

Programmazione: una breve guida di stile

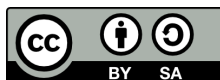
[illegible]

Moreno Marzolla

Università di Bologna, Dipartimento di Informatica—Scienza e Ingegneria

Ultimo aggiornamento: 18 luglio 2023

Copyright 2022, 2023 Moreno Marzolla www.moreno.marzolla.name



Quest'opera è rilasciata con licenza Creative Commons Attribuzione - Condividi allo stesso modo 4.0 Internazionale. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-sa/4.0/> o spedisce una lettera a Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Indice

1 Forma.....	9
1.1 Indentazione.....	9
1.2 Spaziatura.....	10
1.3 Layout.....	12
1.4 Identificatori.....	13
1.5 Commenti.....	14
1.6 Costanti magiche.....	16
1.7 Rendere espliciti gli intenti.....	18
1.8 Priorità degli operatori.....	18
1.9 Lunghezza delle funzioni.....	19
2 Sostanza.....	23
2.1 Evitare ripetizioni.....	23
2.2 Semplificare le strutture condizionali.....	27
2.3 for o while?.....	29
2.4 break e continue.....	34
2.5 Array vs Tuple.....	35
2.6 Variabili globali.....	37
2.7 Ricorsione.....	37
2.8 Asserzioni.....	38
2.9 Programmazione difensiva.....	39
2.10 Evitare soluzioni "astute".....	41
2.11 Macro.....	43

Introduzione

Programmare è l'attività con cui si “istruisce” un dispositivo automatico di calcolo su come risolvere un problema o classe di problemi. In pratica, la programmazione viene associata alla scrittura di un programma usando un apposito linguaggio (C, C++, Java, Python, ...).

Molti scrivono programmi pensando che siano destinati esclusivamente al compilatore che li tradurrà in codice macchina, o all'interprete che li eseguirà. Niente di più sbagliato: un programma è il mezzo con cui si esprime un'idea; programmare è quindi un atto creativo simile alla scrittura di una lettera. Come una lettera, un programma deve esprimere un concetto in modo chiaro e sintetico; a differenza di una lettera, un programma deve essere scritto rispettando regole molto rigide, imposte dal linguaggio di programmazione, affinché venga interpretato o compilato correttamente. Una volta scritto, un programma potrà in seguito richiedere modifiche o estensioni per aggiungere nuove funzionalità o per correggere errori scoperti durante l'uso.

Una volta compreso che un programma è destinato sia alle macchine che alle persone, diventa evidente la necessità di adottare delle regole di stile: un testo non deve rispettare solo la grammatica della lingua in cui è scritto, ma deve anche essere chiaro e scorrevole; dopo tutto, un programma verrà letto molte più volte di quante venga scritto! Per lavoro mi trovo a dover leggere e valutare programmi scritto da terzi, e ho osservato che chi scrive codice disordinato spesso ragiona in modo disordinato e tende a produrre programmi di bassa qualità.

Un programma, quindi, dovrebbe soddisfare alcuni requisiti spesso in conflitto tra loro:

- *Comprensibilità*: l'intento del programmatore deve risultare chiaro dalla lettura del codice e dall'eventuale documentazione allegata (tra cui i commenti nel codice);
- *Mantenibilità*: deve essere facile aggiungere nuove funzionalità o correggere errori. Un programma dovrebbe essere strutturato in modo tale da consentire modifiche localizzate;
- *Efficienza*: un programma non deve richiedere una quantità eccessiva di risorse (tempo, spazio di memoria) per produrre il risultato.

Questa dispensa descrive alcune buone pratiche di programmazione che hanno lo scopo di migliorare la qualità dei programmi. Le buone pratiche illustrate nel seguito non sono il frutto di idee personali; al contrario, si tratta di idee ampiamente condivise dalla comunità informatica che sono già state messe per iscritto in molti libri (si vedano i riferimenti bibliografici in fondo a questo documento).

Le regole che presenteremo vanno intese come suggerimenti da cui è possibile derogare quando ci siano ragioni valide per farlo. Si diventa programmatori esperti quando si matura l'esperienza necessaria per capire quando derogare; fino ad allora consiglio di attenersi il più possibile alle regole.

In questa dispensa vengono usati frammenti di codice di esempio in linguaggio C o Java, per cui è necessario un minimo di familiarità con questi linguaggi. Tuttavia, la maggior parte delle regole sono valide a prescindere dal linguaggio di programmazione che si usa.

Chi desidera approfondire gli argomenti trattati troverà molti altri suggerimenti nel libro:

Steve McConnell, *Code Complete*, 2nd edition, Microsoft Press; 2004, ISBN 978-0735619678

Personalmente ritengo l'acquisto del libro di McConnell uno dei migliori investimenti che abbia fatto: non c'è una pagina da cui non abbia imparato qualcosa di utile!

Un altro libro consigliato è

Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, 2nd edition, McGraw-Hill, 1978, ISBN 0-07-034207-5

Si tratta di un testo molto datato (gli esempi di codice sono in FORTRAN e PL/I) che è comunque rilevante anche oggi: questa dispensa è fortemente ispirata dal libro di Kernighan e Plauger, da cui vengono ripresi molti dei consigli.

È importante sottolineare che, sebbene ampiamente condivise, le regole descritte qui non sono universali: molte organizzazioni hanno le proprie, che potranno in parte differire da quelle presentate nel seguito. Inoltre, linguaggi di programmazione diversi hanno consuetudini diverse: ad esempio, nel linguaggio C si preferiscono identificatori brevi, mentre in Pascal o Java si usano identificatori più lunghi e descrittivi.

Sebbene questo documento sia rivolto principalmente a studenti e alle studentesse dei miei corsi, ritengo che possa essere utile anche a chi si è appena affacciato alla programmazione, e magari a chi ha già fatto della programmazione il proprio lavoro.

Questa dispensa è divisa in due parti. Nella prima parte ("Forma") verranno illustrate alcune regole per migliorare l'aspetto dei programmi. Non è solo una questione estetica: un programma ordinato è più facile da comprendere ed è quindi meno probabile che contenga errori. La seconda parte ("Sostanza") sarà dedicata alle regole per migliorare le funzionalità dei programmi, ad esempio per renderli più efficienti o robusti.

1 Forma

*Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.*

(Martin Fowler e Kent Beck)

1.1 Indentazione

Il termine “indentare” denota la pratica di far “rientrare” le righe di codice sorgente in modo da evidenziarne la struttura a blocchi. Una indentazione appropriata aiuta a cogliere la struttura di un programma a colpo d’occhio.

Ad esempio, consideriamo una funzione `find_max()` in linguaggio C che restituisce il valore massimo in un array non vuoto di lunghezza n . Mostriamo due versioni identiche tranne che per l’indentazione.

✗ Nessuna indentazione	✓ Indentazione corretta
<pre>int find_max(int a[], int n) { int i, m = a[0]; for (i = 1; i < n; i++) { if (a[i] > m) m = a[i]; } return m; }</pre>	<pre>int find_max(int a[], int n) { int i, m = a[0]; for (i = 1; i < n; i++) { if (a[i] > m) m = a[i]; } return m; }</pre>

Nella versione indentata correttamente (a destra) si coglie subito la presenza del ciclo “for” e della struttura condizionale nel suo interno; chi ha un minimo di pratica di programmazione riconosce questo schema e può intuire meglio lo scopo della funzione. Codice non indentato (a sinistra) o indentato male non è solo difficile da leggere, ma può anche nascondere errori. Consideriamo l’esempio seguente in linguaggio C:

✗ Indentazione fuorviante	✓ Indentazione corretta
<pre>if (a > max) a = max; b = max - 1;</pre>	<pre>f (a > max) a = max; b = max - 1;</pre>

Nel codice a sinistra potrebbe sembrare che l'istruzione `b = max - 1` venga eseguita se `a > max`. In realtà non è così: il ramo “vero” non è racchiuso tra parentesi graffe, per cui in base alle regole del C è composto solo dalla riga successiva. A destra viene mostrato lo stesso codice indentato in modo corretto.

Alcuni linguaggi di programmazione, come Python, sfruttano l'indentazione per definire la struttura a blocchi di un programma. In Python non esistono le parentesi graffe per raggruppare istruzioni in blocchi, ma il raggruppamento deriva dall'indentazione.

Esistono diversi stili di indentazione che differiscono nel modo in cui vengono posizionate le parentesi graffe. Alcuni (es., lo “stile GNU”) posizionano le parentesi graffe su righe a se stanti, in modo da rendere ancora più evidente la struttura a blocchi del codice; altri (es., lo “stile K&R”, da Kernighan & Ritchie, gli autori del manuale di riferimento del linguaggio C “The C Programming Language”) favoriscono un layout più compatto.

Indentazione stile “GNU”	Indentazione stile “K&R”
<pre>int power(int x, int n) { int result; if (n < 0) { result = 0; } else { result = 1; while (n > 0) { result *= x; n--; } } return result; }</pre>	<pre>int power(int x, int n) { int result; if (n < 0) { result = 0; } else { result = 1; while (n > 0) { result *= x; n--; } } return result; }</pre>

Personalmente adotto l'indentazione “K&R” perché produce codice più compatto e quindi sfrutta meglio lo spazio a disposizione. In questo stile le parentesi graffe di apertura di un blocco vengono messe sulla stessa riga della parola chiave `if`, `else`, `while`, `for`; le parentesi di chiusura vanno su una riga a se stante, tranne nel caso del ramo vero di un `if` seguito da `else`. Le parentesi graffe di apertura e chiusura del corpo di una funzione vanno sempre su righe a se stanti.

Quasi tutti gli ambienti di sviluppo (IDE) consentono di indentare il codice in modo automatico; non esiste quindi alcun motivo valido per tollerare programmi scritti in modo sciatto.

 Indentare il codice in modo consistente

1.2 Spaziatura

Un programma fitto e privo di spazi è confuso e difficile da leggere. Esempio:

✖ Difficile da leggere

```
int i,j;
while(i<j){//Un commento
    int m=(i+j)/2;//Un altro commento
    if(v[i]<k)
        j=m-1;//Terzo commento
    else if(v[i]>k)
        i=m+1;//Fine
}
```

✔ Spaziatura adeguata

```
int i, j;
while (i < j) {    // Un commento
    int m = (i+j)/2; // Un altro commento
    if (v[i] < k)
        j = m-1;    // Terzo commento
    else if (v[i] > k)
        i = m+1;    // Fine
}
```

La versione di destra è più leggibile, ed è stata ottenuta aggiungendo spazi:

- attorno alle parentesi () nelle espressioni condizionali del `while` e degli `if`;
- attorno ad alcuni operatori (<, > e =);
- prima delle parentesi graffe aperte {
- prima e dopo il delimitatore di inizio dei commenti //

È possibile aggiungere spazi in altri punti (es., attorno agli operatori – e + nelle espressioni).

Quanto sopra è soggetto a preferenze personali; ciascuno potrebbe trovare varie soluzioni più o meno leggibili. Come sempre, in assenza di vincoli stringenti (es., dettati dall'azienda per cui si lavora o dal committente), si usi buon senso.

👉 Usare gli spazi in modo appropriato per favorire la leggibilità del codice

1.3 Layout

I primi videotermini erano in grado di visualizzare 25 righe per 80 colonne di testo; queste dimensioni erano imposte principalmente dai limiti tecnologici dei tubi catodici e da altre convenzioni risalenti agli albori dell'informatica. Oggi questi limiti sono superati: i monitor hanno una risoluzione elevata che, combinata con il formato dell'immagine 16:9, fornisce ampio spazio in orizzontale. Ritengo tuttavia che non sia una buona idea sfruttare questo spazio per scrivere righe di codice lunghe, per almeno due motivi:

- È molto scomodo leggere righe di testo lunghe, come ci si può rendere conto da questa pagina (volutamente stampata in modalità “landscape”). Non è un caso che i libri siano solitamente più alti che larghi, e i quotidiani usino colonne di testo molto più strette della larghezza della pagina;
- Non è detto che chi riceve il codice abbia una risoluzione orizzontale sufficiente per visualizzare le righe senza che vengano mandate a capo o troncate. Anche nel caso in cui si disponga di una risoluzione orizzontale sufficiente, c'è chi ingrandisce i font per leggere meglio, oppure chi preferisce tenere diverse finestre affiancate; questo riduce lo spazio orizzontale a disposizione.

Molti editor di testo usano di default una larghezza di 80 caratteri, oppure mostrano un riferimento (es., una linea verticale) in corrispondenza della colonna 80. Suggerisco di fare il possibile per mantenere le righe di codice e i commenti entro gli 80 caratteri.

☞ Mantenere la lunghezza delle righe di codice e di commento al di sotto di 80 caratteri.

È importante tenere presente che non tutti condividono la mia opinione. Linus Torvalds, l'autore del sistema operativo Linux, ritiene che spezzare righe di codice sia fonte di potenziali problemi, per cui ritiene accettabile scrivere righe di codice lunghe. Le sue argomentazioni, che possono essere lette all'indirizzo <http://lkml.iu.edu/hypermail/linux/kernel/2005.3/08168.html>, sono in parte condivisibili.

1.4 Identificatori

Gli *identificatori* sono nomi a cui vengono associate delle entità del linguaggio di programmazione; le entità possono essere variabili, costanti, tipi di dato, funzioni e altro.

Ogni linguaggio definisce le proprie regole per la costruzione di identificatori validi. Ad esempio, in C/C++ un identificatore può contenere esclusivamente lettere minuscole, maiuscole, cifre numeriche e il simbolo `_`, e non può iniziare con una cifra numerica. In Java è possibile usare anche il simbolo `$`, e certi tipi di identificatori devono iniziare con una lettera maiuscola (es., i nomi di classi). In alcuni linguaggi esiste un limite alla lunghezza di un identificatore. Identificatori non validi vengono rifiutati dal compilatore/interprete.

Ogni linguaggio ha le proprie consuetudini sui nomi degli identificatori. In C si preferiscono identificatori brevi, anche composti da una singola lettera. Nei linguaggi della famiglia del Pascal (Pascal, Ada, Delphi, ecc.) si preferiscono nomi lunghi. Trattandosi di consuetudini e non di regole formali, ciascuno è libero di adottare il proprio stile per la scelta dei nomi, anche se suggerisco di adeguarsi alle consuetudini del linguaggio.

 **Attenersi alle consuetudini del linguaggio di programmazione per la scelta dei nomi degli identificatori**

Il nome di un identificatore dovrebbe soddisfare due requisiti contrastanti:

- essere abbastanza breve, per ridurre la probabilità che il programmatore commetta errori di scrittura;
- suggerire il significato dell'oggetto a cui si riferisce.

Un identificatore breve non è sempre significativo, mentre un identificatore significativo non è sempre breve. In generale può essere difficile trovare un compromesso tra queste due esigenze.

	✗ No	✓ Sì
Indici di cicli	indice1 indice2 indice3	i j k
Temperatura in gradi C	x	temp temperatura
Dimensioni di una matrice	dim1 dim2 a b	n m righe colonne r c
Somma di una sequenza di valori	prod tmp val ¹	sum somma
Albero dei cammini minimi (<i>minimum spanning tree</i>)	a tree	mst minSpanningTree min_spanning_tree
Funzione che crea una struttura “grafo”	f() crea()	creaGrafo() crea_grafo() nuovoGrafo() nuovo_grafo()

Per migliorare la leggibilità degli identificatori lunghi suggerisco di usare il carattere `_` come separatore (es., `min_spanning_tree`) anziché le maiuscole (es., `minSpanningTree`), perché siamo già abituati a leggere parole separate da spazi. Il carattere `_` rende più facile cogliere le singole componenti del nome.

👉 Usare nomi appropriati per gli identificatori

1.5 Commenti

I programmazione consentono di inserire nei programmi dei commenti che vengono ignorati dal compilatore o dall'interprete. I commenti vengono usati per diversi scopi:

- per documentare l'uso di funzioni, classi, metodi;
- per descrivere parti del codice che potrebbero risultare di difficile comprensione;
- per specificare metadati relativi al programma (es., autore, data dell'ultima modifica, licenza, ecc.);
- per specificare direttive particolari da passare a preprocessori o estensioni del compilatore.

Idealmente, ogni funzione non banale dovrebbe essere preceduta da un commento che specifichi:

- cosa calcola (*postcondizione*);
- qual è il significato dei parametri (se presenti);
- quali vincoli ciascun parametro deve soddisfare (es., diverso da NULL? deve avere particolari valori, es. maggiore o uguale a zero?). Tali vincoli sono detti *precondizioni*.

Esempio:

¹ Sì, mi sono stati consegnati programmi in cui una sommatoria veniva chiamata in questi modi.

✓ Commenti dettagliati

```
/**
 * Restituisce la posizione (indice) del valore di rango k in v[]; il
 * valore di rango k è quello che occuperebbe la k-esima posizione se v[]
 * fosse ordinato; questo problema prende il nome di problema della selezione.
 *
 * Input:
 * - v[] array di interi arbitrari non vuoto;
 * - n lunghezza di v[]; è richiesto che n>0;
 * - k rango dell'elemento da selezionare (k=0 indica che si vuole
 *   selezionare il minimo, k=n-1 indica che si vuole selezionare il massimo);
 *   è richiesto che 0 <= k <= n-1
 * Output:
 * - la posizione (indice) in v[] dell'elemento di rango k; si garantisce che
 *   il risultato sia sempre compreso tra 0 e n-1, inclusi.
 */
int select(int v[], int n, int k);
```

👁️ Commentare i punti critici del codice, in modo chiaro e grammaticalmente corretto

Un errore comune è quello di commentare il codice riga per riga. Ciò ha l'unico effetto di infastidire chi lo legge e appesantire inutilmente il sorgente.

✗ Commenti inutili

```
v = v + 1;      /* incrementa v */
if (v > 10) {   /* se v e' maggiore di 10 */
    v = 0;      /* setta v a zero */
}
Prim(G, v); /* esegui l'algoritmo di Prim su G */
```

✓ Commenti appropriati

```
/* Individua la posizione i del primo valore negativo
nell'array a[] di lunghezza n >= 0; al termine si ha
i == n se non esiste alcun valore negativo */
int i = 0;
while (i < n && a[i] >= 0) {
    i++;
}
```

Invece di parafrasare il codice, usare i commenti per descrivere le funzionalità di una porzione di codice o l'algoritmo impiegato per risolvere il problema, nel caso in cui queste informazioni non siano facilmente comprensibili guardando il codice.

👁️ Non parafrasare il codice riga per riga

I commenti possono essere utili per descrivere il significato di variabili o costanti, nei casi in cui i nomi non siano auto-esplicativi. Non è sempre necessario descrivere tutte le variabili usate, ma è buona norma farlo almeno per quelle più importanti o il cui significato non sia immediato.

✓ Commenti dettagliati

```
int boardDim;      /* dimensione (lato) della scacchiera */
float g_cor, g_max; /* guadagno corrente e massimo */
double min_cost;    /* costo del cammino di costo minimo */
```

Un problema frequente dei commenti è che talvolta non sono “allineati” con il codice: spesso capita di fare una modifica ad un frammento di codice dimenticandosi di modificare il commento associato. Esempio:

✘ **Codice e commenti non concordano**

```
/* Restituisce true (nonzero) se la soluzione è valida */  
void check_sol(int *v, int n);
```

Il commento dichiara che la funzione restituisce un valore, ma la funzione è dichiarata essere *void*: chi ha ragione?

👁 Assicurarsi che il codice e i commenti concordino

1.6 Costanti magiche

In molte situazioni occorre fare uso di costanti numeriche i cui valori hanno un significato particolare. Ad esempio, supponiamo di modellare la propagazione di molecole di gas in una matrice in due dimensioni. Ogni cella della matrice può essere vuota, oppure contenere una molecola di gas, oppure contenere un ostacolo. In linguaggio C possiamo rappresentare ciascuno stato con un valore, ad esempio 0, 1, 2. Se però usiamo direttamente i simboli 0, 1, 2 nel codice, chi legge potrebbe non capire immediatamente a cosa corrisponde ciascun valore: lo zero rappresenta la cella vuota? la cella che contiene un ostacolo? I letterali numerici² diversi da 0 e 1 che compaiono in un sorgente sono detti “costanti magiche”.

Ogni ambiguità può essere risolta introducendo degli identificatori, come nell'esempio seguente.

✘ **Cosa rappresentano i valori 0, 1, 2?**

```
const int a = v[i][j];  
const int b = v[i-1][j];  
const int c = v[i][j-1];  
const int d = v[i-1][j-1];  
if (((a == 0) != (b == 0)) &&  
    ((c == 1) != (d == 1))) ||  
    (a == 2) || (b == 2) ||  
    (c == 2) || (d == 2)) {  
    ...
```

✔ **Codice non ambiguo**

```
enum {EMPTY, GAS, WALL};  
  
const int a = v[i][j];  
const int b = v[i-1][j];  
const int c = v[i][j-1];  
const int d = v[i-1][j-1];  
if (((a == EMPTY) != (b == EMPTY)) &&  
    ((c == GAS) != (d == GAS))) ||  
    (a == WALL) || (b == WALL) ||  
    (c == WALL) || (d == WALL)) {  
    ...
```

Nella versione a destra si usano le costanti `EMPTY`, `GAS` e `WALL`, definite tramite un tipo enumerativo del linguaggio C. Il tipo enumerativo consente di definire dei simboli a cui assegnare valori interi costanti; nel caso in cui i valori non siano assegnati esplicitamente, vengono usati gli interi 0, 1, ...

² Un *letterale numerico* è la rappresentazione di un numero. Esempi di letterali numerici in C sono `-13`, `0`, `3.14e2`. In C si possono usare anche letterali di tipo carattere (es., `'a'`) oppure stringa (es., `"ciao"`). Si presti attenzione che c'è differenza tra *costanti* e *letterali*: una costante è un nome a cui è associato un valore che non può essere cambiato da programma. Un letterale è una sequenza di simboli che rappresenta direttamente un valore.

Esistono anche altri meccanismi per introdurre delle costanti: ad esempio, in C sarebbe stato possibile definire costanti globali (*named constants*):

```
const int EMPTY = 0;
const int GAS = 1;
const int WALL = 2;
```

oppure simboli del preprocessore:

```
#define EMPTY 0
#define GAS 1
#define WALL 2
```

oppure costanti enumerative:

```
enum {EMPTY, GAS, WALL};
```

Ciascuna delle soluzioni precedenti ha pregi e difetti, ma risponde ugualmente bene al requisito di sostituire le costanti numeriche con nomi significativi.

Un altro caso in cui spesso si abusa di “costanti magiche” è nella gestione di caratteri ASCII. Ricordiamo che in linguaggio C le espressioni letterali di tipo `'x'` (si notino i singoli apici, da non confondere con `"x"`) hanno valore numerico che corrisponde al codice ASCII del carattere `x`.

Seguono due frammenti di codice che leggono una sequenza di caratteri e stampano solo quelli corrispondenti a cifre numeriche '0', ... '9'. La versione di destra è molto più chiara di quella di sinistra, pur essendo entrambe equivalenti (il codice ASCII di '0' è 48, mentre quello di '9' è 57).

✗ Non chiaro	✓ Intento evidente
<pre>int c; while ((c = getchar()) != EOF) { if (c >= 48 && c <= 57) { printf("%c", c); } }</pre>	<pre>int c; while ((c = getchar()) != EOF) { if (c >= '0' && c <= '9') { printf("%c", c); } }</pre>

Si noti che `c` deve avere tipo `int` e non `char`; la funzione `getchar()` infatti restituisce il codice ASCII del carattere letto da standard input, oppure il valore EOF se l'input è terminato. Normalmente EOF è definito come -1, quindi la funzione `getchar()` restituisce un valore nell'insieme {-1, 0, ... 255} che non può essere rappresentato da un `char`. Per questo motivo il risultato della funzione deve essere assegnato ad una variabile di tipo `int`.

👉 Usare espressioni letterali, simboli o “named constants” al posto di costanti magiche

Si tenga presente che i letterali numerici 0 e 1 sono talmente comuni da non necessitare di costanti simboliche che li definiscano. Inoltre, ci possono essere situazioni in cui non c'è alcun vantaggio nel sostituire letterali numerici con costanti.

1.7 Rendere espliciti gli intenti

Per rendere chiaro il significato di certe operazioni può talvolta essere utile assegnare dei nomi ai valori di certe espressioni, allo scopo di chiarirne l'intento. Come semplice esempio, consideriamo una funzione `reverse()` il cui scopo è di invertire il contenuto di un array (il primo elemento diventa l'ultimo, il secondo diventa il penultimo e così via).

✗ Intento non evidente	✓ Intento evidente
<pre>void reverse(int v[], int n) { for (i = 0; i < n/2; i++) { const int tmp = v[i]; v[i] = v[n-1-i]; v[n-1-i] = tmp; } }</pre>	<pre>void swap(int *a, int *b) { ... } void reverse(int v[], int n) { for (i = 0; i < n/2; i++) { const int opp = n-1-i; swap(&v[i], &v[opp]); } }</pre>

La versione a sinistra è corretta e lo scopo della funzione risulta abbastanza evidente dal nome; tuttavia, il corpo del ciclo potrebbe risultare non immediatamente comprensibile da chi non ha ancora maturato una sufficiente esperienza di programmazione; in particolare, potrebbe non risultare chiaro il significato dell'espressione `n-1-i`, che indica la posizione (indice) dell'elemento in posizione simmetrica rispetto a `v[i]`.

La versione a destra rende l'intento del codice più evidente con due modifiche:

- la creazione di una funzione `swap()`, il cui nome suggerisce che effettui lo scambio tra i valori puntati dagli argomenti;
- la definizione di una costante “opp” (*opposite*) per rappresentare l'indice dell'elemento opposto all'indice `i`

Il corpo del ciclo nella nuova funzione `reverse()` potrebbe ora essere letto come: “scambia `v[i]` con l'elemento nella parte opposta”.

 Rendere esplicito l'intento del codice.

1.8 Priorità degli operatori

Ogni linguaggio di programmazione stabilisce la priorità degli operatori aritmetici e logici usati nelle espressioni. Ad esempio, in linguaggio C l'espressione

```
a == 2 || c == 3
```

viene interpretata come se fosse scritta

```
(a == 2) || (c == 3)
```

perché l'operatore di confronto (`==`) ha precedenza maggiore rispetto all'“or logico” (`||`). Occorre però prestare attenzione ad alcune espressioni che potrebbero essere valutate in modo diverso da quello che ci si aspetta. Ad esempio, supponiamo di voler confrontare il valore di una variabile sta-

tus con una maschera binaria composta utilizzando l'operatore “or bit a bit”. Questo frammento di codice non è corretto

```
if (status == BIT_RD | BIT_WR | BIT_EXCL) { ... } /* ERRATO */
```

perché l'operatore “or bit a bit” (`|`) ha priorità *minore* rispetto all'operatore di confronto, per cui la condizione viene valutata come se fosse scritta:

```
if ((status == BIT_RD) | BIT_WR | BIT_EXCL) { ... }
```

Per confrontare correttamente il valore della variabile “status” con l'espressione

`BIT_RD | BIT_WR | BIT_EXCL` servono le parentesi:

```
if (status == (BIT_RD | BIT_WR | BIT_EXCL)) { ... }
```

Ogni linguaggio riserva sorprese in merito alla priorità degli operatori. Per questo conviene usare sempre le parentesi.

👉 Usare le parentesi per evitare possibili errori legati alla priorità degli operatori

1.9 Lunghezza delle funzioni

Funzione o metodi troppo lunghi sono difficili da comprendere; non è un caso che uno dei principi di base della programmazione sia la *decomposizione* di problemi complessi in problemi più semplici. Secondo il principio di decomposizione, quindi, se una funzione è troppo lunga può essere opportuno decomporla in più funzioni semplici.

Non esistono regole meccaniche per decidere come decomporre una funzione. Un indicatore di cui tener conto è la lunghezza (numero di righe di codice): indicativamente, funzioni più lunghe di 50-60 righe potrebbero necessitare di essere decomposte. Un altro indicatore è il livello di annidamento dei blocchi, inteso come il numero di blocchi inseriti uno nell'altro (es., `if` all'interno di `for` all'interno di `while` all'interno di altri `if`...). Un livello di annidamento superiore a 3-4 rende il codice difficile da comprendere, e può nascondere errori insidiosi. Questi problemi possono essere risolti spostando in funzioni separate il codice che si trova più all'interno.

👉 Decomporre funzioni troppo lunghe o complesse

Come corollario alla regola di decomposizione, la funzione `main()` del linguaggio C dovrebbe limitarsi a invocare altre funzioni per l'esecuzione del programma; tranne che nei casi più semplici, scrivere tutto il codice all'interno del `main()` è una pessima pratica di programmazione.

👉 Non scrivere l'intero programma nel `main()`

Come esempio consideriamo la funzione `floyd_warshall()` che implementa l'algoritmo di Floyd e Warshall per il calcolo dei cammini minimi da singola sorgente. I dettagli non sono importanti; è sufficiente osservare che l'algoritmo si compone di tre parti: nella prima vengono inizializzati la matrice delle distanze `d[][]` e l'array dei predecessori `p[]`; nella seconda si eseguono `n` passi di rilassamento, dove `n` è il numero di nodi del grafo; nell'ultima parte si verifica la presenza di cicli negativi, dato che in presenza di cicli negativi potrebbero non esistere cammini di lunghezza minima.

✗ Prima della decomposizione

```
int floyd_warshall(const Graph *g,
                  double **d, int **p)
{
    int u, v, k;
    const int n = graph_n_nodes(g);
    for (u = 0; u < n; u++) {
        for (v = 0; v < n; v++) {
            d[u][v] = (u==v ? 0.0 : HUGE_VAL);
            p[u][v] = -1;
        }
    }
    for (u = 0; u < n; u++) {
        const Edge *e;
        for (e = graph_adj(g, u);
             e != NULL; e = e->next) {
            d[e->src][e->dst] = e->weight;
            p[e->src][e->dst] = e->src;
        }
    }
    for (k = 0; k < n; k++) {
        for (u = 0; u < n; u++) {
            for (v = 0; v < n; v++) {
                if (d[u][k]+d[k][v]<d[u][v]) {
                    d[u][v] = d[u][k] + d[k][v];
                    p[u][v] = p[k][v];
                }
            }
        }
    }
    for (u = 0; u < n; u++) {
        if ( d[u][u] < 0.0 ) return 1;
    }
    return 0;
}
```

✓ Dopo la decomposizione

```
void init_fw(const Graph *g,
            double **d, int **p)
{
    int u, v;
    const int n = graph_n_nodes(g);
    for (u = 0; u < n; u++) {
        for (v = 0; v < n; v++) {
            d[u][v] = (u==v ? 0.0 : HUGE_VAL);
            p[u][v] = -1;
        }
    }
    for (u=0; u<n; u++) {
        const Edge *e;
        for (e = graph_adj(g, u);
             e != NULL; e = e->next) {
            d[e->src][e->dst] = e->weight;
            p[e->src][e->dst] = e->src;
        }
    }
}

void relax(double **d, int **p,
           int k, int n)
{
    int u, v;
    for (u = 0; u < n; u++) {
        for (v = 0; v < n; v++) {
            if (d[u][k]+d[k][v]<d[u][v]) {
                d[u][v] = d[u][k] + d[k][v];
                p[u][v] = p[k][v];
            }
        }
    }
}

int neg_cycle(double **d, int n)
{
    int u;
    for (u = 0; u < n; u++) {
        if ( d[u][u] < 0.0 ) return 1;
    }
    return 0;
}

int floyd_warshall(const Graph *g,
                  double **d, int **p)
{
    int k;
    const int n = graph_n_nodes(g);
    init_fw(g, d, p);
    for (k=0; k<n; k++) {
        relax(d, p, k, n);
    }
    return neg_cycle(d, n);
}
```

La versione non fattorizzata di `floyd_warshall()` (a sinistra) è in realtà già accettabile così. Non è quindi strettamente necessario decomporla in sotto-funzioni; tuttavia, se lo si fa si ottiene una versione un po' più leggibile, in quanto risultano evidenti le tre fasi dell'algoritmo.

Occorre tenere presente che l'applicazione eccessiva del principio di decomposizione potrebbe portare più problemi che vantaggi: l'abuso di chiamate di funzioni può rendere il codice difficile da leggere, in quanto ci si troverebbe a saltare da una parte all'altra del codice sorgente alla ricerca della definizione delle funzioni invocate. Ci si può rendere conto di questo leggendo la versione a destra della funzione `floyd_warshall()`.

Esistono strumenti automatici in grado di calcolare varie metriche di complessità di un programma in C, come ad il programma `pmccabe` (<https://gitlab.com/pmccabe/pmccabe>). Ad esempio, per esaminare il contenuto del file `floyd-warshall.c` si usa il comando:

```
$ pmccabe -v floyd-warshall.c
```

```
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|   |   # Statements in function
|   |   |   First line of function
|   |   |   |   # lines in function
|   |   |   |   filename(definition line number):function
|   |   |   |   |
12   12   40   149   50   floyd-warshall.c(149): floyd_warshall
3    3    6    202   15   floyd-warshall.c(202): print_path
3    3    15   220   18   floyd-warshall.c(220): print_dist
7    7    38   239   45   floyd-warshall.c(239): main
```

Le colonne di interesse sono le prime due, che indicano due varianti della *complessità ciclomatica di McCabe* (una metrica legata al numero di strutture condizionali e iterative presenti nella funzione), e la quinta che indica la lunghezza di ciascuna funzione. Le funzioni con complessità ciclomatica superiore a 10-15 devono essere esaminate con attenzione perché potrebbero beneficiare di una decomposizione, oppure di essere ristrutturare pesantemente.

2 Sostanza

Beware of bugs in the above code; I have only proved it correct, not tried it.

(Donald Knuth)

2.1 Evitare ripetizioni

Vanno evitate le ripetizioni di codice, perché rendono il codice più lungo e complesso da capire e favoriscono favorire errori insidiosi: ad esempio, se la parte ripetuta deve essere in seguito modificata, c'è il rischio di dimenticarsi di cambiare tutte le istanze. Un indizio sulla presenza di ripetizioni si ha quando si effettuano dei “copia e incolla” durante la scrittura del codice, eventualmente con piccole modifiche tra le copie.

Esistono diversi modi per eliminare codice ripetuto. In molti casi si può dichiarare la parte ripetuta in una funzione con opportuni parametri. In altri casi, potrebbe essere conveniente definire la parte ripetuta all'interno del corpo di uno o più cicli annidati.

Come esempio delle due tecniche precedenti, supponiamo di dover realizzare una funzione `foo(in, out, n)` che riempie una matrice `out[][]` di dimensioni $n \times n$ a partire da una matrice `in[][]` delle stesse dimensioni. Il valore di ciascun elemento non sul bordo `out[i][j]` è la somma dei valori nelle otto celle adiacenti a `in[i][j]` a cui viene applicata una funzione `bar()`. In altre parole, si calcola la sommatoria della funzione `bar()` applicata agli elementi dell'intorno di dimensioni 3×3 centrato in `in[i][j]`, escluso il valore centrale `in[i][j]`.

La soluzione mostrata nella colonna di sinistra realizza in modo diretto quanto sopra usando codice ripetuto. Nella versione a destra le ripetizioni sono eliminate mediante la definizione di una funzione di supporto `nbor()` e l'uso di cicli.

✖ Codice ripetuto

```
void foo(int **in, int **out, int n)
{
    int i, j;
    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            out[i][j] = bar(in[i-1][j-1]);
            out[i][j] += bar(in[i-1][j]);
            out[i][j] += bar(in[i-1][j+1]);
            out[i][j] += bar(in[i][j-1]);
            out[i][j] += bar(in[i][j+1]);
            out[i][j] += bar(in[i+1][j-1]);
            out[i][j] += bar(in[i+1][j]);
            out[i][j] += bar(in[i+1][j+1]);
        }
    }
}
```

✔ Ok

```
int nbor(int **in, int i, int j)
{
    int s, t, result = 0;
    for (s = -1; s <= 1; s++) {
        for (t = -1; t <= 1; t++) {
            if (s || t)
                result += bar(in[i+s][j+t]);
        }
    }
    return result;
}

void foo(int **in, int **out, int n)
{
    int i, j;
    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            out[i][j] = nbor(in, i, j);
        }
    }
}
```

Nella versione a sinistra sono presenti otto righe quasi identiche, in cui l'unica cosa che cambia sono gli indici nella matrice `in[][]`. Il codice non è facilmente modificabile, ad esempio nel caso in cui venisse richiesto di usare un intorno di dimensioni maggiori, oppure di applicare una funzione diversa da `bar()` (in quest'ultimo caso occorrerebbe modificare tutte le otto occorrenze di `bar()`).

Nel codice di destra applichiamo due modifiche: la prima consiste nello spostare il calcolo dei valori da assegnare a `out[i][j]` in una funzione separata `nbor()`; la seconda modifica consiste nell'uso di due cicli annidati per iterare sulle celle adiacenti a `in[i][j]`. Sebbene il programma ottenuto sia più lungo, risulta anche più comprensibile ed è più facile da modificare. La dimensione dell'intorno è determinata dagli estremi dei due cicli "for" nella funzione `nbor()`; nel caso in cui si debba applicare una funzione diversa da `bar()` ai valori `in[i][j]` prima di accumularli, è sufficiente modificare un unico punto del codice.

👁 Limitare al minimo le ripetizioni

Vediamo un ultimo esempio ancora più complesso. Supponiamo di dover scrivere una funzione per calcolare l'importo delle tasse dovute su un reddito R . Le tasse corrispondono ad una percentuale t dell'intero reddito, che dipende da R secondo la tabella seguente:

Reddito (R)	Tasse (t)
$0 \leq R \leq 10000$	0.00%
$10000 < R \leq 15000$	5.00%
$15000 < R \leq 21000$	7.00%
$21000 < R \leq 34000$	8.00%
$34000 < R \leq 49500$	8.50%
$49500 < R \leq 62300$	10.00%

Possiamo calcolare t traducendo la tabella in una catena di condizioni `if ... else`; il programma che si ottiene è piuttosto ripetitivo, e diventa rapidamente ingestibile all'aumentare del numero di scaglioni.

È possibile semplificare il programma spostando la logica dal codice ad una opportuna struttura dati, che anche in questo caso è costituita da una coppia di array paralleli $s[]$ e $t[]$: $s[i]$ rappresenta il limite inferiore dello scaglione i , e $t[i]$ il corrispondente tasso di interesse. Lo scaglione corretto (e quindi la percentuale di imposte) può essere determinata effettuando una ricerca lineare a ritroso nell'array $s[]$.

✗ Codice ripetitivo

```
double t, tasse;
if (R <= 10000) {
    t = 0.0;
} else if (R <= 15000) {
    t = 5.0;
} else if (R <= 21000) {
    t = 7.0;
} else if (R <= 34000) {
    t = 8.0;
} else if (R <= 49500) {
    t = 8.5;
} else if (R <= 62300) {
    t = 10.00;
} else {
    t = 12.75;
}
tasse = R * t / 100;
```

✓ Meno comprensibile, ma più compatto

```
/* s[i] è il limite inferiore dello scaglione i */
const double s[] =
    {0, 10000, 15000, 21000, 34000,
    49500, 62300};
const double t[] =
    {0.0, 5.0, 7.0, 8.0, 8.5, 10.0,
    12.75};
const int N = sizeof(s)/sizeof(s[0]);
double t, tasse;
int i = N-1;
while ((i > 0) && (s[i] <= R))
    i--;
tasse = R * t[i] / 100;
```

Il codice che si ottiene in questo modo non è necessariamente “migliore”. Come si può osservare, la versione di sinistra, che fa uso di una serie di condizioni, è sicuramente più comprensibile. Tuttavia, diventerebbe ingestibile nel caso in cui il numero di scaglioni fosse elevato. La versione di destra, basata sui array paralleli, è più compatta ma necessita di commenti per renderne chiaro l'intento.

👁 Spostare la logica dal codice ai dati quando è opportuno farlo

Vanno assolutamente evitate soluzioni sciatte che dimostrano solo di non aver compreso il problema (né la soluzione). Consideriamo ad esempio la seguente funzione:

```
enum {NORTH, SOUTH, EAST, WEST};

void f(float **d, float **w, int **p, int r, int c) {
    int i, j;
    for (i=1; i<r-1; i++) {
        for (j=1; j<c-1; j++) {
            if (d[i][j] > d[i-1][j] + w[i][NORTH]) {
                d[i][j] = d[i-1][j] + w[i][NORTH];
                p[i][j] = NORTH;
            }
            if (d[i][j] > d[i+1][j] + w[i][SOUTH]) {
                d[i][j] = d[i+1][j] + w[i][SOUTH];
            }
        }
    }
}
```

```

        p[i][j] = SOUTH;
    }
    if (d[i][j] > d[i][j-1] + w[i][EAST]) {
        d[i][j] = d[i][j-1] + w[i][EAST];
        p[i][j] = EAST;
    }
    if (d[i][j] > d[i][j+1] + w[i][WEST]) {
        d[i][j] = d[i][j+1] + w[i][WEST];
        p[i][j] = WEST;
    }
}
}
}

```

Si vede subito che i quattro **if** sono sostanzialmente identici. La seguente soluzione denota che non si è compreso il problema, e va evitata:

/ NO! Soluzione da evitare */*

```

enum {NORTH, SOUTH, EAST, WEST};

void relax_NORTH(float **d, const float **w, int **p, int i, int j) {
    if (d[i][j] > d[i-1][j] + w[i][NORTH]) {
        d[i][j] = d[i-1][j] + w[i][NORTH];
        p[i][j] = NORTH;
    }
}
void relax_SOUTH( ... ) { idem }
void relax_EAST( ... ) { idem }
void relax_WEST( ... ) { idem }

void f(float **d, const float **w, int **p, int r, int c) {
    int i, j;
    for (i=1; i<r-1; i++) {
        for (j=1; j<c-1; j++) {
            relax_NORTH(d, w, p, i, j);
            relax_SOUTH(d, w, p, i, j);
            relax_EAST(d, w, p, i, j);
            relax_WEST(d, w, p, i, j);
        }
    }
}

```

Si noti infatti che il numero di righe di codice non diminuisce, dato che si sono semplicemente spostati quattro blocchi quasi identici in altrettante funzioni quasi identiche. Questo non è accettabile.

Una soluzione accettabile consiste nel definire una *singola* funzione **relax()**, in cui la direzione (NORTH, SOUTH, eccetera) è un ulteriore parametro:

```

enum {NORTH, SOUTH, EAST, WEST};

void relax(float **d, const float **w, int **p, int i, int j, int dir) {
    static const int di[] = {-1, 1, 0, 0}; /* NORTH, SOUTH, EAST, WEST */
    static const int dj[] = {0, 0, 1, -1};
    const int i1 = i + di[dir];
    const int j1 = j + dj[dir];
    if (d[i][j] > d[i1][j1] + w[i][dir]) {
        d[i][j] = d[i1][j1] + w[i][dir];
        p[i][j] = dir;
    }
}

```

```

void f(float **d, const float **w, int **p, int r, int c) {
    int i, j;
    for (i=1; i<r-1; i++) {
        for (j=1; j<c-1; j++) {
            relax(d, w, p, i, j, NORTH);
            relax(d, w, p, i, j, SOUTH);
            relax(d, w, p, i, j, EAST);
            relax(d, w, p, i, j, WEST);
        }
    }
}

```

Si noti l'uso di due array statici *di[]* e *dj[]*, che vengono utilizzati per ottenere le coordinate (*i1*, *j1*) della cella che si trova, rispettivamente, a nord, sud, est e ovest rispetto a (*i*, *j*). Ad esempio, l'espressione *di[NORTH]* ha valore -1, perché per ottenere la prima coordinata della cella che si trova a nord di (*i*, *j*) bisogna sottrarre uno al valore *i*; l'espressione *dj[NORTH]* vale invece zero, perché per ottenere la seconda coordinata della cella che si trova a nord di (*i*, *j*) non bisogna modificare il valore *j*.

2.2 Semplificare le strutture condizionali

Le strutture condizionali (if-then-else) vanno scritte in una delle due forme seguenti:

<pre> if (Cond) Strue; </pre>	<pre> if (Cond) Strue; else Sfalse; </pre>
-----------------------------------	--

dove *Cond* è una espressione, e *Strue*, *Sfalse* sono blocchi *non vuoti* di codice (eventualmente racchiusi tra parentesi graffe se composti da più di una istruzione). Si presti attenzione al fatto che devono essere non vuoti: ciò implica che i blocchi condizionali non dovrebbero mai avere il ramo “vero” vuoto.

<p>✗ No: ramo “vero” vuoto</p> <pre> if (a < b) { /* vuoto */ } else { bar(); } </pre>	<p>✓ Ok</p> <pre> if (a >= b) { bar(); } </pre>
---	--

Se si usino strutture condizionali annidate (cioè posizionate una dentro l'altra), è importante assicurarsi che le condizioni non siano ridondanti. Nell'esempio seguente, la condizione *if (a >= b)* è superflua, in quanto nel ramo “falso” della condizione *if (a < b)* sappiamo già che vale *a >= b*.

✗ Condizione superflua <pre> if (a < b) { foo(); } else { if (a >= b) { /* superfluo */ bar(); } } </pre>	✓ Ok <pre> if (a < b) { foo(); } else { bar(); } </pre>
---	--

Nell'esempio seguente mostriamo un caso in cui non è presente ridondanza in senso stretto, cioè non c'è codice inutile, ma è possibile semplificare il programma rendendolo più compatto e chiaro. I quattro `if` annidati possono essere espressi con una singola condizione ottenuta mediante l'“and” logico di quelle di partenza.

✗ No <pre> if (A) { if (B) { if (C) { if (D) { blah(); } } } } </pre>	✓ Sì <pre> if (A && B && C && D) { blah(); } </pre>
---	---

La semplificazione precedente non è sempre possibile, per cui occorre prestare la massima attenzione. Ad esempio, i due frammenti di codice seguenti non sono equivalenti:

Versione 1 <pre> if (A) { if (B) { foo(); } } else { bar(); } </pre>	Versione 2 (non equivalente) <pre> if (A && B) { foo(); } else { bar(); } </pre>
--	--

Per rendercene conto, esaminiamo tutte le possibili combinazioni dei valori di A e B, e per ciascuna di esse determiniamo quale funzione viene invocata (se ne viene invocata una)


A	B	Versione 1	Versione 2
vero	vero	foo()	foo()
vero	falso	–	bar()
falso	vero	bar()	bar()
falso	falso	bar()	bar()

Il problema è che se A è vera e B è falsa, la versione 1 non esegue alcuna istruzione, mentre la versione 2 invoca la funzione `bar()`.

Un altro uso di una struttura condizionata che può essere semplificato si ha quando si vuole scrivere una funzione che restituisce “vero” o “falso” in base ad una determinata condizione. Ad esempio, supponiamo di voler scrivere una funzione `is_even_positive(n)` che restituisce `!true` (cioè un valore diverso da zero) se *n* è pari e positivo, zero altrimenti.

✗ No	✓ Sì
<pre>int is_even_positive(int n) { if ((n % 2) == 0) && (n > 0)) { return 1; } else { return 0; } }</pre>	<pre>int is_even_positive(int n) { return ((n % 2) == 0) && (n > 0); }</pre>

In casi come questo è utile ricordare che in linguaggio C una espressione logica ha valore 0 se falsa, 1 se vera. Di conseguenza, nel caso in cui la condizione sia sufficientemente semplice, è possibile usarla direttamente come valore di ritorno senza fare ricorso ad una struttura condizionale.

 Semplificare le strutture condizionali quando possibile

2.3 for o while?

Le strutture `for` e `while` nei linguaggi C/C++ e Java sono interscambiabili, nel senso che un ciclo espresso mediante una delle due si può riscrivere in modo equivalente con l'altra. Esempio:

Usando “for”	Usando “while”
<pre>int i; for (i = 0; i < 100; i = i + 5) { blah(i); }</pre>	<pre>int i = 0; while (i < 100) { blah(i); i = i + 5; }</pre>

Nonostante l'equivalenza, esistono delle convenzioni a cui è bene attenersi. Il tipo di costrutto iterativo da usare dipende dal tipo di ciclo:

- Un *ciclo enumerativo* è governato da un contatore che spesso (ma non sempre) viene incrementato/decrementato di una quantità prefissata; il numero di iterazioni risulta quindi prevedibile. Per questo genere di cicli si usa preferibilmente `for`.
- In un *ciclo a valutazione ripetuta* il numero di iterazioni dipende da una condizione complessa che viene valutata ogni volta; di conseguenza, il numero di iterazioni non è prevedibile. Per questo genere di cicli si usa preferibilmente `while`.

- Un *ciclo iteratore* ha come scopo quello di visitare tutti gli elementi di una struttura dati (es., una lista). Per questo genere di cicli si usa preferibilmente **for**; alcuni linguaggi come C++ e Java offrono una sintassi specifica.
- Un *ciclo infinito*, che non termina mai. Questo genere di cicli si trova principalmente nella programmazione di microcontrollori e sistemi embedded in cui il dispositivo deve eseguire continuamente la stessa sequenza di istruzioni. Per questo genere di cicli si usa preferibilmente **while**; in C/C++ di solito si scrive **while(1) { ... }**, mentre in Java si scrive **while(true) { ... }**. Tuttavia, è piuttosto diffusa la rappresentazione di un ciclo infinito usando l'idioma **for(;;) { ... }**

Alcuni esempi:

Cicli enumerativi

```
for (i=0; i<100; i+=4) {
    blah(i);
}
```

```
for (i=1; i<100; i=i*2) {
    blah(i);
}
```

Cicli a valutazione ripetuta

```
/* Esempio C */
File *f =
fopen("test.in", "r");
while (!feof(f)) {
    process_item(f);
}
```

```
// Esempio Java
File f =
    new File("test.in");
Scanner s =
    new Scanner(f);
while (s.hasNextInt()) {
    int val = s.nextInt();
    process(val);
}
```

Cicli iteratori

```
/* Esempio C */
typedef struct List {
    int val;
    struct List *next;
} List;
List *L = ..., *node;
for (node = L;
    node != NULL;
    node = node->next) {
    blah(node);
}
```

```
// Esempio Java
LinkedList<Integer> L = ...;
for (Integer item : L) {
    process(item);
}
```

Ricordando che le strutture generali dei cicli **for** e **while** sono:

for (<i>iniz; test; aggiornamento</i>) { <i>corpo</i> }	while (<i>test</i>) { <i>corpo</i> }
--	---

possiamo definire alcune regole di tipo generale per decidere quando usare i due tipi di cicli:

Usare **for** se:

- Sono presenti tutte e tre le componenti (inizializzazione, test, aggiornamento), e
- Inizializzazione, test e aggiornamento fanno riferimento alla stessa variabile, e
- La variabile che viene modificata nel blocco “aggiornamento” non viene modificata nel corpo del ciclo.

Usare **while** se:

- Il ciclo è del tipo “a valutazione ripetuta”, oppure
- La variabile utilizzata nel test vengono aggiornate in modo non uniforme all’interno del corpo del ciclo.

Il ciclo **for** si usa preferibilmente specificando tutte e tre le componenti (inizializzazione, test, aggiornamento), come nel codice a destra:

✗ No	✓ Sì
<pre>i = 0; for (; i < 10; i++) { blah(i); }</pre>	<pre>for (i = 0; i < 10; i++) { blah(i); }</pre>
<pre>i = 0; for (; i < 10 ;) { blah(i); i++; }</pre>	
<pre>i = 0; for (; ;) { if (i >= 10) break; blah(i); i++; }</pre>	

In generale, un ciclo che itera su tutti gli interi compresi tra 0 e $n - 1$ viene scritto in C o Java nella seguente *forma idiomatica*:

```
for (i = 0; i < n; i++) {
    /* ... */
}
```

In altre parole, i programmatori esperti colgono "a colpo d'occhio" il significato del ciclo guardandone la struttura. Ogni linguaggio di programmazione ha le proprie forme idiomatiche, cioè degli schemi che vengono sistematicamente applicati in certe situazioni come il ciclo precedente. Non è ovviamente obbligatorio attenersi alle forme idiomatiche, ma è importante avere un buon motivo per farlo perché discostarsi dagli idiomi standard rende più difficile la lettura e la comprensione del codice.

👉 Attenersi alle forme idiomatiche del linguaggio di programmazione utilizzato

Come esempi di uso di `for` e `while` descriviamo due funzioni che operano su un array `v[]` di lunghezza `n`. La funzione `esiste_dup()` restituisce `true` se e solo se esiste un valore in `v[]` che compare almeno due volte. La funzione `binsearch()` cerca la posizione di una occorrenza di un valore dato usando l'algoritmo di ricerca binaria.

✔ Preferibile “for”

/ Dato un array v[] di lunghezza n, restituisce “vero” se esiste un valore in v[] che compare almeno due volte, “falso” altrimenti. */*

```
int esiste_dup(int v[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++) {
        for (j = i+1; j < n; j++) {
            if (v[i] == v[j])
                return 1;
        }
    }
    return 0;
}
```

✔ Preferibile “while”

/ Cerca una occorrenza di key nell'array v[] di lunghezza n; l'array deve essere ordinato in senso non decrescente. Restituisce l'indice dell'occorrenza trovata, oppure -1 se la chiave non è presente. */*

```
int binsearch(int v[], int n, int key)
{
    int i = 0; j = n-1;
    while (i <= j) {
        const int m = (i+j) / 2;
        if (v[m] < key) {
            i = m+1;
        } else if (v[m] > key) {
            j = m-1;
        } else {
            return m;
        }
    }
    return -1; /* chiave non presente */
}
```

Nel caso della ricerca di duplicati, la soluzione proposta può essere codificata usando due cicli `for`, in quanto i tre requisiti indicati sopra sono soddisfatti. Nel caso della ricerca binaria, le variabili che compaiono nel test del ciclo vengono aggiornate in modo non uniforme (in certi casi si aggiorna `i`, in altri si aggiorna `j`); ciò suggerisce di usare un ciclo `while`.

👁 Scegliere tra `for` o `while` in modo ragionato

Come detto sopra, è consuetudine che all'interno del corpo di un ciclo `for` non si modifichi la variabile che viene aggiornata nell'intestazione del ciclo. Violare questa consuetudine confonde chi legge e può introdurre errori insidiosi.

Si consideri il problema seguente: è data una stringa `s[]` rappresentata in C mediante un array di caratteri terminato da `'\0'`. Vogliamo esaminare la stringa da sinistra verso destra, stampando il valore intero corrispondente alle sottostringhe massimali di cifre. Ad esempio, se `s = “ab132ccd3ef21”`, vogliamo isolare stampare i valori interi 132, 3 e 21.

Consideriamo le due soluzioni seguenti; quella di sinistra è errata.

✖ Errato

```
for (i = 0; s[i]; i++) {
    if (isdigit(s[i])) {
        int v = 0;
        while (isdigit(s[i])) {
            v = v*10 + (s[i] - '0');
            i++;
        }
        printf("%d\n", v);
    }
}
```

✔ Corretto

```
i = 0;
while (s[i]) {
    if (isdigit(s[i])) {
        int s = 0;
        while (isdigit(s[i])) {
            s = s*10 + (s[i] - '0');
            i++;
        }
        printf("%d\n", s);
    }
}
```

Nella soluzione di sinistra si modifica il valore della variabile “i” sia nell'intestazione del costrutto `for`, sia nel corpo del ciclo. Questo introduce un errore piuttosto insidioso: al termine del ciclo `while` interno, `s[i]` è il carattere immediatamente successivo all'ultima cifra letta. Tuttavia, la variabile `i` viene nuovamente incrementata al termine di ogni iterazione del `for` esterno: questo causa un accesso ad un elemento successivo all'ultimo se la stringa termina con una cifra, ad es., `s = “123a34”`.

Usando una struttura `while` (a destra) l'errore non si presenta; inoltre, la presenza di un `while` suggerisce la possibilità che all'interno del suo corpo il valore di `i` possa venire modificato, come effettivamente avviene.

👉 Non modificare la variabile indice di un ciclo “for” nel corpo del ciclo

Una importante applicazione della regola precedente riguarda l'iterazione su una struttura dati bidimensionale come una matrice. L'idioma standard consiste nell'utilizzare due cicli annidati. Tuttavia, capita di vedere soluzioni ardite quanto inappropriate che realizzano questa operazione con un unico ciclo, come quella a sinistra.

✖ No

/ a[][] è una matrice con R righe e C colonne */*

```
for (i = 0, j = 0; i < R; j++) {
    if (j == C) {
        j = 0;
        i++;
    }
    if (i == R)
        break;
    do_something(a[i][j]);
}
```

✔ Sì

/ a[][] è una matrice con R righe e C colonne */*


```
for (i = 0; i < R; i++) {
    for (j = 0; j < C; j++) {
        do_something(a[i][j]);
    }
}
```

La versione a sinistra presenta molte criticità che dovrebbero renderla quantomeno sospetta:

- Il test del ciclo si svolge su una variabile diversa da quella che viene incrementata nell'intestazione;
- Entrambe le variabili del ciclo vengono modificate all'interno del corpo;

- Si deve usare una seconda condizione per evitare che il corpo del ciclo prenda in considerazione la riga R (che non esiste);
- Si fa uso di una istruzione `break` (vedi Sezione 2.4);
- L'intento del codice non è chiaro (vedi Sezione 1.7).

La versione di sinistra è quindi opaca e fragile, perché richiede controlli ulteriori non banali per funzionare, e va quindi evitata.

 Il modo standard per iterare su matrici è l'utilizzo di cicli annidati.

2.4 `break` e `continue`

I costrutti `break` e `continue` vanno usati con parsimonia: dovrebbero essere l'eccezione più che la regola. L'istruzione `break` introduce una violazione del principio della programmazione strutturata; la programmazione strutturata richiede che ogni blocco abbia un singolo punto di ingresso e un singolo punto di uscita. Il modo consigliato per uscire dai cicli è rendendo falsa la condizione, anziché usare `break`.

Ad esempio, supponiamo di volere cercare la posizione (indice) del primo valore negativo presente in un array $v[]$ di interi di lunghezza n . La soluzione che usa il costrutto `break` (a sinistra) può essere riscritta sfruttando una condizione leggermente più complessa del ciclo `for` (a destra).

✗ No	✓ Sì
<pre> for (i = 0; i < n; i++) { if (v[i] < 0) break; } if (i >= n) { Nessun valore negativo } else { v[i] è il primo valore negativo } </pre>	<pre> for (i=0; (i<n) && (v[i] >= 0); i++) { /* corpo vuoto */ } if (i >= n) { Nessun valore negativo } else { v[i] è il primo valore negativo } </pre>

Lo schema precedente è abbastanza frequente, per cui vale la pena fornire una regola generale che si possa applicare in molti casi (non tutti!). Supponiamo di voler iterare un ciclo enumerativo fino a quando una certa condizione *cond* diventa vera. L'uso di `break` può essere evitato sfruttando un ciclo `for` o `while` in cui il corpo viene eseguito finché la condizione *cond* rimane falsa. Non appena *cond* diventa vera, il ciclo termina:

<p>✗ No</p> <pre>for (init; test; incr) { if (cond) break; else foo(); }</pre>	<p>✓ Sì</p> <pre>for (init; test && !cond; incr) { foo(); }</pre> <hr/> <p><i>/* soluzione alternativa */</i></p> <pre>init; while (test && !cond) { foo(); incr; }</pre>
--	---

Come ultimo esempio di abuso del costrutto `continue`, supponiamo di voler scrivere il corpo di una funzione che, data una matrice `m[][]` di caratteri di dimensioni 3×3 , restituisce 0 se tutti i caratteri nella matrice sono uguali a 'A', 0 altrimenti.

<p>✗ No</p> <pre>int check(char m[3][3]) { int i, j; for (i = 0; i < 2; i++) { for (j = 0; j < 2; j++) { if (m[i][j] == 'A') continue; else return 1; } } return 0; }</pre>	<p>✓ Sì</p> <pre>int check(char m[3][3]) { int i, j; for (i = 0; i < 2; i++) { for (j = 0; j < 2; j++) { if (m[i][j] != 'A') return 1; } } return 0; }</pre>
---	--

La versione senza `continue` risulta più compatta e ugualmente leggibile.

2.5 Array vs Tuple

Un *array* è un tipo di dato in grado di mantenere una sequenza di valori dello stesso tipo; ogni elemento ha lo stesso nome (che coincide con l'identificatore dell'array) ed è univocamente identificato dalla posizione che occupa all'interno della sequenza. Solitamente è possibile accedere ad un elemento dell'array in un tempo che non dipende dalla sua posizione. Esempi di array in C:

```
int giorni_mese[12];
float temp_giornaliero[365];
double distanze[10][10];
```

Una *tupla* (detta anche *record*) è un tipo di dato composto da un insieme finito di valori di tipo arbitrario, anche diverso l'uno dall'altro. Ogni elemento di una tupla può essere acceduto, in base al linguaggio di programmazione, usando un nome oppure la posizione che occupa nella tupla. In linguaggio C, le tuple sono definite tramite la parola chiave `struct`, mentre in Java si usano le classi:

✓ Il tipo “persona” in C

```
struct persona {  
    char *nome;  
    char *cognome;  
    int anno_nascita;  
    float peso;  
};
```

✓ Il tipo “persona” in Java

```
class Persona {  
    String nome;  
    String cognome;  
    int anno_nascita;  
    float peso;  
};
```

Supponiamo di voler rappresentare dei punti materiali nello spazio, ciascuno caratterizzato da posizione, massa e modulo della velocità. La soluzione più appropriata è definire una struttura (in C) o una classe (in Java) contenenti attributi x, y, z, m, v di tipo *float* o *double*. Dato che tutti gli attributi hanno lo stesso tipo, la pigrizia può suggerire di “impacchettarli” in un array di *float* o *double* di lunghezza 5:

✗ Particelle rappresentate da array

```
double p1[5], p2[5];
```

✓ Particelle rappresentate da strutture

```
struct particle {  
    double x, y, z; /* posizione */  
    double m;      /* massa */  
    double v;      /* velocità */  
} p1, p2;
```

La rappresentazione basata su array (a sinistra) è inappropriata perché opaca: rende cioè difficile capire il codice che opera sulle particelle. Ad esempio, supponiamo di voler calcolare l'energia cinetica di una particella, definita dall'espressione $\frac{1}{2}mv^2$; in base alla rappresentazione della particella, si ottengono le due implementazioni seguenti:

✗ Cosa calcola?

```
double kinetic_en( double p[] )  
{  
    return 0.5 * p[3] * (p[4] * p[4]);  
}
```

✓ Calcola l'energia cinetica

```
double kinetic_en( struct particle p )  
{  
    return 0.5 * p.m * (p.v * p.v);  
}
```

La funzione a sinistra è poco comprensibile, perché non è evidente cosa rappresentino $p[3]$ e $p[4]$. Nella versione a destra risulta evidente che si stanno usando gli attributi “m” e “v” della struttura “p”, i cui nomi suggeriscono si tratti di massa e velocità. Se i nomi degli attributi sono scelti con criterio (vedi 1.4), le tuple rendono il codice più leggibile.

Come regola generale un array dovrebbe contenere elementi *logicamente* omogenei (es., tutte temperature, oppure tutti importi di denaro, oppure tutti nomi propri di persone, ecc.). Se i valori non sono logicamente omogenei, si usano tuple.

☞ Usare un array per raggruppare informazioni logicamente omogenee; usare strutture (C) o classi (Java) per informazioni non omogenee

2.6 Variabili globali

Una *variabile globale* è visibile (e quindi modificabile) in qualunque punto del codice. L'uso di variabili globali dovrebbe essere limitato a informazioni che vengono effettivamente usate ovunque nel programma. Occorre però tenere presente che modificare una variabile globale introduce delle dipendenze inaspettate tra le funzioni. Consideriamo il seguente frammento di codice C:

```
/* ... */
int f(int x) { ... }
int g(int x) { ... }
/* ... */
int main(void) {
    int x = 10;
    int a = f(x);
    int b = g(x);
    int c = f(x);
    return 0;
}
```

Non sappiamo cosa facciano le funzioni `f()` e `g()`, ma è ragionevole aspettarsi che le variabili `a` e `c` abbiano lo stesso valore, dato che entrambe vengono inizializzate con il valore `f(x)` e il parametro `x` non cambia. Questo non è vero se introduciamo una variabile globale `k` e facciamo dipendere il risultato di `f()` e `g()` da `k`:

```
int k = 0; /* variabile globale */
int f(int x) { return x+k; }
int g(int x) { k++; return x+k; }
int main(void) {
    int x = 10;
    int a = f(x); // a vale 10
    int b = g(x); // b vale 11
    int c = f(x); // c vale 11
    return 0;
}
```

 Limitare l'uso di variabili globali

2.7 Ricorsione

La modifica di variabili globali in funzioni ricorsive è una pessima pratica di programmazione e va evitata. Supponiamo di volere scrivere una funzione ricorsiva `sum(v, n)` che dato un array di interi `v[]` di lunghezza `n`, restituisce la somma dei valori di `v[]` (l'array vuoto ha somma zero). La soluzione ricorsiva si basa sul seguente ragionamento:

- la somma dei valori di un array vuoto vale zero;
- la somma dei valori di un array `v[0..n - 1]` non vuoto è uguale alla somma del sottovettore `v[0..n - 2]` di lunghezza `n - 1`, a cui si somma `v[n - 1]`:

✗ Con variabile globale

```
int s;

/* Funziona solo se prima di chiamare questa
funzione si azzerava la variabile globale s */
int sum1(int v[], int n)
{
    if (n == 0) {
        return 0;
    } else {
        s += sum1(v, n-1) + v[n-1];
        return s;
    }
}
```

✓ Senza variabile globale

```
int sum2(int v[], int n)
{
    if (n == 0)
        return 0;
    else
        return sum2(v, n-1) + v[n-1];
}
```

Entrambe le soluzioni sfruttano la stessa idea; `sum1()` fa però uso di una variabile globale `s` per accumulare i valori della sommatoria. Questa soluzione è meno leggibile di quella che non fa uso di variabili globali, ed è estremamente fragile perché obbliga il chiamante ad azzerare `s` prima di invocare la funzione.

L'uso di variabili statiche del linguaggio C è di fatto equivalente all'uso di variabili globali. In altre parole, anche la soluzione seguente va evitata:

```
int sum3(int v[], int n)
{
    static int s = 0; /* NO!! equivale a una variabile globale, con l'aggravante che non può essere
"resettata" in modo semplice. */

    if (n == 0) {
        return 0;
    } else {
        s += sum3(v, n-1) + v[n-1];
        return s;
    }
}
```

La funzione `sum3()` è peggiore di `sum1()` perché è garantita restituire il valore corretto solo la prima volta che viene invocata (restituisce il valore corretto anche quando, al termine di una catena di chiamate ricorsive, il valore di `s` diventa zero). A differenza di `sum1()`, non è possibile "resettare" una variabile statica dall'esterno di una funzione, quindi soluzioni simili a `sum3()` vanno evitate.

👉 Non modificare variabili globali o variabili statiche nelle funzioni ricorsive

2.8 Asserzioni

Una asserzione è una proprietà che deve essere vera in un certo punto del codice. Molti linguaggi di programmazione mettono a disposizione dei costrutti per verificare la validità delle asserzioni durante l'esecuzione del programma; nel caso in cui una asserzione non sia vera, il programma viene interrotto oppure viene sollevata una eccezione. Le asserzioni sono utili per intercettare gli errori

prima che essi si propaghino e causino una interruzione del programma magari in punti del codice lontani da dove si è manifestato il problema.

Un esempio di uso delle asserzioni consiste nella verifica delle precondizioni delle funzioni o dei metodi. Una precondizione è una proprietà che deve essere vera affinché una funzione (o un metodo) calcoli il risultato corretto. Ad esempio, consideriamo una funzione C `set_time()` che accetta tre parametri interi rappresentanti l'ora (0–23), i minuti (0–59) e i secondi (0–59) a cui impostare l'orologio di sistema. Possiamo assicurarci che i parametri siano corretti usando la macro `assert()` definita nell'header file `<assert.h>`

```
#include <assert.h>

void set_time(int hh, int mm, int ss)
{
    assert((hh >= 0) && (hh < 24));
    assert((mm >= 0) && (mm < 60));
    assert((ss >= 0) && (ss < 60));
    /* corpo della funzione */
}
```

La macro `assert()` del linguaggio C accetta come unico parametro un valore intero; se il valore è zero (falso) viene stampato a video la riga e il nome del file sorgente in cui si trova la chiamata e il programma termina. Se il valore è diverso da zero (vero) l'esecuzione procede. Il linguaggio Java mette a disposizione una istruzione `assert` che ha lo stesso comportamento, con la differenza che in Java viene sollevata una eccezione di tipo `java.lang.AssertionError` in caso di asserzione falsa; il programmatore potrebbe quindi intercettare l'eccezione e agire di conseguenza senza necessariamente interrompere l'esecuzione del codice. Inoltre, in Java è necessario eseguire il programma passando il flag `-ea` all'interprete per abilitare il controllo delle asserzioni:

```
java -ea MioProgramma
```

Nell'esempio precedente sarebbe stato possibile accorpare le asserzioni in un'unica espressione:

```
assert((hh >= 0) && (hh < 24) &&
      (mm >= 0) && (mm < 60) &&
      (ss >= 0) && (ss < 60));
```

In questo modo, però, non è possibile capire quale variabile sia responsabile di una eventuale violazione dell'asserzione. Mantenendo le asserzioni separate, invece, tale informazione viene fornita dal programma in esecuzione.

 Usare asserzioni per intercettare condizioni anomale

2.9 Programmazione difensiva

Il termine “programmazione difensiva” fa riferimento alle tecniche con cui è possibile sviluppare programmi che reagiscono in modo controllato a situazioni impreviste. “Reagire in modo controllato” non significa necessariamente fornire il risultato corretto anche in presenza di anomalie, quanto piuttosto evitare che le anomalie si propaghino ad altre parti del programma rendendolo instabile.

Consideriamo i due frammenti di codice seguenti:

✗ **La terminazione dipende dai valori di x ed n**

```
int i, x = ...;
for (i = x; i != n-1; i = i+2) {
    blah(i);
}
```

✓ **Termina sempre**

```
int i, x = ...;
for (i = x; i < n; i = i+2) {
    blah(i);
}
```

La variabile i viene incrementata di due ad ogni iterazione. Se x ed n hanno la stessa parità (entrambi pari o entrambi dispari) il ciclo di sinistra non termina, come si può facilmente verificare con $x = 2, n = 4$. Il ciclo di sinistra non termina anche se x è positivo e n negativo, a prescindere dalla loro parità.

L'applicazione del principio di programmazione difensiva porta a modificare la condizione $i \neq n - 1$ in $i < n$. Facendo in questo modo il ciclo termina qualunque sia il valore iniziale x ; naturalmente non è detto che il programma fornisca il risultato corretto, ma si evita il rischio di un ciclo che potrebbe essere infinito oppure no in base a come il programma e/o l'hardware gestiscono l'overflow della variabile indice i .

Come ulteriore esempio si considerino le seguenti implementazioni di una funzione ricorsiva `fib(n)` che calcola l' n -esimo numero della successione di Fibonacci.

✗ **Comportamento indefinito se $n < 0$**

```
int fib(int n)
{
    if ((n == 0) || (n == 1)) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

✓ **`fib(n)` termina sempre**

```
int fib(int n)
{
    if (n <= 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

La versione di sinistra ha un comportamento indefinito se $n < 0$, perché le chiamate ricorsive si allontanano sempre di più dal caso base anziché avvicinarsi. La terminazione della funzione dipende da come il programma e/o l'hardware gestiscono l'*underflow* della variabile n ., cioè cosa succede quando le espressioni $n - 1$ oppure $n - 2$ hanno un valore negativo troppo piccolo per essere rappresentato in una variabile di tipo `int`. La versione a destra rimpiazza la condizione $((n == 0) || (n == 1))$ con $n <= 1$: in questo modo la funzione termina sempre, restituendo 1 quando $n < 0$. Nel caso in cui il caso $n < 0$ rappresenti un errore da segnalare, si può aggiungere l'asserzione `assert(n >= 0)` all'inizio della funzione `fib()`. In questo modo la condizione anomala viene intercettata subito e porta ad un risultato prevedibile (l'interruzione del programma).

🔒 Programmare in modo difensivo

2.10 Evitare soluzioni "astute"

Talvolta capita di farsi prendere la mano esibendo il proprio virtuosismo per proporre “soluzioni astute” a problemi semplici. Col termine “astute” si intende soluzioni contorte e difficili da comprendere a problemi banali che ammettono una soluzione ovvia. Quasi sempre, le soluzioni astute non si rivelano né eleganti né compatte, e spesso nemmeno del tutto corrette. **Proporre soluzioni contorte per problemi semplici non è un pregio, ma un grave difetto.** Un bravo programmatore o programmatrice sviluppa soluzioni comprensibili, facilmente modificabili, ed efficienti. Un programmatore scarso tende a nascondere la propria mediocrità in esibizioni di virtuosismo completamente fuori luogo. Vediamo alcuni esempi.

Consideriamo tre variabili a , b , c di tipo `char`. Supponiamo che le variabili possano assumere esclusivamente i valori `'X'` oppure `'Y'`. Si chiede di scrivere un frammento di codice per verificare se tutte e tre le variabili hanno valore `'X'`.

La soluzione ovvia è l'espressione `(a == 'X') && (b == 'X') && (c == 'X')`. Tuttavia, è anche possibile immaginare una soluzione “astuta” in cui si effettua un solo confronto anziché tre.

✗ Soluzione "astuta"	✓ Soluzione leggibile
<pre>if (a + b + c == 3 * 'X') { printf("tutti uguali a X\n"); } else { printf("non tutti uguali a X\n"); }</pre>	<pre>if ((a == 'X') && (b == 'X') && (c == 'X')) { printf("tutti uguali a X\n"); } else { printf("non tutti uguali a X\n"); }</pre>

La soluzione “astuta” sfrutta il fatto che, *nell'ipotesi in cui i valori possano essere solo 'X' oppure 'Y'*, le variabili hanno lo stesso valore se e solo se la somma dei codici ASCII dei caratteri in esse contenuti è il triplo del codice ASCII del carattere `'X'`. Sebbene ciò sia corretto in questo caso, si può osservare come la soluzione “astuta” non sia né significativamente più compatta né più leggibile di quella normale. A differenza di quella normale, però, non è immediatamente comprensibile, e costringe chi legge il codice a lunghi ragionamenti per intuire cosa si sta facendo. Inoltre, la soluzione “astuta” non è generalizzabile al caso in cui le variabili possano contenere caratteri arbitrari. Ad esempio, se $a = 'w'$, $b = 'X'$, $c = 'Y'$, dato che i codici ASCII dei caratteri sono consecutivi, l'espressione `a + b + c == 3 * 'X'` risulta vera ma le tre variabili non contengono tutte `'X'`. La soluzione “astuta” è quindi opaca (difficile da comprendere) e fragile (non generalizzabile al di fuori di questo caso particolare).

Come altro esempio consideriamo il problema di scambiare tra loro il contenuto di due variabili x e y di tipo `int`. La versione “astuta” non fa uso di alcuna variabile di appoggio:

✗ Soluzione "astuta"	✓ Soluzione leggibile
<pre>x = x - y; y = x + y; x = y - x;</pre>	<pre>const int tmp = x; x = y; y = tmp;</pre>

La soluzione “astuta” può sembrare corretta (è anche possibile dimostrarlo formalmente, come faccio a lezione parlando di asserzioni e invarianti). Purtroppo l'intuizione e la dimostrazione formale trascurano un aspetto importante: il tipo `int` del linguaggio C consente di rappresentare un intervallo *limitato* di valori (di solito quelli nell'intervallo $[-2^{31}, 2^{31} - 1]$ se il tipo `int` ha ampiezza 32 bit). Quindi, se x e y contengono un valore molto grande e l'altra un valore molto piccolo, l'espressione $x - y$ potrebbe causare *overflow*³ e il risultato della computazione sarebbe indefinito. La soluzione diretta, basata sull'uso di una variabile temporanea non soffre di tale problema.

Come altro esempio consideriamo le due implementazioni equivalenti della stessa funzione `f()`.

✗ Soluzione "astuta"	✓ Soluzione leggibile
<pre>int f(int x) { return (x > 0) - (x < 0); }</pre>	<pre>int f(int x) { if (x > 0) return 1; else if (x < 0) return -1; else return 0; }</pre>

La versione leggibile rende evidente che `f(x)` rappresenti la funzione *segno*, che ritorna 1 se x è positivo, -1 se x è negativo, e 0 se x vale zero. La versione “astuta” calcola correttamente lo stesso risultato⁴, ma richiede uno sforzo notevole per essere compresa.

Vediamo un ultimo esempio tratto da Kernighan e Plauger. Consideriamo il seguente frammento di codice:

```
#define N 10
int mat[N][N];
int i, j;

for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        mat[i][j] = ((i+1)/(j+1))*((j+1)/(i+1));
    }
}
```

Il codice inizializza la matrice `mat[][]`, ma non è chiaro con quali valori. Per capirlo osserviamo che:

- Poiché i, j sono interi, allora i rapporti $(i+1)/(j+1)$ e $(j+1)/(i+1)$ vengono valutati secondo le regole del linguaggio C in base alle quali il rapporto tra due interi è un intero ottenuto troncando il risultato;

3 A lezione utilizzo questo esempio per mostrare quanto sia difficile fare dimostrazioni formali di correttezza. Infatti, assumendo che x e y vengano rappresentate con precisione infinita, è effettivamente vero che il codice “astuto” produce sempre il risultato corretto. Purtroppo, nella maggior parte dei linguaggi di programmazione tale assunzione non è vera. Se la si incorpora nella dimostrazione di correttezza -- che diventa molto più complessa -- si scopre che effettivamente ci sono casi in cui il frammento di codice non produce il risultato atteso.

4 La versione “astuta” potrebbe risultare più efficiente su certe architetture hardware, perché evita le istruzioni di salto condizionato che nei moderni processori comportano delle inefficienze. Tuttavia, nella maggior parte delle applicazioni reali il miglioramento è insignificante e non compensato dalla perdita di leggibilità.

- Se $i = j$, allora l'espressione $((i+1)/(j+1)) * ((j+1)/(i+1))$ ha valore 1 perché i numeratori e denominatori sono uguali;
- Se $i \neq j$, allora dobbiamo distinguere due sotto casi:
 - $i < j$: in questo caso $(i+1)/(j+1)$ vale zero, perché il numeratore è minore del denominatore. Di conseguenza, a $mat[i][j]$ viene assegnato il valore zero
 - $i > j$: in questo caso $(j+1)/(i+1)$ vale zero, perché il numeratore è minore del denominatore. Anche in questo caso a $mat[i][j]$ viene assegnato il valore zero

In sostanza, il frammento di codice precedente inizializza la matrice in modo che contenga tutti zeri tranne che nella diagonale principale che contiene; in altre parole, $mat[][]$ viene definita come la matrice identità.

Un modo molto più chiaro per realizzare la stessa computazione è, ad esempio:

```
#define N 10
int mat[N][N];
int i, j;

for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        if (i == j)
            mat[i][j] = 1;
        else
            mat[i][j] = 0;
    }
}
```

oppure, in modo più compatto ma meno leggibile:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        mat[i][j] = (i == j ? 1 : 0);
    }
}
```

oppure, in modo ancora più compatto ma ancor meno leggibile:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        mat[i][j] = (i == j);
    }
}
```

Tutte le versioni precedenti sono preferibili rispetto a quella "astuta".

 Evitare soluzioni "astute"

2.11 Macro

Programmando in linguaggio C/C++, capita talvolta di usare delle macro del preprocessore al posto di funzioni. Ciò non è quasi mai una buona idea, perché le macro potrebbero avere comportamenti inattesi (ma corretti dal punto di vista del linguaggio) che possono portare ad errori difficili da indi-

viduare. Ad esempio, supponiamo di definire una macro `MAX(a, b)` che restituisce il valore massimo tra `a` e `b`:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Consideriamo ora il seguente frammento di codice:

```
int x = 10;
int y = 32;
int z = MAX(x++, y++);
```

Se `MAX()` fosse una funzione, ci aspetteremmo che al termine delle tre istruzioni si abbia `z = 32`, `x = 11` e `y = 33`. In realtà, nelle macro si applica la *sostituzione lessicale dei parametri*, cioè le espressioni `x++` e `y++` vengono sostituite al posto dei parametri formali `a` e `b` della macro. Il risultato è che il codice precedente viene espanso in:

```
int x = 10;
int y = 32;
int z = ((x++) > (y++) ? (x++) : (y++));
```

per cui la variabile che contiene il massimo tra `x` e `y` viene incrementata *due* volte, non una come ci si aspetterebbe dalla chiamata della macro!

Per evitare sorprese del genere è quindi opportuno ridurre al minimo l'uso di macro; la funzione `MAX()` potrà ad esempio essere definita come:

```
int MAX(int x, int y) {
    return (x > y ? x : y);
}
```

Bibliografia

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2nd edition, Prentice Hall, Inc., 1988. ISBN 0-13-110362-8

Brian W. Kernighan, Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999, ISBN 978-0201615869

Brian W. Kernighan, P. J. Plauger, *The Elements of Programming Style*, 2nd edition, McGraw-Hill, 1978, ISBN 0-07-034207-5

Steve McConnell, *Code Complete*, 2nd edition, Microsoft Press; 2004, ISBN 978-0735619678

William Strunk, E. B. White, *The Elements of Style*, 4th ed., Pearson, 1999, ISBN 978-0-205-30902-3