Programmazione: breve guida di stile

[Bozza]

Moreno Marzolla
Università di Bologna, Dipartimento di Informatica—Scienza e Ingegneria
Ultimo aggiornamento: 9 giugno 2022

Indice

Introduzione

Il termine *programmare* indica l'attività con cui si "istruisce" un dispositivo automatico di calcolo per renderlo in grado di risolvere in autonomia un problema o classe di di problemi. Nell'accezione più comune, 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 alla macchina che li eseguirà. Niente di più sbagliato: i programmi sono mezzi con cui si comunica un'idea ad altre persone; di conseguenza, programmare è un atto creativo non troppo diverso dallo scrivere una lettera o un documento. Come una lettera, un programma deve esprimere un concetto in modo chiaro e sintetico; a differenza di una lettera, un programma potrà necessitare di modifiche o estensioni da parte di altri programmatori. 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 produce programmi di bassa qualità.

Una volta compreso che un programma è destinato sia alle macchine che alle persone, diventa chiara la necessità di adottare delle regole di stile, esattamente come un testo non deve rispettare solo le regole grammaticali della lingua italiana, ma deve anche essere chiaro e scorrevole; dopo tutto, un programma viene letto molte più volte di quante viene scritto! Un programma ideale 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.

Questo documento descrive alcune buone pratiche di programmazione che hanno lo scopo di migliorare la qualità dei programmi. Si tratta di pratiche ampiamente condivise dalla comunità informatica, che vanno però 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.

I concetti verranno illustrati con frammenti di codice in linguaggio C, per cui è necessario un minimo di familiarità con il linguaggio. Tuttavia, la maggior parte delle regole sono valide a prescindere dal linguaggio di programmazione che si usa.

Una trattazione più completa si trova nel libro:

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

Personalmente ritengo l'acquisto di questo libro 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

che pur essendo datato (gli esempi di codice sono in FORTRAN e PL/I) resta validissimo.

Sebbene questo documento sia rivolto principalmente agli studenti e alle studentesse dei miei corsi, ritengo che possa essere utile anche tutti coloro che si sono appena affacciati al mondo della programmazione, e forse anche a chi ha fatto della programmazione la propria professione.

Questo documento è diviso in due parti: “Forma” e “Sostanza”. Nella prima parte verranno illustrate alcune regole per migliorare l'aspetto dei programmi. Non è solo una questione solo estetica: un programma ordinato non è più facile da comprendere e per questa ragione è meno probabile che contenga errori. La seconda parte 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 and Kent Beck)

1.1 Indentazione

Con il termine “indentazione” si indica 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 di seguito due versioni, che sono identiche tranne che per l'indentazione.

✖ **Nessuna indentazione**

```
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;
}
```

✔ **Indentazione corretta**

```
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;
}
```

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

✖ **Indentazione fuorviante**

```
if (a > max)
  a = max;
  b = max – 1;
```

✔ **Indentazione corretta**

```
f (a > max)
  a = max;
b = max – 1;
```

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 si intende composto solo dalla riga successiva in base alle regole del linguaggio C. A destra viene mostrato lo stesso codice indentato in modo corretto. Si noti che 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”

```
int power(int x, int n)
{
  int result;

  if (n < 0)
    {
      result = 0;
    }
  else
    {
      result = 1;
      while (n > 0)
        {
          result *= x;
          n--;
        }
    }
  return result;
}
```

Indentazione stile “K&R”

```
int power(int x, int n)
{
  int result;

  if (n < 0) {
    result = 0;
  } else {
    result = 1;
    while (n > 0) {
      result *= x
      n--;
    }
  }
  return result;
}
```

Personalmente adotto l’indentazione “K&R” perché produce codice più compatto e quindi sfrutta meglio lo spazio a disposizione. Nello stile “K&R” le parentesi graffe di apertura di un blocco vengono messe sulla stessa riga di un `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`. Inoltre, 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 c’è quindi nessun 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:

✖ **Nessuno spazio, 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).

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

1.3 Layout

I primi videoterminali 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 sono associati valori. Ogni linguaggio definisce i propri vincoli su come un identificatore possa essere scritto. Ad esempio, in C/C++ un identificatore può contenere 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 molti casi potrebbe essere fissata una lunghezza massima. Gli identificatori non validi vengono rifiutati dal compilatore/interprete.

☞ Fare riferimento alla specifica del linguaggio per sapere quali simboli sono ammessi negli identificatori

Ogni linguaggio ha le proprie consuetudini sulla struttura dei nomi. In C si usano preferibilmente identificatori brevi, in certi casi 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 degli identificatori, anche se suggerisco di adeguarsi per quanto possibile alle consuetudini del linguaggio.

☞ Attenersi alle convenzioni per la scelta dei nomi degli identificatori.

Il nome di un identificatore dovrebbe soddisfare due requisiti contrastanti:

- deve essere abbastanza breve, per ridurre la probabilità che il programmatore commetta errori di scrittura;
- deve 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 (*minimum spanning tree*)	`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 di identificatori lunghi ci sono due possibilità:

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

- usare le maiuscole (es., `minSpanningTree`);
- usare il carattere _ come separatore (es., `min_spanning_tree`).

Suggerisco di usare quest'ultima, perché siamo abituati a leggere parole separate da spazi, per cui il carattere _ rende più facile cogliere le singole componenti del nome.

☞ Usare nomi appropriati per gli identificatori

1.5 Commenti

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

- 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 speciali che vengono interpretate da preprocessori o estensioni del compilatore.

Idealmente, ogni funzione o metodo 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:

✔ **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);
```

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, si usino i blocchi di commento per descrivere le funzionalità di una porzione di codice o l'algoritmo impiegato per risolvere un determinato problema, nel caso in cui queste informazioni non siano facilmente comprensibili guardando il codice.

☞ Non parafrasare il codice nei commenti

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 numerici[2] diversi da 0 e 1 che compaiono in un programma sorgente sono spesso 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, … . Esistono anche altri meccanismi per introdurre delle costanti: ad esempio, in C sarebbe stato possibile definire costanti globali (*named constants*):

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.

```
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
```

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

☞ Usare simboli o “named constants” al posto di costanti magiche

Si tenga presente che i letterali numerici 0 e 1 sono talmente ubiqui 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**

```
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;
  }
}
```

✔ **Intento evidente**

```
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]);
  }
}
```

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 `status` con una maschera binaria composta utilizzando l'operatore “or logico”. 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 il valore della variabile “status” con l'espressione `BIT_RD | BIT_WR | BIT_EXCL` occorre usare le parentesi:

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

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

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

1.9 Lunghezza delle funzioni

Una funzione o metodo troppo lunghi sono difficili da leggere e comprendere. Non è un caso che uno dei principi di base della programmazione sia la *decomposizione* di problemi complessi in problemi più semplici. Il principio di decomposizione si applica in molti contesti, tra cui la scrittura di codice: se una funzione è troppo lunga, può essere opportuno decomporla in più funzioni.

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 vanno esaminate con attenzione. 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. Il problema può essere risolto 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.

☞ Evitare di scrivere l'intero programma nella funzione (o nel metodo) main()

Come esempio consideriamo una funzione `floyd_warshall()` che implementa l'algoritmo di Floyd e Warshall per il calcolo dei cammini minimi da singola sorgente in un grafo orientato pesato *g*. I dettagli dell'algoritmo non sono importanti qui; 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 le funzioni definite nel file sorgente `floyd-warshall.c` è possibile usare 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 la *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. Funzioni con complessità ciclomatica superiore a 10-15 richiedono attenzione.

2 Sostanza

2.1 Evitare ripetizioni

Le ripetizioni di codice vanno evitate, non solo perché rendono il codice più lungo e complesso da capire, ma anche perché possono favorire errori insidiosi: se la parte ripetuta deve essere in seguito modificato, 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; in tal caso occorre fermarsi a riflettere.

Esistono diversi modi per eliminare codice ripetuto. In molti casi si può dichiarare la parte ripetuta in una funzione, con opportuni parametri, da richiamare quando serve. 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**

```
int 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;
}

int 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.

☞ Evitare ripetizioni

In casi più complessi potrebbe non essere possibile usare indici di cicli per parametrizzare le righe di codice ripetuto. Queste situazioni richiedono soluzioni un po' più raffinate. Esempio:

✖ **Codice ripetuto**

```
void foo(int *v, int i, int j)
{
  baz(v, i-1, j+3);
  baz(v, i+1, j-2);
  baz(v, i-3, j+1);
  baz(v, i-4, j  );
  baz(v, i-2, j-2);
  baz(v, i  , j-4);
  baz(v, i-1, j-1);
  baz(v, i-1, j+1);
  baz(v, i+2, j+2);
}
```

✔ **Ok**

```
void foo(int *v, int i, int j)
{
  const int di[] =
   {-1, 1, -3, -4, -2, 0, -1, -1, 2};
  const int dj[] =
   {3, -2, 1, 0, -2, -4, -1, 1, 2};
  const int N =
    sizeof(di)/sizeof(di[0]);
  int t;
  for (t=0; t<N; t++) {
    baz(v, i+di[t], j+dj[t]);
  }
}
```

Nell'esempio sopra usiamo due array paralleli *di*[] e *dj*[], contenenti rispettivamente i valori da sommare agli indici *i* e *j* nelle diverse invocazioni della funzione `baz()`. La lunghezza *N* di questi array viene calcolata; in questo modo è possibile aggiungere o rimuovendo elementi da *di*[] e *dj*[] senza doversi preoccupare di modificare anche *N*; è però indispensabile che *di*[] e *dj*[] abbiano sempre la stessa lunghezza. Ricordiamo che l'operatore `sizeof()` applicato ad un array dichiarato sullo *stack* restituisce il numero di byte occupato dall'array. Non si può fare lo stesso con array definiti nello *heap* con `malloc()`. In altre parole, nel frammento di codice seguente

```
int *a = (int*)malloc(15 * sizeof(int));
const int N = sizeof(a)/sizeof(a[0]); /* Attenzione! */
```

l'espressione `sizeof(a)` ha lo stesso valore di `sizeof(int*)`, cioè il numero di byte occupati da un puntatore a intero, non dall'array *a*[] di 15 interi!

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%
$62300 < R$	12.75%

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-esimo */
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 dal confronto delle due versioni, quella a 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 due array paralleli, è molto più compatta ma probabilmente necessita di commenti al codice per renderne chiaro l'intento.

2.2 Semplificare le strutture condizionali

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

```
if (Cond)
  Strue;
```

```
if (Cond)
  Strue;
else
  Sfalse;
```

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 che devono

essere non vuoti. Ciò significa che i blocchi condizionali if-then-else non dovrebbero mai avere il ramo “vero” vuoto.

✖ Ramo “vero” vuoto

```
if (a < b) {
  /* vuoto */
} else {
  bar();
}
```

✔ Ok

```
if (a >= b) {
  bar();
}
```

☞ Evitare blocchi if-then-else con il ramo “vero” vuoto

Se si usino strutture condizionali annidate (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

```
if (a < b) {
  foo();
} else {
  if (a >= b) { /* superfluo */
    bar();
  }
}
```

✔ Ok

```
if (a < b) {
  foo();
} else {
  bar();
}
```

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

```
if (A) {
  if (B) {
    if (C) {
      if (D) {
         blah();
      }
    }
  }
}
```

✔ Sì

```
if (A && B && C && D) {
  blah();
}
```

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

```
if (A) {
  if (B) {
    foo();
  }
} else {
  bar();
}
```

Versione 2 (non equivalente)

```
if (A && B) {
  foo();
} else {
  bar();
}
```

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()`.

☞ Semplificare le strutture condizionali dove 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"

```
int i;
for (i = 0; i < 100; i = i + 5) {
  blah(i);
}
```

Usando "while"

```
int i = 0;
while (i < 100) {
  blah(i);
  i = i + 5;
}
```

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 (`*`iniz`*`; `*`test`*`; `*`aggiornamento`*`) {`
`  `*`corpo`*
`}`
	`while (`*`test`*`) {`
	`  `*`corpo`*
	`}`

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

`i = 0;`
`for ( ; i < 10; i++) {`
`  blah(i);`
`}`
	`for ( i = 0; i < 10; i++ ) {`
	`  blah(i);`
	`}`

`i = 0;`
`for ( ; i < 10 ; ) {`
`  blah(i);`
`  i++;`
`}`	

`i = 0;`
`for ( ; ; ) {`
`  if (i >= 10) break;`
`  blah(i);`
`  i++;`
`}`	

Come esempi 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 non solo può confondere chi legge il codice, ma può anche 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 i valori 132, 3 e 21.

Consideriamo le due soluzioni seguenti: quella di sinistra fa uso di un ciclo `for` esterno, mentre quella a destra usa un `while`.

✖ **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 significa che il primo carattere immediatamente successivo all’ultima cifra di un numero viene saltato, e questo causa un accesso ad un elemento successivo all'ultimo se la stringa *s* termina con una cifra, ad es., *s* = “123a34”.

Usando una struttura `while` (versione 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 usare un ciclo “for” se la variabile di ciclo viene modificata nel corpo

2.4 `break` e `continue`

I costrutti `break` e `continue` vanno usati con parsimonia, perché dovrebbero essere l'eccezione più che la regola. L'istruzione break introduce una violazione del principio della programmazione strutturata, in cui ogni blocco strutturato dovrebbe avere un singolo punto di ingresso e un singolo punto di uscita. Di conseguenza, il modo preferito 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 per sfruttare una condizione leggermente più complessa del ciclo "for" (a destra).

✖ No

```
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
}
```

✔ Sì

```
/* Si può anche usare "while" */
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
}
```

Lo schema precedente è abbastanza frequente, per cui vale la pena fornire una regola generale che si può applicare in molti casi (non in tutti!). Supponiamo di voler iterare un ciclo enumerativo fino a quando una certa condizione *cond* diventa vera.

✖ No

```
for (init; test; incr) {
  if (cond)
    break;
}
```

✔ Sì

```
for (init; test && !cond; incr) {
  /* corpo vuoto */
}
/* oppure */
init;
while (test && !cond) {
  incr;
}
```

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 propria posizione all'interno della sequenza. Solitamente è possibile accedere ad un elemento dell'array in un tempo che non dipende dalla posizione. Esempio di array in C:

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

Una *tupla* (detta anche *record*) è un tipo di dato in grado di mantenere un insieme finito di valori di tipo anche diverso. Ogni elemento di una tupla può essere acceduto, in base al linguaggio di programmazione, usando un nome diverso per ogni elemento, oppure usando la posizione 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;
};
```

Se tutti gli elementi di una tupla hanno lo stesso tipo, si potrebbe pensare di utilizzare un array per risparmiare qualche riga di codice. Questa sarebbe una pessima idea, come vediamo con un esempio.

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;       /* velocita' */
} 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 risulta incomprensibile, perché non è evidente che valori siano contenuti in *p*[3] e *p*[4]; per saperlo occorre cercare il punto nel codice in cui il parametro attuale viene inizializzato. Invece, nella versione a destra risulta evidente cosa calcoli la funzione, riconoscendo che si tratta dell'energia cinetica. 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;
}
```

Ci sono altri casi in cui l'uso di variabili globali è frutto di una pessima pratica di programmazione. 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;

/* Prima di invocare la funzione occorre azzerare s */
int sum(int v[], int n)
{
  if (n == 0) {
    return 0;
  } else {
    s += sum(v, n-1) + v[n-1];
    return s;
  }
}
```

✔ **Senza variabile globale**

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

Entrambe le soluzioni sfruttano la stessa idea; quella di sinistra, però, fa uso di una variabile globale *s* per accumulare i valori della sommatoria. Questa soluzione non è solo più complessa di quella che non fa uso di variabili globali, ma è estremamente fragile perché richiede che l'utente azzeri *s* ogni volta prima di invocare la funzione.

☞ Limitare l'uso di variabili globali

2.7 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 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

2.8 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 != 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 diventare 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).

2.9 Evitare soluzioni "astute"

A volta mi capita di leggere codice in cui l'autore/autrice hanno pensato (male) di esibire il proprio virtuosismo, lanciandosi in soluzioni “astute” a problemi semplici. Con il termine “soluzioni astute” faccio riferimento a soluzioni non ovvie a problemi tutto sommato semplici, soluzioni che solo in apparenza possono sembrare più eleganti e/o compatte di quelle dirette. Quasi sempre, le soluzioni astute non si rivelano né eleganti, né compatte, e spesso nemmeno del tutto corrette. In generale, **sviluppare soluzioni contorte a problemi semplici non è un pregio, ma un grave difetto**. Un bravo programmatore o una brava programmatrice sviluppano soluzioni leggibili, facilmente modificabili, ed efficienti. Un programmatore scarso tende a nascondere la propria mediocrità in esibizioni di maestria che spesso non padroneggia. Vediamo alcuni esempi.

Supponiamo di avere tre variabili *a*, *b*, *c* di tipo `char`. Supponiamo che le variabili possano assumere come valore solo `'X'` oppure `'Y'`. Vogliamo scrivere un frammento di codice per verificare se tutte e tre le variabili hanno valore `'X'`.

✖ Soluzione "astuta"

```
if (a + b + c == 3 * 'X') {
  printf("tutti uguali a X\n");
} else {
  printf("non tutti uguali a X\n");
}
```

✔ Soluzione leggibile

```
if ((a == 'X') &&
    (b == 'X') &&
    (c == 'X')) {
  printf("tutti uguali a X\n");
} else {
  printf("non tutti uguali a X\n");
}
```

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 tantomeno più leggibile di quella “normale”. Inoltre, 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 sono tutte uguali a `'X'` !

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"

```
x = x - y;
y = x + y;
x = y - x;
```

✔ Soluzione leggibile

```
const int tmp = x;
x = y;
y = tmp;
```

Facendo qualche prova, potremmo convincerci che la soluzione “astuta” sia corretta; è anche possibile dimostrarlo formalmente (lo faccio a lezione parlando di asserzioni e invarianti). Purtroppo l'intuizione empirica 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). Ciò significa che se le variabili *x* e *y* contengono una un valore “molto grande” e l'altra un valore “molto piccolo”, l'espressione `x - y` potrebbe causare overflow[3]. La soluzione più semplice e diretta, oltre a non avere questo problema, è immediatamente comprensibile.

Come ultimo esempio consideriamo le due versioni seguenti di una stessa funzione `f()`.

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.

✖ **Soluzione "astuta"**

```
int f(int x)
{
  return (x > 0) - (x < 0);
}
```

✔ **Soluzione leggibile**

```
int f(int x)
{
  if (x > 0)
    return 1;
  else if (x < 0)
    return -1;
  else
    return 0;
}
```

La versione leggibile rende evidente il fatto che *f*(*x*) è la funzione *segno*, che ritorna 1 se *x* è strettamente positivo, -1 se *x* è strettamente negativo, e 0 se *x* vale zero. La versione “astuta” è corretta[4], ma richiede uno sforzo notevole per capire che cosa calcola.

☞ Scrivere chiaramente evitando soluzioni “astute”

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

Bibliografia

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

Brian W. Kernighan and 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