Capitolo 5

Astrazione procedurale e funzioni

5.1 Astrazioni

Nei capitoli precedenti abbiamo più volte utilizzato il termine astrazione, con riferimento in particolare al controllo, alle operazioni, ai tipi di dato.

Definire un'*astrazione* significa, in generale, definire un'entità (ad esempio un tipo di dato) in base soltanto alle sue proprietà visibili all'esterno, ignorando completamente i dettagli della sua realizzazione concreta.

Nei linguaggi di programmazioni sono normalmente presenti tre tipi principali di forme di astrazione:

- astrazione sui dati
- astrazione sulle operazioni (o *procedurale*)
- astrazione sul controllo.

Molte di queste, di tutti e tre le forme, sono fornite come *primitive*. Altre possono essere definite dall'utente utilizzando opportuni costrutti forniti dal linguaggio stesso.

Le astrazioni primitive presenti in C++ sono state già presentate in massima parte nei capitoli precedenti, mentre per le astrazioni definite dall'utente abbiamo finora visto soltanto il caso delle astrazioni sui dati (Capitolo 4). Prima di addentrarci nella descrizione dettagliata di un altro caso di astrazione supportato dal C++, quella sulle operazioni, riassumiamo brevemente qui di seguito le principali astrazioni primitive e definite da utente presenti nei linguaggi di programmazione convenzionali ed in particolare nel C++.

Astrazioni primitive

1. Sui dati:
   tipi di dato primitivi – in C++: `int`, `float`, `char` e `bool`.

2. Sulle operazioni:
   sono strettamente correlate ai tipi di dato (a rigore, fanno parte della definizione stessa del tipo) – sono tutte e sole le operazioni aritmetiche, di confronto e logiche previste per i tipi primitivi ed elencate, nel caso del C++, nel sottocapitolo 2.3.

3. Sul controllo:
   - valutazione espressioni (con le relative regole di precedenza ed associatività)
   - statement del linguaggio – in C++: assegnamento, `if-else`, `while`, `for`, ecc.
   - gestione delle eccezioni, di cui parleremo in un capitolo successivo, nella II parte di questo testo.

Astrazioni definite da utente

1. Sui dati:
   - tipi definiti da utente – in C++:
     - tipi strutturati, definiti tramite opportuni costruttori di tipo, come *array*, *struct*, ecc.
     - puntatori
     - tipi per enumerazione
   - tipi di dato astratti (ADT), definiti, ad esempio in C++, tramite il costrutto di *class* (che permette di definire astrazioni sui dati e sulle operazione insieme).

2. Sulle operazioni:
   tramite definizione ed uso di sottoprogrammi – in C++ tramite il costrutto di *funzione* che vedremo in questo capitolo. Alcune astrazioni sulle operazioni definite da utente di uso comune sono predefinite e fornite dal linguaggio come *funzioni di libreria* – in C++: funzioni di input/output (come `>>` e `get`), funzioni matematiche (come `sqrt` e `pow`), funzioni per la manipolazione di stringhe (come `size` e `find` della classe `String`).

3. Sul controllo:
   i linguaggi di programmazione convenzionali non danno normalmente la possibilità all'utente di definire le proprie astrazioni sul controllo, sebbene questa possibilità non sia, almeno in linea di principio, priva di interesse.

Nel resto di questo capitolo vedremo in dettaglio le possibilità offerte dal C++ per permettere all'utente di definire le proprie astrazioni sulle operazioni tramite il costrutto di funzione.

5.2 Definizione di funzioni

Consideriamo un semplicissimo esempio per motivare l'introduzione delle funzioni ed illustrarne sinteticamente la definizione e l'uso.

Problema. Realizzare un programma in grado di leggere 3 interi da standard input, determinare il maggiore dei tre, e stampare il risultato su standard output.

È possibile/conveniente vedere l'azione *"determina il maggiore di 3 interi"* come un'operazione astratta, definita e implementata a parte. Questo può essere facilmente realizzato in C++ tramite le *funzioni*. Il costrutto

```
int maggiore3 (int i, int j, int k)
  {
     "implementazione della funzione"
  }
```

definisce una nuova funzione, di nome `maggiore3`, che realizza l'operazione di individuare e restituire come suo risultato il maggiore tra tre numeri interi `i`, `j` e `k`. In termini astratti, `maggiore3` può vedersi come una vera e propria funzione (nel senso matematico del termine) che prende tre numeri interi e restituisce un numero intero e cioè:

$$\texttt{int} \times \texttt{int} \times \texttt{int} \longrightarrow \texttt{int}$$

La funzione `maggiore3` può essere usata come un'operazione qualsiasi nel programma principale. Ad esempio:

```
int main() {
    int A,B,C,R;
    cin >> A >> B >> C;
    R = maggiore3(A,B,C);
    cout << "il maggiore e'" << R << endl;
    return 0;
}
```

A questo punto dobbiamo fornire una possibile implementazione della funzione `maggiore3`. Questo si ottiene specificando in modo opportuno il "corpo" del costrutto di funzione utilizzato per la definizione della `maggiore3`:

```
int maggiore3 (int i, int j, int k) {
     int T;  /*variabile locale*/
     if(i>j) T=j;
     else T=j;
     if(T>k) return T;
     else return k;
}
```

Nel seguito verranno affrontati tre aspetti principali dell'uso di funzioni: la *definizione* di funzione, la *chiamata* di funzione (flusso di controllo), il *passaggio parametri*. In questo sottocapitolo esaminiamo la definizione di funzione (sintassi), con particolare riferimento al C++. Gli altri due aspetti verranno invece esaminati in sottocapitoli successivi.

5.2.1 Il costrutto di funzione

Mostriamo la sintassi del costrutto che permette la definizione di una funzione in C++.

Definizione di funzione in C++.

```
t ALFA (t1 ID1,..., tn IDn)
     { DICHIARAZIONI
             +
         STATEMENT
     }
```

dove:

- `t`: tipi qualsiasi = tipo della funzione, ovvero tipo del risultato restituito dalla funzione
- `ALFA`: identificatore = nome della funzione
- $\texttt{ID}_1,\ldots,\texttt{ID}_n$: identificatori = *parametri formali* della funzione; rappresentano i dati di ingresso della funzione
- $\texttt{t}_1,\ldots,\texttt{t}_n$: tipi qualsiasi = tipi dei parametri formali. □

Esempi:

- ```
  int maggiore3 (int i, int j, int k)
    { ... }
  ```

  definisce una funzione di nome `maggiore3`, con tre parametri formali, di nome `i`, `j`, `k`, tutti e tre di tipo `int`.

- ```
  int main()
    { ... }
  ```

  definisce una funzione di nome `main` (parola chiave), senza parametri formali (e cioè, con $n = 0$).

5.2.2 Programma completo con dichiarazione di funzione

L'esempio seguente mostra come si presenta un programma C++ completo, composto dal programma principale e da una funzione (ovvero, un *sottoprogramma*). Ritorneremo sulla strutturazione di un programma in più sottoprogrammi nel sottocapitolo 5.4.

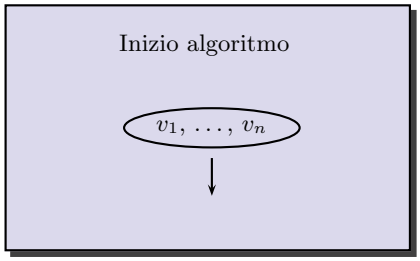
Esempio 5.1 *(Maggiore tra tre numeri)*

```
#include <iostream>
using namespace std;

int maggiore3 (int i, int j, int k) {
     int T;  /*variabile locale*/
     if (i>j) T=j;
     else T=j;
     if (T>k) return T;
     else return k;
}

int main() {
    int A,B,C,R;
    cin >> A >> B >> C;
    R = maggiore3(A,B,C);
    cout << "il maggiore e'" << R << endl;
    return 0;
}
```

Diagrammi di flusso

I blocchi ovali iniziale e finale dei diagrammi di flusso visti nel sottocapitolo 1.2 possono essere utilizzati anche per rappresentare l'inizio e la fine di un algoritmo relativo ad una funzione che preveda dei parametri in ingresso e dei risultati in uscita.

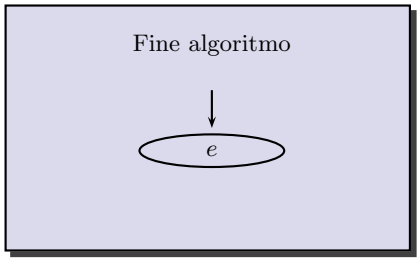
Blocco ovale iniziale



dove $v_1, \ldots, v_n$ sono variabili ($n \geq 0$).

$v_1, \ldots, v_n$ rappresentano i parametri formali della funzione, ovvero i dati in ingresso dell'algoritmo descritto dal diagramma di flusso. I loro valori si assume siano assegnati "dall'esterno" prima dell'inizio dell'esecuzione dell'algoritmo.

Blocco ovale finale



dove e è un'espressione qualsiasi.

Il valore (ottenuto dalla valutazione) di e costituisce il risultato della funzione, ovvero i dati di output prodotti dall'esecuzione dell'algoritmo descritto dal diagramma di flusso.

5.3 Esecuzione di una funzione

5.3.1 Chiamata di funzione

L'esecuzione di una funzione avviene tramite una *chiamata di funzione*. Data una funzione `ALFA` (definita come sopra), la chiamata della funzione `ALFA` in C++ è un'*espressione* avente la seguente forma:

`ALFA(`$\mathtt{E}_1, \ldots, \mathtt{E}_n$`)`

dove $\mathtt{E}_1, \ldots, \mathtt{E}_n$ sono espressioni qualsiasi (i *parametri attuali* della funzione).

La valutazione dell'espressione `ALFA(`$\mathtt{E}_1, \ldots, \mathtt{E}_n$`)` avviene nel modo seguente:

1. valuta i parametri attuali: $\mathtt{E}_1 \rightsquigarrow v_1, \ldots, \mathtt{E}_n \rightsquigarrow v_n$ (dove $\mathtt{E} \rightsquigarrow v =$ indica che la valutazione dell'espressione `E` produce il valore v).
2. associa i valori $v_1, \ldots, v_n$ ai corrispondenti parametri formali $\mathtt{ID}_1, \ldots, \mathtt{ID}_n$ (passaggio parametri)
3. esegue il corpo della funzione `ALFA` (con $\mathtt{ID}_1, \ldots, \mathtt{ID}_n$ inizializzati con $v_1, \ldots, v_n$) fino ad eseguire uno statement di `return E`.
4. restituisce il valore dell'espressione `E` del `return` come risultato della (valutazione della) chiamata di `ALFA`.

Terminata la valutazione della chiamata di `ALFA` l'esecuzione continua nel programma "chiamante". Il flusso di controllo passa dal programma chiamante alla funzione (con la chiamata di funzione) e ritorna dalla funzione al programma chiamante, al punto di chiamata, quando la funzione esegue il `return`.

Si osservi che:

c'è passaggio di dati in input (al momento della chiamata)

```
   A       B       C
   ↓       ↓       ↓
int i   int j   int k
```

e in output (al momento dell'esecuzione del `return`)

```
return T
   ↓
maggiore(A,B,C)
```

I parametri attuali e i parametri formali devono, in generale, coincidere, sia come tipo che come numero (unica eccezione in C++ è la presenza di parametri di default, di cui parleremo in un capitolo successivo). Inoltre, anche il tipo dell'espressione del `return` e il tipo della funzione devono coincidere.

Mostriamo alcuni esempi di chiamata di funzione utilizzando la funzione `maggiore3` definita nel sottocapitolo precedente.

- La chiamata di funzione è un'espressione e può apparire ovunque possa stare un'espressione. Ad esempio:

```
int x;
x = 2*maggiore3(5,2,7);  /* compare come sotto espressione */
```

- I parametri attuali possono essere espressioni qualsiasi. Ad esempio:

```
y=3;
x = 2*maggiore3(y*5, y*y, 10);
```

  Il valore assegnato a `x` sarà 30.

- Le chiamate di funzione possono essere annidate. Ad esempio:

```
int A,B,C,D;
A=5; B=3; C=4; D=2;
x = maggiore3(A, B, maggiore3(C, D, 3));
```

  Il valore assegnato a `x` sarà 5 (partire nella valutazione dalla chiamata di funzione più interna).

5.3.2 Statement `return`

Sintassi:

```
return E
```

dove `E`: espressione qualsiasi
= valore restituito dalla funzione.

Semantica: l'esecuzione continua dall'espressione contenente la chiamata di funzione; il valore di `E` viene restituito come risultato della chiamata di funzione.

Lo statement `return` può apparire ovunque possa stare uno statement semplice in un programma. In particolare può apparire all'interno di cicli o di statement condizionali (`if` e `switch`), a qualsiasi livello di annidamento. L'esecuzione del `return` provoca in ogni caso l'uscita immediata dalla funzione, e quindi l'uscita dal costrutto che lo contiene. Si osservi che questo comportamento è contrario ai principi della programmazione strutturata (i costrutti devono avere una sola uscita), ma risulta molto comodo in pratica in varie occasioni e comunque il suo effetto è limitato alla sola funzione in cui appare.

5.3.3 Un esempio completo

Esempio 5.2 *Leggi (da standard input) un numero intero $n \geq 0$, calcola il fattoriale di n e stampa su standard output il risultato calcolato.*

Sottoproblema: calcolo del fattoriale di un numero naturale n:

$$\mathsf{fatt} : int \to int$$

$$\mathsf{fatt}(n) = \begin{cases} 1 & \text{se } n = 0 \\ 1 * 2 * \ldots * n & \text{altrimenti} \end{cases}$$

Quindi per esempio $\mathsf{fatt}(5) = 1 * 2 * 3 * 4 * 5 = 120$.

Funzione in C++:

```
int fatt(int n) {
    int ris = 1;
    for(i = 1; i <= n; i++)
      ris = ris * i;
    return ris;
}
```

Approfondimento (Prova informale di correttezza). Distinguiamo due casi: $n = 0$ e $n > 0$. **Caso** $n = 0$: La chiamata `fatt(0)` ritorna correttamente 1, in accordo con la definizione di fatt. **Caso** $n > 0$: Assumiamo che la funzione "faccia il suo dovere" per $n - 1$, cioè assumiamo che:

$$\texttt{fatt(n-1)} = 1 * 2 * \ldots n - 1 \tag{5.1}$$

Ora vediamo cosa succede per n. La funzione `fatt` su n ritorna:

$$\texttt{ris} = 1 * 2 * 3 * \ldots * n - 1 * n.$$

Approfondimento (Calcolo informale costo computazionale). Ogni operazione elementare (assegnamenti, prodotti, somme, divisioni, etc...) di un programma richiede per essere eseguita un certo costo computazionale. Dato un numero naturale $n \geq 0$, il programma `fatt` effettua dapprima un assegnamento (`ris = 1`), poi per n volte (ciclo `for`) un prodotto (`ris = ris * i`) ed alla fine un ritorno di funzione (`return ris`). Se il costo computazionale per effettuare l'assegnamento è c_1, il costo del prodotto c_2 e il costo del ritorno da funzione c_3, allora il costo totale della funzione `fatt` sarà:

$$c_1 + c_2 * n + c_3$$

In questo caso possiamo dire che, a meno di costanti moltiplicative e additive (c_1, c_2 e c_3), il costo di `fatt` "cresce come n" o equivalentemente "fatt è in $\mathcal{O}(n)$".

Altro sottoproblema: lettura (e controllo) di un numero naturale.

```
int leggi_naturale() {
     int n;
     cout << Introdurre un numero (>=0):
     do {
        cin >> n;
     } while(n < 0)
     return n;
}
```

A questo punto possiamo scrivere il programma completo:

```
#include <iostream>
int fatt(int n)
    { /*implementazione sopra*/ }
int leggi_naturale()
    { /*implementazione sopra*/ }
int main()
  {int n,z;
   int n = leggi_naturale();
   z = fatt(n);
   return 0;
  }
```

5.3.4 Funzioni senza risultato esplicito (procedure)

Ci sono operazioni che non producono un risultato esplicito, ma piuttosto modificano l'ambiente esterno (operano per *"side effects"*). Ad esempio:

1. stampa/legge dati su/da standard output/input
2. aggiorna un file
3. modifica (il contenuto di) un parametro (passaggio parametri per riferimento – Capitolo 5.5)
4. modifica un dato globale – si veda Capitolo 5.4.2.

In questi casi, il sottoprogramma è indicato come *"procedura"*. In C++ non esiste un costrutto apposito per le procedure; si usano invece le funzioni con tipo del risultato `void`.

Esempio 5.3 *Funzione che stampa i dati di una persona con un opportuno formato di stampa.*

```
struct persona {char nome[32];
                char cognome[32];
                int eta;
               };
```

```
void stampa_persona(persona P) {
     cout<< "Nome:" << P.nome << endl;
     cout<<" Cognome:" << P.cognome << endl;
     cout<<" Eta'"<< P.eta << endl;
     return;
}
```

Si noti che nel caso di funzione `void`, lo statement `return` appare senza espressione (o, meglio, contiene l'espressione vuota).

La chiamata della funzione `stampa_persona` può essere effettuata nel seguente modo:

```
persona A = {"Mario", "Bianchi", "32"};
stampa_persona(A);
```

La chiamata ha la stessa sintassi della chiamata di funzione con risultato, ma viene usata direttamente come statement.

5.4 Struttura di un programma e regole di "scope"

La struttura (statica) di un programma è determinata dall'insieme di sottoprogrammi e dichiarazioni globali che, insieme al programma principale (il `main`), compongono il programma completo. La visibilità dei nomi dei sottoprogrammi e delle dichiarazioni globali è governata dalle regole di "scope" del linguaggio (si veda Capitolo 3.10).

5.4.1 Struttura di un programma C++

La struttura tipica di un programma C++ è la seguente:

```
Dichiarazioni (globali) di costanti, tipi, variabili, ...

tipo_1 f_1(...) {
        Dichiarazioni (locali a f_1)
        Statement di f_1
}
...
tipo_n f_n(...) {
        Dichiarazioni (locali a f_n)
        Statement di f_n
}
int main() {
        Dichiarazioni (locali al main)
        Statement del main
}
```

dove: `Dichiarazioni` e `Statement` sono insiemi, anche vuoti, rispettivamente di dichiarazioni e di statement; $\texttt{f}_1, \ldots, \texttt{f}_n$ sono nomi di funzioni; $\texttt{tipo}_1, \ldots, \texttt{tipo}_n$ sono i tipi delle funzioni $\texttt{f}_1, \ldots, \texttt{f}_n$ ($n \geq 0$). $\texttt{f}_1, \ldots, \texttt{f}_n$ sono i sottoprogrammi, mentre `main` è il programma principale (di fatto una funzione come le altre, ma con nome speciale, che l'ambiente d'esecuzione riconosce e che usa come "entry point" nel programma, cui passare il controllo quando il programma deve essere eseguito).

Si noti che nel caso $n = 0$ è presente il solo programma principale.

Per poter utilizzare una funzione senza dover fornire prima la sua definizione completa, in C++ è possibile dare la dichiarazione della sola testata della funzione (il *prototipo* della funzione), separatamente dal suo corpo. Una volta fornita la dichiarazione della testata, il nome della funzione diventa visibile e quindi utilizzabile all'interno delle altre funzioni del programma. La definizione completa della funzione (testata + corpo) potrà essere fornita a parte (eventualmente compilata separatamente), e potrà essere posta anche dopo le funzioni che la utilizzano.

Ad esempio:

```
int f(int x);       // prototipo della funzione f
int g(char c);      // prototipo della funzione g
int main()
        {...}       // f e g visibili qui
int f(int x)        // definizione completa di f
        {...}
int g(char c)       // definizione completa di g
        {...}
```

Si noti che dopo la dichiarazione del prototipo della funzione va inserito un punto e virgola. Inoltre, nel prototipo è possibile specificare anche soltanto il tipo dei parametri senza dover necessariamente darne anche il nome (ad esempio, `int f(int);`).

La possibilità di specificare prototipi rende di fatto non rigido l'ordine in cui vengono fornite le dichiarazioni delle funzioni (e del `main`).

Nota. Le direttive `#include` che appaiono all'inizio di un programma provvedono, tra l'altro ad aggiungere al programma stesso tutti i *prototipi* delle funzioni di libreria utilizzate nel programma. La definizione completa (codice sorgente) di queste funzioni, invece, non viene aggiunta al programma. Le funzioni di libreria sono infatti *compilate separatamente* e il codice oggetto ottenuto viene *cucito* a quello del programma utente nella successiva fase di "linking".

5.4.2 Dichiarazioni locali e globali

Come abbiamo visto nel sottocapitolo 3.10, le regole di "scope" si basano sulla nozione di blocco. Nel caso delle funzioni, il *blocco della funzione* è costituito non soltanto dallo statement composto che rappresenta il corpo

della funzione, ma anche dalle dichiarazioni dei parametri formali che compaiono nella testata della funzione stessa. Ad esempio, nella funzione `fatt` mostrata sopra (Esempio 5.2), l'ambiente locale della funzione comprende:

- la dichiarazione della variabile `x` (parametro formale)
- la dichiarazione della variabile `ris`.

In base alle regole di "scope" enunciate nel sottocapitolo 3.10, le dichiarazioni locali ad una funzione sono visibili soltanto all'interno della funzione stessa. Pertanto, nomi introdotti in una funzione non sono utilizzabili in altre funzioni, compreso il `main`. Ad esempio:

```
int f(...)
        {...}       // qui x e y non sono visibili
int g(int x)
        { int y;
          ...       // qui x e y sono visibili
int main()
        {...}       // qui x e y non sono visibili
```

Le regole di "scope" del linguaggio ci permettono di dire anche che il campo d'azione delle *dichiarazioni globali* di un programma si estende a tutte le funzioni del programma. In altri termini, un nome introdotto da una dichiarazione globale è utilizzabile in una qualsiasi delle funzioni $\texttt{f}_1, \ldots, \texttt{f}_n$ e `main` che compongono il programma. Ad esempio:

```
const int max = 10;
struct data
        {...};      // qui max e' visibile
int f(...)
        {...}       // qui max e data sono visibili
int g(...)
        {...}       // qui max e data sono visibili
int main()
        {...}       // qui max e data sono visibili
```

Nota. Unica eccezione è costituita dal caso in cui lo stesso nome `id`, introdotto in una dichiarazione globale sia introdotto anche in una dichiarazione locale all'interno di una delle funzioni. In questo caso, la dichiarazione locale "nasconde" temporaneamente – ovvero per tutto il corpo della funzione in cui appare– la dichiarazione globale (in altri termini, un riferimento al nome `id` all'interno della funzione si riferisce alla dichiarazione locale e non a quella globale). Ad esempio:

```
const int max = 10;
int f(...)
        {...}       //qui max "globale" e' visibile
int g(...)
        { int max;
          ...       //qui max "globale" non e' visibile.
```

```
        }              e' visibile invece max "locale"
int main()
        {...}        // qui max "globale" e' visibile
```

Si noti che le dichiarazioni delle funzioni sono di fatto delle dichiarazioni globali e quindi i *nomi delle funzioni* sono visibili in tutto il programma, valendo, comunque, sempre la regola che un nome è visibile dalla sua dichiarazione in poi, e non prima. Ad esempio:

```
int f(...)
        {...}        // f visibile, g non visibile
int g(...)
        {...}        // f e g visibili
int main()
        {...}        // f e g visibili
```

Il fatto che le dichiarazioni globali siano visibili in tutto il programma permette, almeno in linea di principio, di utilizzare *variabili globali* (ovvero, variabili introdotte da dichiarazioni globali) all'interno delle diverse funzioni componenti il programma, anche come alternativa al passaggio di parametri alle funzioni stesse.

Come esempio, riprendiamo il problema del calcolo del fattoriale trattato nel paragrafo 5.3.3 e proviamo a risolverlo utilizzando una variabile globale `n` per contenere il numero di cui si vuol calcolare il fattoriale.

Esempio 5.4

```
#include <iostream>
using namespace std;

int n;
int leggi_naturale() {
   ... // stessa implementazione di 5.3.3
}

int fatt() {
   int ris = 1;
   for(int i=1; i<=n; i++)
      ris = ris * i;
   return ris;
}

int main() {
   int r;
   n = leggi_naturale();
   r = fatt();
   cout<< "Il fattoriale di" << n << "e'" << r <<endl;
   return 0;
}
```

Si noti che in questo esempio la variabile `n` nella funzione `fatt` e nel `main` è utilizzata come variabile globale, definita nel blocco più esterno. Questo permette, tra l'altro, che la funzione `fatt` possa essere definita senza parametri.

Svantaggi dell'uso di variabili globali.

1. Non c'è separazione netta tra programma e sottoprogramma: il sottoprogramma può essere utilizzato soltanto se il programma chiamante contiene la dichiarazione di quelle specifiche variabili che il sottoprogramma usa come globali; un'eventuale modifica dei nomi delle variabili globali richiederebbe la loro modifica anche in tutti i sottoprogrammi che li utilizzano. Di fatto vengono inficiate due proprietà importanti: la *modularità* del programma e la *riusabilità* dei sottoprogrammi.

2. Non è chiara l'*interfaccia* del sottoprogramma (cosa viene comunicato da/verso l'esterno), che normalmente è specificata dalla lista dei parametri formali. Quindi è ridotta anche la *leggibilità* del programma.

In conclusione, le variabili globali vanno evitate il più possibile. Unica eccezione è quando si ha a che fare con dei dati (soprattutto di grosse dimensioni, come tabelle, matrici, ecc.), utilizzati da più funzioni, e si vuole evitare di doverli passare come parametri a tutte le funzioni che li usano.

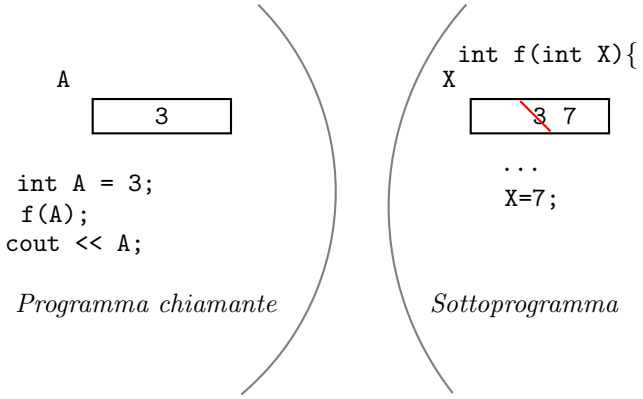
Diverso è invece il discorso per le dichiarazioni globali di costanti e tipi che risultano in molti casi del tutto adeguate.

5.5 Passaggio parametri

5.5.1 Modalità di passaggio parametri

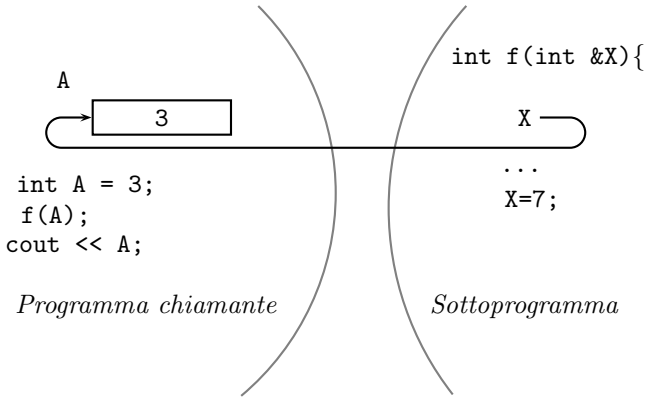
Nei sottocapitoli precedenti abbiamo visto che la chiamata di una funzione comporta, tra l'altro, un passaggio di informazioni tra parametri *attuali* (le espressioni specificate nella chiamata di funzione) e i parametri *formali* (le variabili specificate nella dichiarazione della funzione). Parametri attuali e parametri formali di una stessa funzione vengono normalmente fatti corrispondere in base alla loro posizione. Esistono però diverse modalità di passaggio di parametri. In particolare in C++ sono previste due diverse modalità: per valore e per riferimento.

Per valore. Il valore dei parametri attuali viene *copiato* nelle corrispondenti variabili che costituiscono i parametri formali della funzione.



Ne segue che una eventuale modifica del (valore del) parametro formale non modifica il valore del corrispondente parametro attuale.

Per riferimento. Il parametro formale è un riferimento al (ovvero, un sinonimo o "alias" del) parametro attuale.



Si osservi che la notazione sintattica per indicare che un parametro formale (nell'esempio, `A`) è passato per riferimento piuttosto che per valore consiste semplicemente nel premettere il simbolo `&` al nome del parametro formale. Dunque, con il passaggio parametri per riferimento, parametro attuale e parametro formale *condividono* la stessa area di memoria. Ne segue che un'eventuale modifica del (valore del) parametro formale modifica anche il valore del corrispondente parametro attuale.

Vediamo un semplice esempio che evidenzia la differenza di comportamento tra passaggio parametri per valore e per riferimento.

Esempio 5.5 *Realizzare una funzione, di nome* `scambia`*, che presi come suoi parametri due numeri interi* `x` *e* `y`*, li modifica scambiandone i valori (*`x` $\leftrightarrow$ `y`*).*

Consideriamo dapprima la soluzione con passaggio parametri per valore.

```
void scambia(int x, int y) {
   int t;
   t=x;
   x=y;
   y=t;
}
```

La seguente chiamata di funzione:

```
int a=3;
int b=5;
scambia(a,b);
cout << "a=" << a << "  " << "b=" << b;
```

produrrà:

```
a=3  b=5
```

e cioè un risultato diverso da quello atteso. Il problema è dovuto al fatto che il passaggio parametri per valore comporta la creazione di una copia dei valori dei parametri attuali (le variabili `a` *e* `b` *nell'esempio). Le modifiche fatte dalla funzione sui valori dei corrispondenti parametri formali si applicano in realtà alla copia dei parametri attuali e dunque non si ripercuotono su quest'ultimi. Al ritorno dalla funzione, i parametri attuali avranno ancora il loro valore originario.*

Vediamo lo stesso esempio, ma utilizzando passaggio parametri per riferimento.

```
void scambia(int &x, int &y) {
   int t;
   t=x;
   x=y;
   y=t;
}
```

In questo caso l'esecuzione del frammento di programma chiamante mostrato sopra, contenente la chiamata di funzione,

```
 scambia(a,b);
```

produrrà il seguente risultato:

```
a=5  b=3
```

Dunque, in questo caso, il comportamento della funzione è quello voluto. Infatti il passaggio parametri per riferimento fa sì che `x` *e* `a` *(e* `y` *e* `b`*) siano semplicemente due nomi diversi per lo stesso oggetto denotabile.*[1] *Dunque, le modifiche fatte all'interno della funzione* `scambia` *su* `x` *e* `y` *si ripercuotono sui corrispondenti parametri attuali* `a` *e* `b`*.*

Una volta capite le differenze sintattiche e semantiche tra il passaggio parametri per valore e per riferimento, è importante (e non banale) capirne il diverso utilizzo. Vediamo schematicamente quali possono essere i motivi per cui può risultare utile utilizzare il passaggio parametri per riferimento (piuttosto che quello per valore).

1. Maggior efficienza (non c'è copia dei parametri).
2. Minor occupazione di spazio di memoria (sempre per la mancanza di copia dei parametri).
3. Possibilità di modificare il valore dei parametri attuali (come mostrato con l'esempio della funzione `scambia`).
4. Possibilità di utilizzare un parametro passato per riferimento per restituire al programma chiamante uno (o più) risultati.

Per illustrare l'ultimo dei vantaggi del passaggio per riferimento citati sopra, consideriamo il seguente esempio.

Esempio 5.6 *Scrivere una funzione di nome* `eq2` *che calcola le radici di un'equazione di secondo grado:*

$$ax^2 + bx + c = 0$$

Se il discriminante è negativo la funzione restituisce il valore `false`*; altrimenti restituisce* `true`*. I coefficienti dell'equazione sono passati alla funzione come suoi parametri. Scrivere anche un* `main` *di prova che richiami in modo opportuno la funzione* `eq2`*.*

È evidente che si pone il problema di come far restituire alla funzione il risultato da essa calcolato (nel caso di discriminante non negativo), ovvero le radici dell'equazione data. L'espressione del `return` *è già utilizzata per restituire il valore booleano che ci dice se l'equazione ha soluzioni oppure no. Inoltre la soluzione dell'equazione prevede in generale due valori distinti come risultato, mentre l'espressione del* `return` *permette di specificarne solo uno.*

[1] Si noti che questo è un esempio di una situazione in cui la corrispondenza tra nomi e oggetti denotabili non è univoca. In questo caso allo stesso oggetto—una variabile—corrispondono più nomi.

Una possibile soluzione a questi problemi è l'utilizzo di due parametri in più nella funzione `eq2`*, passati per riferimento, che di fatto fungono da parametri di output, in cui la funzione può memorizzare i risultati da essa calcolati.*

```
#include <iostream>
#include <cmath>

using namespace std;

bool eq2(float a, float b, float c, float &x_1, float &x_2) {
   float delta = pow(b,2) - 4*a*c;
   if(delta>=0) {
      float sqrt_delta = sqrt(delta);
      x_1 = (-b + sqrt_delta) / (2*a)
      x_2 = (-b - sqrt_delta) / (2*a)
   return true;
   }
   else return false;
}
int main() {
   float x,y,z;
   float sol_1, sol_2;
   cout << endl << "Inserisci i coefficienti dell'equazione: ";
   cin >> x >> y >> z;
   if (eq2(x,y,z,sol_1,sol_2))
      cout << "x_1 = " << sol_1 << "e x_2 =" << sol_2 << endl;
   else
      cout << "Discriminante negativo!" << endl;

   return 0;  }
```

L'esecuzione della funzione `eq2` *pone nelle variabili* `x_1` *e* `x_2` *le soluzioni dell'equazione data. Ma* `x_1` *e* `x_2` *altro non sono che riferimenti (sinonimi) delle variabili* `sol_1` *e* `sol_2` *dichiarate nel* `main` *e fatte corrispondere a* `x_1` *e* `x_2` *tramite la chiamata della* `eq2`*. Dunque, al termine di quest'ultima, le variabili* `sol_1` *e* `sol_2` *conterranno proprio le soluzioni all'equazione data calcolate dalla funzione* `eq2`*.*

Vediamo quali possono essere, al contrario, motivi generali per preferire il passaggio parametri per valore a quello per riferimento.

1. Maggiore indipendenza del programma chiamante dalla funzione chiamata. Una modifica (volontaria o meno) del parametro formale non si ripercuote sul programma chiamante.

2. Interfaccia della funzione, specificata dalla testata della funzione, più chiara. Con il passaggio per valore, i parametri della funzione rappresentano gli input alla funzione, mentre il risultato rappresenta il suo

(unico) output. Con il passaggio per riferimento, la funzione può avere diversi output, non chiaramente identificati, come nel caso della funzione `eq2`, o addirittura agire semplicemente per "side-effects", come nel caso della funzione `scambia`.

3. Maggiore riusabilità della funzione, come diretta conseguanza dei due punti precedenti.

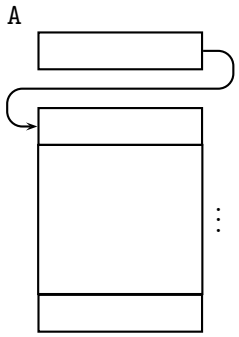
5.5.2 Passaggio parametri di tipo array

In C++ il passaggio parametri di tipo array richiede particolare attenzione. Per capirne bene il funzionamento bisogna tener presente come gli array vengono allocati in memoria in C++.

Nel sottocapitolo 4.3 abbiamo già visto che la dichiarazione di un array, come ad esempio

```
int A[10];
```

corrisponde alla seguente allocazione di memoria



Dunque, l'oggetto denotato dalla variabile `A` non è l'area di memoria contenente gli elementi che costituiscono l'array, quanto piuttosto un puntatore al (e cioè l'indirizzo del) primo di tali elementi. Come già accennato in 4.3, tutti gli accessi agli elementi dell'array avverranno in modo indiretto attraverso la cella puntatore.

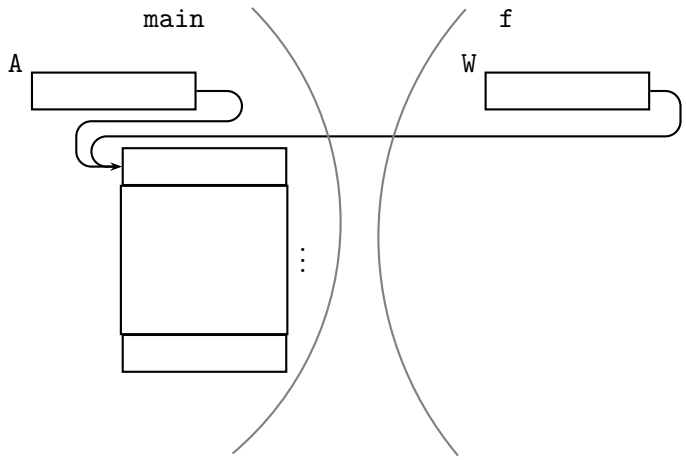
Supponiamo di voler scrivere una funzione `f` con parametro (formale) `W` di tipo array di interi, richiamata dal programma principale con parametro (attuale) `A` di tipo array di interi. La porzione di codice C++ che realizza questo è la seguente:

```
int main(){
   int A[10];
   A[0]=1;
   ...
   f(A)
   ...
}
```

```
int f(int W[]){
   ...
   W[0]=7;
   ...
}
```

La sintassi con le parentesi quadre nella dichiarazione di `W` indica che si tratta di un parametro di tipo array (notare che la dimensione non deve essere specificata quando l'array appare come parametro formale). La modalità di passaggio del parametro è quella per valore. Ora, passare un array `A` per valore significa in realtà fare una copia dell'indirizzo memorizzato nella variabile puntatore `A`, cioè dell'indirizzo di inizio dell'area di memoria dell'array. L'array vero e proprio non viene copiato; il parametro attuale e quello formale condividono la stessa area di memoria.

Graficamente:



Dunque, eventuali modifiche sugli elementi dell'array tramite `W` (ad esempio `W[0]=7`) si ripercuotono anche su `A`, esattamente come accade con il passaggio per riferimento.

Esempio 5.7 *Scrivere una funzione di nome* `scambia_elem` *che scambia tra loro gli elementi di indici* `i1` *e* `i2` *di un vettore* `V`*. Per esempio se* `V` *è il seguente vettore:*

	0	*1*	*2*	*3*	*4*	*5*	*6*
V =	*5*	*11*	*3*	*2*	*-3*	*7*	*8*

dopo l'esecuzione di `scambia_elem(V,2,4)` *l'array* `V` *diventa:*

	0	*1*	*2*	*3*	*4*	*5*	*6*
V =	*5*	*11*	*-3*	*2*	*3*	*7*	*8*

Codice funzione:

```
void scambia_elem(int V[], int i1, int i2) {
  int temp;
  temp = V[i1];
  V[i1] = V[i2];
  V[i2] = temp;
}

main() {    // main di prova
  int A[10];
  ...
  scambia_elem(A,3,5);
  ...
}
```

5.5.3 Esempi di funzioni con array

Esempio 5.8 *Scrivere una funzione di nome* `incrementa` *che presi come suoi parametri un array di interi A ed il numero n di elementi in A incrementa di* 1 *tutti gli elementi di A.*

```
void incrementa(int A[], int n) {
  for(int i=0; i<n; i++)
     A[i]++;
  return;
}

int main() {   // main di prova
  int V[5] = {1,2,3,4,5};
  incrementa(V,5);
  for(int i=0; i<5; i++)
     cout<<V[i]<<" ";
  return 0;
}
```

L'esecuzione di questo codice produce:

```
 2 3 4 5 6
```

Esempio 5.9 *Scrivere una funzione di nome* `ricerca_min` *che presi come suoi parametri un array di interi* `V`*, il numero* `n` *di elementi in* `V` *e un intero* `inizio`*, determina l'indice dell'elemento di* `V` *a partire da* `inizio` *contenente il valore minimo. Per esempio se:*

	0	*1*	*2*	*3*	*4*
V =	*3*	*10*	*5*	*7*	*6*

allora

```
ricerca_min(V, 5, 1) = 2
ricerca_min(V, 5, 0) = 0
```

Codice funzione:

```
int ricerca_min (int V[], int n, int inizio) {
  int min = inizio;
  for(int i=inizio+1; i<n; i++)
      if(V[i] < V[min]) min=i;
  return min;
}
```

Esempio 5.10 *Scrivere una funzione booleana di nome* `appartiene` *che presi come suoi parametri un array di interi A, il numero n di elementi in A e un intero x, determina se $x \in A$.*

```
bool appartiene (int A[], int n, int x) {
  for(int i=0; i<n; i++)
     if(A[i]==x) return true;
  return false;
}
```

Nota. *Un errore tipico è scrivere la funzione* `appartiene` *nel seguente modo:*

```
bool appartiene (int A[], int n, int x) {
  for(int i=0; i<n; i++) {
     if (A[i]==x) return true;
     else return false;
  }
}
```

In questo caso l'esecuzione termina sempre al **primo** *elemento dell'array, restituendo* `true` *o* `false` *a seconda che* `x` *sia uguale o no al* **primo** *elemento.*

Esempio 5.11 *Scrivere un programma principale che legge un intero a e una sequenza di n interi A ($0 \leq n < 100$), con n fornito in input dall'utente, determina se $a \in A$ (utilizzando la funzione* `appartiene`*) e quindi stampa su standard output un opportuno messaggio.*
Si richiede di realizzare la lettura della sequenza di interi e memorizzazione nell'array A tramite una funzione `leggi_vettore` *che restituisce come suo risultato il numero di elementi memorizzati in A.*

```
int leggi_vettore (int A[]) {
  int n;
  cout<< "Dai il numero di elementi da leggere:";
  do cin >> n; while (n<0 || n>dim_max);
  for(int i=0; i<n; i++) {
    cout << "Dammi un elemento:";
    cin>>A[i];
  }
```

```
  return n;
}

const int dim_max 100;
int main() {
  int n, a;
  V[dim_max];
  n = leggi_vettore[V];
  cout << "Dai un numero da cercare:";
  cin >> a;
  if(appartiene(V, n, a))
       cout << "La sequenza letta contiene il numero " << a << endl;
  else cout << "La sequenza letta non contiene il numero " << a << endl;

  return 0;
}
```

5.6 Sviluppo di un programma completo: un esempio

5.6.1 Metodologie di sviluppo programmi

I programmi visti finora in cui abbiamo utilizzato l'astrazione di funzione sono stati realizzati seguendo sostanzialmente un approccio *"bottom-up"*.

Sviluppo programmi bottom-up. Dato un problema, si individuano all'interno di questo dei sottoproblemi; si risolve ciascun sottoproblema, eventualmente utilizzando la stessa metodologia, dandone una soluzione nel linguaggio di programmazione scelto (spesso nella forma di una o più funzioni); quindi si mettono insieme le soluzioni sviluppate a formare un unco programma che risolve il problema generale.

Ad esempio, il programma mostrato nell'Esempio 5.11 è stato ottenuto individuando dapprima due sottoproblemi all'interno del problema dato, e precisamente il sottoproblema di "stampa di un vettore" e di "ricerca di un elemento in un vettore"; quindi abbiamo individuato una soluzione per ciascuno di questi due sottoproblemi (senza scomporli ulteriormente data la loro semplicità) e l'abbiamo implementata in C++ utilizzando il costrutto delle funzioni (funzioni `leggi_vettore` e `appartiene`, rispettivamente); infine abbiamo messo insieme questi "pezzi" a formare un unico programma.

In questo approccio si parte quindi "dal basso" (la soluzione dei sottoproblemi) e si procede verso l'"alto" (da ciò il termine di "bottom-up"), fino ad affrontare e risolvere il problema originale nel suo complesso.

È possibile anche utilizzare un approccio diametralmente opposto, che viene definito *"top-down"*.

Sviluppo programmi top-down. Dato un problema, si individuano all'interno di questo dei sottoproblemi; si fornisce una soluzione del problema originale nel suo complesso, assumendo nota la soluzione dei sottoproblemi; quindi si affronta ciascun sottoproblema, eventualmente utilizzando la stessa metodologia, fino a ridurlo ad elementi (operazioni e strutture dati) direttamente esprimibili nel linguaggio di programmazione scelto.

Nota. Si tratta in entrambi i casi di metodologie basate su un approccio *"divide et impera"* in cui l'aspetto fondamentale per giungere ad una soluzione è la capacità di scomporre il problema dato in sottoproblemi più semplici che, come tali, dovrebbero risultare più facili da affrontare e risolvere.

Come caso particolare di approccio "top-down" consideriamo una metodologia di sviluppo programmi denominata *"per raffinamenti successivi"* (o, in inglese, *"stepwise refinement"*), che illustreremo poi tramite un semplice esempio.

Sviluppo programmi "per raffinamenti successivi". Nella metodologia "per raffinamenti successivi" un programma viene costruito in più passi, o livelli di descrizione, *dal più astratto al più concreto*. Concetto chiave è quello di *astrazione*, concetto che abbiamo già incontrato più volte e che abbiamo approfondito all'inizio di questo capitolo.

- Primo livello. Descrizione dell'algoritmo molto astratta. Usa operazioni, strutture dati e strutture di controllo astratte, quelle più naturali per la formulazione della soluzione del problema. Normalmente scritta in *pseudo-codice*, ovvero in un linguaggio di descrizione algoritmi la cui sintassi ricorda quella di un linguaggio di programmazione vero e proprio (ad esempio, il C++), ma con un uso molto più libero di astrazioni (su operazioni, dati e controllo) descritte tramite linguaggio naturale. Si tratta dunque di una descrizione essenzialmente rivolta ad un esecutore umano.

- Secondo livello. Vengono forniti maggiori dettagli sulla realizzazione concreta di operazioni, strutture dati e strutture di controllo introdotte al primo livello e che non sono astrazioni primitive nel linguaggio di programmazione scelto per l'implementazione del programma. Quella che risulta è pertanto una descrizione più concreta dell'algoritmo descritto al livello precedente, ma non ancora in grado di essere eseguita in modo automatico.

- ...

- Ultimo livello. Tutte le astrazioni introdotte sono astrazioni primitive del linguaggio di programmazione scelto o possono essere sostituite da opportune implementazioni date in termini di quest'ultime. Il risultato è un programma completamente scritto nel linguaggio di programmazione scelto e pertanto eseguibile sul calcolatore.

È importante notare che quanti livelli di descrizione prevedere e quindi quando smettere di "raffinare" la descrizione della soluzione dipende dal linguaggio di programmazione scelto e precisamente da quante e quali astrazioni questo linguaggio supporta come primitive.

Mostriamo ora un esempio di sviluppo di un programma con una tecnica "per raffinamenti successivi" per un problema concreto: l'ordinamento di un vettore. Il linguaggio di implementazione scelto è, ovviamente, il C++.

5.6.2 Ordinamento di un vettore

Problema. Data una sequenza V di n numeri interi, disporre gli elementi di V in ordine crescente.

Nota. Più in generale, il problema dell'ordinamento di un vettore V di elementi di un certo tipo t può essere formulato come il problema di produrre una permutazione degli elementi di V che risulti ordinata rispetto ad una data relazione di ordine ($<$, $>$, $\leq$, $\geq$) sugli elementi di tipo t. Si noti che la scelta della relazione d'ordine determina il tipo di ordinamento ottenuto: crescente ($<$), decrescente ($>$), non-decrescente ($\leq$), non-crescente ($\geq$).

Numerosi sono gli algoritmi che sono stati proposti per risolvere questo problema e per i quali rimandiamo alla letteratura specifica. Qui di seguito ne considereremo uno, particolarmente semplice, noto come *"selection sort"*. Precisamente l'algoritmo che consideriamo è un algoritmo di ordinamento con ricerca del minimo (ordinamento in senso crescente).

L'idea di fondo dell'algoritmo è la seguente. Sia V un vettore di n elementi. Per tutti gli i da 1 a $n-1$, cerca il minimo del sottovettore di V compreso tra V_i a V_n (estremi inclusi), e quindi scambia l'elemento minimo con V_i.

Illustriamo l'algoritmo con un semplicissimo esempio. Per comodità utilizzeremo la notazione $V_{h,k}$, con h e k interi e $h \leq k$, per indicare il sottovettore di V costituito dai $k-h+1$ elementi V_h, V_{h+1}, ..., V_k. Supponiamo che il vettore da ordinare sia

$$[5, 8, 3, 2]$$

(quindi $n = 4$). Il primo passo ($i = 1$) comporta la ricerca del minimo nel sottovettore $V_{1,4}$ che risulta essere l'elemento di indice 4. Si effettua quindi lo scambio di questo elemento con l'elemento iniziale del sottovettore (e cioè V_1)) e quindi il nuovo vettore diventa:

$$[2, 8, 3, 5].$$

Il passo successivo ($i = 2$) comporta la ricerca del minimo nel sottovettore $V_{2,4}$, che risulta essere l'elemento di indice 3. Si effettua quindi lo scambio con l'elemento di indice 2 e quindi il nuovo vettore diventa:

$$[2, 3, 8, 5].$$

Al passo successivo ($i = 3$), dopo aver ricercato il minimo nel sottovettore $V_{3,4}$, e aver scambiato l'elemento minimo (V_4) con l'elemento iniziale (V_3), si ottiene il nuovo vettore:

$$[2, 3, 5, 8]$$

e l'algoritmo termina (essendo $i = n - 1$). Il vettore V risulta ordinato in senso crescente.

Mostriamo ora il codice C++ che implementa questo algoritmo. Lo sviluppo del programma viene fatto con la metodologia "per raffinamenti successivi".

Primo livello

La descrizione dell'algoritmo viene data in pseudo-codice (stile C). Le astrazioni utilizzate sono indicate in corsivo.

```
ordina(vettore V di n elementi di tipo int) {
    int imin;
    while(V ha almeno due elementi) {
        imin = trova l'indice dell'elemento minimo in V;
        scambia V_imin con il primo elemento di V;
        V = sottovettore di V senza il primo elemento;
    }
}
```

Si noti che la descrizione contiene astrazioni sui dati (ad esempio, *vettore di interi*), sulle operazioni (ad esempio, *trova l'indice dell'elemento minimo in V*) e sul controllo (ad esempio, la struttura *while*). Tutte queste astrazioni dovranno essere opportunamente realizzate nei livelli successivi.

Secondo livello

Procediamo con un raffinamento delle astrazioni introdotte al primo livello e che non risultano primitive nel linguaggio di implementazione scelto, ovvero il C++.

1. Astrazioni sui dati:
   - `int`: astrazione primitiva in C++ (= numeri interi)
   - *vettore V di n elementi di tipo* ***int***: può essere realizzato in C++ tramite il costruttore di tipo *array*
     ```
     int V[]
     ```
     più un numero intero `n` (`int n`) che rappresenta il numero di elementi presenti nel vettore.

- *sottovettore di V*: può essere realizzato utilizzando lo stesso array `V`, più l'indice del suo elemento finale `n`, più un intero `inizio` (`int inizio`) che rappresenta l'indice dell'elemento iniziale del sottovettore.

2. Astrazioni sulle operazioni

   - *trova l'indice dell'elemento minimo in V*: può essere realizzata da una funzione così definita (usando la notazione C++ per la dichiarazione di funzioni):
     ```
     int ricerca_min(int V[],int n,int inizio)
     ```
     che restituisce l'indice dell'elemento minimo nel sottovettore di `V` compreso tra `inizio` e `n`.
   - *scambia due elementi di V*: può essere realizzata da una funzione così definita:
     ```
     void scambia_elem(int V[],int i1,int i2)
     ```
     che scambia il contenuto degli elementi di `V` di indici `i1` e `i2`.
   - assegnamento (`=`): primitivo in C++
   - *sottovettore di V senza il primo elemento*: con la realizzazione dell'astrazione sottovettore ipotizzata sopra, questa operazione diventa semplicemente l'aggiornamento dell'indice che individua l'elemento iniziale del sottovettore, e cioè:
     ```
     inizio = inizio + 1
     ```
   - *V ha almeno due elementi*: sempre con le ipotesi sulla realizzazione del sottovettore di sopra, questa operazione diventa semplicemente il test $\texttt{n} - \texttt{inizio} \geq 2$, che è immediatamente realizzabile con l'espressione C++ (primitiva) `n - inizio >= 2` o, equivalentemente, `inizio < n-1`.

3. Astrazioni sul controllo

   - `while`: primitiva in C++
   - sequenza (di statement): : primitivo in C++.

Con queste nuove precisazioni sulle astrazioni utilizzate, la descrizione della soluzione al secondo livello diventa la seguente:

```
ordina(int V[], int n) {
    int imin, inizio = 0;
    while(inizio < n-1) {
        imin = ricerca_min(V,n,inizio);
        scambia_elem(V,inizio,imin);
        inizio++;
    }
}
```

Terzo livello

Le uniche astrazioni non primitive ancora presenti sono le operazioni `ricerca_min` e `scambia_elem`.

Nota. Si noti che se il linguaggio di implementazione scelto fosse ad esempio un linguaggio Assembly non potremmo assumere che il tipo di dato *array* o la struttura di controllo `while` siano primitive, ma dovremmo ulteriormente dettagliare la loro realizzazione in termini delle più semplici astrazioni primitive offerte dal linguaggio Assembly.

Queste due operazioni possono essere realizzate utilizzando il costrutto delle funzioni offerto dal C++ che ci permette di darne l'implementazione in modo separato dal resto del codice.

Questa possibilità ci consente, tra l'altro, di riutilizzare l'implementazione di queste funzioni già mostrata nei sottocapitoli precedenti e, precisamente, nell'Esempio 5.9, per la funzione `ricerca_min`, e nell'Esempio 5.7, per la funzione `scambia_elem`.

L'esempio 5.12 mostra un programma completo in cui si utilizza la funzione `ordina` appena realizzata. Nel programma si riutilizza anche una funzione `leggi_vettore` la cui realizzazione è già stata mostrata nell'Esempio 5.11. Il programma completo dunque è ottenuto seguendo un approccio "bottom-up" in cui si "assemblano" funzioni già realizzate, eventualmente costruite a loro volta con tecniche "bottom-up" o "top-down".

Esempio 5.12 (Ordinamento di un vettore di interi)

```
/* Scrivere un programma che legge una sequenza di n interi V (0
<= n <= 100), con n fornito in input dall'utente, la ordina in
senso crescente e quindi stampa la sequenza ordinata */

#include <iostream>
using namespace std;

const int dim_max = 100;

int ricerca_min(int [],int,int);
void scambia_elem(int [],int,int);

void ordina(int V[], int n) {
  int imin, inizio = 0;
  while (inizio < n -1) {
     imin = ricerca_min(V,inizio,n);
     scambia_elem(V,inizio,imin);
     inizio++;
  }
  return;
}
```

```
int leggi_vettore(int A[]) {
  int n;
  cout << "Dai il numero di elementi da leggere: ";
  do cin >> n; while (n < 0 || n > dim_max);
  for (int i=0; i<n; i++) {
     cout << "Dammi un elemento: ";
     cin >> A[i];
  }
  return n;
}

int main() {
  int n;
  int A[dim_max];
  n = leggi_vettore(A);
  ordina(A,n);
  cout << "Vettore ordinato: " << endl;
  for (int i=0; i < n; i++)
      cout << A[i] << "   ";
  cout << endl;
%  system("pause");
  return 0;
}

int ricerca_min(int V[], int n, int inizio) {
  int imin = inizio;
  for (int j = inizio+1; j < n; j++)
     if (V[j] < V[imin]) imin = j;
  return imin;
}

void scambia_elem(int V[], int i1, int i2) {
  int temp;
  temp = V[i1];
  V[i1] = V[i2];
  V[i2] = temp;
  return;
}
```

5.6.3 Valutazione della complessità computazionale (IN PREP.)

Il problema dell'ordinamento di un vettore può essere usato anche come semplice esempio per introdurre alcune considerazioni sulla valutazione della complessità computazionale degli algoritmi. Rimandiamo ai testi specializzati per una trattazione più completa dell'argomento. Qui vogliamo solo fornire alcuni elementi molto di base e intuitivi per far capire quali sono le problematiche in gioco.

IN PREPARAZIONE

5.7 Funzioni ricorsive

Funzione ricorsiva. Una funzione f si dice *ricorsiva* quando è definita secondo il seguente schema:

- uno o più casi base, in cui f è definita in termini di altre funzioni più semplici note;
- un caso ricorsivo (o induttivo) in cui f è definita in termini di se stessa applicata a dati più "semplici". □

Esempio 5.13 (Funzione fattoriale) *Un esempio "classico" di funzione ricorsiva è la funzione che calcola il fattoriale di un numero naturale x*

$$x! : \mathbb{N} \to \mathbb{N}$$

Definizione iterativa:

$$\texttt{fatt\_it}(x) = \begin{cases} 1 & se\ x = 0 \\ 1 * \ldots * x & altrimenti \end{cases}$$

La definizione ricorsiva invece la possiamo dare nel seguente modo:

$$\texttt{fatt\_rec}(x) = \begin{cases} 1 & se\ x = 0 \quad (caso\ base) \\ x * \texttt{fatt\_ric}(x-1) & altrimenti \quad (caso\ ricorsivo) \end{cases}$$

Si noti come questa definizione rispetti la struttura generale delle funzioni ricorsive. Nel caso base la funzione è data in termini di una funzione nota, la funzione costante 1, mentre nel caso ricorsivo la funzione è data in termini di se stessa applicata a $x - 1$ (il "predecessore" di x) che, nel caso dei numeri naturali, rappresenta il dato più "semplice" previsto nella definizione generale.

Esempio 5.14 (Funzione potenza) *Un altro esempio di funzione ricorsiva è l'elevamento a potenza:*

$$x^y : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

Definizione iterativa:

$$\texttt{pot\_it}(x, y) = \begin{cases} 1 & se\ y = 0 \\ \underbrace{x * \ldots * x}_{y\ volte} & altrimenti \end{cases}$$

Definizione ricorsiva:

$$\texttt{pot\_ric}(x, y) = \begin{cases} 1 & se\ y = 0 \\ \texttt{pot\_ric}(x, y-1) * x & altrimenti \end{cases}$$

Anche in questo caso il dato più "semplice" previsto nella definizione generale è il "predecessore" del numero naturale y. Vedremo più avanti altri esempi di funzioni ricorsive definite su altri domini in cui si utilizzeranno dati più "semplici" diversi.

Molti linguaggi di programmazione, tra cui il C++, danno la possibilità di realizzare una funzione ricorsiva in modo semplice e diretto. Come esempio, mostriamo il codice C++ che realizza la funzione ricorsiva `fatt_ric` mostrata sopra.

```
int fatt_ric(int x) {
   if(x==0)
      return 1;
   else
      return fatt_ric(x-1)*x;
}
```

Si vede che la definizione della `fatt_ric` viene facilmente realizzata con una funzione C++, di nome `fatt_ric`, in cui si distingue il caso base dal caso ricorsivo tramite un `if`, e in cui il risultato restituito dalla funzione è, in un caso, `1` e nell'altro il risultato di un'espressione in cui si richiama la funzione stessa con parametro attuale `x-1`.

Si osservi come tale implementazione corrisponda in modo "1 a 1" alla definizione ricorsiva della `fatt_ric` data sopra. Dalla correttezza di quest'ultima, dunque, si può derivare immediatamente la correttezza dell'implementazione C++.

Approfondimento (Equivalenza definizione ricorsiva e iterativa di $x!$. È possibile dimostrare formalmente che la definizione ricorsiva della funzione fattoriale è equivalente alla (ovvero, dà lo stesso risultato della) definizione iterativa, e cioè che per ogni intero $n \geq 0$ abbiamo $\texttt{fatt\_ric}(n) = \texttt{fatt\_it}(n)$. La prova sarà fornita come Corollario della seguente,

Proposizione 5.15 *Per ogni numero naturale $n \geq 0$,*

$$\texttt{fatt\_it}(n+1) = (n+1) * \texttt{fatt\_it}(n)$$

DIMOSTRAZIONE: *La dimostrazione procede per casi su n:*

$n = 0$ *La tesi si riduce a,*

$$\texttt{fatt\_it}(1) = 1 * \texttt{fatt\_it}(0)$$

che in base alla definizione di `fatt_ric` *e* `fatt_it` *riduce a provare,*

$$1 = 1 * 1$$

$n > 0$ *In base alla definizione di* $\texttt{fatt\_it}(n+1)$ *la tesi si riduce a,*

$$1 * \ldots * n * (n+1) = (n+1) * 1 * \ldots * n$$

che si prova grazie alla proprietà commutativa del prodotto.

□

Corollario 5.16 *Per ogni intero n,*

$$\texttt{fatt\_ric}(n) = \texttt{fatt\_it}(n)$$

DIMOSTRAZIONE: *Per induzione su n.*

Caso base $n = 0$ *Proviamo,*

$$\texttt{fatt\_it}(0) = \texttt{fatt\_ric}(0)$$

che è banalmente vera in base alla definizione di `fatt_ric` *e* `fatt_it`*.*

Passo induttivo *Assumiamo ("ipotesi induttiva") la tesi vera per un intero n qualsiasi; quindi assumiamo,*

$$\texttt{fatt\_it}(n) = \texttt{fatt\_ric}(n) \tag{5.2}$$

e proviamo che la tesi vale per $n + 1$, cioè proviamo:

$$\texttt{fatt\_it}(n + 1) = \texttt{fatt\_ric}(n + 1) \tag{5.3}$$

Grazie alla Proposizione 5.15 e alla definizione di `fatt_ric` *l'eguaglianza (5.3) riduce a:*

$$(n + 1) * \texttt{fatt\_it}(n) = (n + 1) * \texttt{fatt\_ric}(n) \tag{5.4}$$

e tramite l'ipotesi induttiva (5.2) abbiamo la tesi.

□

La presenza della ricorsione in un linguaggio di programmazione aggiunge potere espressivo, ma non potere computazionale al linguaggio stesso. Il maggior potere espressivo deriva dal fatto che la possibilità di scrivere funzioni ricorsive permette di esprimere in modo del tutto naturale definizioni ricorsive che, in molti casi, risultano il modo più immediato di formulare la soluzione di un dato problema (anche se l'esatta formulazione ricorsiva di un problema può richiedere buone capacità di astrazione e una certa confidenza con il ragionamento logico-formale). Il fatto che, invece, la ricorsione non aggiunga potere computazionale significa che la presenza di questo meccanismo non permette di dare una soluzione algoritmica a problemi che altrimenti (e cioè con la sola iterazione) non sarebbero risolvibili: ciò che è computabile con la ricorsione è computabile con l'iterazione, e viceversa.

Va anche osservato che l'esecuzione di un programma ricorsivo è, in generale, più complessa di quella del corrispondente programma iterativo e richiede la presenza di un adeguato meccanismo di esecuzione del programma (*supporto run-time*) che può comportare un peggioramento dei tempi di esecuzione complessivi.

Per dare un'idea del modo in cui avviene l'esecuzione di una funzione ricorsiva, consideriamo l'esecuzione della funzione `fatt_ric(x)` nel caso in cui `x` valga `4`:

$$
\begin{aligned}
\texttt{fatt\_ric}(4) &= 4 * \texttt{fatt\_ric}(3) \\
&= 4 * (3 * \texttt{fatt\_ric}(2)) \\
&= 4 * (3 * (2 * \texttt{fatt\_ric}(1))) \\
&= 4 * (3 * (2 * (1 * \texttt{fatt\_ric}(0)))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

Nell'esempio si nota che l'esecuzione di `fatt_ric(4)` comporta 4 chiamate annidate della funzione `fatt_ric` con i relativi "ritorni" (esecuzione dello statement `return`). Ogni chiamata richiede una nuova allocazione dei dati locali della funzione, mentre l'esecuzione dei `return` comporta la deallocazione di tali dati. Inoltre, il numero delle chiamate annidate dipende dal valore del parametro `x` e quindi non può essere determinato a priori. Tutto ciò implica la presenza a run-time di un meccanismo di allocazione/deallocazione della memoria opportuno.

Sono comunque state sviluppate tecniche per la gestione efficiente della ricorsione che in vari casi permettono di avere la stessa efficienza di esecuzione delle corrispondenti soluzione iterative.

Infine mostriamo due esempi di funzioni ricorsive su domini diversi da quello dei numeri naturali finora considerato, e precisamente il dominio delle stringhe e il dominio dei vettori di interi.

Esempio 5.17 *(Lunghezza di una stringa)*
Definizione ricorsiva.

$$
\texttt{lungh}(s) = \begin{cases} 0 & \textit{se } s = \texttt{""} \\ \texttt{lungh}(\texttt{"}a_2 \ldots a_n\texttt{"}) + 1 & \textit{se } s = \texttt{"}a_1 a_2 \ldots a_n\texttt{"} \end{cases}
$$

Realizzazione in C++.

Assumiamo che le stringhe siano rappresentate, come al solito (si veda il Capitolo 4.5), come array di caratteri terminati da `'\0'`*. Inoltre associamo a ciascuna stringa* `s` *un intero* `inizio` *che contiene l'indice dell'elemento iniziale della sottostringa di* `s` *che stiamo al momento considerando (e che servirà a determinare facilmente la sottostringa* `"`$a_2 \ldots a_n$`"` *nella chiamata ricorsiva della* `lungh`*). La funzione C++ che implementa la* `lungh` *avrà pertanto due parametri: la stringa* `s` *e l'intero* `inizio`*.*

```
int lungh(char s[], int inizio) {
   if(s[inizio] == '\0')
       return 0;
   else
       return (lung(s,inizio+1) + 1);
}
```

Si noti che in questo caso il dato "più semplice" utilizzato nella chiamata ricorsiva è costituito dalla sottostringa ottenuta da `s` *togliendovi il primo elemento.*

La chiamata della funzione

```
cout << lungh("ciao come stai?") << endl;
```

stamperà sullo standard output il valore `15`*.*

Nota. Per "nascondere" il parametro `inizio` presente nella realizzazione della `lungh` in C++ possiamo definire un'altra funzione con il solo parametro `s`—che può essere denominata anch'essa `lungh` dato che il C++ è in grado di distinguere due funzioni con lo stesso nome in base al tipo e numero dei parametri formali—e che avrà il solo scopo di richiamare la `lungh` con due parametri, dando al parametro `inizio` il valore iniziale `0`:

```
int lungh (char s[]) {
   return lungh(s,0);
}
```

Esempio 5.18 *(Appartenenza di un elemento ad un vettore di interi)*

Definizione ricorsiva.

$$\texttt{appartiene}(A,x) = \begin{cases} \texttt{false} & \textit{se } A = \texttt{[]} \\ \texttt{true} & \textit{se } x = A[0] \\ \texttt{appartiene}(\{a_2,\ldots,a_n\},x) & \textit{se } x \neq A[0] \textit{ e} \\ & A = \{a_1,a_2,\ldots,a_n\} \end{cases}$$

Realizzazione in C++.

Anche in questo caso assumiamo di associare al vettore `A`*, oltre che alla sua lunghezza* `n`*, un intero* `inizio` *che contiene l'indice dell'elemento iniziale del sottovettore di* `A` *che si sta considerando al momento.*

```
appartiene(int A[], int n, int inizio, int x) {
  if(inizio == n) return false;
  else if(A[inizio] == x)
          return true;
  else
          return appartiene(A,n,inizio+1,x);
}
```

La chiamata della funzione

```
appartiene(V, 5, 0, 1);
```

con `V` *definito come*

```
int V[5] = {3, 4, 1, 2, 8};
```

restituirà come risultato `true`.

5.8 *Domande per il Capitolo 5*

1. Quali sono le tre forme generali di astrazione tipicamente presenti in un linguaggio di programmazione ad alto livello?
2. Come si definisce in C++ il prototipo di una funzione $f : \mathbb{R} \times \mathbb{Z} \to \mathbb{Z}$, dove $\mathbb{R}$ indica l'insieme dei numeri reali e $\mathbb{Z}$ quello degli interi?
3. Cosa si intende con parametri formali e parametri attuali?
4. Descrivere sinteticamente il meccanismo di esecuzione della chiamata di funzione `alfa(e1,...,en)`, dove `alfa` è il nome della funzione ed `e1`,...,`en` sono espressioni.
5. Che forma sintattica ha lo statement `return`? Qual è il suo significato e funzionamento?
6. Indicare almeno due situazioni in cui risulta naturale/utile definire il risultato di una funzione come di tipo `void` (riferirsi a situazioni generali, non esempi specifici).
7. Quali dichiarazioni comprende il blocco di una funzione in C++?
8. Cosa significa che un riferimento ad una variabile all'interno di una funzione è "locale" oppure "non locale"?
9. Cosa si intende con "ambiente globale"? Indicare almeno due svantaggi dell'uso di variabili globali all'interno di una funzione.
10. Quali modalità di passaggio parametri sono previste in C++.
11. Descrivere (anche con l'ausilio di un disegno) il funzionamento del passaggio parametri per valore e per riferimento in C++.
12. Indicare almeno due vantaggi del passaggio parametri per riferimento rispetto al passaggio per valore.
13. Indicare almeno un vantaggio del passaggio parametri per valore rispetto al passaggio per riferimento.
14. Se `f` è la funzione

    ```
    int f(int& x)
       {return ++x;}
    ```

    qual è il risultato prodotto dall'esecuzione delle seguenti istruzioni:

```
int z = 3;
cout << f(f(z) + z);
```

Giustificare la risposta.

15. Cosa significa, in generale, che una funzione produce un "side-effect"? Illustrare con un esempio.
16. Se `A` è un array di `10` interi ed `f` è una funzione con un parametro `V` di tipo array di interi, mostrare, con l'aiuto di un disegno, come avviene il passaggio parametri nella chiamata di funzione `f(A)`.
17. Perchè non è necessario specificare la dimensione dell'array nella dichiarazione della funzione? Dove e quando viene allocata l'area di memoria destinata a contenere gli elementi dell'array?
18. Cosa significano "sviluppo bottom-up" e "sviluppo top-down" di un programma?
19. In cosa consiste lo "sviluppo programmi per raffinamenti successivi"?
20. Dare una descrizione in pseudo-codice "stile C" dell'algoritmo di ordinamento con selezione del minimo.
21. Qual è la complessità computazionale dell'algoritmo di ordinamento con selezione del minimo?
22. Come è definita una funzione ricorsiva?
23. Esistono problemi che sono risolvibili soltanto se il linguaggio di programmazione offre le funzioni ricorsive?
24. Cosa significa che la possibilità di definire funzioni ricorsive aggiunge potere espressivo al linguaggio di programmazione?
25. Dare la definizione ricorsiva della funzione `inverti(V)` che inverte l'ordine degli elementi nel vettore di interi V (ad esempio, `inverti({3,5,2,8})` = {`8,2,5,3`}).
    Dare anche l'implementazione come funzione ricorsiva in C++.