Boolean)(c:String)
- **For Comprehensions**
 - for(i <- 1.to(5) if i % 2 == 0) yield i
- **Generics** → class Foo[T](bar:T)
- **Package Objects**
- **Partially Applied Functions**
- **Tuples** → val t = (1,"foo","bar")
- **Type Specialization**
- **XML Literals** → val node = <hello>world</hello>
- **etc...**

Programmazione OO in SCALA

Info III

Elenco degli argomenti sugli appunti

Basic scala

Basic Scala

- Use `var` to declare variables

```
var x = 3;  
x += 4;
```

- Use `val` to declare values

```
val y = 3;  
y += 4; // error
```

- Notice no types, but it is statically typed

```
var x = 3;  
x = "hello world"; // error
```

- Type annotations:

```
var x : Int = 3;
```

Defs, Vals, and Vars

Three types of identifier definitions:

`def` defines functions with parameters; RHS expression evaluated each time called

`val` defines **unchanging** values; RHS expression evaluated immediately to initialize

`var` defines storage location whose values can be changed by assignment statements; RHS expression evaluated immediately to initialize

Variables & values, type inference

```
var msg = "Hello"      // msg is mutable
msg += " world"
msg = 5;              // compiler error
```

Variables & values, type inference

```
val msg = "Hello world"      // msg is immutable  
msg += " world"           // compiler error
```

```
val n : Int = 3             // explicit type declaration  
var n2 : Int = 3
```

Immutability

- Why?
 - Immutable objects are automatically thread-safe (you don't have to worry about object being changed by another thread)
 - Compiler can reason better about immutable values -> optimization
 - Steve Jenson from Twitter: “*Start with immutability, then use mutability where you find appropriate.*”

Calling Java from Scala

- Any Java class can be used seamlessly

```
import java.io._  
val url = new URL("http://www.scala-lang.org")
```

demo

Methods

```
def max(x : Int, y : Int) = if (x > y) x else y
```

// equivalent:

```
def neg(x : Int) : Int = -x
```

```
def neg(x : Int) : Int = { return -x; }
```

Types

- Int, Double, String, Char, Byte, BigInt, ...
 - wrappers around Java types

OO programming in Scala

Scala object system

- Class-based
- Single inheritance
- Can define singleton **objects** easily (no need for static which is not really OO)
- Traits, compound types, and views allow for more flexibility

Defining Hello World

```
object HelloWorld {  
    def main(args: Array[String]) {  
        println("Hey world!")  
    }  
}
```

- Singleton object named HelloWorld (also replaces static methods and variables)
- Method main defined (procedure)
- Parameter args of type Array[String]
- Array is generic class with type parameter

Classes

```
/** A Person class.  
 * Constructor parameters become  
 * public members of the class.*/  
class Person(val name: String, var age: Int) {  
    if (age < 0) {  
        throw ...  
    }  
}  
  
var p = new Person("Peter", 21);  
p.age += 1;
```

Constructor

In Scala the *primary constructor* is the class' body and it's parameter list comes right after the class name.

In Scala we create variables (fields) either using the `val` keyword or the `var` keyword. Using `val` we get a read-only variable that's immutable.

```
class Person(n: String){  
    val name = n;  
    def getname() = name  
}
```

```
class Person(val name:  
            String){  
    def getname() = name  
}
```

```
class Person(var name:  
            String){  
    def getname() = name  
}
```

Person
name
getname

Auxiliary
constructors
....

Objects

- Scala's way for “statics”
 - not quite – see next slide
 - (in Scala, there is no *static* keyword)
- “Companion object” for a class
 - = object with same name as the class

demo

An analog to a companion object in Java is having a class with static methods. In Scala you would move the static methods to a Companion object.

singleton object

```
object Main {  
    def sayHi() {  
        println("Hi!");  
    }  
}
```

- This example defines a singleton object called Main. You can call the method sayHi() like this:

```
Main.sayHi();
```

Companion Objects

When a singleton object is named the same as a class, it is called a **companion object**. A companion object must be defined inside the same source file as the class.

A companion class or object can access the private members of its companion.

Use a companion object for methods and values which are not specific to instances of the companion class. (instead of static)

```
class Circle(val radius: Double) {  
    def area: Double = Circle.calculateArea(radius)  
}  
object Circle {  
    private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)  
}  
object ProvaCircle {  
    def main(args: Array[String]): Unit = {  
        val circle1 = new Circle(5.8)  
        println(circle1.area)  
    }  
}
```

Extending classes

```
class Point(xc: Int, yc: Int) {  
    val x: Int = xc  
    val y: Int = yc  
    def move(dx: Int, dy: Int): Point = new Point(x + dx, y + dy)  
}  
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v){  
    val color: String = c  
    def compareWith(pt: ColorPoint): Boolean =  
        (pt.x == x) && (pt.y == y) && (pt.color == color)  
    override def move(dx: Int, dy: Int): ColorPoint =  
        new ColorPoint(x + dy, y + dy, color)
```

ColorPoint adds a new method compareWith - Scala allows member definitions to be *overridden*; Piu' o meno le stesse regole di Java (covarianza del tipo ritornato).

Subclasses define subtypes; this means in our case that we can use ColorPoint objects whenever Point objects are required.

Abstract classes

```
abstract class Greeter {  
    val message: String //abstract  
    def SayHi() = println(message)  
}  
  
class BergemGreeter extends Greeter {  
    val message = "alura"  
}  
  
object prova {  
    val greeter = new BergemGreeter() > greeter :  
BergemGreeter = BergemGreeter@141a32f  
    greeter.SayHi() > alura  
}
```

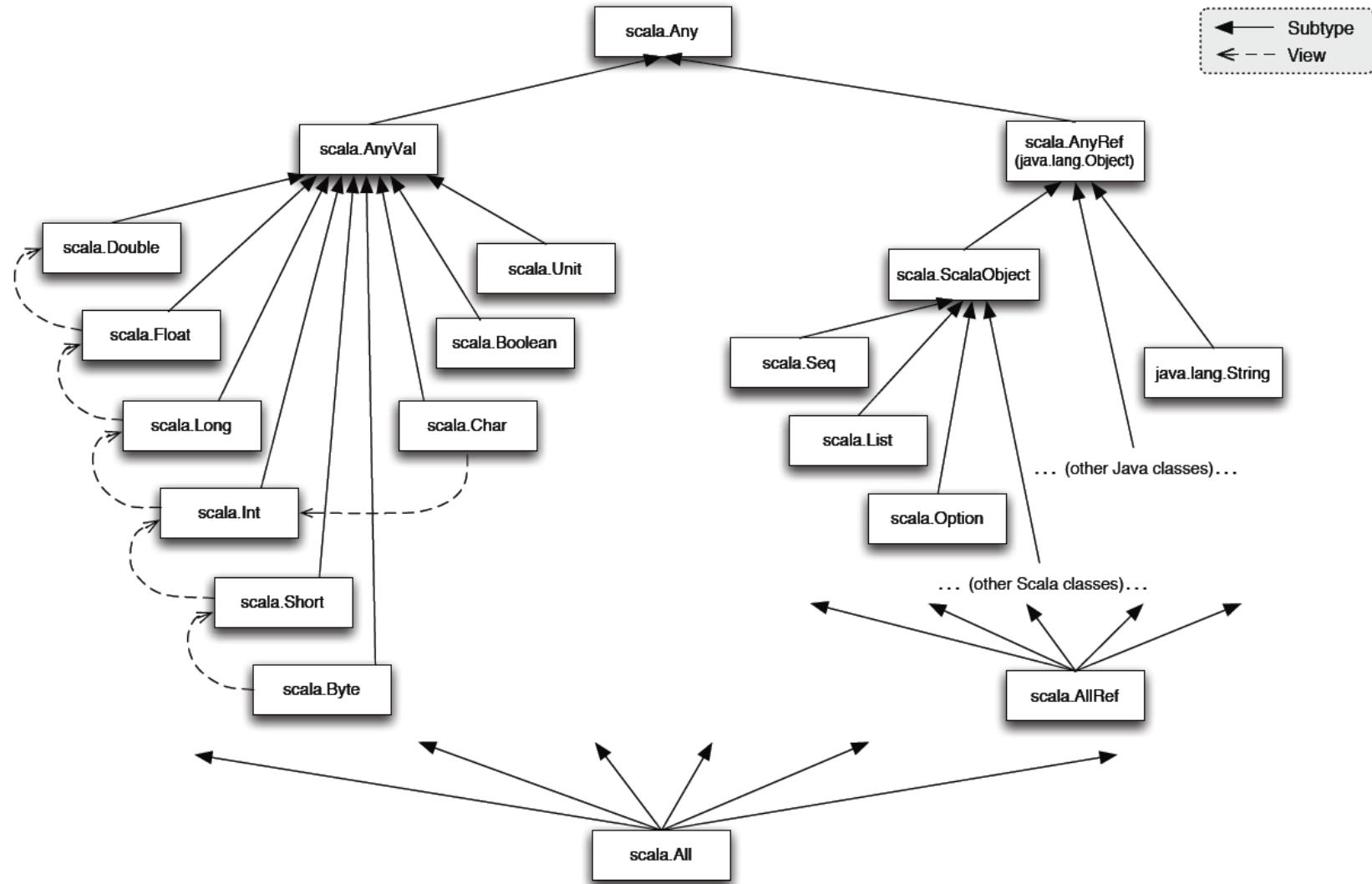
Traits

- Similar to interfaces in Java
 - See Java 8's interfaces
- They may have implementations of methods
- But can't contain state
- Can be multiply inherited from
 - Scala's solution to the Diamond Problem is actually fairly simple: it considers the order in which traits are inherited. If there are multiple implementors of a given member, the implementation in the supertype that is furthest to the right (in the list of supertypes) "wins." Of course, the body of the class or trait doing the inheriting is further to the right than the entire list of supertypes, so it "wins" all conflicts, should it provide an overriding implementation for a member.

Traits example

```
trait Similarity {  
    def isSimilar(x: Any): Boolean  
}  
  
class Student extends Similarity {  
    def isSimilar(x: Student) = true  
}
```

Scala class hierarchy



Introduction to Functional Programming with Scala

Angelo Gargantini

INFO 3A AA 2016/17

credits: Pramode C.E

<https://class.coursera.org/progfun-00>

December 14, 2016

Workshop Plan

Here is what we will do:

- Learn a bit of functional programming in Scala
- Learn some important concepts like (NOT ALL): closures, higher order functions, purity, lazy vs strict evaluation, currying, tail calls/TCO, immutability, persistent data structures, type inference etc!

Workshop material (slide/code samples) sul sito.

Function Definition

```
def add(a:Int, b:Int):Int = a + b  
  
val m:Int = add(1, 2)  
  
println(m)
```

Note the use of the type declaration "Int". Scala is a "statically typed" language. We define "add" to be a function which accepts two parameters of type Int and returns a value of type Int. Similarly, "m" is defined as a variable of type Int.

Function Definition

```
def fun(a: Int):Int = {  
    a + 1  
    a - 2  
    a * 3  
}  
  
val p:Int = fun(10)  
println(p)
```

Note!

There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

Type Inference

```
def add(a:Int, b:Int) = a + b  
  
val m = add(1, 2)  
  
println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

Type Inference

```
def add(a, b) = a + b  
  
val m = add(1, 2)  
  
println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML. Scala is said to do local, "flow-based" type inference while Haskell/ML do Hindley-Milner type inference

References

Expression Oriented Programming

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"

println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

Expression Oriented Programming

```
def errorMsg(errorCode: Int) = errorCode match {  
    case 1 => "File not found"  
    case 2 => "Permission denied"  
    case 3 => "Invalid operation"  
}  
  
println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern.

Evaluation

Applications of parametrized functions are evaluated in a similar way as operators. Expressions are evaluated before passing their value to functions (values are passed to functions)

```
def square(x:Double) = x * x

square(2)

square(2+2)

square(square(2))
```

Evaluation of function application

Given a function application $f(e_1, \dots, e_n)$

- ① Evaluate all function arguments (e_1, \dots, e_n) from left to right.
Let v_1, \dots, v_n the corresponding values.
- ② Replace the function application by the function's right hand side (function body), and, at the same time
- ③ Replace (substitute) the formal parameters of the function by the actual arguments v_1, \dots, v_n

```
def sumOfSquare(x:Double, y: Double) = square(x) +  
square(y)
```

```
sumOfSquare(3,2+2)  
sumOfSquare(3,4)  
square(3) + square(4)  
3 * 3 + square(4)  
9 + square(4)  
....
```

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*

The basic idea is to reduce an expression to a value

It can be proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*

The basic idea is to reduce an expression to a value

It can be proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

```
def loop: Int = loop
```

```
loop
```

Alternative evaluation

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquare(3,2+2)
square(3) + square(2+2)
3*3 + (2+2) * (2+2)
....
```

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

test(3+4,8) CBV faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

test(3+4,8) CBV faster

test(7,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

test(3+4,8) CBV faster

test(7,2*4) CBN faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

test(3+4,8) CBV faster

test(7,2*4) CBN faster

test(3+4,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

test(3+4,8) CBV faster

test(7,2*4) CBN faster

test(3+4,2*4) same number of steps

Call by name functions

Scala normally uses call-by-value

But if the type of a function parameter with \Rightarrow it uses
call-by-name

Example

```
def constOne(x:Int, y: => Int) = 1
```

Using pen and paper, trace the evaluation of the following function calls for the function constOne:

constOne(1+2,loop)
constOne(loop,1+2)

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Consider the Euclid's algorithm

```
def gcd(a: Int, b: Int) = if (b == 0) a else gcd(b,a%b)
```

evaluate gcd(14,21) ...

Consider the factorial algorithm

```
def factorial(n: Int) = if (n == 0) 1 else  
  n*factorial(n-1)
```

evaluate factorial(4) ...

Recursion

```
// sum n + (n-1) + (n-2) + ... + 0
def sum(n: Int): Int =
  if (n == 0) 0 else n + sum(n - 1)

val m = sum(10)
println(m)
```

Try calling the function "sum" with a large number (say 10000) as parameter! You get a stack overflow!

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

```
def factorial(n: Int): Int = {  
    def loop(acc: Int, n: Int)=  
        if ( n == 0) acc  
        else loop(n*acc, n-1)  
    loop(0,n)  
}
```

Tail Calls and TCO

```
def sum(n: Int, acc: Int):Int =  
  if(n == 0) acc else sum(n - 1, acc + n)  
  
val r = sum(10000, 0)  
  
println(r)
```

This is a "tail-recursive" version of the previous function - the Scala compiler converts the tail call to a loop, thereby avoiding stack overflow!

Proviamo con la segnatura semplificata e un loop interno.

Tail Calls and TCO

```
(sum 4)
(4 + sum 3)
(4 + (3 + sum 2))
(4 + (3 + (2 + sum 1)))
(4 + (3 + (2 + (1 + sum 0))))
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 0)))
(4 + (3 + 2))
(4 + 5)
(9)
```

```
(sum 4 0)
(sum 3 4)
(sum 2 7)
(sum 1 8)
(sum 0 9)
(9)
```

Higher-Order Functions

- Functional languages treat functions as *first-class values*
- This means that, like any other value, a function can be passed as a parameter and returned as a result
- Functions that take values and variable are called *first order functions*
- Functions that take other functions as parameters or return functions are called *higher order functions*

Summation once again!

take the sum of the integers from a and b:

```
def sumInts(a: Int, b: Int) =  
  if (a > b) 0 else a + sumInts(a+1,b)
```

Summation once again!

take the sum of the integers from a and b:

```
def sumInts(a: Int, b: Int) =  
  if (a > b) 0 else a + sumInts(a+1,b)
```

If you want to sum the squares or cubes from a and b:

```
def sqr(x: Int) = x * x  
def sumSquares(a: Int, b: Int): Int =  
  if (a > b) 0 else sqr(a) + sumSquares(a + 1, b)  
  
def cube(x: Int) = x * x * x  
def sumCubes(a: Int, b: Int): Int =  
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Exercise

Define the sum of factorial from a and b

Summation

Exercise

Define the sum of factorial from a and b

Idea

Define a sum generic with the respect to the operation applied to each number?

```
def operation(x:Int) = ...  
  
def sumOperation(a: Int, b: Int) =  
  if (a > b) 0 else operation(a) + sumOperation(a+1,b)
```

Higher order functions

```
def sum(f: Int=>Int, a: Int, b: Int): Int =  
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type Int => Int represents a function which accepts an Int and returns an Int.

Higher order functions

```
def sum(f: Int=>Int, a: Int, b: Int): Int =  
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type Int => Int represents a function which accepts an Int and returns an Int.

```
def identity(x: Int) = x  
def sqr(x: Int) = x * x  
def cube(x: Int) = x * x * x  
def fact(x:Int) = ...  
println(sum(identity, 1, 10))  
println(sum(sqr, 1, 10))  
println(sum(cube, 1, 10))
```

Anonymous functions

Passing functions as parameters leads to the creation of many functions. Sometime is tedious. It can be avoided.

Like:

```
def name = "Angelo"; println(name)
```

can be written as

```
println("Angelo")
```

we want to define functions without an explicit name:
anonymous functions

Anonymous functions

Can be written as:

(Parameters) => Body

We can create "anonymous" functions on-the-fly! `x => x*x` is a function which takes an "x" and returns `x*x`

```
(x:Int)=> x *x
```

The parameter type can be omitted if the compiler can infer it:

```
x => x *x
```

```
println(sum(x=>x, 1, 10))
println(sum(x=>x*x, 1, 10))
println(sum(x=>x*x*x, 1, 10))
```

Rewrite sum with the tail recursion?

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

```
def sum(f: Int => Int, a: Int, b: Int): Int = {  
    def loop(a: Int, acc: Int): Int = {  
        if (a > b) acc  
        else loop(a + 1, acc + f(a))  
    }  
    loop(a, 0)  
}
```

Currying

Here is the definition from Wikipedia:

In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument. It was originated by Moses Schönfinkel and later re-discovered by Haskell Curry.

Let's try to do this in Scala!

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

sum can *return* a function that takes two Ints and return an Int

```
def sum(f: Int => Int): (Int,Int) => Int = { ... }
```

sum takes a function f and return a function (Int,Int) => Int

Currying - returning functions

```
def sum(f: Int => Int): (Int,Int) => Int = {  
    def sumF(a: Int, b: Int): Int = {  
        if (a > b) 0  
        else f(a) + sumF(a + 1, b)  
    }  
    sumF  
}
```

Currying - stepwise application

The basic sum functions can be defined without parameters:

```
def sumInts = sum(x=>x)
def sumCubes = sum(x=>x*x)
def sumFactorial = sum(fact)
```

sumInts(3,4) ...

or we could write

```
sum(x=>x)(3,4)
```

Multiple parameter list

```
def sum(f: Int => Int, a: Int, b: Int): Int =
```

Can be rewritten as:

```
def sum(f: Int => Int) : (Int, Int) => Int =
```

or equivalently, by using multiple parameter lists:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
```

the advantage wrt the first is that we can pass only one argument like `sum(cube)` the advantage wrt the second is that we can use `a` and `b` directly in the body

Currying - two argument functions

```
def addA(x: Int, y: Int): Int =  
  x + y  
  
def addB(x: Int):Int=>Int =  
  y => x + y  
  
val a = addA(10, 20)  
  
val b = addB(10)(20)  
  
println(a)  
println(b)
```

Currying - three argument functions

```
def addA(x: Int, y: Int, z: Int) = x + y + z

def addB(x: Int): Int => (Int => Int) =
y => (z => x + y + z)

val a = addA(1, 2, 3)

val b = addB(1)(2)(3)

println(a)
println(b)
```

It is now easy to see how the idea can be generalized to N argument functions!

- ① write a function that calculates the product of the values for the points on a given interval
- ② write a function that calculates the product of the values of a function f for the points on a given interval
- ③ write the factorial in terms of product
- ④ can we write a more general function which generalizes both sum and product

- ① write a function that calculates the product of the values for the points on a given interval
- ② write a function that calculates the product of the values of a function f for the points on a given interval
- ③ write the factorial in terms of product
- ④ can we write a more general function which generalizes both sum and product

END 1

Methods on collections: Map/Filter/Reduce

code/a10.scala

Map applies a function on all elements of a sequence. **Filter** selects a set of values from a sequence based on the boolean value returned by a function passed as its parameter - both functions return a new sequence. **Reduce** combines the elements of a sequence into a single element.

More methods on collections

code/a11.scala

Block structure / Scope

code/a12.scala

The "y" in the inner scope shadows the "y" in the outer scope

Nested functions / functions returning functions

code/a13.scala

```
def fun():Int=>Int says "fun is a function which does not take  
any argument and returns a function which maps an Int to an Int.  
Note that it possible to have "nested" function definitions.
```

Lexical Closure

code/a14.scala

What does it print? 10 or 12?

Lexical Closure

- The function "fun1" returns a "closure".
- A "closure" is a function which carries with it references to the environment in which it was defined.
- When we call `f(10)`, the "add" function gets executed with the environment it had when it was defined - in this environment, the value of "y" is 1.

code/a15.scala

- "y" is now a parameter to fun1
- "fun1" returns an anonymous function - there is absolutely no difference between returning a "named" function and returning an anonymous function.

Simple closure examples

code/a16.scala

Simple closure examples

code/a17.scala

- The anonymous function "score => score >= threshold" is the closure here.
- How do you know that it is a closure? Its body uses a variable "threshold" which is not in its local environment (the local environment, in this case, is the parameter list consisting of a single parameter "score")

Some List operations

code/a20.scala

- Nil and List() are both "empty" lists
- a::b returns a new list with "a" as the first item (the "head") and remaining part b (called the "tail")

code/a32.scala

We are trying to write a function which behaves similar to the built-in "if" control structure in Scala ... does it really work properly? Let's try another example!

Non-strict evaluation

code/a33.scala

Non-strict evaluation

- The behaviour of "if" is "non-strict": In the expression "if (cond) e1 else e2", if "cond" is true e2 is NOT EVALUATED. Also, if "cond" is false, e1 is NOT EVALUATED.
- By default, the behaviour of function calls in Scala is "strict": In the expression "fun(e1, e2, ..., en)", ALL the expressions e1, e2 ... en are evaluated before the function is called.
- There is a way by which we can make the evaluation of function parameters non-strict. If we define a functions as "def fun(e1: => Int)", the expression passed as a parameter to "fun" is evaluated ONLY when its value is needed in the body of the function. This is the "call-by-name" method of parameter passing, which is a "non-strict" strategy.

Non-strict evaluation

code/a34.scala

code/a35.scala

How many times is the message "hello" printed? Is there some way to prevent unnecessary repeated evaluations?

Lazy val's

code/a36.scala

The program prints "hello" once, as expected. The value of the val "a" will be 10.

Lazy val's

code/a37.scala

Strange, the program does NOT print "hello"! Why? The expression which assigns a value to a "lazy" val is executed only when that lazy val is used somewhere in the code!

Lazy val's

code/a38.scala

Unlike a "call-by-name" parameter, a lazy val is evaluated only once and the value is stored! This is called "lazy" or "call by need" evaluation.

If an expression can be replaced by its value without changing the behaviour of the program, it is said to be referentially transparent

- All occurrences of the expression $1+(2*3)$ can be replaced by 7 without changing the behaviour of the program.
- Say the variable x (in a C program) has initial value 5. It is NOT possible to replace all occurrences of the statement ($x = x + 1$) with the value 6.

Pure Functions

code/a39.scala

In what way is our "withdraw" function different from a function like "sin"?

Pure Functions

- A pure function always computes the same value given the same parameters; for example, $\sin(0)$ is always 0. It is "Referentially Transparent".
- Evaluation of a pure function does not cause any observable "side effects" or output - like mutation of global variables or output to I/O devices.

What is Functional Programming?

A style of programming which emphasizes composing your program out of PURE functions and immutable data.

Theoretical foundation based on Alonzo Church's Lambda Calculus

In order for this style to be effective in the construction of real world programs, we make use of most of the ideas seen so far (higher order functions, lexical closures, currying, immutable and persistent datastructures, lazy evaluation etc)

What is Functional Programming?

Questions to ask:

- Is this the "silver bullet"?
- How practical is a program composed completely out of "pure" functions?
- What are the benefits of writing in a functional style?

Is FP the silver bullet?

- Of course, there are NO silver bullets!
(http://en.wikipedia.org/wiki/No_Silver_Bullet)
- Writing software is tough - no one methodology or technique is going to solve all your problems

How practical is a program composed completely out of pure functions?

- Very impractical - unless your aim is to fight the chill by heating up the CPU (which, by the way, is a "side effect")
- The idea is to write your program in such a way that it has a purely functional core, surrounded by a few "impure" functions at the outer layers

Functional core + "impure" outer layers

This example is simple and contrived, but it serves to illustrate the idea. It is taken from the amazing book "Functional Programming in Scala" by Paul Chiusano and Runar Bjarnason.

code/a40.scala

Note that "winner" is an impure function. We will now refactor it a little!

Functional core + "impure" outer layers

code/a41.scala

Now we have separated the computation from the display logic;
"maxScore" is a pure function and "winner" is the impure function
at the "outer" layer!

Benefits of functional style - easy reuse, easy testing

What if we wish to find out the winner among a set of N players?
Easy!

```
val players = List(Player("Ram", 10),  
                   Player("John", 15),  
                   Player("Hari", 20),  
                   Player("Krishna", 17))  
  
println(players.reduceLeft(maxScore))
```

FP as "good software engineering"

- Pure functions are easy to re-use as they have no "context" other than the function parameters. (Think about re-using the "winner" function in our first version to compute the winner among N players).
- Pure functions are also easy to test. (Think about writing an automated test for the "winner" function in the first version).
- Think of FP as "Good Software Engineering"!

- If your function modifies an object which is accessible from many other functions, the effect of calling the function is much more complex to analyse because you now have to analyse how all these other functions get affected by the mutation.
- Similarly, if the value computed by your function depends on the value of an object which can be modified by many other functions, it no longer becomes possible to reason about the working of the function by only looking at the way the function's parameters are manipulated.
- The evaluation of pure functions can be done by a very simple process of "substitution".

Why mutability is tricky - an example

code/a43.java

Why mutability is tricky - an example

This Java example is taken from Prof.Dan Grossman's (University of Washington) excellent coursera.org class on "Programming Languages". Can you identify the problem with the code?

Why mutability is tricky - an example

What if client code does this?

```
p.getAllowedUsers[0] = p.currentUser()  
p.useTheResource()
```

The user can happily use the resource, even if he does not belong to the group of "allowed" users! The fix is to return a copy of "allowedUsers" in the function "getAllowedUsers".

What if we had used an immutable Scala list for representing "allowedUsers"? This problem would never have occurred because an attempt to modify "allowedUsers" simply returns a new object without in any way altering the original!

How it gets even more tricky in the context of concurrency

- Multi core CPU's are becoming commonplace
- We need concurrency in our code to make effective use of the many cores
- This throws up a whole bunch of complex problems
- A function can no longer assume that nobody else is watching when it is happily mutating some data

How it gets even more tricky in the context of concurrency

code/a44.scala

How it gets even more tricky in the context of concurrency

What happens if the functions `changeDate()` and `showDate()` run as two independent threads on two CPU cores? Will `showDate()` always see a consistent date value?

The traditional approach to maintaining correctness in the context of multithreading is to use locks - but people who do it in practice will tell you it is extremely tricky business.

It is claimed that one of the reasons for the resurgence of FP is the emergence of multi-core processors and concurrency - it seems like FP's emphasis on pure functions and immutability is a good match for concurrent programming.

Moving ahead ...

- Join Prof.Grossman's programming languages class on Coursera: <https://www.coursera.org/course/proglang> and learn more about functional programming using SML and Racket
- Join Prof.Odersky's Functional Programming with Scala course on Coursera for an introduction to both Scala and FP: <https://www.coursera.org/course/progfun>
- Watch the classic "SICP" lectures by Abelson and Sussman: <http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/>
- Learn from "HTDP" if you want something simpler: <http://htdp.org/>

- "Programming in Scala, 2nd edition" by Martin Odersky, Bill Venners, Lex Spoon. Perhaps the best introductory book on the language.
- "Functional Programming in Scala" by Paul Chiusano and Runar Bjarnson - this will soon become a classic!
- "Scala in Depth" by Joshua Suereth. This is an advanced book on Scala
- "Learn You a Haskell for Great Good" -
<http://learnyouahaskell.com/>. Scala programmers can definitely benefit from an understanding of Haskell - this is an amazing book which will get you started with Haskell.
- Many others: check out <http://blog.typesafe.com/week-of-scala-with-manning-publications>

Other resources

- "Learning functional programming without growing a neckbeard" - http://marakana.com/s/post/1354/learning_functional_programming_scala_video
- "Scala Days 2012" Videos - <http://skillsmatter.com/event/scala/scala-days-2012>
- "Out of the tar pit" - paper by Ben Mosely and Peter Marks (google it)
- "Persistent Data Structures and Managed References" - talk by Rich Hickey (author of the Clojure programming language).
<http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey>
- "Can programming be liberated from the von Neumann style" - by John Backus. http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf

SCALA- collezioni e metodi

Info 3 AA 18/19

Angelo Gargantini

Immutability

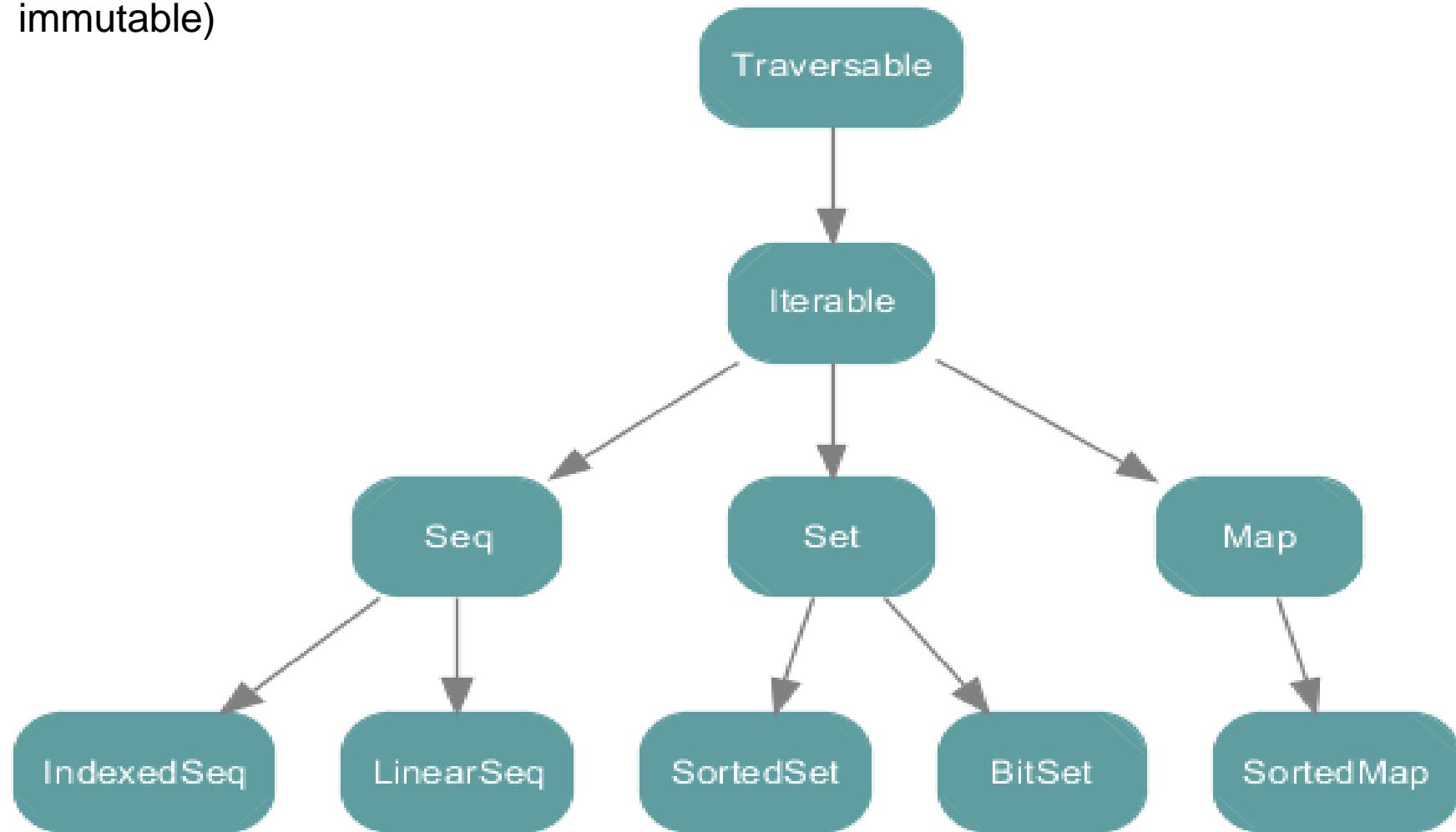
- Why?
 - Immutable objects are automatically thread-safe
(you don't have to worry about object being changed by another thread)
 - Compiler can reason better about immutable values -> optimization
 - Steve Jenson from Twitter: “*Start with immutability, then use mutability where you find appropriate.*”

Collezioni

- Mutable and Immutable Collections
- Scala collections systematically distinguish between mutable and immutable collections.
 - A mutable collection can be updated or extended in place.
 - Immutable collections, by contrast, never change.
 - You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged.

collections in package scala.collection

Higher level (both mutable and immutable)



Immutable



List

- Lists are immutable (= contents cannot be changed)
- List[**String**] contains Strings

```
val lst = List("b", "c", "d")
lst.head          // "b"
lst.tail          // List("c", "d")
val lst2 = "a" :: lst // cons operator
```

Nil = synonym for empty list

```
val l = 1 :: 2 :: 3 :: Nil
```

- List concatenation

```
val l2 = List(1, 2, 3) :: List(4, 5)
```

Foreach

```
val list3 = List("mff", "cuni", "cz")
```

- Following 3 calls are equivalent

```
list.foreach((s : String) => println(s))
```

```
list.foreach(s => println(s))
```

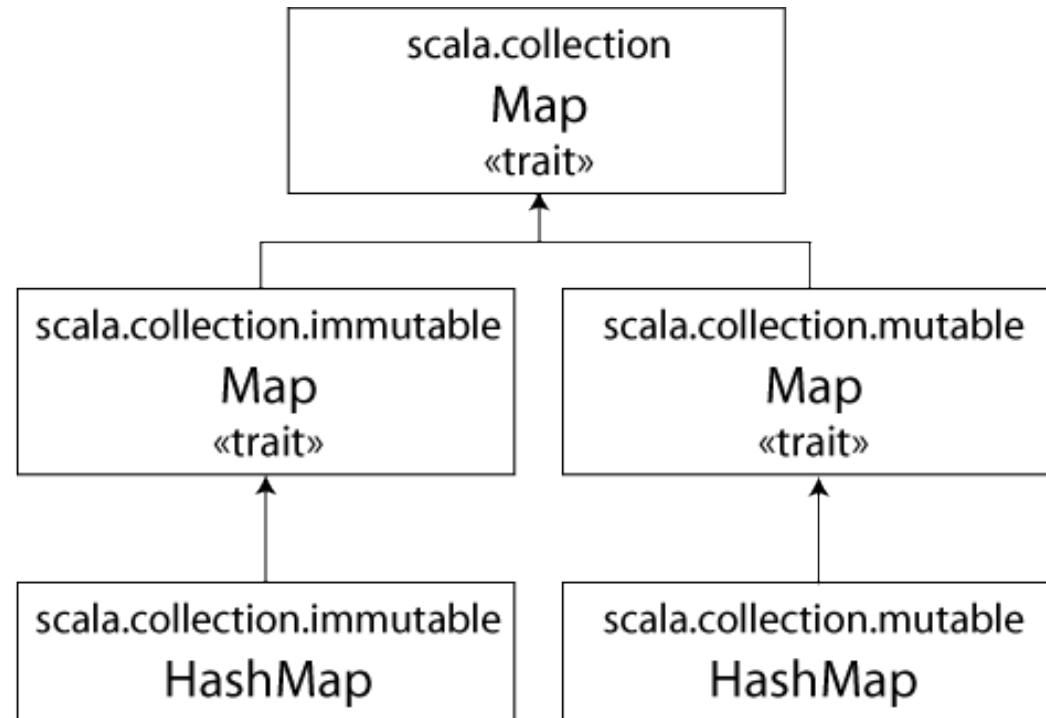
```
list.foreach(println)
```

- **For comprehensions**

```
for (s <- list) println(s)
```

```
for (s <- list if s.length() == 4) println(s)
```

Maps and Sets



MAPS

```
import scala.collection._
```

```
val cache = new mutable.HashMap[String, String];  
cache += "foo" -> "bar";
```

```
val c = cache("foo");
```

- The rest of Map and Set interface looks as you would expect

Mutable List: ListBuffer

- ListBuffer[T] is a mutable List
- Like Java's ArrayList<T>

```
import scala.collection.mutable._
```

```
val list = new ListBuffer[String]  
list += "vicky"  
list += "Christina"
```

```
val str = list(0)                                += add element
```

(*i*) to access the i-th
element

scala.Seq

- scala.Seq is the supertype that defines methods like:
 - filter, fold, map, reduce, take, contains, ...
- List, Array, Maps... descend from Seq

From Java to Scala

- Iterator <=> java.util.Iterator
- Iterator <=> java.utilEnumeration
- Iterable <=> java.langIterable
- Iterable <=> java.utilCollection
- mutable.Buffer <=> java.utilList
- mutable.Set <=> java.utilSet
- mutable.Map <=> java.utilMap
- mutable.ConcurrentMap <=>
java.util.concurrent.ConcurrentMap

algorithms

Iterate – foreach function

- Every collection in Scala's library defines (or inherits) a `foreach` method

```
val names = List("Daniel", "Chris", "Joseph")
  names.foreach { name =>
    println(name)
  }
```

- `foreach` is a “higher-order” method, due to the fact that it accepts a parameter which is itself another method
 - `name => println(name)`

```
names.foreach(println)
```

Foreach - istruzione

- There are times that we just want to use a syntax which is similar to the for-loops available in other languages.

```
val nums = List(1, 2, 3, 4, 5)
```

```
var sum = 0
for (n <- nums) {
    sum += n
}
```

Oppure se volessi usare il metodo:

```
var ss = 0;
def sinc(x:Int) ={
    ss += x;
}

nums.foreach(sinc)
```

Folding

- Looping is nice, but sometimes there are situations where it is necessary to somehow combine or examine every element in a collection, producing a single value as a result.

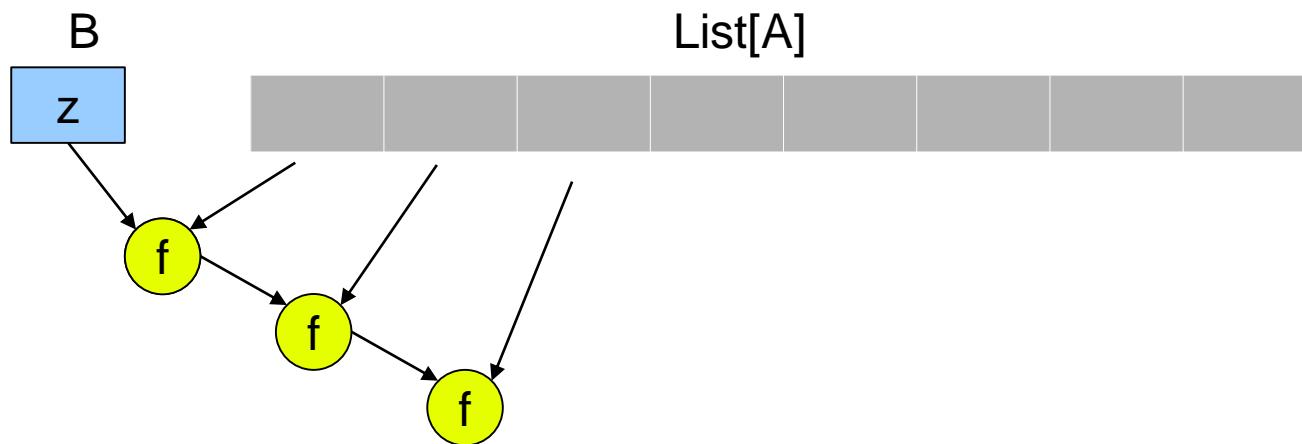
- For List[A]:

```
def foldLeft [B] (z: B) (f: (B, A) => B) : B
```

- The foldLeft function goes through the whole List[A], from head to tail, and passes each value to f. For the first list item, that first parameter, z, is used as the first parameter to f. For the second list item, the result of the first call to f is used as the B type parameter.

foldLeft

```
def foldLeft [B] (z: B) (f: (B, A) => B) : B
```



z: first element
f: function to be applied

Risultato finale

Folding Esempi

Somma di tutti i numeri in nums

```
val sum = nums.foldLeft(0)((total, n) => total + n)
```

oppure

```
def myf(x: Int, y: Int) = x+y
```

```
val sum = nums.foldLeft(0)(myf)
```

Fold --> Reduce

- Fold has a closely related operation in Scala called “reduce” which can be extremely helpful in merging the elements of a sequence where leading or trailing values might be a problem. Consider the ever-popular example of transforming a list of String(s) into a single, comma-delimited value:
- ```
var nn = List("a", "b", "c") // voglio stampare "a,b,c"
println(nn.foldLeft("") ((x, y) => x + "," + y))
```

Stampa però: ,a,b,c

# Reduce

- Solution: use a reduce, rather than a fold.
  - Reduce distinguishes itself from fold in that it does not require an initial value to “prime the sequence”. Rather, it starts with the very first element in the sequence and moves on to the end.

```
def reduceLeft (f: (A, A) => A) : A
```

## Esempi

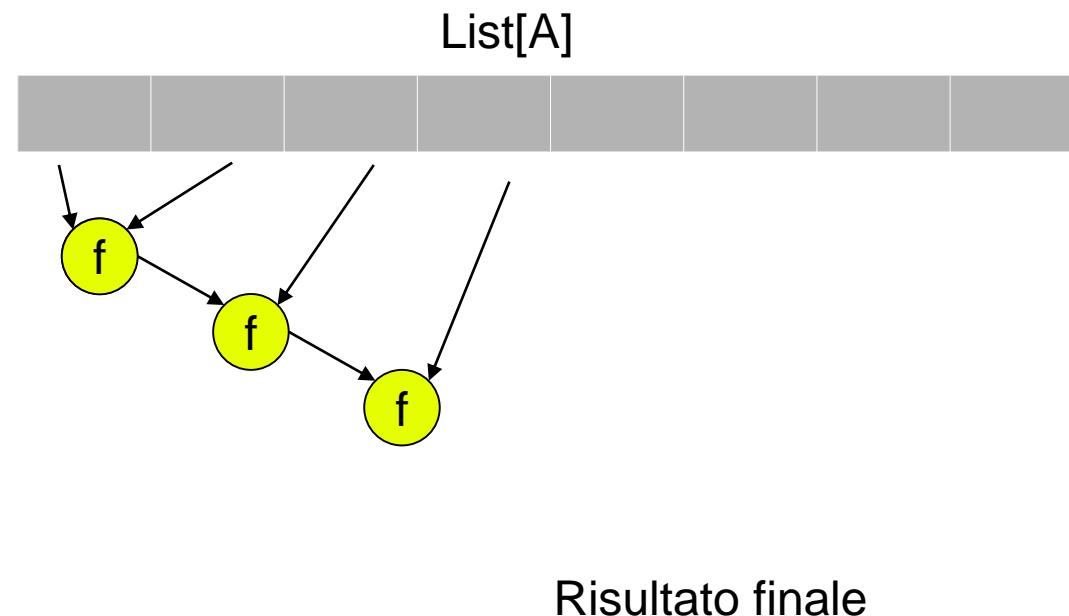
```
println(nn.reduceLeft((x, y) => x + ", " + y))
--> a,b,c
```

- Altro esempio: calcolo del max in lista

```
l.reduceLeft((x, y) => if (x > y) x else y)
```

# foldLeft

```
def reduceLeft (f: (A, A) => A) : A
```



# Esempi di Folder/Reduce

- Fai la somma/prodotto dei numeri in una lista
- Restituisci la stringa piu' lunga
- Trova la dimensione della stringa piu' lunga
- ....

# Filter/map

- fold can be an extremely useful tool for applying a computation to each element in a collection and arriving at a single result
- if we want to apply a method to every element in a collection in-place (as it were), creating a new collection of the same type with the modified elements?
- Esempi, data una lista, costruire la lista dei doppi

```
var ll = List(3, 4, 5)
// lista dei doppi
def doppio(x: Int) = 2 * x
println(ll.map(doppio))
```

# Esempi + Filter

- La lista delle lunghezze di una stringa

```
nomi.map(x => x.length())
```

- Filter
- Alcune volte voglio estrarre delle liste filtrando il contenuto
- Ad esempio: data una lista estrarre la lista pari

```
def pari = (x: Int) => (x % 2 == 0)
println(ll.filter(pari))
```

-

# Using Map+Reduce

- Spesso si usa map insieme a reduce:
  - Con map trasformo i dati per renderli più trattabili
  - Con reduce ottengo un dato sintetico
- Sono algoritmi che si possono parallelizzare
  - Vedi google framework mapreduce
  - <http://it.wikipedia.org/wiki/MapReduce>
- Vedi
  - <http://spark.apache.org/>