Programmazione Avanzata

2024-2025

Introduzione

Il linguaggio di programmazione ideale facilita la scrittura di programmi succinti e chiari. Questo ne permette la comprensione, modifica e mantenimento durante l'intero ciclo di vita. Aiuterà inoltre i programmatori a gestire l'interazione tra le componenti di un sistema software complesso. A un software è richeisto di essere affidabile, manutenibile ed efficiente. Ad un linguaggio di programmazione si chiede di essere scrivibile, ovvero di permettere la stesura di una soluzione in modo non contorto; leggibile, ovvero di permettere di riconoscere la correttezza o gli errori direttamente dalla sintassi, senza eseguire; semplice, ovvero facile da apprendere e applicare; sicuro, ovvero contenere protezioni contro la scrittura di codice malevolo; robusto, ovvero resistente ad eventi indesiderati.

Agli inizi, la programmazione era effettuata direttamente in codice macchina per ottenere programmi piccoli ed efficienti. È negli anni '50 che emergono i primi due linguaggi, Fortran e Cobol. Il primo permette di scrivere programmi in forma matematica, il secondo è adatto all'uso bancario. Le necessità dei programmatori sono cambiate nel corso degli anni. Funzionalità e paradigmi un tempo considerati inefficienti, come la ricorsione e la programmazione ad oggetti, sono oggi diventati la norma. Le caratteristiche del linguaggio sono, in ogni caso, definite al punto d'incontro tra le necessità umane del programmatore e le necessità tecniche dell'architettura di Von Neumann della macchina sottostante. Possiamo distinguere diversi paradigmi. La programmazione procedurale sceglie la *routine* come unità base per la modularizzazione. La programmazione imperativa si basa su istruzioni, definite a passaggi, che modificano valori. La programmazione funzionale segue un approccio simile a quello matematico, basato su espressioni e funzioni. La programmazione a oggetti si basa sul concetto di classe come unità base. La programmazione *abstract data type* usa i tipi astratti come unità base.La programmazione dichiarativa cerca di definire il problema tramite regole, invece di descrivere i passaggi per trovare la soluzione.

Computabilità

Un programma per computer è interpretabile come funzione matematica dello stato della macchina prima dell'esecuzione e degli ingressi forniti dall'utente. Esso può implementare solo funzioni computabili, ovvero in grado di produrre un risultato. Questi può essere impossibile da raggiungere per errori nella funzione, oppure a causa di un tempo di esecuzione infinito. Alcune funzioni possono essere computabili in principio, ma non in pratica (se il tempo di computazione eccede limiti materiali). Si definisce *funzione parziale* una funzione definita solo per certi argomenti. Usando le definizioni matematiche:

- funzione computabile: $f: A \to B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano le seguenti condizioni:
  - $< x, y > \in f$ e $< x, z > \in f \ \to \ y = z$
  - $\forall x \in A, \quad \exists y \in B \ / \ < x, y > \in f$
- funzione parziale $f: A \to B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano la seguente condizione:
  - $< x, y > \in f$ e $zx, z > \in f \ \to \ y = z$

Possiamo fornire una definizione alternativa di computabilità. Una funzione è computabile se esiste un algoritmo che permetta di produrre il risultato desiderato per qualsiasi ingresso appartenente al dominio. Anche quando l'algoritmo esiste, la sua implementabilità dipende dal linguaggio di programmazione scelto. La classe delle funzioni computabili sui numeri naturali coincide con la classe delle funzioni parziali ricorsive. Questo perché la ricorsione è essenziale nella computazione, e perché la maggior parte delle funzioni è parziale.

Un'altra definizione di computabilità si basa sul concetto di macchina di Turing. Una macchina di Turing è un sistema bicomponente. Il primo elemento è un nastro, diviso in celle di memoria, sul quale sia possibile leggere, scrivere e muoversi di una cella per volta. Il secondo elemento è un controllore a stati finiti, che opera sul nastro per leggerlo, per scriverci o per muovervisi di una cella. Una funzione sui numeri naturali è computabile con un metodo efficace se e solo se è computabile da una macchina di Turing. Questo teorema è dimostrabile con tre dimostrazioni: Alonso Church, Alan Turing e Calcolo Lambda. Tutti i linguaggi di programmazione sono *Turing-complete*.

In una quarta definizione, la computabilità è riconducibile all'*halting problem*, che consiste nel determinare se un programma terminerà in corrispondenza di un certo ingresso. Possiamo associare il problema ad una funzione f_{halt} a

doppio ingresso (programma, input) che ritorna *halt* o $\neg halt$ se il programma termina o meno. Supponendo di avere un programma in grado di risolvere il problema, ovvero che abbia lo stesso output di f_{halt}, possiamo utilizzarlo per creare un programma che a volte non termina.

Compilatori, calcolo lambda, semantica denotativa, divisione dei linguaggi

Un compilatore traduce il programma in istruzioni macchina, mentre un interprete traduce ed esegue allo stesso tempo. Il compilatore è divisibile in componenti. Il *lexical analyzer* raggruppa le istruzioni in *token*. Il *syntax analyzer* o *parser* raggruppa i *token* in espressioni, *statement* e dichiarazioni, in vase a regole grammaticali. Il prodotto del *parser* è il *parse tree*, una struttura dati che rappresenta il programma. Il *semantic analyzer* applica regole e procedure aggiuntive in base al contesto delle espressioni, come ad esempio il *type checking*, producendo un *augmented parse tree*. L'*intermediate code generator* produce una prima versione non ottimizzata del codice, in un formato chiamato *intermediate representation*. Il *code optimizer* elimina sottoespressioni, sostituisce variabili duplicate, rimuove istruzioni inutilizzate e rimpiazza le chiamate a funzioni brevi con il rispettivo codice (*inlining*) quando esso è più efficiente. Il *code generator* converte il codice intermedio in codice macchina per il *target* desiderato.

Distinguiamo tra sintassi (il testo di un programma) e semantica (la funzionalità che rappresenta). Una grammatica è composta da un simbolo iniziale, un insieme di simboli non temrinali, un insieme di terminali e un insieme di regole di produzione. Essa fornisce un metodo per definire un insieme infinito di espressioni. I non terminali sono i simboli utilizzati per esprimere la grammatica, mentre i terminali sono i simboli che appaiono nel linguaggio. Una grammatica è detta *ambigua* se la stessa espressione ammette più di un *parse tree*. I linguaggi umani uniscono ambiguità, frasi imperative, dichiarative, e interrogative. I linguaggi imperativi uniscono dichiarazioni e assegnamenti. 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.

Il lambda calculus è una notazione per descrivere la computazione, composta da tre parti. La prima è una notazione per descrivere le funzioni. La seconda è un meccansimo di prova per descrivere equazioni tra epsressioni. La terza è un insieme di regole di calcolo chiamate riduzioni. I due concetti principali del calcolo lambda sono le astrazioni lambda, per cui se M è un'espressione, $\lambda x.M$ è la funzione ottenuta trattando M come una funzione di x, e l'applicazione, ovvero l'anteposizione di un'espressione davanti ad un'altra per ottenere la composizione. Ad esempio, in $(\lambda x.x)M = M$, applichiamo una funzione identità all'espressione M. Un linguaggio di programmazione è interpretabile come un'applicazione del calcolo lambda, ovvero l'unione del calcolo lambda puro con tipi di dati aggiuntivi. Introduciamo ora il concetto di assegnazione delle variabili. Si dice libera una variabile che non è dichiarata nell'espressione (recuperare la definzione di algebra e logica). Nel calcolo lambda, al contrario di quanto avvenga nei linguaggi di programmazione procedurali come C, l'assegnamento di variabili non ha alcun effetto secondario ed è puramente funzionale.

Gestione della memoria

Lo *stack* serve soltanto per la ricorsione, mentre l'*heap* serve soltanto per le strutture dati dinamiche. Quando queste due funzionalità non sono necessarie, la memoria può essere gestita staticamente. Ad esempio il linguaggio FORTRAN, non permettendo la ricorsione, non aveva un record di attivazione. La memoria occupata da un programma era stimabile in modo esatto al momento della compilazione. I linguaggi moderni, invece, permettendo la ricorsione e l'allocazione dinamiche, necessitano di record di attivazione e di una gestione più complessa della memoria. Molti linguaggi sono *block-structured*, ovvero le variabili sono accessibili solo all'interno del loro blocco (variabili locali) o di sottoblocchi interni ad esso. Nell'analisi del ciclo di vita di variabili, distinguiamo tra *scope*, ovvero la regione di spazio (a livello di blocchi) in cui la variabile è attiva, e *lifetime*, ovvero tempo di allocazione. Queste due metriche possono non corrispondere. Ad esempio, se dichiaro una variabile in un blocco interno che abbia lo stesso nome di quella esterna, ottengo un *"hole in scope"* in cui la variabile esterna è ancora attiva ma non accessibile. Ogni blocco vede come globali tutte le variabili dichiarate in blocchi di livello superiore. Lo spazio in memoria viene allocato all'ingresso nel blocco, e deallocato all'uscita. I blocchi possono essere legati a funzioni, *inline*, oppure legati a istruzioni quali controllo di flusso e cicli. C e C++ non permettono la dichiarazione di funzioni locali innestate.

Viene creato un record di attivazione ad ogni ingresso in un blocco. Ogni record di attivazione contiene, in ordine, un *control link* ovvero un puntatore al record precedente sullo stack, delle variabili locali e dei risultati intermedi. L'indirizzo nell'*environment pointer* è al *control link* del record in cima allo *stack*. Una macchina standard contiene dei registri standard, il codice e un registro chiamato *program counter* o *instruction pointer* con l'indirizzo dell'istruzione corrente, i dati (*stack* e *heap*) e un *environment pointer* o *stack pointer* con l'indirizzo della cima dello stack. Le chiamate a funzioni richiedono il passaggio dei parametri, il salvataggio dell'indirizzo di ritorno, il salvataggio di variabili locali e risultati intermedi, e l'allocazione di spazio per il valore di ritorno. I parametri delle funzioni possono essere valutati al momento del passaggio, oppure essere lasciati come funzioni per la *lazy evaluation*, in base al tipo di linguaggio. Il passaggio può avvenire per *reference* o per valore. Il passaggio per valore, richiedendo di copiare il valore del parametro, è più sicuro ma più lento. Riduce però il problema dell'*aliasing*, ovvero il puntamento allo stesso indirizzo di memoria da parte di più variabili. In base alla terminologia il valore effettivo di una variabile può prendere il nome di *R-value* mentre il suo indirizzo è denominato *L-value*.

```c
#include <cstdio>

void f() {
    int x = 5;
    int y = 3;
}

int main(int argc, char *argv[]) {
    f();
    printf("Hello World!");
    return 0;
}
```

```
f():
        sub     sp, sp, #16 ; due byte in più: stack pointer e frame pointer
        mov     w8, #5
        str     w8, [sp, #12]
        mov     w8, #10
        str     w8, [sp, #8]
        add     sp, sp, #16
        ret

main:
        sub     sp, sp, #48
        stp     x29, x30, [sp, #32]
        add     x29, sp, #32
        mov     w8, wzr
        str     w8, [sp, #12]
        stur    wzr, [x29, #-4]
        stur    w0, [x29, #-8]
        str     x1, [sp, #16]
        bl      f()
        adrp    x0, .L.str
        add     x0, x0, :lo12:.L.str
        bl      printf
        ldr     w0, [sp, #12]
        ldp     x29, x30, [sp, #32]
        add     sp, sp, #48
        ret

.L.str:
        .asciz  "Hello World!"
```

In C++ è possibile salvare *reference*. Ad esempio, `int &x = y` crea in `x` una *reference* non modificabile a `y`. Passando per *reference* i parametri per le funzioni, al contrario di quello che avviene con il passaggio per valore, si creano *side effects*. Una versione ibrida è il passaggio per puntatore. L'indirizzo viene passato per copia, ma poi tramite di esso si può modificare la variabile originale come se fosse una *reference*. Per verificare se un linguaggio supporta veramente il passaggio per reference si può provare a costruire una funzione per scambiare il contenuto di due variabili. Non possiamo infatti scambiare il contenuto di due variabili passate per copia, perché la modifica avverrebbe solo sulle copie locali, e verrebbe in ogni caso deallocata al termine dell'esecuzione della funzione. In C++ è possibile scambiare variabili passate per reference. In Java questo non funziona con i tipi base, che sono sempre passati per copia.

```c
int f(int a, int b) {
    if (b==0)
        return a;
    else
        return f(b, a%b);
}

int main(int argc, const char *argv[]) {
    f(15,10);
    return 0;
}
```

```
f:
```

```
        sub     sp, sp, #32
        stp     x29, x30, [sp, #16]   ; store pair, x30=return address
        add     x29, sp, #16    ; x29=frame pointer
        str     w0, [sp, #8]
        str     w1, [sp, #4]
        ldr     w8, [sp, #4]
        cbnz    w8, .LBB0_2
        b       .LBB0_1
.LBB0_1:
        ldr     w8, [sp, #8]
        stur    w8, [x29, #-4]
        b       .LBB0_3
.LBB0_2:    ; il risultato è il resto (divide intero, rimoltiplica, sottrae)
        ldr     w0, [sp, #4]
        ldr     w8, [sp, #8]
        ldr     w10, [sp, #4]
        sdiv    w9, w8, w10
        mul     w9, w9, w10
        subs    w1, w8, w9
        bl      f
        stur    w0, [x29, #-4]
        b       .LBB0_3
.LBB0_3:
        ldur    w0, [x29, #-4]
        ldp     x29, x30, [sp, #16]
        add     sp, sp, #32
        ret

main:
        sub     sp, sp, #48
        stp     x29, x30, [sp, #32]
        add     x29, sp, #32
        mov     w8, wzr
        str     w8, [sp, #12]
        stur    wzr, [x29, #-4]
        stur    w0, [x29, #-8]
        str     x1, [sp, #16]
        mov     w0, #15
        mov     w1, #10
        bl      f
        ldr     w0, [sp, #12]
        ldp     x29, x30, [sp, #32]
        add     sp, sp, #48
        ret
```

Il compilatore può ottimizzare il salvataggio dei valori di ritorno tenendoli in un registro invece di assegnarli allo stack.

Soluzione vecchia (ottimizzata meglio):

Indirizzo	Contenuto	Descrizione
0xF174	00 00 00 0A	10
0xF178	00 00 00 0F	15
0xF17C		Return value
0xF180	...	x29: stack pointerframe pointer
0xF188	...	x30: link registerreturn address

Indirizzo	Contenuto	Descrizione
154	00 00 00 05	5
158	00 00 00 0A	10
15c	...	Return value
160	...	SP

Indirizzo	Contenuto	Descrizione
168	...	LR

Indirizzo	Contenuto	Descrizione
134	00 00 00 00	0 (b)
138	00 00 00 05	5 (a)
13c	...	Return value
140	...	x29 sp
148	...	x30 lr / ra

Gli array a dimensione predefinita sono stack-allocated sia in C che in Java. Quando si passano struct a funzioni, l'intero struct viene copiato nello stack.

Gli array a dimensione fissa sono *stack-allocated* sia in C che in Java. Nel caso di Java, questo è vero soltanto per i tipi base. In C/C++, quando uno struct è argomento (per valore) di una funzione, viene copiato per intero nello stack. Per questo, in questo caso, può essere conveniente passare gli struct per *reference*, pur facendo attenzione alla possibilità di *side effects*. In Java il passaggio per valore di oggetti non è invece possibile.

Si analizzi ora un piccolo esempio di funzione per cambiare il valore dei puntatori, riscritta in due versioni:

```
// VERSIONE 1

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;
}
```

Ritorna 0. `q` punta ancora ad `m`.

Scriviamo una seconda versione, in cui si passa un puntatore a puntatore (o si passa un puntatore per reference, volendolo interpretare così):

```
// Versione 2

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;
}
```

Ritorna 10. `q` punta ad `y`.

Variabili globali

Forniamo un esempio di codice dove sia deliberatamente creato un *hole in scope*, per creare ambiguità nell'uso della variabile `x` nel blocco più interno.

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

Analizziamo il contenuto delle variabili in vari momenti dell'esecuzione del codice. Al momento dell'outer block:

Variabile	Valore
x	1

All'esecuzione di `f(3)`:

Variabile	Valore
y	3
x	4

All'esecuzione di `g(12)`:

Variabile	Valore
z	12

Non è chiaro quale x usare. - *static scope*: variabili globali dal più vicino blocco intorno - *dynamic scope*: variabili dal record di attivazione più recente

L'*access link* è un meccanismo legato al record di attivazione, dedicato all'accesso alle variabili globali secondo le regole di *scope*. La maggior parte dei linguaggi di programmazione è dotata di un *access link* statico, ovvero risolto in compilazione. In C, ad esempio, l'*access link* di un blocco contiene puntatori sia all'*access link* del blocco precedente, che al *return address* del blocco delle variabili globali. In molti linguaggi è possibile dichiarare variabili slegate dal lifetime del proprio *scope* utilizzando la *keyword* `static`. In C e C++, quest'operazione è indistinguibile dalla dichiarazione di variabili globali, per mancanza di una categoria apposita. Le variabili dello *scope* corrente sono viste come locali, e tutte le altre degli *scope* più esterni sono interpretate come globali. Il compilatore cerca le variabili statiche e le carica per prime in memoria, allocandole su *heap*.

Nei linguaggi non sicuri come C e C++, è possibile effettuare un tipo di attacco noto come *buffer overflow*. Esso si basa sull'uscita dai confini di memoria di un *array*, dato che l'accesso non è controllato, per scrivere codice malevolo da qualche parte, e per modificare il *return address* del *record* di attivazione più recente in modo che punti a tale codice.

Tail recursion

Una funzione fa una chiamata tail ad un'altra funzione se il ritorno della prima è il ritorno della seconda. Questo si può determinare in compile time. In tal caso, si può procedere senza impilare le chiamate di funzione sullo stack. Può essere sovrascritto lo stesso record.

L'uso dello *stack* e dei *record* di attivazione è reso necessario solo e soltanto dall'uso della ricorsione. I *record* si impilano perché i valori di ritorno delle chiamate ricorsive devono essere salvati per processarli al momento del loro ritorno. Esiste un caso particolare, nel quale il valore di ritorno della chiamata ricorsiva coincide con il valore di ritorno della funzione chiamante. Quando questo avviene, non è necessario creare un nuovo record di attivazione. Questo tipo di ricorsione è chiamato *ricorsione tail*. È considerata desiderabile perché non riempie lo *stack*. Questo la rende equivalente all'uso di cicli dal punto di vista dell'impiego di memoria. In molti casi, funzioni ricorsive normali possono essere trasformate in funzioni di tipo *tail recursive* con piccoli adattamenti. Questo è ad esempio vero per il più classico esempio di funzione ricorsiva, ovvero il calcolo dei fattoriali:

```
// Versione standard
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
```

```
}

// Versione tail-recursive
int tail_factorial(int n, int accumulator) {
    if (n == 0)
        return accumulator;
    else
        return tail_factorial(n - 1, n * accumulator);
}
```

Esercizio In questo esercizio verifichiamo l'esecuzione della seguente funzione ricorsiva, che richiama sé stessa per quattro volte.

```
int pow_n(int a, int ex) {
    if (ex == 0) return 1;
    if (ex == 1) return a;
    return a * pow_n(a, ex-1);
}

int main(int argc, const char *argv[]) {
    pow_n(5);
}
```

È riportata l'occupazione dello stack alla terza chiamata, su un'architettura ARMv8 a 64 bit.

Indirizzo	Contenuto	Descrizione
ff318		x30
ff314		
ff310		x29
ff30c		
ff308	7d 00 00 00	7.16 + 13 = 112 + 13 = 125 = 25 * 5
304	05 00 00 00	a = 5
300	03 00 00 00	ex = 3
28c	1a 00 00 00	25
288	01 00 00 00	lr x30
284	68 3f 00 00	
280	01 00 00 00	fp x29
2ec	10 f3 df 6f	
2e8	1a 00 00 00	a * pow_n(a, ex-1)
2e4	05 00 00 00	a = 3
2e0	02 00 00 00	x = 2
2dc	05 00 00 00	valore di ritorno
2d8	01 00 00 00	lr x30
2d4	68 2f 00 00	
2d0	01 00 00 00	fp (sp) x29
2dc	f0 f2 df ef	
2c8	05 00 00 00	a * pow(a, ex-1)
2c4	05 00 00 00	a = 5
2c0	01 00 00 00	a=1

Tipizzazione

Una componente importante della sicurezza di un sistema (intesa come *safety*, protezione da falle intrinseche, e non come *security* ovvero resistenza agli attacchi esterni) è il sistema dei tipi. Definiamo *tipo* un insieme di valori omogenei e di operazioni su di esso definite. I tipi definiscono concetti che possono essere elementari (tipi base che rappresentano numeri, lettere, ...) o complessi (collezioni e classi, che sono composizioni di tipi base). L'uso dei tipi permette al compilatore di interpretare correttamente i dati in memoria, e di controllare che essi siano definiti e utilizzati correttamente dall'utente.

Un possibile errore a livello *hardware* può consistere nella confusione tra dati e programmi. Entrambe le categorie sono indistinguibilmente valori binari in memoria centrale. Come descritto in precedenza, questa ambiguità nella rappresentazione fisica può essere utilizzata deliberatamente per produrre un attacco che inserisca codice eseguibile malevolo in un'area altrimenti destinata ai dati. Esistono inoltre errori semantici. La rappresentazione binaria di uno

stesso valore è diversa a seconda che si tratti di un intero o di un numero a virgola mobile. La lettura di una variabile secondo la convenzione sbagliata porta ad errori aritmetici. Nei linguaggi di programmazione ad oggetti il controllo dei tipi deve imporre il rispetto della gerarchia di ereditarietà. Il membro di una sottoclasse può essere promosso a membro di una superclasse perché possiede tutti i campi e metodi necessari, ma non vale il contrario.

Un linguaggio di programmazione si dice *type safe* se non è possibile scrivere con esso un programma in grado di violare il suo sistema di tipi. Le violazioni possono includere la confusione tra tipi, la chiamata di dati come se fossero funzioni, o l'accesso a zone di memoria riservate ad altri dati. La proprietà di prevenzione di quest'ultima problematica è chiamata *memory safety*. C e C++ non sono *type safe* perché le operazioni di *casting* incontrollato e l'aritmetica dei puntatori permettono di violare sia la *type safety* che la *memory safety*. Questo perché C punta alla velocità e all'efficienza, lasciando al programmatore la responsabilità per la sicurezza. Possiamo rendere C più sicuro riducendo l'uso delle feature pericolose. Pascal è parzialmente *type safe* perché richiede la deallocazione manuale della memoria, con il conseguente problema dei *dangling pointers*. Java, Python e Lisp sono considerati *type safe*. Laddove non sia possibile usare linguaggi *type safe*, sono comunque disponibili *tool* esterni di analisi, pur sempre limitati dal precedente nominato problema di Turing.

Elenchiamo in dettaglio le problematiche che rendono C e C++ non *type safe*. Innanzitutto, il *casting* è incontrollato e permette il passaggio tra tipi di dimensioni incompatibili, con rischio di perdita di informazione e *overflow*. Permette inoltre di trasformare tipi per variabili in tipi per funzioni, e dunque di chiamare aree di memoria per dati come se fossero codice. Non solo non esistono meccanismi che impediscano il dereferenziamento dei puntatori nulli, ma addirittura lo standard del linguaggio non definisce cosa fare in tale evenienza. Il valore nullo del puntatore è semplicemente lo 0. La funzione `malloc` usata per l'allocazione dinamica restituisce il valore `null` quando fallisce, e dunque anch'essa può essere fonte di comportamento indefinito. In alcuni sistemi è il sistema operativo a restituire un *segmentation fault* quando si tenta di accedere ad un puntatore nullo, ma questa protezione si perde quando il puntatore nullo è usato per accedere ad una cella di un array diversa dalla prima. Problemi simili si presentano con l'algebra dei puntatori. Ad esempio, `*(p+i)` permette di accedere a celle contigue che potrebbero avere un tipo diverso rispetto a quello di `p`. Equivalentemente, `x = *(p+i)` può permettere di salvare in `x` variabili del tipo sbagliato. Infine esistono problematiche di accesso non valido. Le violazioni possono essere spaziali (*out of bound*, uscire dall'area di memoria assegnata ad un *array*), o temporali (*dangling pointers*, puntatori ad aree deallocate e potenzialmente riscritte). Per quanto riguarda l'accesso non valido, l'allocazione dinamica è meno affetta da questa problematica, ma anche l'allocazione statica può essere resa sicura evitando di non passare mai alla funzione chiamante riferimenti alle variabili locali di una funzione chiamata.

Il controllo a compile time è obbligato a rifiutare programmi potenzialmente validi perché non è possibile determinarne staticamente la correttezza (sarebbe come risolvere il problema di Turing). Il controllo a runtime non soffre di questo problema ma rallenta l'esecuzione. Java utilizza un approccio ibrido introducendo in compilazione dei controlli da effettuare in esecuzione in presenza di problemi di tipo non risolvibili staticamente (ad esempio casting tra classi imparentate).

Il controllo dei tipi può essere effettuato in compilazione o in esecuzione. ML, C, C++ e Java controllano i tipi in compilazione. Python e Lisp li controllano in esecuzione. Il controllo in compilazione deve necessariamente rifiutare programmi potenzialmente validi perché non è possibile determinarne staticamente la correttezza. Se il loro flusso di controllo è programmatico, infatti, determinarne a priori la traiettoria sarebbe equivalente alla risoluzione, impossibile, del problema di *halting*. Il controllo in esecuzione non soffre di questa problematica, ma la maggiore libertà di programmazione si paga in riduzione della velocità, a causa del rallentamento dovuto all'accertamento dinamico dei tipi. Un altro problema del controllo in esecuzione è che gli errori di tipo non vengono individuati fino al momento dell'esecuzione della porzione di codice che li contiene. Java impiega un approccio ibrido. I controlli che sono impossibili da risolvere staticamente sono deferiti al momento dell'esecuzione tramite l'iniezione, al momento della compilazione, di codice di controllo da eseguire dinamicamente. Un esempio di questa strategia riguarda il controllo della promozione a superclasse.

Una distinzione ulteriore tra linguaggi è tra tipizzazione forte e tipizzazione dinamica. Nel primo caso, il tipo di una variabile è fissato per tutto il suo ciclo di vita. Nel secondo, una variabile può cambiare tipo al momento del riassegnamento del suo valore. Alla prima categoria appartengono C, C++ e Java. Alla seconda categoria appartiene Python. Un concetto correlato è l'inferenza dei tipi. Il linguaggio può determinare automaticamente il tipo di una variabile in base al contenuto assegnatole dall'utente. L'algoritmo di inferenza del tipo lavora in tre fasi. Dapprima assassegna un tipo ad ogni espressione e sottoespressione, usando tipi noti. Dopodiché, genera vincoli sui tipi usando l'albero sintattico dell'espressione. Infine, risolve i vincoli per unificazione. Python inferisce dinamicamente i tipi, ma essi sono anche specificabili manualmente. La tipizzazione è dinamica ma ben definita. Essa si perde nella definizione delle funzioni, in cui non è necessario specificare né il tipo dei parametri né il tipo del valore ritorno. Vale in questo caso il concetto di *duck typing* (*if it walks like a duck and quacks like a duck, then it's a duck*), secondo il quale non è necessario lanciare errori di tipo fin quando i valori ricevuti rispettano il contratto minimo necessario, ovvero hanno a disposizione quei campi e quei metodi richiesti durante l'esecuzione. Java, dalla versione 10, introduce la keyword `var` per la dichiarazione di variabili con inferenza automatica del tipo. Dopo un primo assegnamento, il tipo rimane fisso e non modificabile.

Programmi sicuri in C

I linguaggi sicuri tendono ad avere prestazioni inferiori rispetto al C. Questo può avvenire per varie cause, tra cui l'*overhead* del *garbage collector* e del controllo dell'accesso alla memoria. Quest'ultimo aspetto aumenta anche l'occupazione di memoria stessa, perché richiede il salvataggio di informazioni aggiuntive sui tipi e sulle dimensioni degli array. Infine, dato che il C è storicamente il linguaggio più usato, eliminarlo dalla *codebase* di un progetto richiede la riscrittura estensiva di molto codice.

Il C è tuttora la principale scelta per la programmazione dei sistemi operativi e dei *driver*. Rispetto ad altri linguaggi, fornisce prestazioni elevate a parità di complessità algoritmica. Permette il controllo esplicito della memoria e la rappresentazione dei dati a basso livello. Inoltre, è compatibile con codice *legacy* e con una grandissima quantità di librerie scritte nel corso degli anni. Il prezzo da pagare è la grande quantità di problemi di sicurezza. Essi includono violazioni spaziali (possibilità di uscita dalle aree di memoria assegnate: *out of bounds*, *buffer overflow*) e temporali (accesso a variabili deallocate o uscite di scope: *dangling pointers*, perdita di riferimenti a memoria allocata: *memory leak*), ed errori dovuti al *casting* non controllato (overflow, trasformazione di dati in puntatori o funzioni e viceversa). Una possibile soluzione a tutti questi problemi si ottiene creando dialetti del C che ne restringano la funzionalità alle sole componenti sicure. Questo ovviamente accade a scapito dell'espressività. Un'altra famiglia di soluzioni deriva dall'aggiunta di controlli aggiuntivi, che non riducono l'espressività ma potrebbero non prevenire adeguatamente alcuni dei problemi di sicurezza. I controlli possono avvenire in esecuzione, aggiungendo peso alla runtime, oppure essere eseguiti staticamente sul codice prima o durante la compilazione. I problemi dell'analisi statica sono legati al problema di *halting*. Non potendo effettivamente prevedere l'effettivo percorso preso da un programma né la sua terminazione, i *tool* di analisi dovranno rischiare di accettare codice potenzialmente rischioso, o di rifiutare preventivamente codice che potrebbe rivelarsi correttamente funzionante. La scelta del compromesso è un problema a sé.

Esistono diverse soluzioni standard per rendere più sicuro il C. Alcune di esse, come `MISRA C`, dialetto del C utilizzato in ambito automobilistico, seguono la strada dell'aggiunta di regole aggiuntive in compilazione. Altre soluzioni si basano sull'analisi statica. Tra esse si distinguono `Splint`, un linter derivato dal precedente `LCLint` e ispirato al tool `lint` di Unix, e `cppcheck`, un *checker* statico per C++. Esistono librerie, come `Cyclone`, che aggiungono una runtime con controlli dinamici e garbage collection. Altri tool dinamici includono `Purify` di Rational/IBM, che aggiunge checking dinamico, e `Valgrind`, una libreria *open source* per generare *tool* di analisi dinamica personalizzati. Altri sistemi utilizzano invece il C come linguaggio intermedio, permettendo di scrivere il codice in un linguaggio più astratto, e convertendolo in codice C. In questa categoria rientra Vault, un linguaggio molto astratto. Rientra in questo raggruppamento anche anche SafeC, un *transpiler* che aggiunge controlli di sicurezza e `malloc` / `free` esplicite. Esistono infine *tool* che uniscono l'analisi statica all'introduzione di una *runtime* con controlli e *garbage collection*, come ad esempio `CCured`. Infine esistono librerie che, senza introdurre analisi statica o dinamica, forniscono strutture dati più sicure al C. Esse introducono, ad esempio, *array* con controllo all'accesso, puntatori sicuri e stringhe che rispettino il formato ISO/IEC 14651.

Esercizi sulla sicurezza

Esempio 1 In questo esempio di codice, la funzione `foo` alloca staticamente un intero in memoria, di valore pari al parametro `y`, e restituisce al chiamante l'indirizzo di tale variabile. Trattandosi però di una variabile locale, essa viene deallocata all'uscita da `foo`. Il suo valore resta leggibile finché non viene riutilizzato quello spazio sullo stack, causando comportamento non definito.

```c
#include<stdio.h>

int* foo(int y) {
    int h = y ;
    int* p = &h;
    return p;
}

int main(void) {
    int *p = foo(3);
    printf("%d\n", *p);
}
```

Chiamando `foo()` più volte, il programma si rompe. Verrà ritornato lo stesso indirizzo della prima chiamata, riscrivendo il contenuto di tutti i puntatori inizializzati mediante `foo`:

```c
int main(void) {
    int *p = foo(3);
    int *q = foo(4);
```

```
    int *r = foo(5);
    // Viene stampato 5 per tutti e tre i puntatori:
    printf("%d, %q, %r\n", *p, *q, *r);
}
```

Per risolvere il problema, sostituiamo l'allocazione statica con un'allocazione dinamica, che restituirà indirizzi diversi di memoria ad ogni chiamata:

```
#include <stdio.h>
#include <stdlib.h>

int* foo(int y) {
    int* h = (int*) malloc(sizeof(int));
    *h = y;
    return h;
}

int main(void) {
    int* p = foo(3);
    printf("%d\n", *p);
    return 0;
}
```

E se chiamassimo `free()`?

```
#include <stdio.h>
#include <stdlib.h>

int* foo(int y) {
    int* h = malloc(sizeof(int));
    *h = y;
    free(h);
    return h;
}

int main(void) {
    int* p = foo(3);
    printf("%d\n", *p);
    return 0;
}
```

Possiamo osservare valori casuali. Non è colpa di `printf()` perché anche cambiando l'ordine degli statement il risultato continua ad essere casuale. È possibile che si tratti di un meccanismo di sicurezza nell'implementazione della `free` per offuscare le aree liberate.

```
#include<stdio.h>
#include<stdlib.h>

int main(void) {
    long i = 0;
    long* i_ptr = (long*) i;
    printf("%ld", *i_ptr)
}
```

E qui?

```
char *p1 = NULL;
printf("%d", *p1);
```

Wild pointer (puntatore non inizializzato):

```
char *p2;
printf("%c\n", *p2);
printf("%p\n", p2);
```

Cerchiamo di stampare una stringa senza terminatore:

```
char *p3 = malloc(10*sizeof(char));
p3[0] = 'a';
```

```
free(p3);
printf("%c\n", *p3);
```

Esercizio 4.2:

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    char name[5];
    int age;

    scanf("%d", &age);
    scanf("%s", &name);

    printf("Your name is %s and you are %d years old", name, age);
}
```

L'idea è di mettere un nome più lungo di 4 caratteri per andare a sovrascrivere l'età. In pratica questo non succede. Debug:

```
#include<stdio.h>
#include<stdlib.h>

int main() {
    char name[5];
    int age;

    printf("Puntatore name: %p\n", &name);
    printf("Puntatore age: %p\n", &age);
    scanf("%d", &age);
    scanf("%s", &name);

    printf("Your name is %s and you are %d years old", name, age);
}
```

Possiamo notare dalla lettura dei valori che `age` viene messo prima di name nello stack. Cambiando l'ordine delle variabili otteniamo il comportamento sbagliato desiderato.

Programmazione a oggetti

I principi fondamentali della *programmazione orientata agli oggetti* sono:

1. *incapsulamento*: protegge i dettagli interni di un oggetto, rendendo visibili solo le operazioni necessarie.
2. *sottotipazione*: permette di creare nuovi tipi che estendono o modificano il comportamento di tipi esistenti.
3. *ereditarietà*: consente a una classe di ereditare attributi e metodi da un'altra classe.
4. *binding dinamico*: permette di determinare a runtime quale metodo deve essere eseguito, tipicamente attraverso l'override dei metodi.

Il concetto di *decoupling* si riferisce alla separazione tra l'*implementazione* e l'*interfaccia* dei metodi di un oggetto. Definiamo *interfaccia* l'insieme dei metodi esposti da un oggetto. L'*incapsulamento*, in particolare, nasconde l'implementazione all'utente, garantendo che quest'ultimo interagisca solo con l'interfaccia esposta.

Gli *attributi* di un oggetto sono spesso dichiarati privati per due motivi principali:

1. *sicurezza*: proteggere i dati da accessi non autorizzati.
2. *flessibilità*: permettere modifiche interne senza alterare l'interfaccia visibile dall'esterno.

È importante notare che ereditarietà e sottotipazione non sono sinonimi. In Java, questi due concetti coincidono, ma in linguaggi che supportano il *duck typing*, come Python, la sottotipazione si basa sulla somiglianza delle interfacce piuttosto che sulla discendenza del codice.

In OOP si parla di messaggi, che sono essenzialmente chiamate a metodi. Il processo di sviluppo di un *software* orientato agli oggetti può essere suddiviso in quattro passi:

1. *identificare gli oggetti*: a un certo livello di astrazione.
2. *identificare la semantica:* ovvero il comportamento degli oggetti.

3. *definire le relazioni*: tra gli oggetti.
4. *implementare gli oggetti*: in modo iterativo, sia top-down che bottom-up.

In C l'incapsulamento può essere ottenuto utilizzando gli *header file*. Bisogna però ricordare che tipi creati con `typedef` sono comunque visibili nell'*header*, permettendo all'utente di capire la struttura interna e potenzialmente violare l'incapsulamento. Un esempio pratico è il seguente contatore:

```
typedef int Contatore;

void incrementaContatore(Contatore *c);
void riduciContatore(Contatore *c);
int getContatore(Contatore *c);
```

Un'implementazione alternativa sicura prevede l'uso di una variabile statica dichiarata nel modulo, non visibile nell'*header*. Questo permette di avere un solo contatore per volta. L'header `contatore.h` è così strutturato:

```
void incrementaContatore();
void riduciContatore();
int getContatore();
```

Mentre il modulo `contatore.c`:

```
static int contatore = 0;

void incrementaContatore() {
    contatore++;
}

void riduciContatore() {
    contatore--;
}

int getContatore() {
    return contatore;
}
```

La soluzione ottimale prevede un corretto incapsulamento che permette di avere più contatori. Si mette nell'*header* la definizione di un puntatore di tipo `Counter`, mentre la definizione di `Counter` rimane chiusa nel modulo. Questo è il modo di definire Abstract Data Types in C. Osserviamo come si modificherebbe in questo caso `contatore.h`:

```
typedef Contatore* counterRef;

void incrementaContatore(CounterRef c);
void riduciContatore(CounterRef c);
int getContatore(CounterRef c);
```

Per un incapsulamento ancora più rigoroso, si può usare un puntatore opaco (`void*`):

```
typedef void* counterRef;

void incrementaContatore(CounterRef c);
void riduciContatore(CounterRef c);
int getContatore(CounterRef c);
```

In generale, un tipo opaco è un tipo specificato in maniera incompleta dalla propria interfaccia, permettendo la manipolazione solo mediante l'interfaccia.

Il *polimorfismo* consente di usare oggetti di tipi diversi nello stesso modo, a patto che abbiano l'interfaccia minima necessaria. Il concetto è legato a quello di sottotipo. C non permette la sottotipazione. In Java, la sottotipazione e l'ereditarietà sono strettamente correlate. L'ereditarietà permette alle classi figlie di ereditare definizioni di codice dalla classe padre per evitare la duplicazione di codice. In C++, è possibile restringere l'ereditarietà ed ereditare solo parzialmente, così come rendere privati dei metodi pubblici ereditati dalla classe padre. In Java, è consentita la riscrittura di metodi della classe padre senza cambiare la segnatura e rendere pubblici i metodi privati della classe padre: sono permessi allargamenti di visibilità e aggiunte di metodi e campi, ma non restringimenti e rimozioni. Il *binding dinamico* è un meccanismo che permette di risolvere a *runtime* quale metodo deve essere eseguito, tipicamente attraverso l'*override* dei metodi tramite tabelle di *lookup*.
La sottotipazione può correre nel verso opposto rispetto all'ereditarietà. Ad esempio, possiamo usare una lista per implementare una pila o una coda, ma questo non significa che la pila e la coda debbano ereditare dalla lista. Logicamente, è il contrario: la lista è sia una pila che una coda (ereditarietà multipla, non consentita in Java). In

sintesi, la sottotipazione riguarda le interfacce, mentre l'ereditarietà riguarda l'estensione delle classi. La presenza di un metodo viene controllata a *compile time*, ma il *binding* è dinamico perché l'effettiva realizzazione della classe può dipendere dall'esecuzione. Lo stesso discorso vale per l'*overloading* degli operatori. # Design pattern Un *design pattern* è una soluzione generale e riutilizzabile a un problema ricorrente nel contesto del design del *software*. I design pattern non sono algoritmi né codice, ma piuttosto schemi che descrivono come risolvere un problema specifico in modo efficace e flessibile. Essi forniscono una struttura e una guida per risolvere problemi comuni, migliorando la qualità del codice e facilitando la comunicazione tra sviluppatori. Secondo alcune interpretazioni, tuttavia, il bisogno di *pattern* indica mancanza di funzionalità e inadeguatezza da parte dei linguaggi di programmazione. I *design pattern* sono stati introdotti in informatica per la prima volta nel libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* di Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, noti come *Gang of Four*. L'idea di strutture fondamentali riutilizzabili è però ben più vecchia, ed è presente in molti altri settori.

Elenchiamo alcuni tipi di *design pattern*, dividendoli in categorie (quelli segnati in grassetto sono spiegati in dettaglio di seguito): 1. *creazionali*: istanziano oggetti 1. **singleton** 2. *prototype*: istanza (prototipo) clonabile 3. *builder*: separa la costruzione dalla rappresentazione (produce oggetti di un'altra classe, es. `StringBuilder` di Java che concatena stringhe in modo efficiente) 4. *factory method*: costruisce oggetti di varie classi (correlate) 5. *abstract factory*: costruisce varie famiglie di classi 2. *strutturali*: compongono classi e oggetti in strutture più grandi 1. *adapter*: permette di collegare le interfacce di diverse classi 2. *bridge*: separa interfaccia da implementazione 3. *composite*: compone oggetti 4. *decorator*: aggiunge dinamicamente responsabilità agli oggetti (decora i risultati di un metodo) 5. *facade*: singola classe che rappresenta un intero sottosistema 6. *proxy*: un oggetto rappresenta un altro oggetto 3. *comportamentali*: regolano la comunicazione tra oggetti e definiscono algoritmi e responsabilità 1. *observer*: l'osservato invia notifiche agli osservatori sui propri cambi di stato 2. **visitor** 3. **strategy** 4. *iterator*: visita di collezioni

Strategy

Diverse strategie che sono realizzazioni di una stessa interfaccia, tra loro intercambiabili all'interno dell'utilizzatore. Nel seguente esempio, un robot può utilizzare diverse strategie di movimento, rappresentate dall'interfaccia `Movement`.

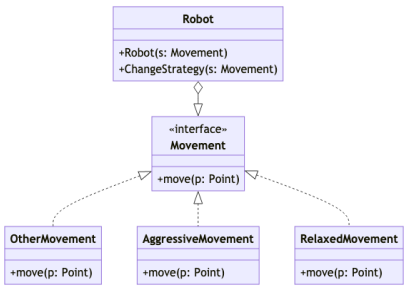


Figura 1: diagram

Singleton

Vogliamo che esista una sola istanza di una data classe. Nel seguente esempio, vogliamo che esista un solo dado all'interno del gioco. Nascondiamo innanzitutto il costruttore agli utilizzatori. Ora esso può essere chiamato solo all'interno della classe `Dado`:

```
class Dado {
    private Dado() {
        ...
    }
}
```

Creiamo ora un metodo statico per ottenere la classe:

```
class Dado {
    private Dado() {
        ...
    }

    public static Dado getDado() {
        ...
    }
```

Il metodo deve essere statico perché altrimenti non potrebbe essere chiamato prima di istanziare la classe. È necessario anche definire una variabile statica per sapere se esiste già un dado istanziato e, nel caso, restituirlo. La classe prende questa forma:

```
class Dado {
    private static Dado d = null;
    ...
    private Dado() {
        ...
    }
    public static Dado getDado() {
        if (d == null)
            d = new Dado();
        return d;
    }
}
```

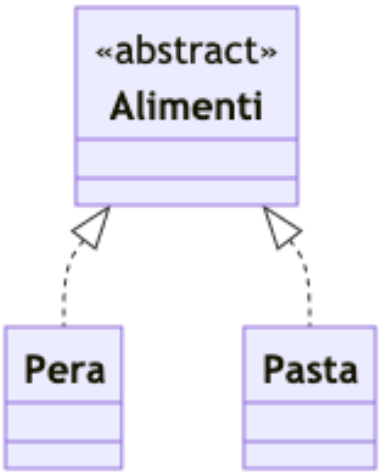
Una possibile riscrittura è:

```
class Dado {
    private static Dado d = new Dado();
    ...
    private Dado() {
        ...
    }
    public static Dado getDado() {
        return d;
    }
}
```

ma in questo caso il dado viene creato indipendentemente dal suo effettivo uso. La prima versione viene detta *lazy singleton*, mentre la seconda è un'implementazione più classica.

La versione non *lazy* del pattern smette di funzionare correttamente se utilizzata in un programma multithread. Se un thread si interrompe durante l'esecuzione dell'if all'interno di `getDado` e l'altro thread crea un dado da zero, al ritorno del primo thread ne verrà creato un altro. Questo si risolve utilizzando la versione *lazy* oppure aggiungendo la keyword `synchronized` al metodo `getDado`.

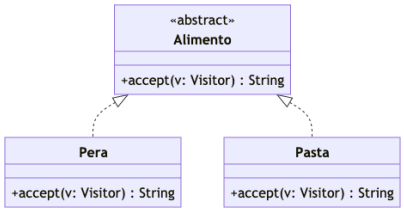
Visitor



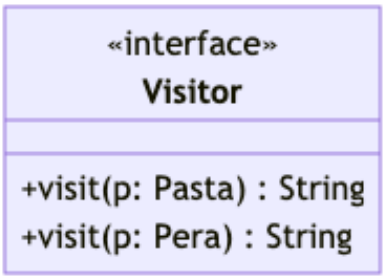
Supponiamo di avere un menu che elenchi degli alimenti. Vogliamo che sia multiingua.

Possibili soluzioni non eleganti: - diversi metodi per reperire lingue diverse: `getItaliano`, `getInglese`, ... - `getNome(Lingua l)`

Soluzione *visitor*:

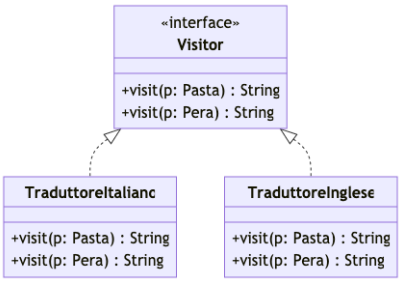


Com'è fatto un *visitor*?



Deve saper visitare ogni singola classe di nostro interesse.

Nel nostro caso ne abbiamo due tipi:




```
class TraduttoreItaliano implements Visitor {
    visit (p: Pasta) {
        return "Pasta";
    }

    visit (p: Pera) {
        return "Pera";
    }
}

class TraduttoreInglese implements Visitor {
    visit (p: Pasta) {
        return "Pasta";
    }

    visit (p: Pera) {
        return "Pear";
    }
}
```

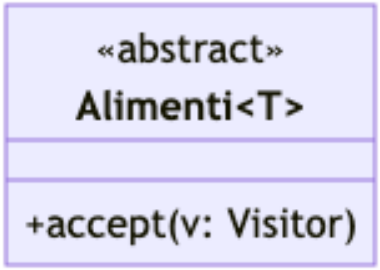
In `Pasta` e `Pera`:

```
class Pera extends Alimento {
    accept(v: Visitor) {
        v.visit(this)
    }
}

class Pasta extends Alimento {
    accept(v: Visitor) {
        v.visit(this)
    }
}
```

Come mai non possiamo spostare il metodo nella classe astratta, se è uguale in tutti i casi? Perché in compilazione il riferimento a `this` sarebbe a `Alimento` e non alle singole realizzazioni; nei visitor non ci sarebbe un metodo per visitare un generico alimento. Come utilizzare il *visitor*?

```
Alimento a = new Pasta();
Visitor v = new TraduttoreItaliano();

a.accept(v);
```

Possiamo rendere il *pattern* ancora più generico, decidendo di poter ritornare generici dal metodo `accept`. A questo

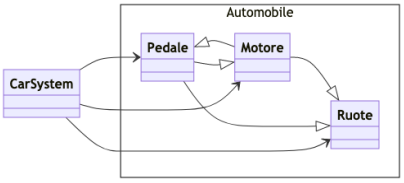


fine possiamo rendere generica la classe astratta: oppure tornare un generico dal metodo:

```
<T> accept(v: Visitor) : T
```

Vale la pena di utilizzare questo *design pattern*? Il codice è più lungo e ha più classi, ma è più flessibile e più ordinato.

Façade



Utilizziamo una classe che permetta di accedere in modo semplice ad un sistema complesso:

Come definire un'interfaccia in `C` in un modo che nasconda l'implementazione? Possiamo definirla in un *header*. Volendo implementare, ad esempio, una libreria che permetta di gestire una rete di computer, definiamo le intestazioni dei metodi nel file `mcomputer.h`:

```
#ifndef MODULE_MCOMPUTER_H_
#define MODULE_MCOMPUTER_H_

typedef int computerid;

void setComputerData(char*, computerid);

char* getComputerName(computerId);

// ...altri metodi

#endif
```

Le istruzioni `#ifndef`, `#define` e `#endif` rappresentano direttive per il preprocessore del compilatore. Nello specifico, `#ifndef` significa *if not defined*: se in nessun altro file preprocessato fino al momento della lettura è mai stato definito il tag `MODULE_MCOMPUTER_H`, bisogna eseguire tutte le righe fino a `#endif`, che rappresenta il termine del blocco condizionale.

```
flowchart
    A[Inizio] --> B{MODULE_MCOMPUTER_H_\ndefinito}
    B --> |Sì| C[Fine]
    B --> |No| D[Definisci MODULE_MCOMPUTER_H_]
    D --> E[Esegui il codice - intestazioni dei metodi]
    E --> C
```

Per eliminare la necessità di questo blocco logico, i compilatori moderni offrono spesso la direttiva `#pragrma once`, da inserire in testa all'*header*:

```
#pragma once

typedef int computerid;
// ... tutto il resto dell'header
```

Scriviamo ora liberamente l'implementazione, ricordando che essa non sarà visibile agli utilizzatori della libreria, come sarà spiegato più dettagliatamente in seguito. Effettuiamo innanzitutto un `include` a inizio codice per importare le intestazioni dei metodi che abbiamo scritto nell'*header*. Includiamo poi le librerie predefinite che ci serviranno per lavorare con i dati. Definiamo la costante `N_COMPUTER` che rappresenta il numero massimo di computer per rete supportato dalla nostra libreria. Scriviamo poi i corpi di ogni metodo dichiarato in precedenza all'interno di `mcomputer.h`. Chiamiamo il nuovo file `computer.c`:

```c
#include "mcomputer.h"

#include<string.h>
#include<stdlib.h>
#include<stdio.h>

#define N_COMPUTER 10

static char* names[N_COMPUTER];

void setComputerData(char* m, int c) {
    if(names[c]==NULL) {
        names[c] = malloc(sizeof(char)*strlen(n)+1);
        strcpy(names[c],n);
    }
}

char* getComputerName(int id) {
    if(id < 0 || id >= N_COMPUTER) {
        return "INVALID ID";
    }
    return names[id];
}

// ...il resto dell'implementazione
```

Per distribuire la libreria in modo offuscato, ovvero lasciando liberamente accessibile solo l'*header*, dobbiamo compilare `computer.c`. Questo permette di distribuirne una versione binaria, il cui codice sorgente sia illeggibile all'utente. La procedura richiede tre passaggi: 1. compilazione di `computer.c` in un *object file* 2. trasformazione dell'*object file* in uno *shared object* che possa essere *linkato* ad altri file durante la compilazione 3. *linking* della libreria al main di eventuali programmi che ne facciano uso

L'intera procedura è riportata di seguito:

```
gcc -fPIC -c computer.c -o mcomputer.o
gcc -shared mcomputer.o -o libmcomputer.so
gcc -o main main.c -L. -lmcomputer
```

Il file `main.c` richiede soltanto l'inclusione di `mcomputer.h` per poter utilizzare i metodi della libreria:

```c
#include "mcomputer.h"

int main(void) {
    computerid id = 3;
    char* computer_name = getComputerName(id);
    // ...
    return 0;
}
```

Java

Scriviamo un esempio in cui si istanzia una sottoclasse come se fosse una superclasse:

```java
package test;

public class Main {
    public static void main(String[] args) {
        S pluto = new T();
    }
}
```

Se `T extends S` allora funziona tutto:

```
classDiagram
    S <|-- T
    G <|-- F
```

Altro esempio:

```
package test;

class S {
    void foo(G a) {
        System.out.println("s");
    }
}

class T extends S {
    void foo(G a) {
        System.out.println("s");
    }
}

public class Main {

    public static void main(String[] args) {
        T s;
        G a;

        if(new Random().nextBoolean()==true)
            a = new F();
        else
            a = new G();

        if(new Random().nextBoolean()==true)
            s = new T();
        else
            s = new Q();

        s.foo(a);    // stampa "s"
        A.f(3,3);
    }
}
```

Il compilatore non può sapere delle promozioni e quindi chiama il tipo della classe più base (quella del tipo delle variabili `T` e `S`).

```
classDiagram
    Q <|-- T
    F <|-- G
```

Facendo così:

```
T s = new T();
s.foo(a);
```

stampa ancora `s`.

Pattern

Facade

```
package patterns;

public class PC {
    private CPU cpu;
    private RAM ram;
    private DISK disk;

    // facade
    float getFreq() {
        return cpu.getFreq()
    }
}
```

In questo esempio possiamo accedere dalla classe PC (la facciata) ai dati contenuti negli oggetti da cui è composto.

```
classDiagram
    direction TD

    Computer  "1" *-- "1" CPU : "has"
    Computer  "1" *-- "1" RAM : "has"
    Computer  "1" *-- "1" DISK : "has"
    class Computer {
        -cpu: CPU
        -ram: RAM
        -disk: DISK
        +getFreq() float
    }
    class CPU {
        ...
        getFreq() float
    }
```

Singleton

```
package patterns.singleton;

public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() {

    }

    static Singleton getInstance() {
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton a = Singleton.getInstance();
        Singleton b = Singleton.getInstance();
        System.out.println(a)
        System.out.println(b)
    }
}

public class LazySingleton {

    private static LazySingleton instance;

    private LazySingleton() {

    }

    static synchronized LazySingleton getInstance() {
        if(instance==null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

Ricordiamo che il *lazy singleton* non è adatto all'uso *multithreaded* e quindi richiede di aggiungere `synchronized` al metodo `getInstance()` perché l'interruzione di un *thread* durante l'esecuzione del metodo e la conseguente chiamata dello stesso metodo da parte di un altro *thread* possono rompere la singolarità dell'oggetto.

Visitor

```
package visitor;

public class Ruota extends CarElement {
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

public class Motore extends CarElement {
    private int cilindrata;

    public Motore(int cilindrata) {
        this.cilindrata = cilindrata
    }

    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }

    public void setCilindrata(int cilindrata) {
        this.cilindrata = cilindrata
    }

    public int getCilindrata() {
        return cilindrata;
    }
}

public class Carrozzeria extends CarElement {
    @Override
    public <T> T accept(Visitor<T> v) {
        return v.visit(this);
    }
}

public interface Visitor<T> {
    T visit(Ruota r);
    T visit(Motore r);
    T visit(Carrozzeria r);
}

public class ToStringVisitor implements Visitor {
    @Override
    public String visit(Ruota r) {
        return "RUOTA";
    }

    @Override
    public String visit(Motore r) {
        return "MOTORE";
    }

    @Override
    public String visit(Carrozzeria r) {
        return "CARROZZERIA";
    }
}

public class ToTestaRossa implements Visitor {
```

```
    @Override
    public void visit(Motore r) {
        r.setCilindrata(10000000);
    }
}

public class Main {
    public static void main(String[] args) {
        Motore m = new Motore(1000);
        Visitor v = new ToStringVisitor();
        System.out.println(m.accept(v));

        // Cilindrata
        System.out.println(m.getCilindrata());
        m.accept(new ToTestaRossa());
        System.out.println(m.getCilindrata());
    }
}
```

L'esempio mostra la creazione di diversi *visitor* che possono eseguire azioni completamente diverse durante la loro visita agli oggetti. ## Covarianza, invarianza e controvarianza

Sia enunciato il *principio di sostituzione di Liskov* (*LSP*), sviluppato da Barbara Liskov 1987:

> Requisito di sottotipazione: sia ϕ una proprietà dimostrabile di oggetti x di tipo T. Allora $\phi(y)$ deve essere vero per oggetti y di tipo S dove S è sottotipo di T.

Simbolicamente:

$$S \leq T \rightarrow (\forall x :\ T.\phi(x) \rightarrow \forall y :\ S.\phi(y))$$

Il seguente diagramma rappresenta il concetto di *invarianza dei parametri*:

```
classDiagram
    class ClassA {
        method(t: T&)
    }
    class ClassB {
        method(t: T&)
    }

    ClassA <|-- ClassB
```

Il seguente, invece, rappresenta la *covarianza del tipo di ritorno*:

```
classDiagram
    class ClassA {
        method(t: T)
    }
    class ClassB {
        method(t': T')
    }

    ClassA <|-- ClassB
```

Ecco invece un esempio di *controvarianza* del parametro:

```
classDiagram
    class ClassA {
        method(t: T)
    }
    class ClassB {
        method(t': T')
    }

    ClassA <|-- ClassB
```

Il seguente diagramma rappresenta infine la *covarianza nel tipo del parametro* (non supportata da Java, perché contraria al principio di Liskov):

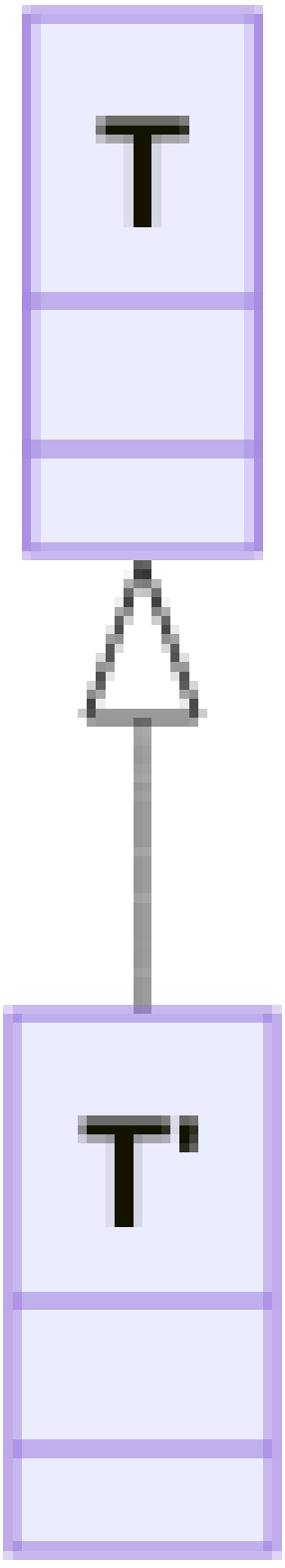


Figura 2: diagram

```
classDiagram
    class ClassA {
        method(t: T)
    }
    class ClassB {
        method(t': T')
    }

    ClassA <|-- ClassB
```

Un di linguaggio con parametri covarianti è il linguaggio Eiffel inventato da Bertrand Meyer.

Array in Java

Gli *array* sono automaticamente definiti per ogni classe o interfaccia. Non sono estendibili (`final`). Possono essere multidimensionali, come array di array. Sono tipi reference e quindi possono essere nulli. Possiamo definirli esplicitamente per dimensione:

```
Circle[] x = new Circle[array_size]
```

O anonimamente:

```
new Circle[] = {c1, c2, c3}
```

Dalla classe base `Object` derivano: - le classi definite dall'utente - gli *array* di tipo `Object[]` e i loro derivati (generati automaticamente) - le eccezioni Non derivano da `Object` i tipi primitivi.

Gli array sono covarianti. Se S è un sottotipo di T, allora l'array di tipo S è sottotipo dell'array di tipo T.

Sia definito un esempio per gli errori:

```
class A {...}

class B extends A {...}

B[] bArray = new B[10];
A[] aArray = bArray;
aArray[0] = new A();
```

Questa porzione di codice compila correttamente: possiamo assegnare `bArray` ad `aArray` perché rispettiamo la covarianza. Dà però errore in esecuzione quando assegniamo un A alla prima cella di `aArray`, che nel frattempo è stato promosso ad array di B.

Generici in Java

I generici in Java sono stati introdotti dalla versione 1.5. Prima si utilizzava la derivazione di tutti i tipi da `Object` come modo improprio per ottenere generici. Non sono stati inseriti da subito in Java perché non era chiara la via per implementarli (estensione alla macchina virtuale? Casting automatici? Duplicazione del codice?).

Segue un esempio di codice da risolvere necessariamente con i generici generici. Volendo definire un metodo per calcolare il massimo tra due numeri, sarebbe necessario reimplementarlo per ogni tipo numerico:

```
int max(int a, int b) {

}

float max(float a, float b) {

}
```

Non è infatti consentito scrivere

```
Object max(Object a, Object b) {

}
```

perché non è possibile verificare che `a` e `b` siano dello stesso tipo, per garantirne la confrontabilità. Il problema è aggirabile localmente con i `Comparable` ma anche questa soluzione richiede indirettamente i generici.

Per esempio, uno *stack* senza generici è implementabile nel seguente modo:

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

Dietro le quinte i generici vengono trasformati in codice con `Object` e i cast. La specializzazione senza i *generics* si può ottenere con la covarianza, ma richiede comunque java > 1.5, perché il concetto è stato inserito nello stesso aggiornamento del linguaggio che ha introdotto i generici.

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

```
class Stack<A> {
    void push(A a) { ... }
    A pop() { ... }
    ...
}

String s = "Hello";
Stack<A> stack = new Stack<A>();     // new Stack() dalla 1.7/1.8
stack.push(s);
```

Metodi generici:

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

I parametri dei generici non rispettano la covarianza. Ad esempio, nonostante `Cane` e `Gatto` siano sottotipo di `Animale`, non è possibile fare

```
List <Animale> a = ...;
List <Cane> c = ...;
a = c;     // no
a[0] = new Gatto();
```

Non è nemmeno possibile questo:

```
faiVerso(List<Animale> a) {...}
faiVerso(c);
```

È necessario scrivere

```
<A extends Animale> faiVerso(List<A> a) {...}
```

e questo risolve parzialmente la mancata covarianza dei generici. Esiste anche la sintassi inversa `<E super T>` per richiedere la necessità di supertipi. Ad esempio, per chiedere l'interfaccia comparable, usiamo `<T extends Comparable<T>>`.

Interfaccia Comparable

```
public interface Comparable<T> {
    int compareTo(T o) {...}
}
```

Il metodo`compareTo` restituisce valori negativi se $A < B$, zero se $A = B$, positivi se $A > B$. Possiamo sfruttarlo per ordinare liste con i metodi statici `Collections.sort` o `Arrays.sort`. L'ordinamento si dice consistente se `a.compareTo(b) == 0` quando vale anche `b.compareTo(a) == 0`, per qualsiasi a, b appartenenti alla classe.

```
List<String> ls = new ArrayList<String>();     // corretto
List<Object> lo = ls;     // sbagliato
```

Volendo scrivere un metodo che stampa gli elementi di una collezione:

```
void printCollection(Collection<Object> s) {
    for (Object e : s) {
```

```
        System.out.println(e)
    }
}
```

Come si potrebbe implementare il metodo in modo che funzioni per tutte le collezioni?

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

Il simbolo `?` è detto *wildcard* e rappresenta il supertipo dei generici. Possiamo limitarlo superiormente e inferiormente:

```
List <? extends Persona>    // accetto tutti i sottotipi di persona: upper bound
```

```
<T extends Comparable<? super T>>
```

significa che T debba essere confrontabile con oggetti della sua stessa classe o della sua superclasse. Rappresenta una estensione rispetto a`<T extends Comparable<T>>` che invece permette di confrontare solo con oggetti della classe specifica di T.

Il *wildcard* permette di reintrodurre l'ereditarietà, ma deliberatamente, e soprattutto permette di mantenere il controllo dei tipi.

Visibilità

È possibile aumentare la visibilità (a patto che quella iniziale non sia `private`) mantenendoci nel caso dell'*overriding*. Non possiamo restringerla. Non è consentito lanciare più eccezioni, solo meno o nessuna, rispetto al metodo di cui si effettua l'*override*.

Esempi su coviarianza e invarianza

Esempio 1

Definiamo due classi, `Persona` e `Studente`:

```
package es1;

public class Persona {

}
```

```
package es1;

public class Studente extends Persona {

}
```

```
package es1;

public class Main {
    public static void main(String[] args) {
        Studente[] ss = new Studente[10];
        Persona[] pp = ss; // è possibile per covarianza degli array

        pp[0] = new Persona();   // il compilatore non lo impedisce
    }
}
```

Il codice compila ma otteniamo un `ArrayStoreException` in esecuzione perché nel frattempo da `Persona` è stato promosso ad un array di `Studente`. Questo comportamento è reso più evidente dal seguente esempio:

```
package es1;

public class Main {
    public static void main(String[] args) {
        Studente[] ss = new Studente[10];

    Persona[] pp;
```

```
    if(new Random().nextBoolean())
        pp = ss;
    else
        pp = new Persona[10];
    }

    pp[0] = new Persona();
}
```

Esempio 2

Creiamo due classi, `A` e `B`:

```
package es2;

public class A {

}
```

```
package es2;

public class B extends A {

}
```

```
package es2;

import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {

    List<B> bb = new ArrayList<B>();
    List<A> aa = bb;
}
```

In questo caso è il compilatore ad accorgersi e dare errore. Volendo inserire sottoclassi in un `ArrayList` della superclasse, è necessario usare la seguente sintassi:

```
// Ora possiamo assegnare bb ad aa
List<? extends A> aa = bb;

// Questa riga però dà errore
aa.add(new Persona());   // è diventata una lista di studente
aa.add(new Studente());  // anche questa non va perché non è possibile determinare il tipo esatto ammissib
```

Le collezioni di generici non godono della proprietà di covarianza tipica degli *array*.

Esempio 3

Il seguente esempio mostra la covarianza del tipo restituito e l'invarianza dei parametri.

```
package es3;

public class A {
    public A m(B y) {
        return new A();
    }
}

public class B extends A {
    // è un overload perché è diverso il tipo del parametro passato
    public A m(A h) {
        return null;
    }
}
```

In altri linguaggi si potrebbe trattare di *overriding* perché si sta chiedendo di meno nei parametri. Java però è invariante nei parametri quindi non lo consente e lo tratta come *overload*.

```
public class B extends A {
    // è covariante nel tipo di ritorno -> override
    @Override
    public B m(B y) {
        return null;
    }

    public A m(A h) {
        return null;
    }
}
```

Non è consentito ridurre la visibilità del metodo `m`, passando ad esempio da `public` a `protected`. Il contrario è invece possibile.

```
// non funziona
@Override
public B m(B y) {
    return null;
}
```

Questo avviene perché Java impone di mantenere visibili dei metodi che potrebbero essere usati da altre classi.

Esempio 4

```
package es4;

public class A {

    int a = 0;

    public String toString() {
        return "A";
    }
}

public class B extends A {

    public boolean equals (B a) {
        return a.a == this.a;
    }

}
```

```
package es4;

import java.util.List;
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        List<A> list = new ArrayList<>();
        list.add(new A());
        list.add(new B());
        for (A a: list) {
            A obj = new B();
            if (a.equals(obj)) {
                System.out.println("Trovato");
            }
        }
```

```
    }
}
```

Non viene trovato. Stiamo chiamando l'`equals` di `A` che, non essendo stato definito, chiama l'`equals` della classe `Object` che controlla semplicemente l'uguaglianza tra reference.

Non funziona neanche nel modo seguente:

```
for (B a: list) {
    A obj = new B();
    if (a.equals(obj)) {
        System.out.println("Trovato");
    }
}
```

l tipo dinamico del parametro è `B`, ma il tipo statico è ancora `A` per colpa di `obj`, quindi viene chiamato nuovamente l'`equals` sbagliato. In `B` abbiamo definito un equals per oggetti di tipo `B`, non per oggetti generici, quindi abbiamo fatto un overload e non un override.

La soluzione corretta è:

```
for (B a: list) {
    B obj = new B();
    if (a.equals(obj)) {
        System.out.println("Trovato");
    }
}
```

Volendo complicare l'esempio:

```
package es4;

public class C extends B {
    // Questo è un vero override
    public boolean equals(Object a) {
        return ((C) a).a  == this.a;
    }

    public String toString() {
        return "C";
    }
}
```

È finalmente mostrato un vero *override* dell'`equals` della classe `Object`. L'*override* per oggetti di tipo `B` è ancora attivo, perché `C` lo eredita da `B`.

In generale, il metodo da eseguire a *runtime* è scelto in base ai tipi statici, ma la scelta è eseguita dinamicamente.

Esempio: definizione di tipi generici

Si cerchi di creare una pila generica senza utilizzare i tipi generici.

```
package es5;

public class Pila {
    Object[] p = new Object[10];
    int cima = 0;
    void push(Object o) {
        p[cima] = o;
        cima++;
    }

    Object pop() {
        cima--;
        return p[cima++];
    }
}
```

La seguente implementazione non permette di specificare il tipo da inserire:

```
public class Main {
    public static void main(String[] args) {
        Pila p = new Pila();
        p.push(new Persona());
        p.push(new String());
    }
}
```

Per specializzare la pila a contenere una sola classe, ad esempio `String`, dovrei copiare e riadattare lo stesso codice:

```
public class PilaStringhe {
    String[] p = new Object[10];
    int cima = 0;
    void push(Object o) {
        p[cima] = o;
        cima++;
    }

    Object pop() {
        cima--;
        return p[cima++];
    }
}
```

Ogni modifica richiede di rivedere il codice in tutte le versioni create. Volendo invece usare l'estensione:

```
package es5;

public class PilaStinghe extends Pila {
    @Override
    public String pop() {
        return (String) super.pop();
    }
}
```

Stiamo rispettando la covarianza: abbiamo ristretto il tipo di ritorno. Si tratta dunque di un override vero e proprio. Continuiamo a lavorare alla classe:

```
public class PilaStinghe extends Pila {

    // Questo non è override
    public void push(String o) {
        super.push(o);
    }

    @Override
    public String pop() {
        return (String) super.pop();
    }
}
```

Non possiamo nascondere il metodo `push` della `Pila` di oggetti. Quello che possiamo fare è impedire il push di oggetti:

```
public class PilaStinghe extends Pila {

    @Override
    public void push(Object o) {
        throw new IllegalArgumentException();
    }

    // Questo non è override
    public void push(String o) {
        super.push(o);
    }

    @Override
```

```
    public String pop() {
        return (String) super.pop();
    }
}
```

I generici nascono proprio per evitare questa problematica. Al loro interno i generici di Java 1.5 memorizzano gli elementi in un *array* di `Object` e producono automaticamente i *casting* appropriati per rispettare il tipo generico specificato. Questo permette di avere metodi generici come `Collections.sort()`, perché la rappresentazione interna è la stessa tra vari tipi di collezioni generiche. Il metodo richiede soltanto che gli elementi della collezione implementino l'interfaccia `Comparable`: `T extends Comparable<? super T>>`. Ad esempio:

```
public class Persona implements Comparable<Persona> {

    private int eta;

    public Persona(int eta) {
        this.eta = eta;
    }

    @Override
    public int compareTo(Persona p) {
        return this.eta - p.eta;
    }
}
```

Creiamo un metodo per stampare il contenuto di una lista contenente persone o classi derivate:

```
public static <T extends Persona> void print(List<T> list) {
    for(T t: list) {
        System.out.println(t);
    }
}
```

C++

Il linguaggio C++ nasce nel 1985 e lo sviluppo della sua prima versione si estende fino al 1997. È stato sviluppato da Bjarne Stroustrup ai Bell Labs. C++ nasce come linguaggio per la creazione di simulazioni ed estende C aggiungendo un sistema di classi ispirato al linguaggio Simula. L'idea base è creare un linguaggio a oggetti che non comprometta la flessibilità e l'efficienza di C. È garantita la retrocompatibilità con C (C++ può essere visto come un suo *superset*), con l'aggiunta di un migliore *type checking* statico, l'astrazione dei dati, la possibilità di definire oggetti e un *focus* sull'efficienza. Quest'ultima si fonda sul seguente principio: il codice scritto senza sfruttare una certa *feature* deve essere efficiente come se il linguaggio non avesse quella feature in assoluto. Non deve esserci, insomma, *overhead* per le funzionalità non utilizzate in un dato momento.

In C++ i tipi di dati astratti corrispondono ai tipi. Rispetto al C, essi diventano centrali a scapito delle procedure. Infatti la cosa più simile ai tipi astratti in C sono i tipi opachi, che però non sono una *feature* definita appositamente.

C++ diventa in poco tempo ampiamente usato, con grande successo. Come il suo predecessore C, alcune aziende e organizzazioni impongono vincoli aggiuntivi, restringendo l'espressività del linguaggio per aumentarne la sicurezza. Alcune di queste modifiche includono non utilizzare l'ereditarietà, le funzioni virtuali (presso SGI) e altre caratteristiche critiche.

Il linguaggio C++ eredita il modello di macchina del C (*heap*, *stack*, indirizzi), e non ha *garbage collection* per ragioni di efficienza. Al contrario di Java, è possibile decidere manualmente se allocare gli oggetti su *heap* o su *stack*. Rispetto al C aggiunge la possibilità di passaggio di parametri per *reference*, un sistema di eccezioni e il costrutto del *copy constructor* per il clonaggio degli oggetti. Un'altra aggiunta è il concetto di *namespace*. Si definisce *namespace* una regione dichiarativa per generare uno scope con un nome definito (una sorta di package). Fully qualified: `nome_namespace::nome_classe` oppure con direttiva `using namespace nome_namespace;`. All'interno del namespace si possono dichiarare delle classi e/o dei metodi.

Esempio: visibilità e classi in C++

Si consideri il seguente esempio di definizione di una struttura dati per rappresentare il tempo in C:

```
// crea una struttura, imposta i suoi membri e stampalo

// definizione della struttura
struct Time {
    int hour;  // 0-23
    int minute;   // 0-59
    int second;   // 0-59
}

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

L'esempio presenta alcune problematiche. Non abbiamo vincoli per impedire l'impostazione degli attributi a valori insensati. Non abbiamo la possibilità di inizializzare correttamente la struttura. Non c'è un'interfaccia da mantenere stabile in caso di modifiche: tutto da rifare in caso di modifiche.

L'approccio C++ consiste nel risolvere il problema mediante l'utilizzo delle classi:

```
#include <iostream.h>

class Time {
public:
    Time();    // costruttore di default
    // prototipi delle funzioni
    void setTime(int h, int m, int s);
    void printMilitary(const Time &);
    void printStandard(const Time &);
private:
    int hour;    // 0-23
    int minute;  // 0-59
    int second;  // 0-59
    ...
};   // notare il punto e virgola

/*
Il costruttore Time inizializza tutti i membri a zero.
```

```
Nessun valore di ritorno.
Si assicura che la classe sia inizializzata con uno stato consistente.
*/
Time::Time()
{
    hour = minute = second = 0;
}

/*
Metodo setTime.
Nessun valore di ritorno.
Permette di impostare l'orario (che è stoccato in variabili private) controllando
la correttezza dei valori inseriti.
*/
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;
}

// ... metodi di stampa ...
```

Possiamo allocare la classe direttamente sullo *stack*:

```
Time sunset;    // oggetto di tipo Time
Time arrayofTimes[5];   // array di Time allocato sullo stack
Time *pointerToTime;    // puntatore a Time sullo stack
```

Possiamo fare *overloading* dei metodi e degli operatori:

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

// Implementazioni
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) }
```

Notiamo che gli *overload* del costruttore chiamano implicitamente il costruttore di *default*.

C++, al contrario di C, permette l'*overloading*. In C non è possibile perché non c'è *runtime* per il *binding* dinamico.

Il costruttore di *default* è chiamato automaticamente (sia implicitamente che esplicitamente) anche all'allocazione su *stack*:

```
Time t1; // Time() chiamato automaticamente
Time t1(); // errore

Time t2(08);   // chiamata implicita per i costruttori alternativi
Time t2 = Time(08);  // chiamata esplicita
```

Per allocare su *heap* si utilizza la *keyword* `new`:

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

Ad esempio, per allocare dinamicamente la classe `Time` definita nell'esempio:

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

La dichiarazione degli array può avvenire sia indirettamente, sia con l'uso di un letterale:

```
Time arrayOfTimes[5];  // chiama implicitamente Time()

// oppure: inizializzazione esplicita
Time secondArray[8] = {3, Time(8), Time(), Time(1,2,11)};
```

Constructor Initializer List

Un altro concetto importante è quello di *constructor initializer list*. Essa rappresenta un elenco di chiamate a costruttori per elementi della classe, da eseguire prima di chiamare il costruttore della classe stessa.

```
class Info {
private:
    const int i;
    double m;
    Time t;
public;
    Info();
    Info(int j, double n);
};

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

Come mai la si impiega? Nell'esempio, `i` è un `const int`: all'interno del corpo del costruttore non è possibile variarne il valore. Va quindi assegnata nella *initializer list*. Allo stesso modo, possiamo avere come campi delle reference (ad esempio `Time &t`) che non possono essere nulle. Devono essere dunque inizializzate prima dell'avvio del costruttore. Lo stesso principio si applica ai campi contenenti oggetti, per lasciare spazio di allocazione.

La necessità per le costanti potrebbe derivare dal fatto che C++ sia costruito sul C.

Copy constructor e distruttori

Sia mostrato un esempio di uso del *copy constructor*:

```
S u(s);   // chiama il copy costructor della classe S
S v = s;
```

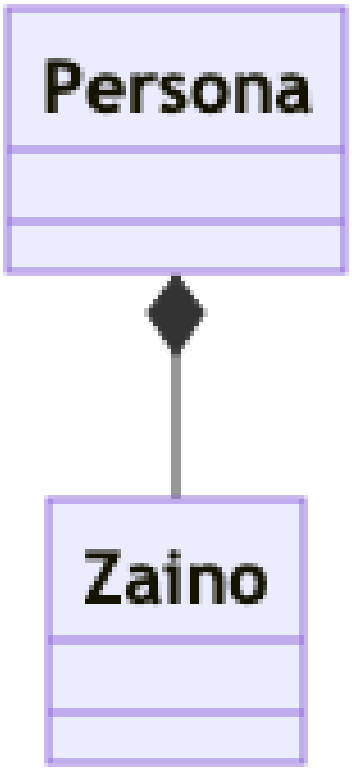
Viene utilizzato il copy constructor per clonare l'oggetto `s` di tipo `S` all'interno della variabile `u`.

Il *copy constructor* si definisce con la seguente sintassi:

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

In caso non lo si definisca esplicitamente, viene creata di *default* dal compilatore una versione che copia tutti i campi (compresi i puntatori a sotto-oggetti). Si parla di *shallow copy*, in contrasto alla *deep copy* che consiste in una copiatura ricorsiva che clona anche i sotto-oggetti e i sotto-oggetti dei sotto-oggetti.

È utile anche il concetto di *distruttore*. Per introdurlo, si pensi al seguente esempio: un oggetto `Persona` possiede uno `Zaino`, e tale zaino non può continuare ad esistere da solo, senza il proprietario. Vogliamo che alla deallocazione di



`Persona` sia deallocato anche `Zaino`.

```
Persona {
    Zaino z;
}
```

Distruttore: prende il nome `~class_name` senza parametri e senza ritorni.

```
Class_name::~class_name() {
    // operazioni di delete
}
```

È eseguito automaticamente all'uscita di scope (dallo stack) o alla chiamata di `delete` (per gli oggetti allocati dinamicamente. È solitamente impiegato per deallocare i sotto-oggetti (o per chiamarne i rispettivi distruttori).

Sintassi del `delete`:

```
delete a;
```

Ci dovrebbe essere un `delete` per ogni `new`. La chiamata ripetuta di `delete` sullo stesso puntatore può causare problemi. La chiamata a `delete` non ha alcun effetto quando eseguita su un `nullptr`. Per questo, è bene azzerare a `nullptr` i puntatori appena dopo la deallocazione.

Dichiarazione di funzioni

```
tipo Nome_classe::nome_metodo(tipo1 parametro1, ...);
```

Possiamo definire metodi `const` incapaci di modificare i propri parametri:

```
tipo Nome_classe::nome_metodo(const tipo1 parametro1, ...);
```

o incapaci di modificare lo stato della propria classe:

```
tipo Nome_classe::nome_metodo(tipo1 parametro1, ...) const;
```

È anche possibile passare per reference (puntatore non nullo) i parametri:

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

Di default il passaggio è per copia: i parametri sono mantenuti come copia locale, modificabile, che sarà deallocata all'uscita dalla funzione. Inefficiente per oggetti grossi.

Variabili statiche, globali, funzioni inline e binding dinamico

Possiamo usare la keyword `exetern` per importare variabili globali da altri file. Variabili globali statiche con keyword `static` non utilizzabili da altri file. Le statiche dentro una classe sono condivise tra tutte le istanze della classe, come

in Java. Le funzioni statiche possono accedere solo a variabili statiche, non possono chiamare funzioni non statiche, usare la reference `this` all'oggetto o essere virtuali. Costruttori e distruttori non possono essere statici.

Funzioni `inline`: sono espanse come una *macro* nel codice compilato per eseguire sul posto invece di effettuare una chiamata di funzione. Le funzioni membri di classi sono automaticamente `inline`. Nell'*inlining* di funzioni ricorsive la funzione viene espansa un numero fissato di volte (tramite parametro da passare al compilatore) sperando che basti a coprire il numero effettivo di chiamate, e rischiando altrimenti ridondanza nel compilato. Analizziamo i costi e benefici. Benefici: non creare record di attivazione. Costi: allungamento della compilazione (relativamente poco grave), allungamento significativo del codice (se la funzione è chiamata molte volte) al punto di non starci tutto in *cache* e rallentare l'esecuzione per *cache miss*. Le funzioni importate dagli *header* sono solitamente *inline*.

Possiamo specificare argomenti di default per le funzioni:

```
void f(int size, int initQuantity = 0);
```

I parametri *inline* vanno messi tutti alla fine della lista dei parametri, compattati.

L'*overloading* delle funzioni è risolto per numero e tipo dei parametri. Non è possibile fare overloading del tipo di ritorno. In C++, per effettuare l'override di un metodo, è necessario dichiararlo come virtuale. Il polimorfismo in esecuzione viene implementato attraverso il lookup delle funzioni virtuali. Quando ciò non è possibile, il lookup è statico. Le ridefinizioni di funzioni non virtuali portano a un overriding per semplice sostituzione, senza possibilità di accesso al metodo della classe padre. Questo approccio permette di pagare il costo del lookup dinamico solo se ritenuto necessario dal programmatore.

L'ereditarietà può essere pubblica o privata.

Le funzioni virtuali devono essere dichiarate esplicitamente come tali. Esse possono ricevere override e attivano il binding dinamico. L'accesso a queste funzioni avviene indirettamente attraverso puntatori nell'oggetto.

```
class A { public: virtual void vi(){...} };
class B : public A{ public: virtual void vi(){...} };
int main() {
    A* pa = new A; a -> vi(); // chiamata virtuale
    A& ra = b; ra.vi(); // chiamata virtuale
    A a = b; a.vi();    // chiamata non virtuale
}
```

Volendo fare un esempio:

```
class Pt {
public:
    Pt(int xv);
    Pt(Pt* pv);  // overload del costruttore
    int getX:    // funzione non virtuale
    virtual void move(int dx);  // funzione virtuale
    virtual void darken(int tint);  // funzione virtuale
protected:
    void setColor(int cv);
private:
    int color;
};

void colorPt::darken(int tint) {color += tint}
```

Ereditarietà e polimorfismo

In C++, un puntatore della classe base può puntare a un oggetto di classi derivate. Poiché è possibile ridurre la visibilità dei metodi in una classe figlia, viene effettuato il *dynamic binding* per determinare quale versione visibile del metodo utilizzare. Questo meccanismo si applica solo agli oggetti mantenuti tramite puntatori o riferimenti, e non agli oggetti allocati sullo stack.

Il puntatore `this` è il primo argomento implicito di ogni funzione membro. Ad esempio, il codice

```
int A::f(int x) { ...g(i)... }
```

viene trasformato internamente in

```
int A::f(A *this, int x) { ...this->g(i)... }
```

È buona pratica dichiarare i distruttori come virtuali per permettere una migliore estensione delle classi derivate. Un esempio di dichiarazione di un distruttore virtuale è il seguente:

```
class A {
    public:
        virtual ~A();
};
```

Le funzioni sono compilate, allocate a un indirizzo di memoria e poi l'indirizzo è salvato nella *tabella dei simboli*. In caso di ridefinizione, viene semplicemente aggiornato l'indirizzo nella tabella statica dei metodi. A differenza di Java, in C++ è possibile effettuare l'*overriding* anche dei metodi privati. L'*overriding* non funziona però per gli oggetti allocati sullo *stack*. La differenza è mostrata nel seguente esempio:

```
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();
    // ...
}
```

Binding delle chiamate alle funzioni

La gestione delle chiamate alle funzioni in C++ è ibrida. Per i metodi non virtuali, viene utilizzato l'*early binding*, simile a quello del linguaggio C. Per le funzioni virtuali, invece, viene impiegato il *late binding* con *lookup* dinamico. Questo meccanismo genera una linea di *assembly* in più e richiede più memoria per i puntatori, ma la criticità si nota solo in ambito *embedded*.

Sottotipazione

La sostituzione delle classi è possibile solo in un caso specifico. Una classe `A` è riconosciuta come sottotipo di `B` solo se `B` è una classe base pubblica di `A`. Questa regola viene meno se viene effettuata una ridefinizione che riduce la visibilità di metodi ereditati.

Consideriamo il seguente diagramma di classe:

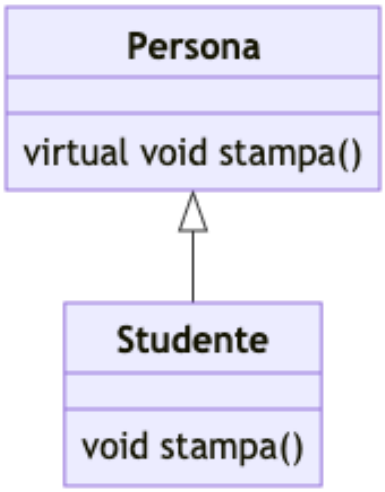


Figura 3: diagram

In questo caso, se creiamo un oggetto di tipo `Studente` e lo assegniamo a un puntatore di tipo `Persona`, accederemo al metodo `stampa` della classe `Persona` perché il metodo `stampa` nella classe `Studente` non è virtuale:

```
Persona* p = new Studente();
p->stampa();
```

Se invece utilizziamo un'assegnazione diretta tra oggetti, accederemo ancora al metodo `stampa` della classe `Persona` a causa del binding statico:

```
Persona p;
Studente s;
p = s;
p.stampa();
```

Infine, se entrambi i metodi `stampa` sono dichiarati come virtuali, otteniamo il binding dinamico al metodo `stampa` della classe `Studente`:

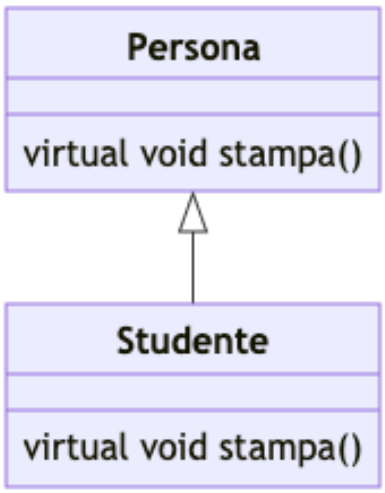


Figura 4: diagram

```
Persona* p = new Studente();
p->stampa();
```

In questo caso, il *binding* dinamico garantisce che venga chiamato il metodo `stampa` della classe `Studente`.

Principio di sostituibilità per la sottotipazione

In C++, il principio di sostituibilità per la sottotipazione permette di assegnare un oggetto di una classe derivata a un puntatore o a un riferimento di una classe base, similmente a quanto accade in Java. Questo meccanismo funziona solo per puntatori o riferimenti e non per oggetti allocati sullo *stack*. Una funzione è considerata sottotipo di un'altra se può essere sostituita al suo posto. In C++ (da C++98 in avanti) vale la covarianza del tipo restituito, ma solo per puntatori a oggetti e solo per funzioni virtuali. I parametri delle funzioni sono invece invarianti, e non è possibile la controvarianza.

Tentare di applicare il polimorfismo su oggetti allocati sullo stack provoca lo slicing. Questo fenomeno si verifica quando viene chiamato il copy constructor, che "taglia" gli attributi in eccesso della classe derivata, lasciando solo quelli della classe base. Ad esempio:

```
class A {
    int foo;
};

class B: public A {
    int bar;
};

B b;
A a = b;  // copy constructor: abbiamo ottenuto un oggetto A con solo l'attributo foo
```

In questo caso, l'oggetto `a` di tipo `A` conterrà solo l'attributo `foo`, mentre l'attributo `bar` della classe `B` verrà perso.

È importante notare che non è possibile restituire un tipo `A` quando il metodo da sovrascrivere era di tipo `*A`.

Le classi astratte sono definite come classi che contengono almeno un membro completamente astratto. Un membro astratto è dichiarato con la sintassi:

```
virtual function_decl = 0;
```

Questo indica che la classe non può essere istanziata direttamente: deve essere ereditata da una classe derivata che fornisce implementazioni concrete per tutti i membri astratti.

Template

I *template* in C++ rappresentano una forma di programmazione generica, permettendo di scrivere codice che può operare con tipi di dati diversi senza dover duplicare le funzioni per ciascun tipo. Consideriamo, ad esempio, una funzione di scambio per interi:

```
void swap(int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

Per rendere questa funzione generica, possiamo utilizzare un *template*:

```
template<class T>
void swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

In questo caso, il tipo `T` deve essere lo stesso per entrambi i parametri, poiché la funzione non può operare correttamente con tipi diversi. I *template* in C++ permettono anche di parametrizzare i tipi di ritorno delle funzioni. A differenza dei generici in Java, ogni parametro del *template* deve essere anche un parametro della funzione. I *template* possono essere utilizzati anche con le classi. Ad esempio, possiamo definire una classe `Complex` che rappresenta un numero complesso con parti reali e immaginarie di tipo generico:

```
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; }
};
```

In questo modo, la classe `Complex` può essere utilizzata con qualsiasi tipo di dato, purché supporti le operazioni necessarie.

Implementazione del *visitor pattern* in C++

Questa sezione rappresenta un esempio di implementazione del *visitor pattern* in C++, seguendo lo stesso schema dell'esempio usato in precedenza per Java.

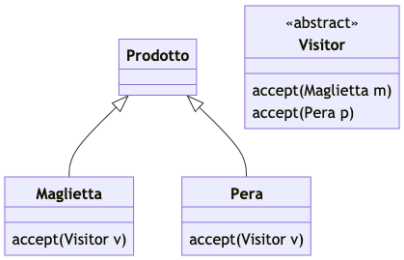


Figura 5: diagram

Sono riportati di seguito i contenuti di alcuni file:

- `Product.h` : classe astratta (interfaccia) `Product` del prodotto, in grado di accettare il *visitor*
- `Pera.cpp` e `Maglietta.cpp`: due classi concrete, `Pera` e `Maglietta`, che estendono la classe `Product` contenuta in `Product.h`
- `Visitor.h`: interfaccia `Visitor` per i *visitor*
- `EnglishVisitor.h` e `EnglishVisitor.cpp`: dichiarazione e implementazione di `EnglishVisitor`, un *visitor* concreto

- `ItalianVisitor.h` e `ItalianVisitor.cpp`: dichiarazione e implementazione di `ItalianVisitor`, un *visitor* concreto

```
/*
 * Product.h
 *
 */

#infdef PRODUCT_H_
#define PRODUCT_H_

class Visitor;   // forward declaration

class Product {
public:
    Product();
    virtual ~Product();
    virtual void accept(Visitor& visitor) = 0;
};

#endif /* PRODUCT_H_ */
```

```
/*
 * Visitor.h
 */

#infdef VISITOR_H_
#define VISITOR_H_

class Pera;
class Maglietta;

class Visitor {
public:
    Visitor();
    virtual ~Visitor();

    virtual void visit(Pera& pera) = 0;
    virtual void visit(Maglietta& maglietta) = 0;
};

#endif /* VISITOR_H_ */
```

```
/*
 * Pera.h
 */

#infdef PERA
#define PERA_H_

#include "Product.h"

class Pera : public Product {
public:
    Pera();
    virtual ~Pera();
    virtual void accept(Visitor& visitor) override;
};

#endif /* PERA_H_ */
```

```
/*
 * Pera.cpp
 */
```

```cpp
#include "Pera.h"
#include "Visitor.h"

Pera::Pera() {
    // TODO Auto-generated constructor stub
}

Pera::~Pera() {
    // TODO Auto-generated destructor stub
}

void Pera::accept(Visitor& visitor) {
    visitor.visit(&this);
}
```

```cpp
/*
 * ItalianVisitor.h
 */

#infdef ITALIAN_H_
#define ITALIAN_H_

#include "Product.h"

class ItalianVisitor : public Visitor {
public:
    ItalianVisitor();
    virtual ~ItalianVisitor();

    virtual void visit(Pera& pera) override;
    virtual void visit(Maglietta& maglietta) override;
};

#endif /* ITALIAN_H_ */
```

```cpp
/*
 * ItalianVisitor.cpp
 */

#include "ItalianVisitor.h"

ItalianVisitor::ItalianVisitor() {
    // TODO Auto-generated constructor stub
}

ItalianVisitor::~ItalianVisitor() {
    // TODO Auto-generated destructor stub
}

void ItalianVisitor::visit(Pera& pera) {
    std::cout << "Questa è una pera" << std::endl;
}

void ItalianVisitor::visit(Maglietta& maglietta) {
    std::cout << "Questa è una maglietta" << std::endl;
}
```

```cpp
/*
 * EnglishVisitor.h
 */

#infdef ENGLISH_H_
#define ENGLISH_H_
```

```
#include "Product.h"

class EnglishVisitor : public Visitor {
public:
    EnglishVisitor();
    virtual ~EnglishVisitor();

    virtual void visit(Pera& pera) override;
    virtual void visit(Maglietta& maglietta) override;
};

#endif /* ENGLISH_H_ */
```

```
/*
 * EnglishVisitor.cpp
 */

#include "EnglishVisitor.h"

EnglishVisitor::EnglishVisitor() {
    // TODO Auto-generated constructor stub
}

EnglishVisitor::~EnglishVisitor() {
    // TODO Auto-generated destructor stub
}

void EnglishVisitor::visit(Pera& pera) {
    std::cout << "This is a pear" << std::endl;
}

void EnglishVisitor::visit(Maglietta& maglietta) {
    std::cout << "This is a t-shirt" << std::endl;
}
```

Di seguito è riportato il `main` per completare l'esempio:

```
int main() {
    // Creazione di alcuni oggetti Product
    Product* pera = new Pera;
    Product* maglietta = new Maglietta;
    Visitor* italianVisitor = new ItalianVisitor;
    Visitor* englishVisitor = new EnglishVisitor;

    // Utiizzo dei Visitors
    std::cout << "Usando l'ItalianVisitor" << std::endl;
    pera->accept(*italianVisitor);
    maglietta->accept(*italianVisitor);

    std::cout << "Usando l'EnglishVisitor" << std::endl;
    pera->accept(*englishVisitor);
    maglietta->accept(*englishVisitor);

}
```

Il *visitor pattern* è conveniente da usare quando l'operazione più frequente è l'aggiunta di nuovi *visitor*. È invece sconveniente quando si tratta di aggiungere nuovi visitabili. È bene utilizzarlo partendo da una collezione di visitabili già esistente e completa.

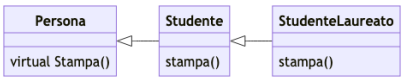


Figura 6: diagram

Altro esempio

```
Persona* p = new Studente;
p->stampa();
```

Il metodo segnato come *virtual* viene automaticamente esteso a tutte le classi figlie. In una qualunque gerarchia di classi si risale per cercare un metodo virtuale. L'unico modo per fermare la catena dell'*override* di un metodo iniziato come virtuale è inserire la keyword `final` in una delle implementazioni derivate. Se una superclasse chiama un proprio metodo nel costruttore, e tale costruttore viene chiamato nel costruttore di una sottoclasse, viene chiamato il metodo della superclasse anche se esso subisce l'override nella sottoclasse. La keyword `override`, per quanto opzionale, dà errore se utilizzata su metodi che non sono virtuali nella classe padre.

STL in C++

Le classi *template* con parametri sono un potente strumento di programmazione che permette di scrivere codice generico e riutilizzabile. La *Standard Template Library* (STL) del C++ è un esempio emblematico di come le classi template possano essere utilizzate efficacemente. Senza l'uso di *template*, per N tipi di dati, M *container* e K$ $algoritmi, sarebbero necessarie $N \times M \times K$ implementazioni nel caso peggiore. Grazie all'uso di template, il numero di implementazioni necessarie si riduce a $N + M + K$.

I *container* nella STL includono *liste*, *adattatori* e *container associativi*. Un esempio di *container* è il `vector<T>`, che rappresenta un array *dinamico* con la sintassi di accesso solita tramite parentesi quadre. Questo *container* permette l'accesso diretto e la modifica degli elementi. I vettori sono confrontabili utilizzando gli operatori `==` e `!=`. C++ permette l'*overloading* degli operatori, rendendo possibile l'uso di questi operatori di confronto in modo naturale. L'uso di container astratti nella STL può portare a problemi di *slicing*, ovvero alla perdita di informazioni specifiche dei tipi derivati quando si lavora con oggetti polimorfici. Questo avviene perché i *container* astratti possono memorizzare solo la parte comune degli oggetti, perdendo le informazioni specifiche dei tipi derivati.

Un altro concetto fondamentale nella STL è l'*iteratore*, un *design pattern* che risolve molti problemi legati alla gestione delle sequenze di dati. Gli iteratori permettono di accedere e manipolare gli elementi dei container in modo uniforme e astratto, indipendentemente dal tipo di container utilizzato.

Smart pointer

Gli *smart pointer* permettono di evitare la necessità di chiamare esplicitamente `delete` sulle variabili puntate, gestendo automaticamente la memoria. Esistono tre tipi principali di *smart pointer*:

- `unique_ptr<T>`: utilizzato per oggetti non condivisi. Questo tipo di *smart pointer* garantisce che ci sia una sola istanza che possiede l'oggetto puntato, assicurando che la memoria venga deallocata automaticamente quando il `unique_ptr` viene distrutto
- `shared_ptr<T>`: utilizzato per oggetti condivisi. Questo tipo di *smart pointer* permette a più `shared_ptr` di condividere la proprietà dello stesso oggetto. La memoria viene deallocata solo dopo che l'ultimo `shared_ptr` che punta all'oggetto viene distrutto
- `weak_ptr<T>`: utilizzato per oggetti condivisi in sola lettura. Questo tipo di *smart pointer* non incrementa il conteggio dei riferimenti dell'oggetto puntato, evitando così cicli di riferimento che potrebbero impedire la deallocazione della memoria.

Segue un esempio di utilizzo di `unique_ptr`:

```
unique_ptr<Song> song2(new Song("Nothing on You"));
```

In questo esempio, `song2` è un `unique_ptr` che possiede un oggetto di tipo `Song` creato dinamicamente. Quando `song2` viene distrutto, la memoria allocata per l'oggetto `Song` viene automaticamente deallocata.

Scala

Scala è un linguaggio di programmazione funzionale sviluppato tra il 2001 e il 2006 da Martin Odersky presso l'École polytechnique fédérale de Lausanne (EPFL). È integrato con Java e tipizzato staticamente, eseguito sulla Java Virtual Machine (JVM), orientato agli oggetti e funzionale, con funzionalità dinamiche. Scala è progettato per essere un *drop-in replacement* per Java, permettendo il riuso delle librerie e dei *tool* esistenti. Questo consente di realizzare progetti misti che utilizzano sia Java che Scala.

Scala è nato per superare i limiti di Java, considerato troppo verboso e con molto codice *boilerplate*, come *getter* e *setter*. Si è voluto sfruttare l'ecosistema di Java, recuperando i suoi punti di forza, quali la popolarità, l'orientamento agli oggetti, la tipizzazione forte, la disponibilità di librerie e la JVM multipiattaforma.

L'interoperabilità tra Scala e Java è quasi completa. Scala può chiamare Java in modo completo, mentre Java può chiamare Scala in modo quasi completo. Scala è compilato in file `.class`, come Java.

In Java quasi tutto è un oggetto, tranne i tipi base per motivi di efficienza. In Scala tutto è un oggetto, con una trasformazione in primitivi gestita nel retroscena. In Java esiste una distinzione tra metodi e operatori, mentre in Scala gli operatori sono metodi e in molti casi è disponibile una doppia sintassi equivalente.

Scala non utilizza il punto e virgola per terminare le istruzioni. I nomi dei tipi iniziano con una lettera maiuscola. I tipi dei parametri e del ritorno seguono il nome invece che precederlo. La *keyword* per i metodi è `def`. Scala distingue tra `val` per le costanti e `var` per le variabili, con type inference statica. L'operatore `return` non è necessario, in quanto viene ritornato il valore dell'ultimo statement dove non specificato. L'operazione di *cast* è sostituita da `.asInstanceOf[Type]` o con funzioni `toType`. L'*import* di un intero package utilizza `._` invece di `.*`. Le chiamate ai metodi senza argomenti non richiedono la coppia di tonde vuote.

In Scala è possibile dichiarare variabili con tipi impliciti o espliciti. Ad esempio, si può definire una somma di numeri, una lista di interi e una mappa che associa una stringa a una lista di interi senza specificare i tipi esplicitamente:

```
val sums = 1 + 2 + 3
val nums = List(1, 2, 3)
val map = Map("abc" -> List(1, 2, 3))
```

Se si desidera specificare i tipi in modo esplicito, il codice diventa:

```
val sum: Int = 1 + 2 + 3
val nums: List[Int] = List(1, 2, 3)
val map: Map[String, List[Int]] = ...
```

A un livello più alto, per verificare se una stringa `name` contiene almeno una lettera maiuscola, in Java si utilizza un ciclo `for`:

```
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

In Scala questa operazione può essere eseguita in modo più conciso utilizzando il metodo `exists`:

```
val hasUpperCase = name.exists(_.isUpperCase)
```

La dichiarazione di una classe in Scala è semplice e diretta. Ad esempio, una classe `Person` con attributi `name` e `age` può essere definita come segue:

```
class Person(var name: String, var age: Int)
```

In Scala i metodi *getter* e *setter* sono definiti implicitamente. Se si desidera definirli esplicitamente, è possibile farlo nel seguente modo:

```
class Person(var name: String, private var _age: Int) {
    // getter
    def age = _age

    // setter
    def age_=(newAge: Int) {
        println("Changing age to: " + newAge)
        _age = newAge
    }
}
```

Una caratteristica distintiva di Scala è l'assenza di riferimenti nulli. Se un metodo potrebbe non restituire nulla, si utilizza l'oggetto `Option`, che può essere `Some(valore di ritorno)` o `None`. Questo oggetto può essere controllato con una espressione `match`. Ecco un esempio dell'uso di `None`:

```
def toInt(in: String): Option[Int] = {
    try {
```

```
        Some(Integer.parseInt(in.trim))
    } catch {
        case e: NumberFormatException => None
    }
}
```

In Scala le variabili sono trattate come funzioni. Un valore è essenzialmente una funzione senza parametri che restituisce il valore stesso. Questo principio è noto come *accesso uniforme*, dove valori e funzioni senza parametri sono indistinguibili. Non è possibile utilizzare le parentesi vuote per chiamare una funzione senza parametri.

I linguaggi funzionali, come ML, OCaml e Haskell, sono spesso considerati accademici e di apprendimento non immediato. La programmazione funzionale offre però soluzioni efficaci per la concorrenza. Introdurre anche piccole sezioni di codice funzionale può semplificare la risoluzione di molti problemi. Questo approccio rappresenta un modo diverso di pensare, difficile da apprendere ma estremamente utile una volta compreso.

Scala può essere utilizzato dinamicamente tramite un ambiente *read-eval-print* direttamente dal terminale. A livello di tipizzazione, pur mantenendo una tipizzazione forte, Scala utilizza un tipo di *duck typing* sicuro. Questo permette di definire funzioni che operano su oggetti con determinati metodi, senza necessariamente condividere una gerarchia di classi. Ad esempio:

```
def doTalk(any: {def talk: String}) {
    println(any.talk)
}

class Duck { def talk = "Quack" }
class Dog { def talk = "Bark" }

doTalk(new Duck)
doTalk(new Dog)
```

Questo tipo di tipizzazione dinamica permette il polimorfismo senza ereditarietà, sostituendo la gerarchia delle classi con metodi e proprietà comuni.

Scala supporta anche i parametri di *default*, noti come *named parameters*:

```
def hello(foo: Int = 0, bar: Int = 0) {
    println("foo: " + foo + " bar: " + bar)
}

hello   // 0, 0
hello(1)      // 1, 0
hello(foo=7)  // 7, 0
hello(foo=5, bar=9)  // 5, 9
```

In Scala, ogni espressione restituisce un valore e non esiste il tipo `void`. Ad esempio:

```
val a = if(true) "yes" else "no"

val b = try {
    "foo"
} catch {
    case _ => "error"
}

val c = {
    println("hello")
    "foo"
}
```

Scala supporta anche la *lazy evaluation*, dove il valore di una variabile è calcolato solo al momento del suo primo utilizzo:

```
lazy val foo = {
    println("init")
    "bar"
}

foo // stampa "init"
```

```
foo // non stampa niente
foo // non stampa niente
```

Scala permette inoltre di definire funzioni annidate, dove una funzione può contenere altre funzioni al suo interno.

```
def foo(x: Int) {
    def bar(y: Double) {
        ...
    }
    bar / x
    ...
}
```

Un'altra caratteristica interessante è il *passaggio per nome*. In questo caso, i valori dei parametri non sono calcolati al momento della chiamata, ma possono essere intere funzioni o composizioni di funzioni, a patto che abbiano il tipo di ritorno uguale al tipo del parametro. Il valore viene calcolato solo al momento dell'uso finale. Questo permette al compilatore di ottimizzare e parallelizzare le chiamate, invece di forzare l'esecuzione in un certo ordine. Ad esempio:

```
// Per valore
def f(x: Int, y: Int) = x

// Per nome
def f(x: => Int, y: => Int) = x
```

Utilizzando la chiamata per nome, si lascia al compilatore più libertà di ottimizzare e parallelizzare le chiamate, migliorando l'efficienza del codice.

Scala: esercitazione 1

Esercizio 1

I metodi si definiscono nel modo seguente. Per sommare due numeri, ad esempio:

```
def add (a: Int, b: Int) : Int = a+b
```

Possiamo definire un valore come ritorno di una funzione:

```
val m: Int = add(1,2)
```

Il valore di `m` può poi essere stampato:

```
println(m)
```

Esercizio 2

```
def fun(a:Int): Int = { // nota bene l'uguale
    a+1
    a-2 // niente ;
    a*3
}
```

Eseguire

```
val p: Int = fun(10)
println(p)
```

stampa `30`. La funzione ritorna 30, perché return è ultima istruzione, quindi `a*3`

Esercizio 3

```
val i=3
val p: if (i>0) -1 else -2
```

In Scala tutto è una funzione. In questo caso questa funzione dipende da `i`, che però è una variabile libera. Scala va quindi a cercare la cosiddetta *chiusura*, cioè va a cercare il valore di `i` per chiudere la funzione. La trova e la usa in modo dinamico, cioè se è dichiarata come `val` non succede nulla perché non cambia mai, ma se è `var` prende l'ultimo valore che ha assunto.

Esercizio 4

L'equivalente per Scala dello *switch statement* è `match`, come mostrato nel seguente esempio:

```
def errorMsg (errorCode: Int) = errorCode.match {
    case 1 => "File not found"
    case 2 => "Permission denied"
    case 3 => "Invalid operation"
}
```

Esercizio 5

```
def sum (n: Int) : Int = if(n==0) 0 else n+sum(n-1)
val m = sum(10)
```

Stampando `m` otteniamo il valore 55. Si tratta di ricorsione non *tail*. L'equivalente *tail recursive* è

```
def sum(n: int, acc: Int) : Int = if(n==0) acc else sum(n-1, acc+n)
```

Esercizio 6

Dato il seguente codice:

```
def sqr(x: Int) = x*x
def cube(x: Int) = x*x*x
```

L'intenzione è costruire una funzione che sommi quadrati (o cubi) di tutti i numeri fra due parametri `a` e `b`. Una possibile soluzione non funzionale è la seguente:

```
def sumSimple(a: Int, b: Int): Int = if (a==b) a else a+sumSimple(a+1,b)
def sumSquares(a: Int, b:Int):Int = if(a==b) sqr(a) else sqr(a)+sumSquares(a+1,b)
def sumCube(a: Int, b:Int):Int = if(a==b) cube(a) else cube(a)+sumCube(a+1,b)
```

La soluzione funzionale, invece, è

```
def identity (x:Int) : Int = x;
def sum(f: Int => Int, a:Int, b:Int) : Int = if(a==b) f(a) else f(a)+sum(f, a+1, b)
sum (identity, 1, 10)
sum(sqr, 1, 10)
sum(cube, 1, 10)
```

In particolare,

```
sum (x=> x*x*x, 1, 10)
```

è equivalente a `sum(cube, 1, 10)`, ma in versione anonima.

Esercizio 7

Si analizzi il comportamento funzionale di Scala nell'ambito delle funzioni passate come parametro.

```
val a = List (1, 2, 3, 4, 5, 6, 7)
val b = a.map(x => x*x)
```

La funzione `map` mappa la funzione anonima `x => x*x` per ogni elemento `x` della lista. Non è importante come venga eseguita la mappatura, la cui implementazione è delegata al compilatore. Quest'ultimo ottimizza solitamente la funzione eseguendo la funzione anonima in parallelo su sottoinsiemi della lista, per poi ricostruire il risultato completo alla fine dell'elaborazione.

Un altro esempio è la funzione `filter`

```
val c = b.filter (x => x<5)
```

che in questo caso restituisce il sottoinsieme degli elementi di `b` minori o uguali a 5.

La funzione `reduce`, invece, prende le coppie di elementi presenti in `b` e ne restituisce la lista delle somme:

```
val d = b.reduce ( (x,y) => x+y )
```

Nel seguente esempio, la funzione `even` ritorna `true` se il suo parametro è pari. Il meccanismo di inferenza dei tipi del compilatore di Scala determina automaticamente che il tipo di ritorno sia `Boolean`.

```
def even(x: Int) = (x%2) == 0
```

Un altro esempio di funzione predefinita per elaborare le liste è`forall`, che esegue un'asserzione su ogni elemento della lista e restituisce `true` soltanto se tutti gli elementi la soddisfano. Nell'esempio riportato di seguito è utilizzata come asserzione la funzione `even` definita in precedenza:

```
a.forall(even)
```

Una funzione simile è `exists`, che ritorna `true` soltanto se almeno uno degli elementi della lista soddisfa l'asserzione passata per parametro:

```
a.exists(even)
```

La funzione predefinita `takeWhile` pesca elementi dalla lista finché non ne trova uno che non rispetti la condizione passata per parametro. A tal punto, la funzione si ferma. Nel seguente esempio:

```
a.takeWhile(even)
```

la funzione si ferma immediatamente, ritornando una lista vuota, perché il primo elemento di `a` è 1, che è dispari.

Infine, la funzione `partition` divide una lista in due sottoliste in base alla condizione passata per parametro. Nel seguente esempio

```
a.partition(even)
```

a partire da `a` vengono create due liste, una di numeri pari e una di numeri dispari.

Esercizio 8

Il seguente esempio riguarda la chiusura.

```
def fun (x: int) = {
    val y=1
    val r = {
    val y=2
    x+y // quale valore prende di y? 2
    }
    Println(r)
    Println(x+y) // quale valore prende di y? 1
}
```

Esercizio 9

Le funzioni Scala possono ritornare altre funzioni, come nel seguente esempio:

```
def fun(): Int => Int = {
    def sqr (x: Int) : Int = x*x
    sqr
}
```

Esercizio 10

Si analizzi il seguente codice Scala:

```
def fun1() : Int => Int = {
    val y=1
    def add(x:Int) = x+y
    add
}

def fun2() = {
    val y=2
    val f = fun1()
    f(10)
}
```

Quando viene calcolata la chiusura? La chiusura viene computata quando `y` viene dichiarata, quindi sceglie `y=1` e ritorna 11.

Esercizio 11

È possibile definire una funzione che componga altre funzioni. Ad esempio:

```
def compose (f: Int => Int, g: Int => Int): Int => Int = x => f(g(x))
def sqr(x:Int) = x*x
def cube(x:Int) = x*x*x
val g = compose(sqr, cube)
```

Scala: esercitazione 2

Esercizio 1

Si consideri la seguente funzione:

```
def addA (x: Int, y: Int): Int = x+y
```

L'intenzione è trasformarla in una sequenza di funzioni che prendano tutte un parametro solo:

```
def addB (x: Int): Int => Int = y => x+y
```

In pratica il parametro `y` è stato fatto diventare l'ingresso di una funzione, ovvero il parametro della funzione di ritorno di `addB`. È possibile scrivere anche `addB(3)(4)`, ossia passare contemporaneamente `x` e `y`. Questa tecnica si chiama *currying*.

Esercizio 2

Si consideri la funzione

```
def addA(x: Int, y: Int, z: Int) = x+y+z
```

L'obiettivo è simile a quello dell'esercizio 1, ma stavolta passando da 3 parametri a 1:

```
def addZ (x: Int) : Int => (Int => Int) = y => (z => x+y+z)
```

Il parametro `y` viene passato alla prima funzione, che ritorna `z`. Quest'ultimo è passato direttamente alla seconda funzione che ritorna `x+y+z`. `AddZ` è quindi una funzione che va da un `Int` a una funzione, che a sua volta va da un `Int` a un `Int`. Anche in questo caso è possibile scrivere in modo compatto `addZ(1)(2)(3)`, che corrisponde a passare in una volta sola `x`, `y` e `z`.

Esercizio 3

Per definire una lista piena:

```
val a = list (1, 2, 3)
```

Per definire una lista nulla:

```
val b = Nil
```

La lista nulla è diversa da una lista senza elementi:

```
val c = List()
```

Per aggiungere `d` (di valore 0) come elemento nella lista `a`:

```
val d= 0::a
```

L'operazione è equivalente a `a.add(0)` in Java.

Si osservi la seguente funzione:

```
def fun (a: List[Int]): Int = a match {
    // se la lista ha 3 elementi di cui il primo 0, allora faccio p+q
    case List(0, p, q) => p+q
    case _ => -1 → case _ // significa in tutti gli altri casi
}
```

La seguente funzione è in grado di ritornare la lunghezza di una lista indipendentemente dal suo essere piena, vuota o nulla:

```
def lenght (a: List[Int]): Int = a match {
    case Nil => 0
    case h::t => 1+lenght(t)
}
```

Per concatenare due liste si procede nel modo seguente:

```
val f = a ++ List(4) oppure val f = a ::: List(4)
```

Esercizio 4

Siano date le funzione `f1` e `f2`:

```
def f1() = {
    println("Sono f1")
    10
}

def f2() = {
    println("Sono f2")
    20
}
```

Si desidera ora definire una funzione `if` in cui se la condizione è vera viene eseguita `f1`, se falsa `f2`. Si vuole però che `f1` e `f2` vengano eseguite solo quando richiesto dall'`if`:

```
def myIf (cond: Boolean, thenPart: => Int, elsePart: => Int) = {
    if(cond) thenPart else elsePart
}
```

La tecnica si chiama passaggio per nome. `: => Int` significa che la funzione va da niente a un `Int`.

Esercizio 5

Questo esercizio spiega il concetto di *referencial transparency*.

A tal fine è presentata la funzione `withdraw`:

```
var balance = 1000

def withdraw (amount: Int) = {
    balance = balance - amount
    balance
}
println(withdraw(100) )
```

Il valore di ritorno della funzione non dipende solo dal parametro della funzione, ma anche dallo stato di `balance`. Questa funzione non è dunque *referencially transparent* perché non è possible prendere il risultato e sostituirlo da altre parti, in quanto dipende dal valore di `balance` che è esterno alla funzione. Dati gli stessi parametri, non è detto che il risultato sia lo stesso ad ogni chiamata. Questo inoltre non permette al compilatore di parallelizzare l'esecuzione.

Esercizio 6

```
def hello () = {
    println ("hello")
    10
}

lazy val a = hello()
```

Il blocco valuta il valore della variabile solo se e quando verrà usata. Cosa cambia rispetto al passaggio per nome? Con il passaggio per nome viene realmente chiamata la funzione, quindi se la chiamata avviene 5 volte, essa viene computata 5 volte. Con la variabile *lazy* viene computato il valore una sola volta, alla prima chiamata, e poi il risultato è salvato in memoria.