Consideriamo la seguente funzione, che immaginiamo sia parte di un programma.

```
int f (int n, int v[]) {
  int x = v[0];
  int c = 1;

  while ( c < n ) {
    if ( x < v[c] )
      x = v[c];
    c = c + 1;
  }

  return x;
}
```

Intuitivamente, la funzione `f` calcola il valore massimo in un vettore di n elementi, e restituisce questo valore. Come possiamo dimostrare che la funzione è ben scritta, cioè "corretta"?

Un primo passo consiste nell'esprimere precisamente che cosa intendiamo quando diciamo che il codice di questa funzione è corretto, cioè fa (sempre) quello che ci aspettiamo.

A questo scopo, dobbiamo indicare che cosa deve valere alla fine dell'esecuzione, ma anche che cosa (eventualmente) presupponiamo dei valori degli argomenti della funzione.

La logica matematica ci dà gli strumenti per esprimere esattamente quello che ci aspettiamo dalla funzione; scriviamo una formula logica che dovrebbe essere vera alla fine dell'esecuzione della funzione:

$$\forall i \in \{0,\ldots,n-1\} : x \geq v[i] \quad \wedge \quad \exists i \in \{0,\ldots,n-1\} : x = v[i] \tag{1.1}$$

La formula esprime il fatto che, al termine dell'esecuzione, la variabile `x` contiene il valore massimo fra tutti i valori contenuti nel vettore (per essere ancora più precisi, dovremmo specificare che i valori nel vettore `v` restano invariati nel corso dell'esecuzione).

Per definire più esattamente il senso di "vera alla fine dell'esecuzione" occorre definire la nozione di *stato* del programma. Supponiamo che V sia l'insieme

delle variabili di un programma (o di una funzione, o di un metodo). Per semplicità, in queste lezioni ci limiteremo a variabili a valori interi. Informalmente, uno stato è una fotografia della memoria in un istante dell'esecuzione; lo stato è quindi definito dal valore delle variabili del programma che stiamo considerando. Diremo allora che uno stato del programma (o della funzione, o del metodo) è una funzione $\sigma : V \to Z$, dove Z denota l'insieme degli interi relativi.

Una formula come la (1.1), che esprime relazioni aritmetiche fra i valori delle variabili, sarà vera in uno stato se in quello stato valgono le relazioni espresse dalla formula. In questo caso, se q denota la formula e σ lo stato, scriveremo $\sigma \models q$, e diremo "σ soddisfa q", oppure "q è valida (o vera) in σ".

In alcuni casi la correttezza di un programma è garantita solo se valgono certe precondizioni sui valori iniziali delle variabili. Nell'esempio qui sopra, ci aspettiamo che il valore dell'argomento n sia non negativo (cioè che ci sia almeno un elemento nel vettore). Anche queste precondizioni si possono esprimere per mezzo di una formula logica.

Riassumendo, possiamo specificare la condizione di correttezza di un programma C indicando la precondizione e la postcondizione, cioè la formula che deve valere alla fine dell'esecuzione. Per convenzione, scriveremo

$$\{p\}\ C\ \{q\} \tag{1.2}$$

per specificare la precondizione p, il programma C, e la postcondizione q, e chiameremo *tripla* questa forma. La tripla si legge così: "se si esegue il programma C a partire da uno stato in cui è valida p, l'esecuzione termina e nello stato finale vale q".

Una tripla è quindi una proposizione, che può essere vera o falsa. Nelle prossime lezioni definiremo delle regole di inferenza che permettono di dimostrare la validità delle triple corrette.

Per ora, cerchiamo di dare una dimostrazione informale, ma il più possibile rigorosa, della correttezza della funzione f. La funzione percorre il vettore, a partire dall'indice 0, conservando nella variabile x il valore più alto incontrato fino a quel momento. Questo valore viene confrontato con il prossimo elemento del vettore e, se si riscontra un valore più alto, si aggiorna la variabile x.

Dopo le prime i iterazioni, quindi, x contiene il valore più alto fra quelli presenti nelle prime i posizioni del vettore (corrispondenti agli indici fra 0 e $i-1$). L'esecuzione della prossima iterazione conserva questa proprietà, che possiamo esprimere formalmente così:

$$\forall i \in \{0,\ldots,c-1\} : x \geq v[i] \quad \wedge \quad \exists i \in \{0,\ldots,c-1\} : x = v[i] \tag{1.3}$$

Notate che, passando da un'iterazione alla successiva, la formula resta valida, ma il valore di c cambia. Chiameremo una formula di questo tipo *invariante di ciclo*. In generale, data un'istruzione iterativa del tipo

```
while (B) S
```

dove `B` è un'espressione booleana e `S` un blocco di codice, diremo che la formula p è un invariante di ciclo se, eseguendo `S` a partire da uno stato in cui valgono sia p sia `B`, si raggiunge uno stato in cui vale p. La formula p può naturalmente contenere variabili del programma, il cui valore cambia per effetto dell'esecuzione di `S`.

Tornando all'esempio, l'equazione (1.3) è un invariante di ciclo. Ad ogni iterazione, il valore di c viene incrementato di 1. Di conseguenza, il valore di c raggiungerà prima o poi n, provocando l'uscita dall'istruzione iterativa. A questo punto, la formula (1.3) sarà valida e c varrà n. Quindi, varrà anche la formula (1.1).

Possiamo considerare conclusa la dimostrazione "semiformale" di correttezza. Nel seguito, definiremo un metodo più rigoroso e completo. Per ora, osserviamo che si può annotare il programma con commenti che riflettono i passi del ragionamento:

```
int f (int v[], int n)
{
    // Per ogni i compreso fra 0 e n-1, v[i] contiene
    // un numero intero.
    int   x = v[0];
    int   a = 1;

    while ( a < n )
    {
        // x = max(v[0],...,v[a-1])
        if ( x < v[a] )
            x = v[a];
        // x = max(v[0],...,v[a])

        a = a + 1;
        // x = max(v[0],...,v[a-1])
    }
    // a = n, x = max(v[0],..., v[n-1])
    return x;
}
```