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.

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:

```c
// 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.

```
#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`:

```
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`.

```
classDiagram
    class Movement {
        <<interface>>
        + move(p: Point)
    }

    class AggressiveMovement {
        + move(p: Point)
    }

    class RelaxedMovement {
        + move(p: Point)
    }

    class OtherMovement {
        +move(p: Point)
    }

    class Robot {
        + Robot(s: Movement)
        + ChangeStrategy(s: Movement)
    }

    Movement <|.. OtherMovement
    Movement <|.. AggressiveMovement
    Movement <|.. RelaxedMovement

    Robot o--|> Movement
```

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.

```
classDiagram
    class Alimenti {
        <<abstract>>
    }

    class Pera
    class Pasta

    Alimenti <|.. Pera
    Alimenti <|.. Pasta
```

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

Soluzione *visitor*:

```
classDiagram
    class Alimento {
        <<abstract>>
        +accept(v: Visitor) String
    }

    class Pera {
        +accept(v: Visitor) String
    }

    class Pasta {
        +accept(v: Visitor) String
    }

    Alimento <|.. Pera
    Alimento <|.. Pasta
```

Com'è fatto un *visitor*?

```
classDiagram
    class Visitor {
        <<interface>>
        +visit(p: Pasta) String
        +visit(p: Pera) String
    }
```

Deve saper visitare ogni singola classe di nostro interesse. Nel nostro caso ne abbiamo due tipi:

```
classDiagram
    class Visitor {
        <<interface>>
        +visit(p: Pasta) String
        +visit(p: Pera) String
    }

    class TraduttoreItaliano {
        +visit(p: Pasta) String
        +visit(p: Pera) String
    }

    class TraduttoreInglese {
        +visit(p: Pasta) String
        +visit(p: Pera) String
    }

    Visitor <|.. TraduttoreItaliano
    Visitor <|.. TraduttoreInglese
```

```
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:

```
classDiagram
    class Alimenti~T~ {
        <<abstract>>
        +accept (v: Visitor)
    }
```

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:

```
classDiagram
    direction LR
    namespace Automobile {
        class Pedale
        class Motore
        class Ruote
    }

    Pedale <|-- Motore
    Motore --|> Ruote
    Pedale --|> Motore
    Pedale --|> Ruote

    class CarSystem
    CarSystem --> Pedale
    CarSystem --> Motore
    CarSystem --> Ruote
```